KOÇ UNIVERSITY

Midterm Review

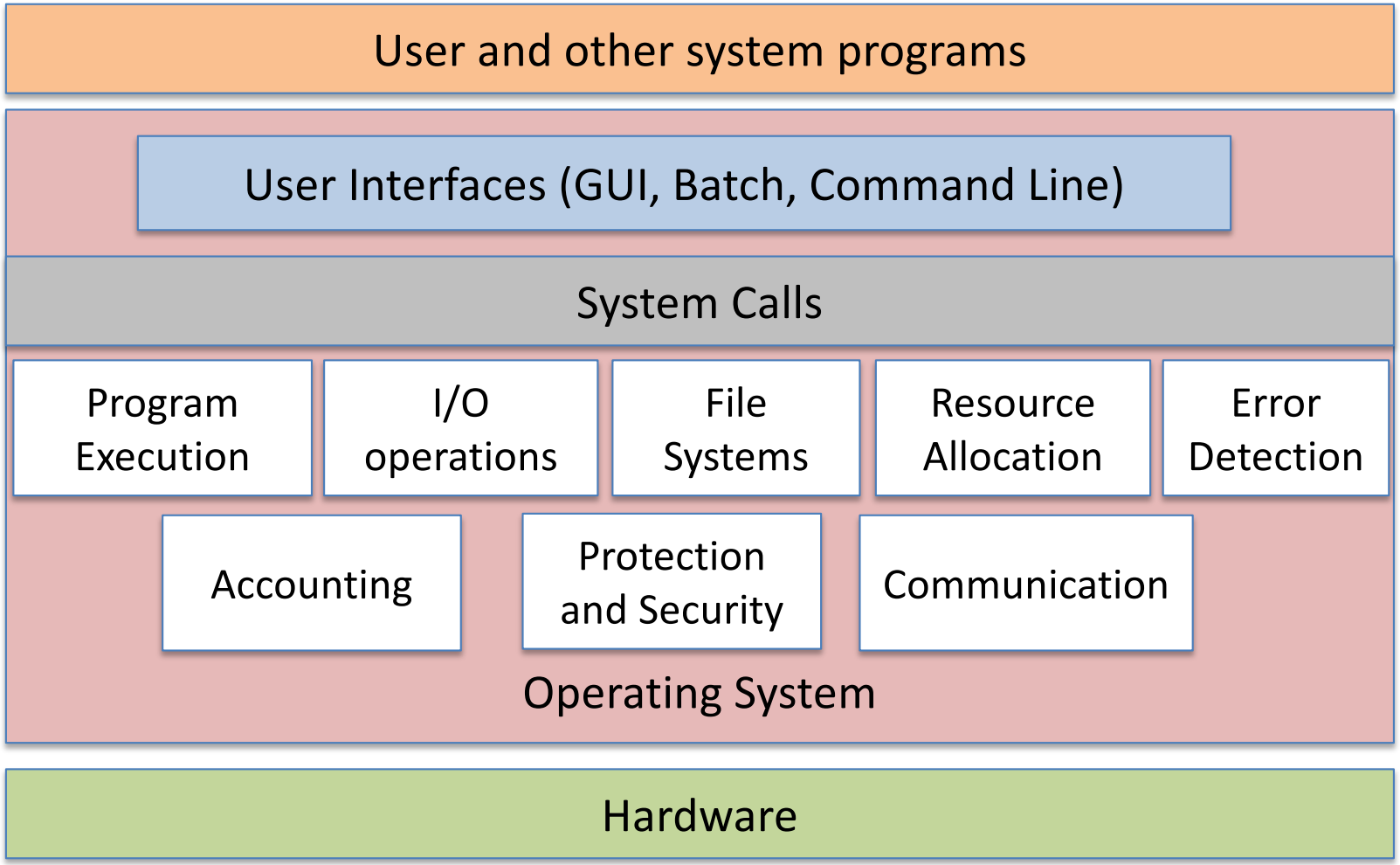
Didem Unat

COMP304 - Operating Systems (OS)

Computer Startup

- **Bootstrap program** is loaded at power-up or reboot
  - Typically stored in ROM, generally known as **firmware**
  - Initializes all aspects of a system
  - Loads operating system **kernel** into main memory and starts execution
    - The first system process is '**init**' (or systemd) in Linux
  - When the system is fully booted, it waits for some event to occur
- Kernel
  - The ``one" program running at all times (the core of OS)
    - Everything else is an application program
- Process
  - An executing program (active program)

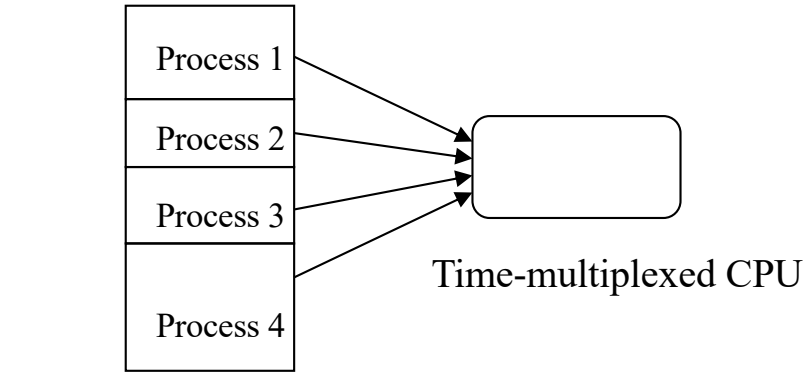
Operating System Services



How Multiprogramming Works

- **Multiprogramming** needed for efficiency
  - Single user or program cannot keep CPU and I/O devices busy at all times
  - Organize processes so that CPU always has one process to execute
  - A subset of total jobs is kept in main memory

- One job selected and run via **CPU scheduling**
  - When it has to wait (for I/O for example), OS switches to another job



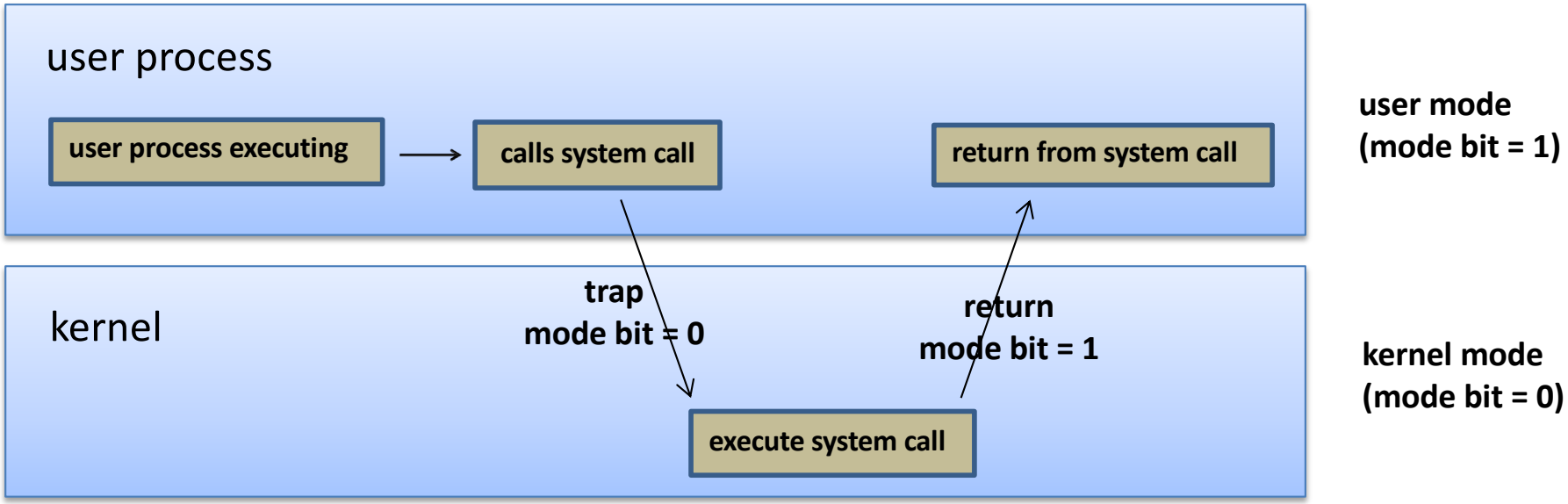
OS Protection: Dual-Mode Operation

- **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
    - For example, I/O related instructions are privileged

- Ensures that an incorrect program cannot cause other programs to execute incorrectly.

Transition from User to Kernel Mode

- **System Call**
  - Results in a transition from user to kernel mode
  - Return from call resets it to user mode
- Software error or a user request creates an exception or trap



System Calls

- Programming interface to the services provided by the OS
  - Well-defined and safe implementation for service requests
  - Typically written in a high-level language (C or C++)
- A typical OS executes 1000s of system calls per second

- Mostly accessed by programs via a high-level Application Program Interface (API) rather than direct system call use
  - Wrapper functions for the system calls

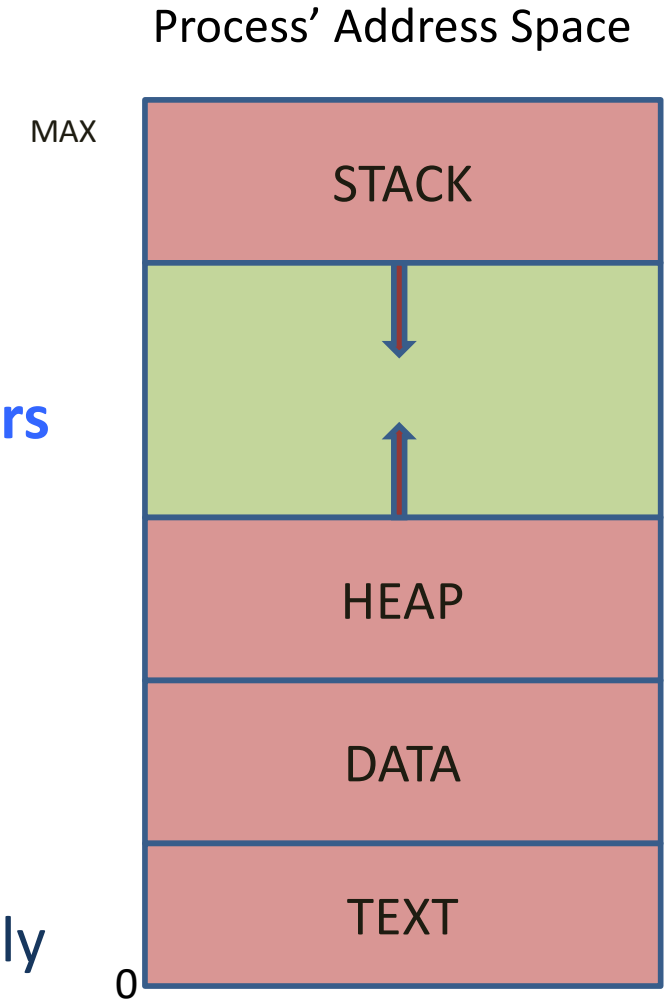
- Three most common APIs are
  - Windows API for Windows,
  - POSIX API for POSIX-based systems (including virtually all versions of UNIX, Linux, and Mac OS X),
  - Java API for the Java virtual machine (JVM)

Privileged Instructions

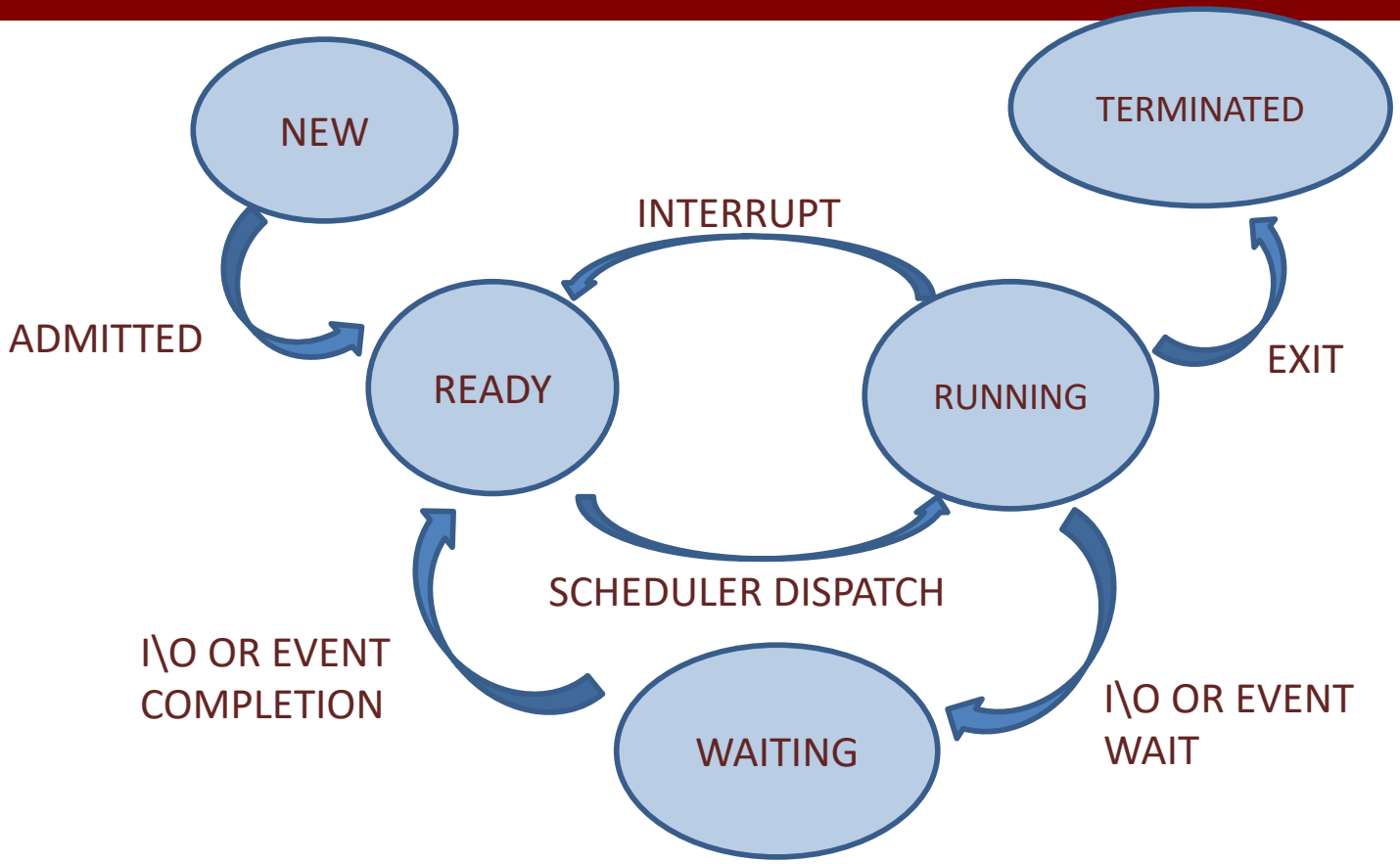
- The dual mode of operation provides us with the means for protecting the operating system from errant users—and errant users from one another.
- We accomplish this protection by designating some of the machine instructions that may cause harm as **privileged instructions**.
  - The hardware allows privileged instructions to be executed only in kernel mode.
  - If an attempt is made to execute a privileged instruction in user mode, the hardware does not execute the instruction but rather treats it as illegal and traps it to the operating system

