CTBTCSE – SIV P4 – Operating System

“Unit-2 Introduction to Operating System”

Dr. Parag Shukla
Assistant Professor,
School of Cyber Security and Digital Forensics
National Forensic Sciences University

INTRODUCTION TO OPERATING SYSTEM

Presentation Outline

- **Process Management**
- **IPC, Thread & Concurrency**
- **Scheduling**
- **Process Synchronization**
- **Deadlocks**

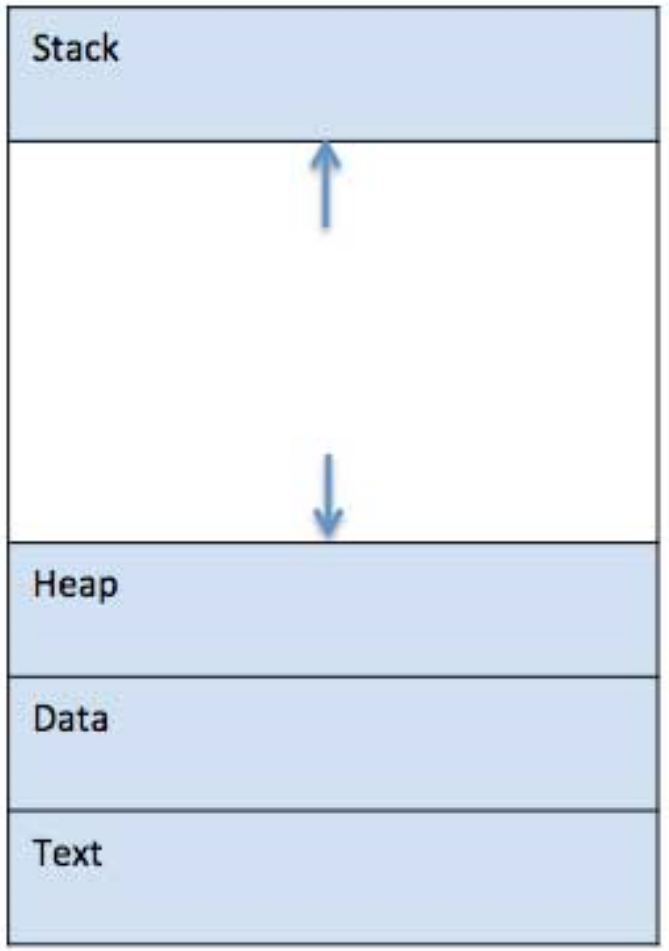
Process Management

- What is Process?
- Process Life Cycle or Process States
- Process Control Block
- Process Scheduling
- Inter-process Communication

Process

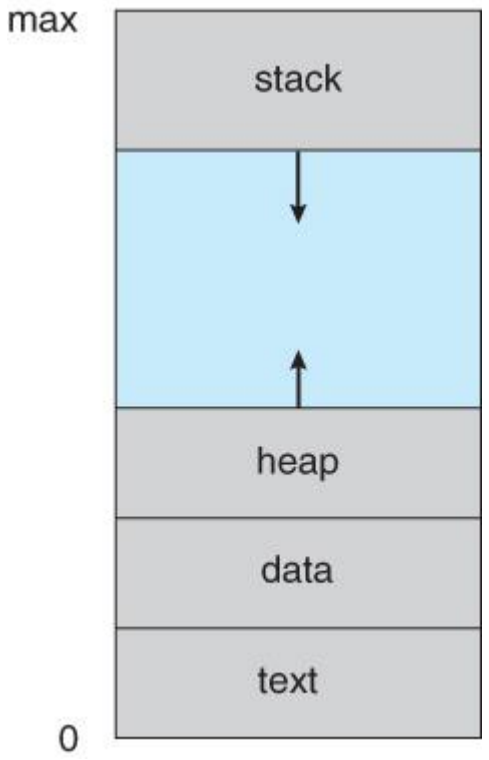
Process

- A process is basically a program in execution.
- The execution of a process must progress in a sequential fashion.
- A process is defined as an entity which represents the basic unit of work to be implemented in the system.
- To put it in simple terms, we write our computer programs in a text file and when we execute this program, it becomes a process which performs all the tasks mentioned in the program.
- When a program is loaded into the memory and it becomes a process, it can be divided into four sections — stack, heap, text and data.
- The image shows a simplified layout of a process inside main memory



Operating System Concepts – Process

S.N.	Component & Description
1	**Stack** The process Stack contains the temporary data such as method/function parameters, return address and local variables.
2	**Heap** This is dynamically allocated memory to a process during its run time.
3	**Text** This includes the current activity represented by the value of Program Counter and the contents of the processor's registers.
4	**Data** This section contains the global and static variables.

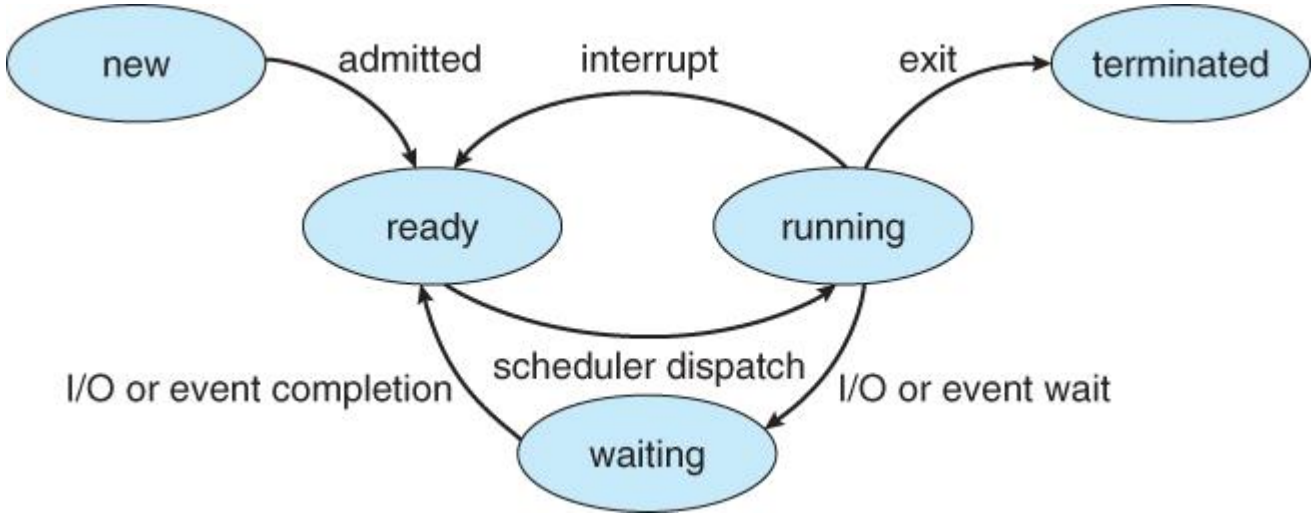


Process in Memory

Process Life Cycle or Process States

Processes may be in one of 5 states, as shown in image

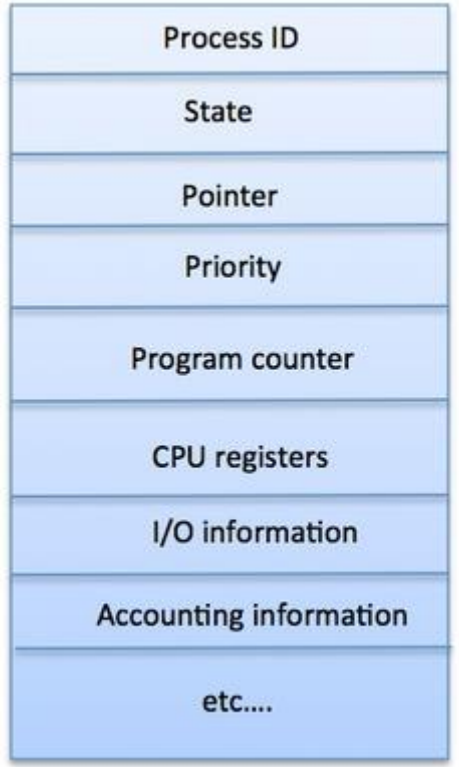
- **New** - The process is in the stage of being created.
- **Ready** - The process has all the resources available that it needs to run, but the CPU is not currently working on this process's instructions.
- **Running** - The CPU is working on this process's instructions.
- **Waiting** - The process cannot run at the moment, because it is waiting for some resource to become available or for some event to occur. For example the process may be waiting for keyboard input, disk access request, inter-process messages, a timer to go off, or a child process to finish.
- **Terminated** - The process has completed.



Process Control Block (PCB)

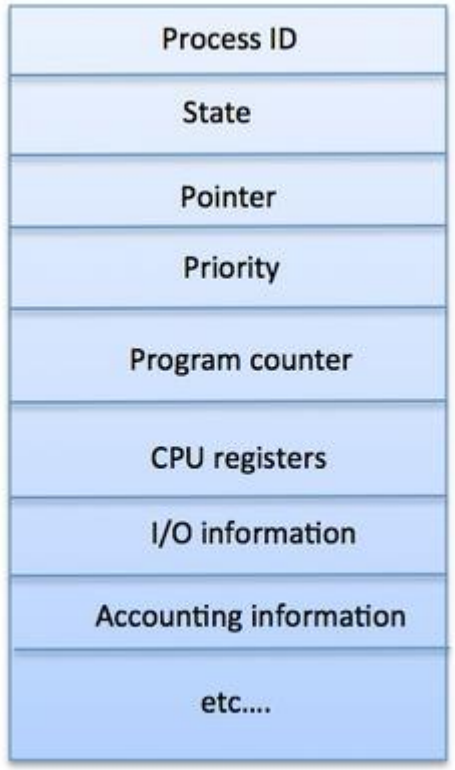
- A Process Control Block is a data structure maintained by the Operating System for every process.
- The PCB is identified by an integer process ID (PID).
- A PCB keeps all the information needed to keep track of a process as listed below in the table

S.N.	Information & Description
1	**Process State** The current state of the process i.e., whether it is ready, running, waiting, or whatever.
2	**Process privileges** This is required to allow/disallow access to system resources.
3	**Process ID** Unique identification for each of the process in the operating system.
4	**Pointer** A pointer to parent process.
5	**Program Counter** Program Counter is a pointer to the address of the next instruction to be executed for this process.
6	**CPU registers** Various CPU registers where process need to be stored for execution for running state.
7	**CPU Scheduling Information** Process priority and other scheduling information which is required to schedule the process.
8	**Memory management information** This includes the information of page table, memory limits, Segment table depending on memory used by the operating system.
9	**Accounting information** This includes the amount of CPU used for process execution, time limits, execution ID etc.
10	**IO status information** This includes a list of I/O devices allocated to the process.



Process Control Block (PCB)

- The architecture of a PCB is completely dependent on Operating System and may contain different information in different operating systems.
- Here is a simplified diagram of a PCB
- The PCB is maintained for a process throughout its lifetime, and is deleted once the process terminates.



