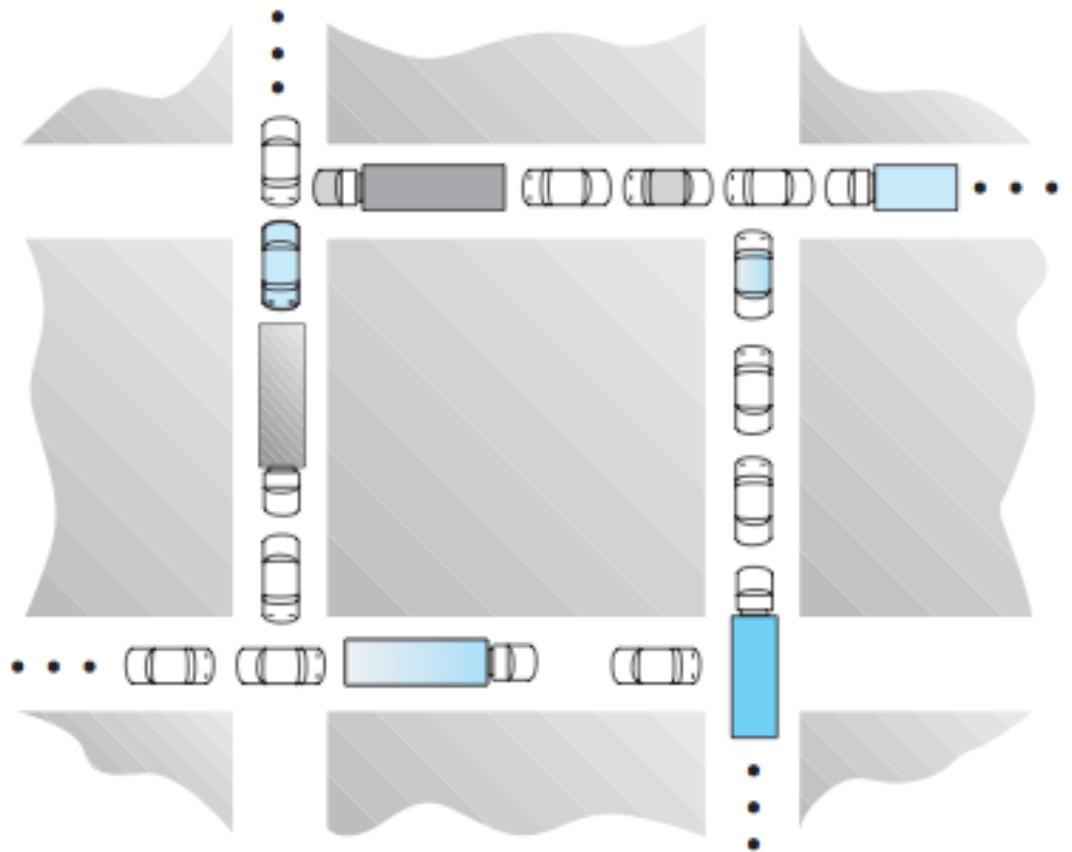


Deadlocks - II

Didem Unat
Lecture 16
COMP304 - Operating Systems (OS)

Traffic Example

- Traffic only in one direction.
- Each street can be viewed as a resource.
- If a deadlock occurs, it can be resolved if one car backs up (preempt resources and rollback).
- Several cars may have to be backed up if a deadlock occurs.
- Starvation is possible.



Resource-Allocation Graph

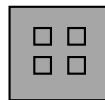
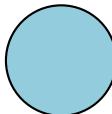
Deadlocks can be described in terms of a directed graph

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

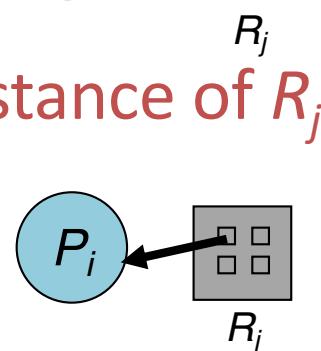
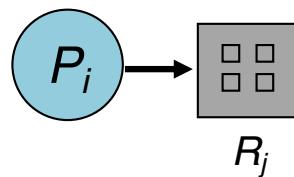
- V is partitioned into two types:
 - $P = \{P_1, P_2, \dots, P_n\}$, the set consisting of all the processes in the system.
 - $R = \{R_1, R_2, \dots, R_m\}$, the set consisting of all resource types in the system.
- **request edge:** directed edge $P_i \rightarrow R_j$
- **assignment edge:** directed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

Resource-Allocation Graph

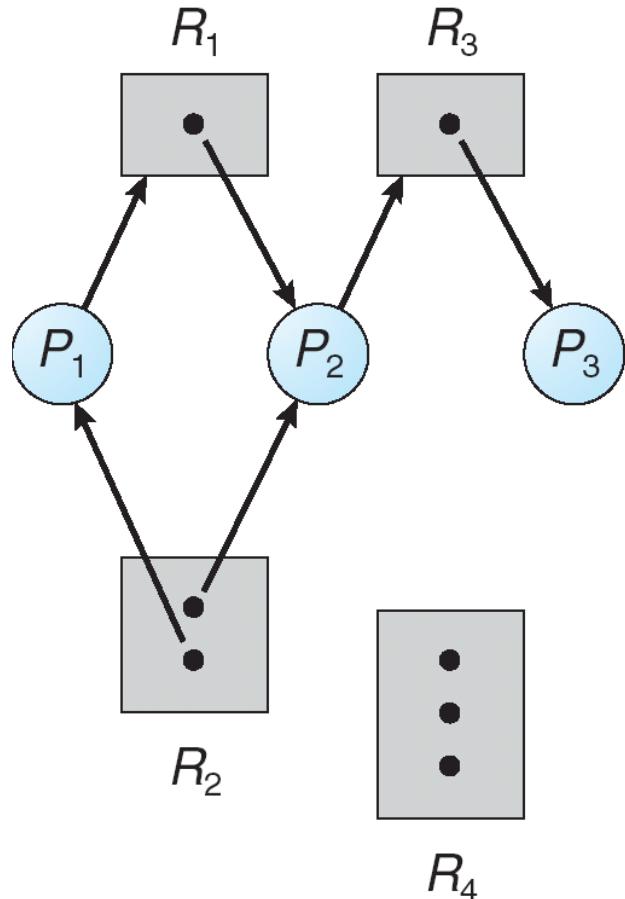
- Process
- Resource type with 4 instances



- P_i requests instance of R_j
- P_i is holding an instance of R_j



Example Resource Allocation Graph



Resource instances:

- One instance of resource type R_1
- Two instances of resource type R_2
- One instance of resource type R_3
- Three instances of resource type R_4

Process states:

- Process P_1 is holding an instance of resource type R_2 and is waiting for an instance of resource type R_1 .
- Process P_2 is holding an instance of R_1 and an instance of R_2 and is waiting for an instance of R_3 .
- Process P_3 is holding an instance of R_3 .

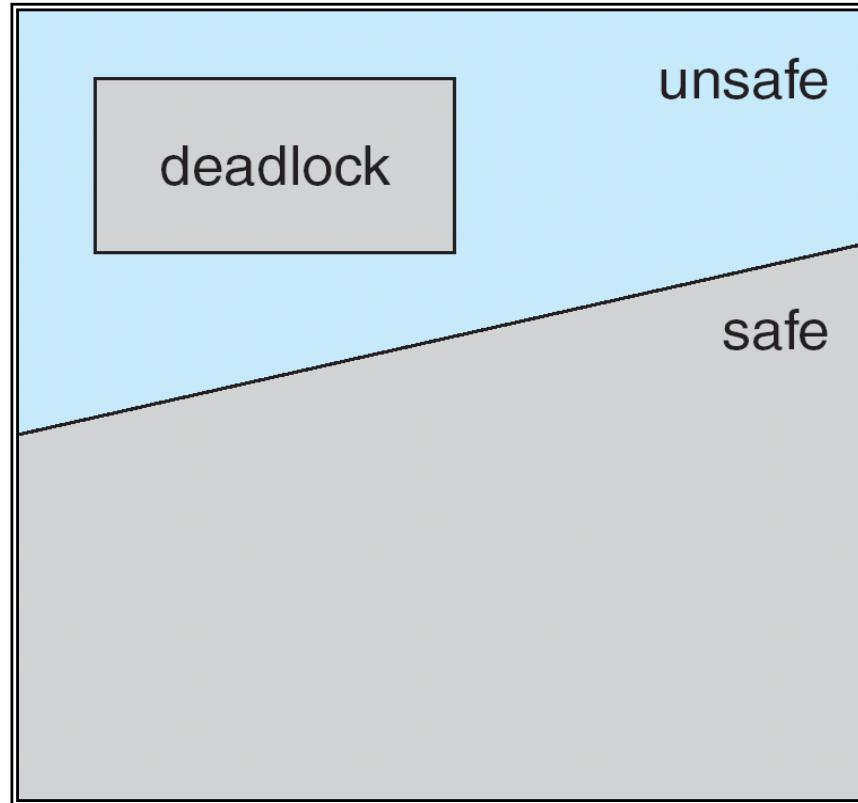
Basic Facts

- 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

- 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**.

Basic Facts

- If a system is in **safe state** \Rightarrow no deadlocks.
- If a system is in **unsafe state** \Rightarrow possibility of deadlock.
- **Avoidance** \Rightarrow ensure that a system will never enter an unsafe state.

Safe State

- System is in safe state if there exists a safe sequence of all processes.
- Sequence $\langle P_1, P_2, \dots, P_n \rangle$ is **safe** if for each P_i , the resources that P_i can still request can be satisfied by currently available resources + resources held by all the P_j , with $j < i$.
 - If P_i 's resource needs are not immediately available, then P_i can wait until all P_j have finished.
 - When P_j is finished, P_i can obtain needed resources, execute, return allocated resources, and terminate.
 - When P_i terminates, P_{i+1} can obtain its needed resources, and so on.

Safe State Sequence Example

- 12 resources, three processes (P_0 , P_1 , and P_2)

	Max Needs	Currently Held #Resources at t_0
P_0	10	5
P_1	4	2
P_2	9	2

- 9 resources are currently used, $12 - 9 = 3$ available
- Current state is safe because a safe sequence exists $\langle P_1, P_0, P_2 \rangle$
 - p_1 can complete with current resources
 - p_0 can complete with current+ p_1
 - p_2 can complete with current + p_1+p_0

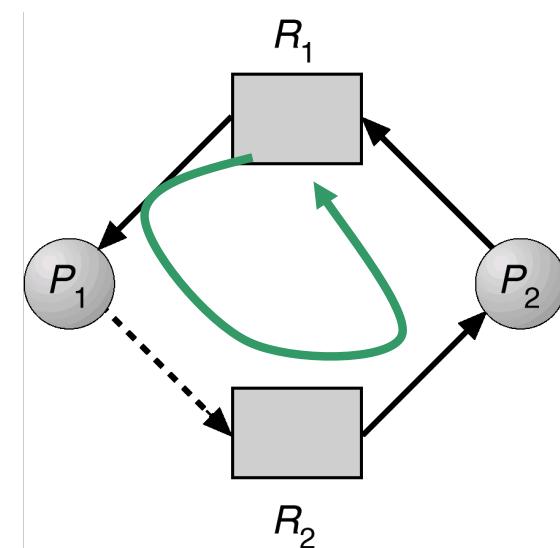
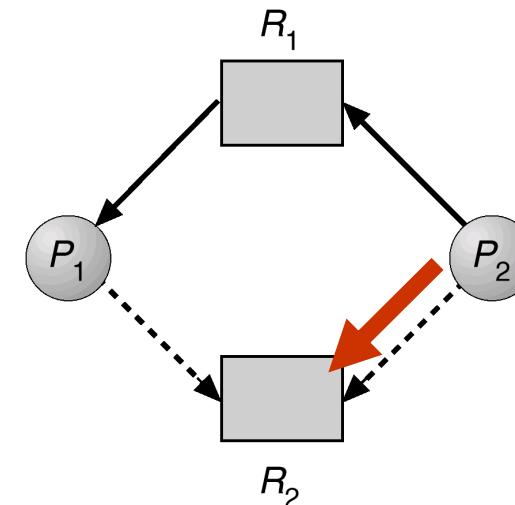
Deadlock Avoidance Algorithms

- Single instance of a resource type: Use a **resource-allocation graph**
- Multiple instances of a resource type: Use the **banker's algorithm**

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**.



Resource-Allocation Graph Algorithm

- Suppose that process P_i requests a resource R_j
- The request can be granted only if converting the **request edge** to an **assignment edge** does not result in the formation of a **cycle** in the resource allocation graph

Convert $P_i \rightarrow R_j$ to $R_j \rightarrow P_i$ only if no cycle

Banker's Algorithm

- Multiple instances of resource types.
- Each process must a priori claim **maximum use**.
- When a process requests a resource it may have to wait.
- When a process requests a set of resources:
 - Will the system be at a safe state after the allocation?
 - Yes → Grant the resources to the process.
 - No → Block the process until the resources are released by some other process.

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 $\text{Request}_i[j] = k$ then process P_i wants k instances of resource type R_j .

If $\text{request}_i > \text{need}_i$

error (asked for too much)

If $\text{request}_i > \text{available}$

wait(can't supply it now)

Resources are available to satisfy the request

Let's assume that we satisfy the request, then

$\text{available} = \text{available} - \text{request}_i$

$\text{allocation}_i = \text{allocation}_i + \text{request}_i$

$\text{need}_i = \text{need}_i - \text{request}_i$

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) $\text{Finish}[i] = \text{false}$
- (b) $\text{Need}_i \leq \text{Work}$

If no such i exists, go to step 3.

Step 2: Found an i

```
Finish[i] = true           //done with this process  
Work = Work + Allocationi  
go to step 1
```

Step 3:

If $\text{Finish}[i] == \text{true}$ 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

Example of Banker's Algorithm

- 5 processes P_0 through P_4
- 3 resource types:
 - A (10 instances), B (5 instances), and C (7 instances).
- Snapshot at time T_0 :

	Allocation			Max			Available		
	A	B	C	A	B	C	A	B	C
P0	0	1	0	7	5	3	3	3	2
P1	2	0	0	3	2	2			
P2	3	0	2	9	0	2			
P3	2	1	1	2	2	2			
P4	0	0	2	4	3	3			

Example (Cont.)

- The content of the matrix Need is defined to be
 $\text{Need} = \text{Max} - \text{Allocation}$.

	Allocation			Max			Need		
	A	B	C	A	B	C	A	B	C
P0	0	1	0	7	5	3	7	4	3
P1	2	0	0	3	2	2	1	2	2
P2	3	0	2	9	0	2	6	0	0
P3	2	1	1	2	2	2	0	1	1
P4	0	0	2	4	3	3	4	3	1

- The system is in a safe state since the sequence
 $\langle P_1, P_3, P_4, P_2, P_0 \rangle$ satisfies safety criteria.

Example P_1 Requests (1,0,2)

- If P_1 requests (1,0,2), should we grant it?
 - Check that $\text{Request}_1 \leq \text{Need}_1$: $(1,0,2) \leq (1,2,2) \Rightarrow \text{true}$.
 - Check that $\text{Request}_1 \leq \text{Available}$: $(1,0,2) \leq (3,3,2) \Rightarrow \text{true}$.

	<u>Allocation</u>			<u>Need</u>			<u>Available</u>		
	A	B	C	A	B	C	A	B	C
P_0	0	1	0	7	4	3	2	3	0
P_1	3	0	2	0	2	0			
P_2	3	0	1	6	0	0			
P_3	2	1	1	0	1	1			
P_4	0	0	2	4	3	1			

- Executing safety algorithm shows that sequence $\langle P_1, P_3, P_4, P_0, P_2 \rangle$ satisfies safety requirement \Rightarrow grant request of P_1 .
 - Can request for (0,2,0) by P_3 be granted in this state?
 - Can request for (3,3,0) by P_4 be granted in this state?
 - Can request for (0,2,0) by P_0 be granted in this state?

Example P_0 Requests (0,2,0)

- P_0 Requests (0,2,0), should we grant the resources?
 - Check that $\text{Request}_1 \leq \text{Need}_1$: $(0,2,0) \leq (7,4,3) \Rightarrow \text{true}$.
 - Check that $\text{Request}_1 \leq \text{Available}$: $(0,2,0) \leq (2,3,0) \Rightarrow \text{true}$.

	<u>Allocation</u>			<u>Need</u>			<u>Available</u>		
	A	B	C	A	B	C	A	B	C
P_0	0	3	0	7	2	3	2	1	0
P_1	3	0	2	0	2	0			
P_2	3	0	1	6	0	0			
P_3	2	1	1	0	1	1			
P_4	0	0	2	4	3	1			

- P_0 's request will be denied because the resulting state is unsafe

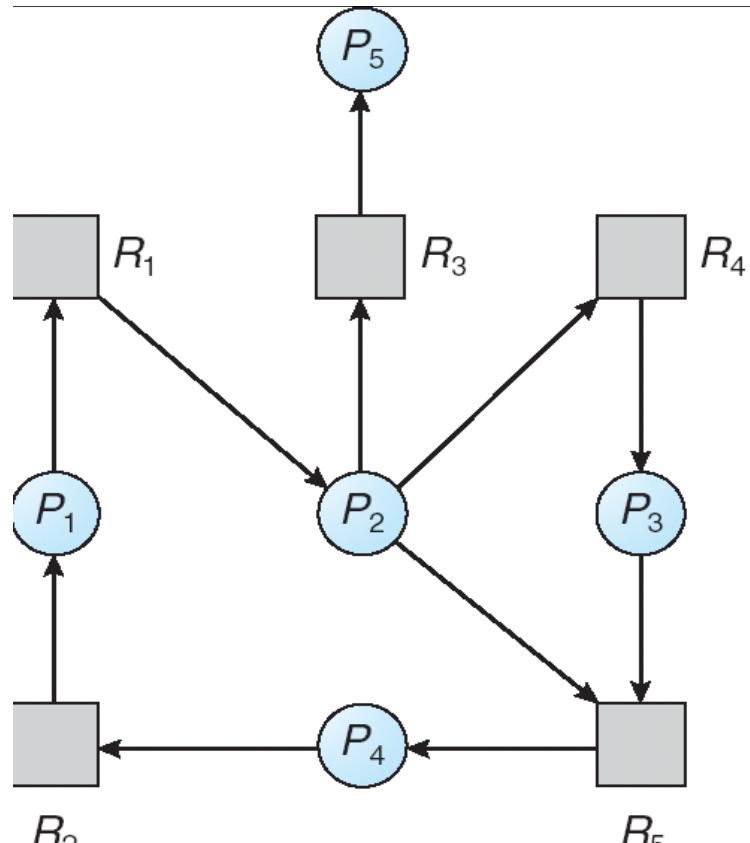
Deadlock Detection

- 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

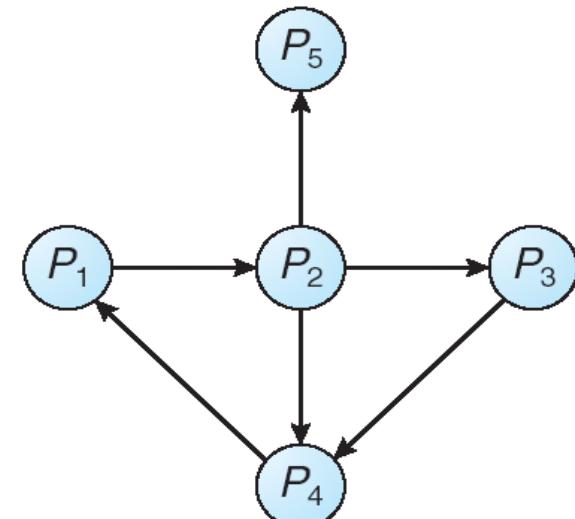
Single instance of each resource type

- 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



(a)



(b)

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.

Example of Detection Algorithm

- Five processes P_0 through P_4
- Three resource types:
 - A (7 instances), B (2 instances), and C (6 instances).
- Snapshot at time T_0 :

	Allocation			Request			Available		
	A	B	C	A	B	C	A	B	C
P_0	0	1	0	0	0	0	0	0	0
P_1	2	0	0	2	0	2			
P_2	3	0	3	0	0	0			
P_3	2	1	1	1	0	0			
P_4	0	0	2	0	0	2			

No deadlock

Sequence $\langle P_0, P_2, P_3, P_1, P_4 \rangle$ will result in $\text{Finish}[i] = \text{true}$ for all i .

Example (Cont.)

- P_2 requests an additional instance of type C.

	<u>Request</u>		
	A	B	C
P_0	0	0	0
P_1	2	0	2
P_2	0	0	1
P_3	1	0	0
P_4	0	0	2

System is deadlocked

- State of system?
 - Can reclaim resources held by process P_0 , but insufficient resources to fulfill other processes requests.
 - Deadlock exists, consisting of processes P_1 , P_2 , P_3 , and P_4 .

Deadlock Detection

- How often should we call deadlock detection algorithm?
 - When there is a low CPU utilization
 - Periodically but not too often
- **Recovery from Deadlock**
- **Killing** one/all deadlocked processes
 - Keep killing processes, until deadlock broken
 - Repeat the entire computation
- **Preempt** resource/processes until deadlock broken
 - Selecting a victim (based on # resources held, how long executed)
 - Rollback -return to some safe state, restart process for that state.
 - Starvation – same process may always be picked as victim, include number of rollback in cost factor.

Acknowledgments

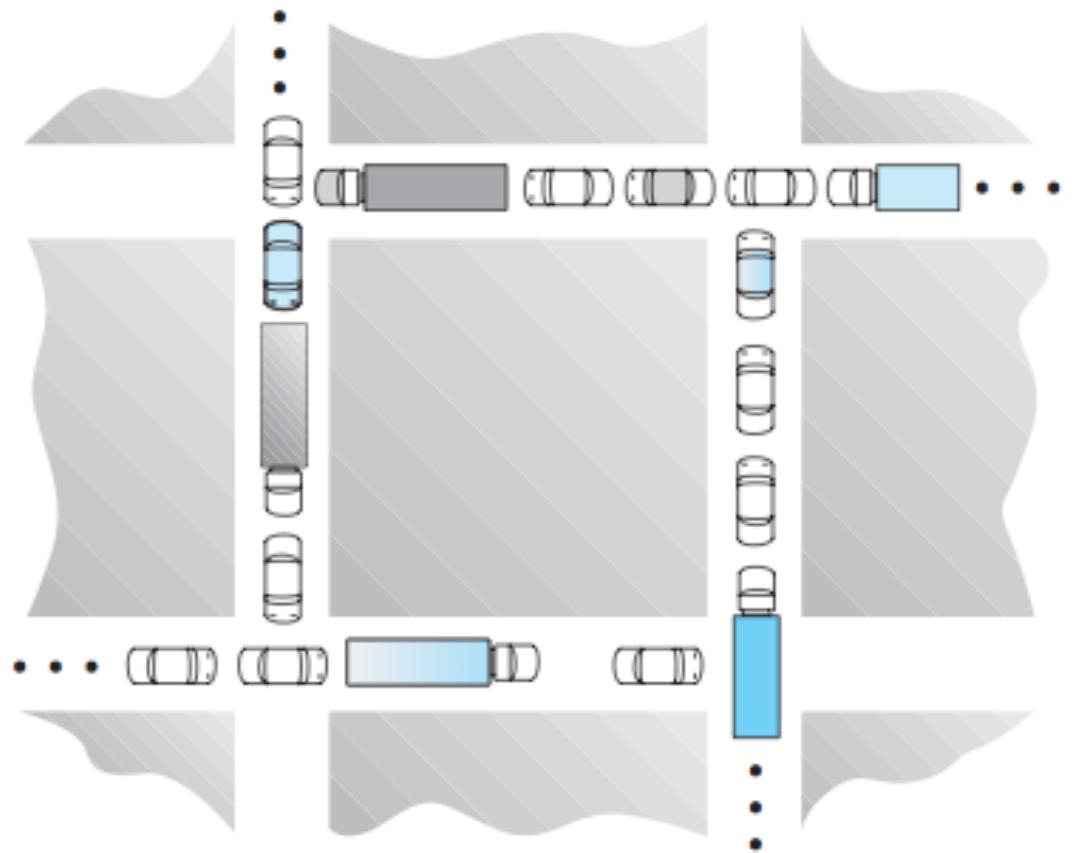
- These slides are adapted from
 - Öznur Özkarap (Koç University)
 - Operating System and Concepts (9th edition) Wiley
 - Cornell University

Deadlocks

Didem Unat
Lecture 15
COMP304 - Operating Systems (OS)

Traffic Example

- Traffic only in one direction.
- Each street can be viewed as a resource.
- If a deadlock occurs, it can be resolved if one car backs up (preempt resources and rollback).
- Several cars may have to be backed up if a deadlock occurs.
- Starvation is possible.



Mutex Code Example

```
/* thread-1 runs in this function */
void *do_work_one(void *param) {
    mutex_lock(&first_mutex);
    mutex_lock(&second_mutex);

    /* Do some work */

    mutex_unlock(&second_mutex);
    mutex_unlock(&first_mutex);
}
```

```
/* thread-2 runs in this function */
void *do_work_two(void *param) {
    mutex_lock(&second_mutex);
    mutex_lock(&first_mutex);

    /* Do some work */

    mutex_unlock(&first_mutex);
    mutex_unlock(&second_mutex);
}
```

In this example, thread one attempts to acquire the mutex locks in the order (1) first mutex, (2) second mutex, while thread two attempts to acquire the mutex locks in the order (1) second mutex, (2) first mutex. **Deadlock** is possible if thread one acquires first mutex while thread two acquires second mutex.

System Model

- Resource types R_1, R_2, \dots, R_m
Ex. CPU cycles, memory space, I/O devices
- Each resource type R_i has W_i instances.
- Each process utilizes a resource as follows:
 - request
 - use
 - release

The deadlock problem: A set of blocked processes each holding a resource and waiting to acquire another resource held by another process in the set.

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 $\{P_0, P_1, \dots, P_n\}$ 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

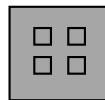
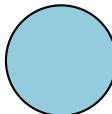
Deadlocks can be described in terms of a directed graph

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

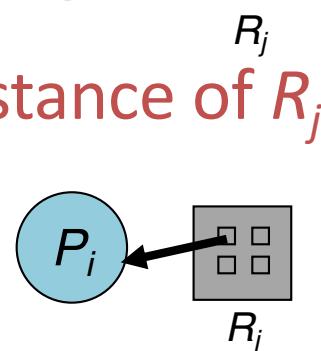
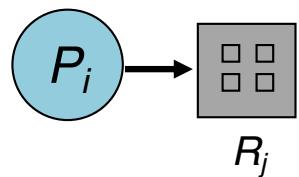
- V is partitioned into two types:
 - $P = \{P_1, P_2, \dots, P_n\}$, the set consisting of all the processes in the system.
 - $R = \{R_1, R_2, \dots, R_m\}$, the set consisting of all resource types in the system.
- **request edge:** directed edge $P_i \rightarrow R_j$
- **assignment edge:** directed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

Resource-Allocation Graph

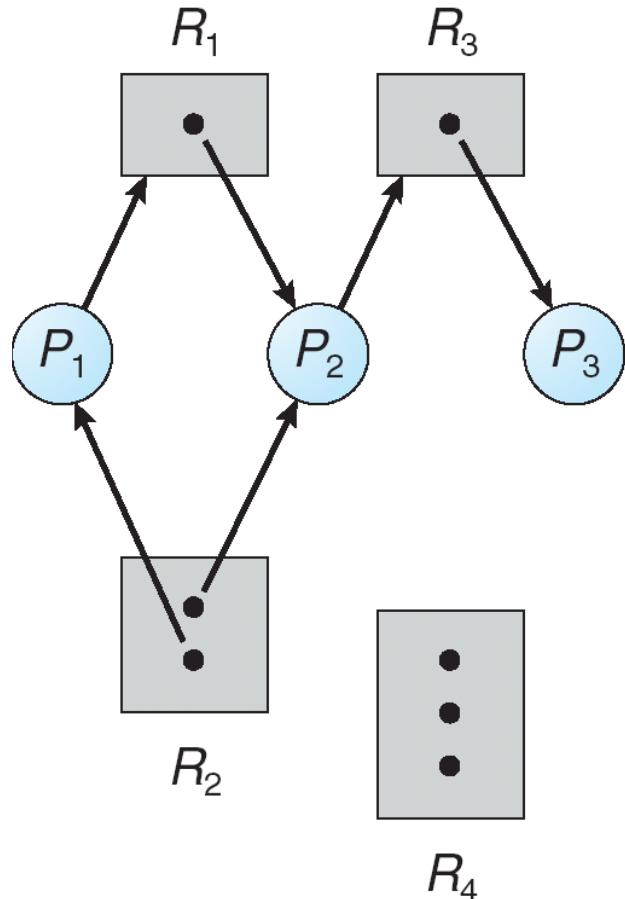
- Process
- Resource type with 4 instances



- P_i requests instance of R_j
- P_i is holding an instance of R_j



Example Resource Allocation Graph



Resource instances:

- One instance of resource type R_1
- Two instances of resource type R_2
- One instance of resource type R_3
- Three instances of resource type R_4

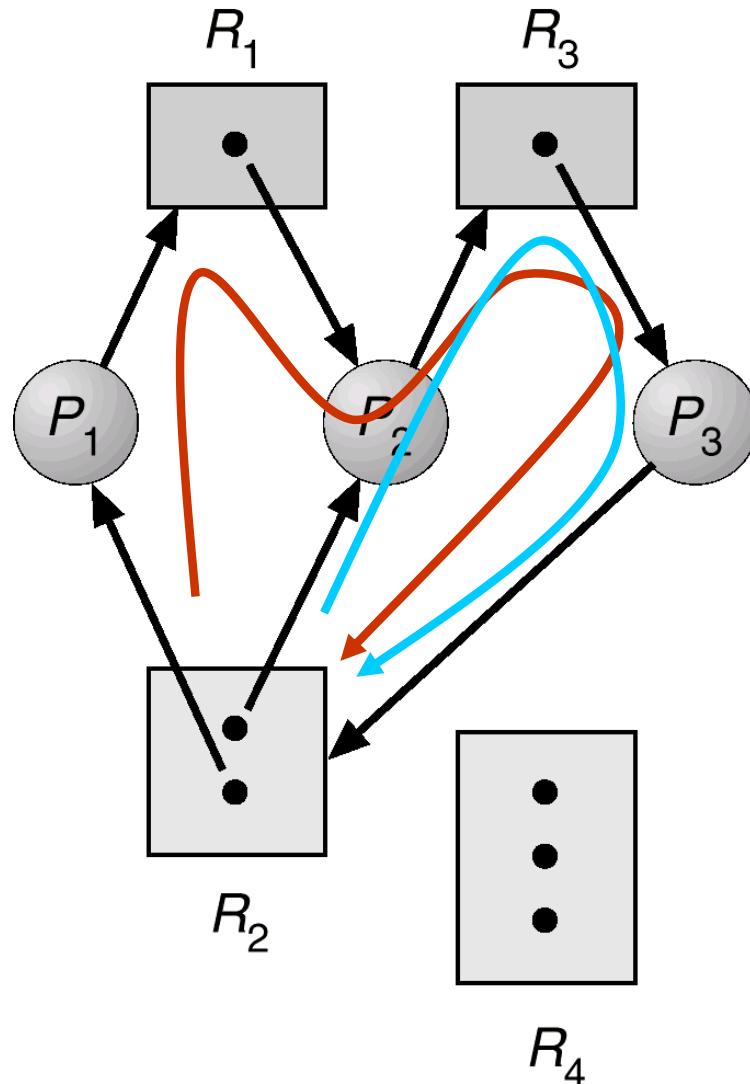
Process states:

- Process P_1 is holding an instance of resource type R_2 and is waiting for an instance of resource type R_1 .
- Process P_2 is holding an instance of R_1 and an instance of R_2 and is waiting for an instance of R_3 .
- Process P_3 is holding an instance of R_3 .

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.

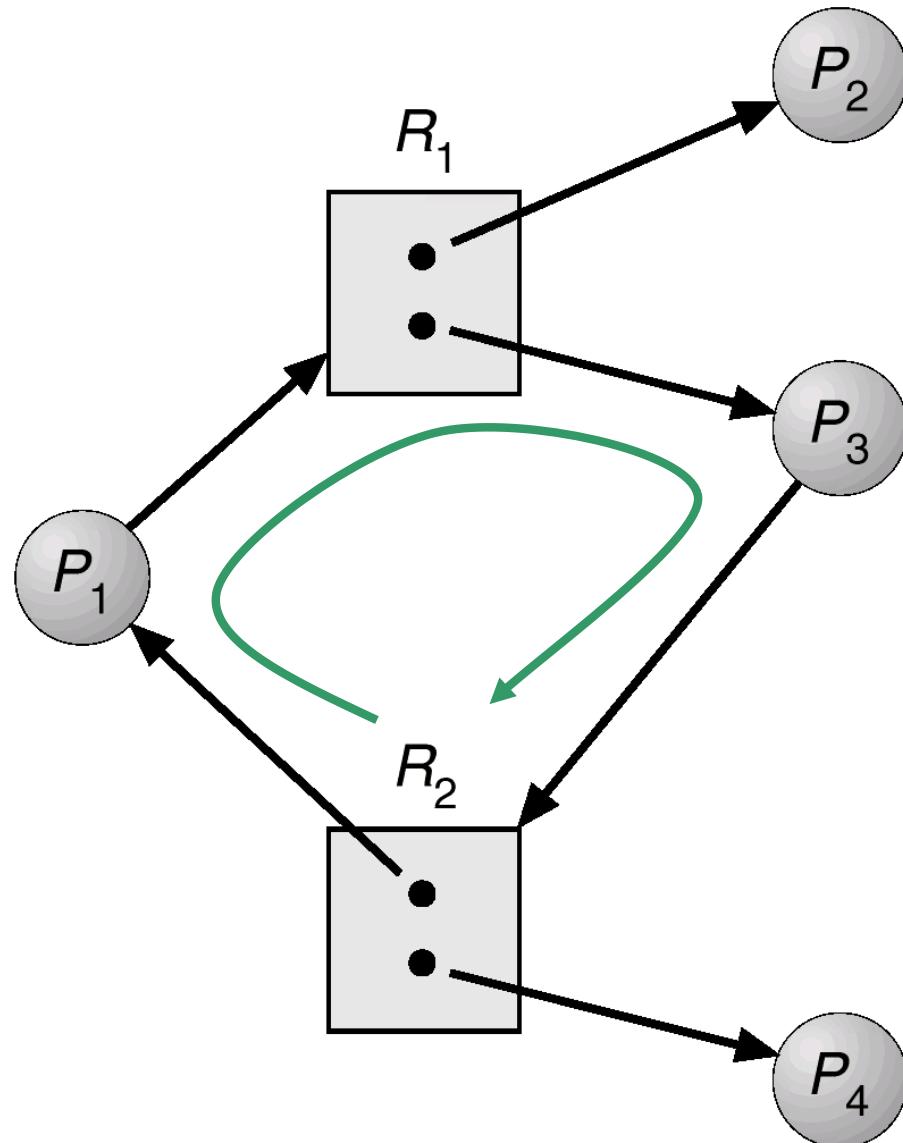
Resource Allocation Graph-1



A resource allocation graph
with a cycle

Is there a deadlock?

Resource Allocation Graph-2



A resource allocation graph
with a cycle

Is there a deadlock?

Deadlock Examples

- Graph-1
 - Deadlock because P1, P2 and P3 are deadlocked. Process P2 is waiting for the resource R3, which is held by process P3. Process P3 is waiting for either process P1 or P2 to release resource R2. In addition process P1 is waiting for process P2 to release resource R1
- Graph-2
 - No deadlock
 - P4 may release its instance of resource type R2. That resource can then be allocated to P3, breaking the cycle

Methods for Handling Deadlocks

- Ensure that the system will **never** enter a deadlock state.

Deadlock prevention or avoidance

- Allow the system to enter a deadlock state and then recover.

Deadlock detection and recovery

- Ignore the problem and pretend that deadlocks never occur in the system

Which one of these methods is used most commonly?

Deadlock Prevention

Methods for ensuring that at least one of the four conditions cannot hold

- **Mutual Exclusion** – not required for **sharable resources**; must hold for **non-sharable resources**.
- **Hold and Wait** –Require process to request and be allocated all its resources before it begins execution, or allow process to request resources only when the process has none.
 - Dining Philosophers' problem: Have both chopsticks otherwise put down the chopstick
 - Low resource utilization; starvation possible.

Deadlock Prevention

- **No Preemption**
 - If a process that is holding some resource requests another resource that cannot be immediately allocated to it, then all resources currently being held are released (preempted).
 - Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting.
- **Circular Wait** – impose a **total ordering of all resource types**, and require that each process requests resources in an increasing order of enumeration.

Total Ordering of Locks

```
/* thread one runs in this function */
void *do_work_one(void *param) {
    mutex_lock(&first_mutex);
    mutex_lock(&second_mutex);

    /* Do some work */

    mutex_unlock(&second_mutex);
    mutex_unlock(&first_mutex);
}
```

```
/* thread two runs in this function */
void *do_work_two(void *param) {
    mutex_lock(&second_mutex);
    mutex_lock(&first_mutex);

    /* Do some work */

    mutex_unlock(&first_mutex);
    mutex_unlock(&second_mutex);
}
```

BSD operating system uses a lock-order verifier, known as **witness**.

Assume that thread one is the first to acquire the locks and does so in the order (1) first mutex, (2) second mutex. Witness records the relationship that first mutex must be acquired before second mutex. If thread two later acquires the locks out of order, witness generates a warning message on the system console.

Example in the Handout

```
void transaction(Account from,
                  Account to, double amount)
{
    mutex lock1, lock2;
    lock1 = get_lock(from);
    lock2 = get_lock(to);
    acquire(lock1);
    acquire(lock2);
    withdraw(from, amount);
    deposit(to, amount);
    release(lock2);
    release(lock1);
}
```

Deadlock Prevention- Summary

- Ensure that at least one of the four conditions cannot hold
- Conservative approach
- Prevention leads to
 - Low utilization of devices
 - Low throughput
 - Frequent starvation
- Alternative method is deadlock avoidance
 - Requires additional information about how resources to be requested

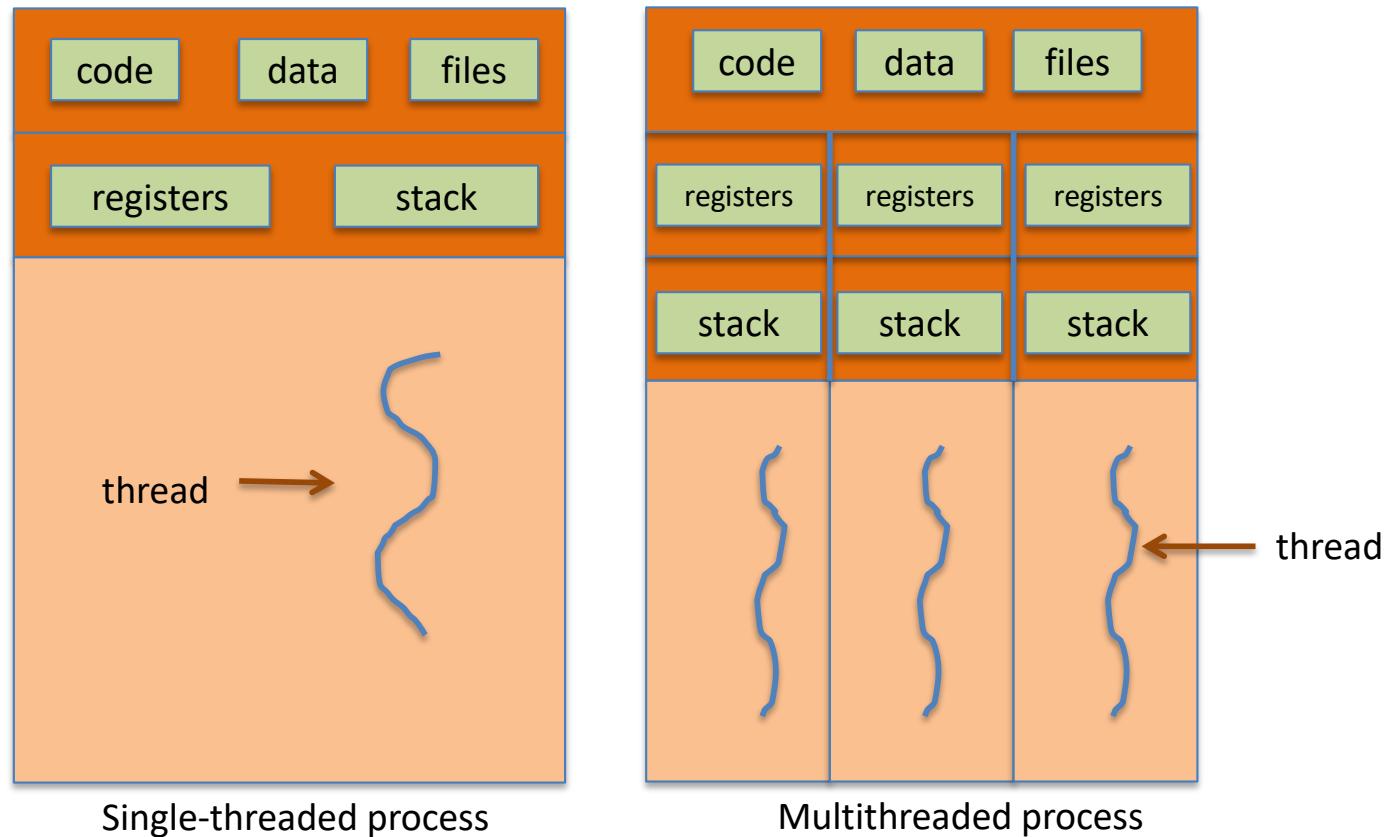
Acknowledgments

- These slides are adapted from
 - Öznur Özkarap (Koç University)
 - Operating System and Concepts (9th edition) Wiley

POSIX Threads

Didem Unat
Lecture 14
COMP304 - Operating Systems (OS)

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

POSIX Threads API (Pthreads)

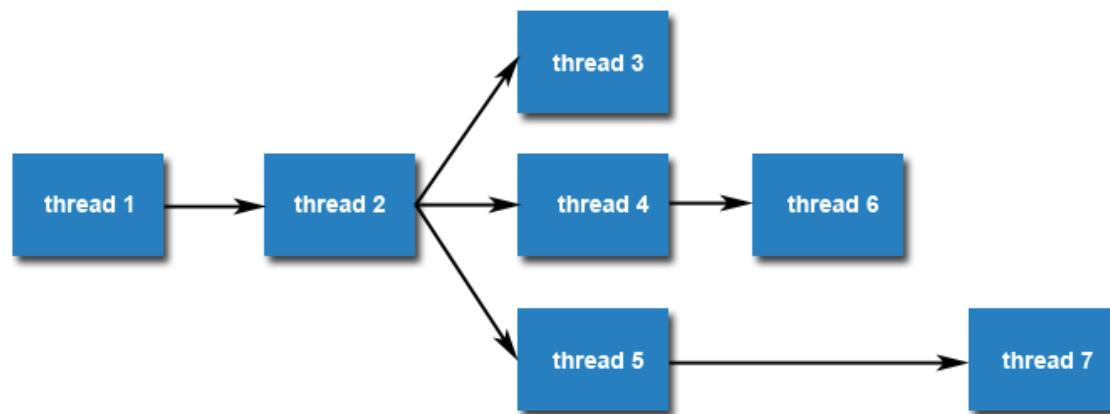
- Pthreads is the POSIX (Portable Operating System Interface for Unix) Thread Library
 - a POSIX standard (IEEE 1003.1c) API for thread creation and synchronization.
- API specifies behavior of the thread library, implementation is up to development of the library.
- Common in UNIX operating systems
 - Solaris, Linux, Mac OS X.
- Is this a Kernel or User Level Thread API?

POSIX Threads API

- Functions of pthreads API provide:
 - Thread management:
 - Creation/termination of threads
 - Set/Query thread attributes
 - Mutexes, semaphores
 - Condition variables
- All identifiers in the thread 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 * (* start_routine)(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
- `errcode` will be set to nonzero if the create operation fails

Thread Creation

- Each thread executes a specific function, `thread_func`
 - For the program to perform different work in different threads, the arguments passed at thread creation distinguish the thread's "id" and any other unique features of the thread.
- After a thread is created, various attributes of it can be set
 - Priority of the thread
 - Stack size
 - Its scheduling policy

Simple Example

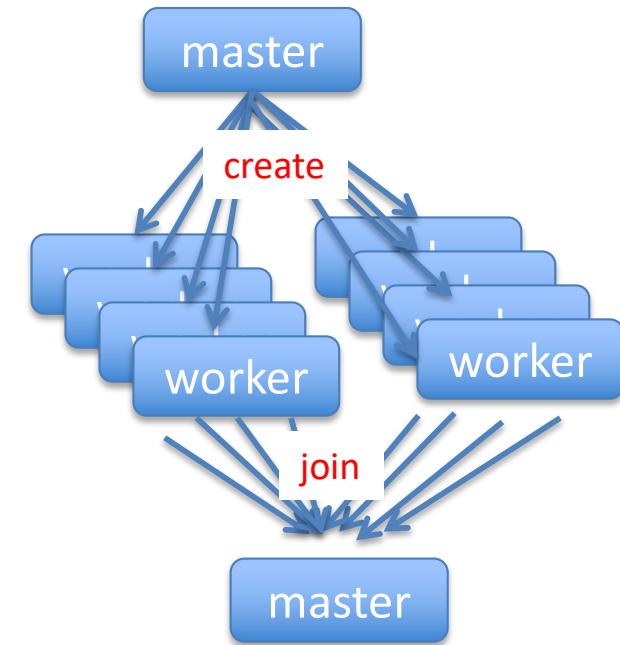
```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

int main() {
    pthread_t threads[8];
    int tn;

    for(tn=0; tn<8; tn++) {
        pthread_create(&threads[tn], NULL, ParFun, NULL);
    }

    for(tn=0; tn<8 ; tn++) {
        pthread_join(threads[tn], NULL);
    }
    return 0;
}
```

- This code creates 8 threads that execute the function “ParFun”.
- What happens to the master while workers are executing?
 - Does master become one of the workers?



Thread Termination

- **Pthread_exit()**
 - A thread returns from its starting routine by default, similar to a process terminating when it reaches to end of main
- **Pthread_cancel()**
 - Thread is cancelled by another thread via this call
- **Pthread_join(. . .)**
 - From Unix specification: “suspends execution of the calling thread until the target thread terminates, unless the target thread has already terminated.”
- After the last thread in a process terminates, the process terminates by calling exit()
- If the entire process is terminated, then all its threads will terminate

“Hello World”

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

int thread_count=16; //accessible by all threads

void *hello(void* rank); //thread function

int main() {
    pthread_t threads[16]; //thread handles

    int tn;
    for(tn=0; tn<16; tn++) {

        pthread_create(&threads[tn] , NULL, hello, (void*) tn);

    }
    for(tn=0; tn<16 ; tn++) {

        pthread_join(threads[tn] , NULL);
    }
    return 0;
}
```

Start function to execute

Arguments to function
e.g. Thread ID

Wait for thread completion

“Hello World”

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

int thread_count=16; //global var, accessible by all threads

void *hello(void* rank); //thread function

int main() {
    pthread_t threads[16]; //thread handles

    int tn;
    for(tn=0; tn<16; tn++) {

        pthread_create(&threads[tn], NULL, hello, (void*)tn);
    }

    pthread_exit(NULL);
}

return 0;
}
```

Prevents the process to die before its threads are done!
main()/master will block and be kept alive to support the threads it created

“Hello World”

```
void *hello(void* id){  
  
    int my_id = (int) id; //typecasting  
  
    printf("Hello from thread %d of %d\n", my_id,  
                           thread_count);  
  
    pthread_exit(0);  
}
```

- By using thread id, different execution for each thread is possible
- How is possible for a thread to access thread_count?

Compiling a Pthread Program

```
gcc -o pth_hello pth_hello.c -lpthread
```

Link the Pthreads library

The Pthreads API is only available on POSIX systems
— Linux, MacOS X, Solaris, HPUX, ...

Because threads share resources:

- Changes made by one thread to shared system resources (such as closing a file) will be seen by all other threads.
- Two pointers having the same value point to the same data.
 - Because of the **shared address space**
- Reading and writing to the same memory locations is possible, and therefore requires **explicit synchronization** by the programmer.

Shared Data

- Variables declared outside of ‘main’ are **global variables**
 - Those variables are shared by all threads
- Object allocated on the **heap** may be shared if pointer is passed as an argument to the thread function

```
char *message = "Hello World!\n";
pthread_create( &thread1, NULL, (void*)&print_fun, (void*)message);
```

- Variables on the **stack** are **private** (locally defined variables)
 - Passing pointer to these around to other threads can cause problems

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.

Mutex (Locks) in Pthreads

```
#include <pthread.h>
pthread_mutex_t mymutex = PTHREAD_MUTEX_INITIALIZER;

//create a mutex
pthread_mutex_init(&mymutex, NULL);

//use it
pthread_mutex_lock(&mymutex);
//mutually excluded code region
pthread_mutex_unlock(&mymutex);

//destroy a mutex
pthread_mutex_destroy(&mymutex);
```

```
pthread_mutex_lock(&our_lock);
counter++
pthread_mutex_unlock(&our_lock);
```

Condition Variables

- While **mutexes** implement synchronization by controlling thread access to data, condition variables allow threads to synchronize based upon the actual value of data.
 - Without condition variables, the programmer would need to have threads continually polling to check (possibly in a critical section) if the condition is met.
- A condition variable is always used in conjunction with a mutex lock.
- Set/query condition variable attributes

See an example usage:

https://hpc-tutorials.llnl.gov posix/example_using_cond_vars/

Condition Variable Code Example

See the handout

1. The main thread creates three threads.
2. Two of those threads increment a "count" variable, while the third thread watches the value of "count".
3. When "count" reaches a predefined limit, the waiting thread is signaled by one of the incrementing threads.
4. The waiting thread "awakens" and then modifies count. The program continues until the incrementing threads reach TCOUNT.
5. The main program prints the final value of count.

Reference link:

https://hpc-tutorials.llnl.gov posix/example_using_cond_vars/

Condition Variables (cont.)

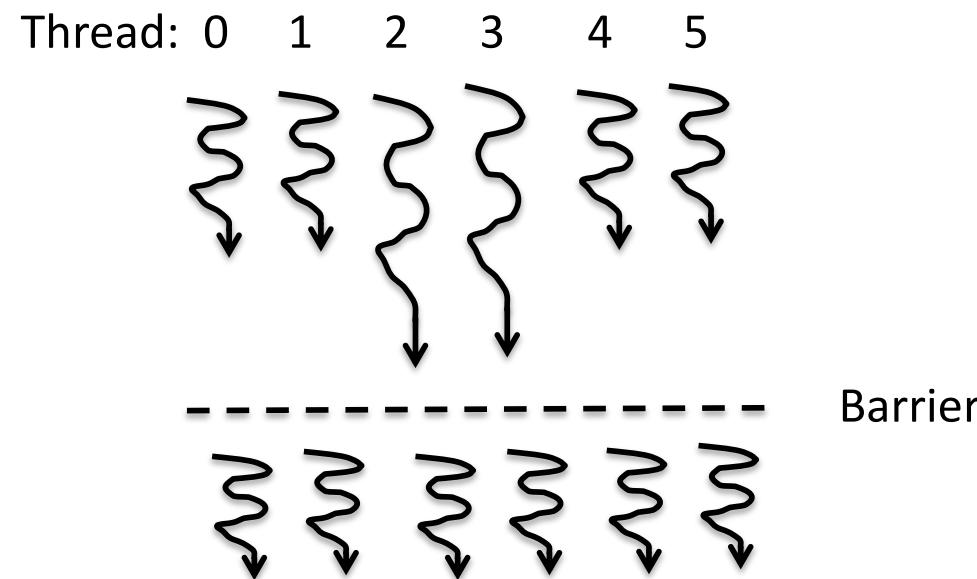
- Main Thread
 - Declare and initialize global data/variables which require synchronization (such as "count")
 - Declare and initialize a condition variable object
 - Declare and initialize an associated mutex
 - Create threads A and B to do work
- Thread A
 - Do work up to the point where a certain condition must occur (such as "count" must reach a specified value)
 - Lock associated mutex and check value of a global variable
 - Call `pthread_cond_wait()` to perform a blocking wait for signal from Thread-B. Note that a call to `pthread_cond_wait()` automatically and atomically unlocks the associated mutex variable so that it can be used by Thread-B.
 - When signaled, wake up. Mutex is automatically and atomically locked.
 - Explicitly unlock mutex
 - Continue
- Thread B
 - Do work
 - Lock associated mutex
 - Change the value of the global variable that Thread-A is waiting upon.
 - Check value of the global Thread-A wait variable. If it fulfills the desired condition, signal Thread-A.
 - Unlock mutex.
 - Continue

Semaphores in Pthreads

- Functions defined in `semaphore.h`:
- A semaphore is represented by a `sem_t` type.
- `sem_init`: for initializing semaphore
- `sem_wait`: for waiting on a semaphore
- `sem_post`: for signaling on a semaphore
- `sem_destroy`: for deallocating a semaphore if you no longer need it

Barriers

- Synchronizing the threads to make sure that they all are at the same point in a program is called a barrier.
- No thread can cross the barrier until all the threads have reached it.



