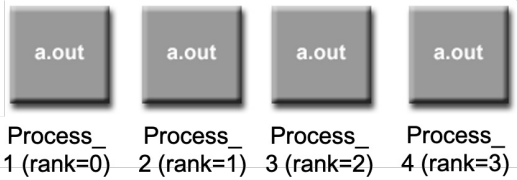
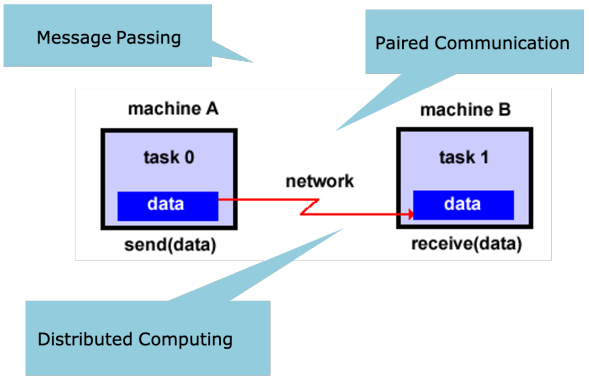
Lecture 11: Introduction to Process Scheduling

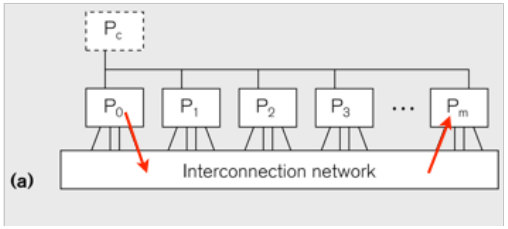
Vivek Kumar
Computer Science and Engineering
IIIT Delhi
vivekk@iiitd.ac.in

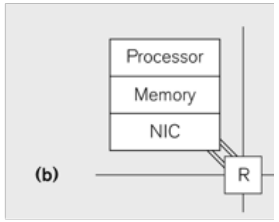
Last Lecture





==> Processors communicate by sending messages via an interconnect






```
int main(int argc, char **argv) {
    int rank=0, nproc=4;
    MPI_Init(&argc, &argv);
    // 1. Get to know your world
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &nproc);
    int array[SIZE]; // initialized and assume (SIZE % nproc = 0)
    // 2. calculate local sum
    int my_sum = 0, chunk = SIZE/nproc;
    for (int i=rank*chunk; i<(chunk+1)*rank; i++) my_sum += array[i];
    // 3. All non-root processes send result to root processes (rank=0)
    if(rank > 0) {
        MPI_Send(&my_sum, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
    }
    else { // executed only at rank=0
        int total_sum = my_sum, tmp;
        for(int src=1; src<nproc; src++) {
          MPI_Recv(&tmp, 1, MPI_INT, src, 0, MPI_COMM_WORLD, NULL);
          total_sum += tmp;
        }
    }
    MPI_Finalize();
}
```

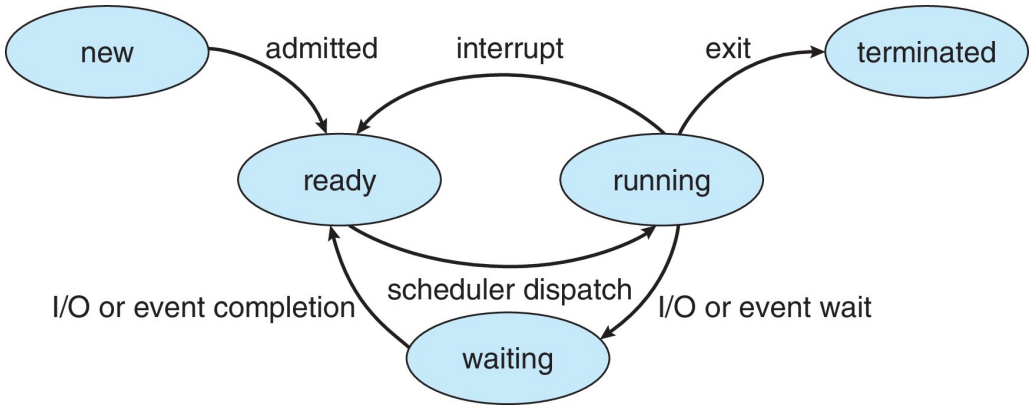
- Several machines (e.g., workstations) connected together in a cluster with some interconnect
- One big computation can be divided equally into N number of sub-computations that can run on N different processes
- Processes use MPI to communicate in distributed memory by exchanging messages with each other
- Amdhal's law: if 50% of your application is parallel and 50% is serial, you can't get more than a factor of 2 speedup, no matter how many processors it runs on

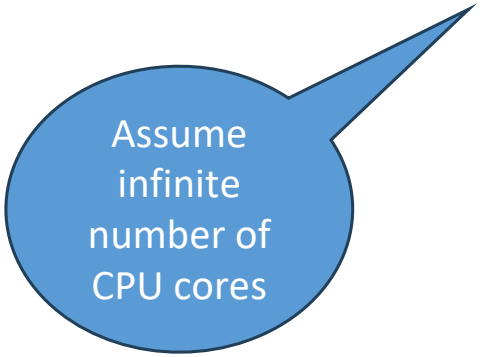
Today's Class

- Introduction to process scheduling
  - Context switch
  - Process scheduling policies

In this course, we will not have any assignment involving carrying out changes inside any operating systems. Still, to understand some concepts, we would revisit well-known OS, such as xv6 OS – as in today's lecture!

