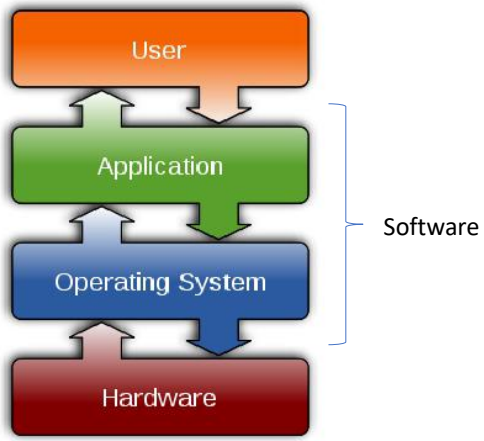
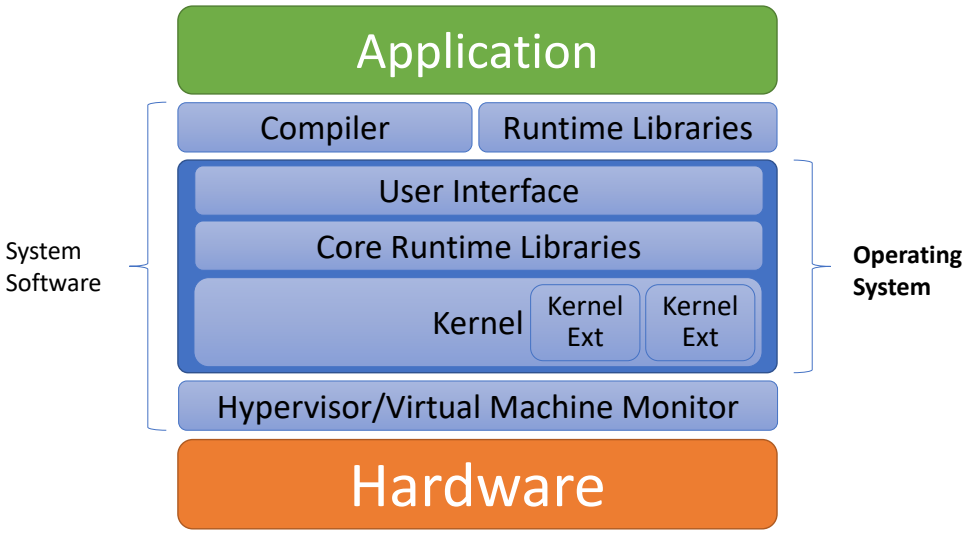
Midterm Review

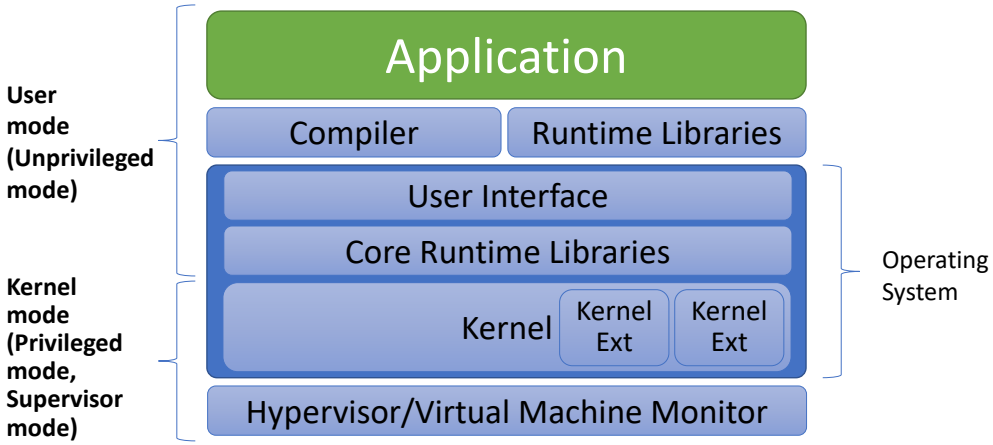
Where the OS Fits In … (1)



