Operating Systems

Vlastimil Babka

Lubomír Bulej

Vojtěch Horký

Petr Tůma

Operating Systems

by Vlastimil Babka, Lubomír Bulej, Vojtěch Horký, and Petr Tůma

Table of Contents

Chapter 1. Introduction

Foreword

Origins

This material originated as a bunch of scribbled down notes for the Charles University Operating Systems lecture. As time went on and the amount of notes grew, I came to realize that the amount of work that went into looking up the information and writing down the notes is no longer negligible. This had two unfortunate implications. First, verifying the notes to maintain the information within updated became difficult. Second, asking the students to locate the information within individually became unrealistic. This material is an attempt at solving both problems. By extending and publishing the notes, I hope to provide the students with a source of information, and me with a source of feedback.

I realize some readers will find this material fragmented, incomplete and unreadable. I also hope other readers will find this material current, detailed and interesting. The notes are being extended and published in good faith and should be taken as such. And remember that you can always revert to other sources of information. Some are listed below.

References

1. Abraham Silberschatz: Operating System Concepts. Wiley 2002. ISBN 0471250600
2. Andrew S. Tannenbaum: Modern Operating Systems, Second Edition. Prentice Hall 2001. ISBN 0130313580
3. Uresh Vahalia: UNIX Internals: The New Frontiers. Prentice Hall 1995. ISBN 0131019082

Structure

It is a laudable trait of technical texts to progress from basic to advanced, from simple to complex, from axioms to deductions. Unfortunately, it seems pretty much impossible to explain a contemporary operating system in this way - when speaking about processes, one should say how a process gets started, but that involves memory mapped files - when speaking about memory mapped files, one should say how a page fault gets handled, but that involves devices and interrupts - when speaking about devices and interrupts, one should say how a process context gets switched, but that involves processes - and so on. This text therefore starts with a look at historic perspective and basic concepts, which gives context to the text that follows. There, forward and backward references are used shamelessly :-).

Historic Perspective

Stone Age

In 1940s, computers were built by Howard Aiken at Harward University, John von Neumann at Princeton University, and others. The computers used relays or vacuum tubes, the former notoriously unreliable, the latter plagued with power consumption and heat generation. The computers were used to perform specialized calculations, which were initially programmed, or, rather, wired into the computer using plug boards. Plug boards were later replaced by punch cards or paper tapes. There was no notion of an operating system.

Hardware	Year	Software
Mark I or *Automatic Sequence Controlled Calculator* - a computer developed by IBM and Harward University, uses relays, program stored on paper tapes, a multiplication operation takes 6 seconds, a division operation takes 12 seconds.	1944	
Electronic Numerical Integrator And Computer (ENIAC) - a computer developed by University of Pennsylvania, uses vacuum tubes, program stored on plug boards, a division operation takes 25 miliseconds.	1946	
Selective Sequence Electronic Calculator - a computer developed by IBM, uses relays and vacuum tubes, program stored on paper tape and in internal memory, a multiplication operation takes 20 miliseconds, a division operation takes 33 miliseconds.	1948	
Electronic Delay Storage Automatic Calculator (EDSAC) - a computer developed by University of Cambridge, uses vacuum tubes, program stored on paper tape and in internal memory, a multiplication operation takes 4.5 miliseconds, a division operation takes 200 miliseconds.	1949	
Electronic Discrete Variable Automatic Computer (EDVAC) - a computer developed by University of Pennsylvania, uses vacuum tubes, program stored on magnetic wires and in internal memory, multiplication and division operations take 3 miliseconds.	1951	

References

1. Weik M. H.: The ENIAC Story. http://ftp.arl.mil/~mike/comphist/eniac-

story.html

2. The Columbia University Computing History Website. http://www.columbia.edu/acis/history
3. The Manchester University Computing History Website. http://www.computer50.org
4. The EDSAC Website. http://www.cl.cam.ac.uk/UoCCL/misc/EDSAC99

Transistors

In 1950s, computers used transistors. The operation times went down from miliseconds to microseconds. To maximize processor utilization, specialized hardware was introduced to handle input and output operations. The computers were running a simple operating system, responsible for loading other programs from punch cards or paper tapes and executing them in batches.

Hardware	Year	Software
Transistor - a semiconductor device capable of amplifying or switching an electric current has been invented by William Shockley at Bell Laboratories.	1947	
IBM 701 - a computer developed by IBM, uses vacuum tubes, multiplication and division operations take 500 microseconds. The first computer that was mass produced (as far as 19 computers can be considered a mass :-).	1952	
IBM 350 - a harddrive developed by IBM, capacity of 5 MB at 50 rotating magnetic discs with a diameter of 61 cm.	1956	
IBM 709 - a computer developed by IBM, uses vacuum tubes, multiplication and division operations take 240 microseconds.	1957	*Fortran* - a programming language developed by John W. Backus at IBM.
IBM 7090 - a computer developed by IBM, uses transistors, a multiplication operation takes 25 microseconds, a division operation takes 30 microseconds.	1958	

One of the most powerful computers of the time was IBM 7094. The computer could perform floating point operations in tens of microseconds and was equipped with 32k words of memory, one word being 36 bits. Specialized hardware provided channels for independent input and output operations that could interrupt the processor.

The IBM 7094 computer run the *Fortran Monitor System (FMS)*, an operating system that executed sequential batches of programs. A program was simply loaded into memory, linked together with arithmetic and input and output libraries and executed. Except for being limited by an execution timeout, the program was in full control of the computer.

Executing sequential batches of programs turned out to be inflexible. At MIT, the first experiments with sharing the computer by multiple programs were made in 1958 and published in 1959. Eventually, a system that can interrupt an executing program, execute another program and then resume the originally interrupted program, was developed. The system was called *Compatible Time Sharing System (CTSS)* and required a hardware modification of the IBM 7094 computer.

Low Integration

In 1960s, integrated circuits appeared alongside transistors. Integration has paved the way for smaller computers, less power consumption, less heat generation, longer uptimes, larger memory and lots of other related improvements. Cabinet-sized minicomputers have appeared alongside room-sized mainframe computers. The computers run operating systems that support executing multiple programs in parallel with virtual memory provided by paging.

Hardware	Year	Software
Integrated circuit - a technology to integrate multiple transistors within a single device has developed by Robert Noyce at Fairchild Semiconductors.	1961	
Mouse - an input device with two wheels developed by Douglas Engelbart at SRI.	1963	
IBM System/360 - a computer developed by IBM. The first computer with configurable assembly from modules.	1964	*Beginner's All Purpose Symbolic Instruction Code (BASIC)* - a programming language developed by J. Kemeny and T. Kurtz at Dartmouth College. *Time Sharing System (TSS)* - an operating system developed at IBM.
	1965	*MULTICS* - an operating system developed at Bell Laboratories.
Dynamic Random Access Memory (DRAM) - a memory circuit developed at IBM.	1966	
ARPANET - a network project at ARPA.	1969	
	1970	*Uniplexed Information and Computing System (UNICS, UNIX)* - an operating system developed at Bell Laboratories.

Hardware	Year	Software
	1971	*Pascal* - a programming language developed by Niklaus Wirth at ETH Zurich.
	1972	*SmallTalk* - a programming language developed by Alan Kay at Xerox PARC.
Mouse - an input device with a single ball developed by Bill English at Xeroc PARC.	1973	

A well known computer of the time, IBM System/360, has been the first to introduce configurable assembly from modules. The computer used the OS/360 operating system, famous for its numerous bugs and cryptic messages. OS/360 supported executing multiple programs in parallel and introduced *spooling* of peripheral operations. Another famous operating system was *Multiplexed Information And Computing Service (MULTICS)*, designed for providing public computing services in a manner similar to telephone or electricity. MULTICS supported memory mapped files, dynamic linking and reconfiguration.

An important line of minicomputers was produced by Digital Equipment Corporation. The first of the line was DEC PDP-1 in 1961, which could perform arithmetic operations in tens of microseconds and was equipped with 4k words of memory, one word being 18 bits. All this at a fraction of the size and cost of comparable mainframe computers.

High Integration

In 1970s, large scale integration made personal computers a reality. The computers run operating systems that are anything from simple bootstrap loader with a BASIC or FORTH interpreter glued on to a full fledged operating system with support for executing multiple programs for multiple users on multiple computers connected by a network.

Hardware	Year	Software
	1976	*Control Program/Memory (CP/M)* - an operating system developed by Gary Kildall at Intergalactic Digital Research, later renamed to just Digital Research :-).
IBM PC - a computer developed by IBM.	1981	*MS-DOS* - an operating system developed at Microsoft.
ZX Spectrum - a computer developed by Richard Altwasser at Sinclair Research.	1982	
	1984	*Finder* - an operating system developed by Steve Capps at Apple.

Basic Concepts

Historically, a contemporary operating system combines the functions of an extended machine and a resource manager. The extended machine separates applications from the low-level platform-dependent details by providing high-level platform-independent abstractions such as windows, sockets, files. The resource manager separates applications from each other by providing mechanisms such as sharing and locking.

Both the extended machine and the resource manager rely on established hardware concepts to build operating system structure and provide operating system abstractions. The following sections summarize well known concepts found in contemporary hardware and well known structure and abstractions found in contemporary operating systems. The sections are styled as a crash course on things either known in general or outside the scope of this book, presented to familiarize the reader with the background and terminology.

Needless to say, none of the things outlined here is definitive. Rather than that, they simply appear as good solutions at this time and can be replaced by better solutions any time in the future.

Hardware Building Blocks

Contemporary hardware builds on semiconductor logic. One of the basic elements of the semiconductor logic is a transistor, which can act as an active switching element, creating a connection between its collector and emitter pins when current is injected into its base pin. The transistor can be used to build gates that implement simple logical functions such as AND and OR, as sketched below.

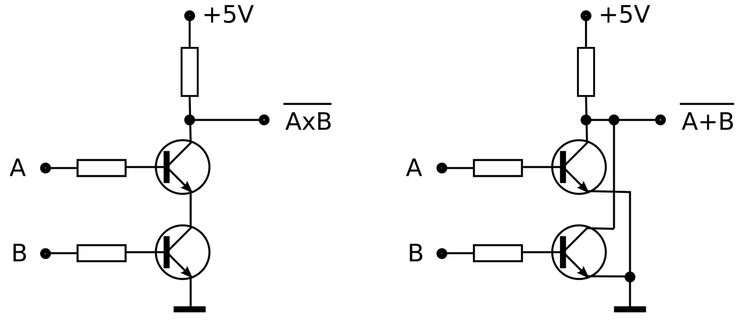

Figure 1-1. Principle Of Composing NAND And NOR Gates From Transistors

Note that the principal illustration uses bipolar transistors in place of more practical field effect transistors, and a simplified composition out of individual transistors in place of more practical direct gate construction.

Gates that implement simple logical functions can be used to construct more complex functions, such as buffers, shifters, decoders, arithmetic units and other circuits. The illustration of constructing a flip flop, which in fact represents a single bit of memory, is below.

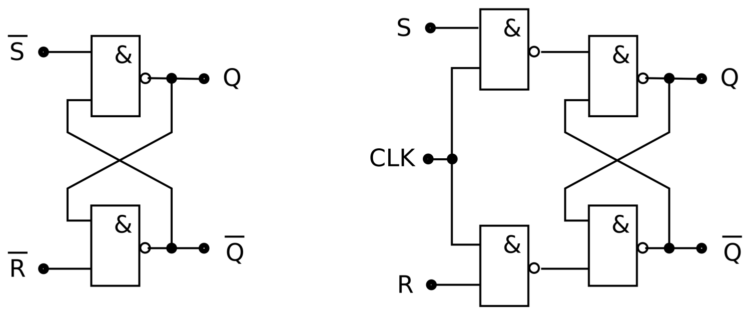

Figure 1-2. Principle Of Composing Flip Flops From Gates

Note that besides well known construction patterns of useful circuits, approaches to design a circuit given the required logical function are also very well established. In fact, many circuits are sold as designs rather than chips, the designs are merged depending on the application and only then embedded in multipurpose chips.

References

1. Ken Bigelow: Play Hookey Digital Logic Tutorial. http://www.play-hookey.com/digital

Basic Computer Architecture

The figure depicts a basic architecture of a desktop computer available in the late 1970s. The architecture is simple but still representative of the contemporary desktop computers. Advances in the architecture since 1970s are outlined in the subsequent sections.

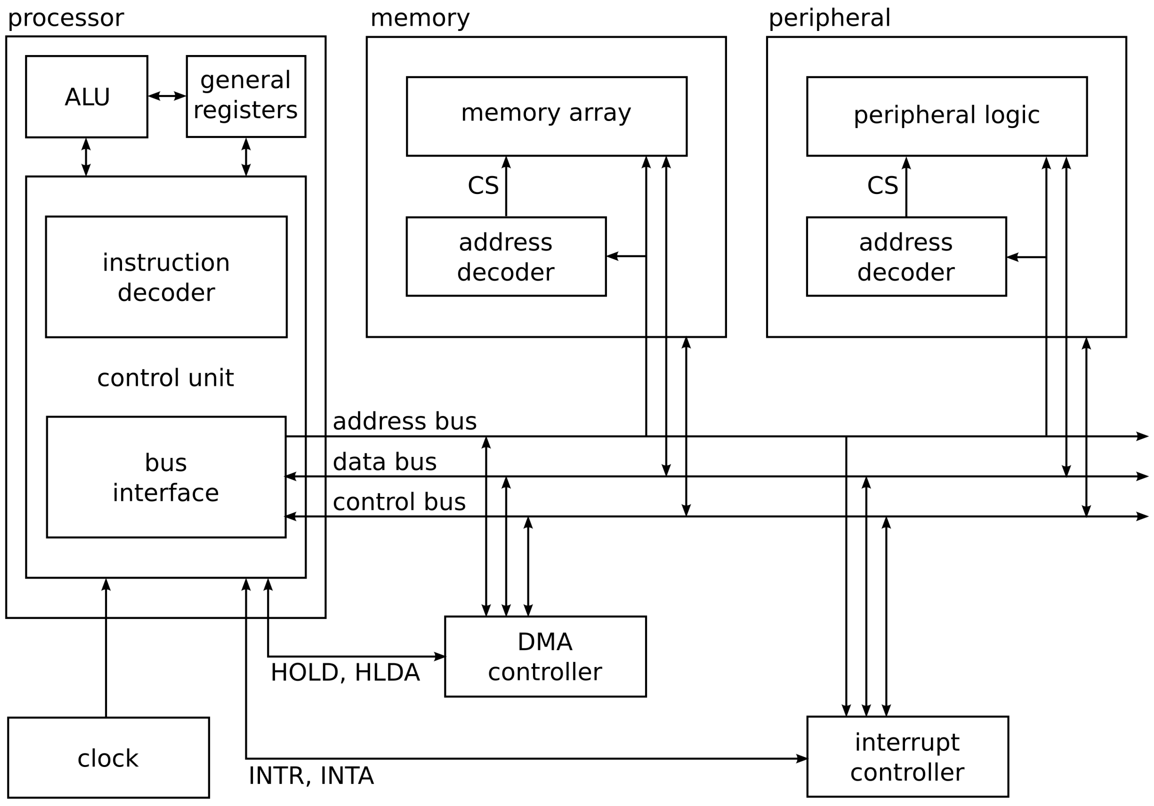

Figure 1-3. Basic Computer Architecture Example

At the core of the architecture is the control unit of the processor. In steps timed by the external clock signal, the control unit repeats an infinite cycle of fetching a code of the instruction to be executed from memory, decoding the instruction, fetching the operands of the instruction, executing the instruction, storing the results of the instruction. The control unit uses the arithmetic and logic unit to execute arithmetic and logic instructions.

Processor Bus

The control unit of the processor communicates with the rest of the architecture through a processor bus, which can be viewed as consisting of three distinct sets of wires denoted as address bus, data bus and control bus. The address bus is a set of wires used to communicate an address. The data bus is a set of wires used to communicate data. The control bus is a set of wires with functions other than those of the address and data buses, especially signals that tell when the information on the address and data buses is valid.

The exact working of the processor bus can be explained by a series of timing diagrams for basic operations such as memory read and memory write.

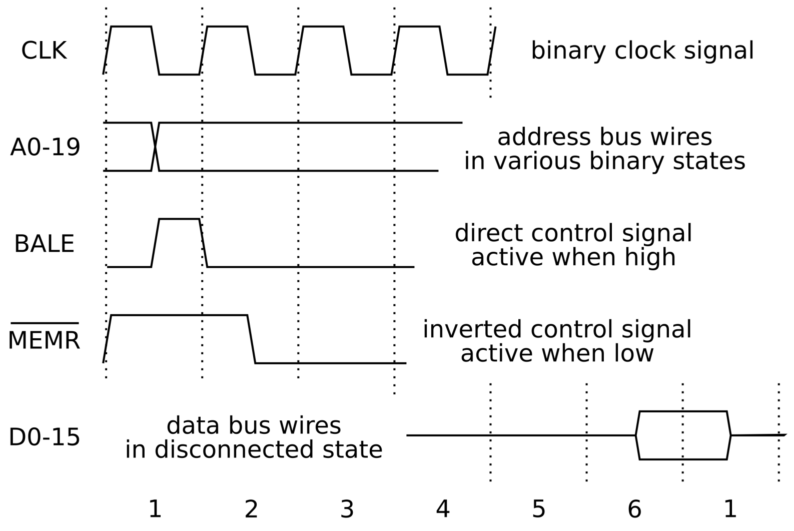
CLK
binary clock signal
A0-19
address bus wires in various binary states
BALE
direct control signal active when high
MEMR
inverted control signal active when low
D0-15
data bus wires in disconnected state
1
2
3
4
5
6
1

Figure 1-4. Timing Diagram Example

What all operations of the processor bus have in common is the general order of steps, which typically starts with the processor setting an address on the address bus and a signal on the control bus that indicates presence of a valid address, and proceeds with the transfer of data. Any device connected to the processor bus is responsible for recognizing its address, usually through an address decoder that sends the chip select signal when the address of the device is recognized.

Example: ISA Bus

The ISA (Industry Standard Architecture) bus is synchronized by a clock signal ticking with the frequency of 8-10 MHz. In the first clock tick of a bus cycle, the bus master, which is typically the processor, sets the address on the address bus and pulses the BALE (Bus Address Latch Enable) signal to indicate that the address is valid.

In a read bus cycle, the bus master activates one of the MEMR (Memory Read) or IOR (Input/Output Read) signals to indicate either reading from memory or reading from an input device. The bus master waits the next four cycles for the memory or the device to recognize its address and set the data on the data bus.

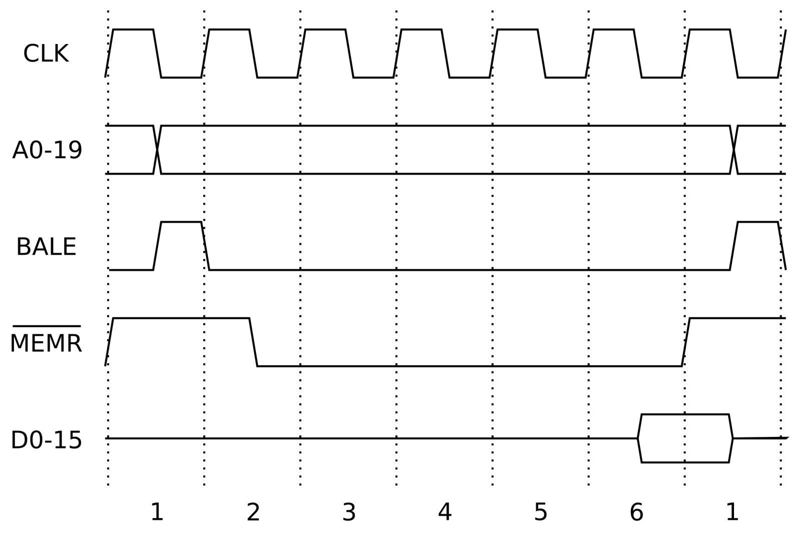

Figure 1-5. ISA Bus Read Cycle

In a write bus cycle, the bus master activates one of the MEMW (Memory Write) or IOW (Input/Output Write) signals to indicate either writing to memory or writing to an output device. The bus master sets the data on the data bus and waits the next four cycles for the memory or the device to recognize its address and data.

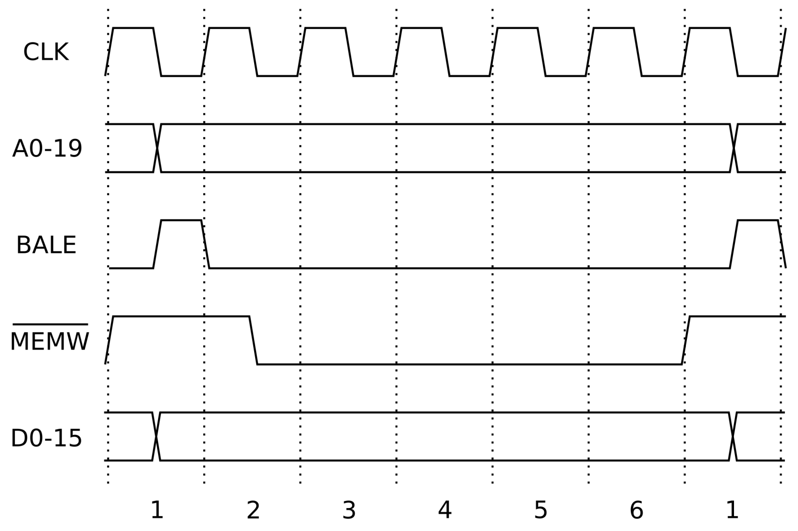

Figure 1-6. ISA Bus Write Cycle

Interrupt Controller

To provide means of requesting attention from outside, the processor is equipped with the interrupt and interrupt acknowledge signals. Before executing an instruction, the control unit of the processor checks whether the interrupt signal is set, and if it is, the control unit responds with setting the interrupt acknowledge signal and setting the program counter to a predefined address, effectively executing a subroutine call instruction.

To better cope with situations where more devices can request attention, the handling of the interrupt request signal is delegated to an interrupt controller. The controller has several interrupt request inputs and takes care of aggregating those inputs into the interrupt request input of the processor using priorities and queuing and providing the processor with information to distinguish the interrupt requests.

Direct Memory Access Controller

To provide means of transferring data without processor attention, the processor is equipped with the hold and hold acknowledge signals. The control unit of the processor checks whether the hold signal is set, and if it is, the control unit responds with setting the hold acknowledge signal and holding access to the processor bus until the hold signal is reset, effectively relinquishing control of the processor bus.

To better cope with situations where more devices can transfer data without processor attention, the handling of the hold signal is delegated to a direct memory access controller. The controller has several transfer request inputs associated with transfer counters and takes care of taking over the processor bus and setting the address and control signals during transfer.

Example: ISA Bus

The ISA (Industry Standard Architecture) bus DMA cycle is commenced by the peripheral device requesting a transfer using one of the DRQ (DMA Request) signals. There are 4 or 8 DRQ signals, DRQ0 to DRQ3 or DRQ7, and 4 or 8 corresponding DACK (DMA Acknowledge) signals, DACK0 to DACK3 or DACK7, each associated with one set of transfer counters in the controller.

When the controller sees the peripheral device requesting a transfer, it asks the processor to relinquish the bus using the HRQ (Hold Request) signal. The processor answers with the HLDA (Hold Acknowledge) signal and relinquishes the bus. This typically happens at the end of a machine cycle.

Once the bus is not used by the processor, the controller performs the device-to-memory or memory-to-device bus transfer in a manner that is very similar to the normal processor-to-memory or memory-to-processor bus transfer. The controller sends the memory address on the address bus together with the AEN (Address Enable) signal to indicate that the address is valid, responds to the peripheral device requesting the transfer using one of the DACK signals, and juggles the MEMW and IOR or the MEMR and IOW signals to synchronize the transfer.

Advances In Processor Architecture

Instruction Pipelining

The basic architecture described earlier executes an instruction in several execution phases, typically fetching a code of the instruction, decoding the instruction, fetching the operands, executing the instruction, storing the results. Each of these phases only employs some parts of the processor and leaves other parts idle. The idea of instruction pipelining is to process several instructions concurrently so that each instruction is in a different execution phase and thus employs different parts of the processor. The instruction pipelining yields an increase in speed and efficiency.

For illustration, Intel Pentium 3 processors sported a 10 stage pipeline, early Intel Pentium 4 processors have extended the pipeline to 20 stages, late Intel Pentium 4 processors use 31 stages. AMD Opteron processors use 12 stages pipeline for fixed point instructions and 17 stages pipeline for floating point instructions. Note that it is not really correct to specify a single pipeline length, since the number of stages an instruction takes depends on the particular instruction.

One factor that makes the instruction pipelining more difficult are the conditional jumps. The instruction pipelining fetches instructions in advance and when a conditional jump is reached, it is not known whether it will be executed or not and what instructions should be fetched following the conditional jump. One solution, statistical prediction of conditional jumps is used. (AMD Athlon processors and Intel Pentium processors do this. AMD Hammer processors keep track of past results for 65536 conditional jumps to facilitate statistical prediction.) Another solution, all possible branches are prefetched and the incorrect ones are discarded.

Superscalar Execution

An increase in speed and efficiency can be achieved by replicating parts of the processor and executing instructions concurrently. The superscalar execution is made difficult by dependencies between instructions, either when several concurrently executing instructions employ the same parts of the processor, or when an instruction uses results of another concurrently executing instruction. Both collisions can be solved by delaying some of the concurrently executing instructions, thus decreasing the yield of the superscalar execution.

An alternative solution to the collisions is replicating the part in the processor. For illustration, Intel Core Duo processors are capable of executing four instructions at once under ideal conditions. Together with instruction pipelining, AMD Hammer processors can execute up to 72 instructions in various stages.

An alternative solution to the collisions is reordering the instructions. This may not always be possible in one thread of execution as the instructions in one thread typically work on the same data. (Intel Pentium Pro processors do this.)

An alternative solution to the collisions is splitting the instructions into micro instructions that are scheduled independently with a smaller probability of collisions. (AMD Athlon processors and Intel Pentium Pro processors do this.)

An alternative solution to the collisions is mixing instructions from several threads of execution. This is attractive especially because instructions in several threads typically work on different data. (Intel Xeon processors do this.)

An alternative solution to the collisions is using previous values of the results. This is attractive especially because the processor remains simple and a compiler can reorder the instructions as necessary without burdening the programmer. (MIPS RISC processors do this.)

References

1. Agner Fog: Software Optimization Resources. http://www.agner.org/optimize

Advances In Memory Architecture

Virtual Memory

Virtual memory makes it possible to create a virtual view of memory by defining a mapping between virtual and physical addresses. Instructions access memory using virtual addresses that the hardware either translates to physical addresses or recognizes as untranslatable and requests the software to supply the translation.

Instruction And Data Caching

Memory accesses used to fetch instructions and their operands and results exhibit locality of reference. A processor can keep a copy of the recently accessed instructions and data in a cache that can be accessed faster than other memory.

A cache is limited in size by factors such as access speed and chip area. A cache may be fully associative or combine a limited degree of associativity with hashing. Multiple levels of caches with different sizes and speeds are typically present to accommodate various memory access patterns.

The copy of the instructions and data in a cache must be coherent with the original. This is a problem when other devices than a processor access memory. A cache coherency protocol solves the problem by snooping on the processor bus and either invalidating or updating the cache when an access by other devices than a processor is observed.

Instruction And Data Prefetching

TODO

Advances In Bus Architecture

Burst Access

The operations of the processor bus were optimized for the common case of transferring a block of data from consecutive addresses. Rather than setting an address on the address bus for each item of the block, the address is only set once at the beginning of the transfer and the block of data is transferred in a burst.

Example: PCI Bus

The PCI (Peripheral Component Interconnect) bus transfers multiple units of data in frames. A frame begins with transporting an address and a command that describes what type of transfer will be done within the frame. Next, data is transferred in bursts of limited maximum duration.

In a single bus cycle, the bus master activates the FRAME signal to denote the start of the cycle, sets the C/BE (Command / Byte Enable) wires to describe the type of transfer (0010 for device read, 0011 for device write, 0110 for memory read, 0111 for memory write, etc.) and sets the A/D (Address / Data) wires to the address to be read from or written to.

After the address and the command is transferred, the bus master uses the IRDY (Initiator Ready) signal to indicate readiness to receive or send data. The target of the transfer responds with DEVSEL (Device Select) to indicate that it has been addressed, and with TRDY (Target Ready) to indicate readiness to send or receive data. When both the initiator and the target are ready, one unit of data is transferred each clock cycle.

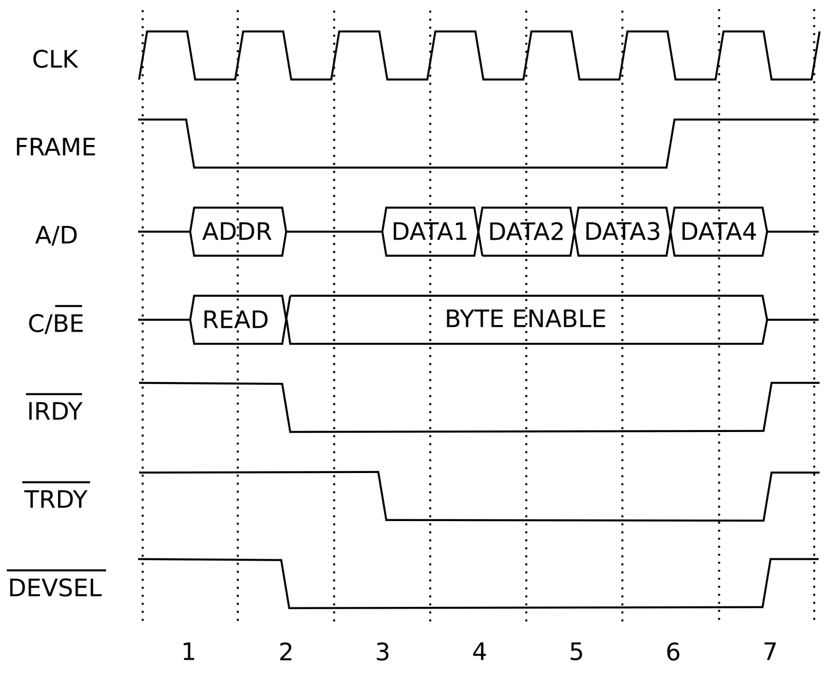

Figure 1-7. PCI Bus Read Cycle

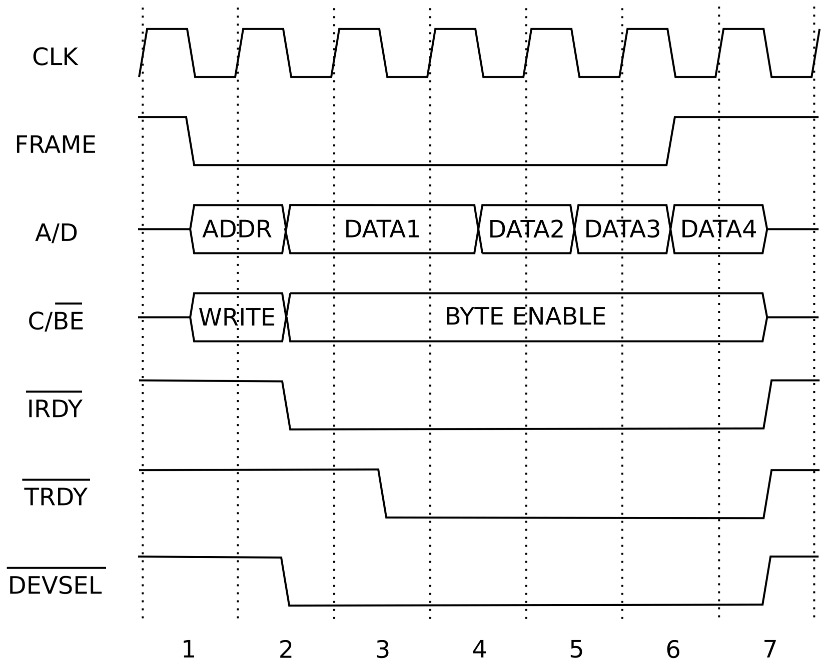

Figure 1-8. PCI Bus Write Cycle

Note that there are many variants of the PCI bus. AGP (Accelerated Graphics Port) is based on PCI clocked at 66 MHz and doubles the speed by transporting data on both the rising and the falling edge of the clock signal. PCI-X (Peripheral Component Interconnect Extended) introduces higher clock speeds and ECC for error checking and correction.

Bus Mastering

Multiprocessor systems and complex devices do not fit the concept of a single processor controlling the processor bus. An arbitration mechanism is introduced to allow any device to request control of the processor bus. (PCI has an arbitrator who can grant the use of the bus to any connected device.)

Example: ISA Bus

The ISA (Industry Standard Architecture) bus DMA cycle can be extended to support bus mastering. After the controller finished the DRQ and DACK handshake, the peripheral device could use the MASTER signal to request bus mastering. The controller responded by relinquishing control of the bus to the peripheral device. Although not typical, this mechanism has been used for example by high end network hardware.

Multiple Buses

The speed of the processor differs from the speed of the memory and other devices. To compensate for the difference, multiple buses are introduced in place of the processor bus from the basic architecture described earlier. (PC has a north bridge that connects processor to memory and graphic card on AGP at one speed and to south bridge at another speed, and a south bridge that connects integrated devices to north bridge at one speed and PCI slots at another speed.)

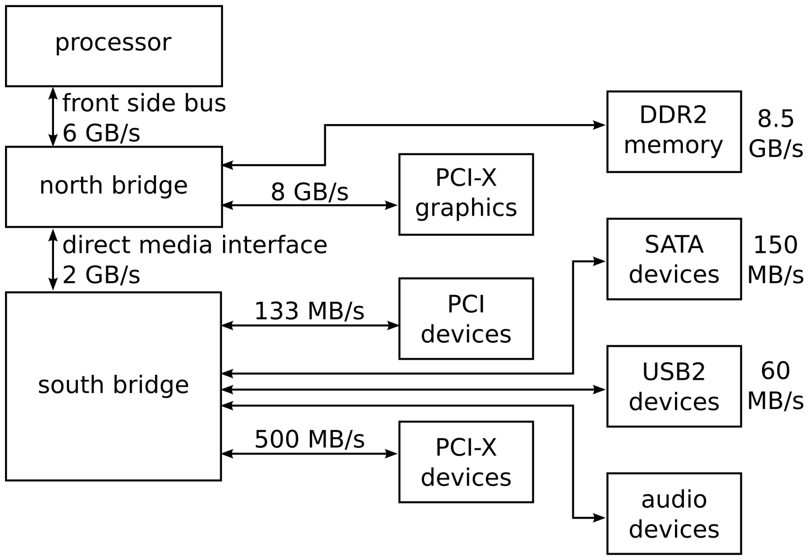

Figure 1-9. Multiple Buses Example

Operating System Structure

The design of an operating system architecture traditionally follows the *separation of concerns* principle. This principle suggests structuring the operating system into relatively independent parts that provide simple individual features, thus keeping the complexity of the design manageable.

Besides managing complexity, the structure of the operating system can influence key features such as robustness or efficiency:

- The operating system posesses various privileges that allow it to access otherwise protected resources such as physical devices or application memory. When these privileges are granted to the individual parts of the operating system that require them, rather than to the operating system as a whole, the potential for both accidental and malicious privileges misuse is reduced.
- Breaking the operating system into parts can have adverse effect on efficiency because of the overhead associated with communication between the individual parts. This overhead can be exacerbated when coupled with hardware mechanisms used to grant privileges.

The following sections outline typical approaches to structuring the operating system.

Monolithic Systems

A monolithic design of the operating system architecture makes no special accommodation for the special nature of the operating system. Although the design follows the separation of concerns, no attempt is made to restrict the privileges granted to the individual parts of the operating system. The entire operating system executes with maximum privileges. The communication overhead inside the monolithic operating system is the same as the communication overhead inside any other software, considered relatively low.

CP/M and DOS are simple examples of monolithic operating systems. Both CP/M and DOS are operating systems that share a single address space with the applications. In CP/M, the 16 bit address space starts with system variables and the application area and ends with three parts of the operating system, namely CCP (Console Command Processor), BDOS (Basic Disk Operating System) and BIOS (Basic Input/Output System). In DOS, the 20 bit address space starts with the array of interrupt vectors and the system variables, followed by the resident part of DOS and the application area and ending with a memory block used by the video card and BIOS.

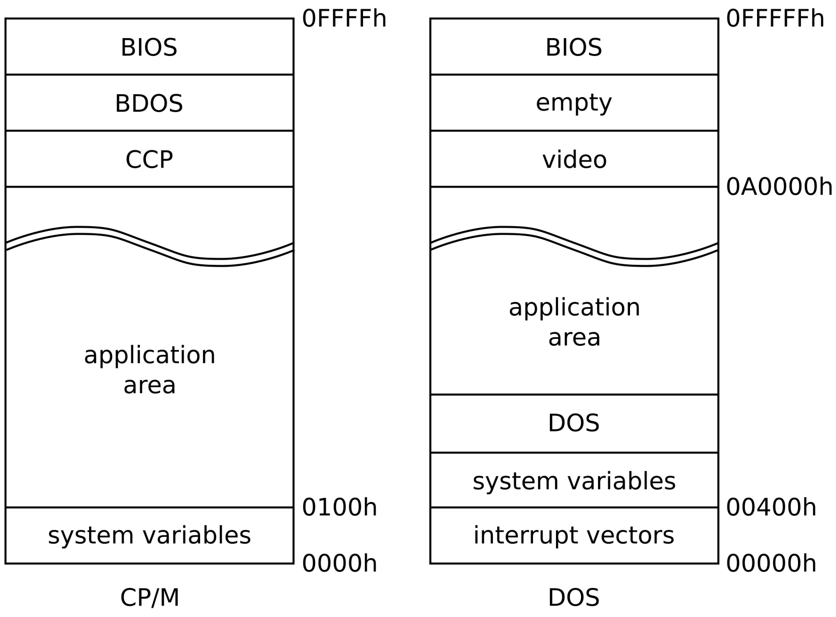

Figure 1-10. Simple Monolithic Operating Systems Example

Most contemporary operating systems, including Linux and Windows, are also considered monolithic, even though their structure is certainly significantly different from the simple examples of CP/M and DOS.

References

1. Tim Olmstead: Memorial Digital Research CP/M Library. http://www.cpm.z80.de/drilib.html

Layered Systems

A layered design of the operating system architecture attempts to achieve robustness by structuring the architecture into layers with different privileges. The most privileged layer would contain code dealing with interrupt handling and context switching, the layers above that would follow with device drivers, memory management, file systems, user interface, and finally the least privileged layer would contain the applications.

MULTICS is a prominent example of a layered operating system, designed with eight layers formed into *protection rings*, whose boundaries could only be crossed using specialized instructions. Contemporary operating systems, however, do not use the layered design, as it is deemed too restrictive and requires specific hardware support.

References

1. Multicians, http://www.multicians.org

Microkernel Systems

A microkernel design of the operating system architecture targets robustness. The privileges granted to the individual parts of the operating system are restricted as much as possible and the communication between the parts relies on a specialized communication mechanisms that enforce the privileges as necessary. The communication overhead inside the microkernel operating system can be higher than the communication overhead inside other software, however, research has shown this overhead to be manageable.

Experience with the microkernel design suggests that only very few individual parts of the operating system need to have more privileges than common applications. The microkernel design therefore leads to a small system kernel, accompanied by additional system applications that provide most of the operating system features.

MACH is a prominent example of a microkernel that has been used in contemporary operating systems, including the NextStep and OpenStep systems and, notably, OS X. Most research operating systems also qualify as microkernel operating systems.

References

1. The Mach Operating System. http://www.cs.cmu.edu/afs/cs.cmu.edu/project/mach/public/www
2. Andrew Tannenbaum, Linus Torvalds: Debate On Linux. http://www.oreilly.com/catalog/opensources/book/appa.html

Virtualized Systems

Attempts to simplify maintenance and improve utilization of operating systems that host multiple independent applications have lead to the idea of running multiple operating systems on the same computer. Similar to the manner in which the operating system kernel provides an isolated environment to each hosted application, virtualized systems introduce a *hypervisor* that provides an isolated environment to each hosted operating system.

Hypervisors can be introduced into the system architecture in different ways.

- A *native* hypervisor runs on bare hardware, with the hosted operating systems residing above the hypervisor in the system structure. This makes it possible to implement an efficient hypervisor, paying the price of maintaining a hardware specific implementation.
- A *hosted* hypervisor partially bypasses the need for a hardware specific implementation by running on top of another operating system. From the bottom up, the system structure then starts with the host operating system that includes the hypervisor, and then the guest operating systems, hosted above the hypervisor.

A combination of the native and the hosted approaches is also possible. The hypervisor can implement some of its features on bare hardware and consult the hosted operating systems for its other features. A common example of this approach is to implement the processor virtualization support on bare hardware and use a dedicated hosted operating system to access the devices that the hypervisor then virtualizes for the other hosted operating systems.

Chapter 2. Process Management

Process Alone

Before delving into how multiple processes are run in parallel and how such processes communicate and synchronize, closer look needs to be taken at what exactly a process is and how a process executes.

Process And Thread Concepts

An obvious function of a computer is executing programs. A program is a sequence of instructions that tell the computer what to do. When a computer executes a program, it keeps track of the position of the currently executing instruction within the program and of the data the instructions of the program use. This gives rise to the concept of a process as an executing program, consisting of the program itself and of the execution state that the computer keeps track of.

The abstract notions of program and state are backed by concrete entities. The program is represented as machine code instructions, which are stored as numbers in memory. The machine code instructions manipulate the state, which is also stored as numbers, either in memory or in registers.

It is often useful to have several processes cooperate. A cooperation between processes requires communication, which may be cumbersome when each process has a completely distinct program and a completely distinct state - the programs have to be kept compatible and the information about the state must be exchanged explicitly. The need to simplify communication gives rise to the concept of threads as activities that share parts of program and parts of state within a process.

The introduction of threads redefines the term process. When speaking about processes alone, the term process is used to refer to both the program and state as passive entities and the act of execution as an active entity. When speaking about processes and threads together, the term process is used to refer to the program and state as a passive entity and the term thread is used to refer to the act of execution as an active entity. A process is a passive shell that contains active threads executing within it.

Starting A Process

Starting a process means loading the program and initializing the state. The program typically expects to begin executing from a specific instruction with only the static variables initialized. The program then initializes the local and heap variables as it executes. Starting a process therefore boils down to loading the program code and the static variables, together called the program image, and setting the position of the currently executing instruction within the program to the instruction where the program expects to begin executing.

Bootstrapping

The first catch to starting a process comes with the question of who loads the program image. The typical solution of having a process load the program image of another process gets us to the question of who loads the program image of the very first process to be started. This process is called the bootstrap process and the act of starting the bootstrap process is called *bootstrapping.*

The program image of the bootstrap process is typically stored in the fixed memory of the computer by the manufacturer. Any of the ROM, PROM, EEPROM or FLASH type memory chips, which keep their contents even with the power switched off, can be used for this purpose. The processor of the computer is hardwired to start executing instructions from a specific address when the power is switched on, the fixed memory with the program image of the bootstrap process is therefore hardwired to reside on the same address.

Computers designed for a specific application purpose can have that purpose implemented by the bootstrap process. Such approach, however, would be too limiting for computers designed for general use, which is why the bootstrap process typically only initializes the hardware and starts another process, whose program image is loaded from whatever source the bootstrap process supports.

Example: Booting IBM PC

The IBM PC line of computers uses the Intel 80x86 line of processors, which start executing from address FFF...FFF0h (exact address depending on the address bus width and hence on the processor model). A fixed memory with BIOS program image resides at that address. The BIOS process initializes and tests the hardware of the computer as necessary and looks for the next process to start.

In the early models of the IBM PC line of computers, the BIOS process expected the program image of the next process to start to reside in the first sector of the first disk connected to the computer, have exactly 512 bytes in size and end with a two byte signature of 55AAh. The BIOS process loaded the program image into memory at address 7C00h and if the two byte signature was present, the BIOS process then begun executing the next process to start from address 7C00h.

In many cases, the fixed size of 512 bytes is too small for the program image of the next process to start. The next process to start is therefore yet another bootstrap process, which loads the program image of the next process to start. This repeats until the operating system itself is loaded. The reason for having a sequence of bootstrap processes rather than a single bootstrap process that loads the operating system straight away is that loading the program image of the operating system requires knowing the structure of the program image both on disk and in memory. This structure depends on the operating system itself and hardcoding the knowledge of the structure in a single bootstrap process which resides in fixed memory would limit the ability of the computer to load an arbitrary operating system.

Example: Booting UEFI

To be done.

Example: Booting MSIM

To be done.

Relocating

The act of loading a program image is further complicated by the fact that the program image may have been constructed presuming that it will reside at a specific range of addresses, and may contain machine code instructions or static variables that refer to specific addresses from that range, using what is denoted as absolute addressing.

Declaring and accessing a global variable in C.

```
static int i;           // declare a global variable
...
i = 0x12345678;         // access the global variable
```

The C code compiled into Intel 80x86 assembler.

```
.comm i,4,4              ;declare i as 4 bytes aligned at 4 bytes boundary
...
movl $0x12345678,i       ;write value 12345678h into target address i
```

The assembler code compiled into Intel 80x86 machine code.

```
C705                     ;movl
C0950408                 ;target address 080495C0h
78563412                 ;value 12345678h
```

Figure 2-1. Absolute Addressing Example

When a program image uses absolute addressing, it must be loaded at the specific range of addresses it has been constructed for. Unfortunately, it is often necessary to load program images at arbitrary ranges of addresses, for example when multiple program images are to share one address space. This requires adjusting the program image by fixing all machine code instructions and static variables that refer to specific addresses using absolute addressing. This process is called *relocation*.

The need for relocation can be alleviated by replacing absolute addressing, which stores addresses as absolute locations in machine code instructions, with relative addressing, which stores addresses as relative distances in machine code instructions. The program image is said to contain *position independent code* when it does not need relocation. Constructing position independent code usually comes at a price, however, because in some cases, relative addressing might be more cumbersome than absolute addressing.

Declaring and accessing a global variable in C.

```
static int i;            // declare a global variable
...
i = 0;                   // access the global variable
```

The C code compiled into position independent Intel 80x86 assembler.

```
.comm i,4,4              ;declare i as 4 bytes aligned at 4 bytes boundary
...
call __get_thunk         ;get program starting address in ECX
addl $_GOT_,%ecx         ;calculate address of global table of addresses in ECX
movl $0,i@GOT(%ecx)      ;write value 0 into target address i relative from ECX
```

The assembler code compiled into position independent Intel 80x86 machine code.

```
E8                       ;call
1C000000                 ;target address 0000001Ch distant from here
81C1                     ;addl target ECX
D9110000                 ;value 000011D9h
C781                     ;movl target address relative from ECX
20000000                 ;target address 00000020h distant from ECX
00000000                 ;value 00000000h
```

Figure 2-2. Relative Addressing Example

Example: Program Image In CP/M

CP/M avoids the need for relocation by expecting a program to always start at the same address, namely 100h. The file with the program image consisted of the program code and the static variables, stored exactly in the same form as when the program is executing in memory.

Example: Program Image In Intel HEX

When the program image was to be stored on media that was originally designed for storing text, such as some types of paper tapes or punch cards, formats such as Intel HEX were used. A program image in Intel HEX consisted of lines starting with a colon and followed by a string of hexadecimal digits.

```
:LLAAAATTxxxxCC
```

In the string, LL is the length of the data, AAAA is the address of the data in memory, TT indicates the last line, the data itself is followed by checksum CC. The data consists of the program code and the static variables, stored exactly in the same form as when the program is executing in memory.

Example: Program Image In DOS

For small programs, DOS employs the segmentation support of the Intel 80x86 processors to avoid the need for relocation. A program is expected to fit into a single segment and to always start at the same address within the segment, namely 100h. The file with the program image consisted of the program code and the static variables, stored exactly in the same form as when the program is executing in memory.

