Expectations of 4411

- Grades usually look not bad if you complete all projects yourselves.
- Understand one important OS concept deeply every week.
  - context-switch, interprocess communication, exception control flow
  - today: priority in scheduling
- Gain some experience on how to build and debug larger project.
- Learn a bit about history: IBM360, UNIX, Turing Awards in systems area, …

Review

- First, operating systems solve time-sharing multi-tasking.
  - context = memory address space + stack pointer + instruction pointer
- Second, operating systems solve interprocess communication (IPC).
  - AT&T UNIX V provides message queue, shared memory and semaphore
- Third, operating systems handle exception control flow: exceptions can be generated by software or hardware and OS handles them.
- Now, we are ready to paint a full picture of scheduling (today's lecture)

A bit more on Exception Control

Why different on MacOS and Linux?

```
void thread_create(){
…
stack_start = malloc(stack_size);
// the following line crashes ctx_start on MacOS
// but it works on Linux
stack_ptr = (address_t)(stack_start + stack_size - 1);
ctx_start(&old_sp, stack_ptr);
…
}
```

Why different on MacOS and Linux?

```
void thread_create(){
…
stack_start = malloc(stack_size);
// the following line crashes ctx_start on MacOS
// but it works on Linux
// stack_ptr = (address_t)(stack_start + stack_size - 1);
// here is the fix
stack_ptr = (address_t)(stack_start + stack_size - 16);
ctx_start(&old_sp, stack_ptr);
…
}
```

Stack pointer needs to be 16-byte aligned!
This is described in the CPU manual.

Why different on MacOS and Linux?

- If stack pointer is not 16-byte aligned, CPU will raise an exception to both MacOS and Linux.
  - For example, in RISC-V, this causes a “misaligned address exception”
  - CPU has many alignment constraints: instruction pointer, data address for atomic operations, etc. This is usually a concern in compilers but sometimes in OS as well.

