UiT

THE ARCTIC UNIVERSITY OF NORWAY

Project 3

Preemptive Scheduling

INF-2201 Operating System Fundamentals
Spring 2023

Department of Computer Science
University of Tromsø

Overview

- You will implement an OS that schedules threads and processes *preemptively*.
- Implement synchronization primitives that work with preemption
  - Re-implement locks
  - Condition variables
  - Semaphores
  - Reusable barriers
- Concurrent programming
  - Implement a fair solution to the dining philosophers problem
- Extra credits
  - Priority scheduling
  - Pthreads implementation of dining philosophers

Previous operating system

- **Non-preemptive** multiprogramming kernel
- System calls by **calling a function in the kernel**
- Processor executes in protected mode
  - All processes run in ring 0
- Synchronization (**locks**) only for kernel threads
  - For non-preemptive kernel

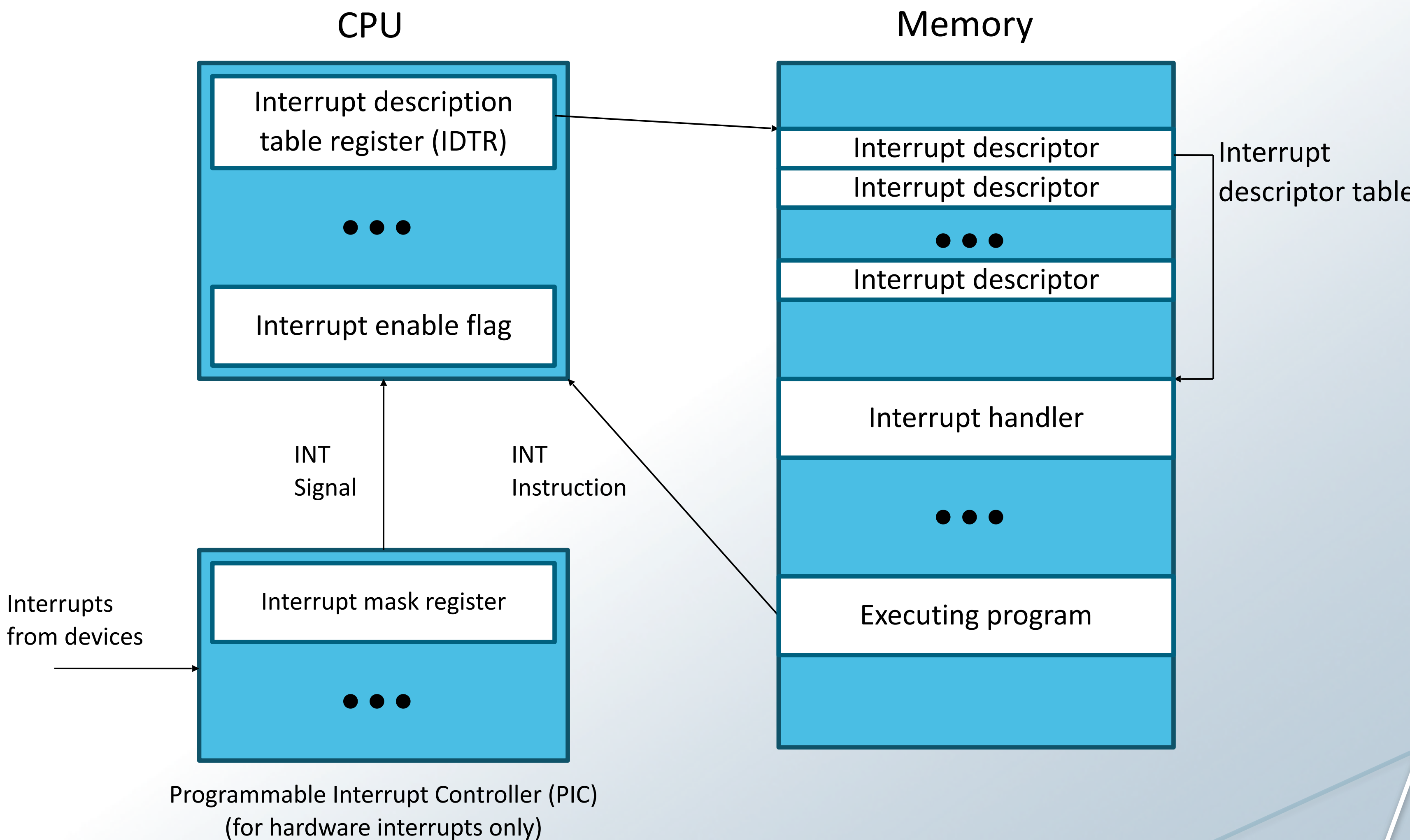
New operating system

- **Preemptive** multiprogramming kernel
- System calls via **software interrupts**
- Processor executes in protected mode
  – Processes will still run in ring 0
- Synchronization primitives that **work with preemption** (but still only for kernel threads)
  – Locks
  – Semaphores
  – Condition variables
  – Barriers

Three types of interrupts

- Hardware interrupts (external interrupts)
  - Example: system timer interrupt
- Software interrupts (INT instructions)
  - Example: (real-mode) BIOS calls
- Software exceptions

Hardware support for interrupts



What does the CPU do when an interrupt occurs from ring 0?

- Get interrupt descriptor address
- Make privilege checks
- Push `EFLAGS`, `CS`, and `EIP` on stack
- Clear `EFLAGS[IF]` (and `EFLAGS[TF]`)
- Load `CS:EIP` from interrupt descriptor
  - Jumps to interrupt handler

And then?

- The interrupt handler executes.
- If interrupt came from PIC (if it was a hardware interrupt), the handler must send an end-of-interrupt (`EOI`) signal to the PIC.
