barkhausen institut

Microkernel Construction

Threads

Nils Asmussen

04/23/2026

- **Threads**
  - **Definition and Basics in NOVA**
  - Thread Switch
- Synchronization
- FPU Handling

What is a Thread?

- An independent flow of control inside an address space
- Communicates with other threads using IPC
- Characterized by a set of registers and the thread state
- Dispatched by the kernel according to a defined schedule

What is a Thread?

- An independent flow of control inside an address space
- Communicates with other threads using IPC
- Characterized by a set of registers and the thread state
- Dispatched by the kernel according to a defined schedule

- Each thread is bound to one core at a time
- Only one thread per core is running at one point in time
- With n cores, n threads can run at once
- All other threads are inactive, waiting inside the kernel

Implementation in NOVA

Execution Context

- Register state
- Continuation
- Address Space (PD)
- UTCB (message buffer)
- IPC partner
- FPU state
- prev/next pointer

Scheduling Context

- Execution Context
- Priority
- Budget
- Remaining budget
- prev/next pointer

Thread Variants

Global Thread

- Needs a scheduling context, i.e., CPU time, to execute
- Causes exception on startup to let creator set register state

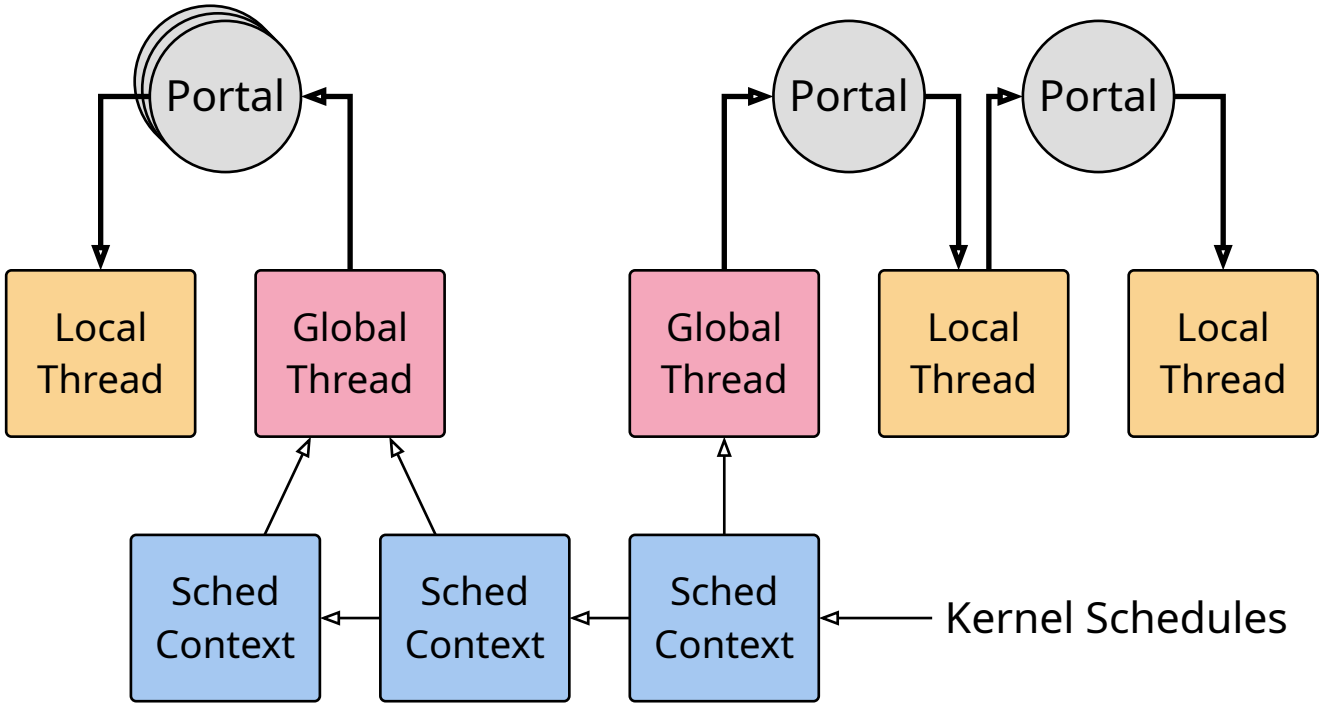
Local Thread

- Has no scheduling context
- Are only used to handle portal calls
- Waits in the kernel until someone called an associated portal

Portals

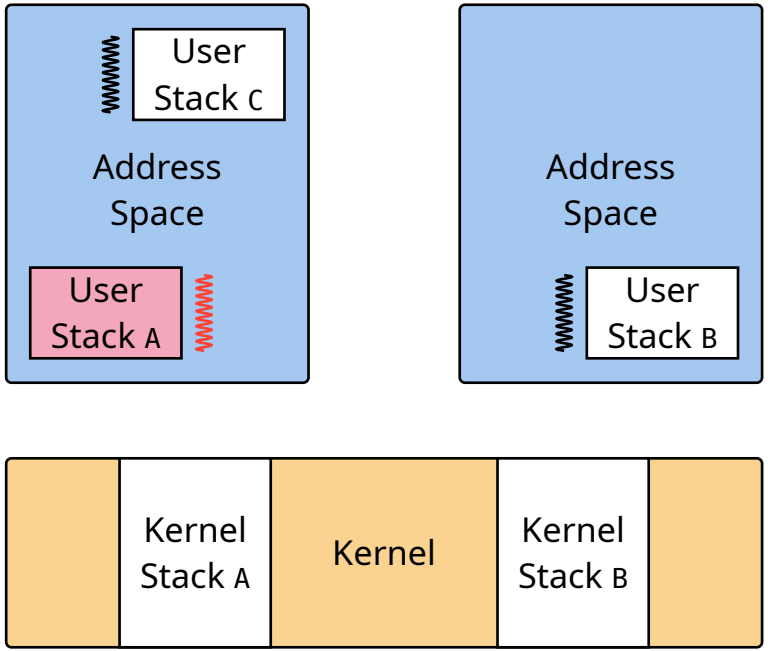
- A portal is an IPC endpoint
- Executed by local threads
- CPU time is donated from caller
- Called via system call
- Message is transferred from sender UTCB to receiver UTCB

Overview


Portal
Portal
Portal
Local Thread
Global Thread
Global Thread
Local Thread
Local Thread
Sched Context
Sched Context
Sched Context
Kernel Schedules

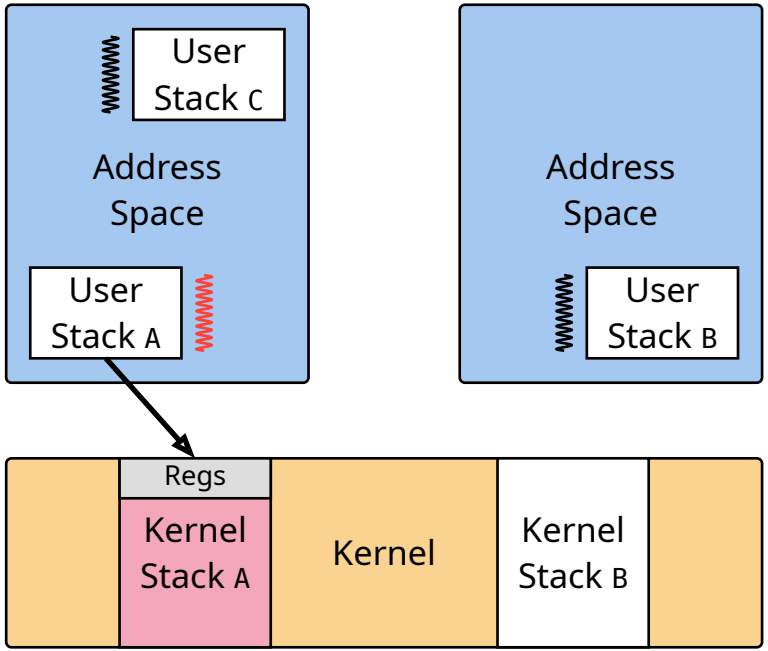
- **Threads**
  - Definition and Basics in NOVA
  - **Thread Switch**
- Synchronization
- FPU Handling

