COP 4600
Operating Systems Design

Spring 2019

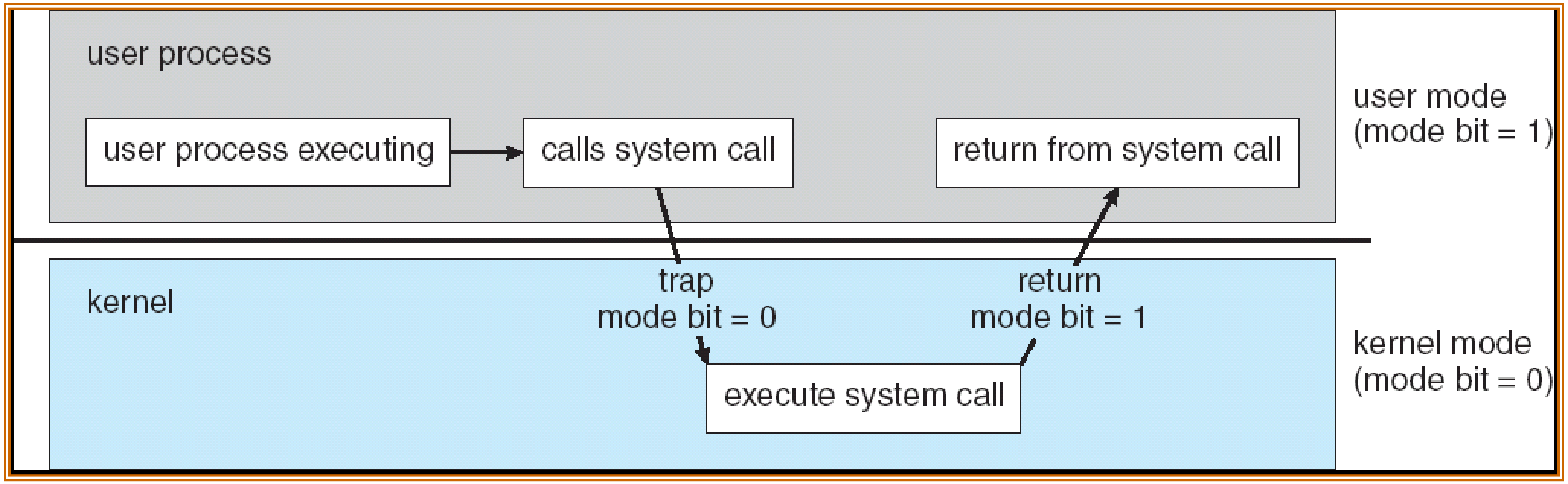
Midterm Review

OS Components

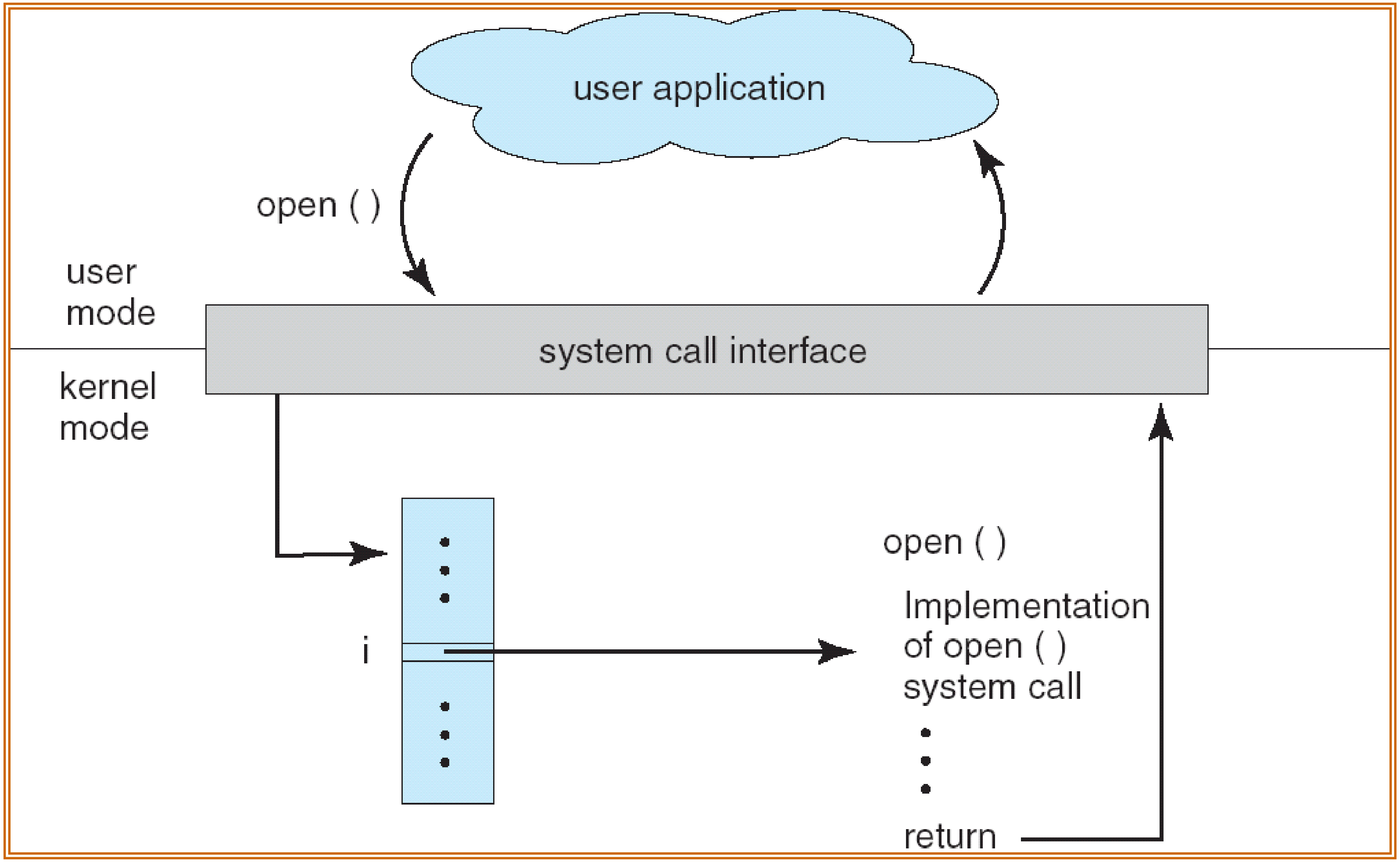
- **Process management**
- **I/O management**
- **Main Memory management**
- **File & Storage Management**
- Networking
- Protection and Security
- User interface

Two-mode of OS

- **Dual-mode** operation allows OS to protect itself and other system components
  - **User mode** and **kernel mode**
  - **Mode bit** provided by hardware
    - Provides ability to distinguish when system is running user code or kernel code
    - Some instructions designated as **privileged**, only executable in kernel mode
    - System call changes mode to kernel, return from call resets it to user