For large programs, DOS introduced the EXE format. Besides the program code and the static variables, the file with the program image also contained a relocation table. The relocation table was a simple list of locations within the program image that need to be adjusted, the adjustment being a simple addition of the program base address to the location. Besides the relocation table, the header of the file also contained the required memory size and the relative starting address.

Offset	Length	Contents
00h	2	Magic (0AA55h)
02h	2	Length of last block
04h	2	Length of file in 512B blocks (L)
06h	2	Number of relocation table entries (R)
08h	2	Length of header in 16B blocks (H)
0Ah	2	Minimum memory beyond program image in 16B blocks
0Ch	2	Maximum memory beyond program image in 16B blocks
0Eh	4	Initial stack pointer setting (SS:SP)
12h	2	File checksum
14h	4	Initial program counter setting (CS:IP)
18h	2	Offset of relocation table (1Ch)
1Ah	2	Overlay number
1Ch	R*4h	Relocation table entries
H*10h	L*200h	Program image

Figure 2-3. DOS EXE Format

Linking

It is common for many programs to share the same libraries. The libraries can be linked to the program either statically, during compilation, or dynamically, during execution. Both approaches can have advantages, static linking creates independent program images robust to system upgrades, dynamic linking creates small program images efficient in memory usage. Both approaches require program image formats that support linking by listing *exported* symbols that the program image provides and *external* symbols that the program image requires.

Example: Executable And Linking Format

ELF is a binary executable file format originally developed for UNIX System V. Gradually, it became the binary executable file format of choice for most UNIX systems. It supports multiple system architectures and can store both modules for linking and programs for execution. It can also carry supplemental information such as debugging data.

An ELF file partitions data into *sections* and *segments*. Sections provide logical separation - there are separate sections for code, symbols, relocations, debugging, and so on. Segments provide structure for execution - there are segments to be loaded in memory, segments for dynamic linking, and so on. Sections and segments are defined in header tables that point to data elsewhere in the file. Sections and segments can also point to the same data and thus overlap.

Headers

The ELF header is the very first part of an ELF file and describes the structure of the file. Besides the usual magic number that identifies the ELF format, it tells the exact type of the file including address size and data encoding, and the processor architecture that the program is for. Other important fields include the starting address of the program.

```
> readelf --file-header /bin/bash

ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2’s complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x41d238
  Start of program headers:          64 (bytes into file)
  Start of section headers:          964960 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         10
  Size of section headers:           64 (bytes)
  Number of section headers:         32
  Section header string table index: 31

> readelf --file-header /lib/libc.so.6

ELF Header:
  Magic:   7f 45 4c 46 01 01 01 03 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2’s complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - GNU
  ABI Version:                       0
  Type:                              DYN (Shared object file)
  Machine:                           Intel 80386
  Version:                           0x1
  Entry point address:               0x44564790
  Start of program headers:          52 (bytes into file)
  Start of section headers:          2009952 (bytes into file)
  Flags:                             0x0
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         10
  Size of section headers:           40 (bytes)
  Number of section headers:         43
  Section header string table index: 42
```

Sections

The section header table lists all the sections of the file. Each section has a name, a type, a position and length within the file, and flags.

```
> readelf --sections /lib/libc.so.6

There are 43 section headers, starting at offset 0x1eab60:

Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
...
  [ 9] .rel.dyn          REL             4455f3e4 0143e4 002a18 08   A  4   0  4
  [10] .rel.plt          REL             44561dfc 016dfc 000058 08   A  4  11  4
  [11] .plt              PROGBITS        44561e60 016e60 0000c0 04  AX  0   0 16
  [12] .text             PROGBITS        44561f20 016f20 14010c 00  AX  0   0 16
...
  [32] .data             PROGBITS        446f9040 1ad040 000e7c 00  WA  0   0 32
  [33] .bss              NOBITS          446f9ec0 1adebc 002bfc 00  WA  0   0 32
  [34] .comment          PROGBITS        00000000 1adebc 00002c 01  MS  0   0  1
  [35] .note.stapsdt     NOTE            00000000 1adee8 0002c4 00      0   0  4
  [36] .symtab           SYMTAB          00000000 1ae1ac 021880 10     37 6229  4
  [37] .strtab           STRTAB          00000000 1cfa2c 01a786 00      0   0  1
...
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings)
  I (info), L (link order), G (group), T (TLS), E (exclude), x (unknown)
  O (extra OS processing required) o (OS specific), p (processor specific)
```

Important sections typically present in a file include:

bss

A section that represents the uninitialized memory of the program.

data

A section that contains the static variables.

text

A section that contains the program code.

init and fini

Sections that contain the program code responsible for initialization and termination.

symtab and dynsym

Sections that contain the symbol tables for static and dynamic linking. Symbol tables list symbols defined by the program. A symbol is defined by its name, value, size and type.

strtab and dynstr

Sections that contain the string tables for static and dynamic linking. String tables list all strings used in the file. A string is referred to using its index in the table rather than quoted.

rel

Sections that contain the relocation information. Relocations are defined by their position, size and type.

```
> readelf --relocs /lib/libc.so.6

Relocation section '.rel.dyn' at offset 0x134e4 contains 1457 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
436f71b0  00000008 R_386_RELATIVE
436f8e74  0000000e R_386_TLS_TPOFF
436f8e90  00058206 R_386_GLOB_DAT    436f9d7c   stderr
436f9008  0004de07 R_386_JUMP_SLOT   435c3b70   malloc
...
```

Segments

The program header table lists all the segments of the file. Each segment has a type, a position and length in the file, an address and length in memory, and flags. The content of a segment is made up of sections. Examples of important types include:

loadable

A segment that will be loaded into memory. Data within a loadable segment must be aligned to page size so that the segment can be mapped rather than loaded.

dynamic

A segment that contains the dynamic linking information. The segment is used by the dynamic loader that is specified in the interpreter segment.

interpreter

A segment that identifies the program interpreter. When specified, the program interpreter is loaded into memory. The program interpreter is responsible for executing the program itself.

```
> readelf --segments /bin/bash

Elf file type is EXEC (Executable file)
Entry point 0x41d238
There are 10 program headers, starting at offset 64

Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  PHDR           0x0000000000000040 0x0000000000400040 0x0000000000400040
                 0x0000000000000230 0x0000000000000230  R E    8
  INTERP         0x0000000000000270 0x0000000000400270 0x0000000000400270
                 0x000000000000001c 0x000000000000001c  R      1
      [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
  LOAD           0x0000000000000000 0x0000000000400000 0x0000000000400000
                 0x00000000000d9cd4 0x00000000000d9cd4  R E    200000
  DYNAMIC        0x00000000000d9df0 0x00000000006d9df0 0x00000000006d9df0
                 0x00000000000001f0 0x00000000000001f0  RW     8
  STACK          0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000000 0x0000000000000000  RW     8
...

 Section to Segment mapping:
  Segment Sections...
   00
   01     .interp
   02     .interp .note .dynsym .rela .init .fini .plt .text ...
...

> readelf --dynamic /bin/bash

Dynamic section at offset 0xd9df0 contains 30 entries:
  Tag        Type                         Name/Value
 0x0000000000000001 (NEEDED)             Shared library: [libtinfo.so.5]
 0x0000000000000001 (NEEDED)             Shared library: [libdl.so.2]
 0x0000000000000001 (NEEDED)             Shared library: [libc.so.6]
 0x000000000000000c (INIT)               0x41adb0
 0x000000000000000d (FINI)               0x4a6374
 0x0000000000000019 (INIT_ARRAY)         0x6d9dd8
 0x0000000000000005 (STRTAB)             0x8e2b30
 0x0000000000000006 (SYMTAB)             0x403988
...
```

Miscellanea

The ELF file format also supports special techniques used to minimize the number of pages modified during relocation and linking. These include using global offset table and procedure linkage table.

The *global offset table* is a table created by the dynamic linker that lists all the absolute addresses that the program needs to access. Rather than accessing the absolute addresses directly, the program uses relative addressing to read the address in the global offset table.

The *procedure linkage table* is a table created by the dynamic linker that wraps all the absolute addresses that the program needs to call. Rather than calling the absolute addresses directly, the program uses relative addressing to call the wrapper in the procedure linkage table.

Calling Operating System

A process needs a way to request services of the operating system. The services provided by the operating system are often similar to the services provided by the libraries, and, in the simplest case, are also called similarly. The situation becomes more complex when access to protected resources, such as hardware devices or application memory, must be considered.

Typically, the privileges that govern access to protected resources are enforced by the processor. Depending on the privileges of the currently executing code, the processor decides whether to allow executing instructions that are used to access the protected resources. To prevent malicious code from accessing protected resources, constraints are imposed on the means by which the currently executing code can change its privileges, as well as the means by which less privileged code can call more privileged code.

The operating system posesses various privileges that allow it to access protected resources. Requesting services of the operating system therefore means calling more privileged code from less privileged code and must be subject to the constraints that prevent malicious code from accessing protected resources. These constraints are met by the *system call interface* of the operating system.

- The system call interface must be efficient. Depending on the processor, this can become an issue especially when calling more privileged code from less privileged code, because the constraints imposed by the processor can make the call slow or make the copying of arguments necessary.
- The system call interface must be robust. Malicious code should not be able to trick the operating system into accessing protected resources on its behalf or into denying service.
- The system call interface must be flexible. Features such as wrapping or monitoring services provided by the operating system should be available. Adding new services without changing the system call interface for the existing services should be possible.

Note that services provided through the system call interface are typically wrapped by libraries and thus look as services provided by libraries. This makes it possible to call all services in a uniform way.

Example: CP/M System Call Interface

CP/M run on processors that did not distinguish any privileges. Its system call interface therefore did not have to solve many of the issues related to efficiency and robustness that concern contemporary systems. Instead, the system call interface has been designed with binary compatibility in mind.

When calling the BDOS module, the application placed a number identifying the requested service in register C, other arguments of the requested service in other registers, and called the BDOS entry point at address 5. The entry point contained a jump to the real BDOS entry point which could differ from system to system. The services provided by BDOS included console I/O and FCB based file operations.

```
ReadKey:    mvi     c,1         ; keyboard read service
            call    5           ; call BDOS entry point
            cpi     a,0Dh       ; is returned key code ENTER ?
            jnz     ReadKey     ; repeat keyboard read until it is
```

Figure 2-4. CP/M BDOS System Call Example

When calling the BIOS module, the application placed arguments of the requested service in registers and called the BIOS entry point for the specific service. The entry point for the specific service could differ from system to system, but its distance from the beginning of the BIOS module was the same for all systems.

```
jmp     BOOT            ;cold boot
jmp     WBOOT           ;warm boot
jmp     CONST           ;console status
jmp     CONIN           ;console input
...
jmp     HOME            ;disk head to track 0
...
jmp     SETDMA          ;set memory transfer address
jmp     READ            ;read sector
jmp     WRITE           ;write sector
```

Figure 2-5. CP/M BIOS System Call Entry Points

Example: Intel 80x86 Processor Privileges

The Intel 80x86 processors traditionally serve as an example of why calling more privileged code from less privileged code can be slow:

- On Intel 80286, an average MOV instruction took 2 clock cycles to execute. A call that changed the privilege level took over 80 clock cycles to execute. A call that switched the task took over 180 clock cycles to execute.
- On Intel 80386, an average MOV instruction took 2 clock cycles to execute. A call that changed the privilege level took over 80 clock cycles to execute. A call that switched the task took over 300 clock cycles to execute.

Modern Pentium processors introduce the SYSENTER and SYSEXIT instructions for efficient implementation of the system call interface:

- The SYSENTER instruction sets the stack pointer and instruction pointer registers to values specified by the operating system in advance to point to the operating system code executing at the most privileged level.
- The SYSEXIT instruction sets the stack pointer and instruction pointer registers to values specified by the operating system in registers ECX and EDX to point to the application code executing at the least privileged level.

Note that the SYSENTER and SYSEXIT instructions do not form a complementary pair that would take care of saving and restoring the stack pointer and instruction pointer registers the way CALL and RET instructions do. It is up to the code using the SYSENTER and SYSEXIT instructions to do that.

Example: Linux System Call API On Intel 80x86

The libraries wrapping the system call interface are called in the same way as any other libraries.

```
ssize_t read (int fd, void *buf, size_t count);
...
int hFile;
ssize_t iCount;
char abBuffer [1024];
iCount = read (hFile, &abBuffer, sizeof (abBuffer));

pushl $1024             ;sizeof (abBuffer)
pushl $abBuffer         ;&abBuffer
pushl hFile             ;hFile
call  read@plt          ;call the library
addl  $12,%esp          ;remove arguments from stack
movl  %eax,iCount       ;save result
```

Figure 2-6. Library System Call Example

The system call interface uses either the interrupt vector 80h, which is configured to lead to the kernel through a trap gate, or the SYSENTER and SYSEXIT instructions. In both cases, the EAX register contains a number identifying the requested service and other registers contain other arguments of the requested service.

Since the system call interface is typically called from within the system libraries, having two versions of the system call code would require having two versions of the libraries that contain the system call code. To avoid this redundancy, the system call interface is wrapped by a virtual library called linux-gate, which does not exist as a file, but is inserted by the kernel into the memory map of every process.

```
__kernel_vsyscall: int $0x80
                   ret
```

Figure 2-7. Linux Gate Library Based On INT 80h

```
__kernel_vsyscall: push %ecx
                   push %edx
                   push %ebp
__resume:          mov  %esp,%ebp
                   sysenter

                   jmp  __resume        ;hack for syscall resume

__return:          pop  %ebp            ;this is where
                   pop  %edx             ;the SYSEXIT
                   pop  %ecx              ;returns
                   ret
```

Figure 2-8. Linux Gate Library Based On SYSENTER And SYSEXIT

References

1. Johan Petersson: What Is linux-gate.so.1 ? http://www.trilithium.com/johan/2005/08/linux-gate
2. Linus Torvalds: System Call Restart. http://lkml.org/lkml/2002/12/18/218

Example: Linux Syslet API

A simplified example of reading a file using syslets is copied from Molnar.

References

1. Ingo Molnar: Syslet and Threadlet Patches. http://people.redhat.com/mingo/syslet-patches

Example: Windows System Call API On Intel 80x86

The libraries wrapping the system call interface are called in the same way as any other libraries.

```
int MessageBox (
    HWND hwndOwner,
    LPCTSTR lpszText,
    LPCTSTR lpszTitle,
    UINT fuStyle);
...
MessageBox (0, zMessageText, zWindowTitle, MB_OK || MB_SYSTEMMODAL || MB_ICONHAND);

push MBOK or MB_SYSTEMMODAL or MB_ICONHAND
push offset zWindowTitle
push offset zMessageTest
push 0
call MessageBoxA                ;call the library
add  esp,16                     ;remove arguments from stack
```

Figure 2-9. Library System Call Example

The system call interface uses either the interrupt vector 2Eh or the SYSENTER and SYSEXIT instructions. In both cases, the EAX register contains a number identifying the requested service and the EDX register points to a stack frame holding arguments of the requested service.

What Is The Interface

Typically, the creation and termination of processes and threads is directed by a pair of `fork` and `join` calls. The `fork` call forks a new process or thread off the active process or thread. The `join` call waits for a termination of a process or thread. The exact syntax and semantics depends on the particular operating system and programming language.

Example: Posix Process And Thread API

To create a process, the Posix standard defines the `fork` and `execve` calls. The `fork` call creates a child process, which copies much of the context of the parent process and begins executing just after the `fork` call with a return value of zero. The parent process continues executing after the `fork` call with the return value providing a unique identification of the child process. The child process typically continues by calling `execve` to execute a program different from that of the parent process.

To terminate a process, the Posix standard defines the `exit` and `wait` calls. The `exit` call terminates a process. The `wait` waits for a child process to terminate and returns its termination code. Additional ways for a process to terminate, both voluntarily or involuntarily, exist.

```
pid_t fork (void);
int execve (const char *filename, char *const argv [], char *const envp []);

pid_t wait (int *status);
pid_t waitpid (pid_t pid, int *status, int options);

void exit (int status);
```

The Posix standard call to create a thread is `pthread_create`, which takes the address of the function executed by the thread as its main argument. The `pthread_join` call waits for a thread to terminate, a thread can terminate for example by returning from the thread function or by calling `pthread_exit`. The `pthread_detach` call indicates that `pthread_join` will not be called on the given thread.

```
int pthread_create (
  pthread_t *thread,
  pthread_attr_t *attr,
  void * (*start_routine) (void *),
  void *arg);

int pthread_join (
  pthread_t thread,
  void **return_value);

void pthread_exit (
  void *return_value);

int pthread_detach (
  pthread_t thread);
```

The Posix standard also allows a thread to associate thread local data with a key and to retrieve thread local data of the current thread given the key.

```
int pthread_key_create (
  pthread_key_t *key,
  void (* destructor) (void *));

int pthread_setspecific (
  pthread_key_t key,
  const void *value);
void *pthread_getspecific (
  pthread_key_t key);
```

Example: Windows Process And Thread API

The Windows API provides the `CreateProcess` call to create a process, two of the main arguments of the call are the name of the program file to execute and the command line to supply to the process. The process can terminate by calling `ExitProcess`, the `WaitForSingleObject` call can be used to wait for the termination of a process.

```
BOOL CreateProcess (
  LPCTSTR lpApplicationName,
  LPTSTR lpCommandLine,
  LPSECURITY_ATTRIBUTES lpProcessAttributes,
  LPSECURITY_ATTRIBUTES lpThreadAttributes,
  BOOL bInheritHandles,
  DWORD dwCreationFlags,
  LPVOID lpEnvironment,
  LPCTSTR lpCurrentDirectory,
  LPSTARTUPINFO lpStartupInfo,
  LPPROCESS_INFORMATION lpProcessInformation
);

VOID ExitProcess (
  UINT uExitCode);

DWORD WaitForSingleObject (
  HANDLE hHandle,
  DWORD dwMilliseconds
);
```

Figure 2-10. Windows Process Creation System Calls

Windows applications can create threads using the `CreateThread` call. Besides returning from the thread function, the thread can also terminate by calling `ExitThread`. The universal `WaitForSingleObject` call is used to wait for a thread to terminate.

```
HANDLE CreateThread (
  LPSECURITY_ATTRIBUTES lpThreadAttributes,
  SIZE_T dwStackSize,
  LPTHREAD_START_ROUTINE lpStartAddress,
  LPVOID lpParameter,
  DWORD dwCreationFlags,
  LPDWORD lpThreadId
);

VOID ExitThread (
  DWORD dwExitCode);
```

Figure 2-11. Windows Thread Creation System Calls

Windows also offers fibers as a lightweight variant to threads that is scheduled cooperatively rather than preemptively. Fibers are created using the `CreateFiber` call, scheduled using the `SwitchToFiber` call, and terminated using the `DeleteFiber` call. Before using fibers, the current thread has to initialize the fiber state information using the `ConvertThreadToFiber` call.

```
LPVOID ConvertThreadToFiber (
  LPVOID lpParameter);

LPVOID CreateFiber (
  SIZE_T dwStackSize,
  LPFIBER_START_ROUTINE lpStartAddress,
  LPVOID lpParameter);

VOID SwitchToFiber (
  LPVOID lpFiber);

VOID DeleteFiber (
  LPVOID lpFiber);
```

Windows also allows a thread or a fiber to associate thread local data or fiber local data with a key and to retrieve the data of the current thread or fiber given the key.

```
DWORD TlsAlloc (void);
BOOL TlsFree (
  DWORD dwTlsIndex);

BOOL TlsSetValue (
  DWORD dwTlsIndex,
  LPVOID lpTlsValue);
LPVOID TlsGetValue (
  DWORD dwTlsIndex);

DWORD FlsAlloc (
  PFLS_CALLBACK_FUNCTION lpCallback);
BOOL FlsFree (
  DWORD dwFlsIndex);

BOOL FlsSetValue (
  DWORD dwFlsIndex,
  PVOID lpFlsValue);
PVOID FlsGetValue (
  DWORD dwFlsIndex);
```

To permit graceful handling of stack overflow exceptions, it is also possible to set the amount of space available on the stack during the stack overflow exception handling.

```
BOOL SetThreadStackGuarantee (
  PULONG StackSizeInBytes);
```

Example: Linux Clone API

Linux offers an alternative process and thread creation API using the `clone` call.

```
int clone (
  int (*fn) (void *),
  void *child_stack,
  int flags,
  void *arg,
  ...);
```

Example: Posix Dynamic Linker API

The dynamic linker can be accessed through a standardized interface as well. The `dlopen` and `dlclose` calls are used to load and drop a dynamic library into and from the current process. Loading and dropping a library also involves calling its constructor and destructor functions. The `dlsym` call locates a symbol by name. Special handles can be used to look up symbols in the default symbol lookup order or in an order that facilitates symbol wrapping.

```
void *dlopen (
  const char *filename,
  int flag);
int dlclose (
  void *handle);

void *dlsym (
  void *handle,
  const char *symbol);
```

Example: Java Thread API

Java wraps the operating system threads with a `Thread`, whose `run` method can be redefined to implement the thread function. A thread begins executing when its `start` method is called, the `stop` method can be used to terminate the thread.

```
class java.lang.Thread implements java.lang.Runnable {
  java.lang.Thread ();
  java.lang.Thread (java.lang.Runnable);

  void start ();
  void run ();
  void interrupt ();
  boolean isInterrupted ();

  void join () throws java.lang.InterruptedException;
  void setDaemon (boolean);
  boolean isDaemon ();

  static java.lang.Thread currentThread ();
  static void yield ();
  static void sleep (long) throws java.lang.InterruptedException;

  ...
}
```

Example: OpenMP Thread API

The traditional imperative interface to creating and terminating threads can be too cumbersome especially when trying to create applications that use both uniprocessor and multiprocessor platforms efficiently. The OpenMP standard proposes extensions to C that allow to create and terminate threads declaratively rather than imperatively.

The basic tool for creating threads is the `parallel` directive, which states that the encapsulated block is to be executed by multiple threads. The `for` directive similarly states that the encapsulated cycle is to be iterated by multiple threads. The `sections` directive finally states that the encapsulated blocks are to be executed by individual threads. More directives are available for declaring thread local data and other features.

```
#pragma omp parallel private (iThreads, iMyThread)
{
  iThreads = omp_get_num_threads ();
  iMyThread = omp_get_thread_num ();
  ...
}

#pragma omp parallel for
  for (i = 0 ; i < MAX ; i ++)
    a [i] = 0;

#pragma omp parallel sections
{
  #pragma omp section
    DoOneThing ();
  #pragma omp section
    DoAnotherThing ();
}
```

Rehearsal

At this point, you should understand how the abstract concept of a running process maps to the specific things happening inside a computer. You should be able to describe how the execution of a process relates to the execution of machine code instructions by the processor and what these instructions look like. You should be able to explain how the abstract concept of a process state maps to the content of memory and registers.

You should be able to outline how a process gets started and where the machine code instructions and the content of memory and registers comes from. You should understand how machine code instructions address memory and how the location of the program image in memory relates to the addressing of memory.

You should understand how an operating system gets to the point where it can start an arbitrary process from the point where the computer has just been turned on.

You should know what facilities enable a process to interact with the system libraries and the operating system.

Based on your knowledge of how processes are used, you should be able to design an intelligent API used to create and destroy processes and threads.

Questions

1. Explain what is a process.
2. Explain how the operating system or the first application process gets started when a computer is switched on.
3. Explain what it means to relocate a program and when and how a program is relocated.
4. Explain what information is needed to relocate a program and where it is kept.
5. Explain what it means to link a program and when and how a program is linked.
6. Explain what information is needed to link a program and where it is kept.
7. Explain what the interface between processes and the operating system looks like, both for the case when the operating system code resides in the user space and for the case when the operating system code resides in the kernel space.
8. Propose an interface through which a process can start another process and wait for termination of another process.

Achieving Parallelism

The operating system is responsible for running processes as necessary. In the simplest case, the operating system runs processes one at a time from beginning to completion. This is called *sequential* processing or *batch* processing. Running processes one at a time, however, means that each process usurps the whole computer for as long as it runs, which can be both inflexible and inefficient. The operating system therefore runs multiple processes in parallel. This is called *multiprocessing.*

Multiprocessing On Uniprocessors

Multiprocessing on machines with a single processor, or uniprocessors, is based on the ability of the operating system to suspend a process by setting its state aside and later resume the process by picking its state up from where it was set aside. Processes can be suspended and resumed frequently enough to achieve an illusion of multiple processes running in parallel.

In multiprocessing terminology, the state of a process is called *process context*, with the act of setting the process context aside and later picking it up denoted as *context switching*. Note that process context is not defined to be strictly equal to the process state, but instead vaguely incorporates those parts of the process state that are most relevant to context switching.

The individual parts of the process state and the related means of context switching are discussed next.

Processor State

The part of the process state that is associated with the processor consists of the processor registers accessible to the process. On most processors, this includes general purpose registers and flags, stack pointer as a register that stores the address of the top of the stack, program counter as a register that stores the address of the instruction to be executed.

The very first step of a context switch is passing control from the executing process to the operating system. As this changes the value of the program counter, the original value of the program counter must be saved simultaneously. Typically, the processor saves the original value of the program counter on the stack of the process whose context is being saved. The operating system typically proceeds by saving the original values of the remaining registers on the same stack. Finally, the operating system switches to the stack of the process whose context will be restored and restores the original values of the registers in an inverse procedure.

When separate notions of processes and threads are considered, the processor context is typically associated with the thread, rather than the process. Exceptions to this rule include special purpose registers whose content does not concern the execution of the thread but rather the execution of the process.

Context switching and similar operations that involve saving and restoring the processor context, especially interrupt and exception handling and system calls, happen very frequently. Processors therefore often include special support for these operations.

Example: Intel Processor Context Switching

The Intel 80x86 line of processors provides multiple mechanisms to support context switching. The simplest of those is the ability to switch to a different stack when switching to a different privilege level. This mechanism makes it possible to switch the processor context without using the stack of the executing process. Although not essential, this ability can be useful when the stack of the executing process must not be used, for example to avoid overflowing or mask debugging.

Another context switching support mechanism is the ability to save and restore the entire processor context of the executing process to and from the TSS (Task State Segment) structure as a part of a control transfer. One issue associated with this ability is efficiency. On Intel 80486, a control transfer using the CALL instruction with TSS takes 170 to 180 clock cycles. A control transfer using the CALL instruction without TSS takes only 20 clock cycles and the processor context can be switched more quickly using common instructions. Specifically, PUSHAD saves all general purpose registers in 11 clock cycles, PUSH saves each of the six segment registers in 3 clock cycles, PUSHF saves the flags in 4 clock cycles. Inversely, POPF restores the flags in 9 clock cycles, POP restores each of the six segment registers in 3 clock cycles, POPAD restores all general purpose registers in 9 clock cycles.

Additional context switching support mechanism takes care of saving and restoring the state of processor extensions such as FPU (Floating Point Unit), MMX (Multimedia Extensions), SIMD (Single Instruction Multiple Data). These extensions denote specialized parts of the processor that are only present in some processor models and only used by some executing processes, thus requiring special handling:

- The processor supports the FXSAVE and FXRSTOR instructions, which save and restore the state of all the extensions to and from memory. This support makes it possible to use the same context switch code regardless of which extensions are present.
- The processor keeps track of whether the extensions context has been switched after the processor context. If not, an exception is raised whenever an attempt to use the extensions is made, making it possible to only switch the extensions context when it is actually necessary.

Example: Linux Processor Context Switching

For examples of a real processor context switching code for many different processor architectures, check out the sources of Linux. Each supported architecture has an extra subdirectory in the `arch` directory, and an extra `asm` subdirectory in the `include` directory. The processor context switching code is usually stored in file `arch/*/kernel/entry.S`.

The following fragment contains the code for saving and restoring processor context on the Intel 80x86 line of processors from the Linux kernel, before the changes that merged the support for 32-bit and 64-bit processors and made the code more complicated. The `__SAVE_ALL` and `__RESTORE_ALL` macros save and restore the processor registers to and from stack. The fixup sections handle situations where segment registers contain invalid values that need to be zeroed out.

Example: Kalisto Processor Context Switching

Kalisto processor context switching code is stored in the `head.S` file. The `SAVE_REGISTERS` and `LOAD_REGISTERS` macros are used to save and load processor registers to and from memory, typically stack. The `switch_cpu_context` function uses these two macros to implement the context switch.

Memory State

In principle, the memory accessible to a process can be saved and restored to and from external storage, such as disk. For typical sizes of memory accessible to a process, however, saving and restoring would take a considerable amount of time, making it impossible to switch the context very often. This solution is therefore used only when context switching is rare, for example when triggered manually as in DOS or CTSS.

When frequent context switching is required, the memory accessible to a process is not saved and restored, but only made inaccessible to other processes. This requires the presence of memory protection and memory virtualisation mechanisms, such as paging. Switching of the memory context is then reduced to switching of the paging tables and flushing of the associated caches.

When separate notions of processes and threads are considered, the memory state is typically associated with the process, rather than the thread. The need for separate stacks is covered by keeping the pointer to the top of the stack associated with the thread rather than the process, often as a part of the processor state rather than the memory state. Exceptions to this rule include thread local storage, whose content does not concern the execution of a process but rather the execution of a thread.

Other State

The process state can contain other parts besides the processor state and the memory state. Typically, these parts of the process state are associated with the devices that the process accesses, and the manner in which they are saved and restored depends on the manner in which the devices are accessed.

Most often, a process accesses a device through the operating system rather than directly. The operating system provides an abstract interface that simplifies the device state visible to the process, and keeps track of this abstract device state for each process. It is not necessary to save and restore the abstract device state, since the operating system decides which state to associate with which process.

In some cases, a process might need to access a device directly. In such a situation, the operating system either has to save and restore the device state or guarantee an exclusive access to the device.

Multiprocessing On Multiprocessors

Parallelism on machines with multiple processors, or multiprocessors, ...

Vedle běhu více procesů pomocí opakovaného přepínání kontextu je možné navrhnout také systém s několika procesory a na každém spouštět jiný proces. Typické jsou SMP (Symmetric Multiprocessor) architektury, kde všechny procesory vidí stejnou paměť a periferie, nebo NUMA (Non Uniform Memory Access) architektury, kde všechny procesory vidí stejnou paměť a periferie, ale přístup na některé adresy je výrazně optimalizován pro některé procesory.

Hyperthreading to be done.

Example: Intel Multiprocessor Standard

Předpokládá SMP. Jeden procesor se definuje jako bootstrap processor (BSP), ostatní jako application processors (AP), spojené jsou přes 82489 APIC. Po resetu je funkční pouze BSP, všechny AP jsou ve stavu HALT, APIC dodává přerušení pouze PIC u BSP (a je povolené maskování A20, ach ta zpětná kompatibilita :-). BIOS vyplní speciální datovou strukturu popisující počet procesorů, počet sběrnic, zapojení přerušení a podobně a spustí whatever system you have. Systém pak připraví startup kód pro všechny AP a pomocí APIC je resetuje (startup adresa se zadá buď přes CMOS + BIOS hack, nebo přes APIC).

Funkci Intel MPS (výčet procesorů, doručování přerušení atd.) v současné době nahrazují části specifikace ACPI.

Jak už jednou procesory běží, problém je jenom s přerušením. Systém si řadiče přerušení každého procesoru nastaví jak potřebuje, pomocí APIC je možné doručit také Inter Processor Interrupts. IPI se hodí například na TLB nebo PTE invalidation.

Cooperative And Preemptive Switching

Přepnutí kontextu může být preemptivní, v takovém případě operační systém sebere počítač jednomu procesu a přidělí ho dalšímu, nebo kooperativní, v takovém případě se proces musí vzdát počítače dobrovolně.

- kooperativní - menší overhead, nedochází k přepnutí v nevhodných okamžicích
- preemptivní - robustnější systém, proces si nemůže uzurpovat počítač

Switching In Kernel Or In Process

Přepínání kontextu procesoru je mimochodem prakticky všechno, co musí dělat implementace threadů. Stačí každému threadu přidělit zásobník a CPU, což je kód, který může vykonávat i aplikace. Tedy pokud operační systém z nějakého důvodu nenabízí thready, aplikace si je může naprogramovat sama. Odtud pojmy *user managed threads* pro thready, které jsou implementovány v aplikaci a *kernel managed threads* pro thready, které jsou implementovány v kernelu. Jiná terminologie používá *user threads* pro thready, které jsou implementovány v aplikaci a kernel o nich neví, *lightweight processes* pro thready, které jsou implementovány v kernelu a aplikace je používá a *kernel threads* pro thready, které jsou implementovány v kernelu a aplikace o nich neví.

- Snaha o efektivitu směřuje k user threadům. Pokud je jejich implementace součást aplikace, šetří se na overheadu volání kernelu, šetří se paměť kernelu, a vůbec je to pohodlnější, člověk si může třeba i ledacos doimplementovat, přepínání může být kooperativní bez problémů s robustností operačního systému.
- Implementace user threadů musí řešit řadu komplikací. Protože přepínání user threadů má na starosti aplikace, pokud některá z nich zavolá kernel a zůstane tam, přepínání se zastaví. Pokud je přepínání preemptivní, může narazit na problémy s reentrantností knihoven či syscalls, typicky malloc. Thready se mohou vzájemně rušit skrz globální kontext, typicky errno, lseek, brk. Thready nemohou bez podpory kernelu použít více procesorů.

Process Lifecycle

Jakmile je k dispozici přepínání kontextu, princip plánování procesů je jasný. Operační systém si udržuje seznam procesů, které mají běžet, a střídavě jim přiděluje počítač. Trocha terminologie: seznam procesů, které mají běžet se jmenuje "ready queue", procesy v tomto seznamu jsou ve stavu "ready to run". Ted se dá kreslit stavový diagram, přepnutím kontextu se proces dostává ze stavu "ready to run" do stavu "running" a zpět, případně můžeme říci přesněji "running in kernel" a doplnit ještě stav "running in application". Mezi těmito stavy se přeskakuje syscalls a návraty z nich a interrupts a návraty z nich. Voláním sleep se proces ze stavu running in kernel dostane do stavu "asleep", z něj se voláním wakeup dostane do stavu "ready to run". Jména těchto volání jsou pouze příklady. Specificky pro UNIX existuje ještě stav "initial", ve kterém se proces vytváří během volání fork, a stav "zombie", ve kterém proces čeká po volání exit dokud parent neudělá wait.

How To Decide Who Runs

The responsibility for running processes and threads includes deciding when to run which process or thread, called scheduling. Scheduling accommodates various requirements such as responsiveness, throughput, efficiency:

- *Responsiveness* requires that a process reacts to asynchronous events within reasonable time. The asynchronous events may be, for example, a user input, where a prompt reaction is required to maintain interactivity, or a network request, where a prompt reaction is required to maintain quality of service.
- *Predictability* requires that the operating system can provide guarantees on the behavior of scheduling.
- *Turnaround* requires that scheduling minimizes the duration of each task.
- *Throughput* requires that scheduling maximizes the number of completed tasks.
- *Efficiency* requires that scheduling maximizes the resource utilization.
- *Fairness* requires that the operating system can provide guarantees on the equal treatment of tasks with the same scheduling parameters.

Many of the requirements can conflict with each other. Imagine a set of short tasks and a single long task that, for sake of simplicity, do not contend for other resources than the processor. A scheduler that aims for turnaround would execute the short tasks first, thus keeping the average duration of each task low. A scheduler that aims for throughput would execute the tasks one by one in an arbitary order, thus keeping the overhead of context switching low. A scheduler that aims for fairness would execute the tasks in parallel in a round robin order, thus keeping the share of processor time used by each task roughly balanced.

When resolving the conflicts between the individual scheduling requirements, it helps to consider classes of applications:

- An *interactive* application spends most of its time waiting for input. When the input does arrive, the application must react quickly. It is often stressed that a user of an interactive application will consider the application slow when a reaction to an input does not happen within about 150 ms. Fluctuations in reaction time are also perceived negatively.
- A *batch* application spends most of its time executing internal computations in order to deliver an external output. In order to benefit from various forms of caching that happen within the hardware and the operating system, a batch application must execute uninterrupted for some time.
- A *realtime* application must meet specific deadlines and therefore must execute long enough and often enough to be able to do so.

When faced with the conflict between the individual scheduling requirements, the operating system would therefore lean towards responsiveness for interactive applications, efficiency for batch applications, predicability for realtime applications.

Na stavovém diagramu je pak zjevně vidět, co je úlohou plánování. Je to rozhodnout, kdy a který proces přepnout ze stavu "ready to run" do stavu "running" a zpět. Pokud není žádný proces ve stavu "ready to run", spouští se umělý idle process alias zahaleč.

HLT

References

1. James Dabrowski, Ethan Munson: Is 100 Milliseconds Too Fast ?

Round Robin

Také cyklické plánování, FIFO, prostě spouštět procesy jeden po druhém. Otázkou je délka kvanta v poměru k režii přepnutí kontextu, příliš krátké kvantum snižuje efektivitu, příliš dlouhé kvantum zhoršuje responsiveness.

Static Priorities

Processes are assigned static priorities, the process with the highest priority is scheduled. Either constant quantum is used or shorter quantum is coupled with higher priority and vice versa.

Confusion can arise when comparing priorities, for numerically lower priority values are often used to meant semantically higher priorities. In this text, the semantic meaning of priorities is used.

Dynamic Priorities

Processes are assigned initial priorities, actual priorities are calculated from the initial priorities and the times recently spent executing, the process with the highest actual priority is scheduled.

Shortest Job First

Pokus o dobrou podporu dávek, spouští se ta co trvá nejkratší dobu, tím se dosáhne průměrně nejkratšího času do ukončení dávky, protože pokud čeká více dávek, jejich časy ukončení se postupně přičítají, a tak je dobře začít od nejkratších časů. Toto se dá zčásti použít i na interaktivní procesy, v situaci kdy více uživatelů na více terminálech čeká na odezvu, spustí se ten proces, kterému bude doručení odezvy trvat nejméně dlouho, tím se dosáhne minimální average response time. Nepříjemné je, že vyžaduje vizionářství, řeší se třeba statisticky.

Fair Share

Také guaranteed share scheduling, procesům se zaručuje jejich procento ze strojového času, buď každému zvlášť nebo po skupinách nebo třeba po uživatelích

Earliest Deadline First

Procesy mají deadlines do kdy musí něco udělat, plánuje se vždy proces s nejbližší deadline. Deadlines se zpravidla rozdělují do *hard realtime deadlines* , ty jsou krátké a nesmí se prošvihnout, *soft realtime deadlines* , ty jsou krátké a občas se prošvihnout můžou dokud bude možné zaručit nějaký statistický limit prošvihnutí, *timesharing deadlines* , ty v podstatě nemají pevný časový limit ale měly by se do někdy stát, *batch deadlines* , ty mají limity v hodinách a obvykle s nimi nebývají problémy.

Example: Archetypal UNIX Scheduler

Nejprve starší scheduler z UNIXu, dynamické priority, nepreemptivní kernel. Priorita je číslo 0-127, nižší číslo vyšší priorita, default 50. Každý proces má current priority 0-127, pro kernel rezervováno 0-49, user priority 0-127, processor usage counter 0-127 a nice factor 0-39, ovlivněný příkazem nice. Current priority v aplikaci je rovna user priority, v kernelu se nastaví podle toho, na co proces čeká, například po čekání na disk se nastaví na 20, čímž se zaručí, že proces po ukončení čekání rychle vypadne z kernelu. Processor usage count se inkrementuje při každém kvantu spotřebovaném procesem, zmenšuje se podle magické formule jednou za čas, třeba na polovinu každou vteřinu, nebo podle load average aby při zatíženém procesoru processor usage count neklesal moc rychle. Load je průměrný počet spustitelných procesů v systému za nějaký čas. User priority se pak vypočítá jako default priority + processor usage counter / 4 + nice factor * 2.

Nevýhody. První, does not scale well. Pokud běží hodně procesů, roste režie na přepočítávání priorit. Další, neumí nic garantovat, zejména ne response time nebo processor share. A závěrem, aplikace mohou priority ovlivňovat pouze přes nice factor, který nenabízí zrovna moc advanced control.

Example: Solaris Scheduler

Podobný je System V Release 4 scheduler. Ten má jako jádro scheduleru fronty ready to run" procesů podle priority 0-160 a rutinu pro plánování, která klasicky vybere proces nejvyšší priority, round robin na stejné prioritě. Každý proces patří do nějaké třídy priorit, která má na starosti všecha rozhodnutí ohledně přidělění priority a délky kvanta. By default jsou k dispozici tři třídy priorit, timesharing, system a realtime:

Timesharing. Používá priority 0-59, procesu se zvyšuje priorita pokaždé když na něco čeká nebo když dlouho trvá než spotřebuje své kvantum, priorita se snižuje pokaždé když proces spotřebuje své kvantum. Přesný způsob změny priority se určuje podle tabulky, jako příklad proces priority 40 spadne po spotřebování kvanta na 30, po ukončení čekání nebo pokud proces nespotřebuje kvantum do 5 vteřin, priorita naopak vyleze na 50 (čekající proces dostane priority 59, změna v normální prioritě se objeví po návratu do user mode). K výsledku se ještě přidává nice value, tím je priorita v user mode určena jednak systémem počítanou prioritou a dvak nice value. Podle systémem počítané priority se udržuje také délka kvanta, nižší priority mají delší kvantum (protože se čeká, že tak často nepoběží, a tak když už se dostanou na řadu, tak ať něco udělají).

System. Používá priority 60-99 pro procesy běžící v kernelu. Tato třída je interní systémová, nečeká se že by do ní hrabali useři, běží v ní třeba page daemon. Proces, který při volání kernelu obdrží kritické resources, dostane dočasně také priority 60-99.

Realtime. Používá priority 100-159, priorita procesu a přidělované kvantum se nastavuje syscallem priocntl a od okamžiku nastavení se nemění. Bordel je v tom, že realtime priority může být větší než system priority, tedy občas by bylo potřeba přerušit kernel. To se ale normálně nedělá, protože preemptivní kernel by byl složitý, procesy se přepínají nejčastěji při opouštění kernel mode, kdy se zkontroluje flag "runrun" indikující nutnost přepnout kontext. Jako řešení se k flagu "runrun" přidá ještě flag "kprunrun" indikující nutnost přenout kontext uvnitř kernelu, a definují se body, kdy je i v kernelu bezpečné přepnout kontext. V těchto bodech se pak testuje "kprunrun". Výsledkem je zkrácení prodlev před rozběhnutím "ready to run" realtime procesů.

Také se počítá s tím, že člověk si bude moci přidávat vlastní třídy priorit. Každá třída implementuje 7 obslužných rutin, jsou to CL_TICK volaná z clock interrupt handleru, CL_FORK a CL_FORKRET pro inicializaci nového procesu, CL_ENTERCLASS a CL_EXITCLASS pro obsluhu situací, kdy proces vstoupí do třídy nebo ji opustí, CL_SLEEP volaná když proces udělá sleep, CL_WAKEUP volaná když proces opustí sleep. Přidání vlastní třídy znamená napsat těchto 7 rutin a přeložit kernel.

Výhoda popsaného scheduleru spočívá jednak v tom, že nikdy nepřepočítává žádné velké seznamy priorit, dvak v tom, že umí podporovat realtime procesy. Pravděpodobně nejznámějším systémem s tímto schedulerem je Solaris, ten má několik drobných změn. Solaris 7 nabízí uživatelům timesharing, interactive a realtime classes, podobně jak bylo popsáno výše, až na to že kernel je mostly preemptive, což zlepšuje responsiveness realtime procesů.

Solaris 7 má příkaz "dispadmin", kterým se dají vypsat tabulky se scheduling parametry. Parametry pro timesharing procesy jsou ve formě tabulky s délkou kvanta (200ms pro prioritu 0, 20ms pro prioritu 59), priority po vypršení kvanta (vždy o 10 menší), priority po sleepu (50 + priority/10 pro většinu priorit), maximální délka starvation (1 vteřina pro všechny priority) a priorita při překročení této délky (50 + priority/10 pro většinu priorit). Parametry pro interactive procesy jsou stejné jako pro timesharing procesy. Parametry pro realtime procesy jsou ve formě tabulky s délkou kvanta (1 sekunda pro prioritu 0, 100ms pro prioritu 59). Admin může tyto tabulky měnit.

Třídy *interactive* a *timesharing* sdílejí tutéž tabulku parametrů, což by naznačovalo, že sdílejí i tentýž plánovací algoritmus, ale nemusí to být pravda. Letmá měření žádný rozdíl neukázala, takže je možné, že TS a IA třídy existují kvůli větší flexibilitě (každé zvlášť se dá nastavit rozsah user priorit) a by default jsou opravdu stejné.

Jako zajímavost, Solaris 7 nabízí ještě volání, které umožnuje lightweight procesu požádat kernel, aby mu dočasně neodebral procesor. To se hodí třeba při implementaci spinlocků v user mode. Detaily viz manpage schedctl_init.

Example: Linux 2.4.X Series Scheduler

In the 2.4.X series of Linux kernels, the scheduler decided what process to run based on a proprietary metric called *goodness value*, simply by picking the process with the highest goodness value. The scheduler distinguished *time sharing* and *realtime* scheduling policies and used different goodness value calculation depending on the policy used by each process.

Under the realtime scheduling policy, each process has a priority value ranging from 1 (low) to 99 (high). The goodness value was calculated as 1000 plus the process priority. This calculation made the scheduler select the realtime process with the highest priority ahead of any other realtime or time sharing process. Depending on whether the scheduling policy was SCHED_RR or SCHED_FIFO, the scheduler either assigned the process a quantum or let the process run with no quantum limit. Naturally, this only made any difference when there were more realtime processes with the same priority.

Under the time sharing scheduling policy, each process has a nice value ranging from -20 (high) to 19 (low). The goodness value was equal to the quantum left to the process in the current scheduling cycle, with the entire range of nice values roughly corresponding to quanta from 210 ms for nice -20 to 10 ms for nice 19. When all processes consumed their quanta, a new scheduling cycle was started and new quanta were assigned to the processes.

Example: Windows Scheduler

Windows uses a priority based scheduler. The priority is an integer from 0 to 31, higher numbers denoting higher priorities. The range of 1-15 is intended for standard applications, 16-31 for realtime applications, memory management worker threads use priorities 28 and 29. The priorities are not assigned to threads directly. Instead, the integer priority is calculated from the combination of one of seven relative thread priorities and one of four process priority classes:

	Idle	**Normal**	**High**	**Realtime**
Idle	1	1	1	16
Lowest	2	6	11	22
Below Normal	3	7	12	23
Normal	4	8	13	24
Above Normal	5	9	14	25
Highest	6	10	15	26
Time Critical	15	15	15	31

The priorities are further adjusted. Threads of the Idle, Normal and High priority class processes receive a priority boost on end of waiting inside kernel and certain other events, and a priority drop after consuming the entire allocated quantum. Threads of the Normal priority class processes receive a priority boost after their window is focused. Similar adjustment is used for process affinity and priority inheritance.

The scheduler always runs the thread with the highest priority. Multiple threads with the same priority are run in a round robin fashion. The time quanta are fixed at around 120 ms in server class Windows versions, and variable from around 20 ms for background processes to around 60 ms for foreground processes in desktop class Windows versions.

Administrative privileges are required to set priorities from the realtime range. To avoid the need for administrative privileges for running multimedia applications, which could benefit from realtime priorities, Windows Vista introduces the Multimedia Class Scheduler Service. Processes can register their threads with the service under classes such as Audio or Games, and the service takes care of boosting the priority of the registered threads based on predefined registry settings.

Example: Nemesis Deadline Scheduler

Operační systém Nemesis, Cambridge University 1994-1998, cílem je podpora Quality of Service pro distributed multimedia applications. V Nemesis se plánují domény, kernel přidělí CPU doméně pomocí activation, což není nic jiného než upcall rutiny specifikované v příslušném domain control bloku. Výjimku tvoří situace, kdy byla doméně odebrána CPU dříve, než ta indikovala připravenost na další activation, v takovém případě se pokračuje v místě odebrání CPU. Krom stavu ready to run může doména ještě čekat na event, což není nic jiného než asynchronně doručovaná zpráva obsahující jeden integer.