Thread Switch: Conventional



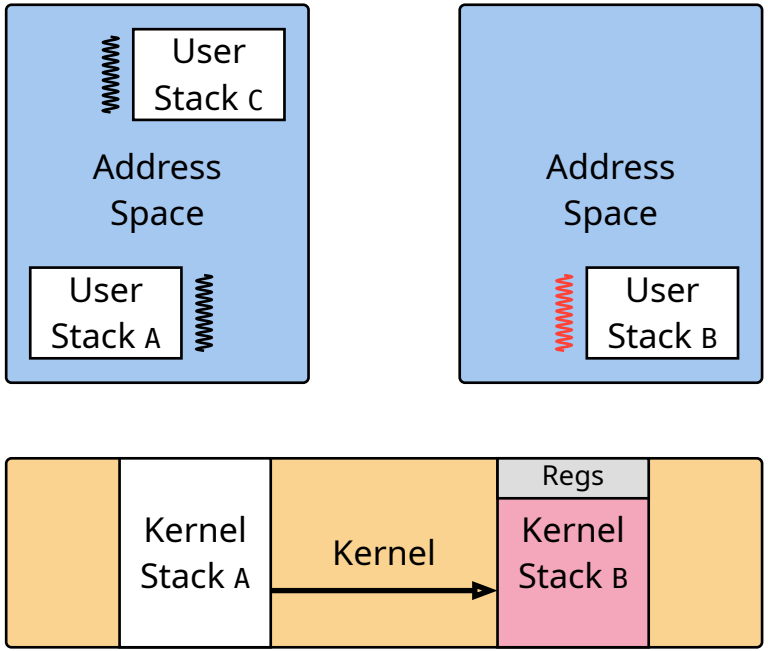
- User thread A is running

Thread Switch: Conventional



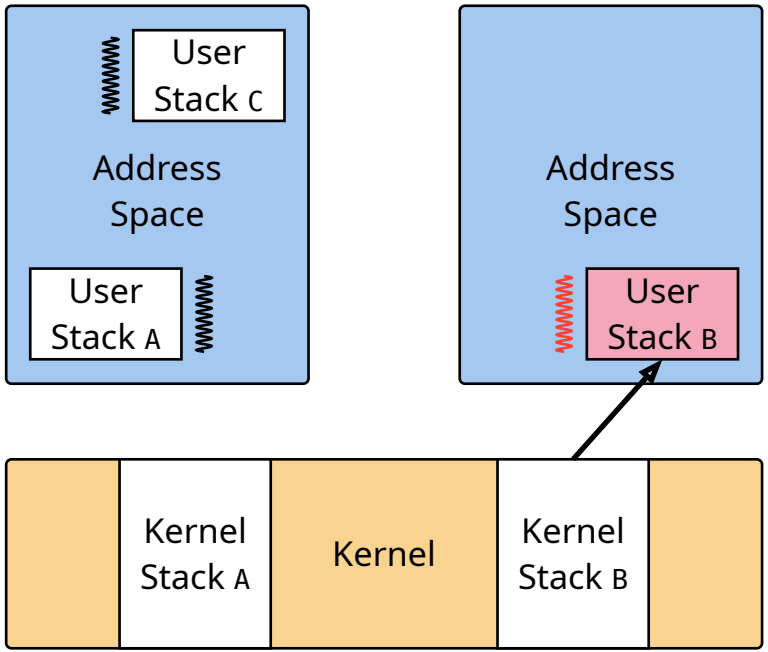
- User thread A is running
- Timer interrupt occurs
- Kernel saves register state onto kernel stack A

Thread Switch: Conventional



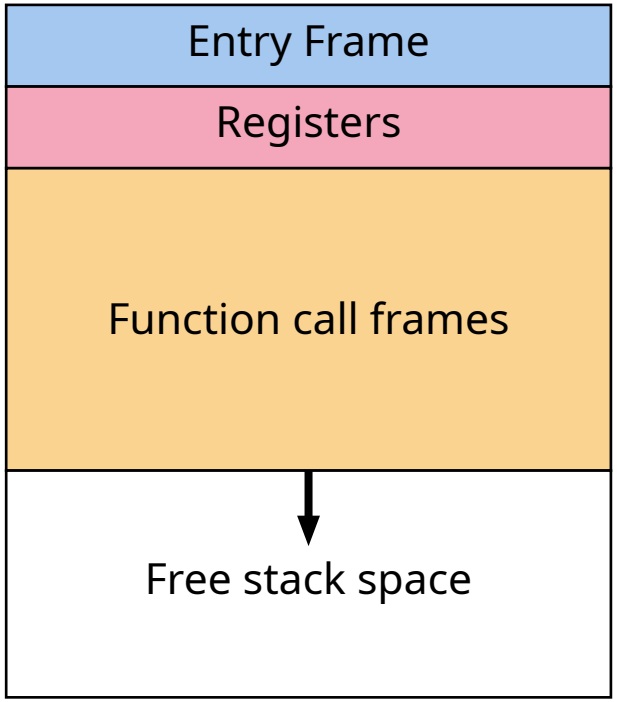
- User thread A is running
- Timer interrupt occurs
- Kernel saves register state onto kernel stack A
- Kernel switches to thread B
- Restores register state from kernel stack B

Thread Switch: Conventional



- User thread A is running
- Timer interrupt occurs
- Kernel saves register state onto kernel stack A
- Kernel switches to thread B
- Restores register state from kernel stack B
- Resumes user thread B

Thread Switch: Conventional Kernel Stack



`SS`, `RSP`, `RFLAGS`, `CS`, `RIP`

`RAX`, `RCX`, `RDX`, `RBX`, `RSP`, `RBP`, `RSI`, `RDI`, `R8`, ..., `R15`

- function arguments
- function return address
- callee-saved registers: `RBX`, `RBP`, `R12`, ..., `R15`
- local variables

Thread Switch: Conventional In-Kernel Switch

Implementation of `thread_switch(reg_t *old, reg_t *new)`

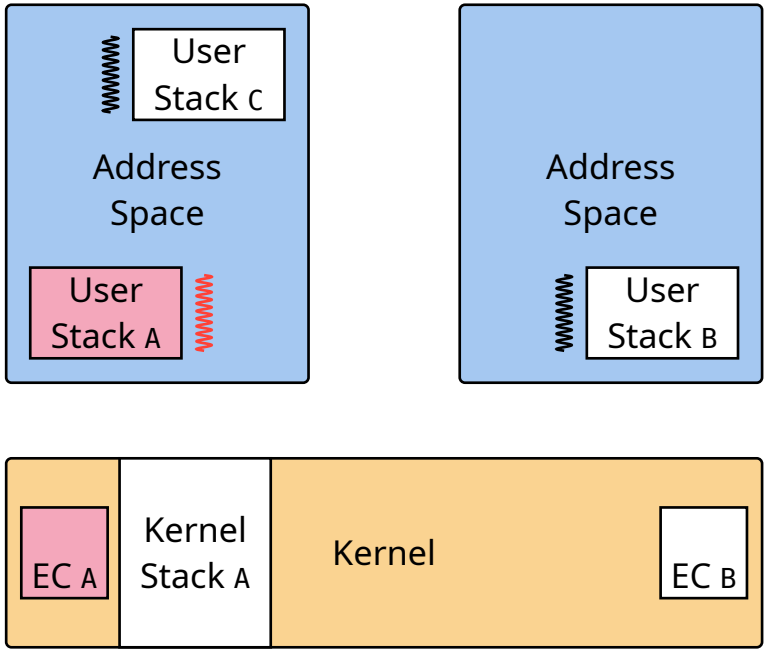
```asm
thread_switch:
    mov     %rbx,  0(%rdi)
    mov     %rsp,  8(%rdi)
    mov     %rbp, 16(%rdi)
    mov     %r12, 24(%rdi)
    mov     %r13, 32(%rdi)
    mov     %r14, 40(%rdi)
    mov     %r15, 48(%rdi)
    # load and save rflags
    pushfq
    popq    56(%rdi)
    mov     48(%rsi), %r15
    mov     40(%rsi), %r14
    mov     32(%rsi), %r13
    mov     24(%rsi), %r12
    mov     16(%rsi), %rbp
    mov      8(%rsi), %rsp
    mov      0(%rsi), %rbx
    # restore rflags
    pushq   56(%rsi)
    popfq
    ret
```

Thread Switch: Conventional In-Kernel Switch

Implementation of `thread_switch(reg_t *old, reg_t *new)`

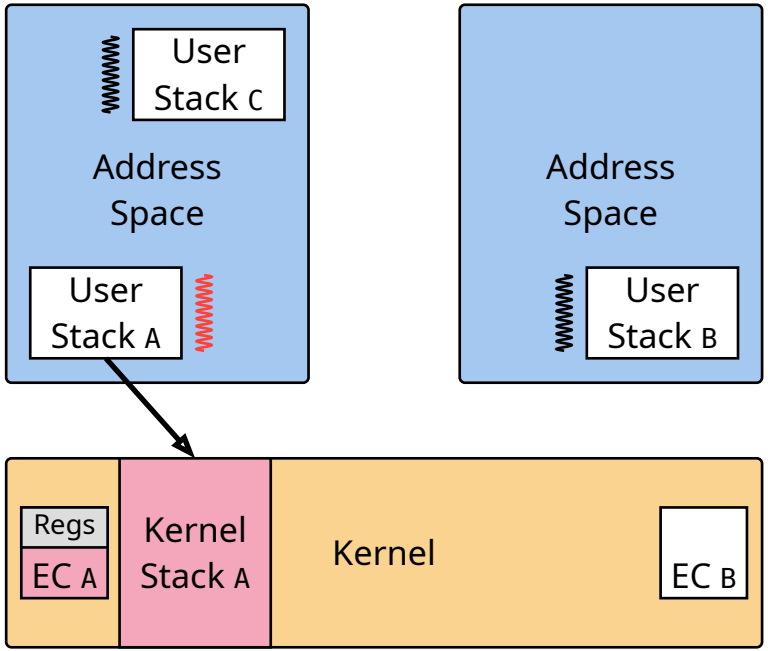
```asm
thread_switch:
    mov     %rbx,  0(%rdi)
    mov     %rsp,  8(%rdi)
    mov     %rbp, 16(%rdi)
    mov     %r12, 24(%rdi)
    mov     %r13, 32(%rdi)
    mov     %r14, 40(%rdi)
    mov     %r15, 48(%rdi)
    # load and save rflags
    pushfq
    popq    56(%rdi)
    mov     48(%rsi), %r15
    mov     40(%rsi), %r14
    mov     32(%rsi), %r13
    mov     24(%rsi), %r12
    mov     16(%rsi), %rbp
    mov      8(%rsi), %rsp
    mov      0(%rsi), %rbx
    # restore rflags
    pushq   56(%rsi)
    popfq
    ret
```

