Process

a process is a program in execution

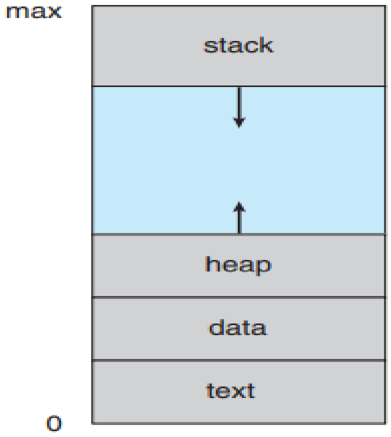


Figure 3.1 Process in memory.

Section	**Description**
Text	Contains the program code
Program Counter(PC)	Contains address of next instruction
Data	Contains global variables
Heap	Memory that is dynamically allocated during process run time
Stack	Contains temporary data (such as function parameters, return addresses, and local variables)

- A program is a passive entity, such as a file containing a list of instructions stored on disk (often called an executable file). In contrast, a process is an active entity, with a program counter specifying the next instruction to execute and a set of associated resources.
- Although two processes may be associated with the same program, they are always treated as separate processes.
- For instance, several users may be running different copies of the mail program, or the same user may invoke many copies of the web browser program. Each of these is a separate process; and although the text sections are equivalent, the data, heap, and stack sections vary.

Process States

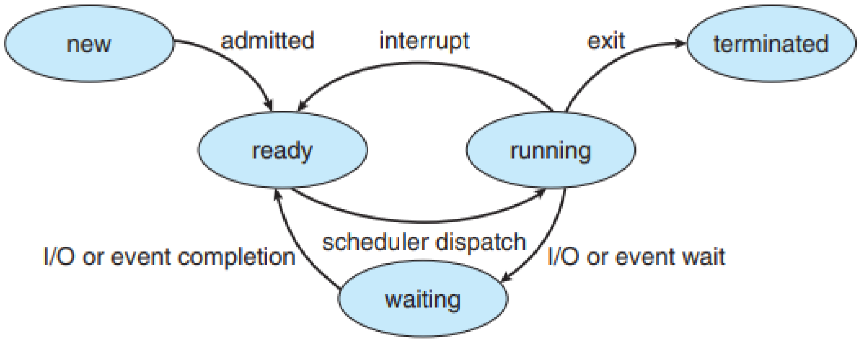
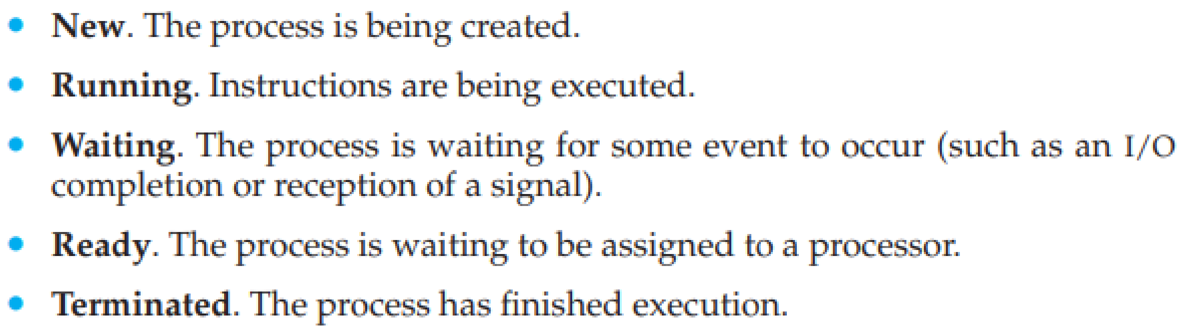


Figure 3.2 Diagram of process state.



- **New**. The process is being created.
- **Running**. Instructions are being executed.
- **Waiting**. The process is waiting for some event to occur (such as an I/O completion or reception of a signal).
- **Ready**. The process is waiting to be assigned to a processor.
- **Terminated**. The process has finished execution.

Process Control Block

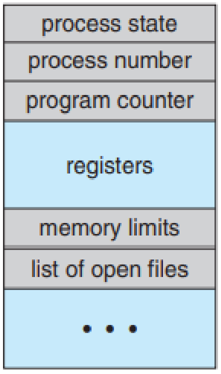


Figure 3.3 Process control block (PCB).

Each process is represented in the operating system by a process control block (PCB)—also called a task control block.

- Process state
  - The state may be new, ready, running, waiting, halted, and so on.
- Program counter.
  - The counter indicates the address of the next instruction to be executed for this process.
- CPU registers.
  - The registers vary in number and type, depending on the computer architecture. They include accumulators, index registers, stack pointers, and general-purpose registers, plus any condition-code information. Along with the program counter, this state information must be saved when an interrupt occurs, to allow the process to be continued correctly afterward (Figure 3.4).
- CPU-scheduling information.
  - This information includes a process priority, pointers to scheduling queues, and any other scheduling parameters. (Chapter 6 describes process scheduling.)
- Memory-management information.
  - This information may include such items as the value of the base and limit registers and the page tables, or the segment tables, depending on the memory system used by the operating system (Chapter 8)

- Accounting information.
  - This information includes the amount of CPU and real time used, time limits, account numbers, job or process numbers, and so on. • I/O status

information. This information includes the list of I/O devices allocated to the process, a list of open files, and so on.

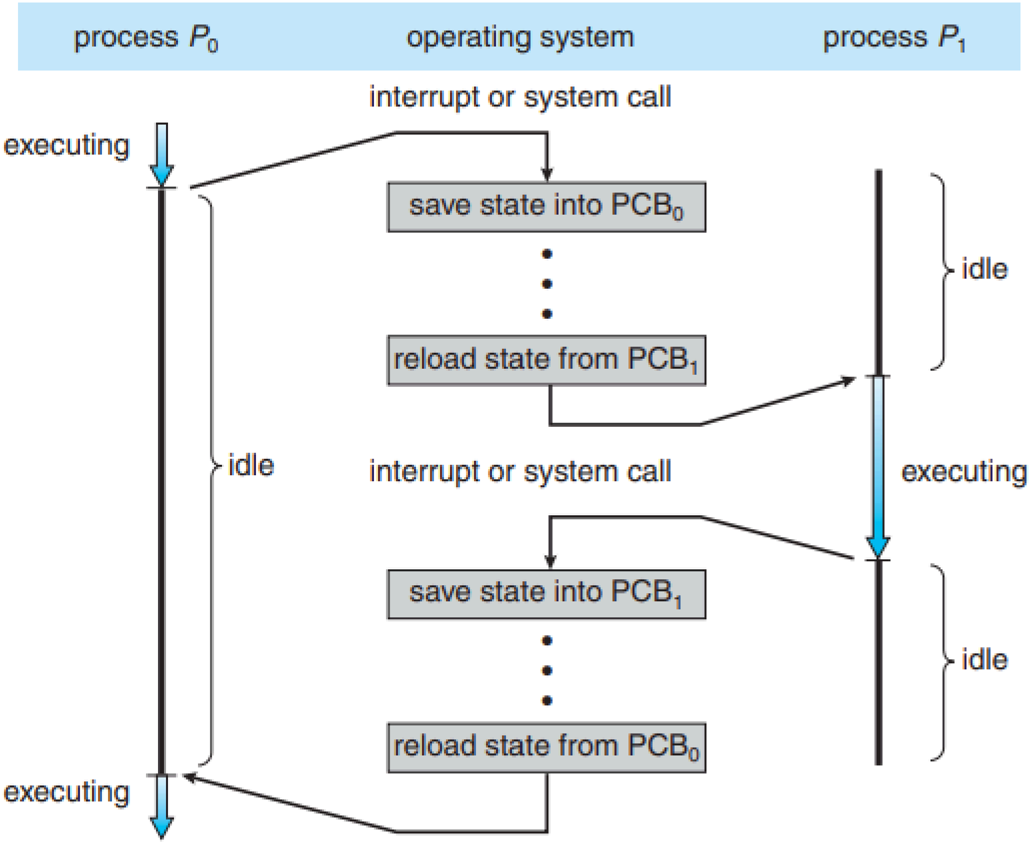


Figure 3.4 Diagram showing CPU switch from process to process.

PROCESS SCHEDULING

- The objective of multiprogramming is to have some process running at all times, to maximize CPU utilization.
- The objective of time sharing is to switch the CPU among processes so frequently that users can interact with each program while it is running.
- To meet these objectives, the process scheduler selects an available process (possibly from a set of several available processes) for program execution on the CPU.

Scheduling Queues

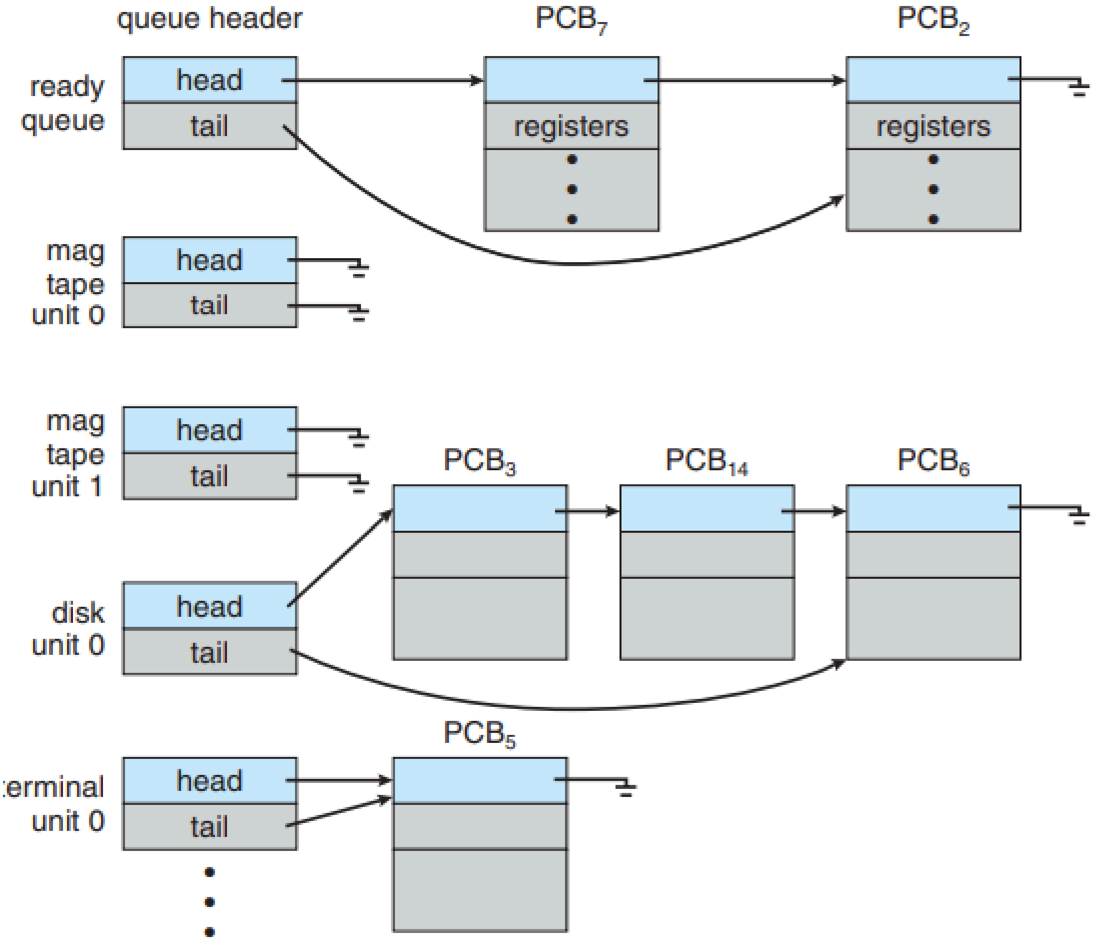


Figure 3.5 The ready queue and various I/O device queues.

- As processes enter the system, they are put into a **job queue**, which consists of all processes in the system.
- The processes that are residing in main memory and are ready and waiting to execute are kept on a list called the **ready queue**.
- Since there are many processes in the system, the disk may be busy with the I/O request of some other process. The process therefore may have to wait for the disk. The list of processes waiting for a particular I/O device is called a device queue. Each device has its own **device queue**.

- This queue is generally stored as a linked list. A ready-queue header contains pointers to the first and final PCBs in the list. Each PCB includes a pointer field that points to the next PCB in the ready queue.

Queueing Diagram

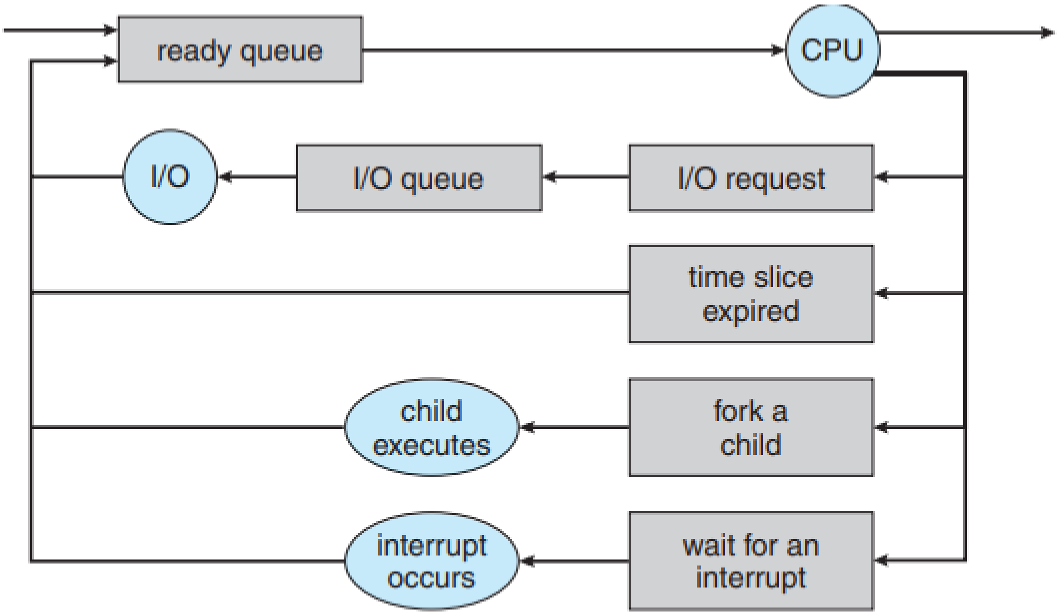


Figure 3.6 Queueing-diagram representation of process scheduling.

Flow in Queueing Diagram

- A new process is initially put in the ready queue. It waits there until it is selected for execution, or dispatched. Once the process is allocated the CPU and is executing, one of several events could occur:
- The process could issue an I/O request and then be placed in an I/O queue.
- The process could create a new child process and wait for the child's termination.
- The process could be removed forcibly from the CPU, as a result of an interrupt, and be put back in the ready queue.
- In the first two cases, the process eventually switches from the waiting state to the ready state and is then put back in the ready queue.
- A process continues this cycle until it terminates, at which time it is removed from all queues and has its PCB and resources deallocated.

Types of Schedulers

S. No.	Long-Term Scheduler	Short-Term Scheduler	Medium-Term Scheduler
1	It is a Job Scheduler	It is a CPU Scheduler	It is a process swapping scheduler
2	It takes process from the job pool	It takes process from the ready state	It takes process from running or wait/dead state
3	Its speed is lesser than short-term scheduler	It is fastest among the two other schedulers	Its speed is in between long-term and short-term
4	It controls the degree of multiprogramming	It has less control over the degree of multiprogramming	It reduces the degree of multiprogramming

Swapping in Medium Term Scheduler

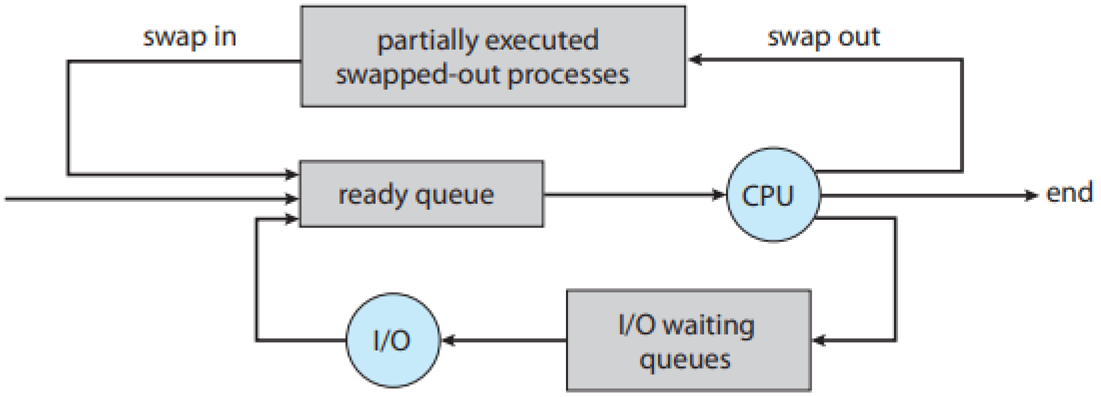


Figure 3.7 Addition of medium-term scheduling to the queueing diagram.

- It removes a process from memory (and from active contention for the CPU) and thus reduces the degree of multiprogramming.
- Later, the process can be reintroduced into memory, and its execution can be continued where it left off. This scheme is called **swapping.**

Context Switching

- Context switching is a technique or method used by the operating system to switch a process from one state to another to execute its function using CPUs in the system.
- When switching is performed in the system, it stores the old running process's PCB in the form of registers and assigns the CPU to a new process by reloading the PCB of the new process to execute its tasks.
- Context-switch time is pure overhead, because the system does no useful work while switching.
- Switching speed varies from machine to machine, depending on the memory speed, the number of registers that must be copied, and the existence of special instructions (such as a single instruction to load or store all registers). A typical speed is a few milliseconds.

CPU SCHEDULING ALGORITHMS

When scheduling is needed?

1. When a process switches from the running state to the waiting state (for example, as the result of an I/O request or an invocation of wait() for the termination of a child process)
2. When a process switches from the running state to the ready state (for example, when an interrupt occurs)
3. When a process switches from the waiting state to the ready state (for example, at completion of I/O)
4. When a process terminates

Two types of Scheduling

Preemptive Scheduling:

Preemptive scheduling is used when a process switches from running state to ready state or from the waiting state to ready state. The resources (mainly CPU cycles) are allocated to the process for a limited amount of time and then taken away, and the process is again placed back in the ready queue if that process still has CPU burst time remaining.

Non-Preemptive Scheduling:

Non-preemptive Scheduling is used when a process terminates, or a process switches from running to the waiting state. In this scheduling, once the resources (CPU cycles) are allocated to a process, the process holds the CPU till it gets terminated or reaches a waiting state.

Scheduling Criteria

1. **CPU utilization**
    - We want to keep the CPU as busy as possible. Conceptually, CPU utilization can range from 0 to 100 percent. In a real system, it should range from 40 percent (for a lightly loaded system) to 90 percent (for a heavily loaded system).
2. **Throughput**

- It is the number of processes that are completed per time unit, called throughput. For long processes, this rate may be one process per hour; for short transactions, it may be ten processes per second.

3. **Turnaround time**

- The interval from the time of submission of a process to the time of completion is the turnaround time. Turnaround time is the sum of the periods spent waiting to get into memory, waiting in the ready queue, executing on the CPU, and doing I/O.

4. **Waiting time**

- The CPU-scheduling algorithm does not affect the amount of time during which a process executes or does I/O. It affects only the amount of time that a process spends waiting in the ready queue. Waiting time is the sum of the periods spent waiting in the ready queue.

5. **Response time**

- In an interactive system, turnaround time may not be the best criterion.
- Response time is the time from the submission of a request until the first response is produced. It is the time it takes to start responding, not the time it takes to output the response.

Various Algorithms

1. **First Come First Served Scheduling**

- FIFO simply queues processes in the order that they arrive in the ready queue.
- In this, the process that comes first will be executed first and next process starts only after the previous gets fully executed.
- Here we are considering that arrival time for all processes is 0.

Drawbacks of FCFS:

1. Non-preemptive
2. Average Waiting Time is not optimal
3. Cannot utilize resources in parallel : Results in Convoy effect (Consider a situation when many IO bound processes are there and one CPU bound process. The IO bound processes have to wait for CPU bound process when CPU bound process acquires CPU. The IO bound process could have better taken CPU for some time, then used IO devices)

2. **Shortest Job First Algorithm**

- **Shortest Job First (SJF)** is an algorithm in which the process having the smallest execution time is chosen for the next execution. This scheduling method can be preemptive or non-preemptive. It significantly reduces the average waiting time for other processes awaiting execution.

Non preemptive	**Preemptive**
once the CPU cycle is allocated to a process, the process holds it till it reaches a waiting state or is terminated.	jobs are put into the ready queue as they come. A process with the shortest burst time begins execution. If a process with even a shorter burst time arrives, the current process is removed or preempted from execution, and the shorter job is allocated CPU cycle.

Advantages	**Drawbacks**
● SJF is frequently used for long term scheduling. ● It reduces the average waiting time over FIFO (First in First Out) algorithm. ● SJF method gives the lowest average waiting time for a specific set of processes.	● Job completion time must be known earlier, but it is hard to predict. ● SJF can't be implemented for CPU scheduling for the short term. It is because there is no specific method to predict the length of the upcoming CPU burst. ● This algorithm may cause very long turnaround times or starvation.

3. **Priority Scheduling**

- A priority is associated with each process, and the CPUis allocated to the process with the highest priority. Equal-priority processes are scheduled in FCFS order.
- Priorities can be defined either internally or externally.
  - Internally defined priorities use some measurable quantity or quantities to compute the priority of a process. For example, time limits, memory requirements etc.
  - External priorities are set by criteria outside the operating system, such as the importance of the process, the type and amount of funds being paid for computer use etc.
- Priority scheduling can be either preemptive or nonpreemptive. When a process arrives at the ready queue, its priority is compared with the priority of the currently running process.
- A preemptive priority scheduling algorithm will preempt the CPU if the priority of the newly arrived process is higher than the priority of the currently running process.
- A nonpreemptive priority scheduling algorithm will simply put the new process at the head of the ready queue.

