Concurrency and Parallels computing

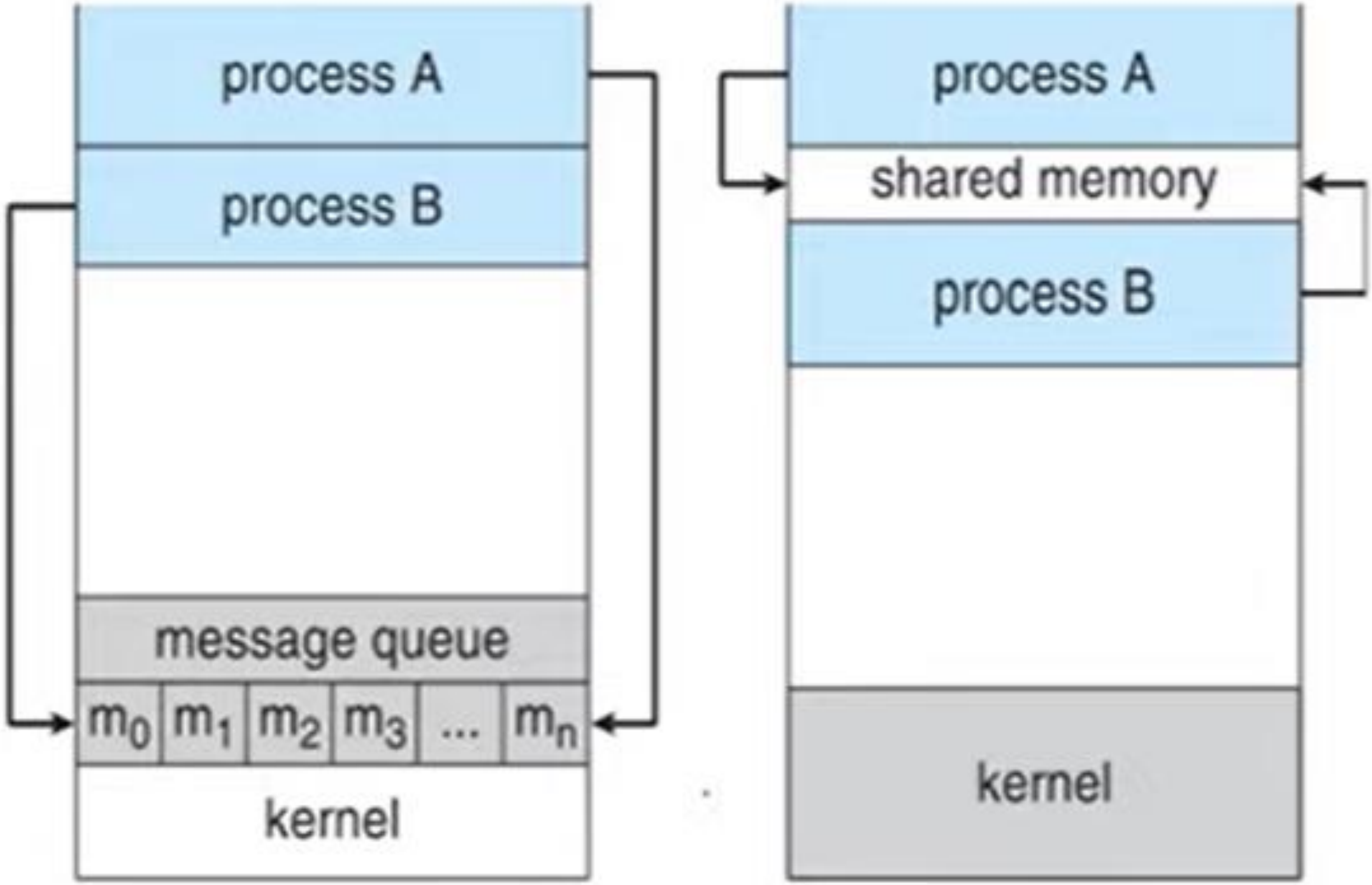
What is Concurrency

- It refers to the execution of multiple instruction sequences at the same time. It occurs in an operating system when multiple process threads are executing concurrently. These threads can interact with one another via shared memory or message passing.

Interprocess communication

- Processes within a system may be **independent** or **cooperating**
- Cooperating process can **affect** or be **affected** by **other processes**, including sharing data.
- Cooperating processes need **interprocess communication** (**IPC**)
- Two models of IPC:
  - ➢ **Shared memory**
  - ➢ **Message passing**

(a) Message passing.   (b) shared memory.



(a)   (b)

Difference between Concurrency and Parallelism in Operating System

- Concurrency and parallelism are related but not the same terms, and they are sometimes confused. The key distinction between concurrency and parallelism is that concurrency is concerned with dealing with several things simultaneously or managing concurrent events while essentially hiding latency. In contrast, parallelism is about doing multiple tasks simultaneously that help to increase the system speed.

Problems in Concurrency

There are various problems in concurrency. Some of them are as follows:

- **Locating the programming errors**

It's difficult to spot a programming error because reports are usually repeatable due to the varying states of shared components each time the code is executed.

- **Sharing Global Resources**

Sharing global resources is difficult. If two processes utilize a global variable and both alter the variable's value, the order in which the many changes are executed is critical.

- **Locking the channel**

It could be inefficient for the OS to lock the resource and prevent other processes from using it.