Process Control Block (PCB)

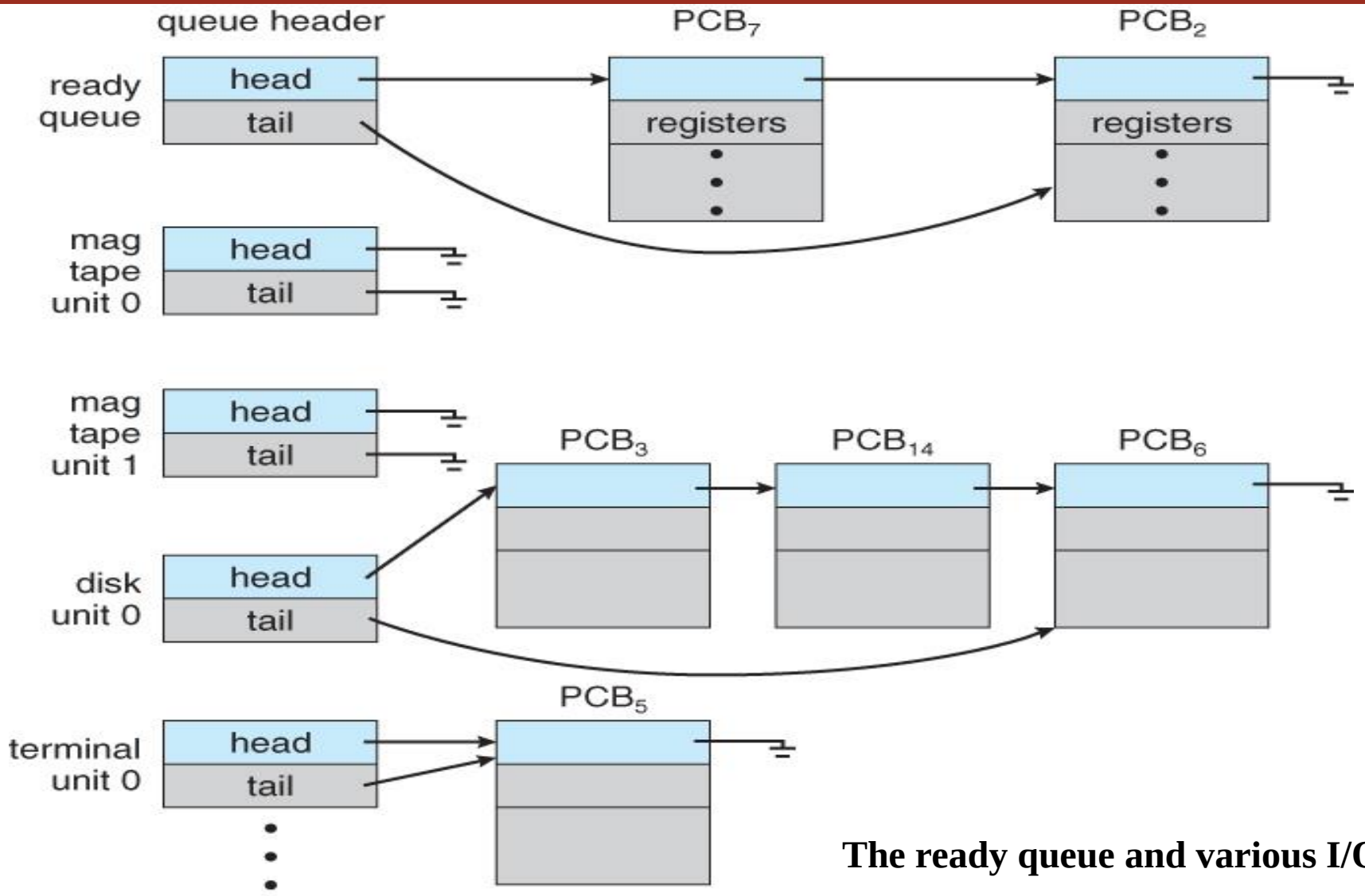
For each process there is a Process Control Block, PCB, which stores the following (types of) process-specific information, as illustrated in image (Specific details may vary from system to system.)

- **Process State** - Running, waiting, etc., as discussed in previous slide.
- **Process ID**, and parent process ID.
- **CPU registers and Program Counter** - These need to be saved and restored when swapping processes in and out of the CPU.
- **CPU-Scheduling information** - Such as priority information and pointers to scheduling queues.
- **Memory-Management information** - E.g. page tables or segment tables.
- **Accounting information** - user and kernel CPU time consumed, account numbers, limits, etc.
- **I/O Status information** - Devices allocated, open file tables, etc.

Process Scheduling

- The two main objectives of the process scheduling system are to keep the CPU busy at all times and to deliver "acceptable" response times for all programs, particularly for interactive ones.
- The process scheduler must meet these objectives by implementing suitable policies for swapping processes in and out of the CPU.
- (Note that these objectives can be conflicting. In particular, every time the system steps in to swap processes it takes up time on the CPU to do so, which is thereby "lost" from doing any useful productive work.
- **Scheduling Queues**
  - All processes are stored in the job queue.
  - Processes in the Ready state are placed in the ready queue.
  - Processes waiting for a device to become available or to deliver data are placed in device queues. There is generally a separate device queue for each device.
  - Other queues may also be created and used as needed.

Process Scheduling



The ready queue and various I/O device queues

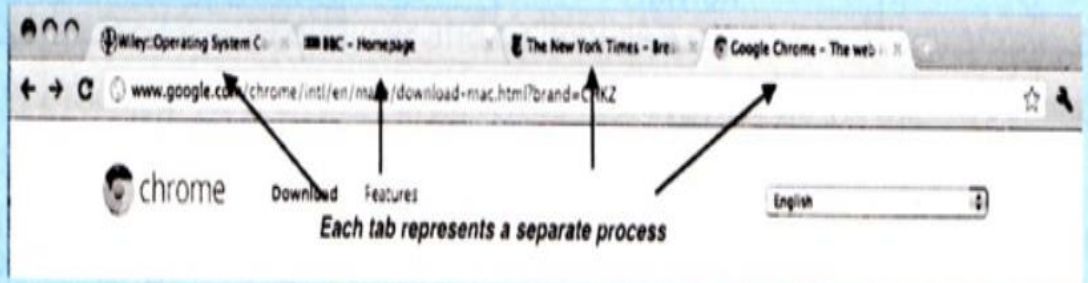
Inter-process Communication

- **Independent Processes** operating concurrently on a systems are those that can neither affect other processes or be affected by other processes.
- **Cooperating Processes** are those that can affect or be affected by other processes. There are several reasons why cooperating processes are allowed:
  - Information Sharing - There may be several processes which need access to the same file for example. (e.g. pipelines.)
  - Computation speedup - Often a solution to a problem can be solved faster if the problem can be broken down into sub-tasks to be solved simultaneously (particularly when multiple processors are involved.)
  - Modularity - The most efficient architecture may be to break a system down into cooperating modules. (E.g. databases with a client-server architecture.)
  - Convenience - Even a single user may be multi-tasking, such as editing, compiling, printing, and running the same code in different windows.

Inter-process Communication – MultiProcess Architecture

MULTIPROCESS ARCHITECTURE—CHROME BROWSER

Many websites contain active content such as JavaScript, Flash, and HTML5 to provide a rich and dynamic web-browsing experience. Unfortunately, these web applications may also contain software bugs, which can result in sluggish response times and can even cause the web browser to crash. This isn't a big problem in a web browser that displays content from only one website. But most contemporary web browsers provide tabbed browsing, which allows a single instance of a web browser application to open several websites at the same time, with each site in a separate tab. To switch between the different sites , a user need only click on the appropriate tab. This arrangement is illustrated below:



A problem with this approach is that if a web application in any tab crashes, the entire process—including all other tabs displaying additional websites —crashes as well.

Google's Chrome web browser was designed to address this issue by using a multiprocess architecture. Chrome identifies three different types of processes: browser, renderers, and plug-ins.

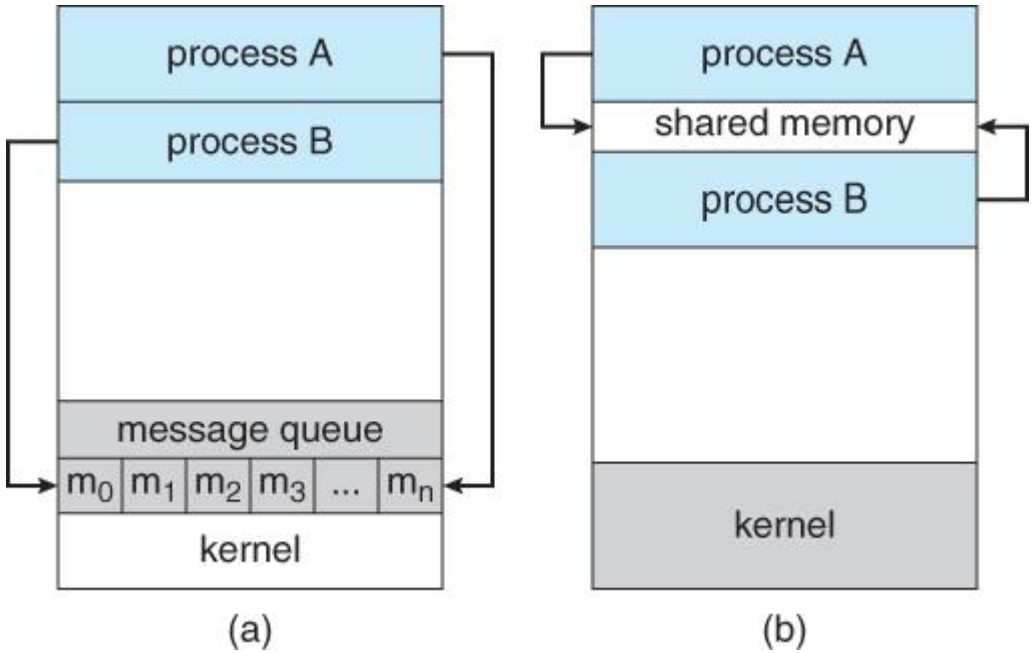
- The **browser** process is responsible for managing the user interface as well as disk and network I/O. A new browser process is created when Chrome is started. Only one browser process is created.
- **Renderer** processes contain logic for rendering web pages. Thus, they contain the logic for handling HTML, Javascript, images, and so forth. As a general rule, a new renderer process is created for each website opened in a new tab, and so several renderer processes may be active at the same time.
- A **plug-in** process is created for each type of plug-in (such as Flash or QuickTime) in use. Plug-in processes contain the code for the plug-in as well as additional code that enables the plug-in to communicate with associated renderer processes and the browser process.

Inter-process Communication – MultiProcess Architecture

- The advantage of the multiprocess approach is that websites run in isolation from one another.
- If one website crashes, only its renderer process is affected; all other processes remain unharmed.
- Furthermore, renderer processes run in a sandbox, which means that access to disk and network I/O is restricted, minimizing the effect of any security exploits.

Inter-process Communication

- Cooperating processes require some type of inter-process communication, which is most commonly one of two types:
  - Shared Memory systems or
  - Message Passing systems.
- Following Figure illustrates the difference between the two systems



Message Passing

Shared Memory

- Shared Memory is faster once it is set up, because no system calls are required and access occurs at normal memory speeds. However it is more complicated to set up, and doesn't work as well across multiple computers. Shared memory is generally preferable when large amounts of information must be shared quickly on the same computer.
- Message Passing requires system calls for every message transfer, and is therefore slower, but it is simpler to set up and works well across multiple computers. Message passing is generally preferable when the amount and/or frequency of data transfers is small, or when multiple computers are involved.