Drawbacks:

- It causes indefinite blocking, or starvation. A process that is ready to run but waiting for the CPU can be considered blocked. A priority scheduling algorithm can leave some low priority processes waiting indefinitely.
- A solution to the problem of indefinite blockage of low-priority processes is aging. Aging involves gradually increasing the priority of processes that wait in the system for a long time.

4. **Round Robin Scheduling**

- Round robin is a pre-emptive algorithm
- The CPU is shifted to the next process after fixed interval time, which is called time quantum/time slice.
- The process that is preempted is added to the end of the queue.
- To implement RR scheduling, we again treat the ready queue as a FIFO queue of processes. New processes are added to the tail of the ready queue.
- The CPU scheduler picks the first process from the ready queue, sets a timer to interrupt after 1 time quantum, and dispatches the process

Advantages:

- It doesn't face the issues of starvation or convoy effect.
- All the jobs get a fair allocation of CPU.
- It deals with all process without any priority

Disadvantages:

- If slicing time of OS is low, the processor output will be reduced.
- This method spends more time on context switching
- Its performance heavily depends on the time quantum.

5. **Multilevel Queue Scheduling**

- It is used when processes are easily classified into different groups (foreground (interactive) processes and background (batch) processes).
- These two types of processes have different response-time requirements, different priorities and so may have different scheduling needs.
- A multilevel queue scheduling algorithm partitions the ready queue into several separate queues (Figure 6.6).
- The processes are permanently assigned to one queue, generally based on some property of the process, such as memory size, process priority, or process type. Each queue has its own scheduling algorithm.
- For example, separate queues might be used for foreground and background processes. The foreground queue might be scheduled by an RR algorithm, while the background queue is scheduled by an FCFS algorithm.

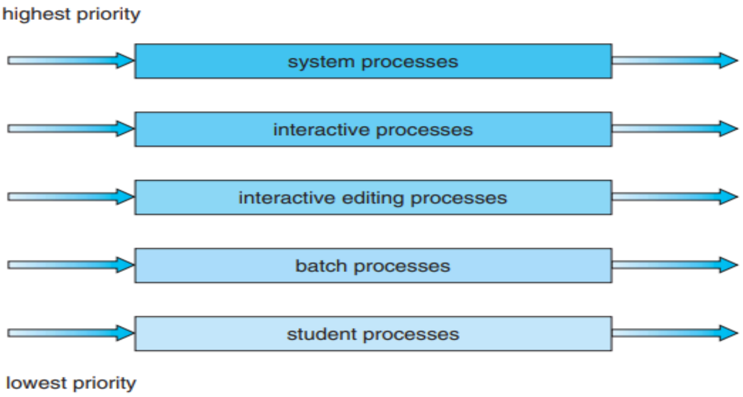


Figure 6.6 Multilevel queue scheduling.

- Each queue has absolute priority over lower-priority queues. No process in the batch queue, for example, could run unless the queues for system processes, interactive processes, and interactive editing processes were all empty.
- If an interactive editing process entered the ready queue while a batch process was running, the batch process would be preempted.
- Another possibility is to time-slice among the queues. Here, each queue gets a certain portion of the CPU time, which it can then schedule among its various processes.
- Eg in foreground–background queue example, the foreground queue can be given 80 percent of the CPU time for RR scheduling among its processes, while the background queue receives 20 percent of the CPU to give to its processes on an FCFS basis.

6. **Multilevel Feedback Queue Scheduling**

- ❖ In multilevel queue scheduling processes are permanently assigned to a queue when they enter the system.
- ❖ If there are separate queues for foreground and background processes, for example, processes do not move from one queue to the other, since processes do not change their foreground or background nature. This setup has the advantage of low scheduling overhead, but it is inflexible.
- ❖ The multilevel feedback queue scheduling algorithm, in contrast, allows a process to move between queues.
- ❖ The idea is to separate processes according to the characteristics of their CPU bursts.
- ❖ If a process uses too much CPU time, it will be moved to a lower-priority queue. This scheme leaves I/O-bound and interactive processes in the higher-priority queues.
- ❖ In addition, a process that waits too long in a lower-priority queue may be moved to a higher-priority queue. This form of aging prevents starvation.

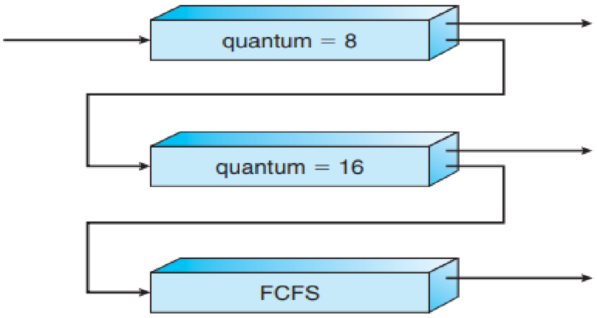


Figure 6.7 Multilevel feedback queues.

Working of multi level feedback scheduling:

- ❖ The scheduler first executes all processes in queue 0. Only when queue 0 is empty will it execute processes in queue 1.
- ❖ Similarly, processes in queue 2 will be executed only if queues 0 and 1 are empty. A process that arrives for queue 1 will preempt a process in queue 2.

- A process in queue 0 is given a time quantum of 8 milliseconds. If it does not finish within this time, it is moved to the tail of queue 1.

It is defined by the following parameters:

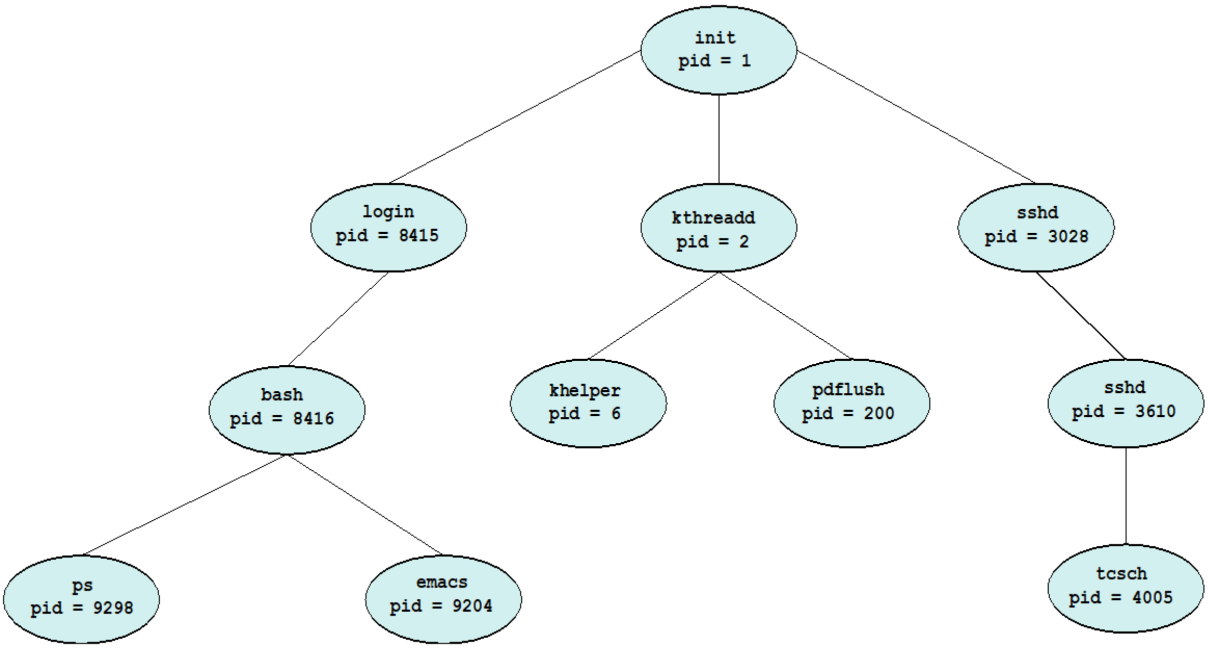
- The number of queues
- The scheduling algorithm for each queue
- The method used to determine when to upgrade a process to a higher priority queue
- The method used to determine when to demote a process to a lower priority queue
- The method used to determine which queue a process will enter when that process needs service

Operations on Processes

- **Parent** process create **children** processes, which, in turn create other processes, forming a **tree** of processes
- Generally, process identified and managed via a **process identifier** (**pid**)
- Resource sharing options
  - Parent and children share all resources
  - Children share subset of parent's resources
  - Parent and child share no resources
- Execution options
  - Parent and children execute concurrently
  - Parent waits until children terminate

Process Creation

Tree of Processes



- The init process (which always has a pid of 1) serves as the root parent process for all user processes.
- Once the system has booted, the init process can also create various user processes, such as a web or print server, an ssh server, and the like.

In Figure 3.8, we see two children of init—kthreadd and sshd.

- The kthreadd process is responsible for creating additional processes that perform tasks on behalf of the kernel (in this situation, khelper and pdflush).
- The sshd process is responsible for managing clients that connect to the system by using ssh (which is short for secure shell).
- The login process is responsible for managing clients that directly log onto the system. In this example, a client has logged on and is using the bash shell,

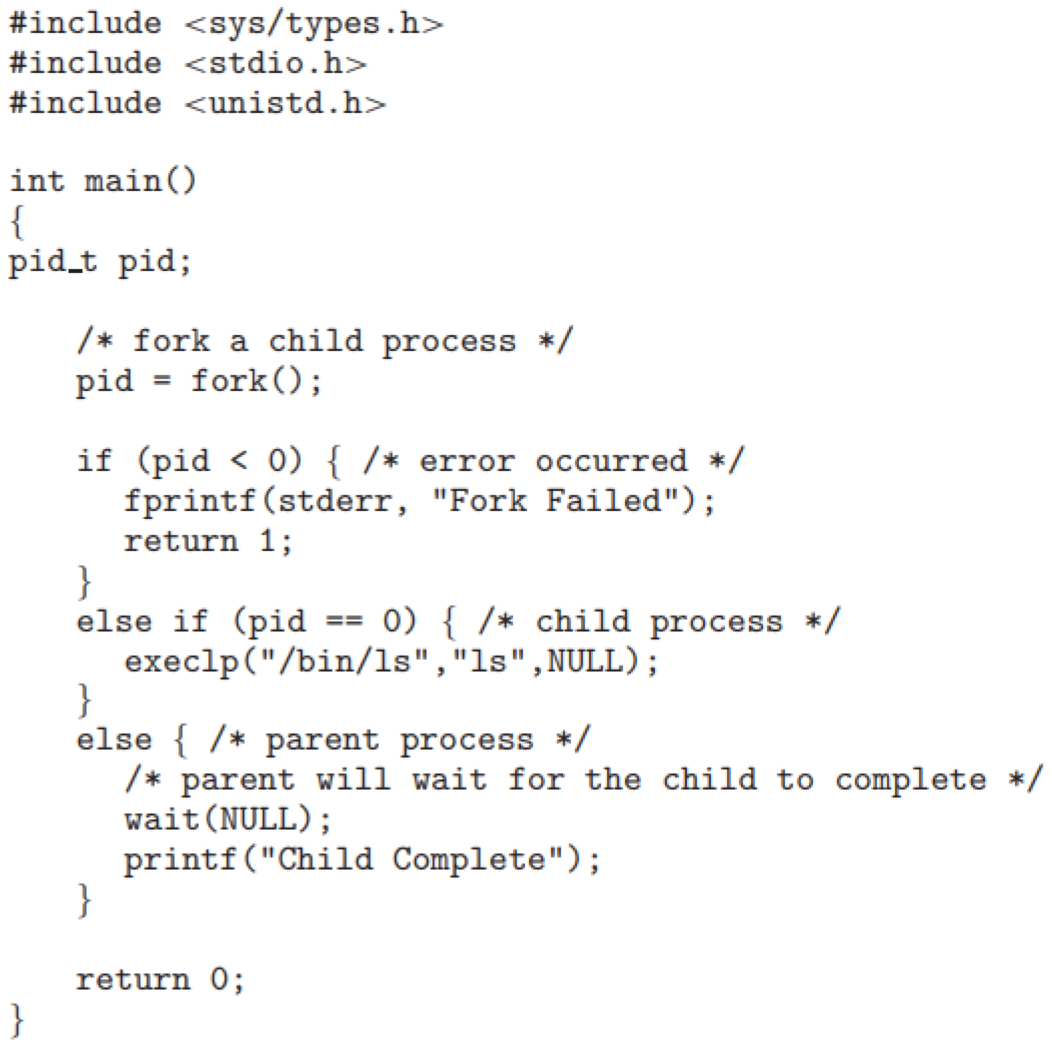
which has been assigned pid 8416.

Two address-space possibilities for the new process:

1. The child process is a duplicate of the parent process (it has the same program and data as the parent).

2. The child process has a new program loaded into it.

After a fork() system call, one of the two processes typically uses the exec() system call to replace the process's memory space with a new program.



```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
pid_t pid;

   /* fork a child process */
   pid = fork();

   if (pid < 0) { /* error occurred */
     fprintf(stderr, "Fork Failed");
     return 1;
   }
   else if (pid == 0) { /* child process */
     execlp("/bin/ls","ls",NULL);
   }
   else { /* parent process */
     /* parent will wait for the child to complete */
     wait(NULL);
     printf("Child Complete");
   }

   return 0;
}
```

Figure 3.9 Creating a separate process using the UNIX fork() system call.

- The parent waits for the child process to complete with the wait() system call.
- When the child process completes (by either implicitly or explicitly invoking exit()), the parent process resumes from the call to wait(), where it completes using the exit() system call. This is also illustrated in Figure 3.10.

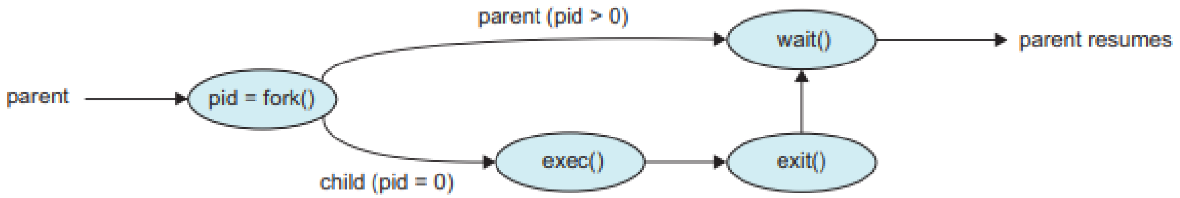


Figure 3.10 Process creation using the `fork()` system call.

Process Termination

- A process terminates when it finishes executing its final statement and asks the operating system to delete it by using the exit() system call.
- At that point, the process may return a status value (typically an integer) to its parent process (via the wait() system call).
- All the resources of the process—including physical and virtual memory, open files, and I/O buffers—are deallocated by the operating system.
- Termination can occur in other circumstances as well.
- A parent may terminate the execution of one of its children for a variety of reasons, such as these:
  - The child has exceeded its usage of some of the resources that it has been allocated. (To determine whether this has occurred, the parent must have a mechanism to inspect the state of its children.)
  - The task assigned to the child is no longer required.
  - The parent is exiting, and the operating system does not allow a child to continue if its parent terminates.
- If a process terminates (either normally or abnormally), then all its children must also be terminated. This phenomenon, referred to as cascading termination.
- A parent process may wait for the termination of a child process by using the wait() system call. The wait() system call is passed a parameter that allows the parent to obtain the exit status of the child.
- A process that has terminated, but whose parent has not yet called wait(), is known as a **zombie process**.
- if a parent did not invoke wait() and instead terminated, thereby leaving its child processes as **orphans.**
- The init process periodically invokes wait(), thereby allowing the exit status of any orphaned process to be collected and releasing the orphan's process identifier and process-table entry

Inter Process Communication

- Processes executing concurrently in the operating system may be either independent processes or cooperating processes.
- A process is **independent** if it cannot affect or be affected by the other processes executing in the system. Any process that does not share data with any other process is independent.
- A process is **cooperating** if it can affect or be affected by the other processes executing in the system.

Need of cooperating processes

1. Information sharing.
   a. Since several users may be interested in the same piece of information (for instance, a shared file), we must provide an environment to allow concurrent access to such information.
2. Computation speedup.
   a. If we want a particular task to run faster, we must break it into subtasks, each of which will be executing in parallel with the others. Notice that such a speedup can be achieved only if the computer has multiple processing cores.
3. Modularity.
   a. We may want to construct the system in a modular fashion, dividing the system functions into separate processes or threads.
4. Convenience.
   a. Even an individual user may work on many tasks at the same time. For instance, a user may be editing, listening to music, and compiling in parallel.

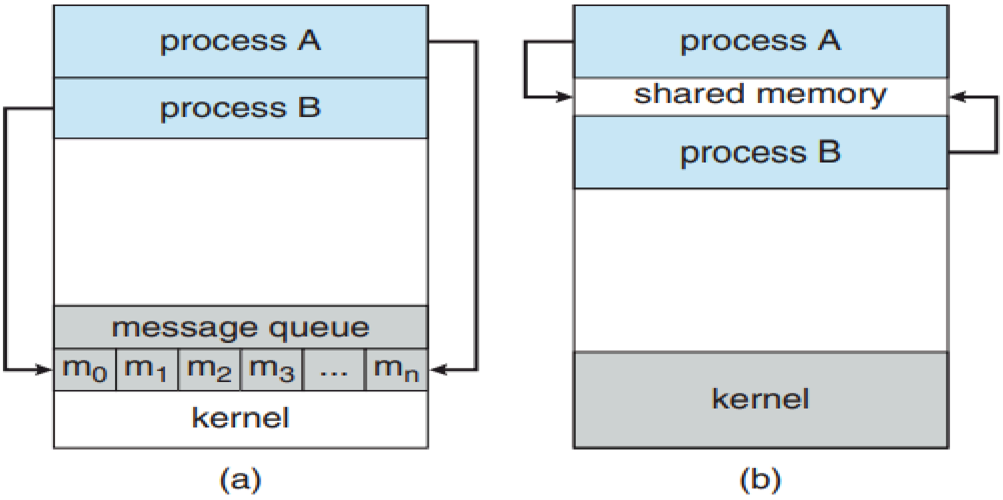


Figure 3.12 Communications models. (a) Message passing. (b) Shared memory.

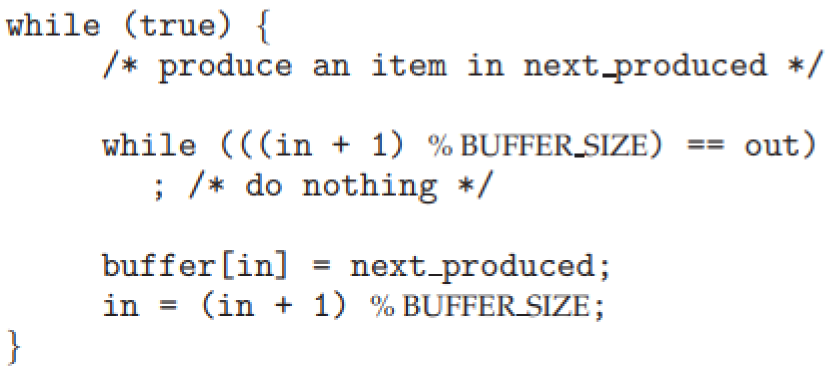
- In the shared-memory model, a region of memory that is shared by cooperating processes is established. Processes can then exchange information by reading and writing data to the shared region.
- In the message-passing model, communication takes place by means of messages exchanged between the cooperating processes.

Shared-Memory Systems

- A shared-memory region resides in the address space of the process creating the shared-memory segment. Other processes that wish to communicate using this shared-memory segment must attach it to their address space.
- They can then exchange information by reading and writing data in the shared areas.
- The form of the data and the location are determined by these processes and are not under the operating system's control. The processes are also responsible for ensuring that they are not writing to the same location simultaneously

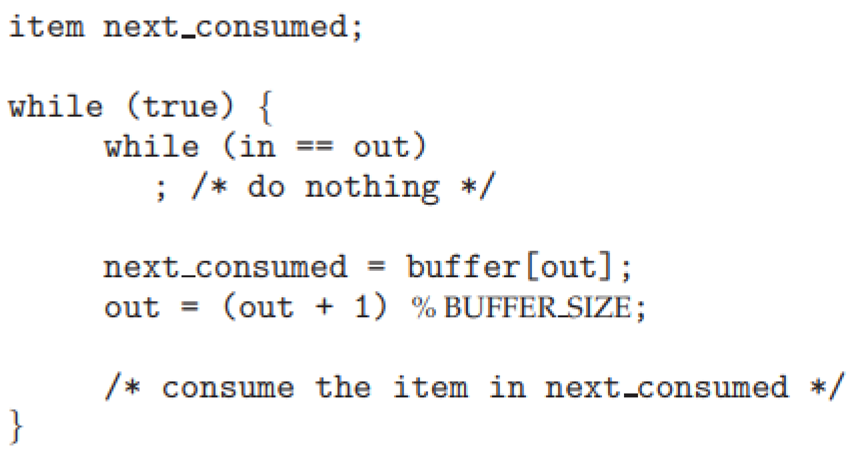
Eg of Shared Memory in Producer Consumer problem

A producer process produces information that is consumed by a consumer process.



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

     while (((in + 1) % BUFFER_SIZE) == out)
        ; /* do nothing */

     buffer[in] = next_produced;
     in = (in + 1) % BUFFER_SIZE;
}
```

Figure 3.13 The producer process using shared memory.