Even though threads 2 and 3 reached barrier, they will wait for others to arrive.
Then all threads cross the barrier point together.

Barriers in Pthreads

- To (dynamically) initialize a barrier, use code similar to this (which sets the number of threads to 3):

```
pthread_barrier_t b;  
pthread_barrier_init(&b,NULL,3);
```

- The second argument specifies an object attribute; using NULL yields the default attributes.
- To wait at a barrier, a process executes:

```
pthread_barrier_wait(&b);
```

- To destroy a barrier

```
pthread_barrier_destroy(&b);
```

Thread Scheduling

- Threads can be scheduled by the operating system and run as independent entities
- Many-to-one and many-to-many models, thread library schedules user-level threads
 - Known as **process-contention scope (PCS)** since scheduling competition is within the process
- Kernel thread scheduled onto available CPU is **system-contention scope (SCS)** – competition among all threads in system

Pthread Scheduling

- API allows specifying either PCS or SCS during thread creation
 - PTHREAD_SCOPE_PROCESS schedules threads using PCS scheduling
 - PTHREAD_SCOPE_SYSTEM schedules threads using SCS scheduling
- Can be limited by OS – Linux and Mac OS X only allow PTHREAD_SCOPE_SYSTEM

Pthread Scheduling API

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5
int main(int argc, char *argv[]) {

    int i, scope, policy;
    pthread_t tid[NUM_THREADS];
    pthread_attr_t attr;

    /* get the default attributes */
    pthread_attr_init(&attr);
    /* first inquire on the current scope */
    if (pthread_attr_getscope(&attr, &scope) != 0)
        fprintf(stderr, "Unable to get scheduling scope\n");
    else {
        if (scope == PTHREAD_SCOPE_PROCESS)
            printf("PTHREAD_SCOPE_PROCESS");
        else if (scope == PTHREAD_SCOPE_SYSTEM)
            printf("PTHREAD_SCOPE_SYSTEM");
        else
            fprintf(stderr, "Illegal scope value.\n");
    }
}
```

Pthread Scheduling API (cont.)

```
/* get the current scheduling policy */
if (pthread_attr_getschedpolicy(&attr, &policy) != 0)
    fprintf(stderr, "Unable to get policy.\n");
else {
    if (policy == SCHED_OTHER) printf("SCHED OTHER\n");
    else if (policy == SCHED_RR) printf("SCHED RR\n");
    else if (policy == SCHED_FIFO) printf("SCHED FIFO\n");
}

/* set the scheduling policy - FIFO, RR, or OTHER */
if (pthread_attr_setschedpolicy(&attr, SCHED_FIFO) != 0)
    fprintf(stderr, "Unable to set policy.\n");
...
```

Pthread Scheduling API (cont.)

```
/* set the scheduling algorithm to PCS or SCS */
pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);
/* create the threads */
for (i = 0; i < NUM_THREADS; i++)
    pthread_create(&tid[i], &attr, run_method, NULL);
/* now join on each thread */
for (i = 0; i < NUM_THREADS; i++)
    pthread_join(tid[i], NULL);
}
/* Each thread will begin control in this function */
void *run_method(void *param)
{
    /* do some work ... */
    pthread_exit(0);
}
```

Summary of Pthreads

- Pthreads are user-level threads for POSIX systems
 - Familiar language for most programmers, particularly for systems people
 - Ability to shared data is convenient
 - Various supports for synchronization
- Pthread Tutorial
 - https://hpc-tutorials.llnl.gov posix/example_using_cond_vars/
- Reading from Book
 - 4.1, 4.3, 4.6 , 4.7
- Acknowledgements
 - These slides are adapted from
 - Öznur Özkasap (Koç University)
 - Operating System and Concepts (9th edition) Wiley
 - <https://computing.llnl.gov/tutorials/pthreads/>

Processes vs Threads

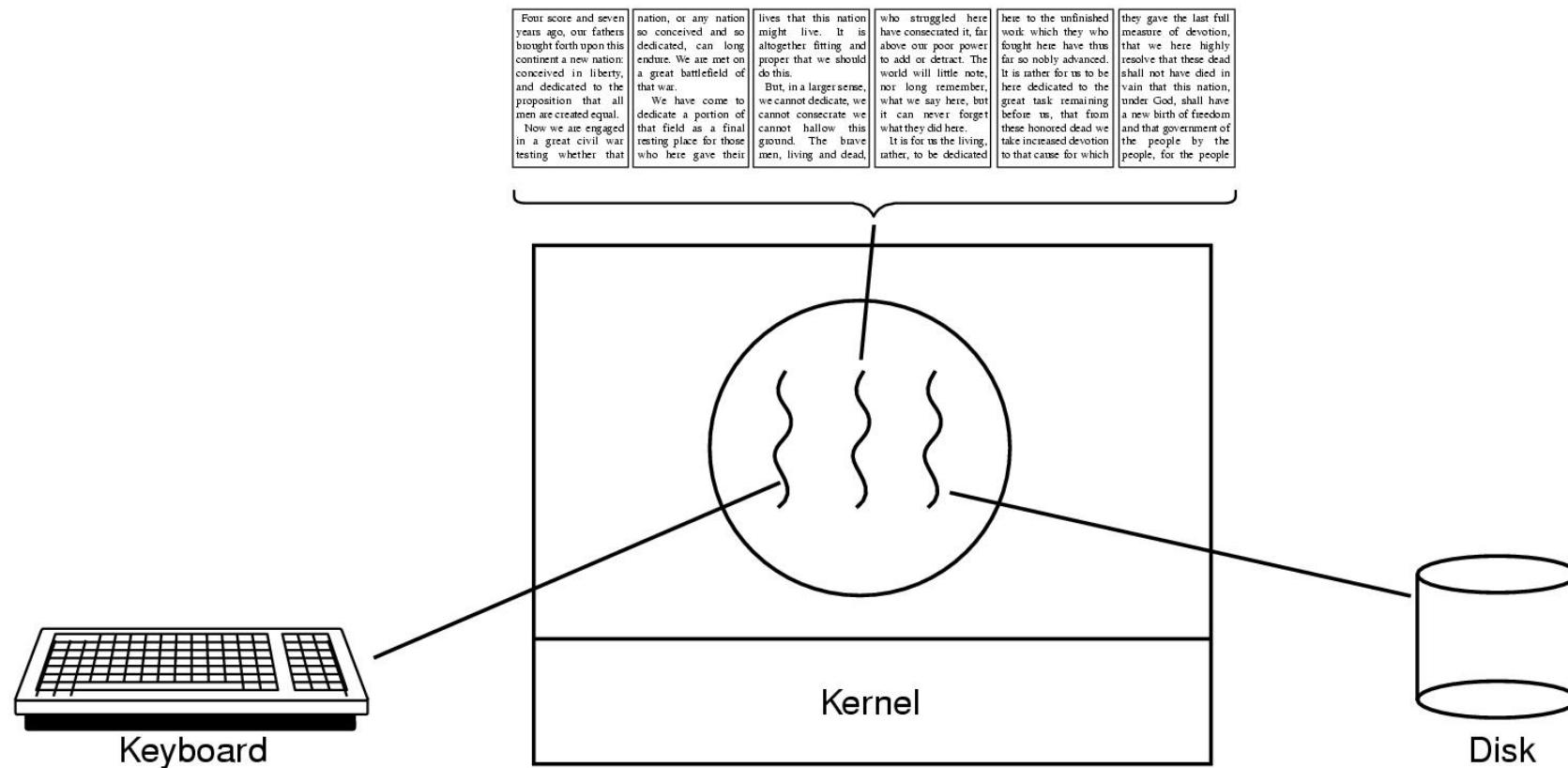
Didem Unat
Lecture 13
COMP304 - Operating Systems (OS)

Threads <= Processes

- A process is an executing program with at least one thread of control
- A process can contain multiple threads of control
 - Threads run within application in parallel (concurrently)
- Process creation is relatively **heavy-weight** while thread creation is **light-weight**
- Modern kernels are generally multithreaded
- Reading from textbook
 - Chapter 4.1, 4.3, 4.6 , 4.7

Thread Usage - Example

- An editor process with three threads

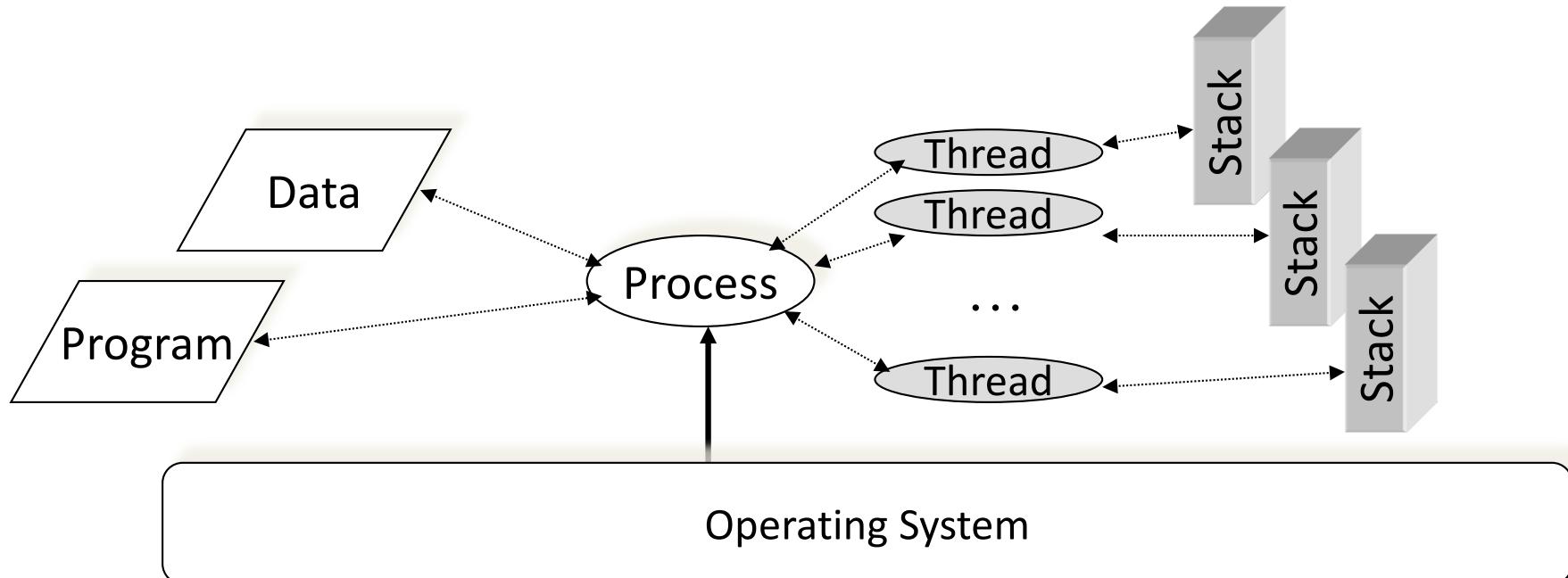


Why use threads?

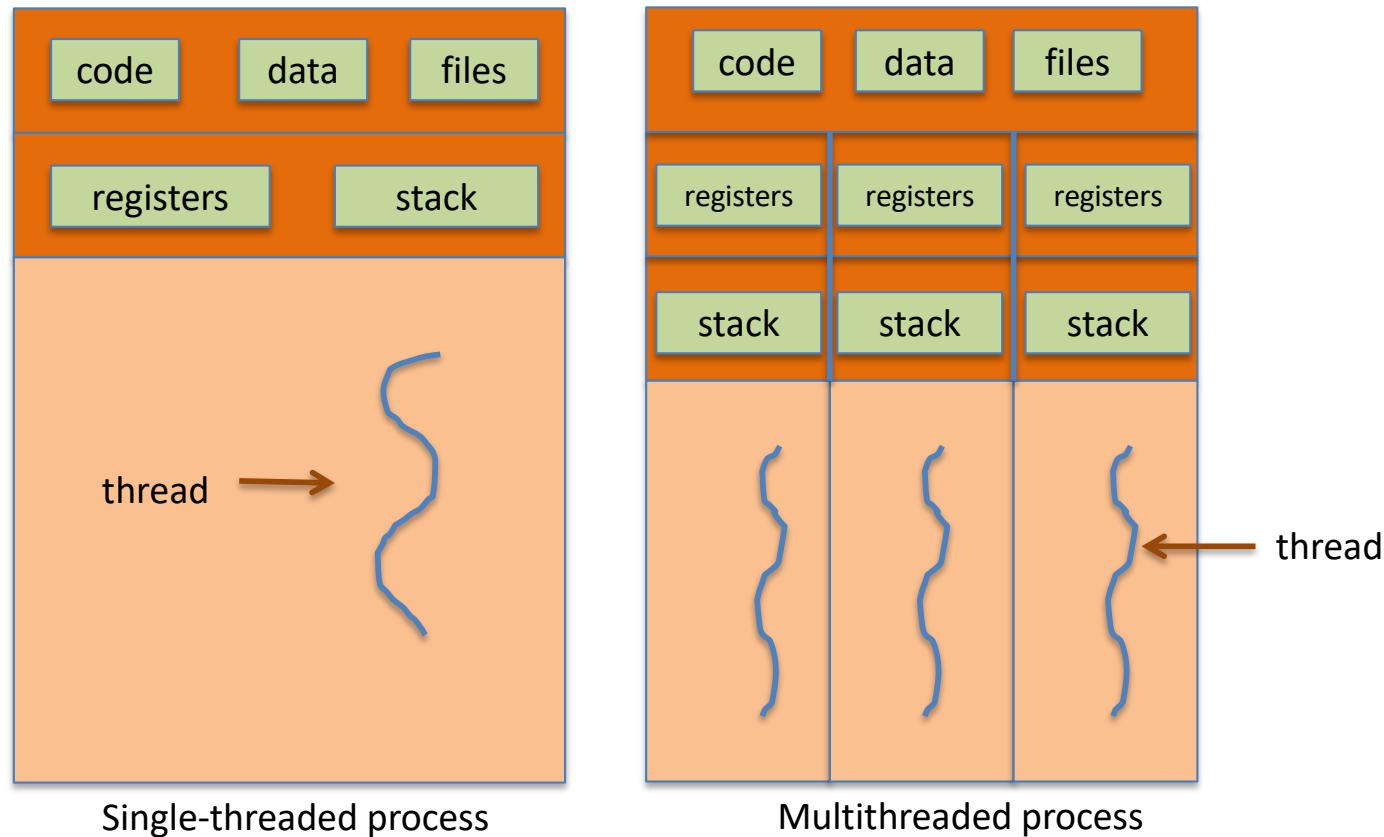
- **Responsiveness** – may allow continued execution if part of process is blocked, especially important for user interactivity
- **Resource Sharing** – threads share resources of a process, easier than shared memory or message passing communication between processes
- **Economy** – cheaper than process creation, thread switching lower overhead than context switching in some systems

Process and Thread

- **Process** is an infrastructure in which execution takes place – address space + resources
- **Thread** is a program in execution within a process context – each thread has its own stack



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

Thread States

- Like a process, a thread can be in one of 3 states: ready, blocked (waiting), running
- A thread may wait:
 - For some external event to occur (such as I/O completion)
 - For some other thread to unblock it.
- When a program runs, it starts as a single-threaded process
 - Then it can create other threads
 - Typically the first thread is referred as the **master** thread and the others are **worker** threads

Thread Libraries

- A **thread library** provides a programmer with API for creating and managing threads
- Two primary ways of implementing
 - Library entirely in user space
 - Kernel-level library supported by the OS

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

Java Threads

- Java threads are managed by the JVM
- Typically implemented using the thread model provided by underlying OS
- Java threads may be created by:
 - Extending Thread class
 - Implementing the Runnable interface

```
public interface Runnable
{
    public abstract void run();
}
```

Java Multithreaded Program

```
class Sum
{
    private int sum;

    public int getSum() {
        return sum;
    }

    public void setSum(int sum) {
        this.sum = sum;
    }
}

class Summation implements Runnable
{
    private int upper;
    private Sum sumValue;

    public Summation(int upper, Sum sumValue) {
        this.upper = upper;
        this.sumValue = sumValue;
    }

    public void run() {
        int sum = 0;
        for (int i = 0; i <= upper; i++)
            sum += i;
        sumValue.setSum(sum);
    }
}
```

Creates java threads

This method is executed by a thread when it is created

Java Multithreaded Program (cont.)

```
public class Driver
{
    public static void main(String[] args) {
        if (args.length > 0) {
            if (Integer.parseInt(args[0]) < 0)
                System.err.println(args[0] + " must be >= 0.");
            else {
                Sum sumObject = new Sum();
                int upper = Integer.parseInt(args[0]);
                Thread thrd = new Thread(new Summation(upper, sumObject));
                thrd.start(); Thread start calls run() method
                try {
                    thrd.join();
                    System.out.println
                        ("The sum of "+upper+" is "+sumObject.getSum());
                } catch (InterruptedException ie) { }
            }
        }
        else
            System.err.println("Usage: Summation <integer value>"); }
    }
```

Thread joins with master thread

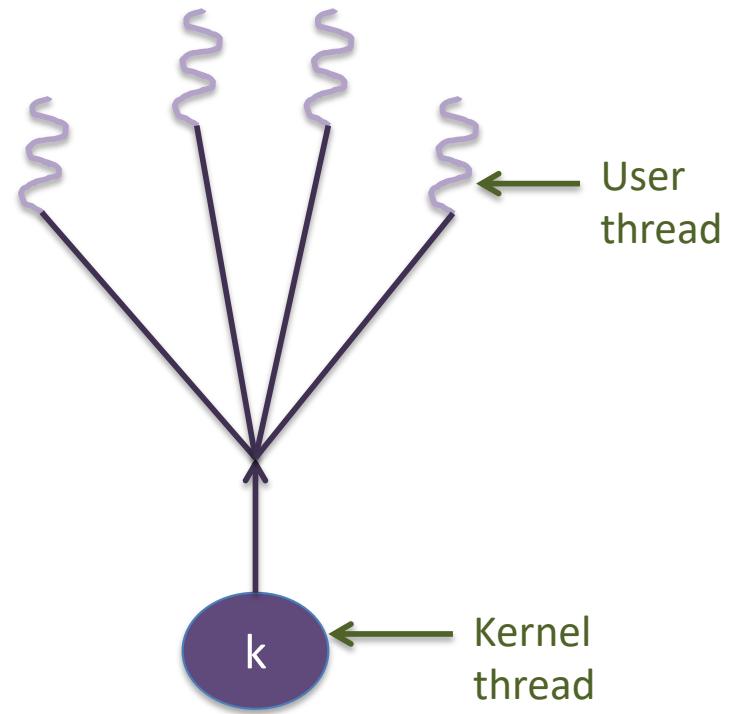
- How can threads share data?

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

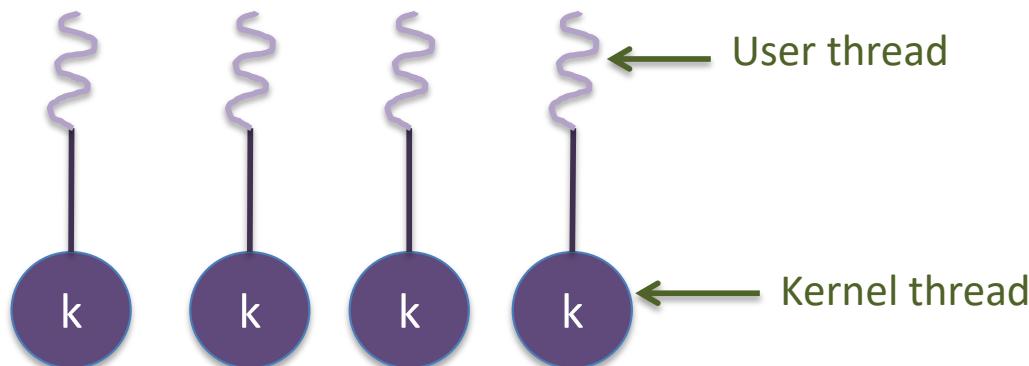
Many-to-One

- Many user-level threads mapped to single kernel thread.
- Used on systems that do not support kernel threads.
- Examples:
 - Solaris Green Threads
 - GNU Portable Threads
 - Not common anymore
- Multiple threads may not run in parallel on multicore system because only one may be in kernel at a time
- Entire process will block if a thread makes a blocking system call



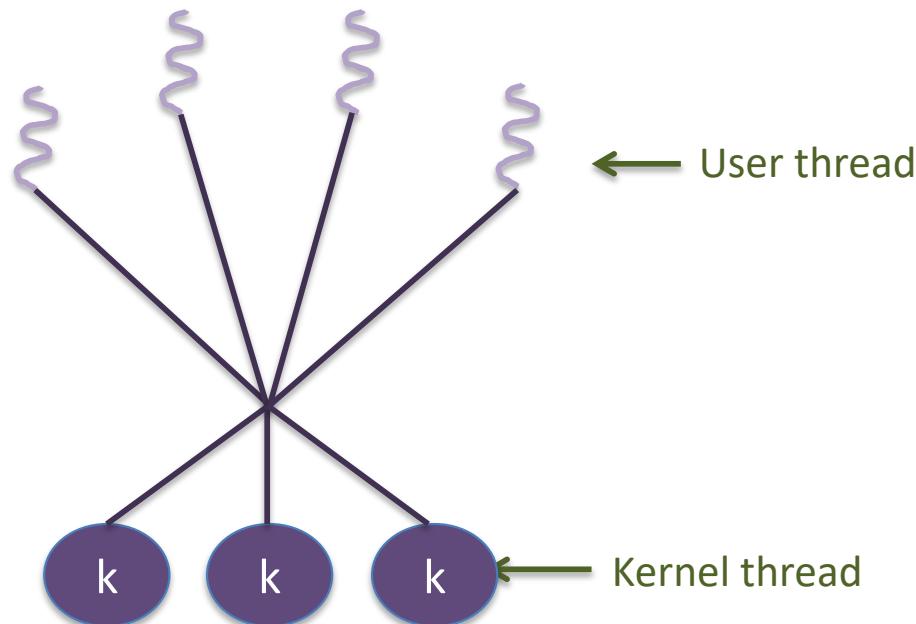
One-to-One

- Each user-level thread maps to a kernel thread
 - Creating a user-level thread creates a kernel thread
- More concurrency than many-to-one
- Number of threads per process sometimes restricted due to overhead
- Examples
 - Windows NT/XP/2000, Linux, Solaris 9 and later



Many-to-Many Model

- Allows many user level threads to be mapped to many kernel threads (usually fewer kernel threads)
 - Allows the operating system to create a sufficient number of kernel threads
- Solaris prior to version 9
- Windows NT/2000 with the *ThreadFiber* package



Exercise

- Which of the following components of program state are shared across threads?
 - Register values
 - Heap memory
 - Stack
 - Global variables
 - Program counter
 - Scheduling properties (e.g. policy, priority)

Question

- A system with two dual-core processors has four processors available for scheduling.
 - A CPU-intensive application is running on this system.
 - All input is performed at program start-up, when a single file must be opened.
 - Similarly, all output is performed just before the program terminates, when the program results must be written to a single file.
 - Between startup and termination, the program is entirely CPU-bound. Your task is to improve the performance of this application by multithreading it.
 - The application runs on a system that uses the one-to-one threading model (each user thread maps to a kernel thread).
- How many threads will you create to perform the input and output? Explain.
- How many threads will you create for the CPU-intensive portion of the application? Explain.

Answer

- It only makes sense to create as many threads as there are blocking system calls, as the threads will be blocking. Creating additional threads provides no benefit. Thus, it makes sense to create a single thread for input and a single thread for output unless you can also perform parallel I/O
- Four. There should be as many threads as there are processing cores. Fewer would be a waste of processing resources, and any number > 4 would be unable to run simultaneously.

Threading Issues

- Semantics of **fork()** and **exec()** system calls
- Signal handling
 - Synchronous and asynchronous
- Thread cancellation of target thread
 - Asynchronous or deferred
- Thread-local storage
- Scheduler Activations

Semantics of fork() and exec()

- Semantics of fork and exec system calls change in a multithreaded program
- Threads and fork(): **think twice before mixing them.**
- Does **fork()** duplicate only the calling thread or all threads?
 - Some UNIXes have two versions of fork
 - Version 1: The child has only one thread
 - Version 2: The child has all the threads
- **Exec()** usually works as normal – replace the entire process including all threads
 - Call exec right after fork()

Signal Handling

- **Signals** are used in UNIX systems to notify a process that a particular event has occurred.
 - Synchronous : illegal memory access, division by zero
 - Asynchronous: key strokes to cancel a process
- In a single-threaded program, a signal is delivered to the process
- In a multi-threaded program, where should a signal be delivered?
 - Deliver the signal to the thread to which the signal applies
 - Deliver the signal to every thread in the process
 - Deliver the signal to certain threads in the process
 - Assign a specific thread to receive all signals for the process
- Depends on the type of signal generated.

Thread Cancellation

- Terminating a thread before it has finished
- Thread to be canceled is called **target thread**
- Two general approaches:
 - **Asynchronous cancellation** terminates the target thread immediately
 - **Deferred cancellation** allows the target thread to periodically check if it should be cancelled
- Pthread code to create and cancel a thread:

```
pthread_t tid;  
  
/* create the thread */  
pthread_create(&tid, 0, worker, NULL);  
  
.  
. . .  
  
/* cancel the thread */  
pthread_cancel(tid);
```

Thread-Local Storage

- **Thread-local storage (TLS)** allows each thread to have its own copy of data
- Useful when you do not have control over the thread creation process (i.e., when using a thread pool)
- Different from local variables
 - Local variables visible only during single function invocation
 - TLS visible across function invocations
- Similar to **static** data
 - TLS is unique to each thread

OS Example: Windows Threads

- Implements the one-to-one mapping, kernel-level
- Each thread contains
 - A thread id
 - Register set representing state of processor
 - Separate user and kernel stacks for when thread runs in user mode or kernel mode
 - Private data storage area used by run-time libraries and dynamic link libraries (DLLs)
- The register set, stacks, and private storage area are known as the **context** of the thread
- The primary data structures of a thread include:
 - ETHREAD (executive thread block) – includes pointer to process to which thread belongs and to KTHREAD, in kernel space
 - KTHREAD (kernel thread block) – scheduling and synchronization info, kernel-mode stack, pointer to TEB, in kernel space
 - TEB (thread environment block) – thread id, user-mode stack, thread-local storage, in user space

OS Example: Linux Threads

- Linux refers to them as ***tasks*** rather than ***threads***
- Thread creation is done through **clone()** system call
- **clone()** allows a child task to share the address space of the parent task (process)
 - Different than fork()

flag	meaning
CLONE_FS	File-system information is shared.
CLONE_VM	The same memory space is shared.
CLONE_SIGHAND	Signal handlers are shared.
CLONE_FILES	The set of open files is shared.

- **struct task_struct** points to thread data structures

Question

- Linux does not distinguish between processes and threads. However, many operating systems—such as Windows XP and Solaris—treat processes and threads differently. Typically, such systems use a notation wherein the data structure for a process contains pointers to the separate threads belonging to the process. Contrast these two approaches for modeling processes and threads within the kernel.
- Answer:
- On one hand, in systems where processes and threads are considered as similar entities, some of the operating system code could be simplified. A scheduler, for instance, can consider the different processes and threads on an equal footing without requiring special code to examine the threads associated with a process during every scheduling step. On the other hand, this uniformity could make it harder to impose process-wide resource constraints in a direct manner. Instead, some extra complexity is required to identify which threads correspond to which process and perform the relevant accounting tasks.

Reading

- From text book
 - Read Chapter 4.1, 4.3, 4.6 , 4.7
- Next lecture on POSIX Threads
- Acknowledgments
 - These slides are adapted from
 - Öznur Özkasap (Koç University)
 - Operating System and Concepts (9th edition) Wiley

Synchronization-IV Monitors

Didem Unat

Lecture 12

COMP304 - Operating Systems (OS)

Semaphores

- We want to be able to write more complex constructs so 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 resources
- Binary semaphore – integer value can range only between 0 and 1; can be simpler to implement
 - This is the same as mutex locks

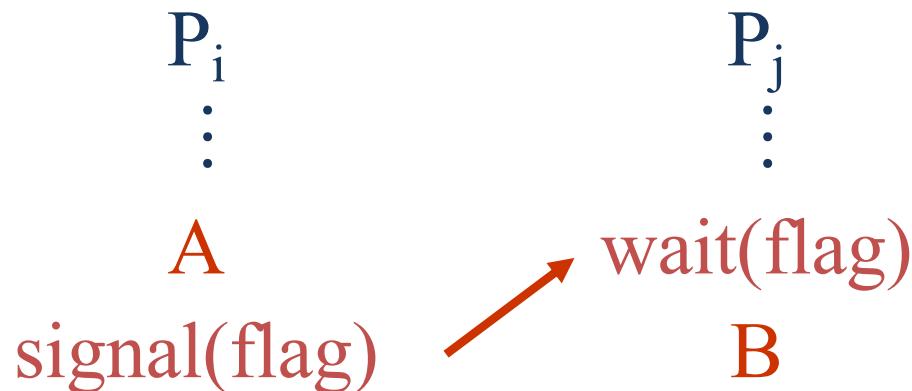
Semaphore as a general synchronization tool

- Semaphores can be used to force synchronization (precedence) if the **preceder** 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 executed in P_i
- Use semaphore flag initialized to **0**

Code:



Blocking Semaphores

```
typedef struct {
    int      value;
    struct process *list; /* list of processes waiting on S */
} SEMAPHORE;
```

```
SEMAPHORE s;
wait(s) {
    s.value = s.value - 1;
    if ( s.value < 0 ) {
        add this process to s.list;
        block ();
    }
}
```

```
SEMAPHORE s;
signal(s) {
    s.value = s.value + 1;
    if ( s.value <= 0 ) {
        remove a process P from s.L;
        wakeup(P);
    }
}
```

Block – place the process invoking the operation on the appropriate waiting queue if semaphore is not available

Wakeup – Wakes up one of the blocked processes upon getting a signal and places the process to ready queue

Semaphores vs Locks

- Semaphores: processes that are blocked at the level of program logic are placed on queues, rather than busy-waiting
- Locks: Busy-waiting may be used for the mutual exclusion
 - But these should be very short critical sections
- Unlike locks, counting semaphores can take an integer value representing total number of resources

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**

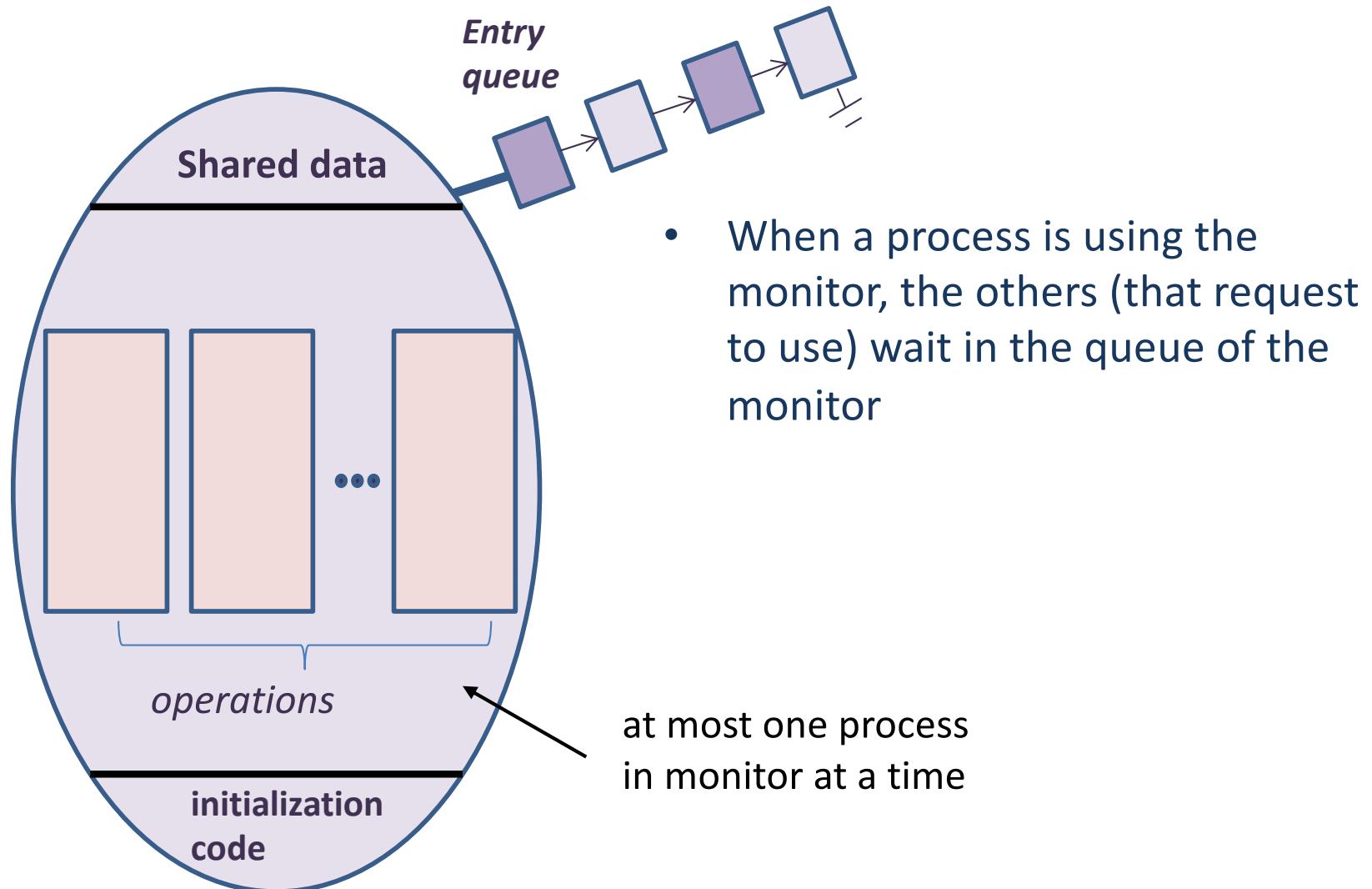
Monitors

- 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

Schematic View of a Monitor



Example: Shared Balance

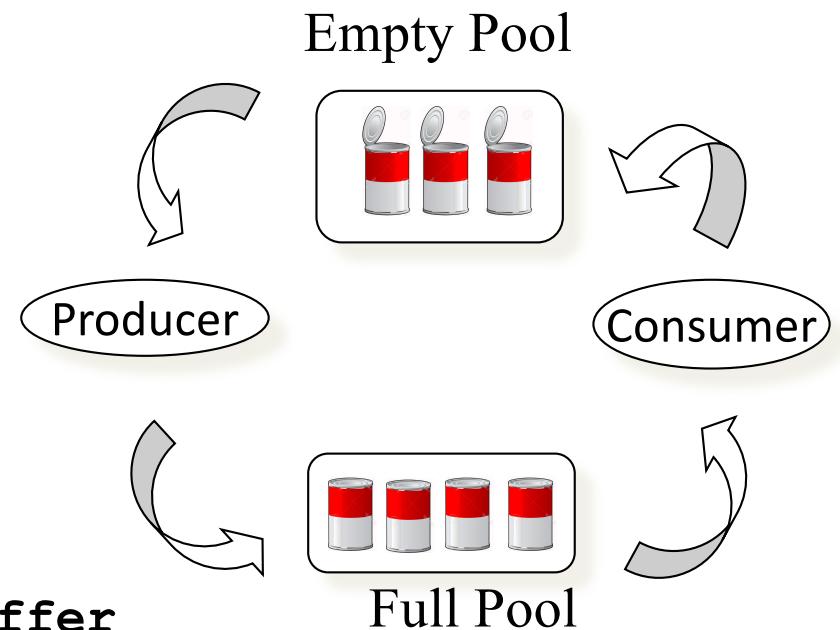
```
monitor sharedBalance {  
  
    double balance;  
  
    void deposit(double amount) {  
        balance += amount;  
    }  
    void withdraw(double amount) {  
        balance -= amount;  
    }  
    . . .  
}
```

- Balance is a **shared** variable
- Deposit and withdraw are **procedures**
- Processes do not directly read/write into the shared variables
but access them via these procedures

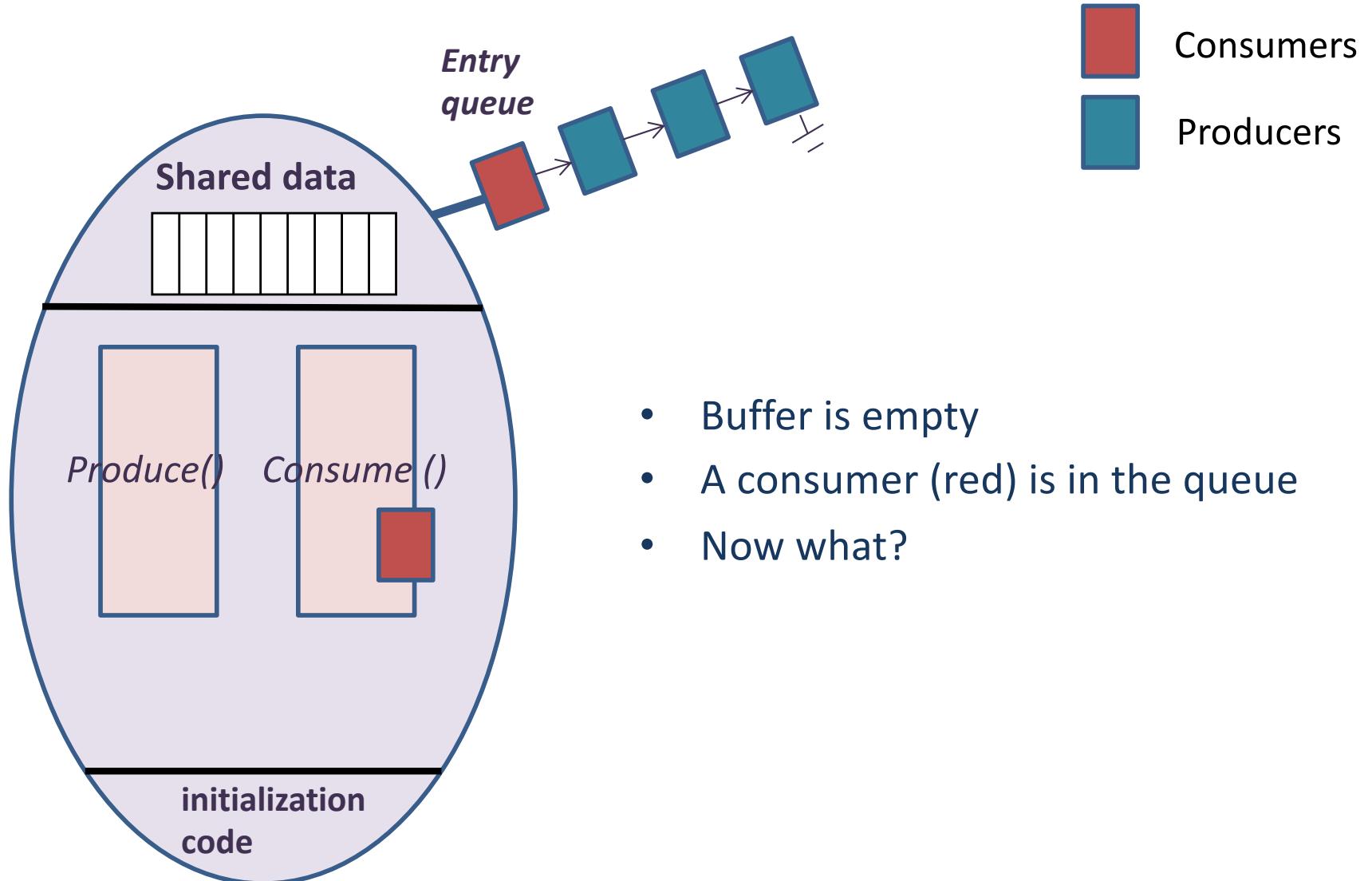
Producer-Consumer Problem

- Buffer size: buffer can hold **n** items

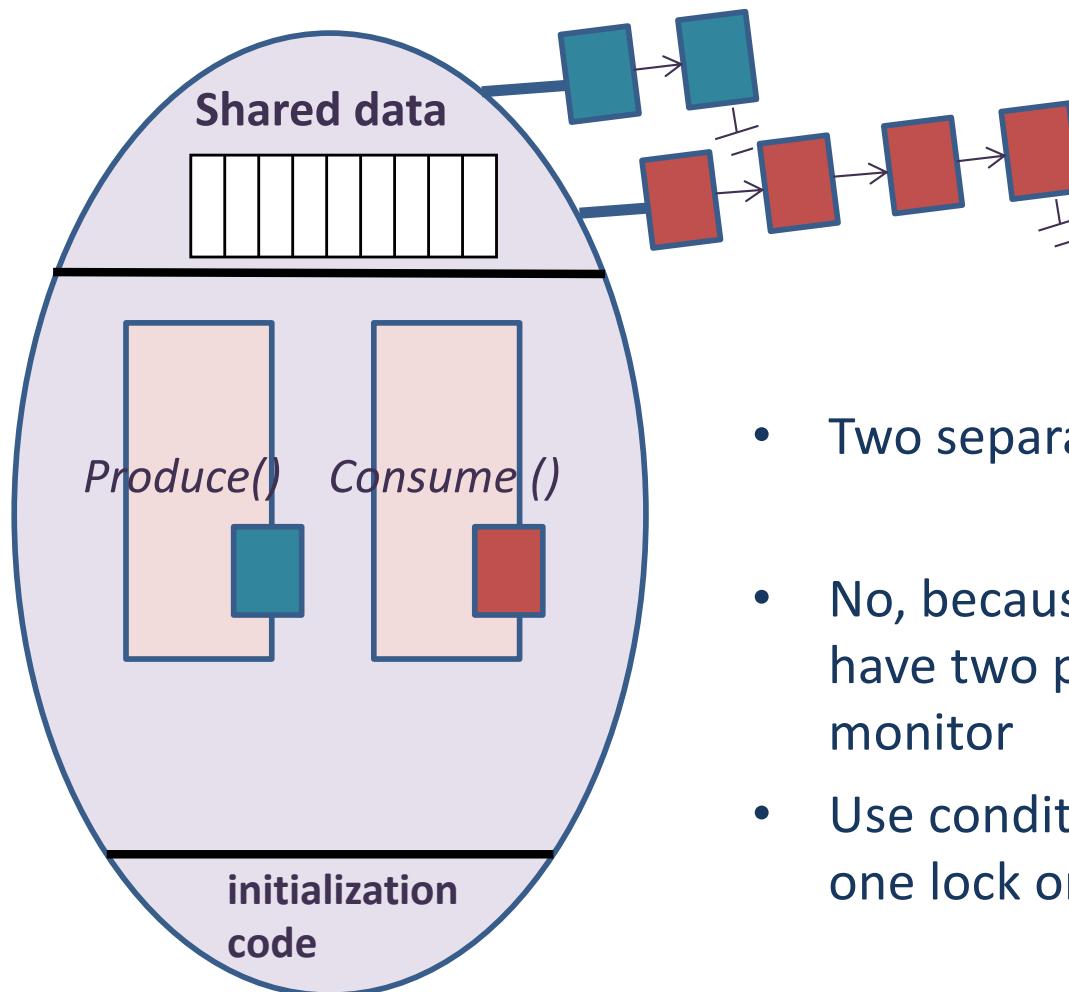
```
monitor ProducerConsumer {  
  
    item buffer[N];  
  
    void produce(item x) {  
        //add an item to the buffer  
    }  
    void consume(item *x) {  
        //remove an item from the buffer  
    }  
    . . .  
}
```



Example: Producer-Consumer Problem



Example: Producer-Consumer Problem

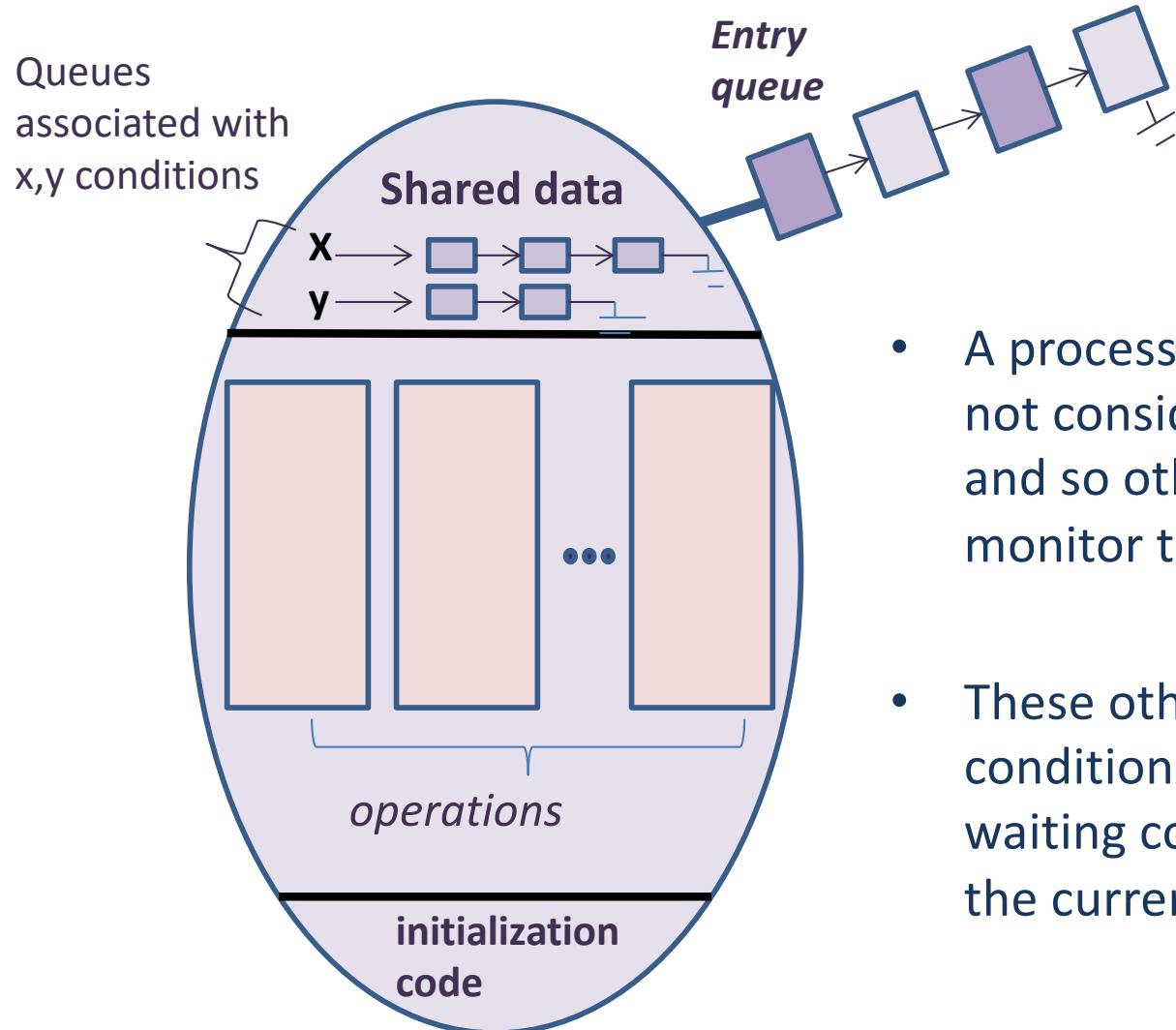


- Two separate queues?
- No, because then we would have two processes in the monitor
- Use condition variables sharing one lock on the queue

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 waiting for a condition 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

Producer and Consumer with Monitors

```
Monitor ProducerConsumer {
    item buffer[N];
    condition not_full, not_empty;

    produce(item x) {
        if (array "buffer" is full)// determined by a count
            wait(not_full);
        insert "x" in array "buffer"
        signal(not_empty);
    }

    consume(item *x) {
        if (array "buffer" is empty) //determined maybe by a count
            wait(not_empty);
        *x = get item from array "buffer"
        signal(not_full);
    }
}
```

Producer and Consumer with Monitors

```
Monitor ProducerConsumer {
    item buffer[N];
    condition not_full, not_empty;

    produce(item x) {
        if (array "buffer" is full)// determined by a count
            wait(not_full);
        insert "x" in array "buffer"
        signal(not_empty);
    }

    consume(item *x) {
        if (array "buffer" is empty) //determined maybe by a count
            wait(not_empty);
        *x = get item from array "buffer"
        signal(not_full);
    }
}
```

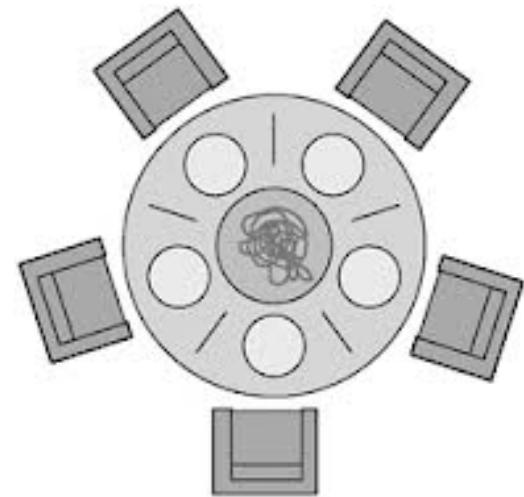
EnterMonitor

ExitMonitor

- Monitor inserts the lock operations before at the entry of produce/consume and releases the lock at the exit of produce/consume
- That's why we don't need a separate mutex lock for multiple producers (or consumers)

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 to eat
 - Pick one chopstick at a time
- While a philosopher is thinking, she drops the chopsticks on the table



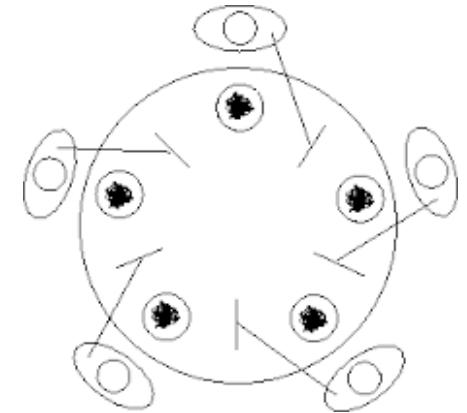
Dining Philosophers Problem

- Shared data

semaphore chopstick[5]; Initially all values are 1

Philosopher i :

```
while (true) {
    wait(chopstick[i])
    wait(chopstick[(i+1) % 5])
    // eat
    signal(chopstick[i]);
    signal(chopstick[(i+1) % 5]);
    // think
}
```



Deadlock
may occur!

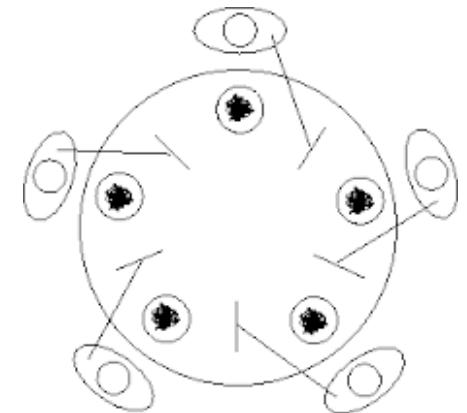
Dining Philosophers Problem

- Shared data

semaphore chopstick[5]; Initially all values are 1

Philosopher i :

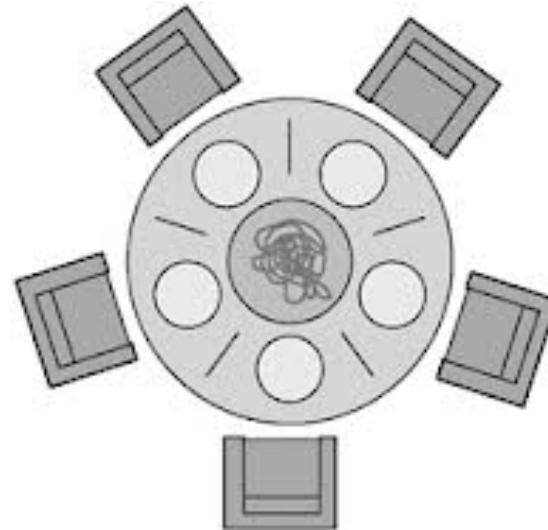
```
while (true) {
    wait(chopstick[i])
    wait(chopstick[(i+1) % 5])
    // eat
    signal(chopstick[i]);
    signal(chopstick[(i+1) % 5]);
    // think
}
```



Starvation
may occur!

Several Solutions

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



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

Monitor Solution to Dining Philosophers

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

`dp.pickup (i)`

EAT

`dp.putdown (i)`

THINK

Monitor Solution to Dining Philosophers

- 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 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);
    }
}
```

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

Language Support

- Java, C# and several other languages have support for Monitors
 - Here is a Java example

```
// To make a method synchronized, simply add the synchronized  
// keyword to its declaration:
```

```
public class SynchronizedCounter {  
    private int c = 0;  
    public synchronized void increment() {  
        c++;  
    }  
    public synchronized void decrement() {  
        c--;  
    }  
    public synchronized int value() {  
        return c;  
    }  
}
```

Summary of Monitors

- A **monitor** is a synchronization construct that allows processes
 - To have both mutual exclusion and
 - The ability to wait (block) for a certain condition to become true.
- Monitors also have a mechanism for signaling other processes that their condition has been met.
- A monitor consists of a mutex object and **condition variables**, procedures to access them

Summary of Synchronization

- Mutual exclusion is required to ensure no two concurrent processes are in their critical section at the same time
 - To prevent race conditions (corruption of shared data)
 - First identified and solved by Dijkstra in 1965
- Hardware solutions
 - Test-and-Test
 - Compare-and-Swap
- Software solutions (only provide higher-level abstractions to their hardware solutions)
 - Locks (busy-wait)
 - Semaphores (blocking, counting semaphores)
 - Monitors (high level language constructs)

OS Support for Synchronization

- Windows
 - Uses **spinlocks** on multiprocessor systems.
 - Also provides **dispatcher objects** which may act as mutexes and semaphores.
 - Dispatcher objects may also provide **events**. An event acts much like a **condition variable**.
- Linux
 - Disables interrupts to implement short critical sections
 - Provides semaphores and spinlocks
 - Pthreads: provides mutex locks and condition variables

Reading

- Read Chapter 6
- Wikipedia Article on Monitors
 - [http://en.wikipedia.org/wiki/Monitor %28synchronization%29](http://en.wikipedia.org/wiki/Monitor_%28synchronization%29)
- Acknowledgments
 - These slides are adapted from
 - Öznur Özkasap (Koç University)
 - Operating System and Concepts (9th edition) Wiley
 - Jerry Breecher (Worcester Polytechnic Institute)
 - Mark Zbikowski and Gary Kimura (Univ. of Washington)

Synchronization-III

Semaphores

Didem Unat

Lecture 11

COMP304 - Operating Systems (OS)

Mutex Locks

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

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

- Software interface for locks
- Calls to **acquire()** and **release()** must be atomic
 - implemented via hardware atomic instructions
- But this solution requires **busy waiting**
 - And also called a **spinlock**

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.)