Thread Switch: Continuation Style



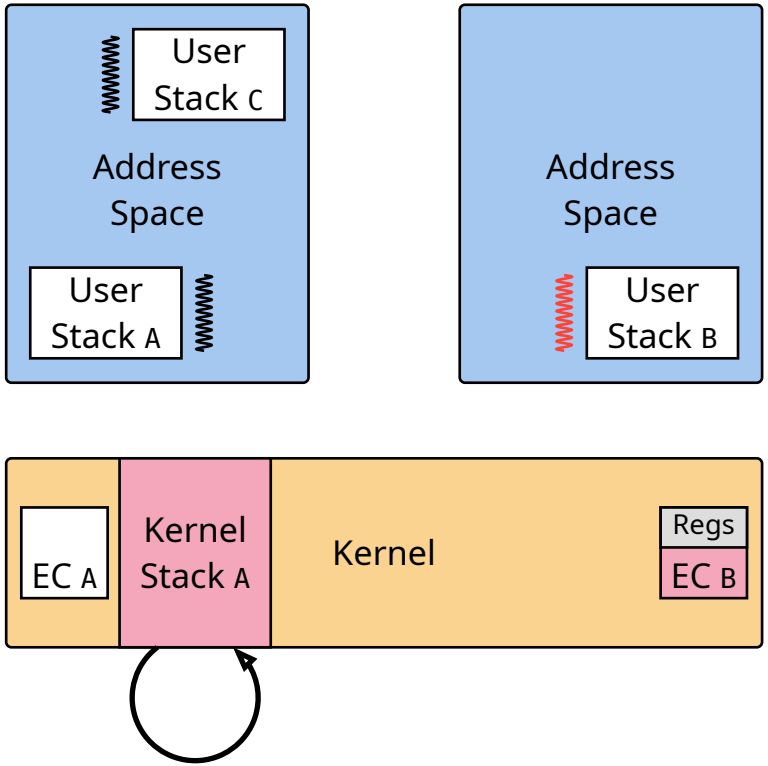
- User thread A is running

Thread Switch: Continuation Style



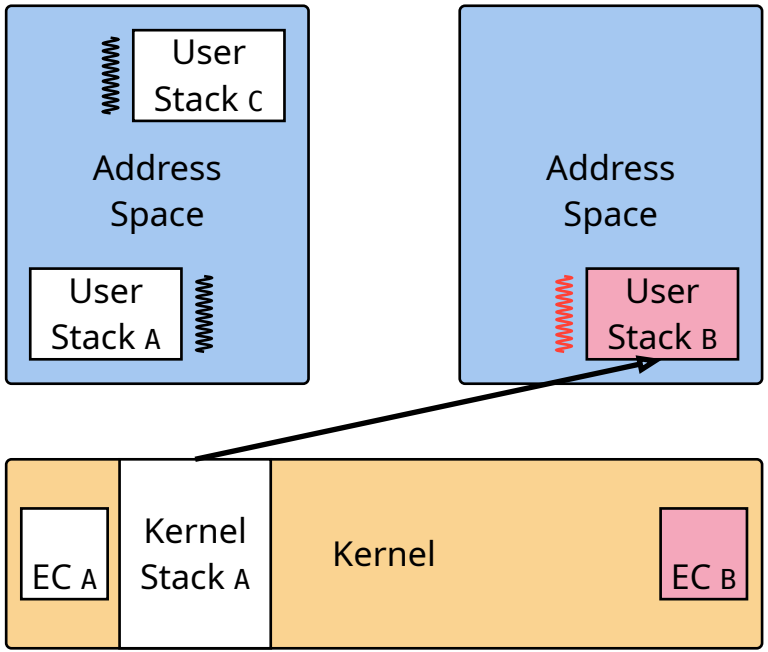
- User thread A is running
- Timer interrupt occurs
- Kernel saves register state into thread A's TCB

Thread Switch: Continuation Style



- User thread A is running
- Timer interrupt occurs
- Kernel saves register state into thread A's TCB
- Kernel switches to thread B, starting with fresh kernel stack

Thread Switch: Continuation Style



- User thread A is running
- Timer interrupt occurs
- Kernel saves register state into thread A's TCB
- Kernel switches to thread B, starting with fresh kernel stack
- Restores register state from thread B's TCB
- Resumes user thread B

Thread Switch: In-Kernel Switch

- Traditional kernels save/restore the current CPU state
- Each thread has own stack → stack frames are kept
- In NOVA, stack frames and CPU state are lost

Thread Switch: In-Kernel Switch

- Traditional kernels save/restore the current CPU state
- Each thread has own stack → stack frames are kept
- In NOVA, stack frames and CPU state are lost

Example in-kernel switch: `sys_call`

C++

```
1  current->cont = ret_user_sysexit;
2  current->set_partner (ec);
3  ec->cont = recv_user;
4  ec->regs.set_ip (pt->ip);
5  ec->regs.set_pt (pt->id);
6  ec->make_current();
```

Thread Switch: In-Kernel Switch

Switching to an Ec

C++

```
1  void Ec::make_current() {
2    current = this;
3    Tss::run.sp0 = reinterpret_cast<mword>(exc_regs());
4    pd->make_current();
5    asm volatile (
6      "mov %0, %%rsp;"
7      "jmp *%1;"
8      :
9      : "g" (CPU_LOCAL_STCK + PAGE_SIZE),
10       "rm" (cont)
11     : "memory"
12   );
13   UNREACHED;
14 }
```

Continuation Style: Pros and Cons

- 🙁 Context switches more restricted
  - thread loses all state
  - switches only possible at specific points
- 🙂 Less kernel memory usage
- 🙂 Performance
  - less cache pressure
  - less TLB pressure

- Threads
- **Synchronization**
  - **Basic Concepts**
  - Blocking Synchronization
  - Lock-Free Synchronization
  - Wait-Free Synchronization
- FPU Handling

Synchronization Properties

Granularity

- Fine-grained synchronization allows high concurrency
- Coarse-grained synchronization reduces overhead

Synchronization Properties

Granularity

- Fine-grained synchronization allows high concurrency
- Coarse-grained synchronization reduces overhead

Fairness

- Avoid starvation of threads when entering critical sections

Synchronization Properties

Granularity

- Fine-grained synchronization allows high concurrency
- Coarse-grained synchronization reduces overhead

Fairness

- Avoid starvation of threads when entering critical sections

Lock holder preemption

- Run lock holder until completion of critical section

Synchronization Properties

Granularity

- Fine-grained synchronization allows high concurrency
- Coarse-grained synchronization reduces overhead

Fairness

- Avoid starvation of threads when entering critical sections

Lock holder preemption

- Run lock holder until completion of critical section

Multi-processor synchronization

- Higher synchronization effort

Preemptibility and Concurrency

No preemptibility (example: seL4)

- Only one thread executes code inside the kernel
- Big kernel lock

Preemptibility and Concurrency

No preemptibility (example: seL4)

- Only one thread executes code inside the kernel
- Big kernel lock

Restricted preemptibility (example: NOVA)

- Allowing preemption at preemption points
- Preemption point defines consistent state

Preemptibility and Concurrency

No preemptibility (example: seL4)

- Only one thread executes code inside the kernel
- Big kernel lock

Restricted preemptibility (example: NOVA)

- Allowing preemption at preemption points
- Preemption point defines consistent state

High preemptibility (example: Linux)

- Thread can be preempted at (almost) any point in time
- State has to be consistent for all possible preemptions

Synchronization Primitives

Disabling Interrupts

- Not multi-core safe
- Core-local data structures, kernel entry/exit

Synchronization Primitives

Disabling Interrupts

- Not multi-core safe
- Core-local data structures, kernel entry/exit

Blocking synchronization

- Semaphores
- Spin locks, Ticket locks, etc.

Synchronization Primitives

Disabling Interrupts

- Not multi-core safe
- Core-local data structures, kernel entry/exit

Blocking synchronization

- Semaphores
- Spin locks, Ticket locks, etc.

Non-blocking synchronization

- Lock-free: some thread makes progress
- Wait-free: all threads make progress

- Threads
- **Synchronization**
  - Basic Concepts
  - **Blocking Synchronization**
  - Lock-Free Synchronization
  - Wait-Free Synchronization
- FPU Handling

Blocking Synchronization

Sleeping locks: Semaphores

- Suspend thread when lock is already taken
- Wake up thread when lock is released
- High overhead: suitable for long critical sections
- Not used by NOVA; only in user space

Blocking Synchronization

Sleeping locks: Semaphores

- Suspend thread when lock is already taken
- Wake up thread when lock is released
- High overhead: suitable for long critical sections
- Not used by NOVA; only in user space

Spinning locks: Spinlocks

- Waiting thread spins until lock is released
- Burns CPU time: suitable for short critical sections

Spinlock Variants

Test and set lock

- Try to acquire lock, spin if it fails (writing)

Spinlock Variants

Test and set lock

- Try to acquire lock, spin if it fails (writing)

Test and test and set lock

- Try to acquire lock only if free
- Otherwise spin (reading)
- Avoids cache-line bouncing

Spinlock Variants

Test and set lock

- Try to acquire lock, spin if it fails (writing)

Test and test and set lock

- Try to acquire lock only if free
- Otherwise spin (reading)
- Avoids cache-line bouncing

Ticket lock

- Every thread acquires a ticket
- Tickets get access in FIFO order; guarantees fairness

Ticket Lock

NOVA's ticket lock implementation

C++

```
1  void lock() {
2      uint16 tmp = 0x100; // high: reserved ticket, low: active ticket
3      asm volatile ("     lock; xadd %0, %1;  "  // reserve ticket; tmp = val; val += 0x100
4                    "1:   cmpb %h0, %b0;      "  // compare our ticket with active ticket
5                    "     je 2f;              "  // we are next? -> done
6                    "     pause;              "  // relax CPU
7                    "     movb %1, %b0;       "  // update active ticket
8                    "     jmp 1b;             "  // retry
9                    "2:                       "  // enter critical section
10                   : "+Q" (tmp), "+m" (val) : : "memory");
11 }
12 void unlock() {
13     asm volatile ("incb %0" : "=m" (val) : : "memory");
14 }
```