```
item next_consumed;

while (true) {
     while (in == out)
        ; /* do nothing */

     next_consumed = buffer[out];
     out = (out + 1) % BUFFER_SIZE;

     /* consume the item in next_consumed */
}
```

Figure 3.14 The consumer process using shared memory.

- To allow producer and consumer processes to run concurrently, we must have available a buffer of items that can be filled by the producer and emptied by the consumer.
- This buffer will reside in a region of memory that is shared by the producer and consumer processes.
- A producer can produce one item while the consumer is consuming another item.
- The producer and consumer must be synchronized, so that the consumer does not try to consume an item that has not yet been produced.

Types of buffers

1. The **unbounded buffer** places no practical limit on the size of the buffer. The consumer may have to wait for new items, but the producer can always produce new items.
2. The **bounded buffer** assumes a fixed buffer size. In this case, the consumer must wait if the buffer is empty, and the producer must wait if the buffer is full.

Message Passing

Message passing provides a mechanism to allow processes to communicate and to synchronize their actions without sharing the same address space.

- Mechanism for processes to communicate and to synchronize their actions
- Message system – processes communicate with each other without resorting to shared variables
- IPC facility provides two operations:
  - **send**(*message*)
  - **receive**(*message*)
- The *message* size is either fixed or variable
- If processes *P* and *Q* wish to communicate, they need to:
  - Establish a ***communication link*** between them
  - Exchange messages via send/receive
- Implementation issues:
  - How are links established?
  - Can a link be associated with more than two processes?
  - How many links can there be between every pair of communicating processes?
  - What is the capacity of a link?
  - Is the size of a message that the link can accommodate fixed or variable?
  - Is a link unidirectional or bi-directional?
- Logical:
  - 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
- **Mailbox sharing**
  - P_1, P_2, and P_3 share mailbox A
  - P_1, sends; P_2 and P_3 receive
  - Who gets the message?
- **Solutions**
  - Allow a link to be associated with at most two processes
  - Allow only one process at a time to execute a receive operation
  - Allow the system to select arbitrarily the receiver. Sender is notified who the receiver was.

Synchronization

- Message passing may be either blocking or non-blocking
- **Blocking** is considered **synchronous**
  - **Blocking send --** the sender is blocked until the message is received
  - **Blocking receive --** the receiver is blocked until a message is available
- **Non-blocking** is considered **asynchronous**
  - **Non-blocking send --** the sender sends the message and continue
  - **Non-blocking receive** the receiver receives either valid message or null.

Buffering

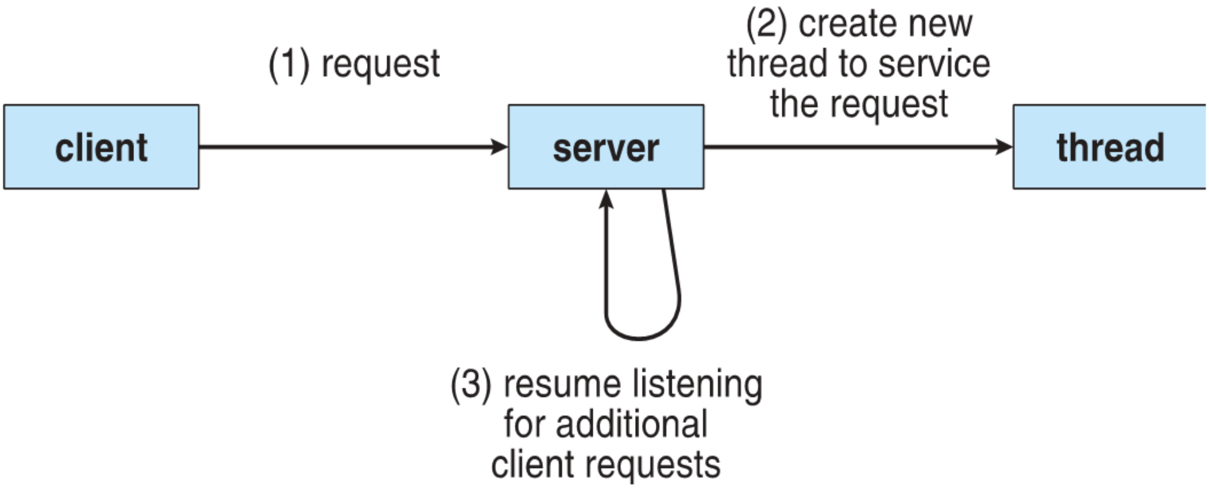
- Queue of messages attached to the link.
- implemented in one of three ways

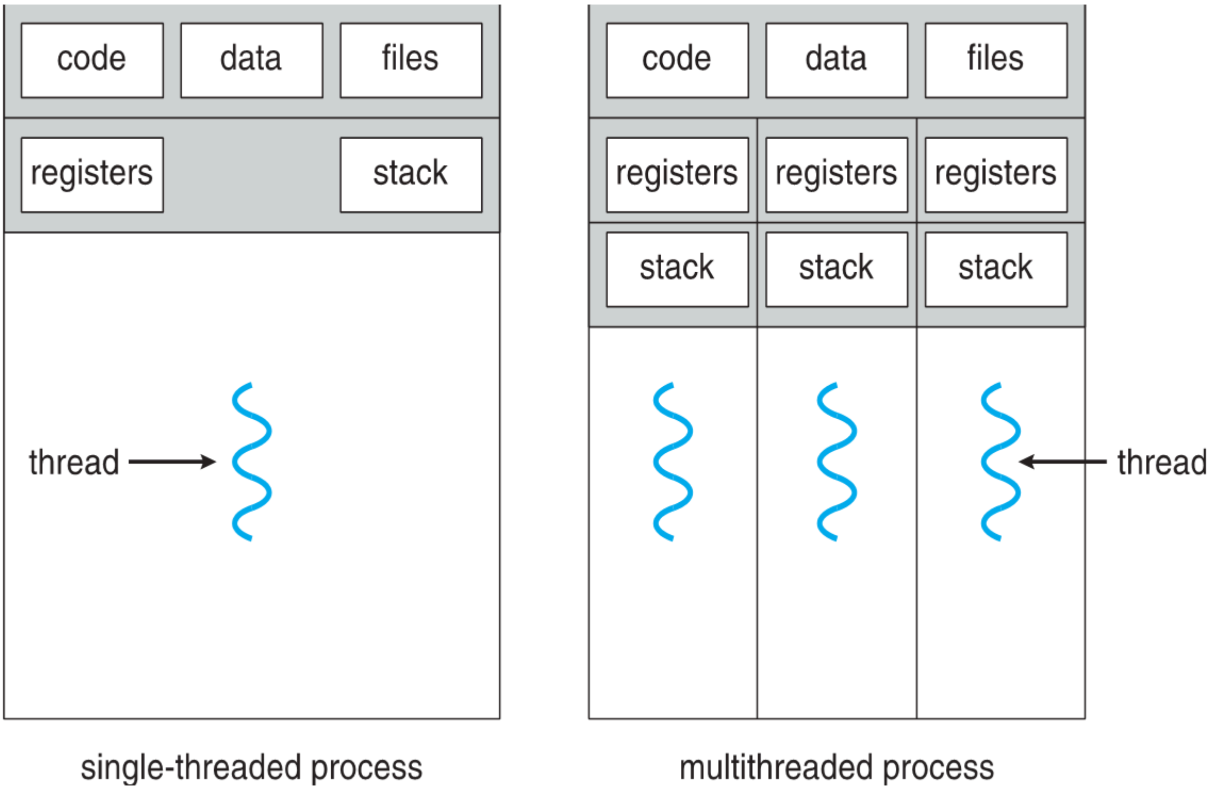
1. Zero capacity – no messages are queued on a link.
   Sender must wait for receiver (rendezvous)
2. Bounded capacity – finite length of *n* messages
   Sender must wait if link full
3. Unbounded capacity – infinite length & Sender never waits

Multi threading

A thread is a basic unit of CPU utilization; it comprises a thread ID, a program counter, a register set, and a stack. It shares with other threads belonging to the same process its code section, data section, and other operating-system resources, such as open files and signals. A traditional (or heavyweight) process has a single thread of control. If a process has multiple threads of control, it can perform more than one task at a time.

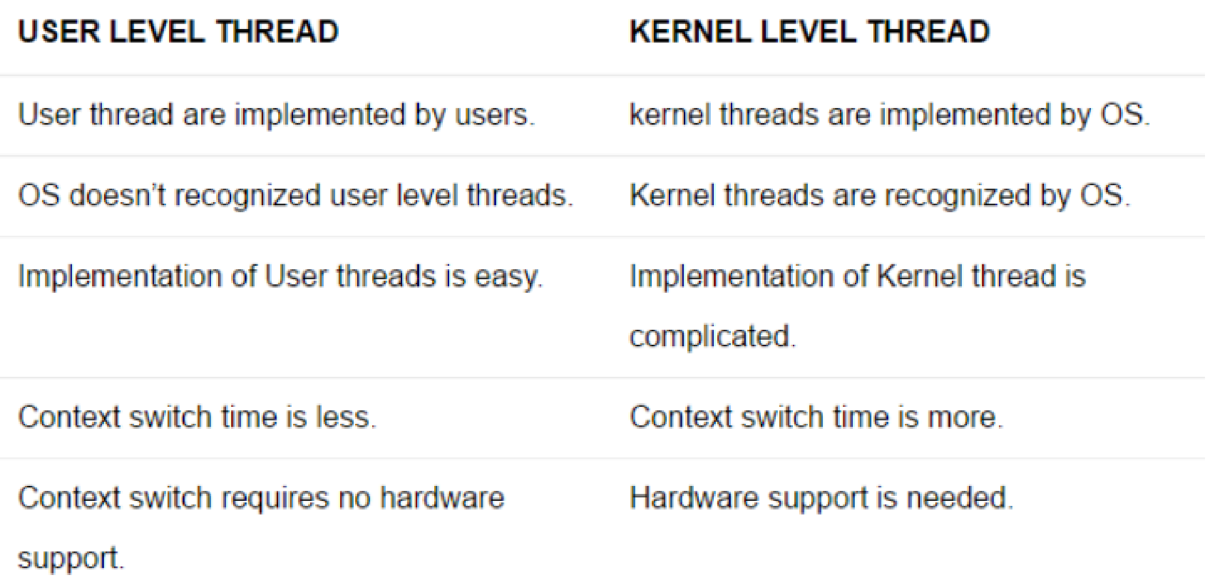
Multi threaded server architecture


(1) request
(2) create new thread to service the request
client
server
thread
(3) resume listening for additional client requests



Benefits of Multi threading

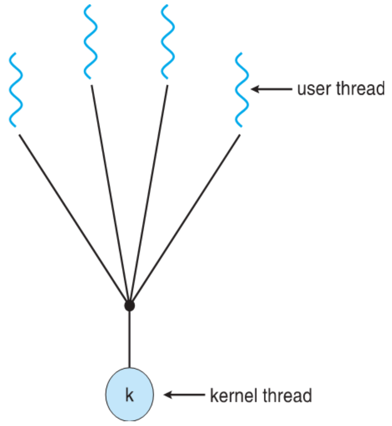
1. **Responsiveness.** Multithreading an interactive application may allow a program to continue running even if part of it is blocked or is performing a lengthy operation, thereby increasing responsiveness to the user.
2. **Resource sharing.** Threads access shared resources more easily than processes as it can only share resources through techniques such as shared memory and message passing.
3. **Economy.** Allocating memory and resources for process creation is costly. Because threads share the resources of the process to which they belong, it is more economical to create and context switch threads.
4. **Scalability.** The benefits of multithreading can be even greater in a multiprocessor architecture, where threads may be running in parallel on different processing cores.



USER LEVEL THREAD	KERNEL LEVEL THREAD
User thread are implemented by users.	kernel threads are implemented by OS.
OS doesn’t recognized user level threads.	Kernel threads are recognized by OS.
Implementation of User threads is easy.	Implementation of Kernel thread is complicated.
Context switch time is less.	Context switch time is more.
Context switch requires no hardware support.	Hardware support is needed.

Multithreading Models

1. Many to one



Many user-level threads mapped to single kernel thread

One thread blocking causes all to block

Multiple threads may not run in parallel on muticore system because only one may be in kernel at a time

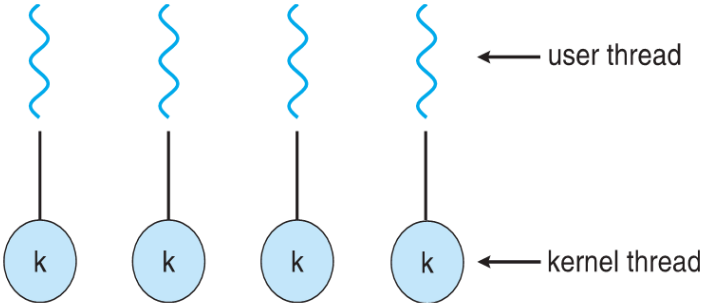
Few systems currently use this model

Examples:

¡Solaris Green Threads

¡GNU Portable Threads

2. One to one



Each user-level thread maps to kernel thread

Creating a user-level thread creates a kernel thread

More concurrency than many-to-one

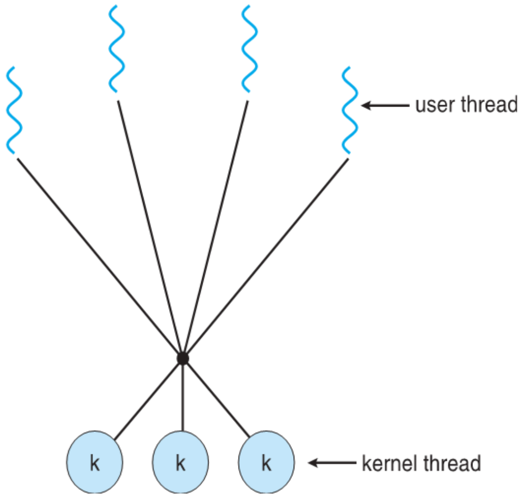
Number of threads per process sometimes restricted due to overhead

Examples

¡Windows

¡Linux

3. Many to Many



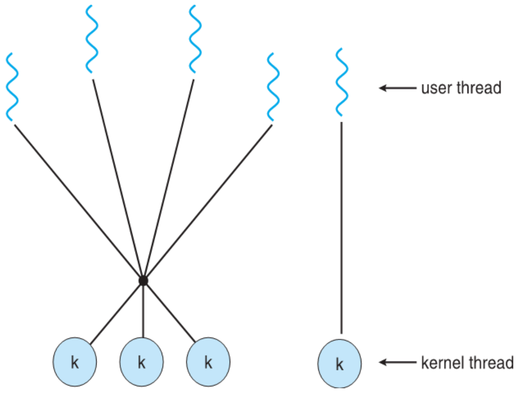
Allows many user level threads to be mapped to many kernel threads

Allows the operating system to create a sufficient number of kernel threads

Solaris prior to version 9

Windows with the *ThreadFiber* package

4. Two level model



Similar to M:M, except that it allows a user thread to be **bound** to kernel thread

Examples

¡IRIX

Threading Issues

1. The fork() and exec() system calls

- The fork() is used to create a duplicate process. The meaning of the fork() and exec() system calls change in a multithreaded program.
- If one thread in a program which calls fork(), does the new process duplicate all threads, or is the new process single-threaded? If we take, some UNIX systems have chosen to have two versions of fork(), one that duplicates all threads and another that duplicates only the thread that invoked the fork() system call.
- If a thread calls the exec() system call, the program specified in the parameter to exec() will replace the entire process which includes all threads.

2. Signal Handling

- Generally, signal is used in UNIX systems to notify a process that a particular event has occurred. A signal received either synchronously or asynchronously, based on the source of and the reason for the event being signalled.
- All signals, whether synchronous or asynchronous, follow the same

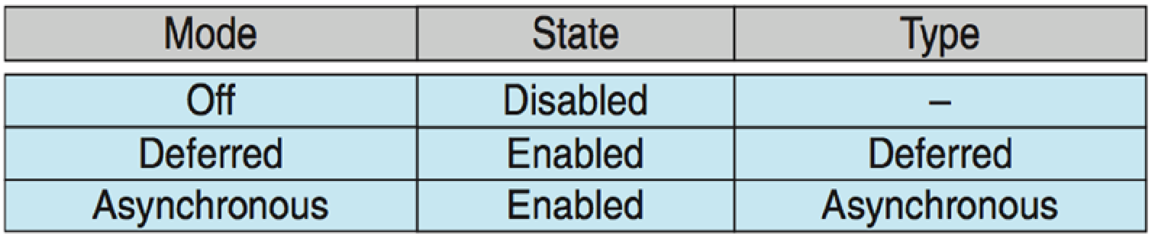
pattern as given below –

  - A signal is generated by the occurrence of a particular event.
  - The signal is delivered to a process.
  - Once delivered, the signal must be handled.

- Where should a signal be delivered for multi-threaded?
  - 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

3. Cancellation

- Thread cancellation is the task of terminating a thread before it has completed.
- For example – If multiple database threads are concurrently searching through a database and one thread returns the result the remaining threads might be cancelled.
- A target thread is a thread that is to be cancelled, cancellation of target thread may occur in two different scenarios –
  - **Asynchronous cancellation** – One thread immediately terminates the target thread.
  - **Deferred cancellation** – The target thread periodically checks whether it should terminate, allowing it an opportunity to terminate itself in an ordinary fashion.
- Invoking thread cancellation requests cancellation, but actual cancellation depends on thread state . (default- deferred)



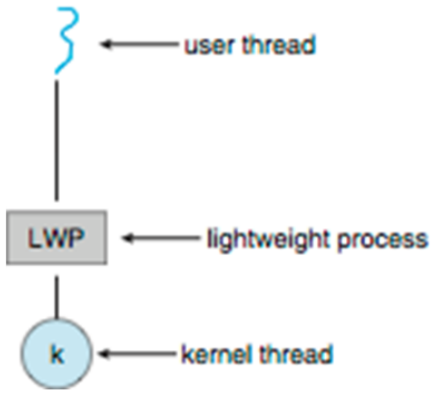
Mode	State	Type
Off	Disabled	–
Deferred	Enabled	Deferred
Asynchronous	Enabled	Asynchronous

- The issue related to the target threads are listed below:
  - What if the resources had been allotted to the cancel target thread?
  - What if the target thread is terminated when it was updating the data, it was sharing with some other thread.

4. **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

5.**Scheduler Activation**



- Both M:M and Two-level models require communication to maintain the appropriate number of kernel threads allocated to the application
- Typically use an intermediate data structure between user and kernel threads – lightweight process (LWP)
  - Appears to be a virtual processor on which process can schedule user thread to run
  - Each LWP attached to kernel thread
  - How many LWPs to create?
- Scheduler activations provide upcalls - a communication mechanism from the kernel to the upcall handler in the thread library
- This communication allows an application to maintain the correct number kernel threads

Example of Scheduler Activation

- One event that triggers an upcall occurs when an application thread is about to block.
- In this scenario, the kernel makes an upcall to the application informing it that a thread is about to block and identifying the specific thread.
- Then the kernel allocates a new virtual processor to the application. The application runs an upcall handler on this new virtual processor, that saves the state of the blocking thread and relinquishes the virtual processor on which the blocking thread is running.
- Another thread that is eligible to run on the new virtual processor is scheduled then by the upcall handler.
- Whenever the event that the blocking thread was waiting for occurs, the kernel makes another upcall to the thread library informing it that the previously blocked thread is now eligible to run.
- A virtual processor is also required for The upcall handler for this event, and the kernel may allocate a new virtual processor or preempt one of the user threads and then run the upcall handler on its virtual processor.
- The application schedules an eligible thread to run on an available virtual processor, after marking the unblocked thread as eligible to run.

PROCESS SYNCHRONIZATION

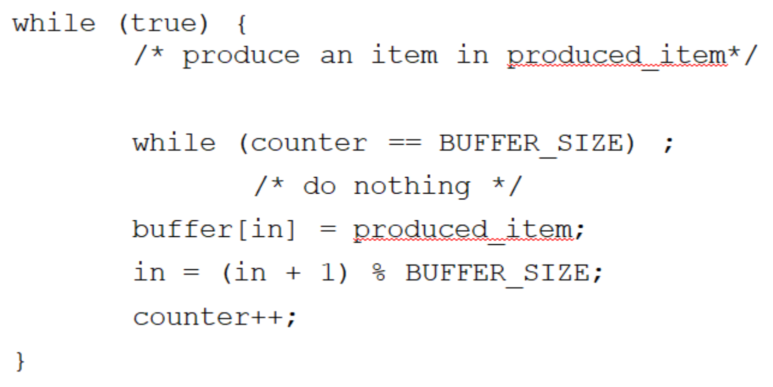
- Concurrent access to shared data may result in data inconsistency. Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes.
- Shared-memory solution to bounded-butter problem allows at most n – 1 items in buffer at the same time.
- A solution, where all N buffers are used is not simple.
- Suppose that we modify the producer-consumer code by adding a variable counter, initialized to 0 and increment it each time a new item is added to the buffer
- **Race condition:** The situation where several processes access – and manipulate shared data concurrently. The final value of the shared data depends upon which process finishes last.
- To prevent race conditions, concurrent processes must be synchronized.

Producer Consumer Problem

Problem: Given the common fixed-size buffer, the task is to make sure that the producer can't add data into the buffer when it is full and the consumer can't remove data from an empty buffer.

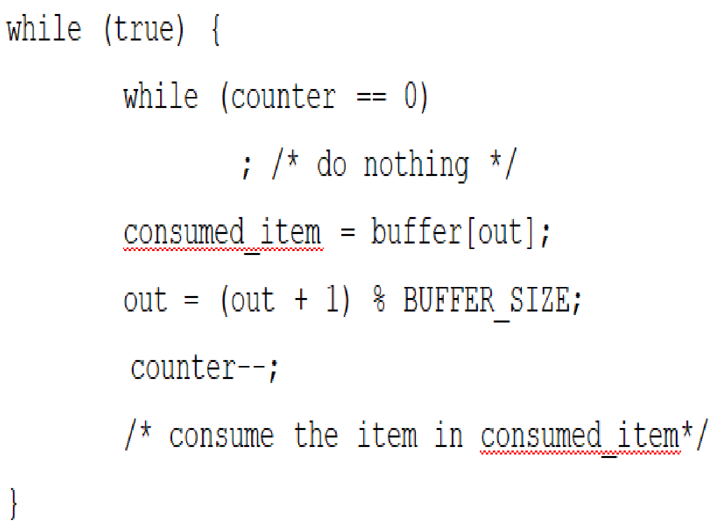
Solution: The producer is to either go to sleep or discard data if the buffer is full. The next time the consumer removes an item from the buffer, it notifies the producer, who starts to fill the buffer again. In the same manner, the consumer can go to sleep if it finds the buffer to be empty. The next time the producer puts data into the buffer, it wakes up the sleeping consumer.

Producer code

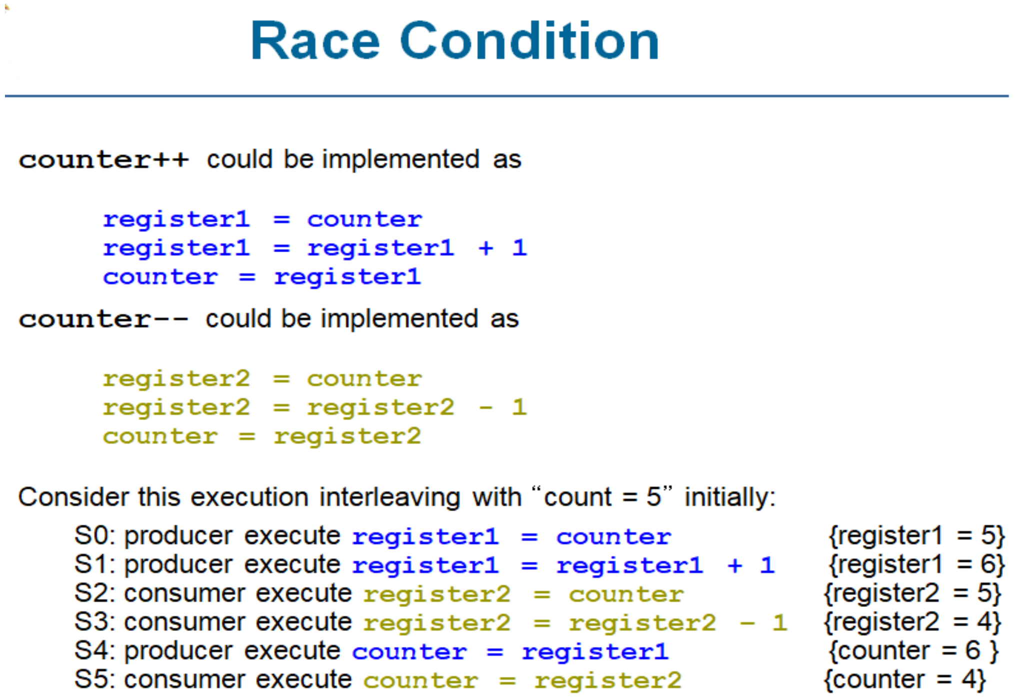


