Advanced Operating Systems

Summer Semester 2022/2023

Martin Děcký

7

Communication and Concurrency

Classical IPC

- **POSIX signals**
  - Since UNIX Version 4
  - Asynchronous notification sent to a process (thread)
    - Similar to level-triggered interrupts (including masking)
    - Sender uses the `kill(2)` syscall
      - Run-time exceptions and state changes also cause signals (`SIGFPE`, `SIGSEGV`; `SIGPIPE`, `SIGINT`, `SIGSTOP`/`SIGTSTP`, `SIGCONT`, `SIGTRAP`)
    - Receiver thread is interrupted and a signal handler is executed (installed using `signal(2)` or `sigaction(2)`)
      - Race conditions due to nested signals
      - Calling non-reentrant functions (e.g. `malloc()`, `printf()`) is undefined behavior
      - Interruption of some syscalls
    - Real-time signals
      - Queued, guaranteed sending order

Classical IPC

- **Anonymous pipes**
- **Named pipes**
  - Persistent uni-directional pipes
    - Same API as files (anonymous pipes)
    - Pipe identification: File system i-node (bound to a directory entry)
    - No identification of senders on the receiver end
      - Writes of data larger than `PIPE_BUF` bytes can be interleaved
  - Windows named pipes
    - Dedicated namespace (Named Pipe File System `\\.\pipe\`)
    - Non-persistent (removed when all clients close the pipe)
    - Anonymous pipes are named pipes with random names

Classical IPC

- **UNIX domain sockets**
  - Reliable bi-directional stream of bytes (akin to TCP), or …
  - Unordered unreliable datagrams (akin to UDP), or …
  - Reliable ordered stream of datagrams between local processes
    - Same API as BSD sockets
    - Socket identification: File system i-node (bound to a directory entry or to an abstract socket namespace)
    - Sending file descriptors (`sendmsg()`, `rescvmsg()`) as ancillary data
      - Rudimentary capabilities

Classical IPC

- **Software shared memory**
  - POSIX Shared Memory, System V Shared Memory
    - Persistent shared memory objects in dedicated namespace
      - In Linux, objects created as tmpfs files (usually `/dev/shm`)
    - `shm_open(3)`, `mmap(2)`, `munmap(2)`, `shm_unlink(3)`
    - `shmget(2)`, `shmat(2)`, `shmdt(2)`
  - Memory mapped files
    - Shared memory backed by a file (or anonymous memory)
    - `mmap(2)`, `munmap(2)`
    - `memfd_create(2)`
      - Removed when no longer referenced
      - File sealing

Classical IPC

- **Message passing**
  - Synchronous / asynchronous, blocking / non-blocking sending
  - Synchronous / asynchronous, blocking / non-blocking receiving
  - Symmetrical / asymmetrical / indirect addressing
  - POSIX message queues, System V Message Passing
    - Indirect addressing using a message queue (key for `msgget(2)`, i-node for `mq_open(3)`)
    - `msgsnd(2)`, mq_send(3) asynchronous non-blocking (unless the queue is full)
    - `msgrcv(2)`, `mq_receive(3)` synchronous blocking by default
  - Windows Messages
    - Symmetrical addressing using window/thread handles
    - `SendMessage()` synchronous non-blocking, `SendMessageCallback()`, `SendNotifyMessage()`, `PostMessage()` asynchronous non-blocking
    - `GetMessage()` synchronous blocking, `PeekMessage()` synchronous non-blocking

Classical IPC

- **IPC abstractions**
  - D-Bus
    - Single-node middleware replacing CORBA (GNOME) and DCOP (KDE)
    - Software bus abstraction (end-points communicating over a shared virtual channel)
      - System bus vs. session bus
      - End-points identified by a component string and unique connection (instance) name
      - Method calls and signals implemented on top of message passing
        - Synchronous one-to-one request-response (libdbus)
          - UNIX domain sockets, TCP sockets, kdbus (abandoned), BUS1
        - Asynchronous publish/subscribe (dbus-daemon)