Process

- The program code, also called **text section**
  - **Binary code**
- Current activity includes **program counter** and other processor **registers**
- **Stack** containing temporary data
  - Function parameters, return addresses, local variables
- **Data section** containing global variables
- **Heap** containing memory dynamically allocated during run time



Transition between Process States



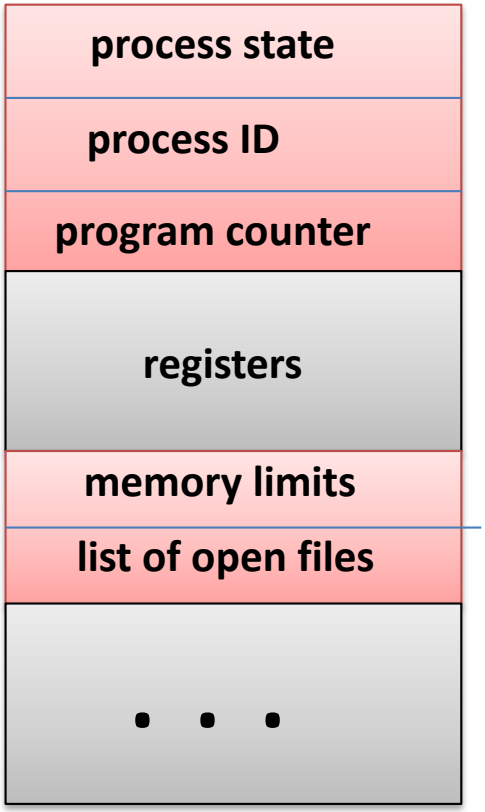
- Process transitions from one state to another
- An animation for process states
  - https://www.youtube.com/watch?v=Y3mQYaQsrvg

Process Control Block (PCB)

Keeps the process context

- **Process state** – running, waiting, etc
- **Program counter** – location of instruction to next execute
- **CPU registers** – contents of all process registers
- **CPU scheduling information**- priorities, scheduling queue pointers
- **Memory-management information** – memory allocated to the process
- **Accounting information** – CPU used, clock time elapsed since start, time limits
- **I/O status information** – I/O devices allocated to process, list of open files

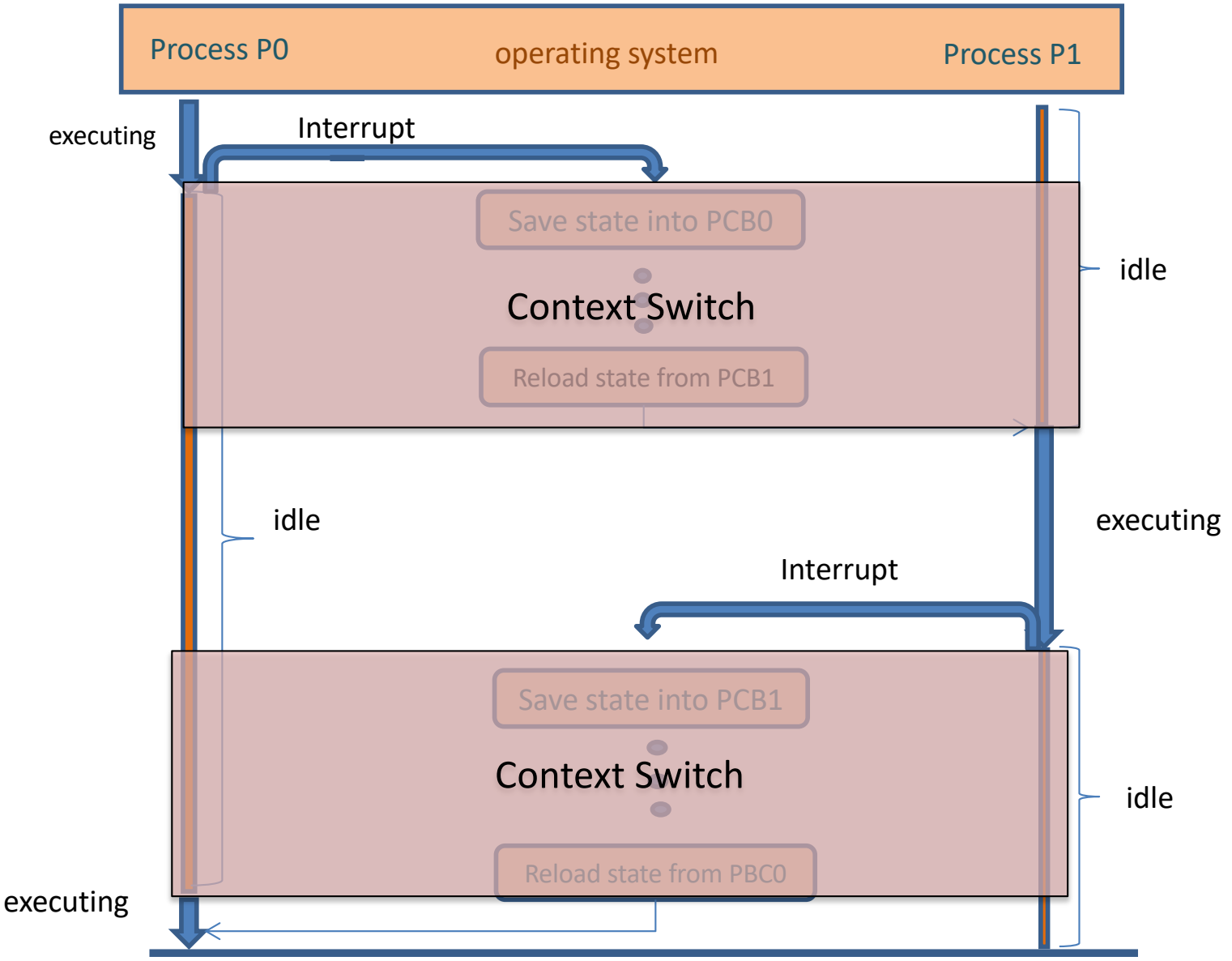
Metadata about a process



Context Switch

- OS needs to store and restore the context of a process
  - So that execution of the process can be resumed from the same point at a later time
  - This is called **context switch**

- When does OS switch context?
  - In case of an interrupt
  - When process's time is up
    - Even though process has still some work to do
  - When a process terminates



- Switching between threads of a single process can be faster than between two separate processes

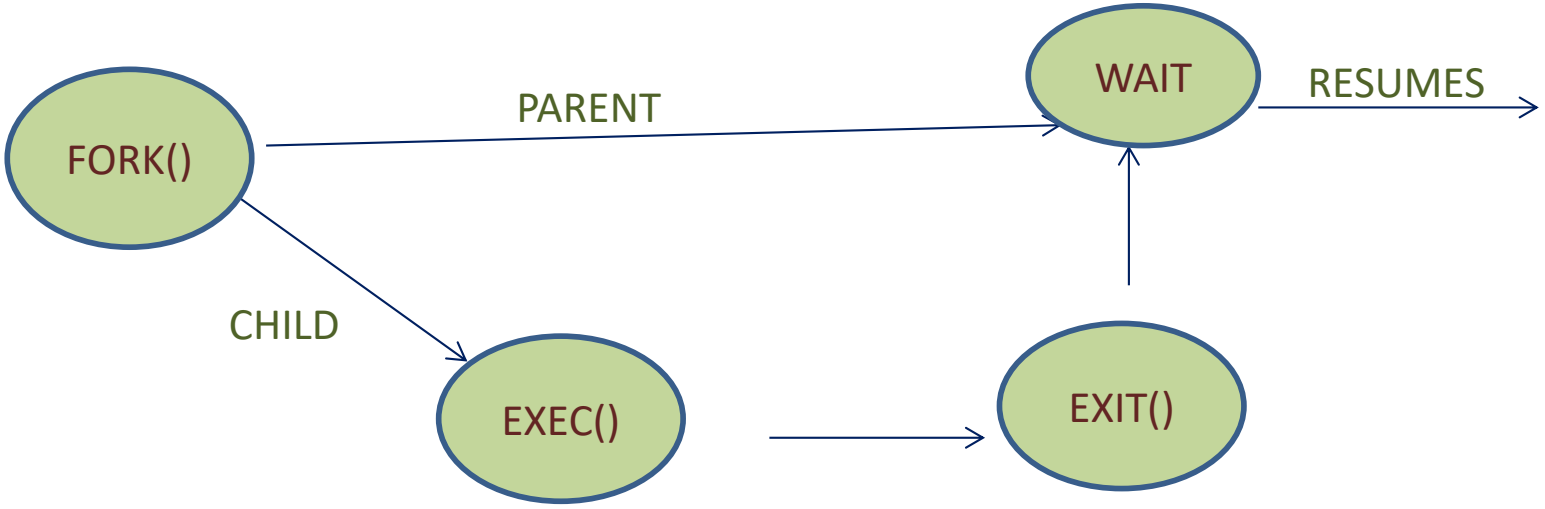
Context Switch

- Context switch is **pure overhead**
  - Why?
  - Should be very small
    - 100s of nanoseconds to couple microseconds
    - Exact time depends on the CPU and size of the context
  - Hardware support for context switching improves the performance

Process Creation

- Address space
  - Child duplicate of parent
  - Child has a program loaded into it

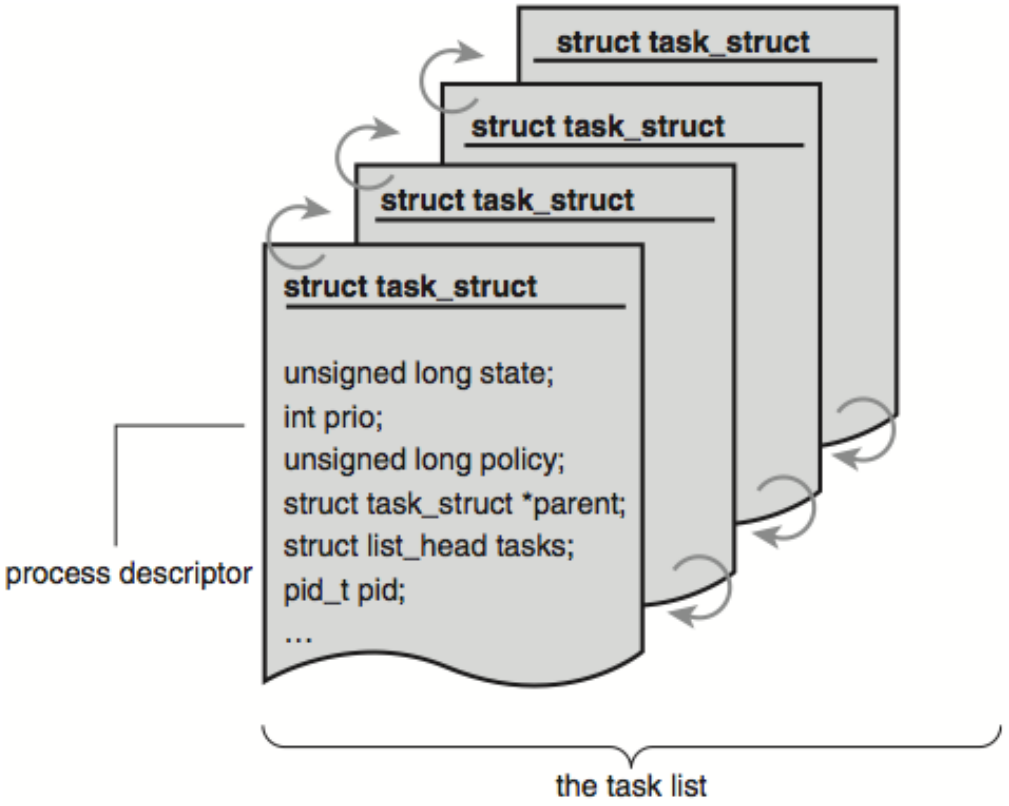
- UNIX examples
  - **fork()** system call creates a new process
  - **exec()** system call is used after a **fork()** to replace the process' memory space with a new program



task_struct

- Represented by the C structure

```
task_struct {
    pid_t pid;
    /* process identifier */
    long state;
    /* state of the process */
    unsigned int time_slice
    /* scheduling information */
    struct task_struct *parent;
    /* this process's parent */
    struct list_head children;
    /* this process's children */
    struct files_struct *files;
    /* list of open files */
    struct mm_struct *mm;
    /* address space of this process */
   ...
```