- **Optimal Allocation of Resources**

It is challenging for the OS to handle resource allocation properly

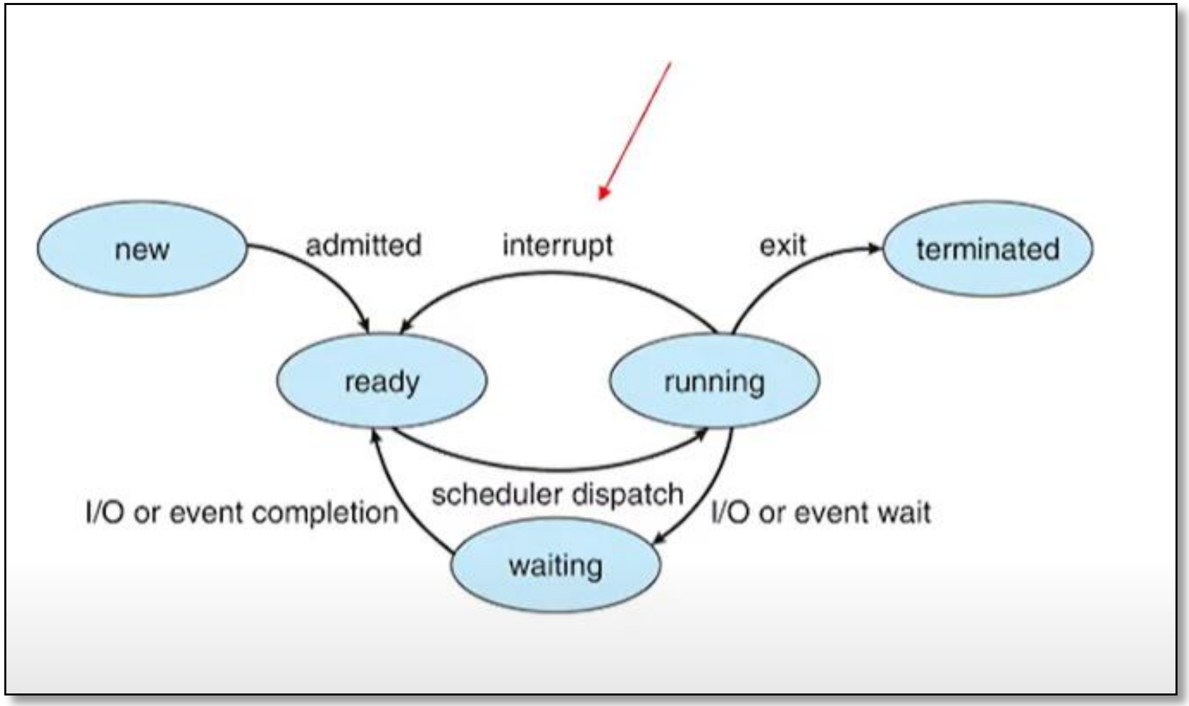
Concurrency and Parallels computing

Process concept Program vs process

Program is passive entity stored on disk (executable file), **process** is active

- Program becomes process when executable file loaded into memory •
- Execution of program started via GUI mouse clicks, command line entry of its name, etc •
- One program can be several processes

Diagram of process state



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
- Illustration of the problem:
  Suppose that we wanted to provide a solution to the consumer-producer problem that fills ***all*** the buffers. We can do so by having an integer `counter` that keeps track of the number of full buffers. Initially, `counter` is set to 0. It is incremented by the producer after it produces a new buffer and is decremented by the consumer after it consumes a buffer.

Producer

```
while (true) {
        /* produce an item in next produced */

        while (counter == BUFFER_SIZE)
                ; /* do nothing */
        buffer[in] = next_produced;
        in = (in + 1) % BUFFER_SIZE;
        counter++;
}
```

Consumer

```
while (true) {
        while (counter == 0)
                ; /* do nothing */
        next_consumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
         counter--;
        /* consume the item in next consumed */
}
```

Race condition

- **`counter++`** could be implemented as

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

- **`counter--`** could be implemented as

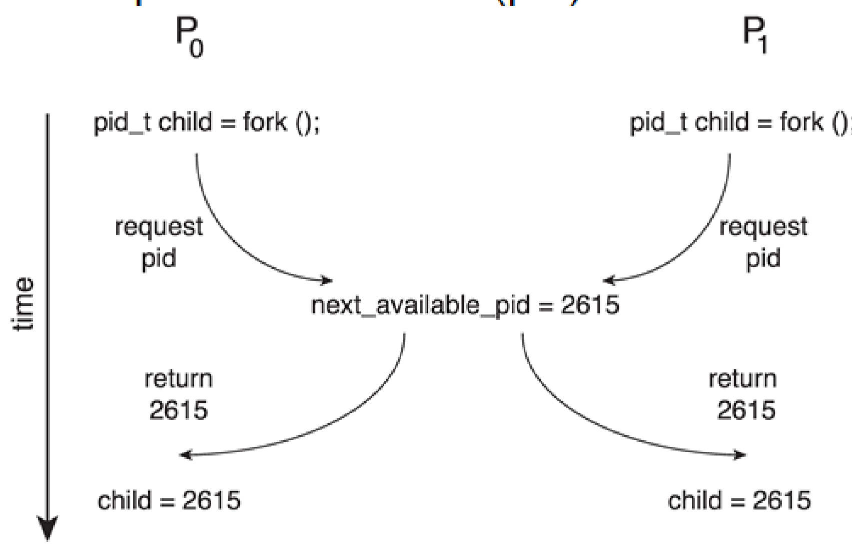
```
register2 = counter
register2 = register2 - 1
counter = register2
```

- Consider this execution 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}

Race condition

- Processes P_0 and P_1 are creating child processs using the `fork()` system call
- Race condition on kernel variable `next_available_pid` which represents the next available process identifier (pid)



- Unless there is mutual exclusion, the same pid could be assigned to two different processes!