Classical IPC

- **IPC abstractions**
  - Doors
    - Synchronous remote procedure call
    - Originally implemented for Spring, later ported to Solaris
    - Request and reply buffer, request and reply list of file descriptors
  - Binder (OpenBinder)
    - Middleware and component framework (similar to Microsoft COM), originally implemented for BeOS, now used by Android
    - Synchronous remote method invocation
      - Custom kernel IPC mechanism
        - Method invocation implemented as thread migration
          - `BINDER_WRITE_READ` ioctl with a request and reply buffer
        - Object reference tracking
  - Windows Dynamic Data Exchange, Object Linking and Embedding, Component Object Model
    - Based on [Advanced] Local Procedure Call ([A]LPC)

Mach IPC

- **Prototypical microkernel asynchronous message passing**
  - Ports
    - Receive end-points and associated message queues
  - Port rights
    - Client capabilities for accessing a port (send, receive, send-once)
      - Only a single server can have a receive right
    - Each task has an initial set of port rights
      - Communicating with the kernel, etc.
  - Tagged message structure
    - Kernel enforces type correctness
    - Port rights can be also passed
    - Timeouts

Mach IPC

- **“When poor implementation casts a shadow on the whole idea”**
  - IPC overhead of 50 % compared to monolithic UNIX
    - With a single UNIX server
    - Root causes
      - Complex non-optimized kernel-side code
        - Tagged data type evaluation, handling of timeouts, etc.
        - Dynamic data structures
          - But the implementation only uses linked lists
        - Excessive cache footprint
      - Asynchronicity rarely used in practice
        - User space tasks (mostly ported from UNIX) use synchronous communication and blocking I/O

The Era of Synchronous IPC

- **L3 (1988), L4 (1993) by Jochen Liedtke**
  - IPC overhead of 3 % compared to monolithic UNIX
    - With a single UNIX server
    - Single IPC call overhead comparable to single syscall overhead in UNIX (approx. 20 times faster than on Mach)
  - Synchronous IPC
    - Explicit client/server rendez-vous and thread migration
      - No need for *full* context switch (address space switch is sufficient)
      - No buffering, no scheduling, data passed mostly directly in registers
    - Highly target-optimized implementation
      - Small working set, cache-friendly code
      - No complex algorithms or dynamic data structures

The Era of Synchronous IPC

- **L3 (1988), L4 (1993) by Jochen Liedtke**
  - Drawbacks
    - Non-portable microkernel (by design)
      - Poor code readability and maintainability
      - Preoccupation with single-threaded performance conflicts with other goals (e.g. throughput)
    - Design issues of synchronous IPC
      - Unresponsive server blocks the client indefinitely
        - Originally solved using timeouts (in hindsight not a great solution)
      - Asynchronous communication on top of synchronous IPC
        - Abstraction inversion anti-pattern (i.e. requires multithreading)
      - Scalability suffers on modern massively parallel architectures

The Return of Asynchronous IPC

- **The best of both worlds**
  - Synchronous IPC still superior in specific use cases
    - Synchronous blocking semantics, single-core communication
  - Asynchronous IPC reasonably simple, cache-friendly with fast-path kernel code
    - Bounded kernel buffers (additional buffering possible on the client user space side)
    - Intelligent bookkeeping data structures (hash tables, trees)
    - Simple IPC message structure (only integer payload that fits into registers)
      - Additional semantics for memory copying and memory sharing possible
    - Possibility to build rich abstractions in user space
      - Actors, agents, continuations, futures, promises

HelenOS IPC

- **Basic design**
  - Asynchronous message passing over uni-directional connections
    - 6-integer payload (1st integer interpreted as interface/method ID)
    - Bounded kernel buffers
    - Every message paired with a reply (6-integer return value)
    - New connections established via existing connections (capabilities)
      - Security policy delegated to the connection brokers
      - Every client initially connected to the Naming Service (default broker)
    - Message forwarding (recursive)
    - Kernel events and hardware interrupts converted to IPC messages (no reply)

HelenOS IPC

- **Kernel API**
  - Global method IDs with special semantics
    - `IPC_M_CONNECTION_CLONE` (clone a connection capability from the client to the server)
    - `IPC_M_CONNECT_TO_ME` (establish a callback connection)
    - `IPC_M_CONNECT_ME_TO` (establish a new connection)
      - When forwarded, the connection is potentially established to the next receiver
        - Broker (Naming Service, Location Service, Device Manager, VFS, etc.) connects the client to the target server
    - `IPC_M_SHARE_IN` / `IPC_M_SHARE_OUT` (receive/send a shared virtual address space area)
    - `IPC_M_DATA_READ` / `IPC_M_DATA_WRITE` (receive/send bulk data)
    - `IPC_M_STATE_CHANGE_AUTHORIZE` (update a server state on behalf of a different client)
      - Three-way handshake
    - `IPC_M_PHONE_HUNGUP` (connection close)