Problems of Locks

- Overhead
- Deadlocks
- Lock-holder preemption
- Priority inversion

- Threads
- **Synchronization**
  - Basic Concepts
  - Blocking Synchronization
  - **Lock-Free Synchronization**
  - Wait-Free Synchronization
- FPU Handling

Lock-Free Synchronization

- Principle:
  - prepare data out of line
  - atomically try to exchange old data with new one
  - retry if failed
- Less overhead than locks
- Deadlock free
- *Some* thread makes progress
- Many threads may starve (see wait-free synchronization)

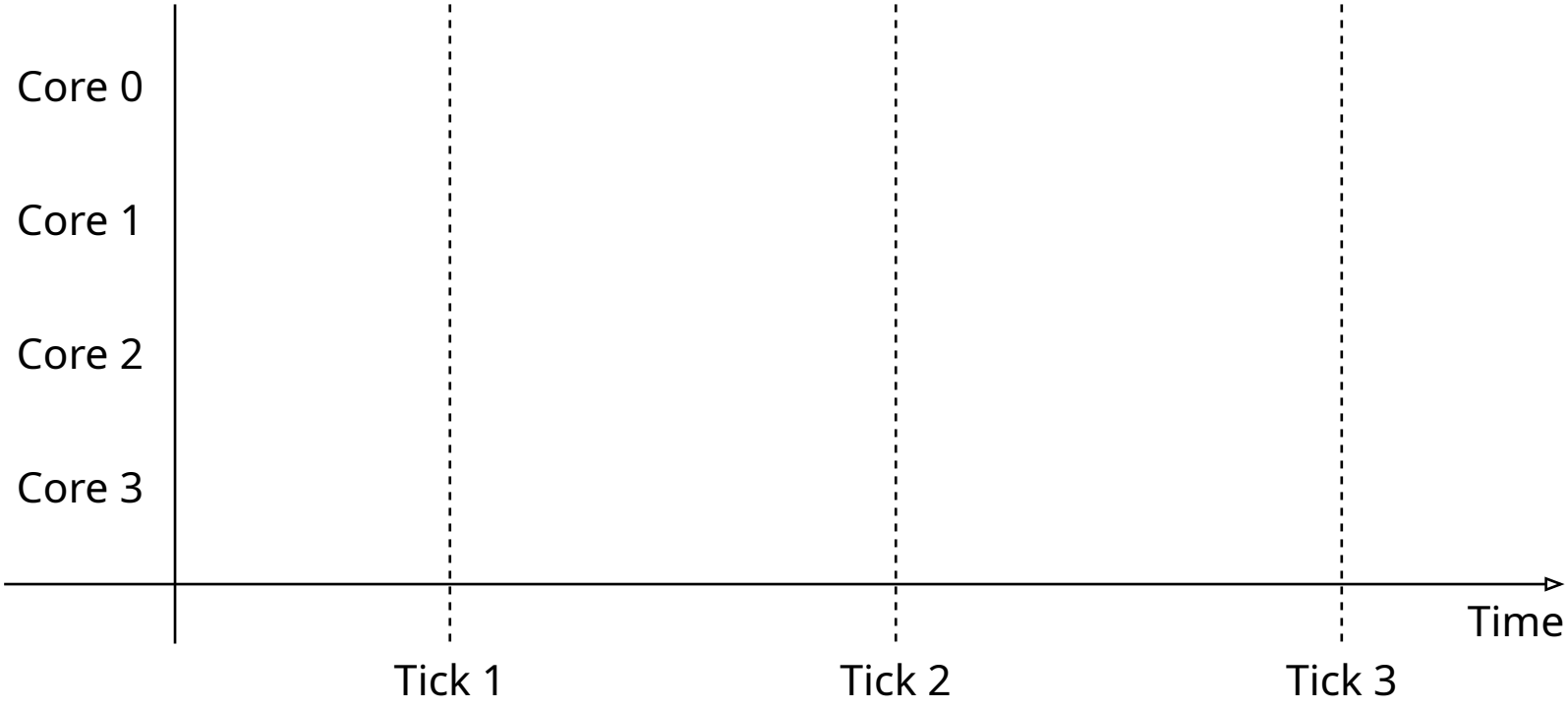
Example in NOVA: Deletion of Kernel Objects

- When revoking, kernel objects should be destructed
- But what if somebody accesses them at the same time?
- We could lock them during each access
- But this is expensive
- We don’t care that much when exactly they are destructed
- Can’t we destruct them if nobody accesses them anymore?

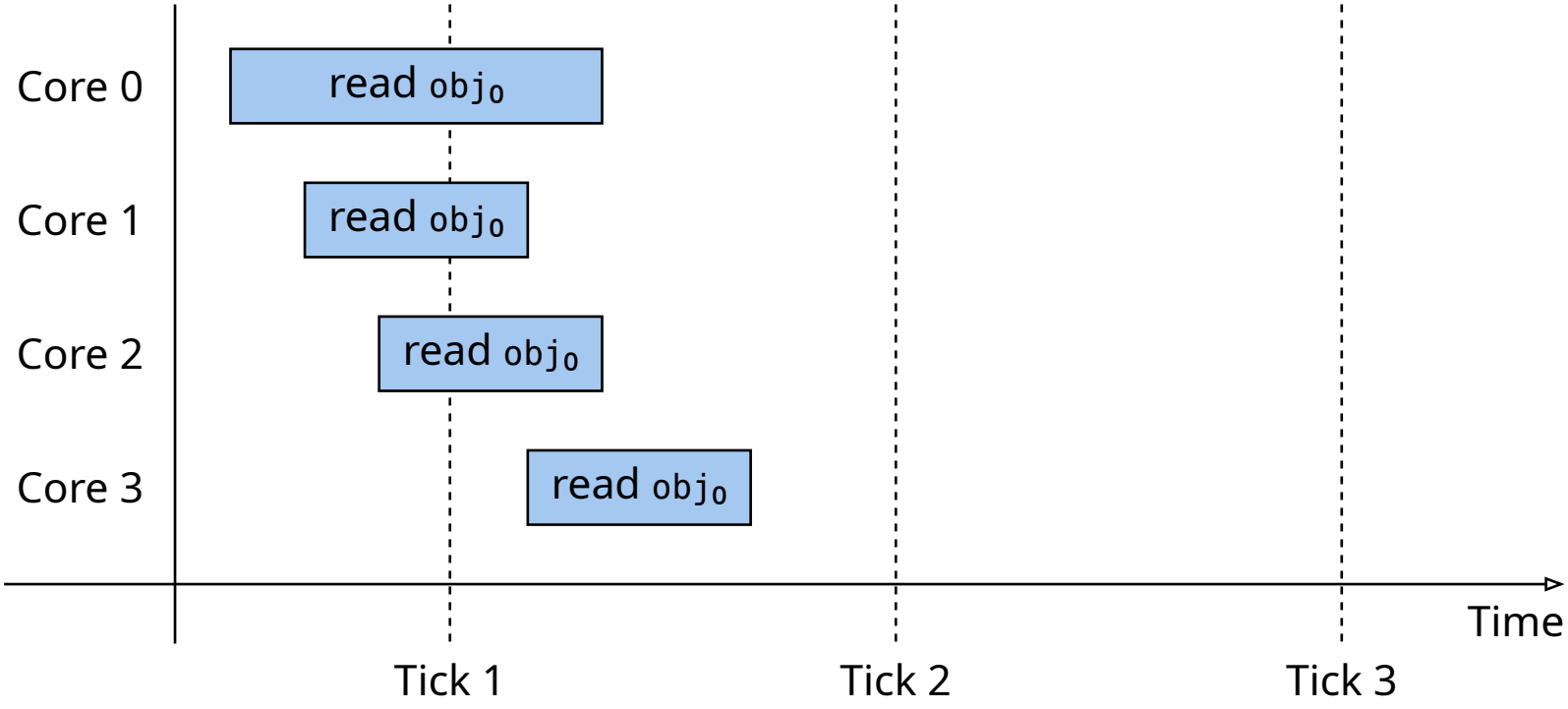
Read-Copy-Update (RCU)

- Basically: copy-on-write with lazy delete
- Don’t change objects, but copy them and change the copy
- Don’t delete objects immediately, but when readers are done
- In case of NOVA: no copy-on-write, but only lazy delete
- On revoke, object is removed first
- Then, the object is registered for deletion
- Timer IRQ is used to delete only if all readers are gone