Každá doména si řekne o processor share, který chce, ve formě čísel slice a period, obě v nějakých timer ticích. Systém pak zaručuje, že v každé periodě dostane aplikace nejméně slice tiků, s podmínkou, že suma všech slice / period v systému je menší než 1 (jinak by nebylo dostatek CPU času na uspokojení všech domén).

Interní je scheduler založen na earliest deadline first algoritmu. Udržuje se fronta domén, které ještě nebyly v dané periodě uspokojeny, seřazná podle času konce této periody, a fronta domén, které již uspokojeny byly, seřazená podle času začátku nové periody. Scheduler vždy spustí první doménu ve frontě neuspokojených domén, pokud je tato fronta prázdná, pak náhodnou doménu ve frontě uspokojených domén. Následující scheduler action se naplánuje na čas nejbližší deadline nebo do konce slice, whichever comes sooner.

Mimochodem, původní popis algoritmu nezmiňoval scheduler action při vyčerpání slice, což se pak projevovalo jednak v anomáliích při rozjezdu algoritmu, dvak v neřízeném rozdělování přebytečného času procesoru. Divné.

* *Příklad, tři procesy A B C, A chce share 1 per 2, B chce share 1 per 5, C chce share 2 per 10. Tabulka, řádky čas, sloupce zbývající slices a period deadlines pro A B C.*

Jako detaily, předpokládá se stabilní běh systému a nulová režie scheduleru. Další drobnost, domain si může říci, zda chce nebo nechce dostávat přebytečný čas procesoru. Mezi ty domény, které ho chtějí dostávat, se přebytečný čas procesoru rozděluje náhodně, s kvantem nějakých 100 us, nic lepšího zatím nevymysleli a prý to stačí.

Jeden detail, co dělat s doménami, které čekaly na event? Prostě se nacpou zpátky do fronty neuspokojených domén jako kdyby se nově zařazovaly, přinejhorším tím budou spotřebovávat přebytečný čas procesoru. Pro domény, kterým stačí jen malé procento času procesoru, ale potřebují reagovat rychle, se dá zadat ještě latency hint, ten se použije pro výpočet deadline místo periody v případě, že doména čekala déle než svou periodu. Použití pro interrupt handling.

Interrupt handling je neobvyklý, zato však odstraňuje jeden ze základních problémů dosud zmíněných schedulerů, totiž že část aktivit spojená s devices je v podstatě plánovaná signály od hardware a nikoliv operačním systémem. Když přijde interrupt, jeho handler v kernelu jej pouze zamaskuje a pošle event doméně zodpovědné za jeho obsluhu. Předpokládá se, že tato doména čeká na event, scheduler jí tedy v souladu s jejími parametry naplánuje, doména obslouží device a znovu povolí interrupt. Pokud pak systém nestíhá, není ucpaný interrupty, ale prostě vyřídí to co stíhá a zbytek interruptů ignoruje. Drivery mohou instalovat handlery v kernelu.

Téma nestíhání serverů a driverů je ještě o něco hlubší. Nad kernel schedulerem totiž má běžet ještě Quality of Service monitor aplikace, který umí detekovat situace, kdy server nebo driver nestíhá odpovídat na dotazy. Processor share serveru se udržuje dostatečně vysoký na to, aby stíhal odpovídat, to je vždy možné protože s rostoucím share se blíží extrém, kdy server sežere veškerý čas procesoru a nepoběží driver, který mu dodává data jež ho zatěžují. Processor share driveru se udržuje dostatečně vysoký na zpracování příchozích dat, nejvýše však takový, aby se kvůli němu nemusel snižovat processor share serveru. A tím je vystaráno, počítač dělá co stihne, nadbytečný traffic prostě ignoruje a nezahltí se.

Ještě drobný detail o tom, proč se domény aktivují tak divně od stejného místa. Počítá se s tím, že každá doména bude mít v sobě něco jako user threads, aktivace domény pak spustí user scheduler, který si skočí na který thread uzná za vhodné. Systém nabízí doménám možnost specifikovat, kam se má při preempci uložit kontext procesoru, předpokládá se, že každá doména bude zvlášť ukládat kontexty jednotlivých threadů.

Scheduling On Multiprocessors

Plánování začne být ještě o něco zajímavější v multiprocesorových systémech. Některé problémy už byly naznačené, totiž multiprocesorové plánování by nemělo být pouhým rozšířením singleprocesorového, které má jednu ready frontu a z ní posílá procesy na všechny procesory. Proč vlastně ?

Důvod spočívá v tom, jak vypadá multiprocesorový hardware. I u velmi těsně vázaného systému mají procesory lokální informace nasbírané za běhu procesu, jako třeba cache překladu adres, memory cache, branch prediction a podobně. Z toho je snadno vidět, že výkonu systému prospěje, pokud scheduler bude plánovat tytéž procesy, případně thready téhož procesu, na stále stejné procesory. Tomu se někdy říká processor affinity.

Již zmíněnými příklady byly Solaris, Linux, a Windows NT, které na multiprocesorovém hardware zohledňují processor affinity a mírně se snaží plánovat stejné procesy na stejné procesory. Další by byl třeba Mach.

Samozřejmě zůstávají další problémy, jeden z nich je například sdílení ready fronty. Čím více procesů sdílí libovolný prostředek, tím více na něm budou čekat, a ready frontu sdílí každý procesor a kernel do ní hrabe každou chvíli. Drobným vylepšením je například definování local a global ready front s rozlišením, kdy se bude sahat do které. Toto rozlišení může být různé, například realtime procesy v globální frontě a ostatní v lokálních, nebo dynamické přesouvání mezi globální a lokální frontou podle zatížení procesoru.

Ještě zajímavější je scheduling na loosely coupled hardware, kde se například nesdílí paměť. Tam se už musí zohledňovat i cena migrace procesu, cena vzdáleného přístupu k prostředkům a podobně, ale to teď necháme.

Example: Mach Scheduler

Mach plánuje na úrovni threadů. Základní princip plánování je stále stejný, priority a zohlednění CPU usage, zaměřme se na multiprocessor support.

Za prvé, Mach nemá cross processor scheduler events. Co to je napovídá název — pokud se na některém procesoru objeví událost, která povolí běh threadu s prioritou vyšší než je priorita nějakého jiného threadu na jiném procesoru, tento druhý thread se nepřeruší, ale poklidně doběhne své kvantum, než se scheduler dostane ke slovu.

Za druhé, Mach zavádí processor sets, množiny procesorů určených k vykonávání threadů. Každý thread má přidělen jeden processor set, na kterém je plánován, processor sets definuje a přiděluje admin. Tak je možné vyhradit například rychlé procesory na realtime úlohy, či procesory se speciálním hardware na úlohy, které jej využijí. Procesory sice mohou patřit pouze do jednoho setu, ale za běhu systému mohou mezi sety cestovat.

Handoff scheduling.

Example: Linux Early 2.6.X Series Scheduler

The early 2.6 series of kernels uses a scheduler that provides constant time scheduling complexity with support for process preemption and multiple processors.

The scheduler keeps a separate pair of an active and an expired queue for each processor and priority, the active queue being for processes whose quantum has not yet expired, the expired queue being for processes whose quantum has already expired. For priorities from 1 to 140, this makes 280 queues per processor. The scheduler finds first non empty queue pair, switches the active and expired queues when the active queue is empty, and schedules the first process from the active queue. The length of the quantum is calculated linearly from the priority, with higher quanta for lower priority.

An interactivity metric is kept for all processes as a ratio of the time spent calculating to the time spent waiting for input or output. Processes with the priority range between 100 and 140, which is reserved for user processes, get their priority adjusted so that processes that calculate a lot are penalized and processes that wait for input or output a lot are rewarded. Processes that wait for input or output a lot are never moved from the active to the expired queue.

An extra kernel thread redistributes processes between processors.

References

1. Rohit Agarwal: Process Scheduler For Kernel 2.6.x

Example: Linux Late 2.6.X Series Scheduler

The late 2.6 series of kernels distinguish Completely Fair Scheduler (CFS) and Real Time (RT) classes, handled by separate scheduler modules with separate per processor run queue structures. The scheduler calls the `put_prev_task` function of the appropriate module to put a previously scheduled process among other runnable processes, and the `pick_next_task` function of the highest priority module to pick the next scheduled process from other runnable processes.

The per processor structure of the RT class scheduler contains an array of queues, one for each process priority. The `pick_next_task` function picks the process with the highest priority. The quantum is set to 100 ms for the SCHED_RR scheduling policy and to infinity for the SCHED_FIFO scheduling policy.

The per processor structure of the CFS class scheduler contains a tree of processes, indexed by a virtual time consumed by each process. The `pick_next_task` function achieves fairness by picking the process with the least consumed time, maintaining overall balance in the consumed times. Rather than setting a fixed quantum, the scheduler is configured to rotate all ready processes within a given time period and calculates the quantum from this time period and the number of ready processes.

The virtual time consumed by each process is weighted by the nice value. A difference of one nice point translates into a difference of 10% processor time. After the quantum expires, the consumed time is adjusted accordingly. Processes that become ready are assigned the minimum of the virtual consumed times, unless they become ready after a short sleep, in which case their virtual consumed time is preserved.

An extra kernel thread redistributes processes between processors.

Example: Advanced Linux Scheduling Features

Multiple advanced features control the scheduler behavior on groups of processes. The groups can be defined using the control group mechanism, which is exported through a virtual filesystem. There can be multiple control group hierarchies exported through multiple virtual filesystems. Each hierarchy is associated with a particular controller, the scheduler being one. Individual groups are created as directories in the virtual filesystem. Group membership and controller parameters are set through files.

```
> tree -d /sys/fs/cgroup

/sys/fs/cgroup
|-- blkio
|-- cpu
|   `-- system
|       |-- acpid.service
|       |-- chronyd.service
|       |-- crond.service
|       |-- dbus.service
|       |-- httpd.service
|       |-- mdmonitor.service
...
|       |-- sshd.service
|       |-- udev.service
|       `-- upower.service
|-- devices
|-- memory
...
```

Each scheduler control group can define its processor share in the `cpu.shares` tunable. The shares are dimensionless numbers that express processor usage relative to other control groups. Processor usage can also be regulated in absolute rather than relative terms. The `cpu.cfs_period_us` and `cpu.cfs_quota_us` tunables set the maximum processor time used (sum across processors) per period for the CFS scheduler, the `cpu.rt_period_us` and `cpu.rt_runtime_us` tunables provide similar feature for the RT scheduler.

The processors in a multiprocessor system can be grouped using the *cpuset* control groups. Each cpuset control group contains a subset of processors of its parent, the top level group has all processors. Assignment of processes to processors can be restricted by placing them in a particular cpuset. Load balancing within a cpuset can be disabled entirely or adjusted so that immediate load balancing is limited. Immediate load balancing takes place when there are no ready processes available to a particular processor.

What Is The Interface

As illustrated by the individual examples, the interface to the scheduler is mostly determined by the scheduler itself.

Example: Linux Scheduler API

Linux offers the old style scheduler interface where priority is a single integer, ranging roughly from -20 to +20. This used to be the nice value in the old style scheduler algorithm. Meaning in the new style scheduler algorithms varies. The interface allows setting the priority for a single process, all processes in a group, and all processes owned by a user. Note the interface design error where return value can be either legal priority or error code.

```
int getpriority (int which, int who);
int setpriority (int which, int who, int prio);
```

The old style scheduler interface is accompanied by a new style scheduler interface that allows changing scheduling type. The supported types are SCHED_OTHER, SCHED_BATCH and SCHED_IDLE for time sharing processes and SCHED_FIFO and SCHED_RR for processes with static priorities. For time sharing processes, the nice value set through the old style interface still applies. For processes with static priorities, the priority value can be set through the new style interface.

```
int sched_setscheduler (
  pid_t pid,
  int policy,
  const struct sched_param *param);
int sched_getscheduler (pid_t pid);

int sched_setparam (pid_t pid, const struct sched_param *param);
int sched_getparam (pid_t pid, struct sched_param *param);

struct sched_param
{
  ...
  int sched_priority;
  ...
};

int sched_yield (void);
```

A process can also be constrained to run only on certain processors.

```
int sched_setaffinity (
  pid_t pid,
  size_t cpusetsize,
  cpu_set_t *mask);
int sched_getaffinity (
  pid_t pid,
  size_t cpusetsize,
  cpu_set_t *mask);
```

Example: Windows Scheduler API

```
BOOL SetPriorityClass (
  HANDLE hProcess,
  DWORD dwPriorityClass);
DWORD GetPriorityClass (
  HANDLE hProcess);

BOOL SetThreadPriority (
  HANDLE hThread,
  int nPriority);
int GetThreadPriority (
  HANDLE hThread);

BOOL SetProcessPriorityBoost (
  HANDLE hProcess,
  BOOL DisablePriorityBoost);
BOOL SetThreadPriorityBoost (
  HANDLE hThread,
  BOOL DisablePriorityBoost);

BOOL SetProcessAffinityMask (
  HANDLE hProcess,
  DWORD_PTR dwProcessAffinityMask);
DWORD_PTR SetThreadAffinityMask (
  HANDLE hThread,
  DWORD_PTR dwThreadAffinityMask);
```

Windows also provides an interface that implements the thread pool scheduling pattern, where a pool of threads with predefined minimum and maximum size is used to handle incoming work requests.

```
PTP_POOL CreateThreadpool (
  PVOID reserved);
VOID CloseThreadpool (
  PTP_POOL ptpp);

BOOL SetThreadpoolThreadMinimum (
  PTP_POOL ptpp,
  DWORD cthrdMic);
VOID SetThreadpoolThreadMaximum (
  PTP_POOL ptpp,
  DWORD cthrdMost);

VOID SubmitThreadpoolWork (
  PTP_WORK pwk);
```

Figure 2-12. Windows Thread Pool Calls

It is also possible to query various information on process timing.

```
BOOL GetProcessTimes (
  HANDLE hProcess,
  LPFILETIME lpCreationTime,
  LPFILETIME lpExitTime,
  LPFILETIME lpKernelTime,
  LPFILETIME lpUserTime);
```

Figure 2-13. Windows Process Timing Call

Rehearsal

At this point, you should understand how multiple processes can run in parallel on a computer with multiple processors, and how an illusion of multiple processes running in parallel can be provided even when the number of processes is higher than the number of processors. You should be able to recognize important parts of process context and explain how efficient context switching can be done with each part of the process context.

You should see why it can be useful to split an activity of a process into multiple threads. You should understand why and which parts of the entire context remain shared parts of the process context and which parts become private parts of the thread context.

You should be able to design meaningful rules telling when to switch a context and what context to switch to, related to both the architecture of the operating system and the requirements of the applications. You should be able to explain the working of common scheduling algorithms in the light of these rules.

Questions

1. Explain how multiple processes can run concurrently on a single processor hardware.
2. Explain what is a thread and what is the relationship between threads and processes.
3. Explain what happens when the thread context is switched.
4. Explain what happens when the process context is switched.
5. Using a step by step description of a context switch, show how an implementation of threads in the user space and an implementation of threads in the kernel space differ in the way the context is switched.

   Na popisu přepnutí kontextu krok po kroku ukažte, jak se implementace vláken v uživatelském prostoru a implementace vláken v prostoru jádra liší ve způsobu přepínání kontextu.
6. Explain how the requirements of interactive and batch processes on the process scheduling can contradict each other.
7. List the typical requirements of an interactive process on the process scheduling.
8. List the typical requirements of a batch process on the process scheduling.
9. List the typical requirements of a realtime process on the process scheduling.
10. Explain the difference between soft and hard realtime scheduling requirements.
11. Define typical phases of a process lifecycle and draw a transition diagram explaining when and how a process passes from one phase to another.
12. Explain cooperative context switching and its advantages.
13. Explain preemptive context switching and its advantages.
14. Explain the round robin scheduling algorithm by outlining the code of a function `GetProcessToRun` that will return a process to be scheduled and a time after which another process should be scheduled.
15. Explain the simple priority scheduling algorithm with static priorities by outlining the code of a function `GetProcessToRun` that will return a process to be scheduled and a time after which another process should be scheduled.
16. Explain the simple priority scheduling algorithm with dynamically adjusted priorities by outlining the code of a function `GetProcessToRun` that will return a process to be scheduled and a time after which another process should be scheduled.
17. Explain the earliest deadline first scheduling algorithm by outlining the code of a function `GetProcessToRun` that will return a process to be scheduled and a time after which another process should be scheduled.
18. Explain the function of the Solaris process scheduler by describing how the algorithm decides what process to run and for how long.
19. Explain the function of the Linux process scheduler by describing how the algorithm decides what process to run and for how long.
20. Explain the function of the Windows process scheduler by describing how the algorithm decides what process to run and for how long.
21. Define processor affinity and explain how a scheduler observes it.

22. Propose an interface through which a thread can start another thread and wait for termination of another thread, including passing the initial arguments and the termination result of the thread.

Exercises

1. Design a process scheduler that would support coexistence of batch, realtime and interactive processes. Describe how the processess communicate their scheduling requirements to the scheduler and what data structures the scheduler keeps. Describe the algorithm that uses these requirements and data structures to decide what process to run and for how long and analyze the time complexity of the algorithm.

Process Communication

Means Of Communication

Běžně používané prostředky pro komunikaci mezi procesy jsou:

- sdílení paměti a výměna informací skrz tuto paměť
- zasílání zpráv mezi procesy v různých formách

Shared Memory

To be done.

Example: System V Shared Memory

To be done.

```
int shmget (key_t key, size_t size, int shmflg);
void *shmat (int shmid, const void *shmaddr, int shmflg);
int shmdt (const void *shmaddr);
```

```
> ipcs -m
key        shmid      owner      perms      bytes      nattch     status
0x00000000 12345      root       600        123456     2          dest
0x00000000 123456     root       600        234567     2          dest
0x00000000 1234567    nobody     777        345678     2          dest
```

Example: POSIX Shared Memory

To be done.

Example: Windows Shared Memory

To be done.

```
HANDLE CreateFileMapping (
  HANDLE hFile,
  LPSECURITY_ATTRIBUTES lpFileMappingAttributes,
  DWORD flProtect,
  DWORD dwMaximumSizeHigh,
  DWORD dwMaximumSizeLow,
  LPCTSTR lpName);
```

Flag PAGE_READONLY gives read only access to the committed region. Flag PAGE_READWRITE gives read write access to the committed region of pages. Flag PAGE_WRITECOPY gives copy on write access to the committed region.

Flag SEC_COMMIT allocates physical storage in memory or in the paging file on disk for all pages of a section. Flag SEC_IMAGE says file is executable, mapping and protection are taken from the image. Flag SEC_NOCACHE disables caching, used for shared structures in some architectures. Flag SEC_RESERVE reserves all pages of a section without allocating physical storage, reserved range of pages cannot be used by any other allocation operations until it is released.

If hFile is 0xFFFFFFFF, the calling process must also specify a mapping object size in dwMaximumSize. The function creates a file mapping object backed by the operating system paging file rather than by a named file.

```
LPVOID MapViewOfFile (
  HANDLE hFileMappingObject, DWORD dwDesiredAccess,
  DWORD dwFileOffsetHigh, DWORD dwFileOffsetLow,
  DWORD dwNumberOfBytesToMap);

LPVOID MapViewOfFileEx (
  HANDLE hFileMappingObject, DWORD dwDesiredAccess,
  DWORD dwFileOffsetHigh, DWORD dwFileOffsetLow,
  DWORD dwNumberOfBytesToMap, LPVOID lpBaseAddress);
```

Flags FILE_MAP_WRITE, FILE_MAP_READ, FILE_MAP_ALL_ACCESS, FILE_MAP_COPY. Address is suggested, if the address is not free the call fails.

```
BOOL UnmapViewOfFile (LPCVOID lpBaseAddress);
```

The address must be from a previous MapViewOfFile(Ex) call.

Example: Windows Clipboard

To be done.

Message Passing

Message passing is a mechanism that can send a *message* from one process to another. The advantage of message passing is that it can be used between processes on a single system as well as between processes on multiple systems connected by a network without having to change the interface between the processes and message passing.

Message passing is *synchronous* when the procedure that sends a message can not return until the message is received. Message passing is *asynchronous* when the procedure that sends a message can return before the message is received.

The procedures that send or receive a message are *blocking* when they can wait before returning, and *non blocking* when they do not wait before returning. When a non blocking procedure needs to wait, it can replace blocking by *polling* or *callbacks*.

Message passing can use *symmetrical*, *asymmetrical* and *indirect* addressing. The symmetrical addressing requires both the sender and the receiver to specify the address of the other party. The asymmetrical addressing requires the sender to specify the address of the receiver. The indirect addressing requires both the sender and the receiver to specify an address of the same message queue.

The message sent from the sender to the receiver can be anything from a single integer number through an unformatted stream of bytes to a formatted structure of records.

Example: Posix Signals

Signals are messages that can be delivered to processes or threads. A signal is identified by a number, with numbers from 1 to 31 allocated to standard signals with predefined meaning and numbers from SIGRTMIN to SIGRTMAX allocated to real time signals.

Name	Number	Meaning
SIGHUP	1	Controlling terminal closed
SIGINT	2	Request for interrupt sent from keyboard
SIGQUIT	3	Request for quit sent from keyboard
SIGILL	4	Illegal instruction
SIGTRAP	5	Breakpoint instruction
SIGABRT	6	Request for abort
SIGBUS	7	Illegal bus cycle
SIGFPE	8	Floating point exception
SIGKILL	9	Request for kill
SIGUSR1	10	User defined signal 1
SIGSEGV	11	Illegal memory access
SIGUSR2	12	User defined signal 2
SIGPIPE	13	Broken pipe
SIGALRM	14	Timer alarm
SIGTERM	15	Request for termination
SIGTERM	16	Illegal stack access
SIGCHLD	17	Child process status changed
SIGCONT	18	Request to continue when stopped
SIGSTOP	19	Request to stop
SIGTSTP	20	Request for stop sent from keyboard
SIGTTIN	21	Input from terminal when on background
SIGTTOU	22	Output to terminal when on background

Figure 2-14. Standard Signals

Signals are processed by signal handlers. A signal handler is a procedure that is called by the operating system when a signal occurs. Default handlers are provided by the operating system. New handlers can be registered for some signals.

```
typedef void (*sighandler_t) (int);

sighandler_t signal (int signum, sighandler_t handler);
```

- SIG_DFL - use default signal handler
- SIG_IGN - ignore the signal

```
struct sigaction
{
  void (*sa_handler) (int);
  void (*sa_sigaction) (int, siginfo_t *, void *);
  sigset_t sa_mask;
  int sa_flags;
}

struct siginfo_t
{
  int      si_signo;    // Signal number
  int      si_errno;    // Value of errno
  int      si_code;     // Additional signal code
  pid_t    si_pid;      // Sending process PID
  uid_t    si_uid;      // Sending process UID
  int      si_status;   // Exit value
  clock_t  si_utime;    // User time consumed
  clock_t  si_stime;    // System time consumed
  sigval_t si_value;    // Signal value
  int      si_int;      // Integer value sent with signal
  void *   si_ptr;      // Pointer value sent with signal
  void *   si_addr;     // Associated memory address
  int      si_fd;       // Associated file descriptor
}

int sigaction (int signum, const struct sigaction *act, struct sigaction *oldact);
```

- sa_handler - signal handler with limited arguments
- sa_sigaction - signal handler with complete arguments
- sa_mask - what other signals to mask while in signal handler

- SA_RESETHAND - restore default signal handler after one signal
- SA_NODEFER - allow recursive invocation of this signal handler
- SA_ONSTACK - use alternate stack for this signal handler

Figure 2-15. Signal Handler Registration System Call

Due to the ability of signals to interrupt processes at arbitrary times, the actions that can be taken inside a signal handler are severely limited. Access to shared variables and system calls are not safe in general. This can be somewhat alleviated by masking signals.

```
int sigprocmask (int how, const sigset_t *set, sigset_t *oset);
int pthread_sigmask (int how, const sigset_t *set, sigset_t *oset);
```

- SIG_BLOCK - add blocking to signals that are not yet blocked
- SIG_UNBLOCK - remove blocking from signals that are blocked
- SIG_SETMASK - replace existing mask

Figure 2-16. Signal Masking System Call

Signals are usually reliable, even though unreliable signals did exist. Signals are delivered asynchronously, usually on return from system call. Multiple instances of some signals may be queued.

```
int kill (pid_t pid, int sig);
int pthread_kill (pthread_t thread, int sig);

union sigval
{
  int sival_int;
  void *sival_ptr;
}

int sigqueue (pid_t pid, int sig, const union sigval value);
```

Figure 2-17. Signal Send System Call

Example: System V Message Passing

Jako první příklad message passing lze asi uvést System V message passing API. Zpráva tam vypadá jednoduše, na začátku je long message type, za ním následuje pole bajtů, jejichž délka se udává jako další argument při volání API. Volání jsou pak triviální:

```
int msgsnd (int que, message *msg, int len, int flags);
int msgrcv (int que, message *msg, int len, int type, int flags);
```

Při odesílání zprávy lze specifikovat, zda se má při zaplnění bufferu zablokovat volající proces nebo vrátit chyba, jako drobný detail i zablokovanému volajícímu procesu se může vrátit chyba třeba pokud se zruší message queue.

Při příjmu zprávy se udává maximální velikost bufferu, flagy říkají zda se větší zprávy mají oříznout nebo zda se má vrátit chyba. Typ zprávy může být buď 0, což znamená any message, nebo konkrétní typ, pak se ve flazích dá říci zda se vrátí první zpráva uvedeného nebo jiného než uvedeného typu. Záporný argument pak znamená přijmout zprávu s nejnižším typem menším než je absolutní hodnota argumentu. Ve flazích se samozřejmě dá také říci, zda se čeká na zprávu.

Adresuje se pomocí front zpráv. Fronta se vytvoří voláním int msgget (key, flags), ve kterém se uvádí identifikátor fronty a flagy. Identifikátor je globální, případně může mít speciální hodnotu IPC_PRIVATE, která zaručuje vytvoření nové fronty. Přístup ke frontě ovlivňují access rights ve flazích, ty jsou stejné jako například u souborů.

```
int msgget (key_t key, int msgflg);
```

Example: D-Bus Message Passing

To be done.

Example: Mach Message Passing

V Machu jsou procesy vybaveny frontami zpráv spravovanými kernelem, těmto frontám se říká porty. Oprávnění k práci s portem jsou uložena v tabulkách pro každý proces spravovaných kernelem, těmto oprávněním se říká capabilities. Proces identifikuje port jeho handlerem, což je index do příslušné tabulky capabilities. Capability může opravňovat číst z portu, zapisovat do portu, nebo jednou zapsat do portu. Pouze jeden proces může mít právo číst z portu, to je jeho vlastník. Při startu je proces vybaven několika významnými porty, například process portem pro komunikaci s kernelem, syscalls jsou pak posílání zpráv na tento port.

Zpráva v Machu se skládá z hlavičky, ta obsahuje destination a reply port handler, velikost zprávy, kernelem ignorované message kind a function code pole, a potom posloupnost datových polí tvořících zprávu. Zvláštností Machu je, že data zprávy jsou tagged, tedy před každou položkou je napsáno co je zač. Tag obsahuje out of line flag, velikost dat v počtu položek, velikost položky v bitech, a konečně typ položky, ten je ze standardních typů plus handler portu. Kernel interpretuje předání handleru portu jako předání příslušné capability.

Pro odeslání a příjem zpráv slouží volání mach_msg, to má jako argument adresu hlavičky zprávy, flags s bity jako expect reply, enable timeout, enable delivery notification, velikost zprávy, velikost místa na odpověď, plus porty.

Remote Procedure Call

Volání služby serveru pomocí zprávy z klienta má obvykle charakter volání procedury, a tak se kód pro manipulaci se zprávami na klientovi a serveru odděluje a automaticky generuje, nápad zhruba kolem roku 1984.

Když se volá normální procedura, uloží se na stack parametry, procedura si je vyzvedne a něco udělá, vrátí výsledky. Když se volá služba na serveru, parametry se uloží do zprávy, server ji přijme a něco udělá, vrátí výsledky. RPC udělá lokální proceduru, která vezme parametry ze stacku, uloží je do zprávy, zavolá server, přijme výsledky a vrátí je volajícímu. A aby i programátoři serveru měli pohodu, udělá se to samé také na druhé straně - říká se tomu client a server stub, případně client stub a server skeleton.

Uložení parametrů do zprávy se říká marshalling, opačně zase unmarshalling. Závisí na typu parametrů, které se předávají.

- Passed by value. Jediným problémem může být nekompatibilita reprezentací parametru. Ta se řeší bud' stanovením společného formátu (+ krátké zprávy, - někdy oba zbytečně převádí), nebo uváděním formátu ve zprávě (+ flexibilní, - složité stuby a delší zprávy).
- Passed by reference. Nejtěžší varianta. U reference na dobře typovaná malá data se dá převést na obousměrné by value (+ jednoduché a efektivní, - nemá přesně tutéž sémantiku při existenci cyklů referencí), u velkých dat je vhodnější když server žádá dodatečně klienta o data (+ flexibilnější, - složitější protokoly a neefektivní dereference). Některé reference se prakticky nedají přenést, typickým příkladem je předání funkce jako parametru.

S předáváním jsou ještě další záludnosti, které nejsou na první pohled zřejmé. Mezi ně patří:

- Global variables. Pochopitelně nejsou u serveru dostupné, ale ze sémantiky procedure callu to není zjevné, tak se na to zapomíná. Hlavně to vadí u takových věcí jako jsou globální error resulty. Ručně vytvořené stuby to umí dodělat, automaticky generované už ne.
- System identifiers. Pokud se předává nějaká hodnota, která má význam pro kernel klienta, nemusí už znamenat totéž u serveru. Typicky handlery souborů, čísla portů a podobně. Zmínit konverzi při posílání zpráv u Machu.

Další problém je error handling. S tím moc chytristiky udělat nejde. Možné varianty selhání jsou známé, je prostě nutné počítat s tím, že RPC může selhat ještě pár jinými způsoby než normální call a ošetřit to v programu.

Při implementaci RPC je důležitá efektivita, stojí na ní celý systém. Kritická cesta při RPC - call stub, prepare msg buffer, marshall params, fill headers, call kernel send, context switch, copy stub buffer to kernel space, fill headers, set up interface - receive interrupt, check packet, find addressee, copy buffer to addressee, wake up addressee, context switch, unmarshall params, prepare stack, call server.

Co trvá dlouho - marshalling, buffer copying (při špatné implementaci header filling). Řeší se obvykle mapováním a scatter and gather network hardware (efektivní jen pro delší zprávy).

Stuby a skeletony je potřeba automaticky generovat. Jako vstup generátoru slouží definice hlaviček procedur, ty jazykové ale nejsou zpravidla dostatečně informativní, takže se definuje nějaký jazyk pro popis hlaviček procedur (IDL), podle kterého se pak jednak generují stuby a skeletony a jednak hlavičky procedur v nějakém programovacím jazyce.

Example: Spring Remote Procedure Call

Na právě popsaném principu běží například Spring, kde se procesy volají skrz doors. Při volání door se předává buffer, který může obsahovat data, identifikátor door, out of line data. Předávání je buď consume nebo copy, s jasnou sémantikou. Thread na straně klienta se pozastaví, na straně serveru se vybere thread z thread pool příslušejícího k door, který vykoná kód spojený s door. Interfaces jsou popsané v IDL, překládá se do client a server stubů, pod nimi jsou ještě subcontracts, ignore.

Pro marshalling Spring původně používal buffer fixní velikosti spojený s každým threadem, to se ale ukázalo špatné ze dvou důvodů. Za prvé, většina volání přenášela méně než 128 bajtů dat (90% pod 48 bajtů), a několikakilobajtový buffer byl pak zbytečně velký. Za druhé, buffery se rezervovaly staticky, čímž spotřebovávaly paměť. Jako řešení se udělal stream interface s metodami put_int, put_short, put_char, put_bulk, put_door_identifier, put_aligned (a odpovídajícími get metodami). Stream si by default alokuje buffer 128 bajtů, do kterého od konce ukládá structured data (door identifiers a out of line data) a od začátku unstructured data (všechno ostatní). Structured data se překládají, unstructured kopírují, při zaplnění se alokuje extra overflow buffer.

Rehearsal

At this point, you should know how processes can exchange information. You should be able to distinguish the various ways of exchanging information based on their applicability, efficiency and utility.

You should be able to characterize basic properties of message passing mechanisms and to relate these properties to both the architecture of the operating system and the requirements of the applications.

Based on your knowledge of how processes communicate using message passing, you should be able to design an intelligent message passing API.

You should be able to explain how remote procedure calls mimic local procedure calls and how certain issues limit ideal substitutability of the two mechanisms. You should be able to explain why the code of stubs can be generated and what information is necessary for that.

Questions

1. Propose an interface through which a process can set up a shared memory block with another process.
2. Define synchronous and asynchronous message passing.
3. Define blocking and non blocking message sending and reception.
4. Explain how polling can be used to achieve non blocking message reception.

5. Explain how callbacks can be used to achieve non blocking message reception.
6. Explain when a synchronous message sending blocks the sender.
7. Explain when an asynchronous message sending blocks the sender.
8. Propose a blocking interface through which a process can send a message to another process. Use synchronous message passing with direct addressing.
9. Propose a blocking interface through which a process can send a message to another process. Use asynchronous message passing with indirect addressing.

Exercises

1. Design a process communication mechanism based on message passing suitable for a microkernel operating system. Describe the interface used by a process to communicate with other processes. Include specialized support for very short messages that would be communicated as quickly as possible, and specialized support for very long messages that would be communicated as efficiently as possible. Compare the overhead introduced by the process communication mechanism with the overhead of a local procedure call.

Process Synchronization

When concurrently executing processes communicate among themselves or use shared resources, they can obviously influence each other. This influence can lead to errors that only exhibit themselves in certain scenarios of concurrent execution. Such errors are called *race conditions.*

Bernstein conditions from 1966 state that given sets of inputs and sets of outputs for concurrently executing processes, race conditions can only occur when either sets of outputs of two processes overlap, or a set of inputs of a process overlaps with a set of outputs of other processes.

Race conditions are notoriously difficult to discover. Process synchronization provides means of avoiding race conditions by controlling or limiting the concurrency when executing code where race conditions can occur. This code is typically denoted as *critical sections.*

Synchronization Problems

To better understand what kind of process synchronization is necessary to avoid race conditions, models of synchronization problems are used. Each model depicts a particular scenario of concurrent execution and presents particular requirements on process synchronization.

Petri nets are often used to describe the synchronization problems. Petri net consists of places and transitions. Places can hold tokens, transitions can fire by consuming input tokens and producing output tokens. Roughly, places correspond to significant process states, transitions correspond to significant changes of process state.

References

1. Carl Adam Petri: Kommunikation mit Automaten.

Mutual Exclusion

Mutual Exclusion models a scenario where several processes with critical sections execute concurrently. The synchronization problem requires that no two processes execute their critical sections simultaneously.

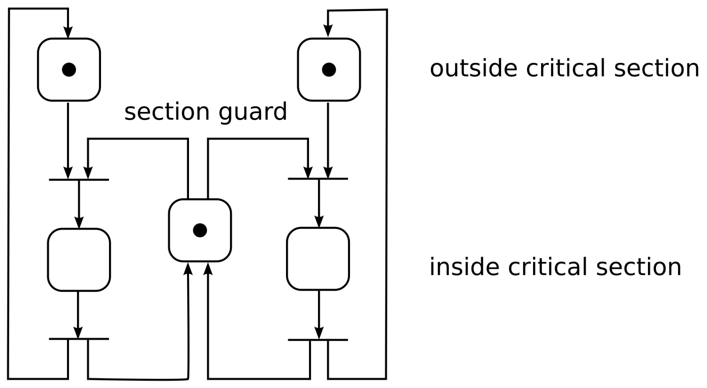

Figure 2-18. Mutual Exclusion Petri Net

Rendez Vous

Rendez Vous models a scenario where several processes must reach a given state simultaneously.

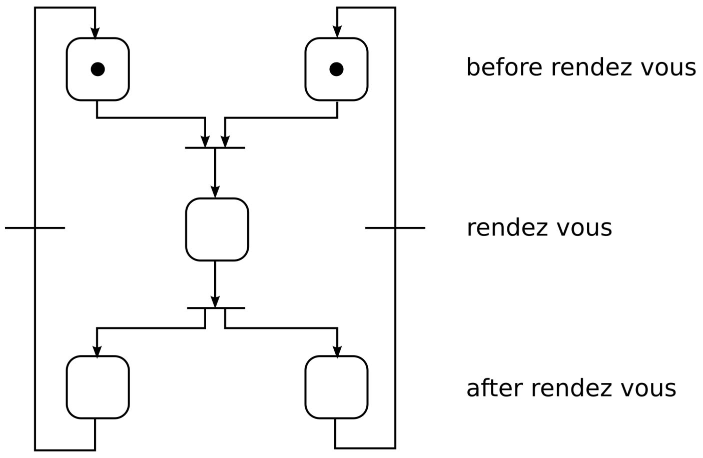

Figure 2-19. Rendez Vous Petri Net

Producer And Consumer

Producer And Consumer models a scenario where several processes produce items and several processes consume items. The items are stored in a buffer of a limited size. The synchronization problem requires that the buffer neither underflows nor overflows, or, in other words, that no producer attempts to put an item into a full buffer and that no consumer attempts to get an item from an empty buffer.

Readers And Writers

Readers And Writers models a scenario where several processes write shared data and several processes read shared data. The synchronization problem requires that no two writers write the data simultaneously and that no reader reads the data while it is being written.

Dining Philosophers

Dining Philosophers models a scenario where several philosophers alternatively think and dine at a round table. The table contains as many plates and forks as there are philosophers. A philosopher needs to pick two forks adjacent to his plate to dine.

The problem approximates a situation where several processes compete for an exclusive use of resources with the possibility of a deadlock.

Sleeping Barber

Sleeping Barber models a scenario where several customers visit a barber in a barber shop. The shop contains a limited number of seats for waiting customers. The barber serves customers one at a time or sleeps when there are no customers. A customer enters the shop and either wakes the barber to be served immediately, or waits in a seat to be served later, or leaves when there are no free seats.

The problem approximates a situation where several processes queue to get served by another process.

Means For Synchronization

The most trivial example of process synchronization is exclusive execution, which prevents all but one process from executing. Technical means of achieving exclusive execution include disabling processor interrupts and raising process priority.

Disabling interrupts yields exclusive execution in an environment that uses interrupts to schedule multiple processes on a single processor, simply because when no interrutps arrive, no scheduling happens. Since interrupts are used to service devices, disabling interrupts can lead to failure in servicing devices. As such, disabling interrupts is only permitted to privileged processes, which should limit disabling interrupts to short periods of time. Disabling interrupts does not yield exclusive execution on systems with multiple processors.

Active Waiting

Active waiting is an approach to process synchronization where a process that waits for a condition does so by repeatedly checking whether the condition is true. In the following, multiple solutions to the mutual exclusion problem based on active waiting are developed to illustrate the concept.

Assume availability of shared memory that can be atomically read and written. A shared boolean variable `bCriticalSectionBusy` can be used to track the availability of the critical section. A naive solution to the mutual exclusion problem would consist of waiting for the variable to become false before entering the critical section, setting it to true upon entering, and setting it to false upon leaving.

```
while (bCriticalSectionBusy)
{
  // Active waiting cycle until the
  // bCriticalSectionBusy variable
  // becomes false
}
bCriticalSectionBusy = true;

// Code of critical section comes here
...

bCriticalSectionBusy = false;
```

The principal flaw of the naive solution can be revealed by considering what would happen if two processes attempted to enter the critical section at exactly the same time. Both processes would wait for the `bCriticalSectionBusy` to become false, both would see it become false at exactly the same time, and both would leave the active waiting cycle at exactly the same time, neither process noticing that the other process is about to enter the critical section.

Staying with two processes, the flaw of the naive solution can be remedied by splitting the `bCriticalSectionBusy` variable into two, each indicating the intent of one process to enter the critical section. A process first indicates its intent to enter the critical section, then checks if the other process indicates the same intent, and enters the critical section when alone or backs off when not.

```
while (true)
{
  // Indicate the intent to enter the critical section
  bIWantToEnter = true;
  // Enter the critical section if the other
  // process does not indicate the same intent
  if (!bHeWantsToEnter) break;
  // Back off to give the other process
  // a chance and continue the active
  // waiting cycle
  bIWantToEnter = false;
}

// Code of critical section comes here
...

bIWantToEnter = false;
```

The solution is *safe* in that, unlike its naive predecessor, it never lets more than one process into the critical section. Unfortunately, a process waiting to enter the critical section can be overtaken infinitely many times, violating the *fairness* property. Additionally, all processes waiting to enter the critical section can form an infinite cycle, violating the *liveness* property.

A safe solution that also guarantees bounded waiting is known as the *Dekker Algorithm*.

```
// Indicate the intent to enter the critical section
bIWantToEnter = true;
while (bHeWantsToEnter)
{
  // If the other process indicates the same intent and
  // it is not our turn, back off to give the other
  // process a chance
  if (iWhoseTurn != MY_TURN)
  {
    bIWantToEnter = false;
    while (iWhoseTurn != MY_TURN) { }
    bIWantToEnter = true;
  }
}

// Code of critical section comes here
...

iWhoseTurn = HIS_TURN;
bIWantToEnter = false;
```

Another similar algorithm is the *Peterson Algorithm.*

```
// Indicate the intent to enter the critical section
bIWantToEnter = true;
// Be polite and act as if it is not our
// turn to enter the critical section
iWhoseTurn = HIS_TURN;
// Wait until the other process either does not
// intend to enter the critical section or
// acts as if its our turn to enter
while (bHeWantsToEnter && (iWhoseTurn != MY_TURN)) { }

// Code of critical section comes here
...

bIWantToEnter = false;
```

Other variants of the two algorithms exist, supporting various numbers of processes and providing various fairness guarantees. When the only means for synchronization is a shared memory that supports atomic reads and writes, any fair deterministic solution of the mutual exclusion problem for N processes has been proven to need at least N shared variables.

From practical point of view, our assumption that shared memory can only be atomically read and written is broadly correct but often too stringent. Many processors offer atomic operations such as test-and-set or compare-and-swap, which test wheter a shared variable meets a condition and set its value only if it does. The utility of these operations is illustrated by fixing the naive solution to the mutual exclusion problem, which is made safe by using the `AtomicSwap` operation. The operation sets a new value of a shared variable and returns the previous value.

```
while (AtomicSwap (bCriticalSectionBusy, true))
{
  // Active waiting cycle until the
  // value of the bCriticalSectionBusy
  // variable has changed from false to true
}

// Code of critical section comes here
...

bCriticalSectionBusy = false;
```

When the only means for synchronization is a shared memory that supports atomic compare-and-swap alongside atomic reads and writes, any fair deterministic solution of the mutual exclusion problem for N processes has been proven to need at least N/2 shared variables.

Active waiting is useful when the potential for contention is relatively small and the duration of waiting is relatively short. In such cases, the overhead of active waiting is smaller than the overhead of passive waiting, which necessarily includes context switching. Some situations also require the use of active waiting, for example when there is no other process that would wake up the passively waiting process.

Passive Waiting

Passive waiting is an approach to process synchronization where a process that waits for a condition does so by sleeping, to be woken by another process that has either caused or observed the condition to become true.

Consider the solution to the mutual exclusion problem using the `AtomicSwap` operation. A naive extension of the solution to support passive waiting uses the `Sleep` and `Wake` operations to remove and add a process from and to the ready queue, and the `oWaitingProcesses` shared queue variable to keep track of the waiting processes.

```
if (AtomicSwap (bCriticalSectionBusy, true))
{
  // The critical section is busy, put
  // the process into the waiting queue
  oWaitingProcesses.Put (GetCurrentProcess ());
  // Wait until somebody wakes the process
  Sleep ();
}

// Code of critical section comes here
...

// See if any process is waiting in the queue
oWaitingProcess = oWaitingProcesses.Get ();

if (oWaitingProcess)
{
  // A process was waiting, let it enter the critical section
  Wake (oWaitingProcess);
}
else
{
  // No process was waiting, mark the critical section as free
  bCriticalSectionBusy = false;
}
```

One major flaw of the naive solution is that the decision to wait and the consecutive adding of the process to the wait queue and removing of the process from the ready queue are not performed atomically. It is possible that a process decides to wait just before the critical section becomes free, but is only added to the wait queue and removed from the ready queue after the critical section becomes free. Such a process would continue waiting even though the critical section would be free.

Another major flaw of the naive solution is that the access to the shared queue variable is not synchronized. The implementation of the shared queue variable would be a critical section in itself.

Both flaws of the naive solution can be remedied, for example by employing active waiting both to make the decision to wait and the consecutive queue operations atomic and to synchronize access to the shared queue variable. The solution is too long to be presented in one piece though.

Passive waiting is useful when the potential for contention is relatively high or the duration of waiting is relatively long. Passive waiting also requires existence of another process that will wake up the passively waiting process.

Nonblocking Synchronization

From practical perspective, many synchronization problems include bounds on waiting. Besides the intuitive requirement of fairness, three categories of solutions to synchronization problems are defined:

- A *wait free* solution guarantees every process will finish in a finite number of its own steps. This is the strongest category, where bounds on waiting always exist.
- A *lock free* solution guarantees some process will finish in a finite number of its own steps. This is a somewhat weaker category, with the practical implication that starvation of all processes will not occur and progress will not stop should any single process block or crash.
- An *obstruction free* solution guarantees every process will finish in a finite number of its own steps provided other processes take no steps. This is an even weaker category, with the practical implication that progress will not stop should any single process block or crash.

To be done.

Wait free hierarchy based on consensus number. Shared registers (1), test and set, swap, queue, stack (2), atomic assignment to N shared registers (2N-2), memory to memory move and swap, queue with peek, compare and swap (infinity).

Impossibility results. Visible object for N processes is an object for which any of N processes can execute a sequence of operations that will necessarily be observed by some process, regardless of the operations executed by other processes. An implementation of a visible object for N processes over shared registers takes at least N of those shared registers if the registers can do either read and write or conditional update, or at least N/2 of those shared registers if the registers can do both read and write and conditional update. The same goes for starvation free mutual exclusion.

Randomized synchronization primitives.

References

1. Maurice Herlihy: Wait-Free Synchronization.
2. Faith Fich, Danny Hendler, Nir Shavit: On the Inherent Weakness of Conditional Synchronization Primitives.

Memory Models

The synchronization examples presented so far have assumed that operations happen in the same order as specified in code. While this facilitates understanding, it is not necessarily true - the operations in code are translated by the compiler into processor instructions, and the processor instructions direct the processor to execute series of memory operations - and both the compiler and the processor may decide to change the order of operations, typically to improve execution efficiency.

Obviously, it is not possible to permit arbitrary changes to memory operation order. Compilers and processors therefore adhere to *memory models*, which are formal sets of rules that define how memory operations behave. A memory model is often designed to approximate a reasonably intuitive behavior for most programs, and to only provide exceptions where such behavior would lead to a significant loss of efficiency. Such reasonable behavior may be *sequential consistency* or *linearizability*.

- Sequential consistency assumes that a program consists of multiple threads and each thread executes a sequence of atomic operations. A sequentially consistent behavior is a behavior that can be observed in some interleaving of the operations executed by the threads, the order of operations of individual threads is not changed.

- Linearizability assumes that a program consists of multiple threads and each thread executes a sequence of steps that perform operations on objects. Each operation is called and returns at some step. A linearizable behavior is a behavior that can be observed if each operation takes place atomically some time between the call and the return without breaking the semantics of the operation, the order of steps of individual threads is not changed.