HelenOS IPC

- **User space API**
  - Async framework
    - Goal: Writing single-threaded sequential client code that makes effective use of the asynchronous IPC
      - User space-scheduled cooperative threads (fibrils)
        - Efficient parallelism (preempted only when blocking on waiting for IPC replies)
    - Abstracting the low-level IPC connections into sessions
      - Each session can have a different threading model
    - Abstracting the atomic low-level IPC messages into logical exchanges
      - Easily implementing complex communication protocols

HelenOS IPC

```
async_exch_t *ns_exch = async_exchange_begin(session_ns);

async_sess_t *sess =
    async_connect_me_to_iface(ns_exch, INTERFACE_VFS, SERVICE_VFS, 0);

async_exchange_end(ns_exch);

async_exch_t *exch = async_exchange_begin(sess);

ipc_call_t answer;
aid_t req =
    async_send_3(exch, VFS_IN_OPEN, lflags, oflags, 0, &answer);

async_data_write_start(exch, path, path_size);

async_exchange_end(exch);

// Do some other useful work in the meantime

sysarg_t rc;
async_wait_for(req, &rc);

if (rc == EOK)
    fd = (int) IPC_GET_ARG1(answer);
```

Synchronization Mechanisms

- **Mutual exclusion**
  - Locks, semaphores, condition variables, etc.
    - Based on atomic test-and-set operations
  - Temporal separation, intuitive semantics, well-known characteristics
  - Overhead, restriction of concurrency, blocking
  - Adverse effects
    - Convoying, priority inversion, starvation, deadlock

Synchronization Mechanisms

- **Non-blocking mechanisms**
  - Lock-free data structures, transactional memory, hazard pointers, read-copy-update, etc.
    - Based on atomic read-modify-write operations
  - Logical separation vs. eventual consistency
  - No restriction on concurrency
    - Especially suitable for concurrent workloads (e.g. asynchronous IPC)
  - Less intuitive semantics, surprising characteristics

Non-blocking Taxonomy

- **Wait-freedom**
  - Guaranteed system-wide progress and starvation-freedom (all operations are finitely bounded)
  - Wait-freedom algorithms always exist [1], but the performance of general methods is usually inferior to blocking algorithms
  - Wait-free queue by Kogan & Petrank [2]
- **Lock-freedom**
  - Guaranteed system-wide progress, but individual threads can starve
  - Four phases: Data operation, assisting obstruction, aborting obstruction, waiting
- **Obstruction-freedom**
  - Guaranteed single thread progress if isolated for a bounded time (obstructing threads need to be suspended)

Read-Copy-Update

- **Family of generic non-blocking synchronization mechanisms**
  - Many different implementations with various characteristics
  - Targeting read-mostly pointer-based data structures with immutable values
    - Useful for many practical data structures (e.g. linked lists, hash tables, etc.)
    - Unlimited number of readers without blocking (running concurrently with other readers and writers)
      - Little to no overhead on the reader side (smaller than taking an uncontended lock)
      - Readers have to tolerate “stale” data and late updates
      - Readers have to observe “safe” access patterns
    - Synchronization among writers out of scope of the mechanism
      - RCU only guarantees consistency between readers and writers
    - Optional provisions for asynchronous reclamation