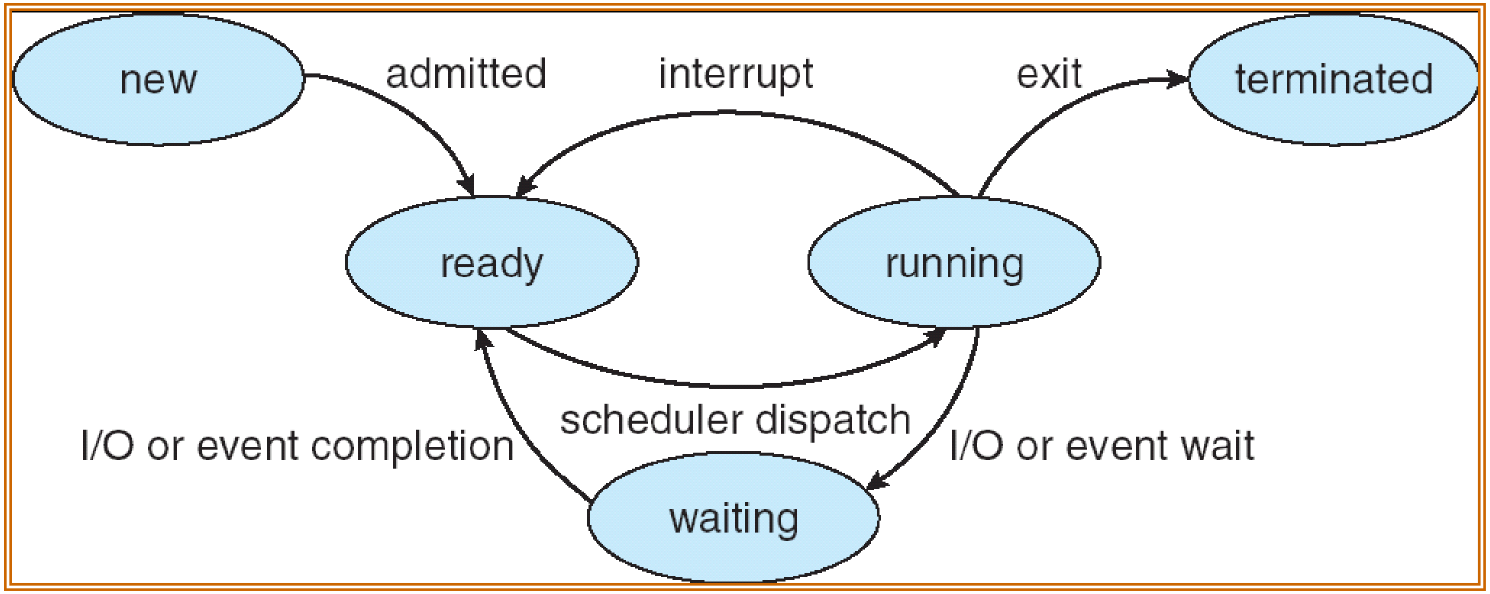


API – System Call – OS Relationship

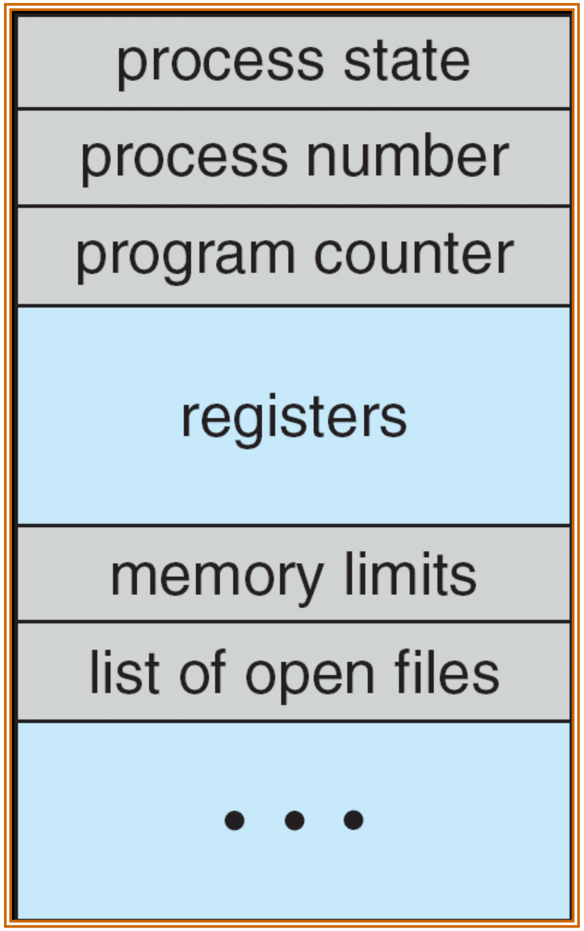


Process State: 3 or 5 states

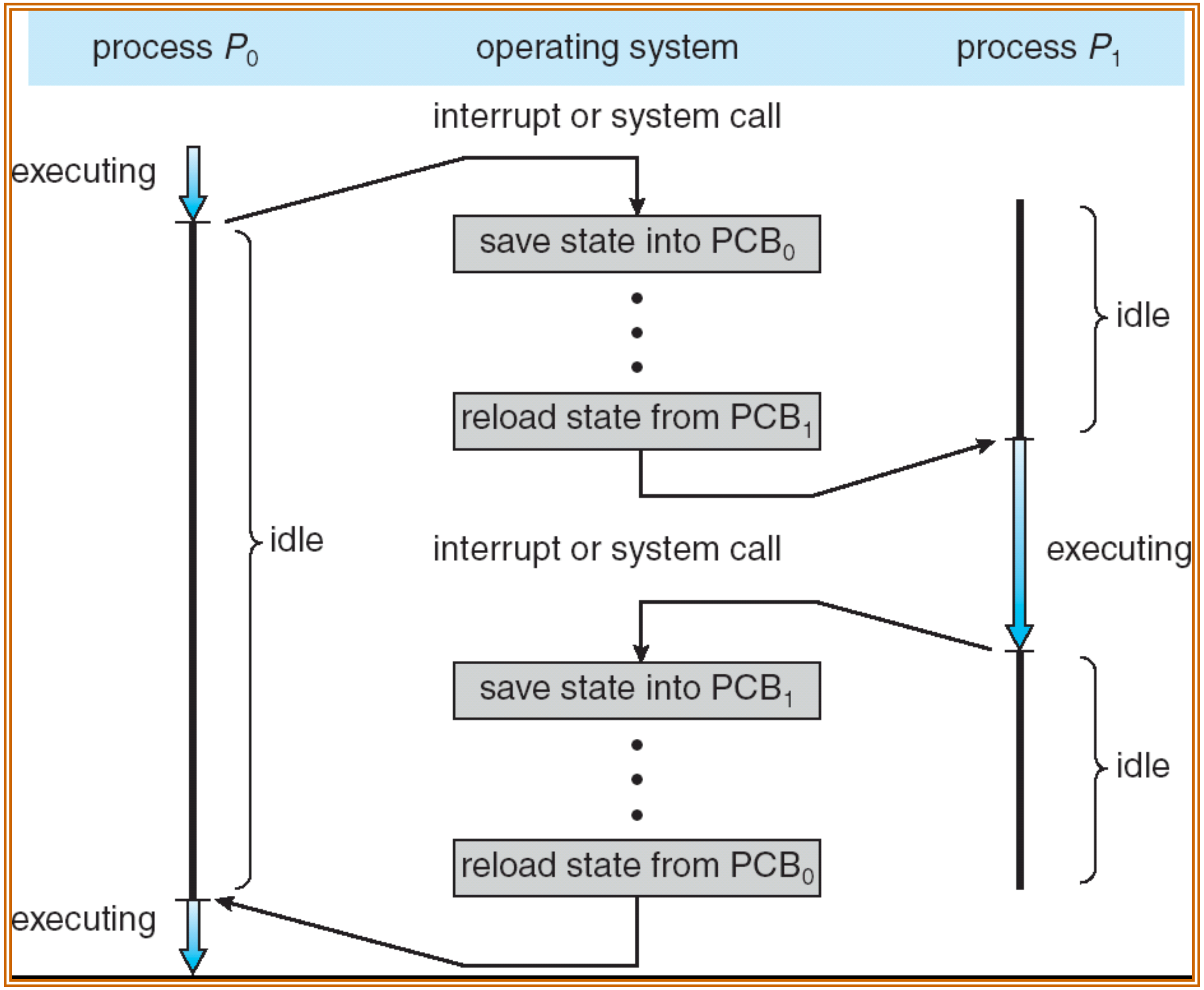