Inter-process Communication

Shared-Memory Systems

- In general the memory to be shared in a shared-memory system is initially within the address space of a particular process, which needs to make system calls in order to make that memory publicly available to one or more other processes.
- Other processes which wish to use the shared memory must then make their own system calls to attach the shared memory area onto their address space.
- Generally a few messages must be passed back and forth between the cooperating processes first in order to set up and coordinate the shared memory access.

Producer-Consumer Example Using Shared Memory

- This is a classic example, in which one process is producing data and another process is consuming the data. (In this example in the order in which it is produced, although that could vary.)
- The data is passed via an intermediary buffer, which may be either unbounded or bounded.
- With a bounded buffer the producer may have to wait until there is space available in the buffer, but with an unbounded buffer the producer will never need to wait.
- The consumer may need to wait in either case until there is data available.
- This example uses shared memory and a circular queue.
- Example in Java

Inter-process Communication

Message Passing Systems

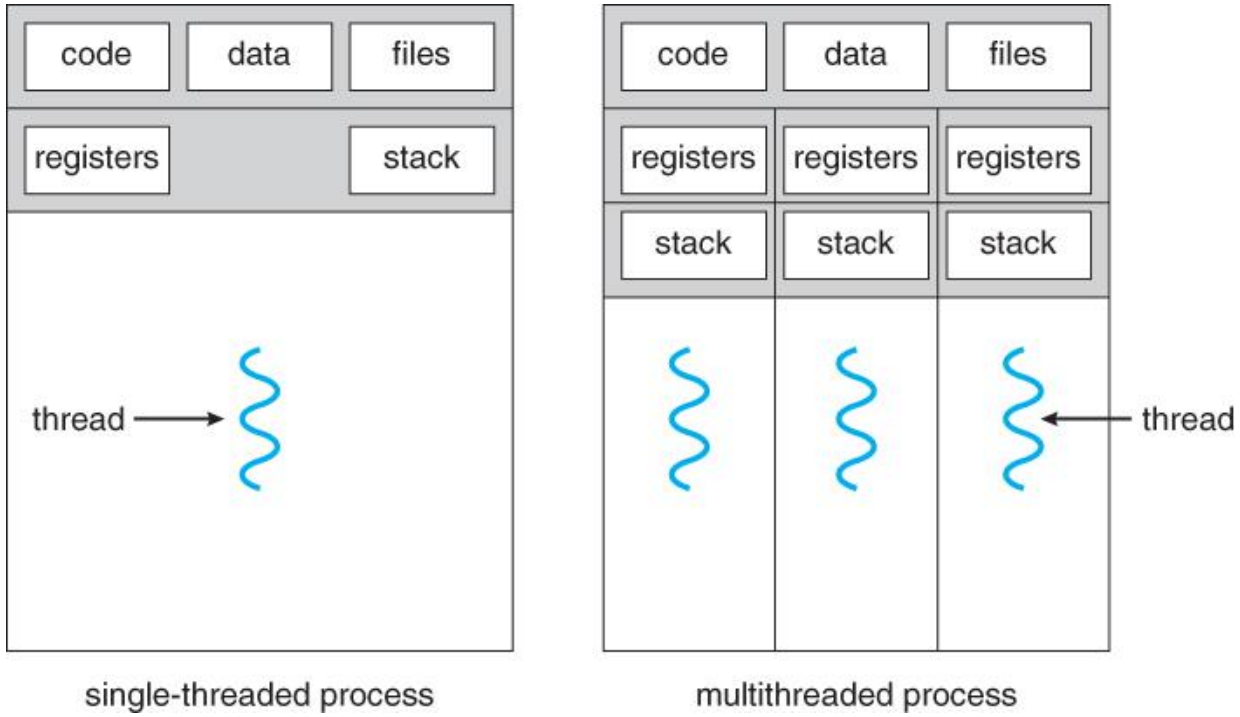
- Message passing systems must support at a minimum system calls for "send message" and "receive message".
- A communication link must be established between the cooperating processes before messages can be sent.
- There are three key issues to be resolved in message passing systems as further explored in the next three subsections:
  - Direct or indirect communication (naming)
  - Synchronous or asynchronous communication
  - Automatic or explicit buffering.

Naming

- With **direct communication** the sender must know the name of the receiver to which it wishes to send a message.
  - There is a one-to-one link between every sender-receiver pair.
  - For **symmetric** communication, the receiver must also know the specific name of the sender from which it wishes to receive messages.
    For **asymmetric** communications, this is not necessary.
- **Indirect communication** uses shared mailboxes, or ports.
  - Multiple processes can share the same mailbox or boxes.
  - Only one process can read any given message in a mailbox. Initially the process that creates the mailbox is the owner, and is the only one allowed to read mail in the mailbox, although this privilege may be transferred.
    - (Of course the process that reads the message can immediately turn around and place an identical message back in the box for someone else to read, but that may put it at the back end of a queue of messages.)
  - The OS must provide system calls to create and delete mailboxes, and to send and receive messages to/from mailboxes.
- Example in Java

Threads

- A ***thread*** is a basic unit of CPU utilization, consisting of a program counter, a stack, and a set of registers, (and a thread ID.)
- Traditional (heavyweight) processes have a single thread of control - There is one program counter, and one sequence of instructions that can be carried out at any given time.
- As shown in Figure, multi-threaded applications have multiple threads within a single process, each having their own program counter, stack and set of registers, but sharing common code, data, and certain structures such as open files.

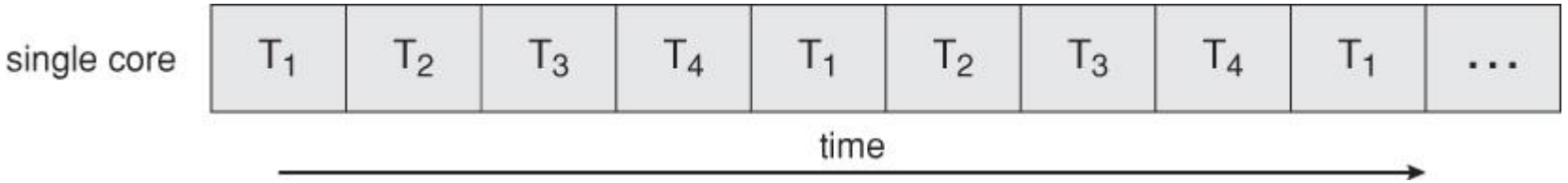


Threads

- Threads are very useful in modern programming whenever a process has multiple tasks to perform independently of the others.
- This is particularly true when one of the tasks may block, and it is desired to allow the other tasks to proceed without blocking.
- For example in a word processor, a background thread may check spelling and grammar while a foreground thread processes user input (keystrokes), while yet a third thread loads images from the hard drive, and a fourth does periodic automatic backups of the file being edited.
- Another example is a web server - Multiple threads allow for multiple requests to be satisfied simultaneously, without having to service requests sequentially or to fork off separate processes for every incoming request. (The latter is how this sort of thing was done before the concept of threads was developed.
- A daemon would listen at a port, fork off a child for every incoming request to be processed, and then go back to listening to the port.)

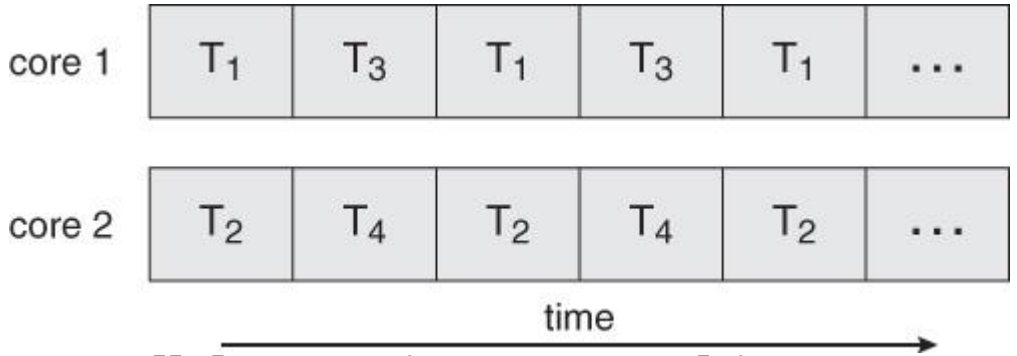
Threads – Multicore Programming

- A recent trend in computer architecture is to produce chips with multiple ***cores***, or CPUs on a single chip.
- A multi-threaded application running on a traditional single-core chip would have to interleave the threads, as shown in Figure.



Concurrent execution on a single core system

- On a multi-core chip, however, the threads could be spread across the available cores, allowing true parallel processing, as shown in Figure



Parallel execution on a multicore system

Threads – Multicore Programming

- For operating systems, multi-core chips require new scheduling algorithms to make better use of the multiple cores available.
- As multi-threading becomes more pervasive and more important (thousands instead of tens of threads), CPUs have been developed to support more simultaneous threads per core in hardware.

Programming Challenges (New section, same content ?)

- For application programmers, there are five areas where multi-core chips present new challenges:
- Identifying tasks - Examining applications to find activities that can be performed concurrently.
- Balance - Finding tasks to run concurrently that provide equal value. I.e. don't waste a thread on trivial tasks.
- Data splitting - To prevent the threads from interfering with one another.
- Data dependency - If one task is dependent upon the results of another, then the tasks need to be synchronized to assure access in the proper order.
- Testing and debugging - Inherently more difficult in parallel processing situations, as the race conditions become much more complex and difficult to identify.

Threads – Multicore Programming

Types of Parallelism

- In theory there are two different ways to parallelize the workload:

- Data parallelism divides the data up amongst multiple cores (threads), and performs the same task on each subset of the data. For example dividing a large image up into pieces and performing the same digital image processing on each piece on different cores.
- Task parallelism divides the different tasks to be performed among the different cores and performs them simultaneously.
- In practice no program is ever divided up solely by one or the other of these, but instead by some sort of hybrid combination.