What is OS? (1)



What is OS? (2)



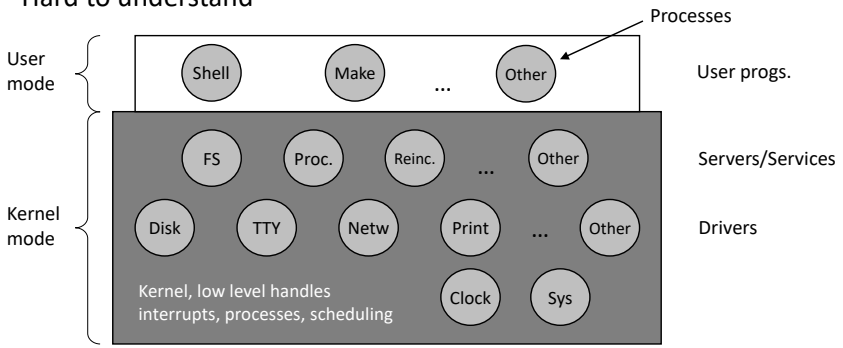
NOTE there exist OSes that do not use modes, there is hardware that doesn't support modes

Operating Systems Structure

- What an OS looks like from the inside
  - Monolithic
  - Microkernel
  - …
  - Hybrids

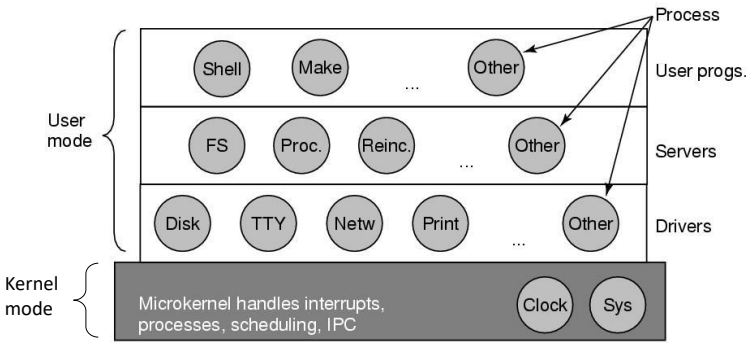
Monolithic OS

- Entire OS runs as a single program in kernel mode
- Collection of procedures linked together into a single large executable binary
  - Every function can call the other
  - Very performant
  - Crash affects everything
  - Hard to understand

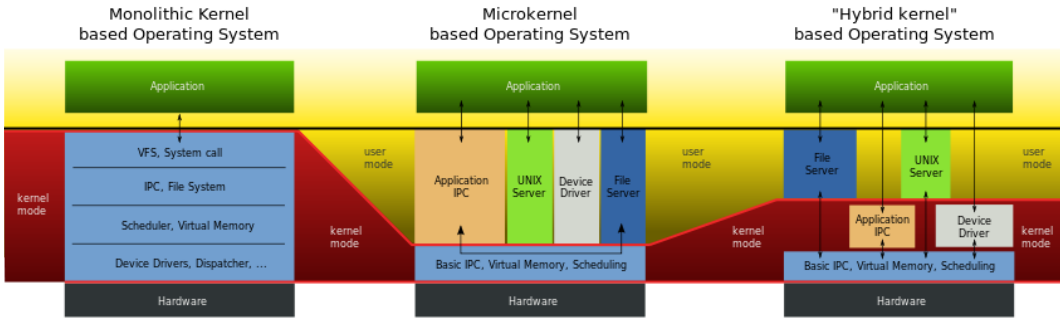


Microkernels OS

- Split the OS up into small, well-defined modules
  - only one of which runs in kernel mode (microkernel)
  - the rest run as relatively powerless ordinary user processes (a bug cannot crash the entire system)



Hybrid of Monolithic and Microkernel

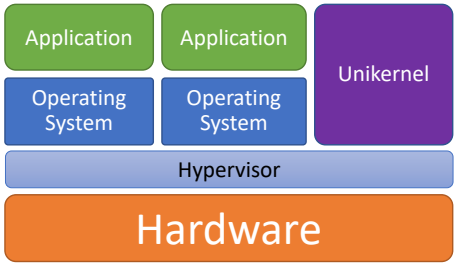


What OS has what internals?