- As a process executes, it changes *state*
  - **new**: The process is being created
  - **running**: Instructions are being executed
  - **waiting**: The process is waiting for some event to occur
  - **ready**: The process is waiting to be assigned to a processor
  - **terminated**: The process has finished execution



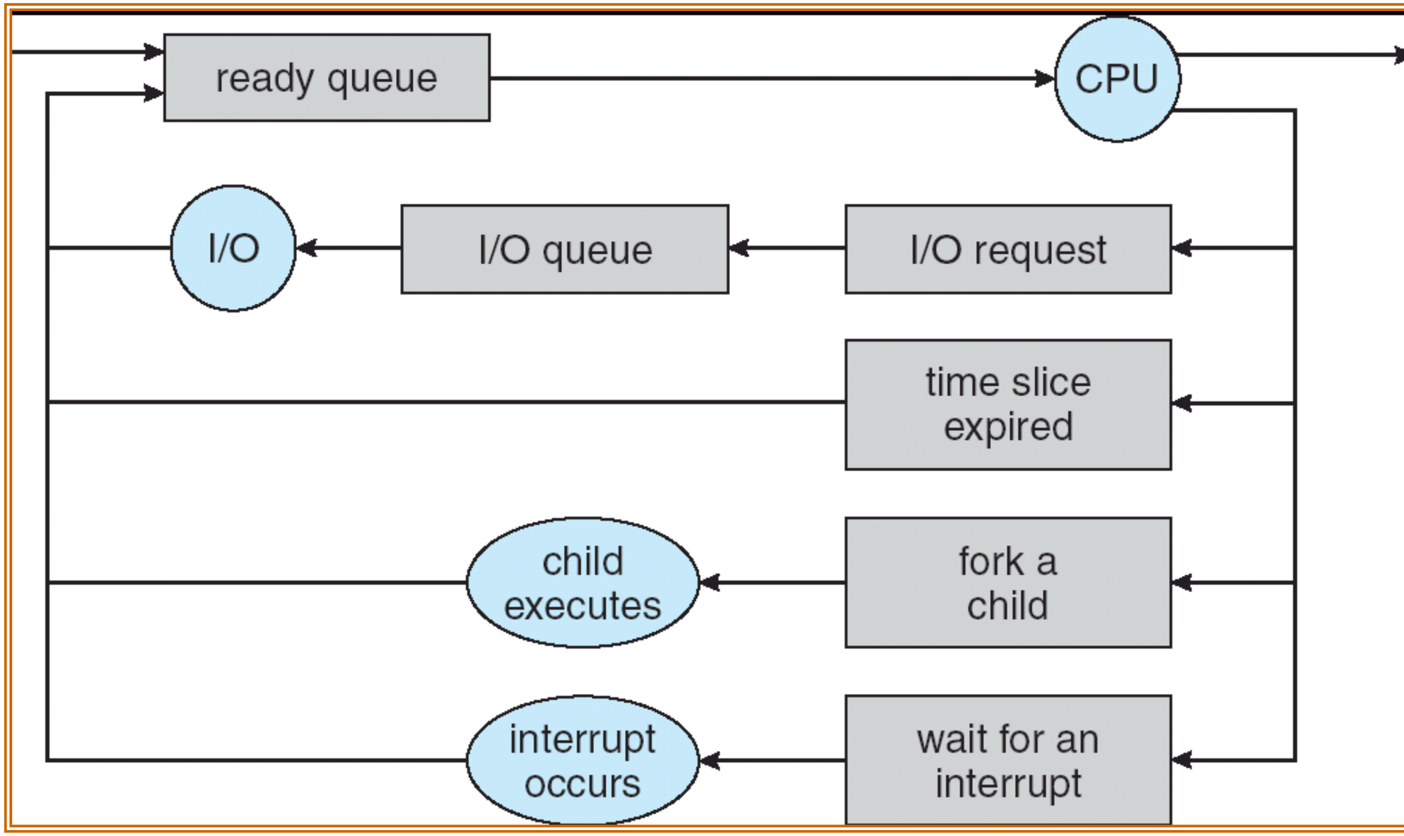
Process Control Block (PCB)