Critical Section Problem

- Consider system of ***n*** processes $\{p_0, p_1, \ldots p_{n-1}\}$
- Each process has **critical section** segment of code
  - Process may be changing common variables, updating table, writing file, etc
  - When one process in critical section, no other may be in its critical section
- ***Critical section problem*** is to design protocol to solve this
- Each process must ask permission to enter critical section in **entry section**, may follow critical section with **exit section**, then **remainder section**

Critical section

- General structure of process P_i

```
do {
    entry section

        critical section

    exit section

        remainder section

} while (true);
```

Solution to Critical-Section Problem

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
   - Assume that each process executes at a nonzero speed
   - No assumption concerning **relative speed** of the n processes

Critical-Section Handling in OS

Two approaches depending on if kernel is preemptive or non- preemptive

- **Preemptive** – allows preemption of process when running in kernel mode
- **Non-preemptive** – runs until exits kernel mode, blocks, or voluntarily yields CPU
  - Essentially free of race conditions in kernel mode

Concurrency and Parallels computing

Algorithm 1 strict alternation

- Shared variables:
  - **int turn**;
    initially **turn = 0**
  - **turn - i** ⇒ P_i can enter its critical section
- Process P_i

```
do {
    while (turn != i) ;
        critical section
    turn = j;
        reminder section
} while (1);
```

pi

```
do{
  while(turn!=i);
  critical section
  turn=j;
       reminder section
} while(1);
```

pj

```
do{
  while(turn!=j);
  critical section
  turn=i;
       reminder section
} while(1);
```

Satisfies mutual exclusion, but not progress

Algorithm 2

- Shared variables
  - **boolean flag[2]**;
    initially **flag [0] = flag [1] = false.**
  - **flag [i] = true** ⇒ P_i ready to enter its critical section
- Process P_i

```
do {
    flag[i] := true;
    while (flag[j]) ;
        critical section
    flag [i] = false;
        remainder section
} while (1);
```

pi

```
do{
  flag[i]:=true;
  while(flag[j]);
  critical section
  flag[i]=false;
       reminder
section
}while(1);
```

pj

```
do{
  flag[j]:=true;
  while(flag[i]);
  critical section
  flag[j]=false;
       reminder
section
}while(1);
```

Satisfies mutual exclusion, but not progress

Algorithm 2

- Shared variables
  - **boolean flag[2]**;
    initially **flag [0] = flag [1] = false.**
  - **flag [i] = true** ⇒ P_i ready to enter its critical section
- Process P_i

```
do {
    while (flag[j]) ;      <- Swap
    flag[i] := true;       <- Swap
        critical section
    flag [i] = false;
        remainder section
} while (1);
```

mutual exclusion is not satisfied

Peterson's solution

- Not guaranteed to work on modern architectures! (But good algorithmic description of solving the problem)
- Two process solution
- Assume that the **load** and **store** machine-language instructions are atomic; that is, cannot be interrupted
- The two processes share two variables:
  - **int turn;**
  - **boolean flag[2]**
- The variable **turn** indicates whose turn it is to enter the critical section
- The **flag** array is used to indicate if a process is ready to enter the critical section. **flag[i] = *true*** implies that process P_i is ready!

Algorithm for Process P_i

```
while (true){
        flag[i] = true;
        turn = j;
        while (flag[j] && turn = = j)
              ;

        /* critical section */

        flag[i] = false;

        /* remainder section */

}
```

```
while (true){
        flag[i] = true;
        turn = j;
        while (flag[j] && turn = = j)
                ;

        /* critical section */

        flag[i] = false;

        /* remainder section */

}
```

```
while (true){
        flag[i] = true;
        turn = j;
        while (flag[j] && turn = = j)
                ;

        /* critical section */

        flag[i] = false;

        /* remainder section */

}
```

Peterson's solution

- Provable that the three CS requirement are met:
  1. Mutual exclusion is preserved

     P_i enters CS only if:

     either `flag[j] = false` or `turn = i`
  2. Progress requirement is satisfied
  3. Bounded-waiting requirement is met

Peterson's solution

- Although useful for demonstrating an algorithm, Peterson's Solution is not guaranteed to work on modern architectures.
- Understanding why it will not work is also useful for better understanding race conditions.
- To improve performance, processors and/or compilers may reorder operations that have no dependencies.
- For single-threaded this is ok as the result will always be the same.
- For multithreaded the reordering may produce inconsistent or unexpected results!

Code Optimizations Wreak Havoc with Multiple Threads

```
// Compile with "/platform:x86 /o" and run it NOT under the debugger
internal static class StrangeBehavior {
   private static Boolean s_stopWorker = false;

   public static void Main() {
      Console.WriteLine("Main: letting worker run for 5 seconds");
      Thread t = new Thread(Worker); t.Start();
      Thread.Sleep(5000);
      s_stopWorker = true;
      Console.WriteLine("Main: waiting for worker to stop");
      t.Join();
   }

   private static void Worker(Object o) {
      Int32 x = 0;
      while (!s_stopWorker) x++;
      Console.WriteLine("Worker: x={0}", x);
   }
}
```

```
//compiler optimizes to this:
Int32 x=0;
If(!s_stopWorker)
While(true) x++;
Console.WriteLine("Worker: x={0}", x);
```

Code Optimizations Cause Problems

```
class OutOfProgramOrder {
   private Boolean m_flag  = false;
   private Int32   m_value = 0;

   public void Thread1() {
      // These could execute in reverse order
      m_value = 5;
      m_flag = true;
   }

   public void Thread2() {
      // m_value could be read before m_flag
      if (m_flag)
         Display(m_value); // Nothing or 5?
   }
}
```

- Two threads share the data:

```
boolean flag = false;
int x = 0;
```

- Thread 1 performs

```
while (!flag)
    ;
print x
```

- Thread 2 performs

```
x = 100;
flag = true
```

- What is the expected output?