- Monolithic
  - UNIX
  - Linux
  - Most BSD variants
  - …
- Microkernel
  - MINIX
  - Mac
  - L4 family of kernels
  - …
- Hybrids
  - Windows
  - Xnu/Darwin
  - DragonFly BSD
  - …

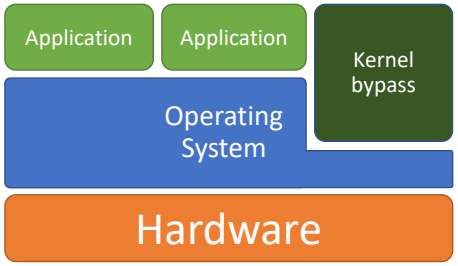
What else? (1)

- Virtual machines/hypervisors (e.g., VirtualBox, Xen)
  - Are not OSes
    - They interface with the hardware (below)
    - They provide an hardware interface (above)
  - Run OSes
    - Linux, Windows, BSD, etc.
    - **Unikernels (“libOS”, cf. exokernel)**



What else? (2)

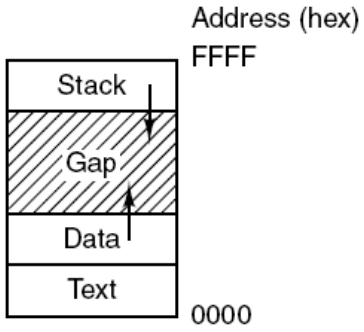
- Kernel-bypass solutions (e.g., DPDK, SPDK)
  - Are not OSes
    - Interface with the OS and the hardware
  - Run part of OSes
    - Mostly BSD



- HW #1 Problem 6

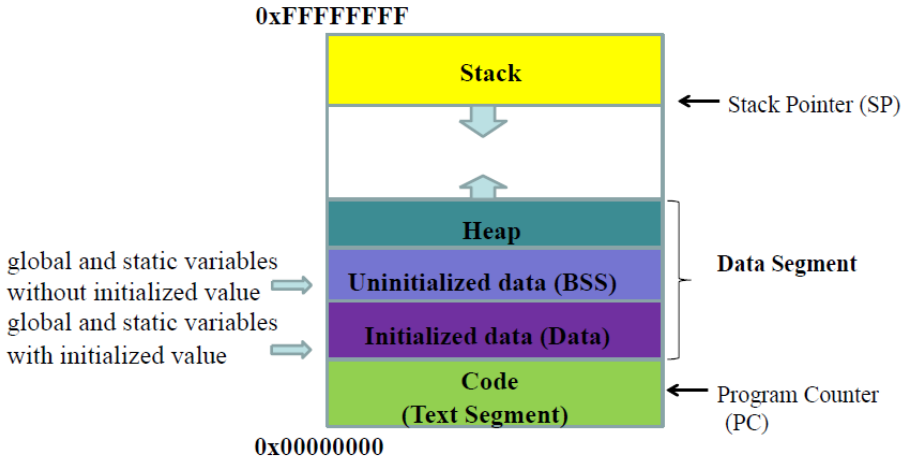
- What are the differences between
  - OS vs hypervisor
  - Monolithic vs microkernel vs hybrid

Address Space/Memory Layout (1)



Processes have three **main** segments: text, data, and stack. (MOS Figure 1-20)

Address Space/Memory Layout (2)