Search for task_struct (line ~1200)
https://elixir.bootlin.com/linux/latest/source/include/linux/sched.h

Creating/Destroying Processes

- UNIX `fork()` creates a process
  - Creates a new address space
  - Copies text, data, & stack into new address space
  - Provides child with access to open files of its parent

- UNIX `wait()` allows a parent to wait for a child to change its state
  - This is a blocking call, parent waits until it receives a signal
  - http://linux.die.net/man/2/wait

- UNIX `exec()` system call variants (e.g.execve()) allow a child to run a new program

Zombie

- A **zombie process** or **defunct process** is a process that has completed execution (via the exit system call)
- It still has an entry in the process table: it is a process in the "Terminated state".
- This occurs for child processes, where the entry is still needed to allow the parent process to read its child's exit status:
  - Once the exit status is read via the wait system call, the zombie's entry is removed from the process table and it is said to be "reaped"

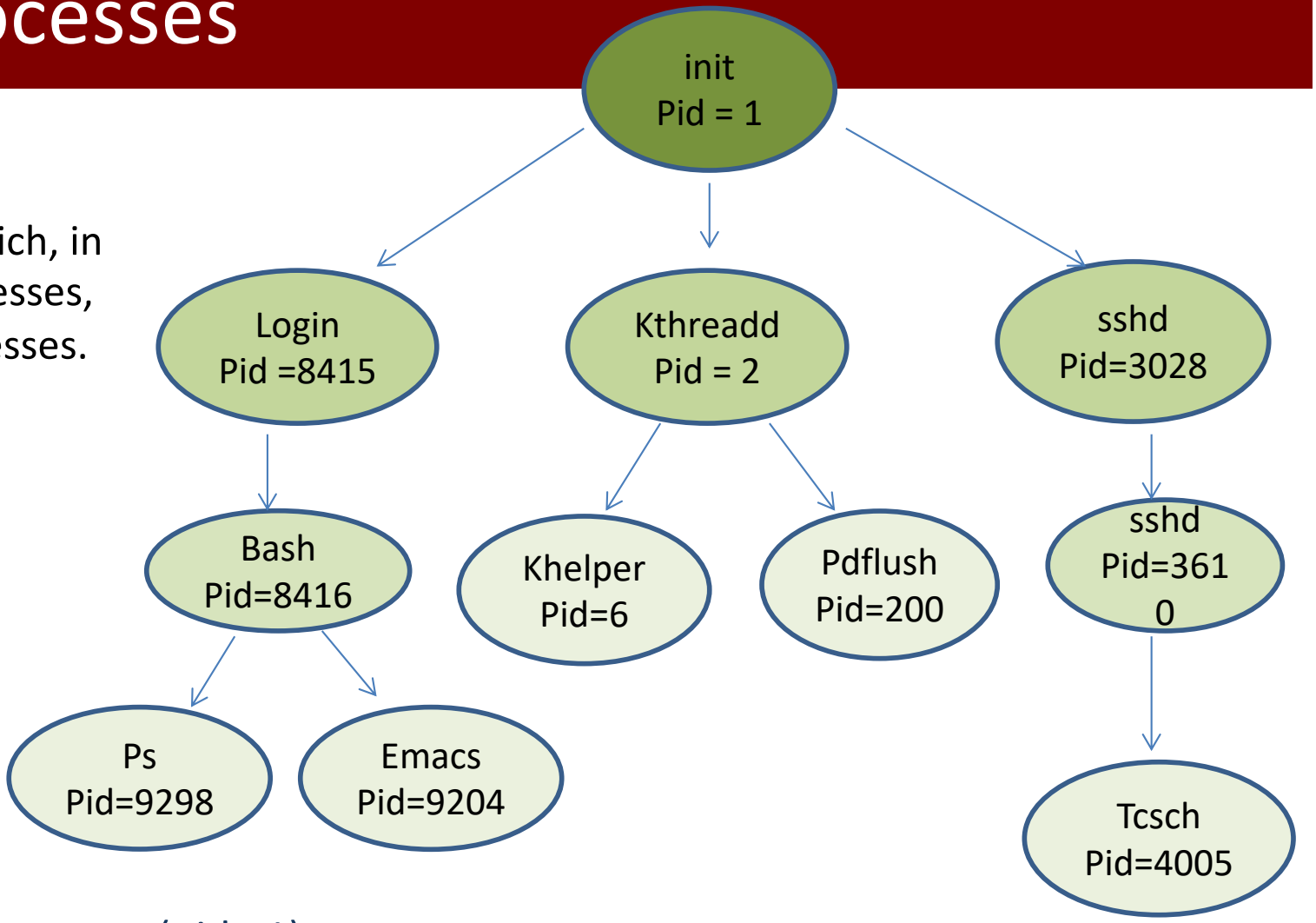
What happens if the parent dies before the child?

Orphans

- Some operating systems do not allow child to continue without its parent
  - All children are terminated - **cascading termination**
- More common: If parent terminates, still executing children processes are called **orphans**
  - Those are adopted by *init* process
- **Init** periodically calls wait to terminate orphans

Tree of Processes

Parent process creates children processes, which, in turn create other processes, forming a tree of processes.

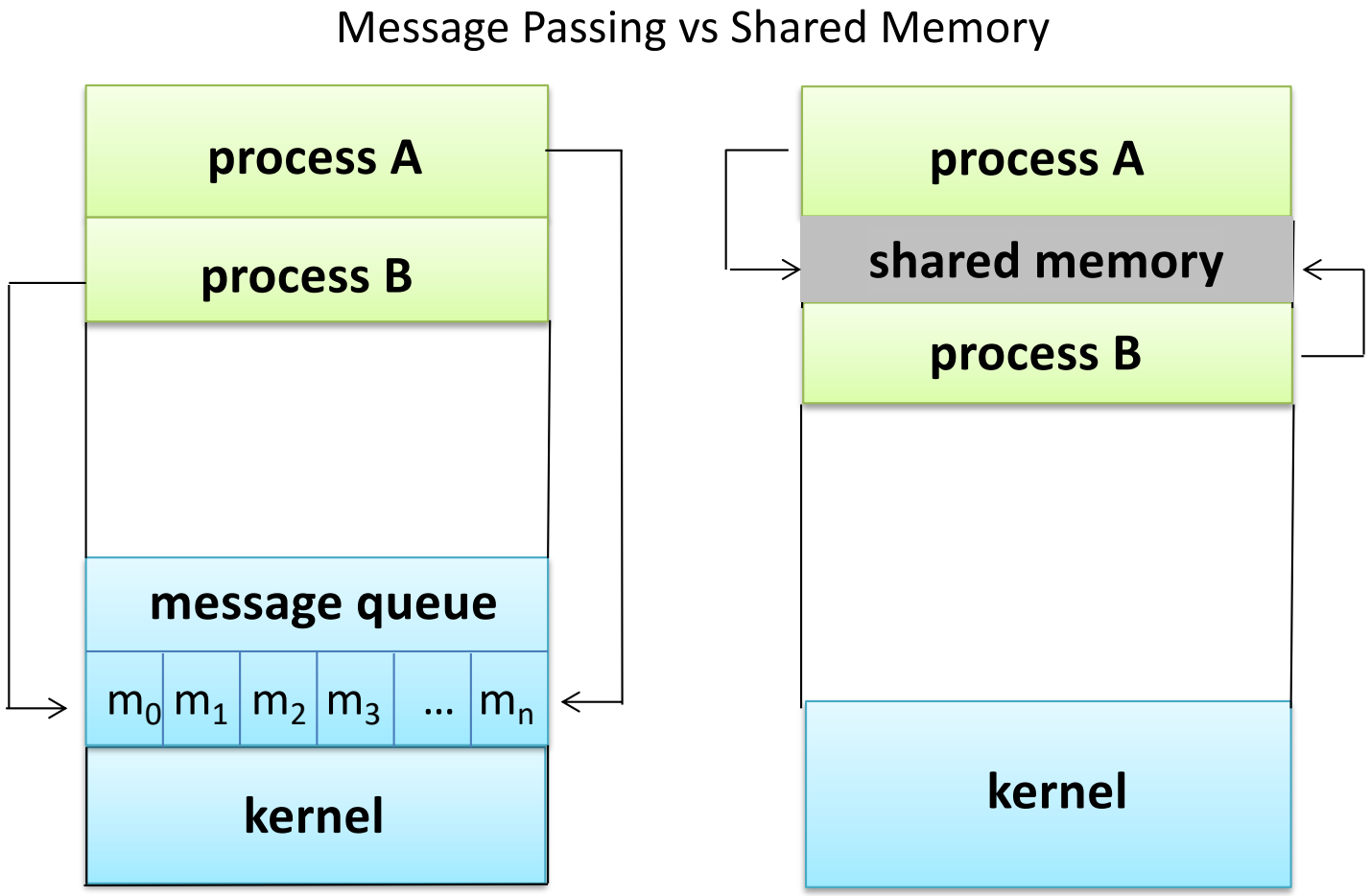


- init is very first process (pid =1)
- kthread is for system processes (pid=2)
- login process is for users directly logged in to the system
- sshd process is for users remotely logged in to the system
  - Starts an openSSH SSH daemon

Inter-process Communication (IPC)

- *An independent* process cannot affect or be affected by the execution of another process.
- *Cooperating* processes can affect or be affected by the execution of other processes
- Cooperating processes need methods for **inter-process communication (IPC)**
- There are two models of IPC
  - **Shared memory**
  - **Message passing**

Two Models of Communication



- Message passing requires the message of A to be copied to a buffer and copied to process B's memory – thus it is slightly slower but safer

Blocking or Nonblocking ?

- Message passing may be either blocking or non-blocking

- **Blocking** is considered **synchronous**
  - **Blocking send** has the sender block until the message is received
  - **Blocking receive** has the receiver block until a message is available
- **Non-blocking** is considered **asynchronous**
  - **Non-blocking** send has the sender send the message and continue
  - **Non-blocking** receive has the receiver receive a valid message or null

Blocking or non-blocking?

Blocking or non-blocking?

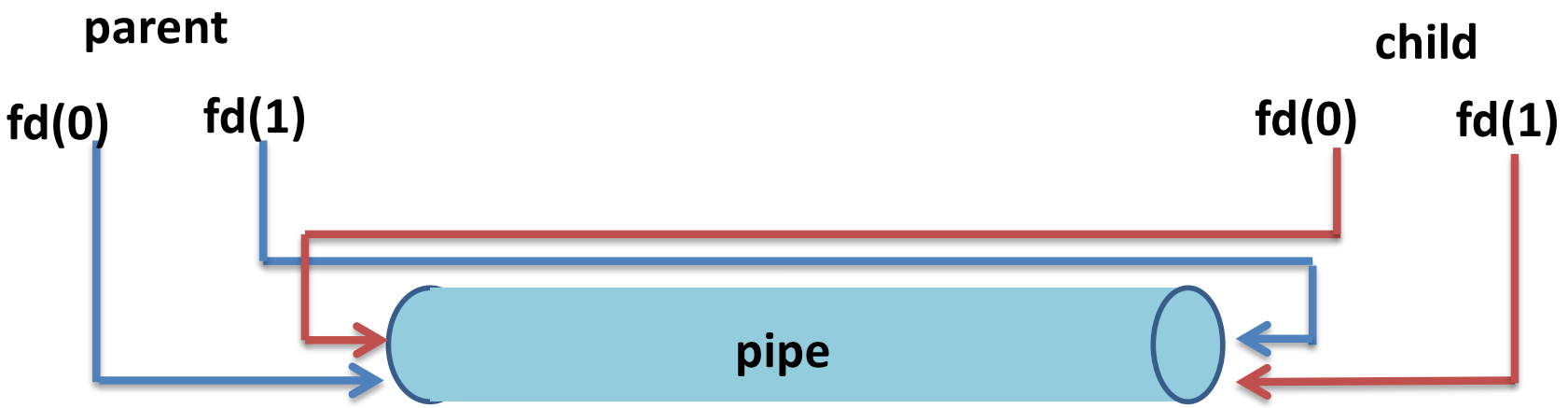
Pipes

- Acts as a conduit allowing two processes to communicate
  - Ordinary Pipes
  - Named Pipes

- **Issues**
  - Is communication unidirectional or bidirectional?
  - Must there exist a relationship (i.e. ***parent-child***) between the communicating processes?
  - Can the pipes be used over a network?

Ordinary Pipes

- Ordinary Pipes allow communication in standard producer-consumer style
- Producer writes to one end (the **write-end** of the pipe)
- Consumer reads from the other end (the **read-end** of the pipe)
- Ordinary pipes are therefore **unidirectional**
- Require parent-child relationship between communicating processes



int fd[] file descriptors: fd[0] is the read-end of the pipe, and fd[1] is the write-end.