Peterson's Solution

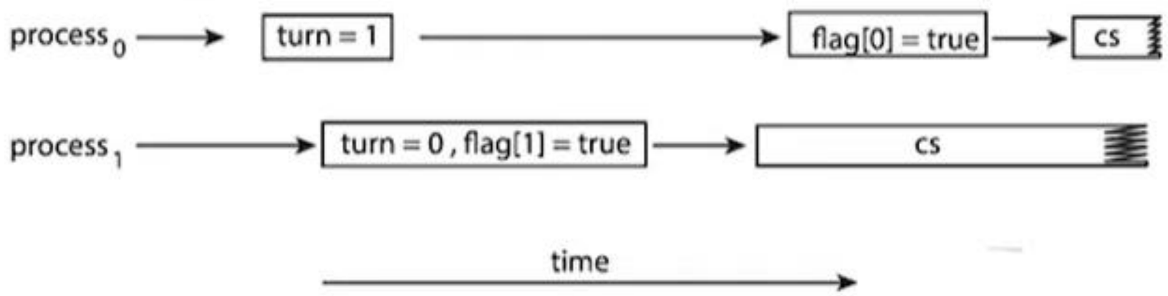
100 is the expected output.

However, the operations for Thread 2 may be reordered:

```
flag = true;
x = 100;
```

If this occurs, the output may be 0!

The effects of instruction reordering in Peterson's Solution



Synchronization hardware

- Many systems provide hardware support for implementing the critical section code.
- Uniprocessors – could disable interrupts
  - Currently running code would execute without preemption
  - Generally too inefficient on multiprocessor systems
    - Operating systems using this not broadly scalable
- We will look at three forms of hardware support:

1. Memory barriers
2. Hardware instructions
3. Atomic variables

Memory Barriers

- **Memory model** are the memory guarantees a computer architecture makes to application programs.
- Memory models may be either:

- **Strongly ordered** – where a memory modification of one processor is immediately visible to all other processors.
- **Weakly ordered** – where a memory modification of one processor may not be immediately visible to all other processors.

- A **memory barrier** is an instruction that forces any change in memory to be propagated (made visible) to all other processors.

Memory Barrier

- We could add a memory barrier to the following instructions to ensure Thread 1 outputs 100:
- Thread 1 now performs

```
while (!flag)
    memory_barrier();
print x
```

- Thread 2 now performs

```
x = 100;
memory_barrier();
flag = true
```

Hardware Instructions

- Special hardware instructions that allow us to either *test-and-modify* the content of a word, or to *swap* the contents of two words atomically (uninterruptibly.)
- **Test-and-Set** instruction
- **Compare-and-Swap** instruction

Atomic Variables

- Typically, instructions such as compare-and-swap are used as building blocks for other synchronization tools.
- One tool is an **atomic variable** that provides *atomic* (uninterruptible) updates on basic data types such as integers and booleans.
- For example, the **`increment()`** operation on the atomic variable **`sequence`** ensures **`sequence`** is incremented without interruption:

```
increment(&sequence);
```

Atomic Variables

- The **`increment()`** function can be implemented as follows:

```
void increment(atomic_int *v)
{
   int temp;

   do {
         temp = *v;
   }
   while (temp !=
(compare_and_swap(v,temp,temp+1));
}
```

Atomic Variables

It is important to note that although atomic variables provide atomic updates, they do not entirely solve race conditions in all circumstances. For example, in the bounded-buffer problem described in Section 6.1, we could use an atomic integer for `count`. This would ensure that the updates to `count` were atomic. However, the producer and consumer processes also have `while` loops whose condition depends on the value of `count`. Consider a situation in which the buffer is currently empty and two consumers are looping while waiting for `count` > 0. If a producer entered one item in the buffer, both consumers could exit their `while` loops (as `count` would no longer be equal to 0) and proceed to consume, even though the value of `count` was only set to 1.

Solution to the critical section problem

semaphore

- Semaphore **S** – integer variable
- Can only be accessed via two indivisible (atomic) operations
  - **wait()** and **signal()**
    - Originally called **P()** and **V()**
- Definition of the **wait()** **operation**

```
wait(S) {
    while (S <= 0)
       ; // busy wait
    S--;
}
```