Following are brief examples of some memory models. As a disclaimer, these are all necessarily inaccurate explanations that summarize multiple pages of formal memory model definitions into a few paragraphs.

References

1. Hans Boehm: Threads Basics.
2. Hans Boehm: Threads Cannot Be Implemented As A Library.

Example: Memory Model On Intel 80x86 Processors

The Intel 80x86 processors guarantee that all read and write instructions operating on shared memory are atomic when using aligned addresses. Other instructions may or may not be atomic depending on the particular processor. In particular, read and write instructions operating within a single cache line are often atomic, while read and write instructions operating across cache lines are not. Read-modify-write instructions can be made atomic using a special LOCK prefix.

Also starting with the Intel Pentium 4 processors, multiple memory ordering models were introduced to enable optimization based on reordering of memory accesses. The basic memory ordering model works roughly as follows:

- Reads by a single processor are carried out in the program order.
- Most writes by a single processor are carried out in the program order.
- Reads and writes by a single processor to the same address are carried out in the program order.
- Younger reads and older writes by a single processor to different addresses are not necessarily carried out in program order.
- Writes by a single processor are observed in the program order by the other processors.
- Writes by multiple processors are not necessarily observed in the same order by these processors, but are observed in the same order by the other processors.
- Reads and writes are causally ordered.

Other memory ordering models include the *strong ordering* model, where all reads and writes are carried out in the program order, or the *write back ordering* model, where writes to the same cache line can be combined. The memory ordering models can be set on a per page and a per region basis.

The LFENCE, SFENCE, MFENCE instructions can be used to force ordering. The LFENCE instruction forms an ordering barrier for all load instructions, the SFENCE instruction forms an ordering barrier for all store instructions, the MFENCE instruction does both LFENCE and SFENCE.

Starting with the Intel Pentium 4 processors, the processor family introduced the MONITOR and MWAIT instruction pair. The MONITOR instruction sets up an address to monitor for access. The MWAIT instruction waits until the address is accessed. The purpose of the instruction pair is to optimize multiprocessor synchronization, because the processor is put into power saving mode while waiting for the access.

The PAUSE instruction can be used inside an active waiting cycle to reduce the potential for collisions in instruction execution and to avoid executing the active waiting cycle to the detriment of other work on hyperthreading processors.

References

1. Intel: Intel 64 and 32 Architectures Software Developer Manual.

Example: Memory Model On MIPS32 Processors

The MIPS32 processors may implement a cache coherency protocol. One of five memory coherency models can be set on a per page basis.

- Uncached - the data are never cached, reads and writes operate directly on memory.
- Noncoherent - the data are cached, reads and writes operate on cache, no coherency is guaranteed.
- Sharable - the data are cached, reads and writes operate on cache, write by one processor invalidates caches of other processors.
- Update - the data are cached, reads and writes operate on cache, write by one processor updates caches of other processors.
- Exclusive - the data are cached, reads and writes by one processor invalidate caches of other processors.

The LL and SC instructions can be used to implement a variety of test-and-set and compare-and-swap operations. The LL instruction reads data from memory, and additionally stores the address that was read in the LLaddr register and sets the LLbit register to true. The processor will set the LLbit register to false whenever another processor performs a cache coherent write to the cache line containing the address stored in the LLaddr register. The SC instruction stores data to memory if the LLbit register is true and returns the value of the LLbit register.

References

1. MIPS Technologies: MIPS32 4K Processor Core Family Software User Manual.
2. Joe Heinrich: MIPS R4000 Microprocessor User Manual.

Example: Memory Model In C++

The memory model defines the order within a single thread by a *sequencing* relation. Even a single thread may have operations that are not sequenced with respect to each other, these are often expressions that have multiple side effects on the same data. Where a result depends on the order of operations that are not sequenced, the result is generally undefined.

Threads can perform *synchronization operations*. Typically, these are either atomic variable operations or library synchronization operations. A synchronization operation is classified as *release*, *acquire* or *consume*. An atomic variable write can be a release operation, an atomic variable read can be an acquire or a consume operation. A lock operation is an acquire operation, an unlock operation is a release operation.

- Release and later acquire of the same object gives rise to *synchronization order*.
- Release and later consume of the same object gives rise to *dependency order*.
- Together with sequencing, this creates the *happens-before order*.

Intuitively, the happens-before order captures operation ordering that is explicit in the program, that is, ordering that is either due to sequencing in a single thread or due to synchronization operations between threads. Operations that access the same object and include writes must be ordered by the happens-before order, otherwise they are considered a *data race* and their behavior is undefined. In general, the last write operation in the happens-before order is also the visible one.

The memory model considers locations that are unique except for bit fields, multiple adjacent bit fields share the same memory location. Memory locations are updated without interference.

Example: Memory Model In Java

The memory model of the Java programming language distinguishes *correctly synchronized* programs and programs with *races*. The formal definition of the two cases relies on programming language constructs that necessarily introduce ordering:

- The execution order prescribed by the program code for a single thread defines the *program order* of actions. Simply put, statements that come earlier in the program code are ordered before statements that come later.
- The execution order prescribed by synchronization between threads defines the *synchronization order* of actions. For example, unlocking a lock comes before locking the same lock later. Besides locks, synchronization order also considers thread lifecycle actions and accesses to volatile variables.

Both the program order and the synchronization order are evident and intuitive. The memory model further defines a *happens-before order*, which can be roughly described as a transitive closure of the program order and the synchronization order.

To define whether a program is correctly synchronized, the memory model asks whether potentially dangerous accesses to shared variables can occur in certain reasonable executions of the program:

- Each execution of a program is defined as a total order over all program actions.
- Reasonable executions are executions where actions observe the program order and the synchronization order and where all reads return values written by the latest writes. These executions are called *sequentially consistent*.
- Potentially dangerous accesses to shared variables are accesses to the same variable that include writes, when the accesses are not ordered by the happens-before order. Such accesses constitute a *data race*.

A program is said to be correctly synchronized when its sequentially consistent executions contain no data race.

When accessed in all its formal glory, the definition of correctly synchronized programs is complex. Still, it has a very intuitive motivation. We know that it is potentially dangerous to access shared variables without synchronization. The memory model tells us that a program is correctly synchronized if all accesses to shared variables are ordered through synchronization in any possible interleaving of the individual program threads.

In reality, program execution is not limited to interleaving of the individual threads. Both the compiler and the processor may decide to change the order of certain actions. For programs that are correctly synchronized, the memory model guarantees that any execution will appear to be sequentially consistent - that is, whatever reordering happens under the hood, the observed effects will always be something that could happen if the individual threads were simply interleaved.

For programs with races, the memory model guarantees are weaker. Reads will not see writes in contradiction of the happens-before order. One, a read will not see a write when that write should come later than the read. Two, a read will not see a write when that write should be hidden by a later write that itself comes sooner than the read. The memory model also prohibits causality loops, the formal requirement is based on committing individual actions within executions in a manner that prevents causality loops from occuring.

Accesses to all basic types except long and double are atomic in Java.

References

1. James Gosling, Bill Joy, Guy Steele, Gilad Bracha: The Java Language Specification.

Synchronization And Scheduling

Convoys

To be done.

Priority Inversion

Priority inversion je situace, kdy procesy s vyšší prioritou čekají na něco, co vlastní procesy s nižší prioritou. V nejhorším případě může priority inversion vést i k deadlocku. Řešením priority inversion může být priority inheritance.

Inversion and active and passive waiting ...

Z principu se podpora priority inheritance zdá být jednoduchá. V okamžiku, kdy proces začne na něco čekat, se jeho priorita propůjčí procesu vlastnícímu to, na co se čeká. Problém je v tom, že tohle funguje dobře u zámků, které mají jednoho vlastníka, ale u semaforů nebo condition variables se už nedá zjistit, kdo vlastně bude ten proces, který vázaný prostředek uvolní, a komu se tedy má priorita půjčit.

Řešení například v Solarisu, priority inheritance funguje přímočaře u mutexů, read write zámky zvýší prioritu prvního vlastníka a dalších už ne, condition variables nedělají nic.

Starvation And Deadlock

Ke hladovění dochází v případě, kdy je některý proces neustále odkládán, přestože by mohl běžet. Tohle lze dobře ukázat například u čtenářů a písařů, kde písař může prostě čekat, protože někdo pořád čte. Při řešení synchronizačních úloh je proto často důležité, aby použité algoritmy zaručovaly, že nedojde ke hladovění.

Pokud jde o uváznutí synchronizovaných procesů, v podstatě se nabízí tři možnosti, jak se s uváznutím vypořádat, totiž zotavení, prevence a vyhýbání se.

Zotavení je technika, při které systém detekuje vznik deadlocku a odstraní ho. Hned dva problémy, jak detekovat a jak odstranit. Dokud systém ví, kdo na koho čeká, může detekovat prostým hledáním cyklů v grafu. Jakmile se ale neví, kdo na koho čeká, nejde to (e.g. aktivní čekání, user level implementace synchronizace a threadů). Mohou se použít náhradní techniky, například watchdogs, ale to není spolehlivé.

Pokud připustíme, že umíme deadlock detekovat, jeho odstranění také není triviální. Když se podíváme na prostředky, lze je rozdělit na preemptivní a nepreemptivní, pouze ty první lze procesu bezpečně odebrat. Mezi preemptivní prostředky se dá řadit například fyzická paměť, nepreemptivní je skoro všechno ostatní. Sebrat procesu nepreemptivní prostředek jen tak nejde, násilně ukončit proces může způsobit další problémy. Částečné řešení nabízejí transakce s možností rollbacku a retry.

Prevence uváznutí znamená, že procesy se naprogramují tak, aby nemohly uváznout. Aby mohly procesy uváznout, musí současně platit čtyři podmínky:

- procesy musí čekat na prostředky v cyklu
- procesy musí prostředky vlastnit výhradně
- procesy musí být schopny přibírat si vlastnictví prostředků
- prostředky nesmí být možné vrátit

Řešení jsou pak založena na odstranění některé z těchto podmínek, na první je to například uspořádání prostředků a jejich získávání v pořadí určeném tímto uspořádáním, na druhou virtualizace, na třetí současné zamykání všech prostředků, na čtvrtou spin styl zamykání prostředků.

Vyhýbání se deadlocku spočívá v tom, že procesy předem poskytují dost informací o tom, na které prostředky budou ještě čekat. Samozřejmě, to je problematické, ale občas se to dělá.

Ze škatulky vyhýbání se deadlocku je i bankéřův algoritmus. Jeho jméno pochází z modelové situace, kdy bankéř nabízí zákazníkům půjčky do určitého limitu, a jeho celkový kapitál je menší než počet zákazníků krát limit. Při každé žádosti o půjčku bankéř zkontroluje, zda po půjčení zůstane dost peněz na to, aby si alespoň jeden zákazník mohl vybrat plný limit, postupně pro všechny zákazníky. Pokud ano, půjčí, pokud ne, čeká. Předpokládá se, že pokud si alespoň jeden zákazník může vybrat plný limit, časem bude muset něco vrátit, a tím budou peníze na uspokojení ostatních zákazníků.

What Is The Interface

Když už víme, kdy a proč a jak synchronizovat, zbývá se ještě podívat na to, jaké prostředky k synchronizaci má operační systém dát aplikacím. Samozřejmě, z těchto prostředků vypadávají takové věci jako je zakázání a povolení přerušení, protože k těm nemůže operační systém aplikaci pustit. Podobně těžké je to s aktivním čekáním, protože procesy nemusí vždy sdílet paměť. Takže co zbývá ...

Důležitým faktem je, že dokud se pohybujeme v oblasti procesů sdílejících paměť, stačí nám jeden synchronizační prostředek k naprogramování ostatních. Odtud pak oblíbené úlohy na naprogramování jednoho synchronizačního prostředku pomocí jiného.

Atomic Operations

To be done.

Barriers

To be done.

Locks

Zámky alias mutexy jsou jednoduchým prostředkem k synchronizaci na kritické sekci. Zámek má metody lock a unlock se zjevnou funkcí, pouze jeden proces může mít v kterémkoliv okamžiku zamknutý zámek. Implementace jednoduchá, lock otestuje zda je zamčeno, pokud ne tak zamkne, pokud ano tak nechá proces čekat, unlock spustí další čekající proces, pokud nikdo nečeká tak odemkne. Samozřejmě, v implementaci jsou potřeba atomické testy.

Nakreslit implementaci zámku a ukázat, jak je důležitý atomický test, a jak jej lze zajistit bud' atomickou instrukcí, nebo zákazem přerušení.

Ukázat příklad, jak lze pomocí mutexu vyřešit nějakou synchronizační úlohu, nejlépe prostě vzájemné vyloučení.

V Linuxu je k dispozici mutex od pthreadů. Je reprezentován datovou strukturou pthread_mutex_t, inicializuje se voláním int pthread_mutex_init (pthread_mutex_t *, pthread_mutexattr_t *), ničí se voláním int pthread_mutex_destroy (pthread_mutex_t *) na odemčeném mutexu, pro práci s ním jsou metody _lock (), _trylock () a _unlock (). Atributy mutexu nastavují co se stane pokud thread zkusí znovu zamknout mutex, který již jednou zamknul, fast mutexy se deadlocknou, error checking mutexy vrátí chybu, recursive mutexy si pamatují kolikrát se zamklo. Podobná situace je při odemykání, fast a recursive mutexy může odemknout každý, u recursive mutexů se testuje vlastník, to je ale na rozdíl od zamykání nepřenositelný detail, zmiňuje se jen aby bylo vidět že existuje také koncept vlastníka zámku.

Když se podíváme na implementaci, pthread_mutex_t je malinká struktura obsahující krom prázdných polí frontu čekajících threadů, stav mutexu, counter rekurzivního zamykání, pointer na vlastníka a typ mutexu. Implementace vlastních operací je pak jednoduchá, sdílení mezi procesy (pokud jej systém umí) se zařizuje pomocí shared memory.

For active waiting, Posix threads library provides spin locks available through the pthread_spinlock_t data structure and pthread_spin_ functions, which are analogical to the mutex functions (except there is no _timedlock variant). Upon initialization, the pshared flag specifies if the spin lock can be used only by threads inside one process, or also by different processes, provided that the spin lock is allocated in a shared memory area.

With mutexes available in user space but threads implemented in kernel space, it is unavoidable that some mutex operations have to call the kernel. It is, however, possible to optimize mutex operations for the case without contention, so that the kernel does not have to be called when no scheduling is needed. Linux makes this optimization possible through its futex interface.

When called with `op` set to `FUTEX_WAIT`, the interface suspends the current thread if the value of the futex equals to `val`. When called with `op` set to `FUTEX_WAKE`, the interface wakes at most `val` threads suspended on the futex.

A simplified example of implementing a mutex using a futex is copied from Drepper.

References

1. Ulrich Drepper: Futexes Are Tricky. http://people.redhat.com/drepper/futex.pdf

Windows NT mají také mutexy, a to hned dvojího druhu. Jedním se říká critical sections, druhým mutexes. Nejprve critical sections:

Critical sections ve Windows NT si pamatují vlastníka, je možné je zamknout jedním threadem několikrát (a tolikrát se musí odemknout). Jsou (víceméně) rychlé, ale nefungují mezi procesy (pochopitelně).

Pro synchronizaci mezi procesy se ve Windows NT používají kernel objekty. Z těch nás momentálně zajímá mutex.

Parametr lpsa určuje security sdělení, nezajímá nás. FInitialOwner říká, zda bude mutex po vytvoření okamžitě zamčený pro volajícího. LpszMutexName umožňuje pojmenovat mutex. Dva procesy mohou sdílet mutex tak, že jej vytvoří pod stejným jménem (případně lze zavolat HANDLE OpenMutex (DWORD fdwAccess, BOOL fInherit, LPTSTR lpszName)). Jinou metodou sdílení (beze jména) je volání BOOL DuplicateHandle (HANDLE hSourceProcess, HANDLE hSource, HANDLE hTargetProcess, LPHANDLE lphTarget, DWORD fdwAccess, BOOL fInherit, DWORD fdwOptions), které umožňuje zduplikovat handle (handle se nedá předat rovnou, je process-specific).

Čeká se pomocí DWORD WaitForSingleObject (object, timeout), vrací OK, TIMEOUT, OWNER_TERMINATED. Také funguje WaitForMultipleObjects, viz výše. Mutexy mají vlastníky a lock count stejně jako critical sections. Odemyká se pomocí BOOL ReleaseMutex (mutex).

Zámků může existovat více verzí, konec konců podobně jako jiných synchronizačních primitiv. Tak se můžete setkat s termíny spin lock pro zámek, který čeká na uvolnění aktivně (termínem spinning se rozumí právě opakované testování přístupu), blocking lock pro zámek, který čeká pasivně, recursive lock pro zámek, který lze zamknout vícekrát stejným threadem, read write lock pro zámek, který má režim zamčení pro čtení a pro zápis, atd. Více zamykání v transakcích.

Implementace odemknutí zámku může být v jednom detailu naprogramovaná dvěma způsoby. Poslední vlastník buď odemkne zámek a rozeběhne některý čekající proces, který zámek znovu zamkne, nebo jej prostě předá zamčený některému čekajícímu procesu. Druhá metoda se sice zdá efektivnější, ale má nepříjemnou vlastnost v situaci, kdy se někdo pokusí zamknout zámek ve chvíli, kdy se jej již vzdal starý vlastník, ale ještě se nerozběhl nový vlastník. V takové situaci skončí pokus o zamčení zablokováním volajícího, což může být pokládáno za špatnou věc (aktivní proces musí čekat na pasivní). Vytváření těchto závislostí mezi procesy se říká *convoys*.

Read Write Locks

To implement the Readers And Writers Synchronization Problem, a variant of a lock that distinguishes between readers and writers is typically provided. The lock can be locked by multiple readers, but only by a single writer, and never by both readers and writers.

Read write locks can adopt various strategies to resolve contention between readers and writers. Often, writers take precedence over readers.

Linux provides Read Write Locks as a part of the Posix Threads library.

Windows provide Slim Reader Writer Locks that can be used within a single process.

Seq Lock

To be done.

Read Copy Update

To avoid some of the blocking associated with implementing the Readers And Writers Synchronization Problem using read write locks, the read copy update interface lets readers operate on past copies of data when updates are done. This is achieved by splitting an update into the *modification* and *reclamation* phases. In the modification phase, the writer makes the updated copy of data visible to new readers but the past copy of data is retained for existing readers. In between the modification and reclamation phases, the writer waits until all readers of the past copy of data finish. In the reclamation phase, the writer discards the past copy of data. The interface does not deal with writer synchronization.

Linux provides Read Copy Update as a part of the kernel. The `rcu_read_lock` and `rcu_read_unlock` functions delimit readers. The `synchronize_rcu` function synchronizes writers by waiting until there are no active readers. The `rcu_assign_pointer` and `rcu_dereference` macros make sure atomicity or ordering does not break synchronization. For simplicity, the interface does not care what data is accessed, all readers are synchronized against all writers.

The interface permits many different implementations. When context switching can be prevented by readers, a straightforward implementation can leave the reader synchronization empty and wait for a context switch on each processor for writer synchronization.

Semaphores

Velmi podobné zámkům jsou semafory, BTW vymyslel je Dijkstra někdy v roce 1965. Semafor má metody signal a wait, často bohužel právě díky Dijkstrovi z holandštiny pojmenované P (passern, projít kolem) a V (vrijgeven, uvolnit), a initial count, který říká, kolik procesů smí současně vlastnit semafor.

Opět stručně nastínit implementaci s atomickou operací a řešení nějakého synchronizačního problému, například producent a konzument.

V Unixech podle System V, a tedy i v Linuxu, jsou semafory poskytovány systémem. Tyto semafory synchronizují procesy s odděleným adresovým prostorem, čemuž odpovídá i jejich interface. Semafor lze vytvořit voláním int semget (key, number, flags), které vrátí sadu semaforů globálně pojmenovanou daným klíčem. Se semafory se pak pracuje voláním int semop (key, ops_data *, ops_number). Každá ze sady operací obsahuje číslo semaforu v sadě, jednu ze tří operací se semaforem, a flagy. Operace je bud' přičtení čísla k semaforu, nezajímavé, nebo test semaforu na nulu, čeká se do okamžiku než semafor dosáhne nuly, nebo odečtení čísla od semaforu, čeká se do okamžiku než semafor bude mít dostatečně velkou hodnotu. Z flagů jsou zajímavé IPC_NOWAIT, který říká že se nemá čekat, a SEM_UNDO, který zajišt'uje, že operace na semaforu bude vrácena zpět, pokud proces, který jí volal, skončí. Operace se bud' udělají všechny nebo žádná. Pak je ještě semctl pro různé dalčí operace.

V Unixu jsou ještě semafory podle POSIX specifikace. Jejich interface je pochopitelně podobný pthread mutexům, inicializují se sem_init, další funkce jsou čekání, pokus o čekání, čtení hodnoty, signalizace, zničení semaforu.

Ve Windows jsou semafory podobně jako mutexy, jen nemají vlastníky a mají reference count.

ReleaseSemaphore se nepovede, pokud by se čítač semaforu zvětšil přes maximum specifikované při jeho vytvoření. Mimochodem není možné zjistit okamžitou hodnotu semaforu bez jeho změny, protože ReleaseSemaphore vyžaduje nenulový cRelease.

Condition Variables

Semafory a mutexy vždycky čekají na situace typu uvolnění obsazeného prostředku. Často je potřeba pasivně čekat na složité podmínky, například když nějaký thread zobrazuje stav jiných threadů v GUI a při změně má udělat repaint. Tam se pak hodí například condition variables.

Condition variable má metody wait, signal a broadcast. Pokud proces zavolá wait, začne pasivně čekat. Pokud někdo zavolá signal, vzbudí se jeden z právě čekajících procesů, pokud někdo zavolá broadcast, vzbudí se všechny právě čekající procesy. Využití na pasivní čekání na složité podmínky je pak nasnadě. Proces, který čeká, v cyklu střídá wait s testováním podmínky, kdokoliv pak může ovlivnit vyhodnocení podmínky dělá signal či broadcast. Jen jedno drobné zdokonalení, protože test podmínky musí být atomický, condition variable je svázána s mutexem, který chrání testovanou podmínku.

Implementace condition variable, nastínění použití uvnitř while.

Samozřejmě, operace condition variables je třeba používat s rozmyslem. Jednak je třeba mít na paměti, že u condition variable se signal před broadcast nezapočítá, na rozdíl od zámků a semaforů. Dvak, když se udělá broadcast, může dojít ke spuštění zbytečně velkého počtu procesů naráz.

V pthread library condition variables samozřejmě jsou. Vytvářejí se int pthread_cond_init (pthread_cond_t *, pthread_condattr_t *), další metody jsou _signal (), _broadcast (), _wait (), _timedwait (timeout) a _destroy (). Zřejmě není co dodat.

Úplně stranou, podobný mechanismus je možné najít třeba v Javě, kde thread může zavolat metodu wait na libovolném objektu, jiné thready ho pak pomocí notify nebo notifyAll mohou vzbudit. Podobně jako u klasických condition variables, i tady je možné tyto metody volat pouze pokud je příslušný objekt zamčen.

Mimochodem, condition variables jdou napsat dvěma způsoby, u jednoho po signal běží dál signaller, u druhého běží jeden z čekajících procesů. První způsob se občas také nazývá *Mesa Semantics*, druhý *Hoare Semantics*.

Windows provide Condition Variables that can be used within a single process.

Events

Windows events. Parametry jako obvykle, fManualReset říká, zda je event potřeba explicitně shazovat. čeká se také stejně (pomocí WaitForXxx), ale signalizuje se jinak. BOOL SetEvent (HANDLE hEvent) ohlásí event, u non manual reset events se po rozběhnutí jednoho čekajícího threadu event zase shodí. BOOL ResetEvent (HANDLE hEvent) shodí manual reset event. BOOL PulseEvent (HANDLE hEvent) nahodí manual reset event, počká až se rozběhnou všichni kdo na ní čekají a závěrem jí shodí.

PulseEvent údajně občas nemusí fungovat a jeho používání se nedoporučuje.

Monitors

Monitor dovoluje omezit paralelní přístup k objektu v programovacím jazyce, vymyslel ho pan Hoare v roce 1974 a najdete ho například v Concurrent Pascalu, Module, Javě.

Zřejmě nejjednodušší bude demonstrovat monitory právě na Javě. Ta má klíčové slovo synchronized, které lze uvést u metod objektu. Pokud je nějaká metoda takto označena, před jejím vykonáním se zamkne zámek spojený s jejím objektem, čímž je zajištěna synchronizace. Mimochodem Java nabízí také synchronizaci bloku kódu na explicitně uvedeném objektu, to je původem mírně starší koncept, který v podstatě dovoluje označit v kódu kritické sekce.

Pro případy, kdy je potřeba čekat na něco právě uvnitř monitoru, se k němu doplňují extra funkce. Jedno z možných provedení doplňuje funkce delay (queue) pro umístění procesu do fronty a continue (queue) pro vzbuzení jednoho procesu z fronty.

Note that spurious wake up in monitor wait is typically possible.

Guards

Poněkud méně známý synchronizační prostředek dovoluje zapsat před blok kódu podmínku, která musí být splněna, než se daný blok kódu začne vykonávat. Tím je v podstatě dosaženo podobné funkce jako u klasického použití condition variables., až na to, že nikdo nesignalizuje okamžik, kdy se má znovu otestovat podmínka. Což je také důvod, proč obecné guards téměř nikde nejsou k dispozici (okamžik otestování podmínky je těžké určit). Takže se dělají guards, které mají okamžiky otestování podmínky omezené na události jako je volání metody apod.

Příkladem takového guardu může být select a rendez vous v Adě. Ten se zapisuje pomocí příkazů select a accept, které jinak vypadají jako case a deklarace procedury:

Funkce je přímočará, select nejprve vyhodnotí všechny podmínky, pokud je u nějaké splněné podmínky k dispozici rendez vous, provede se, jinak se provede náhodně větev nějaké splněné podmínky nebo větev else, pokud není splněná žádná podmínka tak se hodí výjimka. Accept čeká dokud jej někdo nezavolá.

Rehearsal

At this point, you should be able to defend the need for process synchronization using a variety of practical examples. You should be able to describe the practical examples using formalisms that abstract from the specific details but preserve the essential requirement for synchronization.

You should be able to define precise requirements on process synchronization related to both correctness and liveness.

You should be able to demonstrate how process synchronization can be achieved using a variety of practical tools, including disabling interrupts, atomic reading and atomic writing of shared memory, atomic test and set over shared memory, atomic compare and swap over shared memory, message passing.

You should understand how process synchronization interacts with process scheduling. You should be able to explain how process synchronization can lead to scheduling anomalies.

You should demonstrate familiarity with both implementation and application of common synchronization primitives including barriers, signals, locks, semaphores, condition variables, monitors. You should be able to select and apply proper synchronization primitives to common synchronization problems.

Questions

1. Explain what is a race condition.
2. Explain what is a critical section.
3. Explain the conditions under which a simple

   ```
   I++
   ```
   code fragment on an integer variable can lead to a race condition when executed by multiple threads.
4. Explain the conditions under which omitting the

`volatile`
declaration on an integer variable can lead to a race condition when the variable is accessed by multiple threads.

5. Describe the Mutual Exclusion synchronization task. Draw a Petri net illustrating the synchronization task and present an example of the task in a parallel application.
6. Describe the Rendez Vous synchronization task. Draw a Petri net illustrating the synchronization task and present an example of the task in a parallel application.
7. Describe the Producer And Consumer synchronization task. Draw a Petri net illustrating the synchronization task and present an example of the task in a parallel application.
8. Describe the Readers And Writers synchronization task. Draw a Petri net illustrating the synchronization task and present an example of the task in a parallel application.
9. Describe the Dining Philosophers synchronization task. Draw a Petri net illustrating the synchronization task and present an example of the task in a parallel application.
10. Explain how a deadlock can occur in the Dining Philosophers synchronization task. Propose a modification of the synchronization task that will remove the possibility of the deadlock.
11. Explain the difference between active and passive waiting. Describe when active waiting and when passive waiting is more suitable.
12. Present a trivial solution to the mutual exclusion problem without considering liveness and fairness. Use active waiting over a shared variable. Explain the requirements that your solution has on the operations that access the shared variable.
13. Present a solution to the mutual exclusion problem of two processes including liveness and fairness. Use active waiting over a shared variable. Explain the requirements that your solution has on the operations that access the shared variable.
14. Describe the priority inversion problem.
15. Explain how priority inheritance can solve the priority inversion problem for simple synchronization primitives.
16. Present a formal definition of a deadlock.
17. Present a formal definition of starvation.
18. Present a formal definition of a wait free algorithm.
19. Describe the interface of a lock and the sematics of its methods.
20. Describe the interface of a read-write lock and the sematics of its methods.
21. Explain when a lock is a spin lock.
22. Explain when a lock is a recursive lock.
23. Explain why Windows offer `Mutex` and `CriticalSection` as two implementations of a lock.
24. Implement a solution for the mutual exclusion synchronization problem using locks.
25. Describe the interface of a semaphore and the sematics of its methods.
26. Implement a solution for the producer and consumer synchronization problem over a cyclic buffer using semaphores.
27. Describe the interface of a condition variable and the sematics of its methods.

28. Explain what is a monitor as a synchronization tool and what methods it provides.

Exercises

1. Implement a spin lock and then a recursive lock using the spin lock and the `Sleep` and `Wake` functions, as well as suitable functions of your choice for managing lists and other details, all without any guarantees as to parallelism. Explain how this implementation works on a multiprocessor hardware.

Chapter 3. Memory Management

Management Among Processes

Multiple Processes Together

To be done.

Single Partition

Primitivní řešení, bootstrap natáhne program, který má pro sebe celou paměť.

Nevýhody jsou zřejmé, chybí device drivery, software není přenositelný. Potřeba přenositelnosti, vznikají operační systémy, tou dobou v podstatě jen device drivery. Example CP/M.

Table 3-1. Struktura paměti CP/M

Adresa	Obsah
0000h-0002h	Warm start vector (JM
0005h-0007h	System call vector (JM
005Ch-006Bh	Parsed FCB 1
006Ch-007Bh	Parsed FCB 2
0080h-00FFh	Command tail area
0100h-BDOS	Transient program ar
BDOS-BIOS	BDOS
BIOS-RTOP	BIOS

Fixed Partitions

Paměť se při startu systému pevně rozdělí na partitions, do každé partition se umístí jedna aplikace. V závislosti na architektuře systému se mohou udělat buď oddělené fronty aplikací, které se budou zpracovávat v jednotlivých partitions, nebo jedna společná fronta aplikací.

Example IBM OS/360, říkal tomu multiprogramming with a fixed number of tasks (MFT). Později bylo zavedené multiprogramming with a variable number of tasks (MVT).

Klasické problémy jsou vnitřní fragmentace a umisťování aplikaci do partitions, s tím souvisí také problém relokace a ochrany dat.

Relokace se řeší buď bez podpory hardware, prostou úpravou binárního kódu aplikace, nebo s podporou hardware, pak je zpravidla k dispozici bázový registr. Bázový registr má jednu drobnou přednost, tou je relokace za běhu aplikace.

Ochrana je možná buď zavedením práv ke stránkám, nebo omezením adresového prostoru.

Example IBM 360, paměť rozdělená na 4KB bloky, každý měl 4b klíč a příznak fetch protect. Při čtení fetch protected stránky nebo při zápisu libovolné stránky musel mít program v registru PSW shodný klíč. Registr PSW bylo možné nastavit pouze v supervisor režimu.

Example CDC Cyber 6000, each application had to be allocated a single partition, starting at the address in Reference Address (RA) register, limit at the length in Field Length (FL) register.

U fixed partitions bylo navíc vidět, že pár malých aplikací může zablokovat systém na neúnosně dlouhou dobu, pokud je malý počet partitions. Aby se tomu zabránilo, zavedlo se periodické odkládání procesů na disk (swapping).

U fixed partitions se také začalo více narážet na situaci, kdy se program vůbec nevešel do fyzické paměti. Zavedlo se postupné nahrávání částí programu tak, jak byly používány (overlaying). Bohužel toto mírně zpomaluje volání procedur, mírně zatěžuje programátora a neřeší problém velkého heapu.

Variable Partitions

Protože fixed partitions mají vysokou vnitřní fragmentaci, nejsou pro swapping příliš vhodné. Zavedly se tedy variable partitions, princip je zřejmý.

Problémem variable partitions je externí fragmentace, případně také možnost změny velikosti segmentů za běhu. Fragmentace by se mohla řešit setřásáním segmentů za běhu, ale to se raději nedělá, protože to dlouho trvá a kvůli relokaci to nemusí být triviální. Example CDC Cyber 6000, mainframe ve výrobě kolem 1970, jeho feritová paměť se slovem šířky 60 bitů měla speciální hardware, který uměl přesouvat paměť rychlostí 40 MB za vteřinu. Relokace byla snadná díky adresaci bázovým registrem.

Přibývá samozřejmě také nutnost pamatovat si rozvržení variable partitions, což je problém, který se objevuje i v mnoha podobných situacích, jako je správa heapu, správa paměti v kernelu, správa swapu.

Separating Multiple Processes

Zatím všechno povídání snad s výjimkou overlaying počítalo s tím, že se do fyzické paměti vejde několik programů najednou. Situace je ale občas opačná, program se vůbec nemusí vejít. Takže se vymyslela virtuální paměť, překvapivě už někdy kolem 1961.

Page Translation

Princip stránkování je v pohodě. Paměť se rozdělí na bloky stejné délky a udělá se tabulka, která mapuje virtuální adresy na fyzické. Problém je samozřejmě v té tabulce ...

Tabulka musí být schopna pokrýt celý adresový prostor procesu, případně jich může být i víc pro víc procesů. To vede na problém s velikostí tabulky, pro adresové prostory 32 bitů a stránky o velikosti 4KB zbývá 20 bitů adresy na číslo stránky, při 4 bajtech na položku vychází tabulka kolem 4MB. To je moc, proto se dělají ...

- Víceúrovňové tabulky, kde není potřeba rezervovat prostor pro tabulku stránek celé virtuální paměti, ale jen pro použitou část, navíc mohou být části tabulky také stránkovány.
- Různé velikosti stránek, kde je potřeba menší počet položek na namapování stejného objemu paměti.
- Inverted page tables, které mají položku pro každou fyzickou stránku, a jsou tedy vlastně asociativní pamětí, která vyhledává podle virtuální adresy. Mají tu výhodu, že jejich velikost závisí na velikosti fyzické paměti, nikoliv virtuální, ovšem špatně se prohledávají. Protože inverted page tables se prohledávají zpravidla hashováním a řešit kolize v hardware by bylo nákladné, jsou vedle

samotné inverted page v paměti ještě další hashovací struktury, které používá operační systém.

Protože prohledávání tabulek při každém přístupu do paměti by bylo pomalé, vymyslel se Translation Lookaside Buffer, který je ovšem (jako každá asociativní paměť) nákladný. S TLB souvisí ještě dvě důležité věci, jedna je vyprazdňování TLB při přepnutí adresového prostoru, druhá je idea nechat správu a prohledávání stránkovacích tabulek výhradně na operačním systému a v hardware mít pouze TLB.

Page Replacement

Nahrazování stránek je pochopitelně věda, jde o to vyhodit vždycky tu správnou stránku. Zjevným kritériem je minimalizace počtu výpadků stránek za běhu aplikace, tedy optimální algoritmus by vyhodil vždycky tu stránku, která bude potřeba za nejdelší dobu. To se sice nedá předem zjistit, ale jde udělat jeden průchod programu pro změření výpadků stránek a další už s optimálním stránkováním (pokud program běží deterministicky). Tento algoritmus slouží spolu s algoritmem vybírajícím náhodnou stránku jako měřítko pro hodnocení bežných algoritmů, které se vesměs snaží vyrobit nějakou smysluplnou predikci chování programu podle locality of reference (náhodný a optimální výběr stránky představují limitní situace pro nulovou a dokonalou predikci aplikace).

[Tady vypadají moc pěkně ty grafy, co vyšly v ACM OS Review 10/97, je na nich vidět chování různých druhů aplikací při přístupu k paměti. Tedy, možná ne moc pěkně, ale na začátku by asi nebylo špatné je zmínit.]

- First In First Out replaces the page that has been replaced longest time ago.
- Not Recently Used presumes that a read access to a page sets the *accessed* bit associated with the page and that a write access to a page sets the *dirty* bit associated with the page. The operating system periodically resets the accessed bit. When a page is to be replaced, pages that are neither accessed nor dirty are replaced first, pages that are dirty but not accessed are replaced second, pages that are accessed but not dirty are replaced third, and pages that are accessed and dirty are replaced last.
- One Hand Clock is a variant of Not Recently Used that arranges pages in a circular list walked by a clock hand. The clock hand advances whenever page replacement is needed, a page that the hand points to is replaced if it is not accessed and marked as not accessed otherwise.
- Two Hand Clock is a variant of Not Recently Used that arranges pages in a circular list walked by two clock hands. Both clock hands advance whenever page replacement is needed, a page that the first hand points to is replaced if it is not accessed, the page that the second hand points to is marged as not accessed. The angle between the two hands determines the aggressivnes of the algorithm.
- Least Recently Used replaces the page that has been accessed longest time ago. Since the information on when a page has been accessed last is rarely available, approximations are used. The algorithm exhibits very inappropriate behavior when large structures are traversed.
- Least Frequently Used replaces the page that has been accessed least frequently lately. Since the information on how frequently a page has been accessed lately is rarely available, approximations are used. The algorithm exhibits very inappropriate behavior when usage frequency changes.

U algoritmů, které nezohledňují používání paměti procesem, může dojít k Beladyho anomálii, totiž v určitých situacích se přidáním frames zvýší počet výpadků stránek. Příkladem může být třeba algoritmus FIFO 012301401234 ve třech a čtyřech stránkách (je potřeba počítat už načítání prvních stránek jako výpadky).

V systému s více procesy to pak vypadá tak, že jsou namapovány nějaké množiny stránek pro každý proces. Algoritmy se dají aplikovat různým způsobem, klasické je rozdělení na lokální aplikování algoritmu v rámci jednoho procesu a globální aplikování algoritmu v rámci celého počítače. Množině stránek, které proces právě používá, se říká working set, její obsah se mění tak, jak proces běží. Ve chvíli, kdy běží příliš mnoho aplikací, se jejich working sets do paměti nevejdou. Pak se vždy spustí proces, který potřebuje naswapovat nějakou stránku, tedy se najde oběť a začne se swapovat, mezitím se spustí jiný proces, který potřebuje naswapovat nějakou stránku, a tak pořád dokola, takže se nic neudělá. Říká se tomu thrashing.

U lokálních algoritmů se dají lépe poskytovat záruky například realtime aplikacím (protože se nestatne, že jedna aplikace sebere druhé paměť), ale obecně vyhrávají spíš globální algoritmy, spolu s nějakým minimem stránek pro každý proces. Do vyhazovaných stránek se pak počítají také kernel caches. I globální algoritmy většinou fungují tak, že iterují postupně přes jednotlivé procesy, protože tím budou spíš vyhazovat nejdřív stránky z jednoho procesu a pak z dalšího, čímž zvyšují pravděpodobnost, že některé procesy budou mít v paměti celý working set.

Zmínit memory mapped files a copy on write.

References

1. Al-Zoubi et al.: Performance Evaluation of Cache Replacement Policies for the SPEC CPU2000 Benchmark Suite.
2. Sleator et al.: Amortized Efficiency of List Update and Paging Rules

Hardware Implementation

Intel IA32 Address Translation

Procesor má dvě vrstvy adresace, jedna převádí logické adresy na lineární a druhá převádí lineární adresy na fyzické. První vrstvu zatím necháme, pro stránkování je zajímavá jen ta druhá. Základní verze, logická adresa 32 bitů, fyzická adresa 32 bitů. Překlad simple, CR3 je base of page directory, prvních 10 bitů adresy offset, odtamtud base of page table, druhých 10 bitů adresy offset, odtamtud base of page, zbylých 12 bitů adresy offset.

Directory entry má krom 20 bitů base ještě 3 bity user data, jeden bit size, jeden bit accessed, jeden bit cache disabled, jeden bit write through, jeden bit user/supervisor, jeden bit read/write, jeden bit present.

Page entry má krom 20 bitů base ještě 3 bity user data, jeden bit global, jeden bit dirty, jeden bit accessed, jeden bit cache disabled, jeden bit write through, jeden bit user/supervisor, jeden bit read/write, jeden bit present.

- Pokud je nastaven bit global, mapování dané stránky se považuje za přítomné ve všech adresových prostorech a nevyhazuje se z TLB při změně CR3.
- Pokud je nastaven bit page size, directory entry neukazuje na page table, ale rovnou na stránku, která je pak velká 4 MB.
- Bity accessed a dirty nastavuje příslušným způsobem procesor, používají se pro page replacement algoritmy.
- Bity s právy a podobné jsou jasné, víc se stejně proberou na nějakém jiném předmětu.
- Když je položka not present, všechny ostatní bity jsou user defined.

No a aby se to pletlo, od Pentia Pro je ještě Physical Address Extension bit v CR4, když se nahodí tak jsou directory entry a page entry dlouhé 64 bitů, v CR3 se objeví pointer na page directory pointer table a překládá se trojúrovňově, 2 bity z adresy do pointer table, 9 bitů do directory table, 9 bitů do page table, 12 bitů do page, nebo 2 bity do pointer table, 9 bitů do directory table, 21 bitů do page. Fyzická adresa je pak 36 bitů.

Intel IA64 Address Translation

Velikosti stránek 4, 8, 16, 64 a 256 KB a 1, 4, 16 a 256 MB. Virtuální adresa 54 bitů (51 bitů adresa, 3 bity region index), fyzická adresa 44 bitů (ale aplikace vidí virtuální adresu 64 bitů, 61 bitů adresa, 3 bity region index). Region index ukazuje na jeden z osmi region registrů šířky 24 bitů, region je v podstatě address space a region index je ve virtuální adrese proto, aby procesy mohly koukat do address space jiným procesům.

Překládá se pomocí TLB, položka obsahuje krom adresy a regionu obvyklé bity present, cacheable, accessed, dirty, access rights, nic zvláštního. Pokud TLB neobsahuje překlad, hardware může prohledat ještě Virtual Hash Page Table, což je jednoduchá hash table pevného formátu. Pokud ani VHPT neobsahuje překlad, hodí se fault a operační systém naplní TLB.

Zajímavý je systém ochran. Každá položka TLB obsahuje klíč, při jejím použití se tento klíč hledá v Protection Key Registers, které obsahují klíče přidělené aktuálnímu procesu. Pokud se klíč nenajde, hodí se Key Miss Fault (pro virtualizaci PKR), pokud se najde, zkontroluje se, zda klíč nezakazuje read, write nebo execution access.

Také zajímavý je systém registrů. General purpose registers mají jména GR0 až GR31. Ty jsou jako všude jinde. Krom nich jsou ještě k dispozici registry GR32 až GR127, které fungují jako register stack. Z register stacku je část vyhrazena jako input area, část jako local area, část jako output area. Při volání procedury se z output area volajícího stane input area volaného, velikost local area a output area volaného je 0, pomocí instrukce alloc se pak dá nastavit local a output area, simple. Pokud dojde těch 96 registrů procesoru, které se interně používají jako cyklický buffer, existuje extra stack BSP, na který se ukládá, co se nevejde.

Motorola 680x0 Address Translation

Ze 32 bitů adresy je 7 bitů pointer do root table (registry URP a SRP obsahují user a supervisor root table pointer), 7 pointer do pointer table, 5 nebo 6 pointer do page table, zbyte pointer do stránky (8 nebo 4 KB, podle bitu P registru TCR). Také je k dispozici možnost vymezit čtyřmi TTR registry čtyři bloky virtuálních adres, které se nepřekládají. Jinak umí v podstatě všechno co Intel, s jednou věcí navíc, totiž directory table může krom normálního page descriptoru obsahovat ještě indirect page descriptor, který ukazuje na skutečný descriptor uložený někde jinde v paměti. To se hodí pokud se jedna stránka sdílí na více virtuálních adresách, pak totiž může stále mít jen jeden dirty bit.

[Obrázek překladu je v MC68060 User's Manual, Section 4 Memory Management Unit. Málo zajímavý je obrázek 4-1, který jen ukazuje, že se odděluje adresová a datová cache, TLB se říká ATC a mají 64 položek stupně asociace 4 každá. Obrázek 4-4 ukazuje formát translation control registru. Obrázek 4-5 ukazuje formát transparent translation registrů. Obrázek 4-7 ukazuje rozdělení virtuální adresy. Obrázky 4-10 a 4-11 ukazují formát stránkovacích tabulek, G je global, U je accessed, M je dirty, W je read only, CM je cache, PDT a UDT jsou typy položky s hlavní funkcí rozlišení present. Obrázek 4-12 ukazuje příklad překladu adresy. Obrázek 4-13 ukazuje příklad překladu adresy sdílené stránky. Obrázek 4-14 ukazuje příklad překladu adresy copy on write stránky. Obrázek 4-19 vysvětluje strukturu částečně asociativní TLB.]

[Obrázek cache je v MC68060 User's Manual, Section 5 Caches. Obrázek 5-4 vysvětluje strukturu částečně asociativní cache. Cache dovoluje čtyři režimy práce, write through cachuje čtení a zapisuje data rovnou, copy back cachuje čtení i zápis, inhibited čte i zapisuje data rovnou, ve verzi precise zaručuje pořadí přístupu shodné s pořadím instrukcí, ve verzi imprecise dovoluje některým čtením předběhnout zápisy.]

MIPS32 Address Translation

Tohle se zná z Nachosu. Na čipu je pouze TLB se 48 položkami, každá položka mapuje dvě stránky o velikosti od 4 KB do 16 MB po čtyřnásobcích. V položce je jeden address space ID (porovnává se s ASID v CP0) a jedna virtuální adresa, dvě fyzické adresy pro sudou a lichou stránku (smart protože se porovnává podle virtuální adresy), pro virtuální adresu maska (určuje velikost) a flag global (ignoruje se ASID), pro každou stránku dirty a valid flag a detaily pro řízení cache coherency, nezajímavé.

Pro naplnění položky TLB je k dispozici extra instrukce, může buď naplnit náhodnou položku nebo vybranou. Náhoda se odvozuje od počítání instrukčních cyklů, také je k dispozici wired TLB entry index registr, který říká, do kolika prvních položek TLB se náhoda nemá strefovat.

Mimochodem je to všechno dost zjednodušené, ale nevadí, podstatný je address translation mechanism a ten je popsaný přesně. Jinak existují varianty tohoto procesoru, které mají zjednodušenou MMU.

[Hezký obrázek je v MIPS32 4K Processor Core Family Software User's Manual (MIPS32-4K-Manual.pdf), Memory Management 3.3 Translation Lookaside Buffer]

Alpha Address Translation

Procesor od Compaqu, z návodu pro Alpha 21264. Virtuální adresa 48 nebo 43 bitů (podle bitu v registru I_CTL), fyzická adresa 44 bitů (nejvyšší bit je 0 pro paměť a 1 pro zařízení). TLB pro instrukce a pro data s round robin alokací, každá 128 bitů, mapují 8 KB stránky buď po jedné, nebo po skupině 8, 64 nebo 512, s 8 bity ID procesu. S TLB pracuje takzvaný PAL (Privileged Architecture Library) code, což je v podstatě privilegovaný kód blízký mikrokódu, který je uložený v normální paměti.

Zajímavě je řešené vyhazování položek z TLB. Procesor má registry ITB_IA, ITB_IS, DTB_IAP, DTB_IA a DTB_IS. Zápis do ?TB_IA vyhodí z datové nebo instrukční TLB všechny položky, DTB_IAP vyhodí všechny položky daného procesu, ?TB_IS vyhodí všechny položky týkající se zapisované adresy.