CPU Switch From Process to Process



Representation of Process Scheduling



Schedulers

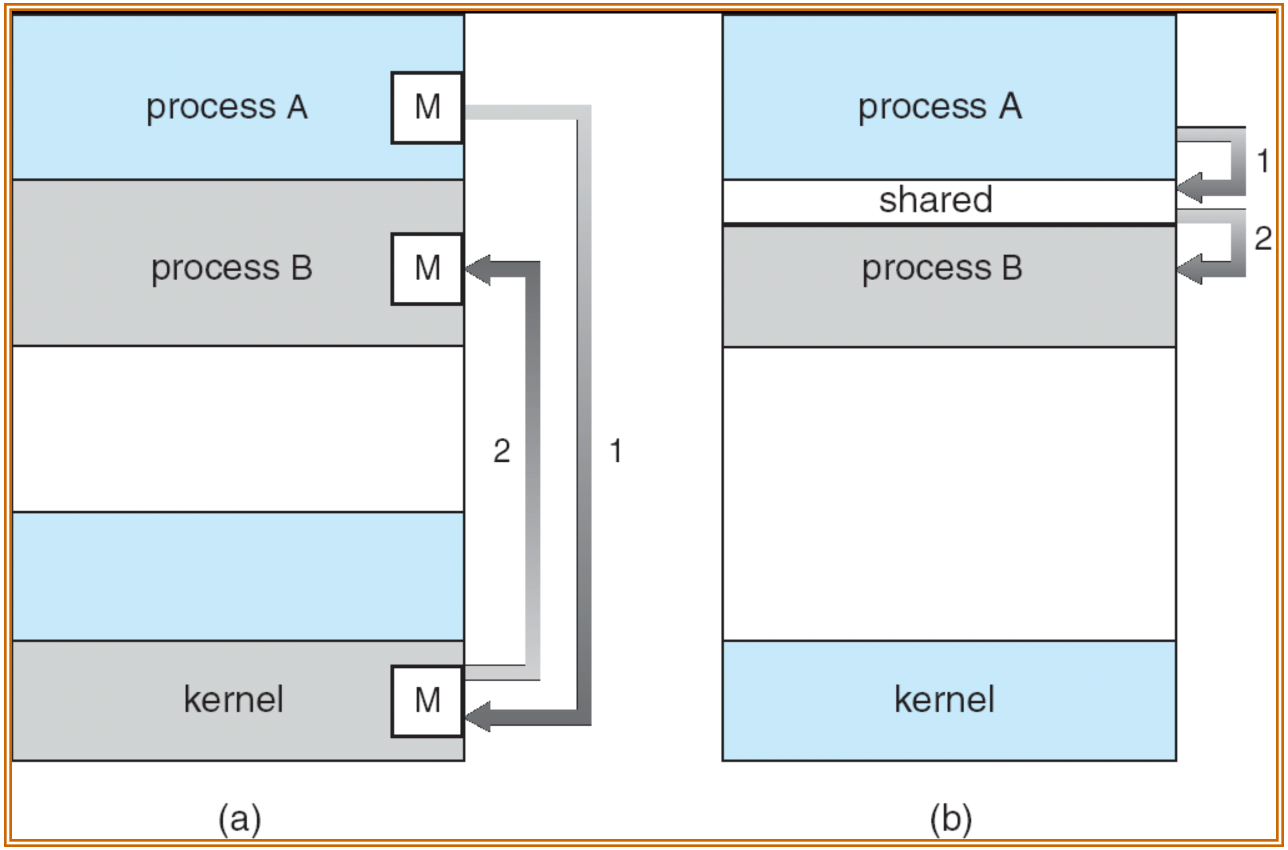
- Long-term scheduler
- Short-term scheduler
- Medium Term Scheduling