Examples of IPC Systems - POSIX

- POSIX Shared Memory
  - Process first creates shared memory segment

    ```
    shm_fd = shm_open(name, O_CREAT | O_RDRW, 0666);
    ```

  - Also used to open an existing segment to share it
  - Set the size of the object

    ```
    ftruncate(shm_fd, SIZE);
    ```

  - Memory-mapped the file

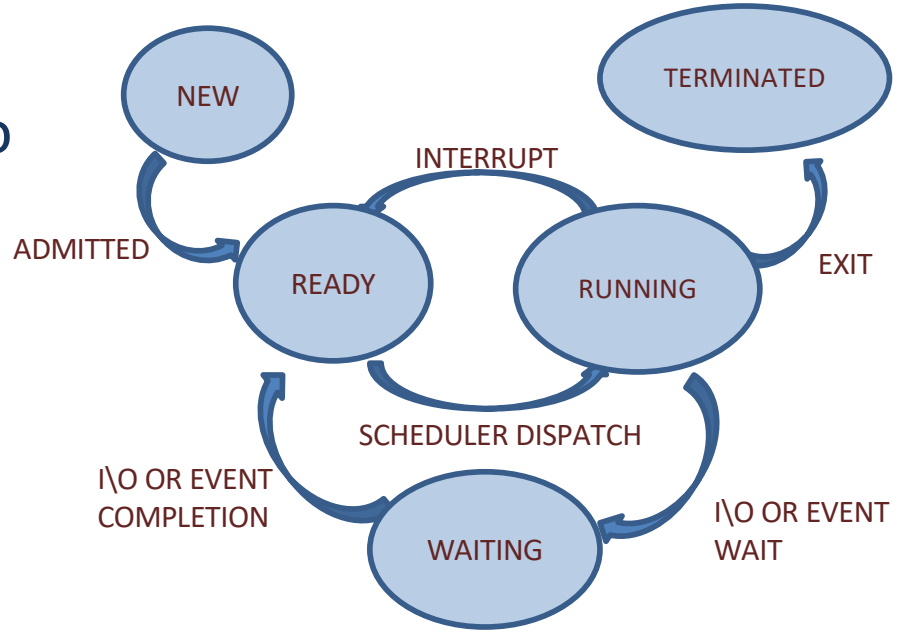
    ```
    ptr = mmap (start, length, PROT_WRITE, MAP_SHARED, shm_fd,  offset);
    ```

  - Now the process could write to the shared memory

    ```
    sprintf(ptr, "Writing to shared memory");
    ```

CPU Scheduler

- Selects among the processes in memory that are ready to execute, and allocates the CPU to one of them.
- CPU scheduling decisions may take place when a process:
  1. Switches from running to waiting state.
  2. Switches from running to ready state.
  3. Switches from waiting to ready.
  4. Terminates.



(Non)-preemptive

- Non-preemptive
  - Process voluntarily releases CPU
- Preemptive
  - OS kicks the process out from the CPU

- 1 and 4 non-preemptive
- 2 and 3 are preemptive

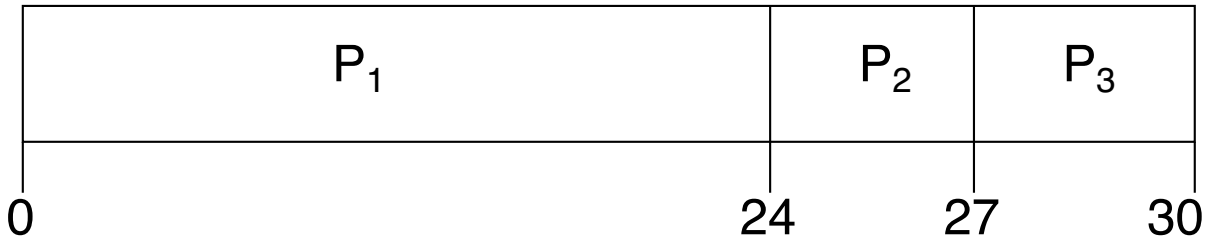
Scheduling Criteria

- CPU utilization – keep the CPU as busy as possible
- Throughput – # of processes that completes their execution per time unit
- Turnaround time – amount of time to execute a particular process (time between entry and exit)
- Waiting time – amount of time a process has been waiting in the ready queue
- Response time – amount of time it takes from when a request was submitted until the first response is produced, **not** output (for time-sharing environment)

1. First-Come, First Served (FCFS)

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
  The Gantt Chart for the schedule is:



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

2. Shortest Job First (SJF) Scheduling

- Associate with each process the length of its next CPU burst. Use these lengths to schedule the process with the shortest time.
- Two schemes:
  - **Non-preemptive** – once CPU given to the process it cannot be preempted until completes its CPU burst.
  - **preemptive** – if a new process arrives with CPU burst length less than remaining time of current executing process, preempt. This is known as the **Shortest-Remaining-Time-First (SRTF)**.
- SJF is **optimal** – gives minimum average waiting time for a given set of processes.

3. 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).
- Can be preemptive and nonpreemptive
- FCFS and SJF are special cases of priority scheduling
  - Why?
  - In FCFS priority is based on the arrival time
  - SJF is a priority scheduling where priority is the predicted next CPU burst time.

Problem with Priority Scheduling

- Problem ≡ **Starvation** – low priority processes may never execute.
- Solution ≡ **Aging** – as time progresses increase the priority of the process.

IBM 7094 operator's console

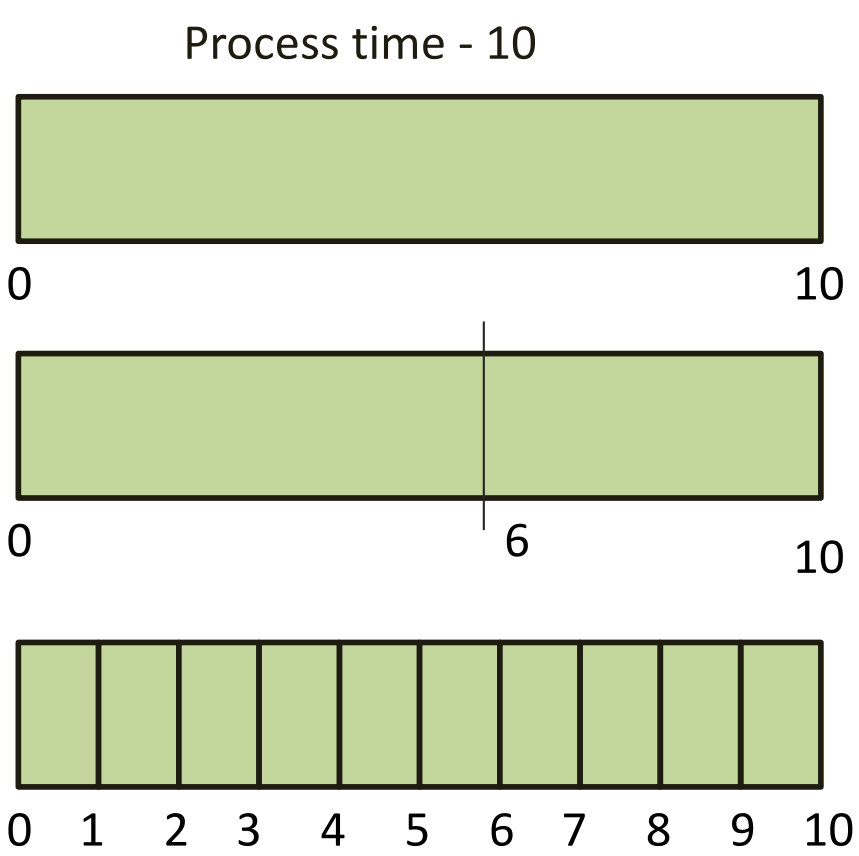
- Rumor has it that,
  - when they shut down the IBM 7094 at MIT in 1973, they found a low-priority process that had been submitted in 1967 and had not yet been run.

4. Round Robin (RR)

- Each process gets a small unit of CPU time (**time quantum**), usually 10-100 milliseconds.
- After this time has elapsed, the process is **preempted** and added to the end of the ready queue.

- If there are n processes in the ready queue and the time quantum is q, then each process gets $1/n$ of the CPU time in chunks of at most q time units at once.
- No process waits more than $(n-1)q$ time units.

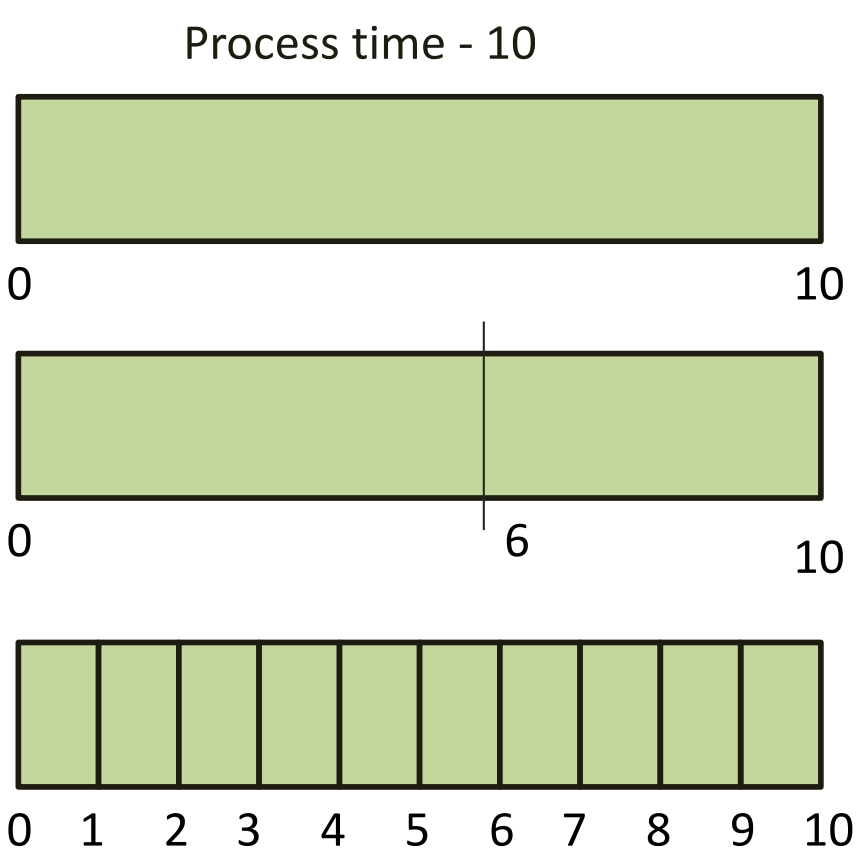
Time Quantum and Context Switch Time



quantum	Context switches
12	0
6	1
1	9

- *quantum* large ⇒ FIFO
- *quantum* small ⇒ *quantum* must be large enough with respect to context switch, otherwise overhead is too high.

Time Quantum and Context Switch Time



quantum	Context switches
12	0
6	1
1	9

CPU Utilization =

CPU doing useful work / (CPU doing useful work + Overhead)

5. Multilevel Queue

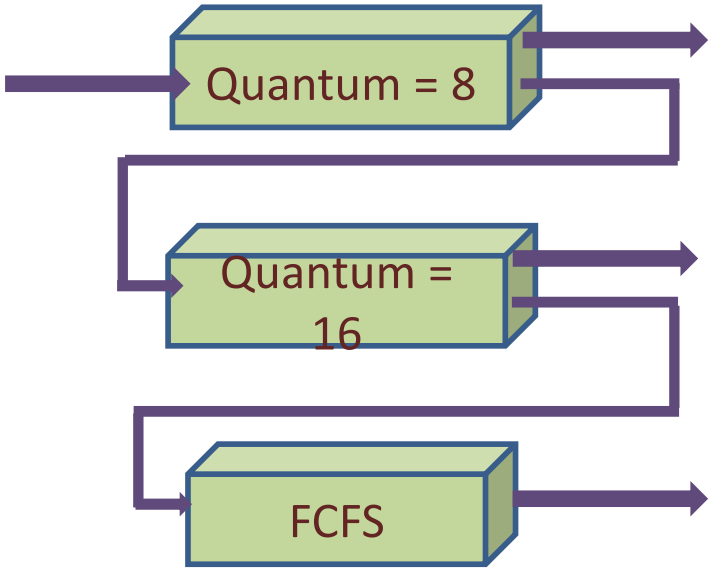
- Ready queue is partitioned into separate queues:
  foreground (interactive)
  background (batch)
- Each queue has its own scheduling algorithm,
  foreground – RR
  background – FCFS
- Scheduling must be done between the queues.
  - Fixed priority scheduling – (i.e., serve all from foreground then from background). Possibility of **starvation**.
  - Time slice – each queue gets a certain amount of CPU time which it can schedule amongst its processes; i.e., 80% to foreground in RR, 20% to background in FCFS

6. Multilevel Feedback Queue

- Multilevel queue with feedback scheduling is similar to multilevel queue; however, it allows processes to move between queues.
- **Aging** can be implemented this way.
- Multilevel-feedback-queue scheduler defined by the following parameters:
  - number of queues
  - scheduling algorithms for each queue
  - method used to determine when to upgrade a process
  - method used to determine when to demote a process
  - method used to determine which queue a process will enter when that process needs service

Example of Multilevel Feedback Queue

- Three queues:
  - Q_0 – time quantum 8 milliseconds
  - Q_1 – time quantum 16 milliseconds
  - Q_2 – FCFS non-preemption
- Example Scheduling
  - A new job enters queue Q_0. 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 job is again served FCFS and receives 16 additional milliseconds.
  - If it still does not complete, it is preempted and moved to queue Q_2.