Critical Section for n Processes

- Shared semaphore:

semaphore mutex = 1; //initial value

- Process Pi:

```
do {  
    wait(mutex)  
        critical section  
    signal(mutex)  
        remainder section  
} while (true);
```

Shared Account Balance Example

```
semaphore mutex = 1;
```

```
proc_0() {  
    . . .  
    /* Enter the CS */  
wait(mutex);  
    balance += amount;  
signal(mutex);  
    . . .  
}
```

```
proc_1() {  
    . . .  
    /* Enter the CS */  
wait(mutex);  
    balance -= amount;  
signal(mutex);  
    . . .  
}
```

//CS stands for critical section

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

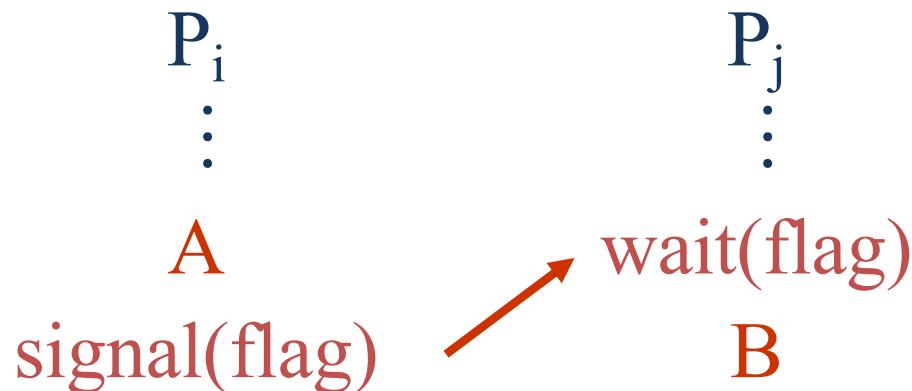
Semaphore as a general synchronization tool

- Semaphores can be used to force synchronization (precedence) if the **preceder** 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:



Classical Problems of Synchronization

- Bounded-Buffer Problem
- Readers and Writers Problem
- Dining Philosophers Problem

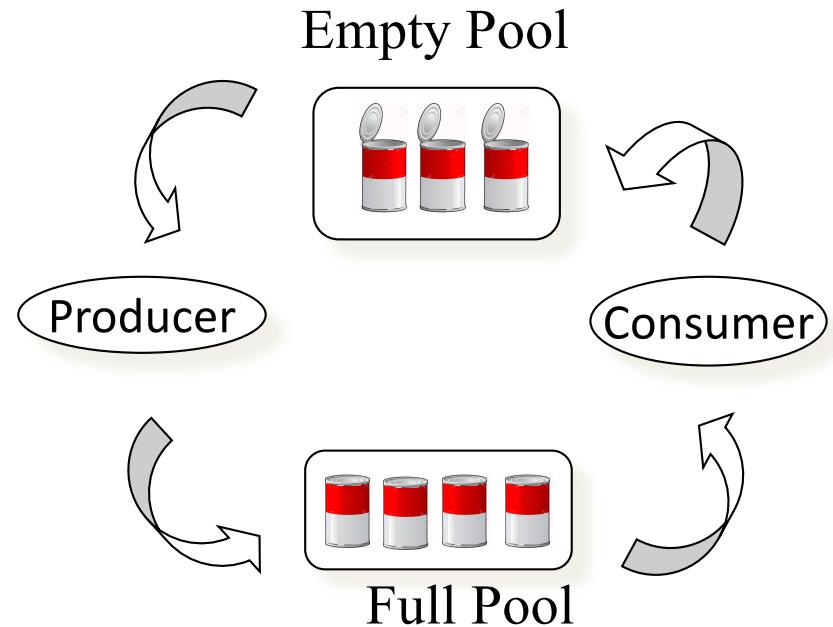
These classical problems are used for testing newly proposed synchronization methods.

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

- Empty and Full are counting semaphores

```
producer:  
do {  
    /* produce an item in nextp */  
    wait (empty);           /* Do action */  
  
    /* add nextp to buffer */  
  
    signal (empty);  
} while(TRUE);  
  
consumer:  
do {  
    wait (full);  
  
    /* remove an item from buffer to nextc */  
  
    signal (full);  
    /* consume an item in nextc */  
} while(TRUE);
```

Signals are wrong!
Producer (consumer)
needs to wake up the
consumer (producer)!

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);
```

Does this work for multiple producers and consumers?

```
consumer:  
do {  
    wait (full);  
  
    /* remove an item from buffer to nextc */  
  
    signal (empty);  
  
    /* consume an item in nextc */  
}  
} while (TRUE);
```

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

Bounded-Buffer Problem

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

```
producer:  
do {  
    /* produce an item in nextp */  
    wait (empty);           /* Do action */  
    wait (mutex);          /* Buffer guard*/  
  
    /* add nextp to buffer */  
  
    signal (mutex);  
    signal (full);  
  
} while(TRUE);  
  
consumer:  
do {  
    wait (full);  
    wait (mutex);  
  
    /* remove an item from buffer to nextc */  
  
    signal (mutex);  
    signal (empty);  
  
    /* consume an item in nextc */  
} while(TRUE);
```

We need a mutex in addition to empty and full semaphores because multiple producers (hence consumers) are operating on the same buffer

Semaphores

- **Spinlocks** (mutexes) are useful in a system since no context switch is required
- A disadvantage of mutex solutions so far:
 - they all require **busy waiting**.
- To overcome busy waiting → blocking a process

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

Blocking Semaphores

```
typedef struct {
    int      value;
    struct process *list; /* list of processes waiting on S */
} SEMAPHORE;
```

```
SEMAPHORE s;
wait(s) {
    s.value = s.value - 1;
    if ( s.value < 0 ) {
        add this process to s.list;
        block ();
    }
}
```

```
SEMAPHORE s;
signal(s) {
    s.value = s.value + 1;
    if ( s.value <= 0 ) {
        remove a process P from s.list;
        wakeup(P);
    }
}
```

Block – place the process invoking the operation on the appropriate waiting queue if semaphore is not available

Wakeup – Wakes up one of the blocked processes upon getting a signal and places the process to ready queue

Deadlock and Starvation

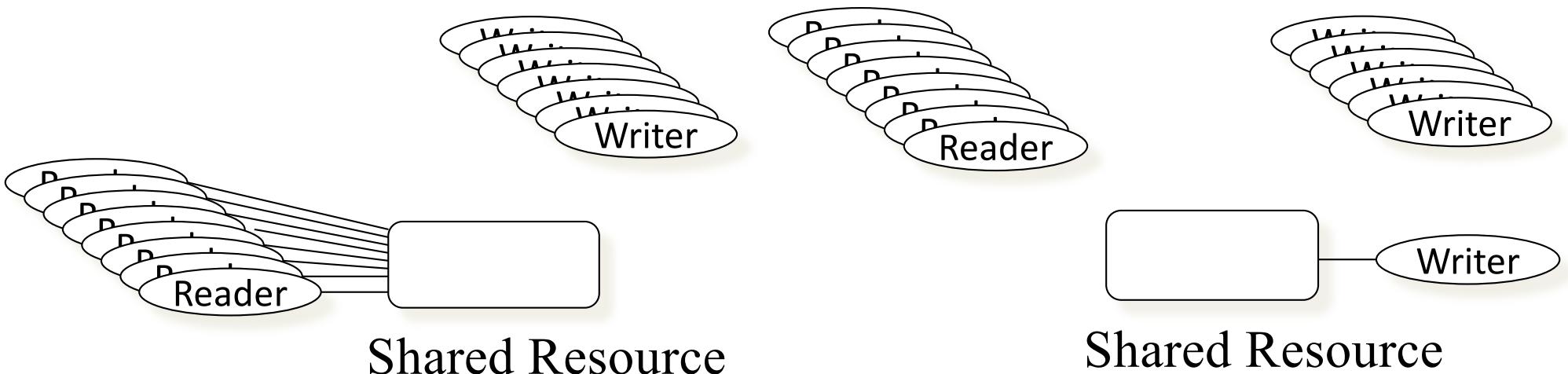
- **Deadlock** – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes.
- Let S and Q be two semaphores initialized to 1

P_0	P_1
wait(S);	wait(Q);
wait(Q);	wait(S);
:	:
signal(S);	signal(Q);
signal(Q)	signal(S);

- **Starvation** – indefinite blocking. A process may never be removed from the semaphore queue in which it is suspended.
- May occur if we add and remove processes from the list in LIFO order or based on priority.

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.



Readers/Writers Problem

Locks:

- are **shared** (for the readers) and **exclusive** (for the writer).

Two possible (contradictory) guidelines can be used:

- No reader is kept waiting unless a writer holds the lock (the readers have precedence).
 - First readers problem
- If a writer is waiting for access, no new reader gains access (writer has precedence).
 - Second readers problem

Can starvation occur on either of these rules?

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);
```

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 );  
  
} while(TRUE);
```



A reader might be
reading while a writer is
writing at the same
time or two writers can
perform their writes!
/*last reader frees writer */

How can that happen?

First-Readers Problem

Correct Implementation, 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 );  
    readcount = readcount + 1;  
    if readcount == 1 then  
        wait( rdwrt );  
    signal( cntmutex );  
  
    /* reading is performed */  
  
    wait( cntmutex );  
    readcount = readcount - 1;  
    if readcount == 0 then  
        signal( rdwrt );  
    signal( cntmutex );  
  
} while(TRUE);
```

/* Allow 1 reader in entry*/

Assume there are two readers left (r1 and r2). r1 may be interrupted before it checks `readcount == 0`. The second reader(r2) decrements the `readcount` from 1 to 0 and both will check `readcount==0` and get true, then both send a `signal(wrt)` allowing two writers or one reader and one writer in the system.

Question

- Many events on campus take place in SGKM. Assume that the show room can only allow N many people. As soon as N audience are in the room, the staff will not accept another incoming audience until an existing audience leaves the event.
- Explain how semaphores can be used by the staff to limit the number of people in the SGKM event room.
- Show your pseudo-code.

Solution

- Semaphore is initialized to the number of allowable people in SGKM. When a reservation is accepted, the acquire() method is called; when someone leaves the event, the release() method is called. If the system reaches the number of allowable people, subsequent calls to acquire() will block until an existing person leaves the event and the release method is invoked.

Reading

- Read Chapter 6
- Acknowledgments
 - These slides are adapted from
 - Öznur Özkarap (Koç University)
 - Operating System and Concepts (9th edition) Wiley
 - Jerry Breecher

Synchronization-II

Didem Unat
Lecture 10

COMP304 - Operating Systems (OS)

Race Condition

```
shared double balance;
```

Code for p₁ (deposit)

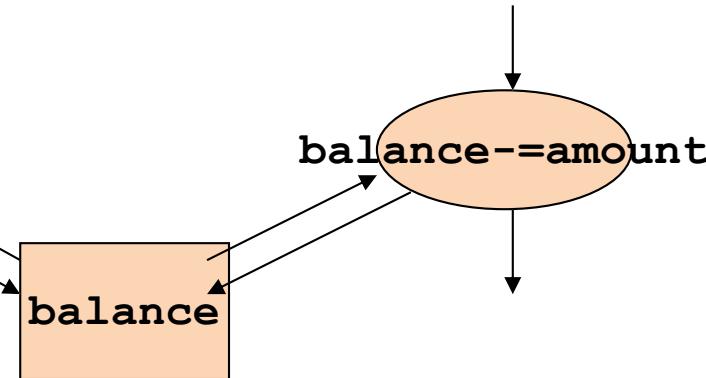
```
. . .
balance = balance + amount;
. . .
```



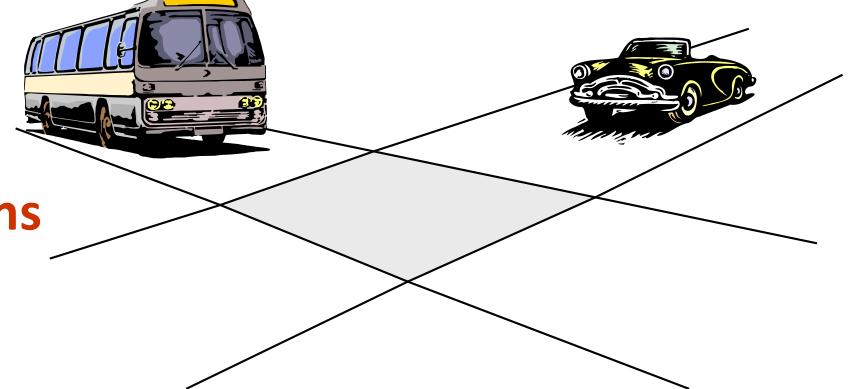
Interleaved Printing

Code for p₂ (withdraw)

```
. . .
balance = balance - amount;
. . .
```



Traffic Intersections



Load, Execute, Store

Execution of p₁

```
...
load R1, balance
load R2, amount
```

Timer interrupt (process p₁ preemption)

Execution of p₂

```
...
load R1, balance
load R2, amount
sub R1, R2
store balance, R1
```

...

Timer interrupt (process p₂ preemption)

```
add R1, R2
store balance, R1
```

...

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

- 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.
- Three properties for any solution for the critical section problem:
 - Mutual Exclusion, Progress, Bounded Waiting Time

Atomic Instructions

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

- The statements

counter++;

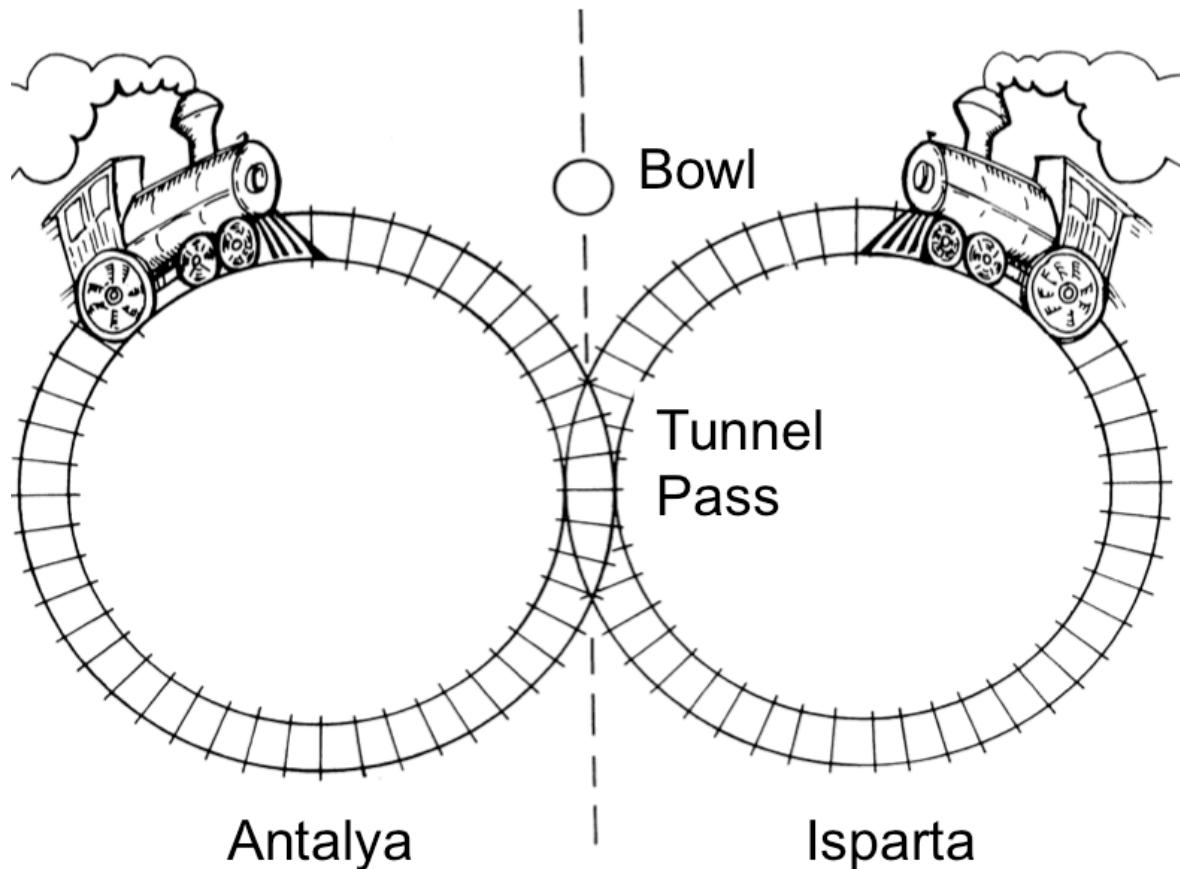
counter--;

```
register1 = counter  
register1 = register1 + 1  
counter = register1
```

```
register2 = counter  
register2 = register2 - 1  
counter = register2
```

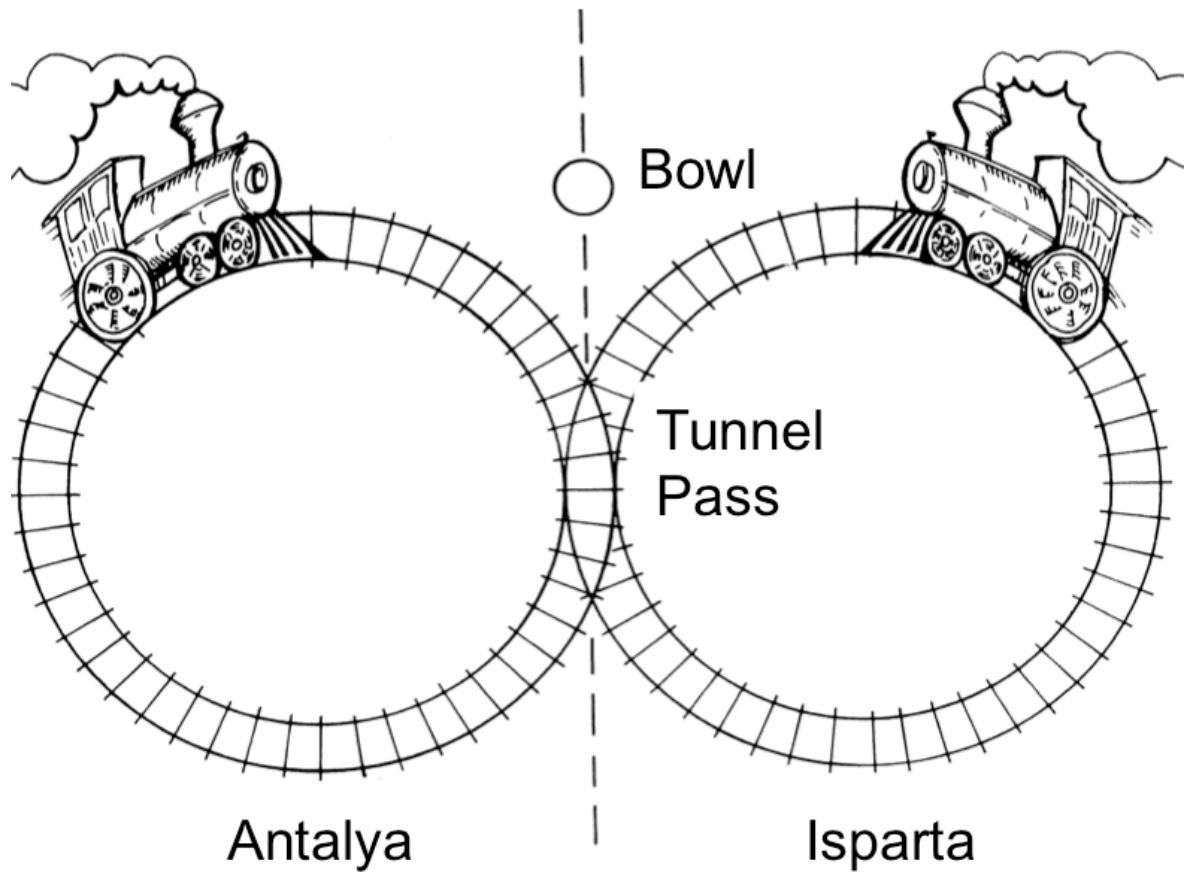
- must be performed **atomically**.
- Atomic operation means an operation that completes in its entirety **without interruption**.

Question



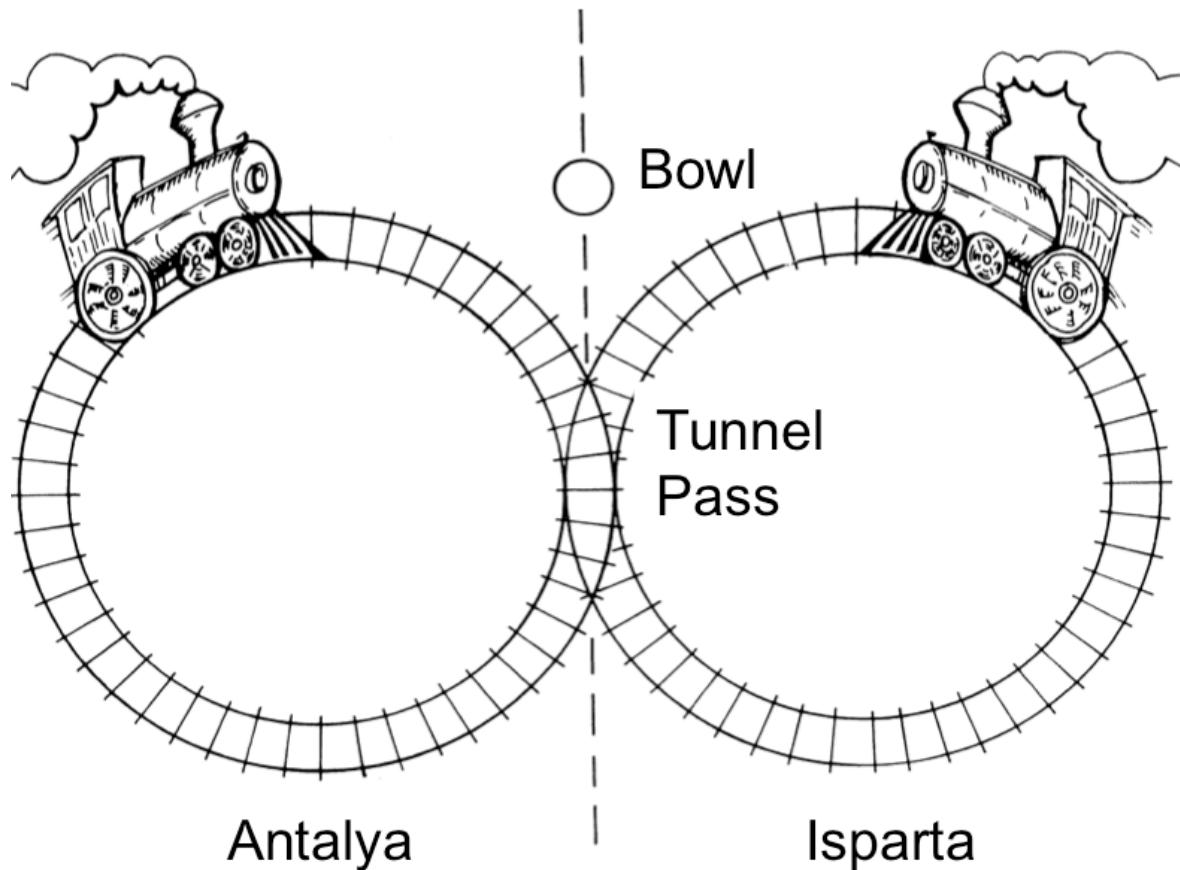
- High in the Toros mountains, there are two circular railway lines. One line is in Antalya, the other in Isparta.
- The Antalya train runs twenty times a day, the Isparta train runs ten times a day.
- They share a common section of track where the lines cross a tunnel pass that lies on the city border.
- 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.**

Method 1



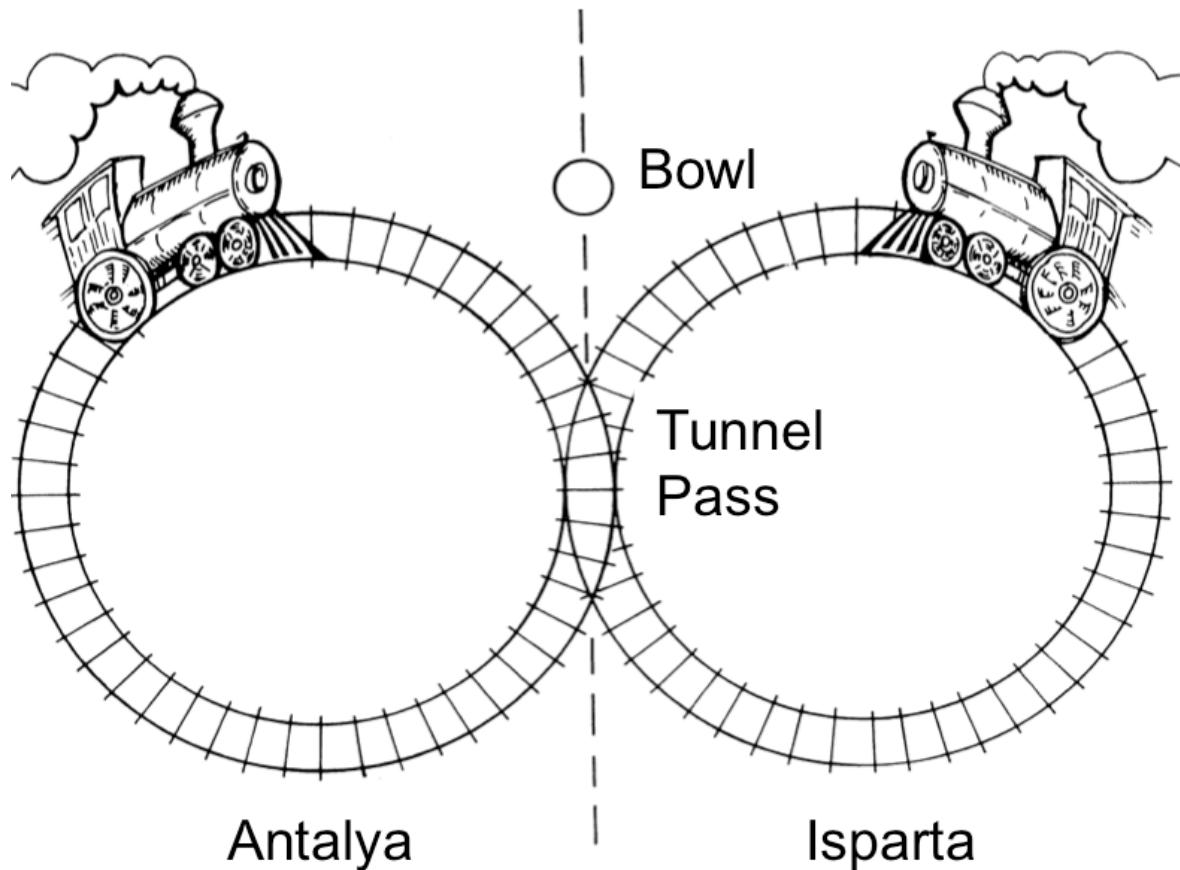
- First method suggests that they set up a large bowl at the entrance to the tunnel.
- Before entering the tunnel, a driver must stop his train, walk over to the bowl, and reach into it to see if it contains a rock.
- If the bowl is empty, the driver finds a rock and drops it in the bowl, indicating that his train is entering the tunnel;
- Once his train has cleared the tunnel, he must walk back to the bowl and remove his rock, indicating that the tunnel is no longer being used. Finally, he walks back to the train and continues down the line.
- If a driver arriving at the tunnel finds a rock in the bowl, he leaves the rock there; he repeatedly takes a nap and rechecks the bowl until he finds it empty. Then he drops a rock in the bowl and drives his train into the pass.

Method 2



- This method uses the bowl in a different way.
- The driver from the Isparta train must wait at the entry to the pass until the bowl is empty, drive through the pass and walk back to put a rock in the bowl. The driver from Antalya must wait at the entry until the bowl contains a rock, drive through the pass and walk back to remove the rock from the bowl.

Method 3

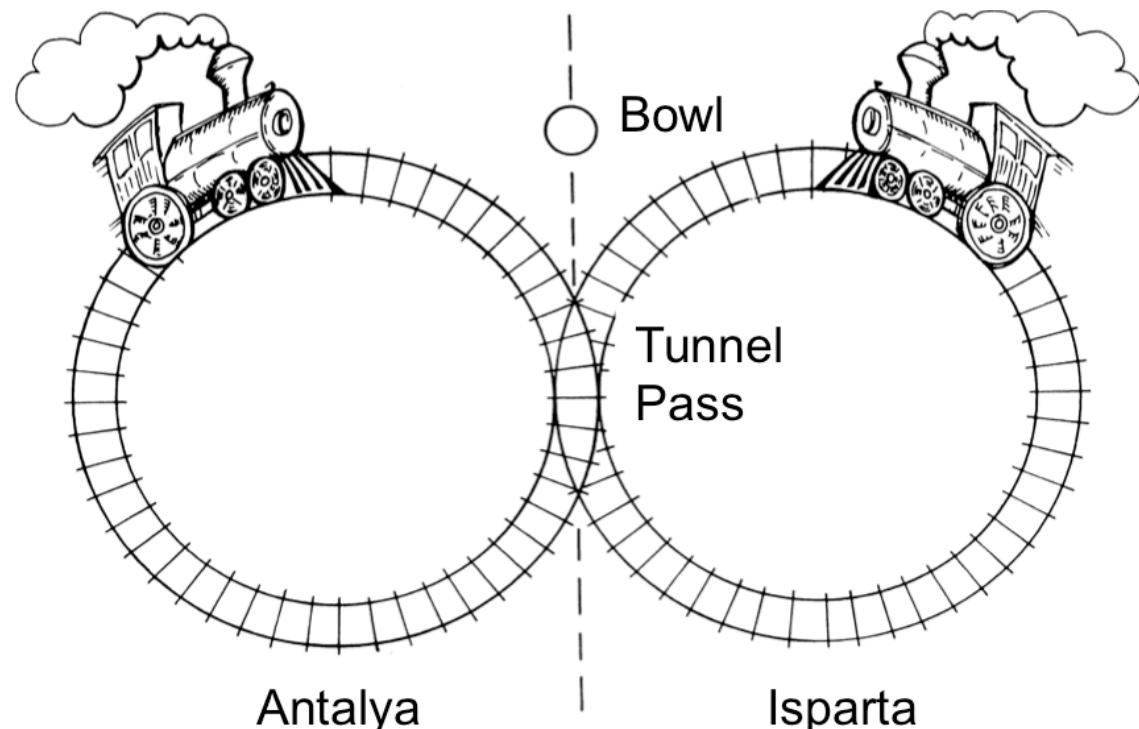


The third method suggests to use two bowls, one for each driver:

- When a driver reaches the entry, he first drops a rock in his bowl, then checks the other bowl to see if it is empty.
- If he finds a rock in the other bowl, he goes back to his bowl and removes his rock. Then he takes a nap, again drops a rock in his bowl and re-checks the other bowl, and so on, until he finds the other bowl empty.
- If the bowl is empty, then he drives his train through the tunnel pass. Stops and walks back to remove his rock.

Your suggestion

- Suggest a method that works
 - Collision free
 - Starvation free
 - Respects the train schedule



Attempt 3: Peterson's Solution

Combined shared variables of algorithms 1 and 2.

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

```
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 1
- Process 1 may see turn 0
- Resulting in data race
- We need hardware support for avoiding race conditions.

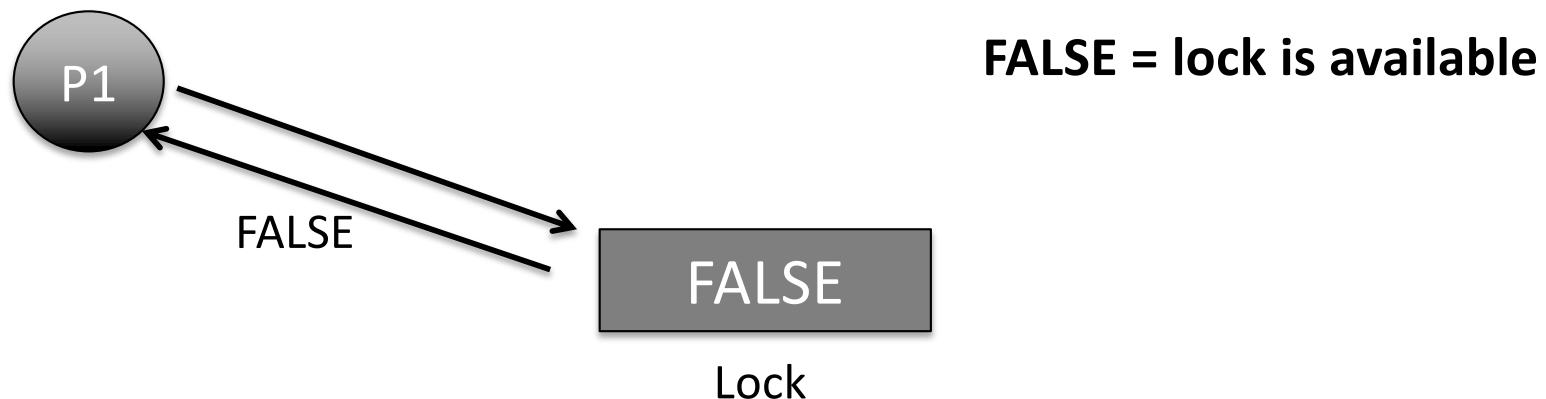
Atomic Test-and-Set Instruction

- 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;  
}
```

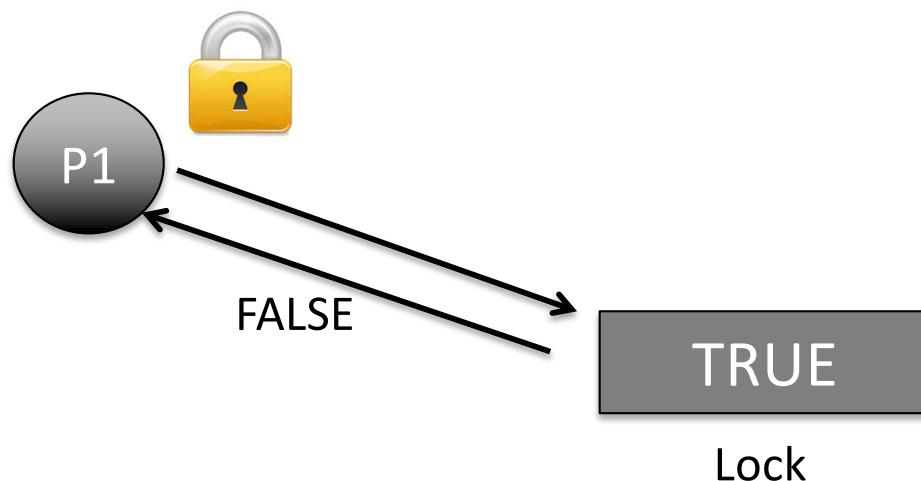
Test-and-set

- Test-and-set does two things atomically:
 - Test a lock (whose value is returned)
 - Set the lock
- Lock obtained when the return value is FALSE
 - If TRUE, someone already had the lock (and still has it)



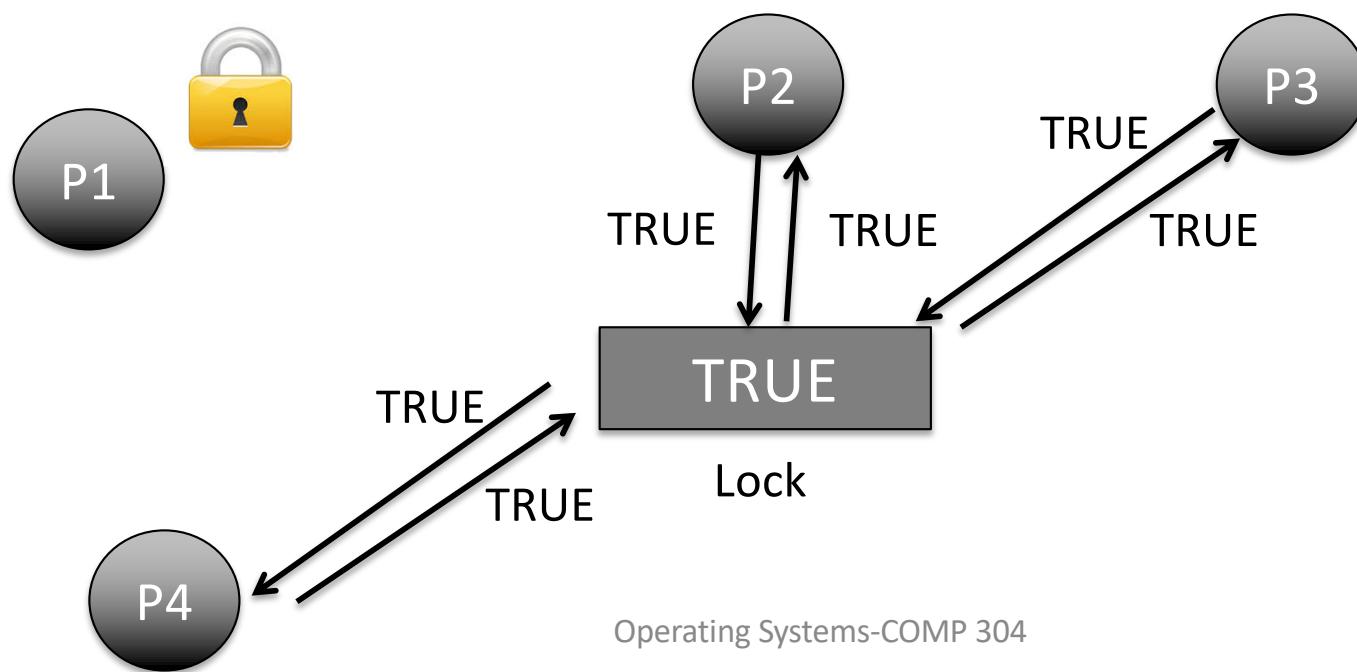
Test-and-set

- Test-and-set does two things atomically:
 - Test a lock (whose value is returned)
 - Set the lock
- Lock obtained when the return value is FALSE
 - If TRUE, someone already had the lock (and still has it)



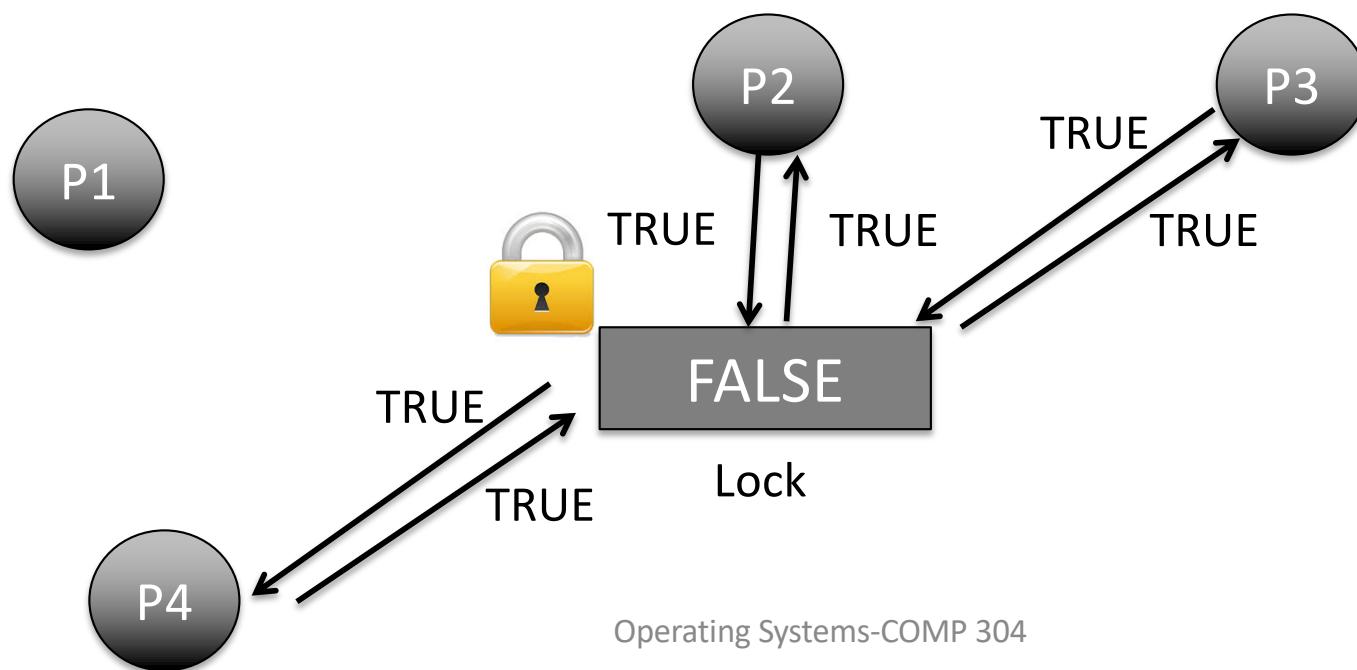
Test-and-set

- Test-and-set does two things atomically:
 - Test a lock (whose value is returned)
 - Set the lock
- Lock obtained when the return value is FALSE
 - If TRUE, someone already had the lock (and still has it)



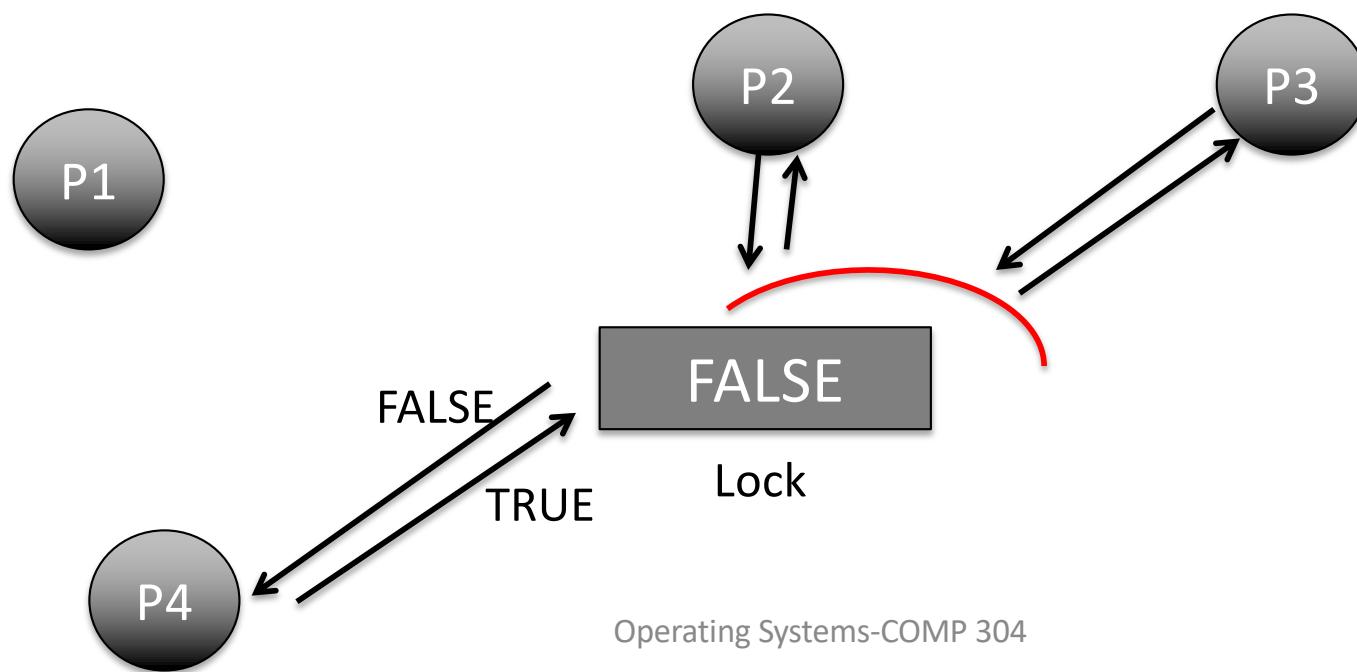
Test-and-set

- Test-and-set does two things atomically:
 - Test a lock (whose value is returned)
 - Set the lock
- Lock obtained when the return value is FALSE
 - If TRUE, someone already had the lock (and still has it)



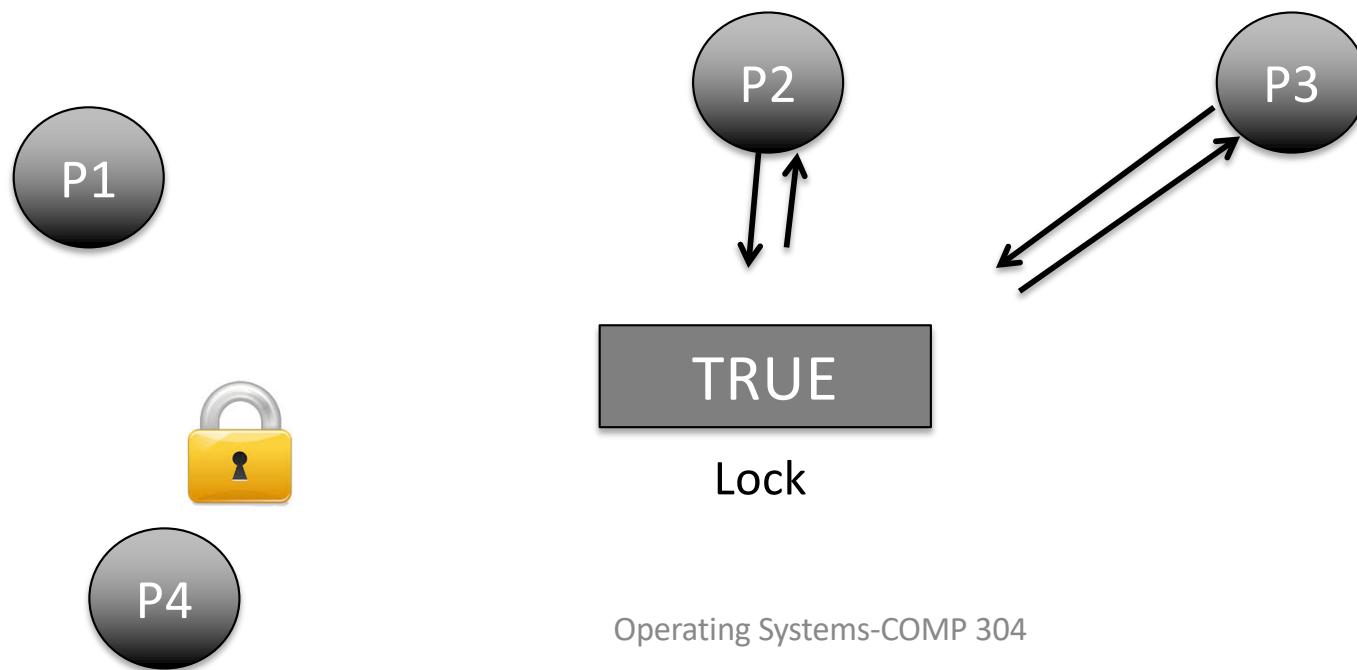
Test-and-set

- Test-and-set does two things atomically:
 - Test a lock (whose value is returned)
 - Set the lock
- Lock obtained when the return value is FALSE
 - If TRUE, someone already had the lock (and still has it)



Test-and-set

- Test-and-set does two things atomically:
 - Test a lock (whose value is returned)
 - Set the lock
- Lock obtained when the return value is FALSE
 - If TRUE, someone already had the lock (and still has it)



Mutual Exclusion with Test-and-Set

- Shared variable:

```
boolean lock = false;
```

- Process P_i

```
do {
    while (TestAndSet(&lock)) { };
    critical section
    lock = false;
    remainder section
} while (true);
```

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

The calling process obtains the lock if the old value was **False**. It spins until it acquires the lock. When it acquires, the value turns to **True** preventing other processes to acquire the lock.

