Principles of Computers
19th Lecture

Pavel Ježek, Ph.D.
pavel.jezek@d3s.mff.cuni.cz
Selected Faults/Traps/Exceptions of x86 ISA

<table>
<thead>
<tr>
<th>CPU Exception</th>
<th>Interrupt vector (all push IP of faulting instruction)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invalid opcode</td>
<td>6</td>
</tr>
<tr>
<td>Divide by zero (DIV0)</td>
<td>0</td>
</tr>
<tr>
<td>Alignment check</td>
<td>17 ($11)</td>
</tr>
</tbody>
</table>
### Selected Faults/Traps/Exceptions of x86 ISA

<table>
<thead>
<tr>
<th>CPU Exception</th>
<th>Interrupt vector (all push IP of faulting instruction)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invalid opcode</td>
<td>6</td>
</tr>
<tr>
<td>Divide by zero (DIV0)</td>
<td>0</td>
</tr>
<tr>
<td>Alignment check</td>
<td>17 ($11)</td>
</tr>
<tr>
<td><strong>General Protection Fault</strong></td>
<td>13 ($0D)</td>
</tr>
</tbody>
</table>
## Selected Faults/Traps/Exceptions of x86 ISA

<table>
<thead>
<tr>
<th>CPU Exception</th>
<th>Interrupt vector (all push IP of faulting instruction)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invalid opcode</td>
<td>6</td>
</tr>
<tr>
<td>Divide by zero (DIV0)</td>
<td>0</td>
</tr>
<tr>
<td>Alignment check</td>
<td>17 ($11)</td>
</tr>
<tr>
<td><strong>General Protection Fault</strong></td>
<td>13 ($0D)</td>
</tr>
<tr>
<td><strong>Page Fault</strong></td>
<td>14 ($0E)</td>
</tr>
</tbody>
</table>
Kernel code

Kernel code
stack
stack
free
free
free
free
free
free
free
free
free
free
9
free
8
A data
7
A data
6
A data
5
A code
4
A code
3
A code
2
A code
1
A code

PT entry

...
<table>
<thead>
<tr>
<th>Non-present</th>
<th>Kernel/Supervisor</th>
<th>User Read/Only</th>
<th>User Read/Write</th>
<th>NX/XD No Execute</th>
</tr>
</thead>
</table>

**Kernel Data**

- ... IVT
- ... kernel data
- ... page tbl
- ... kernel code
- ... stack
- ... free
- ... free
- ... free
- ... free
- ... free
- ... free
- ... free
- ... free
- ... free
- ... free
- ... free
- ... free
- ... free
- ... free
- ... free
- ... free
- ... free

**Stack**

- stack
- stack

**Free**

- free
- free
- free
- free
- free
- free
- free
- free
- free
- free

**PT Entry**

- 10 free
- 9 free
- 8 A data
- 7 A data
- 6 A data
- 5 A code
- 4 A code
- 3 A code
- 2 A code
- 1 A code

**API Calls**

- Call to RTL's `New(1000)`
- Call to kernel's `AllocMem` API function
AllocMem API will return based address of a free page (9) to RTL → RTL will manage subsequent allocations in that page.

call to kernel’s AllocMem API function

call to RTL’s New(1000)
Heap state during:

A := New(1000 bytes);
Heap state after:

```plaintext
A := New(1000 bytes);
```

### Heap Management

<table>
<thead>
<tr>
<th>Heap Management Record</th>
<th>Size</th>
<th>Next</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>size: 1008</td>
<td>nil</td>
</tr>
<tr>
<td></td>
<td>size: 3088</td>
<td>nil</td>
</tr>
</tbody>
</table>

- **FirstFreeBlock**: 1000 bytes of allocated memory
- **FirstUsedBlock**: 3080 bytes of free memory

### Permissions Table

<table>
<thead>
<tr>
<th>Page Table Entry</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Non-present</td>
</tr>
<tr>
<td></td>
<td>Kernel/Supervisor</td>
</tr>
<tr>
<td></td>
<td>User Read/Only</td>
</tr>
<tr>
<td></td>
<td>User Read/Write</td>
</tr>
<tr>
<td></td>
<td>NX/XD No Execute</td>
</tr>
</tbody>
</table>

- **PT entry**: 1
- **Kernel code**: 1
- **Page table**: 1
- **IVT**: 1

---

### Table

<table>
<thead>
<tr>
<th>Page Number</th>
<th>Type</th>
<th>Size</th>
<th>Next</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Code</td>
<td>A</td>
<td>nil</td>
</tr>
<tr>
<td>2</td>
<td>Code</td>
<td>A</td>
<td>nil</td>
</tr>
<tr>
<td>3</td>
<td>Code</td>
<td>A</td>
<td>nil</td>
</tr>
<tr>
<td>4</td>
<td>Code</td>
<td>A</td>
<td>nil</td>
</tr>
<tr>
<td>5</td>
<td>Code</td>
<td>A</td>
<td>nil</td>
</tr>
<tr>
<td>6</td>
<td>Code</td>
<td>A</td>
<td>nil</td>
</tr>
<tr>
<td>7</td>
<td>Code</td>
<td>A</td>
<td>nil</td>
</tr>
<tr>
<td>8</td>
<td>Code</td>
<td>A</td>
<td>nil</td>
</tr>
<tr>
<td>9</td>
<td>Code</td>
<td>A</td>
<td>nil</td>
</tr>
<tr>
<td>10</td>
<td>Code</td>
<td>A</td>
<td>nil</td>
</tr>
<tr>
<td>Page</td>
<td>Type</td>
<td>Content</td>
<td></td>
</tr>
<tr>
<td>------</td>
<td>----------------</td>
<td>-------------------------------</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>A code</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>A code</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>A code</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>A code</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>A code</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>A data</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>A data</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>A data</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>used (RTL heap)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>free</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**PT entry**