Multiple-Processor Scheduling

- CPU scheduling is more complex when multiple CPUs are available
  - **Asymmetric multiprocessing** – only one processor accesses the system data structures, alleviating the need for data sharing
  - **Symmetric multiprocessing (SMP)** – each processor is self-scheduling, all processes in common ready queue, or each has its own private queue of ready processes
- **Processor affinity** – process has affinity for processor on which it is currently running
  - **soft affinity** (can be changed at a later time)
  - **hard affinity** (process doesn't move to another processor)
  - Variations including **processor sets**

Real-Time CPU Scheduler

- **Real-time programs** must guarantee response within strict time constraints, often referred to as deadlines
- **Soft real-time systems** – no guarantee as to when critical real-time process will be scheduled, degrades the system's quality of service
  - Ex: the flight plan updates for an airline, live broadcasting
- **Hard real-time systems** – missing a deadline is a total system failure
  - Mission critical: a real-time deadline must be met, regardless of system load
  - Ex: Anti-lock brakes on a car, heart pacemakers and many medical devices
- Not all the Operating Systems are real-time operating systems

Basic Philosophies in Linux

- Priority is the primary scheduling mechanism
- Priority is *dynamically adjusted* at run time

- Try to distinguish interactive processes from non-interactive ones
- Use large time quanta for important processes
  - Modify quanta based on CPU usage for the next run
- Associate processes to CPUs in a multicore systems
  - Process affinity

Completely Fair Scheduler (CFS)

- Core ideas: dynamic time slice and order
- Don’t use fixed time slice per task
  - Instead, fixed time slice across all tasks
  - Scheduling Latency
- Don’t use round robin to pick next task
  - Pick task which has received the least CPU time so far
  - Equivalent to dynamic priority

Race Condition

- **Race condition**: The situation where several processes access and manipulate shared data concurrently. The final value of the shared data is **non-deterministic** and depends upon which process finishes last.

- To prevent race conditions, concurrent processes must be **synchronized**.

The Critical Section Problem

- Consider system of ***n*** processes {$p_0, p_1, \ldots p_{n-1}$} and all competing to use some shared data
  - Updating a table, writing into a file etc.

- Each process has a code segment, called **critical section**, in which the shared data is accessed.

- Problem – ensure that when one process is executing in its critical section, no other process is allowed to execute in its critical section.

Critical Section

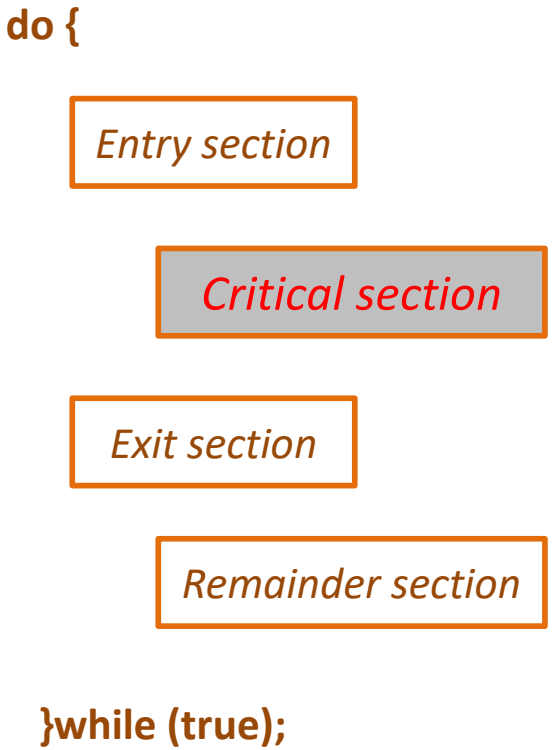
A Critical Section Environment contains:

Entry Section Code requesting entry into the critical section.

Critical Section Code in which only one process can execute at any one time.

Exit Section The end of the critical section, releasing or allowing others in.

Remainder Section Rest of the code AFTER the critical section.



Solution to Critical Section Problem

Any solution to Critical Section problem must provide all of the followings

1. **Mutual Exclusion**. If process P_i is executing in its critical section, then no other processes can be executing in their critical sections.

2. **Progress**. If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely.

3. **Bounded Waiting**. A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

Attempt 3: Peterson's Solution

Combined shared variables of algorithms 1 and 2.

```
bool flag[0] = false;
bool flag[1] = false;
int turn;  //shared variable
```

```
P0: flag[0] = true;
P0: turn = 1;

while (flag[1] && turn == 1) {
// busy wait }

// critical section

flag[0] = false;

//remainder section
```

```
P1: flag[1] = true;
P1: turn = 0;

while (flag[0] && turn == 0) {
// busy wait }

// critical section

flag[1] = false;

//remainder section
```

Meets all three requirements; solves the critical-section problem for two processes.

Peterson's Solution

- Peterson solution is **not correct** on today's modern computers
  - Because update to turn is not specified as **atomic**
  - We have **caches and multiple copies** of the same data (turn variable) in the hardware
- Process 0 may see turn as 1
- Process 1 may see turn as 0
- Resulting in data race

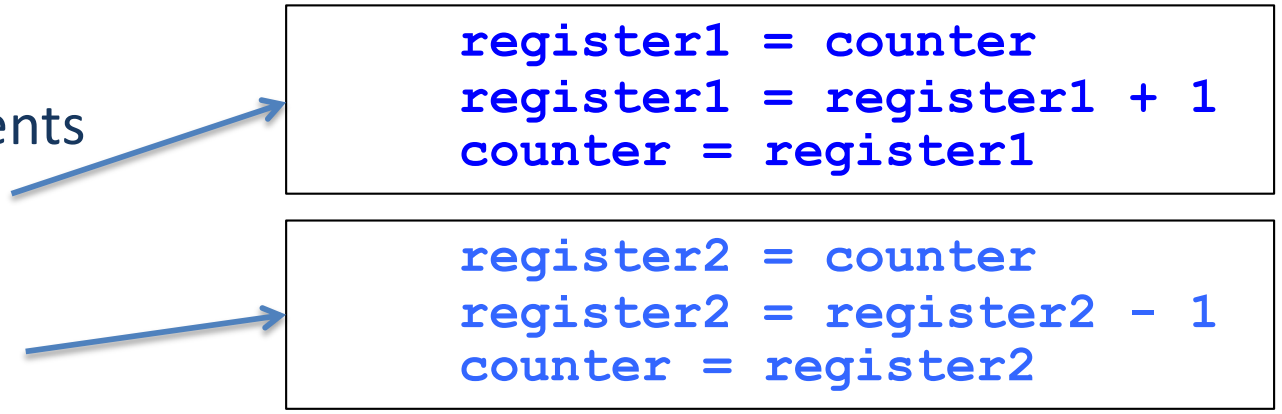
- We need hardware support for avoiding race conditions.

Atomic Instructions

- Peterson's solution is a software-based solution
- Modern machines provide special atomic hardware instructions
  - **Atomic** = indivisible instructions
- The statements

  counter++;

  counter--;



- must be performed **atomically**.
- Atomic operation means an operation that completes in its entirety **without interruption (preemption)**.

Atomic Test-and-Set Instruction

- The term *locks* are used to indicate getting a key to enter a critical section.
- The **test-and-set** instruction is an instruction used to write to a memory location and return its old value as a single atomic (i.e., non-interruptible) operation.
- If multiple processes may access the same memory location, and if a process is currently performing a test-and-set, no other process may begin another test-and-set until the first process is done.

```
boolean TestAndSet(boolean *lock) {
  boolean initial = *lock;
  *lock = true;
  return initial;
}
```

compare_and_swap instruction

- It compares the contents of a memory location to a given value and, **only if they are the same**, modifies the contents of that memory location to a given new value.
- Done as a **single atomic operation**
- if the value had been updated by another process in the meantime, the write would fail.

```
int compare_and_swap(int *value,
                     int expected, int new_value) {
   int oldValue = *value;
   if (*value == expected)
      *value = new_value;
   return oldValue;
}
```

Mutex Locks

- Previous hardware solutions are complicated and generally inaccessible to application programmers
- OS designers build software tools to solve critical section problem using the support in the hardware
- Enter critical regions by first **acquire()** a lock then **release()** it
  - Boolean variable indicates if lock is available or not
  - Next lecture, we will cover those
- Note that these solutions still use hardware solutions/support underneath

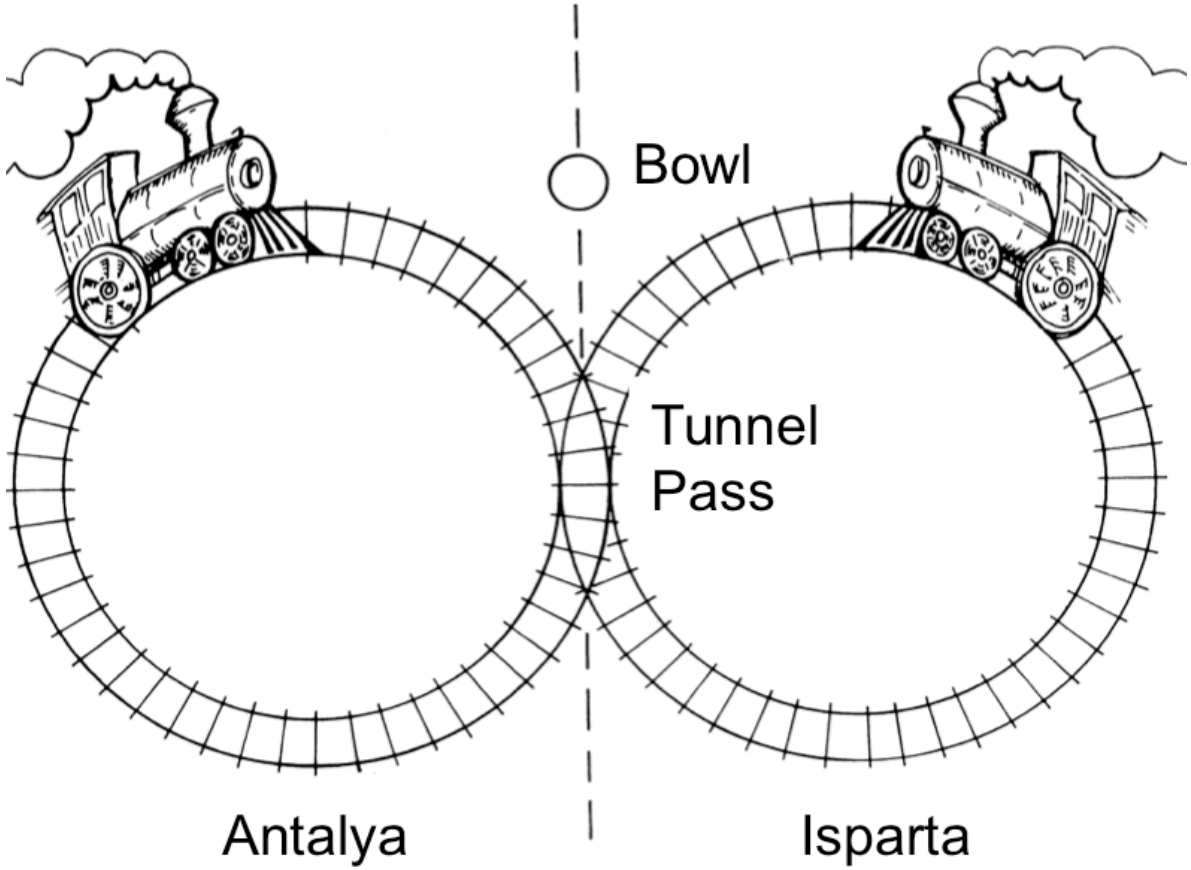
acquire() and release()

```
do {
    acquire lock
        critical section
    release lock
        remainder section
} while (true);
```

```
acquire() {
    while (!available)
        ; /* busy wait */
    available = false;;
}

release() {
    available = true;
}
```

- Calls to **acquire()** and **release()** must be atomic
  - Usually implemented via *hardware atomic instructions* discussed few slides back.
- This solution requires **busy waiting**
  - This type of lock is called a **spinlock**

Question



- High in the Toros mountains, there are two circular railway lines. One line is in Antalya, the other in Isparta.
- The Antalya train runs twenty times a day, the Isparta train runs ten times a day.
- They share a common section of track where the lines cross a tunnel pass that lies on the city border.
- Unfortunately, the two trains occasionally collide when simultaneously entering the common section of track (the tunnel pass).
- The problem is that the drivers of the two trains are both blind and deaf, so they can neither see nor hear each other.
- Three OS students suggested the following methods to prevent accidents.
- For each method, explain if the method **prevents collision, whether it is starvation-free and respects the train schedule.**

Semaphores

- We want to be able to write more complex constructs
  - need a language to do so. We define semaphores which we assume are atomic operations.
- Semaphores are more general synchronization tools
  - Operating System Primitive
  - Two standard atomic operations modify semaphore variable S: wait() and signal()

```
WAIT ( S ):

    while   ( S  <=  0 );
    S  =  S  -  1;
```

```
SIGNAL ( S ):

    S  =  S  +  1;
```

- As given here, these are not atomic as written in "macro code". We define these operations, however, to be atomic (Protected by a hardware lock.)