Tenhle hrůzný procesor má dokonce i virtuální registry. Běžné registry se jmenují R0 až R31, ale když je programátor používá, přemapují se na interní registry procesoru tak, aby se minimalizoval počet falešných write-after-read a writer-after-read závislostí mezi instrukcemi v pipeline.

[Zdá se, že žádné pěkné obrázky nejsou.]

UltraSparc Address Translation

Je velmi podobná MIPS procesorům, s délkami stránek 8, 64, 512 a 4096 KB, virtuální adresa 44 bitů (ale rozdělená na dvě poloviny na začátku a konci prostoru 64 bitů), fyzická adresa 41 bitů. Opět je k dispozici kontext určující kterému procesu patří položka TLB, bit na jeho ignorování u globálních mapování, page size ve dvou bitech, z dalších je třeba bit indikující endianness dat uložených na dané stránce (jmenuje se IE od Invert Endianness, další údaj o endianness je v address space ID, další v instrukci).

Aby měl TLB miss handler jednodušší život, nabízí hardware ještě další podporu. Při TLB miss se z registrů MMU dá vyčíst adresa do translation table v paměti, kde by podle jednoduchých hash rules měla být potřebná položka TLB. Pokud tam je, MMU ji umí na pokyn od handleru načíst do TLB.

[Obrázky UltraSparc 2 User's Manual, Chapter 15 MMU Internal Architecture. Obrázek 15-1 ukazuje formát položky TLB, CONTEXT je address space ID, V je valid, NFO je cosi, IE je invert endianness, L je lock entry, P je privileged, W je read only, G je global. Obrázek 15-2 ukazuje formát translation table v paměti, split říká jestli se budou 8k a 64k stránky hashovat společně nebo ne.]

ARM Address Translation

Řada procesorů od ARM, z návodu pro ARM10E. Instrukční a datová TLB, každá 64 položek, tabulka stránek podporovaná hardware MMU. Umí stránky o velikosti 1, 4, 16, 64 a 1024 kB, pro zmatení nepřítele jim říká tiny pages, small pages, large pages a sections, některé umí dělit do čtyř subpages. Tabulka stránek je dvouúrovňová až na velikosti stránek 1024 kB. Ochrany jsou řešeny zavedením 16 domén, v 16 registrech jsou popsána práva supervisor a user procesů k doméně, každá stránka patří do nějaké domény. Je také možné používat pouze TLB.

[Obrázky ARM 1022E Technical Reference Manual, Chapter 4 Memory Management Units. Obrázek 4-1 ukazuje překlad adresy. Obrázek 4-3 ukazuje formát položky stránkovací tabulky úrovně 1. Obrázek 4-5 ukazuje formát položky stránkovací tabulky úrovně 2. C je cacheable, B je write back bufferable, AP je cosi co rozlišuje subpages, SBZ should be zero :). Nejsou bity accessed a dirty, nepředpokládá se paging.]

Další zajímavé vlastnosti procesoru. V kódu všech instrukcí je možné uvést podmínku, kdy se má provést, což odstraňuje nutnost branch prediction a nebezpečí prediction misses pro malé větve kódu.

Software Implementation

To be done.

Krom obvyklých problémů s přístupem ke sdíleným strukturám mají víceprocesorové systémy problémy ještě v situacích, kdy jednotlivé procesory cachují informace související s memory managementem. Dvě situace:

- Mapování v TLB. Pokud se změní address space mapppings, na uniprocesoru se flushuje TLB. Na multiprocesorech je potřeba flushnout TLB na všech procesorech, z toho vyplývá nutnost synchronizace při změně mapování, a to je pomalé. Trikem se to dá řešit třeba u R4000, kde se procesu prostě posune ASID, čímž se invalidují všechny jeho položky v TLB.
- Virtual address caches. Většina caches sice používá fyzické adresy, ale protože hardware s caches na virtuální adresy může běžet rychleji, občas se také objeví. Tam je pak stejný problém jako u TLB.

Example: Linux

HAL jako rozhraní, které zpřístupňuje memory manager dané platformy, zbytek kernelu předpokládá trojúrovňové stránkování. [Linux 2.4.20 například /include/asm-i386/pgtable-2level.h a /include/asm-i386/pgtable-3level.h] [Linux 2.6.9 například pgtable-2level.h a pgtable-2level-defs.h a pgtable-3level.h a pgtable-3level-defs.h a pgtable.h v /include/asm-i386] Neznamená to, že by se nějak simulovaly 3 úrovně kernelu pro 2 úrovně procesoru, prostě se v makrech řekne, že ve druhé úrovni je jen 1 položka.

Fyzické stránky se evidují strukturami struct page {} mem_map_t v seznamu mem_map, jako algoritmus pro výběr oběti se zřejmě používá LRU, bez bližších detailů, protože kernel vypadá všelijak. [Linux 2.4.20 /include/linux/mm.h]

Fyzické stránky jsou přiřazeny do zón, které odrážejí omezení pro některé rozsahy fyzické paměti, například ZONE_DMA, ZONE_NORMAL, ZONE_HIGHMEM. Zóny mají seznamy volných stránek per CPU, aby nedocházelo ke kolizím na multiprocesorech. V každé zóně jede něco, čemu autoři říkají LRU, kód pro zájemce hlavně v [Linux 2.6.9 /mm/vmscan.c]. [Linux 2.6.9 /include/linux/mmzone.h]

Pro každý proces se pamatuje mapa jeho adresového prostoru ve struktuře mm_struct [Linux 2.4.22 /include/linux/sched.h], které je seznamem struktur vm_area_struct [Linux 2.4.22 /include/linux/mm.h]. Každá area má začátek, délku, flagy (code, data, shared, locked, growing ...) a případně associated file. Potřebné areas se vytvoří při startu procesu, uživatel pak může volat už jen pár syscalls jako mmap pro memory mapped files (dnes již v podstatě běžná záležitost) a shared memory (rovněž nic nového pod sluncem) nebo brk pro nastavení konce heapu. Nic moc. [Linux 2.4.20] [Linux 2.6.9]

[Mel Gorman: Understanding The Linux Virtual Memory Manager]

Example: Solaris

Klasické rozdělení na HAL, protože je potřeba nezávislost na platformě, pak správce segmetů a správce adresových prostorů. Mapa paměti klasicky kód, heap, stack. Stack roste on demand.

Každý segment má svého správce, v podstatě virtuální metody pro typy segmentů, hlavní je seg_vn driver pro vnodes souborů, seg_kmem pro nestránkovatelnou paměť kernelu, seg_map pro vnodes cache. Každý správce umí advise (jak se bude přistupovat k segmentu), checkprot (zkontrolování ochrany), fault (handle page fault na dané adrese), lockop (zamknutí a odemknutí stránek), swapout (žádost o uvolnění co nejvíce stránek), sync (žádost o uložení dirty stránek) a samozřejmě balík dalších.

Správce seg_vn umí mapovat soubory jako shared nebo private. Při private mapování se používá copy on write, který při zápisu přemapuje stránku do anonymní paměti. Jako drobné rozhodnutí, když je dost paměti, vyhradí se nové stránka a nakopírují se data, když ne, použije se sdílená stránka, která tím pádem přestane být sdílená.

Anonymní paměť je zajímavá, při prvním použití je automaticky zero filled, což je důležité. Spravuje jí swapfs layer, který ale nefunguje přímočaře tak, že by si pro každou stránku pamatoval pozici ve swap partition. Kdyby to tak totiž bylo, spotřebovával by se swap ještě než by se vůbec začalo stránkovat, takže místo toho je ke každé anonymní stránce struktura anon_map, která si v případě vyswapování zapamatuje pozici na disku. Prostor swapu je rezervován, ale nikoliv alokován, už v okamžiku alokace stránky, což dovoluje synchronní hlášení out of memory (do dostupného swapu se počítá i nezamčená fyzické paměť). Prý například AIX tohle nemá a out of memory se hlásí signálem.

Velmi zajímavá je také integrace správce souborů se správcem paměti. Jednak kvůli paměťově mapovaným souborů, ale hlavně kvůli cache. Aby se zabránilo sémantickým chybám při přístupu k souborům současně pomocí read a write a pomocí mmap, implementuje se read a write interně jako mmap do kernel bufferu. Když se takový buffer uvolní, stránka se sice eviduje jako volná, ale zůstane informace o tom, který vnode a offset obsahovala, takže až do té doby, než ji někdo použije, je součástí cache a dá se znovu použít. Kvůli tomu se udržuje hash stránek podle vnode a offsetu.

Page reclaim algoritmus je hodinový se dvěmi ručičkami, spouští se tím víc, čím víc dochází paměť, vzdálenost a rychlost obíhání se nastavuje v konfiguraci kernelu při bootu. Zajímavý efekt nastal, když se integroval správce souborů a správce paměti a začaly být rychlé disky, totiž když se zaplní cache a hledá se oběť pro vyhození, ručičky začnou obíhat příliš rychle (při dnešních discích desítky, v případě diskových polí stovky MB za vteřinu, což znamená, že se i bity o přístupu nulují v řádech vteřin) a tedy začnou příliš agresivně vyhazovat stránky programů, protože ty je prostě za tak krátkou dobu nestihnou použít. Solaris 7 tak upřednostňuje cache před stránkami programů, dokud mu nezačne docházet paměť, Solaris 8 už dělá něco úplně jiného, o čem nemám informace.

Jako jiná zajímavá funkce existují watchpoints, možnost dostat signál při přístupu na konkrétní adresu, ovládá se přes /proc file systém, tady jen pro zajímavost.

Detaily. Na thrashing se reaguje vyswapováním celého procesu. Dělá se page coloring kvůli caches. Shared pages se nevyhazují dokud to není nutné. Kernel allocator dělá slaby, což jsou bloky dané délky, umí reuse už inicializovaných objektů, další info skipped.

[Mauro, McDougall: Solaris Internals, ISBN 0-13-022496-0]

Example: Mach And Spring

O lepší memory manager se pokusil například Mach, do relativně dokonalé podoby byl celý mechanismus dotažen ve Springu. Dá se dobře odpřednášet podle technical reportu z Evry.

Interně má velmi podobnou strukturu jako Spring také Unix SVR4, ale tam není přístupná uživateli. Memory objektům se říká segments, pagery jsou reprezentované pomocí vnodes pokud přistupují k objektům file systému, krom nich existuje ještě anonymous pager pro memory objekty, které přímo neodpovídají žádnému objektu file systému.

Example: Cluster Memory Management

It can be observed that reading a page from a disk is not necessarily faster than reading a page from a network. It can also be observed that physical memory of a system is rarely used in its entirety except for caches. These two observations give rise to the idea of using spare physical memory of a cluster of systems instead of a disk for paging.

A prototype of a cluster memory management has been implemented for OSF/1 running on DEC Alphas connected by ATM. The prototype classifies pages on a system node as local or global depending on whether they are accessed by this node or cached for another node. The page fault algorithm of the prototype distinguishes two major situations:

- The faulted page for node X is cached as global on another node Y. (The page can be fetched from network, a space for it has to be made on X.) The faulted page from Y is exchanged for any global page on X. If X has no global page, a LRU local page is used instead.
- The faulted page for node X is not cached as global on any node. (The page must be fetched from disk, a space for it has to be made in cluster and on X.) A cluster wide LRU page on another node Y is written to disk. Any global page on X is written to Y. If X has no global page, a LRU local page is used instead. The faulted page is read from disk, where all pages are stored as a backup in case a node with global pages becomes unreachable.

To locate a page in the cluster, the prototype uses a distributed hash table. For each page in the cluster, the table contains the location of the page. Each node in the cluster manages part of the table.

To locate a cluster wide LRU page, the prototype uses a probabilistic LRU algorithm. The lifetime of the cluster is divided into epochs with a maximum epoch duration and a maximum eviction count. Each epoch, a coordinator is chosen as the node with most idle pages from the last epoch. The coordinator collects summary of page ages from each node in the cluster and determines the percentage of oldest pages within the maximum eviction count on each node in the cluster. The coordinator distributes this percentage to each node in the cluster and appoints a coordinator for the next epoch. Each eviction, a node is chosen randomly with the density function using the distributed percentages, the node then evicts LRU local page.

[Michael J. Feeley, William E. Morgan, Frederic H. Pighin, Anna R. Karlin, Henry M. Levy, Chandramohan A. Thekkath: Implementing Global Memory Management in a Workstation Cluster]

What Is The Interface

To be done.

Rehearsal

Questions

1. Internal fragmentation leads to poor utilization of memory that is marked as used by the operating system. Explain how internal fragmentation occurs and how it can be dealt with.
2. External fragmentation leads to poor utilization of memory that is marked as free by the operating system. Explain how external fragmentation occurs and how it can be dealt with.
3. Explain how memory virtualization through paging works and what kind of hardware and operating system support it requires.
4. At the level of individual address bits and entry flags, describe the process of obtaining physical address from virtual address on a processor that only has the Translation Lookaside Buffer. Explain the role of the operating system in this process.

    Hint: Do not concentrate on the addresses alone. The address translation process also reads and writes some flags in the entries of the Translation Lookaside Buffer.

    A thorough answer should also explain the relationship between the widths of the address fields and the sizes of the address translation structures.

5. At the level of individual address bits and entry flags, describe the process of obtaining physical address from virtual address on a processor that supports multilevel page tables. Explain the role of an operating system in this process.

    Hint: Do not concentrate on the addresses alone. The address translation process also reads and writes some flags in the entries of the multilevel page tables.

    A thorough answer should also explain the relationship between the widths of the address fields and the sizes of the address translation structures.

6. Explain the relation between the page size and the size of the information describing the mapping of virtual addresses to physical memory and list the advantages and disadvantages associated with using smaller and larger page sizes.

7. List the advantages and disadvantages of using multilevel page table as a data structure for storing the mapping of virtual to physical memory.
8. List the advantages and disadvantages of inverse page table as a data structure for storing the mapping of virtual to physical memory.
9. Explain what a Translation Lookaside Buffer is used for.
10. Describe the hardware realization of a Translation Lookaside Buffer and explain the principle and advantages of limited associativity.
11. How does the switching of process context influence the contents of the Translation Lookaside Buffer ? Describe ways to minimize the influence.
12. Provide at least two criteria that can be used to evaluate the performance of a page replacement algorithm.
13. Explain the principle of the First In First Out page replacement algorithm and evaluate the feasibility of its implementation on contemporary hardware along with its advantages and disadvantages.
14. Explain the principle of the Not Recently Used page replacement algorithm and evaluate the feasibility of its implementation on contemporary hardware along with its advantages and disadvantages.
15. Explain the principle of the Least Recently Used page replacement algorithm and evaluate the feasibility of its implementation on contemporary hardware along with its advantages and disadvantages.
16. Explain the principle of the Least Frequently Used page replacement algorithm and evaluate the feasibility of its implementation on contemporary hardware along with its advantages and disadvantages.
17. Explain what is a working set of a process.
18. Explain what is a locality of reference and how it can be exploited to design or enhance a page replacement algorithm.
19. Explain what is thrashing and what causes it. Describe what can the operating system do to prevent thrashing and how can the system detect it.
20. Explain the concept of memory mapped files.
21. Explain the priciple of the copy-on-write mechanism and provide an example of its application in an operating system.

Exercises

1. Consider a system using 32 bit virtual and 32 bit physical addresses. Choose and describe a way the processor in this system should translate virtual addresses to physical. Choose a page size and explain what kind of operation is the page size well suited for.

   Design a data structure for mapping virtual addresses to physical and describe in detail all the records used in the structure. When designing the data structure, take into account the choice of the address translation mechanism and explain why is the resulting data structure well suited for the system.

   Describe the involvement of the operating system in the process of translating a virtual address to physical (if any), and provide a sketch of the algorithms for handling a page fault and selecting a page for eviction.

   Hint: Among other things, approaches to address translation differ in the range of exceptions that the operating system must handle. These might include access protection exception, address translation exception, page fault exception. Does your description of the operating system involvement cover all the exceptions applicable in the chosen approach to address translation ?

Besides the addresses, the data structure for address mapping typically contains many other fields that control access protection or help page replacement. Is your description of the data structure detailed enough to include these fields ?

2. Consider the previous example except the system is using 32 bit virtual and 36 bit physical addresses.
3. Consider the previous example except the system is using 54 bit virtual and 44 bit physical addresses.

Allocation Within A Process

Process Memory Layout

A typical process runs within its own virtual address space, which is distinct from the virtual address spaces of other processes. The virtual address space typically contains four distinct types of content:

- Executable code. This part of the virtual address space contains the machine code instructions to be executed by the processor. It is often write protected and shared among processes that use the same main program or the same shared libraries.
- Static data. This part of the virtual address space contains the statically allocated variables to be used by the process.
- Heap. This part of the virtual address space contains the dynamically allocated variables to be used by the process.
- Stack. This part of the virtual address space contains the stack to be used by the process for storing items such as return addresses, procedure arguments, temporarily saved registers or locally allocated variables.

Each distinct type of content typically occupies one or several continuous blocks of memory within the virtual address space. The initial placement of these blocks is managed by the loader of the operating system, the content of these blocks is managed by the process owning them.

The blocks that contain executable code and static data are of little interest from the process memory management point of view as their layout is determined by the compiler and does not change during process execution. The blocks that contain stack and heap, however, change during process execution and merit further attention.

While the blocks containing the executable code and static data are fixed in size, the blocks containing the heap and the stack may need to grow as the process owning them executes. The need for growth is difficult to predict during the initial placement of the blocks. To avoid restricting the growth by placing either heap or stack too close to other blocks, they are typically placed near the opposite ends of the process virtual address space with an empty space between them. The heap block is then grown upwards and the stack block downwards as necessary.

When multiple blocks of memory within the virtual address space need to grow as the process owning them executes, the initial placement of the blocks becomes a problem. This can be partially alleviated by using hardware that supports large virtual addresses, where enough empty space can be set aside between the blocks without exhausting the virtual address space, or by using hardware that supports segmentation, where blocks can be moved in the virtual address space as necessary.

Example: Virtual Address Space Of A Linux Process

In Linux, the location of blocks of memory within the virtual address space of a process is exported by the virtual memory manager of the operating system in the `maps` file of the `proc` filesystem.

```
> cat /proc/self/maps
00111000-00234000 r-xp 00000000 03:01 3653725    /lib/libc-2.3.5.so
00234000-00236000 r-xp 00123000 03:01 3653725    /lib/libc-2.3.5.so
00236000-00238000 rwxp 00125000 03:01 3653725    /lib/libc-2.3.5.so
00238000-0023a000 rwxp 00238000 00:00 0
007b5000-007cf000 r-xp 00000000 03:01 3653658    /lib/ld-2.3.5.so
007cf000-007d0000 r-xp 00019000 03:01 3653658    /lib/ld-2.3.5.so
007d0000-007d1000 rwxp 0001a000 03:01 3653658    /lib/ld-2.3.5.so
008ed000-008ee000 r-xp 008ed000 00:00 0          [vdso]
08048000-0804d000 r-xp 00000000 03:01 3473470    /bin/cat
0804d000-0804e000 rw-p 00004000 03:01 3473470    /bin/cat
09ab8000-09ad9000 rw-p 09ab8000 00:00 0          [heap]
b7d88000-b7f88000 r--p 00000000 03:01 6750409    /usr/lib/locale/locale-archive
b7f88000-b7f89000 rw-p b7f88000 00:00 0
b7f96000-b7f97000 rw-p b7f96000 00:00 0
bfd81000-bfd97000 rw-p bfd81000 00:00 0          [stack]
```

The example shows the location of blocks of memory within the virtual address space of the `cat` command. The first column of the example shows the address of the blocks, the second column shows the flags, the third, fourth, fifth and sixth columns show the offset, device, inode and name of the file that is mapped into the block, if any. The blocks that contain executable code are easily distinguished by the executable flag. Similarly, the blocks that contain read-only and read-write static data are easily distinguished by the readable and writeable flags and the file that is mapped into the block. Finally, the blocks with the readable and writeable flags but no file contain the heap and the stack.

The address of the blocks is often randomized to prevent buffer overflow attacks on the process. The attacks are carried out by supplying the process with an input that will cause the process to write past the end of the buffer allocated for the input. When the buffer is a locally allocated variable, it resides on the stack and being able to write past the end of the buffer means being able to modify return addresses that also reside on the stack. The attack can therefore overwrite some of the input buffers with malicious machine code instructions to be executed and overwrite some of the return addresses to point to the malicious machine code instructions. The process will then unwittingly execute the malicious machine code instructions by returning to the modified return address. Randomizing the addresses of the blocks makes this attack more difficult.

Stack

The process stack is typically used for return addresses, procedure arguments, temporarily saved registers and locally allocated variables. The processor typically contains a register that points to the top of the stack. This register is called the *stack pointer* and is implicitly used by machine code instructions that call a procedure, return from a procedure, store a data item on the stack and fetch a data item from the stack.

The use of stack for procedure arguments and locally allocated variables relies on the fact that the arguments and the variables reside in a constant position relative to the top of the stack. The processor typically allows addressing data relative to the top of the stack, making it possible to use the same machine code instructions to access the procedure arguments and the locally allocated variables regardless on their absolute addresses in the virtual address space, as long as their addresses relative to the top of the stack do not change.

Allocating the block that contains stack requires estimating the stack size. Typically, the block is allocated with a reasonable default size and an extra page protected against reading and writing is added below the end of the allocated block. Should the stack overflow, an attempt to access the protected page will be made, causing an exception. The operating system can handle the exception by growing the block that contains stack and retrying the machine code instruction that caused the exception.

A multithreaded program requires as many stacks as there are threads. This makes placing the block that contains stack more difficult with respect to growing the block later, unless segmentation or split stack is used.

Example: Stack On Intel 80x86 Processors

The Intel 80x86 processors have a stack pointer register called `ESP` . The CALL machine code instruction decrements the `ESP` register by the size of a return address and stores the address of the immediately following machine code instruction to the address pointed to by the `ESP` register. Symetrically, the RET machine code instruction fetches the stored return address from the address pointed to by the `ESP` register and increments the `ESP` register by the size of a return address. The PUSH and POP machine code instructions can be used to store and fetch an arbitrary register to and from the stack in a similar manner.

Note that the stack grows towards numerically smaller addresses. This simplifies the process memory management when only one stack block is present, as it can be placed at the very end of the virtual address space rather than in the middle of the virtual address space, where it can collide with other blocks that change during process execution.

To address the stack content, the Intel 80x86 processors have a base pointer register called `EBP` . The `EBP` register is typically set to the value of the `ESP` register at the beginning of a procedure, and used to address the procedure arguments and locally allocated variables throughout the procedure. Thus, the arguments are located at positive offsets from the `EBP` register, while the variables are located at negative offsets from the `EBP` register.

```
void SomeProcedure (int anArgument)
{
  int aVariable;
  aVariable = anArgument;
}

SomeProcedure:

    push    ebp              ;save original value of EBP on stack
    mov     ebp,esp          ;store top of stack address in EBP
    sub     esp,4            ;allocate space for aVariable on stack

    mov     eax,[ebp+8]      ;fetch anArgument into EAX, which is
                             ;8 bytes below the stored top of stack
    mov     [ebp-4],eax      ;store EAX into aVariable, which is
                             ;4 bytes above the stored top of stack

    mov     esp,ebp          ;free space allocated for aVariable
    pop     ebp              ;restore original value of EBP
    ret                      ;return to the caller
```

In the example, the stack at the entry to `SomeProcedure` contains the return address on top, that is 0 bytes above the value of `ESP` , and the value of `anArgument` one item below the top, that is 4 bytes above the value of `ESP` . Saving the original value of `EBP` stores another 4 bytes to the top of the stack and therefore decrements the value of `ESP` by another 4 bytes, this value is then stored in `EBP` . During the execution of `SomeProcedure`, the value of `anArgument` is therefore 8 bytes above the value of `EBP` . Note that the machine code instructions used to access the procedure arguments and the locally allocated variables do not use absolute addresses in the virtual address space of the process.

Heap

The process heap is used for dynamically allocated variables. The heap is stored in one or several continuous blocks of memory within the virtual address space. These blocks include a data structure used to keep track of what parts of the blocks are used and what parts of the blocks are free. This data structure is managed by the *heap allocator* of the process.

In a sense, the heap allocator duplicates the function of the virtual memory manager, for they are both responsible for keeping track of blocks of memory. Typically, however, the blocks managed by the heap allocator are many, small, short-lived and aligned on cache boundaries, while the blocks managed by the virtual memory manager are few, large, long-lived and aligned on page boundaries. This distinction makes it possible to design the heap allocator so that it is better suited for managing blocks containing dynamically allocated variables than the virtual memory manager. Usually, the heap allocator resides within a shared library used by the processes of the operating system. The kernel of the operating system has a separate heap allocator.

Heap Allocators

Obvyklými požadavky na alokátor jsou rychlost (schopnost rychle alokovat a uvolnit paměť), úspornost (malá režie dat alokátoru a malá fragmentace) funkčnost (resizing, align, zero fill).

Alokátory evidují volnou a obsazenou paměť zpravidla buď pomocí seznamů nebo pomocí bitmap. Bitmapy mají dobrou efektivitu při alokaci bloků velikosti blízké jejich granularitě, nevýhodou je interní fragmentace, taky se v nich blbě hledá volný blok požadované délky. U linked lists asi taky není co dodat, režie na seznam, externí fragmentace, sekvenční hledání, oddělené seznamy plných a prázdných bloků, zvláštní seznamy bloků obvyklých velikostí aka zones, scelování volných bloků.

Při alokaci nového bloku je možné použít několik strategií. Nejjednodušší je first fit, případně modifikace next fit. Dalším je best fit, který ovšem vytváří malé volné bloky. Zkusil se tedy ještě worst fit, který také nebyl nic extra. Udržování zvláštních seznamů častých velikostí se někdy nazývá quick fit. Sem asi patří i buddy system, to jest dělení partitions na poloviční úseky u seznamů bloků obvyklých velikostí, problém s režií bloků délek přesně mocnin dvou.

Statistiky overheadu pro konkrétní aplikace uvádějí 4% pro best fist, 7% pro first fit na FIFO seznamu volných bloků, 50% pro first fit na LIFO seznamu volných bloků, 60% pro buddy system.

[M. S. Johnstone & P. R. Wilson: The Memory Fragmentation Problem Solved, ACM SIGPLAN 34/3, 3/1999]

Podívat se na [P. R. Wilson & M. S. Johnstone & M. Neely & D. Boles: Dynamic Storage Allocation A Survey And Critical Review, International Workshop on Memory Management, September 1995, ftp://ftp.cs.utexas.edu/pub/garbage/allocsrv.ps]

Buddy system. Výhodou buddy systému má být zejména to, že se při uvolňování bloku dá snadno najít kandidát na spojení do většího volného bloku. Nevýhodou je potenciálně vysoká interní fragmentace, daná pevnou sadou délek bloku.

Implementace buddy systému potřebuje někde uschovávat informace o blocích a seznamy volných bloků. To se dá dělat například v hlavičkách u samotných bloků, čímž jsou vlastní bloky o něco menší, ale v hlavičce není potřeba příliš mnoho informací. Alternativně se vedle alokované paměti umístí bitmapa s jedním bitem pro každý blok a každou úroveň buddy systému.

Mimochodem, když už jsme u toho, multiprocesorové systémy mají u alokátorů podobné problémy jako plánovače nad ready frontou, tedy příliš mnoho souběhů je zpomaluje. Proto se dělají hierarchické alokátory, local free block pools, které se v případě potřeby přelévají do global free block poolu.

Example: dlmalloc Heap Allocator

dlmalloc is a general purpose allocator written and maintained by Doug Lea since 1992. It was used in early versions of GNU LibC and other allocators, such as ptmalloc or nedmalloc, are derivations. The allocator is suitable for workloads with block sizes from tens of bytes and few concurrent allocations. The current description is for version 2.8.6 from August 2012.

Internally, the allocator distinguishes three ranges of block sizes. [1]

- For blocks of less than 256 bytes, the allocator keeps an array of linked lists called *bins*. Each bin contains free blocks of one particular size, from minimum block size of 16 bytes to 256 bytes in steps of 8. Requests for blocks in this range are satisfied from the bins using exact fit. If exact fit is not available, the last best fit block that was split is used.
- For blocks between 256 bytes and 256 kilobytes, the allocator keeps an array of bins for ranges of sizes in powers of two. Each bin is a binary trie whose elements are lists of blocks of equal size. In earlier versions, each bin was a sorted list of blocks. Best fit is used for allocation from bins.
- Finally, `mmap` is used to allocate blocks larger than 256 kilobytes if free block is not available.

Both free and allocated blocks, called *chunks*, have headers and footers. Free blocks also carry trie or list references where allocated blocks store user data. This simplifies management of allocator structures as well as block splitting and coalescing operations.

```
chunk +------------------------------------------------+
      | Size of previous chunk (if P = 0)              |
      +--------------------------------------------+---+
      +--------------------------------------+---+ | P |
      | Size of this chunk                   | 1 | +---+
data  +--------------------------------------+---+-----+
      |                                                |
      : User data (size - sizeof (size_t) bytes)       :
      |                                                |
chunk +------------------------------------------------+
      | Size of this chunk again                       |
      +--------------------------------------------+---+
      +--------------------------------------+---+ | 1 |
      | Size of next chunk                   | U | +---+
data  +--------------------------------------+---+-----+

// Adjusted from dlmalloc source code comments.
```

Figure 3-1. Allocated Chunk Structure

```
chunk +------------------------------------------------+
      | Size of previous chunk (if P = 0)              |
      +--------------------------------------------+---+
      +--------------------------------------+---+ | P |
      | Size of this chunk                   | 0 | +---+
      +--------------------------------------+---+-----+
      | Pointer to next chunk in bin list              |
      +------------------------------------------------+
      | Pointer to previous chunk in bin list         |
      +-----------------------------------------------+
      | Pointer to left child when in bin trie        |
      +-----------------------------------------------+
      | Pointer to right child when in bin trie       |
      +-----------------------------------------------+
      | Pointer to parent when in bin trie            |
      +-----------------------------------------------+
      | Bin index when in bin trie                    |
      +-----------------------------------------------+
      |                                               |
      : Free                                          :
      |                                               |
chunk  +-----------------------------------------------+
      | Size of this chunk again                      |
      +-------------------------------------------+---+
      +-----------------------------------+---+ | 1 |
      | Size of next chunk                | U | +---+
data   +-----------------------------------+---+-----+

// Adjusted from dlmalloc source code comments.
```

Figure 3-2. Free Chunk Structure

There is no support for efficient handling of concurrent allocations. The allocator uses a single big lock to protect its structures.

References

1. Doug Lea: A Memory Allocator. http://gee.cs.oswego.edu/dl/html/malloc.html

ptmalloc Heap Allocator

ptmalloc is a general purpose allocator written and maintained by Wolfram Gloger since 1999. It is derived from dlmalloc and used in late versions of GNU LibC. The current description is for version 2 from May 2006.

As far as the core allocation structures and algorithms go, ptmalloc is much the same as dlmalloc. Some minor differences are:

- The allocator uses sorted lists instead of binary tries in bins for large blocks.
- The allocator uses a special array of *fast bins* for blocks of up to 80 bytes. When freed, these blocks are kept marked as used and put into the appropriate fast bin. Allocation is satisfied using exact fit from fast bins when possible. Fast bins are emptied under multiple heuristic conditions including when an exact fit allocation from any fast bin fails, when a large block is allocated or freed, when fast bins contain too many blocks or when the allocator needs more memory.
- Even blocks that do not fit into fast bins are not returned to standard bins immediately. Instead, they are kept marked as used and put into a special unsorted list. On allocation, this list is traversed and if an exact fit is found, the block is reused. All blocks that are not an exact fit are put into bins during the traversal. To avoid anomalies, at most 10000 blocks are examined in one allocation.

The big difference between ptmalloc and dlmalloc is in support for concurrent allocations. The ptmalloc allocator maintains multiple *arenas*. Each arena is an independent instance of the allocator protected by a single big lock. On allocation, the thread invoking the allocator attempts to lock an arena, starting with the one it used last. If an arena is locked successfully, it is used to satisfy the allocation request, otherwise a new arena is allocated.

To find an arena given a block pointer, simple pointer arithmetic is used. All arenas except one start at addresses that are multiples of maximum arena size, one megabyte by default. Masking the block pointer bits that address within the arena therefore returns the arena start pointer. One arena is designated as *main arena* and not subject to the size limit. Main arena blocks are identified by header bits.

References

1. Wolfram Gloger: A Memory Allocator. http://www.malloc.de

Example: tcmalloc Heap Allocator

tcmalloc is a general purpose allocator written and maintained by Sanjay Ghemawat since 2004. The current description is for version 2.1 from July 2013.

The allocator fetches memory from the system in *spans*, which are continous sequences of pages. A span is either free or assigned to a single large block or assigned to small blocks of the same size class. The allocator maintains a tree to convert a block pointer page to a span reference, augmented with a mapping cache and stored in system allocated memory blocks. In this arrangement, small blocks do not need individual headers.

Free memory in spans is referenced by free lists from either thread local caches or central heap. The free lists themselves are LIFO single linked lists stored within the free blocks. The allocator behavior differs based on block size:

- Small blocks are blocks with size below 32 kilobytes. Each cache has free lists pointing to free blocks of given size. There are about 60 size classes spaced from minimum size to 32 kilobytes. Spacing starts at 8 bytes for smallest classes and ends at 256 bytes for largest classes. Central heap has free lists of the same structure.

  Allocation goes into the local free lists. Since there are not block headers, size is always rounded up to nearest class. If the local exact fit fails, an adaptive number of free blocks is pulled from the central heap. If even the central heap exact fit fails, a new span is allocated and split into free blocks of same size.

- Large blocks are blocks with size above 32 kilobytes. These are allocated only on central heap, which has 256 free span lists for sizes from 32 kilobytes in steps of 4 kilobytes, last for all larger sizes.

  Allocation uses best fit on the free lists. If the best fit fails, a new span is allocated.

When freeing a block, the allocator first identifies the span. Small blocks are returned to the thread cache of current thread. If the number of free blocks exceeds an adaptive threshold, the cache is shrunk. Large blocks are merged with free neighbors and returned to the span list of the central heap.

References

1. Sanjay Ghemawat: TCMalloc: Thread-Caching MAlloc. http://github.com/gperftools/gperftools/blob/master/docs/tcmalloc.html

Example: Hoard Allocator

To be done.

Example: Posix Heap Allocator Interface

To be done.

Example: Linux Kernel Slab Allocator

To be done.

```
// Create a slab cache
kmem_cache_t * kmem_cache_create (
  const char *name,
  size_t size,
  size_t align,
  unsigned long flags,
  void (* ctor) (void *, kmem_cache_t *, unsigned long),
  void (* dtor) (void *, kmem_cache_t *, unsigned long));

// Allocate and free slabs of the cache
void *kmem_cache_alloc (kmem_cache_t *cachep, int flags);
void kmem_cache_free (kmem_cache_t *cachep, void *objp);
```

Two implementations of the allocator exist in the kernel, called SLAB and SLUB. The allocators differ in the way they keep track of slabs and objects, SLAB being more complex, SLUB being more streamlined. Usage statistics is available with both allocators.

```
> cat /proc/slabinfo
slabinfo - version: 2.1
# name            <active_objs> <num_objs> <objsize> <objperslab> <pagesperslab> ...
ext4_inode_cache         49392      49392      1016           16              4
ext4_free_data             128        128        64           64              1
ext4_xattr                  92         92        88           46              1
blkdev_requests            273        273       376           21              2
blkdev_queue                30         30      2080           15              8
vm_area_struct            3520       3520       184           22              1
task_struct                160        160      1952           16              8
inode_cache              11899      11928       568           14              2
dentry                  401373     401373       192           21              1
...
```

[This information is current for kernel 2.6.23.]

References

1. Jeff Bonwick: The Slab Allocator: An Object-Caching Kernel Memory Allocator.

Garbage Collectors

A traditional interface of a heap allocator offers methods for explicit allocating and freeing of blocks on the heap. Explict allocating and freeing, however, is prone to memory leaks when a process fails to free an allocated block even though it no longer uses it. A *garbage collector* replaces the explicit freeing of blocks with an automatic freeing of blocks that are no longer used.

A garbage collector needs to recognize when a block on the heap is no longer used. A garbage collectors determines whether a block is no longer used by determining whether it is *reachable*, that is, whether a process can follow a chain of references from statically or locally allocated variables, called *roots*, to reach the block.

Note that there is a difference between blocks that are no longer used and blocks that can no longer be used. This difference means that a garbage collector will fail to free blocks that can be used but are no longer used. In other words, a garbage collector exchanges the burden of having to explicitly free dynamically allocated variables for the burden of having to discard references to unused dynamically allocated variables. Normally, this is a benefit, because while freeing variables is always explicit, discarding references is often implicit.

Reference Tracing

Reference tracing algorithms. Copying. Mark and sweep. Mark and compact.

Reference Counting

Reference counting algorithms. Cycles. Distribution.

Distinguishing Generations

It has been observed that objects differ in lifetime. Especially, many young objects quickly die, while some old objects never die. Separating objects into generations therefore makes it possible to collect a generation at a time, especially, to frequently collect the younger generation using a copying collector and to seldomly collect the older generation using a mark and sweep collector. Collecting a generation at a time requires keeping remembered sets of references from other generations. Typically, all generations below certain age are collected, therefore only references from older to younger generations need to be kept in remembered sets.

[Dave Ungar: Generation Scavenging: A Non-Disruptive High Performance Storage Reclamation Algorithm] [Richard E. Jones, Rafael Lins: Garbage Collection: Algorithms for Automatic Dynamic Memory Management]

Additional Observations

Note that having garbage collection may simplify heap management. Copying and compacting tends to maintain heap in a single block, making it possible to always allocate new objects at the end of a heap, making allocation potentially as simple as a single pointer addition operation. Similarly, tracing does not concern dead objects, making deallocation potentially an empty operation. All of this gets a bit more complicated when destructors become involved though, for a call to a destructor is not an empty operation.

The asynchronous nature of calls to destructors makes them unsuitable for code that frees contented resources. A strict enforcement of referential integrity also requires garbage collection to handle situations where a call to a destructor associated with an unreachable block makes that block reachable again.

Rehearsal

By now, you should understand what a memory layout of a typical process looks like. You should be able to describe how executable code, static data, heap and stack are stored in memory and what are their specific requirements with respect to process memory management.

Concerning the stack, you should be able to explain how return addresses, function arguments and local variables are stored on stack and how the contents of the stack can be elegantly accessed using relative addressing.

Concerning the heap, you should be able to outline the criteria of efficient heap management and relate them to typical heap usage patterns. You should be able to explain the working of common heap management algorithms in the light of these criteria and outline heap usage patterns for which these algorithms excel and fail.

You should be able to explain the principal approach to identifying garbage in garbage collecting algorithms and to discuss the principal differences between process memory management that relies on explicit garbage disposal and implicit garbage collection. You should understand the working of basic reference counting and reference tracing algorithms and see how the typical heap usage patterns lead to optimizations of the algorithms.

Based on your knowledge of how process memory management is used, you should be able to design an intelligent API that not only allows to allocate and free blocks of memory, but also helps to debug common errors in allocating and freeing memory.

Questions

1. Identify where the following function relies on the virtual address space to store executable code, static data, heap and stack.

   ```
   void *SafeAlloc (size_t iSize)
   {
     void *pResult = malloc (iSize);
     if (pResult == NULL)
     {
       printf ("Failed to allocate %zi bytes.\n", iSize);
       exit (ENOMEM);
     }
     return (pResult);
   }
   ```

2. List four distinct types of content that reside in a virtual address space of a typical process .
3. Explain the advantages of segmentation over flat virtual memory address space.
4. Explain why random placement of allocated blocks in the virtual address space of a process can contribute to improved security.
5. Explain what the processor stack is used for with typical compiled procedural programming languages.
6. For a contemporary processor, explain how the same machine instructions with the same arguments can access local variables and arguments of a procedure regardless of their absolute address in the virtual address space. Explain why this is important.
7. Explain what is the function of a heap allocator.
8. Explain why the implementation of the heap allocator for user processes usually resides in user space rather than kernel space.
9. Design an interface of a heap allocator.
10. Explain the problems a heap allocator implementation must solve on multiprocessor hardware and sketch appropriate solutions.
11. Explain the rationale behind the Buddy Allocator and describe the function of the allocation algorithm.
12. Explain the rationale behind the Slab Allocator and describe the function of the allocation algorithm.
13. Describe what a heap allocator can do to reduce the overhead of the virtual memory manager.
14. Explain the function of a garbage collector.

15. Define precisely the conditions under which a memory block can be freed by a reference tracing garbage collector.
16. Describe the algorithm of a copying garbage collector.
17. Describe the algorithm of a mark and sweep garbage collector.
18. Describe the algorithm of a generational garbage collector. Assume knowledge of a basic copying garbage collector and a basic mark and sweep garbage collector.

    Hint: Essential to the generational garbage collector is the ability to collect only part of the heap. How is this possible without the collector missing some references ?

Notes

1. The text uses specific constants. They should be used as rough guidelines. Particular allocator configuration may differ.

Chapter 4. Device Management

Device Drivers

Traditionally, the operating system is responsible for controlling devices on behalf of applications. Even though applications could control devices directly, delegating the task to the operating system keeps the applications device independent and makes it possible to safely share devices among multiple applications.

The operating system concentrates the code for controlling specific devices in *device drivers*. The details of controlling individual devices tend to depend on the device model, version, manufacturer and other factors. A device driver can hide these details behind an interface that is the same for a class of similar devices. This makes it possible to keep the rest of the operating system code largely device independent as well.

To be done.

Architektura I/O systému. Přítomnost přerušení ovlivňuje strukturu driveru, bude mít část obsluhující požadavky na přerušení od hardware, která je volaná asynchronně (kdykoliv přijde přerušení) a část obsluhující požadavky na operace od software, která je volaná synchronně (když aplikace nebo operační systém zavolají ovladač). Mezi těmito částmi se komunikuje většinou pomocí front a bufferů, vzniká problém se zamykáním takto sdílených dat, protože obsluha přerušení od hardware může běžet současně s obsluhou operace od software. Tento problém se řeší použitím mechanismů, které dovolují naplánovat na později vykonání operací, které jsou součástí obsluhy přerušení od hardware (v Linuxu bottom half handlers a tasklets, ve Windows deferred procedure calls, v Solarisu pinned interrupt thread pools).

Pro označení těchto dvou částí driveru se používají termíny bottom half (asynchronně volaná část driveru, která se stará převážně o požadavky hardware) a top half (synchronně volaná část driveru, která se stará převážně o požadavky software). Toto označení odpovídá chápání architektury, kde na nejnižší úrovni je hardware, následují drivers, pak operační systém, pak aplikace.

Zmíněné označení koliduje s termíny v Linuxu, který jako top half označuje okamžitě vykonávanou a jako bottom half odloženou část obsluhy přerušení. Zahlédl jsem označení této terminologie jako Linuxové a té zmíněné výše jako BSD, řada textů se zdá se oběma terminologiím vyhýbá.

Asynchronous Requests

Example: Linux Tasklets

The interrupt handling code in the kernel is not reenterant. When an interrupt handler executes, additional interrupts are disabled and a statically allocated stack is used. This simplifies the code but also implies that interrupt handling must be short lest the ability of the kernel to respond to an interrupt promptly is affected. The kernel offers four related tools to postpone work done while servicing an interrupt, called soft irqs, tasklets, bottom half handlers and work queues.

Soft irqs are functions whose execution can be requested from within an interrupt handler. All pending soft irqs are executed by the kernel on return from an interrupt handler with additional interrupts enabled. Soft irqs that were raised again after return from an interrupt handler are executed by a kernel thread called ksoftirqd. A soft irq can execute simultaneously on multiple processors. The number of soft irqs is limited to 32, soft irqs are used within the kernel for example to update kernel timers and handle network traffic.

Two soft irqs are dedicated to executing low and high priority tasklets. Unlike a handler of a soft irq, a handler of a tasklet will only execute on one processor at a time. The number of tasklets is not limited, tasklets are the main tool to be used for scheduling access to resources within the kernel.

Finally, bottom half handlers are implemented using tasklets. To preserve backward compatibility with old kernels, only one bottom half handler will execute at a time. Bottom half handlers are deprecated.

When executed on return from an interrupt handler, soft irqs are not associated with any thread and therefore cannot use passive waiting. Since tasklets and bottom half handlers are implemented using soft irqs, the same constraint applies there as well. When passive waiting is required, work queues must be used instead. A work queue is similar to a tasklet but is always associated with a kernel thread, trading the ability to execute immediately after an interrupt handler for the ability to wait passively.

[This information is current for kernel 2.6.23.]

References

1. Matthew Wilcox: I'll Do It Later: Softirqs, Tasklets, Bottom Halves, Task Queues, Work Queues and Timers.

Example: Windows Deferred Procedure Calls

Windows kernel provides the option of postponing work done while servicing an interrupt through the deferred procedure call mechanism. The interrupt service routine can register a deferred procedure call, which gets executed later. The decision when to execute a deferred procedure call depends on the importance of the call, the depth of the queue and the rate of the interrupts.

```
// Registers DPC for a device
VOID IoInitializeDpcRequest (
  IN PDEVICE_OBJECT DeviceObject,
  IN PIO_DPC_ROUTINE DpcRoutine
);

// Schedules DPC for a device
VOID IoRequestDpc (
  IN PDEVICE_OBJECT DeviceObject,
  IN PIRP Irp,
  IN PVOID Context
);

// DPC
VOID DpcForIsr (
  IN PKDPC Dpc,
  IN struct _DEVICE_OBJECT *DeviceObject,
  IN struct _IRP *Irp,
  IN PVOID Context
);
```

Example: Solaris Pinned Threads

Solaris obsluhuje přerušení ve vyhrazených vláknech. Protože inicializace vlákna při přerušení by byla dlouhá, používají se *interrupt threads* jako omezená varianta *kernel threads*. Při přerušení se aktivní vlákno označí jako *pinned thread*, což znamená, že se uspí, ale že nemůže být naplánováno na jiný procesor, protože jeho kontext není úplně uschován. Spustí se interrupt thread, který obslouží přerušení, po jeho ukončení se vzbudí pinned thread. Pokud by interrupt thread zavolal funkci, která ho potřebuje uspat, systém ho automaticky konvertuje na kernel thread, ten uspí a vzbudí pinned thread.

Synchronous Requests

Rozhrani pro tridy devices. Problemy s kopirovanim dat na rozhranich.

User interface, blokující funkce, funkce s asynchronní signalizací. Problémy asynchronní signalizace při chybách, signalizace chyb (indikace result kódem, indikace globální proměnnou, asynchronní indikace á la DOS, chytřejší asynchronní indikace).

Example: Unix Driver Model

Block devices, přenos dat po blocích, velikost bloků dána vlastnostmi zařízení. Bloky jsou adresovatelné, přímý přístup k datům. Mají cache, mají fronty a obslužné strategie.

Character devices, přenos dat po jednotlivých bajtech, sekvenční přístup. Nemají cache, mají read a write rutiny.

Výše uvedené rozdělení na character a block devices má kořeny v dobách, kdy se pro I/O operace používaly tzv. kanálové procesory. Ty podporovaly právě dva režimy přenosu dat z periferních zařízení a to buď po jednotlivých bajtech nebo po blocích.

V současné době není toto členení příliš opodstatněné a rozhodujícím činitelem je spíše sekvenční či náhodný přístup k datům. Příkladem toho budiž zařízení pro digitalizaci videa, ke kterému se přistupuje jako ke znakovému zařízení, které však poskytuje data s granularitou celých frames, nikoliv jednotlivých bajtů. Ovladač zařízení podporuje mapování do paměti, což aplikaci umožňuje snadný přístup k jednotlivým bajtům jednoho frame, není však možné žádat zařízení o předchozí frames.

V UNIXu říká major device number typ ovladače, minor device number pořadové číslo zařízení (zhruba).

Example: Linux Driver Model

The driver model facilitates access to common features of busses with devices and to drivers with classes and interfaces. The structure maintained by the driver model is accessible via the sysfs filesystem.