  - Because PIC sets a bit in its In-Service Register (`ISR`), corresponding to the hardware interrupt vector, to block consecutive interrupts on that vector until handler completes.
  - `EOI` clears interrupt vector bit in `ISR`.
  - In other words, if you forget to clear `EOI`, you will only get one interrupt from each hardware vector …ever!
- To return from interrupt handler, execute `IRET` instruction
  - Pops `CS`, `EIP` and `EFLAGS`.
  - When restoring `CS:EIP`

Pseudo-C template of an interrupt handler

```
void irqX(void)
{
  /* save context of interrupted thread/process */
  …
  /* do the work */
  …
  /* restore context of interrupted thread/process */
  …
  /* restore EIP, CS and EFLAGS */
  asm volatile("iret");
}
```

Remember: if this is a H/W interrupt handler, you also need to issue EOI:

```
outb(0x20, 0x20);
```

Preemptive Scheduling

- Most of it is already setup for you
- Setup system timer interrupt so that an interrupt will be generated on IRQ line 0 every 10 milliseconds (given in `kernel.c`).
  - You may want to change this value for debugging and testing.
- You need to implement the code for the timer interrupt handler in `entry.S`.
  - Switch between user and kernel stack
  - Save and restore the context
- Modify `yield()`, `lock_acquire()`, `lock_release()`, etc. to deal with preemptive scheduling.
  - Critical regions (`enter_critical()`, `leave_critical()` in `entry.S`

Synchronization primitives

- Semaphores, barriers, and condition variables with semantics as discussed in lectures
  - Also check how this is done in Pthreads
  - The template for code you need to write is in `thread.c`.
- You will also need to re-implement locks (like in P2), but this time make sure that they can be used in a preemptive context (avoid race conditions).
- You have to design data structures for semaphores, barriers and condition variables.
  - Empty structures are defined in `thread.h`.