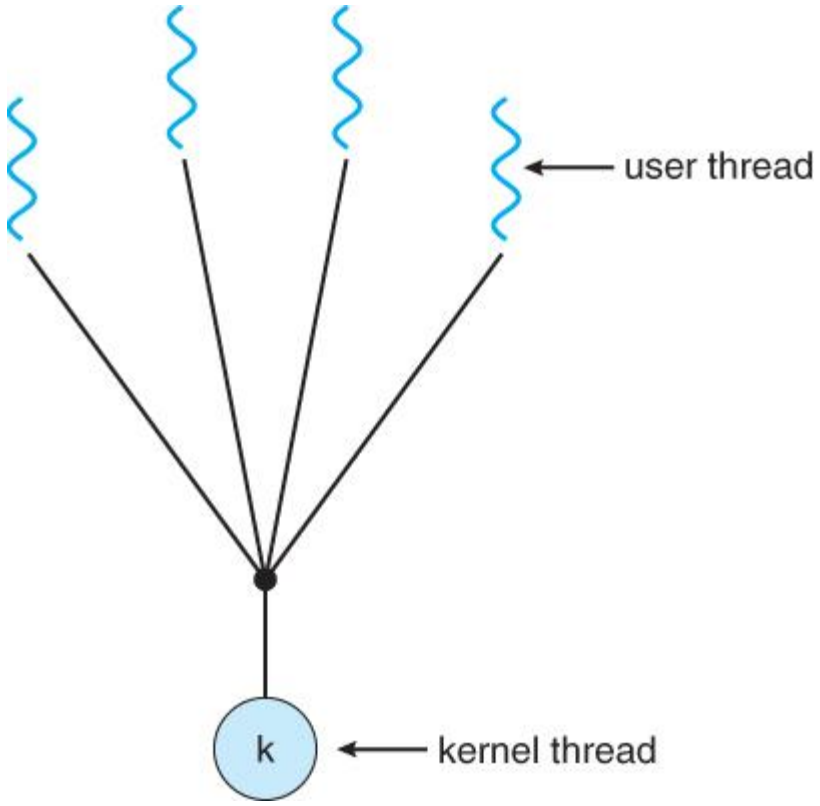
Threads – Multithreading Models

- There are two types of threads to be managed in a modern system:
  - User threads and
  - kernel threads.
- User threads are supported above the kernel, without kernel support. These are the threads that application programmers would put into their programs.
- Kernel threads are supported within the kernel of the OS itself. All modern OSes support kernel level threads, allowing the kernel to perform multiple simultaneous tasks and/or to service multiple kernel system calls simultaneously.
- In a specific implementation, the user threads must be mapped to kernel threads, using one of the following strategies.
  - Many to One Model
  - One to One Model
  - Many to Many Model

Threads – Multithreading Models

Many-to-one Model

- In the many-to-one model, many user-level threads are all mapped onto a single kernel thread.
- Thread management is handled by the thread library in user space, which is very efficient.
- However, if a blocking system call is made, then the entire process blocks, even if the other user threads would otherwise be able to continue.
- Because a single kernel thread can operate only on a single CPU, the many-to-one model does not allow individual processes to be split across multiple CPUs.
- Green threads for Solaris and GNU Portable Threads implement the many-to-one model in the past, but few systems continue to do so today.

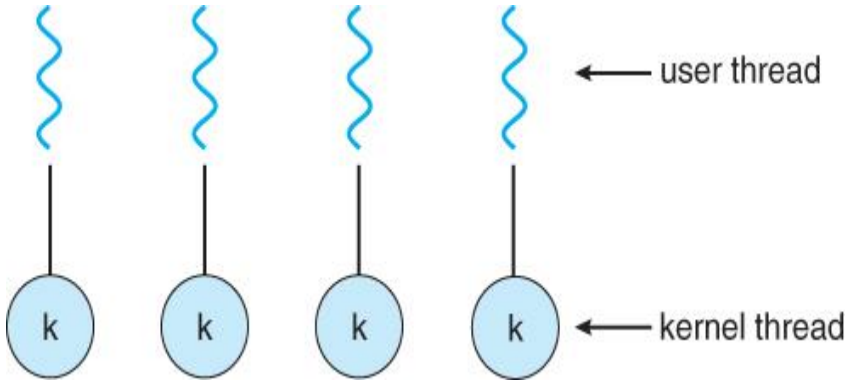


Many-to-one Model

Threads – Multithreading Models

One-To-One Model

- The one-to-one model creates a separate kernel thread to handle each user thread.
- One-to-one model overcomes the problems listed above involving blocking system calls and the splitting of processes across multiple CPUs.
- However the overhead of managing the one-to-one model is more significant, involving more overhead and slowing down the system.
- Most implementations of this model place a limit on how many threads can be created.
- Linux and Windows from 95 to XP implement the one-to-one model for threads.

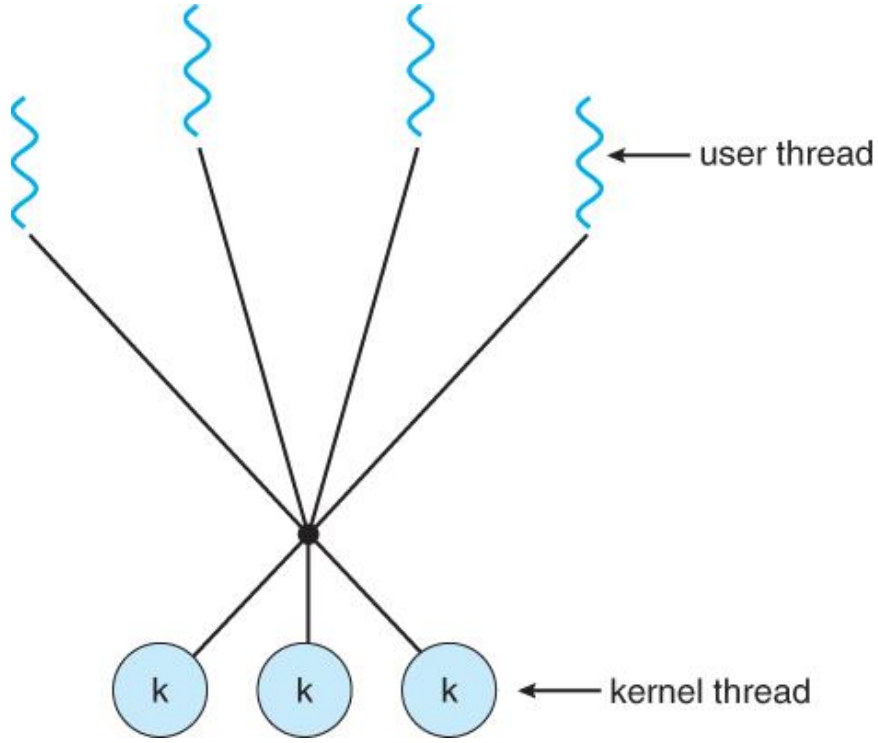


One-to-one Model

Threads – Multithreading Models

Many-to-Many Model

- The many-to-many model multiplexes any number of user threads onto an equal or smaller number of kernel threads, combining the best features of the one-to-one and many-to-one models.
- Users have no restrictions on the number of threads created.
- Blocking kernel system calls do not block the entire process.
- Processes can be split across multiple processors.
- Individual processes may be allocated variable numbers of kernel threads, depending on the number of CPUs present and other factors.



Many-to-Many Model

Thread Libraries

- Thread libraries provide programmers with an API for creating and managing threads.
- Thread libraries may be implemented either in user space or in kernel space. The former involves API functions implemented solely within user space, with no kernel support. The latter involves system calls, and requires a kernel with thread library support.
- There are three main thread libraries in use today:
  1. POSIX Pthreads - may be provided as either a user or kernel library, as an extension to the POSIX standard.
  2. Win32 threads - provided as a kernel-level library on Windows systems.
  3. Java threads - Since Java generally runs on a Java Virtual Machine, the implementation of threads is based upon whatever OS and hardware the JVM is running on, i.e. either Pthreads or Win32 threads depending on the system.

Thread Libraries

Pthreads

- The POSIX standard (IEEE 1003.1c) defines the *specification* for pThreads, not the *implementation*.
- pThreads are available on Solaris, Linux, Mac OSX, Tru64, and via public domain shareware for Windows.
- Global variables are shared amongst all threads.
- One thread can wait for the others to rejoin before continuing.
- pThreads begin execution in a specified function, in this example the runner( ) function:

Thread Libraries

Multithreaded C Program using Pthreads API

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

int sum; /* this data is shared by the thread(s) */
void *runner(void *param); /* the thread */

int main(int argc, char *argv[])
{
   pthread_t tid; /* the thread identifier */
   pthread_attr_t attr; /* set of thread attributes */

   if (argc != 2) {
      fprintf(stderr,"usage: a.out <integer value>\n");
      return -1;
   }
   if (atoi(argv[1]) < 0) {
      fprintf(stderr,"%d must be >= 0\n",atoi(argv[1]));
      return -1;
   }

   /* get the default attributes */
   pthread_attr_init(&attr);
   /* create the thread */
   pthread_create(&tid,&attr,runner,argv[1]);
   /* wait for the thread to exit */
   pthread_join(tid,NULL);

   printf("sum = %d\n",sum);
}

/* The thread will begin control in this function */
void *runner(void *param)
{
   int i, upper = atoi(param);
   sum = 0;

   for (i = 1; i <= upper; i++)
      sum += i;

   pthread_exit(0);
}
```

Thread Libraries

Windows Thread

- Similar to pThreads. Examine the code example to see the differences, which are mostly syntactic & nomenclature:

Multithreaded C Program using Win32 API

```
#include <windows.h>
#include <stdio.h>
DWORD Sum; /* data is shared by the thread(s) */
/* the thread runs in this separate function */

DWORD WINAPI Summation(LPVOID Param)
{
   DWORD Upper = *(DWORD*)Param;
   for (DWORD i = 0; i <= Upper; i++)
      Sum += i;
   return 0;
}

int main(int argc, char *argv[])
{
   DWORD ThreadId;
   HANDLE ThreadHandle;
   int Param;
   /* perform some basic error checking */
   if (argc != 2) {
      fprintf(stderr,"An integer parameter is required\n");
      return -1;
   }
   Param = atoi(argv[1]);
   if (Param < 0) {
      fprintf(stderr,"An integer >= 0 is required\n");
      return -1;
   }

   // create the thread
   ThreadHandle = CreateThread(
      NULL, // default security attributes
      0, // default stack size
      Summation, // thread function
      &Param, // parameter to thread function
      0, // default creation flags
      &ThreadId); // returns the thread identifier

   if (ThreadHandle != NULL) {
      // now wait for the thread to finish
      WaitForSingleObject(ThreadHandle,INFINITE);

      // close the thread handle
      CloseHandle(ThreadHandle);

      printf("sum = %d\n",Sum);
   }
}
```

Thread Libraries

Java Threads

- ALL Java programs use Threads - even "common" single-threaded ones.
- The creation of new Threads requires Objects that implement the Runnable Interface, which means they contain a method "public void run( )" . Any descendant of the Thread class will naturally contain such a method. (In practice the run( ) method must be overridden / provided for the thread to have any practical functionality.)
- Creating a Thread Object does not start the thread running - To do that the program must call the Thread's "start( )" method. Start( ) allocates and initializes memory for the Thread, and then calls the run( ) method. (Programmers do not call run( ) directly.)
- Because Java does not support global variables, Threads must be passed a reference to a shared Object in order to share data, in this example the "Sum" Object.
- Note that the JVM runs on top of a native OS, and that the JVM specification does not specify what model to use for mapping Java threads to kernel threads. This decision is JVM implementation dependant, and may be one-to-one, many-to-many, or many to one.. (On a UNIX system the JVM normally uses PThreads and on a Windows system it normally uses windows threads.)

Thread Libraries – Java Threads

```
class Sum
{
  private int sum;

  public int getSum() {
    return sum;
  }

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

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

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

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

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

Java Program for summation of Non-negative integer

Threading Issues

The fork( ) and exec( ) System Calls

- Q: If one thread forks, is the entire process copied, or is the new process single-threaded?
- A: System dependant.
- A: If the new process execs right away, there is no need to copy all the other threads. If it doesn't, then the entire process should be copied.
- A: Many versions of UNIX provide multiple versions of the fork call for this purpose.