Common features of busses include listing devices connected to the bus and drivers associated with the bus, matching drivers to devices, hotplugging devices, suspending and resuming devices.

```
> ls -R /sys/bus
/sys/bus:
pci  pci_express  pcmcia  scsi  usb
/sys/bus/pci:
devices  drivers
/sys/bus/pci/devices:
0000:00:00.0  0000:00:1a.7  0000:00:1c.3  0000:00:1d.7  0000:00:1f.3
0000:00:01.0  0000:00:1b.0  0000:00:1c.4  0000:00:1e.0  0000:01:00.0
/sys/bus/pci/drivers:
agpgart-intel  ata_piix  ehci_hcd  ohci_hcd  uhci_hcd  ahci  e1000  HDA Intel
...
```

Common features of devices include listing interfaces provided by the device and linking to the class and the driver associated with the device. The driver provides additional features specific to the class or the interfaces or the device.

```
> ls -R /sys/devices
/sys/devices:
pci0000:00
/sys/devices/pci0000:00:
0000:00:19.0
/sys/devices/pci0000:00/0000:00:19.0:
class  config  device  driver  irq  net  power  vendor
/sys/devices/pci0000:00/0000:00:19.0/net:
eth0
/sys/devices/pci0000:00/0000:00:19.0/net/eth0:
address  broadcast  carrier  device  features  flags  mtu  power  statistics
...
```

To be done.

Network devices include/linux/netdevice.h struct net_device.

Block devices include/linux/blkdev.h struct request_queue.

Character devices include/linux/cdev.h struct cdev.

IOCTL with strace

When a new device is connected to a bus, the driver of the bus notifies the udevd daemon, providing information on the identity of the device. The daemon uses this information to locate the appropriate driver in the driver database, constructed from information provided by the modules during module installation. When the appropriate driver is loaded, it is associated with the device, which thus becomes ready to use. The notifications can be observed using the udevmonitor command.

```
> udevmonitor --env
UEVENT[12345.67890] add /devices/pci0000:00/0000:00:1a.7/usb1/1-3/1-3:1.0 (usb)
ACTION=add
DEVPATH=/devices/pci0000:00/0000:00:1a.7/usb1/1-3/1-3:1.0
SUBSYSTEM=usb
DEVTYPE=usb_interface
DEVICE=/proc/bus/usb/001/006
PRODUCT=457/151/100
INTERFACE=8/6/80
MODALIAS=usb:v0457p0151d0100dc00dsc00dp00ic08isc06ip50
```

[This information is current for kernel 2.6.23.]

References

1. Patrick Mochel: The Linux Kernel Device Model.

Power Management

Rehearsal

Questions

1. Popište obecnou architekturu ovladače zařízení a vysvětlete, jak tato architektura dovoluje zpracovávat současně asynchronní požadavky od hardware a synchronní požadavky od software.

2. Popište obvyklý průběh obsluhy přerušení jako asynchronního požadavku na obsluhu hardware ovladačem zařízení. Průběh popište od okamžiku přerušení do okamžiku ukončení obsluhy. Předpokládejte obvyklou architekturu ovladače, kde spolu asynchronně a synchronně volané části ovladače komunikují přes sdílenou frontu požadavků. Každý krok popište tak, aby bylo zřejmé, kdo jsou jeho účastníci a kde získají informace potřebné pro vykonání daného kroku.
3. Popište obvyklý průběh systémového volání jako synchronního požadavku na obsluhu software ovladačem zařízení. Předpokládejte obvyklou architekturu ovladače, kde spolu asynchronně a synchronně volané části ovladače komunikují přes sdílenou frontu požadavků. Průběh popište od okamžiku volání do okamžiku ukončení obsluhy. Každý krok popište tak, aby bylo zřejmé, kdo jsou jeho účastníci a kde získají informace potřebné pro vykonání daného kroku.

Devices

Busses

Although busses are not devices in the usual sense, devices that represent busses are sometimes available to control selected features of the busses. Most notable are features for bus configuration.

Example: SCSI

The SCSI bus provides configuration in a form of an inquiry command. Devices are addressed using ID (0-7 or 0-15) and LUN (0-7). ID selects a device, LUN selects a logical unit within the device. Devices can communicate with each other by sending commands using command descriptor blocks. Examples of commands include Test Unit Ready (0), Sequential Read (8), Sequential Write (0Ah), Seek (0Bh), Inquiry (12h), Direct Read (28h), Direct Write (2Ah). Commands can be queued and reordered.

Each device responds to the Inquiry command.

```
+=====-========-========-========-========-========-========-========-========+
|  Bit|   7    |   6    |   5    |   4    |   3    |   2    |   1    |   0    |
|Byte |        |        |        |        |        |        |        |        |
|=====+========-========-========-========-========-========-========-========|
| 0   | Operation Code: Inquiry (12h)                                         |
|-----+-----------------------------------------------------------------------|
| 1   | Logical Unit Number      | Reserved                          | EVPD   |
|-----+-----------------------------------------------------------------------|
| 2   | Page Code                                                             |
|-----+-----------------------------------------------------------------------|
| 3   | Reserved                                                              |
|-----+-----------------------------------------------------------------------|
| 4   | Allocation Length: Inquiry Reply Length (96)                          |
|-----+-----------------------------------------------------------------------|
| 5   | Control                                                               |
+=====-=======================================================================+
```

```
+=====-========-========-========-========-========-========-========-========+
|  Bit|   7    |   6    |   5    |   4    |   3    |   2    |   1    |   0    |
|Byte |        |        |        |        |        |        |        |        |
|=====+========-========-========+========-========-========-========-========|
| 0   | Peripheral Qualifier     | Peripheral Device Type                     |
|-----+-----------------------------------------------------------------------|
| 1   | RMB    | Device-Type Modifier                                          |
|-----+----------------------------------------------------------------------|
| 2   | ISO Version     | ECMA Version             | ANSI Version             |
|-----+-----------------+-----------------------------------------------------|
| 3   | AENC   | TrmIOP | Reserved        | Response Data Format              |
|-----+----------------------------------------------------------------------|
| 4   | Additional Length (n-4)                                               |
|-----+----------------------------------------------------------------------|
| 5   | Reserved                                                              |
|-----+----------------------------------------------------------------------|
| 6   | Reserved                                                              |
|-----+----------------------------------------------------------------------|
| 7   | RelAdr | WBus32 | WBus16 |  Sync  | Linked |Reserved| CmdQue | SftRe  |
|-----+----------------------------------------------------------------------|
| 8   | (MSB)                                                                 |
|- - -+---                        Vendor Identification                  ---|
| 15  |                                                              (LSB) |
|-----+----------------------------------------------------------------------|
| 16  | (MSB)                                                                 |
|- - -+---                        Product Identification                 ---|
| 31  |                                                              (LSB) |
|-----+----------------------------------------------------------------------|
| 32  | (MSB)                                                                 |
|- - -+---                        Product Revision Level                 ---|
| 35  |                                                              (LSB) |
|-----+----------------------------------------------------------------------|
| 36  |                                                                       |
|- - -+---                        Vendor Specific                        ---|
| 55  |                                                                       |
|-----+----------------------------------------------------------------------|
| 56  |                                                                       |
|- - -+---                        Reserved                               ---|
| 95  |                                                                       |
|=====+======================================================================|
| 96  |                                                                       |
|- - -+---                        Additional Vendor Specific             ---|
| n   |                                                                       |
+=====-=======================================================================+
```

```
> cat /proc/scsi/scsi
Attached devices:
Host: scsi0 Channel: 00 Id: 00 Lun: 00
  Vendor: QUANTUM  Model: ATLAS10K2-TY184L Rev: DA40
  Type:   Direct-Access                    ANSI SCSI revision: 03
Host: scsi1 Channel: 00 Id: 05 Lun: 00
  Vendor: NEC      Model: CD-ROM DRIVE:466 Rev: 1.06
  Type:   CD-ROM                           ANSI SCSI revision: 02
Host: scsi2 Channel: 00 Id: 00 Lun: 00
  Vendor: PLEXTOR  Model: DVDR   PX-708A   Rev: 1.02
  Type:   CD-ROM                           ANSI SCSI revision: 02
```

References

1. SCSI-1 Standard.
2. SCSI-2 Standard.
3. SCSI-3 Standard.
4. Heiko Eißfeldt: The Linux SCSI Programming HowTo.

Example: PCI

The PCI bus provides configuration in a form of a configuration space, which is separate from memory space and port space. Apart from the usual port read, port write, memory read, memory write commands, the C/BE signals can also issue configuration read (1010b) and configuration write (1011b) commands. Because address bus cannot be used to address devices whose address is not yet known, each slot has separate IDSEL signal which acts as CS signal for configuration read and configuration write commands.

Each device can have up to 8 independent functions, each function can have up to 64 configuration registers, the first 64 bytes are standardized. The standardized registers contain vendor ID and device ID, subsystem vendor ID and subsystem device ID, flags, memory address ranges, port address ranges, interrupts, etc.

Devices are addressed using domains (0-0FFFFh), busses (0-0FFh), slots (0-1Fh), functions (0-7). A domain typically addresses a host bridge. A bus typically addresses a bus controller, a slot typically addresses a device.

```
> lspci -t
-[0000:00]-+-00.0
           +-01.0-[0000:01]----00.0
           +-02.0-[0000:02-03]----1f.0-[0000:03]----00.0
           +-1e.0-[0000:04]--+-0b.0
           |                 +-0c.0
           |                 \-0d.0
           +-1f.0
           +-1f.1
           +-1f.2
           +-1f.3
           +-1f.4
           \-1f.5
```

The example shows a computer with one domain, which has three bridges from bus 0 to busses 1, 2 and 4, one bridge from bus 2 to bus 3, one device with six functions on bus 0, one device on bus 1, one device on bus 3, three devices on bus 4.

```
> lspci
00:00.0 Host bridge: Intel Corp. 82860 860 (Wombat) Chipset Host Bridge (MCH) (rev 04)
00:01.0 PCI bridge: Intel Corp. 82850 850 (Tehama) Chipset AGP Bridge (rev 04)
00:02.0 PCI bridge: Intel Corp. 82860 860 (Wombat) Chipset AGP Bridge (rev 04)
00:1e.0 PCI bridge: Intel Corp. 82801 PCI Bridge (rev 04)
00:1f.0 ISA bridge: Intel Corp. 82801BA ISA Bridge (LPC) (rev 04)
00:1f.1 IDE interface: Intel Corp. 82801BA IDE U100 (rev 04)
00:1f.2 USB Controller: Intel Corp. 82801BA/BAM USB (Hub #1) (rev 04)
00:1f.3 SMBus: Intel Corp. 82801BA/BAM SMBus (rev 04)
00:1f.4 USB Controller: Intel Corp. 82801BA/BAM USB (Hub #2) (rev 04)
00:1f.5 Multimedia audio controller: Intel Corp. 82801BA/BAM AC'97 Audio (rev 04)
01:00.0 VGA compatible controller: ATI Technologies Inc Radeon RV100 QY [Radeon 7000/VE
02:1f.0 PCI bridge: Intel Corp. 82806AA PCI64 Hub PCI Bridge (rev 03)
03:00.0 PIC: Intel Corp. 82806AA PCI64 Hub Advanced Programmable Interrupt Controller (
04:0b.0 Ethernet controller: 3Com Corporation 3c905C-TX/TX-M [Tornado] (rev 78)
04:0c.0 FireWire (IEEE 1394): Texas Instruments TSB12LV26 IEEE-1394 Controller (Link)
04:0d.0 Ethernet controller: Intel Corp. 82544EI Gigabit Ethernet Controller (Copper) (
```

Check the example to see what are the bridges from the previous example. Bus 1 is on board AGP going to ATI VGA, bus 2 is on board AGP going to PCI64 with APIC, bus 4 is on board PCI going to network cards.

Check the example to see what are the devices from the previous example. Device 00:1f is single chip integrating ISA bridge, IDE, USB, SMB, audio.

```
> lspci -vvs 04:0b.0
04:0b.0 Ethernet controller: 3Com Corporation 3c905C-TX/TX-M [Tornado] (rev 78)
        Subsystem: Dell: Unknown device 00d8
        Control: I/O+ Mem+ BusMaster+ SpecCycle- MemWINV+ VGASnoop- ParErr- Stepping- S
        Status: Cap+ 66Mhz- UDF- FastB2B- ParErr- DEVSEL=medium >TAbort- <TAbort- <MAbo
        Latency: 64 (2500ns min, 2500ns max), Cache Line Size 10
        Interrupt: pin A routed to IRQ 23
        Region 0: I/O ports at dc80 [size=128]
        Region 1: Memory at ff3ffc00 (32-bit, non-prefetchable) [size=128]
        Expansion ROM at ff400000 [disabled] [size=128K]
        Capabilities: [dc] Power Management version 2
                Flags: PMEClk- DSI- D1+ D2+ AuxCurrent=0mA PME(D0+,D1+,D2+,D3hot+,D3col
                Status: D0 PME-Enable- DSel=0 DScale=2 PME-
```

Check the example to see what the configuration registers reveal. The identification of the device actually says class 200h, vendor ID 10B7h, device ID 9200h, subsystem vendor ID 1028h, subsystem device ID 0D8h. This means class Ethernet, vendor 3Com, device 3C905C, subsystem vendor Dell, subsystem device unknown.

Example: USB

The USB bus provides configuration in a form of a device descriptor. Devices are addressed by unique addresses (0-127), communication uses message or stream pipes between endpoints. A device connect as well as supported speed is recognized electrically by a hub, which indicates a status change to the host. The host queries the hub to determine the port on which the device is connected and issues power and reset command to the hub for the port. The host assigns a unique address to the device using the default address of 0 and the default control pipe with endpoint 0 and then queries and sets the device configuration.

```
> lsusb -t
Bus#  1
`-Dev#   1 Vendor 0x0000 Product 0x0000
  `-Dev#   2 Vendor 0x046d Product 0xc01b

> lsusb
Bus 001 Device 002: ID 046d:c01b Logitech, Inc. MX310 Optical Mouse
Bus 001 Device 001: ID 0000:0000

> lsusb -vv -s 1:2

Bus 001 Device 002: ID 046d:c01b Logitech, Inc. MX310 Optical Mouse
Device Descriptor:
  bLength                18
  bDescriptorType         1
  bcdUSB               2.00
  bDeviceClass            0 (Defined at Interface level)
  bDeviceSubClass         0
  bDeviceProtocol         0
  bMaxPacketSize0         8
  idVendor           0x046d Logitech, Inc.
  idProduct          0xc01b MX310 Optical Mouse
  bcdDevice           18.00
  iManufacturer           1 Logitech
  iProduct                2 USB-PS/2 Optical Mouse
  iSerial                 0
  bNumConfigurations      1
  Configuration Descriptor:
    bLength                 9
    bDescriptorType         2
    wTotalLength           34
    bNumInterfaces          1
    bConfigurationValue     1
    iConfiguration          0
    bmAttributes         0xa0
      Remote Wakeup
    MaxPower               98mA
```

```
Interface Descriptor:
  bLength                 9
  bDescriptorType         4
  bInterfaceNumber        0
  bAlternateSetting       0
  bNumEndpoints           1
  bInterfaceClass         3 Human Interface Devices
  bInterfaceSubClass      1 Boot Interface Subclass
  bInterfaceProtocol      2 Mouse
  iInterface              0
  Endpoint Descriptor:
    bLength                 7
    bDescriptorType         5
    bEndpointAddress     0x81  EP 1 IN
    bmAttributes            3
      Transfer Type            Interrupt
      Synch Type               none
      Usage Type               Data
    wMaxPacketSize     0x0005  bytes 5 once
    bInterval              10
```

Check the example to see what the device descriptor reveals. The interface class HID means a human interface device, the interface subclass BOOT means a device useful at boot, the interface protocol MOUSE means a pointing device. A report descriptor would be used to describe the interface but a parser for the report descriptor is complicated. Devices useful at boot can therefore be identified from the interface class, interface subclass and interface protocol. The interrupt mentioned in the descriptor does not mean processor interrupt but interrupt transfer as one of four available transfer types for specific transfer pipe.

References

1. Universal Serial Bus Specification 1.0.
2. Universal Serial Bus Specification 1.1.
3. Universal Serial Bus Specification 2.0.

Clock

Využití hodin, kalendář, plánování procesů v preemptivním multitaskingu, účtování strojového času, alarmy pro user procesy, watchdogs pro systém, profilování, řízení.

Principy hodinového hardware, odvození od sítě a od krystalového oscilátoru. Možné funkce hardware, tedy samotný čítač, one time counter, periodic counter, kalendář.

Využití hodinového hardware pro různé využití hodin - pro kalendář - pro plánování (ekvidistantní tiky, chce interrupt) - pro eventy (nastavování one time counteru je nejlepší, lze i jinak) - watchdog timer - profiling (buď statisticky koukat, kde je program, nebo přesně měřit).

Example: PC Clock

First source, Intel 8253 or Intel 8254 counter with 65536 default divisor setting yielding 18.2 Hz interrupt, which roughly corresponds to the original PC processor clock of 4.77 MHz divided by 4.

Second source, Motorola 146818 real time clock with CMOS RAM with 32768 kHz clock yielding 1024 Hz interrupt.

Other sources ?

Keyboard

Klasický příklad character device. Klávesnice bez řadiče. Klávesnice s řadičem, překódování kláves, type ahead buffer, přepínání focusu (zmínka o X-Windows na pomalých počítačích).

Example code keyboard handleru.

Mouse

Microsoft mouse. Serial 1200 bps, 7N1, 3 byte packets (sync + buttons + high bits X and Y, low bits X, low bits Y. Mouse Systems mouse. Serial 1200 bps, 8N1, 5 byte packets (sync + buttons, X, Y, delta X since X, delta Y since Y).

PS/2 mouse. Serial 10000-16667 bps, 8O1, 3 byte packets (sync + buttons + direction + overflow, delta X, delta Y). Mouse can receive commands, 0FFh reset, recognizes 3 modes of operation (stream - sends data when mouse moves, remote - sends data when polled, wrap - echoes received data).

Video Devices

Rozdělení na command interface a memory mapped interface. Popis vlastností terminálů s command interface, standardy řídících příkazů.

ANSI Escape Sequences treba **ESC [** <n> **J** (clear screen, 0 from cursor, 1 to cursor, 2 entire), **ESC [** <line> **;** <column> **H** (goto line and column), něco na barvy atd.

Popis terminálů s memory mapped interface, znakové a grafické displeje, práce s video RAM, akcelerátory, hardwarová podpora kurzoru, kreslení apod.

VGA režim 320x200x256, co byte to pixel, paleta 256 barev nastavovaná v registrech, video RAM souvislá oblast paměti. VGA režim 800x600x256, co byte to pixel, paleta 256 barev nastavovaná v registrech, video RAM window posouvané po 64K. Další režimy třeba bit planes nebo linear memory.

Zde je mimochodem vidět, jak se dá standardizovat na různých úrovních, stejné registry, různé registry ale stejná mapa video RAM, parametrizovatelná mapa video RAM, grafická primitiva.

Modern cards can do MPEG decoding, shading, lighting, whatever.

Audio Devices

To be done.

Disk Storage Devices

A disk is a device that can read and write fixed length sectors. Various flavors of disks differ in how sectors are organized. A hard disk has multiple surfaces where sectors of typically 512 bytes are organized in concentric tracks. A floppy disk has one or two surfaces where sectors of typically 512 bytes are organized in concentric tracks. A compact disk has one surface where sectors of typically 2048 bytes are organized in a spiral track.

Addressing

Initially, sectors on a disk were addressed using the surface, track and sector numbers. This had several problems. First, implementations of the ATA hardware interface and the BIOS software interface typically limited the number of surfaces to 16, the number of cylinders to 1024, and the number of sectors to 63. Second, the fact that the length of a cylinder depends on the distance from the center of the disk makes it advantageous to vary the number of sectors per cylinder. Lately, sectors on a disk are therefore addressed using a logical block address that numbers sectors sequentially.

Example: ATA Disk Access

An ATA disk denotes a disk using the Advanced Technology Attachment (ATA) or the Advanced Technology Attachment with Packet Interface (ATAPI) standard, which describes an interface between the disk and the computer. The ATA standard allows the disk to be accessed using the command block registers, the ATAPI standard allows the disk to be accessed using the command block registers or the packet commands.

The command block registers interface relies on a number of registers, including the Cylinder High, Cylinder Low, Device/Head, Sector Count, Sector Number, Command, Status, Features, Error, and Data registers. Issuing a command entails reading the Status register until its BSY and DRDY bits are cleared, which indicates that the disk is ready, then writing the other registers with the required parameter, and finally writing the Command register with the required command. When the Command register is written, the disk will set the Status register to indicate that a command is being executed, execute the command, and finally generate an interrupt to indicate that the command has been executed. Data are transferred either through the Data register or using Direct Memory Access.

The packet commands interface relies on the command block registers interface to issue a command that sends a data packet, which is interpreted as another command. The packet commands interface is suitable for complex commands that cannot be described using the command block registers interface.

Request Queuing

Because of the mechanical properties of the disk, the relative speed of the computer and the disk must be considered. A problem arises when the computer issues requests for accessing consecutive sectors too slowly relative to the rotation speed, this can be solved by interleaving of sectors. Another problem arises when the computer issues requests for accessing random sectors too quickly relative to the access speed, this can be solved by queuing of requests. The strategy of processing queued requests is important.

- The FIFO strategy of processing requests directs the disk to always service the first of the waiting requests. The strategy can suffer from excessive seeking across tracks.
- The Shortest Seek First strategy of processing requests directs the disk to service the request that has the shortest distance from the current position of the disk head. The strategy can suffer from letting too distant requests starve.
- The Bidirectional Elevator strategy of processing requests directs the disk to service the request that has the shortest distance from the current position of the disk head in the selected direction, which changes when no more requests in the selected direction are waiting. The strategy lets too distant requests starve at most two passes over the disk in both directions.
- The Unidirectional Sweep strategy of processing requests directs the disk to service the request that has the shortest distance from the current position of the disk head

in the selected direction, or the longest distance from the current position of the disk head when no more requests in the selected direction are waiting. The strategy lets too distant requests starve at most one pass over the disk in the selected directions.

The strategy used to process the queue of requests can be implemented either by the computer in software or by the disk in hardware. The computer typically only considers the current track that the disk head is on, because it does not change without the computer commanding the disk to do so, as opposed to the current sector that the disk head moves over.

Most versions of the ATA interface do not support issuing a new request to the disk before the previous request is completed, and therefore cannot implement any strategy to process the queue of requests. On the contrary, most versions of the SCSI and the SATA interfaces do support issuing a new request to the disk before the previous request is completed.

Example: SATA Native Command Queuing

A SATA disk uses Native Command Queuing as the mechanism used to maintain the queue of requests. The mechanism is coupled with First Party Direct Memory Access, which allows the drive to instruct the controller to set up Direct Memory Access for particular part of particular request.

Example: Linux Request Queuing

[Linux 2.2.18 /drivers/block/ll_rw_blk.c] Linux sice ve zdrojácích vytrvale používá název Elevator, ale ve skutečnosti řadí příchozí požadavky podle lineárního čísla sektorů, s výjimkou požadavků, které příliš dlouho čekají (256 přeskočení pro čtení, 512 pro zápis), ty se nepřeskakují. Tedy programy, které intenzivně pracují se začátkem disku, blokují programy, které pracují jinde.

[Linux 2.4.2 /drivers/block/ll_rw_blk.c & elevator.c] Novější Linux se polepšil, nové požadavky nejprve zkouší připojit do sekvence se stávajícími (s omezením maximální délky sekvence), pak je zařadí podle čísla sektoru, nikoliv však na začátek fronty a nikoliv před dlouho čekající požadavky. Výsledkem je one direction sweep se stárnutím.

[Linux 2.6.x] The kernel makes it possible to associate a queueing discipline with a block device by providing modular request schedulers. The three schedules implemented by the kernel are anticipatory, deadline driven and complete fairness queueing.

- The Anticipatory scheduler implements a modified version of the Unidirectional Sweep strategy, which permits processing of requests that are close to the current position of the disk head but in the opposite of the selected direction. Additionally, the scheduler enforces an upper limit on the time a request can starve.

  The scheduler handles read and write requests separately and inserts delays between read requests when it judges that the process that made the last request is likely to submit another one soon. Note that this implies sending the read requests to the disk one by one and therefore giving up the option of queueing read requests in hardware.

- The Deadline Driven scheduler actually also implements a modified version of the Unidirectional Sweep strategy, except that it assigns deadlines to all requests and when a deadline of a request expires, it processes the expired request and continues from that position of the disk head.

- The Complete Fairness Queueing scheduler is based on the idea of queueing requests from processes separately and servicing the queues in a round robin fashion, or in a weighted round robin fashion directed by priorities.

[This information is current for kernel 2.6.19.]

References

1. Hao Ran Liu: Linux I/O Schedulers. http://www.cs.ccu.edu.tw/~lhr89/linux-kernel/Linux IO Schedulers.pdf

Failures

Obsluha diskových chyb, retries, reset řadiče, chyby v software. Správa vadných bloků, případně vadných stop, v hardware, SMART diagnostics. Caching, whole track caching, read ahead, write back. Zmínit mirroring a redundantní disková pole.

RAID 0 uses striping to speed up reading and writing. RAID 1 uses plain mirorring and therefore requires pairs of disks of same size. RAID 2 uses bit striping and Hamming Code. RAID 3 uses byte striping and parity disk. RAID 4 uses block striping and parity disk. RAID 5 uses block striping and parity striping. RAID 6 uses block striping and double parity striping. The levels were initially defined in a paper of authors from IBM but vendors tend to tweak levels as they see fit. RAID 2 is not used, RAID 3 is rare, RAID 5 is frequent. RAID 0+1 and RAID 1+0 or RAID 10 combine RAID 0 and RAID 1.

Example: SMART Diagnostics

Linux 2.6.10 smartctl -a /dev/hda prints all device information. Attributes have raw value and normalized value, raw value is usually but not necessarily human readable, normalized value is 1-254, threshold 0-255 is associated with normalized value, worst lifetime value is kept. If value is less or equal to threshold then the attribute failed. Attributes are of two types, pre failure and old age. Failed pre failure attribute signals imminent failure. Failed old age attribute signals end of life. Attributes are numbered and some numbers are standardized.

Partitioning

Zmínit partitioning and logical volume management.

Example: IBM Volume Partitioning

To be done.

Example: GPT Volume Partitioning

To be done.

Example: Linux Logical Volume Management

Physical volumes, logical volumes, extents (size e.g. 32M), mapping of extents (linear or striped), snapshots.

```
> vgdisplay

  --- Volume group ---
  VG Name               volumes
  System ID
  Format                lvm2
  Metadata Areas        2
  Metadata Sequence No  10
  VG Access             read/write
  VG Status             resizable
  MAX LV                0
  Cur LV                3
  Open LV               3
  Max PV                0
  Cur PV                2
  Act PV                2
  VG Size               1.27 TiB
  PE Size               32.00 MiB
  Total PE              41695
  Alloc PE / Size       24692 / 771.62 GiB
  Free  PE / Size       17003 / 531.34 GiB
  VG UUID               fbvtrb-GFbS-Nvf4-Ogg3-J4fX-dj83-ebh39q

> pvdisplay --map

  --- Physical volume ---
  PV Name               /dev/md0
  VG Name               volumes
  PV Size               931.39 GiB / not usable 12.56 MiB
  Allocatable           yes
  PE Size               32.00 MiB
  Total PE              29804
  Free PE               17003
  Allocated PE          12801
  PV UUID               hvfcSD-FvSp-xJn4-lsR3-40Kx-LdDD-wvfGfV

  --- Physical Segments ---
  Physical extent 0 to 6875:
    Logical volume /dev/volumes/home
    Logical extents 0 to 6875
  Physical extent 6876 to 6876:
    Logical volume /dev/volumes/var
    Logical extents 11251 to 11251
  Physical extent 6877 to 12800:
    Logical volume /dev/volumes/home
    Logical extents 6876 to 12799
  Physical extent 12801 to 29803:
    FREE

> lvdisplay --map

  --- Logical volume ---
  LV Name                /dev/volumes/home
  VG Name                volumes
  LV UUID                OAdf3v-zfI1-w5vq-tFVr-Sfgv-yvre-GWFb3v
  LV Write Access        read/write
  LV Status              available
  LV Size                400.00 GiB
  Current LE             12800
  Segments               2
  Allocation             inherit
  Read ahead sectors     auto
  - currently set to     256
  Block device           253:2

  --- Segments ---
  Logical extent 0 to 6875:
    Type  linear
    Physical volume /dev/md0
    Physical extents 0 to 6875
  Logical extent 6876 to 12799:
    Type  linear
    Physical volume /dev/md0
    Physical extents 6877 to 12800
```

Memory Storage Devices

Similar to disks are various memory devices, based most notably on NOR and NAND types of FLASH memory chips. These memory chips retain their content even when powered off, but reading and writing them is generally slower and explicit erasing is required before writing. Erasing is only possible on whole blocks at a time. The NOR chips allow reading and writing arbitrary addresses. The NAND chips allow reading and writing whole pages at a time only. Blocks are typically tens to hundreds of kilobytes in size, pages are typically hundreds of bytes to kilobytes in size, all naturally powers of two.

The individual blocks of the memory chips wear down by erasures, with the typical lifetime ranging between tens of thousands and tens of millions of erasures. To make sure the wear is spread evenly across the entire device, block remapping is used to achieve *wear levelling*. Most memory storage devices that masquerade as disks support wear levelling in hardware.

Devices that masquerade as disks must mimic the ability to rewrite individual sectors. This requires block erasures that happen at different granularity than sector writes. Rewriting only some sectors of a block triggers copying of remaining sectors, leading to *write amplification*. Some controllers support a special trim command that marks a sector as unused. These sectors are not copied when their block is erased.

Network Cards

Scatter gather. Checksumming. Segmentation.

Parallel Ports

To be done.

Serial Ports

To be done.

Printers

To be done.

Modems

To be done.

Rehearsal

Questions

1. List the features that a hardware device that represent a bus typically provides.
2. List the features that a hardware clock device typically provides.
3. List the features that a hardware keyboard device typically provides.
4. List the features that a hardware mouse device typically provides.
5. List the features that a hardware terminal device with command interface typically provides.
6. List the features that a hardware terminal device with memory mapped interface typically provides.
7. List the features that a hardware disk device typically provides.
8. Explain the properties that a hardware disk interface must have to support hardware ordering of disk access requests.
9. Describe at least three strategies for ordering disk access requests. Evaluate how the strategies optimize the total time to execute the requests.
10. Explain the role of a disk partition table.
11. List the features that a network interface hardware device typically provides.

Rehearsal

Exercises

1. Navrhněte rozhraní mezi ovladačem disku nabízejícím funkce pro čtení a zápis bloku sektorů, schopné zpracovávat více požadavků současně, a vyššími vrstvami operačního systému. Pro vámi navržené rozhraní popište architekturu ovladače disku, která je schopna obsluhovat přerušení a volání z vyšších vrstev operačního systému, včetně algoritmu ošetření přerušení a algoritmů funkcí pro čtení a zápis bloku sektorů.

Chapter 5. File Subsystem

File systém poskytuje abstrakce adresářů a souborů nad disky, případně i jinými typy paměťových médií. Tytéž abstrakce adresářů a souborů se mohou použít i k jiným účelům, například ke zpřístupnění stavu systému nebo ke zprostředkování síťové komunikace.

Základní požadavky kladené na file systém jsou schopnost ukládat velký počet i velký objem dat s co nejmenší kapacitní a časovou režií, schopnost odolat výpadkům systému bez poškození uložených dat, schopnost zabezpečit uložená data před neoprávněným přístupem, schopnost koordinovat sdílení uložených dat.

Abstractions And Operations

Stream File Operations

Mezi nejjednodušší operace patří sekvenční přístup k souborům po záznamech nebo po bajtech, následují operace pro náhodný přístup. Téměř vždy mají podobu pětice operací pro otevření a zavření souboru, nastavení pozice v souboru, čtení a zápis.

Těmto operacím v podstatě odpovídá dnešní představa souboru jako streamu bajtů, případně někdy více streamů bajtů. Výjimkami jsou specializované systémy souborů, které dovolují vnitřní členění souborů například v podobě stromu, ale ty jsou spíše z urban legends.

Za úvahu stojí, proč jsou operace na soubory typicky rozděleny právě do zmiňované pětice. Je totiž zjevně možné udělat jen operace read a write, které budou specifikovat jméno souboru, pozici a velikost bloku.

Důvody pro pětici operací jsou možnost odstranit při běžném způsobu práce se soubory opakování operací jako je nalezení pozice na disku ze jména souboru a pozice v souboru, možnost spojit otevírání souboru s dalšími operacemi jako je zamykání nebo kontrola oprávnění.

Důvodem pro dvojici operací je možnost implementovat file systém bezestavově, což přináší výhody v distribuovaných systémech.

Operace bývají k dispozici v synchronní i asynchronní verzi.

Example: Linux Stream File Operations

```
int open (char *pathname, int flags);
int open (char *pathname, int flags, mode_t mode);
int creat (char *pathname, mode_t mode);
int close (int fd);
```

The open, creat and close operations open and close a file stream. The O_RDONLY, O_WRONLY, O_RDWR open mode flags tell whether the file is opened for reading, writing, or both. This information is useful for access right checks and potentially also for sharing and caching support. These flags can be combined with O_CREAT to create the file if needed, O_EXCL to always create the file, O_TRUNC to truncate the file if applicable, O_APPEND to append to the file.

The O_NONBLOCK flag indicates that operations on the file stream should not block, O_SYNC requests that operations that change the file stream block until the changes are safely written to the underlying media.

The value of mode contains the standard UNIX access rights.

```
off_t lseek (int fildes, off_t offset, int whence);
```

The lseek operation sets the position in the file stream. The whence argument is one of SEEK_SET, SEEK_CUR, SEEK_END, indicating whether the offset argument is counted relatively from the beginning, current position, or end of the file stream.

```
ssize_t read (int fd, void *buf, size_t count);
ssize_t write (int fd, void *buf, size_t count);
ssize_t pread (int fd, void *buf, size_t count, off_t offset);
ssize_t pwrite (int fd, void *buf, size_t count, off_t offset);
```

The read and write operations are trivial, more interesting are their vectorized and asynchronous counterparts.

```
ssize_t readv (int fd, struct iovec *vector, int count);
ssize_t writev (int fd, struct iovec *vector, int count);

struct iovec {
  void    *iov_base;
  size_t  iov_len;
};

int aio_read (struct aiocb *aiocbp);
int aio_write (struct aiocb *aiocbp);

int aio_error (struct aiocb *aiocbp);
ssize_t aio_return (struct aiocb *aiocbp);

int aio_suspend (
  struct aiocb *cblist [],
  int n, struct timespec *timeout);

int aio_cancel (int fd, struct aiocb *aiocbp);

int lio_listio (
  int mode, struct aiocb *list [],
  int nent, struct sigevent *sig);

struct aiocb {
  int               aio_fildes;
  off_t             aio_offset;
  void              *aio_buf;
  size_t            aio_nbytes;
  int               aio_reqprio;
  struct sigevent   aio_sigevent;
  int               aio_lio_opcode;
  ...
}

int posix_fadvise (int fd, off_t offset, off_t len, int advice);
int posix_fallocate (int fd, off_t offset, off_t len);
```

Advice can be given on future use of the file. Flags describing the future use include POSIX_FADV_NORMAL for no predictable pattern, POSIX_FADV_SEQUENTIAL and POSIX_FADV_RANDOM for specific patterns, POSIX_FADV_NOREUSE for data accessed once, and POSIX_FADV_WILLNEED and POSIX_FADV_DONTNEED to determine whether data will be accessed in the near future.

Example: Windows Stream File Operations

```
HFILE OpenFile (LPCSTR lpFileName, LPOFSTRUCT lpReOpenBuff, UINT uStyle);

HANDLE CreateFile (
  LPCTSTR lpFileName,
  DWORD dwDesiredAccess,
  DWORD dwShareMode,
  LPSECURITY_ATTRIBUTES lpSecurityAttributes,
  DWORD dwCreationDisposition,
  DWORD dwFlagsAndAttributes,
  HANDLE hTemplateFile);

HANDLE ReOpenFile (HANDLE hOriginalFile, DWORD dwDesiredAccess, DWORD dwShareMode, DWOR

BOOL CloseHandle (HANDLE hObject);
```

The OpenFile, CreateFile and CloseHandle operations open and close a file stream. Various combinations of flags can direct the calls to create or truncate the file if needed, other flags tell whether the file is opened for reading, writing, or both, and what sharing is allowed.

The OpenFile operation, currently deprecated, illustrates some less common design features. Given appropriate flags, it can open a dialog box when the requested file does not exist. When the file name does not contain a path, the operation searches system directories.

The ReOpenFile operation can open an already open file with different flags.

Mapped File Operations

S rozmachem stránkování se stalo běžné, že každá stránka paměti je spojena s daty na disku, což je možné v principu využít také pro přístup k souborům, pokud operační systém dá aplikacím možnost specifikovat, s jakými daty na disku jsou stránky spojené. Tato možnost se označuje termínem memory mapped files.

Memory mapped files uměl například již MULTICS kolem roku 1965, dnes je podporují prakticky všechny systémy včetně Linuxu a Windows.

Typickými operacemi je dvojice map a unmap, kde map říká, kterou část kterého souboru mapovat kam do paměti, unmap pak toto mapování ruší. Inherentním problémem memory mapped files je problém změny délky souboru při zápisu, neboť nelze prostě říci, že zápis za namapovaný blok paměti má soubor prodloužit. Další problémy vznikají v situaci, kdy se pro přístup k souboru používají současně stream i mapped operace, tam operační systém zpravidla převádí stream operace na mapped přístup k bufferům kernelu.

Example: Linux Mapped File Operations

```
void *mmap (void *start, size_t length,
            int prot, int flags,
            int fd, off_t offset);
int munmap (void *start, size_t length);
```

Pokud flags neuvádějí jinak, adresa se bere pouze jako nápověda, systém může namapovat soubor od jiné adresy, kterou vrátí. Adresa musí být zarovnána na hranici stránky. To je pochopitelné, paměťově mapované soubory implementuje file systém ve spolupráci se správcem virtuální paměti, který žádá o data při výpadcích stránek.

Na rozdíl od adresy už nemusí být délka zarovnána na hranici stránky, pro kratší soubory bude poslední stránka doplněna nulami a data zapsaná za konec souboru se při odmapování souboru zahodí.

Ochrana daná parametrem prot je PROT_READ, PROT_WRITE, PROT_EXEC nebo PROT_NONE, případně kombinace, ale opět kvůli způsobu implementace je jasné, že ne všechny kombinace budou k dispozici.

Hlavní flags jsou MAP_SHARED pro normální sdílení změn, MAP_PRIVATE pro vytváření kopií technikou copy on write, MAP_FIXED při požadavku mapovat právě na uvedenou adresu, MAP_ANONYMOUS pro mapování bez souboru.

Flags MAP_PRIVATE a MAP_FIXED mají zřejmý význam při nahrávání aplikací do paměti.

```
void *mremap (
  void *old_address, size_t old_size,
  size_t new_size, unsigned long flags);
```

Snad jediný zajímavý flag MREMAP_MAYMOVE.

```
int msync (void *start, size_t length, int flags);

int posix_madvise (void *addr, size_t len, int advice);
```

Advice can be given on future use of the mapped file. Flags describing the future use include POSIX_MADV_NORMAL for no predictable pattern, POSIX_MADV_SEQUENTIAL and POSIX_MADV_RANDOM for specific patterns, and POSIX_MADV_WILLNEED and POSIX_MADV_DONTNEED to determine whether data will be accessed in the near future.

Example: Windows Mapped File Operations

```
HANDLE CreateFileMapping (HANDLE hFile,
                          LPSECURITY_ATTRIBUTES lpFileMappingAttributes,
                          DWORD flProtect,
                          DWORD dwMaximumSizeHigh,
                          DWORD dwMaximumSizeLow,
                          LPCTSTR lpName);
```

Toto volání vytvoří abstraktní objekt reprezentující mapovaný soubor, ještě ale nic nenamapuje.

Flagy PAGE_READONLY, PAGE_READWRITE, PAGE_READCOPY, význam zřejmý. Flag SEC_COMMIT vyžaduje přidělení fyzického prostoru v paměti či na disku. Flag SEC_IMAGE upozorňuje na mapování spustitelného souboru. Flag SEC_NOCACHE, význam zřejmý. Flag SEC_RESERVE vyžaduje rezervaci bez přidělení fyzického prostoru v paměti či na disku.

Handle souboru může být 0xFFFFFFFF, pak musí být uvedena i velikost mapovaného bloku, systém vyhradí požadovaný prostor podobně jako při odkládání paměti při stránkování.

```
LPVOID MapViewOfFile (HANDLE hFileMappingObject, DWORD dwDesiredAccess,
                      DWORD dwFileOffsetHigh, DWORD dwFileOffsetLow,
                      DWORD dwNumberOfBytesToMap);
LPVOID MapViewOfFileEx (HANDLE hFileMappingObject, DWORD dwDesiredAccess,
                        DWORD dwFileOffsetHigh, DWORD dwFileOffsetLow,
                        DWORD dwNumberOfBytesToMap, LPVOID lpBaseAddress);
BOOL UnmapViewOfFile (LPCVOID lpBaseAddress);
```

Namapuje objekt reprezentující mapovaný soubor.

Flags FILE_MAP_WRITE, FILE_MAP_READ, FILE_MAP_ALL_ACCESS, FILE_MAP_COPY. Teď opravdu nevím, co se stane, když tyto flagy odporují flagům u CreateFileMapping, asi chyba.

Whole File Operations

To be done.

Example: Linux Whole File Operations

```
ssize_t sendfile (int out_fd, int in_fd, off_t *offset, size_t count);
ssize_t splice (int fd_in, loff_t *off_in, int fd_out, loff_t *off_out, size_t len, uns
```

To minimize the data copying overhead, it is possible to copy the content of one file to another.

Example: Windows Whole File Operations

Directory Operations

První operační systémy začínaly s jednoúrovňovým adresářem. Řešil se hlavně formát jména a atributy. Přišly problémy s vyhledáváním a kolizí jmen. Objevily se víceúrovňové adresáře a zavedení relativních odkazů vůči current directory. Jako poslední se objevila koncepce linků, kterou se dotvořil koncept adresářového grafu jak je znám dnes.

Mimochodem, stromovou strukturu adresářů vymysleli v AT&T Bell Labs v roce 1970.

Jako moderní koncepce se dnes ukazuje úplné oddělení adresářové struktury od souborů. Soubory jsou objekty, které obsahují data, programy operují s referencemi. V případě potřeby je pak možno v adresáři svázat takovou referenci se jménem.

Adresářová položka zpravidla obsahuje jméno souboru a atributy jako jsou přístupová práva, čas vytvoření a změny, některé systémy dovolují specifikovat libovolné atributy jako named values.

Základní operace na adresářích jsou otevření a zavření a čtení či prohledávání obsahu. Pro zápis obsahu jsou zvláštní funkce, které vytvářejí, přejmenovávají a mažou adresáře a soubory a nastavují jejich atributy, aby aplikace nemohly poškodit strukturu adresáře.

Example: Linux Directory Operations

```
DIR *opendir (const char *name);
int closedir (DIR *dir);
struct dirent *readdir (DIR *dir);
```

Pomocí těchto funkcí je možné číst adresář, zajímavá je samozřejmě struktura dirent. O té ale POSIX standard říká pouze, že bude obsahovat zero terminated jméno souboru a případně inode number, kterému ale říká sériové číslo souboru.

```
int scandir (const char *dir, struct dirent ***namelist,
             int (*select) (const struct dirent *),
             int (*compar) (const struct dirent **,
                            const struct dirent **));
```

Funkce scandir prohledá adresář a vrátí seznam položek, funkce select říká, které položky uvažovat, funkce compare říká, jak uvažované položky seřadit.

```
int stat (char *path, struct stat *buf);

struct stat {
  dev_t      st_dev;      // File device
  ino_t      st_ino;      // File inode
  mode_t     st_mode;     // Access rights
  nlink_t    st_nlink;
  uid_t      st_uid;      // Owner UID
  gid_t      st_gid;      // Owner GID
  dev_t      st_rdev;     // Device ID for special files
  off_t      st_size;     // Size in bytes
  blksize_t  st_blksize;  // Block size
  blkcnt_t   st_blocks;   // Size in blocks
  time_t     st_atime;    // Last access time
  time_t     st_mtime;    // Last modification time
  time_t     st_ctime;    // Last status change time
}
```

The stat system call provides information about a single directory entry.

Example: Windows Directory Operations

```
HANDLE FindFirstFile (LPCTSTR lpFileName, LPWIN32_FIND_DATA lpFindFileData);
BOOL FindNextFile (HANDLE hFindFile, LPWIN32_FIND_DATA lpFindFileData);

typedef struct _WIN32_FIND_DATA {
  DWORD dwFileAttributes;
  FILETIME ftCreationTime;
  FILETIME ftLastAccessTime;
  FILETIME ftLastWriteTime;
  DWORD    nFileSizeHigh;
  DWORD    nFileSizeLow;
  DWORD    dwReserved0;
  DWORD    dwReserved1;
  TCHAR    cFileName [MAX_PATH (= 260)];
  TCHAR    cAlternateFileName [14];
} WIN32_FIND_DATA;
```

Funkce převzaté z CP/M.

Sharing Support

Pokud přistupuje k souboru více procesů, je samozřejmě potřeba nějak definovat jak to bude vypadat. Minimální řešení je zajištění atomičnosti jednotlivých operací, což má jako default například UNIX či MS-DOS bez nataženého share.

Důmyslnější řešení je možnost zamykání celých souborů, to je například k dispozici v MS-DOSu při nataženém share. Při volání INT 21h fn 3Dh File Open se dalo zadat, zda se povolí další otevírání pro čtení a pro zápis. Podobnou věc umí UNIX pomocí volání flock.

Ještě o něco důmyslnější je možnost zamykat části souborů pro čtení či pro zápis. Tohle umí jak UNIX přes fcntl, tak třeba i nešťastný MS-DOS se share. Zadá se offset a délka zamykaného bloku a režim zamykání, ten je zpravidla shared (alias read) lock nebo exclusive (alias write) lock. Zamykání části souboru má jednu nevýhodu, totiž u každého souboru se musí pamatovat seznam existujících zámků, který se musí kontrolovat při relevantních operacích.

Aby se omezila velikost seznamu zámků, DOS například vyžaduje, aby odemykání specifikovalo pouze přesně takové bloky, které byly zamčené. Tedy není možné zamknout velký blok a odemknout kousek z jeho prostředka, čímž se odstraní problémy s fragmentací bloků.

Example: Linux Sharing Operations

Unix rozlišuje advisory a mandatory locking. Od začátku implementované jsou pouze advisory locks, totiž zámky, které se projeví pouze pokud se na ně proces zeptá. To samozřejmě není příliš bezpečné, a tak se doplnily ještě mandatory locks, které kontroluje kernel. Aby mandatory locks neblokovaly ve chvílích, kdy to stávající aplikace nečekaly, řeklo se, že budou automaticky nasazené na soubory s nastaveným group ID bitem a shozeným group execute bitem.

Mandatory locks uměl první tuším UNIX System V.

Nepříjemná vlastnost mandatory locks je, že mají trochu složitější sémantiku než advisory locks, a ne všechny systémy se do ní vždycky trefí. Sice existuje specifikace UNIX System V Interface Definition, ale tu snad nikdo přesně nedodržuje. Pěkný seznam odchylek je v dokumentaci o zamykání v Linux kernelu.

Mandatory locking také může způsobovat deadlock. Oblíbeným hackem bývalo zamknout si lokálně mandatory nějaký soubor a pak zkusit porušit tenhle zámek přes NFS, čímž se s trochou štěstí dal zablokovat NFS server.