```
while (true) {
       /* produce an item in produced_item*/

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

Consumer code



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



Race Condition

counter++ could be implemented as

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

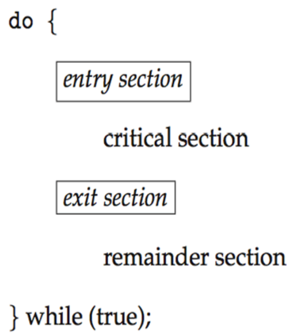
counter-- could be implemented as

```
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}

The Critical-Section Problem:



```
do {
     entry section
          critical section
     exit section
          remainder section
} while (true);
```

- There are n processes that are competing to use some shared data
- Each process has a code segment, called critical section, in which the shared data is accessed.
- Each process must ask permission to enter critical section in **entry section**, may follow critical section with **exit section**, then **remainder section**

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

Requirements to be satisfied for a Solution to the Critical-Section Problem:

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

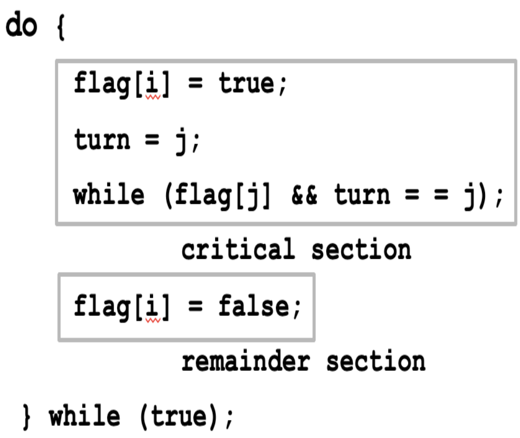
2. **Progress** - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely.

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.

Peterson’s Solution

- 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!

Code:



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

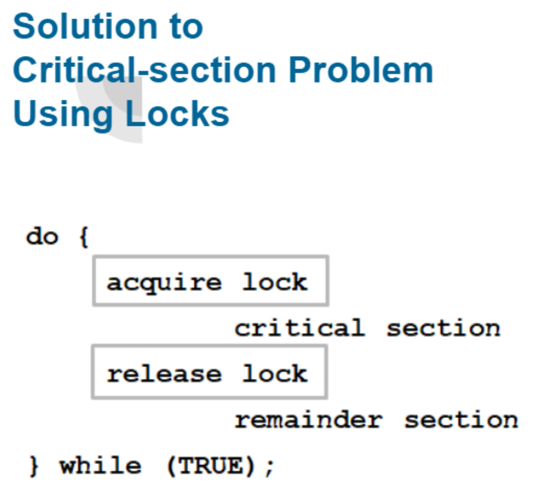
Steps:

- In the solution i represents Process P1 and j represents process P2. Initially the flags are false.
- When a process wants to execute it’s critical section, it sets it’s flag to true and turn as the index of the other process.
- This means that the process wants to execute but it will allow the other process to run first. The process performs busy waiting until the other process has finished it’s own critical section.
- After this the current process enters it’s critical section and accesses the shared variables.
- After completing the critical section, it sets it’s own flag to false, indication it does not wish to execute anymore.

1. Mutual exclusion is preserved
    i. **P_i** enters CS only if: either **flag[j] = false** or **turn = i**
2. Progress requirement is satisfied
    i. Progress is satisfied as any waiting process can enter their critical section as soon as existing process in critical section sets its own flag to false.
3. Bounded-waiting requirement is met

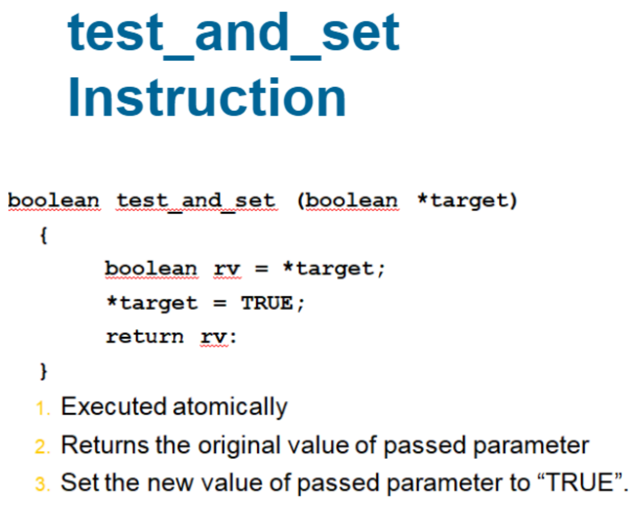
Synchronization Hardware

- Many systems provide hardware support for implementing the critical section code.
- All solutions below based on idea of **locking**
    - Protecting critical regions via locks
- Uniprocessors – could disable interrupts
    - Currently running code would execute without preemption
    - Generally too inefficient on multiprocessor systems
    - Operating systems using this not broadly scalable
- Modern machines provide special atomic hardware instructions
    - **Atomic** = non-interruptible instructions
    - Either test memory word and set value
    - Or swap contents of two memory words



Solution to Critical-section Problem Using Locks

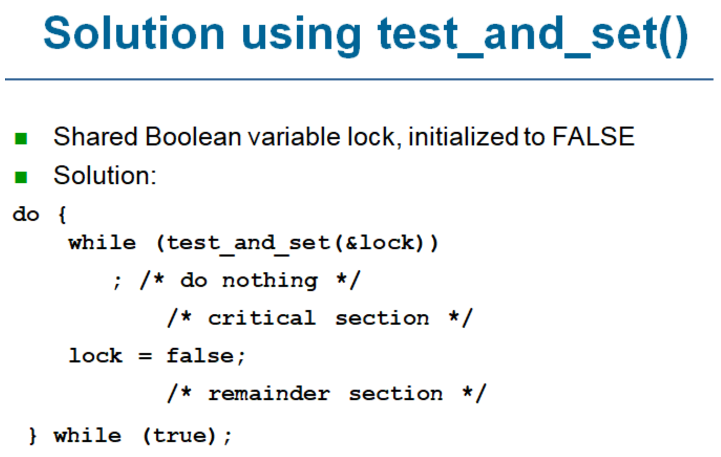
```
do {
      acquire lock
                 critical  section
      release lock
                 remainder  section
} while  (TRUE);
```



test_and_set Instruction

```
boolean  test_and_set  (boolean  *target)
  {
       boolean  rv  =  *target;
       *target  =  TRUE;
       return  rv:
  }
```

1. Executed atomically
2. Returns the original value of passed parameter
3. Set the new value of passed parameter to “TRUE”.



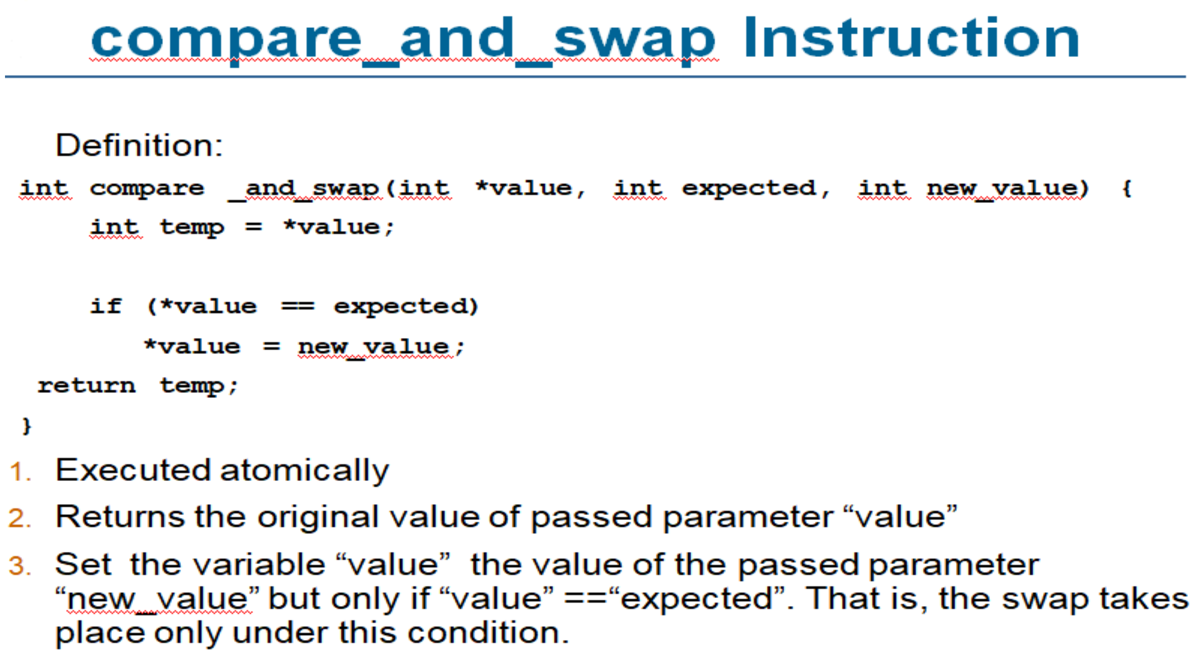
Solution using test_and_set()

- Shared Boolean variable lock, initialized to FALSE
- Solution:

```
do {
    while  (test_and_set(&lock))
       ; /* do nothing */
           /* critical  section */
    lock  = false;
           /* remainder  section */
 } while  (true);
```

Working of Test and Set:

1. Initially the lock value = False.
2. Any process that needs to enter its critical section calls test_and_set(lock).
3. The test_and_set sends back the original lock value to the process and modified the lock=True.
4. The process keeps on waiting till it gets False from the test_and_set method.
5. The process enters its critical section when it gets True from the method.
6. After completing its execution in critical section, the lock is again set to false.



compare_and_swap Instruction

Definition:

```
int compare _and_swap(int *value, int expected, int new_value) {
    int temp = *value;

    if (*value == expected)
       *value = new_value;
 return temp;
}
```

1. Executed atomically
2. Returns the original value of passed parameter “value”
3. Set the variable “value” the value of the passed parameter “new_value” but only if “value” ==“expected”. That is, the swap takes place only under this condition.

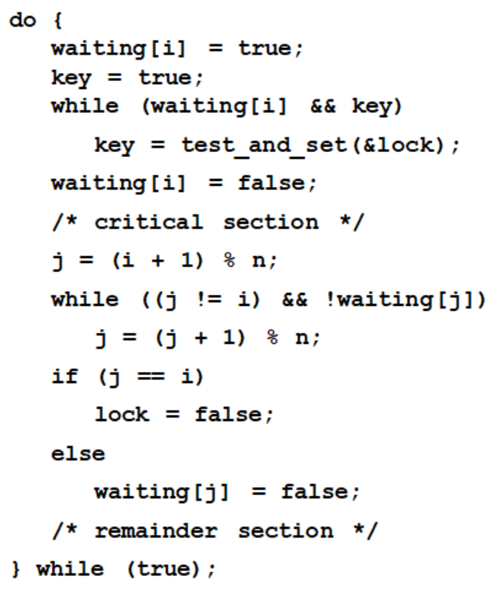
Working of Compare_and_Swap:

1. Initially the lock value = 0.
2. Any process that needs to enter its critical section calls compare_and_swap(lock,0,1).
3. The compare_and_swap sends back the original lock value to the process and modifies the lock=1 if lock is the same as expected(lock =0 = expected).
4. The process keeps on waiting till it gets 0 from the compare_and_swap method.
5. The process enters its critical section when it gets 1 from the method.
6. After completing its execution in critical section, the lock is again set to 0.

Drawback:

Only mutual exclusion is satisfied.

Solution that satisfies bounded waiting and progress also:



```
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);
```

1. **Mutual exclusion requirement:**

- Process Pi can enter its critical section only if either waiting[i] == false or key == false. The value of key can become false only if the test and set() is executed. The first process to execute the test and set() will find key == false; all others must wait. The variable waiting[i] can become false only if another process leaves its critical section; only one waiting[i] is set to false, maintaining the mutual-exclusion requirement.

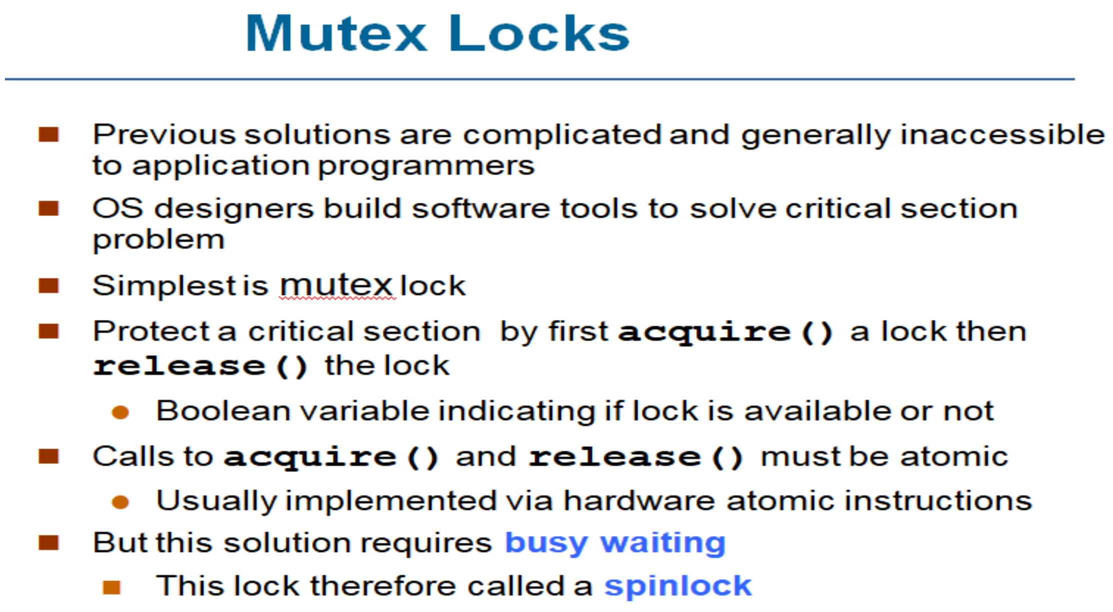
2. **Progress requirement :**

- the arguments presented for mutual exclusion also apply here, since a process exiting the critical section either sets lock to false or sets waiting[j] to false. Both allow a process that is waiting to enter its critical section to proceed.

3. **Bounded-waiting requirement i:**

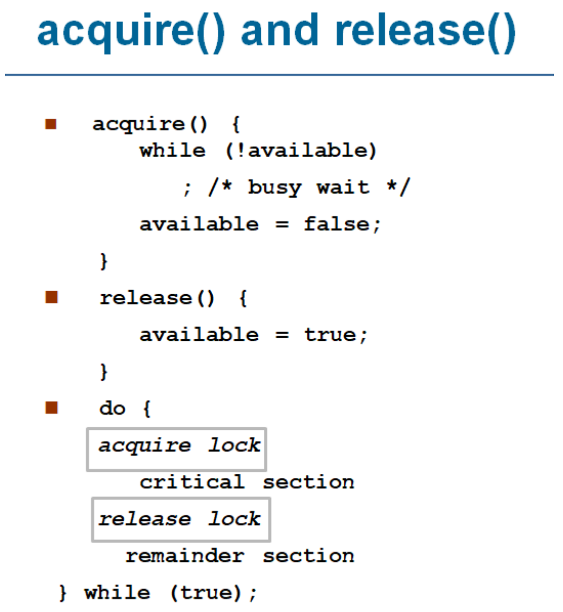
- when a process leaves its critical section, it scans the array waiting in the cyclic ordering (i + 1, i + 2, ..., n − 1, 0, ..., i − 1). It designates the first process in this ordering that is in the entry section (waiting[j] == true) as the next one to enter the critical section. Any process waiting to enter its critical section will thus do so within n − 1 turns.

Mutex Locks



Mutex Locks

- Previous solutions are complicated and generally inaccessible to application programmers
- OS designers build software tools to solve critical section problem
- Simplest is mutex lock
- Protect a critical section by first **`acquire()`** a lock then **`release()`** the lock
  - Boolean variable indicating if lock is available or not
- Calls to **`acquire()`** and **`release()`** must be atomic
  - Usually implemented via hardware atomic instructions
- But this solution requires **busy waiting**
  - This lock therefore called a **spinlock**



acquire() and release()

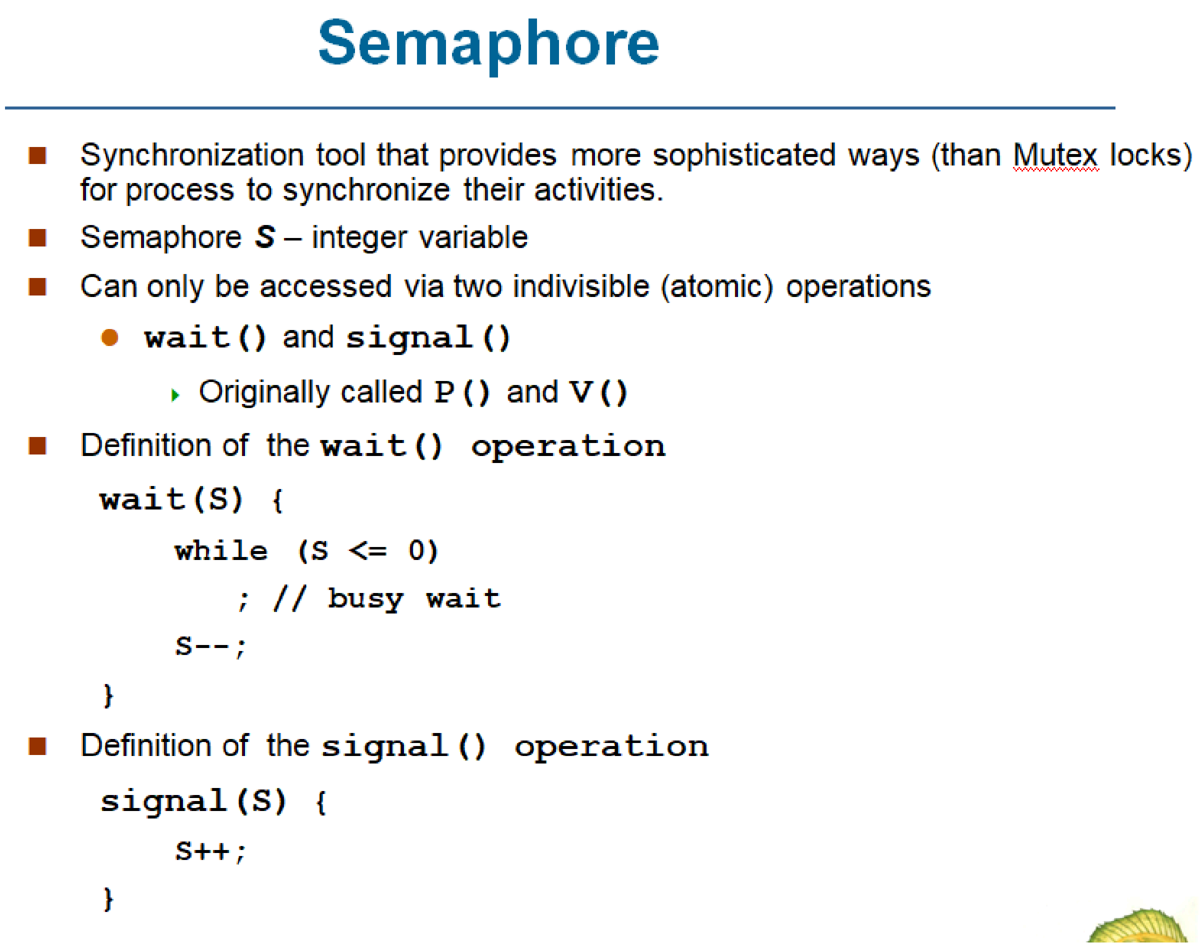
```
■  acquire() {
      while (!available)
         ; /* busy wait */
      available = false;
   }
■  release() {
      available = true;
   }
■  do {
   acquire lock
      critical section
   release lock
     remainder section
 } while (true);
```

- A mutex lock has a boolean variable available whose value indicates if the lock is available or not.
- If the lock is available, a call to acquire() succeeds, and the lock is then considered unavailable.
- A process that attempts to acquire an unavailable lock is blocked until the lock is released.

Busy waiting. While a process is in its critical section, any other process that tries to enter its critical section must loop continuously in the call to acquire(). In fact, this type of mutex lock is also called a **spinlock** because the process “spins” while waiting for the lock to become available.

Spinlocks do have an advantage, however, in that no context switch is required when a process must wait on a lock, and a context switch may take considerable time. Thus, when locks are expected to be held for short times, spinlocks are useful.

Semaphores



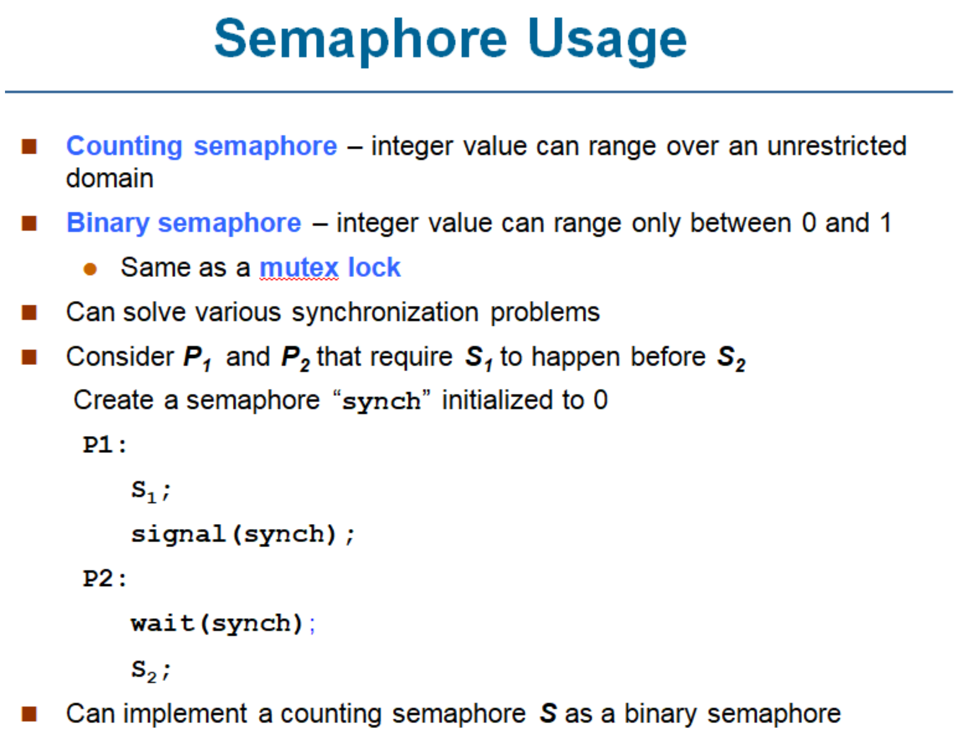
Semaphore

- Synchronization tool that provides more sophisticated ways (than Mutex locks) for process to synchronize their activities.
- 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++;
}
```