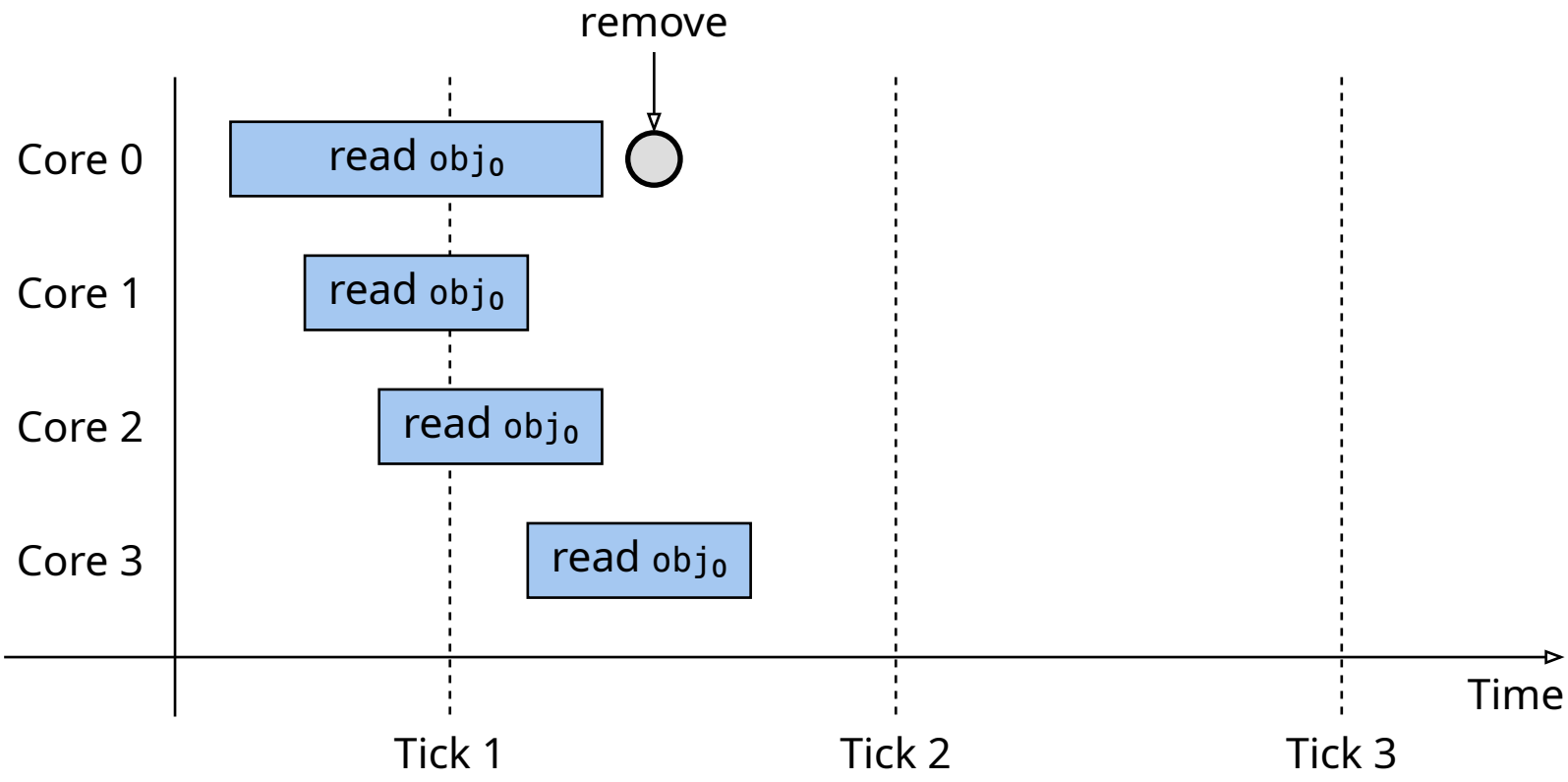
RCU In Action



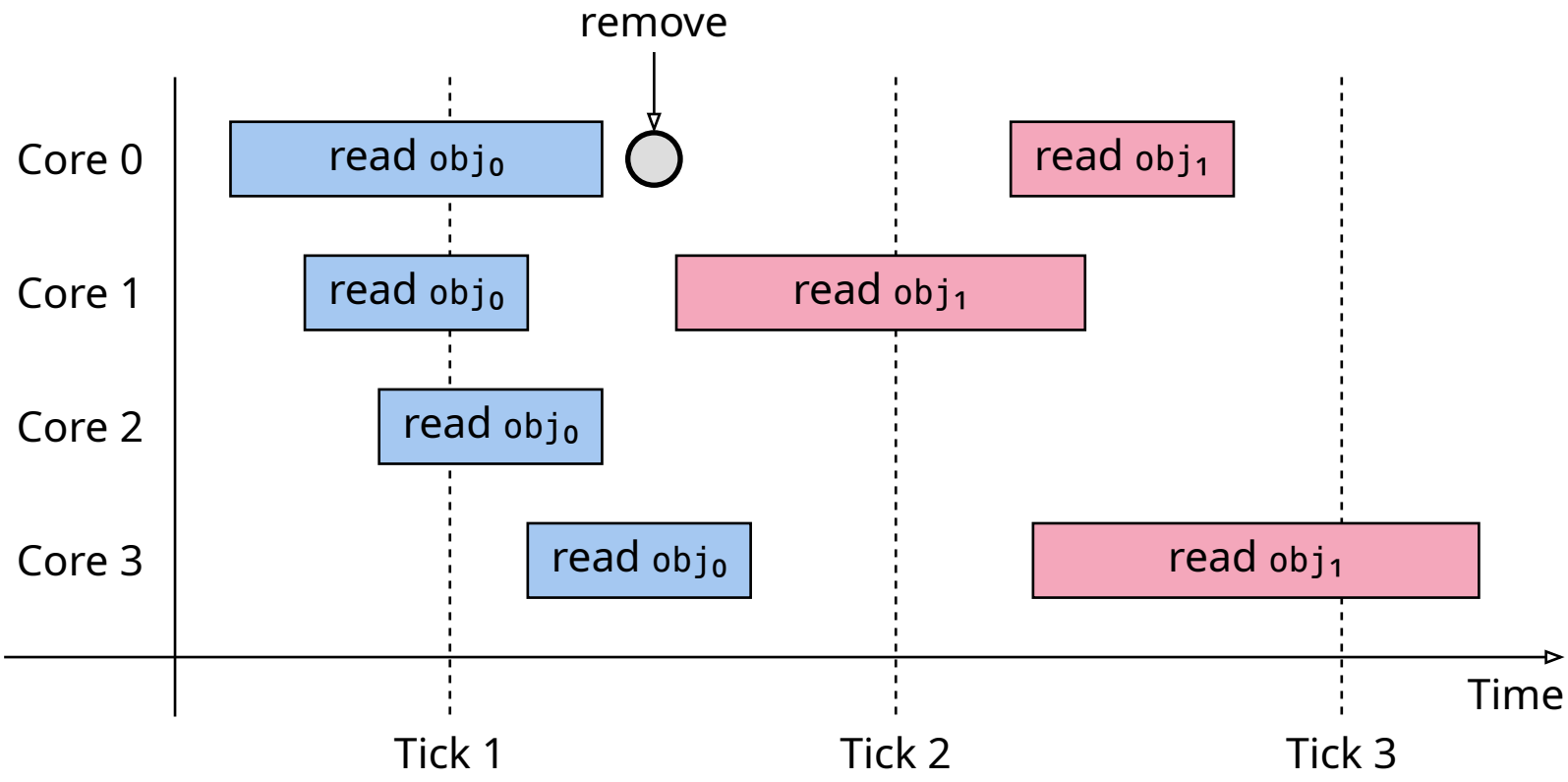
RCU In Action



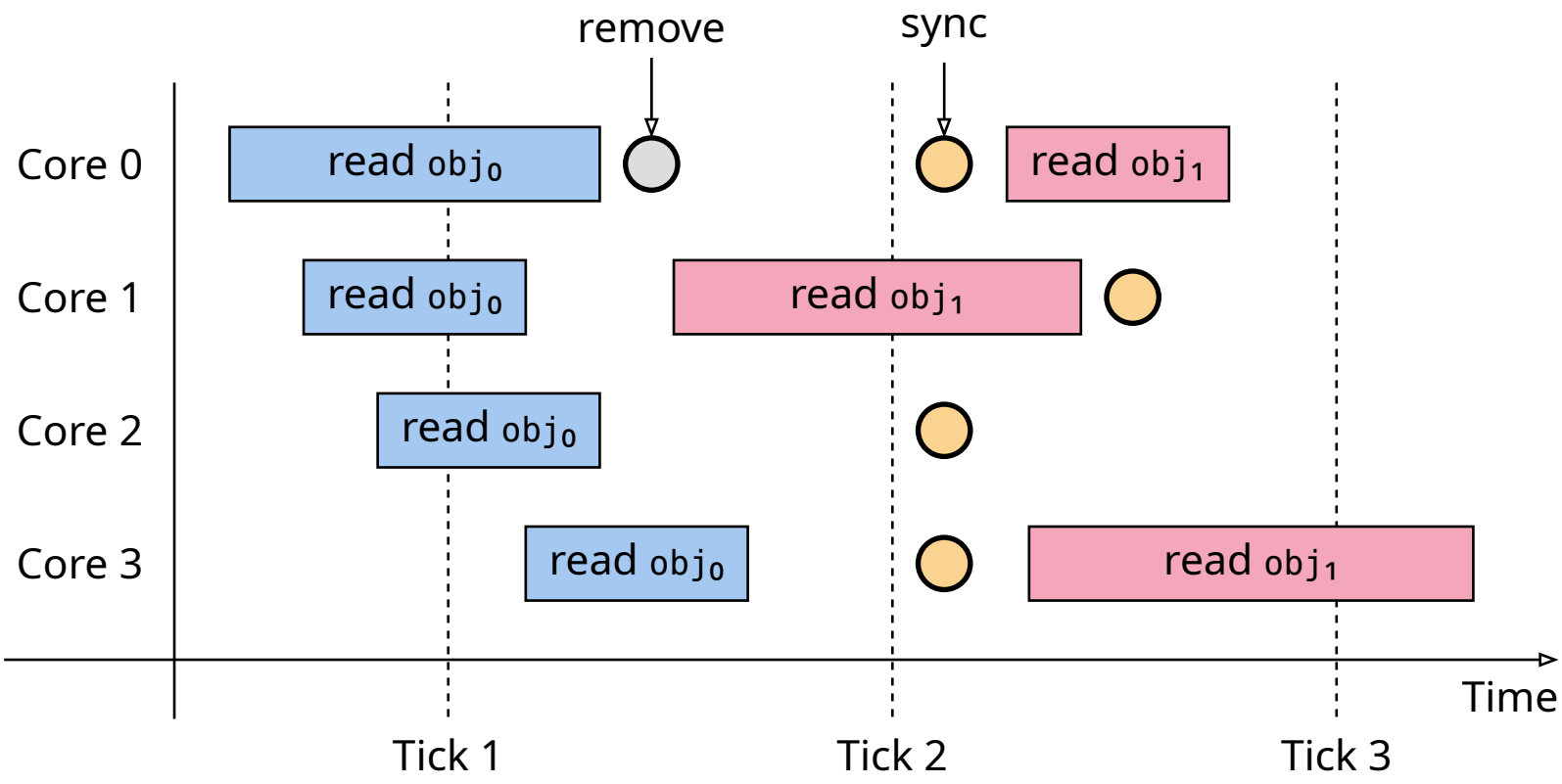
RCU In Action



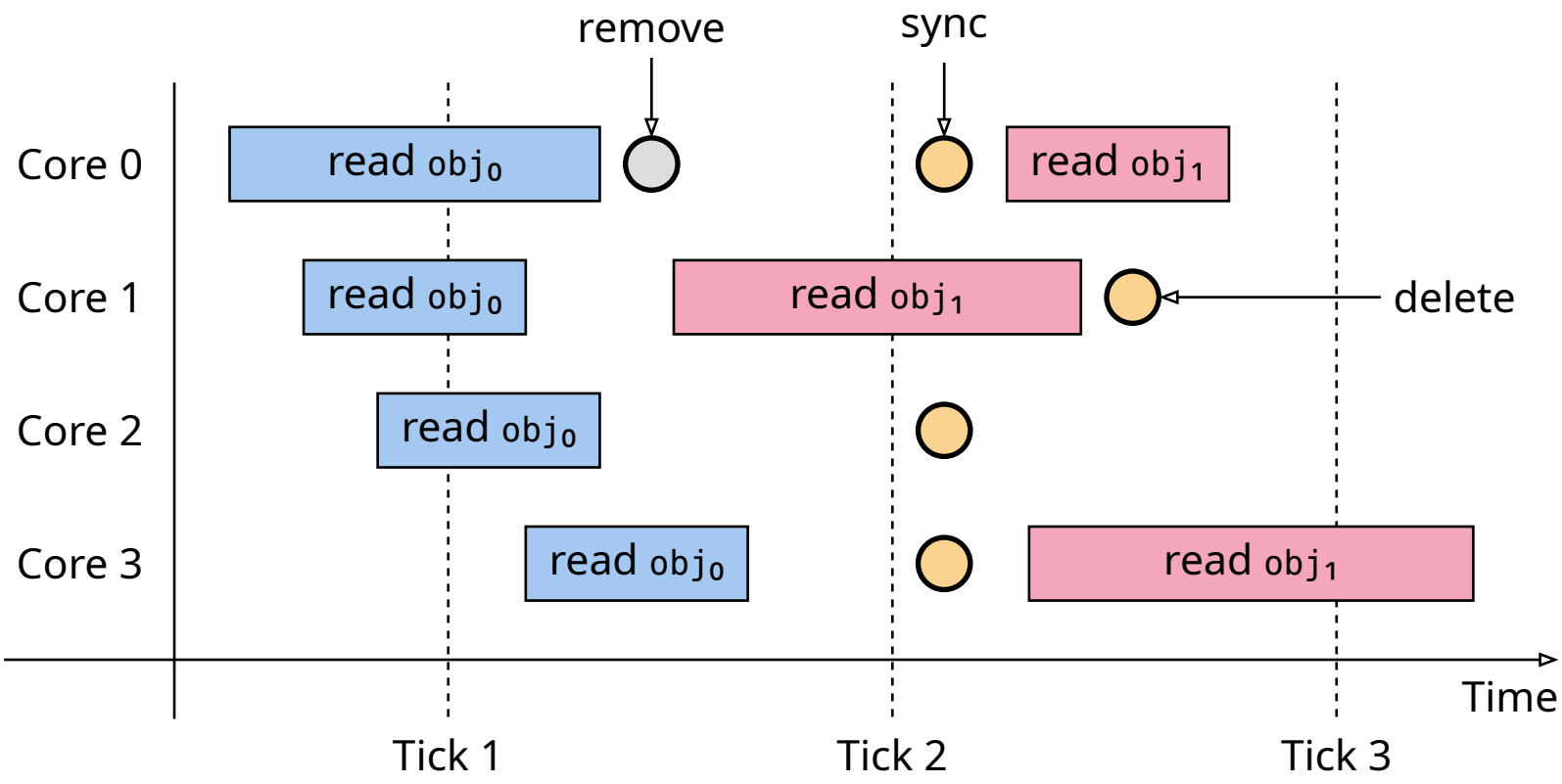
RCU In Action



RCU In Action

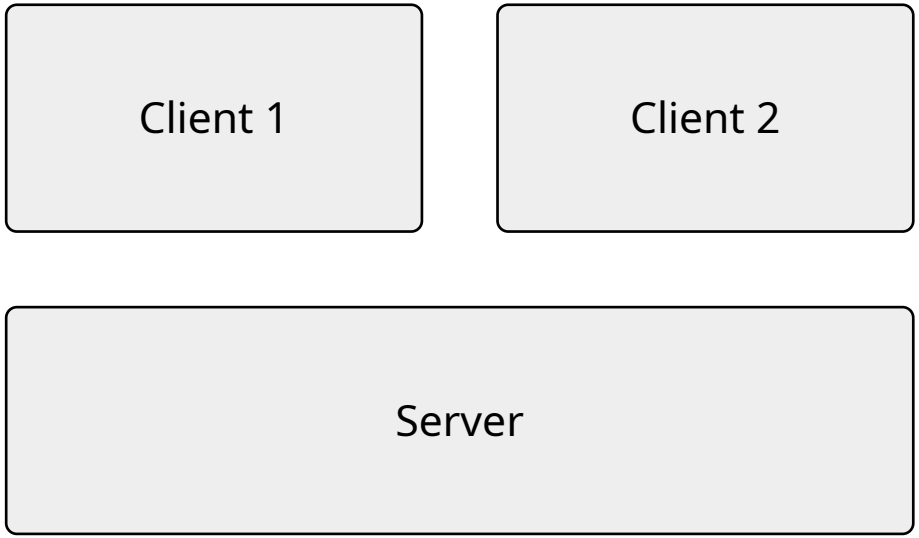


RCU In Action