Read-Copy-Update

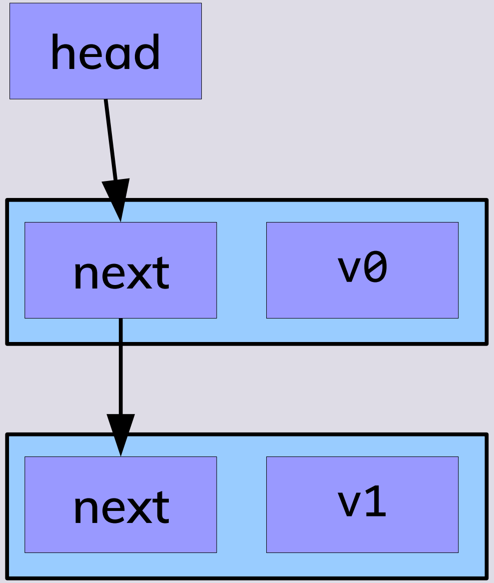
- **Read-side critical section**
  - Delimited by `read_lock()` and `read_unlock()` non-blocking methods
    - Protected data cannot be referenced outside of the critical section
  - Safe `access()` methods for reading pointers
    - Each pointer can be read at most once in a critical section
    - No restriction on reading the pointed values
- **Quiescent state**
  - A scheduling entity (thread, CPU, etc.) being outside its critical section
- **Grace period**
  - A point in time when all scheduling entities have passed through a quiescent state (at least once)

Read-Copy-Update

- **Synchronous write-side update**
  - Atomically unlinking an old element
  - Running the `synchronize()` method
    - Blocks until a grace period elapses
      - All readers pass their quiescent state (i.e. they no longer reference the unlinked data)
  - Possibly reclaiming/freeing the unlinked data
  - Inserting a new element using a safe `assign()` method
    - Avoiding store reordering on architectures with weak memory ordering

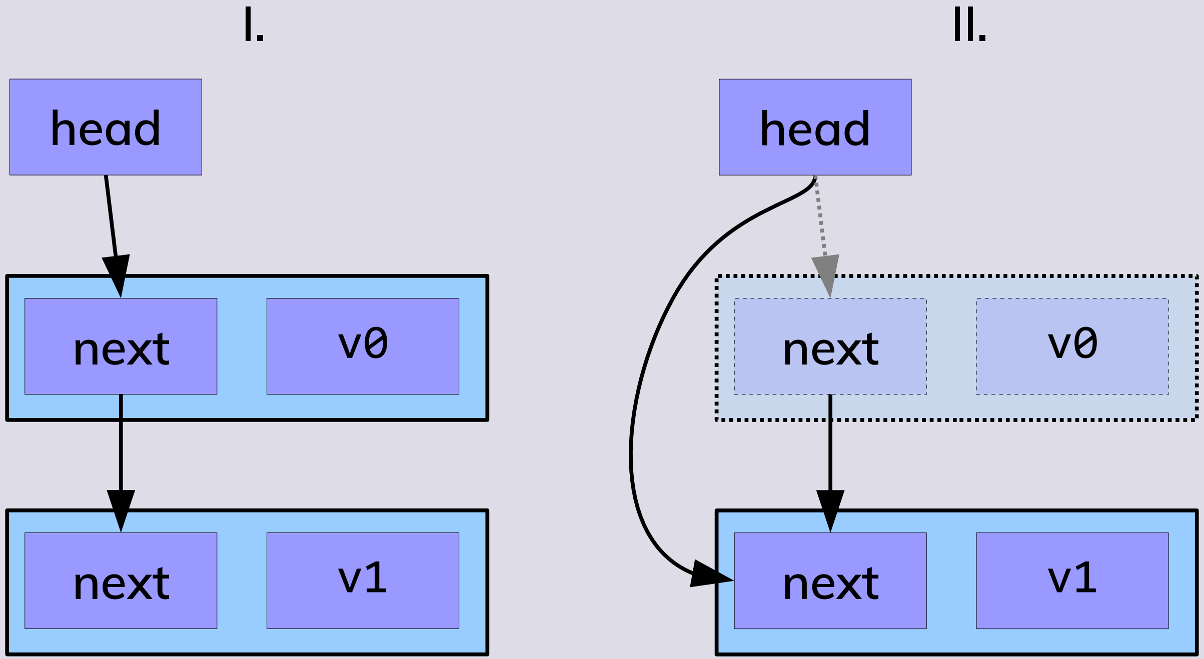
Synchronous Update Example

I.