Creating a New Process - fork()

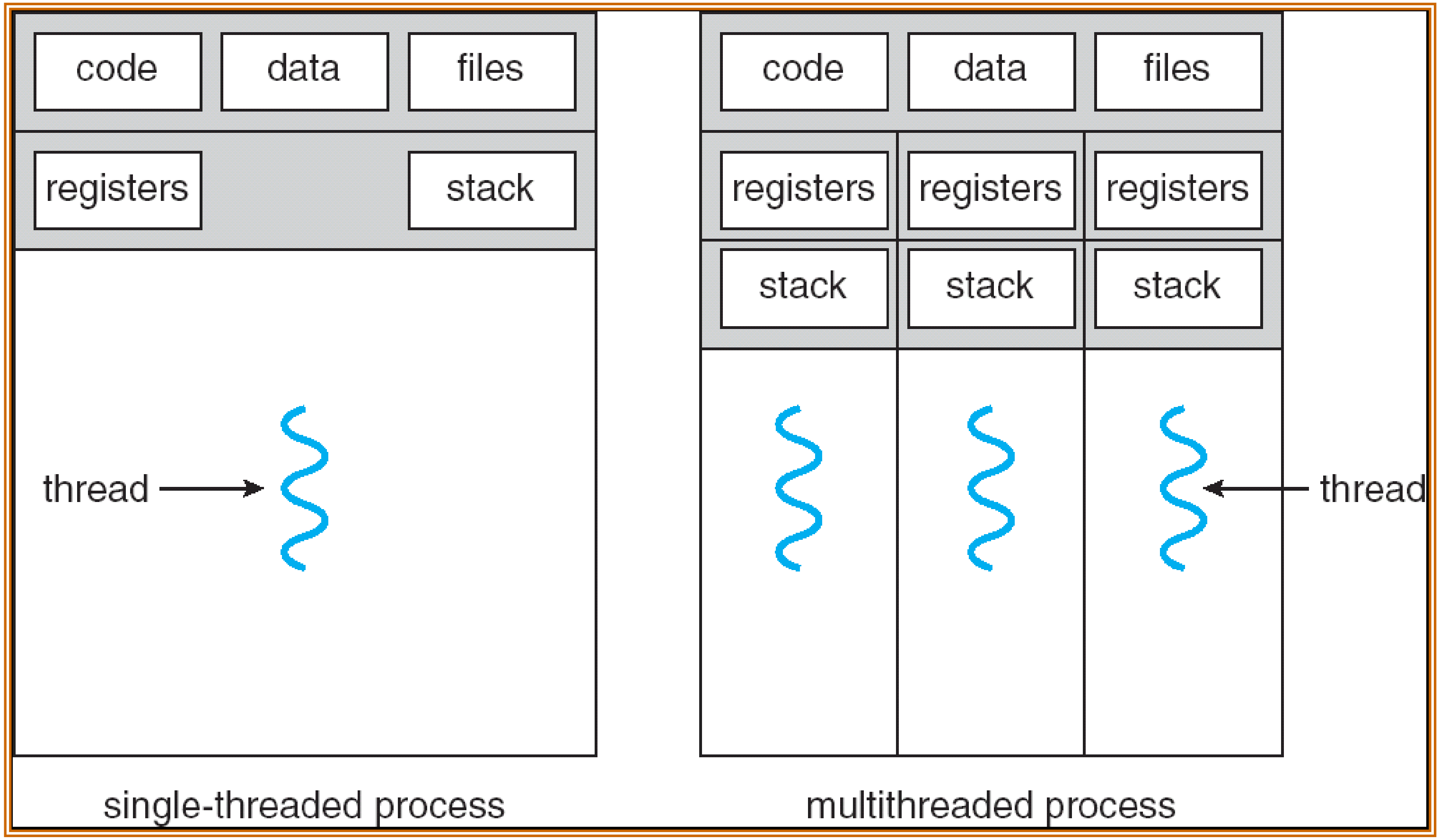
```
pid = fork();

if (pid == -1) {
    fprintf(stderr, "fork failed\n");
    exit(1);
}

if (pid == 0) {
    printf("This is the child\n");
    exit(0);
}

if (pid > 0) {
    printf("This is parent. The child is %d\n", pid);
    exit(0);
}
```