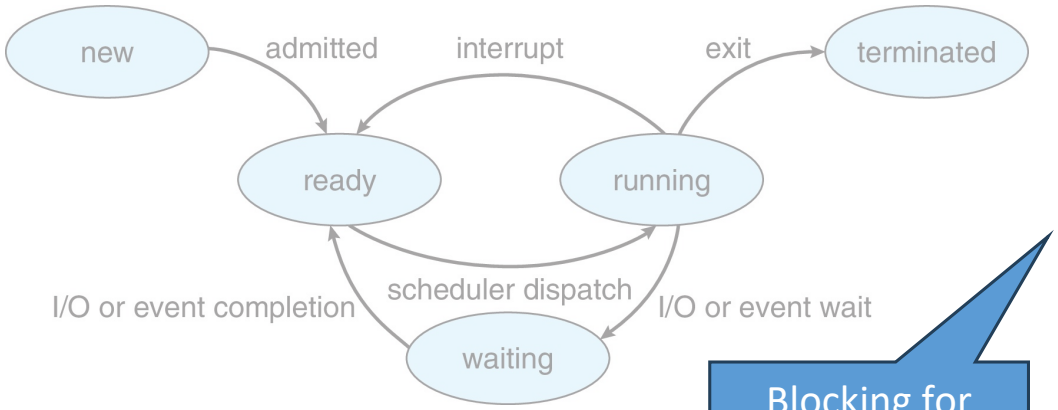
Process State in these Scenarios?





1. Write on pipe that is currently full
2. scanf() waiting for user input
3. Recursive Fibonacci number calculation
4. Process executing sem_wait when
   - Semaphore > 0
   - Semaphore <= 0
5. Process executing sem_post
6. sleep()
7. wait()

Process State in these Scenarios?



Blocking for some resources

Assume infinite number of CPU cores

Running until termination

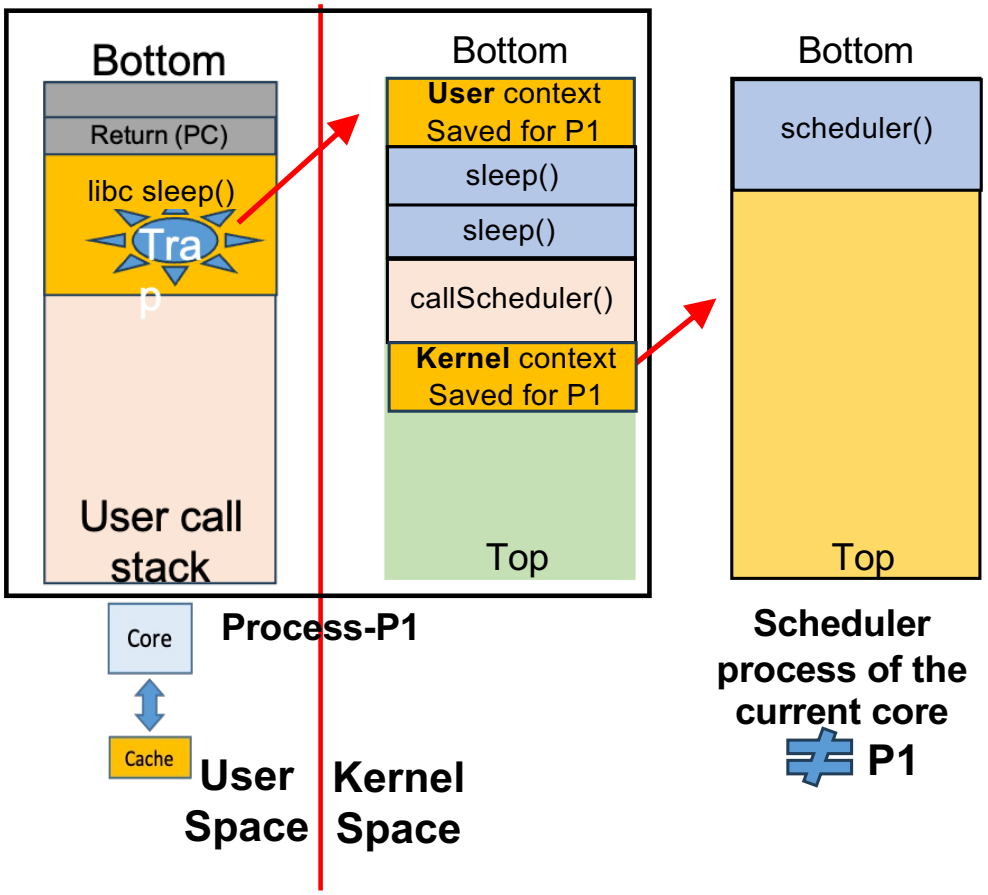
1. Write on pipe that is currently full
2. scanf() waiting for user input
3. sleep()
4. wait()
5. Process executing sem_wait when
   - Semaphore <= 0

6. Recursive Fibonacci number calculation
7. Process executing sem_wait when
   - Semaphore > 0
8. Process executing sem_post

How OS Can Juggle Between Processes?