UNIX Process Creation Example

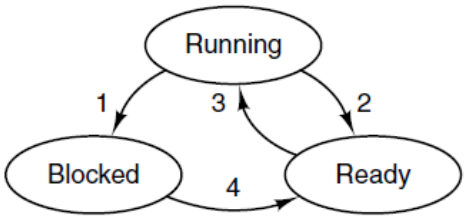
```
#include <stdio.h>
#include <unistd.h>
int main()
{
   int pid1 = getpid(); int pid2;
   int ret = fork();
   if (ret < 0) { /* error */
      printf("error pid1: %d ret: %d\n", pid1, ret);
   } else if (ret > 0) { /* parent */
      pid2 = getpid();
      printf("parent pid2: %d pid1: %d ret: %d\n", pid2, pid1, ret);
   } else { /* child */
      pid2 = getpid();
      printf("child pid2: %d pid1: %d ret: %d\n", pid2, pid1, ret);
   }
   return 0;
}
```

Process State

- Execution state – What a process is currently doing
  - **Running** – Executing instructions on the CPU
    - It is the process that has control of the CPU
  - **Ready** – Waiting to be assigned to the CPU
    - Ready to execute, but another process is executing on the CPU
  - **Waiting/blocked** – Waiting for an event
    - It cannot make progress until event is signaled (e.g., IO completes)

For 1-CPU system, how many processes can be in the running state simultaneously?

Process State Transitions



1. Process blocks for input
2. Scheduler picks another process
3. Scheduler picks this process
4. Input becomes available

A process can be in running, blocked, or ready state. Transitions between these states are as shown. (MOS Figure 2-2)

Why are there transitions missing?

Process Data Structures

- How does the OS represents a process?

- Maintains a table called **Process Table**
- Process Table has **one entry per process**
- The entry is called **Process Control Block (PCB)**
- A PCB contains **all the infos about a process**

Process Table

PCB0
PCB1
PCB2
...

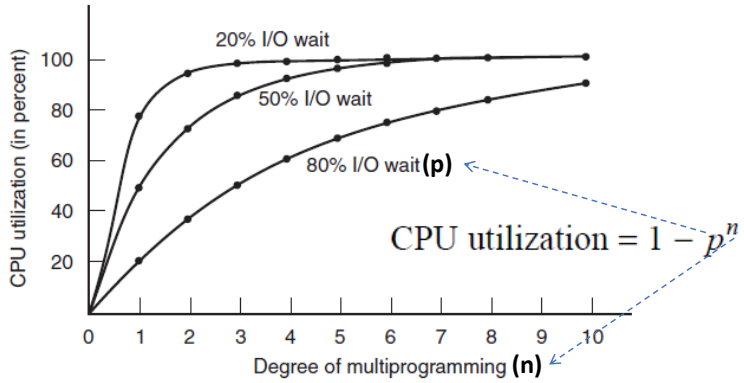
Process Table Entry (or PCB)

Process management	**Memory management**	**File management**
Registers	Pointer to text segment info	Root directory
Program counter	Pointer to data segment info	Working directory
Program status word	Pointer to stack segment info	File descriptors
Stack pointer		User ID
Process state		Group ID
Priority		
Scheduling parameters		
Process ID		
Parent process		
Process group		
Signals		
Time when process started		
CPU time used		
Children's CPU time		
Time of next alarm		

Some of the fields of a typical process table entry. (MOS Figure 2-4)

Modeling Multiprogramming (1)

On a Single CPU



CPU utilization as a function of the number of processes in memory. (MOS Figure 2-6)

Modeling Multiprogramming (2)

On a Single CPU

- **Approximate model (for single CPU)**
- Can be used for simple CPU performance prediction
  - OS requires 128MB, each program requires 128MB, 80% wait
    - 512MB RAM –> 3 applications, CPU utilization 48%
    - 1024MB RAM -> 7 applications, CPU utilization 79%
    - 1536MB RAM -> …
    - 2048MB RAM -> …

Process-to-OS Communication (1)

- **System Calls**
  - Vary from one OS to another
- Invoking a System Call (or Syscall)
  - When a running application needs a system service
- Mechanics highly **machine dependent**
  - Assembly code
    - A procedure library is provided (C, other languages)
  - Interrupt instruction
  - Special machine instruction (Intel's `sysenter`/`sysret`)
- Syscalls are numbered (generally the case)