Souborové zámky zpravidla nejsou vhodné pro časté zamykání s malou granularitou.

Example: Windows Sharing Operations

Locked and unlocked regions must match, it is not possible to lock a region and then unlock part of a region, or to lock multiple adjacent regions and then unlock the regions together.

Locking does not prevent reading through memory mapping.

Locks are unlocked on closing the locked file or terminating the owning process. Arbitrary time may elapse between closing or terminating and unlocking.

Consistency Support

To be done.

Example: Windows Transaction Operations

```
HANDLE CreateTransaction (
  LPSECURITY_ATTRIBUTES lpTransactionAttributes,
  LPGUID UOW,
  DWORD CreateOptions,
  DWORD IsolationLevel,
  DWORD IsolationFlags,
  DWORD Timeout,
  LPWSTR Description);

BOOL CommitTransaction (
  HANDLE TransactionHandle);
BOOL RollbackTransaction (
  HANDLE TransactionHandle);
```

Transaction context can be used to group together multiple operations and provide multiple readers with a consistent past snapshot of data in presence of a single writer. Most arguments of the context creation call are ignored and should be set to zero.

```
HANDLE CreateFileTransacted (
  LPCTSTR lpFileName,
  DWORD dwDesiredAccess,
  DWORD dwShareMode,
  LPSECURITY_ATTRIBUTES lpSecurityAttributes,
  DWORD dwCreationDisposition,
  DWORD dwFlagsAndAttributes,
  HANDLE hTemplateFile,
  HANDLE hTransaction,
  PUSHORT pusMiniVersion,
  PVOID pExtendedParameter);

BOOL DeleteFileTransacted(
  LPCTSTR lpFileName,
  HANDLE hTransaction);

BOOL CreateDirectoryTransacted (...);
BOOL RemoveDirectoryTransacted (...);

BOOL MoveFileTransacted (...);
BOOL CopyFileTransacted (...);
```

The support for transactions is generic, driven by a system transaction manager and cooperating resource managers. Transactional operations can therefore be provided by other parts of the system, such as registry.

Rehearsal

Questions

1. Popište obvyklé rozhraní operačního systému pro přístup k souborům pomocí operací čtení a zápisu. Funkce rozhraní uveďte včetně argumentů a sémantiky.
2. Vysvětlete, proč obvyklé rozhraní operačního systému pro přístup k souborům pomocí operací čtení a zápisu odděluje operace otevření a zavření souboru a operaci nastavení aktuální pozice v souboru od vlastních operací čtení a zápisu.
3. Popište obvyklé rozhraní operačního systému pro přístup k souborům pomocí operací mapování do paměti. Funkce rozhraní uveďte včetně argumentů a sémantiky.
4. Popište obvyklé rozhraní operačního systému pro práci s adresáři. Funkce rozhraní uveďte včetně argumentů a sémantiky.
5. Popište obvyklé rozhraní operačního systému pro zamykání souborů. Funkce rozhraní uveďte včetně argumentů a sémantiky.
6. Vysvětlete rozdíl mezi advisory a mandatory zámky pro zamykání souborů. Vysvětlete, proč tyto druhy zámků existují.

File Subsystem Internals

Disk Layout

Bunch of blocks. Tree. Log.

Handling of Files

Přechod z pásek na disky, první nápad se sekvenčním ukládáním souborů. Má to dvě výhody, totiž rychlost a malou režii. Nevýhodou je potřeba znát předem délku souboru a pochopitelně fragmentace.

První nápad jak tohle odstranit je udělat linked list. C64 měl tohle na floppy, nevýhody jsou zřejmé. Extrémně pomalý random access, velikost bloků není mocnina dvou, špatně se maže a tak.

Další modifikace je nechat linked list, ale vytáhnout ho z bloků a dát do tabulky. Typické řešení MS-DOSu. Nevýhodné to začne být když se celá tahle tabulka nevejde do paměti, lidi napadlo mít jí po kouskách u souborů, výsledek je třeba CP/M nebo UNIX. Co dělat když je tahle tabulka moc velká, CP/M přidává lineárně další bloky, UNIX stromově větví I-nodes.

Ukládání alokačních informací o souborech do adresářových položek, třeba alá CP/M, má ještě jednu značnou nevýhodu. Pokud totiž není možné oddělit jméno souboru od jeho alokační informace, není možné dělat hard linky.

Handling of Directories

Triviální případ jednoúrovňového adresáře, koncept ROOTu v MS-DOSu. Hierarchické adresáře, ukládání podadresářů do nadřazených adrešářů. DOSácká klasika, totéž u UNIXu. Jak do adresáře zadělat linky, koncept hard linku a symbolic linku.

Výhody a nevýhody hardlinku, rychlost, špatné mazání, nepřekročí hranici file systému. Symbolický link, totéž. Drobnost při zálohování a kopírování souborů, nutnost služby rozeznávající link od normálního souboru.

Handling of Free Space

Přidělování volného místa, problém velikosti bloků. Hlediska pro větší bloky, rychlost, malá režie, hlediska pro menší bloky, malá fragmentace.

Evidence volného místa, seznam volných bloků a bitmapy. Funkce bitmap je jasná, seznam volných bloků se zdá být nevýhodný, až na schopnost mít v paměti jen malou část tablice a přesto uspokojovat velký počet dotazů na volné bloky, a možnost ukládání právě ve volném místě. Možnost využití seznamu alá FAT. Evidence vadných bloků, vadný soubor, označení vadných bloků.

Diskové kvóty, mechanizmus hard a soft kvóty. Princip implementace, tablice otevřených souborů, tablice majitelů otevřených souborů.

Performance

Malá rychlost a malý počet bloků cache. Vhodná strategie závisí na aplikaci, úprava pro přednostní caching adresářů a I-nodes. Write-back caching, rozdělení místa mezi write-back a read cache.

Minimalizace pohybu hlavičky, umístění adresářů do středu disku, rozdělení velkého disku na segmenty. Alokace souborů do sousedních bloků, defragmentace.

Reliability

Požadavek spolehlivosti, nejjednodušším řešením je zálohování. Podpora pro zálohování, archivní atribut, detekce linků, snapshot. Zálohování na pásky, na disky, mirroring.

Konzistence systému, důležitost některých oblastí disku. Přednostní zápis adresářů, I-nodes, FAT, alokačních map a tak. Periodický sync. Kontrola konzistence při bootování systému. Unerase, nevýhody dolepovaných unerase, podpora v systému.

Example: FAT File System

Klasika, boot sektor s rozměry filesystému, za ním dvakrát FAT, za ní root directory, za ním data area. Adresářová položka obsahuje name, attributes, first cluster. Bad clusters mají extra známku ve FAT.

Nevýhody zahrnují nepohodlnou práci s FAT (je velká, nelze z ní snadno vytáhnout data týkající se jednoho souboru), nepohodlnou práci s adresáři (krátká jména souborů, málo atributů, špatné prohledávání, možnost fragmentace adresářů vymazanými jmény), režie na velké clustery.

Modifikace s rozšířením na větší čísla clusterů a delší jména souborů. Větší čísla jsou prostě tak, rozšířená na 32 bitů, žádný problém. Delší jména souborů jsou uložena do vhodných míst extra položek v adresáři, označených nesmyslnou kombinací atributů, v unicode. Hypotéza je, že to je kvůli kompatibilitě se staršími systémy, když se na diskety nahraje něco s dlouhým jménem.

Example: HPFS File System

Ačkoliv z OS/2, produkt Microsoftu. Citace z roku 1989 říká, že "HPFS solves all the problems of the FAT file system and is designed to meet the demands expected into the next few decades."

Na začátku disku je vyhrazeno 16 sektorů na bootstrap loader, následuje superblock s rozměry disku a pointery na bad block list, directory band, root directory a spareblock. Zbytek disku je rozdělen na 8 MB bands, každý band má free sectors bitmap a data area, které se střídají tak, aby bands sousedily buď bitmapami, nebo data areas.

Každý soubor je reprezentovaný strukturou F-node, která se přesně vejde do sektoru. Každý F-node obsahuje různé access control lists, attributes, usage history, last 15 chars of file name, plus an allocation structure. Allocation structure je buď 8 runs přímo v F-node, každý run je 32 bitů starting sector a 32 bitů number of sectors, nebo B+ strom o 12 větvích, jehož leaf nodes obsahují až 40 runs. Zajímavou věcí jsou extended attributes, u každého souboru se může uložit až 64 KB name/value párů, které jsou buď přímo v F-node, nebo v extra runu.

Adresáře jsou podobně jako soubory reprezentované strukturou F-node, pointer na F-node root directory je v superbloku. Adresář má položky různé délky ve 2 KB blocích uspořádaných jako B strom, položek se při jménech kolem 10 znaků vejde do 2 KB bloku tak 40. V každé položce je jméno, usage count, F-node pointer.

Jednou výhodou HPFS je alokační strategie, díky které jsou soubory ukládány pokud možno v souvislých blocích, a díky které je F-node blízko u dat souborů. Používá se prealokace po 4 KB, přebytečné bloky se vrací při zavření souboru. Samozřejmostí je read ahead a write back; dokumentace tvrdí, že se u souboru pamatuje usage pattern a podle něj se toto řídí.

Zajímavá je také fault tolerance. Systém si udržuje hotfix map, pokud narazí na chybu sektoru, tak jej do této mapy přidá a zobrazá warning, ve vhodné chvíli se pak soubory ležící v hotfixed sektorech přesunou jinam a hotfix se vyprázdní. Při power outage se podle dirty flagu ve spareblocku pozná, že vše není v pořádku, recovery pak může použít magic identifiers, které jsou přítomné ve všech zajímavých strukturách pro nalezení F-nodes a directories, které jsou navíc linked to each other.

Poznámka stranou, B strom je vyvážený strom s daty ve všech uzlech, B+ strom je vyvážený strom s daty pouze v listech. Jinak snad normální ...

Example: EXT2 And EXT3 And EXT4 File Systems

The filesystem uses the classical structure starting with a bootstrap area and continuing with blocks, each block containing a copy of the superblock, filesystem descriptors, free block bitmap, free inode bitmap, inode table fragent, and data area. These blocks serve the same role as bands or groups in other filesystems and should not be confused with equal sized blocks within the data area.

Free space is allocated by blocks. A free block bitmap is used to keep track of free blocks.

A file is represented by an inode. The inode contains information such as file type, access rights, owners, timestamps, size, and the information about where the data of the file resides, stored either as a block map or as an extent tree.

The block map contains pointers to direct blocks, a pointer to a single level indirect block, which contains pointers to direct blocks, a pointer to a double level indirect block, which contains pointers to single level indirect blocks, and a pointer to a triple level indirect block, which contains pointers to double level indirect blocks. By default, 12 pointers to direct blocks reside in the inode, 1024 pointers to indirect blocks reside in a block.

The extent tree, available starting with EXT4, describes consecutive sequences of blocks called extents. The extent tree has up to 4 entries in the inode, additional entries reside in potential interim blocks, both interim entries and leaves are 12 bytes long. This means that by default, the extent tree would need 5 levels to cover a hypothetical file with 2^32 extents.

Some versions of the filesystem could store file tails in block fragments. The inode structure therefore contains block fragment related fields, which, however, are not used in the current filesystem implementations.

```
struct ext2_inode {
  __u16  i_mode;   /* File mode */
  __u16  i_uid;                       /* Owner ID */
  __u32  i_size;                      /* Size in bytes */
  __u32  i_atime;                     /* Access time */
  __u32  i_ctime;                     /* Creation time */
  __u32  i_mtime;                     /* Modification time */
  __u32  i_dtime;                     /* Deletion Time */
  __u16  i_gid;                       /* Group ID */
  __u16  i_links_count;               /* Links count */
  __u32  i_blocks;                    /* Blocks count */
  __u32  i_flags;                     /* File flags */
  __u32  i_block [EXT2_N_BLOCKS];     /* Ptrs to blocks */
  __u32  i_version;                   /* File version for NFS */
  __u32  i_file_acl;                  /* File ACL */
  __u32  i_dir_acl;                   /* Directory ACL */
  __u32  i_faddr;                     /* Fragment address */
  __u8   l_i_frag;                    /* Fragment number */
  __u8   l_i_fsize;                   /* Fragment size */
};

#define EXT2_DIR_BLOCKS  12
#define EXT2_IND_BLOCK          EXT2_DIR_BLOCKS
#define EXT2_DIND_BLOCK  (EXT2_IND_BLOCK + 1)
#define EXT2_TIND_BLOCK  (EXT2_DIND_BLOCK + 1)
#define EXT2_N_BLOCKS           (EXT2_TIND_BLOCK + 1)

#define EXT2_SECRM_FL    0x00000001      /* Secure del */
#define EXT2_SYNC_FL     0x00000008      /* Sync update */
#define EXT2_IMMUTABLE_FL       0x00000010      /* Immutable */
#define EXT2_APPEND_FL          0x00000020      /* Only ap */
```

Directories are stored either unsorted, or with a tree index of file name hashes. When stored unsorted, a directory is simply an array of variable length entries. Entries never cross block boundary, unused space within blocks is marked with empty entries. When stored with a tree index, a directory starts with a tree root block, which points to potential interim blocks and eventually blocks with leaves, which are standard directory entries. All tree blocks start with a fake empty directory entry so that directory content remains backwards compatible.

```
struct ext2_dir_entry_2 {
  __u32  inode;              /* Inode number */
  __u16  rec_len;                      /* Directory entry length */
  __u8   name_len;                     /* Name length */
  __u8   file_type;   /* File type */
  char   name [EXT2_NAME_LEN];    /* File name */
};

#define EXT2_NAME_LEN 255

#define EXT2_FT_REG_FILE        1
#define EXT2_FT_DIR             2
#define EXT2_FT_CHRDEV   3
#define EXT2_FT_BLKDEV   4
#define EXT2_FT_SYMLINK  7
```

A quick overview of other features includes a bad block map kept in reserved inode 1, an administrator space reservation.

Pro odhad, v Linuxu je na 8GB partition celkem 1M I-nodes, z toho jsou pro běžné soubory tak 4% použitých, každý blok má 130MB. Extra atributy se dají číst a měnit přes lsattr a chattr, patří mezi ně IMM (immutable), APP (append only), SYNC (synchronous update), COMPRESS, UNDELETE, SAFEDEL (tyto tři ale kernel ignoruje ?).

Journalling mode for data is either writeback, ordered, or journal. Writeback means data are not journalled. Ordered means data are written to normal location before corresponding metadata are journalled. Journal means both data and metadata are journalled. Journalling is done to a special file.

References

1. Tweedie S. C.: Journaling the Linux ext2fs Filesystem

Example: NTFS File System

Na začátku disku je jen bootsektor s rozměry disku a pointerem na MFT, jeho kopie je uložena ještě kdesi na (konci ?) partition. Celý zbytek disku se adresuje po clusterech, které jsou podobně jako u FAT mocninné násobky sektorů. Na disku neleží nic než soubory, informace o nich jsou uloženy v MFT aneb Master File Table. Každý soubor je jednoznačně identifikován indexem do MFT, MFT sama je také soubor s indexem 0, další význačné indexy jsou 1 pro MFT mirror, 2 pro transaction log, 3 pro root directory, 4 pro allocationj bitmap, 5 pro bootstrap, 6 for bad cluster file atd.

Některé ze skrytých souborů lze vypsat, chodí příkazy dir /ah $bitmap, dir /ah $badclus, dir /ah $mftmirr atd. (ovšem krom vypsání v root adresáři už tuším nic nejde). Ve Windows 2000 už zdá se nejde ani tohle.

Každý soubor je set of attributes, jeden z atributů je default, to je data stream. Každý záznam v MFT obsahuje magic, update sequence (podobná NFS, je potřeba aby při reuse záznamu bylo možné poznat staré reference), reference count, flags, potenciálně pointer na base file record pokud toto je extension file record (když se vše nevejde do base file recordu). Následují file attributes, u nich záznam obsahuje jméno, typ a data, data mohou být buď resident, v tom případě následují přímo v záznamu, nebo non-resident, v tom případě následuje v záznamu run list, což je sekvence bloků clusterů podobně jako u HPFS.

Adresáře jsou uložené jako soubory, jejichž obsah je B strom s referencemi na obsažené soubory.

Mírně zjednodušeno. Největší nevýhodou se zdá být fragmentace. To prevent fragmentation of MFT, NTFS makes sure a free area called MFT zone exists after MFT. Each time the disk becomes full, MFT zone is halved.

Multiple Streams

Jednotlivé streams v souboru jsou označené jmény a lze k nim přistupovat otevřením souboru se jménem file:stream_name:$stream_type. Default stream se nijak nejmenuje a typ má data, takže file a file::$data je totéž (to se používalo pro útok na Microsoft Information Server, který u jmen s explicitně uvedeným streamem nepoznal příponu, takže člověk mohl číst zdrojáky skriptů). Legračně délka souboru odráží pouze default stream, takže data v dalších streamech zabírají místo na disku, ale v adresářích nejsou vidět (také se leckdy nekopírují, Explorer OK, ale FAR ne).

Některé konkrétní atributy, krom $DATA nepřístupné aplikaci:

Atribut	**Obsah**
$VOLUME_VERSION	Volume version
$VOLUME_NAME	Disk's volume name
$VOLUME_INFORMATION	NTFS version and dirty flag
$FILE_NAME	File or directory name
$STANDARD_INFORMATION	File time stamps and hidden, system, and read-only flags
$SECURITY_DESCRIPTOR	Security information
$DATA	File data
$INDEX_ROOT	Directory content
$INDEX_ALLOCATION	Directory content
$BITMAP	Directory content mapping
$ATTRIBUTE_LIST	Describes nonresident attribute headers

Ještě pozoruhodněji, Windows dlouhou dobu neměly pro práci se streams žádné API, tedy nešel snadno vypsat seznam streams apod. Řešení nabízela funkce BackupRead, která ze souboru vyrobí speciální backup stream, určený pro zálohování. Tento stream obsahuje data potřebná pro kompletní rekonstrukci soubor, tedy i streams a jeho formát je známý. Zdá se, že i ACL jsou uložené jako stream (?).

```
HANDLE FindFirstStreamW (
  LPCWSTR lpFileName,
  STREAM_INFO_LEVELS InfoLevel,
  LPVOID lpFindStreamData,
  DWORD dwFlags);
BOOL FindNextStreamW (
  HANDLE hFindStream,
  LPVOID lpFindStreamData);

typedef enum _STREAM_INFO_LEVELS {
  FindStreamInfoStandard
} STREAM_INFO_LEVELS;

typedef struct _WIN32_FIND_STREAM_DATA {
  LARGE_INTEGER StreamSize;
  WCHAR cStreamName [MAX_PATH + 36];
} WIN32_FIND_STREAM_DATA;
```

The only thing worth noting on the stream enumeration interface is probably the inconsistent use of system constants suggested by the need to add an arbitrary constant to MAX_PATH.

Cache Manager

Quoted from [Mark Russinovich, David Solomon: Windows XP: Kernel Improvements Create a More Robust, Powerful, and Scalable OS]

In order to know what it should prefetch, the Windows XP Cache Manager monitors the page faults, both those that require that data be read from disk (hard faults) and those that simply require data already in memory be added to a working set (soft faults), that occur during the boot process and application startup. By default, it records 120 seconds of the boot process, 60 seconds after all services have finished initializing, or 30 seconds after the shell starts, whichever occurs first. The Cache Manager also monitors the first 10 seconds of application startup.

After collecting a trace organized into faults taken on MFT (if the application accesses files or directories), the files referenced, and the directories referenced, the Cache Manager notifies the prefetch component of the Task Scheduler that performs a call to the internal NtQuerySystemInformation system call requesting the trace data. After performing post-processing on the trace data, the Task Scheduler writes it out to a file in the /Windows/Prefetch folder. The file's name is the name of the application to which the trace applies followed by a dash and the hexadecimal representation of a hash of the file's path. The file has a .pf extension, so an example would be NOTEPAD.EXE-AF43252301.PF. An exception to the file name rule is the file that stores the boot's trace, which is always named NTOSBOOT-B00DFAAD.PF (a convolution of the hexadecimal-compatible word BAADF00D, which programmers often use to represent uninitialized data).

When the system boots or an application starts, the Cache Manager is called to give it an opportunity to perform prefetching. The Cache Manager looks in the prefetch directory to see if a trace file exists for the prefetch scenario in question. If it does, the Cache Manager calls NTFS to prefetch any MFT references, reads in the contents of each of the directories referenced, and finally opens each file referenced. It then calls the Memory Manager to read in any data and code specified in the trace that's not already in memory. The Memory Manager initiates all of the reads asynchronously and then waits for them to complete before letting an application's startup continue.

Backup Support

Quoted from [Mark Russinovich, David Solomon: Windows XP: Kernel Improvements Create a More Robust, Powerful, and Scalable OS]

A new facility in Windows XP, called volume shadow copy, allows the built-in backup utility to record consistent views of all files, including open ones. The shadow copy driver is a type of driver, called a storage filter driver, that layers between file system drivers and volume drivers (the drivers that present views of the disk sectors that represent a logical drive) so that it can see the I/O directed at a volume.

When the backup utility starts a backup operation it directs the volume shadow copy driver (/Windows/System32/Drivers/Volsnap.sys) to create a volume shadow copy for the volumes that include files and directories being recorded. The volume shadow copy driver freezes I/O to the volumes in question and creates a shadow volume for each. For example, if a volume's name in the Object Manager namespace is /Device/HarddiskVolume0, the shadow volume might be named /Device/HarddiskVolumeShadowCopyN, where N is a unique ID.

Instead of opening files to back up on the original volume, the backup utility opens them on the shadow volume. A shadow volume represents a point-in-time view of a volume, so whenever the volume shadow copy driver sees a write operation directed at an original volume, it reads a copy of the sectors that will be overwritten into a paging file-backed memory section that's associated with the corresponding shadow volume. It services read operations directed at the shadow volume of modified sectors from this memory section, and services reads to non-modified areas by reading from the original volume.

Because the backup utility won't save the paging file or the contents of the system-managed /System Volume Information directory located on every volume, the snapshot driver uses the defragmentation API to determine the location of these files and directories, and does not record changes to them.

By relying on the shadow copy facility, the Windows XP backup utility overcomes both of the backup problems related to open files.

The shadow copy driver is actually only an example of a shadow copy provider that plugs into the shadow copy service (/Windows/System32/Vssvc.exe). The shadow copy service acts as the command center of an extensible backup core that enables ISVs to plug in writers and providers. A writer is a software component that enables shadow copy-aware applications to receive freeze and thaw notifications in order to ensure that backup copies of their data files are internally consistent, whereas providers allow ISVs with unique storage schemes to integrate with the shadow copy service. For instance, an ISV with mirrored storage devices might define a shadow copy as the frozen half of a split-mirrored volume.

Summary

References

1. Russinovich M.: Inside NTFS
2. Wijk J. v.: NTFS Disk Structure Definitions

Example: XFS File System

XFS has been designed by SGI and provides support for large numbers of large files in large directories accessed by large numbers of clients. This is achieved by using balanced trees for most structures and by providing metadata logging.

XFS divides the disk into groups, each group contains metadata and data areas. The group metadata area contains a copy of the superblock, pointers to roots of two group free block trees, a pointer to the root of a group inode tree, and a reserved group free block list. The data area is split into equal sized blocks. Most references used by XFS come in two flavors, a relative reference within a group and an absolute reference, which is created by prepending the group identifier to the relative reference.

Free space is allocated by blocks. The two free block trees allow locating free space by block number and by extent size, each leaf of a tree points to a free extent. The reserved free block list contains free blocks reserved for growing the free block trees.

Files are represented by inodes. The inode tree allows locating an inode by inode number, each leaf of the tree points to a block with a sparse array of 64 inodes. An inode contains basic information about a file and points to file data and extended attributes using structures called a data fork and an attribute fork. Depending on the size of the data referenced by the fork, the data is stored:

- directly within the fork inode. The default size of an inode is 256 bytes, out of which 100 bytes are used by the basic information, leaving 156 bytes for the forks.
- in extents listed within the fork inode. The default size of an inode provides enough space for up to 19 extents.
- data in a tree. When the file data is stored in a tree, the keys of the tree are offsets within the file and the leaves of the tree are extents.

Directories use either short form, block form, leaf form, node form, or tree form, depending on their size. All forms have a variable length entry containing the file name and the file inode.

- A short form directory stores entries directly within its inode.
- A block form directory stores entries in a single extent, which also contains a sorted array of file name hashes and an array of a few largest free entries in the extent.
- A leaf form directory stores entries in multiple entry extents and single extent with a sorted array of file name hashes and an array of a few largest free entries in the entry extents.
- A node form directory stores entries in multiple entry extents, a tree of file name hashes and an extent with an array of a few largest free entries in the entry extents.
- Finally, a tree form directory uses trees to store the array of entries, the file name hashes, and the array of a few largest free entries in the entry extents.

Attributes use either short form, leaf form, node form, or tree form, depending on their size. The forms of attribute storage are similar to the forms of directory storage, except that the names and values of attributes are kept together with name hashes, while the entries were kept separate from name hashes.

Metadata modifications are journalled.

References

1. SGI: XFS Filesystem Structure. http://oss.sgi.com/projects/xfs/papers/xfs_filesystem_structure.pd
2. SGI: XFS Overview and Internals. http://oss.sgi.com/projects/xfs/training/index.html

Example: CD File System

Standard ISO9660 a ECMA119. Disk rozdělen na sektory zpravidla 2048 bytes, prvních 16 sektorů prázdných pro bootstrap loader. Zbytek disku je popsán sekvencí volume descriptors, jeden per sektor, nejdůležitější je Primary Volume Descriptor s adresou root directory, path table a dalšími zbytečnostmi (copyright, abstract, bibinfo). Adresáře jsou usual stuff, name of 30 chars max, attributes, soubor je udán počátečním sektorem a délkou (teoreticky je možné uvést víc adresářových položek pro soubor z více fragmentů, ale nepoužívá se, stejně jako se zdá se nepoužívá interleaving).

Jedním zajímavým detailem v ISO9660 jsou path tables. Aby se adresáře nemusely prohledávat item by item, uloží se do path table seřazený (podle hloubky a dalších kritérií) seznam všech cest na disku, pro každou cestu obsahuje path table sektor příslušného adresáře a jeho parenta.

The standard imposes a number of weird limits on the file system structure, such as maximum directory nesting depth of 8, only capital letters, digits and underscores in file names, no extensions in directory names, etc. For this reason, extensions such as Joliet and Rock Ridge have appeared.

References

1. Erdelsky P. J.: ISO9660 Simplified For DOS/Windows

Example: UDF File System

Standard ISO13346 a UDF a ECMA167. Základní principy podobné ISO9660 a ECMA116.

Zajímavý je koncept Prevailing Descriptors pro připisovatelná média. Každý deskriptor má u sebe verzi, či lépe pořadové číslo, a pokud se v seznamu deskriptorů najde více deskriptorů téhož typu, uvažuje se ten s nejvyšší verzí. Protože seznam deskriptorů není (nemusí být) ukončený, lze na jeho konec připisovat nové deskriptory, které nahradí staré. S tím souvisí ještě koncept Virtual Allocation Table, která mapuje logické na fyzické sektory disku. Všechny údaje na disku jsou v logických sektorech, když je potřeba například přepsat část souboru, mohou se adresáře i zbytek souboru nechat tam kde jsou a jen se upraví mapování.

Example: JFFS2 File System

JFFS2 is a journalling file system that accommodates the specific nature of the flash memory devices, which are organized in blocks that need to be erased before writing.

The file system views the entire flash memory device as a log consisting of arbitrarily arranged blocks. The log contains records called nodes, nodes fill up blocks but may not span block boundaries so that independent garbage collection of blocks remains possible. There are three types of nodes:

- An inode node, which contains metadata and optionally a fragment of data belonging to a file. Compression of the fragments is supported, a fragment should generally fit a memory page on the host.
- A dirent node, which contains the inode number of the directory that the entry belongs to, and the name and inode number of the file that the entry describes.
- A cleanmarker node, which marks a successfully erased block.

All the inode and dirent nodes contain a version number. An update of a node is done by writing a new version of the node at the tail of the log. When the file system is mounted, the blocks of the log are scanned (not necessarily from head to tail, due to independent garbage collection of blocks), creating an overview of the latest versions of all nodes.

Garbage collection frees space for the tail of the log by picking a random block and copying whatever of its content is not outdated to the tail of the log. Statistical preference is given to blocks with at least some outdated content, so that proper balance between precise wear levelling and increased wear associated with copying is maintained.

References

1. David Woodhouse: JFFS: The Journalling Flash File System. http://sources.redhat.com/jffs2/jffs2.pdf

Example: Spiralog File System

Tohle je zajímavý systém od Digitalu, založený na log structure.

The file system consists of multiple servers and clerks. Clerks run near client applications and are responsible for caching and cache coherency and ordered write back. Servers run near disks and are reponsible for carrying out idempotent atomic sets of operations on behalf of clerks. Disks can be attached to multiple servers but only one of those servers accesses the disks, one of the remaining servers is chosen to access the disks if the current server fails.

Clerks present clients with files that can have user defined attributes and multiple data streams. Servers store files in an infinite log.

At top level, server handles objects with unique identification, with multiple named cells for storing data accessed in one piece, and with multiple numbered streams for storing data accessed in multiple pieces. Files are mapped onto objects with attributes in cells and contents in streams. Directories are mapped onto objects with attributes and entries in cells.

At medium level, server handles infinite log. Objects are mapped into B tree stored in the log, the keys are object identifiers, the leaves are objects. Cells and streams are mapped into B trees of a single leaf of the object B tree. When cells are stored in a B tree, the keys are names of the cells and the leaves denote cell data. When a stream is stored in a B tree, the keys are positions in the stream and the leaves denote stream extents. Optimizations that store short extents within their leaves also apply.

At bottom level, server handles segments. Segments are blocks of consecutive sectors 256 kB long. A segment consists of a data area and a commit record area that are written in two physical phases for each logical write. Log is mapped into segments using a segment array. Cleaner process compacts the old segments by copying. Checkpointing process keeps the number of B tree change records that have to be applied during tree reconstruction down to a reasonable limit.

References

1. Johnson J.E., Laing W.A.: Overview of the Spiralog File System, Digital Technical Journal 8(2), DEC, 1996
2. Whitaker C., Bayley J., Widdowson R.: Design of the Server for the Spiralog File System, Digital Technical Journal 8(2), DEC, 1996

Example: Reiser File System

Další méně obvyklý systém, chodí pod Linuxem a zaměruje se na efektivitu při práci s velkým množstvím malých souborů. Problém s malými soubory je overhead při alokaci, který je tady řešený tak, že se na celý disk pohlíží jako na jeden B* strom. Uzly tohoto stromu jsou v blocích, které jsou násobky velikosti sektoru, uzel je buď nepřímý, pak obsahuje pouze klíče a pointery na potomky, nebo přímý formátovaný, pak obsahuje seznam prvků uložený tak, že od začátku uzlu narůstají hlavičky (directory item, indirect data, direct data) a od konce těla prvků, nebo přímý neformátovaný, pak obsahuje data velkého souboru do násobku velikosti bloku.

Celý tenhle cirkus zaručuje, že se malé soubory budou ukládat pohromadě do jednoho bloku, čímž se spoří místo. To, kam přesně se co uloží, je dané klíčem. Klíče jsou proto udělané tak, že obsahují vždy parent object ID, local object ID, offset, uniqueness, čímž se zaručuje, že všechny objekty budou pohromadě u svého parenta (například directory entries z jednoho directory pohromadě). Bloky jsou alokované near each other, evidují se v bitmapě, bitmapy jsou rozmístěny mezi datovými bloky, vždy jeden blok bitmapa a pak tolik bloků data, kolik se dá popsat v jednom bloku bitmapy.

Ještě zajímavá je konzistence. Problémem u takto složitého file systému je situace, kdy se kvůli vyvážení stromu musí přepisovat již existující struktury. Pokud v tu chvíli systém spadne, hrozí poškození starých dat. Původní verze file systému toto řešily tak, že zavedly uspořádání na všech zápisech na disk tak, aby zápisy dat v nových pozicích předcházaly smazání dat ve starých pozicích. To bylo ale závěrem příliš složité, takže teď se při odebrání položky změní pozice bloku tak, aby se nepřepisovala stará verze (hledá se nejbližší volné místo) a stará verze se uloží do preserve listu. Preserve list se vyprázdní, když v paměti nejsou žádné bloky, do kterých se přidávaly položky. Ještě novější verze mají log.

Krom zjevné úspornosti má také problémy, jedna je s rychlostí u souborů, které jsou mírně menší než bloky, protože ty se ukládají jako dva direct items. Druhá je u preserve listu při velkém počtu malých souborů, je potřeba často flushovat aby se mohl vyprázdnit. Třetí jsou problémy s memory mapped files když soubor není aligned. Plus samozřejmě kdo to má psát, celý EXT2 má pod 200K zdrojáků, ReiserFS má včetně patchů kernelu víc než mega.

References

1. ReiserFS Whitepaper
2. Kurz G.: The ReiserFS Filesystem
3. Buchholz F.: The Structure of the ReiserFS Filesystem

Example: BTRFS File System

To be done.

Integration Of File Subsystem With Memory Management

Caches.

Integration Of Multiple File Subsystems

Zmínit integraci více file systémů do kernelu, princip mount points pro poskytnutí jednoho prostoru jmen. Stackable file systems přes V-nodes.

Example: Linux Virtual File System

Provedení v Linuxu je přímočaré. Při volání open se systém podívá na začátek jména souboru a podle toho zda je absolutní či relativní vezme dentry buď root directory nebo current directory. Pak už se jen postupně parsuje jméno a každá jeho část se zkusí najít v dentry cache, pokud tam není, tak se použije lookup funkce parent dentry.

Do tohoto mechanizmu celkem přímočaře zapadá i mounting. Pokud se do adresáře něco namountuje, jeho dentry bude obsahovat pointer na root dentry namountovaného file systému. Tento dentry zůstane díky busy locku vždy v dentry cache. Při parsování cesty se pak u každého dentry ještě kontroluje, zda nemá mounted file systém, pokud ano, vezme se jeho root dentry.

Example: Linux Union File System

Stackable filesystems. Whiteout files.

Rehearsal

Questions

1. Vysvětlete hlediska ovlivňující volbu velikosti bloků jako alokačních jednotek na disku.
2. Uveďte, jakými způsoby lze na disku ukládat informaci o blocích, ve kterých jsou umístěna data souborů. Jednotlivé způsoby ilustrujte na existujících systémech souborů a zhodnoťte.
3. Uveďte, jakými způsoby lze na disku ukládat strukturu adresářů. Jednotlivé způsoby ilustrujte na existujících systémech souborů a zhodnoťte.
4. Vysvětlete rozdíl mezi hard linkem a symbolic linkem. Porovnejte výhody a nevýhody obou typů linků.
5. Uveďte, jakými způsoby lze na disku ukládat informaci o volných blocích. Jednotlivé způsoby ilustrujte na existujících systémech souborů a zhodnoťte.
6. Popište způsob uložení informace o umístění dat souborů v systému souborů FAT. Uveďte přednosti a nedostatky tohoto způsobu uložení informace.
7. Popište způsob uložení informace o struktuře adresářů v systému souborů FAT. Uveďte přednosti a nedostatky tohoto způsobu uložení informace.
8. Popište způsob uložení informace o umístění volných bloků v systému souborů FAT. Uveďte přednosti a nedostatky tohoto způsobu uložení informace.
9. Popište způsob uložení informace o umístění dat souborů v systému souborů EXT2. Uveďte přednosti a nedostatky tohoto způsobu uložení informace.
10. Popište způsob uložení informace o struktuře adresářů v systému souborů EXT2. Uveďte přednosti a nedostatky tohoto způsobu uložení informace.
11. Popište způsob uložení informace o umístění volných bloků v systému souborů EXT2. Uveďte přednosti a nedostatky tohoto způsobu uložení informace.
12. Popište způsob uložení informace o umístění dat souborů v systému souborů NTFS. Uveďte přednosti a nedostatky tohoto způsobu uložení informace.
13. Popište způsob uložení informace o struktuře adresářů v systému souborů NTFS. Uveďte přednosti a nedostatky tohoto způsobu uložení informace.
14. Popište způsob uložení informace o umístění dat souborů v systému souborů na CD. Uveďte přednosti a nedostatky tohoto způsobu uložení informace.
15. Popište způsob uložení informace o struktuře adresářů v systému souborů na CD. Uveďte přednosti a nedostatky tohoto způsobu uložení informace.
16. Vysvětlete princip integrace více systémů souborů v operačním systému do jednoho prostoru jmen.

Exercises

1. Popište strukturu systému souborů FAT na disku. Ilustrujte použití této struktury v operacích čtení dat ze souboru při zadané cestě a jménu souboru a pozici

a délce dat v souboru a zápisu dat do nově vytvořeného souboru při zadané cestě a jménu souboru a délce dat. Uveďte přednosti a nedostatky tohoto systému souborů.

2. Popište strukturu systému souborů EXT2 na disku. Ilustrujte použití této struktury v operacích čtení dat ze souboru při zadané cestě a jménu souboru a pozici a délce dat v souboru a zápisu dat do nově vytvořeného souboru při zadané cestě a jménu souboru a délce dat. Uveďte přednosti a nedostatky tohoto systému souborů.
3. Navrhněte systém souborů, který je schopen efektivně podporovat neomezeně dlouhá jména souborů a linky. Popište strukturu dat ukládaných na disk a algoritmy přečtení a zapsání dat z a do souboru daného jménem včetně cesty a pozicí v rámci souboru. Vysvětlete přednosti vašeho návrhu.
4. Navrhněte systém souborů, který je schopen efektivně podporovat velmi krátké i velmi dlouhé soubory. Popište strukturu dat ukládaných na disk a algoritmy přečtení a zapsání dat z a do souboru daného jménem včetně cesty a pozicí v rámci souboru. Vysvětlete přednosti vašeho návrhu.

Chapter 6. Network Subsystem

Podpora sítí se dá zhruba rozdělit do dvou částí. První částí je pouhé zpřístupnění sítě aplikacím kvůli přenosu dat, druhou částí je vystavění nějakých zajímavých mechanizmů nad vlastním přenosem dat.

Abstractions And Operations

Sockets

The most traditional interface of the network subsystem is the Berkeley socket interface. Historically, the Berkeley socket interface was developed at the University of California at Berkeley as a part of BSD 4.2 from 1981 to 1983. These days, it is present in virtually all flavors of Unix and Windows.

The Berkeley socket interface centers around the concept of a socket as an object that facilitates communication. The socket can be bound to a local address and connected to a remote address. Data can be sent and received over a socket.

```
int socket (int domain, int type, int protocol);
```

Domain specifies socket protocol class:

- PF_UNIX - local communication
- PF_INET - IPv4 protocol family
- PF_INET6 - IPv6 protocol family
- PF_IPX - IPX protocol family
- PF_NETLINK - kernel communication
- PF_PACKET - raw packet communication

Type specifies socket semantics:

- SOCK_STREAM - reliable bidirectional ordered stream
- SOCK_RDM - reliable bidirectional unordered messages
- SOCK_DGRAM - unreliable bidirectional unordered messages
- SOCK_SEQPACKET - reliable bidirectional ordered messages
- SOCK_RAW - raw packets

Protocol specifies socket protocol:

- 0 - class and type determine protocol
- other - identification of supported protocol

The `socket` call creates the socket object. An error is returned if the combination of class, type, protocol is not supported.

```
int bind (int sockfd, struct sockaddr *my_addr, socklen_t addrlen);
```

```
#define __SOCKADDR_COMMON(sa_prefix) \
  sa_family_t sa_prefix##family

struct sockaddr_in
{
  __SOCKADDR_COMMON (sin_);
  in_port_t      sin_port;
  struct in_addr sin_addr;
  unsigned char  sin_zero [sizeof (struct sockaddr) -
                           __SOCKADDR_COMMON_SIZE -
                           sizeof (in_port_t) -
                           sizeof (struct in_addr)];
};

struct sockaddr_in6
{
  __SOCKADDR_COMMON (sin6_);
  in_port_t       sin6_port;
  uint32_t        sin6_flowinfo;
  struct in6_addr sin6_addr;
  uint32_t        sin6_scope_id;
};
```

The `bind` call binds the socket to a given local address. The binding is typically necessary to tell the socket what local address to listen on for incoming connections.

```
int listen (int sockfd, int backlog);
```

The `listen` call tells the socket to listen for incoming connections and sets the length of the incoming connection queue.

```
int accept (int sockfd, struct sockaddr *addr, socklen_t *addrlen);
```

The `accept` call accepts an incoming connection on a listening socket that is `SOCK_SEQPACKET`, `SOCK_STREAM` or `SOCK_RDM`. The function returns a new socket and an address that the new socket is connected to and keeps the original socket untouched.

```
int connect (int sockfd,
             const struct sockaddr *serv_addr,
             socklen_t addrlen);
```

The `connect` call connects a socket that is `SOCK_SEQPACKET`, `SOCK_STREAM` or `SOCK_RDM` to a remote address. For other socket types, it sets a remote address of the socket.

```
ssize_t send (int sockfd, const void *buf, size_t len, int flags);
ssize_t sendto (int sockfd, const void *buf, size_t len, int flags,
                const struct sockaddr *to, socklen_t tolen);
ssize_t sendmsg (int sockfd, const struct msghdr *msg, int flags);

struct msghdr
{
  void         *msg_name;       // optional address
  socklen_t    msg_namelen;     // optional address length
  struct iovec *msg_iov;        // array for scatter gather
  size_t       msg_iovlen;      // array for scatter gather length
  void         *msg_control;    // additional control data
  socklen_t    msg_controllen;  // additional control data length
  int          msg_flags;
};
```

The `send` family of calls sends data over a socket. Either the socket is connected or the remote address is specified. The `write` call can also be used but the flags cannot be specified in that case.

```
ssize_t recv (int sockfd, void *buf, size_t len, int flags);
ssize_t recvfrom (int sockfd, void *buf, size_t len, int flags,
                  struct sockaddr *from, socklen_t *fromlen);
ssize_t recvmsg (int sockfd, struct msghdr *msg, int flags);

struct msghdr
{
  void         *msg_name;       // optional address
  socklen_t    msg_namelen;     // optional address length
  struct iovec *msg_iov;        // array for scatter gather
  size_t       msg_iovlen;      // array for scatter gather length
  void         *msg_control;    // additional control data
  socklen_t    msg_controllen;  // additional control data length
  int          msg_flags;
};
```

The `recv` family of calls receives data over a socket. The `read` call can also be used but the flags cannot be specified in that case.

The additional control data can provide data such as list of queued errors or additional protocol and transport information. The additional control data is structured as a list with headers and payload, which is protocol specific.

```
int select (int setsize,
            fd_set *readfds,
            fd_set *writefds,
            fd_set *exceptfds,
            struct timeval *timeout);

int poll (struct pollfd *ufds,
          unsigned int nfds,
          int timeout);

struct pollfd
{
  int fd;
  short events;         // requested events
  short revents;        // returned events
};
```

The `select` call is used to wait for data on several sockets at the same time. The arguments are sets of file descriptors, usually implemented as bitmaps. The file descriptors in *`readfds`* are waited for until a read would not block, the file descriptors in *`writefds`* are waited for until a write would not block, the file descriptors in *`exceptfds`* are waited for until an exceptional condition occurs. The call returns the number of file descriptors that meet the condition of the wait.

The `poll` call makes it possible to more precisely distinguish what events to wait for.

```
int getsockopt (int sockfd, int level,
                int optname, void *optval, socklen_t *optlen);

int setsockopt (int sockfd, int level,
                int optname, const void *optval, socklen_t optlen);
```

References

1. Hewlett Packard: BSD Sockets Interface Programmers Guide

Example: Unix Sockets

Unix sockets represent a class of sockets used for local communication between processes. The sockets are represented by a file name or an abstract socket name.

```
struct sockaddr_un
{
  sa_family_t  sun_family;              // set to AF_UNIX
  char         sun_path [PATH_MAX];     // socket name
};
```

It is also possible to use sockets without names, the `socketpair` function creates a pair of connected sockets that can be inherited by child processes and used for communication.

```
int socketpair (int domain,
                int type,
                int protocol,
                int sockets [2]);
```

Unix sockets can use additional control data to send file descriptors or to send process credentials (PID, UID, GID) whose correctness is verified by kernel.

Important uses of the Unix sockets include the X protocol.

```
> netstat --unix --all (servers and established)
Proto RefCnt Flags    Type   State     Path
unix  2      [ ACC ] STREAM LISTENING /var/run/acpid.socket
unix  2      [ ACC ] STREAM LISTENING /tmp/.font-unix/fs7100
unix  2      [ ACC ] STREAM LISTENING /tmp/.gdm_socket
unix  2      [ ACC ] STREAM LISTENING /tmp/.X11-unix/X0
unix  2      [ ACC ] STREAM LISTENING /tmp/.ICE-unix/4088
unix  2      [ ACC ] STREAM LISTENING /var/run/dbus/system_bus_socket
unix  3      [ ]     STREAM CONNECTED /var/run/dbus/system_bus_socket
unix  2      [ ]     DGRAM            @/var/run/hal/hotplug_socket
unix  2      [ ]     DGRAM            @udevd
unix  2      [ ACC ] STREAM LISTENING /tmp/xmms_ceres.0
unix  3      [ ]     STREAM CONNECTED /tmp/.X11-unix/X0
unix  3      [ ]     STREAM CONNECTED /tmp/.ICE-unix/4088
```

Example: Linux Netlink Sockets

Netlink sockets represent a class of sockets used for communication between processes and kernel. The sockets are represented by a netlink family that is specified in place of protocol when creating the socket.

- NETLINK_ARPD - ARP table
- NETLINK_ROUTE - routing updates and modifications of IPv4 routing table
- NETLINK_ROUTE6 - routing updates and modifications of IPv6 routing table
- NETLINK_FIREWALL - IPv4 firewall
- ...

Messages sent over the netlink socket have a standardized format. Macros and libraries are provided for handling messages of specific netlink families.

Example: Windows Winsock Sockets

From the application programmer perspective, Winsock sockets offer an interface that is, in principle, based on that of the Berkeley sockets. From the service programmer perspective, Winsock offers an interface that allows service providers to install multiple protocol libraries underneath the unified API. The interface, called SPI (Service Provider Interface), distinguishes two types of services, transport and naming, and allows layering of protocol libraries.

Remote Procedure Call

This is described in detail in the Middleware materials.

Rehearsal

Questions

1. Popište *socket* jako abstrakci rozhraní operačního systému pro přístup k síti podle Berkeley sockets. Uveďte základní funkce tohoto rozhraní včetně hlavních argumentů a sémantiky.
2. Vysvětlete účel funkce `select` v rozhraní operačního systému pro přístup k síti podle Berkeley sockets.
3. Vysvětlete, k čemu slouží sockety v doméně `PF_UNIX` .
4. Vysvětlete, k čemu slouží sockety v doméně `PF_NETLINK` .
5. Popište princip funkce mechanismu vzdáleného volání procedur a načrtněte obvyklou architekturu jeho implementace.

Network Subsystem Internals

Queuing Architecture

The architecture of the network subsystem typically follows the architecture of the protocols used by the network subsystem. At the lowest level, the device drivers provide access to the network interfaces. At the highest level, the socket module implements the Berkeley socket interface. In between, the protocol modules implement the ARP, IP, UDP, TCP and other protocols. The modules typically communicate through queues of packets.

As described, the architecture has two pitfalls, both related to a potential loss of efficiency when a large number of modules processes packets.

The first pitfall is caused by excessive data copying. The individual modules that process packets may need to add headers or footers to the data, which may prompt a need for moving the data to make room for the headers or footers. With top desktop systems moving data in memory in hundreds to thousands of MB per second and top network systems moving data in wires in thousands of MB per second, even a small amount of data copying may be a problem.

The second pitfall is caused by excessive data dispatching. Many solutions exist, the traditional ones including hash tables, the wilder ones ranging from dispatcher shortcut caching to dispatcher code generation and dispatcher code upload.