Must be careful if these approaches are to satisfy a bounded wait condition - must use round robin

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;
}
```

Use of compare_and_swap instruction

- Shared boolean variable *lock* initialized to FALSE (0)

```
do {  
    while (compare_and_swap(&lock, 0, 1) != 0)  
        ; /* do nothing */  
  
    /* critical section */  
  
    lock = FALSE;  
    /* remainder section */  
  
} while (true);
```

Expected value New value


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);
```

- 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**

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

Bounded-waiting Mutual Exclusion with test_and_set

```
//Round-robin implementation
Boolean waiting[N];
int j;
//takes on values from 0 to N-1
Boolean key;
```

Each process tries to test_and_set for the lock. Only one succeeds with key=false, then it sets its waiting to false. Enters the critical section. Before it exits, it tries to find a process j who is waiting. If j is not i, then process i hands in the lock to process j.

```
do {
    waiting[i] = true;
    key = true;
    while (waiting[i] && key)
        key = test_and_set(&lock);
    waiting[i] = false;

    /* critical section */

    j = (i + 1) % n;
    while ((j != i) && !waiting[j])
        j = (j + 1) % n;
    if (j == i)
        lock = false;
    else
        waiting[j] = false;

    /* remainder section */

} while (true);
```

Reading

- Read Chapter 6
- Acknowledgments
 - These slides are adapted from
 - Öznur Özkarap (Koç University)
 - Operating System and Concepts (9th edition) Wiley
 - Jerry Breecher

Process Synchronization

Didem Unat

Lecture 9

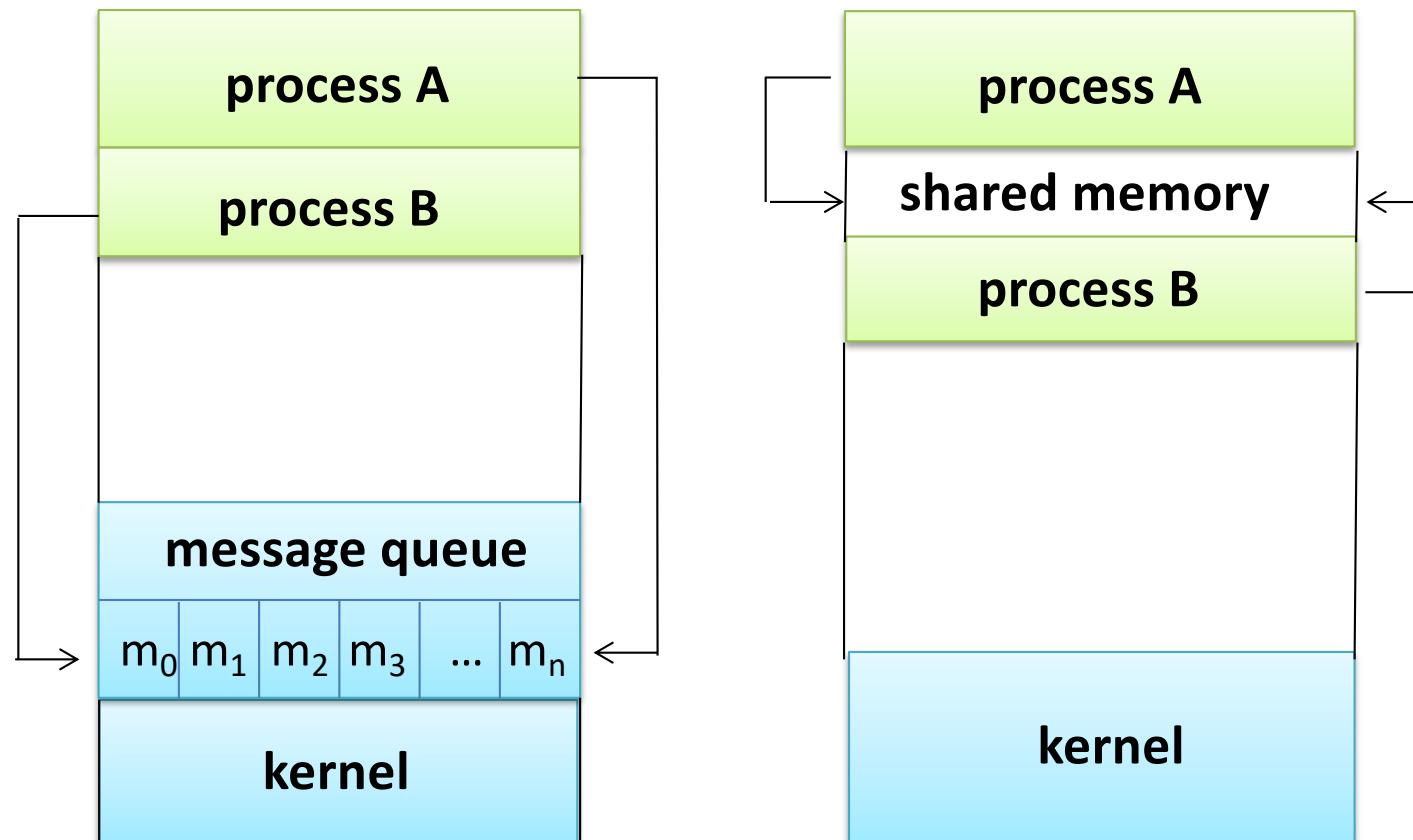
COMP304 - Operating Systems (OS)

Process Synchronization

- Race Condition
- The Critical-Section Problem
- Synchronization Hardware
- Semaphores
- Classical Problems of Synchronization
- Monitors
- Synchronization Examples

Two Models of Communication

Message Passing vs Shared Memory



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

Shared Memory

- Communication through shared memory takes place with shared variables.
- Threads or processes share common variables
- Access to these variables should be coordinated (synchronized) so that the data is not corrupted.

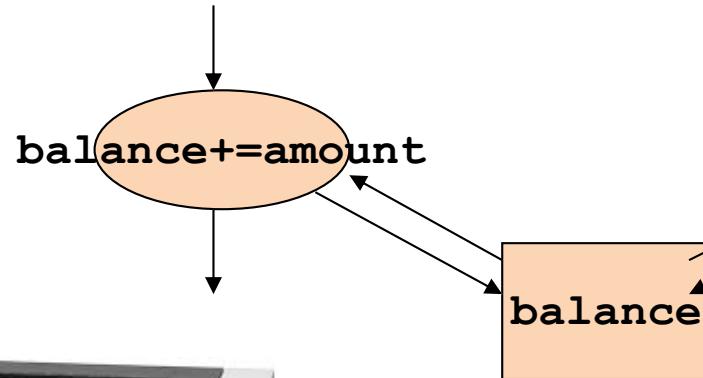
Problem

```
shared double balance;
```

Code for p₁ (deposit)

```
    . . .
balance = balance + amount;
```

```
    . . .
```

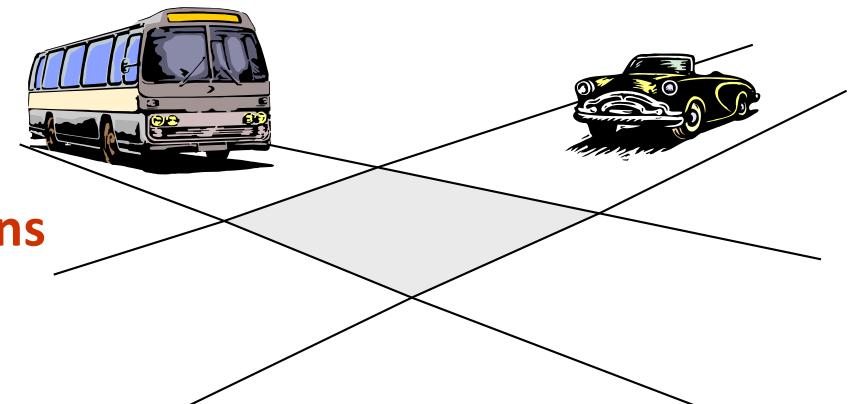
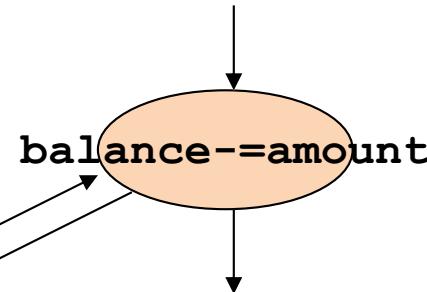


Interleaved Printing

Code for p₂ (withdraw)

```
    . . .
balance = balance - amount;
```

```
    . . .
```



Traffic Intersections

Background

- Processes can execute concurrently
 - May be interrupted at any time, partially completing execution
- Concurrent access to shared data may result in data inconsistency.
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes.

Producer-Consumer Problem

- A **producer** process "produces" information "consumed" by a **consumer** process.
- Here are the variables needed to define the problem:

```
#define BUFFER_SIZE 10
typedef struct {
    DATA      data;
} item;

item   buffer[BUFFER_SIZE];
int    in = 0;           // Location of next input to buffer
int    out = 0;          // Location of next removal from buffer
int    counter = 0;      // Number of items in the buffer
```

Counter is initialized to 0 and incremented each time a new item is added to the buffer and decremented when an item is consumed.

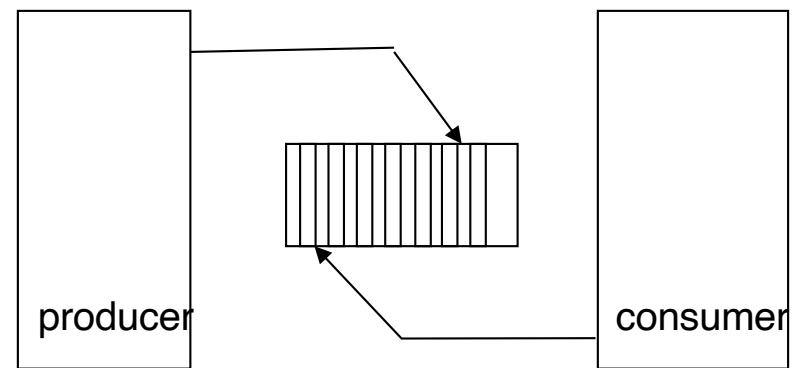
Producer-Consumer Problem

```
item    nextProduced;      //PRODUCER

while (TRUE) {
    while (counter == BUFFER_SIZE);
    buffer[in] = nextProduced;
    in = (in + 1) % BUFFER_SIZE;
counter++;
}
```

```
item    nextConsumed;      //CONSUMER

while (TRUE) {
    while (counter == 0);
    nextConsumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
counter--;
}
```



Example:

Producer produces papers
to be printed.
Consumer prints the papers.
They use a common queue.

Counter

- The statements

counter++;

counter--;

```
register1 = counter  
register1 = register1 + 1  
counter = register1
```

```
register2 = counter  
register2 = register2 - 1  
counter = register2
```

- Even though these are single statements in the code, they are compiled into and executed as multiple instructions in the hardware.

Race Condition

Producer	Consumer		Counter
			5
read value		←	5
increase value			5
write back		→	6
	read value	←	6
	Decrease value		6
	write back	→	5

Producer	Consumer		Counter
			5
read value		←	5
	read value	←	5
increase value			5
	decrease value		5
write back		→	6
	write back	→	4

Consider this execution (shown on the right) interleaving with “count = 5” initially:

```

S0: producer execute register1 = counter           {register1 = 5}
S1: producer execute register1 = register1 + 1     {register1 = 6}
S2: consumer execute register2 = counter           {register2 = 5}
S3: consumer execute register2 = register2 - 1     {register2 = 4}
S4: producer execute counter = register1          {counter = 6 }
S5: consumer execute counter = register2          {counter = 4}

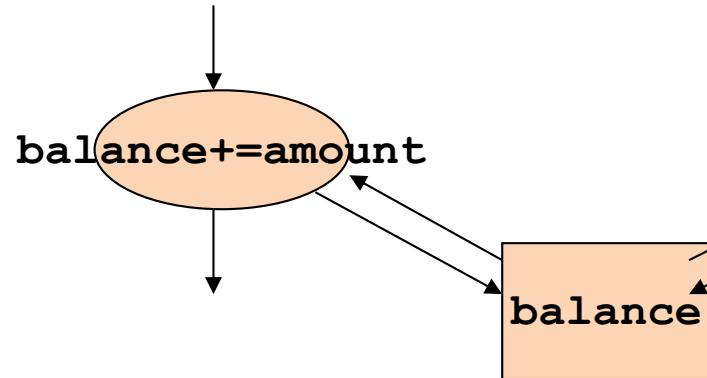
```

Problem

```
shared double balance;
```

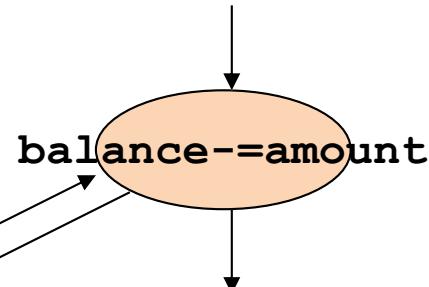
Code for p₁ (deposit)

```
. . .
balance = balance + amount;
. . .
```



Code for p₂ (withdraw)

```
. . .
balance = balance - amount;
. . .
```



Load, Execute, Store

Execution of p₁

```
...  
load R1, balance  
load R2, amount
```

Timer interrupt (process p₁ preemption)

Execution of p₂

```
...  
load R1, balance  
load R2, amount  
sub R1, R2  
store balance, R1
```

...

Timer interrupt (process p₂ preemption)

```
add R1, R2  
store balance, R1  
...
```

This store is lost
because it is
overwritten by the
process 1's store.

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**.

Preemptive Kernels

- A non-preemptive kernel is free from race conditions on kernel data structures
- A preemptive kernel allows a process to be preempted while it is running in kernel mode
 - Need to ensure that shared kernel data are free from race conditions
 - Examples of shared kernel data
 - List of open files, interrupt handlers, process list / queues, memory allocation management

The Critical Section Problem

- Consider system of n processes $\{p_0, p_1, \dots p_{n-1}\}$ 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:

do {

Entry Section Code requesting entry into the critical section.

Entry section

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

Critical section

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

Exit section

Remainder Section Rest of the code AFTER the critical section.

}while (true);

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.

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.

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 1 to Solve Problem

- Only 2 processes, P_0 and P_1

i is the current process, j the "other" process.

```
//Pi:  
//turn =>shared variable, initially i  
//if turn == i => Pi can enter its  
critical  
  
do {  
    while ( turn != i );  
    /* critical section */  
    turn = j;  
    /* remainder section */  
  
} while(TRUE);
```

```
//Pj:  
//turn =>shared variable, initially i  
//if turn == j => Pj can enter its  
critical  
  
do {  
    while ( turn != j );  
    /* critical section */  
    turn = i;  
    /* remainder section */  
  
} while(TRUE);
```

Are the three Critical Section Requirements Met?

Satisfies mutual exclusion, but not progress because P_i needs to enter the critical section, then only P_j can enter.

Attempt 2 to Solve Problem

Shared variables

- ◆ **boolean flag[2];**
//initially flag[0] = flag[1] = false.
- ◆ If flag[i] == true $\Rightarrow P_i$ ready to enter its critical section

```
do {  
    flag[i] = true;  
    while (flag[j]) ;  
  
    //critical section  
  
    flag [i] = false;  
  
    //remainder section  
} while (true);
```

Each process maintains a flag indicating that it wants to get into the critical section. It checks the flag of the other process and doesn't enter the critical section if other process wants to get in.

Are the three Critical Section Requirements Met?

Satisfies mutual exclusion, but not progress and not bounded waiting

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

- **Mutual exclusion:** P0 and P1 can never be in the critical section at the same time: If P0 is in its critical section, then flag[0] is true. In addition, either flag[1] is false (meaning P1 has left its critical section), or turn is 0 (meaning P1 is just now trying to enter the critical section, but graciously waiting), or P1 is trying to enter its critical section, after setting flag[1] to true but before setting turn to 0. So if both processes are in their critical sections then we conclude that the state must satisfy flag[0] and flag[1] and turn = 0 and turn = 1. No state can satisfy both turn = 0 and turn = 1, so there can be no state where both processes are in their critical sections.
- **Progress:** A process cannot immediately re-enter the critical section if the other process has set its flag to say that it would like to enter its critical section.
- **Bounded waiting:** In Peterson's algorithm, a process will not wait longer than one turn for entrance to the critical section: After giving priority to the other process, this process will run to completion and set its flag to 0, thereby allowing the other process to enter the critical section.

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--;

```
register1 = counter  
register1 = register1 + 1  
counter = register1
```

```
register2 = counter  
register2 = register2 - 1  
counter = register2
```

- 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;  
}
```

Mutual Exclusion with Test-and-Set

- Shared variable:

```
boolean lock = false;
```

- Process P_i

```
do {
    while (TestAndSet(&lock)) { };
    critical section
    lock = false;
    remainder section

} while (true);
```

Mutual Exclusion with Test-and-Set

- Shared variable:

```
boolean lock = false;
```

- Process P_i

```
do {
    while (TestAndSet(&lock)) { };
    critical section
    lock = false;
    remainder section
} while (true);
```

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

The calling process obtains the lock if the old value was **False**. It spins until it acquires the lock. When it acquires, the value turns to **True** preventing other processes to acquire the lock.

Must be careful if these approaches are to satisfy a bounded wait condition - must use round robin

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;
}
```

Use of compare_and_swap instruction

- Shared boolean variable *lock* initialized to FALSE (0)

```
do {  
    while (compare_and_swap(&lock, 0, 1) != 0)  
        ; /* do nothing */  
  
    /* critical section */  
  
    lock = FALSE;  
    /* remainder section */  
  
} while (true);
```

Expected value New value


Mutex Locks

- Previous hardware solutions are complicated and generally too low level to application programmers
- OS designers build software tools to solve critical section problem on top of these hardware solutions
- 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);
```

- 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**

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

Bounded-waiting Mutual Exclusion with test_and_set

```
//Round-robin implementation
Boolean waiting[N];
int j;
//takes on values from 0 to N-1
Boolean key;
```

Each process tries to test_and_set for the lock. Only one succeeds with key=false, then it sets its waiting to false. Enters the critical section. Before it exits, it tries to find a process j who is waiting. If j is not i, then process i hands in the lock to process j.

```
do {
    waiting[i] = true;
    key = true;
    while (waiting[i] && key)
        key = test_and_set(&lock);
    waiting[i] = false;

    /* critical section */

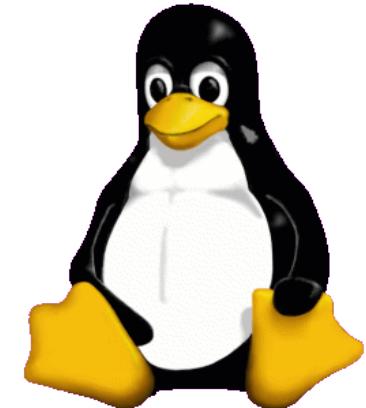
    j = (i + 1) % n;
    while ((j != i) && !waiting[j])
        j = (j + 1) % n;
    if (j == i)
        lock = false;
    else
        waiting[j] = false;

    /* remainder section */

} while (true);
```

Reading

- Read Chapter 6
- Acknowledgments
 - These slides are adapted from
 - Öznur Özkasap (Koç University)
 - Operating System and Concepts (9th edition) Wiley
 - Jerry Breecher



Real-Time CPU Scheduling and Linux Scheduler

Didem Unat

Lecture 8

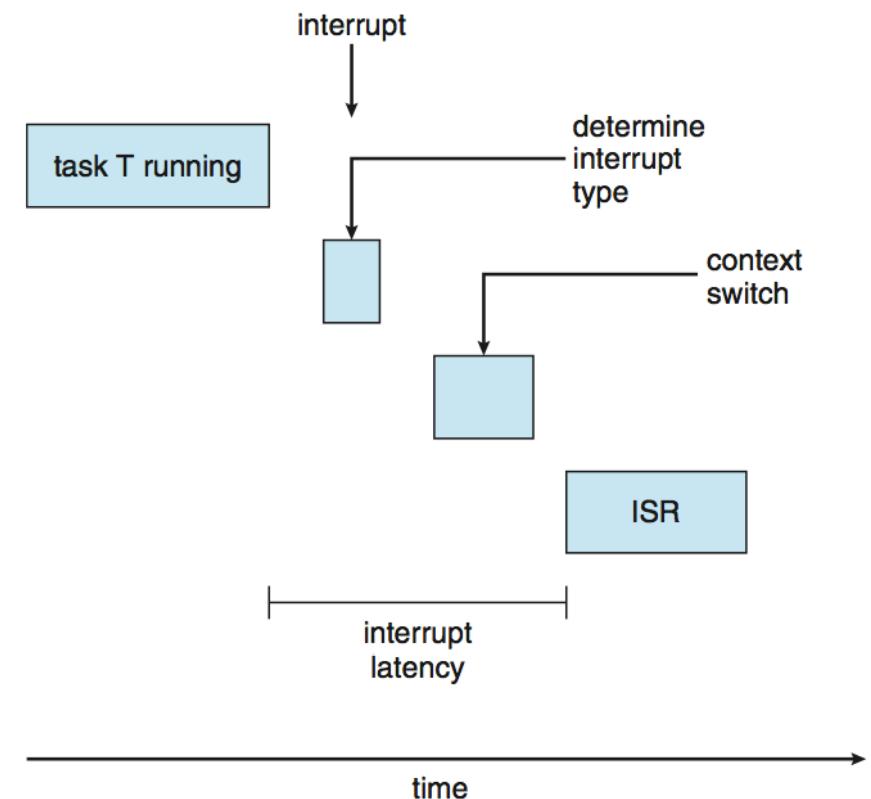
COMP304 - Operating Systems (OS)

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

Real-Time CPU Scheduling

- **Event latency:** time that elapses from when an event occurs to when it is serviced
- Two types of latencies affect performance
 1. **Interrupt latency** – time from arrival of interrupt to start of routine that services the interrupt
 2. **Dispatch latency** – time for scheduler to take current process off from CPU and switch to another



ISR Stands for "*Interrupt Service Routine*".

Real-Time OSs

- **Event driven systems** switch between tasks based on their priorities while time sharing systems switch the task based on clock interrupts.
- Design goal is not high throughput, but rather a guarantee of service for a high priority job
- Real-time OS is more frequently dedicated to a narrow set of applications.
 - Targeted usage is typically embedded systems, robots etc.
 - Supporting industrial, automotive, smart city and smart home
- Some open-source real-time OSs:
 - Zephyr : <https://www.zephyrproject.org/>
 - uKOS
 - FreeRTOS
- http://www.wikiwand.com/en/Comparison_of_real-time_operating_systems

Zephyr (an example RTOS)

- **Extensive suite of Kernel services:**
 - *Multi-threading Services* for cooperative, priority-based, threads with optional round robin time-slicing. Includes POSIX pthreads compatible API support.
 - *Interrupt Services* for compile-time registration of interrupt handlers.
 - *Memory Allocation Services* for dynamic allocation and freeing of fixed-size or variable-size memory blocks.
 - *Inter-thread Synchronization Services* for binary semaphores, counting semaphores, and mutex semaphores.
 - *Inter-thread Data Passing Services* for basic message queues ...
 - *Power Management Services* such as tickless idle and an advanced idling infrastructure.
- **Multiple Scheduling Algorithms:**
 - Preemptive Scheduling
 - Earliest Deadline First (EDF)
 - Timeslicing: Enables time slicing between preemptible threads of equal priority
 - Multiple queuing strategies:
 - Simple linked-list ready queue
 - Red/black tree ready queue
 - Traditional multi-queue ready queue

<https://docs.zephyrproject.org/latest/introduction/index.html>

Linux Scheduler

History

Linux Version	Scheduler
Pre 2.5	Multi-level Feedback Queue
Pre 2.6.23	O(1) Scheduler
Post 2.6.23	Completely fair scheduler

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

Priority

- Each task has a **static priority** that is set based upon the nice value specified by the task.
 - *static_prio* in *task_struct*
 - *Default is 120*
- For normal tasks, the **static priority** is $100 + \text{nice}$.
- Each task has a **dynamic priority** that is set based upon a number of factors
 - *prio* in *task_struct*

Niceness

- Niceness
 - a process is nicer to others if it has a higher nice value
 - Default is inherited from its parent (usually 0)
 - Ranges from -20 to +19
- Nice value of -20 maps to global priority 100
- Nice value of +19 maps to priority 139

Value can be set via **nice()** system call or **nice** command