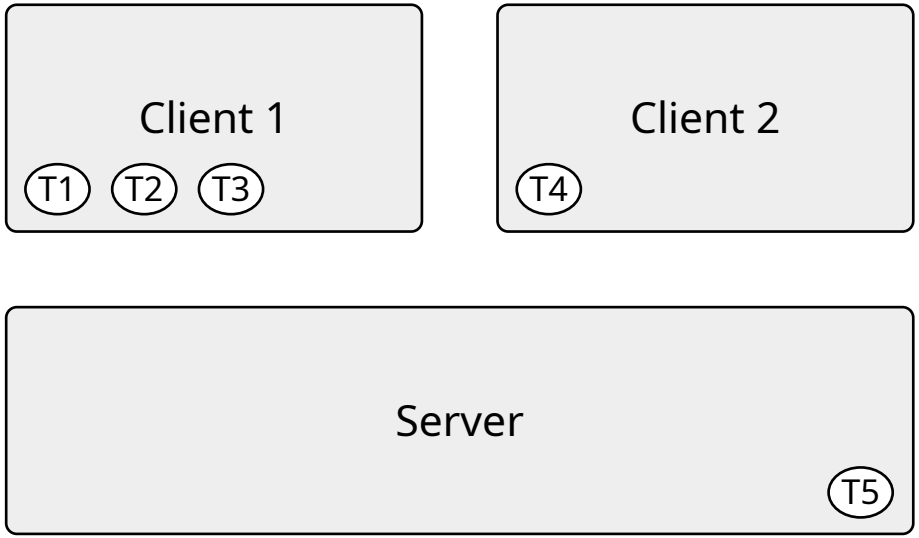


- Threads
- **Synchronization**
  - Basic Concepts
  - Blocking Synchronization
  - Lock-Free Synchronization
  - **Wait-Free Synchronization**
- FPU Handling

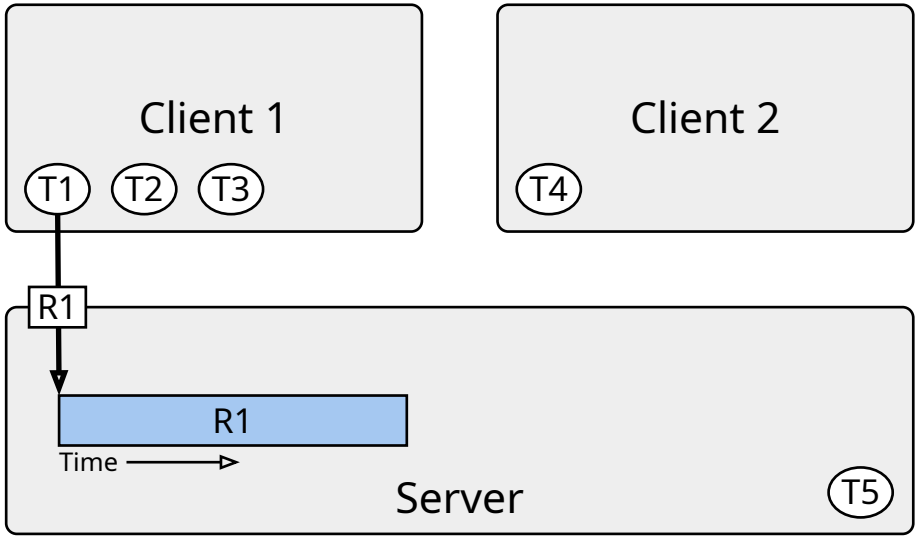
Separation of Scheduling and Execution Context