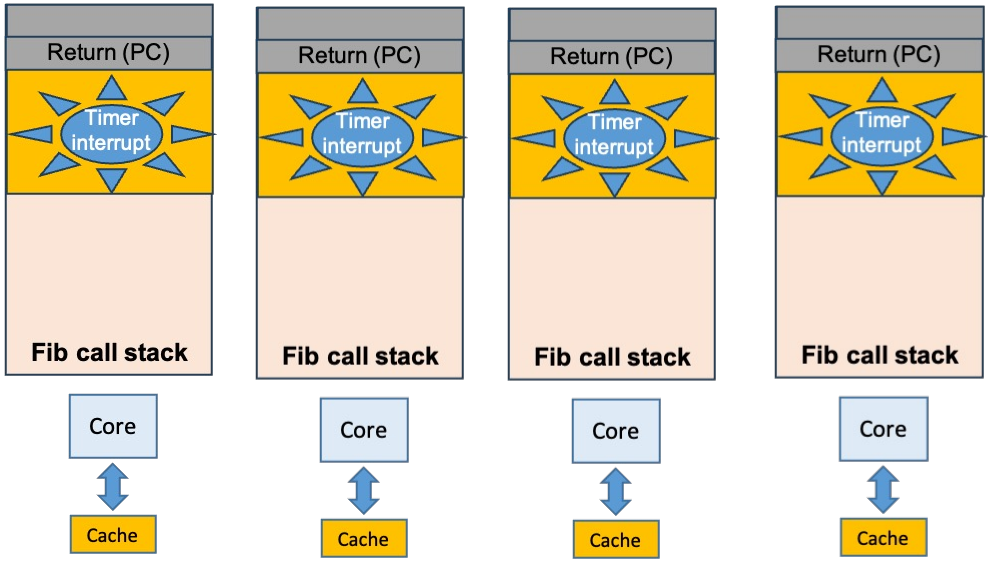
- Case-1: Whenever a process is blocking for some resource
  - It does not make any sense to let a blocked process running on the CPU (e.g., waiting for IO, sem_wait, reading from an empty pipe, etc.)
    - Even if the CPUs are idle
  - OS can safely move the process from the running to waiting queue
    - Does not affect the performance as if CPUs are idle, the process can be brought back into the running queue as soon as it unblocks
- Case-2: If a process is not blocking for some resource (running until termination)
  - Timer interrupt is generated by the CPU
    - It is a hardware generated interrupt at a fixed interval
  - It is used to trigger the scheduling by moving the currently running process into the ready queue, and another process from the ready queue into the running queue

Case-1: Process is Going to Block



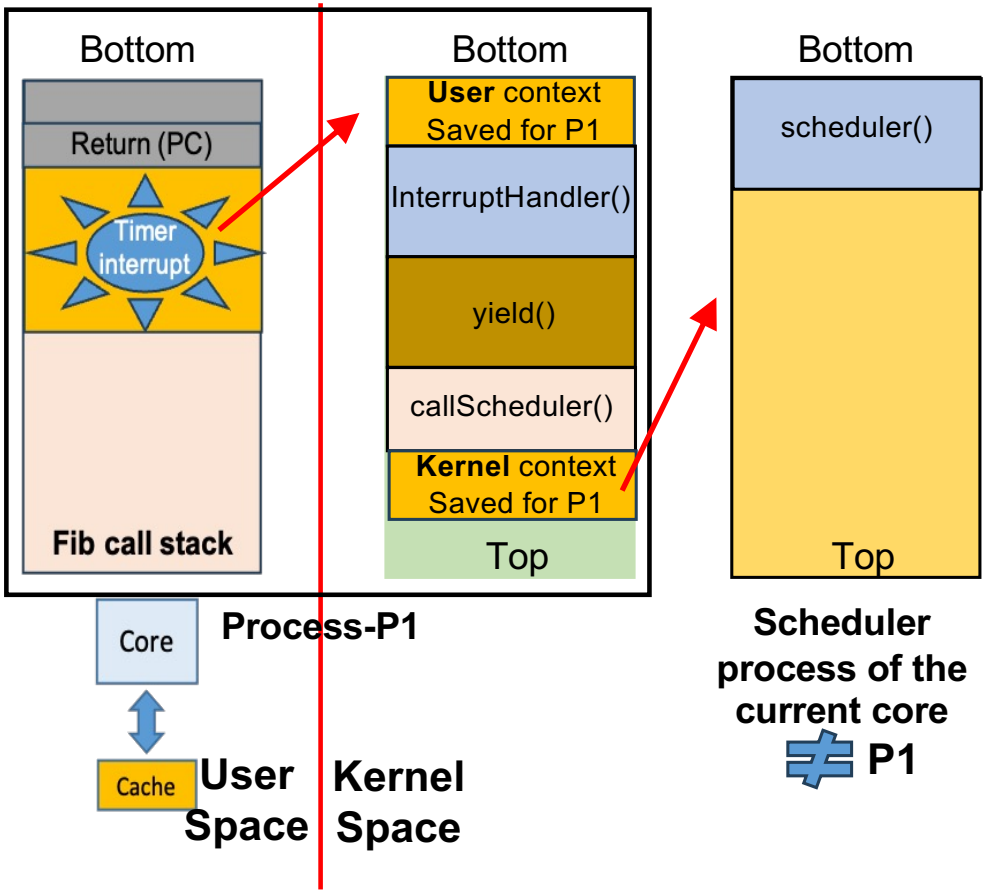
- The user process P1 calls a blocking API (e.g., **sleep** on an empty pipe)
- P1 switches its execution from the user stack to its kernel stack (each user process has its own kernel stack)
  - User context is saved on the bottom of the kernel stack and interrupt handler is invoked
  - The blocking system call will eventually call some sleep implementation
- The sleep will invoke a call to the CPU scheduler, which will: a) save the kernel context of the user process at the top of the kernel stack, b) load the context of the CPU scheduler from the corresponding stack, and c) jump to the CPU scheduler stack where it will call the schedule() routine
- Each CPU has its own CPU scheduler process that runs on a separate kernel stack than that of the process it was originally executing
  - Why?