```
bash$ nice -n 19 tar cvzf archive.tgz largefile
```

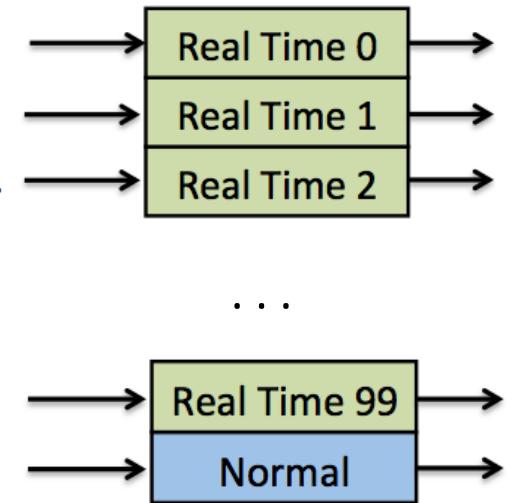
Prior to Kernel 2.5

In the 2.4 kernel, this was the scheduling algorithm:

- Each task got a number of CPU ticks (*jiffies*) made available to the task each scheduling interval, or epoch.
- The number of new ticks given was determined from the nice value for the task. It was roughly:
$$((20-\text{nice}) * \text{HZ}/800) + 1.$$
- Each task had a *counter*, which was the number of CPU ticks still left for the task to use in the current epoch.
- Unused ticks in a particular epoch decayed by 50% for use in the next interval.

Linux O(1) Scheduler

- Version 2.5 moved to constant order $O(1)$ scheduling time
 - Preemptive, priority based
 - Two priority ranges: time-sharing and real-time
 - **Real-time** range from 0 to 99 and **normal** (time-sharing) range from 100 to 139
 - Higher priority gets larger time quantum
 - Scales well with the number of processes



Real-Time Scheduling

- Linux has a soft real-time scheduler
 - No hard real-time guarantees
 - All real-time processes are higher priority than any normal processes
- Processes with priorities [0, 99] are real-time
 - saved in *rt_priority* in the *task_struct*
 - scheduling priority of a real time task is: 99 - *rt_priority*
- A process can be converted to real-time via *sched_setscheduler* system call

Scheduling Policies

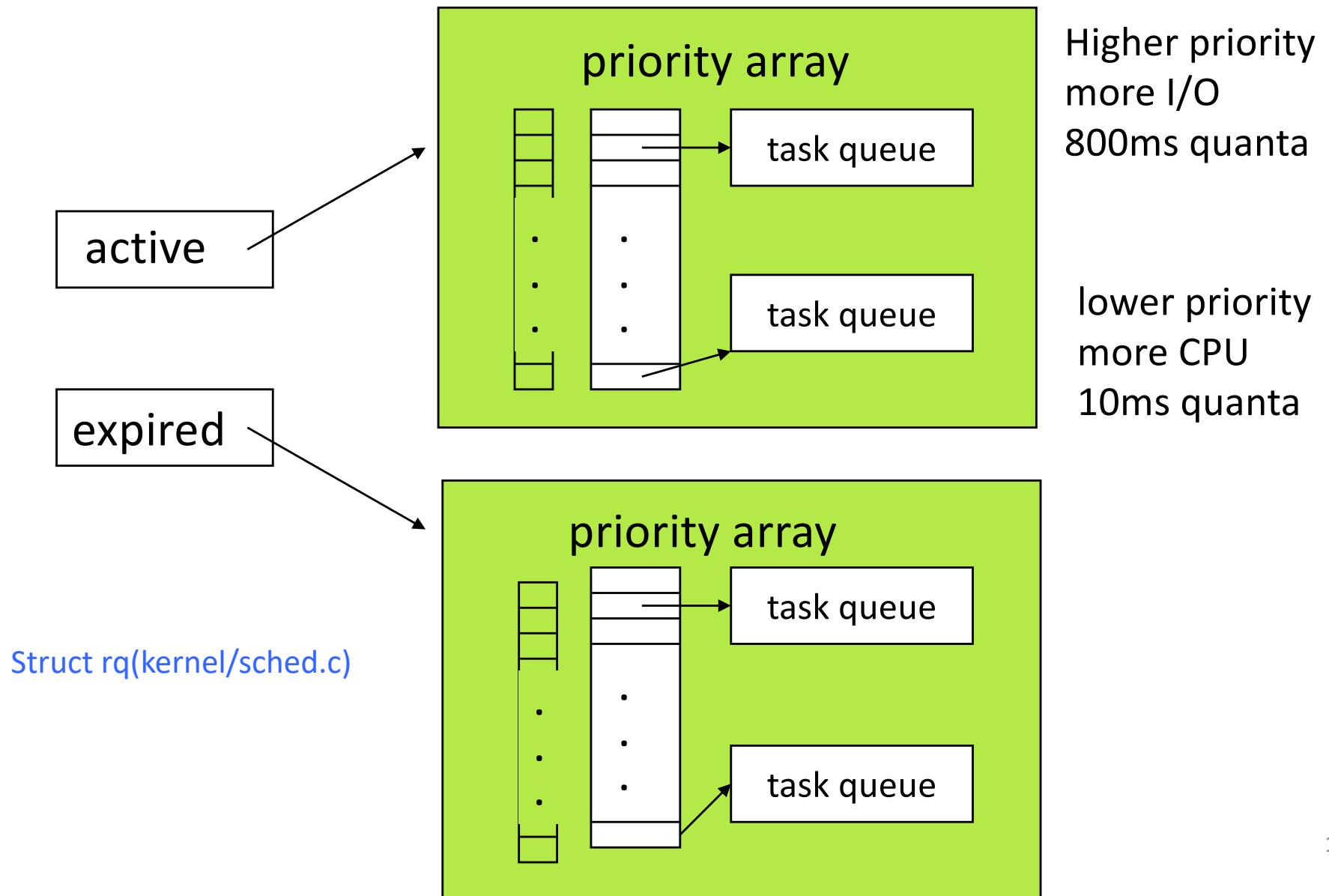
- Real-time processes
 - First-in, first-out: **SCHED_FIFO**
 - Static priority
 - Process is only preempted for a higher-priority process
 - No time quanta; it runs until it blocks or yields voluntarily
 - Round-robin: **SCHED_RR**
 - RR within the same priority level
 - A time quanta (800 ms)
- Normal processes have
 - **SCHED_OTHER**: standard processes
 - **SCHED_BATCH**: batch style processes
 - **SCHED_IDLE**: low priority tasks

O(1) Scheduler

- Task runnable as long as time left in time slice (**active**)
- If no time left (**expired**), not runnable until all other tasks use their slices
- All runnable tasks tracked in per-CPU **runqueue** data structure
 - Two priority arrays (active, expired)
 - When no more active, arrays are swapped

Runqueues

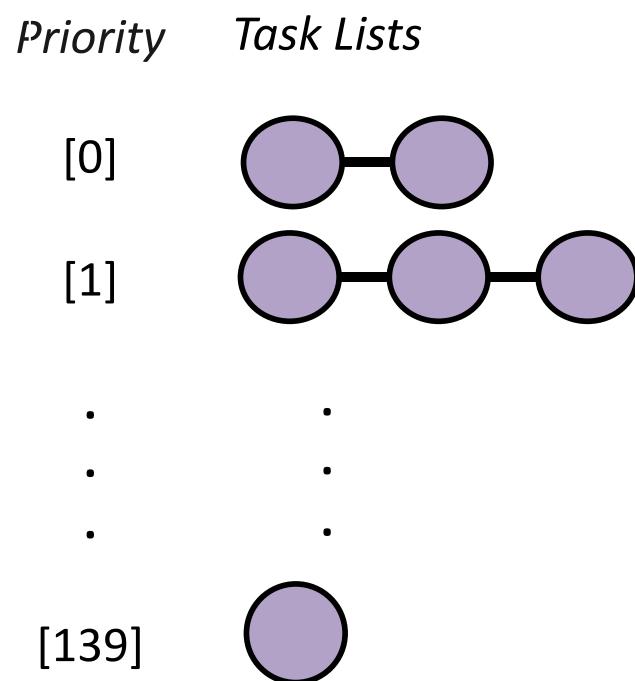
140 separate queues, one for each priority level in two sets: active and expired



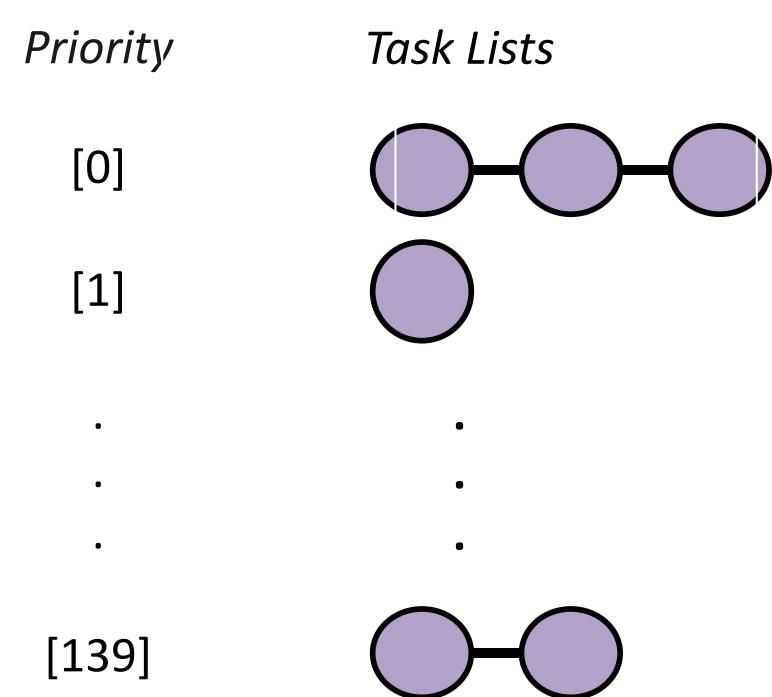
Runqueues

- Two arrays of priority queues: **active** and **expired**
 - Total 140 priorities [0, 140)
 - Smaller integer = higher priority

Active Array



Expired Array



Scheduling Algorithm for Normal Processes

- Find the highest-priority non-empty queue in `rq->active`; if none, simulate aging by swapping active with expired
- `Next` = Find the first process on that queue
- Calculate `next's` quantum size and its `next's` priority
- Context switch to `next`
- Let it run
- When its time is up, put it on the `expired` list
- Repeat

Simulate Aging

- After running all of the active queues, the active and expired queues are swapped
- There are pointers to the current arrays; at the end of a cycle, the pointers are switched
- Swapping active and expired gives low priority processes a chance to run
- Advantage: $O(1)$
 - Processes are touched only when they start or stop running

Find highest priority non-empty queue

- Time complexity $O(1)$
 - Depends on the number of priority levels, not the number of processes
- Implementation: a bitmap for fast look up
 - 140 queues
 - A few comparisons to find the first non-zero bit

Calculating Time Slices

- *time_slice* in the *task_struct*
- Calculate Quantum where
 - If ($SP < 120$): $\text{Quantum} = (140 - SP) \times 20$
 - if ($SP \geq 120$): $\text{Quantum} = (140 - SP) \times 5$
where SP is the *static priority*
- Higher priority process gets longer quanta
- Basic idea: important processes should run longer

Typical Quanta

Priority:	Static Pri	Niceness	Quantum
Highest	100	-20	800 ms
High	110	-10	600 ms
Normal	120	0	100 ms
Low	130	10	50 ms
Lowest	139	19	5 ms

Issues with O(1) RR Scheduler

- Not easy to distinguish between CPU and I/O bound
 - I/O bound typically needs better interactivity
- Finding right time slice isn't easy
 - Too small: good for I/O bound but high overhead
 - Too large: good for CPU bound but poor interactivity
- Priority is relative but time slice is absolute
 - Nice 0, 1: time slice 100 and 95 msec: 5% difference
 - Nice 18,19: time slice 10 and 5 msec: 100 % difference

Completely Fair Scheduler (CFS)

- Starting from Linux kernel version 2.6.23 since 2007
- Not based on runqueues as in O(1) scheduler
- Not based on time slices
- Note that CFS is used only for normal processes, for real-time processes, Linux still use priority based FCFS and RR schedulers

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

CFS

- CFS calculates how long a process should run as a function of the total number of runnable processes.
 - If there are N runnable processes, then each should be afforded $1/N$ of the processor's time.
 - CFS adjusts the allotment by weighting each process's allotment by its nice value.
 - Small nice value => higher weight
 - Large nice value => lower weight
 - Then process's time slice is proportional to its weight divided by the total weight of all runnable processes.

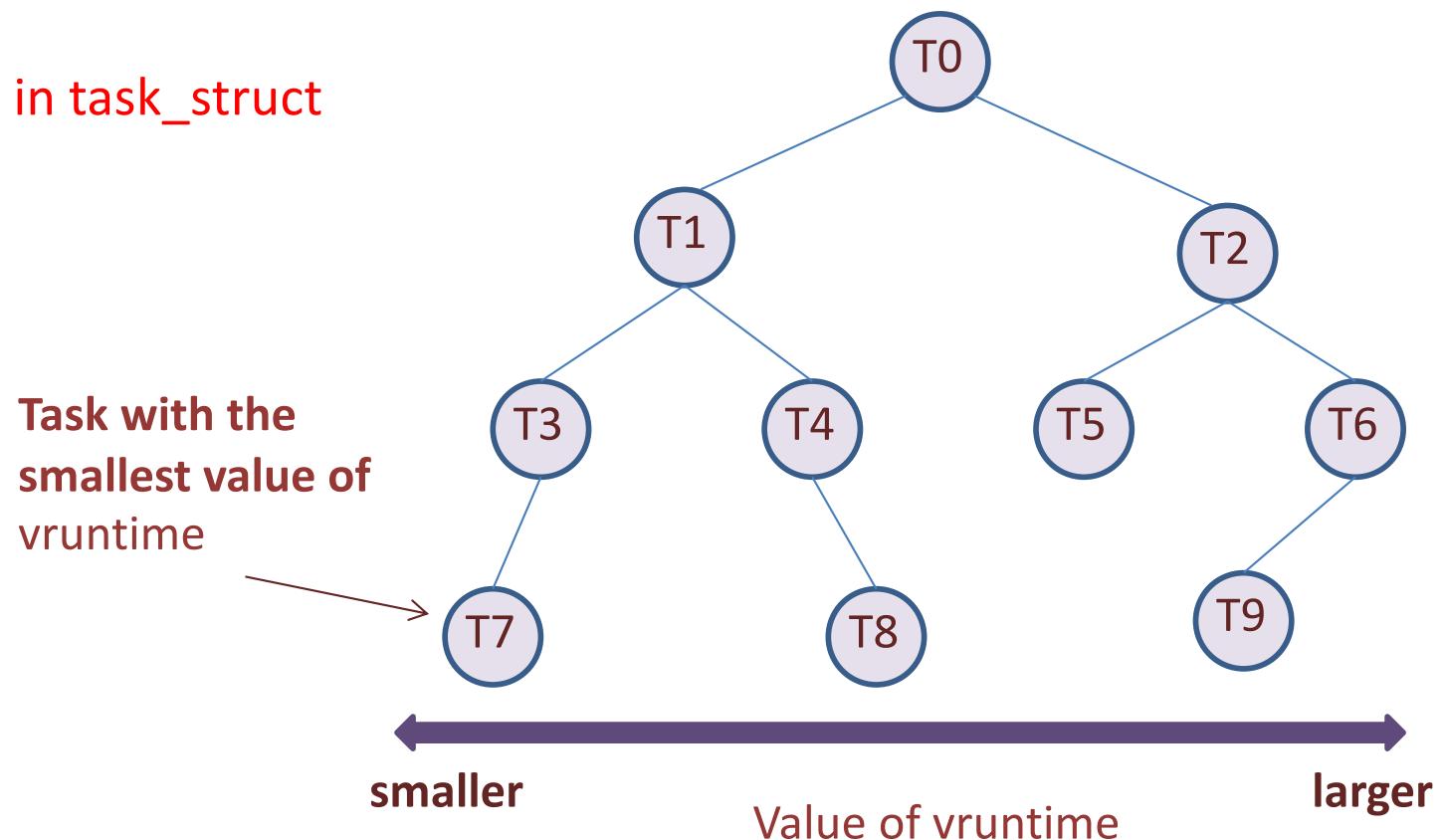
$$\text{Timeslice(task)} = \text{Timeslice}(t) * \text{prio}(t) / \text{Sum_all_t}'(\text{prio}(t'))$$

$$\text{Timeslice (t)} = \text{latency} / \text{nr_tasks}$$

CFS Tree

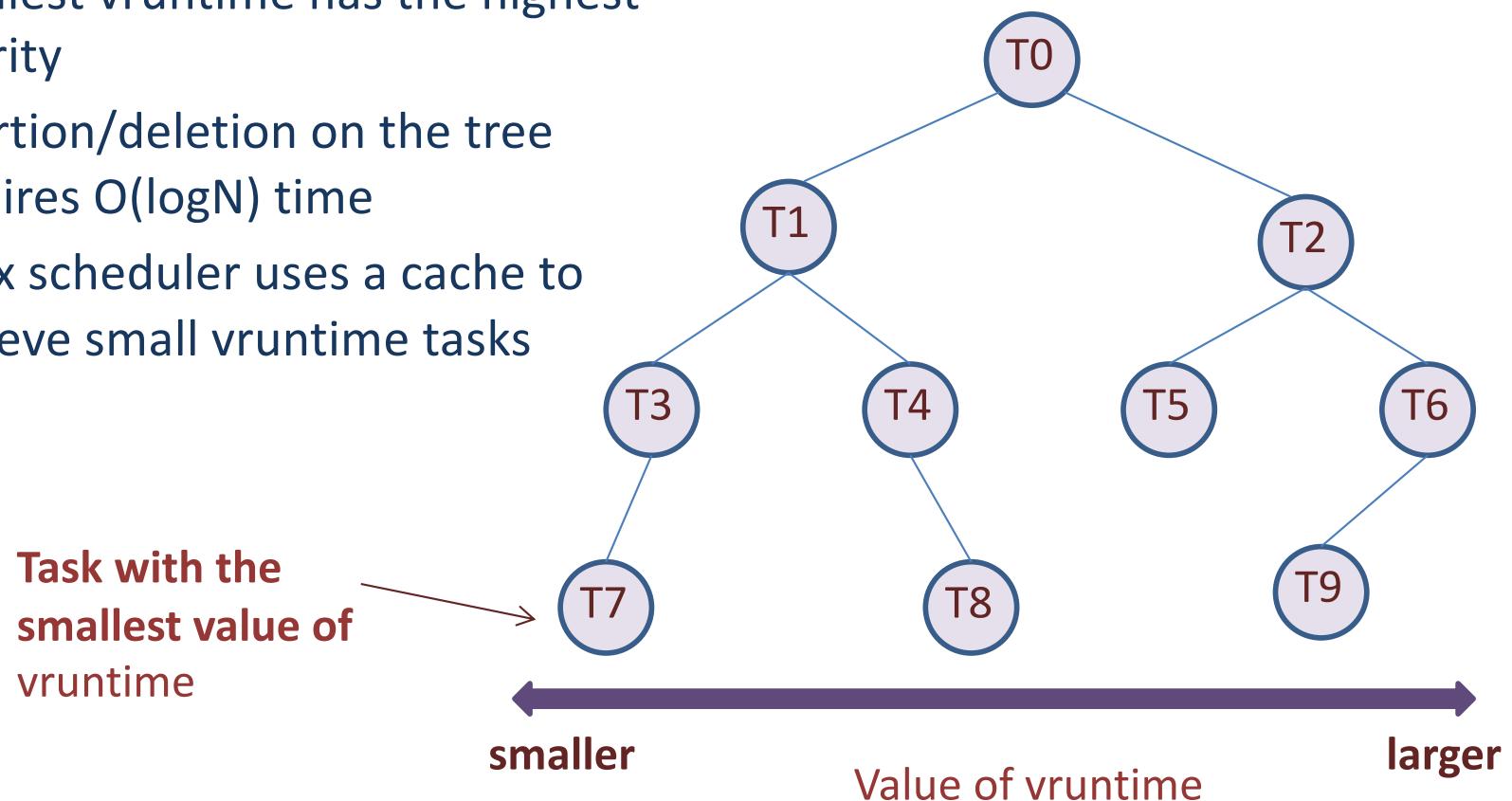
- Each runnable task is placed in a red-black tree
 - A balanced binary search tree whose key is based on the value of vruntime (task with min runtime so far)

sched entity in task struct



CFS Tree

- When a task becomes runnable, it is added to the tree (red/black tree)
 - Not runnable tasks (e.g. waiting for I/O) are removed from the tree
 - Smallest vruntime has the highest priority
 - Insertion/deletion on the tree requires $O(\log N)$ time
 - Linux scheduler uses a cache to retrieve small vruntime tasks



CFS (con.t)

- Two tasks have the same nice values
- One task is I/O bound, other is CPU-bound
 - I/O bound normally runs for a short period before it is interrupted for an I/O operation
 - CPU-bound normally exhausts all its quantum
- Vruntime will eventually be lower for the I/O bound task than for the CPU-bound task
 - Thus I/O bound will get access to CPU more often
 - Vruntime is weighted by process priority

Picking the next process

- Pick task with minimum runtime so far
- Every time process runs for t ns
 - $V_{\text{runtime}} += t$
- How does this impact I/O vs CPU bound tasks?
 - Task A needs CPU for 1 msec every 100 msec (I/O bound)
 - Task B, C need CPU for 80 msec every 100 msec (CPU bound)
 - After 10 times that A, B and C have been scheduled.
 - $V_{\text{runtime}}(A) = 10$
 - $V_{\text{runtime}}(B,C) = 800$
 - Overtime task A gets priority, but it quickly releases CPU.

CFS Algorithm

- The leftmost node of the scheduling tree is chosen (as it will have the lowest spent *execution time*), and sent for execution.
- If the process simply completes execution, it is removed from the system and scheduling tree.
- If the process reaches its *maximum execution time* or is otherwise stopped (voluntarily or via interrupt) it is reinserted into the scheduling tree based on its new spent *execution time*.
- The new leftmost node will then be selected from the tree, repeating the iteration.

Choosing a task can be done in constant time, but reinserting a task after it has run requires $O(\log N)$ operations

Multiprocessor Scheduling

- Each processor maintains a red/black tree
- Each processor only selects processes from its own tree to run
- It's possible for one processor to be idle while others have jobs waiting in their run queues
- Periodically, rebalance
 - `void load_balance()`
 - Attempts to move tasks from one CPU to another

Processor Affinity

- Each process has a bitmask saying what CPUs it can run on
- Normally, of course, all CPUs are listed
- Processes can change the mask
- The mask is inherited by child processes (and threads), thus tending to keep them on the same CPU
 - not allowed to run on the current CPU (as indicated by the *cpus_allowed* bitmask in the *task_struct*)

Adding a new Scheduler Class to Linux

- The Scheduler is modular and extensible
- Each scheduler class has priority within hierarchical scheduling hierarchy
 - Priorities defined in sched.h, e.g. #define SCHED_RR 2
 - Linked list of sched_class sched_class.next reflects priority
- Core functions:
 - kernel/sched.c, include/linux/sched.h
 - Additional classes: kernel/sched_fair.c, sched_rt.c
- Process changes class via
 - sched_setscheduler syscall
- Each class needs
 - New sched_class structure implementing scheduling functions
 - New sched_entity in the task_struct

OS Schedulers

Operating System	Preemption	Algorithm
Amiga OS	Yes	Prioritized round-robin scheduling
FreeBSD	Yes	Multilevel feedback queue
Linux kernel before 2.6.0	Yes	Multilevel feedback queue
Linux kernel 2.6.0–2.6.23	Yes	O(1) scheduler
Linux kernel after 2.6.23	Yes	Completely Fair Scheduler
classic Mac OS pre-9	None	Cooperative scheduler
Mac OS 9	Some	Preemptive scheduler for MP tasks, and cooperative for processes and threads
macOS	Yes	Multilevel feedback queue
NetBSD	Yes	Multilevel feedback queue
Solaris	Yes	Multilevel feedback queue
Windows 3.1x	None	Cooperative scheduler
Windows 95, 98, Me	Half	Preemptive scheduler for 32-bit processes, and cooperative for 16-bit processes
Windows NT (including 2000, XP, Vista, 7, and Server)	Yes	Multilevel feedback queue

A fun read

- The Linux Scheduler: A Decade of Wasted Cores
 - <https://people.ece.ubc.ca/sasha/papers/eurosys16-final29.pdf>
- Talks about performance bugs in multi-core version of Completely Fair Scheduler and how they fixed them

Reading

- Read Chapter 16.5 Linux Scheduling
- Read Chapter 5
- Read Chapter 4 (Linux Kernel Development)
- Acknowledgments
 - These slides are adapted from
 - Öznur Özkasap (Koç University)
 - Operating System and Concepts (9th edition) Wiley
 - Linux Scheduling
 - Linux Overview. COMS W4118 Spring 2008 slideserve.com
 - Prof. Kaustubh R. Joshi from Columbia
 - <http://www.algorithmsandme.com/2014/03/scheduling-o1-and-completely-fair.html#.VPgpMZLOWc>

CPU Scheduling Algorithms

Didem Unat
Lecture 7

COMP304 - Operating Systems (OS)

Project 1

- Project 1 will be up in two days
- There is a PS on Monday related to Project 1
- Fill the project partner form online
- Start EARLY
 - You won't finish the project in a week
 - Do not ask for an extension
 - You will encounter a lot of issues related to your system
- Add to your CV

Terminology

- CPU utilization – keep the CPU as busy as possible
- Throughput – # of processes that complete their execution per time unit
- Turnaround time – amount of time to execute a particular process
- 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)

<u>Process</u>	<u>Burst Time</u>
P ₁	24
P ₂	3
P ₃	3

- Suppose that the processes arrive in the order: P₁, P₂, P₃
The Gantt Chart for the schedule is:



- Waiting times for: P₁ = 0; P₂ = 24; P₃ = 27
- Average waiting time: $(0 + 24 + 27)/3 = 17$

FCFS Scheduling (Cont.)

Suppose that the processes arrive in the order

$$P_2, P_3, P_1$$

- The Gantt chart for the schedule is:



- Waiting time for P₁ = 6; P₂ = 0; P₃ = 3
- Average waiting time: $(6 + 0 + 3)/3 = 3$
- Much better than previous case.
- **Convoy effect:** short process behind long process

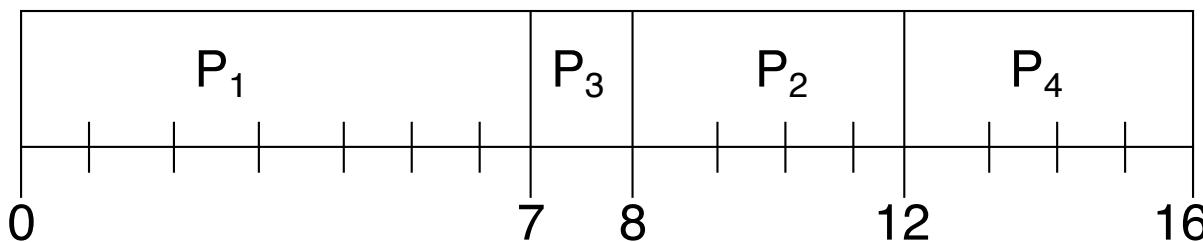
2. Shortest Job First (SJF) Scheduling

- 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.

Non-preemptive (SJF) Scheduling

<u>Process</u>	<u>Arrival Time</u>	<u>Burst Time</u>
P ₁	0.0	7
P ₂	2.0	4
P ₃	4.0	1
P ₄	5.0	4

- SJF (non-preemptive)

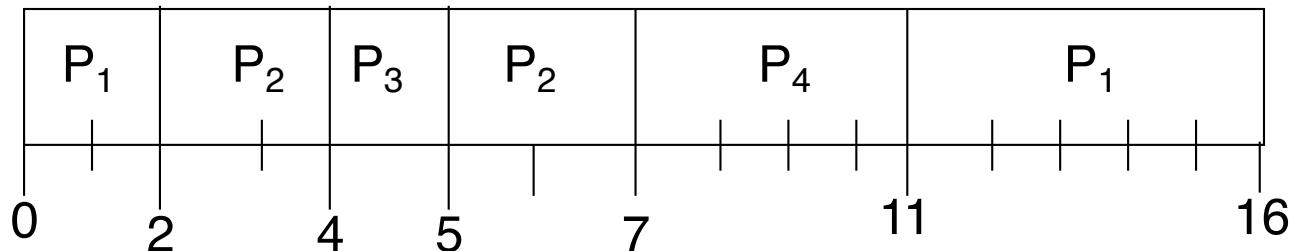


- Average waiting time ?
 $= (0 + 6 + 3 + 7)/4 = 4$

Example of Preemptive SJF

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 (preemptive)



- Average waiting time ?
 $= (9 + 1 + 0 + 2)/4 = 3$

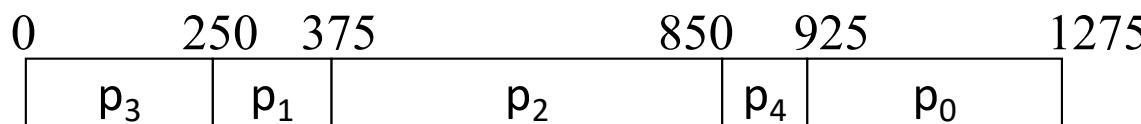
3. Priority Scheduling

- A **priority** number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority (smallest integer \equiv 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.

Priority Scheduling

Pid	$\tau(p_i)$	Priority
0	350	5
1	125	2
2	475	3
3	250	1
4	75	4

- Nonpreemptive example



$$T_{TRnd}(p_0) = \tau(p_0) + \tau(p_4) + \tau(p_2) + \tau(p_1) + \tau(p_3) = 350 + 75 + 475 + 125 + 250 = 1275 \quad R(p_0) = 925$$

$$T_{TRnd}(p_1) = \tau(p_1) + \tau(p_3) = 125 + 250 = 375$$

$$T_{TRnd}(p_2) = \tau(p_2) + \tau(p_1) + \tau(p_3) = 475 + 125 + 250 = 850$$

$$T_{TRnd}(p_3) = \tau(p_3) = 250$$

$$T_{TRnd}(p_4) = \tau(p_4) + \tau(p_2) + \tau(p_1) + \tau(p_3) = 75 + 475 + 125 + 250 = 925$$

$$\text{Average response time } R_{avg} = (925 + 250 + 375 + 0 + 850) / 5 = 2400 / 5 = 480$$

Problem with Priority Scheduling

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



IBM 7094 operator's console

Priority

- What if a high-priority process needs to access the data that is currently being held by a low- priority process?
- The high-priority process is blocked by the low- priority process. This is **priority inversion**.
- This can be solved with **priority-inheritance protocol**.
 - The low priority process accessing the data inherits the high priority until it is done with the resource.
 - When the low-priority process finishes, its priority reverts back to the original.

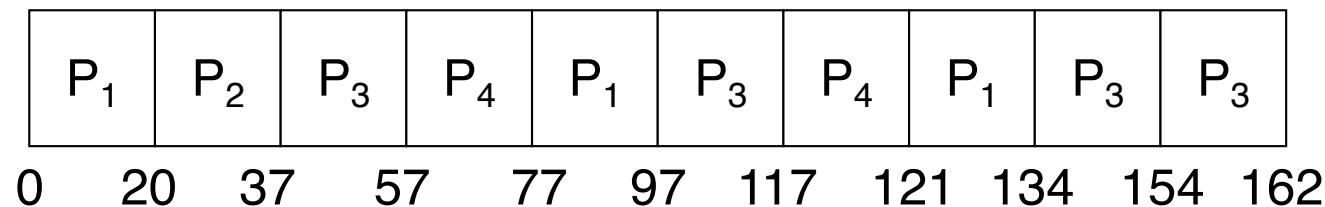
4. Round Robin (RR)

- 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.

Example of RR with Time Quantum = 20

<u>Process</u>	<u>Burst Time</u>
P ₁	53
P ₂	17
P ₃	68
P ₄	24

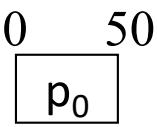
- The Gantt chart is:



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

Round Robin (Time Quantum=50)

i	$\tau(p_i)$
0	350
1	125
2	475
3	250
4	75

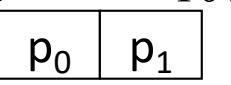


A Gantt chart illustrating the execution of process p_0 . The horizontal axis represents time, starting at 0 and ending at 50. A single horizontal bar represents the execution of p_0 , which begins at time 0 and ends at time 50.

$$R(p_0) = 0$$

Round Robin (Time Quantum=50)

i	$\tau(p_i)$
0	350
1	125
2	475
3	250
4	75

0 100


$$R(p_0) = 0$$

$$R(p_1) = 50$$

Round Robin (Time Quantum=50)

i $\tau(p_i)$

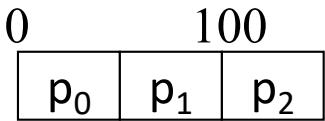
0 350

1 125

2 475

3 250

4 75



$$R(p_0) = 0$$

$$R(p_1) = 50$$

$$R(p_2) = 100$$

Round Robin (Time Quantum=50)

i	$\tau(p_i)$
0	350
1	125
2	475
3	250
4	75

0 100 200 300

p₀ | p₁ | p₂ | p₃ | p₄ | p₀

$$R(p_0) = 0$$

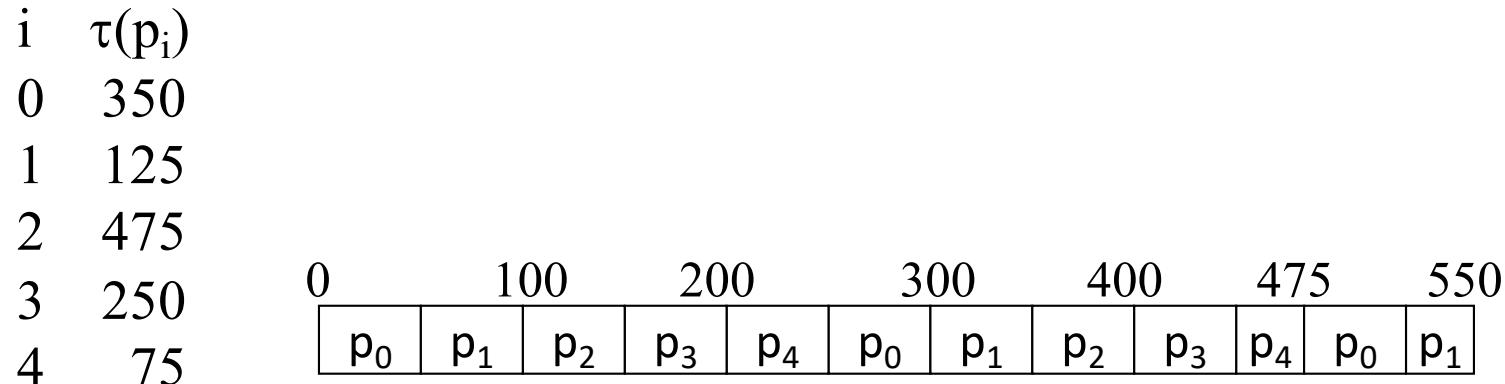
$$R(p_1) = 50$$

$$R(p_2) = 100$$

$$R(p_3) = 150$$

$$R(p_4) = 200$$

Round Robin (Time Quantum=50)



$$\begin{array}{ll} R(p_0) = 0 & \\ T_{TRnd}(p_1) = 550 & \begin{array}{l} R(p_1) = 50 \\ R(p_2) = 100 \\ R(p_3) = 150 \\ R(p_4) = 200 \end{array} \\ T_{TRnd}(p_4) = 475 & \end{array}$$

Round Robin (Time Quantum=50)

i $\tau(p_i)$

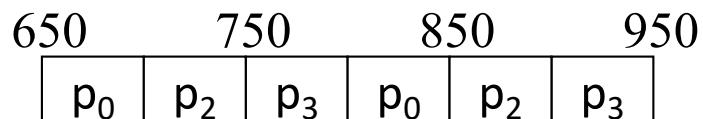
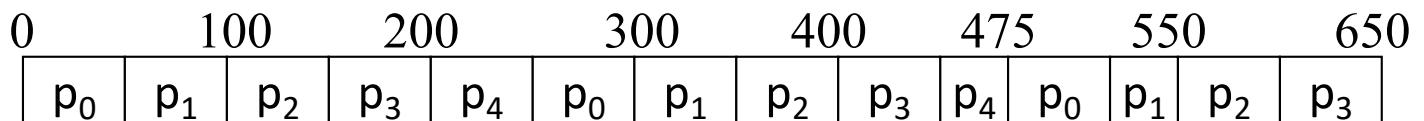
0 350

1 125

2 475

3 250

4 75



$$T_{TRnd}(p_1) = 550$$

$$T_{TRnd}(p_3) = 950$$

$$T_{TRnd}(p_4) = 475$$

$$R(p_0) = 0$$

$$R(p_1) = 50$$

$$R(p_2) = 100$$

$$R(p_3) = 150$$

$$R(p_4) = 200$$

Round Robin (Time Quantum=50)

i $\tau(p_i)$

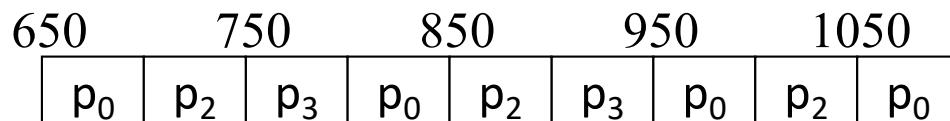
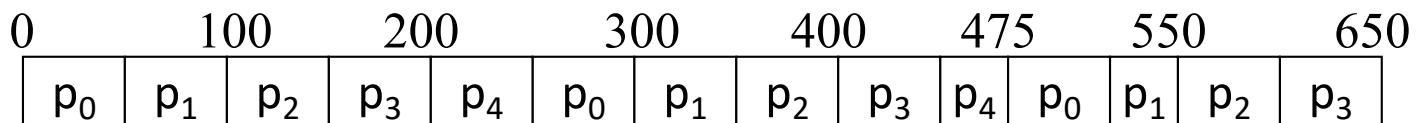
0 350

1 125

2 475

3 250

4 75



$$T_{TRnd}(p_0) = 1100$$

$$T_{TRnd}(p_1) = 550$$

$$T_{TRnd}(p_3) = 950$$

$$T_{TRnd}(p_4) = 475$$

$$R(p_0) = 0$$

$$R(p_1) = 50$$

$$R(p_2) = 100$$

$$R(p_3) = 150$$

$$R(p_4) = 200$$

Round Robin (Time Quantum=50)

i $\tau(p_i)$

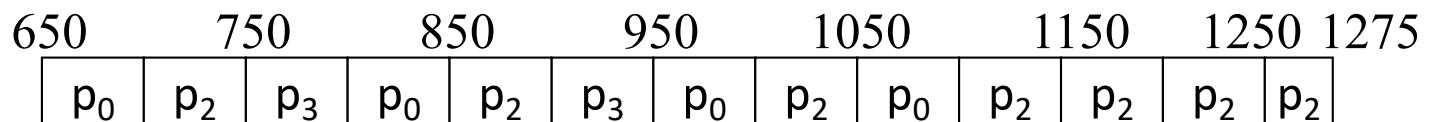
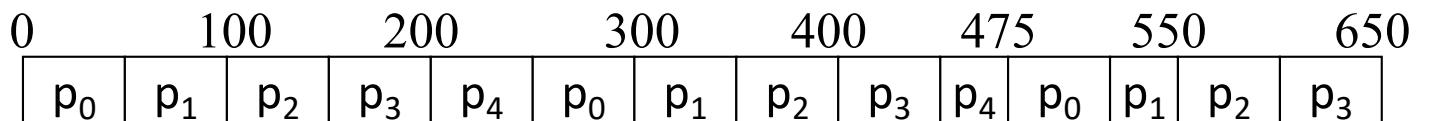
0 350

1 125

2 475

3 250

4 75



$$T_{TRnd}(p_0) = 1100$$

$$T_{TRnd}(p_1) = 550$$

$$T_{TRnd}(p_2) = 1275$$

$$T_{TRnd}(p_3) = 950$$

$$T_{TRnd}(p_4) = 475$$

$$R(p_0) = 0$$

$$R(p_1) = 50$$

$$R(p_2) = 100$$

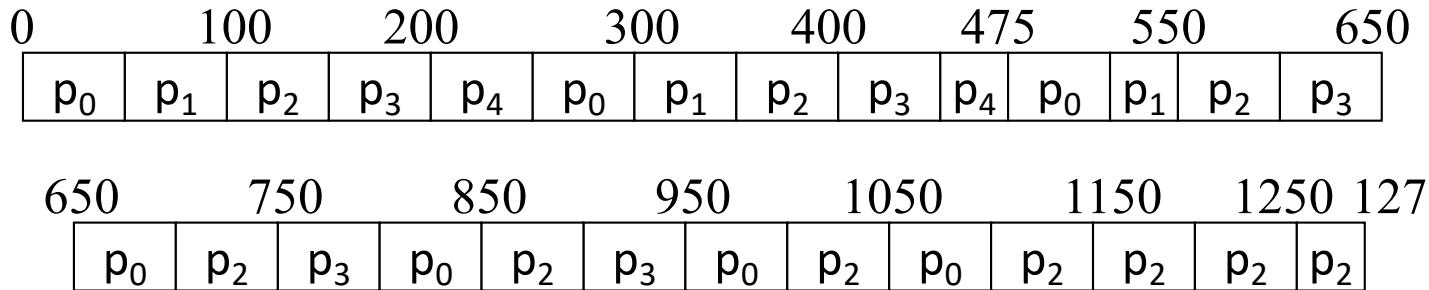
$$R(p_3) = 150$$

$$R(p_4) = 200$$

Round Robin (Time Quantum=50)

- Equitable
- Most widely-used

i	$\tau(p_i)$
0	350
1	125
2	475
3	250
4	75



$$T_{TRnd}(p_0) = 1100$$

$$R(p_0) = 0$$

$$T_{TRnd}(p_1) = 550$$

$$R(p_1) = 50$$

$$T_{TRnd}(p_2) = 1275$$

$$R(p_2) = 100$$

$$T_{TRnd}(p_3) = 950$$

$$R(p_3) = 150$$

$$T_{TRnd}(p_4) = 475$$

$$R(p_4) = 200$$

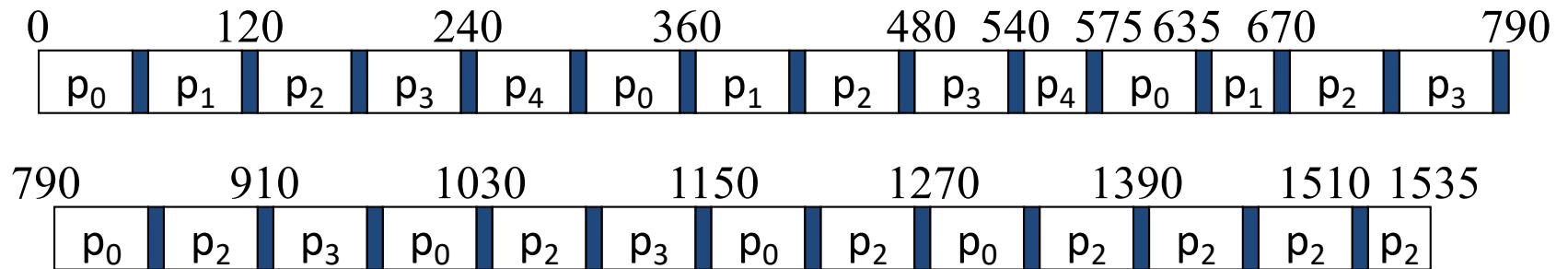
$$T_{TRnd_avg} = (1100 + 550 + 1275 + 950 + 475) / 5 = 4350 / 5 = 870$$

$$\text{Average response (wait) time } R_{avg} = (0 + 50 + 100 + 150 + 200) / 5 = 500 / 5 = 100$$

RR (TQ=50, Overhead =10)

i	$\tau(p_i)$
0	350
1	125
2	475
3	250
4	75

•Context Switch Overhead must be considered



$$T_{TRnd}(p_0) = 1320$$

$$R(p_0) = 0$$

$$T_{TRnd}(p_1) = 660$$

$$R(p_1) = 60$$

$$T_{TRnd}(p_2) = 1535$$

$$R(p_2) = 120$$

$$T_{TRnd}(p_3) = 1140$$

$$R(p_3) = 180$$

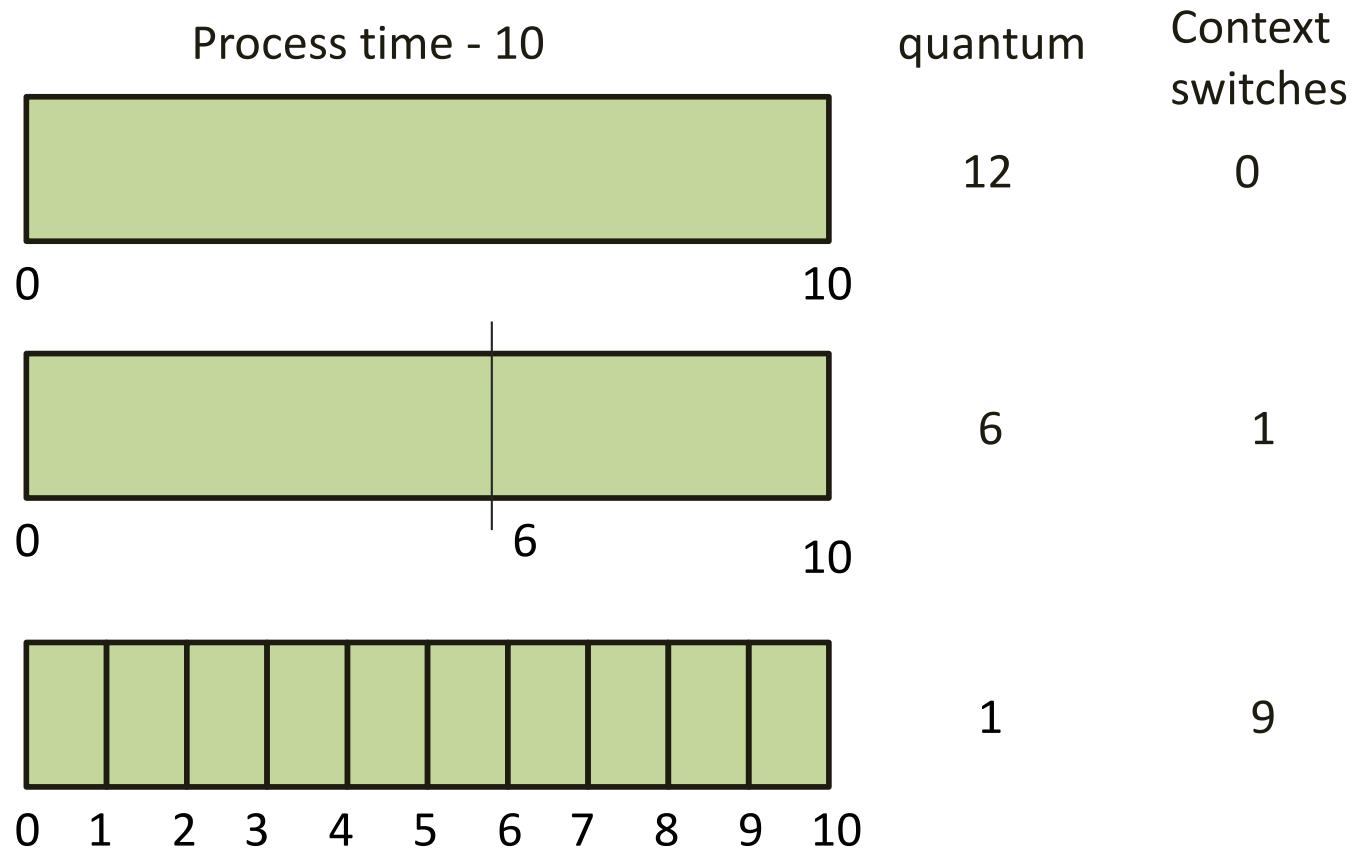
$$T_{TRnd}(p_4) = 565$$

$$R(p_4) = 240$$

$$T_{TRnd_avg} = (1320 + 660 + 1535 + 1140 + 565)/5 = 5220/5 = 1044$$

$$\text{Average response (wait) time } R_{avg} = (0 + 60 + 120 + 180 + 240)/5 = 600/5 = 120$$

Time Quantum and Context Switch Time



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

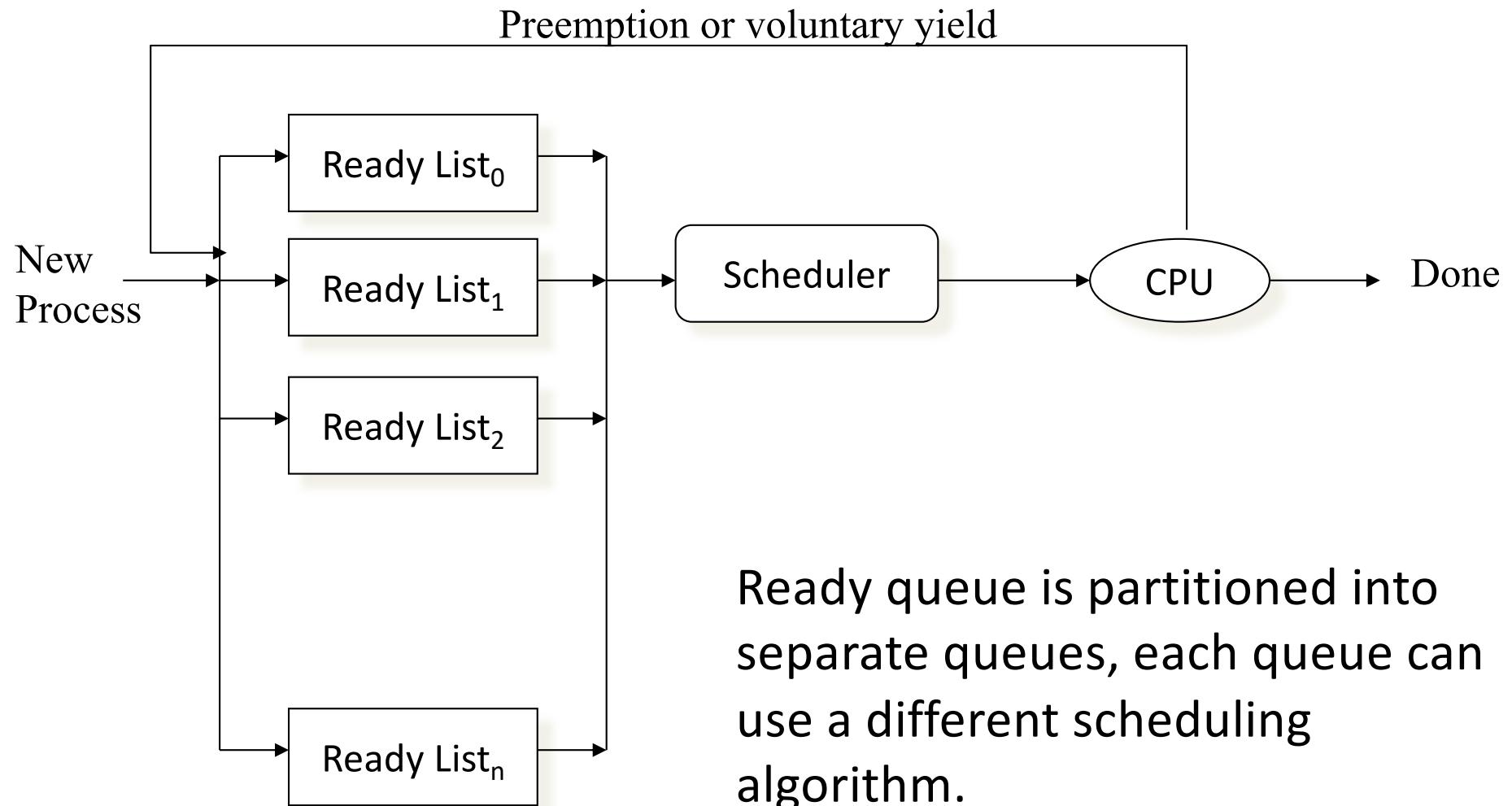
RR algorithm

- Consider a variant of the RR scheduling algorithm where the entries in the ready queue are pointers to the PCBs.
- What would be the effect of putting two pointers to the same process in the ready queue?
- How would you modify the basic RR algorithm to achieve the same effect without the duplicate pointers?

5. Multilevel Queue

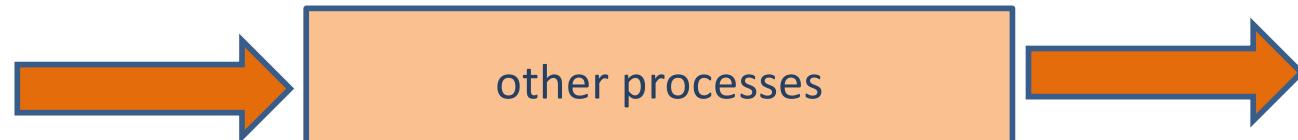
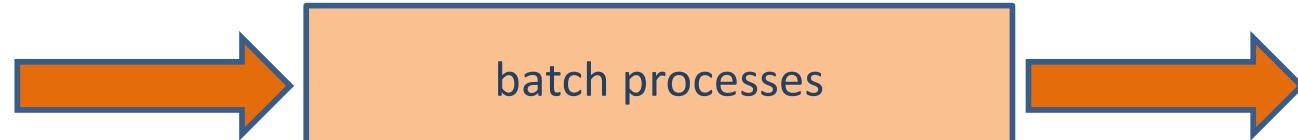
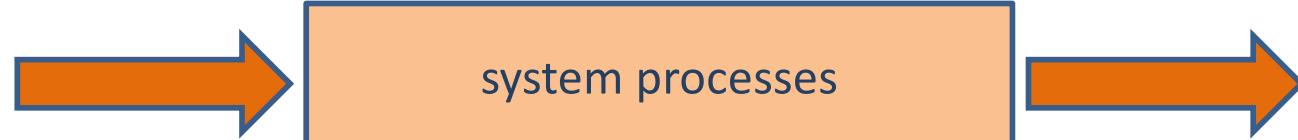
- Ready queue is partitioned into separate queues:
foreground (interactive)
background (batch)
- Each queue has its own scheduling algorithm,
foreground – RR
background – FCFS
- Scheduling must be done between the queues.
 - **Fixed priority scheduling** – (i.e., serve all from foreground then from background). 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

Multi-Level Queues



Example- Multi-level Queue

Highest priority



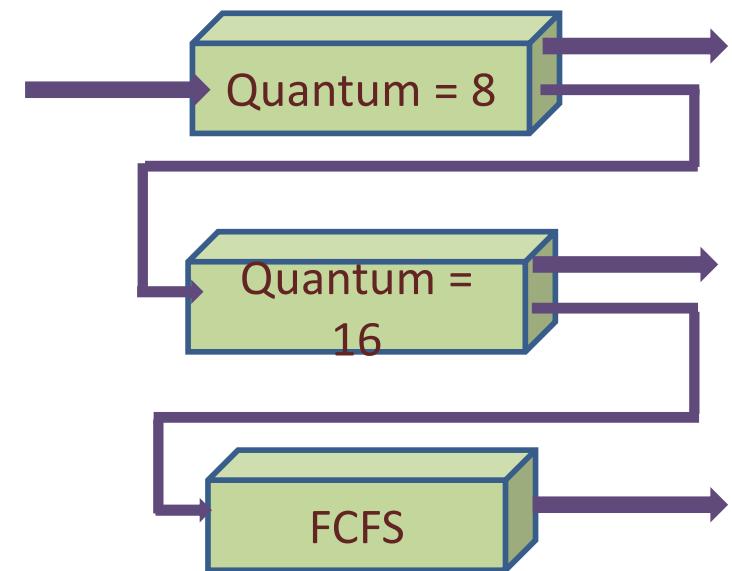
Lowest priority

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

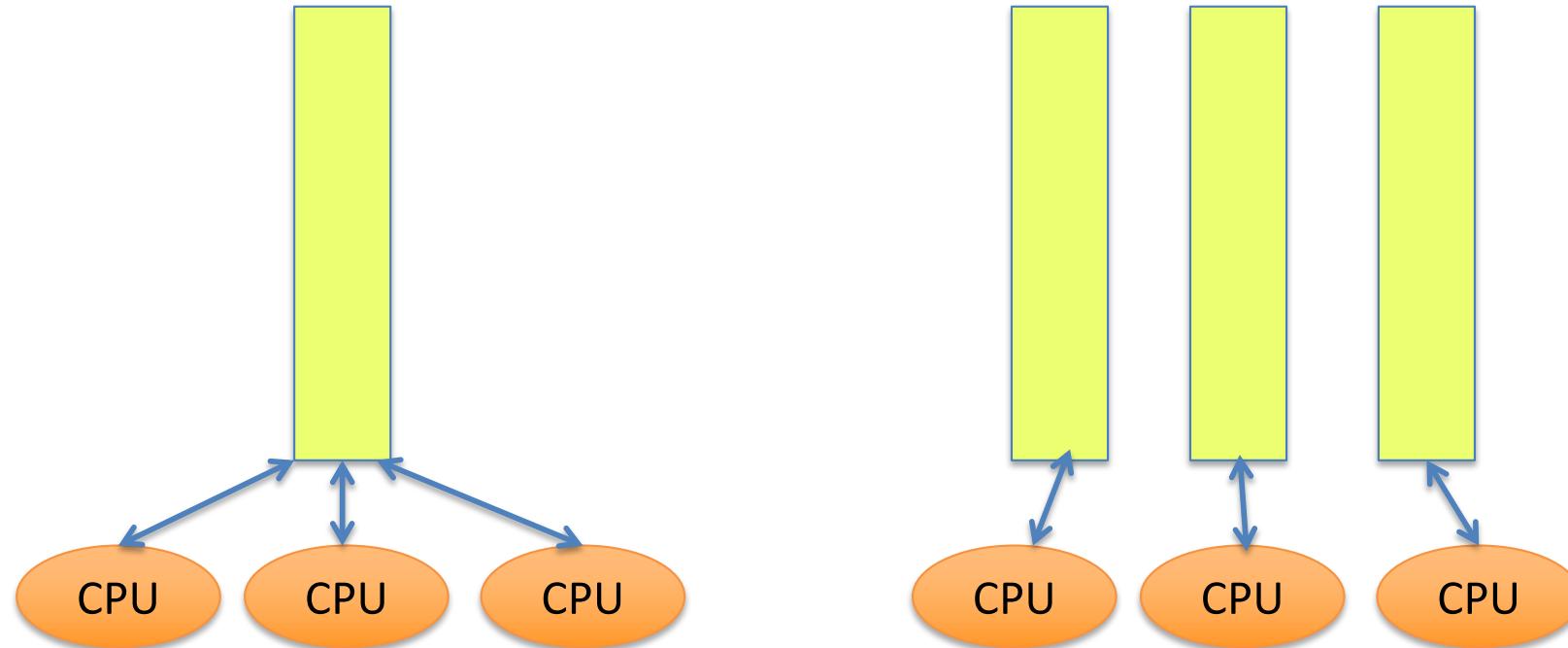
- Three queues:
 - Q_0 – time quantum 8 milliseconds
 - Q_1 – time quantum 16 milliseconds
 - Q_2 – FCFS non-preemption
- Example Scheduling
 - A new job enters queue Q_0 . When it gains CPU, job receives 8 milliseconds. If it does not finish in 8 milliseconds, job is moved to queue Q_1 .
 - At Q_1 job 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**

Multicore Scheduling



- In SMP, each CPU has its own private ready queue for the processes waiting to be run on that CPU

Process Migration

- If SMP, need to keep all CPUs loaded for efficiency
- **Load balancing** attempts to keep workload evenly distributed
- **Push migration** – periodically checks load on each processor, and if found any of them overloaded, pushes task from overloaded CPU to other CPUs
- **Pull migration** – idle processors pull waiting tasks from busy processor

Starvation

- Which of the following scheduling algorithms could result in starvation?
 - First-come, first serve
 - Shortest job first
 - Round robin
 - Priority

Scheduling in Linux

- How does Linux schedule processes?

Reading

- Read Chapter 5 (Textbook)
- Read Chapter 4 (Linux Kernel Development)
- Acknowledgments
 - These slides are adapted from
 - Öznur Özkasap (Koç University)
 - Operating System and Concepts (9th edition) Wiley

CPU Scheduling

Didem Unat
Lecture 6

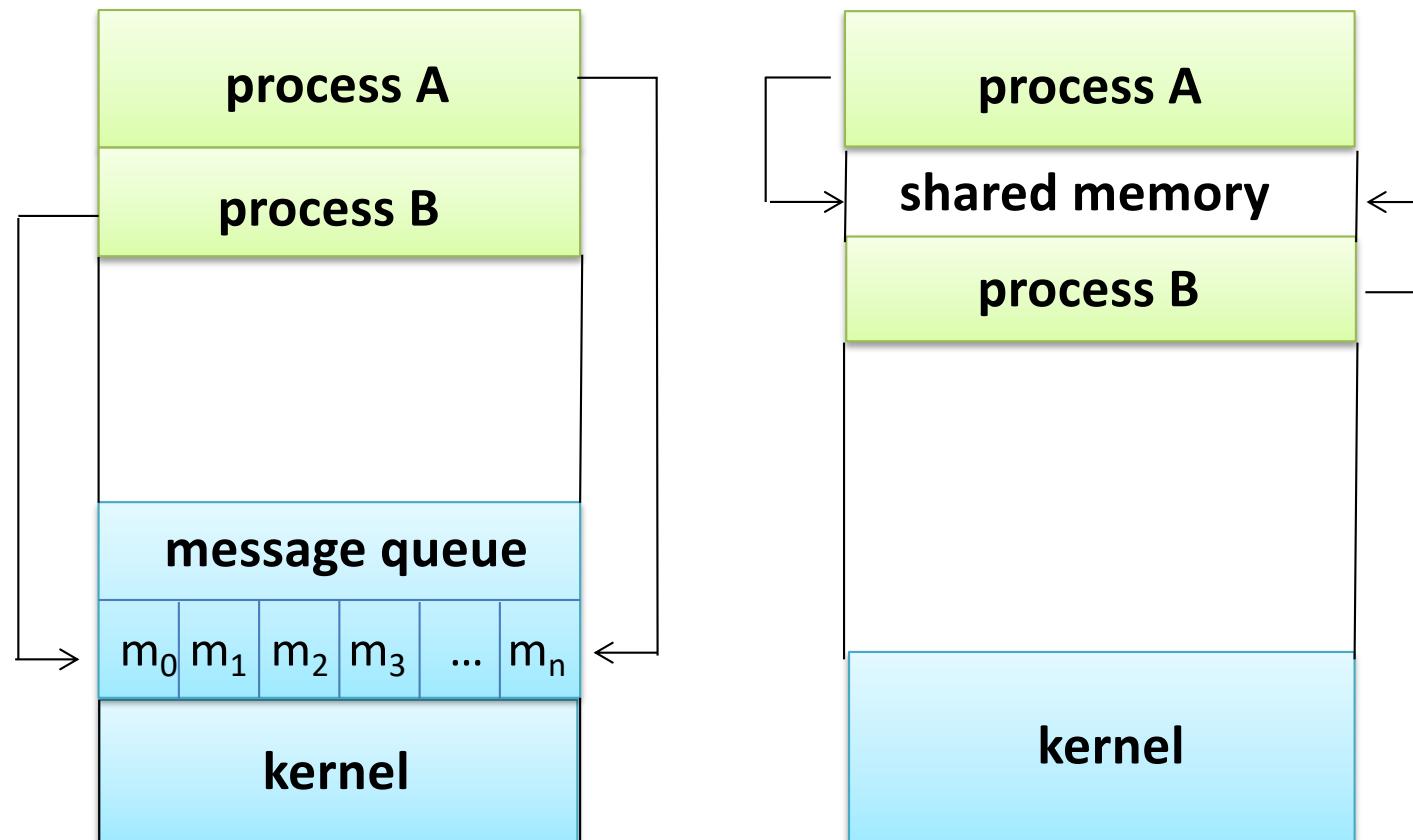
COMP304 - Operating Systems (OS)

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 another processes
- Cooperating processes need **inter-process communication**
- Two models of IPC
 - **Shared memory**
 - **Message passing**

Two Models of Communication

Message Passing vs Shared Memory



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

Scheduling

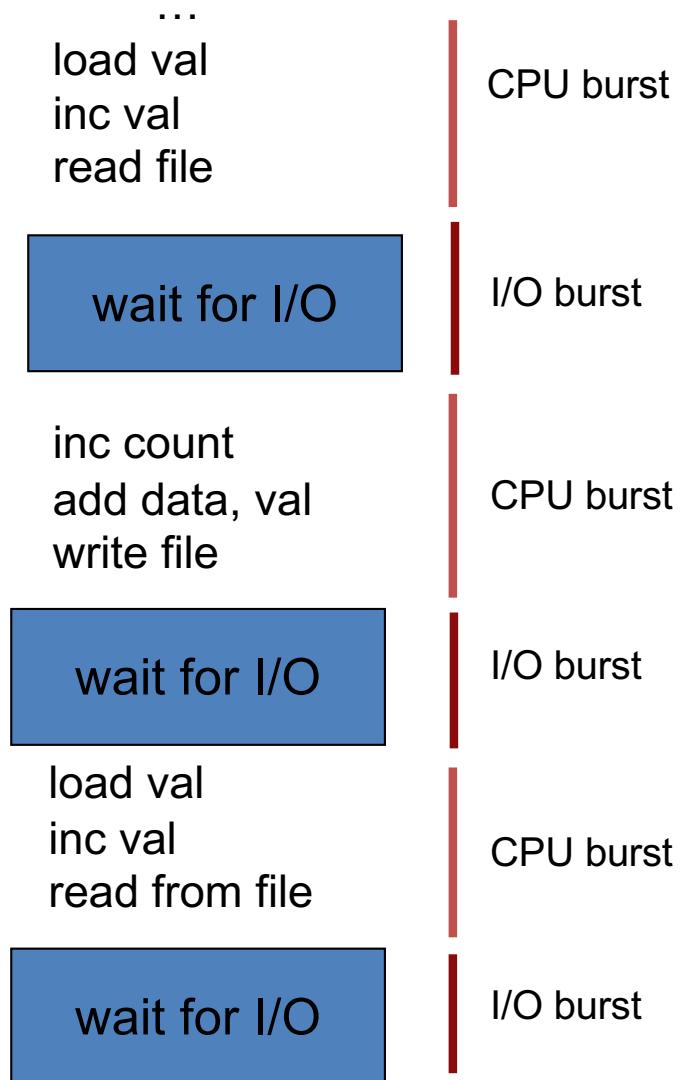
- One of the main tasks of an OS is to schedule processes to execute.
- Long-term scheduler (or job scheduler) – selects which processes should be brought into the ready queue.
- Short-term scheduler (or CPU scheduler) – selects which process should be executed next and allocates CPU to that process.

Schedulers

- Short-term scheduler is invoked very frequently (milliseconds)
 - ⇒ must be fast.
- Long-term scheduler is invoked very infrequently (seconds, minutes)
 - ⇒ can be slow.
- The long-term scheduler controls the *degree of multiprogramming*.

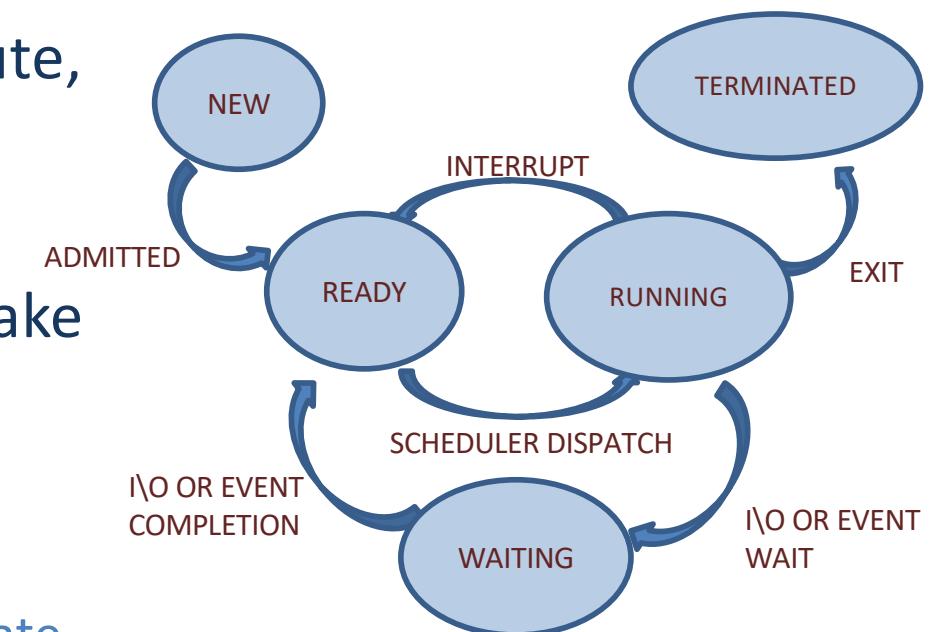
CPU-I/O Burst Cycle

- CPU-I/O Burst Cycle – Process execution consists of a *cycle* of CPU execution and I/O wait
- Processes can be described as either:
 - *I/O-bound process* – spends more time doing I/O than computations; many, short CPU bursts.
 - *CPU-bound process* – spends more time doing computations; few, very long CPU bursts.



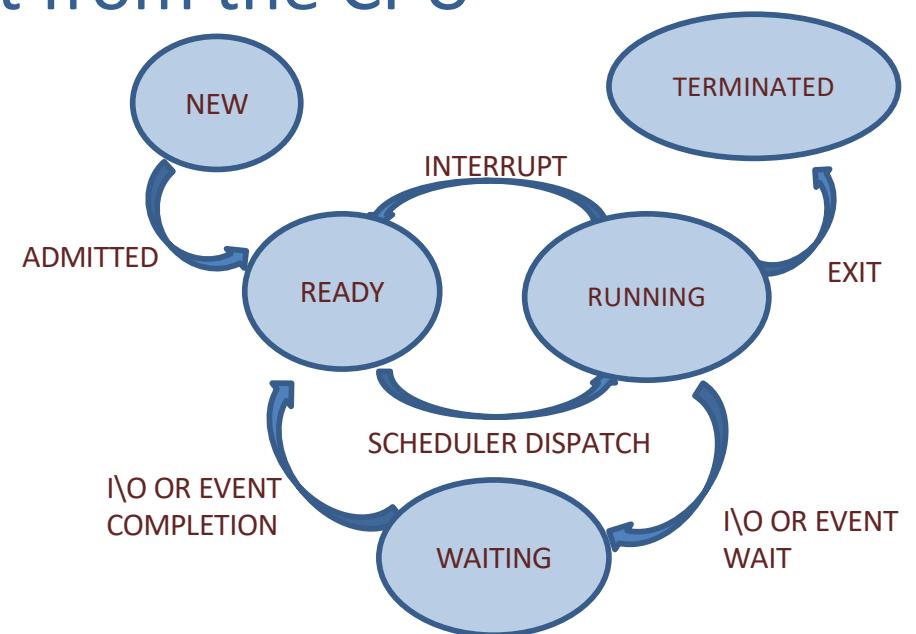
CPU Scheduler

- 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)

Scheduling

- Let $P = \{p_i \mid 0 \leq i < n\}$ = set of processes
- Let $S(p_i) \in \{\text{running, ready, waiting}\}$
- Let $t(p_i)$ = Time process needs to be in running state (the service time, CPU burst)
- Let $T_{TRnd}(p_i)$ = Time from p_i first enters ready to last exits system (turnaround time)
- Batch Throughput rate = inverse of avg T_{TRnd}
- Let $R(p_i)$ = Time p_i is in ready state before first transition to running (or response time) (different than “waiting time”)

Optimization Criteria

- Max CPU utilization
- Max throughput
- Min turnaround time
- Min waiting time
- Min response time

It is important to minimize the variance in response time than minimize the average response time – provides fairness

Dispatcher

- **Dispatcher** module is part of the OS that gives control of the CPU to the process selected by the short-term scheduler; this involves:
 - switching context
 - switching to user mode
 - jumping to the proper location in the user program to restart that program
- *Dispatch latency* – time it takes for the dispatcher to stop one process and start another running.

Scheduling Algorithms

If you were the OS, how would you decide who should run next?

First-Come, First Served (FCFS)

<u>Process</u>	<u>Burst Time</u>
P ₁	24
P ₂	3
P ₃	3

- Suppose that the processes arrive in the order: P₁, P₂, P₃
The Gantt Chart for the schedule is:



- Waiting times for: P₁ = 0; P₂ = 24; P₃ = 27
- Average waiting time: $(0 + 24 + 27)/3 = 17$

FCFS Scheduling (Cont.)

Suppose that the processes arrive in the order

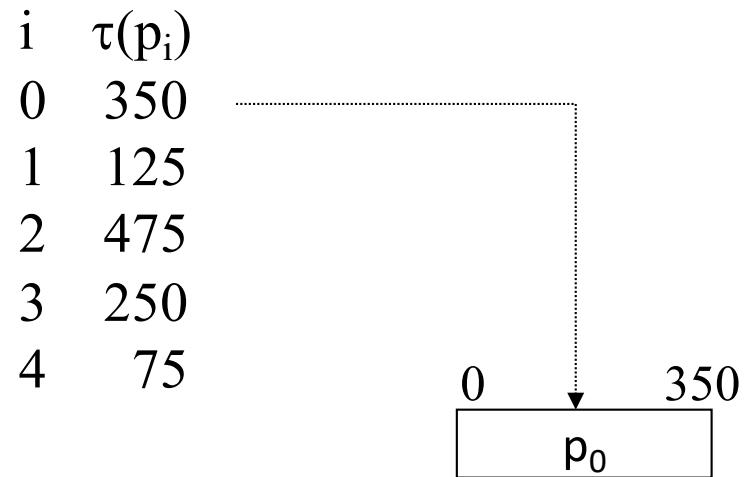
$$P_2, P_3, P_1$$

- The Gantt chart for the schedule is:



- Waiting time for P₁ = 6; P₂ = 0; P₃ = 3
- Average waiting time: $(6 + 0 + 3)/3 = 3$
- Much better than previous case.
- **Convoy effect:** short process behind a long process

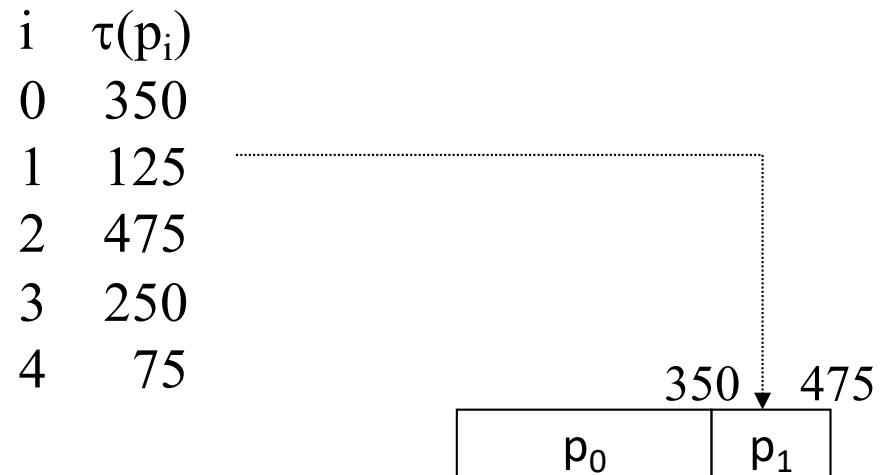
FCFS Scheduling (Cont.)



$$T_{TRnd}(p_0) = \tau(p_0) = 350$$

$$R(p_0) = 0$$

FCFS Scheduling (Cont.)



$$T_{TRnd}(p_0) = \tau(p_0) = 350$$

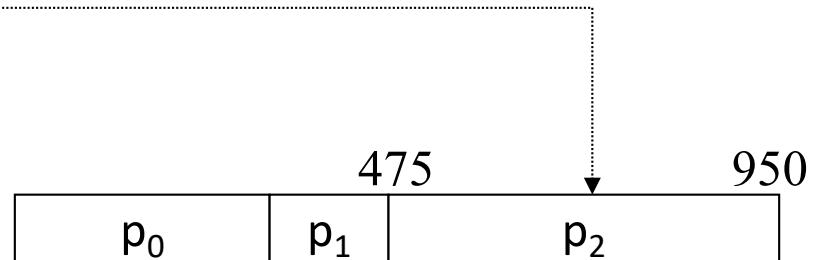
$$T_{TRnd}(p_1) = (\tau(p_1) + T_{TRnd}(p_0)) = 125 + 350 = 475$$

$$R(p_0) = 0$$

$$R(p_1) = T_{TRnd}(p_0) = 350$$

FCFS Scheduling (Cont.)

i	$\tau(p_i)$
0	350
1	125
2	475
3	250
4	75



$$T_{TRnd}(p_0) = \tau(p_0) = 350$$

$$R(p_0) = 0$$

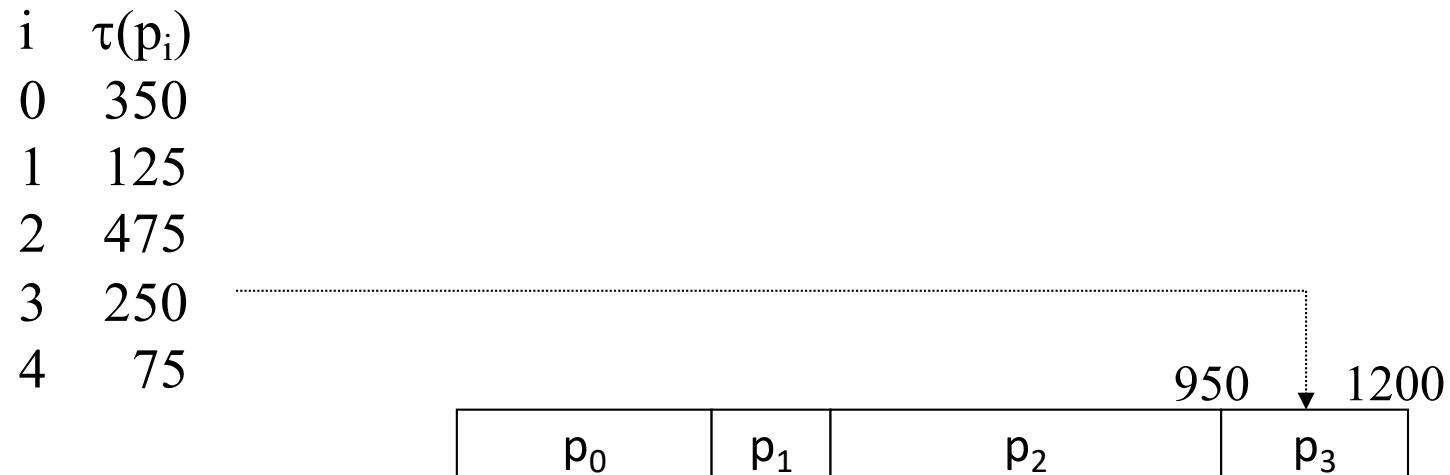
$$T_{TRnd}(p_1) = (\tau(p_1) + T_{TRnd}(p_0)) = 125 + 350 = 475$$

$$R(p_1) = T_{TRnd}(p_0) = 350$$

$$T_{TRnd}(p_2) = (\tau(p_2) + T_{TRnd}(p_1)) = 475 + 475 = 950$$

$$R(p_2) = T_{TRnd}(p_1) = 475$$

FCFS Scheduling (Cont.)



$$T_{TRnd}(p_0) = \tau(p_0) = 350$$

$$R(p_0) = 0$$

$$T_{TRnd}(p_1) = (\tau(p_1) + T_{TRnd}(p_0)) = 125 + 350 = 475$$

$$R(p_1) = T_{TRnd}(p_0) = 350$$

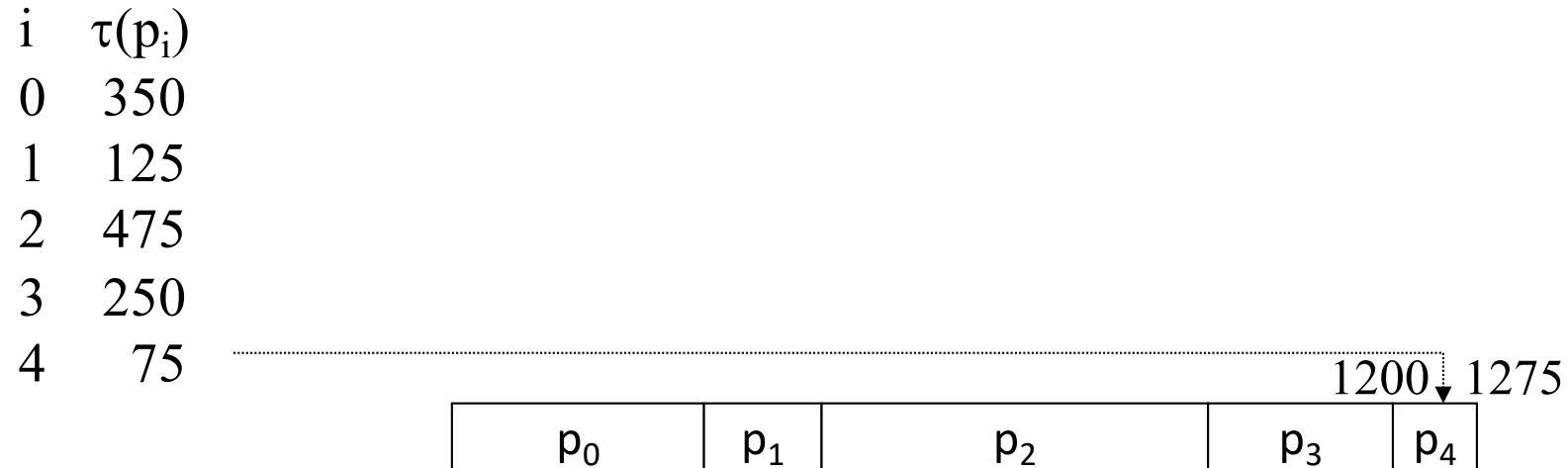
$$T_{TRnd}(p_2) = (\tau(p_2) + T_{TRnd}(p_1)) = 475 + 475 = 950$$

$$R(p_2) = T_{TRnd}(p_1) = 475$$

$$T_{TRnd}(p_3) = (\tau(p_3) + T_{TRnd}(p_2)) = 250 + 950 = 1200$$

$$R(p_3) = T_{TRnd}(p_2) = 950$$

FCFS Scheduling (Cont.)



$$T_{TRnd}(p_0) = \tau(p_0) = 350$$

$$T_{TRnd}(p_1) = (\tau(p_1) + T_{TRnd}(p_0)) = 125 + 350 = 475$$

$$T_{TRnd}(p_2) = (\tau(p_2) + T_{TRnd}(p_1)) = 475 + 475 = 950$$

$$T_{TRnd}(p_3) = (\tau(p_3) + T_{TRnd}(p_2)) = 250 + 950 = 1200$$

$$T_{TRnd}(p_4) = (\tau(p_4) + T_{TRnd}(p_3)) = 75 + 1200 = 1275$$

$$R(p_0) = 0$$

$$R(p_1) = T_{TRnd}(p_0) = 350$$

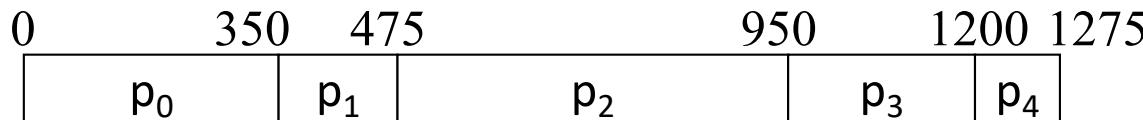
$$R(p_2) = T_{TRnd}(p_1) = 475$$

$$R(p_3) = T_{TRnd}(p_2) = 950$$

$$R(p_4) = T_{TRnd}(p_3) = 1200$$

FCFS Scheduling- Average Wait Time

i	$\tau(p_i)$
0	350
1	125
2	475
3	250
4	75



- Easy to implement
- Not a great performer
- Non-preemptive

$$T_{TRnd}(p_0) = \tau(p_0) = 350$$

$$R(p_0) = 0$$

$$T_{TRnd}(p_1) = (\tau(p_1) + T_{TRnd}(p_0)) = 125 + 350 = 475$$

$$R(p_1) = T_{TRnd}(p_0) = 350$$

$$T_{TRnd}(p_2) = (\tau(p_2) + T_{TRnd}(p_1)) = 475 + 475 = 950$$

$$R(p_2) = T_{TRnd}(p_1) = 475$$

$$T_{TRnd}(p_3) = (\tau(p_3) + T_{TRnd}(p_2)) = 250 + 950 = 1200$$

$$R(p_3) = T_{TRnd}(p_2) = 950$$

$$T_{TRnd}(p_4) = (\tau(p_4) + T_{TRnd}(p_3)) = 75 + 1200 = 1275$$

$$R(p_4) = T_{TRnd}(p_3) = 1200$$

$$\text{Average response (wait) time } R_{avg} = (0+350+475+950+1200)/5 = 2974/5 = 595$$

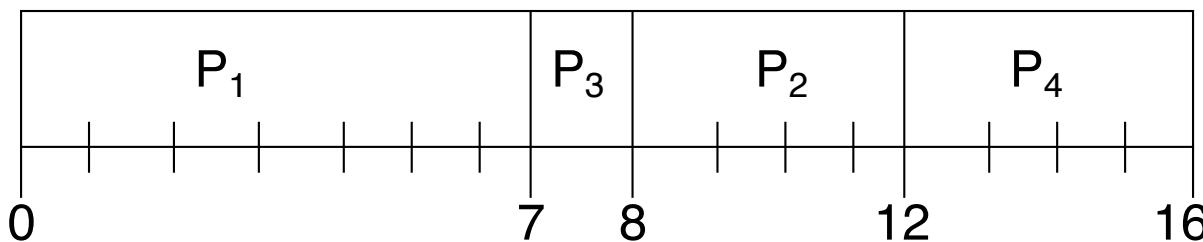
2. Shortest Job First (SJF) Scheduling

- 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.

Non-preemptive (SJF) Scheduling

<u>Process</u>	<u>Arrival Time</u>	<u>Burst Time</u>
P ₁	0.0	7
P ₂	2.0	4
P ₃	4.0	1
P ₄	5.0	4

- SJF (non-preemptive)

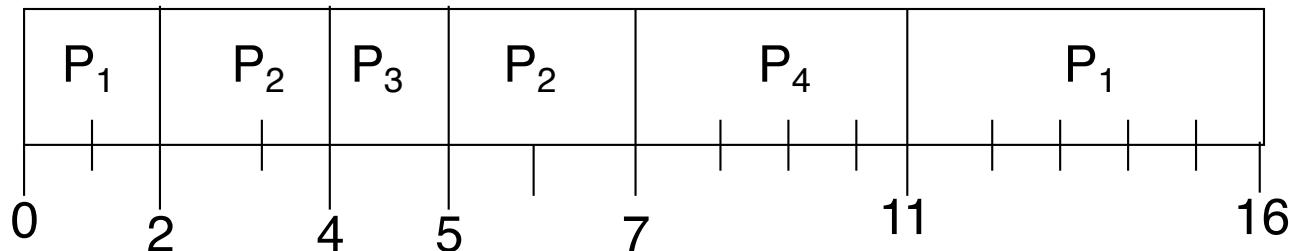


- Average waiting time ?
 $= (0 + 6 + 3 + 7)/4 = 4$

Example of Preemptive SJF

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 (preemptive)



- Average waiting time ?
 $= (9 + 1 + 0 + 2)/4 = 3$

Example of SJF

i	$\tau(p_i)$
0	350
1	125
2	475
3	250
4	75

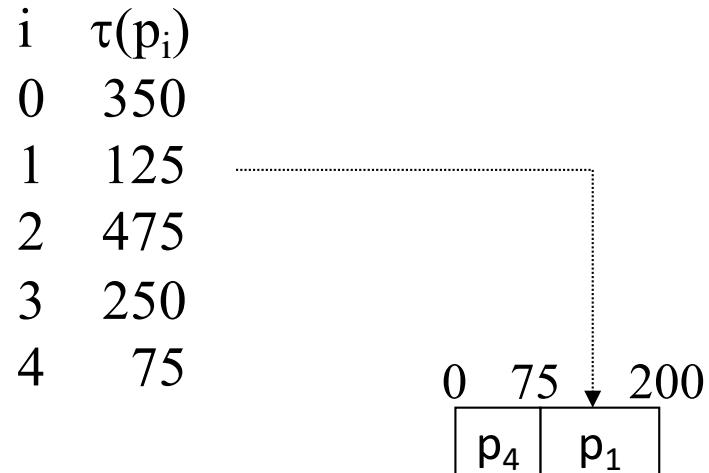
.....



$$T_{TRnd}(p_4) = \tau(p_4) = 75$$

$$R(p_4) = 0$$

Example of SJF



$$T_{TRnd}(p_1) = \tau(p_1) + \tau(p_4) = 125 + 75 = 200$$

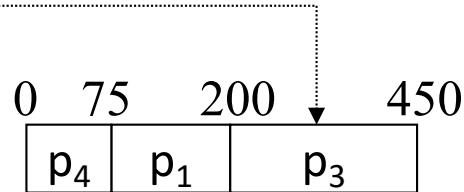
$$R(p_1) = 75$$

$$T_{TRnd}(p_4) = \tau(p_4) = 75$$

$$R(p_4) = 0$$

Example of SJF

i	$\tau(p_i)$
0	350
1	125
2	475
3	250
4	75



$$T_{TRnd}(p_1) = \tau(p_1) + \tau(p_4) = 125 + 75 = 200$$

$$R(p_1) = 75$$

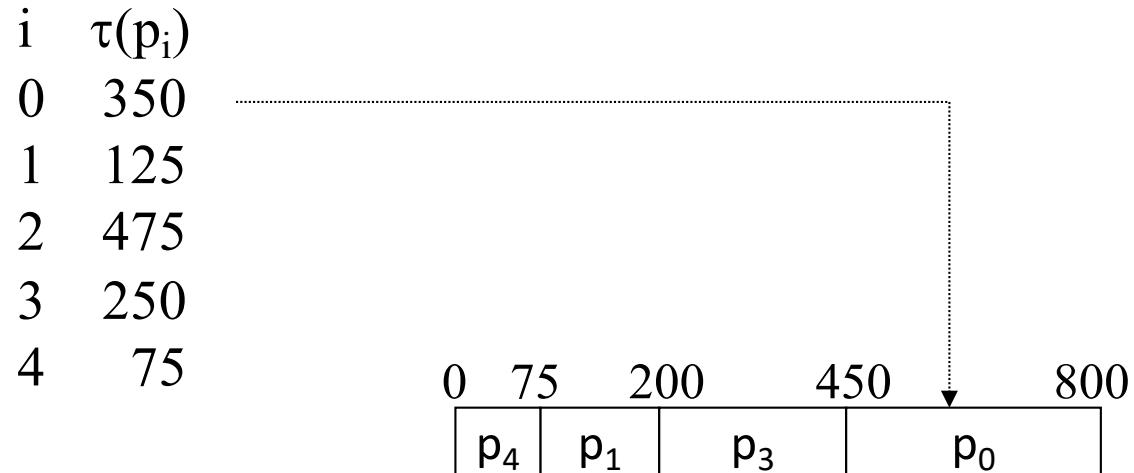
$$T_{TRnd}(p_3) = \tau(p_3) + \tau(p_1) + \tau(p_4) = 250 + 125 + 75 = 450$$

$$R(p_3) = 200$$

$$T_{TRnd}(p_4) = \tau(p_4) = 75$$

$$R(p_4) = 0$$

Example of SJF



$$T_{TRnd}(p_0) = \tau(p_0) + \tau(p_3) + \tau(p_1) + \tau(p_4) = 350 + 250 + 125 + 75 = 800$$

$$T_{TRnd}(p_1) = \tau(p_1) + \tau(p_4) = 125 + 75 = 200$$

$$R(p_0) = 450$$

$$R(p_1) = 75$$

$$T_{TRnd}(p_3) = \tau(p_3) + \tau(p_1) + \tau(p_4) = 250 + 125 + 75 = 450$$

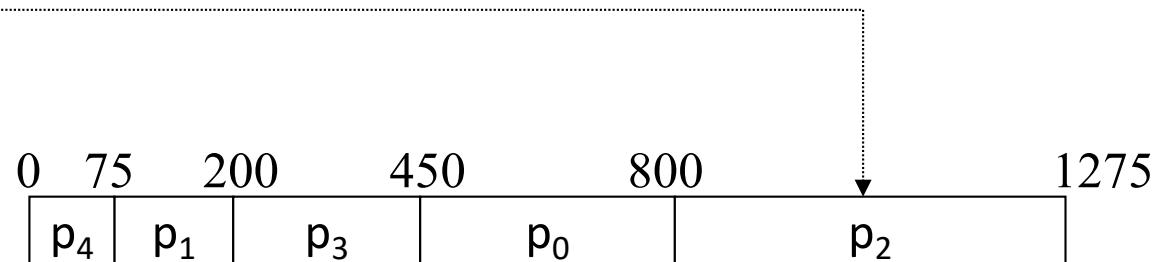
$$T_{TRnd}(p_4) = \tau(p_4) = 75$$

$$R(p_3) = 200$$

$$R(p_4) = 0$$

Example of SJF

i	$\tau(p_i)$
0	350
1	125
2	475
3	250
4	75



$$T_{TRnd}(p_0) = \tau(p_0) + \tau(p_3) + \tau(p_1) + \tau(p_4) = 350 + 250 + 125 + 75 = 800$$

$$R(p_0) = 450$$

$$T_{TRnd}(p_1) = \tau(p_1) + \tau(p_4) = 125 + 75 = 200$$

$$R(p_1) = 75$$

$$\begin{aligned} T_{TRnd}(p_2) &= \tau(p_2) + \tau(p_0) + \tau(p_3) + \tau(p_1) + \tau(p_4) \\ &= 475 + 350 + 250 + 125 + 75 \\ &= 1275 \end{aligned}$$

$$R(p_2) = 800$$

$$T_{TRnd}(p_3) = \tau(p_3) + \tau(p_1) + \tau(p_4) = 250 + 125 + 75 = 450$$

$$R(p_3) = 200$$

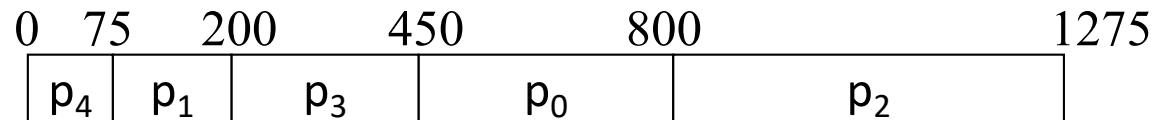
$$T_{TRnd}(p_4) = \tau(p_4) = 75$$

$$R(p_4) = 0$$

Example of SJF

i	$\tau(p_i)$
0	350
1	125
2	475
3	250
4	75

- Minimizes waiting time – Why?
- May starve large jobs



$$T_{TRnd}(p_0) = \tau(p_0) + \tau(p_3) + \tau(p_1) + \tau(p_4) = 350 + 250 + 125 + 75 = 800$$

$$R(p_0) = 450$$

$$T_{TRnd}(p_1) = \tau(p_1) + \tau(p_4) = 125 + 75 = 200$$

$$R(p_1) = 75$$

$$\begin{aligned} T_{TRnd}(p_2) &= \tau(p_2) + \tau(p_0) + \tau(p_3) + \tau(p_1) + \tau(p_4) \\ &= 475 + 350 + 250 + 125 + 75 \\ &= 1275 \end{aligned}$$

$$R(p_2) = 800$$

$$T_{TRnd}(p_3) = \tau(p_3) + \tau(p_1) + \tau(p_4) = 250 + 125 + 75 = 450$$

$$R(p_3) = 200$$

$$T_{TRnd}(p_4) = \tau(p_4) = 75$$

$$R(p_4) = 0$$

$$\text{Average response (wait) time } R_{avg} = (450+75+800+200+0)/5 = 1525/5 = 305$$

Shortest Job First

- The SJF is provably optimal
- It gives the minimum average waiting time for a given set of processes
- Moving a short process before a long one decreases the waiting time of the short process more than it increases the waiting time of the long process
- What is the difficulty of using the shortest job first scheduler?

Determining the Length of the Next CPU Burst

- Can only estimate the length.
- Can be done by using the length of previous CPU bursts, using exponential averaging.

1. t_n = actual length of n^{th} CPU burst
2. τ_{n+1} = predicted value for the next CPU burst
3. $\alpha, 0 \leq \alpha \leq 1$
4. Define :

$$\tau_{n+1} = \alpha t_n + (1 - \alpha) \tau_n$$

Examples of Exponential Averaging

- $\alpha = 0$
 - $\tau_{n+1} = \tau_n$
 - Recent information does not count.
- $\alpha = 1$
 - $\tau_{n+1} = t_n$
 - Only the actual last CPU burst counts.
- If we expand the formula, we get:

$$\begin{aligned}\tau_{n+1} &= \alpha t_n + (1 - \alpha) \alpha t_{n-1} + \dots \\ &\quad + (1 - \alpha)^j \alpha t_{n-j} + \dots \\ &\quad + (1 - \alpha)^{n+1} \tau_0\end{aligned}$$

- Since both α and $(1 - \alpha)$ are less than or equal to 1, each successive term has less weight than its predecessor.

Question

_____ is the number of processes that are completed per time unit.

- A) CPU utilization
- B) Response time
- C) Turnaround time
- D) Throughput

Question

- The strategy of making processes that are logically runnable to be temporarily suspended is called :
 - a) Non preemptive scheduling
 - b) Preemptive scheduling
 - c) Shortest job first
 - d) First come First served

Answer is b)

Puzzle

- A group of people wants to get through a tunnel.
 - A can make it in 1 minute,
 - B can in 2 minutes,
 - C can in 4 and
 - D can in 5 minutes.
- Unfortunately, not more than two persons can go through the narrow tunnel at one time, moving at the speed of the slower one.
- They have a single torch to show their way in dark
- What is the minimum time for all to make it to the other side of the tunnel?

Reading

- Read Chapter 5
- Read Chapter 4 (Linux Kernel Development)
- Acknowledgments
 - These slides are adapted from
 - Öznur Özkasap (Koç University)
 - Operating System and Concepts (9th edition) Wiley

Inter-process Communication

Didem Unat
Lecture 5

COMP304 - Operating Systems (OS)

Outline

- Last Lecture: Process Management
 - Process State
 - Context Switch
 - Process Creation and Termination
- Today: Inter-Process Communication (IPC)
 - Cooperating Processes
 - Direct Communication
 - Indirect Communication
 - IPC on Unix, Mac and Windows
 - Pipes

Quiz Question

- Each process has its own process control block
 - True or False?
- From waiting state, a process can only enter into _____.
 - A) running state
 - B) ready state
 - C) new state
 - D) terminated state

Question