**Kernel/Supervisor**

**User Read/Only**

**User Read/Write**

**NX/XD No Execute**

**IVT**

**Kernel data**

**Page tbl**

**kernel code**

**stack**

**free**

**NO call to kernel’s AllocMem API function**

2nd call to RTL’s *New(2000)* – enough space for allocation in page 9
Heap state after:

A := New(1000 bytes);
B := New(2000 bytes);

```plaintext
1072 bytes of free memory
1000 bytes of allocated memory
2000 bytes of allocated memory
```
AllocMem API will return base address of next free page (10) to RTL → RTL will manage subsequent allocations in that page.

3rd call to RTL’s New(3000) – will not fit current heap (into page 9)
Heap state after:

A := New(1000 bytes);
B := New(2000 bytes);

and during:
C := New(3000 bytes);
Heap state after:

A := New(1000 bytes);
B := New(2000 bytes);
C := New(3000 bytes);

If pagesize = 4096:
2 pages = 2*4096 = 8192 bytes

8192-3008-2008-1008 = 2168 bytes of free memory
Heap state after:

A := New(1000 bytes);
B := New(2000 bytes);
C := New(3000 bytes);
Dispose(B);
firmware test and configures HW
firmware checks and calls option ROMs
firmware tries to find bootable drive:
  load boot sector to RAM
  jump/call to boot sector entrypoint in RAM
    boot loader running
  repeat
    call ReadSector firmware API
      firmware communicates with drive’s controller
      firmware stores received sectors data into RAM
  until all sectors with kernel code/data loaded to RAM
  call kernel entrypoint
firmware test and configures HW
firmware checks and calls option ROMs
firmware tries to find bootable drive:
  load boot sector to RAM
  jump/call to boot sector entrypoint in RAM
    boot loader running
    repeat
      call ReadSector firmware API
        firmware communicates with drive’s controller
        firmware stores received sectors data into RAM
      until all sectors with kernel code/data loaded to RAM
    call kernel entrypoint

kernel init
call Exec('Prog.exe')
  load Prog.exe
  call Prog.exe entrypoint
    ... Prog.EXE’s RTL initialization code
    ... main program of Prog.EXE
    ...

PURPLE = computer firmware code executed by CPU
GREEN = boot loader code executed by CPU
RED = kernel code executed by CPU
BLUE = user program code executed by CPU
Loading drivers needed to access disk drives...
Searching for Puppy files in computer disk drives...
Loading the 'luma_525.sfs' main file... copying to ram
Setting up the layered filesystem...
Performing a 'switch_root' to the layered filesystem...Kernel panic - not syncing
   Attempted to kill init!
firmware test and configures HW
firmware checks and calls option ROMs
firmware tries to find bootable drive:
  load boot sector to RAM
  jump/call to boot sector entrypoint in RAM
    boot loader running
    repeat
      call ReadSector firmware API
        firmware communicates with drive’s controller
        firmware stores received sectors data into RAM
    until all sectors with kernel code/data loaded to RAM
    call kernel entrypoint

kernel init
call Exec('shell.exe')
  load shell.exe
  call shell.exe entrypoint
...
  repeat
    call ReadKey kernel API
      kernel communicates with keyboard controller
      key := result of ReadKey
    call PrintCharacter kernel API
      kernel communicates with monitor controller
    until key <> ENTER
firmware test and configures HW
firmware checks and calls option ROMs
firmware tries to find bootable drive:
  load boot sector to RAM
  jump/call to boot sector entrypoint in RAM
  boot loader running
  repeat
    call ReadSector firmware API
    firmware communicates with drive’s controller
    firmware stores received sectors data into RAM
  until all sectors with kernel code/data loaded to RAM
  call kernel entrypoint

kernel init
  call Exec(‘shell.exe’)
  load shell.exe
  call shell.exe entrypoint
  ...
  repeat
    call ReadKey kernel API
    kernel communicates with keyboard controller
    key := result of ReadKey
    call PrintCharacter kernel API
    kernel communicates with monitor controller
  until key <> ENTER
  call Exec(‘Prog.EXE’)
  load Prog.EXE
  call Prog.EXE entrypoint
  ...
  Prog.EXE’s RTL initialization code
  ...
  main program of Prog.EXE
  ...

PURPLE = computer firmware code executed by CPU
GREEN = boot loader code executed by CPU
RED = kernel code executed by CPU
BLACK = shell code executed by CPU
BLUE = user program code executed by CPU