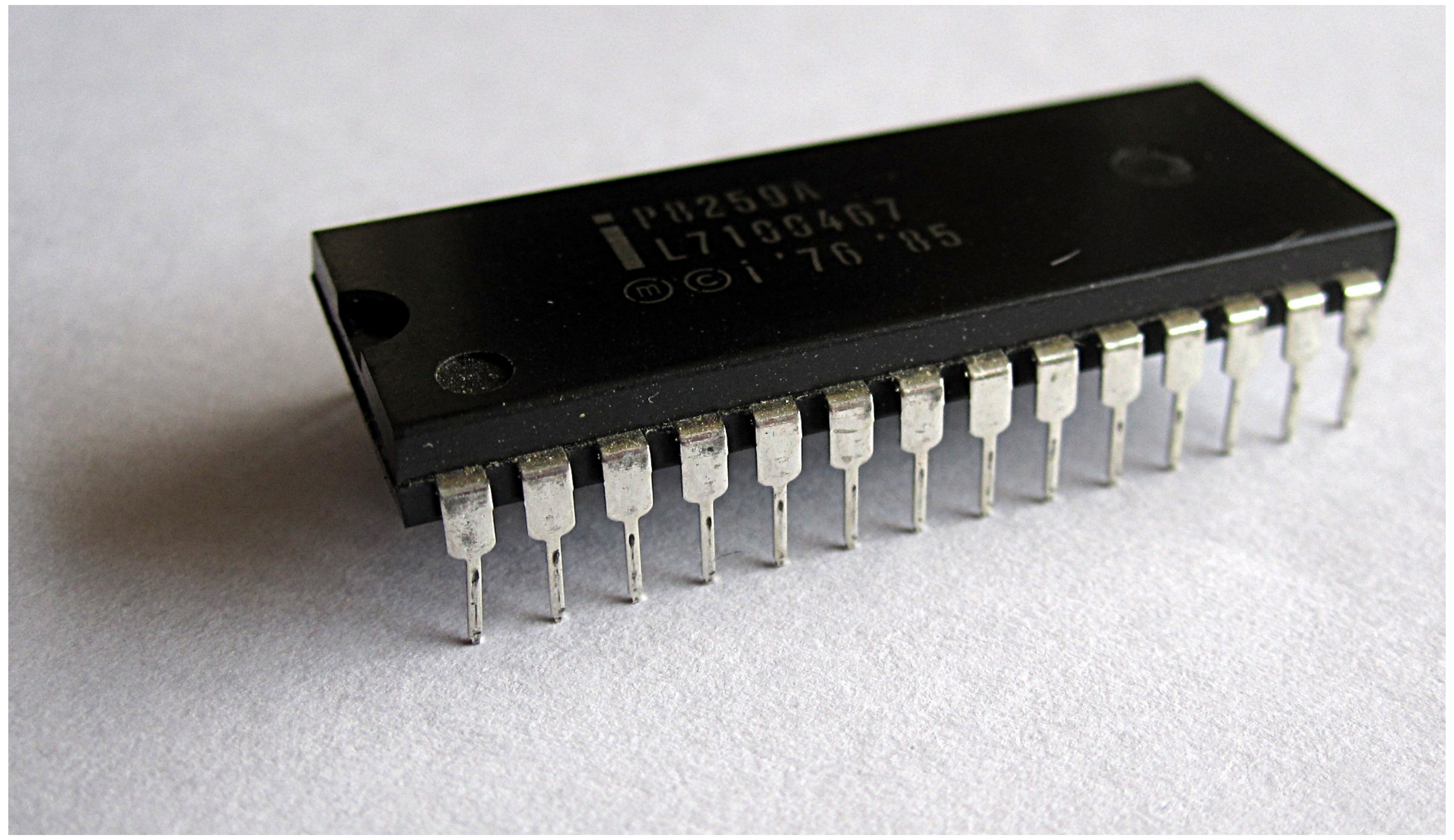
One possibility

- MacOS and Linux can handle the misaligned exception differently:
  - MacOS decides to forward this exception to user application. If user application doesn’t handle this exception, it crashes.
  - Linux decides to mask this exception. Linux aligns the stack pointer for the user application and return to the user application normally.
- In the real-world, different OS can have different behaviors. This is common in real-world software engineering.

Exceptions have priorities.

Priority in exception handling

- OS can set the priority of exceptions by writing to registers of programmable interrupt controller hardware.
  - When multiple interrupts come at the same time, higher priority ones are delivered first by this piece of hardware.



Intel 8259 interrupt controller

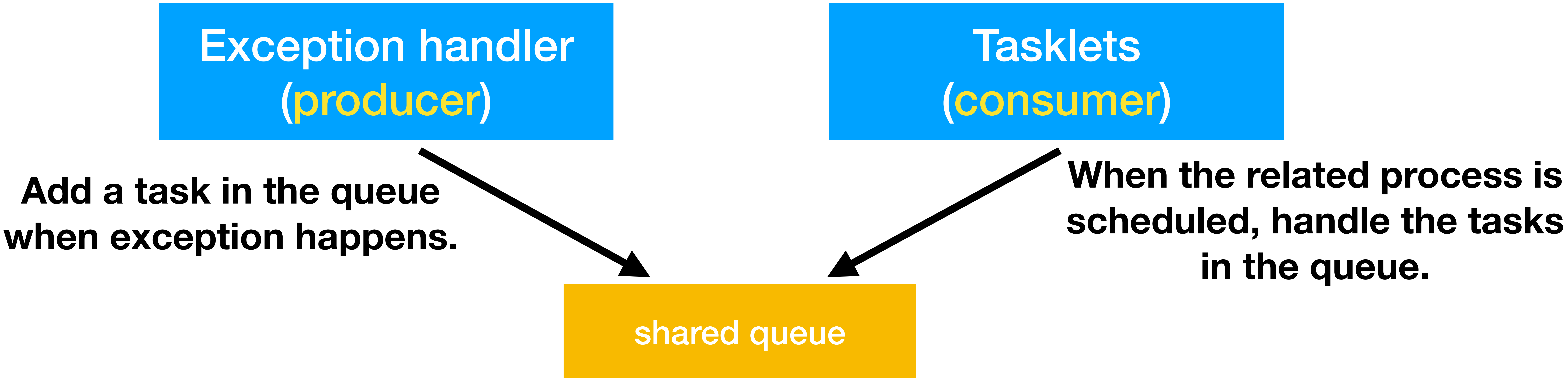
https://en.wikipedia.org/wiki/Intel_8259

Exception handling priority in Linux

Priority	Exceptions
0	Handling tasklets (next slide)
1	Timer interrupt
2	Network card send interrupt
3	Network card receive interrupt
4	Block device (e.g., disk) interrupt
...	...

Why Tasklets in Linux?

- Goal: exception handler in OS should be as fast as possible.
  - For example, if an exception handler for the disk runs very long time, it will block the pending exceptions from the network card for a long time.
  - Tasklets use the spirit of producer-consumer just like P1!



Exception handler (producer)

Tasklets (consumer)

Add a task in the queue when exception happens.

When the related process is scheduled, handle the tasks in the queue.

shared queue

Exceptions have priorities.

Process scheduling also has priorities.