Case-2: Process is Running until Termination



- There could be several Fibonacci processes being displayed in **running** state from htop on a 4-core processor
  - However, only four of them are **actually** in running state any given time (1-1 mapping between a process and a core)
- Timer interrupt (IDT index 32 on x86) is generated by the hardware after every fixed intervals

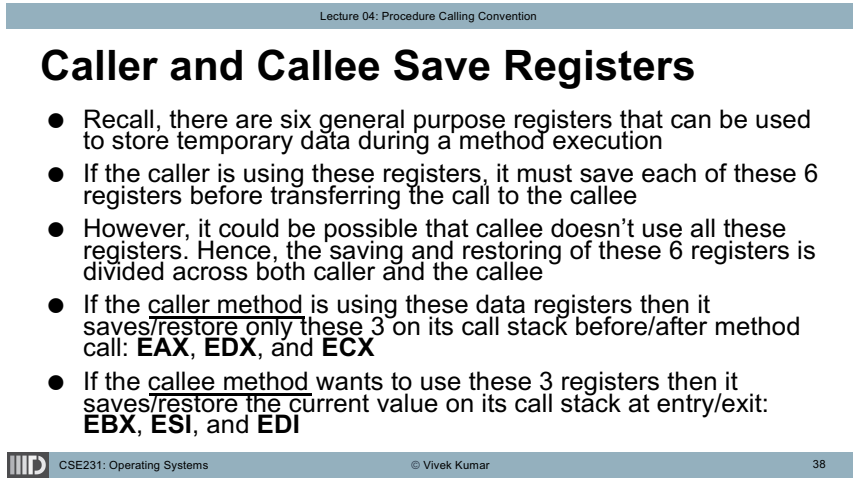
Case-2: Process is Not Going to Block



- Timer interrupt moves the process execution from user stack to its kernel stack (each user process has its own kernel stack)
- User context is saved on the bottom of the kernel stack and interrupt handler is invoked
- The handler will call yield() method as it was a timer interrupt
- The **yield** will invoke a call to the CPU scheduler, which will: a) save the kernel context of the user process at the top of the kernel stack, b) load the context of the CPU scheduler from the corresponding stack, and c) jump to the CPU scheduler stack where it will call the schedule() routine
- Each CPU has its own CPU scheduler process that runs on a separate kernel stack than that of the process it was originally executing

The Context Switch – swtch(OldCtx, NewCtx)

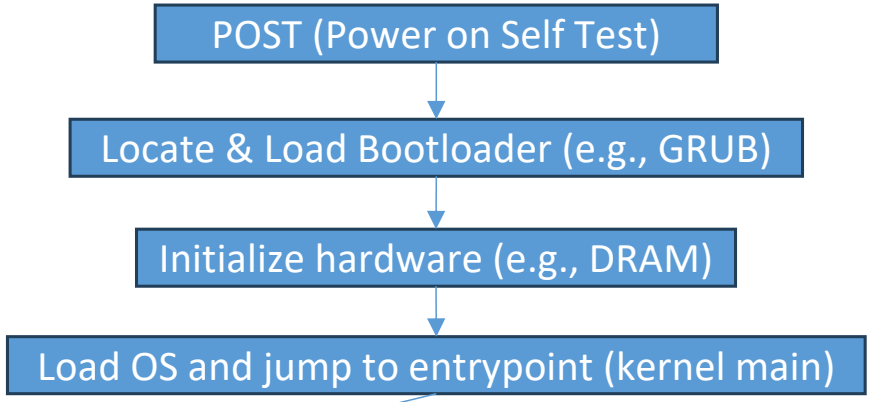
- Typical steps for calling a context switch on x86
  1. Load the parameters passed to the context switching routine in registers, as it is currently on the kernel stack of the user process
     - Parameters would be the context of current process that has to be saved, and the context of the scheduler process that has to be restored
       - The context of the current process will no longer be accessible as the ESP will not be pointing to the old stack after context switch
  2. Load context of the scheduler process
     a) **Load** the **callee save registers** (EBX, ESI, and EDI), EBP, and ESP from the "**top**" of scheduler process stack
     b) Load EIP from scheduler stack and jump to that instruction address



The Juggling API: `scheduler`

- **Basic questions about the design**
  - How many scheduler processes should be there in an OS?
  - How many copies of process table should be there in the OS?
    - Process table is an array of PCB
  - Can there be any race on the process table accesses?
    - How to resolve?

Who Calls the scheduler




```
kernel_main() {
    setup();
    init_all_cpus();
    start_init_process(); // on CPU-0
    for(int cpu=1; cpu<numCPUs; cpu++) {
        launch(cpu, scheduler);
    }
    scheduler(); // on CPU-0
}
```

- OS is booted with the help of another program called bootloader
  - Reads the kernel ELF executable
  - Loads the program segments
  - Jump to the entry point (kernel_main)
  - Call never returns to bootloader (similar to exec)
- Kernel main
  - Complete the basic setup (initializing CPUs, memory, disk, registers, etc.)
  - Launch the first user process (?)
  - Launch one scheduler process on each CPU that will eventually pick a process from the process table and start executing it until it gives up the CPU (see case-1 and case-2 in previous slides)

The Juggling API: `scheduler` Pseudocode

```
void scheduler() {
    while(true) {

        foreach(Process p:                          (process_table)) {




            swtch(scheduler_process, p);
            // p is done for now..

        }

    }
}
```

- Scheduler job is to simply pick a process from the process table
- Context switch of the scheduler process with the process picked from the process table
- Execute the process from the point it was previously stopped
- When the control will be passed back to scheduler from this process?