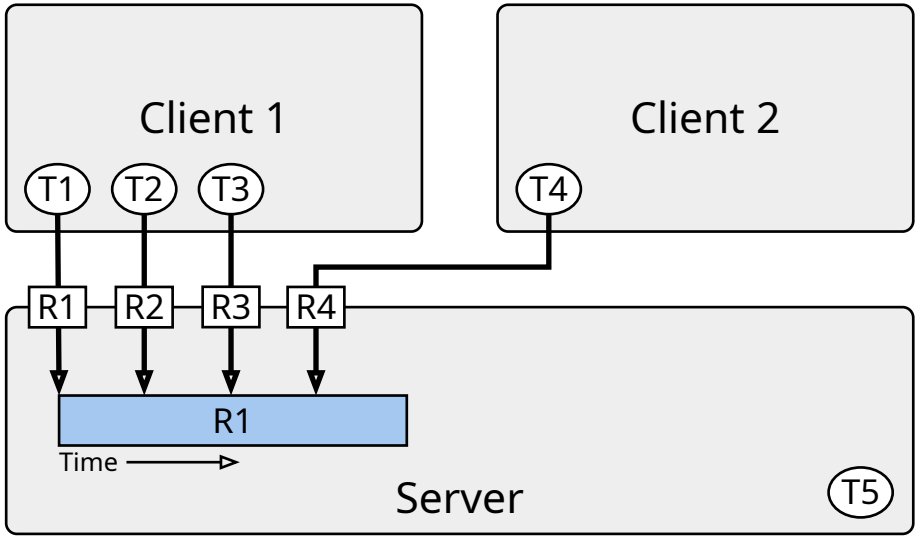
Separation of Scheduling and Execution Context



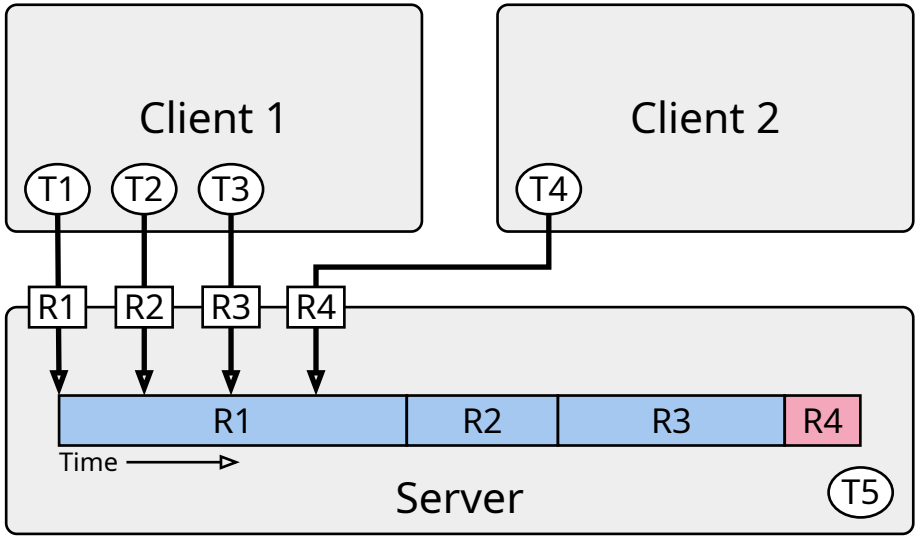
Separation of Scheduling and Execution Context



Separation of Scheduling and Execution Context

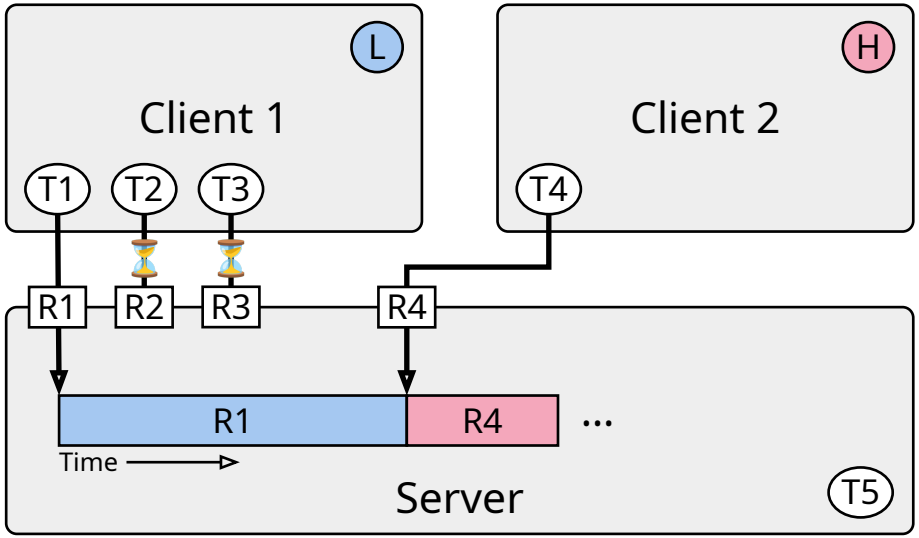


Separation of Scheduling and Execution Context



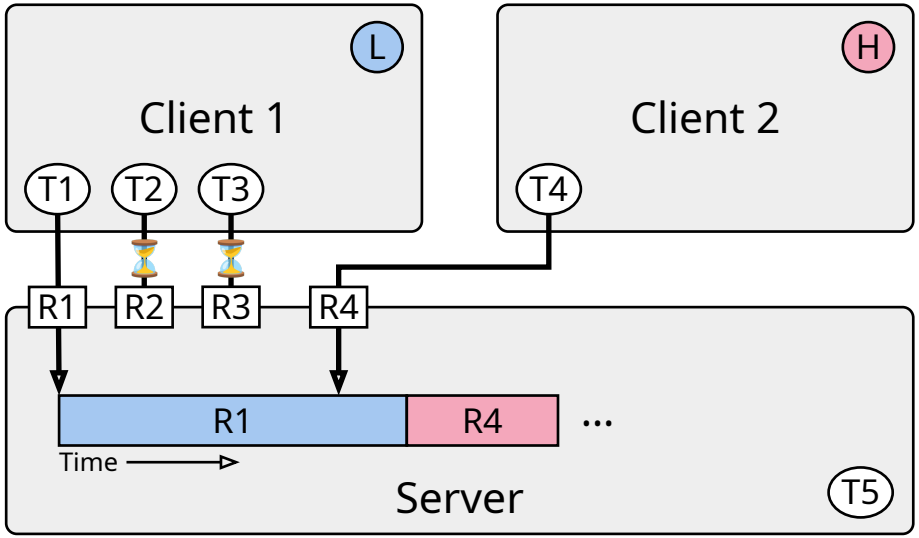
- clients that send the most and longest requests win
- how can we restrict the time budget for requests?
- how can we give clients different priorities?
- what priority should the server thread have?

Separation of Scheduling and Execution Context



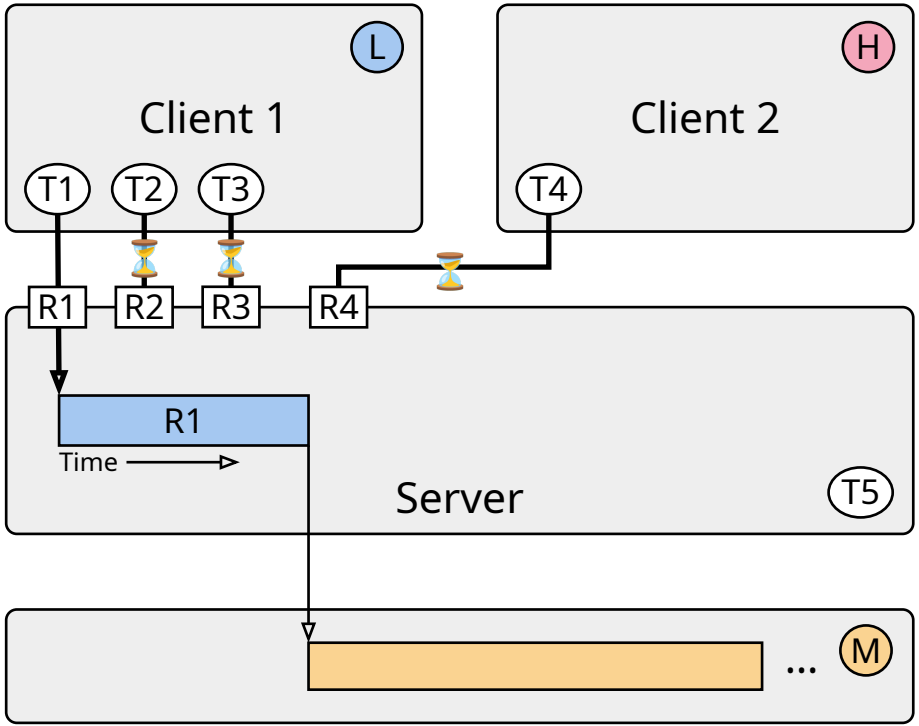
- separate CPU time from execution
- servers do not have CPU time on their own
- requests executed on senders time
- (H) will be served next (higher prio)

Separation of Scheduling and Execution Context



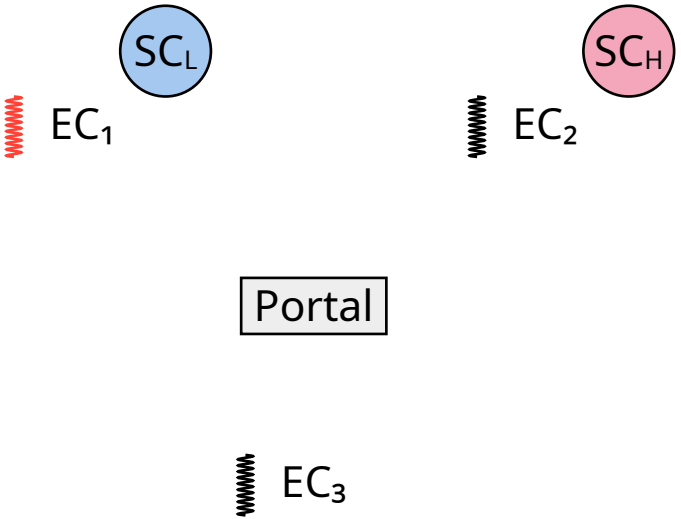
- separate CPU time from execution
- servers do not have CPU time on their own
- requests executed on senders time
- H will be served next (higher prio)
- what if H becomes ready during R1?