Dining philosophers

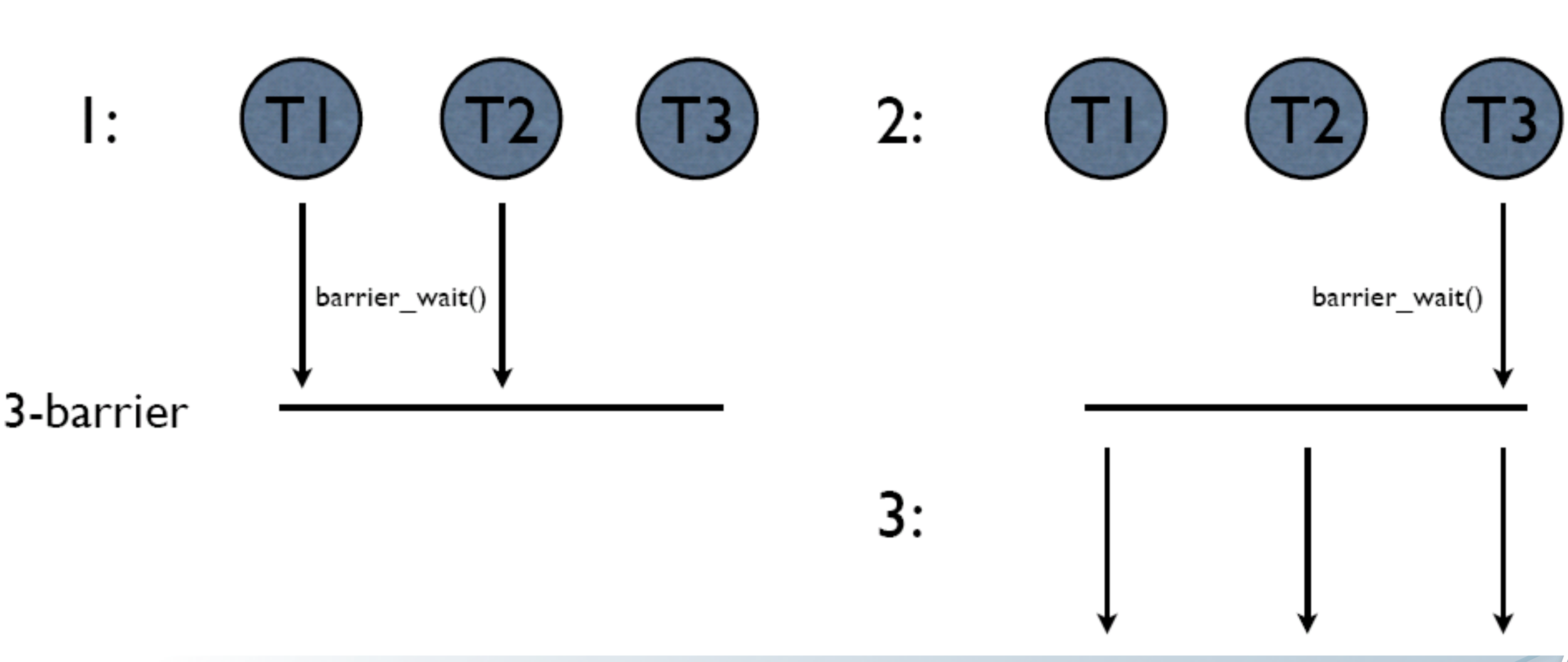
- Precode comes with a solution using semaphores
  - Three philsophers: Caps, Num and Scroll
    - Watch the keyboard LEDs.
  - But this solution is not fair.
  - Favors Caps (why?)
- Come up with and implement a solution that is “fair”
  - Every philosopher can eat for the same amount of time
- Document your solution

Barriers (1)

- A barrier is a synchronization mechanism where several threads can be "realigned", that is, once they leave the barrier, they will all be in the same stage of execution.
- An n-barrier works by blocking threads calling `barrier_wait()`, then unblocks <u>all</u> the threads when the *n*th thread calls `barrier_wait()`.