The Juggling API: `scheduler` Pseudocode

```
void scheduler() {
    while(true) {

        foreach(Process p:                          (process_table)) {
            if(p->state != READY) {
                continue;
            }
            p->state = RUNNING;

            swtch(scheduler_process, p);
            // p is done for now..

        }

    }
}
```

- The scheduler should pay attention that the process picked up from the process table is in READY state (should not be in WAITING state)
- It should change the state of this process from READY to RUNNING
- When control is passed back to the scheduler from this process, its state would have changed to either READY or WAITING

The Juggling API: `scheduler` Pseudocode

```
void scheduler() {
    while(true) {
        lock(process_table);
        foreach(Process p:                          (process_table)) {
            if(p->state != READY) {
                continue;
            }
            p->state = RUNNING;
            unlock(process_table);
            swtch(scheduler_process, p);
            // p is done for now..
            lock(process_table);
        }
        unlock(process_table);
    }
}
```

Does it looks familiar? What would happen if unlock isn't called?

- There is a single process table on a system irrespective of the number of CPUs
- Proper locking mechanism must be ensured to avoid any race condition on the process table by the scheduler processes running at other CPUs
- Modern OS may use scalable implementations of locking to improve performance

The Juggling API: `scheduler` Pseudocode

```
void scheduler() {
    while(true) {
        lock(process_table);
        foreach(Process p: scheduling_algorithm(process_table)) {
            if(p->state != READY) {
                continue;
            }
            p->state = RUNNING;
            unlock(process_table);
            swtch(scheduler_process, p);
            // p is done for now..
            lock(process_table);
        }
        unlock(process_table);
    }
}
```

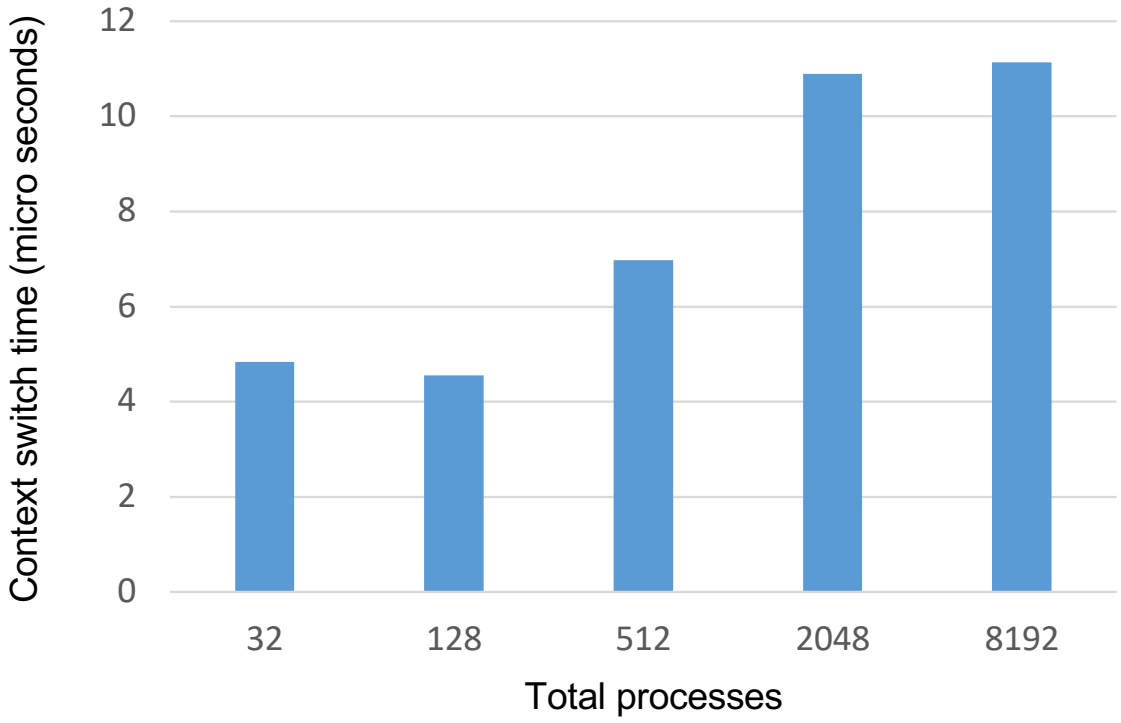
- Process scheduling algorithm plays an important role in the design of operating system
- Different algorithms are chosen according to the need

Today's Class

- Introduction to process scheduling
  - Context switch
  - **Process scheduling policies**
    - First In First Out / First Come First Serve
    - Shortest Job First