- Definition of the **signal()** **operation**

```
signal(S) {
    S++;
}
```

Semaphore

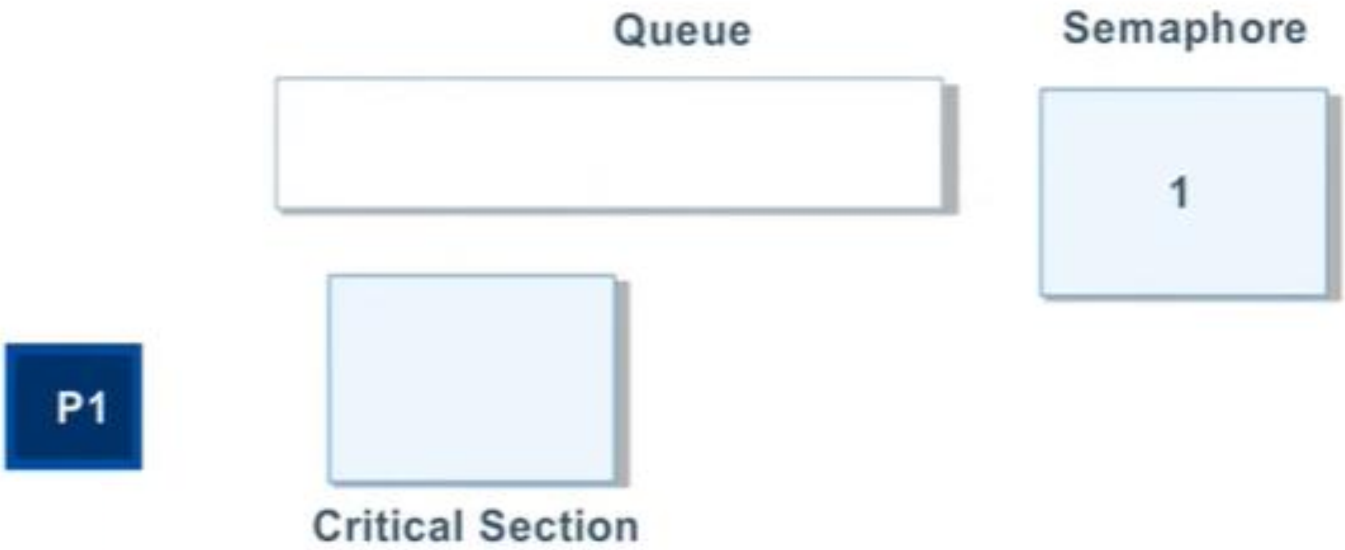
- To overcome the **problem** of **busy waiting in semaphore**, we can modify the defining of the wait and signal operation.
- When a process executes the wait operation and find the value of semaphore is not positive, it must wait. However rather than engaging in a busy waiting, the process can block itself in waiting queue associated with each semaphore.
- In this case the state of the process is waiting in the **semaphore queue**.

Implementation of wait:

```
wait (S){
value--;
if (value < 0) {
add this
process to waiting queue
block(); }
}
```

Implementation of signal:

```
Signal (S){
value++;
if (value <= 0) {
remove a
process P from the waiting queue
wakeup(P); }
}
```



Queue
Semaphore
1
P1
Critical Section

- **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

```
   P0                    P1
wait(S);              wait(Q);
wait(Q);              wait(S);
 ...                   ...
signal(S);            signal(Q);
signal(Q);            signal(S);
```

- Other forms of deadlock:
- **Starvation** – indefinite blocking
  - A process may never be removed from the semaphore queue in which it is suspended
- **Priority Inversion** – Scheduling problem when lower-priority process holds a lock needed by higher-priority process
- Solved via **priority-inheritance protocol**

Priority inheritance protocol

- Consider the scenario with three processes **P1, P2**, and **P3**. **P1** has the highest priority, **P2** the next highest, and **P3** the lowest. Assume a resouce **P3** is assigned a resource **R** that **P1** wants. Thus, **P1** must wait for **P3** to finish using the resource. However, **P2** becomes runnable and preempts **P3**. What has happened is that **P2** - a process with a lower priority than **P1** - has indirectly prevented **P3** from gaining access to the resource.

- To prevent this from occurring, a **priority inheritance protocol** is used. This simply allows the priority of the highest thread waiting to access a shared resource to be assigned to the thread currently using the resource. Thus, the current owner of the resource is assigned the priority of the highest priority thread wishing to acquire the resource.

System model

- System consists of resources
- Resource types $R_1, R_2, \ldots, R_m$

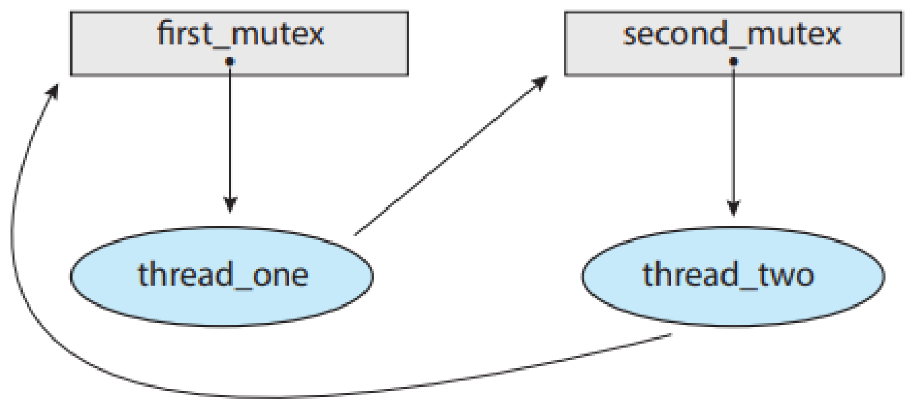
  CPU cycles, memory space, I/O devices
- Each resource type R_i has W_i instances.
- Each process utilizes a resource as follows:
  - **request**
  - **use**
  - **release**

Deadlock in Multithreaded Applications

```
/* thread_one runs in this function */
void *do_work_one(void *param)
{
   pthread_mutex_lock(&first_mutex);
   pthread_mutex_lock(&second_mutex);
   /**
    * Do some work
    */
   pthread_mutex_unlock(&second_mutex);
   pthread_mutex_unlock(&first_mutex);

   pthread_exit(0);
}

/* thread_two runs in this function */
void *do_work_two(void *param)
{
   pthread_mutex_lock(&second_mutex);
   pthread_mutex_lock(&first_mutex);
   /**
    * Do some work
    */
   pthread_mutex_unlock(&first_mutex);
   pthread_mutex_unlock(&second_mutex);

   pthread_exit(0);
}
```

Deadlock in Multithreaded Applications

- Deadlock is possible if thread 1 acquires **`first_mutex`** and thread 2 acquires **`second_mutex`**. Thread 1 then waits for **`second_mutex`** and thread 2 waits for **`first_mutex`**.
- Can be illustrated with a **resource allocation graph**:



Deadlock characterization

Deadlock can arise if four conditions hold simultaneously.