```
#include <stdio.h>
#include <unistd.h>

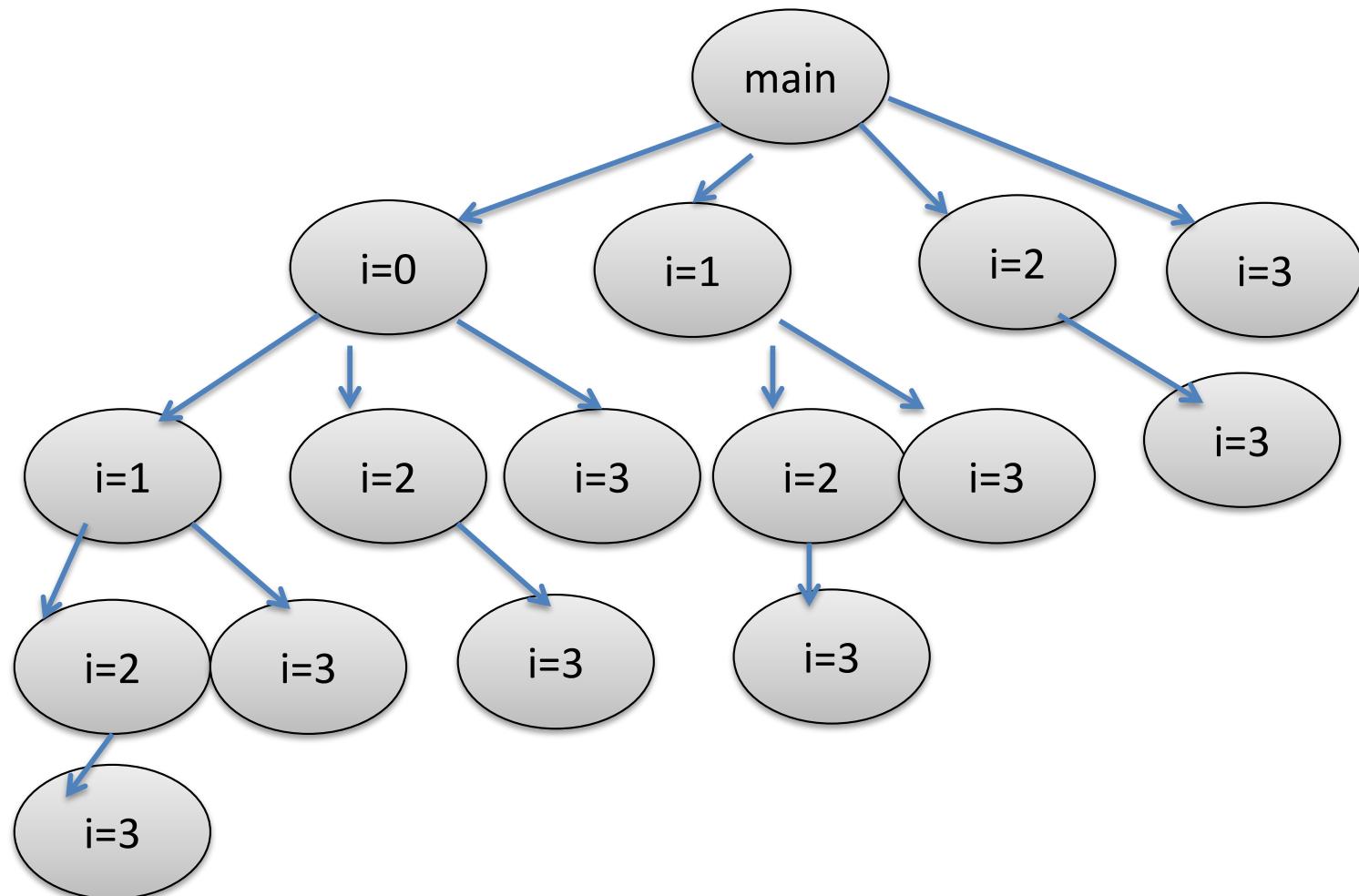
int main()
{
    int i;
    for(i=0; i < 4 ; i++)
        fork();

    printf("PID %d\n", getpid());
    return 0;
}
```

Including the initial parent process,
How many processes are created?

Draw a process tree starting from
the initial parent process as the root!

Process Tree for Question

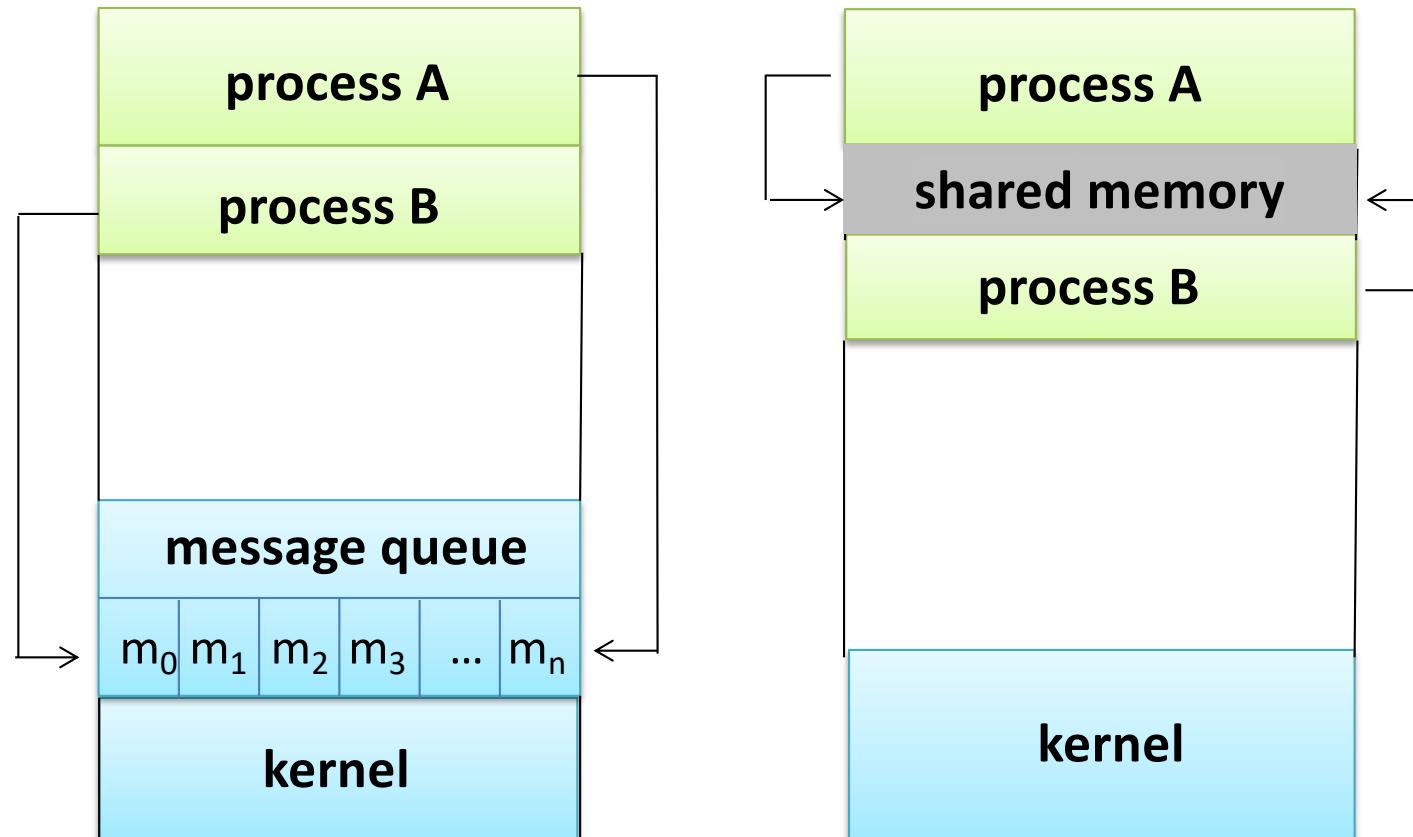


Inter-process Communication (IPC)

- *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 vs Shared Memory



- 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

Why support IPC?

There are several reasons for supporting IPC

- Sharing information
 - for example, web servers use IPC to share web documents and media with users through a web browser
- Distributing work across systems
 - for example, Wikipedia uses multiple servers that communicate with one another using IPC to process user requests
- Separating privilege
 - for example, network systems are separated into layers based on privileges to minimize the risk of attacks. These layers communicate with one another using encrypted IPC
- Processes within the same computer or across computers use similar techniques for communication

Message Passing

- Message system – processes communicate with each other without resorting to shared variables
- IPC facility provides two operations:
 - `send(message)` – message size fixed or variable
 - `receive(message)`
- If P and Q wish to communicate, they need to:
 - establish a ***communication link*** between them
 - exchange messages via send/receive
- Implementation of communication link
 - Direct or indirect,
 - Synchronous or asynchronous,
 - Automatic or explicit buffering

Direct Communication

- Processes must name each other explicitly:
 - **send** (P , *message*) – send a message to process P
 - **receive**(Q , *message*) – receive a message from process Q
- Properties of communication link
 - Links are established automatically
 - A link is associated with exactly one pair of communicating processes
 - Between each pair there exists exactly one link
 - The link may be unidirectional, but is usually bi-directional

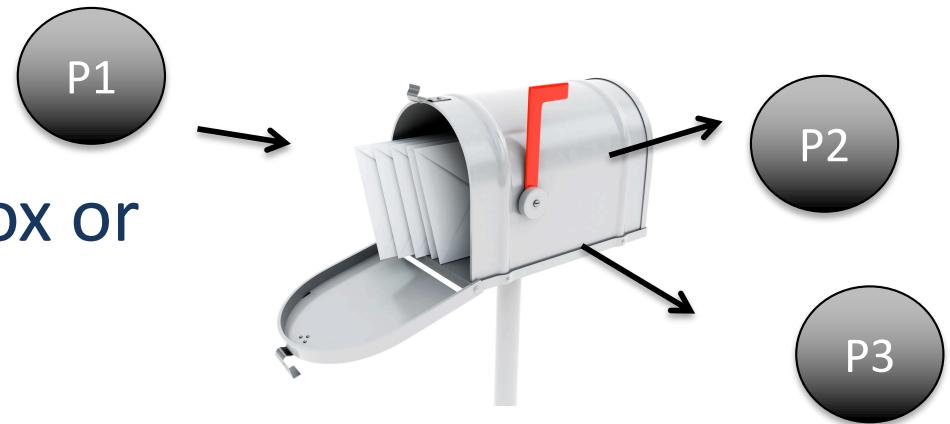
Indirect Communication

- Messages are directed and received from mailboxes (also referred to as ports)
 - Each mailbox has a unique id
 - Processes can communicate only if they share a mailbox
- Properties of communication link
 - Link established only if processes share a common mailbox
 - A link may be associated with many processes
 - Each pair of processes may share several communication links
 - Link may be unidirectional or bi-directional



Indirect Communication

- A process may own a mailbox or
- OS provides operations to
 - create a new mailbox
 - send and receive messages through mailbox
 - destroy a mailbox
- Primitives are defined as:
send(A, message) – send a message to mailbox A
receive(A, message) – receive a message from mailbox A



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?



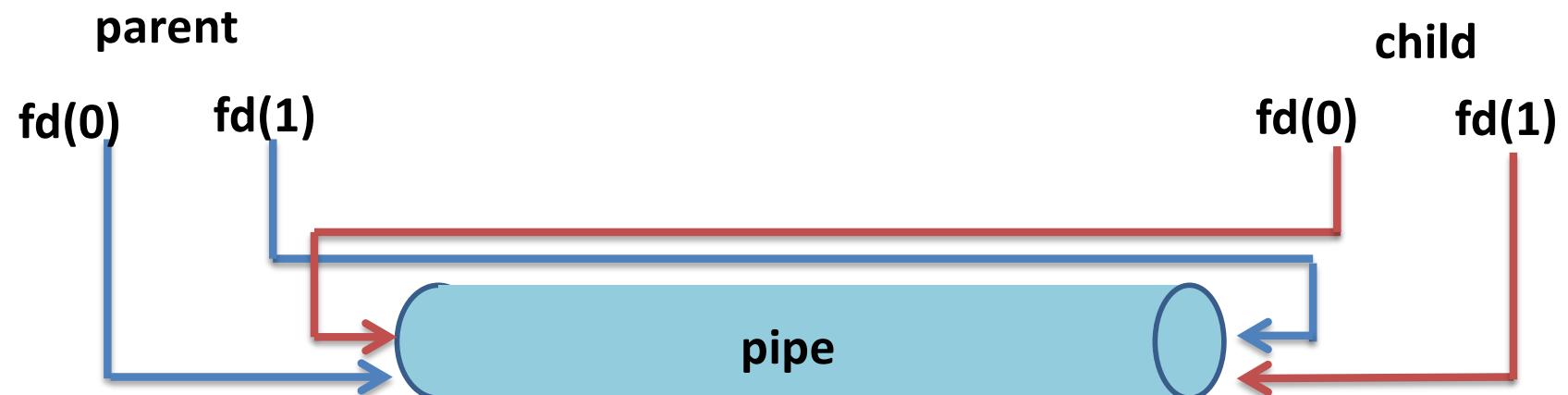
Blocking or
non-blocking?

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.

An Example of Ordinary Pipes

- Powerful command for I/O redirection
- Connects multiple commands together
- With pipes, the standard output of one command is fed into the standard input of another.

```
bash$> ls -l | less
```

```
bash$> history | less
```

Named Pipes

- Named Pipes are more powerful than ordinary pipes.
- Communication is bidirectional.
- No parent-child relationship is necessary between the communicating processes.
- Several processes can use the named pipe for communication.
- Provided on both UNIX and Windows systems.
- An example here:
 - <https://www.geeksforgeeks.org/named-pipe-fifo-example-c-program/>

Examples of IPC Systems - POSIX

- POSIX Shared Memory

- Process first creates shared memory segment

```
shm_fd = shm_open(name, O_CREAT | O_RDWR, 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");
```

IPC POSIX Producer

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>

int main()
{
    /* the size (in bytes) of shared memory object */
    const int SIZE 4096;
    /* name of the shared memory object */
    const char *name = "OS";
    /* strings written to shared memory */
    const char *message_0 = "Hello";
    const char *message_1 = "World!";

    /* shared memory file descriptor */
    int shm_fd;
    /* pointer to shared memory obect */
    void *ptr;

    /* create the shared memory object */
    shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);

    /* configure the size of the shared memory object */
    ftruncate(shm_fd, SIZE);

    /* memory map the shared memory object */
    ptr = mmap(0, SIZE, PROT_WRITE, MAP_SHARED, shm_fd, 0);

    /* write to the shared memory object */
    sprintf(ptr,"%s",message_0);
    ptr += strlen(message_0);
    sprintf(ptr,"%s",message_1);
    ptr += strlen(message_1);

    return 0;
}
```

Need to compile with `-lrt` flag

Create a shared memory segment

Memory-mapped file

Writing into the shared memory object

IPC POSIX Consumer

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>

int main()
{
    /* the size (in bytes) of shared memory object */
    const int SIZE 4096;
    /* name of the shared memory object */
    const char *name = "OS";
    /* shared memory file descriptor */
    int shm_fd;
    /* pointer to shared memory obect */
    void *ptr;

    /* open the shared memory object */
    shm_fd = shm_open(name, O_RDONLY, 0666);

    /* memory map the shared memory object */
    ptr = mmap(0, SIZE, PROT_READ, MAP_SHARED, shm_fd, 0);

    /* read from the shared memory object */
    printf("%s", (char *)ptr);

    /* remove the shared memory object */
    shm_unlink(name);

    return 0;
}
```

Create a shared memory segment for readonly

Memory-mapped file for reading

Read from the shared memory object

Examples of IPC Systems - Mach

- Mach communication is message based
 - Even system calls are messages
 - Each task gets two mailboxes at creation- Kernel and Notify
 - Only three system calls needed for message transfer
`msg_send()`, `msg_receive()`, `msg_rpc()`
 - Mailboxes needed for communication, created via
`port_allocate()`
 - Send and receive are flexible, for example four options if mailbox full:
 - Wait indefinitely
 - Wait at most n milliseconds
 - Return immediately
 - Temporarily cache a message

Examples of IPC Systems – Windows

- Message-passing centric via **advanced local procedure call (LPC)** facility
 - Only works between processes on the same system
 - Uses ports (like mailboxes) to establish and maintain communication channels
 - Communication works as follows:
 - The client opens a handle to the subsystem's **connection port** object.
 - The client sends a connection request.
 - The server creates two private **communication ports** and returns the handle to one of them to the client.
 - The client and server use the corresponding port handle to send messages or callbacks and to listen for replies.

Reading

- Read Chapter 3.4-3.7
 - Excluding client-server communication
- Acknowledgments
 - These slides are adapted from
 - Öznur Özkasap (Koç University)
 - Operating System and Concepts (9th edition) Wiley

Process Creation/Termination

COMP304

Operating Systems (OS)

Didem Unat

Lecture 4

Outline

- Process Concepts
- Process State
- Context Switch
- Process Creation and Termination
- Announcements
 - HW #1 will be out today
 - Monday, there will be a PS on HW1

Process

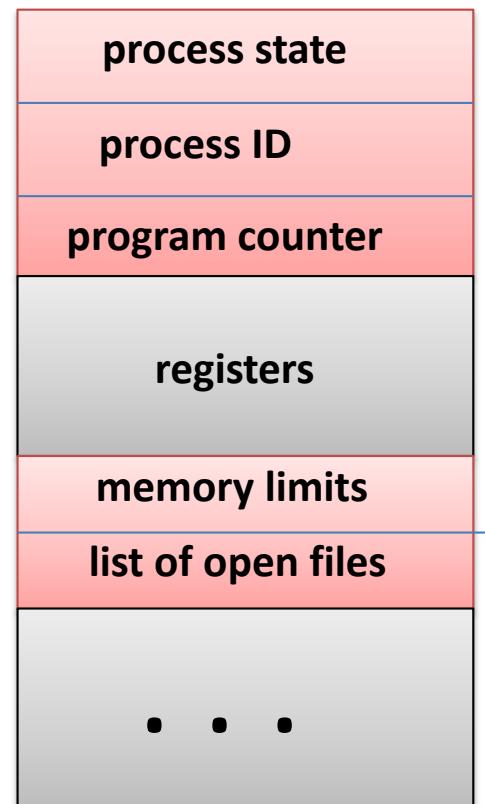
- **Process** – a program in execution;
- A program is ***passive*** entity stored on disk (**executable file**), process is ***active***
 - A program becomes a process when executable file loaded into memory
- Terms ***job***, ***task*** and ***process*** are almost interchangeably used
- Execution of a program starts via GUI mouse clicks, command line entry of its name, etc
- One program can have several processes
 - Consider multiple users executing the same program
 - Ex. Multiple browsers running at the same time

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

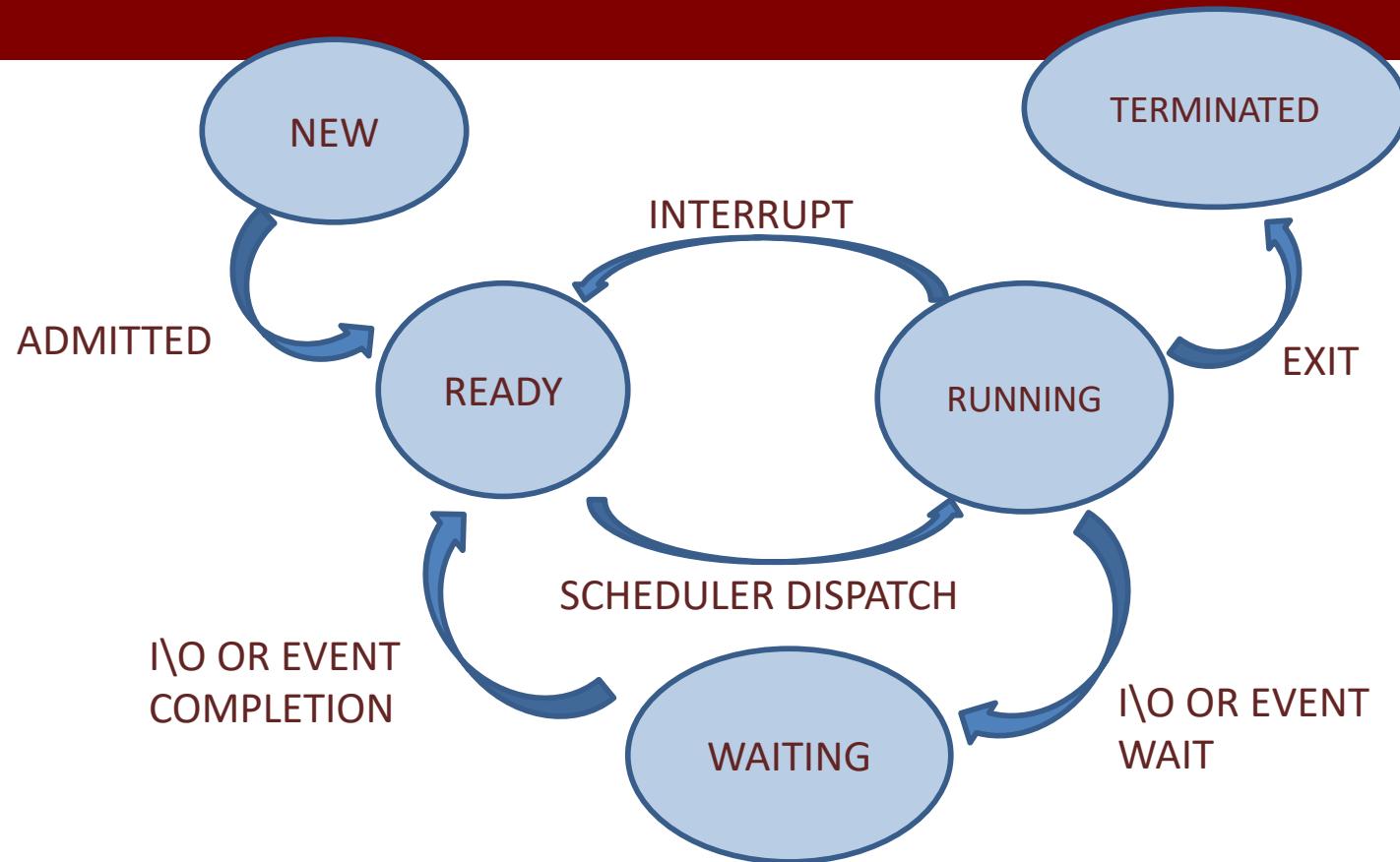
Metadata about a process



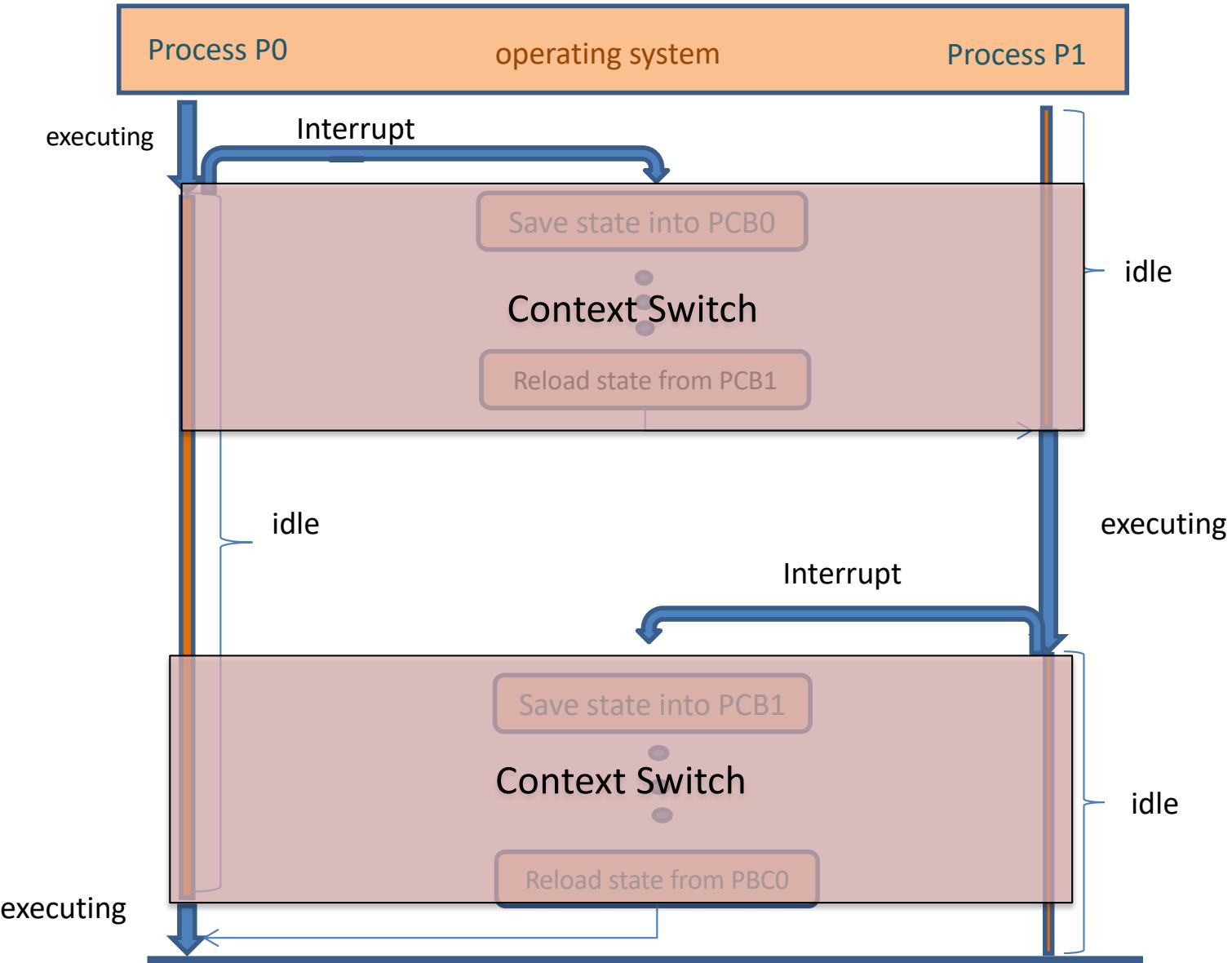
Process State

- As a process executes, it changes its **state**
 - **new**: The process is being created
 - **running**: Instructions are being executed
 - **waiting**: The process is waiting for some event (e.g., IO) to occur
 - **ready**: The process is waiting to be assigned to a CPU
 - **terminated**: The process has finished execution

Transition between Process States



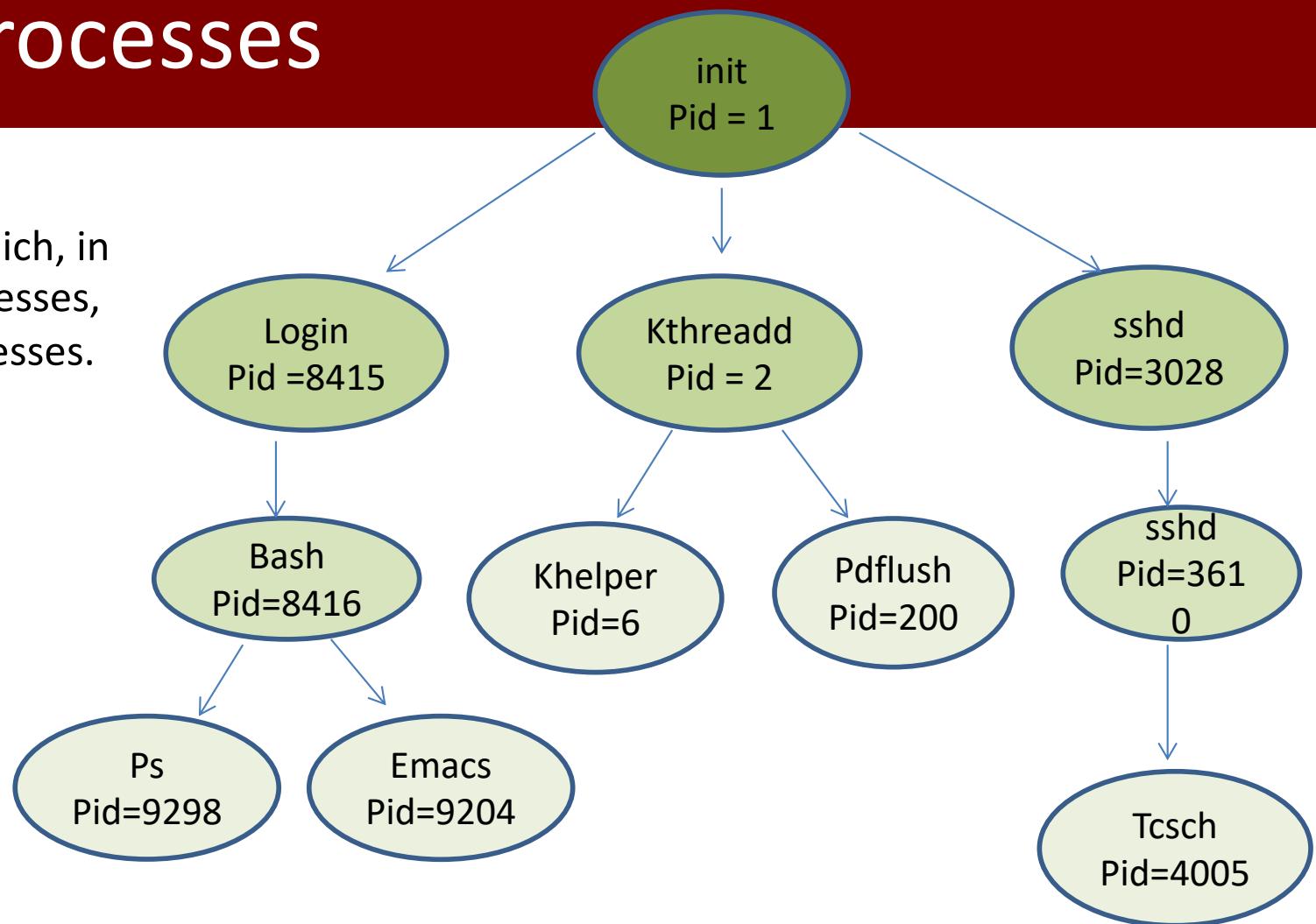
- Process transitions from one state to another
- An animation for process states
 - <https://www.youtube.com/watch?v=Y3mQYaQsrgv>



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

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

Creating/Destroying Processes

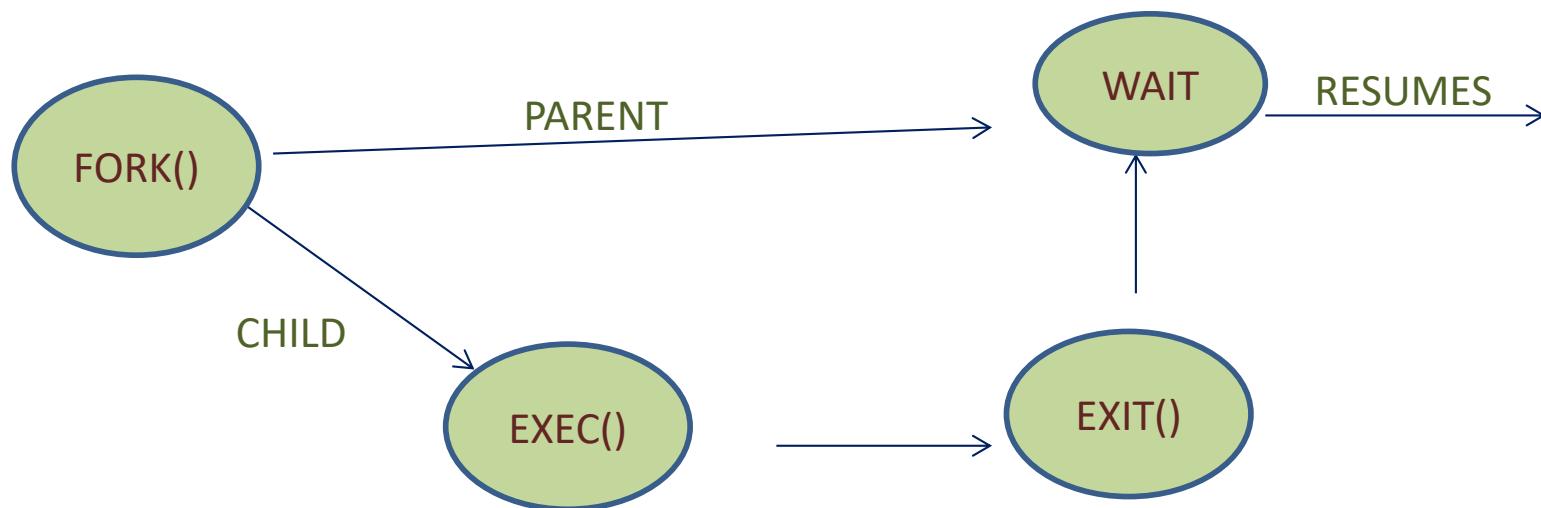
- UNIX `fork()` creates a process
 - Creates a new address space
 - Copies text, data, & stack into new address space
 - Provides child with access to open files of its parent
- UNIX `wait()` allows a parent to wait for a child to change its state
 - This is a blocking call, parent waits until it receives 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

Creating a UNIX process

```
int value, mypid=-1;  
.  
value = fork(); /* Creates a child process */  
  
/* value is 0 for child, nonzero for parent */  
if(value == 0) {  
    /* The child executes this code concurrently with parent */  
    mypid = getpid();  
    printf("Child's Process ID: %d\n", mypid);  
    exit(0);  
}  
/* The parent executes this code concurrently with child */  
  
parentWorks(...);  
wait(...);  
. . .
```

Process Creation

- Address space
 - Child duplicates the address space of the 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



Child Executing a Different Program

```
int mypid;  
...  
/* Set up the argv array for the child */  
...  
/* Create the child */  
if((mypid = fork()) == 0) {  
    /* The child executes its own absolute program */  
    execve("childProgram.out", argv, 0);  
    /* Only return from an execve call if it fails */  
    printf("Error in the exec ... terminating the child ...");  
    exit(0);  
}  
...  
wait(...) ;      /* Parent waits for child to terminate */  
...
```

```

#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
#define SIZE 5

int nums[SIZE] = {0,1,2,3,4};