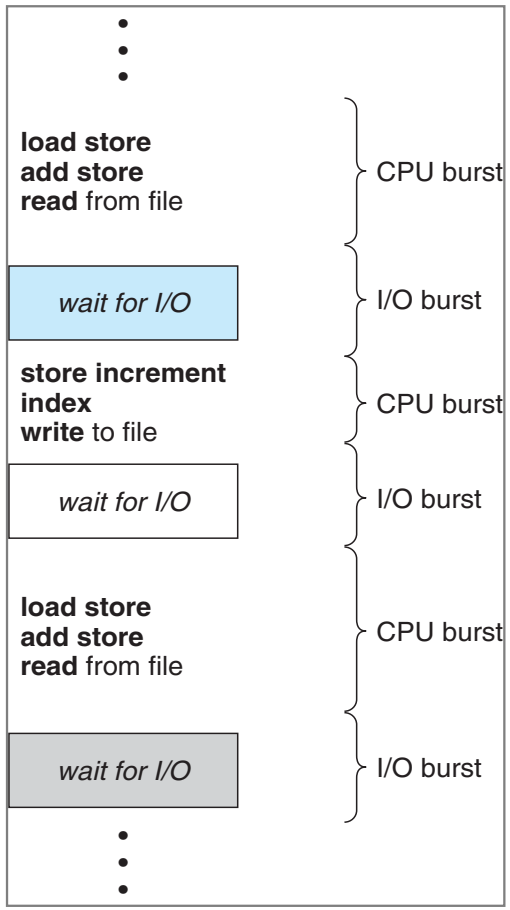
Acknowledgements: Lecture content derived from the CS162 course offered at University of California, Berkeley

Context Switch: Cost?



- Context switch overhead measured on an AMD EPYC 32-core processor running Ubuntu 18.04.3 LTS
  - Data generated using lmbench benchmark (./lat_ctx –s 0 32 128 512 2048 8192)
- Overheads
  - Timer interrupt latency
  - Saving restoring context
  - Process scheduling
  - Loss in cache locality

CPU and I/O Bursts



- Process execution model: programs alternate between bursts of CPU and I/O
  - Program typically uses the CPU for some period of time, then does I/O, then uses CPU again
  - Each scheduling algorithm is about which process to give to the CPU for use by its next CPU burst
  - With time slicing, process may be forced to give up CPU before finishing current CPU burst

Key Scheduling Goals

- **Response Time**
  - What user sees: time to echo a keystroke in editor
- **Throughput**
  - Total operations per second
    - Affected by the overheads of context switching
- **Fairness**
  - Conflicts with the response time
- **Starvation**
  - Due to improper resource allocation policy

First-Come, First-Served (FCFS)

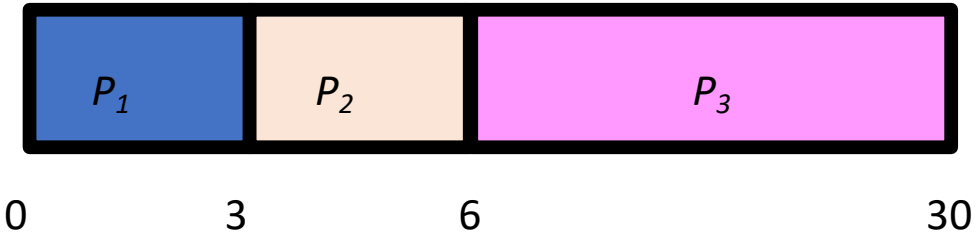
Run tasks in order of arrival.

Run task until completion (or blocks on IO).
No preemption

Also called FIFO

First-Come, First-Served (FCFS)

Process	Burst Time
P_1	3
P_2	3
P_3	24

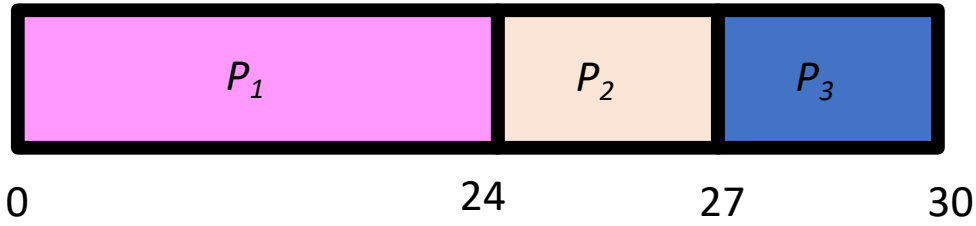


What is the average completion time? $(\frac{3+6+30}{3} = 13)$

What is the average waiting time? $(\frac{0+3+6}{3} = 3)$

First-Come, First-Served (FCFS)