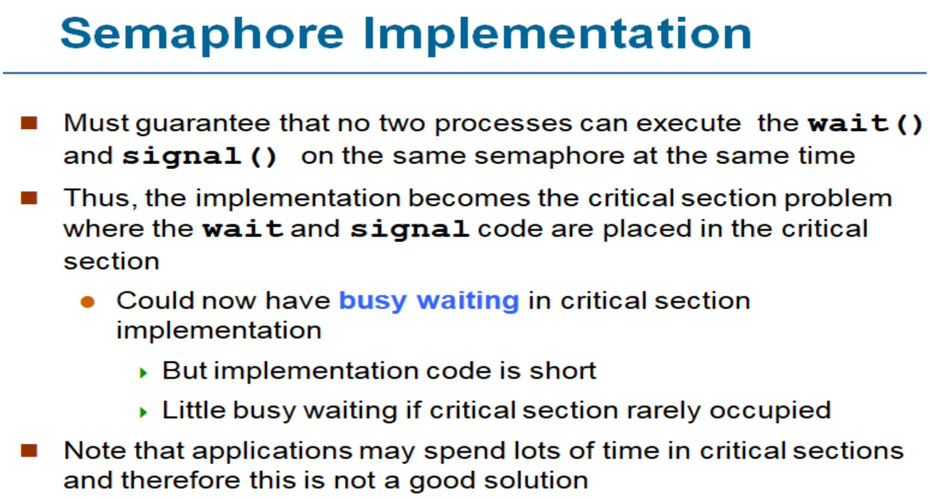
Semaphore Usage

- **Counting semaphore** – integer value can range over an unrestricted domain
- **Binary semaphore** – integer value can range only between 0 and 1
  - Same as a **mutex lock**
- Can solve various synchronization problems
- Consider ***P₁*** and ***P₂*** that require ***S₁*** to happen before ***S₂***

  Create a semaphore "`synch`" initialized to 0

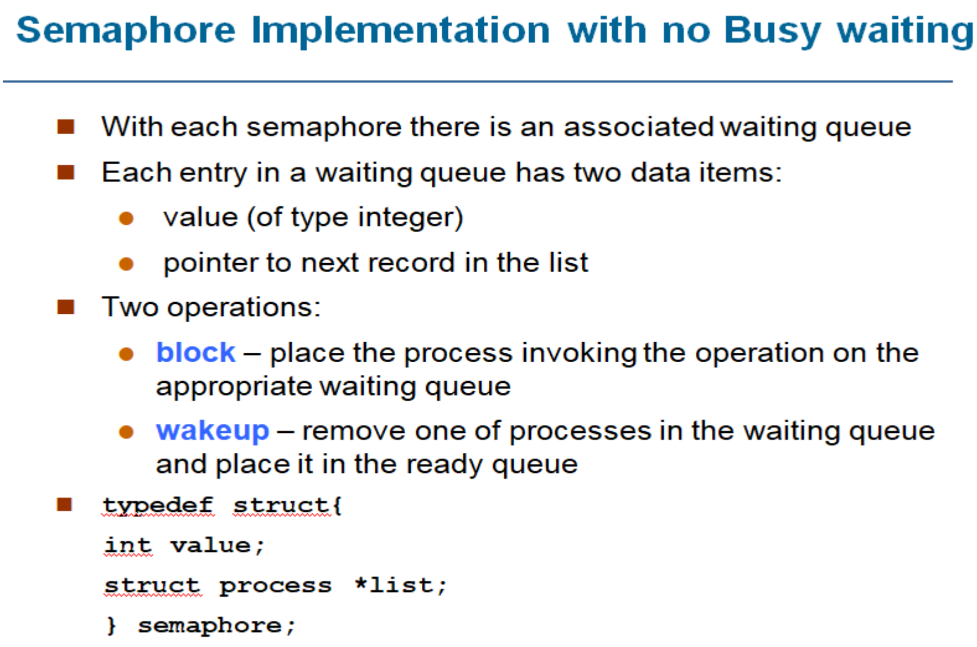
```
P1:
    S1;
    signal(synch);
P2:
    wait(synch);
    S2;
```

- Can implement a counting semaphore ***S*** as a binary semaphore



Semaphore Implementation

- Must guarantee that no two processes can execute the **`wait()`** and **`signal()`** on the same semaphore at the same time
- Thus, the implementation becomes the critical section problem where the **`wait`** and **`signal`** code are placed in the critical section
  - Could now have **busy waiting** in critical section implementation
    - But implementation code is short
    - Little busy waiting if critical section rarely occupied
- Note that applications may spend lots of time in critical sections and therefore this is not a good solution



Semaphore Implementation with no Busy waiting

- With each semaphore there is an associated waiting queue
- Each entry in a waiting queue has two data items:
  - value (of type integer)
  - pointer to next record in the list
- Two operations:
  - **block** – place the process invoking the operation on the appropriate waiting queue
  - **wakeup** – remove one of processes in the waiting queue and place it in the ready queue

-
  ```
  typedef struct{
  int value;
  struct process *list;
  } semaphore;
  ```

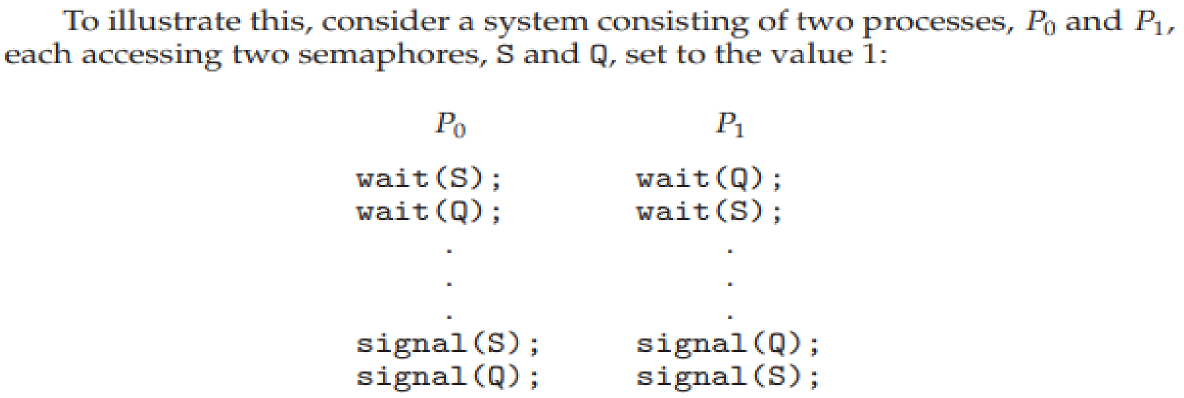
Code for wait() and signal():

```
wait(semaphore *S) {
    S->value--;
    if (S->value < 0) {
        add this process to S->list;
        block();
    }
}

signal(semaphore *S) {
    S->value++;
    if (S->value <= 0) {
        remove a process P from S->list;
        wakeup(P);
    }
}
```

- In this implementation, semaphore values may be negative, whereas semaphore values are never negative under the classical definition of semaphores with busy waiting.
- If a semaphore value is negative, its magnitude is the number of processes waiting on that semaphore. This fact results from switching the order of the decrement and the test in the implementation of the wait() operation.
- The list of waiting processes can be easily implemented by a link field in each process control block (PCB). Each semaphore contains an integer value and a pointer to a list of PCBs.
- One way to add and remove processes from the list so as to ensure bounded waiting is to use a FIFO queue, where the semaphore contains both head and tail pointers to the queue.

Deadlock and Starvation in Semaphores:

- The implementation of a semaphore with a waiting queue may result in a situation where two or more processes are waiting indefinitely for an event that can be caused only by one of the waiting processes. The event in question is the execution of a signal() operation. When such a state is reached, these processes are said to be **deadlocked.**



To illustrate this, consider a system consisting of two processes, P_0 and P_1, each accessing two semaphores, S and Q, set to the value 1:

```
      P0                P1

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

- Suppose that P0 executes wait(S) and then P1 executes wait(Q). When P0 executes wait(Q), it must wait until P1 executes signal(Q). Similarly, when P1 executes wait(S), it must wait until P0 executes signal(S). Since these signal() operations cannot be executed, P0 and P1 are deadlocked.
- **Indefinite blocking or starvation**, a situation in which processes wait indefinitely within the semaphore. Indefinite blocking may occur if we remove processes from the list associated with a semaphore in LIFO (last-in, first-out) order.

Priority Inversion

- A scheduling challenge arises when a higher-priority process needs to read or modify kernel data that are currently being accessed by a lower-priority process—or a chain of lower-priority processes.
- Since kernel data are typically protected with a lock, the higher-priority process will have to wait for a lower-priority one to finish with the resource. The situation becomes more complicated if the lower-priority process is preempted in favor of another process with a higher priority.

- As an example, assume we have three processes—L, M, and H—whose priorities follow the order L < M < H. Assume that process H requires resource R, which is currently being accessed by process L. Ordinarily, process H would wait for L to finish using resource R. However, now suppose that process M becomes runnable, thereby preempting process L. Indirectly, a process with a lower priority—process M—has affected how long process H must wait for L to release resource R. This problem is known as priority inversion.

Solutions:

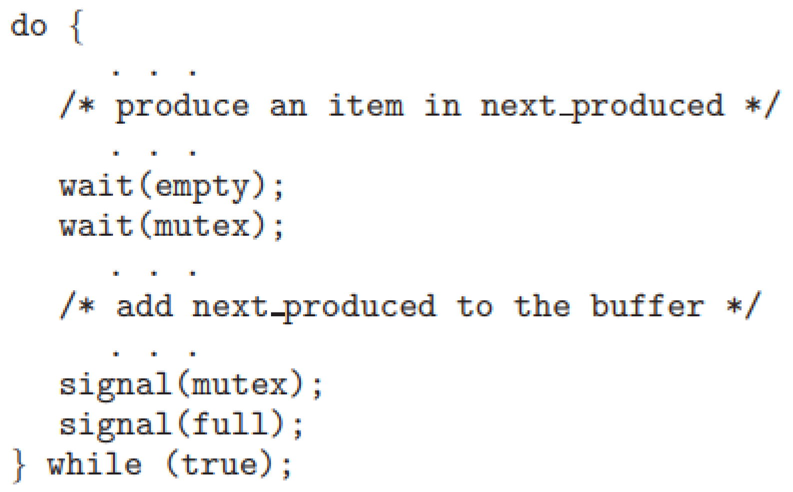
1. Have only two priorities in the system.
2. **Priority-inheritance protocol.** According to this protocol, all processes that are accessing resources needed by a higher-priority process inherit the higher priority until they are finished with the resources in question. When they are finished, their priorities revert to their original values.

Classical Problems of Synchronization

1. Bounded buffer problem

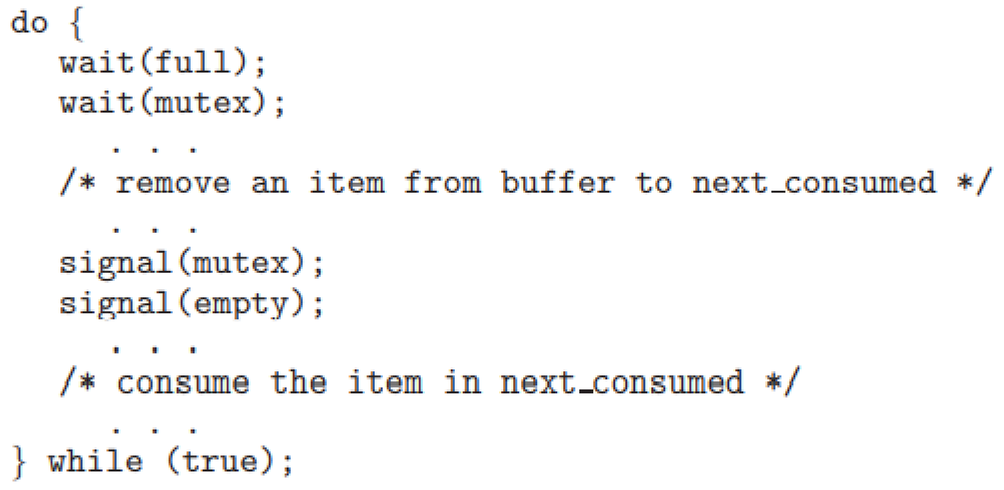
In our problem, the producer and consumer processes share the following data structures: int n; semaphore mutex = 1; semaphore empty = n; semaphore full = 0 .

We assume that the pool consists of n buffers, each capable of holding one item. The mutex semaphore provides mutual exclusion for accesses to the buffer pool and is initialized to the value 1. The empty and full semaphores count the number of empty and full buffers. The semaphore empty is initialized to the value n; the semaphore full is initialized to the value 0.



```
do {
     . . .
   /* produce an item in next_produced */
     . . .
   wait(empty);
   wait(mutex);
     . . .
   /* add next_produced to the buffer */
     . . .
   signal(mutex);
   signal(full);
} while (true);
```

Figure 5.9 The structure of the producer process.

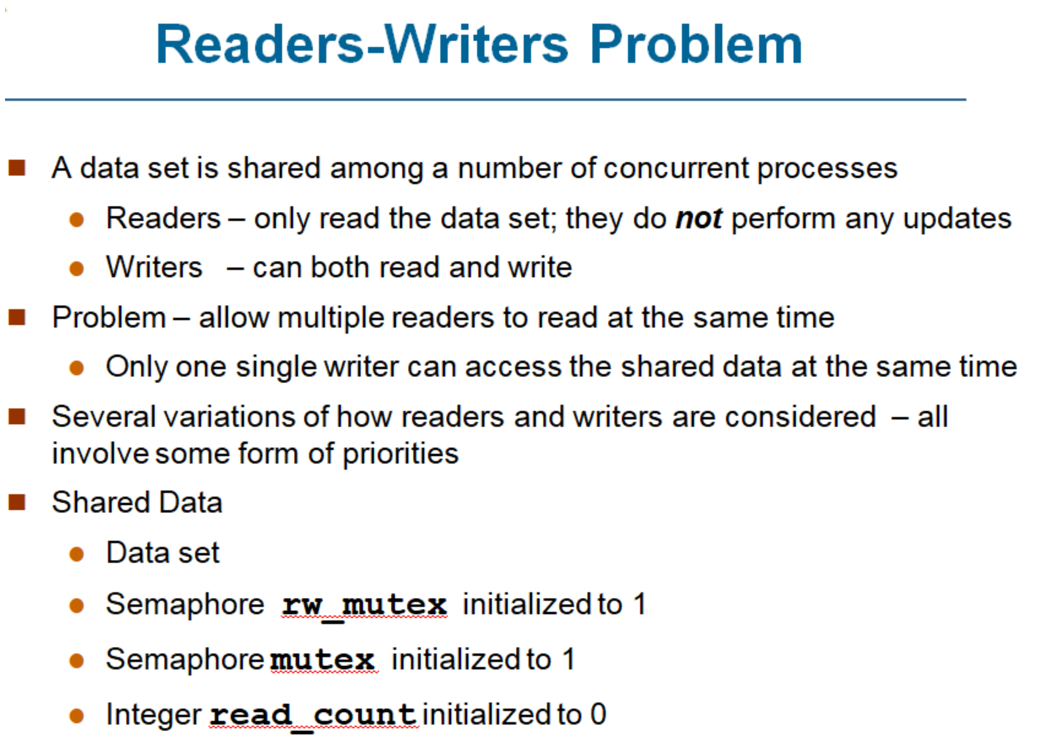


```
do {
   wait(full);
   wait(mutex);
     . . .
   /* remove an item from buffer to next_consumed */
     . . .
   signal(mutex);
   signal(empty);
     . . .
   /* consume the item in next_consumed */
     . . .
} while (true);
```

Figure 5.10 The structure of the consumer process.

Write explanation as per the video

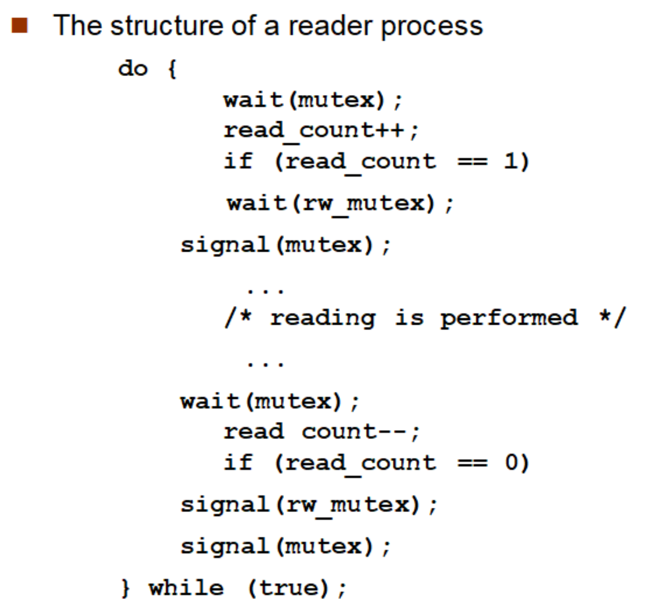
2. Readers Writers Problem



Readers-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 the same time
- Several variations of how readers and writers are considered – all involve some form of priorities
- Shared Data
  - Data set
  - Semaphore **`rw_mutex`** initialized to 1
  - Semaphore **`mutex`** initialized to 1
  - Integer **`read_count`** initialized to 0

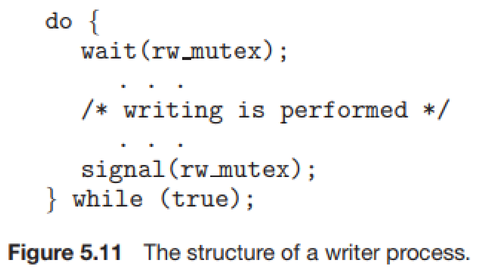
Reader Process



- The structure of a reader process