Separation of Scheduling and Execution Context



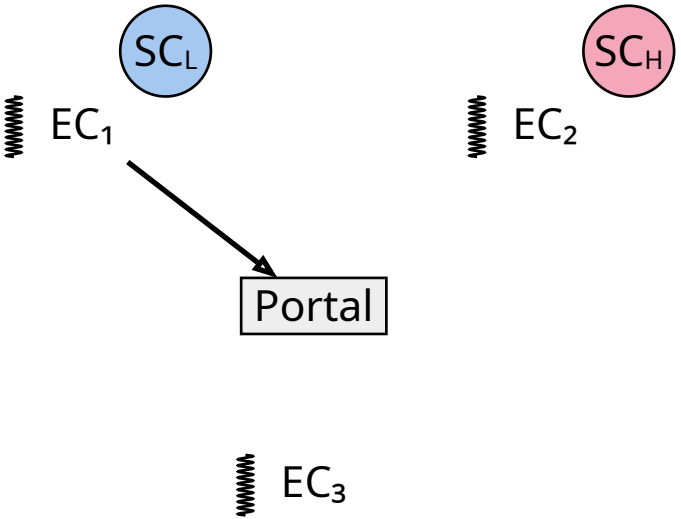
- separate CPU time from execution
- servers do not have CPU time on their own
- requests executed on senders time
- H will be served next (higher prio)
- what if H becomes ready during R1?
- priority inversion?

Solution in NOVA: Priority Inheritance



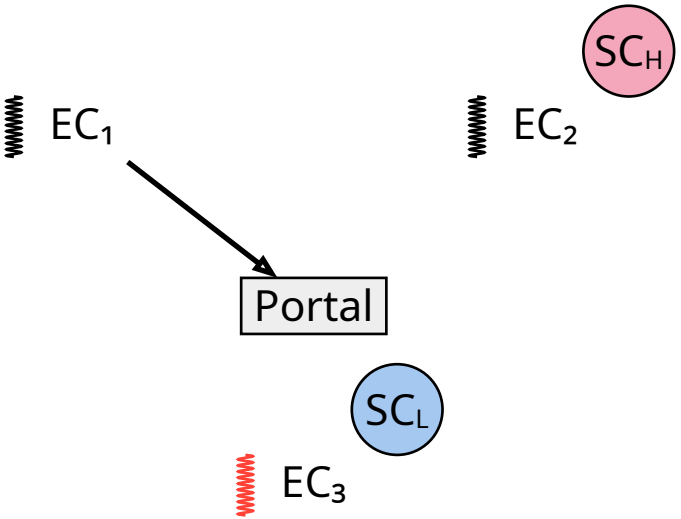
- Timeslice donation:
  - EC_1 calls portal with SC_1
  - SC_L is donated to EC_3

Solution in NOVA: Priority Inheritance



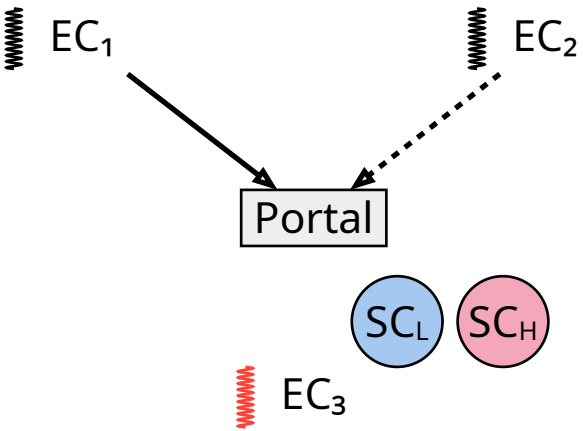
- Timeslice donation:
  - EC_1 calls portal with SC_1
  - SC_L is donated to EC_3

Solution in NOVA: Priority Inheritance



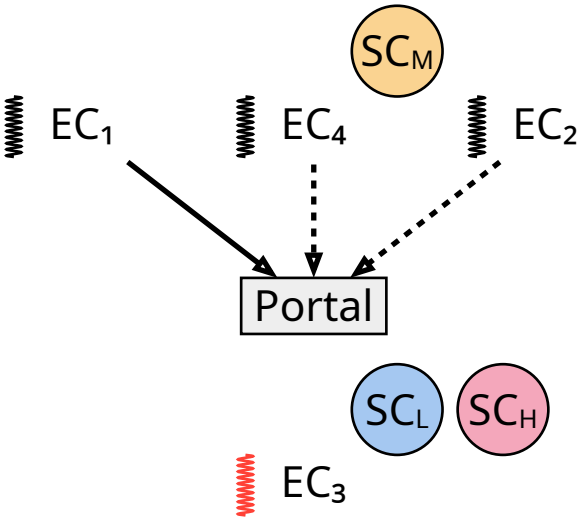
- Timeslice donation:
  - EC_1 calls portal with SC_1
  - SC_L is donated to EC_3

Solution in NOVA: Priority Inheritance



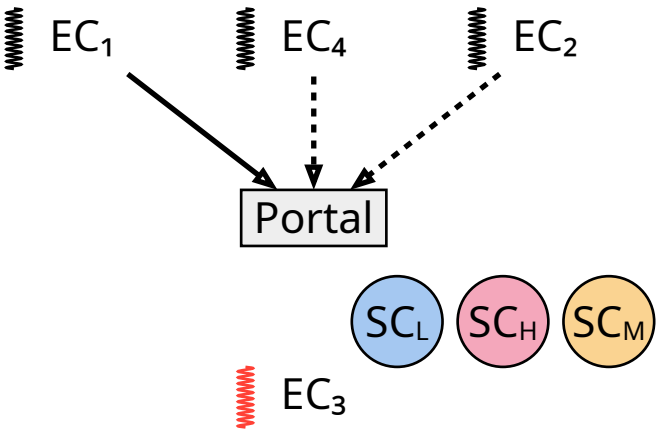
- Timeslice donation:
  - EC_1 calls portal with SC_1
  - SC_L is donated to EC_3
- Helping:
  - if SC_L has no time left, SC_H helps EC_3
  - EC_3 runs with SC_H

Solution in NOVA: Priority Inheritance



- Timeslice donation:
  - EC_1 calls portal with SC_1
  - SC_L is donated to EC_3
- Helping:
  - if SC_L has no time left, SC_H helps EC_3
  - EC_3 runs with SC_H

Solution in NOVA: Priority Inheritance



- Timeslice donation:
  - EC_1 calls portal with SC_1
  - SC_L is donated to EC_3
- Helping:
  - if SC_L has no time left, SC_H helps EC_3
  - EC_3 runs with SC_H

Wait-Free Synchronization

- Strongest non-blocking guarantees
- *All* threads make progress
- Achieved by helping
- Priority inversion prevented by priority inheritance

- Threads
- Synchronization
- **FPU Handling**

Floating Point Unit

- CPU has dedicated functional units for FP computations
- Are accessed with specific instructions
- Have their own state, which is large (512 bytes)
- Each thread has its own FPU state

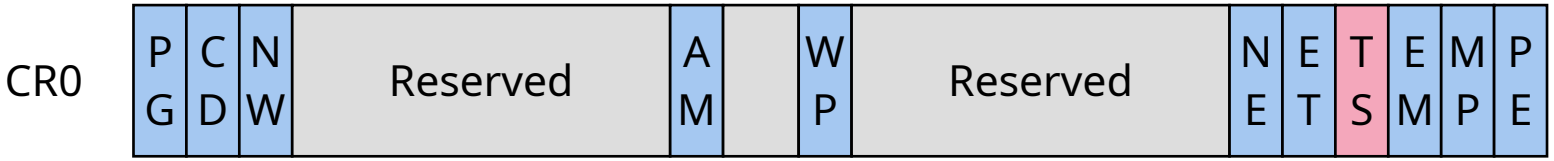
➡️ Save/restore FPU on each context switch is expensive

➡️ However, many OSes on x86 today save it on every switch (vector instructions, LazyFPU vulnerability)

FPU Switch: General Idea

- We want to know if/when a thread uses the FPU
- We only want to save the FPU state if it has been modified
- We don’t want to save the FPU state when switching from a thread that used the FPU to a thread that is not going to use the FPU and then later restore the old (unmodified) FPU state

Lazy FPU Switch on x86



`CR0.TS`

If `CR0.TS` (task switched) flag is set, FPU instructions are not executed, but cause `#NM` exception.

Implementation in NOVA

Handling the #NM exception

C++

```
1   void handle_exc_nm() {
2     CR0.TS = 0;
3     hzd |= HZD_FPU;
4     if (current == fpowner) return;
5     if (fpowner)
6       fpowner->fpu->save();
7     if (current->fpu)
8       current->fpu->load();
9     else {
10      current->fpu = new Fpu;
11      Fpu::init();
12    }
13    fpowner = current;
14  }
```

Before leaving to user

C++

```
1  void handle_hazards() {
2    if ((hzd & HZD_FPU) &&
3          current != fpowner) {
4      CR0.TS = 1;
5      hzd &= ~HZD_FPU;
6    }
7  }
```