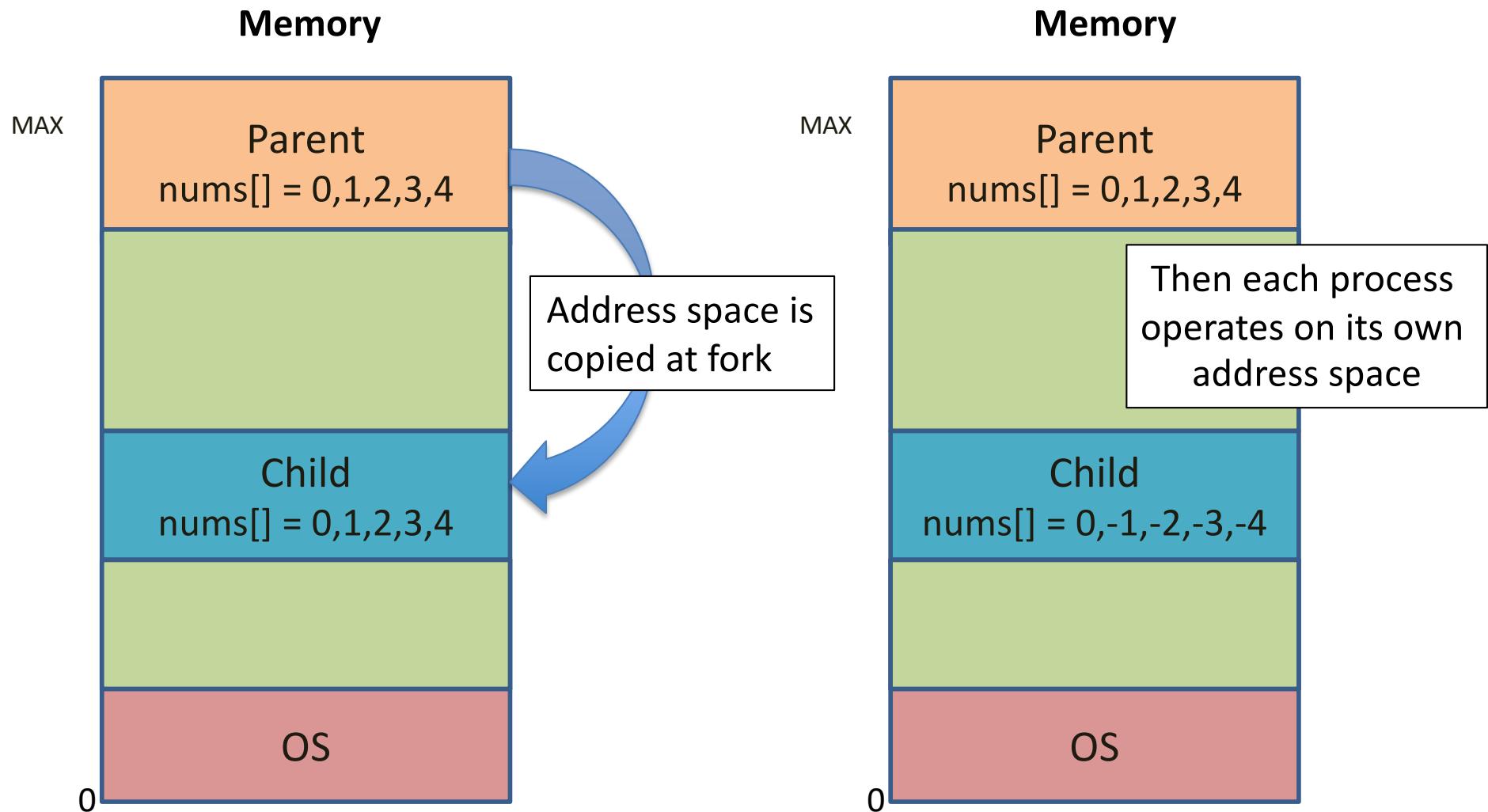
int main()
{
    int i;
    pid_t pid;
    pid = fork();

    if (pid == 0) {
        for (i = 0; i < SIZE; i++) {
            nums[i] = -i;
            printf("CHILD: %d \n",nums[i]); /* LINE X */
        }
    }
    else if (pid > 0) {
        wait(NULL);
        for (i = 0; i < SIZE; i++)
            printf("PARENT: %d \n",nums[i]); /* LINE Y */
    }
    return 0;
}

```

What output will be at
Line X and Line Y?

Address Spaces



Process Termination

- Process executes last statement and asks the operating system to delete it (**exit()**
 - Output data from child to parent (via **wait()**)
 - Terminated process' resources are deallocated by operating system
- Parent may terminate execution of children processes
 - Via **kill()** system call
- A terminated process is a **zombie**, until its parent calls **wait()**
 - Still has an entry in the process table

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

Question

```
#include <stdio.h>
```

```
int main()
{
```

```
    printf("Print 1: %d\n",getpid());
```

```
    fork(); /*fork 1*/
```

```
    printf("Print 2: %d\n",getpid());
```

```
    fork(); /*fork 2*/
```

```
    printf("Print 3: %d\n",getpid());
```

```
    fork(); /*fork 3*/
```

```
    printf("Print 4: %d\n",getpid());
```

```
    return 0;
```

```
}
```

How many processes are there?

How many prints are called?

Question

```
#include <stdio.h>
#include <unistd.h>

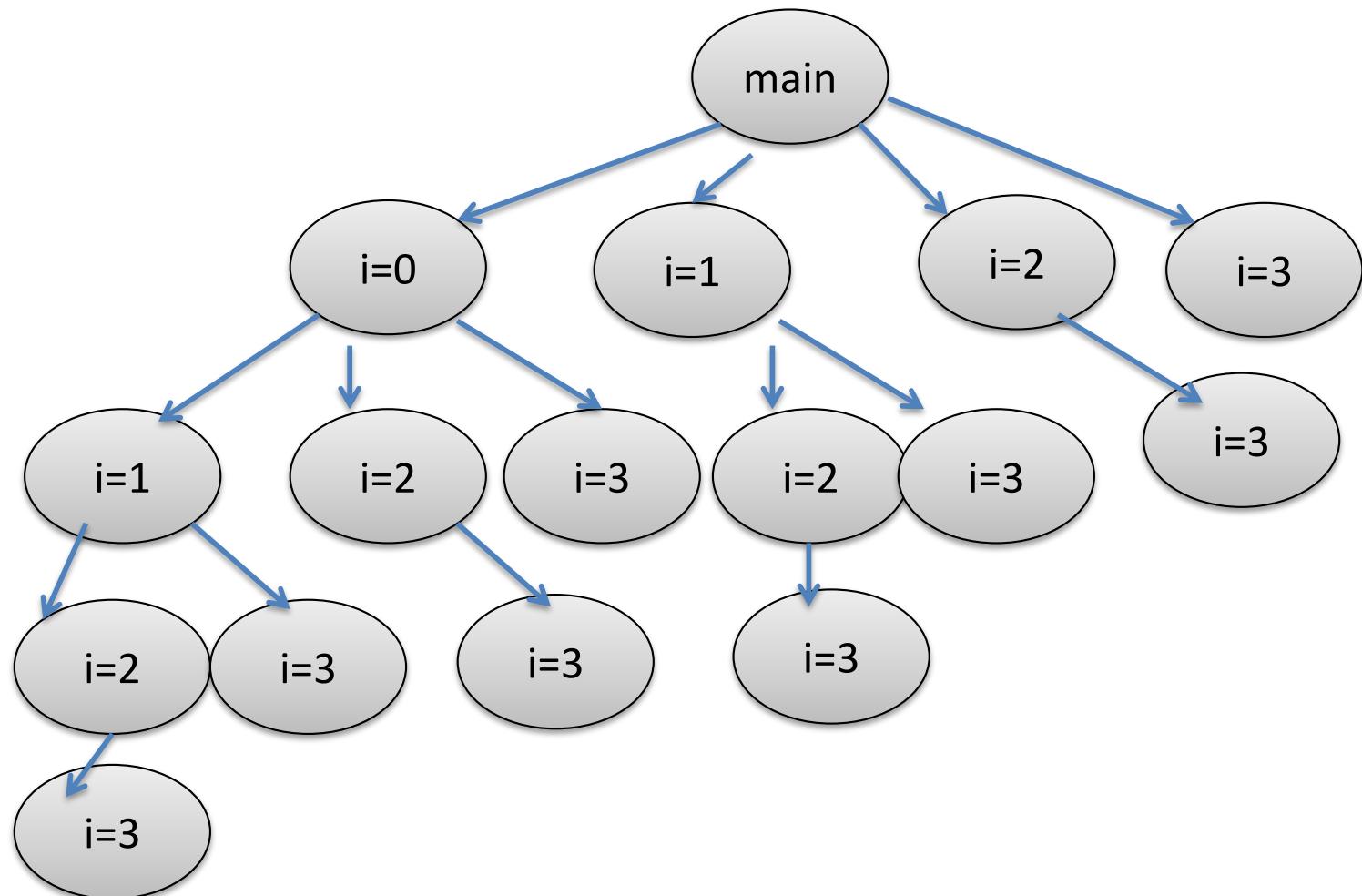
int main()
{
    int i;
    for(i=0; i < 4 ; i++)
        fork();

    printf("PID %d\n", getpid());
    return 0;
}
```

Including the initial parent process,
How many processes are created?

Draw a process tree starting from
the initial parent process as the root!

Process Tree for the Question



Reading

- From text book
 - Read Chapter 3.1-3.4
 - Linux Kernel Development (Chapter 3)
- Acknowledgments
 - These slides are adapted from
 - Öznur Özkasap (Koç University)
 - Operating System and Concepts (9th edition) Wiley

Process Management

COMP304
Operating Systems (OS)

Didem Unat
Lecture 3

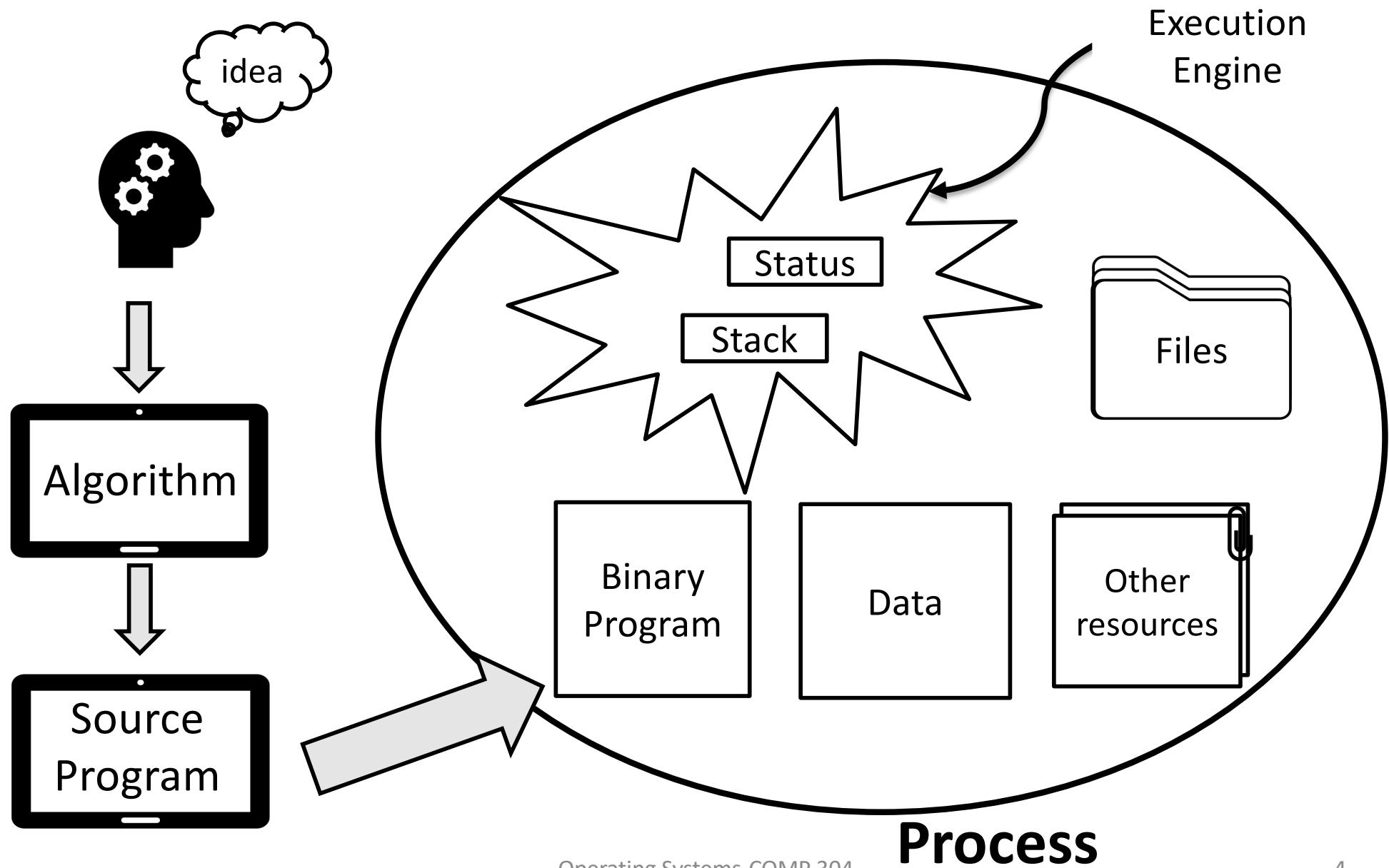
Outline

- Process Concepts
- Process State
- Context Switch
- Schedulers
- Process Creation and Termination
- Reading
 - Chapter 3.1-3.4 from textbook – Very Good!
 - Linux Kernel Development – Chapter 3
 - HW #1 will be out this week
 - Requires Linux environment with a sudo access

Process

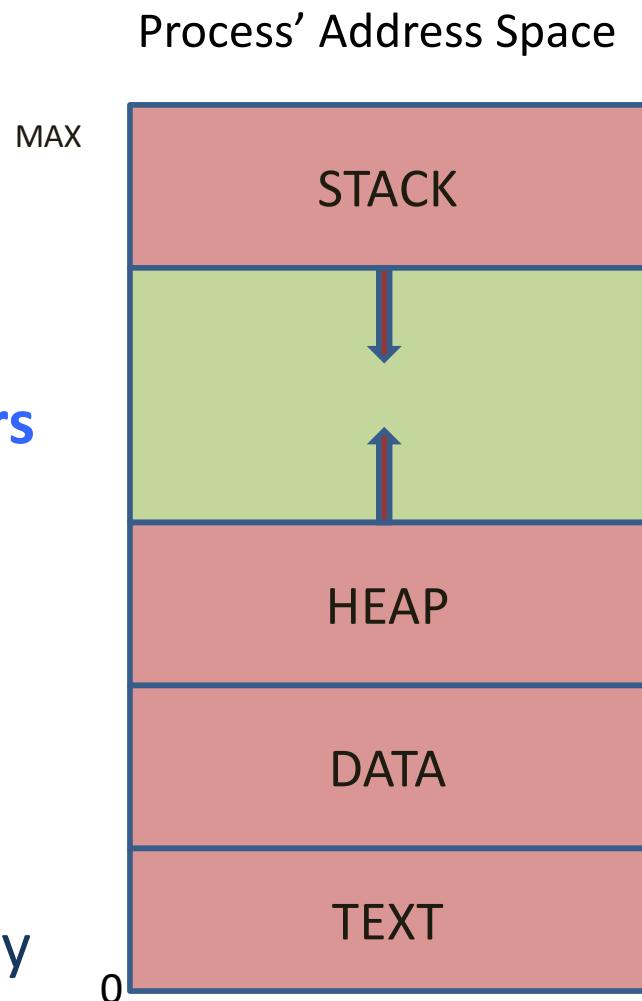
- **Process** – a program in execution;
- A program is ***passive*** entity stored on disk (**executable file**), process is ***active***
 - A program becomes a process when executable file loaded into memory
- Terms ***job***, ***task*** and ***process*** are almost interchangeably used
- Execution of a program starts via GUI mouse clicks, command line entry of its name, etc
- One program can have several processes
 - Consider multiple users executing the same program
 - Ex. Multiple browsers running at the same time

Algorithm, Program and Process



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



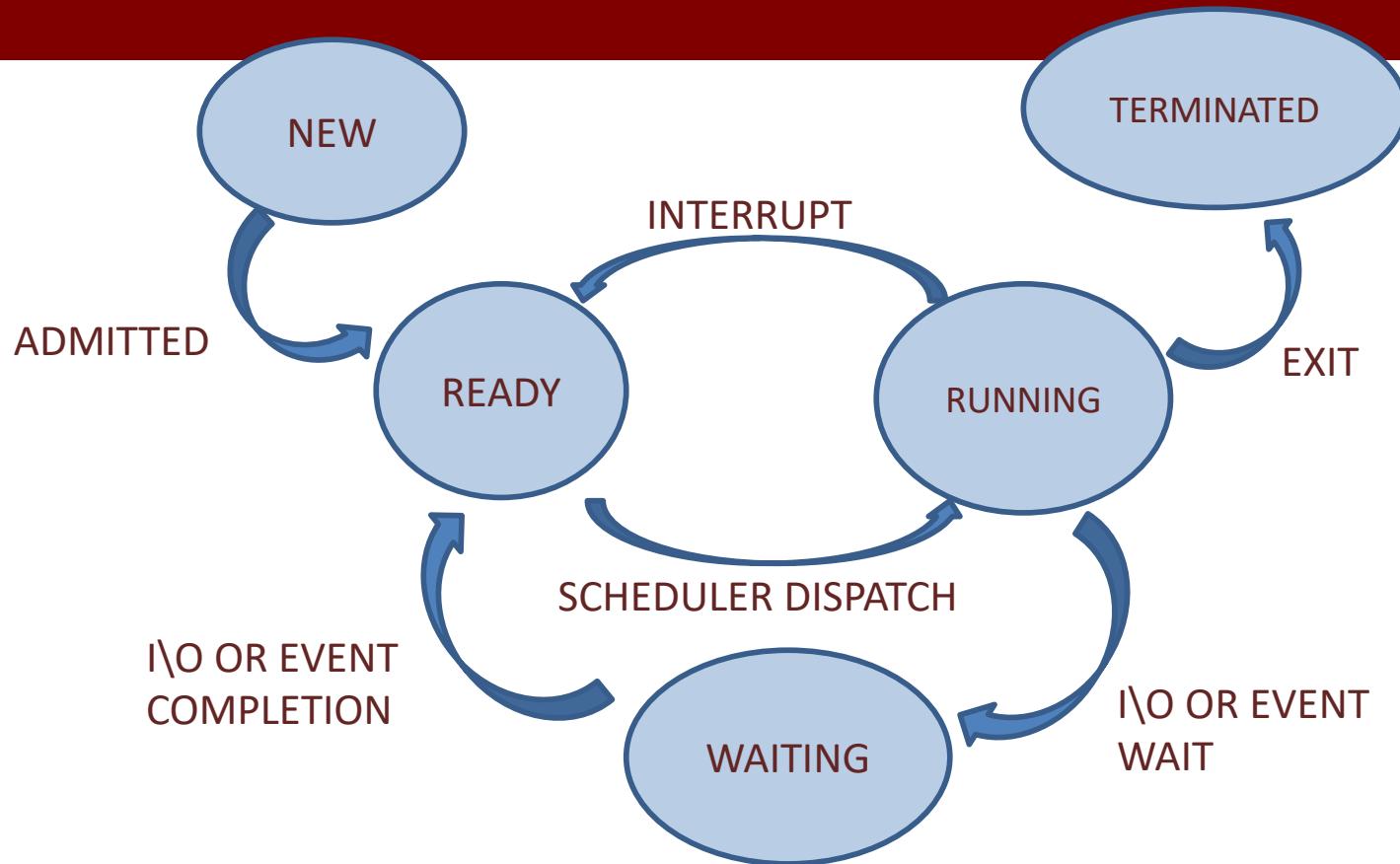
Concurrent Execution

- OS implements an abstract machine per process
- *Multiprogramming* enables
 - N programs to be *space-muxed* in executable memory, and *time-muxed* across the physical machine processor.
- Result: Have an environment in which there can be multiple programs in execution *concurrently**, each as a process
 - Concurrently means processes appear to execute simultaneously, they all make some progress over time

Process State

- As a process executes, it changes its **state**
 - **new**: The process is being created
 - **running**: Instructions are being executed
 - **waiting**: The process is waiting for some event (e.g., IO) to occur
 - **ready**: The process is waiting to be assigned to a CPU
 - **terminated**: The process has finished execution

Transition between Process States



- Process transitions from one state to another
- An animation for process states
 - <https://www.youtube.com/watch?v=Y3mQYaQsrgv>

Process Context

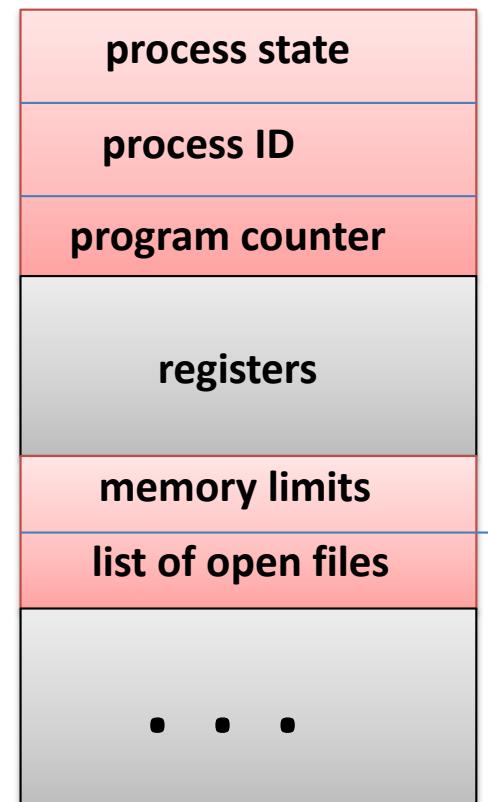
- Also called process control block
- When an interrupt occurs, what information OS needs to keep around so that we can reconstruct process's context as if it was never interrupted its execution?

Process Control Block (PCB)

Keeps the process context

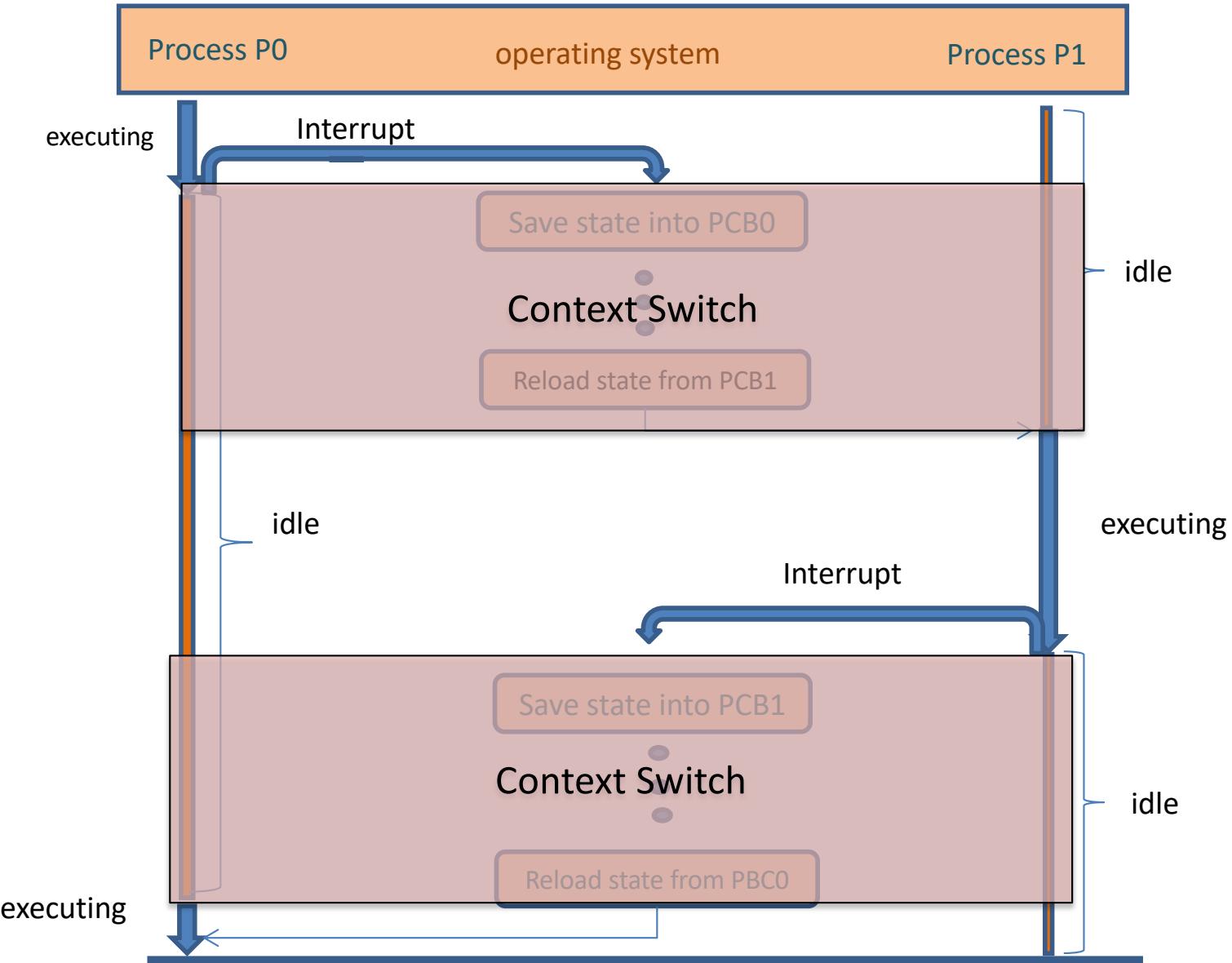
- **Process state** – running, waiting, etc
- **Program counter** – location of instruction to next execute
- **CPU registers** – contents of all process registers
- **CPU scheduling information** - priorities, scheduling queue pointers
- **Memory-management information** – memory allocated to the process
- **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 (couple millisecond)
 - Hardware support for context switching improves the performance

Threads

- A process has at least one thread of execution
- Consider having multiple program counters per process
 - Multiple locations can execute at once
 - Word document
 - Spell checker – 1 thread
 - Typing text – 1 thread
- Must then have storage for thread details, multiple program counters in PCB or each thread has a PCB
- More on threads later (in Chapter 4)

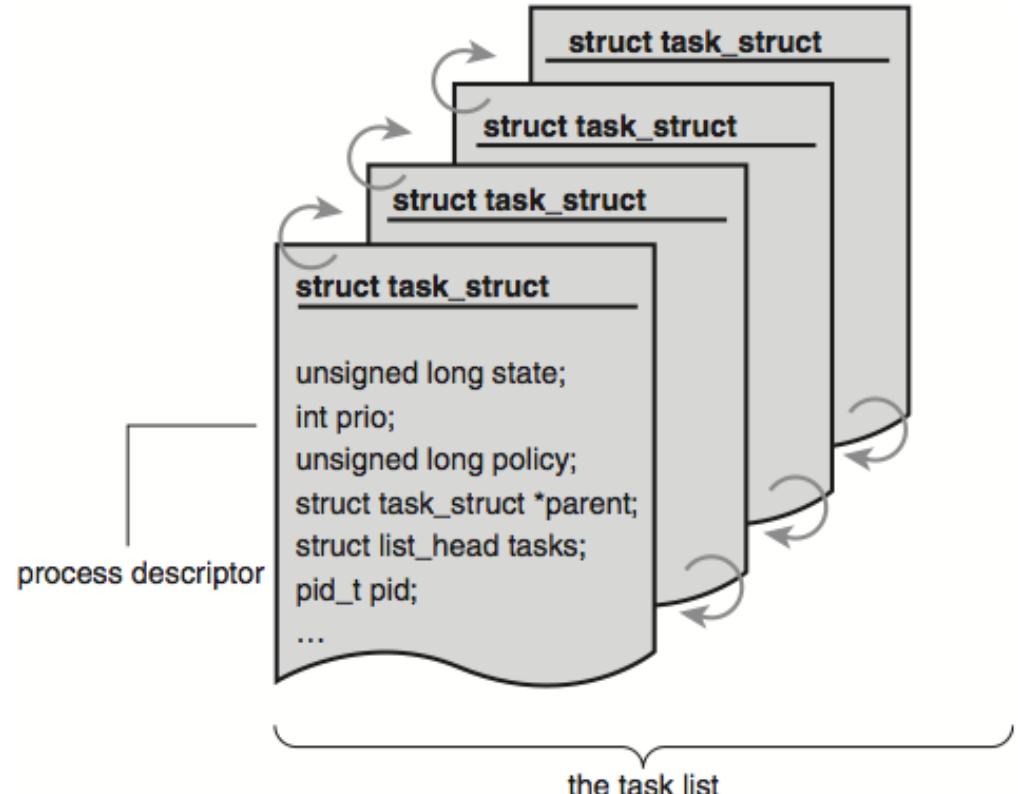
Unix Processes

- Each process has its own address space
 - Subdivided into text, data, & stack segment – a.out file describes the address space
- OS kernel creates *descriptor (PCB)* to manage process
- *Process identifier (PID)*: User handle for the process (descriptor)

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 */  
    ...  
}
```

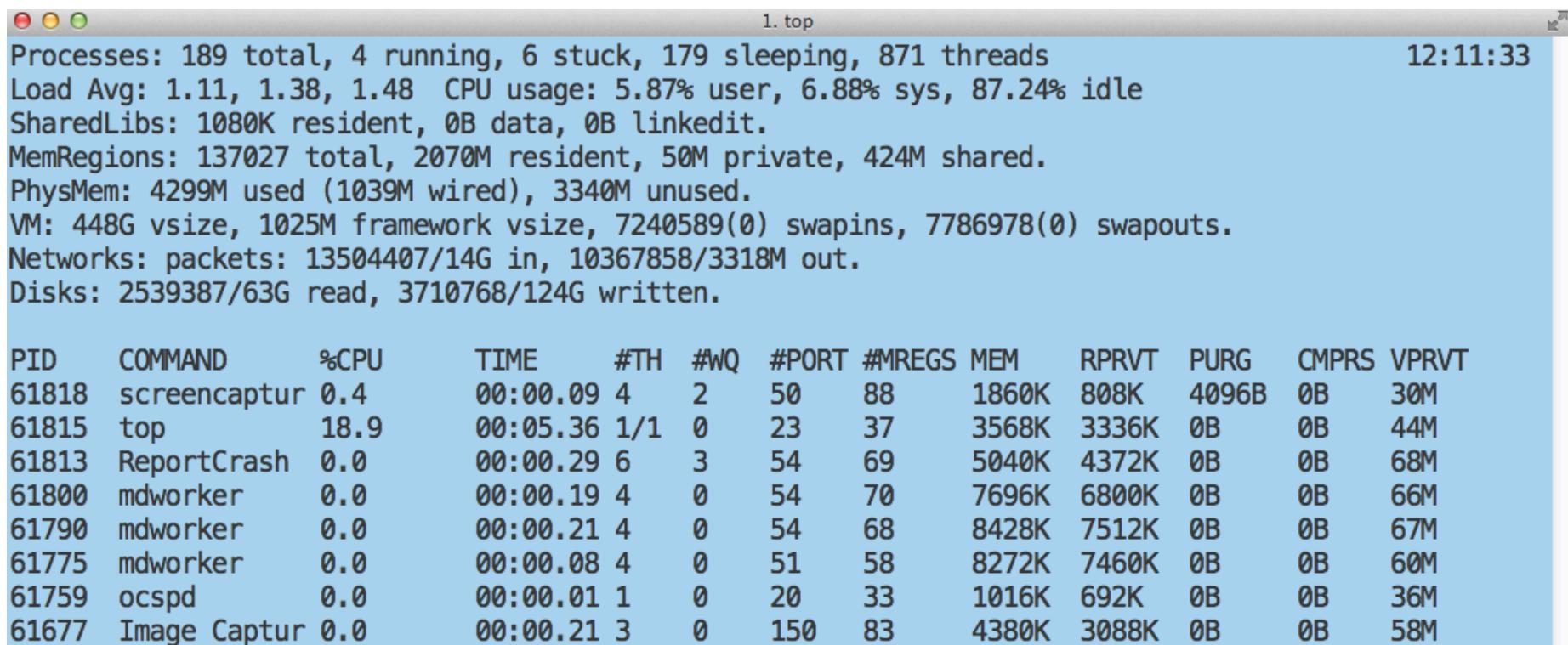


Search for `task_struct` (line ~1200)

<https://elixir.bootlin.com/linux/latest/source/include/linux/sched.h>

‘top’ and ‘ps’ commands

- top: Displays processor activity of POSIX-based OS and also displays tasks managed by kernel in real-time.
- ps: snapshot of process states



1. top

Processes: 189 total, 4 running, 6 stuck, 179 sleeping, 871 threads 12:11:33

Load Avg: 1.11, 1.38, 1.48 CPU usage: 5.87% user, 6.88% sys, 87.24% idle

SharedLibs: 1080K resident, 0B data, 0B linkedit.

MemRegions: 137027 total, 2070M resident, 50M private, 424M shared.

PhysMem: 4299M used (1039M wired), 3340M unused.

VM: 448G vsize, 1025M framework vsize, 7240589(0) swapins, 7786978(0) swapouts.

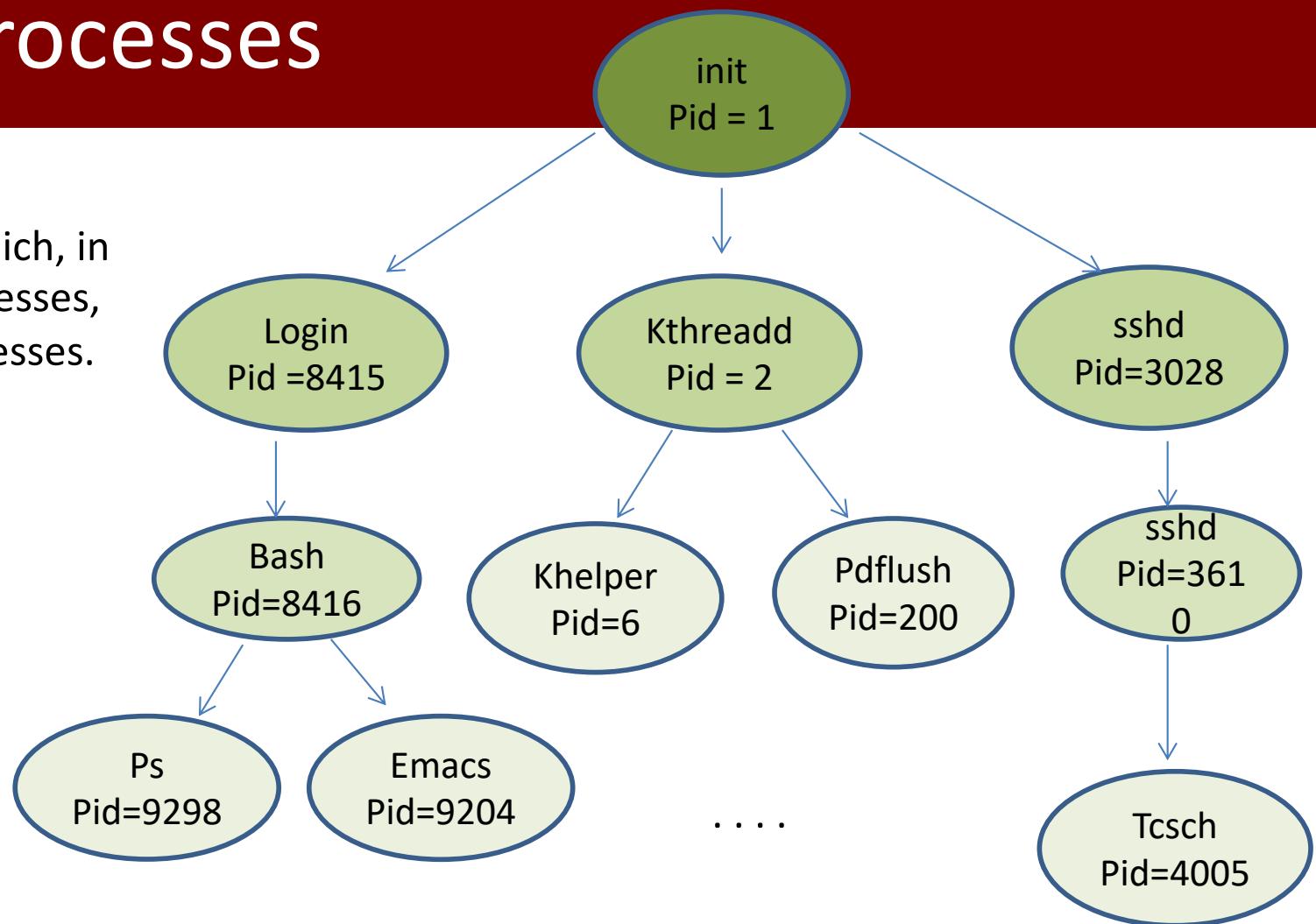
Networks: packets: 13504407/14G in, 10367858/3318M out.

Disks: 2539387/63G read, 3710768/124G written.

PID	COMMAND	%CPU	TIME	#TH	#WQ	#PORT	#MREGS	MEM	RPRVT	PURG	CMPRS	VPRVT
61818	screencaptur	0.4	00:00.09	4	2	50	88	1860K	808K	4096B	0B	30M
61815	top	18.9	00:05.36	1/1	0	23	37	3568K	3336K	0B	0B	44M
61813	ReportCrash	0.0	00:00.29	6	3	54	69	5040K	4372K	0B	0B	68M
61800	mdworker	0.0	00:00.19	4	0	54	70	7696K	6800K	0B	0B	66M
61790	mdworker	0.0	00:00.21	4	0	54	68	8428K	7512K	0B	0B	67M
61775	mdworker	0.0	00:00.08	4	0	51	58	8272K	7460K	0B	0B	60M
61759	ocspd	0.0	00:00.01	1	0	20	33	1016K	692K	0B	0B	36M
61677	Image Captur	0.0	00:00.21	3	0	150	83	4380K	3088K	0B	0B	58M

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

Creating/Destroying Processes

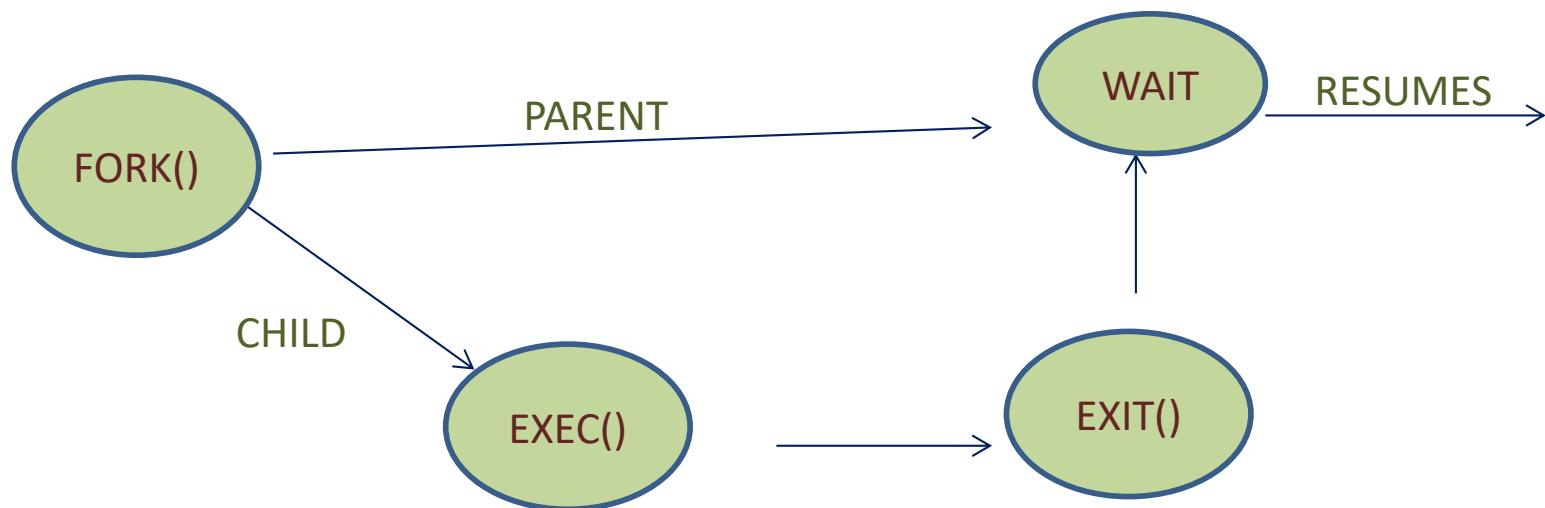
- UNIX `fork()` creates a process
 - Creates a new address space
 - Copies text, data, & stack into new address space
 - Provides child with access to open files of its parent
- UNIX `wait()` allows a parent to wait for a child to change its state
 - This is a blocking call, parent waits until it receives 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

Creating a UNIX process

```
int value, mypid=-1;  
.  
value = fork(); /* Creates a child process */  
  
/* value is 0 for child, nonzero for parent */  
if(value == 0) {  
    /* The child executes this code concurrently with parent */  
    mypid = getpid();  
    printf("Child's Process ID: %d\n", mypid);  
    exit(0);  
}  
/* The parent executes this code concurrently with child */  
  
parentWorks(...);  
wait(...);  
. . .
```

Process Creation

- Address space
 - Child duplicates the address space of the 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



Child Executing a Different Program

```
int mypid;  
...  
/* Set up the argv array for the child */  
...  
/* Create the child */  
if((mypid = fork()) == 0) {  
    /* The child executes its own absolute program */  
    execve("childProgram.out", argv, 0);  
    /* Only return from an execve call if it fails */  
    printf("Error in the exec ... terminating the child ...");  
    exit(0);  
}  
...  
wait(...) ;      /* Parent waits for child to terminate */  
...
```

Reading

- From text book
 - Read Chapter 3.1-3.4
 - Linux Kernel Development (Chapter 3)
- Acknowledgments
 - These slides are adapted from
 - Öznur Özkasap (Koç University)
 - Operating System and Concepts (9th edition) Wiley

COMP304

Operating Systems (OS)

Operating System Structure

Didem Unat

Lecture 2

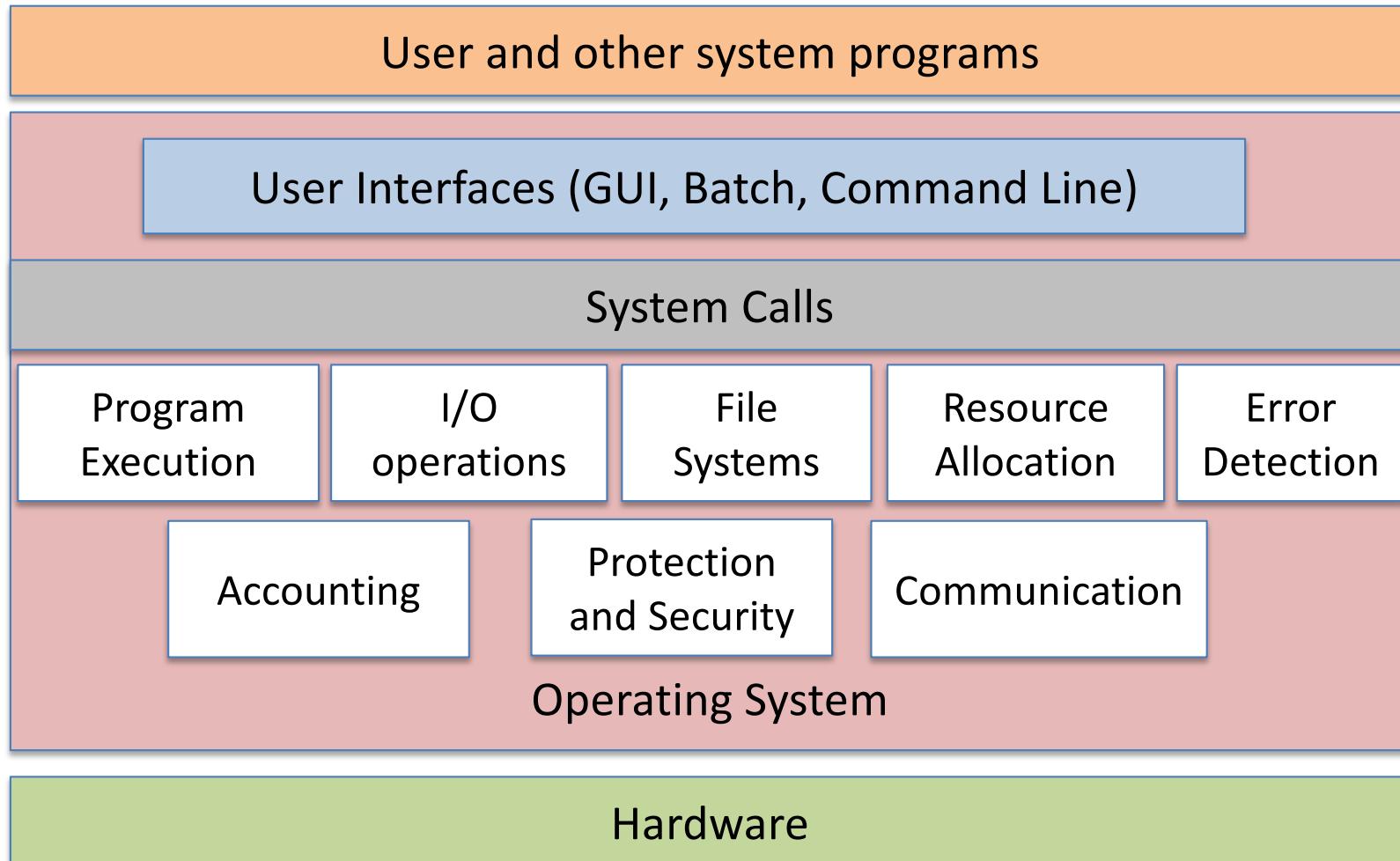
Outline

- Operating System Services
- Command Interpreter
- Dual Mode Operation
- System Calls and Types
- I/O, Memory and CPU Protection
- Operating System Design Structure

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**’ 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



Operating System Services (1/3)

- **User interface** - Almost all operating systems have a user interface (**UI**).
 - Varies between **Command-Line (CLI)**, **Graphics User Interface (GUI)**, or **Batch**
- **Program execution** - The system must be able to load a program into memory and to run that program, end execution, either normally or abnormally (indicating error)
- **I/O operations** - A running program may require I/O, which may involve a file or an I/O device
- **File-system manipulation** - Programs need to read and write files and directories, create and delete them, search them, list file information, manage permissions.

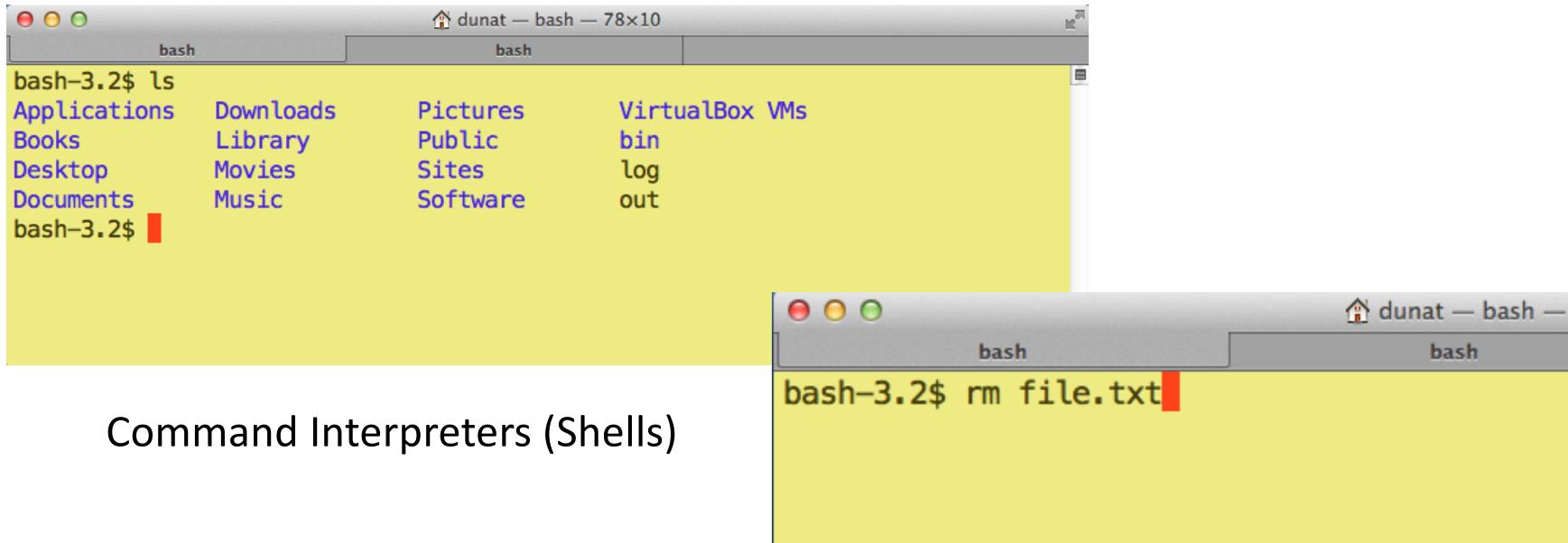
Operating System Services (2/3)

- **Communications** – Processes may exchange information, on the same computer or between computers over a network
 - Communications may be via shared memory or through message passing (packets moved by the OS)
- **Error detection** – OS needs to be constantly aware of possible errors
 - May occur in the CPU and memory hardware, in I/O devices, in user program
 - For each type of error, OS should take the appropriate action to ensure correct and consistent computing

Operating System Services (3/3)

- **Resource allocation** - When multiple users or multiple jobs running concurrently, resources must be allocated to each of them
 - Many types of resources - Some (such as CPU cycles, main memory, and file storage) may have special allocation code, others (such as I/O devices) may have general request and release code
- **Accounting** - To keep track of which users use how much and what kind of computer resources, improve response time to users
- **Protection and security** - The owners of information stored in a multiuser or networked computer system may want to control use of that information, concurrent processes should not interfere with each other

Command Interpreters



```
bash-3.2$ ls
Applications  Downloads  Pictures  VirtualBox VMs
Books         Library    Public     bin
Desktop       Movies     Sites     log
Documents     Music     Software  out
bash-3.2$
```

Command Interpreters (Shells)

```
bash-3.2$ rm file.txt
```

- In UNIX everything is a file
 - Command interpreter does not understand the command (e.g. “rm”)
 - It merely uses the command to identify a file to be loaded into memory and executed.
 - For example, shell would search for a file called ‘rm’, load the file into memory and execute it with the parameter file.txt
 - Thus, programmer can add new commands to the system easily by creating new files

Src code of Linux Commands

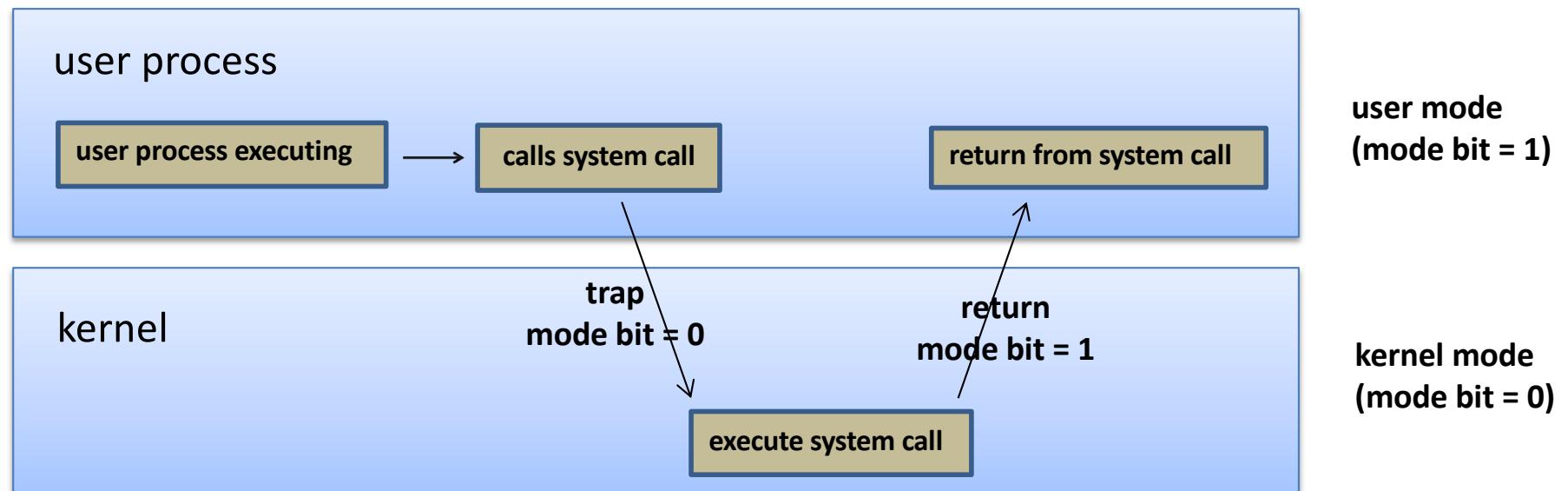
- All these basic commands are part of the **coreutils** package.
 - <http://www.gnu.org/software/coreutils/>
 - commands such as rm, ls, chmod, cp ...
 - <https://github.com/coreutils/coreutils/tree/master/src>
- For example, “ls” command:
 - <https://github.com/coreutils/coreutils/blob/master/src/ls.c>
 - Only 5000+ code lines for a command 'easy enough'

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.