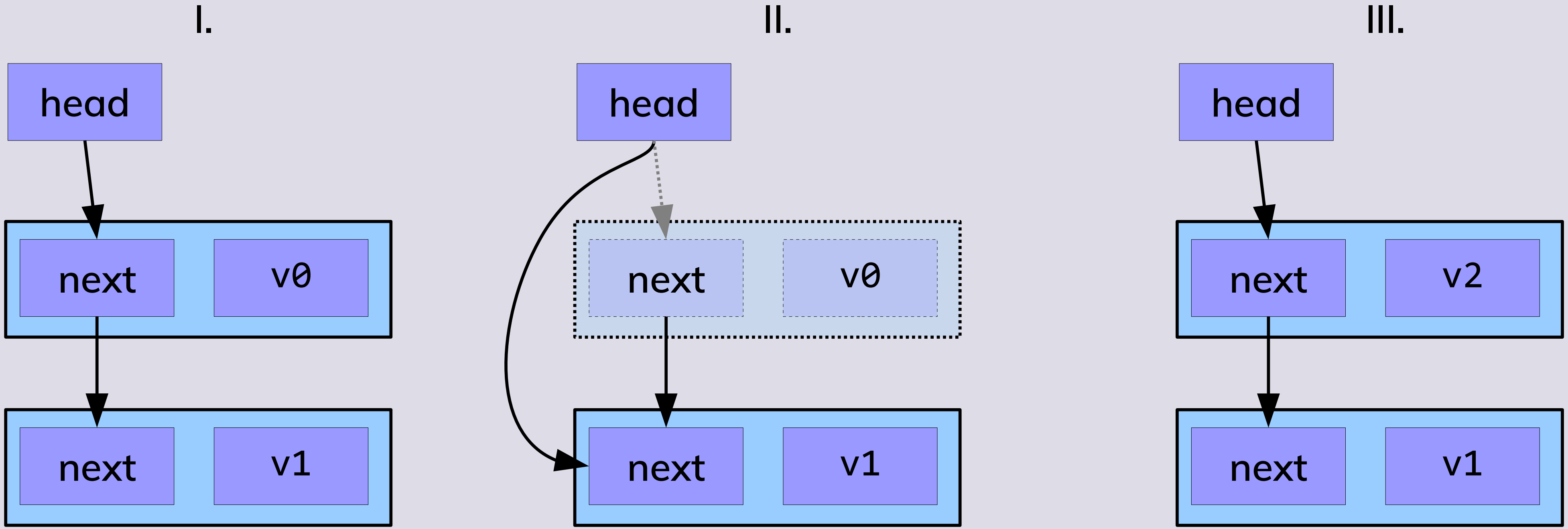
Atomic pointer update to remove the element with `v0` from the list

Synchronous Update Example



Blocking on `synchronize()`
During the grace period, preexisting readers
can still access the “stale” element with `v0`

Synchronous Update Example



No reader can reference the element with `v0` anymore,
it can be safely reclaimed
New element with `v2` can be atomically inserted