Semaphore as a general synchronization tool

- Provides mutual exclusion

```
Semaphore S =  1;    //  initialized to 1 or initialized to # of resources
wait (S);
    Critical Section
signal (S);
```

- Counting semaphore – integer value can range over an unrestricted domain
  - For example: resources in the hardware: semaphore is initialized to number of printers
- Binary semaphore – integer value can range only between 0 and 1; can be simpler to implement
  - This is the same as mutex locks

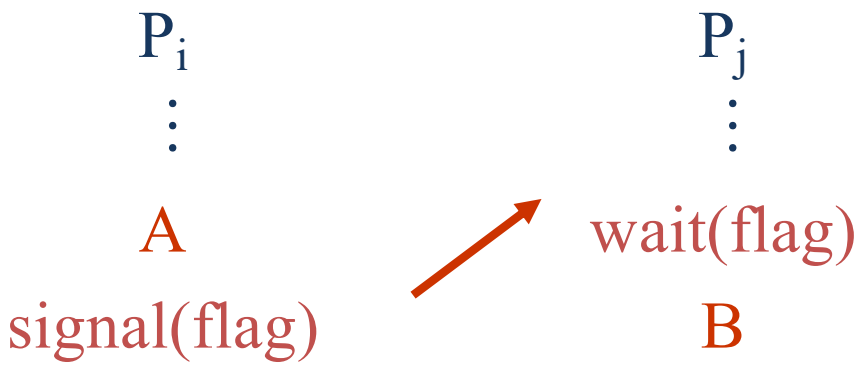
Semaphore as a general synchronization tool

- Semaphores can be used to force synchronization (precedence) if the **preceeder** does a signal at the end, and the **follower** does wait at beginning.

For example, here we want P1 to execute before P2.

- Execute B in P_j **only after** A is executed in P_i
- Use semaphore flag initialized to **0**

Code:



No busy waiting (blocking) Semaphores

- With each semaphore there is an associated waiting queue:
  - Keeps list of processes waiting on the semaphore

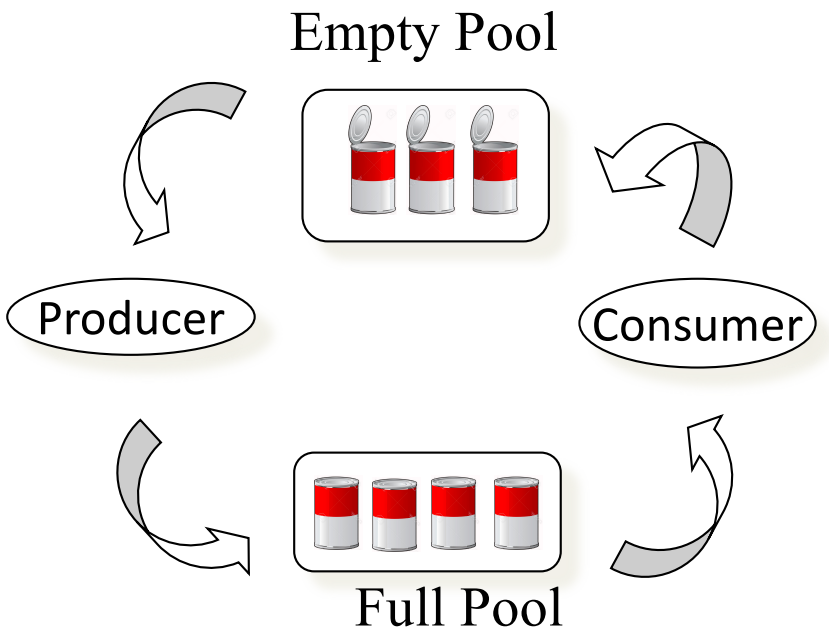
- Two operations:
  - Block – place the process invoking the operation on the appropriate waiting queue if semaphore == false
  - Wakeup – Wakes up one of the blocked processes upon getting a signal and places the process to ready queue

Bounded-Buffer Problem

- Buffer size: buffer can hold **n** items
- Binary semaphores
  `semaphore mutex;`
- Counting semaphores
  `semaphore full, empty;`

Initially:

`full = 0, empty = n, mutex = 1`



This is the same producer / consumer problem as before. But now we'll do it with signals and waits. Remember: a **wait decreases** its argument and a **signal increases** its argument.

Bounded-Buffer Problem

- **Mutex** is binary semaphore, Empty and Full are counting semaphores

```
producer:
do {
      /* produce an item in nextp */
      wait (empty);            /* Do action     */


       /* add nextp to buffer  */


      signal (full);

  } while(TRUE);
```

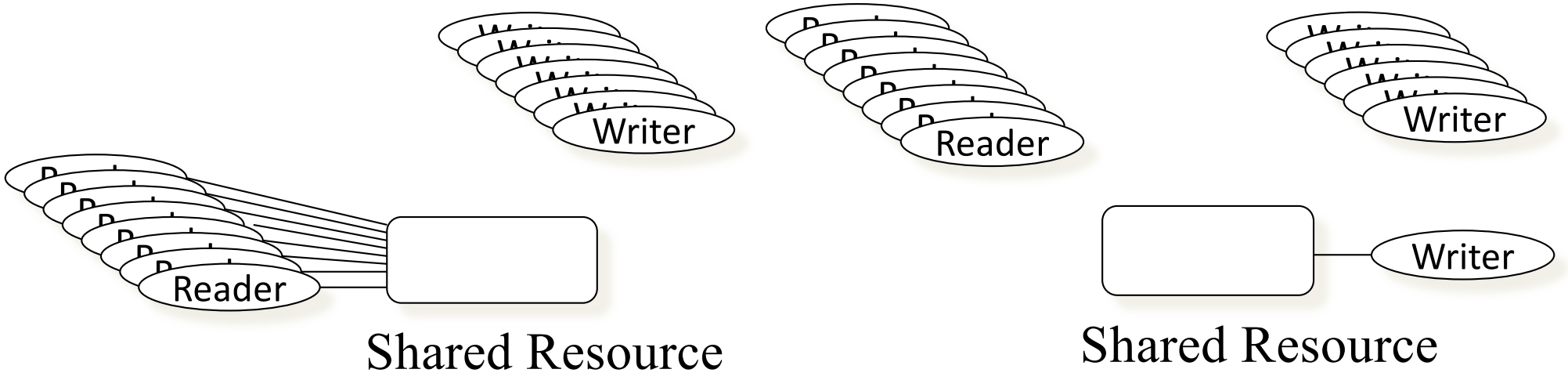
```
consumer:
do {
      wait (full);


       /* remove an item from buffer to nextc */


      signal (empty);

      /* consume an item in nextc */
   } while(TRUE);
```

Does this work for multiple producers and consumers?

Only works for one producer and consumer. Need the mutex to prevent multiple producers writing into the same buffer.

Readers and Writers Problem

- A data set is shared among a number of concurrent processes
  - Readers – only read the data set; they do not perform any updates
  - Writers – can both read and write.
- Problem
  - Allow multiple readers to read at the same time.
  - Only one single writer can access the shared data at a time.



First-Readers Problem

Favoring readers over writers

```
semaphore rdwrt = 1;
semaphore cntmutex = 1;
int readcount = 0;
```

Writer:

```
do {
      wait( rdwrt );
        /*   writing is performed    */
      signal(  rdwrt  );

} while(TRUE);
```

Reader:

```
do {
       wait( cntmutex );                /* Allow 1 reader in entry*/
       readcount = readcount + 1;
       if readcount == 1 then
          wait(rdwrt );               /* 1st reader locks rdwrt */
       signal( cntmutex );

          /*   reading is performed  */

       wait( cntmutex );
       readcount = readcount - 1;
       signal( cntmutex );
       if readcount == 0  then
          signal(rdwrt );           /*last reader frees writer */

} while(TRUE);
```

Problems with Semaphores (and Locks)

- Semaphores are shared global variables
  - Can be accessed from anywhere
- Used for both critical sections (mutual exclusion) and for coordination (scheduling or ordering execution)
- Incorrect use of semaphore operations
  - Call signal first and later on call wait
    - signal(mutex) …. wait(mutex)
  - Call wait after another wait
    - wait(mutex) …. wait(mutex)
  - Omitting of wait or signal
- Thus, they are prone to bugs
- To deal with such issues,
  - Introduce a high-level synchronization construct - **monitors**

Monitors

- A monitor is a **programming language construct** that supports controlled access to shared data
  - First developed in Concurrent Pascal
  - It resembles an object-oriented approach for synchronization
- *A monitor encapsulates*
  - shared data structures
  - procedures that operate on the shared data (protects shared data)
  - synchronization between concurrent processes that invoke those procedures

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

    procedure Pn (…) {……}

    initialization(…) { … }

}
```

Monitor construct ensures that only one process at a time can be active within the monitor

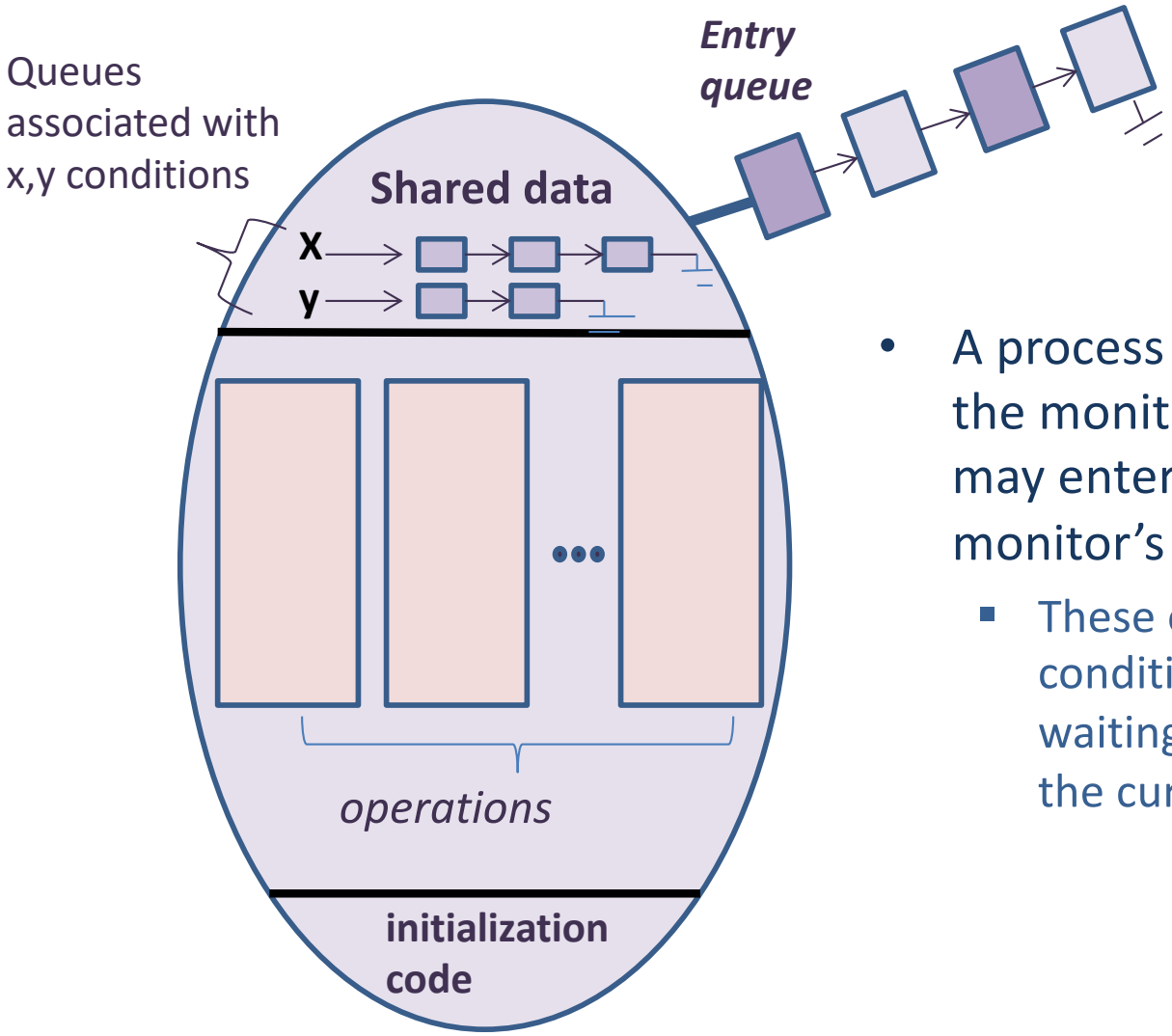
Conditional Variables

- Conceptually a condition variable is a queue of processes, associated with a monitor on which a process may wait for some condition to become true

- Sometimes called a rendezvous point
  - To allow a process to wait within the monitor, a **condition** variable must be declared, as
    condition c;

- Three operations on condition variables 'c'
  - wait(c)
    - The calling process is suspended until another process invokes it
  - signal(c)
    - wake up at most one waiting process
    - if no waiting processes, signal has no effect
      - this is different than semaphores: no history!
  - broadcast(c)
    - wake up all waiting processes

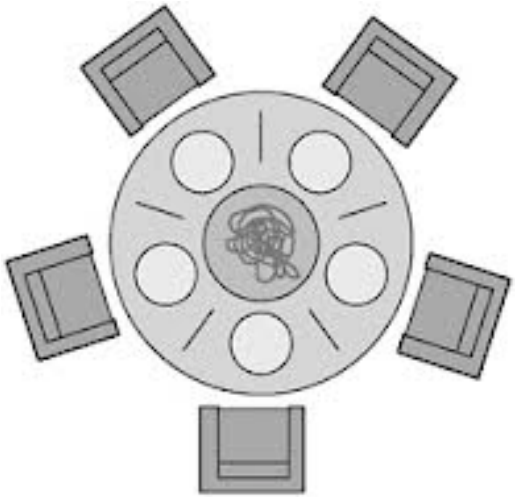
Monitor with Condition Variables



- A process is not considered to occupy the monitor, and so other processes may enter the monitor to change the monitor's state
  - These other processes may signal the condition variable to indicate that waiting condition has become true in the current state

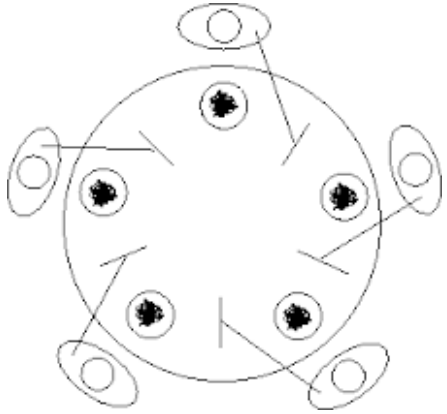
Dining Philosophers Problem

- 5 philosophers with 5 chopsticks sit around a circular table.
  - They each want to eat at random times
  - Must pick up the 2 chopsticks
  - Pick one chopstick at a time
- While a philosopher is thinking, she drops the chopsticks



Dining Philosophers Problem

- Shared data
  semaphore chopstick[5]; Initially all values are 1



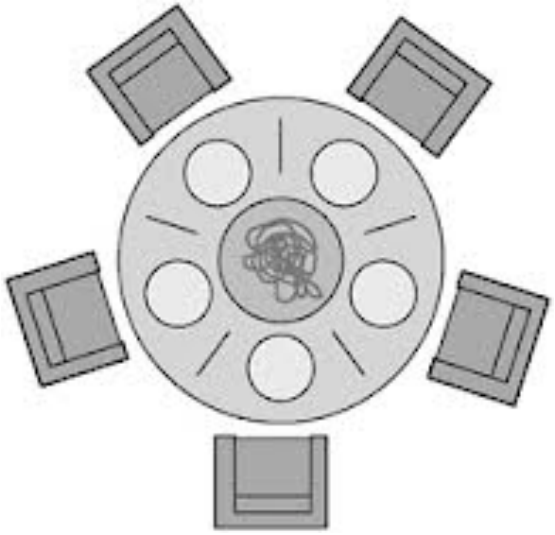
Deadlock may occur!

Structure of Philosopher *i*:

```
while (true) {
  wait(chopstick[i])
  wait(chopstick[(i+1) % 5])

  // eat

  signal(chopstick[i]);
  signal(chopstick[(i+1) % 5]);

  // think
}
```

Several Solutions

- Allow pickup only if both chopsticks are available
- Odd # philosophers always picks up left chopstick first, even # philosophers always picks up right chopstick first



A deadlock-free solution does not necessarily eliminate the possibility of starvation.

Monitor Solution to Dining Philosophers

```
monitor DP
 {
   enum { THINKING, HUNGRY, EATING) state [5] ;
   condition self [5];

   void pickup (int i) {
          state[i] = HUNGRY;
          test(i);
          if (state[i] != EATING)
              self[i].wait;
   }

   void putdown (int i) {
          state[i] = THINKING;
           // test left and right neighbors
          test((i + 4) % 5);
          test((i + 1) % 5);
  }
```

- This implements a deadlock-free solutions with a restriction that a philosopher may pick up chopsticks only if both of them are available
- Is this solution starvation free?

Monitor Solution to Dining Philosophers (cont.)

```
void test (int i) {
    if ( (state[(i + 4) % 5] != EATING) &&
         (state[i] == HUNGRY) &&
         (state[(i + 1) % 5] != EATING) ) {

         state[i] = EATING ;
          self[i].signal () ;
     }
}

initialization_code() {
   for (int i = 0; i < 5; i++)
       state[i] = THINKING;
   }
}
```

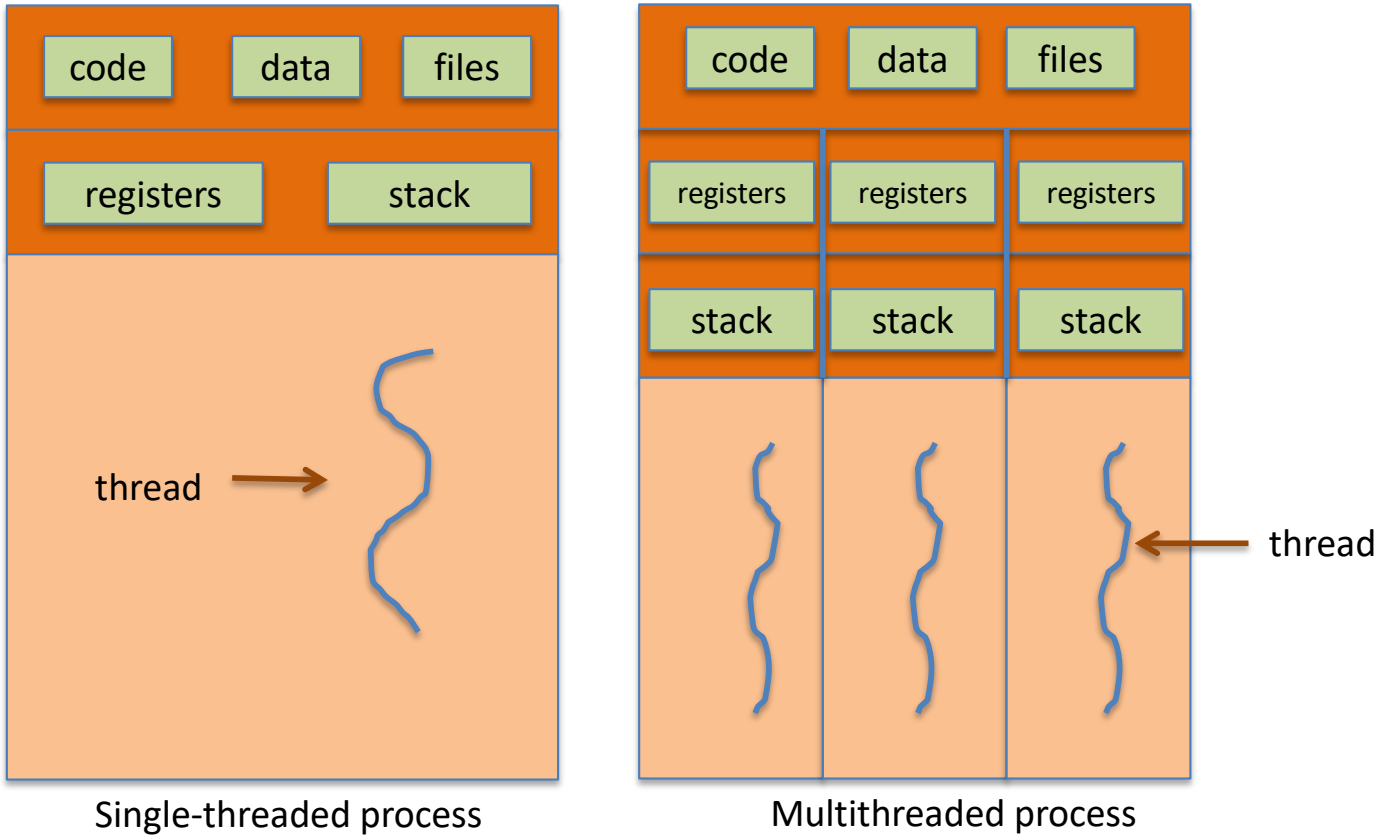
- If my neighbors are not in eating state and I am hungry, then I can eat!
- All philosophers' states are set to Thinking initially

Solution to DP Problem (cont.)

- Each philosopher *i* invokes the operations pickup() and putdown() in the following sequence:

```
dp.pickup (i)

   EAT

dp.putdown (i)

   THINK
```

Single vs Multithreaded Process



- A thread has an ID, a program counter, a register set, and a stack
- Shares the code section, data section and OS resources (e.g. files) with other threads within the same process

User Threads and Kernel Threads

- **User threads** - management done by user-level thread library
- Three primary thread libraries:
  - POSIX **Pthreads**
  - Win32 threads
  - Java threads

- **Kernel threads** - Supported by the Kernel
- Examples – virtually all general-purpose operating systems, including:
  - Windows
  - Solaris
  - Linux
  - Tru64 UNIX
  - Mac OS X

Multi-Threading Models

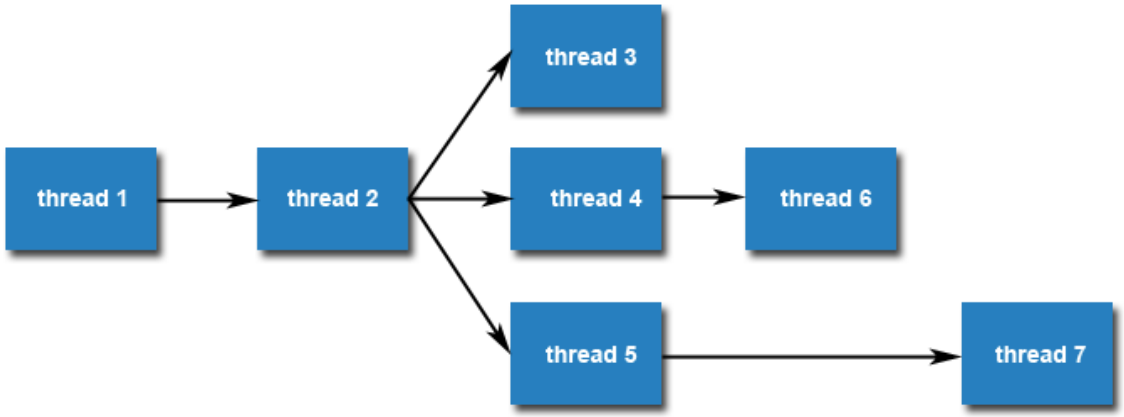
- Many systems support both user and kernel level threads, resulting in different multi-threading models
  - Many-to-one
  - One-to-one
  - Many-to-many

POSIX Threads API

- Functions of pthreads API:
  - Thread management:
    - Creation/termination of threads
    - Set/Query thread attributes
  - Mutexes, semaphores
  - Condition variables
- All identifiers in the threads library begin with pthread_

Creating Threads

- Initially, a main() program comprises a single, default thread. All other threads must be explicitly created by the programmer.
- pthread_create
  - creates a new thread and makes it executable.
- The maximum number of threads that may be created by a process is implementation dependent.
  - Programs that attempt to exceed the limit can fail or produce wrong results.
- Threads can create other threads (but there is no hierarchy)



Pthread_create()

- Forking Pthreads

Signature:

```
int pthread_create(pthread_t *,
                   const pthread_attr_t *,
                   void * (*)(void *),
                   void *);
```

Example call:

```
errcode = pthread_create(&thread_id, &thread_attribute,
                         &thread_func, &func_arg);
```

- thread_id is the thread id or handle (used to halt, etc.)
- thread_attribute various attributes
  - standard default values obtained by passing a NULL pointer
- thread_func the function to be run (takes and returns void*)
- func_arg an argument can be passed to thread_fun when it starts
- errorcode will be set to nonzero if the create operation fails

Thread Synchronization

- Need to protect the shared data and synchronize threads
- Pthread provides several ways to synchronize threads:
- Mutexes (Locks)
- Semaphores
- Condition Variables
- Barriers
  - Synchronizing the threads to make sure that they all are at the same point in a program is called a barrier.

Deadlock Characterization

Deadlock can arise if four conditions hold simultaneously.

Mutual exclusion: only one process at a time can use a resource.

Hold and wait: a process holding at least one resource is waiting to acquire additional resources held by other processes.

No preemption: a resource can be released only voluntarily by the process holding it, after that process has completed its task.

Circular wait: there exists a set $\{P_0, P_1, \ldots, P_n\}$ of waiting processes such that P_0 is waiting for a resource that is held by P_1, P_1 is waiting for a resource that is held by P_2, …, P_n is waiting for a resource that is held by P_0.

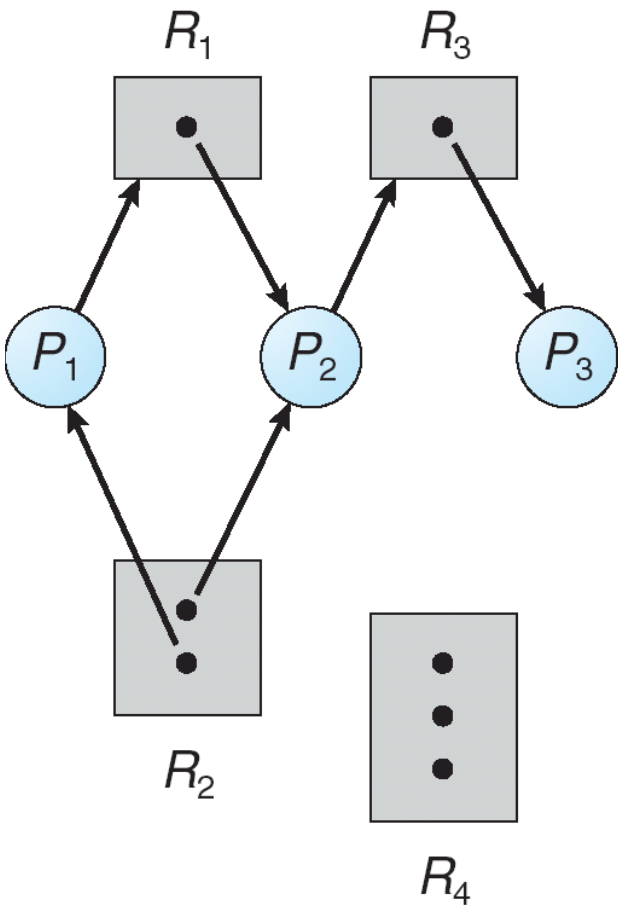
Resource-Allocation Graph

Deadlocks can be described in terms of a directed graph

A set of vertices V and a set of edges E.

- V is partitioned into two types:
  - $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system.
  - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system.
- **request edge:** directed edge $P_i \rightarrow R_j$
- **assignment edge:** directed edge $R_j \rightarrow P_i$

Example Resource Allocation Graph



Resource instances:

- One instance of resource type *R1*
- Two instances of resource type *R2*
- One instance of resource type *R3*
- Three instances of resource type *R4*

Process states:

- Process *P1* is holding an instance of resource type *R2* and is waiting for an instance of resource type *R1*.
- Process *P2* is holding an instance of *R1* and an instance of *R2* and is waiting for an instance of *R3*.
- Process *P3* is holding an instance of *R3*.

Basic Facts

- If graph contains no cycles $\Rightarrow$ no deadlock.
- If graph contains a cycle $\Rightarrow$
  - if only one instance per resource type, then deadlock.
  - if several instances per resource type, possibility of deadlock.

Deadlock Avoidance Algorithms

- Single instance of a resource type: Use a **resource-allocation graph**

- Multiple instances of a resource type: Use the **banker's algorithm**

Deadlock Avoidance

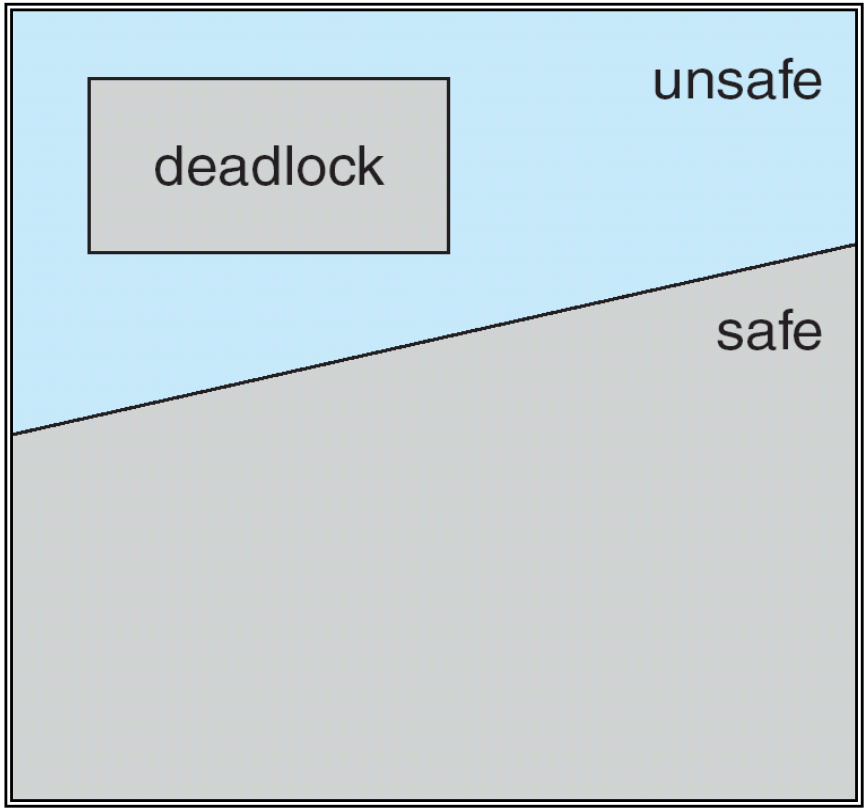
- The **system knows** the complete sequence of requests and releases for each process.
  - Priori information is available

- The **system decides** for each request whether or not the process should wait in order to avoid a deadlock.

- Each **process declares** the maximum number of resources of each type that it may need.

- The system should always be at a **safe state**.

Safe, Unsafe , Deadlock State

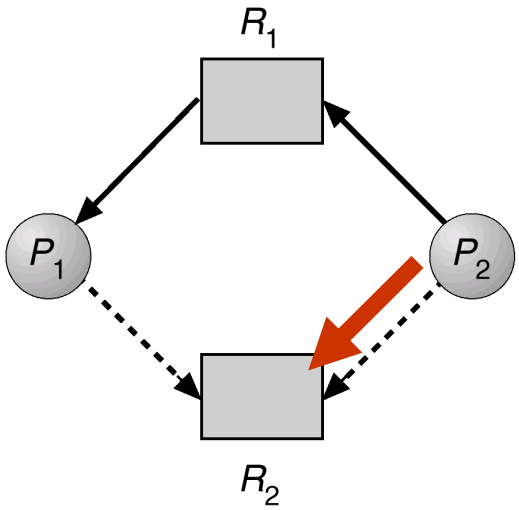


We can define avoidance algorithms that ensure the system will never deadlock.

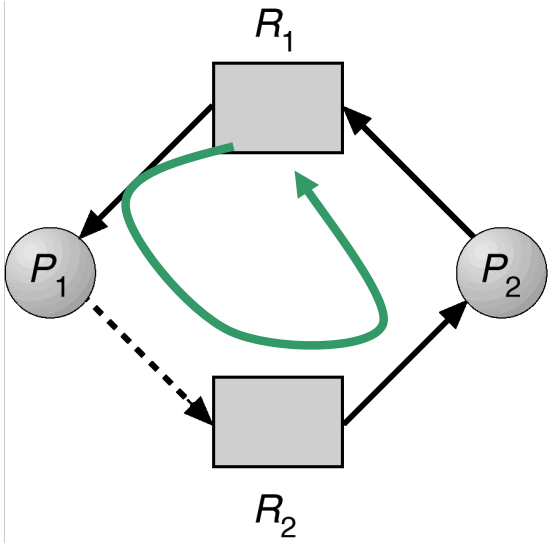
A resource is granted only if the allocation leaves the system in a **safe state.**

Resource-Allocation Graph Algorithm

- Works only if each resource type has one instance
- Algorithm
  - **Add a claim edge** $P_i \rightarrow R_j$ indicating that process P_i may request resource R_j;
  - Represented by a dashed line.



- A request $P_i \rightarrow R_j$ can be granted only if
  - Adding assignment edge $R_j \rightarrow P_i$ does not result in a cycle in the graph



A cycle indicates an unsafe state.

Banker's Algorithm

```
n: integer    # of processes
m: integer    # of resource-types


Available[1:m]
 #Available[j] is # of avail resources of type j
Max[1:n,1:m]
 #Max demand of each Pi for each Rj
Allocation[1:n,1:m]
 #current allocation of resource Rj to Pi
Need[1:n,1:m]
 #max # resource Rj that Pi may still request
 #Need[i,j] = Max[i,j] - Allocation[i,j]
```

Banker's Algorithm

$Request_i$ = request vector for process P_i.
If $Request_i[j] = k$ then process P_i wants k instances of resource type R_j.

If $request_i > need_i$
    error (asked for too much)
If $request_i > available$
    wait(can't supply it now)
Resources are available to satisfy the request
    Let's assume that we satisfy the request, then
        $available = available - request_i$
        $allocation_i = allocation_i + request_i$
        $need_i = need_i - request_i$

Now check if this would leave us in a safe state:
    If yes, grant the request
    If no, leave the state as is and cause process to wait

Banker's Safety Algorithm

Algorithm for finding out whether or not a system is in a safe state

```
Initialize:
  Work[1:m] = Available[1:m] //how many resources available
  Finish[1:n] = false        //none of the processes finished yet

Step 1:Find a process i such that both:
    (a) Finish[i] = false
    (b) Need_i ≤ Work
    If no such i exists, go to step 3.

Step 2: Found an i
    Finish[i] = true       //done with this process
    Work = Work + Allocation_i
    go to step 1

Step 3:
    If Finish[i] == true for all i, then
       the system is in a safe state.
    Else
       Not safe
```

Requires $O(mn^2)$ operations to decide whether a state is safe

Deadlock Detection

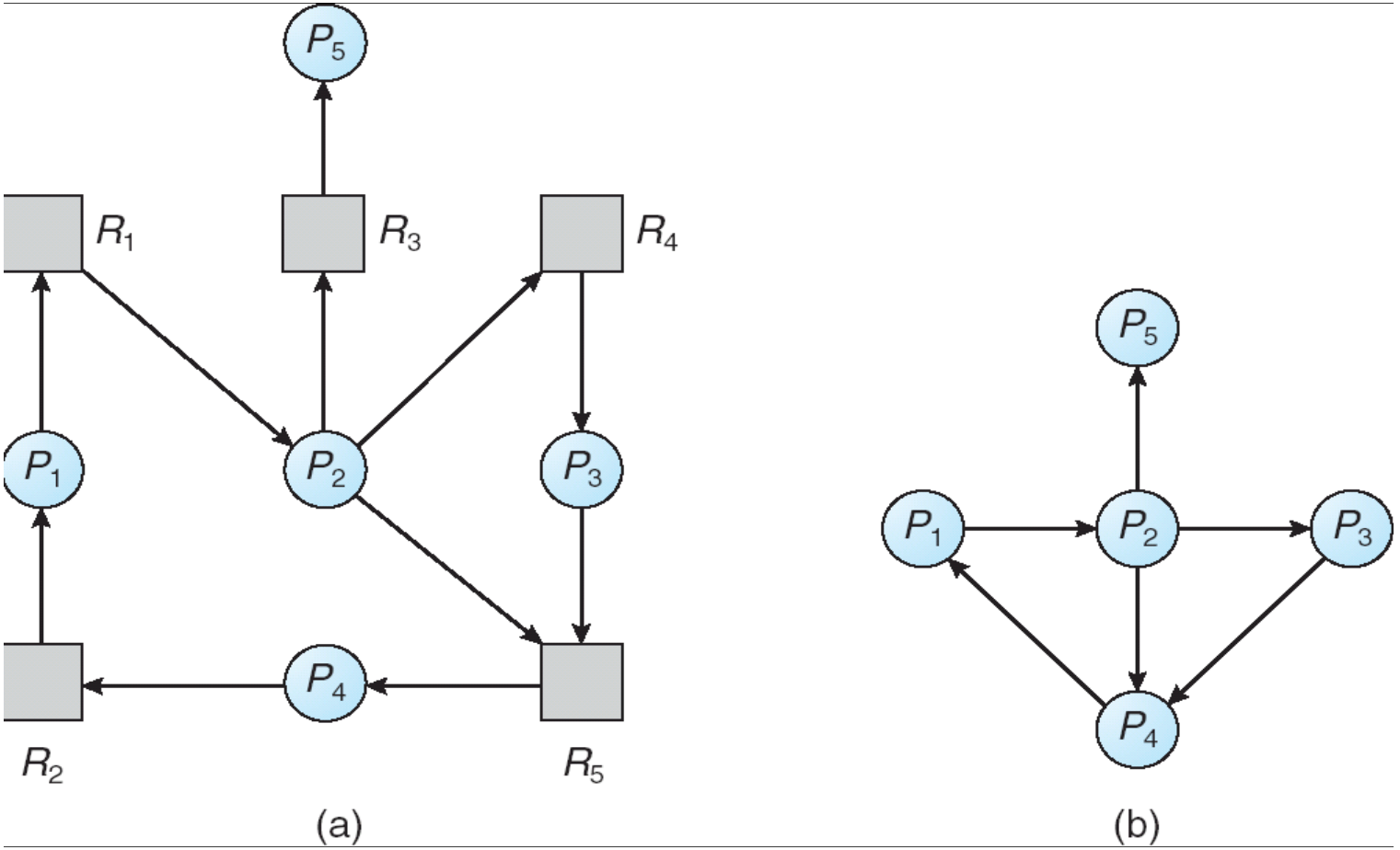
- We saw that you can **prevent** deadlocks
  - By negating one of the four necessary conditions.
- We saw that the OS can schedule processes in a careful way so as to **avoid** deadlocks.
  - Using a resource allocation graph.
  - Banker's algorithm.

- Deadlock Detection
  - Allow system to enter deadlock state
  - Detection algorithm
  - Recovery scheme

Single instance of each resource type

- Maintain **wait-for** graph
  - Nodes are processes.
  - $P_i \rightarrow P_j$ if P_i is waiting for P_j (to release a resource that P_i needs)
- Periodically invoke an algorithm that searches for a cycle in the graph.
- An algorithm to detect a cycle in a graph requires an order of n^2 operations, where n is the number of vertices in the graph.

Resource-Allocation and Wait-for Graphs



Resource-Allocation Graph

Corresponding wait-for graph

Multiple instances of a resource type

Let n = number of processes, and m = number of resources types.

```
n: integer    # of processes
m: integer    # of resource-types

Available[1:m]
 #Available[i] is # of avail resources of type i
Request[1:n,1:m]
 #Current demand of each Pi for each Rj
Allocation[1:n,1:m]
 #current allocation of resource Rj to Pi
finish[1:n]
 #true if Pi's request can be satisfied
```

Detection Algorithm

Let Work and Finish be vectors of length m and n, respectively

1. **Initialize:**
   (a) Work = Available
   (b) For i=1:n,
       if Allocation[i] ≠ 0, then
         Finish[i] = false
       otherwise, Finish[i] = true.
2. **Find an index i such that both:**
   (a) Finish[i] == false
   (b) Request[i] ≤ Work
   If no such i exists, go to step 4.
3. **Work = Work + Allocation[i]**
   Finish[i] = true
   go to step 2.

4. **If Finish[i] == false**, for some i, then
   the system is in deadlock state with P_i is deadlocked.

Requires an order of $O(mn^{2})$ operations to detect whether the system is in deadlocked state.

- GOOD LUCK