Interprocess Communication (IPC)

- Mechanism for processes to communicate and to synchronize their actions
  - Shared Memory (shown in b)
  - Message Passing (shown in a)



Single and Multithreaded Processes



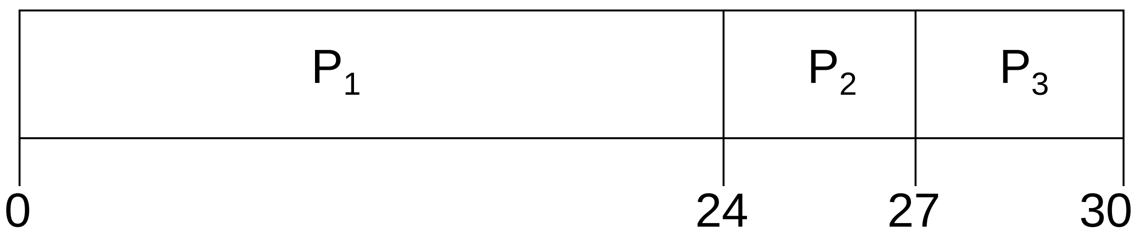
CPU Scheduler

- Selects from among the processes in the ready queue, and allocates the CPU to one of them
- Non-preemptive Scheduling
- Preemptive scheduling

First-Come, First-Served (FCFS) Scheduling

Process	Burst Time
P_1	24
P_2	3
P_3	3

- Suppose that the processes arrive in the order: P_1 , P_2 , P_3

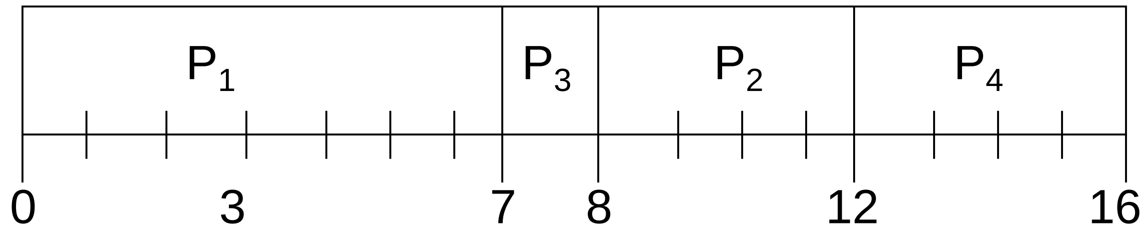


- Waiting time
  - $P_1 = 0$; $P_2 = 24$; $P_3 = 27$
- Average waiting time:
  - $(0 + 24 + 27)/3 = 17$

Example of Non-Preemptive Shortest-Job-First

Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

- SJF (non-preemptive)



- Average waiting time = (0 + 6 + 3 + 7)/4 = 4

Priority Scheduling

- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority (smallest integer ≡ highest priority)
  - Preemptive
  - nonpreemptive
- SJF is a priority scheduling where priority is the predicted next CPU burst time
- Problem
  - Starvation – low priority processes may never execute
- Solution
  - Aging – as time progresses increase the priority of the process