Threading Issues

Signal Handling

- Q: When a multi-threaded process receives a signal, to what thread should that signal be delivered?
- A: There are four major options:
  - 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 in a process.
- The best choice may depend on which specific signal is involved.
- UNIX allows individual threads to indicate which signals they are accepting and which they are ignoring. However the signal can only be delivered to one thread, which is generally the first thread that is accepting that particular signal.
- UNIX provides two separate system calls, **kill(pid, signal)** and **pthread_kill(tid, signal)**, for delivering signals to processes or specific threads respectively.
- Windows does not support signals, but they can be emulated using Asynchronous Procedure Calls (APCs). APCs are delivered to specific threads, not processes.

Threading Issues

Thread Cancellation

- Threads that are no longer needed may be cancelled by another thread in one of two ways:
  - **Asynchronous Cancellation** cancels the thread immediately.
  - **Deferred Cancellation** sets a flag indicating the thread should cancel itself when it is convenient. It is then up to the cancelled thread to check this flag periodically and exit nicely when it sees the flag set.
- (Shared) resource allocation and inter-thread data transfers can be problematic with asynchronous cancellation.

Scheduling – CPU Scheduling Overview

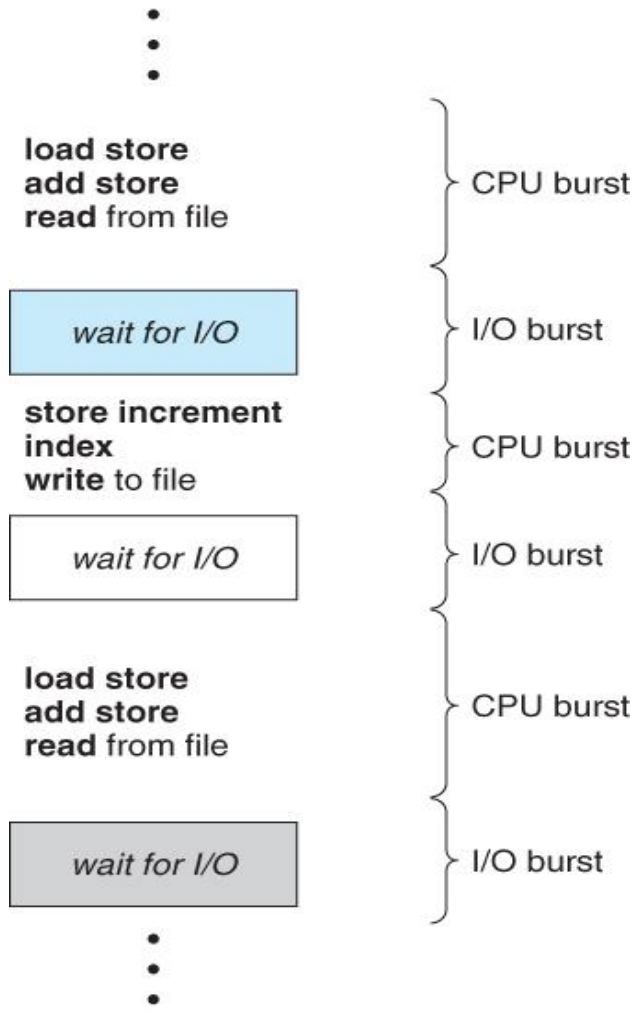
Basic Concepts

- Almost all programs have some alternating cycle of CPU number crunching and waiting for I/O of some kind. (Even a simple fetch from memory takes a long time relative to CPU speeds.)
- In a simple system running a single process, the time spent waiting for I/O is wasted, and those CPU cycles are lost forever.
- A scheduling system allows one process to use the CPU while another is waiting for I/O, thereby making full use of otherwise lost CPU cycles.
- The challenge is to make the overall system as "efficient" and "fair" as possible, subject to varying and often dynamic conditions, and where "efficient" and "fair" are somewhat subjective terms, often subject to shifting priority policies.

Scheduling – CPU Scheduling Overview

CPU-I/O Burst Cycle

- Almost all processes alternate between two states in a continuing ***cycle***, as shown in Figure
  - A CPU burst of performing calculations, and
  - An I/O burst, waiting for data transfer in or out of the system.

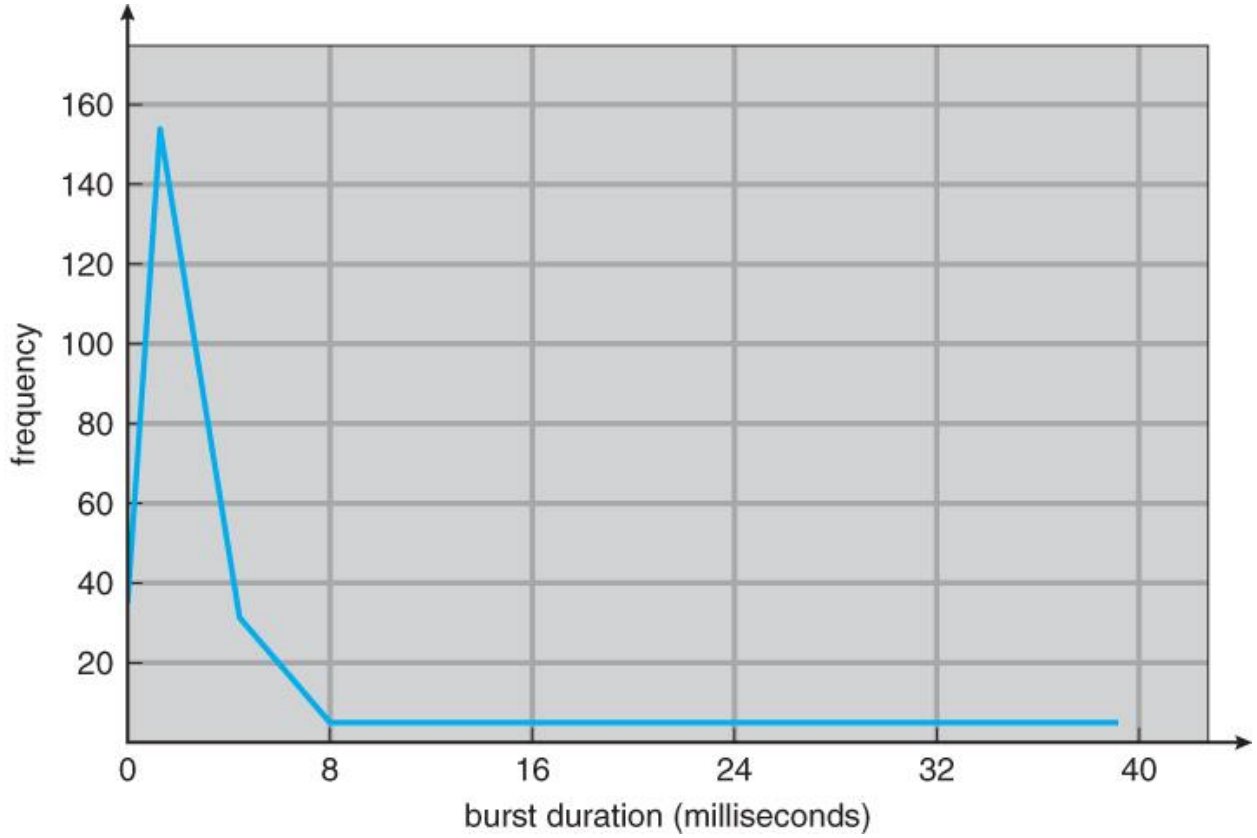


Alternative Sequence of CPU and I/O Burst

Scheduling – CPU Scheduling Overview

CPU-I/O Burst Cycle

- CPU bursts vary from process to process, and from program to program, but an extensive study shows frequency patterns similar to that shown in Figure



Histogram of CPU-burst durations

Scheduling – CPU Scheduling Overview

CPU Scheduler

- Whenever the CPU becomes idle, it is the job of the CPU Scheduler (a.k.a. the short-term scheduler) to select another process from the ready queue to run next.
- The storage structure for the ready queue and the algorithm used to select the next process are not necessarily a FIFO queue.
- There are several alternatives to choose from, as well as numerous adjustable parameters for each algorithm.

Scheduling – CPU Scheduling Overview

Pre-emptive Scheduling

- CPU scheduling decisions take place under one of four conditions:
  1. When a process switches from the running state to the waiting state, such as for an I/O request or invocation of the wait( ) system call.
  2. When a process switches from the running state to the ready state, for example in response to an interrupt.
  3. When a process switches from the waiting state to the ready state, say at completion of I/O or a return from wait( ).
  4. When a process terminates.
- For conditions 1 and 4 there is no choice - A new process must be selected.
- For conditions 2 and 3 there is a choice - To either continue running the current process, or select a different one.
- If scheduling takes place only under conditions 1 and 4, the system is said to be ***non-preemptive***, or ***cooperative***. Under these conditions, once a process starts running it keeps running, until it either voluntarily blocks or until it finishes. Otherwise the system is said to be ***preemptive.***
- Windows used non-preemptive scheduling up to Windows 3.x, and started using pre-emptive scheduling with Win95. Macs used non-preemptive prior to OSX, and pre-emptive since then. Note that pre-emptive scheduling is only possible on hardware that supports a timer interrupt.

Scheduling – Algorithms

- The following are scheduling strategies, looking at only a single CPU burst each for a small number of processes.
- Obviously, real systems have to deal with a lot more simultaneous processes executing their CPU-I/O burst cycles.

  - First-Come First-Serve Scheduling, FCFS
  - Shortest-Job-First Scheduling, SJF
  - Priority Scheduling
  - Round Robin Scheduling
  - Multilevel Queue Scheduling
  - Multilevel Feedback-Queue Scheduling

Scheduling – Algorithms

First-Come First-Serve Scheduling, FCFS

- FCFS is very simple - Just a FIFO queue, like customers waiting in line at the bank or the post office or at a copying machine.
- Unfortunately, however, FCFS can yield some very long average wait times, particularly if the first process to get there takes a long time. For example, consider the following three processes.

Process	Burst Time
P1	24
P2	3
P3	3

In this Gantt chart, process P1 arrives first.
The average waiting time for the three processes is
(0 + 24 + 27) / 3 = 17.0 ms.

In the following Gantt chart below, the same three processes have an average wait time of (0 + 3 + 6) / 3 = 3.0 ms.
The total run time for the three bursts is the same, but in the second case two of the three finish much quicker, and the other process is only delayed by a short amount.

Scheduling – Algorithms

First-Come First-Serve Scheduling, FCFS (Continue…)

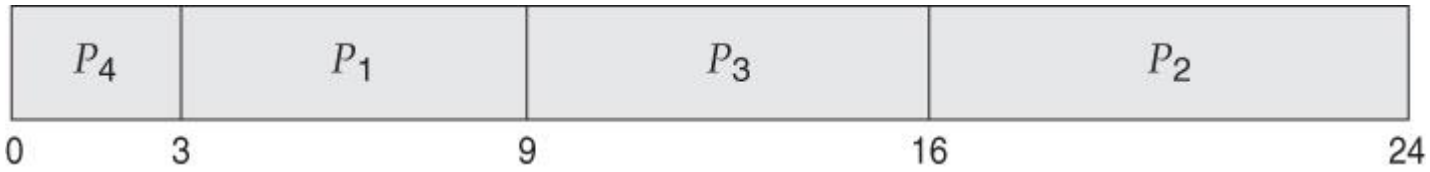
- FCFS can also block the system in a busy dynamic system in another way, known as the ***convoy effect***.
- When one CPU intensive process blocks the CPU, a number of I/O intensive processes can get backed up behind it, leaving the I/O devices idle.
- When the CPU hog finally relinquishes the CPU, then the I/O processes pass through the CPU quickly, leaving the CPU idle while everyone queues up for I/O, and then the cycle repeats itself when the CPU intensive process gets back to the ready queue.