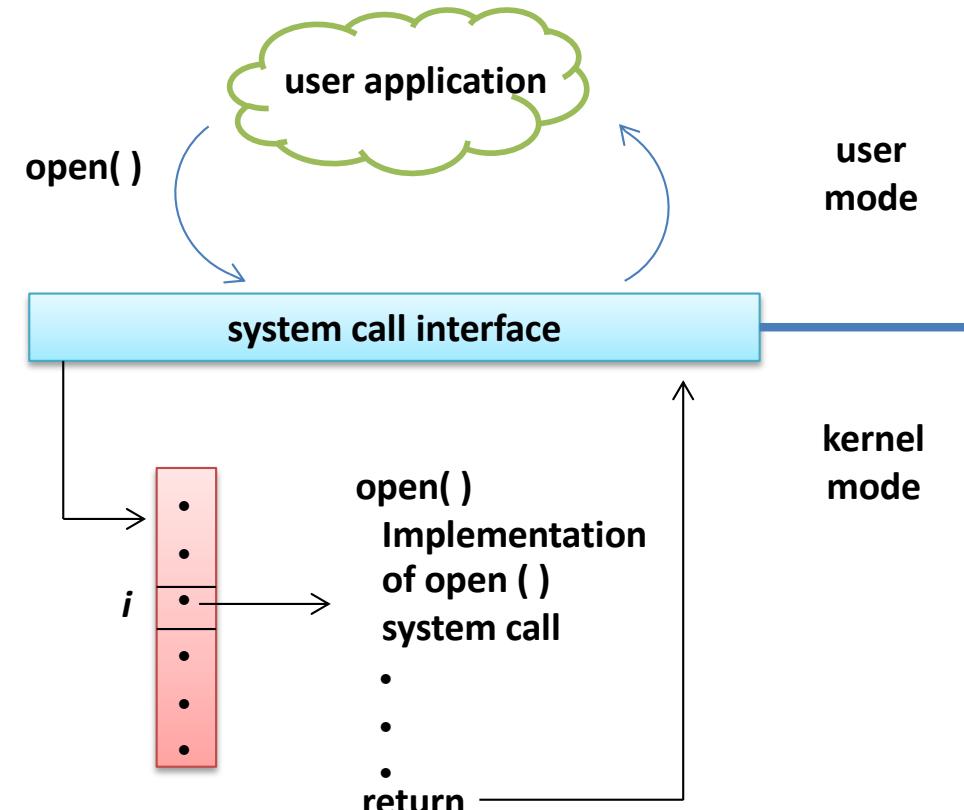
Transition from User to Kernel Mode

- **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



Example: Linux System Calls

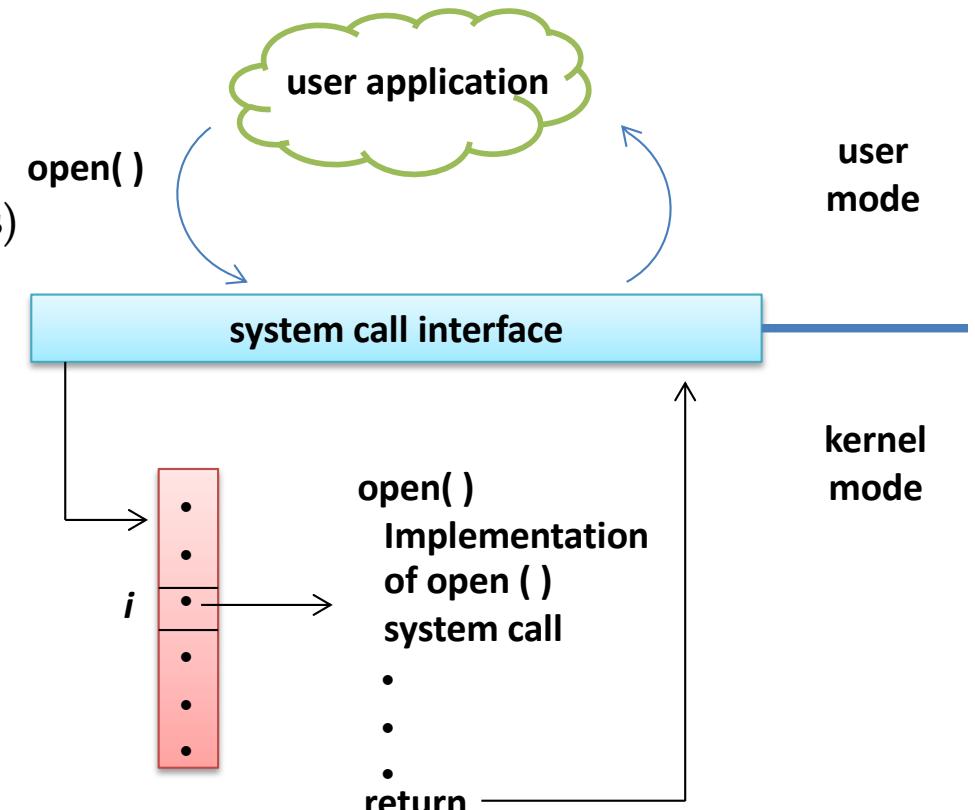
- A system call number is a unique integer in Unix-based OSs
 - There are about >300 system calls in Linux
 - A list of all registered system calls is maintained in the *system call table*
 - *Those numbers cannot be changed or recycled*
 - See the list of system calls with a command
 - (Location might differ depending on the Unix distribution)



```
cat /usr/include/asm/unistd.h | less
```

Example: Linux System Calls

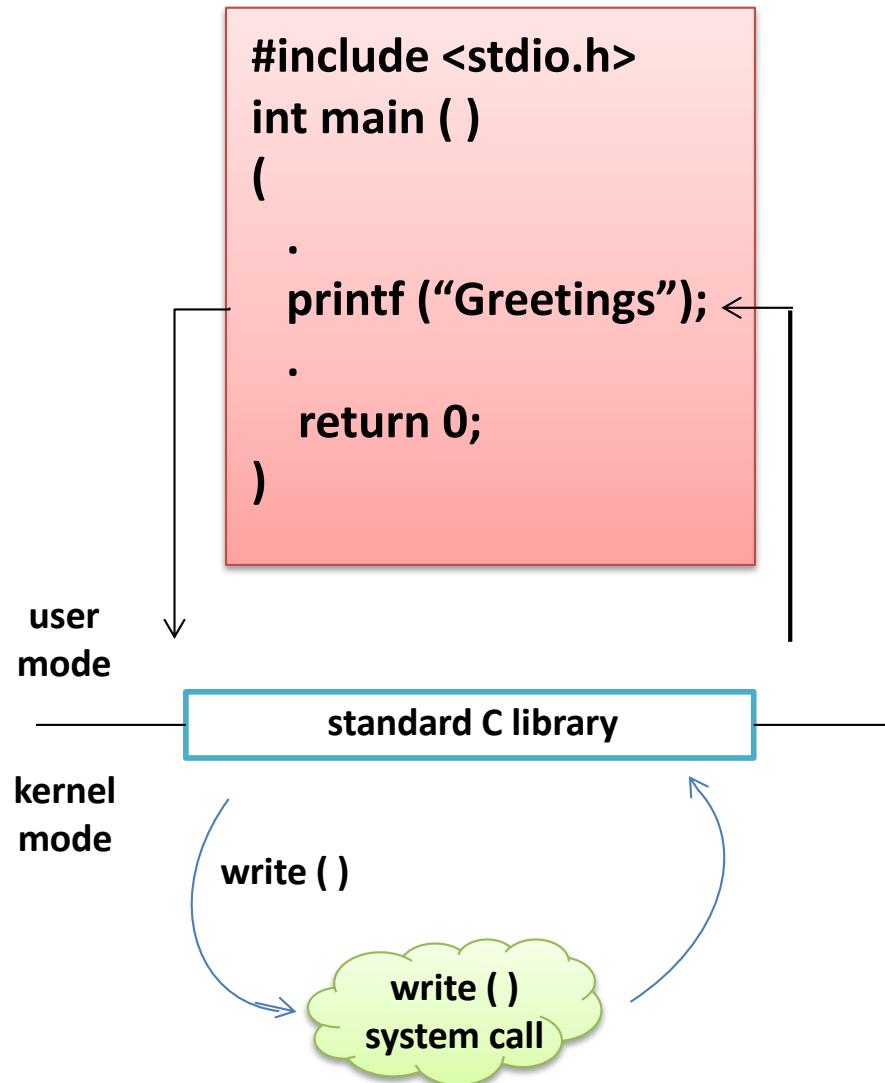
- One can also call a service by directly using its number
 - `syscall(system_call_number, arguments)`
- Actual Implementation of a system call is in different files in the kernel src
 - <https://elixir.bootlin.com/linux/latest/source>
- An example, open system call
 - <https://0xax.gitbooks.io/linux-insides/content/SysCall/linux-syscall-5.html>



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)

Standard C Library Example



- C program invoking `printf()` library call, intercepts function call in the API and invokes the necessary system calls within the operating system
 - Calls `write()` system call
- Caller needs to know nothing about
 - how the system call is implemented
 - what it does during execution

Why use APIs instead of using system calls directly?

Examples for Types of System Calls

	Windows	Unix
Process Control	CreateProcess() ExitProcess() WaitForSingleObject()	fork() exit() wait()
File Manipulation	CreateFile() ReadFile() WriteFile() CloseHandle()	open() read() write() close()
Device Manipulation	SetConsoleMode() ReadConsole() WriteConsole()	ioctl() read() write()
Information Maintenance	GetCurrentProcessID() SetTimer() Sleep()	getpid() alarm() sleep()
Communication	CreatePipe() CreateFileMapping() MapViewOfFile()	pipe() shmget() mmap()
Protection	SetFileSecurity() InitializeSecurityDescriptor() SetSecurityDescriptorGroup()	chmod() umask() chown()

- Types of system calls classified under 6 categories. Table gives an example for Windows and Unix syscalls.

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

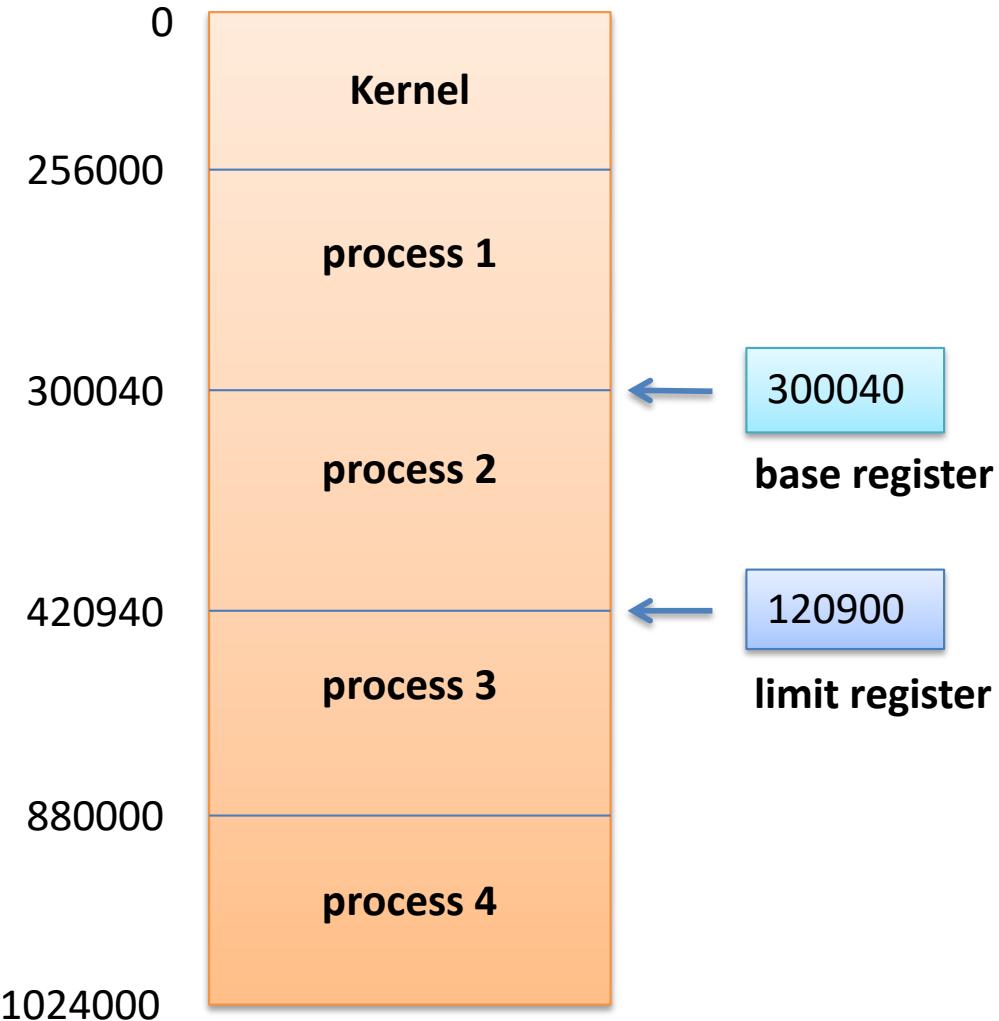
I/O Protection

- All I/O instructions are *privileged instructions*.
 - Must ensure that a user program could never gain control of the computer in **kernel mode** (i.e., a user program that, as part of its execution, stores a new address in the interrupt vector).
 1. “normal” instructions, e.g., add, sub, etc.
 2. “privileged” instructions, e.g., initiate I/O switch state vectors or contexts load/save from protected memory etc.

Memory Protection

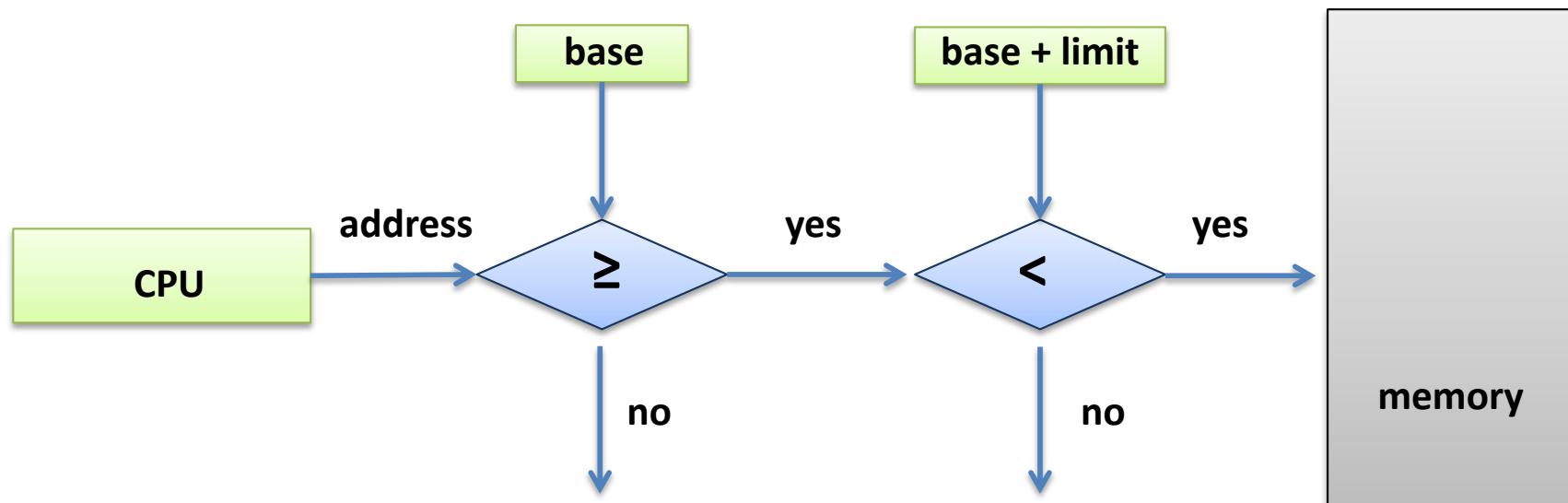
- Must provide memory protection at least for the interrupt vector and the interrupt service routines.
- In order to have memory protection, add two registers that determine the range of legal addresses a process may access:
 - **Base register** – holds the smallest legal physical memory address.
 - **Limit register** – contains the size of the range
- Memory outside the defined range is protected.

Use of a Base and Limit Registers



Hardware Protection

- When executing in kernel mode, the operating system has unrestricted access to both kernel and user's memory.
- The load instructions for the *base* and *limit* registers are **privileged instructions**.



A fault raised by hardware, notifying the operating system about an addressing error

CPU Protection

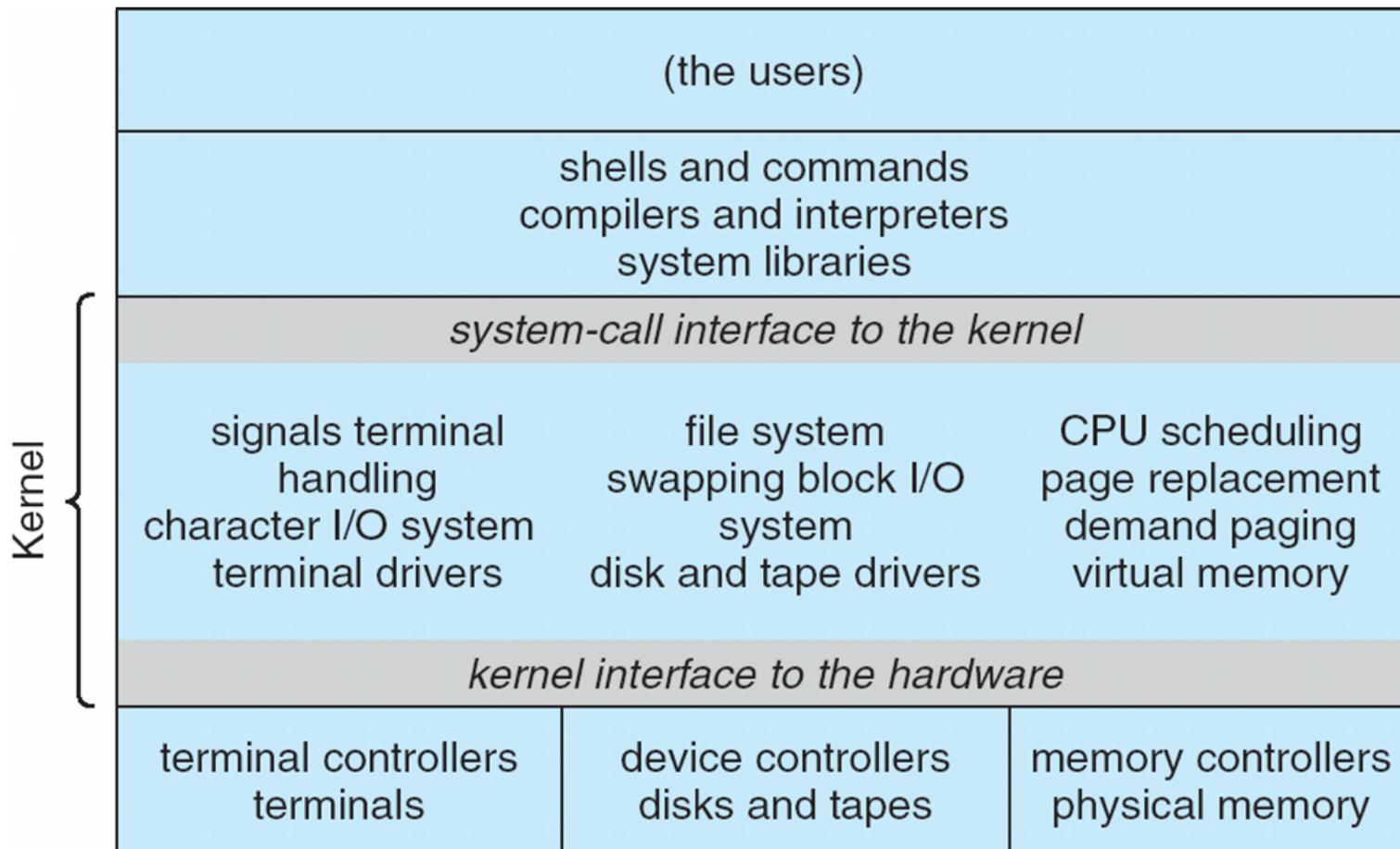
- *Timer* – interrupts computer after specified period to ensure operating system maintains control.
 - Timer is decremented every clock tick.
 - When timer reaches the value 0, an interrupt occurs.
- Timer commonly used to implement time sharing systems.
- Clearly, instructions that modify the content of the timer are privileged.

Operating System Structure

- General-purpose OS is very large program
 - Typically written in assembly, C/C++, some scripts in Perl or Python
- Various ways to structure it
 - **Monolithic Kernel:** All the OS services are implemented in the kernel.
Fast OS but hard to extend
 - Ex: MS-DOS, Unix
 - **Microkernel:** Moves all the nonessential components from the kernel to user level. Smaller kernel, uses messages with system and user-level programs
 - Ex: Mach
 - **Modular Approach:** Loadable kernel modules, load additional services if needed at boot or run time
 - Ex: Solaris
- Most current OS combines all three approaches nowadays
 - Ex: Windows, Mac OS X, Linux

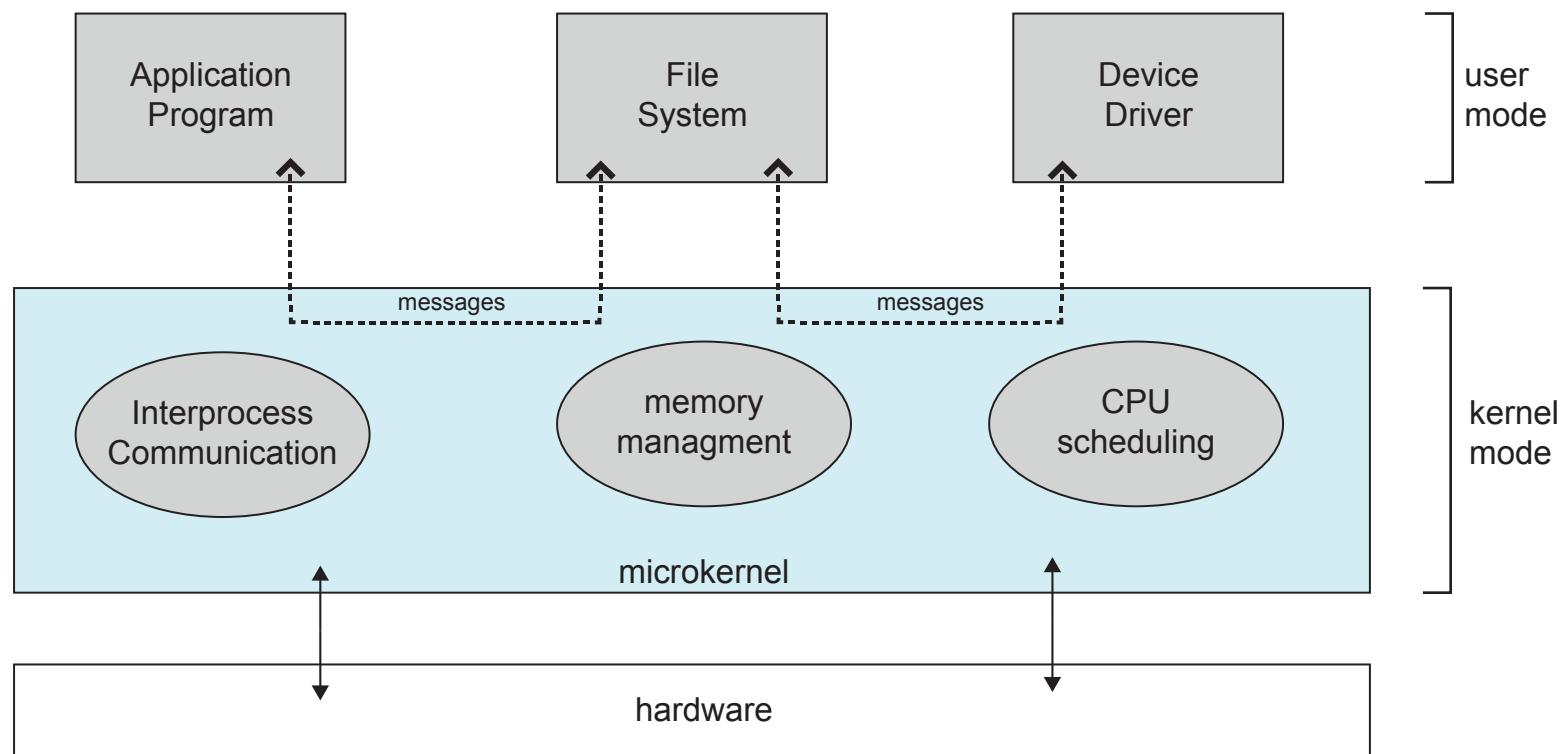
Monolithic Kernel

All the OS services are implemented in the kernel. Fast OS but hard to extend



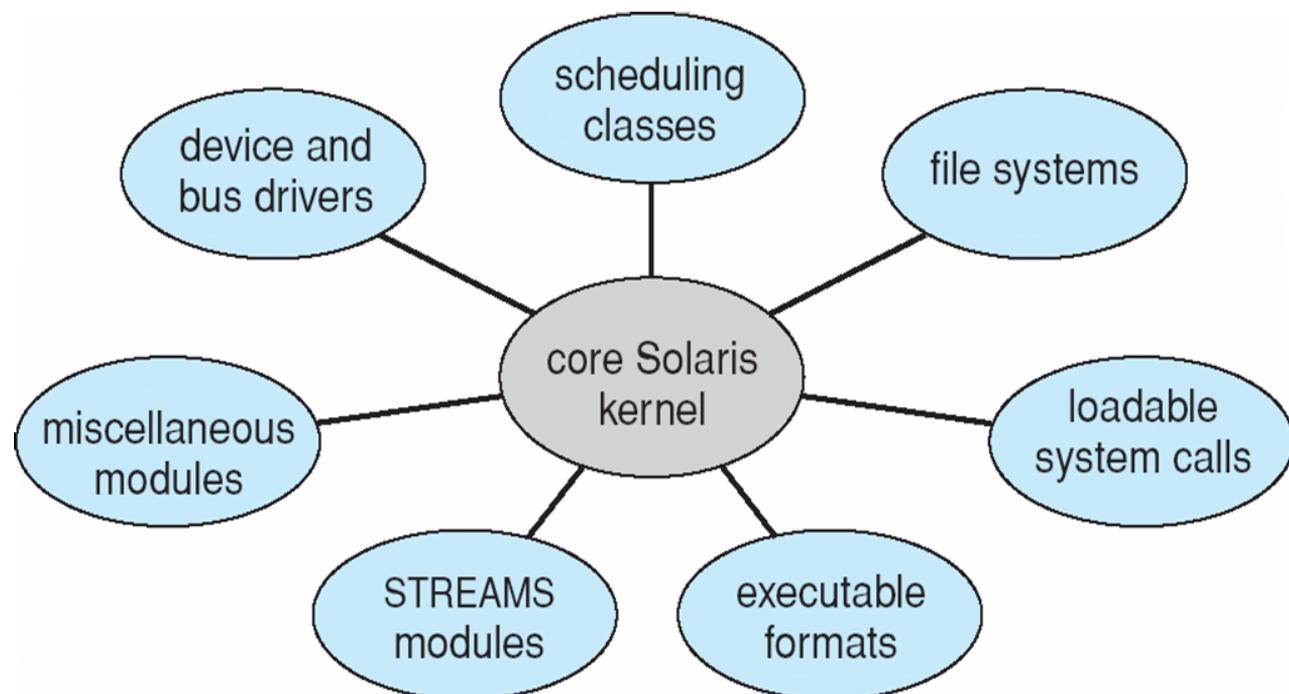
Microkernel

- Moves all the nonessential components from the kernel to user level.
Smaller kernel, uses messages with system and user-level programs

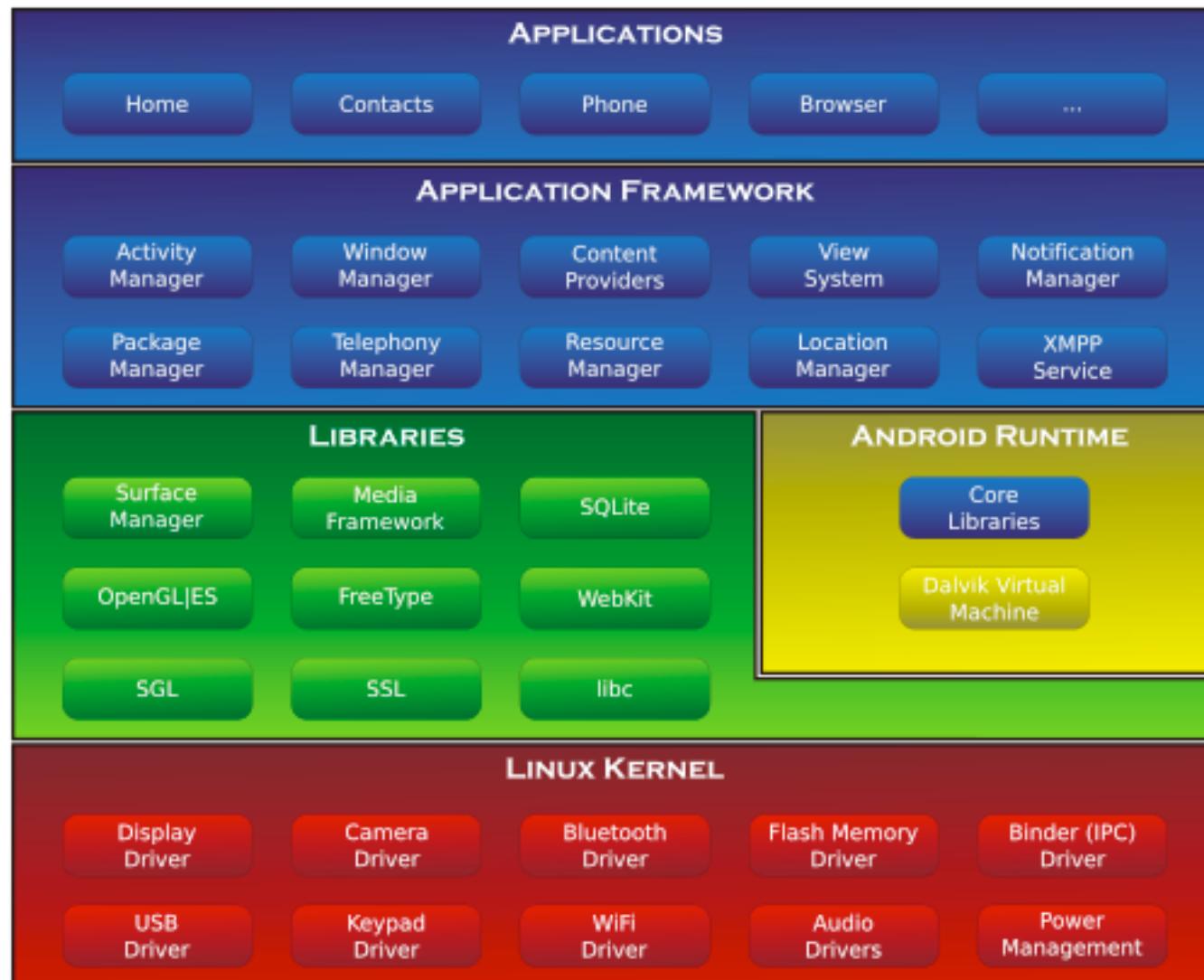


Modular Kernel

- **Modular Approach:** Loadable kernel modules, load additional services if needed at boot or run time



Android



Question

- Which of the following instructions should be privileged?
 - a. Set value of timer.
 - b. Read the clock.
 - c. Clear memory.
 - d. Issue a trap instruction.
 - e. Turn off interrupts.
 - f. Modify entries in device-status table.
 - g. Access I/O device.
- a, c, e, f, g

Question

- A _____ can be used to prevent a user program from never returning control to the operating system.
 - Dual mode
 - Program counter
 - Kernel module
 - Timer

D

Reading

- From text book
 - Read Chapter 2: Section 2.1-2.4, 2.10
- Linux System Call References
 - <https://man7.org/linux/man-pages/man2/syscalls.2.html>
- Kernel source code
 - <https://elixir.bootlin.com/linux/latest/source>
- Acknowledgments
 - These slides are adapted from
 - Öznur Özkarap (Koç University)
 - Operating System and Concepts (9th edition) Wiley

COMP304

Operating Systems (OS)

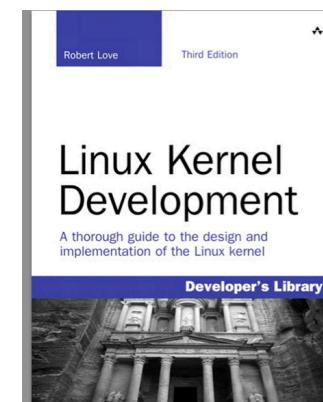
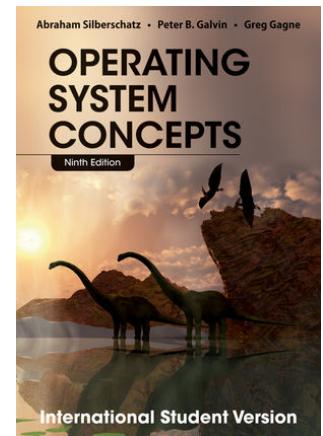
Introduction

Didem Unat

Lecture 1

Course Basics

- Website
 - Blackboard : <https://ku.blackboard.com/>
 - All course materials will be posted
- Main Book
 - *Operating System and Concepts (10th edition)*
 - By Silberschatz, Galvin and Gagne
- Additional Book
 - *Linux Kernel Development (3rd Edition)*
 - By Robert Love
 - <http://it-ebooks.info/book/819/>



Linux Operating System

- In assignments and projects, we will be using Linux environment. You have two options:

BACK UP YOUR DATA

1) Install a Linux OS environment **(recommended)**

Installation package (latest distributions of Ubuntu or Fedora)

Install as dual boot on your own computer

2) Install Linux Virtual Machine on your computer

Have it ready by next week for the PS, consult your TAs or peers if you have any problems

Linux Tutorial

- Learning Unix commands
 - <http://www.ee.surrey.ac.uk/Teaching/Unix/>
- Study the intro and first 2 sections by next PS hour
 - Experiment with these basic commands
 - First PS will go over the commands
- First assignment will require you to have a running Linux environment and basic Unix command knowledge

Grading

- **Grading**

- %10 Written/Coding Assignments (5+3+2 of them)
- %38 Projects (14+14+10 of them)
- %20 Midterm
- %30 Final
- %02 Attendance

Final makeup exam and remedial exam will take place on the same day at the same time. A student can take either of them but not both.

Midterm makeup exam is on the last week of the instructions at the PS hour. Midterm makeups are not cumulative.

TAs and PS Hours

- PS is on Thursdays at 5.30 pm
 - Not every week.
 - We will announce it when it is happening.
- TA Office Hour in-person (ENG 230)
 - 5.30-7 pm Tuesdays, Wednesdays
- My Office Hour
 - Tuesdays before the class
- TAs
 - Ilyas Turimbetov (ENG 230)
 - Kefah Issa (ENG 230)
 - Ismayil Ismayilsoy (ENG 230)

About me



2006
Graduated from
Boğaziçi University

2012
PhD at
University of California,
San Diego



About me



2012-2014

Luis Alvarez Postdoctoral
Fellowship

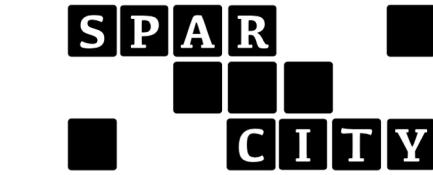
Lawrence Berkeley National
Laboratory

2014 -

Koç University



About me



2.6M Euro

1.5M Euro

6 European partners

First in Turkey
in Computer Eng.



EuroHPC
Joint Undertaking



European Research Council
Established by the European Commission

A screenshot of the Communications of the ACM website. The header includes the ACM logo, the text "COMMUNICATIONS OF THE ACM", and navigation links for HOME, CURRENT ISSUE, NEWS, BLOGS, OPINION, RESEARCH, PRACTICE, CAREERS, and ARCHIVE. A search bar is also present.

[Home](#) / [Careers](#) / [Didem Unat Named SIGHPC Emerging Woman Leader in Technical...](#) / [Full Text](#)

ACM CAREERS

Didem Unat Named SIGHPC Emerging Woman Leader in Technical Computing

September 8, 2021

[Comments](#)

VIEW AS: [List](#) [Grid](#) SHARE:



Didem Unat is an assistant professor of computer engineering at Ko University in Istanbul, Turkey.

From [HPCwire](#)
[View Full Article](#)

Didem Unat of Koç University has been named the winner of the 2021 ACM SIGHPC Emerging Woman Leader in Technical Computing award. Unat was recognized for innovations in the field of programming models for data locality in high performance and scientific computing and for her leadership role in the international high performance computing community.

Unat's work on simplifying software development for current and future supercomputing architectures resulted in architecture-independent abstractions. These allow for the development of scientific software that maps to complex memory hierarchies and accelerator structures, with high-performance results.

"Unat's rigorous technical work has directly impacted the productivity of application scientists," says award committee chair Cristina Beldica of Intel. "This is critical not only for high performance computing, but science in general."

SIGN IN for Full Access

User Name
Password
» [Forgot Password?](#)
» [Create an ACM Web Account](#)

MORE NEWS & OPINIONS
[Phone's Dark Mode Doesn't Necessarily Save Much Energy](#)
Tech Explorist

[Lessons From the Loop](#)
Lorrie Faith Cranor

[Inclusive Integration of Computing in School Dis](#)
[Two Essential Tradeoffs](#)
Merijke Coenraad, Brian Giovanini, Mills, Jeremy Roschelle

Motivation

- Operating Systems: Major field of Computer Science and Engineering
 - One of the MOST important and enjoyable course
- Around 20% of questions in GRE Computer Science subject test are from the OS concepts
- Forms a good knowledge base for other subject areas
- Provides a complete understanding of software/hardware infrastructure

Elements

- Good knowledge in
 - C programming
 - Data structures
 - Computer Systems
 - CPU and Memory Subsystem
 - Algorithms

What is an Operating System?

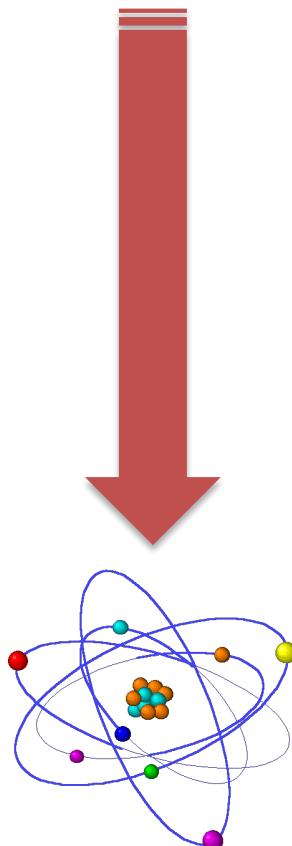
- A program that acts as an intermediary (supervisor) between a user of a computer and the computer resources
- Duties of an OS
 - 1) Provide resource abstraction
 - 2) Manage and coordinate resources
 - 3) Provide *security and protection*
 - 4) Provide *fairness* among users (or programs)

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**’ 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)

(1) OS creates resource abstractions

```
foo(int x) { ... }
```



Users/Other Machines

Applications

Compiler

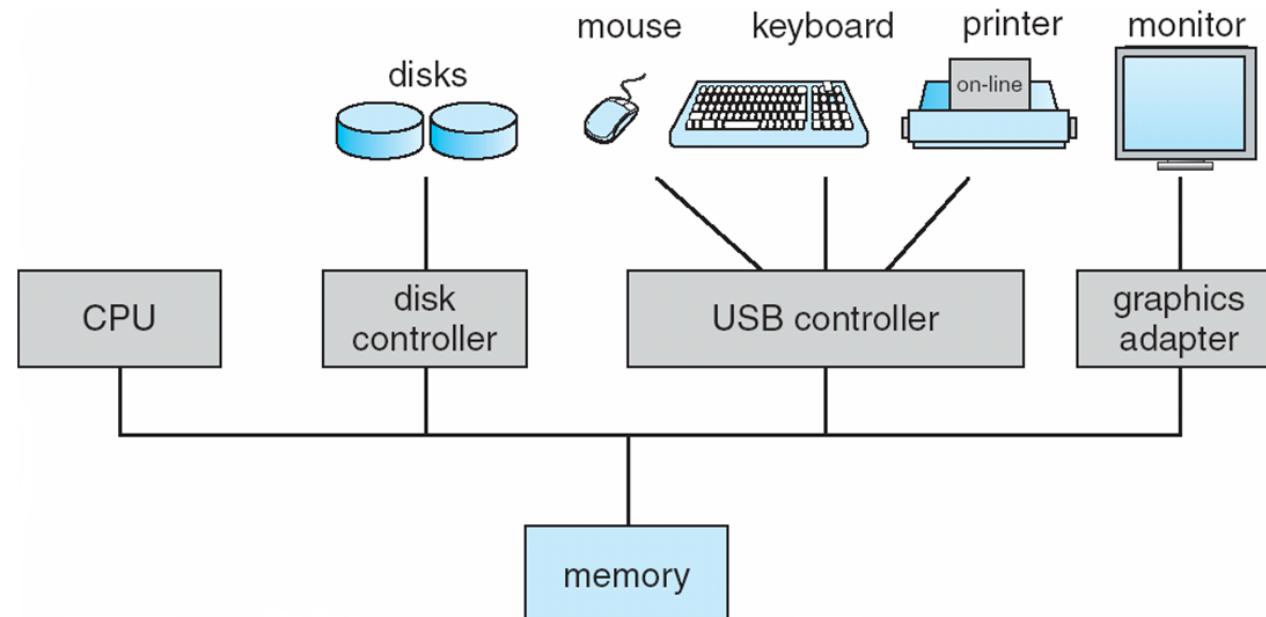
Operating System

Hardware

There are other layers in software stack such as runtime, libraries etc.
Operating System and Compilers are essentials.

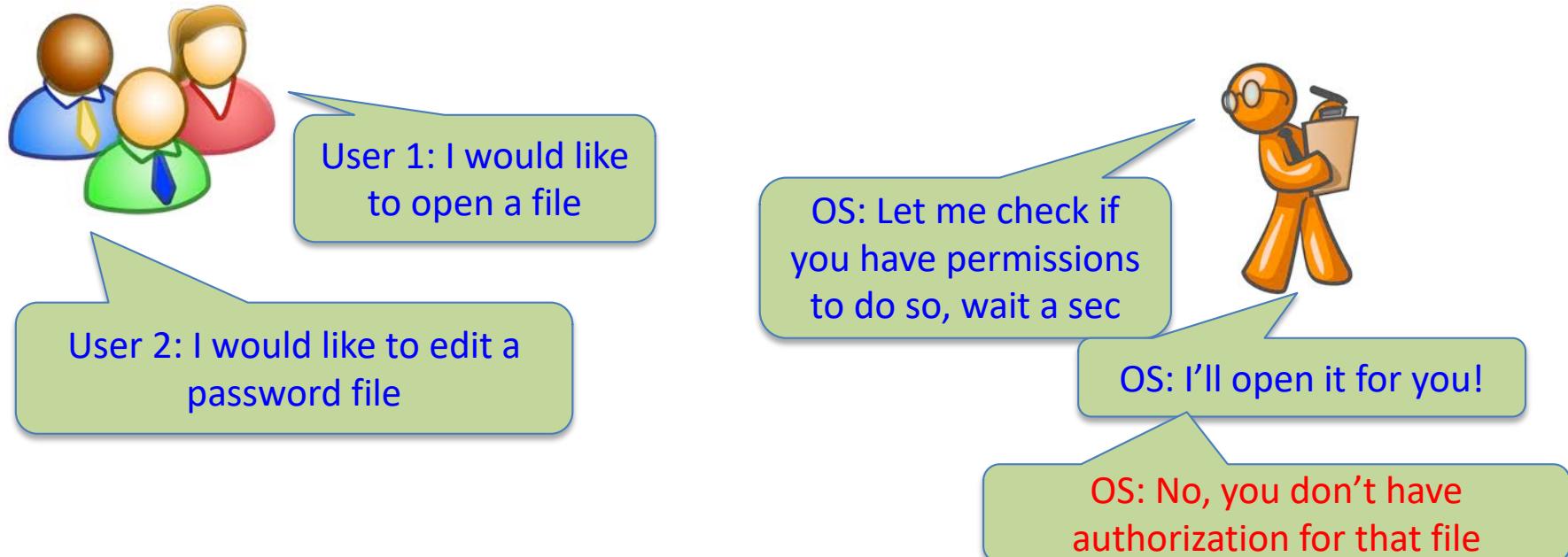
(2) OS manages resources

- OS is a **resource allocator**
 - Manages all resources for processes
 - Decides between conflicting requests for efficient and fair resource use



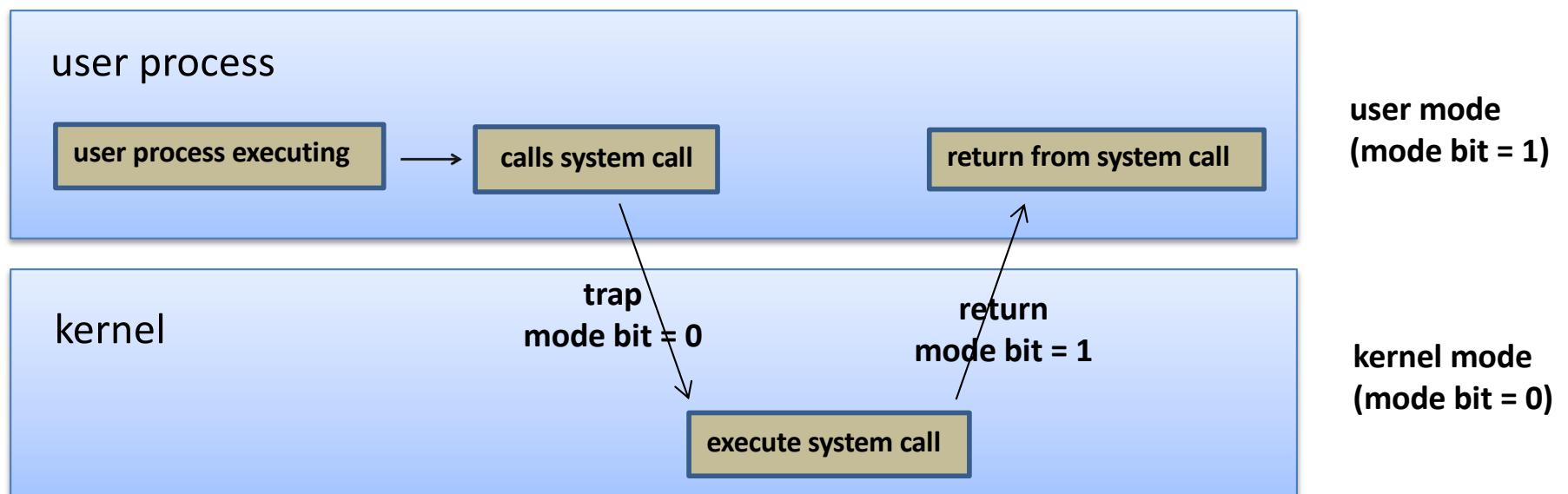
(3) OS provides protection and security

- OS is a **control program**
 - Controls execution of programs to prevent errors and improper/malicious use of the computer
 - Dual mode and Multimode OS
 - User mode and Kernel mode



(3) OS provides protection and security

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

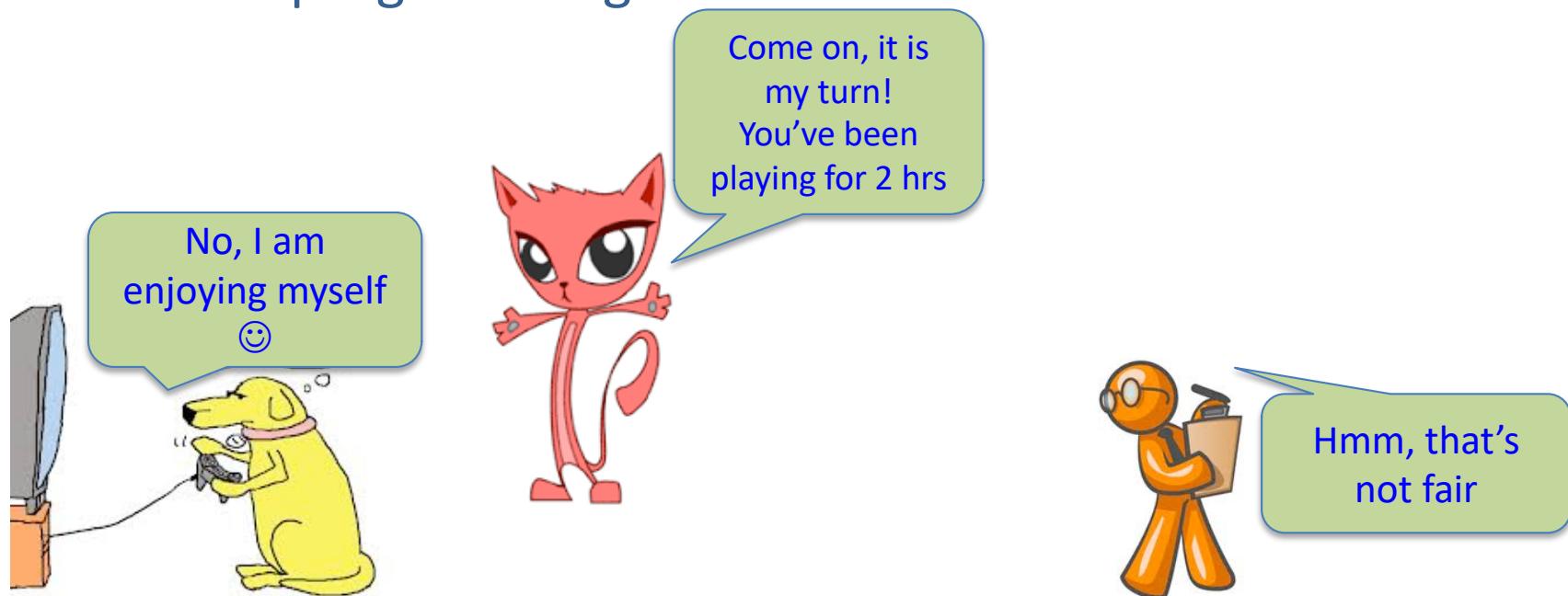


Interrupts

- An operating system is **interrupt driven**
 - It sits and waits for an event to occur
- Device or hardware interrupts
 - I/O device is done or
 - Hardware throws an exception (e.g. overflow)
- Software interrupts
 - A **trap** or **exception** is a software-generated interrupt caused either by an error or a user request (system call)
- OS has an **interrupt vector**, which contains the addresses of all the service routines for interrupt handling

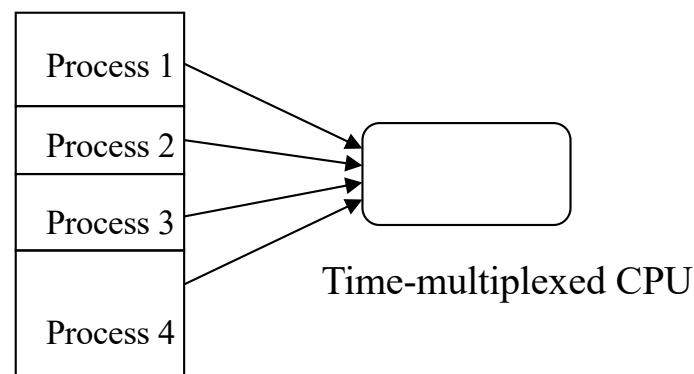
(4) OS provides *fair* execution

- OS provides fair execution and resource sharing between users and programs
 - via multiprogramming



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



Space-multiplexed Memory

Brief History of OS

- No operating system 1940s
 - Computers are exotic
 - Program in machine language
 - Programs manually loaded
 - No concurrency: no multiple jobs, no multiple users
- 1950s
 - First compiler is developed
 - OS uses batch scheduling
 - No human-computer interaction
 - Still used in servers, clusters and data centers today

Brief History of OS

- 1960s
 - Multics – one of the most important real OS
 - Hierarchical file system (directory structure)
 - Access control list and protection
 - <https://multicians.org/>
- 1970s
 - Computers became affordable
 - UNIX is born at Bell Labs by Ken Thompson and Dennis Ritchie
 - Written in C, allows people to experiment

Brief History of OS

- 1980s
 - MS-DOS
 - IBM needed software for their personal computers
 - Approached Bill Gates (Microsoft) and he created MS-DOS
 - BSD Unix
 - University of California developed BSD Unix
 - Became open source later
 - Mach
 - Carnegie Mellon Univ. developed Mach to replace Unix
 - Apple chose BSD/Mach as the foundation for MacOS X
- 1983
 - Richard Stallman started the GNU project
 - Advocates free, open-source UNIX compatible operating system
 - GNU General Public License (GPL) is now a common license under which free software is released

Brief History of OS

- 1990s
 - Linux
 - Developed by a student (Linus Torvalds) in Finland
 - Unix-based
 - Several distributions: SUSE, Fedora, Ubuntu, Redhat
 - Open-source operating system under GNU General Public License
 - Windows 95 and MacOS X became mature and complex
- 2000s
 - Mobile devices: Android (based on Linux)
 - Trend is to have a smaller OS (network storage)
 - Virtualization has become common (Vmware Player, VirtualBox etc.)

Reading

- From text book
 - Read 1.1, 1.4-1.10 (OS Structure – Kernel Data Structures)
 - Read 1.12 (Open-Source OS)
 - Read 1.2-1.3 if you want to refresh your Computer Architecture knowledge
- Install the Linux Distribution or Virtual Machine by next PS
- Subscribe to Blackboard Discussion Forum

Acknowledgments

- These slides are adapted from
 - Öznur Özkasap (Koç University)
 - Operating System and Concepts (9th edition) Wiley