- **Mutual exclusion**: only one process at a time can use a resource
- **Hold and wait**: a process holding at least one resource is waiting to acquire additional resources held by other processes
- **No preemption**: a resource can be released only voluntarily by the process holding it, after that process has completed its task
- **Circular wait**: there exists a set $\{P_0, P_1, \ldots, P_n\}$ of waiting processes such that P_0 is waiting for a resource that is held by P_1, P_1 is waiting for a resource that is held by P_2, …, P_{n-1} is waiting for a resource that is held by P_n, and P_n is waiting for a resource that is held by P_0.

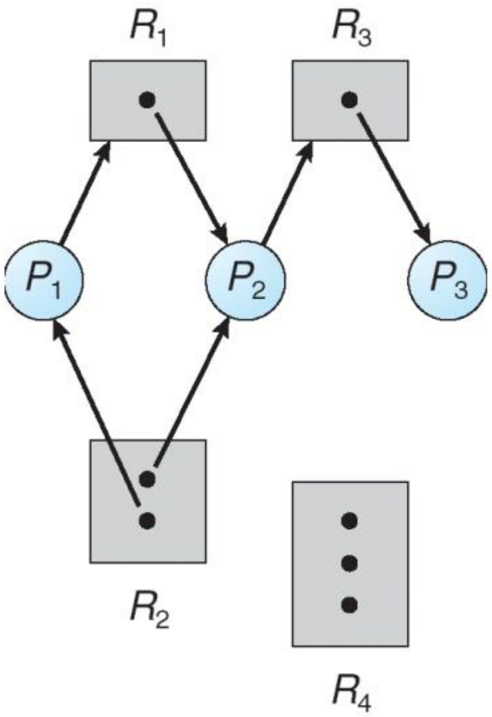
Resource allocation graph

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

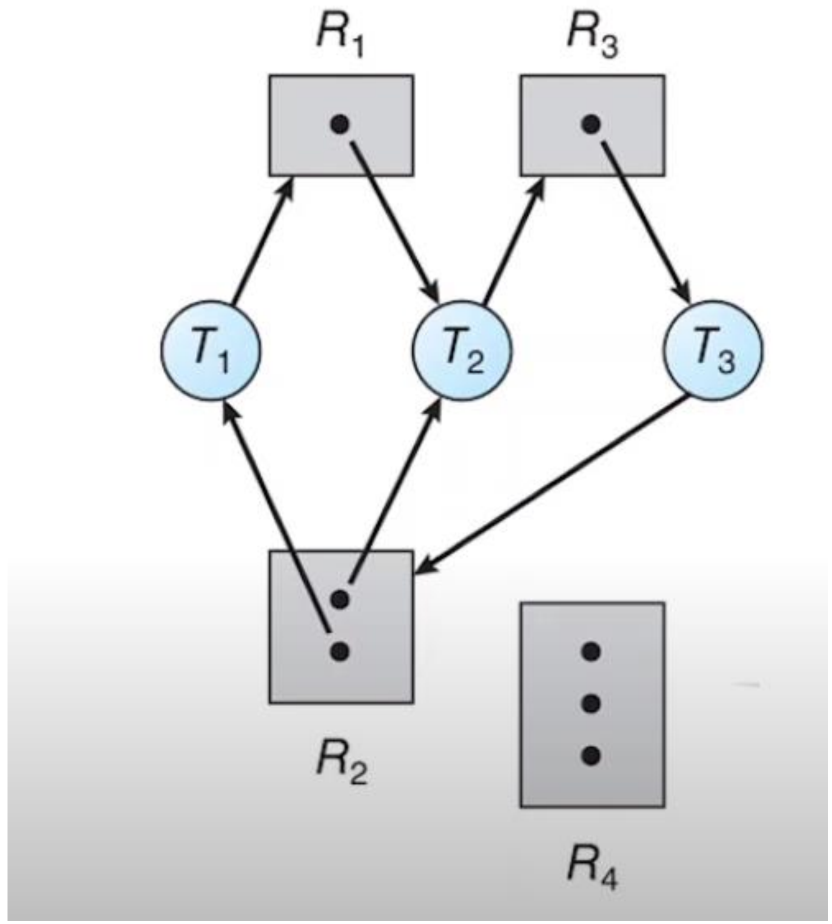
- V is partitioned into two types:
  - $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system
  - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system
- **request edge** – directed edge $P_i \rightarrow R_j$
- **assignment edge** – directed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