Scheduling – Algorithms – Shorted Job First

Shortest Job First Scheduling SJF

- The idea behind the SJF algorithm is to pick the quickest fastest little job that needs to be done, get it out of the way first, and then pick the next smallest fastest job to do next.
- (Technically this algorithm picks a process based on the next shortest **CPU burst**, not the overall process time.)
- For example, the Gantt chart below is based upon the following CPU burst times, (and the assumption that all jobs arrive at the same time.)

Process	Burst Time
P1	6
P2	8
P3	7
P4	3



- In the case above the average wait time is (0 + 3 + 9 + 16) / 4 = 7.0 ms,

Scheduling – Algorithms

Shortest Job First Scheduling SJF (Continue…)

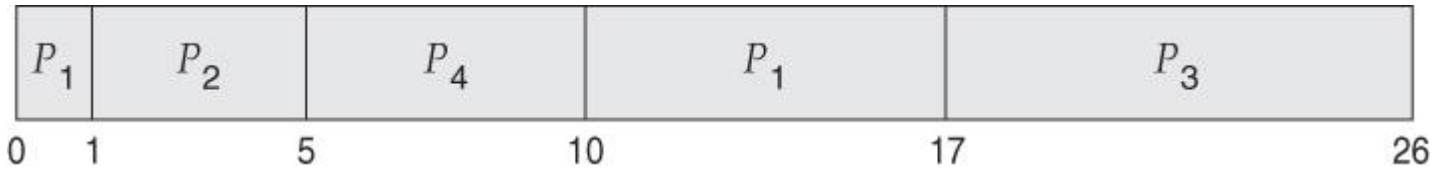
- SJF can be proven to be the fastest scheduling algorithm, but it suffers from one important problem: How do you know how long the next CPU burst is going to be
  - For long-term batch jobs this can be done based upon the limits that users set for their jobs when they submit them, which encourages them to set low limits, but risks their having to re-submit the job if they set the limit too low. However that does not work for short-term CPU scheduling on an interactive system.
  - Another option would be to statistically measure the run time characteristics of jobs, particularly if the same tasks are run repeatedly and predictably. But once again that really isn't a viable option for short term CPU scheduling in the real world.
  - A more practical approach is to ***predict*** the length of the next burst, based on some historical measurement of recent burst times for this process. One simple, fast, and relatively accurate method is the ***exponential average***, which can be defined as follows. (The book uses tau and t for their variables, but those are hard to distinguish from one another and don't work well in HTML.)

Scheduling – Algorithms

•SJF can be either preemptive or non-preemptive. Preemption occurs when a new process arrives in the ready queue that has a predicted burst time shorter than the time remaining in the process whose burst is currently on the CPU. Preemptive SJF is sometimes referred to as ***shortest remaining time first scheduling.***

•For example, the following Gantt chart is based upon the following data:

Process	Arrival Time	Burst Time
P1	0	8
P2	1	4
P3	2	9
p4	3	5



•The average wait time in this case is ((5 - 3) + (10 - 1) + (17 - 2)) / 4 = 26 / 4 = 6.5 ms.

Scheduling – Algorithms – Priority Scheduling

Priority Scheduling

- Priority scheduling is a more general case of SJF, in which each job is assigned a priority and the job with the highest priority gets scheduled first. (SJF uses the inverse of the next expected burst time as its priority - The smaller the expected burst, the higher the priority.)
- Note that in practice, priorities are implemented using integers within a fixed range, but there is no agreed-upon convention as to whether "high" priorities use large numbers or small numbers.
- For example, the following Gantt chart is based upon these process burst times and priorities, and yields an average waiting time of 8.2 ms:

Process	Burst Time	Priority
P1	10	3
P2	1	1
P3	2	4
P4	1	5
P5	5	2

Scheduling – Algorithms – Priority Scheduling

- Priorities can be assigned either internally or externally. Internal priorities are assigned by the OS using criteria such as average burst time, ratio of CPU to I/O activity, system resource use, and other factors available to the kernel. External priorities are assigned by users, based on the importance of the job, fees paid, politics, etc.
- Priority scheduling can be either preemptive or non-preemptive.
- Priority scheduling can suffer from a major problem known as ***indefinite blocking***, or ***starvation***, in which a low-priority task can wait forever because there are always some other jobs around that have higher priority.
  - If this problem is allowed to occur, then processes will either run eventually when the system load lightens (at say 2:00 a.m.), or will eventually get lost when the system is shut down or crashes. (There are rumors of jobs that have been stuck for years.)
  - One common solution to this problem is ***aging***, in which priorities of jobs increase the longer they wait. Under this scheme a low-priority job will eventually get its priority raised high enough that it gets run.

Scheduling – Algorithms – Round Robin Scheduling

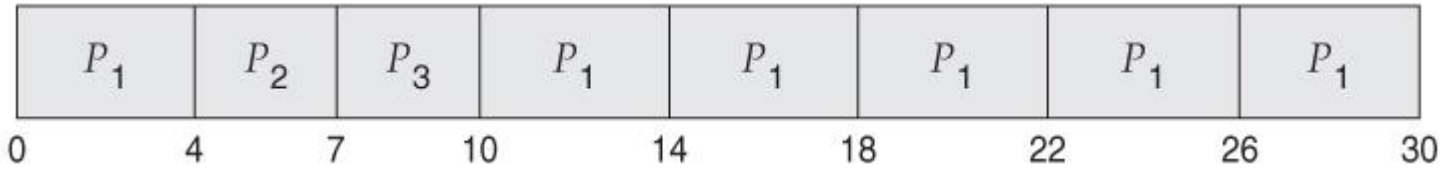
- Round robin scheduling is similar to FCFS scheduling, except that CPU bursts are assigned with limits called ***time quantum***.
- When a process is given the CPU, a timer is set for whatever value has been set for a time quantum
  - If the process finishes its burst before the time quantum timer expires, then it is swapped out of the CPU just like the normal FCFS algorithm.
  - If the timer goes off first, then the process is swapped out of the CPU and moved to the back end of the ready queue.
- The ready queue is maintained as a circular queue, so when all processes have had a turn, then the scheduler gives the first process another turn, and so on.

Scheduling – Algorithms – Round Robin Scheduling

- RR scheduling can give the effect of all processors sharing the CPU equally, although the average wait time can be longer than with other scheduling algorithms.
- In the following example the average wait time is 5.66 ms.

Process	Burst Time
P1	24
P2	3
P3	3

Here, Quantum = 4



Average Waiting Time

(4-0) + (7-0) + (10-4) = 4 + 7 + 6 = 17 / 3 = 5.66 ms

P2 + P3 + P1

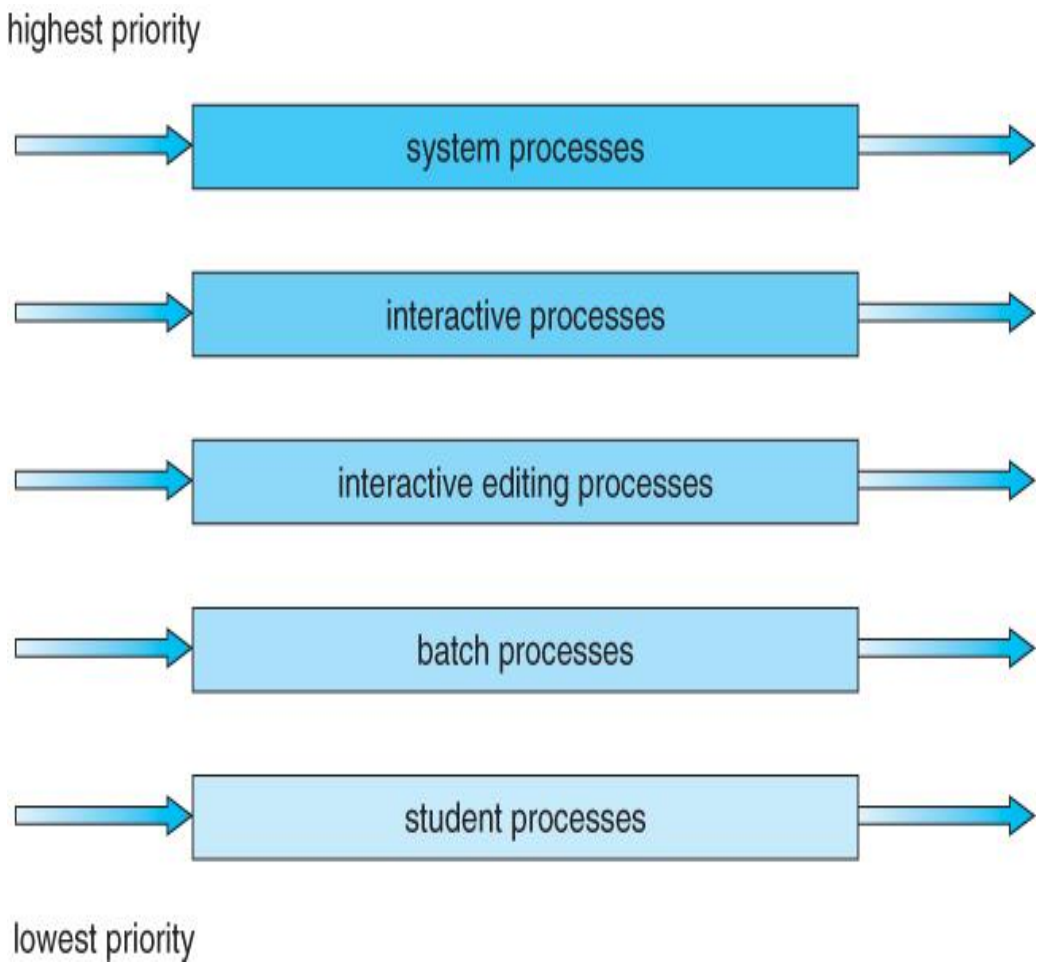
Scheduling – Algorithms – Round Robin Scheduling

- The performance of RR is sensitive to the time quantum selected.
- If the quantum is large enough, then RR reduces to the FCFS algorithm; If it is very small, then each process gets 1/nth of the processor time and share the CPU equally.
- **BUT**, a real system invokes overhead for every context switch, and the smaller the time quantum the more context switches there are. Most modern systems use time quantum between 10 and 100 milliseconds, and context switch times on the order of 10 microseconds, so the overhead is small relative to the time quantum.