Example of Round Robin with Time Quantum = 20

Process	Burst Time
P_1	53
P_2	17
P_3	68
P_4	24

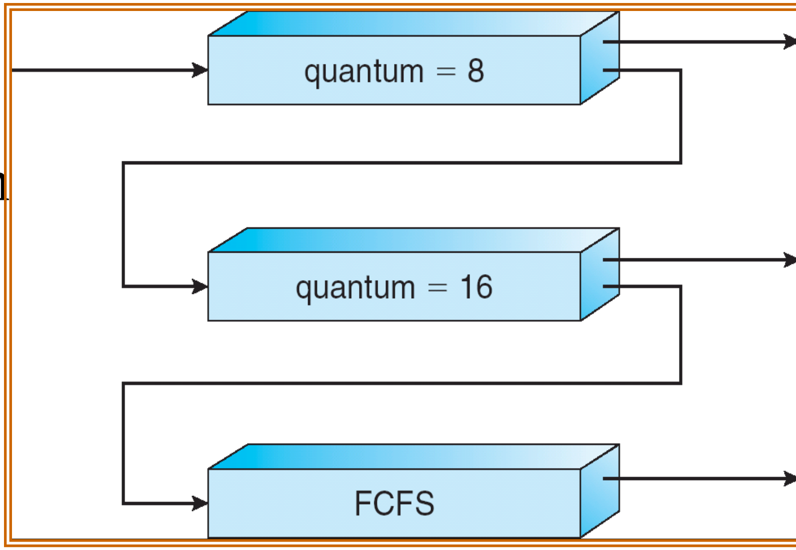
- The Gantt chart is:

P_1	P_2	P_3	P_4	P_1	P_3	P_4	P_1	P_3	P_3

0 20 37 57 77 97 117 121 134 154 162

- Typically, higher average turnaround than SJF, but better *response*

Example of Multilevel Feedback Queue



- Three queues:
  - Q_0 – RR with time quantum 8 millisecon
  - Q_1 – RR time quantum 16 milliseconds
  - Q_2 – FCFS
- Scheduling
  - A new job enters queue Q_0 which is served *RR*. When it gains CPU, job receives 8 milliseconds. If it does not finish in 8 milliseconds, job is moved to queue Q_1.
  - At $Q_{1,}$ the job is again served and receives 16 additional milliseconds. If it still does not complete, it is preempted and moved to queue Q_2.

Race Condition

- A race condition occurs when you have 2 or more processes sharing resources and what happens depends on the order that they run in.
- How to avoid race condition?
  - Ensure no two processes are <u>ever</u> in their shared critical section at the same time.

Semaphore

- Synchronization tool that **does not require busy waiting**
- Semaphore *S* – integer variable
  - Provides a way to count the number of sleep/wakeups invoked
- Two standard operations modify S: wait() and signal()
  - Originally called P() and V()
- Semantics of wait() and signal()

Semaphore as General Synchronization Tool

- Two kinds of semaphore
  - Counting semaphore
    - integer value can range over an unrestricted domain
  - Binary semaphore
    - integer value can range only between 0 and 1; can be simpler to implement
    - Also known as mutex locks
  - Provides mutual exclusion

```
Semaphore S;    //  initialized to 1
wait (S);
  Critical Section
signal (S);
```

Monitors

- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- Only one process may be active within the monitor at a time

```
monitor monitor-name
{
  // shared variable declarations
  procedure P1 (…) { …. }
        …

  procedure Pn (…) {……}

   Initialization code ( ….) { … }
        …
  }
}
```

Condition Variables

- condition x, y;

- Two operations on a condition variable:
  - x.wait () – a process that invokes the operation is suspended.
  - x.signal () – resumes one of processes (if any) that invoked x.wait ()

Application of Synchronization

- Classical problems
  - Bounded-Buffer Problem
  - Dining-Philosophers Problem
  - Readers and Writers Problem
- How to use semaphore/monitor to solve the problems?

Solaris Synchronization Example

- Uses *adaptive mutexes* for efficiency when protecting data from short code segments.
  - Protects every critical data item
  - With a multiprocessor system, it has two functions and if the data is locked
    1. If lock is held by running thread, then use a ***spinlock*** (busy loop).
    2. If lock is held by blocked thread, then go to sleep.
  - Single processor, uses only #2
- Uses *readers-writers* locks when longer sections of code need access to data.
  - Readers-Writers locks protect data is read often, but needs little updating.

Deadlock Characterization

Deadlock can arise if four conditions hold simultaneously.

- Mutual exclusion
- Hold and wait
- No preemption
- Circular wait

Methods for Handling Deadlocks

- Prevention and avoidance
  - Ensure that the system will *never* enter a deadlock state.
- Detection and recovery
  - Allow the system to enter a deadlock state and then recover.
- Ostrich strategy
  - Ignore the problem and pretend that deadlocks never occur in the system; used by most operating systems.

Example of Banker's Algorithm

- 5 processes P_0 through P_4;

  3 resource types:

  A (10 instances), B (5instances), and C (7 instances).

- Snapshot at time T_0:

	Allocation	*Max*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	7 5 3	3 3 2
P_1	2 0 0	3 2 2	
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_4	0 0 2	4 3 3	