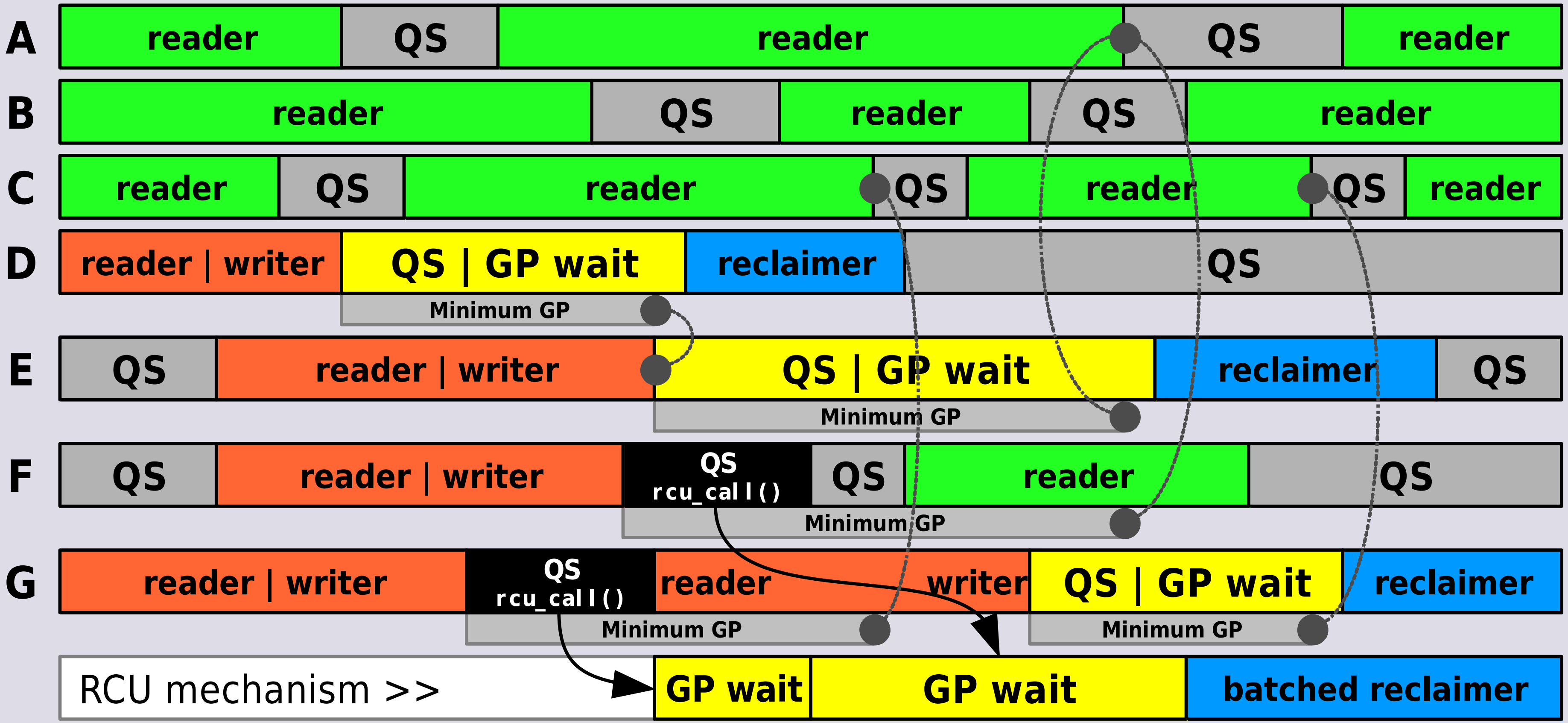
Read-Copy-Update

- **Asynchronous write-side update**
  - Using a `call()` method
    - Non-blocking operation registering a callback
    - Callback is executed after a grace period elapses
  - Using a `barrier()` method
    - Waiting for all queued asynchronous callbacks to finish

Grace Period Detection

- **Cornerstone of any RCU implementation**
  - Balancing the trade-off between precision and overhead
    - Any extension of a grace period is also a grace period
    - Long (imprecise) grace periods
      - Blocking synchronous writers for a longer time
      - Increasing memory usage due to unreclaimed “stale” data
    - Short (precise) grace periods
      - Increasing overhead on the reader side
        - More heavy-weight operations needed (memory barriers, atomics)
  - Looking for *naturally occurring quiescent states*
    - Events that automatically guarantee the end of a critical section (for the given RCU implementation)
      - Context switch, exception (timer tick), exit to user space, etc.

Grace Period Detection



A
reader
QS
reader
QS
reader

B
reader
QS
reader
QS
reader

C
reader
QS
reader
QS
reader
QS
reader

D
reader | writer
QS | GP wait
reclaimer
QS
Minimum GP

E
QS
reader | writer
QS | GP wait
reclaimer
QS
Minimum GP

F
QS
reader | writer
QS
`rcu_call()`
QS
reader
QS
Minimum GP

G
reader | writer
QS
`rcu_call()`
reader
writer
QS | GP wait
reclaimer
Minimum GP
Minimum GP

RCU mechanism >>
GP wait
GP wait
batched reclaimer

[3]

References

[1] Herlihy M. P.: *Impossibility and universality results for wait-free synchronization*, in Proceedings of the 7th Annual ACM Symposium on Principles of Distributed Computing, ACM, 1988

[2] Kogan A., Petrank E.: *Wait-free queues with multiple enqueuers and dequeuers*, in Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming, ACM, 2011

[3] Podzimek A., Děcký M., Bulej L., Tůma P.: *A Non-Intrusive Read-Copy-Update for UTS*, in Proceedings of the 18th IEEE International Conference on Parallel and Distributed Systems (ICPADS 2012), IEEE Computer Society, 2012

D3S

Thank you!

Questions?