Process	Burst Time
P_3	24
P_2	3
P_1	3

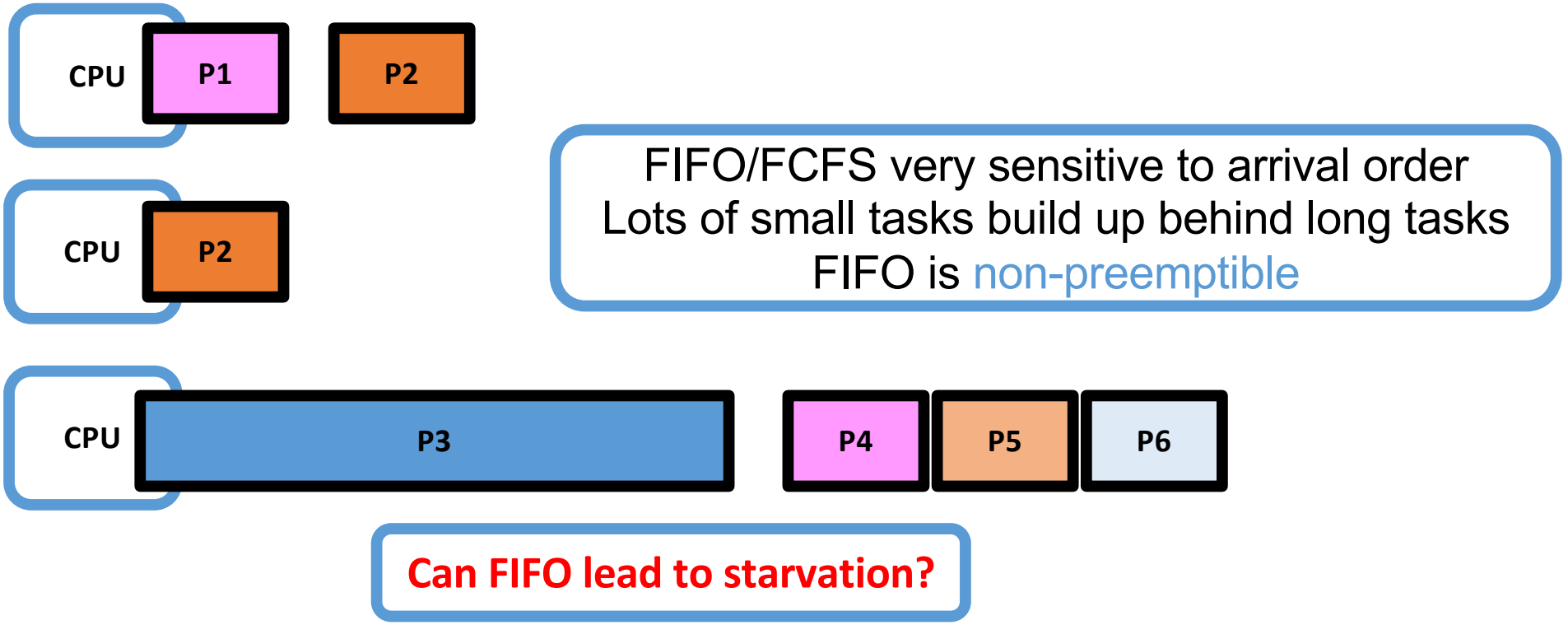


What is the average completion time? $(\frac{24+27+30}{3} = 27)$

What is the average waiting time? $(\frac{0+24+27}{3} = 17)$

Can there be Convoy Effect in FCFS?

Convoy effect: Short process stuck behind long process



Today's Class

- Introduction to process scheduling
  - Context switch
  - **Process scheduling policies**
    - First In First Out / First Come First Serve
    - **Shortest Job First**

Acknowledgements: Lecture content derived from the CS162 course offered at University of California, Berkeley

Shortest Job First

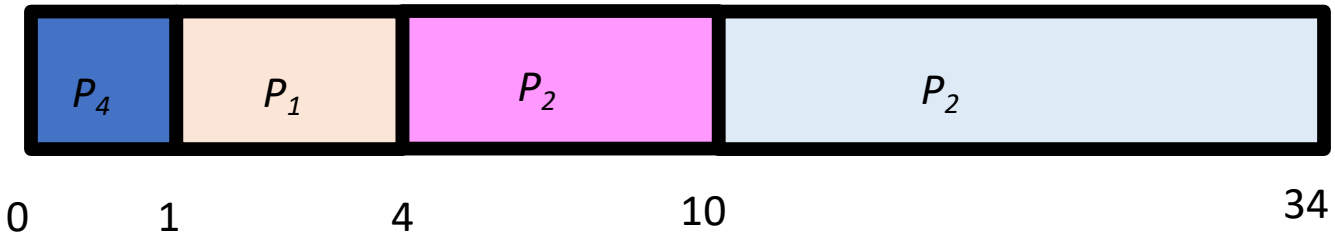
How can we minimize average completion time?