Resource allocation graph example

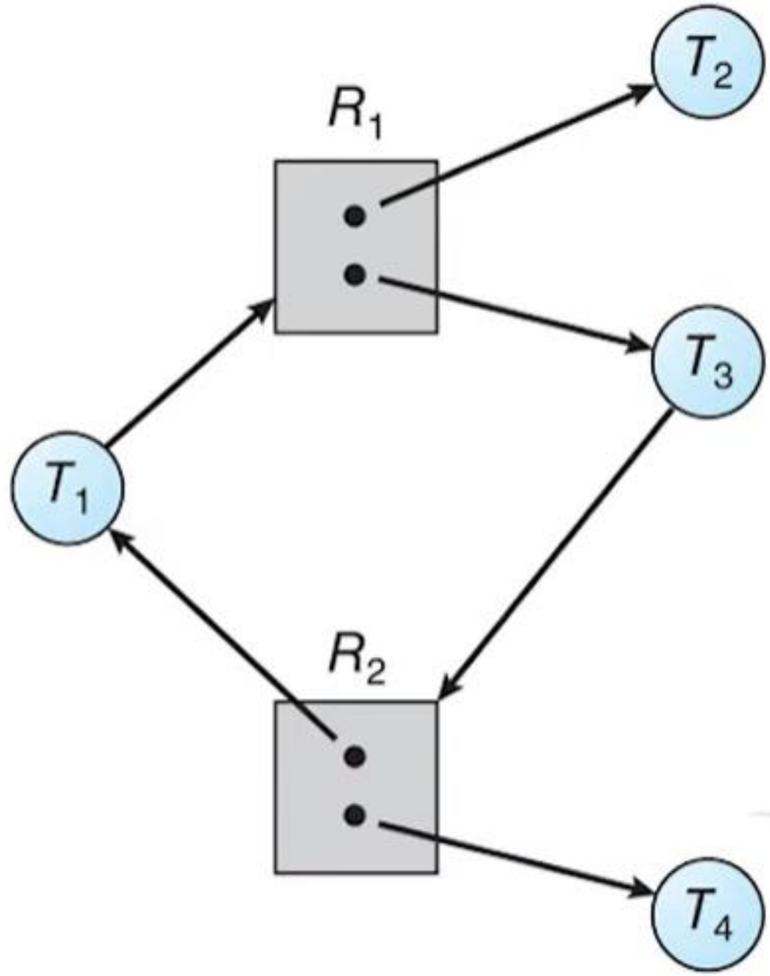
- One instance of R1
- Two instances of R2
- One instance of R3
- Three instance of R4
- T1 holds one instance of R2 and is waiting for an instance of R1
- T2 holds one instance of R1, one instance of R2, and is waiting for an instance of R3
- T3 is holds one instance of R3



Resource allocation graph with a deadlock



Graph with a cycle but no Deadlock



Basic Facts

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

Methods for handling Deadlocks

- Ensure that the system will ***never*** enter a deadlock state:
  - Deadlock prevention
  - Deadlock avoidance
- Allow the system to enter a deadlock state and then recover
- Ignore the problem and pretend that deadlocks never occur in the system.

Deadlock prevention

Invalidate one of the four necessary conditions for deadlock:

- **Mutual Exclusion** – not required for sharable resources (e.g., read-only files); must hold for non-sharable resources
- **Hold and Wait** – must guarantee that whenever a process requests a resource, it does not hold any other resources
  - 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 allocated to it.
  - Low resource utilization; starvation possible

- **No Preemption** –
  - If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released
  - Preempted resources are added to the list of resources for which the process is waiting
  - 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

Solution to the critical section problem

Deadlock avoidance

Requires that the system has some additional ***a priori*** information available

- Simplest and most useful model requires that each process declare the ***maximum number*** of resources of each type that it may need
- The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition
- Resource-allocation *state* is defined by the number of available and allocated resources, and the maximum demands of the processes

Safe State

- When a process requests an available resource, system must decide if immediate allocation leaves the system in a safe state
- System is in **safe state** if there exists a sequence <P_1, P_2, …, P_n> of ALL the processes in the systems such that 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 $j < i$
- That is:
  - If P_i 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

Basic Facts

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

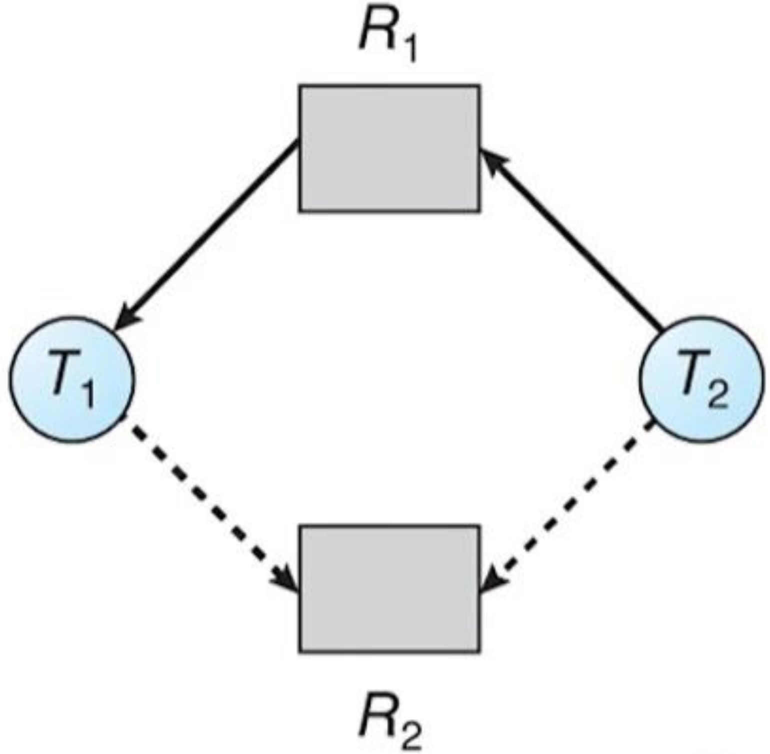
Avoidance Algorithms

- Single instance of a resource type
  - Use a resource-allocation graph
- Multiple instances of a resource type
  - Use the Banker's Algorithm