Priority in process scheduling

- We have seen single-queue scheduling in 4411 P1.
- Multi-level feedback queue (MLFQ) is a scheduling algorithm that assign different priorities to different processes.
- Demo of `htop`

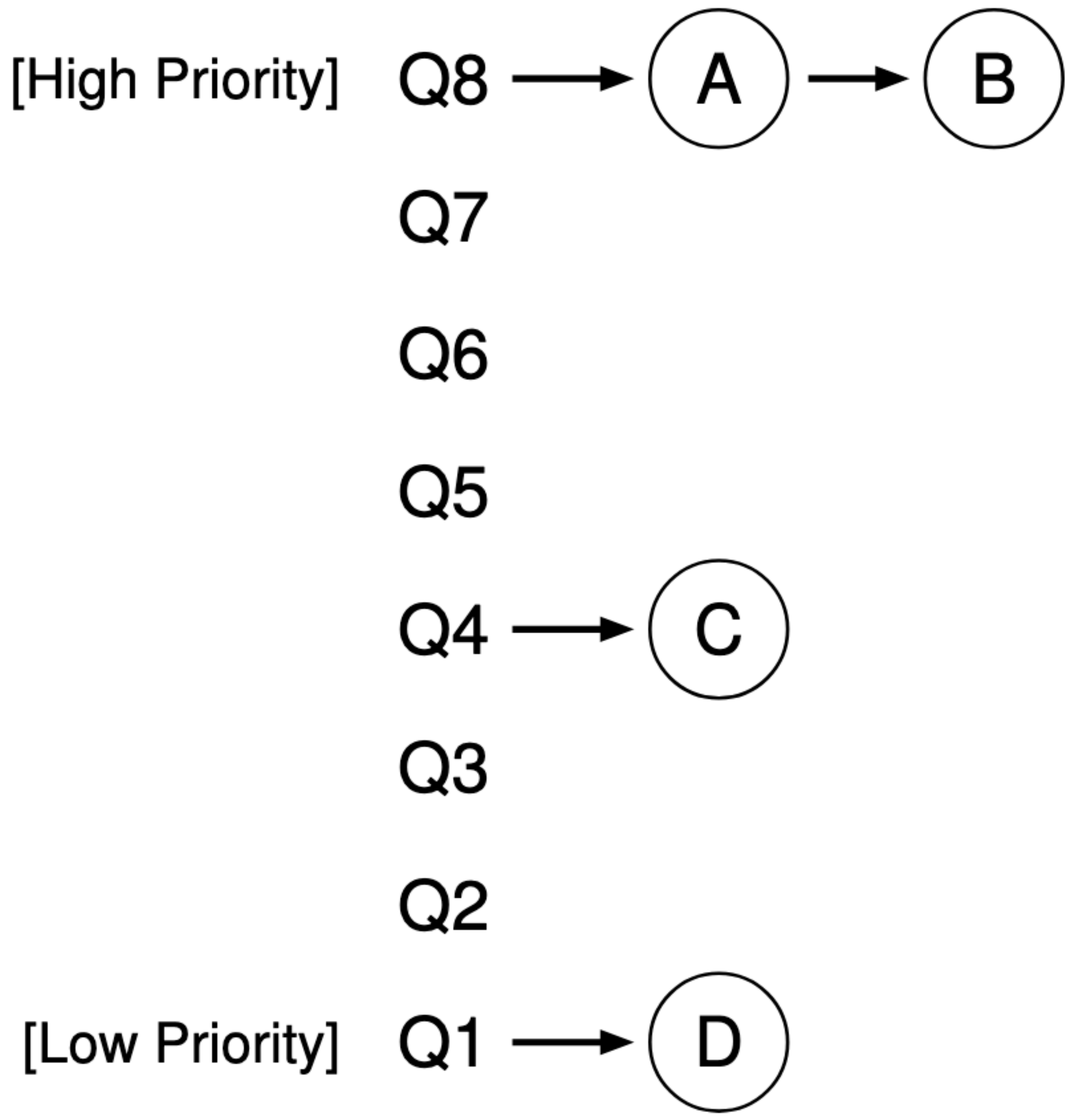


Figure 8.1: **MLFQ Example**

*** OSTEP chapter8 is a good handout for understanding MLFQ**

I/O-bound vs. CPU-bound

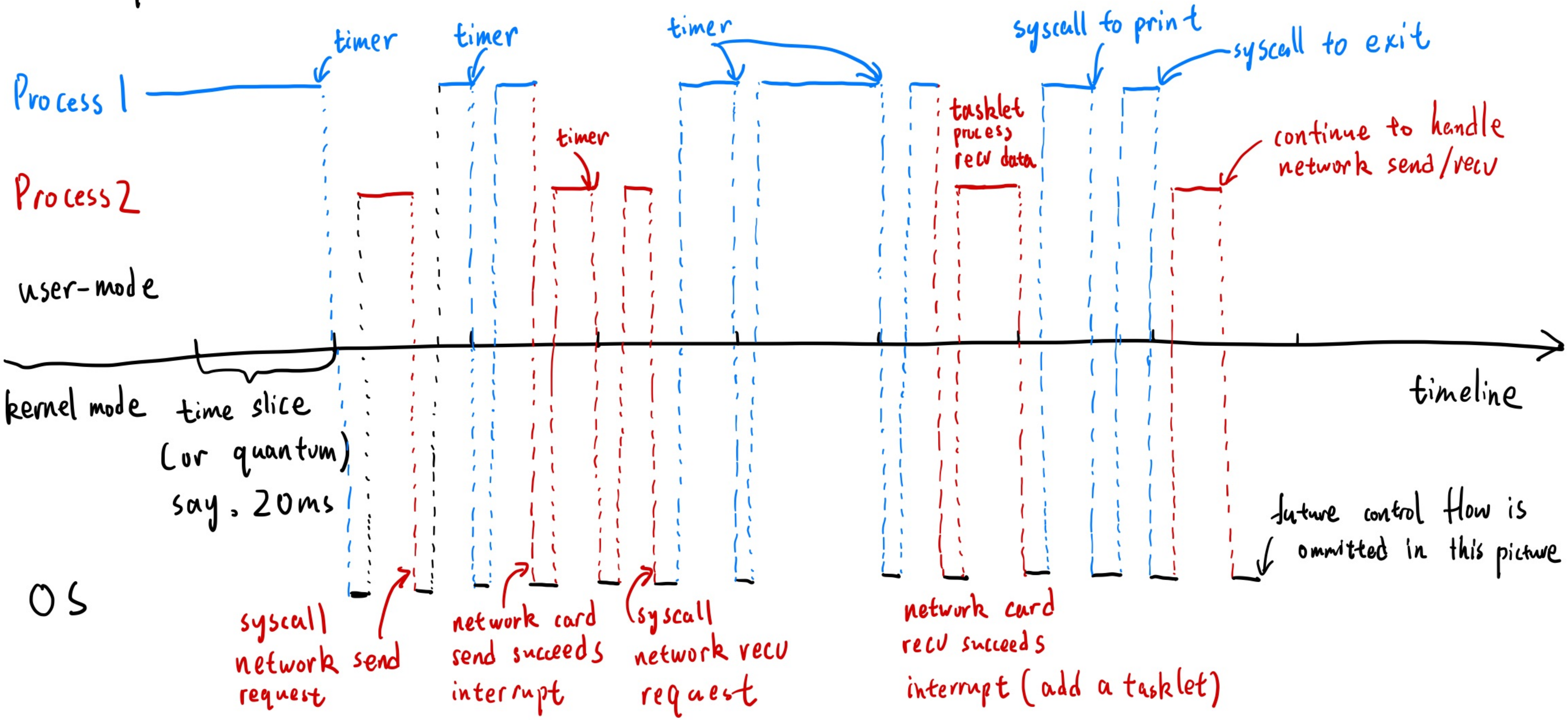
- Key lesson for MLFQ: user applications can be I/O-bound or CPU-bound.
  - For example,
    - `ls` is an I/O-bound application, it reads a directory in the file system
    - loop is a CPU-bound application, it runs a loop
  - Your goal in P1 is to maintain `ls` a higher priority and `loop` a lower priority by detecting the I/O behaviors of the processes (i.e., OS does not know that they are `ls` and `loop`)
  - Demo in EGOS

Full Picture of Scheduling

Full picture of scheduling

process 1 : CPU-bound, print result at the end (e.g. matrix computation)

process 2 : I/O-bound, frequent I/O to network (e.g. zoom)



Take-aways

- OS manages priorities for interrupts, processes scheduling, etc.
- User applications can be categorized into CPU-bound and I/O-bound.
  - Scheduler can give different priorities to processes by detecting whether a process is CPU-bound or I/O-bound.
- A full picture of scheduling is painted.

Homework

- P2 is due on Oct 23. Implement the MLFQ scheduling algorithm (read OSTEP chapter 8).
- Read `src/apps/ls.c` and `src/apps/loop.c` in EGOS and understand I/O-bound vs. CPU-bound.
- No lecture on Oct. 14; next lecture on Oct. 21 introducing memory hierarchy