```
do {
       wait(mutex);
       read_count++;
       if (read_count == 1)
       wait(rw_mutex);
    signal(mutex);
        ...
       /* reading is performed */
        ...
    wait(mutex);
       read count--;
       if (read_count == 0)
    signal(rw_mutex);
    signal(mutex);
} while (true);
```

Writer Process



```
do {
   wait(rw_mutex);
      . . .
   /* writing is performed */
      . . .
   signal(rw_mutex);
} while (true);
```

Figure 5.11 The structure of a writer process.

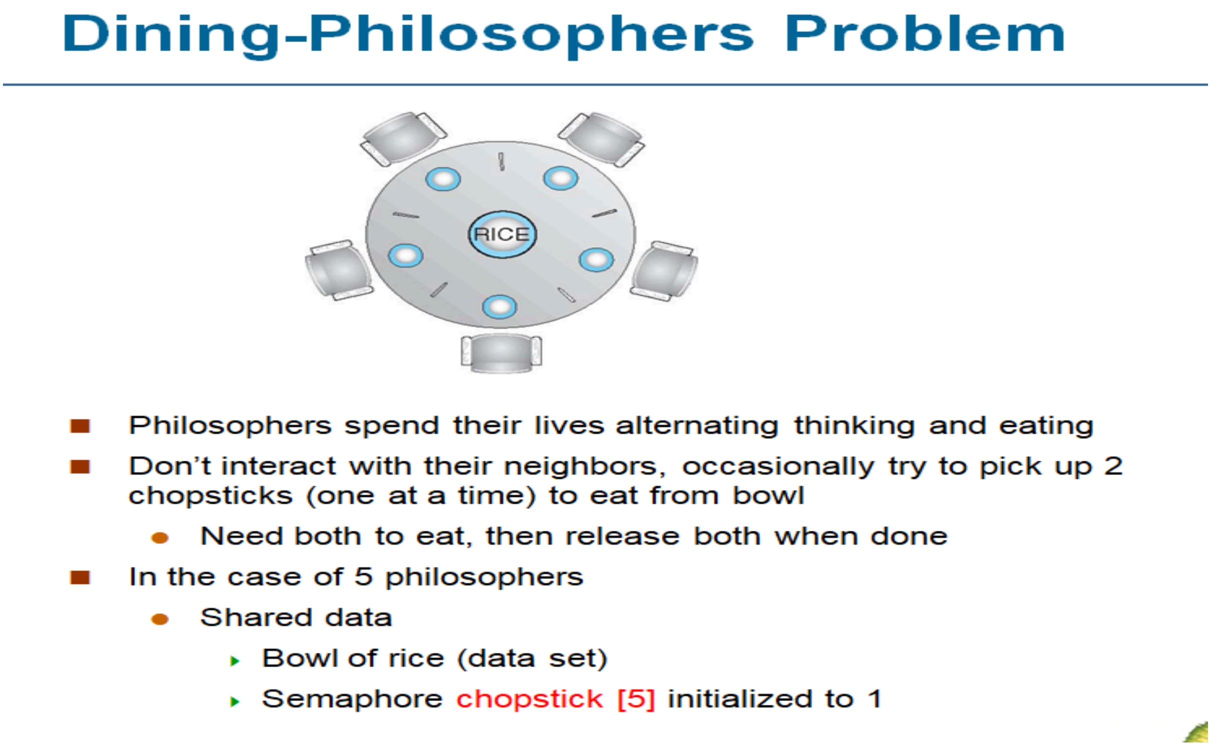
Working:

- The mutex semaphore is used to ensure mutual exclusion when the variable read count is updated.
- The read count variable keeps track of how many processes are currently reading the object.
- The semaphore rw mutex functions as a mutual exclusion semaphore for the writers. It is also used by the first or last reader that enters or exits the critical section. It is not used by readers who enter or exit while other readers are in their critical sections.
- If a writer is in the critical section and n readers are waiting, then one reader is queued on rw mutex, and n − 1 readers are queued on mutex.
- When a writer executes signal(rw mutex), we may resume the execution of either the waiting readers or a single waiting writer. The selection is made by the scheduler.

Variations:

1. First readers–writers problem, requires that no reader be kept waiting unless a writer has already obtained permission to use the shared object. In other words, no reader should wait for other readers to finish simply because a writer is waiting.
2. The second readers –writers problem requires that, once a writer is ready, that writer perform its write as soon as possible. In other words, if a writer is waiting to access the object, no new readers may start reading. A solution to either problem may result in starvation. In the first case, writers may starve; in the second case, readers may starve.

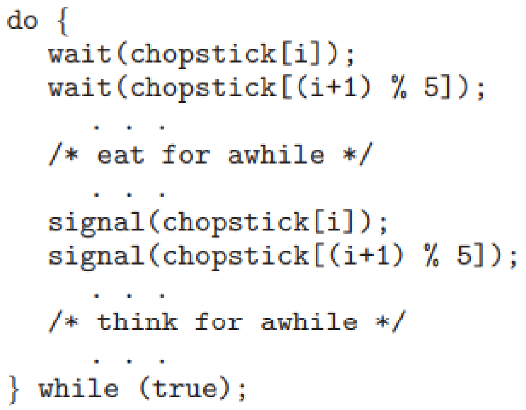
3. Dining Philosophers problem



Dining-Philosophers Problem

- Philosophers spend their lives alternating thinking and eating
- Don't interact with their neighbors, occasionally try to pick up 2 chopsticks (one at a time) to eat from bowl
  - Need both to eat, then release both when done
- In the case of 5 philosophers
  - Shared data
    - Bowl of rice (data set)
    - Semaphore chopstick [5] initialized to 1

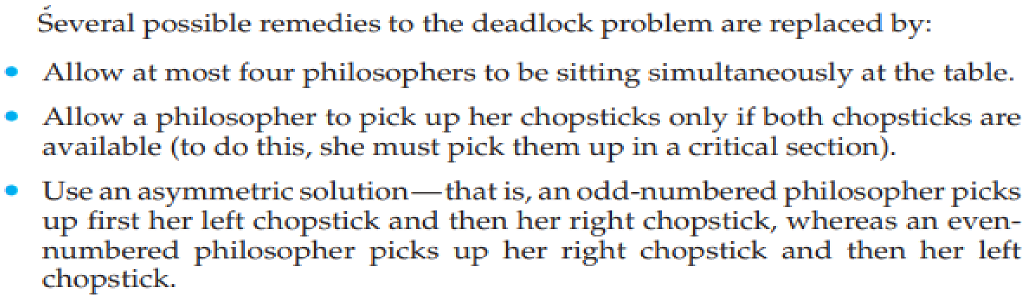
Solution using semaphore



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

Problem with this solution : Deadlock

Solution to avoid deadlock:

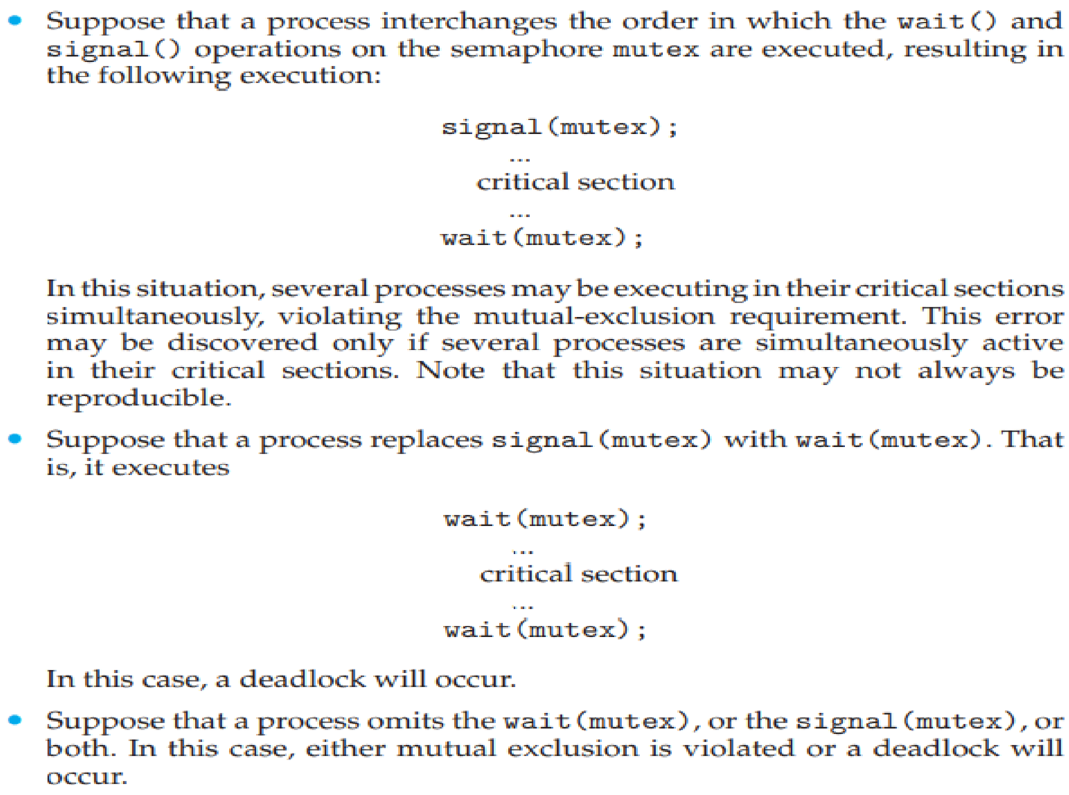


Several possible remedies to the deadlock problem are replaced by:

- Allow at most four philosophers to be sitting simultaneously at the table.
- Allow a philosopher to pick up her chopsticks only if both chopsticks are available (to do this, she must pick them up in a critical section).
- Use an asymmetric solution—that is, an odd-numbered philosopher picks up first her left chopstick and then her right chopstick, whereas an even-numbered philosopher picks up her right chopstick and then her left chopstick.

MONITORS

Issues with Semaphore



- Suppose that a process interchanges the order in which the `wait()` and `signal()` operations on the semaphore `mutex` are executed, resulting in the following execution:

```
signal(mutex);
   ...
   critical section
   ...
wait(mutex);
```

  In this situation, several processes may be executing in their critical sections simultaneously, violating the mutual-exclusion requirement. This error may be discovered only if several processes are simultaneously active in their critical sections. Note that this situation may not always be reproducible.

- Suppose that a process replaces `signal(mutex)` with `wait(mutex)`. That is, it executes

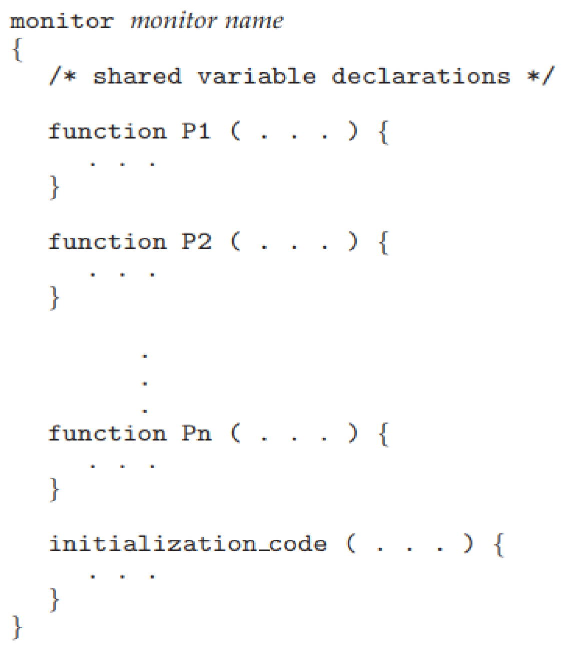
```
wait(mutex);
   ...
   critical section
   ...
wait(mutex);
```

  In this case, a deadlock will occur.

- Suppose that a process omits the `wait(mutex)`, or the `signal(mutex)`, or both. In this case, either mutual exclusion is violated or a deadlock will occur.

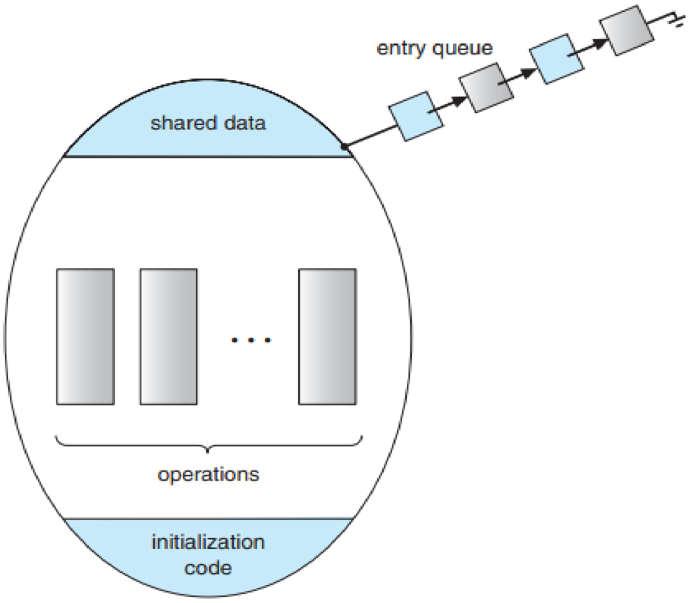
- An abstract data type—or **ADT**—encapsulates data with a set of functions to operate on that data that are independent of any specific implementation of the ADT. A monitor type is an ADT that includes a set of programmer defined operations that are provided with mutual exclusion within the monitor.
- The monitor type also declares the variables whose values define the state of an instance of that type, along with the bodies of functions that operate on those variables.
- A function defined within a monitor can access only those variables declared locally within the monitor and its formal parameters.
- The monitor construct ensures that only one process at a time is active within the monitor.

Syntax:



```
monitor monitor name
{
   /* shared variable declarations */

   function P1 ( . . . ) {
      . . .
   }

   function P2 ( . . . ) {
      . . .
   }

         .
         .
         .
   function Pn ( . . . ) {
      . . .
   }

   initialization_code ( . . . ) {
      . . .
   }
}
```

Schematic View of Monitor



- The monitor construct is not sufficiently powerful for modeling some synchronization schemes.

  Solution - **Condition variables**

  Syntax

  Condition x,y;

  Operations

  1. x.wait() - the process invoking this operation is suspended until another process invokes x.signal()
  2. The x.signal() operation resumes exactly one suspended process. If no process is suspended, then the signal() operation has no effect; that is, the state of x is the same as if the operation had never been executed

- Now suppose that, when the x.signal() operation is invoked by a process P, there exists a suspended process Q associated with condition x.
- Clearly, if the suspended process Q is allowed to resume its execution, the signaling process P must wait.
- Otherwise, both P and Q would be active simultaneously within the monitor.

1. **Signal and wait.** P either waits until Q leaves the monitor or waits for another condition.

2. **Signal and continue**. Q either waits until P leaves the monitor or waits for another condition.

Monitor and Condition Variables

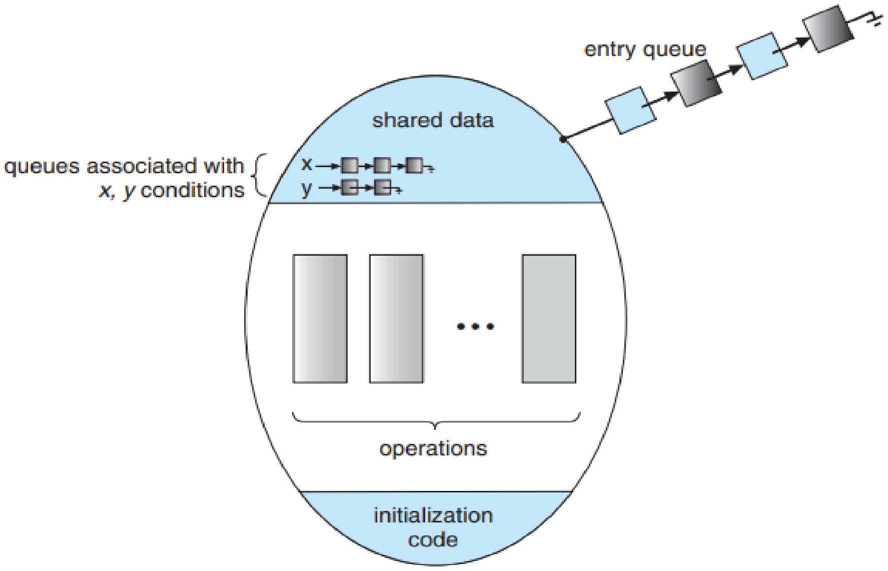
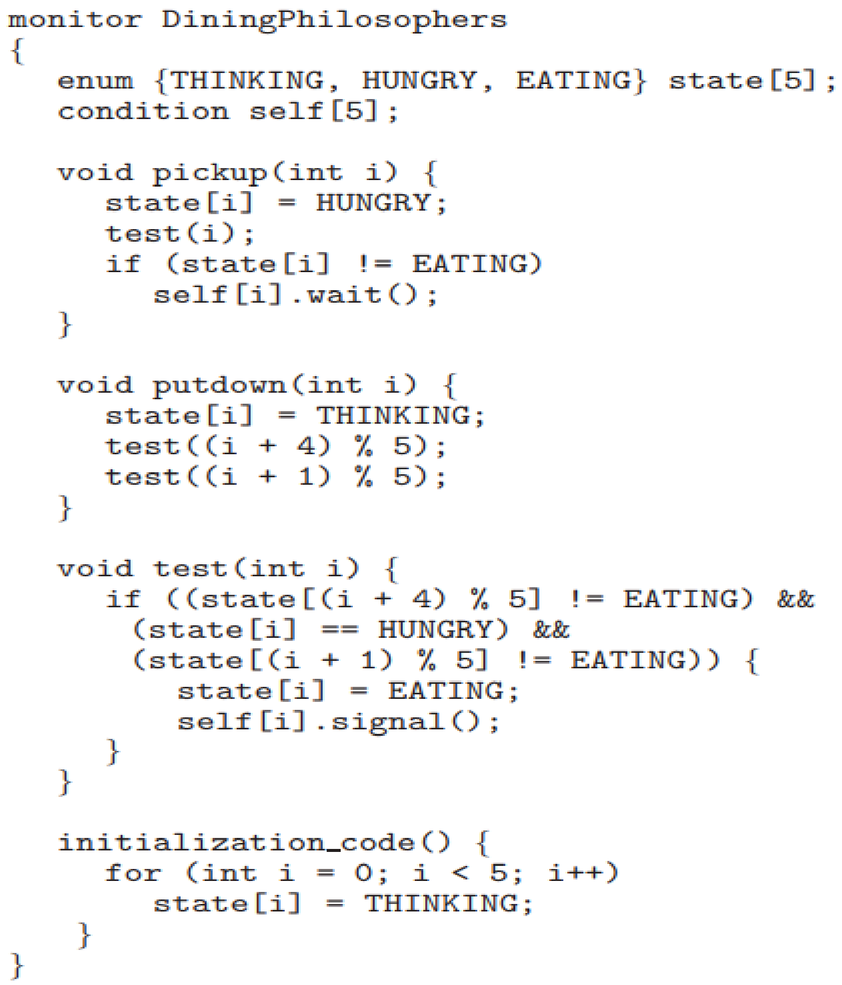


Figure 5.17 Monitor with condition variables.

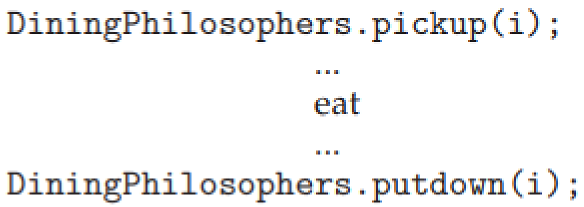
Dining Philosopher Solution using Monitors

- This solution imposes the restriction that a philosopher may pick up her chopsticks only if both of them are available. To code this solution, we need to distinguish among three states in which we may find a philosopher. For this purpose, we introduce the following data structure:
- **enum {THINKING, HUNGRY, EATING} state[5];**
- Philosopher i can set the variable state[i] = EATING only if her two neighbors are not eating: (state[(i+4) % 5] != EATING) and (state[(i+1) % 5] != EATING).