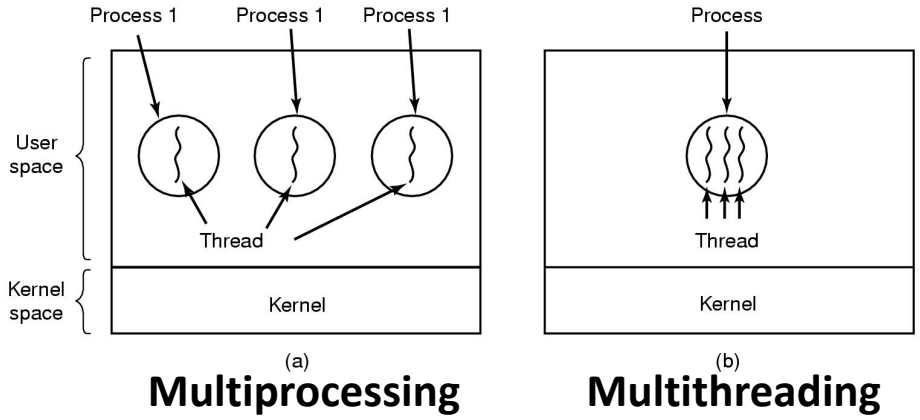
- QUIZ #1
- HW #1 Problem 1
- HW #1 Problem 3

QUIZ #1

- **What is the output?**
- the process that starts the execution of main() has the pid 231. The first fork will create the child process 232. The second fork will create the child with pid 233 from main() and the child with pid 234 from the process with pid 232

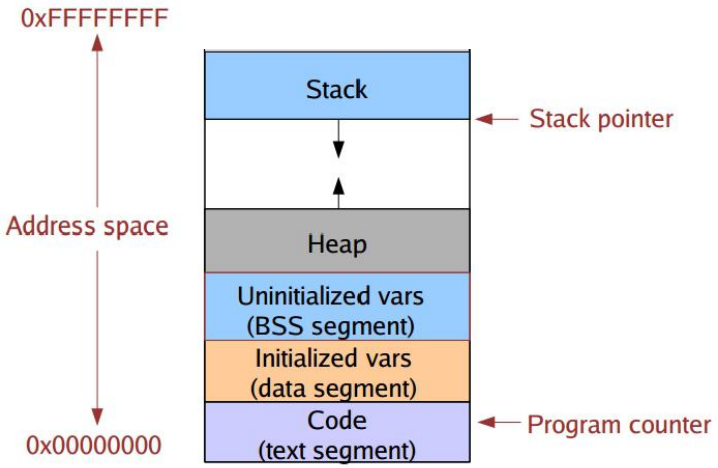
```
int main()
{
  int x=1, y=2;
  int pid;

  x = fork();
  if(x==0)
  {
    printf("x = %d\n", x);
    printf("y = %d\n", y);
    pid = getpid();
    printf("I am process: %d\n", pid);
  }

  y = fork();
  if(y==0)
  {
    printf("x = %d\n", x);
    printf("y = %d\n", y);
    pid = getpid();
    printf("I am process: %d\n", pid);
  }

  return 0;
}
```