Both pitfalls can be sidestepped by using smart hardware.

Example: Linux SK Buff Structure

To avoid data copying, the individual modules that process packets keep data in the sk_buff structure. The structure reserves space before and after data so that headers or footers can be added without data copying.

```
struct sk_buff *alloc_skb (unsigned int size, int priority);
void skb_reserve (struct sk_buff *skb, unsigned int len);
int skb_headroom (const struct sk_buff *skb);
int skb_tailroom (const struct sk_buff *skb);

unsigned char *skb_put (struct sk_buff *skb, unsigned int len);
unsigned char *skb_push (struct sk_buff *skb, unsigned int len);

unsigned char *skb_pull (struct sk_buff *skb, unsigned int len);
void skb_trim (struct sk_buff *skb, unsigned int len);
```

References

1. Alan Cox: Network Buffers and Memory Management

Packet Filtering

The networking layer must decide what to do with each packet. A packet can be delivered to a local recipient, forwarded to a remote recipient, or even dropped. This mechanism is configurable to avoid abuse of default rules for delivering, forwarding, discarding.

Example: Linux Packet Filter

The packet filter framework defines several points where a packet can be classified and a decision can be taken based upon the classification. The points are identified by chains that are grouped into tables.

The `filter` table is for normal packets:

- INPUT - chain for incoming packets
- OUTPUT - chain for outgoing packets
- FORWARD - chain for packets that pass through

The `nat` table is for packets that open new connections:

- PREROUTING
- OUTPUT
- POSTROUTING

The `mangle` table is for packets that need special modifications:

- PREROUTING

- INPUT
- OUTPUT
- FORWARD
- POSTROUTING

Each point contains a sequence of rules. A rule can classify packets using information from packet header (source and destination address, protocol ...) or from packet processing (source and destination interface ...). Modules that classify packets can be added, available modules include file conditions, connection marks, connection rates, connection state, security context, random and others.

The action of the first matching rule is used. An action is either a chain name or ACCEPT, DROP, QUEUE, RETURN. ACCEPT means process packet, DROP means discard, QUEUE means queue for user space application to decide, RETURN means continue previous chain. Modules that process packets can be added, available modules include marking, address translation and redirection, logging, routing and others.

```
> cat /etc/sysconfig/iptables
*filter
:INPUT ACCEPT [0:0]
:FORWARD ACCEPT [0:0]
:OUTPUT ACCEPT [0:0]
:INPUT_FROM_LOCAL - [0:0]
:INPUT_FROM_WORLD - [0:0]
:FORWARD_FROM_LOCAL - [0:0]
:FORWARD_FROM_WORLD - [0:0]

# Sort traffic
-A INPUT -i lo -j INPUT_FROM_LOCAL
-A INPUT -i eth0 -j INPUT_FROM_LOCAL
-A INPUT -i tun0 -j INPUT_FROM_LOCAL
-A INPUT -i tun1 -j INPUT_FROM_LOCAL
-A INPUT -j INPUT_FROM_WORLD
-A FORWARD -i lo -j FORWARD_FROM_LOCAL
-A FORWARD -i eth0 -j FORWARD_FROM_LOCAL
-A FORWARD -i tun0 -j FORWARD_FROM_LOCAL
-A FORWARD -i tun1 -j FORWARD_FROM_LOCAL
-A FORWARD -j FORWARD_FROM_WORLD

# Input from local machines
-A INPUT_FROM_LOCAL -j ACCEPT

# Input from world machines
-A INPUT_FROM_WORLD -p tcp --dport ssh -j ACCEPT
-A INPUT_FROM_WORLD -p tcp --dport http -j ACCEPT
-A INPUT_FROM_WORLD -p tcp --dport smtp -j ACCEPT
-A INPUT_FROM_WORLD -m state --state ESTABLISHED,RELATED -j ACCEPT
-A INPUT_FROM_WORLD -j REJECT

# Forward from local machines
-A FORWARD_FROM_LOCAL -j ACCEPT

# Forward from world machines
-A FORWARD_FROM_WORLD -m state --state ESTABLISHED,RELATED -j ACCEPT
-A FORWARD_FROM_WORLD -j REJECT

COMMIT

*nat
:PREROUTING ACCEPT [0:0]
:POSTROUTING ACCEPT [0:0]
```

```
:OUTPUT ACCEPT [0:0]
-A PREROUTING -s 192.168.0.128/25 -p tcp --dport http -j REDIRECT --to-ports 3128
-A PREROUTING -s 192.168.0.128/25 -p tcp --dport smtp -j REDIRECT --to-ports 25
-A POSTROUTING -o ppp0 -s 192.168.0.128/25 -j MASQUERADE
COMMIT
```

Use **iptables -L -v** to list the current rules.

References

1. Graham Shaw: Implement Port Knocking Using IPTables

Packet Scheduling

Given that neither the network capacity nor the queues capacity is infinite, it is possible to overload the network or the queues with packets. To prevent that, packet policing is used to discard input packets and packet scheduling is used to time output packets.

Stochastic Fair Queuing

The stochastic fair queuing algorithm is used when many flows need to compete for bandwidth. The algorithm approximates having a queue for each flow and sending data from the queues in a round robin fashion. Rather than having as many queues as flows, however, the algorithm hashes a potentially large number of flows to a relatively small number of queues. To compensate for the possibility of a collision that would make multiple flows share one queue, the algorithm changes the hash function periodically.

Token Bucket

The token bucket algorithm is used when single flow needs to observe bandwidth. The flow is assigned a bucket for tokens with a defined maximum capacity. Tokens are added regularly and removed when data is sent, no tokens are added to a full bucket, no data can be sent when no tokens are available. The speed of adding tokens determines bandwidth limit. The capacity of token bucket determines fluctuation limit.

Hierarchical Token Bucket

To be done.

Class Based Queuing

The class based queuing algorithm is used when multiple flows need to share bandwidth. The flows are separated into hierarchical classes that specify their bandwidth requirements and can borrow unused bandwidth from each other.

A class has a *level*. The level of a leaf class is 1, the level of a parent class is one higher than the maximum level of its children.

A class is *under limit* if it transmits below the allocated capacity. A class is *over limit* if it transmits above the allocated capacity. A class is *on limit* otherwise.

A class is *unsatisfied* if it is under limit and it or its siblings have data to transmit. A class is *satisfied* otherwise.

A class is *regulated* if the class based queuing algorithm prevents it from sending data. A class is *unregulated* otherwise.

V klasické Formal Sharing implementaci může třída zůstat unregulated pokud není over limit, nebo pokud má předka na úrovni i, který není over limit a ve stromu nejsou žádné unsatisfied třídy úrovně nižší než i.

Drobný nedostatek algoritmu je příliš složitá podmínka regulace. Proto se definuje Ancestor Only Sharing, ve kterém třída zůstává unregulated pokud není over limit, nebo pokud má předka, který je under limit. Nevýhodou tohoto přístupu pochopitelně je, že bude omezovat over limit třídy i tehdy, pokud tyto momentálně nikomu nevadí.

Další variantou je Top Level Sharing, které definuje maximální úroveň, ze které si ještě třídy smí půjčovat přenosové pásmo. Třída pak smí zůstat unregulated pokud není over limit nebo pokud má předka do dané úrovně, který je under limit. Úpravou maximální úrovně se pak dá tento algoritmus regulovat, pro nekonečnou úroveň je stejný jako Ancestor Only Sharing, pro stejnou úroveň jako je nejmenší úroveň unsatisfied třídy je téměř stejný jako Formal Sharing, pro úroveň 1 algoritmus reguluje všechny over limit třídy a tím vyprazdňuje fronty.

Pro nastavování maximální úrovně pro Top Level Sharing se zpravidla používá heuristika. Jedna z možných funguje následujícím způsobem:

- Kdykoliv přijde paket třídy, která není over limit, maximum je 1 (tj. heuristika se snaží zaručit třídám jejich přenosové pásmo).
- Kdykoliv přijde paket třídy, která je over limit, ale má under limit předka třídy nižší než je aktuální maximum, maximum je tato třída (tj. na rostoucí zatížení reaguje snižováním možnosti půjčovat si přenosové pásmo).
- Kdykoliv třída odešle paket a má bud’ prázdnou frontu nebo se stane regulovanou, maximum je nekonečno (tj. heuristika uvolňuje omezení když se mění podmínky).

References

1. Floyd S., Jacobson V.: Link-Sharing and Resource Management Models for Packet Networks, IEEE/ACM Transactions on Networking 3(4), August 1995

Random Early Detection

The goal of the random early detection queuing algorithm is to avoid anomalies associated with algorithms that fill a queue first and drop a queue tail when the queue is filled. A weighted average of the queue length is kept and within a range of minimum and maximum queue lengths, packets are marked or dropped with a probability proportional to the current weighted average of the queue length. This gives the flow control algorithms of the transport protocols an early warning before the queue is filled.

References

1. Floyd S., Jacobson V.: Random Early Detection Gateways for Congestion Avoidance

Example: Linux Packet Scheduling

Linux uses queuing disciplines associated with network devices to determine how packets should be scheduled. Some queuing disciplines can combine other queuing disciplines. Queueing disciplines are connected through classes. Filters tell what packets go to what class.

```
# Root qdisc is prio with 3 bands
tc qdisc add dev ppp0 root handle 1: prio bands 3

# Band 1 qdisc is sfq and filter is ICMP & SSH & DNS & outbound HTTP
tc qdisc add dev ppp0 parent 1:1 sfq perturb 16
tc filter add dev ppp0 parent 1: protocol ip prio 1 u32 match ip protocol 1 0xff flowid
tc filter add dev ppp0 parent 1: protocol ip prio 1 u32 match ip sport 22 0xffff flowid
tc filter add dev ppp0 parent 1: protocol ip prio 1 u32 match ip dport 22 0xffff flowid
tc filter add dev ppp0 parent 1: protocol ip prio 1 u32 match ip sport 53 0xffff flowid
tc filter add dev ppp0 parent 1: protocol ip prio 1 u32 match ip dport 53 0xffff flowid
tc filter add dev ppp0 parent 1: protocol ip prio 1 u32 match ip sport 80 0xffff flowid
tc filter add dev ppp0 parent 1: protocol ip prio 1 u32 match ip sport 443 0xffff flowi

# Band 2 qdisc is sfq and filter is anything unfiltered
tc qdisc add dev ppp0 parent 1:2 sfq perturb 16
tc filter add dev ppp0 parent 1: protocol ip prio 9 u32 match u8 0 0 flowid 1:2

# Band 3 qdisc is tbf and filter is outbound SMTP
tc qdisc add dev ppp0 parent 1:3 tbf rate 128kbit buffer 100000 latency 100s
tc filter add dev ppp0 parent 1: protocol ip prio 1 u32 match ip dport 25 0xffff flowid
tc filter add dev ppp0 parent 1: protocol ip prio 1 u32 match ip dport 465 0xffff flowi
```

The example first attaches a priority queuing discipline to ppp0. The queuing discipline distinguishes three priority bands and schedules higher priority bands before lower priority bands.

Next, the example attaches a shared fair queuing discipline as a child of the priority queuing discipline with priority 1. The queuing discipline schedules packets from multiple streams in round robin manner. A series of filters then tells that ICMP (IP protocol 1), SSH (port 22), DNS (port 53) and outgoing web replies (port 80) packets belong to class 1:1, which is this queuing discipline.

Next, the example attaches a shared fair queuing discipline as a child of the priority queuing discipline with priority 2. A filter then tells that all packets belong to class 1:2, which is this queuing discipline. The filter has a priority 9 as opposed to priority 1 of other filters, this makes it the last filter matched.

Next, the example attaches a token bucket discipline as a child of the priority queuing discipline with priority 3. The queuing discipline schedules packets with a bandwidth limit. A pair of filters then tells that outgoing SMTP (port 80) packets belong to class 1:3, which is this queuing discipline.

Together, the filter tells Linux to first send ICMP, SSH, DNS and outgoing web replies to ppp0. If there are no such packets, the filter tells Linux to send any packets except outgoing SMTP. If there are no such packets, the filter tells Linux to send outgoing SMTP with a bandwidth limit.

Rehearsal

Questions

1. Vysvětlete, proč při implementaci přístupu k síti v operačním systému může kopírování přenášených dat být problém. Zhodnoťte míru tohoto problému a uveďte, jakým způsobem jej lze odstranit.

2. Vysvětlete roli filtrování paketů v operačním systému. Uveďte příklady kritérií, podle kterých mohou být pakety filtrovány a příklady akcí, které mohou filtry s pakety vykonávat.
3. Vysvětlete roli plánovače paketů v operačním systému.
4. Popište *Token Bucket* algoritmus pro plánování paketů a vysvětlete, co je jeho cílem.
5. Popište *Stochastic Fair Queuing* algoritmus pro plánování paketů a vysvětlete, co je jeho cílem.
6. Popište *Class Based Queuing* algoritmus pro plánování paketů a vysvětlete, co je jeho cílem.
7. Popište *Random Early Detection* algoritmus pro plánování paketů a vysvětlete, co je jeho cílem.

Network Subsystem Applications

File Systems

Example: Network File System

Three major versions of the NFS standard are 2, 3 and 4.

Version 2 of the NFS standard is designed to permit stateless server implementation. A stateless server holds no client related state, server failure and recovery is therefore transparent to connected clients. The design relies on identifying directories and files using special file handles that refer directly to server disk data, rather than to data in memory. From the client perspective, an NFS file handle is an opaque array of 32 bytes, the server uses the NFS file handle to store the file system ID, the file ID (I-node number), and the generation ID (basically I-node version number).

```
const MNTPATHLEN = 1024;  /* maximum bytes in a pathname argument */
const MNTNAMLEN = 255;    /* maximum bytes in a name argument */
const FHSIZE = 32;        /* size in bytes of a file handle */

typedef opaque fhandle [FHSIZE];
typedef string name <MNTNAMLEN>;
typedef string dirpath <MNTPATHLEN>;

union fhstatus switch (unsigned fhs_status) {
  case 0:
    fhandle fhs_fhandle;
  default:
    void;
};
```

The NFS standard introduces the NFS protocol and the mount protocol, both built over RPC. The purpose of the mount protocol is to provide reference to the root of the exported directory tree, the NFS protocol provides operations upon directories and files.

The basic operation of the mount protocol is MNT, which returns the file handle for the root of an exported directory tree identified by a path. Complementary to MNT is UMNT, which the server uses to track currently mounted paths, however, because of the stateless design the list of currently mounted paths can become stale. The DUMP operation provides the list of paths the server believes are mounted by the client, the EXPORT operation lists the paths of all exported directory trees.

```
typedef struct mountbody *mountlist;
struct mountbody {
  name ml_hostname;
  dirpath ml_directory;
  mountlist ml_next;
};

typedef struct groupnode *groups;
struct groupnode {
  name gr_name;
  groups gr_next;
};

typedef struct exportnode *exports;
struct exportnode {
  dirpath ex_dir;
  groups ex_groups;
  exports ex_next;
};

program MOUNTPROG {
  version MOUNTVERS {
    void MOUNTPROC_NULL (void) = 0;
    fhstatus MOUNTPROC_MNT (dirpath) = 1;
    mountlist MOUNTPROC_DUMP (void) = 2;
    void MOUNTPROC_UMNT (dirpath) = 3;
    void MOUNTPROC_UMNTALL (void) = 4;
    exports MOUNTPROC_EXPORT (void) = 5;
    exports MOUNTPROC_EXPORTALL (void) = 6;
  } = 1;
} = 100005;
```

The operations of the NFS protocol resemble those of common file system interfaces, such as reading and writing files and manipulating directories. Due to the stateless design, there is no concept of opening and closing a file - instead of the pair of open and close operations, the protocol introduces the LOOKUP operation, which accepts a file name and returns a file handle that remains valid for the lifetime of the file.

```
program NFS_PROGRAM {
  version NFS_VERSION {
    ...
    diropres NFSPROC_LOOKUP (diropargs) = 4;
    ...
  } = 2;
} = 100003;

struct diropargs {
  nfs_fh  dir;            /* directory file handle */
  filename name;          /* file name */
};

union diropres switch (nfsstat status) {
case NFS_OK:
  diropokres diropres;
default:
  void;
};

struct diropokres {
  nfs_fh file;
  fattr attributes;
};

struct fattr {
  ftype type;             /* file type */
  unsigned mode;          /* protection mode bits */
  unsigned nlink;         /* number of hard links */
  unsigned uid;           /* owner user id */
  unsigned gid;           /* owner group id */
  unsigned size;          /* file size in bytes */
  unsigned blocksize;     /* preferred block size */
  unsigned rdev;          /* special device number */
  unsigned blocks;        /* used size in kilobytes */
  unsigned fsid;          /* device number */
  unsigned fileid;        /* inode number */
  nfstime atime;          /* time of last access */
  nfstime mtime;          /* time of last modification */
  nfstime ctime;          /* time of last change */
};

struct nfs_fh {
  opaque data [NFS_FHSIZE];
};

enum nfsstat {
  NFS_OK=0,               /* No error */
  NFSERR_PERM=1,          /* Not owner */
  NFSERR_NOENT=2,         /* No such file or directory */
  NFSERR_IO=5,            /* I/O error */
  NFSERR_NXIO=6,          /* No such device or address */
  NFSERR_ACCES=13,        /* Permission denied */
  NFSERR_EXIST=17,        /* File exists */
  NFSERR_NODEV=19,        /* No such device */
  NFSERR_NOTDIR=20,       /* Not a directory*/
  NFSERR_ISDIR=21,        /* Is a directory */
  NFSERR_FBIG=27,         /* File too large */
  NFSERR_NOSPC=28,        /* No space left on device */
  NFSERR_ROFS=30,         /* Read-only file system */
  NFSERR_NAMETOOLONG=63,  /* File name too long */
  NFSERR_NOTEMPTY=66,     /* Directory not empty */
  NFSERR_DQUOT=69,        /* Disc quota exceeded */
  NFSERR_STALE=70,        /* Stale NFS file handle */
  NFSERR_WFLUSH=99        /* Write cache flushed */
};

program NFS_PROGRAM {
  version NFS_VERSION {
    ...
    readres NFSPROC_READ (readargs) = 6;
    ...
  } = 2;
} = 100003;

struct readargs {
  nfs_fh file;            /* handle for file */
  unsigned offset;        /* byte offset in file */
  unsigned count;         /* immediate read count */
  unsigned totalcount;    /* total read count (from this offset)*/
};

union readres switch (nfsstat status) {
case NFS_OK:
  readokres reply;
default:
  void;
};

struct readokres {
  fattr attributes;       /* attributes */
  opaque data <NFS_MAXDATA>;
};
```

```
struct fattr {
  ftype type;             /* file type */
  unsigned mode;          /* protection mode bits */
  unsigned nlink;         /* number of hard links */
  unsigned uid;           /* owner user id */
  unsigned gid;           /* owner group id */
  unsigned size;          /* file size in bytes */
  unsigned blocksize;     /* preferred block size */
  unsigned rdev;          /* special device number */
  unsigned blocks;        /* used size in kilobytes */
  unsigned fsid;          /* device number */
  unsigned fileid;        /* inode number */
  nfstime atime;          /* time of last access */
  nfstime mtime;          /* time of last modification */
  nfstime ctime;          /* time of last change */
};

struct nfs_fh {
  opaque data [NFS_FHSIZE];
};

enum nfsstat {
  NFS_OK=0,               /* No error */
  NFSERR_PERM=1,          /* Not owner */
  NFSERR_NOENT=2,         /* No such file or directory */
  NFSERR_IO=5,            /* I/O error */
  NFSERR_NXIO=6,          /* No such device or address */
  NFSERR_ACCES=13,        /* Permission denied */
  NFSERR_EXIST=17,        /* File exists */
  NFSERR_NODEV=19,        /* No such device */
  NFSERR_NOTDIR=20,       /* Not a directory*/
  NFSERR_ISDIR=21,        /* Is a directory */
  NFSERR_FBIG=27,         /* File too large */
  NFSERR_NOSPC=28,        /* No space left on device */
  NFSERR_ROFS=30,         /* Read-only file system */
  NFSERR_NAMETOOLONG=63,  /* File name too long */
  NFSERR_NOTEMPTY=66,     /* Directory not empty */
  NFSERR_DQUOT=69,        /* Disc quota exceeded */
  NFSERR_STALE=70,        /* Stale NFS file handle */
  NFSERR_WFLUSH=99        /* Write cache flushed */
};
```

The stateless design is not entirely transparent to clients. Places where differences from local file system appear include permissions, which are normally tested on opening a file. In absence of the open and close operations, permissions are checked on each read and write instead. Furthermore, the permissions rely on the UID and GID concepts, which need to be synchronized across clients and the server. In absence of network wide authentication mechanism, the server must trust the clients to supply correct credentials.

Among the more subtle differences, certain permission checks must be relaxed - for example, a client can have a right to execute a file without the right to read the file, however, for the server both operations imply accessing file content. Lack of open and close changes behavior when deleting an open file. Finally, there is a limit on RPC argument size that forces clients to split large data operations, this breaks atomicity.

Version 3 of the NFS protocol introduces the NLM protocol for managing locks, which can be used with any version of the NFS protocol. Recovery of locks after crash is solved by introducing lease and grace periods. The server only grants a lock for a lease period. The server enters grace period longer than any lease period after crash and only grants lock renewals during the grace period.

Version 4 of the NFS protocol abandons statelessness and integrates the mount, NFS and NLM protocols, and introduces security, compound operations that can pass file handle to each other, extended attributes, replication and migration, client caching.

References

1. RFC 1094: NFS Network File System Protocol Specification
2. RFC 1813: NFS Version 3 Protocol
3. RFC 3530: NFS Version 4 Protocol

Example: Server Message Block And Common Internet File System

TODO: Some description, at least from RFC and SMB & CIFS protocol.

Example: Andrew File System

The Andrew File System or AFS is a distributed file system initially developed at CMU. AFS organizes files under a global name space split into cells, where a cell is an administrative group of nodes. Servers keep subtrees of files in volumes, which can be moved and read only replicated across multiple servers, and which are listed in volume location database replicated across database servers.

Clients cache files, writes are propagated on close or flush. A server sends file data together with a callback, which is a function that notifies of outdated file data in cache. When a write is propagated to the server, the server notifies all clients that cache the file data that their callback has been broken. Clients renew callbacks when opening files whose file data were sent some time ago.

AFS uses Rx, which is a proprietary RPC implementation over UDP. AFS uses Kerberos for authentication. AFS uses identities that are separate from system user identities.

Example: Coda File System

The Coda File System sports a design similar to AFS, with global name space, replicated servers and caching clients. Servers keep files in volumes, which can be moved and read write replicated across multiple servers. Files are read from one server and written to all servers. Clients check versions on all servers and tell servers to resolve version mismatches.

Clients work in strongly connected, weakly connected and disconnected modes. The difference between connected and disconnected modes is that in the connected modes, the client hoards files, while in the disconnected mode, the client uses the hoarded files. The difference between strongly connected and weakly connected modes is that in the strongly connected mode, writes are synchronous, while in the weakly connected mode, writes are reintegrated.

Reintegration happens whenever there is a write to be reintegrated and the client is connected. Writes are reintegrated using an optimized replay log of mutating operations. Conflicts are solved manually.

Global File System

The Global File System is a distributed file system based on shared access to media rather than shared access to files. Conceptually, the file system uses traditional disk layout with storage pools of blocks, bitmaps to keep track of block usage, distributed index nodes that point to lists of blocks stored in as many levels of a branching hierarchy as required by file size, and journals to maintain metadata consistency. The distribution relies on most data structures occupying entire blocks and on introducing a distributed block locking protocol.

GFS supports pluggable block locking protocols. Three block locking protocols currently available are:

- DLM (Distributed Locking Manager) uses distributed architecture with a distributed directory of migrating lock instances.
- GULM (Grand Unified Locking Manager) uses client server architecture with replicated servers and majority quora.
- NOLOCK makes it possible to completely remove locking and use GFS locally.

Computational Resource Sharing

Network Load Balancing

Klasické aplikace v distribuovaném systému, kde se procesy přesouvají na méně zatížené uzly. Snaží se o ni i klasické systémy, například Mosix či Beowulf pro Linux. Problem with uniform resource access.

Example: Mosix

The goal of Mosix is to build clusters of homogeneous computers that allow transparent load balancing. Mosix has been developed since 1981 for various flavors of Unix and finally settled on Linux.

Mosix spreads load among the computers in a cluster by migrating processes from their home nodes to remote nodes. The decision to migrate a process is based on multiple criteria, which include the communication cost, the memory requirements, the processor usage. To avoid thrashing, overriding importance is assigned to memory requirements.

When accessing resources, a migrated process can either access the local resources of the remote node, or the remote resources of the home node. In general, access to local resources is faster than access to remote resources, but some remote resources cannot be replaced by local resources. Mosix therefore intercepts accesses of migrated processes to resources and directs them towards local resources when transparency can be preserved and towards remote resources otherwise. To facilitate access to remote resources, the migrated process communicates with its process deputy on the home node.

To guarantee transparency when accessing user credentials, Mosix requires that all computers in a cluster share the same UID and GID space.

To guarantee transparency when accessing files but avoid doing all file system operations remotely, Mosix relies on DFSA (Direct File System Access) optimizations. These optimizations recognize cluster file systems that are mounted across the entire cluster and do most file system operations locally on such file systems.

Mosix refuses to migrate processes that use shared memory etc.

References

1. Mosix, http://www.mosix.org
2. openMosix, http://www.openmosix.org

Network Global Memory

Existují další věci, které se dají se sítí dělat. Například se na síť dá swapovat, to má výhodu v low latency. Také distributed shared memory.

Single System Image

Example: Amoeba

Drobný popis Amoeby, distribuovaný systém od pana Tanenbauma, pro komunikaci RPC generované z AIL, předpokládá dostatek CPU a dostatek paměti. Dva hlavní rysy file systému jsou oddělení jmen od souborů a immutable soubory.

- Naming separation. Jména má na starosti directory server, který není vázaný na zbytek file systému, svazuje jména s capabilities.
- Immutable files. Se souborem se smí dělat pouze CREATE, READ, DELETE, SIZE (vytvoří soubor z dat, přečte soubor, smaže soubor, vrátí velikost souboru). Má to spoustu výhod, například caching a replication se nemusí starat o konzistenci. Protože je dost paměti, vždycky to projde.

Když se z Amoeby začal stávat použitelný systém, přiznalo se, že ne vždycky může být dost paměti. Pak se soubory rozdělily na committed a uncommitted. committed jsou viz výše, uncommited jsou v procesu vytváření a dá se do nich připisovat než se commitnou. Dalším drobným ústupkem je možnost čtení po částech.

Filesystem jsou Bullet Server (jako že rychlý) a Directory Server (jako že adresář ?).

Bullet Server se stará o soubory, má operace CREATE (s parametrem zda commited nebo uncommited), MODIFY, INSERT, DELETE (na data uncommitted souborů, jako parametr říkají, zda commitnout), READ (na committed file), SIZE. Soubory jsou reprezentované pomocí capabilities, soubory bez capabilities se automaticky mažou. Protože se neví, kdo má capabilities, používají se timeouts (uncommitted files se mažou za 10 minut, committed files mají parametr age, který Directory Server posouvá voláním touch, typicky se volá touch jednou za hodinu a zmizí za 24 hodin od posledního touch).

Directory Server je obecný naming server, který přiřazuje jména k capabilities. Základní operace jsou CREATE, DELETE (vytvoření a zrušení adresáře), APPEND (vložení capability do directory), REPLACE (nahrazení capability v directory, důležité pro atomický update souborů), LOOKUP, GETMASKS, CHMOD (čtení a nastavení práv).

Example: Mach

To be done.

Example: Plan 9

Plan 9 is an experimental operating system developed around 1990 in Bell Labs. Plan 9 builds on the idea that all resources should be named and accessed uniformly.

To be done.

Rehearsal

Questions

1. Popište architekturu síťového systému souborů NFS včetně používaných protokolů a hlavních operací těchto protokolů.
2. Vysvětlete, co to je a jaké položky zpravidla obsahuje *file handle* v síťovém systému souborů NFS.
3. Vysvětlete, jaké problémy přináší bezstavovost NFS při testování přístupových práv a jak jsou tyto problémy řešeny.
4. Vysvětlete, jaké problémy přináší bezstavovost NFS při mazání otevřených souborů a jak jsou tyto problémy řešeny.
5. Vysvětlete, jaké problémy přináší bezstavovost NFS při zamykání souborů a jak jsou tyto problémy řešeny.

Chapter 7. Security Subsystem

Authentication

Authentication je problém ověření toho, zda je aktivita (proces, uživatel) tím, za koho se vydává. Zpravidla se používá kombinace jména a hesla, typicky je pak v systému nějaká centrální autorita, která toto ověřuje, ostatní se už jen ptají téhle autority.

Linux PAM Example

PAM je sada knihoven, která poskytuje API pro ověření identity. Jejím hlavním rysem je schopnost dynamicky konfigurovat, jaké aplikace budou používat jaké metody ověření identity. Funkce jsou rozděleny do čtyřech skupin:

- Account management
- Authentication management
- Password management
- Session management

PAM je konfigurována souborem, který pro každou službu (aplikace, která chce PAM používat) uvádí, pro jakou skupinu bude použitý jaký modul a jak se zachovat při jeho selhání.

```
> cat /etc/pam.d/login
auth       required     pam_securetty.so
auth       required     pam_stack.so service=system-auth
auth       required     pam_nologin.so
account    required     pam_stack.so service=system-auth
password   required     pam_stack.so service=system-auth
session    required     pam_stack.so service=system-auth
session    optional     pam_console.so
```

Uvedený příklad říká, že přihlášení pomocí služby login bude vyžadovat úspěšné vykonání modulů securetty, stack a nologin. Modul securetty testuje, zda se uživatel root přihlašuje z terminálu uvedeného v /etc/securetty, pro ostatní uživatele uspěje vždy. Modul nologin testuje, zda neexistuje soubor /etc/nologin, pokud ano, uspěje pouze uživatel root. Modul stack zařadí všechny testy služby system-auth, kde jsou ještě moduly env (podle /etc/security/pam_env.conf nastaví proměnné prostředí), unix (podle /etc/passwd a /etc/shadow ověří jméno a heslo) a deny (jako default volba vždy selže). Novější verze mají místo modulu stack volby include a substack.

Obecně lze použít volby requisite - selhání modulu způsobí okamžité vrácení chyby, required - selhání modulu způsobí vrácení chyby po zpracování ostatních modulů, sufficient - úspěch modulu způsobí okamžité vrácení úspěšného výsledku, optional - úspěch či selhání modulu je důležité pouze pokud je jediný. Krom toho existují ještě složitější metody kombinace modulů, které dovolují pro každý možný způsob ukončení modulu (hlavně úspěch a selhání, ale ještě mnoho dalších) uvést, zda se má modul ignorovat, zda má stack okamžitě nebo nakonec selhat nebo uspět, a ještě pár maličkostí.

Z pohledu programátora je pak použití PAM přímočaré, hlavní je asi funkce pam_authenticate pro ověření uživatele, další funkce jsou k dispozici pro zbývající funkce knihovny. Zvláštností je použití konverzační funkce, to je callback funkce poskytnutá aplikací knihovně tak, aby tato mohla v případě potřeby vyzvat uživatele například k zadání hesla.

```
#include <security/pam_appl.h>
#include <security/pam_misc.h>
```

```
static struct pam_conv conv = { misc_conv, NULL };

int main(int argc, char *argv[])
{
  pam_handle_t *pamh = NULL;
  char *user;
  int retval;

  // ...

  retval = pam_start ("check_user", user, &conv, &pamh);
  if (retval == PAM_SUCCESS)
    retval = pam_authenticate (pamh, 0); // Is user really himself ?
  if (retval == PAM_SUCCESS)
    retval = pam_acct_mgmt (pamh, 0);  // Is user account valid ?
  if (retval == PAM_SUCCESS)

  // ...

  pam_end (pamh, retval);
}
```

References

1. Linux Man Pages
2. Morgan A. G.: Linux PAM Application Developer's Guide
3. Morgan A. G.: Linux PAM System Administrator's Guide

Kerberos Example

Problémem s centrální autoritou pro ověření identity je možnost falšovat její výsledky. To hrozí zejména v distribuovaných systémech, kde je snažší zachytit komunikaci mezi aplikacemi a touto autoritou. Aby se ošetřil tento problém, používají se bezpečnostní protokoly na základě návrhu Needhama a Schroedera, jejichž typickým představitelem je Kerberos z MIT, RFC 1510.

Princip zmíněného protokolu je jednoduchý. Předpokládá se použití symetrické kryptografie a existence autority, která má k dispozici tajné klíče všech účastníků protokolu. Pokud pak klient chce komunikovat se serverem, použije následující sekvenci:

- Klient pošle autoritě žádost o spojení se serverem, ve které uvede své jméno, jméno serveru a unikátní číslo U1.
- Autorita ověří právo klienta spojit se se serverem.
- Autorita pošle klientovi zprávu zašifrovanou jeho tajným klíčem KC, ve které uvede unikátní číslo U1 předtím zaslané klientem, náhodný klíč KR pro komunikaci se serverem a tiket T, což je ještě jednou klíč KR a jméno klienta, vše zašifrované tajným klíčem serveru KS.
- Klient ověří pravost autority tím, že byla schopna vrátit zaslané unikátní číslo U1 zašifrované jeho tajným klíčem KC.
- Klient pošle serveru zprávu, ve které uvede tiket T.
- Server pošle klientovi zprávu zašifrovanou klíčem KR, ve které uvede unikátní číslo U2.

- Klient pošle serveru zprávu zašifrovanou klíčem KR, ve které uvede domluvenou transformaci unikátního čísla U2.
- Server ověří pravost klienta tím, že byl schopen provést transformaci unikátního čísla U2 se znalostí klíče KR.

Zbývající slabinou tohoto protokolu je možnost vydávat se za klienta v situaci, kdy skutečný klient například havaruje a v paměti zůstane klíč KR. Kerberos tento problém řeší doplněním časového razítka tak, aby tiket T a tedy klíč KR bylo možné použít jen omezenou dobu, po které klient požádá autoritu o obnovení.

References

1. Coulouris G., Dollimore J., Kindberg T.: Distributed Systems Concepts And Design

Rehearsal

Questions

1. Vysvětlete termín *authentication* .

Authorization

Authorization je problém rozhodnutí, zda je daná aktivita (proces, uživatel) oprávněna udělat nějakou akci nad nějakým prostředkem (soubor, zařízení).

Activities do Actions on Resources

Ověření práv se s oblibou modeluje tak, že se definuje množina aktivit, množina prostředků a množina akcí, a pak se do tabulky kde osy určují aktivitu a prostředek zapisují povolené akce. Udržovat takovou tabulku v kuse by však bylo nepraktické, takže se ukládá po skupinách, odtud access control lists a capabilities.

Access Control Lists

Access control lists je technika, kde se s každým prostředkem uloží seznam aktivit a jim dovolených akcí. ACL umí leckteré UNIXy, v těch jsou zpravidla jako aktivity bráni users nebo groups a akce jsou klasické RWX nad soubory. Z téhož principu vlastně vycházejí i standardní atributy u UNIX souborů.

Nevýhod ACL je řada, zřejmě největší z nich je statičnost vzhledem k aktivitám, kvůli které se ACL dělají pro users a ne pro processes. To může vést k situaci, kdy procesy mají zbytečně silná práva, řeší se mimo jiné vytvářením pseudo users pro některé procesy či dodatečným omezováním práv.

Další věcí je scalability, vlastně nutím prostředky ukládat informace o aktivitách, kterých může být hafo. Odtud pokusy o dědění práv z hierarchicky nadřazených objektů a zaznamenávání změn, sdružování práv do skupin a podobně.

Capabilities

Capabilities je technika, kdy si každá aktivita nese seznam prostředků a nad nimi povolených akcí. Při přístupu k prostředku se pak aktivita prokáže svou capability, kterou systém verifikuje. Toto je mechanizmus, který běžné systémy příliš často nemívají, ale u distribuovaných systémů nachází značné uplatnění, příklady jsou capabilities u Amoeby, Machu či EROSu nebo credentials v CORBE.

Problémem capabilities je otázka kam je umístit. Samozřejmě není možné dát je jen tak k dispozici procesům, protože ty by mohly zkoušet je padělat. Jedním z řešení je mít capabilities v protected paměti, to je třeba příklad Machu (procesy mají jen handles do svých tabulek capabilities, tabulky samy jsou v kernelu). Jiné řešení je šifrování capabilities, to dělá Amoeba. Každý objekt má u sebe 48 bitů náhodné číslo, toto číslo plus rights z capability se proženou oneway funkcí a ta se přidá do capability, kterou má uživatel k dispozici. Pokud nemá to štěstí, nemůže si změnit capability aby ukazovala na jiný objekt, ani aby nešla jiná práva.

Ačkoliv to na první pohled vypadá jako že capabilities a access control lists jsou ekvivalentní, jsou v nich důležité rozdíly. Capabilities mohou náležet jednotlivým procesům, tedy je možné je použít například při ochraně dat před vyzrazením tím, že se untrusted procesům omezí initial capabilities.

Levels delimit Security and Integrity

Zpět na o něco vyšší úroveň, model ochran založený na zmiňované tabulce má jeden vážný nedostatek, totiž není z něj jasně patrné co a jak bude chránit. Je zřejmé, že práva budou tranzitivní, ale bohužel pokud nejsou k dispozici informace o výměně informací mezi aktivitami, což zpravidla nejsou, není rozhodnutelné, zda může existovat posloupnost akcí dovolující v konečném efektu aktivitě nějakou akci.

Proto se vymýšlejí ještě jiné modely. Další z klasických je založený na security levels a integrity levels. Aktivity mají clearances, data mají classes. Řekne se, že není přípustné číst informace z vyšších security classes než máme clearances ani zapisovat informace do nižších security classes než máme clearances, a podobně že není přípustné zapisovat informace do vyšších integrity classes, ani číst informace z nižších integrity classes.

Tohle má ovšem jiný problém, totiž k dokonalé implementaci by bylo potřeba sledovat každý bit informace, což by bylo nákladné, a tedy se používají zjednodušení. Ta ovšem nesmí porušit security ani integrity, a tak jsou raději pesimističtější. Výsledkem toho je pozvolný drift dat do vyšších security a nižších integrity classes.

Example: Security Enhanced Linux

The framework introduces policies that tell how subjects (processes) can manipulate objects (devices, files, sockets ...). Subjects and objects have types, which are stored in a security context in the form of a triplet of user, role, type. Security context of files is stored in extended attributes.

To be done.

```
> ls -Z /
       system_u:object_r:bin_t:s0 bin
      system_u:object_r:boot_t:s0 boot
    system_u:object_r:device_t:s0 dev
       system_u:object_r:etc_t:s0 etc
 system_u:object_r:home_root_t:s0 home
       system_u:object_r:lib_t:s0 lib
       system_u:object_r:lib_t:s0 lib64
       system_u:object_r:mnt_t:s0 media
       system_u:object_r:mnt_t:s0 mnt
       system_u:object_r:usr_t:s0 opt
      system_u:object_r:proc_t:s0 proc
system_u:object_r:admin_home_t:s0 root
   system_u:object_r:var_run_t:s0 run
       system_u:object_r:bin_t:s0 sbin
       system_u:object_r:var_t:s0 srv
     system_u:object_r:sysfs_t:s0 sys
...
> semanage fcontext -l
SELinux fcontext        type               Context
/                       directory          system_u:object_r:root_t:s0
/.*                     all files          system_u:object_r:default_t:s0
/bin                    all files          system_u:object_r:bin_t:s0
/bin/.*                 all files          system_u:object_r:bin_t:s0
/bin/bash               regular file       system_u:object_r:shell_exec_t:s0
/bin/dmesg              regular file       system_u:object_r:dmesg_exec_t:s0
/bin/ip                 regular file       system_u:object_r:ifconfig_exec_t:s0
...
/dev                    directory          system_u:object_r:device_t:s0
/dev/.*                 all files          system_u:object_r:device_t:s0
/dev/.*mouse.*          character device   system_u:object_r:mouse_device_t:s0
/dev/[0-9].*            character device   system_u:object_r:usb_device_t:s0
/dev/[shmxv]d[^/]*      block device       system_u:object_r:fixed_disk_device_t:s0
...
/home                   directory          system_u:object_r:home_root_t:s0
/home/[^/]+             directory          unconfined_u:object_r:user_home_dir_t:s0
/home/[^/]+/www(/.+)?   all files          unconfined_u:object_r:httpd_user_content_t:s
...

> ps -Z
LABEL                             PID TTY          TIME CMD
unconfined_u:unconfined_r:unconfined_t:s0-s0:c0.c1023 4891 pts/0 00:00:00 ps
unconfined_u:unconfined_r:unconfined_t:s0-s0:c0.c1023 5124 pts/0 00:00:00 bash
> id -Z
unconfined_u:unconfined_r:unconfined_t:s0-s0:c0.c1023

> semanage module -l
Module Name               Priority  Language
abrt                      100       pp
accountsd                 100       pp
acct                      100       pp
afs                       100       pp
aiccu                     100       pp
aide                      100       pp
ajaxterm                  100       pp
alsa                      100       pp
amanda                    100       pp
...
> sesearch -A -t sshd_key_t -p write
allow ssh_keygen_t sshd_key_t:file { append create getattr ioctl link lock open read re
allow sshd_keygen_t sshd_key_t:file { append create getattr ioctl link lock open read r
...
allow files_unconfined_type file_type:file { append audit_access create execute execute
...
allow ftpd_t non_security_file_type:file { append create getattr ioctl link lock open r
allow kernel_t non_security_file_type:file { append create getattr ioctl link lock open
...
allow sysadm_t non_security_file_type:file { append create getattr ioctl link lock open
...

> getsebool -a
antivirus_can_scan_system --> off
antivirus_use_jit --> off
...
daemons_dump_core --> off
daemons_enable_cluster_mode --> off
```

```
daemons_use_tcp_wrapper --> off
daemons_use_tty --> off
...
ftpd_anon_write --> off
ftpd_full_access --> off
ftpd_use_nfs --> off
...
git_cgi_enable_homedirs --> off
git_cgi_use_nfs --> off
...
httpd_anon_write --> off
httpd_builtin_scripting --> on
httpd_can_check_spam --> off
httpd_can_connect_ftp --> off
httpd_can_network_connect --> off
httpd_can_network_memcache --> off
httpd_can_sendmail --> off
httpd_enable_cgi --> on
httpd_enable_homedirs --> off
httpd_use_nfs --> off
...

> tail /var/log/audit/audit.log
type=AVC msg=audit(1515657259.550:620585): avc:  denied  { open } for  pid=8358 comm="s
...
> audit2allow < /var/log/audit/audit.log
#============= nagios_t ==============
allow nagios_t initrc_var_run_t:file open;
...
> ls -Z /run/utmp
system_u:object_r:initrc_var_run_t:s0 /run/utmp

policy_module(ssh, 2.4.2)

gen_tunable(allow_ssh_keysign, false)
gen_tunable(ssh_sysadm_login, false)

attribute ssh_server;
attribute ssh_agent_type;

type ssh_t;
type ssh_exec_t;
type ssh_home_t;
type sshd_exec_t;
...

allow ssh_t self:capability { setuid setgid ... };
allow ssh_t self:tcp_socket create_stream_socket_perms;
allow ssh_t self:unix_dgram_socket { create_socket_perms sendto };
allow ssh_t self:unix_stream_socket { create_stream_socket_perms connectto };
...

allow ssh_t sshd_key_t:file read_file_perms;
allow ssh_t sshd_tmp_t:dir manage_dir_perms;
allow ssh_t sshd_tmp_t:file manage_file_perms;
...

tunable_policy (‘allow_ssh_keysign’,‘
    domain_auto_trans (ssh_t, ssh_keysign_exec_t, ssh_keysign_t)
    allow ssh_keysign_t ssh_t:fd use;
    allow ssh_keysign_t ssh_t:process sigchld;
    allow ssh_keysign_t ssh_t:fifo_file rw_file_perms;
’)
...
```

Rehearsal

Questions

1. Vysvětlete termín *authorization* .
2. Vysvětlete, co to je *access control list* .
3. Vysvětlete, co to je *capability* .

Security Subsystem Implementation

Problémem implementace je dodržení deklarovaného bezpečnostního modelu ...

Example: DoD TCSEC Classification

Security klasifikace ... Trusted Computer System Evaluation Criteria (TCSEC or Orange Book), the Canadian Trusted Computer Product Evaluation Criteria (CTCPEC), and the Information Technology Security Evaluation Criteria (ITSEC). The goal of these documents is to specify a standard set of criteria for evaluating the security capabilities of systems.

DoD TCSEC Level D: Systems that fail to meet requirements of any higher class.

Level C1: Provides separation of users and data and access control on individual basis so that users can prevent other users from accidentaly accessing or deleting their data.

Level C2: In addition requires auditing of security related events.

Level B1: In addition requires informal statement of the security policy model and no errors with respect to that statement.

Level B2: In addition requires formal statement of the security policy model and no covert channels.

Level B3: In addition requires testability of the formal statement of the security policy model.

Level A1: In addition requires verifiability of the formal statement of the security policy model on the architecture level and verifiability of the informal statement of the security policy model on the implementation level.

Ze stránky http://www.radium.ncsc.mil/tpep/epl/epl-by-class.html existují v roce 2000 tyto secure systémy:

A1 žádný operační systém, žádná aplikace, dva routery od Boeing a Gemini Computers.

B3 operační systémy XTS-200 a XTS-300 od Wang Federal (binárně kompatibilní s UNIX System V na Intel platformách, ale aby měl B3, musí mít speciální hardware, používá security a integrity levels ala Bell, LaPadula, Biba), žádná aplikace, žádný router.

B2 operační systémy Trusted XENIX 3.0 a 4.0 od Trusted Information Systems (binárně kompatibilní s IBM XENIX), žádná aplikace, router DiamondLAN od Cryptek Secure Communications.

B1 operační systémy UTS/MLS od Amdahl Corporation, CA-ACF2 MVS od Computer Associates, SEVMS VAX 6 od DEC, ULTRIX MLS+ od DEC, CX/SX 6 od Harris Computer Systems, HP-UX BLS 9 od HP, Trusted IRIX/B od SGI, OS1100/2200 od Unisys, aplikace INFORMIX/Secure 5 od Informixu, Trusted Oracle 7 od Oracle, Secure SQL 11 od SyBase, routery ...

C2 operační systémy AOS/VS 2 od Data General, OpenVMS VAX 6 od DEC, OS/400 na AS/400 od IBM, Windows NT 4 od Microsoftu, Guardian 90 od Tandem, aplikace Microsoft SQL2000 8 ...

C1 se již nevyhodnocuje.

Seznam obsahuje pouze komerčně dostupné systémy, navíc se zhruba od roku 2000 již nepoužívá, ale stále je známý a proto zasluhuje zmínku.

Example: NIST CCEVS

Současně používaná je Common Criteria Evaluation and Validation Scheme (CCEVS) od NIST a NSA, ta hodnotí podle ISO Standard 15408 aneb Common Criteria for IT Security Evaluation. Urovně se označují EAL1 až EAL7 a obsahují kombinace požadavků v různých třídách, EAL1 je nejjednodušší (v podstatě nějak funguje), až do EAL4 se zvyšuje úroveň testování ale nikoliv nároky (produkty navržené bez uvažování CC by měly obstát na EAL4), EAL5 požaduje semiformal design and testing, EAL6 požaduje ještě semiformal verification of design, EAL7 požaduje formal design and testing a formal verification of design.

Pro malé srovnání, z operačních systémů je na EAL3 SGI IRIX, na EAL4 Solaris 8, HP-UX, Windows 2000, z databází je na EAL4 Oracle 8, ze smart cards je na EAL5 GemPlus JavaCard, vyšší levels se zřejmě zatím neudělují.