```
monitor DiningPhilosophers
{
   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((i + 4) % 5);
      test((i + 1) % 5);
   }

   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;
    }
}
```

Explanation from my video

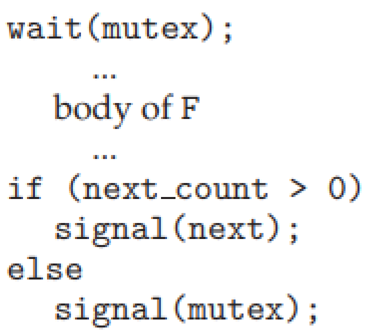


```
DiningPhilosophers.pickup(i);
                   ...
                   eat
                   ...
DiningPhilosophers.putdown(i);
```

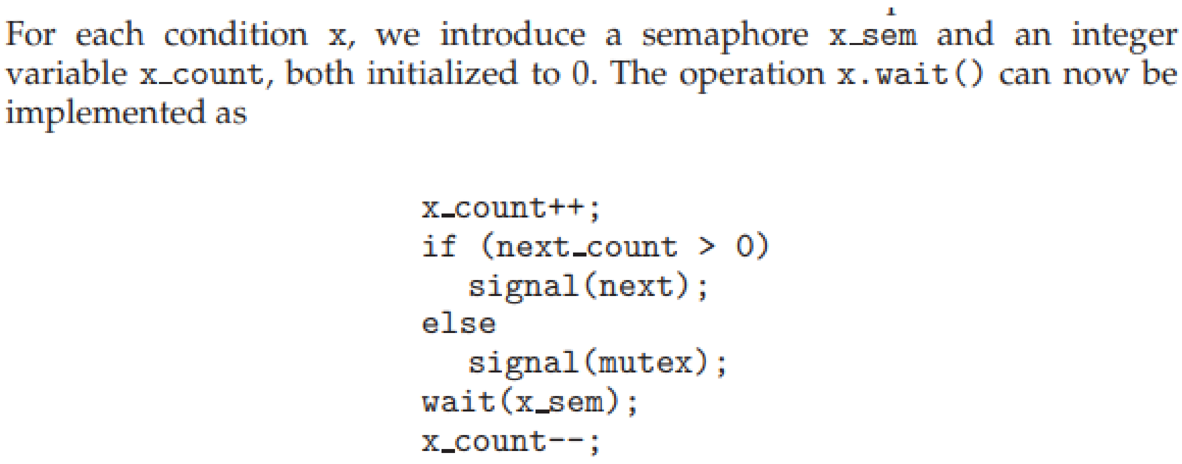
- It is easy to show that this solution ensures that no two neighbors are eating simultaneously and that no deadlocks will occur but starvation may occur.

Implementing a monitor using semaphores

- For each monitor, a semaphore mutex (initialized to 1) is provided. A process must execute wait(mutex) before entering the monitor and must execute signal(mutex) after leaving the monitor.
- Since a signaling process must wait until the resumed process either leaves or waits, an additional semaphore, next, is introduced, initialized to 0.
- The signaling processes can use next to suspend themselves.
- An integer variable next count is also provided to count the number of processes suspended on next.



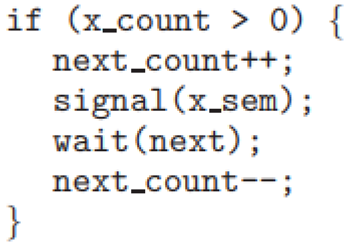
```
wait(mutex);
   ...
   body of F
   ...
if (next_count > 0)
   signal(next);
else
   signal(mutex);
```



For each condition x, we introduce a semaphore x_sem and an integer variable x_count, both initialized to 0. The operation x.wait() can now be implemented as

```
x_count++;
if (next_count > 0)
   signal(next);
else
   signal(mutex);
wait(x_sem);
x_count--;
```

x.signal()



```
if (x_count > 0) {
   next_count++;
   signal(x_sem);
   wait(next);
   next_count--;
}
```

Resuming Processes within Monitors

- If several processes are suspended on condition x, and an x.signal() operation is executed by some process, then how do we determine which of the suspended processes should be resumed next?
- One simple solution is to use a first-come, first-served (FCFS) ordering, so that the process that has been waiting the longest is resumed first. In many circumstances, however, such a simple scheduling scheme is not adequate.
- For this purpose, the conditional-wait construct can be used.
  - **x.wait(c);**
  - **where c is an integer expression that is evaluated when the wait() operation is executed.**
- The value of c, which is called a priority number, is then stored with the name of the process that is suspended. When x.signal() is executed, the process with the smallest priority number is resumed next.

Resource Allocation

A process that needs to access the resource in question must observe the following sequence:

```
R.acquire(t);
... access the resource; ...
R.release();
```

where R is an instance of type ResourceAllocator.

Unfortunately, the monitor concept cannot guarantee that the preceding access sequence will be observed. In particular, the following problems can occur.

1. A process might access a resource without first gaining access permission to the resource.
2. A process might never release a resource once it has been granted access to the resource.
3. A process might attempt to release a resource that it never requested.
4. A process might request the same resource twice (without first releasing the resource).

DEADLOCKS

System Model

- A system consists of a finite number of resources to be distributed among a number of competing processes.
- The resources may be partitioned into several types (or classes), each consisting of some number of identical instances. CPU cycles, files, and I/O devices (such as printers and DVD drives) are examples of resource types. If a system has two CPUs, then the resource type CPU has two instances.
- If a process requests an instance of a resource type, the allocation of any instance of the type should satisfy the request. If it does not, then the instances are not identical, and the resource type classes have not been defined properly.

Steps in Utilizing a resource

1. **Request.** The process requests the resource. If the request cannot be granted immediately (for example, if the resource is being used by another process), then the requesting process must wait until it can acquire the resource.
2. **Use**. The process can operate on the resource (for example, if the resource is a printer, the process can print on the printer).
3. **Release.** The process releases the resource

Deadlock Characterization - Necessary Conditions

1. **Mutual exclusion.** At least one resource must be held in a nonsharable mode; that is, only one process at a time can use the resource. If another process requests that resource, the requesting process must be delayed until the resource has been released.
2. **Hold and wait.** A process must be holding at least one resource and waiting to acquire additional resources that are currently being held by other processes.
3. **No preemption.** Resources cannot be preempted; that is, a resource can be released only voluntarily by the process holding it, after that process has completed its task.
4. **Circular wait.** A set {P0, P1, ..., Pn} of waiting processes must exist such that P0 is waiting for a resource held by P1, P1 is waiting for a resource held by P2, ..., Pn−1 is waiting for a resource held by Pn, and Pn is waiting for a resource held by P0.

Resource Allocation Graph

- This graph consists of a set of vertices V and a set of edges E.
- The set of vertices V is partitioned into two different types of nodes: P = {P1, P2, ..., Pn}, the set consisting of all the active processes in the system, and R = {R1, R2, ..., Rm}, the set consisting of all resource types in the system.
- A directed edge from process Pi to resource type Rj is denoted by Pi → Rj ; it signifies that process Pi has requested an instance of resource type Rj and is currently waiting for that resource.
- A directed edge from resource type Rj to process Pi is denoted by Rj → Pi ; it signifies that an instance of resource type Rj has been allocated to process Pi .
- A directed edge Pi → Rj is called a request edge; a directed edge Rj → Pi is

called an assignment edge.

Representation

Process - circle

Resource - rectangle.

- Since resource type Rj may have more than one instance, we represent each such instance as a dot within the rectangle.
- When process Pi requests an instance of resource type Rj , a request edge is inserted in the resource-allocation graph.
- When this request can be fulfilled, the request edge is instantaneously transformed to an assignment edge.
- When the process no longer needs access to the resource, it releases the resource. As a result, the assignment edge is deleted.

Example 1

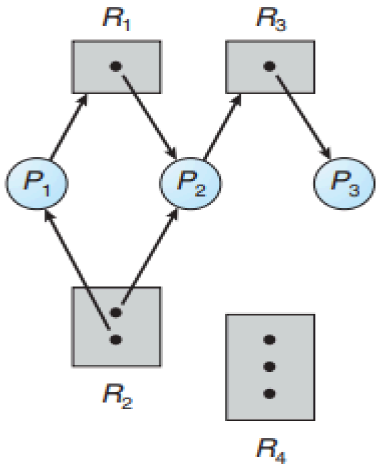


Figure 7.1 Resource-allocation graph.

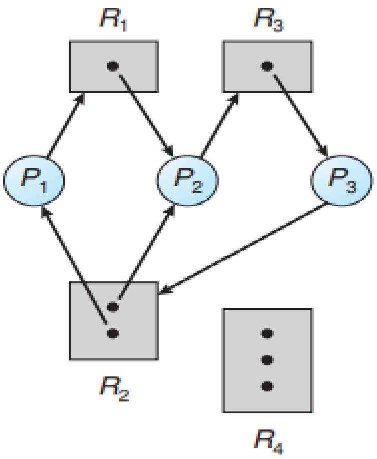
  - $R = \{R_1, R_2, R_3, R_4\}$
  - $E = \{P_1 \rightarrow R_1, P_2 \rightarrow R_3, R_1 \rightarrow P_2, R_2 \rightarrow P_2, R_2 \rightarrow P_1, R_3 \rightarrow P_3\}$

- 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.

Deadlock using Resource Allocation Graph

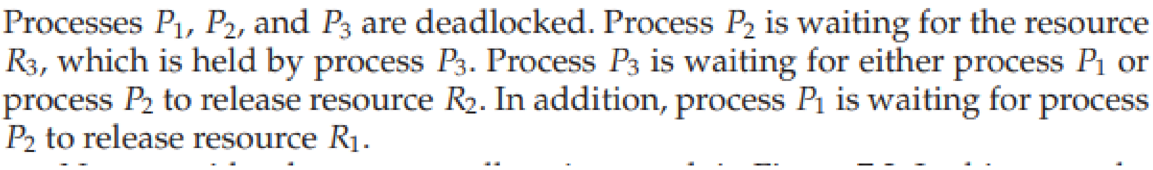
- Given the definition of a resource-allocation graph, it can be shown that, if the graph contains no cycles, then no process in the system is deadlocked.
- If the graph does contain a cycle, then a deadlock may exist.
- If each resource type has exactly one instance, then a cycle implies that a deadlock has occurred.
- If each resource type has several instances, then a cycle does not necessarily imply that a deadlock has occurred.

Example 2 : Graph with deadlock

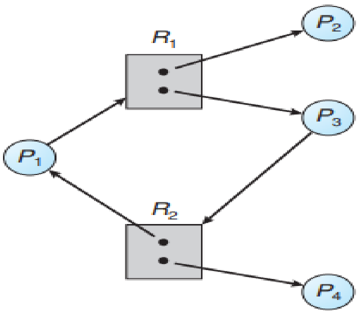


Cycles are as follows:

$$P_1 \rightarrow R_1 \rightarrow P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$$
$$P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_2$$


Processes P_1, P_2, and P_3 are deadlocked. Process P_2 is waiting for the resource R_3, which is held by process P_3. Process P_3 is waiting for either process P_1 or process P_2 to release resource R_2. In addition, process P_1 is waiting for process P_2 to release resource R_1.

Example 3: Cycle but no deadlock



Process P4 may release its instance of resource type R2. That resource can then be allocated to P3, breaking the cycle.

METHODS FOR HANDLING DEADLOCKS

3 WAYS

• We can use a protocol to prevent or avoid deadlocks, ensuring that the system will never enter a deadlocked state.

• We can allow the system to enter a deadlocked state, detect it, and recover.

• We can ignore the problem altogether and pretend that deadlocks never occur in the system.

1. DEADLOCK PREVENTION

By ensuring that at least one of these conditions cannot hold, we can prevent the occurrence of a deadlock.

a) **Mutual Exclusion**

   i) The mutual exclusion condition must hold. That is, at least one resource must be nonsharable.

   ii) Sharable resources, in contrast, do not require mutually exclusive access and thus cannot be involved in a deadlock. Read-only files are a good example of a sharable resource.

   iii) If several processes attempt to open a read-only file at the same time, they can be granted simultaneous access to the file.

   iv) A process never needs to wait for a sharable resource.

   v) In general, however, we cannot prevent deadlocks by denying the mutual-exclusion condition, because some resources are intrinsically nonsharable. For example, a mutex lock cannot be simultaneously shared by several processes

b) Hold and wait

   i) To ensure that the hold-and-wait condition never occurs in the system, we must guarantee that, whenever a process requests a resource, it does not hold any other resources.

   ii) One protocol that we can use requires each process to request and be allocated all its resources before it begins execution.

   iii) An alternative protocol allows a process to request resources only when it has none. A process may request some resources and use them. Before it can request any additional resources, it must release all the resources that it is currently allocated.

   iv) Drawbacks:

      1) resource utilization may be low, since resources may be allocated but unused for a long periods.

      2) starvation is possible.

c) No preemption

   i) If a process is holding some resources and requests another resource that cannot be immediately allocated to it (that is, the process must wait), then all resources the process is currently holding are preempted.

   ii) In other words, these resources are implicitly released.

   iii) The preempted resources are added to the list of resources for which the process is waiting. The process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting.

iv) Alternatively, if a process requests some resources, we first check whether they are available. If they are, we allocate them.

v) If they are not, we check whether they are allocated to some other process that is waiting for additional resources. If so, we preempt the desired resources from the waiting process and allocate them to the requesting process.

d) Circular Wait

i) Impose a total ordering of all resource types and to require that each process requests resources in an increasing order of enumeration.

ii) To illustrate, we let R = {R1, R2, ..., Rm} be the set of resource types. We assign to each resource type a unique integer number, which allows us to compare two resources and to determine whether one precedes another in our ordering.

iii) Example:

1) F(tape drive) = 1 F(disk drive) = 5 F(printer) = 12
2) Each process can request resources only in an increasing order of enumeration.
3) a process that wants to use the tape drive and printer at the same time must first request the tape drive and then request the printer
4) a process requesting an instance of resource type Rj must have released any resources Ri such that F(Ri) ≥ F(Rj)

2. **DEADLOCK AVOIDANCE**

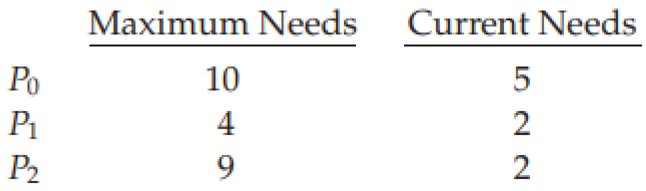
- Effects of preventing deadlocks by this method, however, are low device utilization and reduced system throughput.
- An alternative method for avoiding deadlocks is to require additional information about how resources are to be requested.
- For example, in a system with one tape drive and one printer, the system might need to know that process P will request first the tape drive and then the printer before releasing both resources, whereas process Q will request first the printer and then the tape drive.
- With this knowledge of the complete sequence of requests and releases for each process, the system can decide for each request whether or not the process should wait in order to avoid a possible future deadlock.
- Each request requires that in making this decision the system consider the resources currently available, the resources currently allocated to each process, and the future requests and releases of each process.

Safe State

- A state is safe if the system can allocate resources to each process (up to its maximum) in some order and still avoid a deadlock.
- A system is in a safe state only if there exists a safe sequence.
- A sequence of processes is a safe sequence for the current allocation state if, for each Pi , the resource requests that Pi can still make can be satisfied by the currently available resources plus the resources held by all Pj , with j < i.
- A safe state is not a deadlocked state.

Example

- We consider a system with twelve magnetic tape drives and three processes: P0, P1, and P2. Process P0 requires ten tape drives, process P1 may need as many as four tape drives, and process P2 may need up to nine tape drives.
- Suppose that, at time t0, process P0 is holding five tape drives, process P1 is holding two tape drives, and process P2 is holding two tape drives. (Thus, there are three free tape drives.

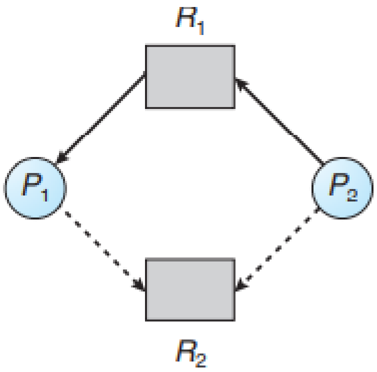


	Maximum Needs	Current Needs
P_0	10	5
P_1	4	2
P_2	9	2

- At time t0, the system is in a safe state. The sequence satisfies the safety condition. Process P1 can immediately be allocated all its tape drives and then return them (the system will then have five available tape drives); then process P0 can get all its tape drives and return them (the system will then have ten available tape drives); and finally process P2 can get all its tape drives and return them (the system will then have all twelve tape drives available).

- A system can go from a safe state to an unsafe state. Suppose that, at time t1, process P2 requests and is allocated one more tape drive. The system is no longer in a safe state.

Resource-Allocation-Graph Algorithm

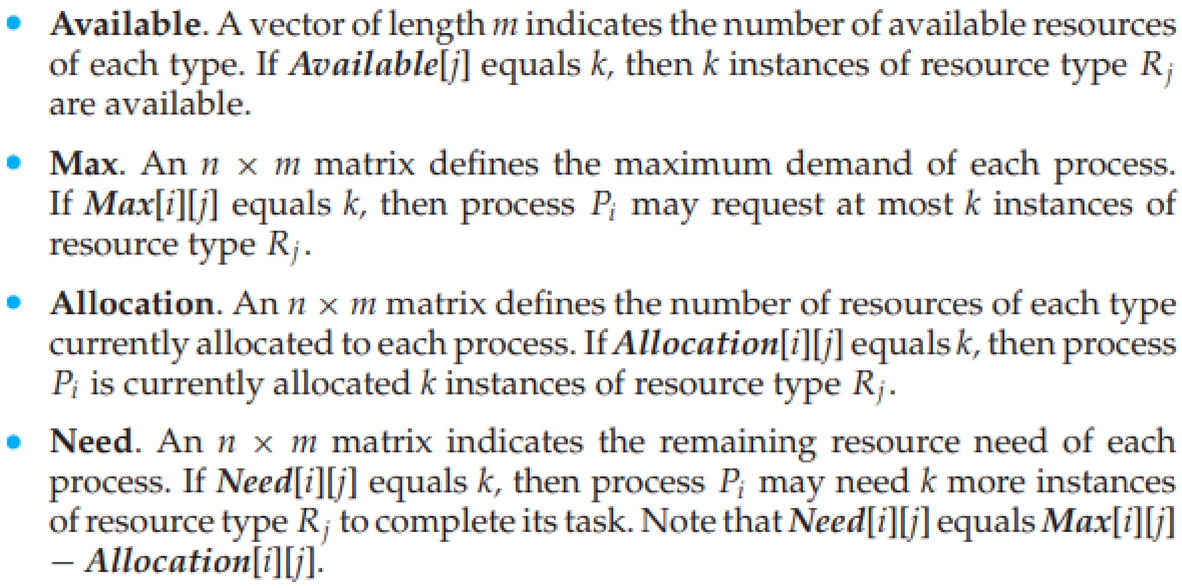


- A claim edge Pi → Rj indicates that process Pi may request resource Rj at some time in the future. This edge resembles a request edge in direction but is represented in the graph by a dashed line.
- When process Pi requests resource Rj , the claim edge Pi → Rj is converted to a request edge. Similarly, when a resource Rj is released by Pi , the assignment edge Rj → Pi is reconverted to a claim edge Pi → Rj .
- Note that the resources must be claimed a priori in the system. That is, before process Pi starts executing, all its claim edges must already appear in the resource-allocation graph.
- Now suppose that process Pi requests resource Rj . The request can be granted only if converting the request edge Pi → Rj to an assignment edge Rj → Pi does not result in the formation of a cycle in the resource-allocation graph.
- If no cycle exists, then the allocation of the resource will leave the system in a safe state.

Bankers Algorithm

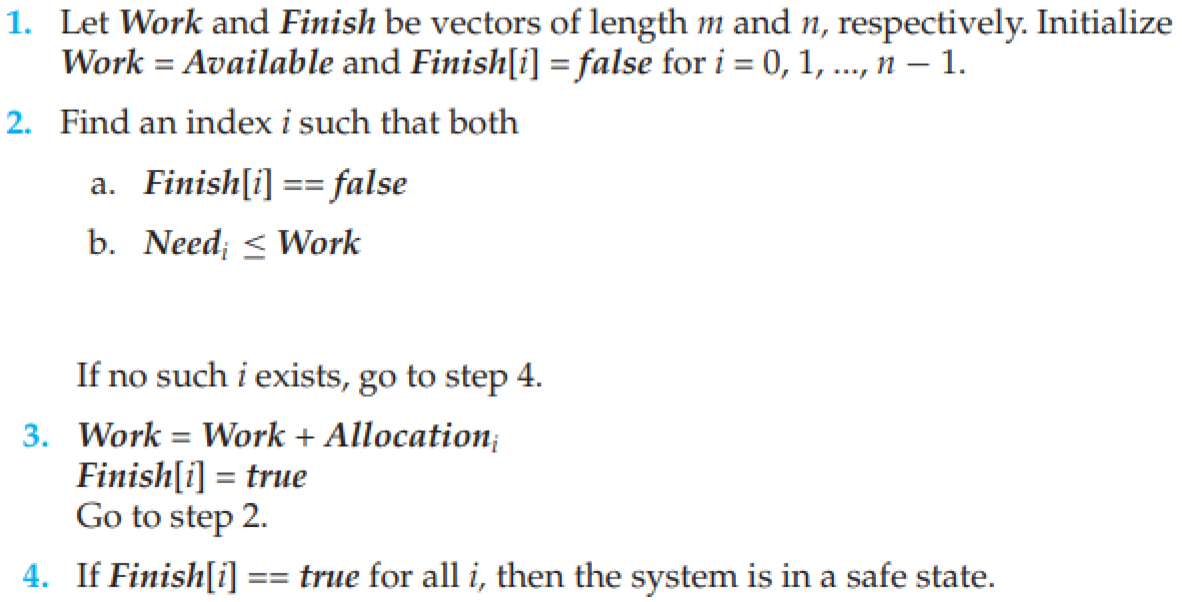
- The resource-allocation-graph algorithm is not applicable to a resource allocation system with multiple instances of each resource type.
- The name was chosen because the algorithm could be used in a banking system to ensure that the bank never allocated its available cash in such a way that it could no longer satisfy the needs of all its customers.
- When a new process enters the system, it must declare the maximum number of instances of each resource type that it may need. This number may not exceed the total number of resources in the system.
- When a user requests a set of resources, the system must determine whether the allocation of these resources will leave the system in a safe state. If it will, the resources are allocated; otherwise, the process must wait until some other process releases enough resources.

Data Structures used in Banker's Algorithm



- **Available**. A vector of length m indicates the number of available resources of each type. If ***Available***$[j]$ equals k, then k instances of resource type R_j are available.
- **Max**. An $n \times m$ matrix defines the maximum demand of each process. If ***Max***$[i][j]$ equals k, then process P_i may request at most k instances of resource type R_j.
- **Allocation**. An $n \times m$ matrix defines the number of resources of each type currently allocated to each process. If ***Allocation***$[i][j]$ equals k, then process P_i is currently allocated k instances of resource type R_j.
- **Need**. An $n \times m$ matrix indicates the remaining resource need of each process. If ***Need***$[i][j]$ equals k, then process P_i may need k more instances of resource type R_j to complete its task. Note that ***Need***$[i][j]$ equals ***Max***$[i][j]$ − ***Allocation***$[i][j]$.

Safety Algorithm - Used to find if system is in safe state



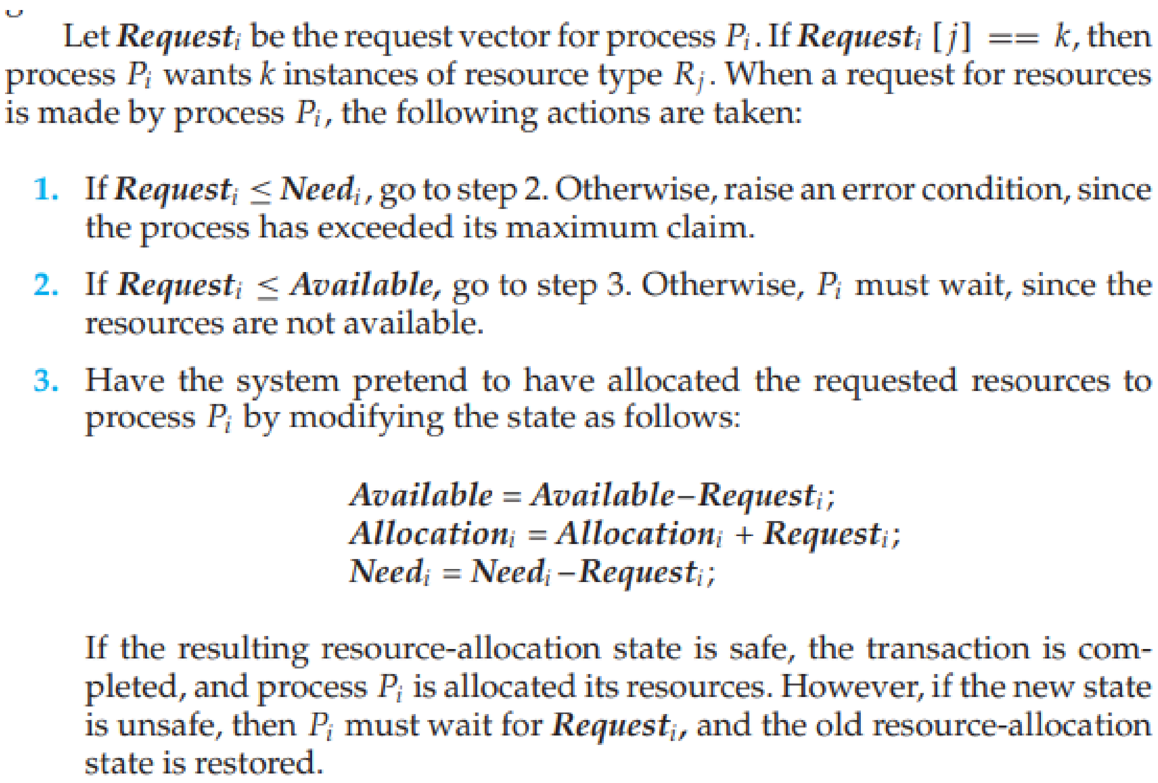
1. Let ***Work*** and ***Finish*** be vectors of length m and n, respectively. Initialize ***Work*** = ***Available*** and ***Finish***$[i]$ = ***false*** for $i = 0, 1, ..., n - 1$.
2. Find an index i such that both

   a. ***Finish***$[i]$ == ***false***

   b. ***Need***$_i \leq$ ***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***$[i]$ == ***true*** for all i, then the system is in a safe state.

Resource-Request Algorithm



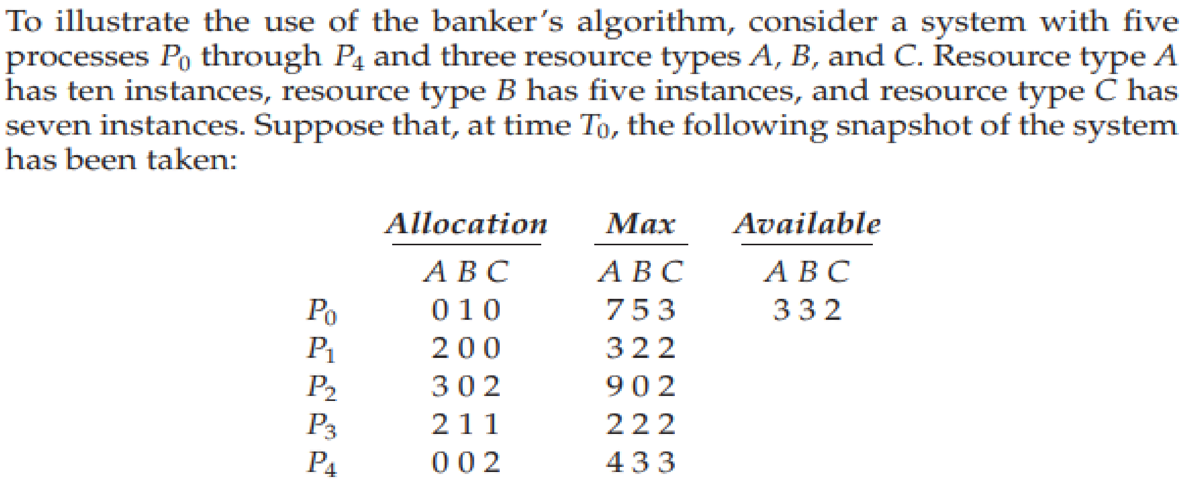
Let $\textbf{\textit{Request}}_i$ be the request vector for process P_i. If $\textbf{\textit{Request}}_i\,[j] == k$, then process P_i wants k instances of resource type R_j. When a request for resources is made by process P_i, the following actions are taken:

1. If $\textbf{\textit{Request}}_i \le \textbf{\textit{Need}}_i$, go to step 2. Otherwise, raise an error condition, since the process has exceeded its maximum claim.
2. If $\textbf{\textit{Request}}_i \le \textbf{\textit{Available}}$, go to step 3. Otherwise, P_i must wait, since the resources are not available.
3. Have the system pretend to have allocated the requested resources to process P_i by modifying the state as follows:

$$\begin{aligned}&\textbf{\textit{Available}} = \textbf{\textit{Available}} - \textbf{\textit{Request}}_i;\\&\textbf{\textit{Allocation}}_i = \textbf{\textit{Allocation}}_i + \textbf{\textit{Request}}_i;\\&\textbf{\textit{Need}}_i = \textbf{\textit{Need}}_i - \textbf{\textit{Request}}_i;\end{aligned}$$

If the resulting resource-allocation state is safe, the transaction is completed, and process P_i is allocated its resources. However, if the new state is unsafe, then P_i must wait for $\textbf{\textit{Request}}_i$, and the old resource-allocation state is restored.

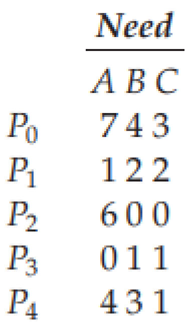
Example 1:



To illustrate the use of the banker's algorithm, consider a system with five processes P_0 through P_4 and three resource types A, B, and C. Resource type A has ten instances, resource type B has five instances, and resource type C has seven instances. Suppose that, at time T_0, the following snapshot of the system has been taken:

	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	

Need = Max - Allocation

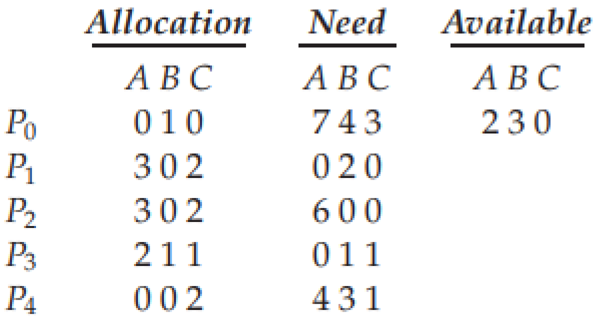


	Need
	A B C
P_0	7 4 3
P_1	1 2 2
P_2	6 0 0
P_3	0 1 1
P_4	4 3 1

Detailed solution in video

- The sequence (P1,P3,P4,P2,P0) satisfies the safety criteria.
- Suppose now that process P1 requests one additional instance of resource type A and two instances of resource type C, so Request1 = (1,0,2). To decide whether this request can be immediately granted, we first check that Request1 ≤ Available—that is, that (1,0,2) ≤ (3,3,2), which is true.

New state:



	Allocation	*Need*	*Available*
	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 2	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

- We must determine whether this new system state is safe. To do so, we execute our safety algorithm and find that the sequence satisfies the safety requirement.
- Hence, we can immediately grant the request of process P1.
- When the system is in this state, a request for (3,3,0) by P4 cannot be granted, since the resources are not available.
- Furthermore, a request for (0,2,0) by P0 cannot be granted, even though the resources are available, since the resulting state is unsafe

DEADLOCK DETECTION

- If a system does not employ either a deadlock-prevention or a deadlockavoidance algorithm, then a deadlock situation may occur.
- In this environment, the system may provide:
  - An algorithm that examines the state of the system to determine whether a deadlock has occurred
  - An algorithm to recover from the deadlock

Single Instance of Each Resource Type

- If all resources have only a single instance, then we can define a deadlock detection algorithm that uses a variant of the resource-allocation graph, called a wait-for graph. We obtain this graph from the resource-allocation graph by removing the resource nodes and collapsing the appropriate edges.
- An edge from Pi to Pj in a wait-for graph implies that process Pi is waiting for process Pj to release a resource that Pi needs. An edge Pi → Pj exists in a wait-for graph if and only if the corresponding resource allocation graph contains two edges Pi → Rq and Rq → Pj for some resource R.

Example:

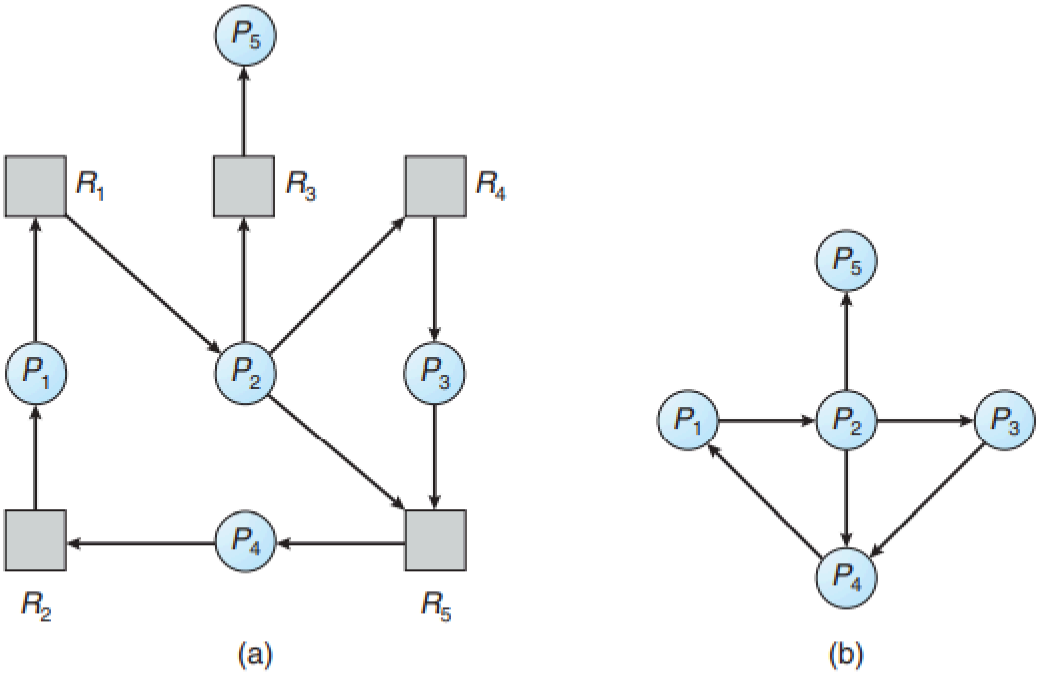
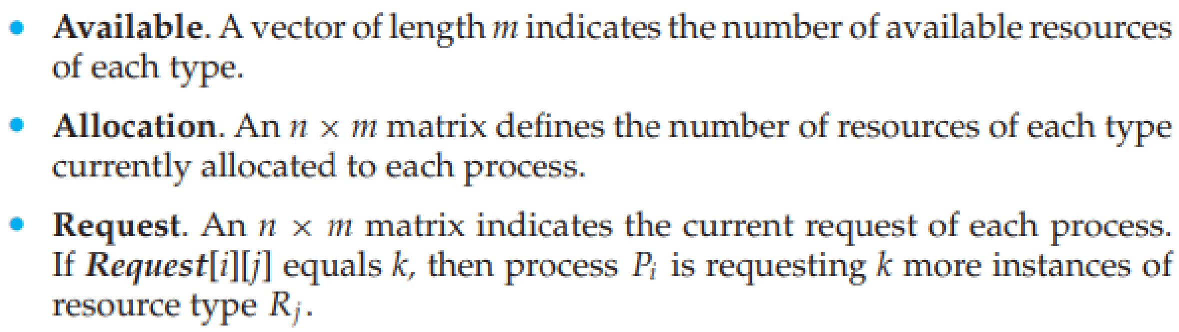


Figure 7.9 (a) Resource-allocation graph. (b) Corresponding wait-for graph.

- As before, a deadlock exists in the system if and only if the wait-for graph contains a cycle.
- To detect deadlocks, the system needs to maintain the waitfor graph and 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-square operations, where n is the number of vertices in the graph.

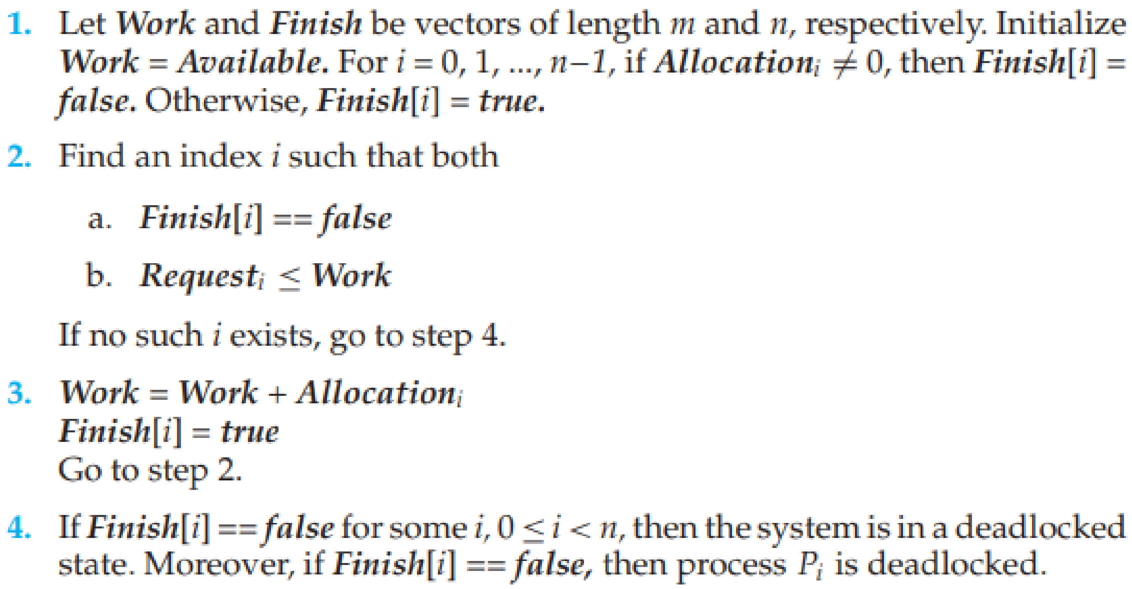
Several Instances of a Resource Type - Deadlock Detection Algorithm

Data Structures



- **Available**. A vector of length m indicates the number of available resources of each type.
- **Allocation**. An $n \times m$ matrix defines the number of resources of each type currently allocated to each process.
- **Request**. An $n \times m$ matrix indicates the current request of each process. If ***Request***$[i][j]$ equals k, then process P_i is requesting k more instances of resource type R_j.

Steps



1. Let ***Work*** and ***Finish*** be vectors of length m and n, respectively. Initialize ***Work*** = ***Available***. For $i = 0, 1, ..., n-1$, if ***Allocation***$_i \neq 0$, then ***Finish***$[i]$ = ***false***. Otherwise, ***Finish***$[i]$ = ***true***.
2. Find an index i such that both

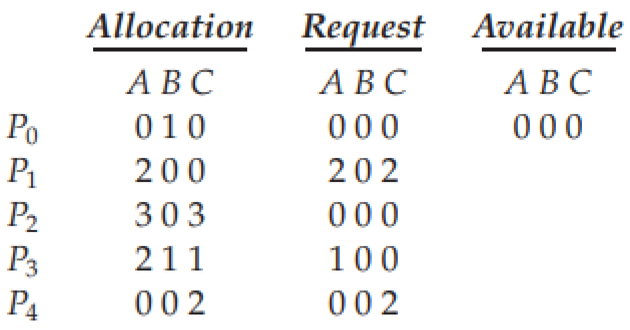
   a. ***Finish***$[i]$ == ***false***

   b. ***Request***$_i \leq$ ***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, $0 \leq i < n$, then the system is in a deadlocked state. Moreover, if ***Finish***$[i]$ == ***false***, then process P_i is deadlocked.

Example

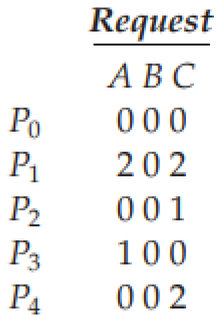
- a system with five processes P0 through P4 and three resource types A, B, and C. Resource type A has seven instances, resource type B has two instances, and resource type C has six instances. Suppose that, at time T0, we have the following resource-allocation state.



	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	

We claim that the system is not in a deadlocked state. Indeed, if we execute our algorithm, we will find that the sequence results in Finish[i] == true for all i.

- Suppose now that process P2 makes one additional request for an instance of type C. The Request matrix is modified as follows.



	Request *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

- The system is now deadlocked. Although we can reclaim the resources held by process P0, the number of available resources is not sufficient to fulfill the requests of the other processes. Thus, a deadlock exists, consisting of processes P1, P2, P3, and P4.

Detection-Algorithm Usage

When should we invoke the detection algorithm?

The answer depends on two factors:

1. How often is a deadlock likely to occur?

2. How many processes will be affected by deadlock when it happens?

- If deadlocks occur frequently, then the detection algorithm should be invoked frequently. Resources allocated to deadlocked processes will be idle until the deadlock can be broken. In addition, the number of processes involved in the deadlock cycle may grow.
- Invoking the deadlock-detection algorithm for every resource request will incur considerable overhead in computation time. A less expensive alternative is simply to invoke the algorithm at defined intervals.
- If the detection algorithm is invoked at arbitrary points in time, the resource graph may contain many cycles. In this case, we generally cannot tell which of the many deadlocked processes “caused” the deadlock.

RECOVERY FROM DEADLOCK

- There are two options for breaking a deadlock. One is simply to abort one or more processes to break the circular wait. The other is to preempt some resources from one or more of the deadlocked processes.

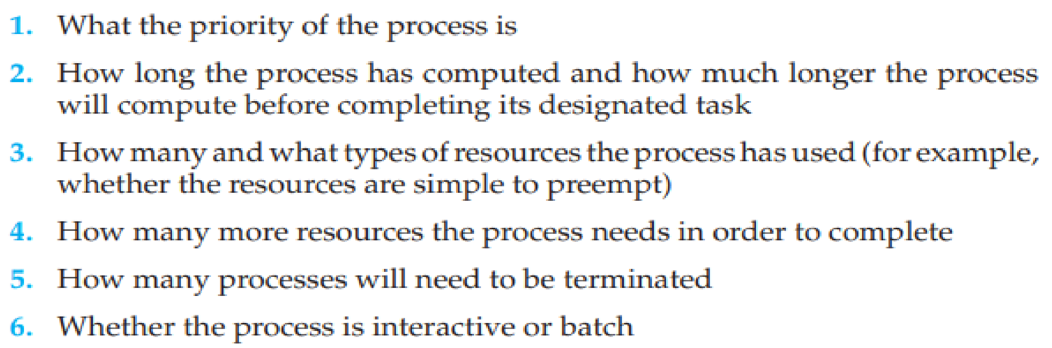
Process Termination

1. **Abort all deadlocked processes.**
   a. This method clearly will break the deadlock cycle, but at great expense.
   b. The deadlocked processes may have computed for a long time, and the results of these partial computations must be discarded and probably will have to be recomputed later.
2. **Abort one process at a time until the deadlock cycle is eliminated.**
   a. This method incurs considerable overhead, since after each process is aborted, a deadlock-detection algorithm must be invoked to determine whether any processes are still deadlocked.

- Aborting a process may not be easy. If the process was in the midst of updating a file, terminating it will leave that file in an incorrect state. Similarly, if the process was in the midst of printing data on a printer, the system must reset the printer to a correct state before printing the next job.

How a process is selected for termination?

Many factors may affect which process is chosen, including:



1. What the priority of the process is
2. How long the process has computed and how much longer the process will compute before completing its designated task
3. How many and what types of resources the process has used (for example, whether the resources are simple to preempt)
4. How many more resources the process needs in order to complete
5. How many processes will need to be terminated
6. Whether the process is interactive or batch

Resource Preemption

1. **Selecting a victim.**
   a. Which resources and which processes are to be preempted? As in process termination, we must determine the order of preemption to minimize cost.
   b. Cost factors may include such parameters as the number of resources a deadlocked process is holding and the amount of time the process has thus far consumed.
2. **Rollback.**
   a. If we preempt a resource from a process, what should be done with that process? Clearly, it cannot continue with its normal execution; it is missing some needed resource.
   b. We must roll back the process to some safe state and restart it from that state. Since, in general, it is difficult to determine what a safe state is, the simplest solution is a total rollback: abort the process and then restart it.
   c. Although it is more effective to roll back the process only as far as necessary to break the deadlock, this method requires the system to keep more information about the state of all running processes.
3. **Starvation.** The same resources may be terminated again and again.