Multiprocessing vs Multithreading

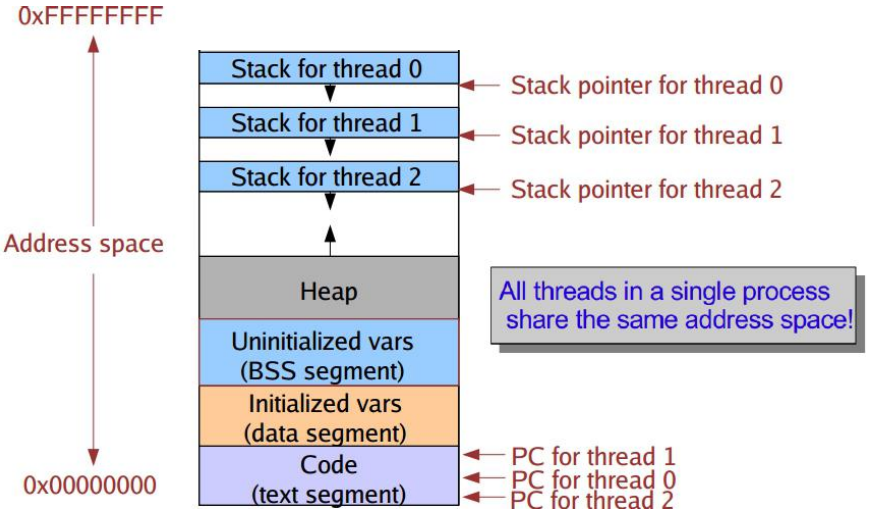


(a) Three processes each with one thread. (b) One process with three threads. (MOS Figure 2-11)

Process Address Space

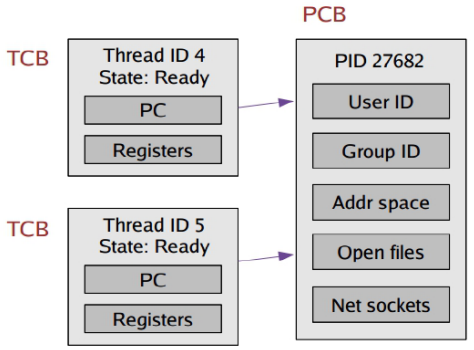


Address Space **with Threads**



Thread Control Block (TCB)

- Break the PCB into two pieces
- Info on program execution stored in Thread Control Block (**TCB**)
  - Program counter
  - CPU registers
  - Scheduling information
  - Pending I/O information
- Other infos stored in Process Control Block (**PCB**)
  - Memory management information
  - Accounting information

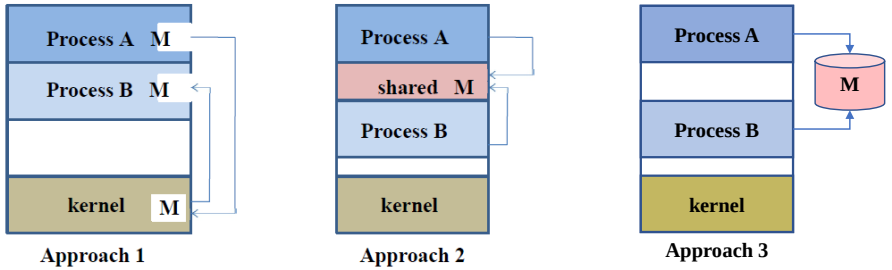


- HW #1 Problem 2
- HW #1 Problem 4
- HW #1 Problem 5

- What are the differences between
  - Threads and processes (practically)
  - Threads and processes (internal structures)
  - Usr threading vs krn threading

How One Process Can Pass Information to Another? (1)

1. By passing messages through the kernel
2. By sharing memory
3. By sharing a file
4. Through asynchronous signals or alerts
5. …



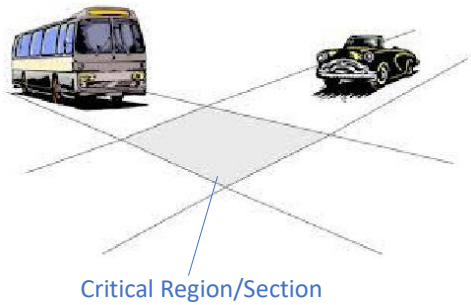
Race Conditions

- **Race condition**
  - Two or more processes reading or writing shared data
  - The final result depends on who runs precisely when

- **How to avoid race conditions?**

Modeling Programs to Solve Race Conditions

- **Critical Region or Section**
  - Part of the program where the shared data is accessed
    - Uncoordinated read/write of the data in critical section may lead to races



Mechanisms for Mutual Exclusion

- Hardware
  - Disabling interrupts
- Busy waiting
  - Lock variable
  - Strict alternation
  - Peterson's solution
  - Test-and-Set Lock (TSL) Instruction (hardware support)
- Sleep and wakeup (Part 2)
  - sleep() and wakeup()
  - Semaphores
  - Mutexes
  - Monitors