By scheduling jobs in order of

estimated completion time

Shortest Job First

Process	Burst Time
P_1	3
P_2	6
P_3	24
P_4	1

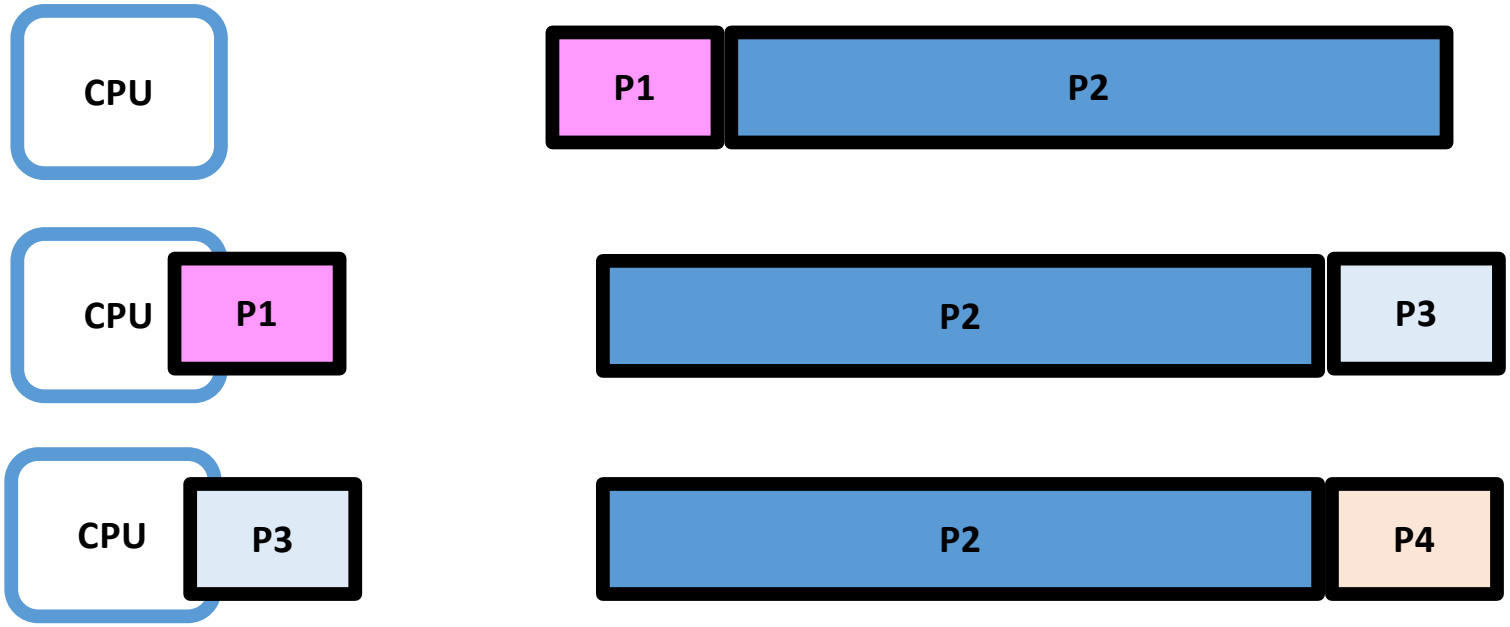


What is the average completion time?

$(\frac{1+4+10+34}{4} = 12.25)$

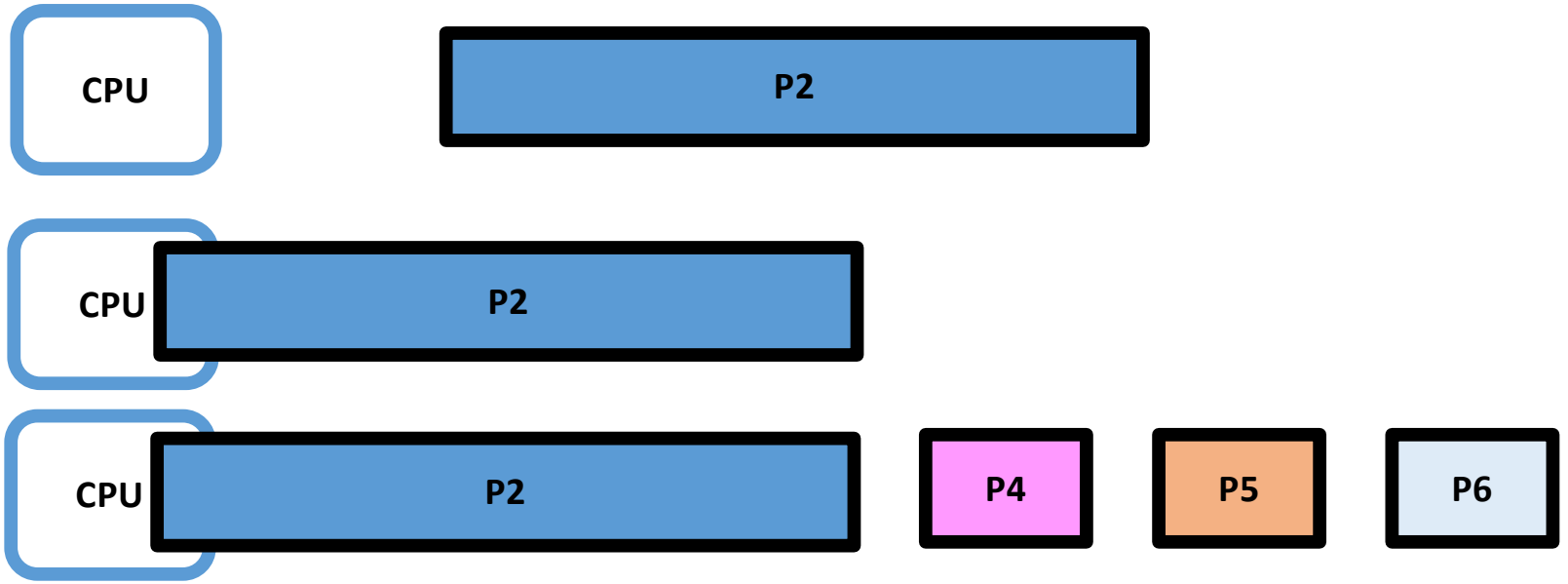
Can SJF Lead to Starvation?

Any scheduling policy that always favours a fixed property for scheduling leads starvation



Can there be Convoy Effect in SJF?

Any non-preemptible scheduling policy suffers from convoy effect



Summary

If jobs arrive simultaneously

Property	FCFS	SJF
Optimize Average Completion Time	✗	✓
Prevent Starvation	✓	✗
Prevent Convoy Effect	✗	✗
Psychic Skills **NOT** Needed	✓	✗

Any scheduling policy that always favors a fixed property for scheduling leads to starvation

Reference Material

- https://pdos.csail.mit.edu/6.828/2018/xv6/book-rev11.pdf
  - You may read the chapter-5 just to have a high-level understanding of scheduling and context switch in a simple OS
  - You can avoid referring to the xv6 codebase as we will not have any assignment on reading/modifying xv6 code

Next Lecture

- Process scheduling policies