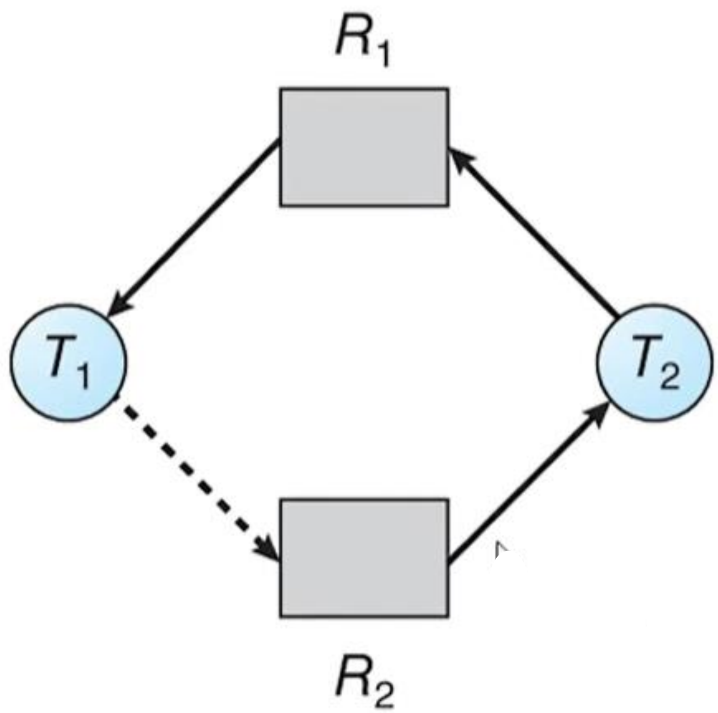
Resource- Allocation Graph Scheme

- **Claim edge** $P_i \rightarrow R_j$ indicated that process P_j may request resource R_j; represented by a dashed line
- Claim edge converts to request edge when a process requests a resource
- Request edge converted to an assignment edge when the resource is allocated to the process
- When a resource is released by a process, assignment edge reconverts to a claim edge
- Resources must be claimed *a priori* in the system

Resource-Allocation Graph



Unsafe State In Resource-Allocation Graph



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

Banker's Algorithm

- Multiple instances of resources
- Each process must a priori claim maximum use
- When a process requests a resource it may have to wait
- When a process gets all its resources it must return them in a finite amount of time

To illustrate, consider a system with twelve resources and three threads: T_0, T_1, and T_2. Thread T_0 requires ten resources, thread T_1 may need as many as four, and thread T_2 may need up to nine resources. Suppose that, at time t_0, thread T_0 is holding five resources, thread T_1 is holding two resources, and thread T_2 is holding two resources. (Thus, there are three free resources.)

	Maximum Needs	Current Needs
T_0	10	5
T_1	4	2
T_2	9	2

Data Structures for Banker's Algorithm

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

- **Available**: Vector of length m. If available $[j] = k$, there are k instances of resource type R_j available
- **Max**: $n \times m$ matrix. If $Max[i,j] = k$, then process P_i may request at most k instances of resource type R_j
- **Allocation**: $n \times m$ matrix. If Allocation$[i,j] = k$ then P_i is currently allocated k instances of R_j
- **Need**: $n \times m$ matrix. If $Need[i,j] = k$, then P_i may need k more instances of R_j to complete its task

$$Need[i,j] = Max[i,j] - Allocation[i,j]$$

Safety Algorithm

1. Let ***Work*** and ***Finish*** be vectors of length *m* and *n*, respecti
   Initialize:

   Work* = *Available

   ***Finish* [*i*] = *false* for *i* = 0, 1, …, *n*- 1**

2. Find an ***i*** such that both:

   (a) ***Finish* [*i*] = *false***

   (b) $\boldsymbol{Need_i \leq Work}$

   If no such ***i*** exists, go to step 4

3. $\boldsymbol{Work = Work + Allocation_i}$
   Finish*[*i*] = *true
   go to step 2

4. If ***Finish* [*i*] == *true*** for all *i*, then the system is in a safe state

Resource-Request Algorithm for Process P_i

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

1. If $Request_i \leq Need_i$ go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim
2. If $Request_i \leq Available$, go to step 3. Otherwise P_i must wait, since resources are not available
3. Pretend to allocate requested resources to P_i by modifying the state as follows:

$$Available = Available - Request_i;$$

$$Allocation_i = Allocation_i + Request_i;$$

$$Need_i = Need_i - Request_i;$$

- If safe $\Rightarrow$ the resources are allocated to P_i
- If unsafe $\Rightarrow$ P_i must wait, and the old resource-allocation state is restored

- 5 processes P_0 through P_4;

  3 resource types:

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

- Snapshot at time T_0:

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

The content of the matrix ***Need*** is defined to be ***Max*** – ***Allocation***

	Need
	A B C
P_0	7 4 3
P_1	1 2 2
P_2	6 0 0
P_3	0 1 1
P_4	4 3 1

The system is in a safe state since the sequence < P_1, P_3, P_4, P_2, P_0> satisfies safety criteria

	Allocation			Max			Need			Available		
p0	0	1	0	7	5	3	7	4	3	3	3	2
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			