- Make the Quantum of 10 in previous example and Calculate the Average Waiting Time

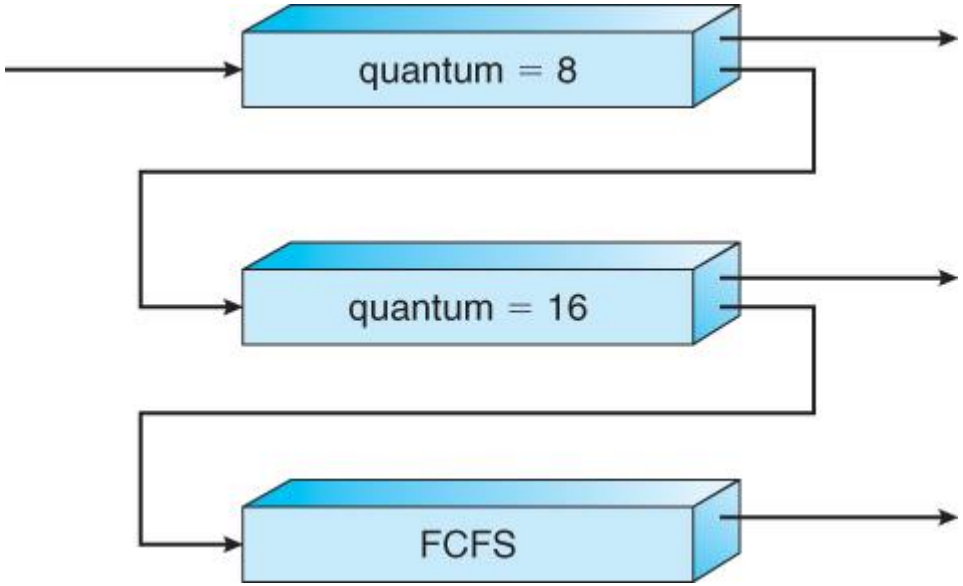
Scheduling – Algorithms – Multi-Level Queue Scheduling

- When processes can be readily categorized, then multiple separate queues can be established, each implementing whatever scheduling algorithm is most appropriate for that type of job, and/or with different parametric adjustments.
- Scheduling must also be done between queues, that is scheduling one queue to get time relative to other queues.
- Two common options are strict priority (no job in a lower priority queue runs until all higher priority queues are empty) and round-robin (each queue gets a time slice in turn, possibly of different sizes.)
- Note that under this algorithm jobs cannot switch from queue to queue - Once they are assigned a queue, that is their queue until they finish.



Scheduling – Algorithms – Multi-Level Feeback Queue Scheduling

- Multilevel feedback queue scheduling is similar to the ordinary multilevel queue scheduling described above, except jobs may be moved from one queue to another for a variety of reasons:
  - If the characteristics of a job change between CPU-intensive and I/O intensive, then it may be appropriate to switch a job from one queue to another.
  - Aging can also be incorporated, so that a job that has waited for a long time can get bumped up into a higher priority queue for a while.
- Multilevel feedback queue scheduling is the most flexible, because it can be tuned for any situation. But it is also the most complex to implement because of all the adjustable parameters.
- Some of the parameters which define one of these systems include:
  - The number of queues.
  - The scheduling algorithm for each queue.
  - The methods used to upgrade or demote processes from one queue to another. (Which may be different.)
  - The method used to determine which queue a process enters initially.



Thread Scheduling

- The process scheduler schedules only the kernel threads.
- User threads are mapped to kernel threads by the thread library - The OS (and in particular the scheduler) is unaware of them.
- **Contention Scope**
  - ***Contention scope*** refers to the scope in which threads compete for the use of physical CPUs.
  - On systems implementing many-to-one and many-to-many threads, ***Process Contention Scope, PCS,*** occurs, because competition occurs between threads that are part of the same process. (This is the management / scheduling of multiple user threads on a single kernel thread, and is managed by the thread library.)
  - ***System Contention Scope, SCS,*** involves the system scheduler scheduling kernel threads to run on one or more CPUs. Systems implementing one-to-one threads (XP, Solaris 9, Linux), use only SCS.
  - PCS scheduling is typically done with priority, where the programmer can set and/or change the priority of threads created by his or her programs. Even time slicing is not guaranteed among threads of equal priority.
- **Pthread Scheduling**
- The Pthread library provides for specifying scope contention:
  - PTHREAD_SCOPE_PROCESS schedules threads using PCS, by scheduling user threads onto available LWPs using the many-to-many model.
  - PTHREAD_SCOPE_SYSTEM schedules threads using SCS, by binding user threads to particular LWPs, effectively implementing a one-to-one model.
- getscope and setscope methods provide for determining and setting the scope contention respectively

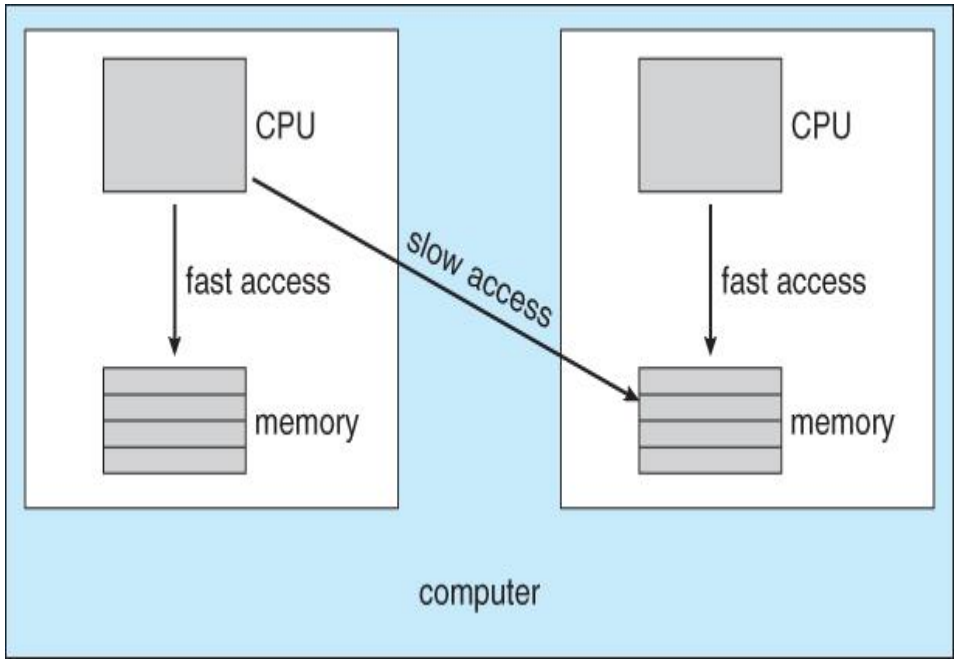
Multi-Processor Scheduling

- When multiple processors are available, then the scheduling gets more complicated, because now there is more than one CPU which must be kept busy and in effective use at all times.
- ***Load sharing*** revolves around balancing the load between multiple processors.
- Multi-processor systems may be **heterogeneous,** (different kinds of CPUs), or **homogenous,** (all the same kind of CPU). Even in the latter case there may be special scheduling constraints, such as devices which are connected via a private bus to only one of the CPUs. This book will restrict its discussion to homogenous systems.
- **Approaches to Multiple-Processor Scheduling**
  - One approach to multi-processor scheduling is ***asymmetric multiprocessing,*** in which one processor is the master, controlling all activities and running all kernel code, while the other runs only user code. This approach is relatively simple, as there is no need to share critical system data.
  - Another approach is ***symmetric multiprocessing, SMP,*** where each processor schedules its own jobs, either from a common ready queue or from separate ready queues for each processor.
  - Virtually all modern OSes support SMP, including XP, Win 2000, Solaris, Linux, and Mac OSX.
- **Processor Affinity**
  - Processors contain cache memory, which speeds up repeated accesses to the same memory locations.
  - If a process were to switch from one processor to another each time it got a time slice, the data in the cache (for that process) would have to be invalidated and re-loaded from main memory, thereby obviating the benefit of the cache.
  - process has an affinity for a particular CPU, then it should preferentially be assigned memory storage in "local" fast access areas.

Multi-Processor Scheduling

Load Balancing

- Obviously an important goal in a multiprocessor system is to balance the load between processors, so that one processor won't be sitting idle while another is overloaded.
- Systems using a common ready queue are naturally self-balancing, and do not need any special handling. Most systems, however, maintain separate ready queues for each processor.
- Balancing can be achieved through either ***push migration*** or ***pull migration:***
  - ***Push migration*** involves a separate process that runs periodically, (e.g. every 200 milliseconds), and moves processes from heavily loaded processors onto less loaded ones.
  - ***Pull migration*** involves idle processors taking processes from the ready queues of other processors.
  - Push and pull migration are not mutually exclusive.
- Note that moving processes from processor to processor to achieve load balancing works against the principle of processor affinity, and if not carefully managed, the savings gained by balancing the system can be lost in rebuilding caches. One option is to only allow migration when imbalance surpasses a given threshold.



Process Synchronization

- Recall that back in we looked at cooperating processes (those that can effect or be effected by other simultaneously running processes), and as an example, we used the producer-consumer cooperating processes

Producer Process:

```
while (true)
{
     /* produce an item in nextProduced */
     while (counter == BUFFER_SIZE)
        ; /* do nothing */
     buffer[in] = nextProduced;
     in = (in + 1) % BUFFER_SIZE;
     counter++;
}
```

- Unfortunately we have now introduced a new problem, because both the producer and the consumer are adjusting the value of the variable counter, which can lead to a condition known as a ***race condition***.
- In this condition a piece of code may or may not work correctly, depending on which of two simultaneous processes executes first, and more importantly if one of the processes gets interrupted such that the other process runs between important steps of the first process

Consumer Process:

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

Process Synchronization

- The particular problem above comes from the producer executing "counter++" at the same time the consumer is executing "counter--".
- If one process gets part way through making the update and then the other process butts in, the value of counter can get left in an incorrect state.
- Note that race conditions are ***notoriously difficult*** to identify and debug, because by their very nature they only occur on rare occasions, and only when the timing is just exactly right. (or wrong! :-))
- Race conditions are also very difficult to reproduce. :-(
- Obviously the solution is to only allow one process at a time to manipulate the value "counter". This is a very common occurrence among cooperating processes, so lets look at some ways in which this is done, as well as some classic problems in this area

Producer:

$$register_1 = \texttt{counter}$$
$$register_1 = register_1 + 1$$
$$\texttt{counter} = register_1$$

Consumer:

$$register_2 = \texttt{counter}$$
$$register_2 = register_2 - 1$$
$$\texttt{counter} = register_2$$

Interleaving:

T_0:	*producer*	execute	$register_1 = \texttt{counter}$	$\{register_1 = 5\}$
T_1:	*producer*	execute	$register_1 = register_1 + 1$	$\{register_1 = 6\}$
T_2:	*consumer*	execute	$register_2 = \texttt{counter}$	$\{register_2 = 5\}$
T_3:	*consumer*	execute	$register_2 = register_2 - 1$	$\{register_2 = 4\}$
T_4:	*producer*	execute	$\texttt{counter} = register_1$	$\{counter = 6\}$
T_5:	*consumer*	execute	$\texttt{counter} = register_2$	$\{counter = 4\}$

Process Synchronization – Critical Section Problem

- The producer-consumer problem described above is a specific example of a more general situation known as the ***critical section*** problem.
- The general idea is that in a number of cooperating processes, each has a critical section of code, with the following conditions and terminologies:
  - Only one process in the group can be allowed to execute in their critical section at any one time. If one process is already executing their critical section and another process wishes to do so, then the second process must be made to wait until the first process has completed their critical section work.
  - The code preceding the critical section, and which controls access to the critical section, is termed the entry section. It acts like a carefully controlled locking door.
  - The code following the critical section is termed the exit section.
  - It generally releases the lock on someone else's door, or at least lets the world know that they are no longer in their critical section.
  - The rest of the code not included in either the critical section or the entry or exit sections is termed the remainder section.