Semaphores (1)

- Purpose
  - Solve the lost wakeup problem
  - Avoid race conditions
- Ideas
  - Save the number of ***wakeups***
  - Do check and sleep/wakeup ***atomically***
- How does it work?
  - Define a count variable, the ***semaphore***
  - Operations
    - ***Down(semaphore)*** equivalent of check and sleep()
    - ***Up(semaphore)*** equivalent of check and wakeup()

Semaphores (2) (Book)

- E.W. Dijkstra, 1965
  - Semaphore variable ***value***
    - Number of resources protected by the semaphore

```
int value=1; /* protecting one resource */
```

  - **Proberen (value)** `/* P, Down */`

```
if (value > 0)
  value--;
else if (value == 0)
  sleep(); /* sleep without completing down */
```

  - **Verhogen (value)** `/* V, Up */`

```
if (sleeping_processes())
  wakeup(); /* wakeup one sleeper with a policy */
else
  value++;
```

  - `sleeping_processes()` returns true if there are processes sleeping on the semaphore

Semaphores (2) (Updated)

- E.W. Dijkstra, 1965
  - Semaphore variable ***value***
    - Negative or zero values, pending wakeups
    - Positive values, no pending wakeups
      - Number of resources protected by the semaphore

```
int value=1; /* protecting one resource */
```

  - **Proberen (value)** `/* P, Down */`

```
value--;
if (value < 0)
  sleep(); /* sleep without completing down */
```

  - **Verhogen (value)** `/* V, Up */`

```
value++;
if (value =< 0)
  wakeup(); /* wakeup one sleeper with a policy */
```

Semaphores (3)

- *Down()* and U*p()* are **atomic**
  - Checking values, changing values, going to sleep, or wakeup, all is done as a single action
    - Once the operation has started, no other process can access the semaphore until completed or blocked
  - Implementation in kernel-mode
    - **Disabling interrupts** to avoid scheduling events
    - **TSL/XCHG instructions** to protect from other CPUs
- Sleeping in *Down()* won't miss wakeups from *Up()*
- Atomicity is essential to solving synchronization problems and avoiding race conditions

- QUIZ #2
- HW #2 Problem 1
- HW #2 Problem 2

- What are the differences between
  - Busy waiting and sleep & wakeup
  - The different mechanisms presented
  - Semaphore and mutexes

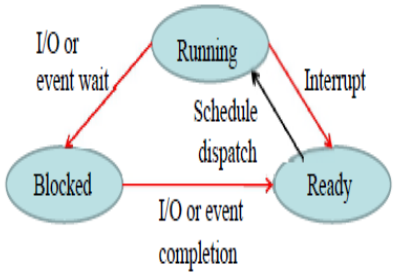
QUIZ #2

- Write down a sequence of code lines (just the list of line code numbers) that may take the following code into a deadlock

```
0:  const int N=2;
1:  semaphore full=N, empty=0;
2:  semaphore mux=1;
3:  Function Producer:
4:  while (1){
5:   produce an item A;
6:   down(full);
7:   down(mux);
8:   insert item;
9:   up(mux);
10:  up(empty);
11: }

12: Function Consumer:
13: while (1){
14:   down(mux);
15:   down(empty);
16:   remove item;
17:   up(mux);
18:   up(full);
19:   consume an item;
20: }
```

When to Schedule

- Scheduling decisions may take place when a **process/thread**
  - Is created
  - In running state exits
  - Blocks on IO, or an event
    - Switch from *Running* to *Blocked*
- Scheduling decisions may take place when an **interrupt occurs**
  - Clock interrupt
    - Switch from *Running* to *Ready*
  - IO interrupt, or (unblocking) syscall
    - Switch from *Blocked* to *Ready*



First-come First-served (FCFS)

- Processes are assigned to the CPU in the order they request it (or they arrive)
- Non-preemptive

FCFS Example (1)

- Arrival **order** for the processes
  - P1, P2, P3
- Turnaround time
  - P1 = 24
  - P2 = 27
  - P3 = 30
- Average turnaround time
  - (24+27+30)/3 = 27
- Short process delayed by long process
  - **Convoy effect**