Barriers (2)

```
0: barrier_init(3)
```



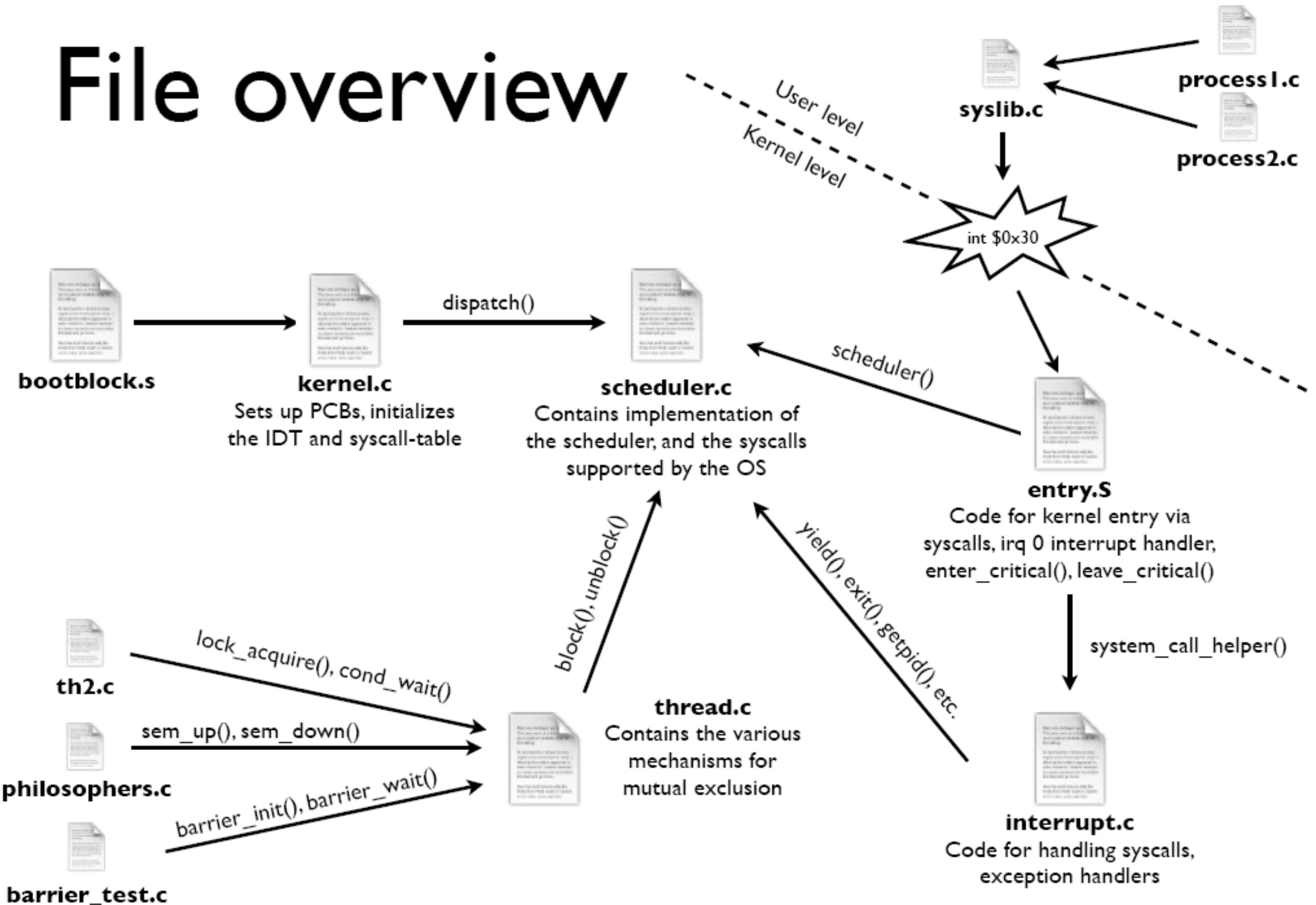
Barriers (3)

- Beware of race conditions!
  - What happens when one thread manages to call `barrier_wait()` a second time before all the other threads have been unblocked

Extra Credits

- Priority scheduling
  - Process 1 calls `set_priority()`
  - Modify `scheduler()`
- Dining philosophers implemented using Pthreads
  - You will do this in Linux!
  - You need to create the source files and Makefile (there is no pre-code)
  - Refer to online documentation for Pthreads details

File overview



Where to start?

- Do the preemptive-multitasking bit first!
- Once that is done…
  - Complete the locks
  - Semaphores
  - Condition variables
  - Barriers
- Remember, all synchronization primitives must work with preemption present!
- Finally, work on the dining philosophers problem

- You really should learn how to use these mechanism in Pthreads

Design review, code, report, hand-in

- Same as for P1
- …using GitHub