Process	CPUTime
P1	24
P2	3
P3	3

Turnaround Time – average time taken by a job to complete after submission

FCFS Example (2)

- Arrival **order** for the processes
  - P2, P3, P1
- Turnaround time
  - P1 = 30
  - P2 = 3
  - P3 = 6
- Average turnaround time
  - (30+3+6)/3 = 13
- Much better than the previous case

Process	CPUTime
P1	24
P2	3
P3	3

Shortest Job First (SJR)

- Associate with each process the **length of its CPU time**
- Use the CPU time length to schedule the process with the shortest CPU time first
- Two variations
  - *Non-preemptive* – once CPU is given to the process, it cannot be taken away until completion (or blocking)
  - *Preemptive* – if a new process arrives with CPU time less than the remaining time of current executing process, preempt. **(*Shortest Remaining Time Next, SRTN*)**

Non-Preemptive SJF Example

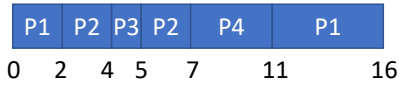
- Arrival time for the processes
  - P1 at 0, P2 at 2, P3 at 4, P4 at 5
- Turnaround time
  - P1 = 7
  - P2 = 12 – 2 = 10
  - P3 = 8 – 4 = 4
  - P4 = 16 – 5 = 11
- Average turnaround time
  - (7+10+4+11)/4 = 8
- 3 ctx switches

Process	ArrivalTime	CPUTime
P1	0	7
P2	2	4
P3	4	1
P4	5	4

Preemptive SJF Example

- Arrival time for the processes
  - P1 at 0, P2 at 2, P3 at 4, P4 at 5
- Turnaround time
  - P1 = 16
  - P2 = 7 – 2 = 5
  - P3 = 5 – 4 = 1
  - P4 = 11 – 5 = 6
- Average turnaround time
  - (16+5+1+6)/4 = 7
- 5 ctx switches

Process	ArrivalTime	CPUTime
P1	0	7
P2	2	4
P3	4	1
P4	5	4



Scheduling in Real-time Systems

- Categorization
  - **Hard real-time**
    - There are absolute deadlines that must be met
  - **Soft real-time**
    - Missing an occasional deadline is undesirable but tolerable
- Assumption(s)
  - **Processes behavior is predictable and known in advance**
    - **Release time (R_i), execution time (C_i), deadline (D_i)**
  - Periodic process
  - Sporadic process
  - Aperiodic process

Scheduling Periodic Processes

- How many periodic processes are **schedulable**?
  - "can fit/run on a processor" – single CPU
  - All combination of processes that satisfy the formula

$$\sum_{i=1}^{m} \frac{C_i}{P_i} \leq 1$$

- ***m*** periodic events
- Event ***i*** occurs
  - with period $\boldsymbol{P_i}$
  - requires $\boldsymbol{C_i}$ time on the CPU

- HW #2 Problem 3
- HW #2 Problem 4

Scheduling Problem #1

- Five batch jobs. A through E, arrive at a computer center at 0, 2, 4, 7, and 10 minute. They have estimated running times of 10, 6, 2, 4, and 8 minutes. Their (externally determined) priorities are 3, 5, 2, 1, and 4, respectively, with 5 being the highest priority. For each of the following scheduling algorithms, determine the mean process turnaround time. Ignore process switching overhead.
  - Round robin
  - Priority scheduling
  - First-come, first-served (run in order 10, 6, 2, 4, 8)
  - Shortest job first
- For (a), assume that the system is multiprogrammed, and that each job gets its fair share of the CPU. For (b) through (d), assume that only one job at a time runs, until it finishes. All jobs are completely CPU bound.

Scheduling Problem #2

- A soft real-time system has four periodic events with periods of 50, 100, 150, and 200 msec each. Suppose that the four events require 30, 20, 30, and 8 msec of CPU time, respectively. Is the system schedulable?