Process Synchronization – Critical Section Problem

- Solution to Critical Section Problem
- A solution to the critical section problem must satisfy the following three conditions:
  - **Mutual Exclusion -** Only one process at a time can be executing in their critical section.
  - **Progress -** If no process is currently executing in their critical section, and one or more processes want to execute their critical section, then only the processes not in their remainder sections can participate in the decision, and the decision cannot be postponed indefinitely. (I.e. processes cannot be blocked forever waiting to get into their critical sections.)
  - **Bounded Waiting -** There exists a limit as to how many other processes can get into their critical sections after a process requests entry into their critical section and before that request is granted. (I.e. a process requesting entry into their critical section will get a turn eventually, and there is a limit as to how many other processes get to go first.)

Process Synchronization – Semaphores

- A more robust alternative to simple mutexes is to use ***semaphores***, which are integer variables for which only two (atomic) operations are defined, the wait and signal operations, as shown in the figure.
- Note that not only must the variable-changing steps (S-- and S++) be indivisible, it is also necessary that for the wait operation when the test proves false that there be no interruptions before S gets decremented.
- It IS okay, however, for the busy loop to be interrupted when the test is true, which prevents the system from hanging forever.

Wait:

```
wait(S) {
    while S <= 0
        ; // no-op
    S--;
}
```

Signal:

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

Process Synchronization – Semaphores

- In practice, semaphores can take on one of two forms:
- **Binary semaphores** can take on one of two values, 0 or 1.
  - They can be used to solve the critical section problem as described above, and can be used as mutexes on systems that do not provide a separate mutex mechanism.
  - The use of mutexes for this purpose is shown in figure.
- **Counting semaphores** can take on any integer value, and are usually used to count the number remaining of some limited resource.
  - The counter is initialized to the number of such resources available in the system, and whenever the counting semaphore is greater than zero, then a process can enter a critical section and use one of the resources.
  - When the counter gets to zero (or negative in some implementations), then the process blocks until another process frees up a resource and increments the counting semaphore with a signal call. (The binary semaphore can be seen as just a special case where the number of resources initially available is just one.)

```
do {
   waiting(mutex);

      // critical section

   signal(mutex);

      // remainder section
}while (TRUE);
```

Classic Problems of Synchronization

- The following classic problems are used to test virtually every new proposed synchronization algorithm.
  1. Bounded-Buffer Problem
  2. Readers-Writers Problem
  3. Dining-Philosophers Problem

- **The Bounded-Buffer Problem**
  - This is a generalization of the producer-consumer problem wherein access is controlled to a shared group of buffers of a limited size.
  - In this solution, the two counting semaphores "full" and "empty" keep track of the current number of full and empty buffers respectively (and initialized to 0 and N respectively.)
  - The binary semaphore mutex controls access to the critical section. The producer and consumer processes are nearly identical - One can think of the producer as producing full buffers, and the consumer producing empty buffers.

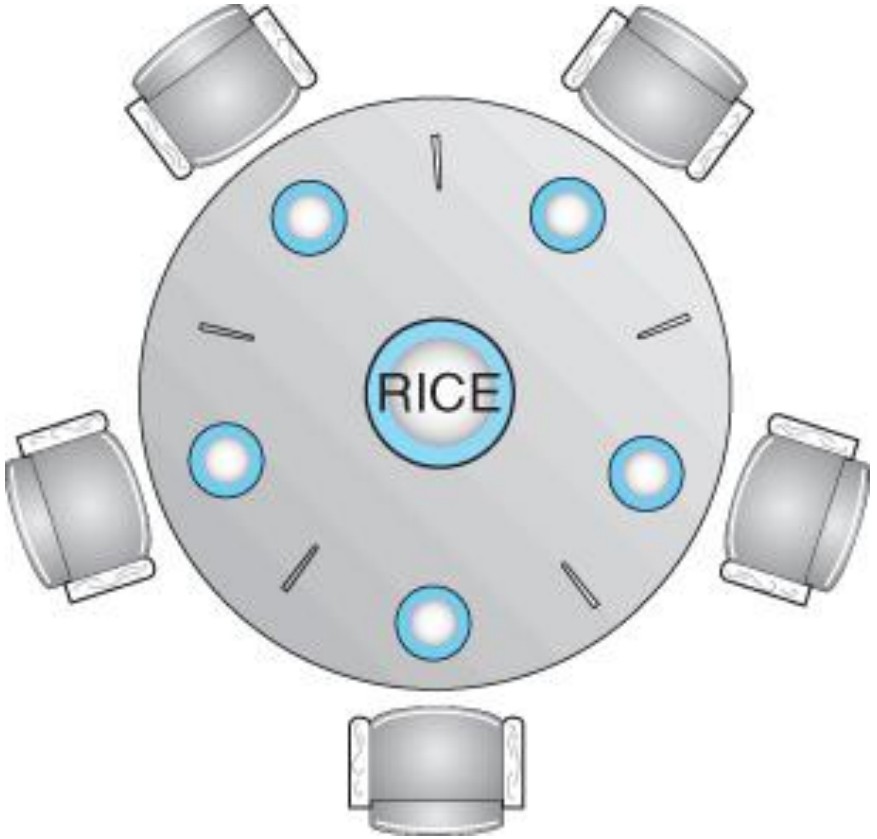
Classic Problems of Synchronization

- **The Readers-Writers Problem**
- In the readers-writers problem there are some processes (termed readers) who only read the shared data, and never change it, and there are other processes (termed writers) who may change the data in addition to or instead of reading it.
- There is no limit to how many readers can access the data simultaneously, but when a writer accesses the data, it needs exclusive access.
- There are several variations to the readers-writers problem, most centered around relative priorities of readers versus writers.
- The *first* readers-writers problem gives priority to readers.
- In this problem, if a reader wants access to the data, and there is not already a writer accessing it, then access is granted to the reader.
- A solution to this problem can lead to starvation of the writers, as there could always be more readers coming along to access the data. (A steady stream of readers will jump ahead of waiting writers as long as there is currently already another reader accessing the data, because the writer is forced to wait until the data is idle, which may never happen if there are enough readers.)
- The *second* readers-writers problem gives priority to the writers. In this problem, when a writer wants access to the data it jumps to the head of the queue - All waiting readers are blocked, and the writer gets access to the data as soon as it becomes available.
- In this solution the readers may be starved by a steady stream of writers.

Classic Problems of Synchronization

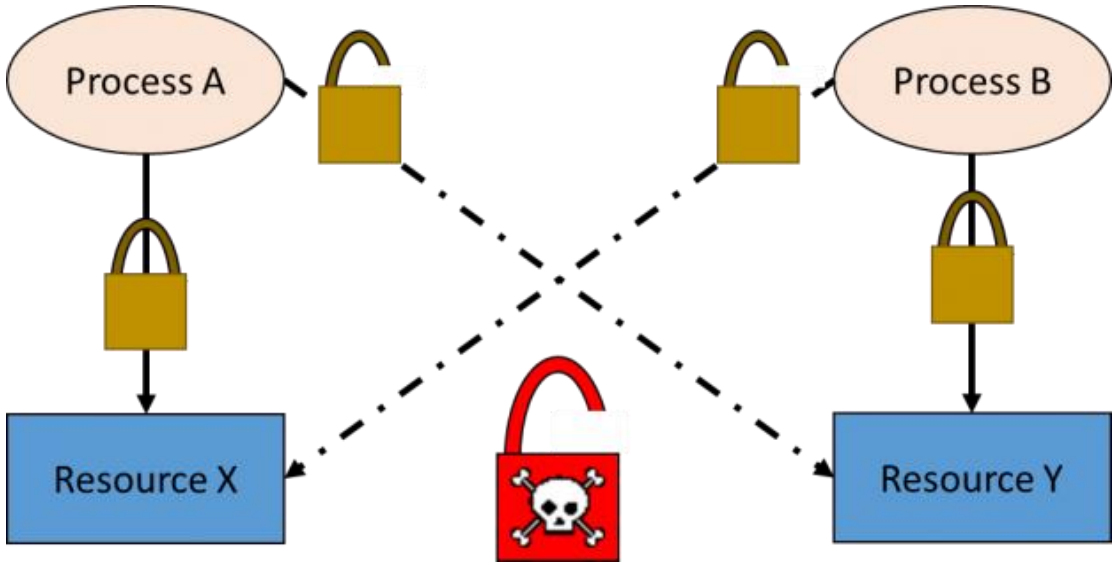
The Dining-Philosophers Problem

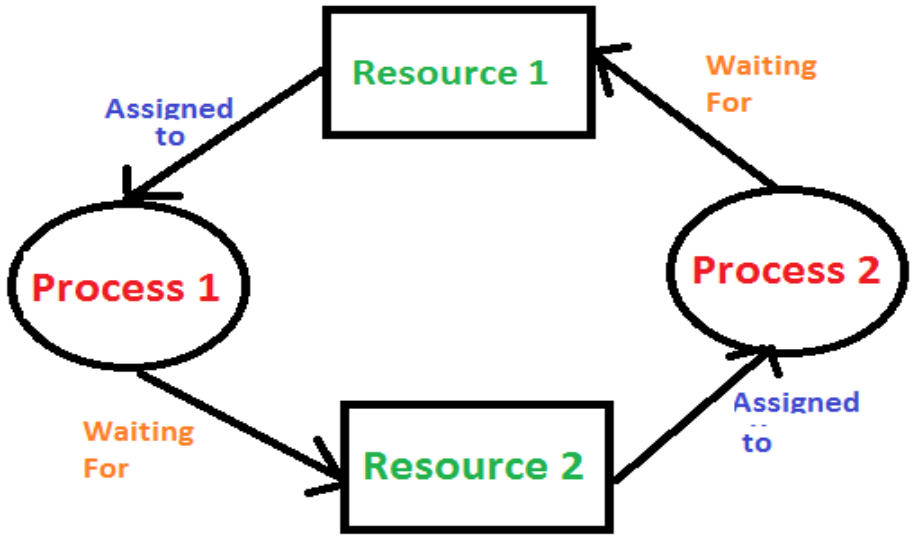
- The dining philosophers problem is a classic synchronization problem involving the allocation of limited resources amongst a group of processes in a deadlock-free and starvation-free manner:
  - Consider five philosophers sitting around a table, in which there are five chopsticks evenly distributed and an endless bowl of rice in the center, as shown in the diagram below. (There is exactly one chopstick between each pair of dining philosophers.)
  - These philosophers spend their lives alternating between two activities: eating and thinking.
  - When it is time for a philosopher to eat, it must first acquire two chopsticks - one from their left and one from their right.
  - When a philosopher thinks, it puts down both chopsticks in their original locations.

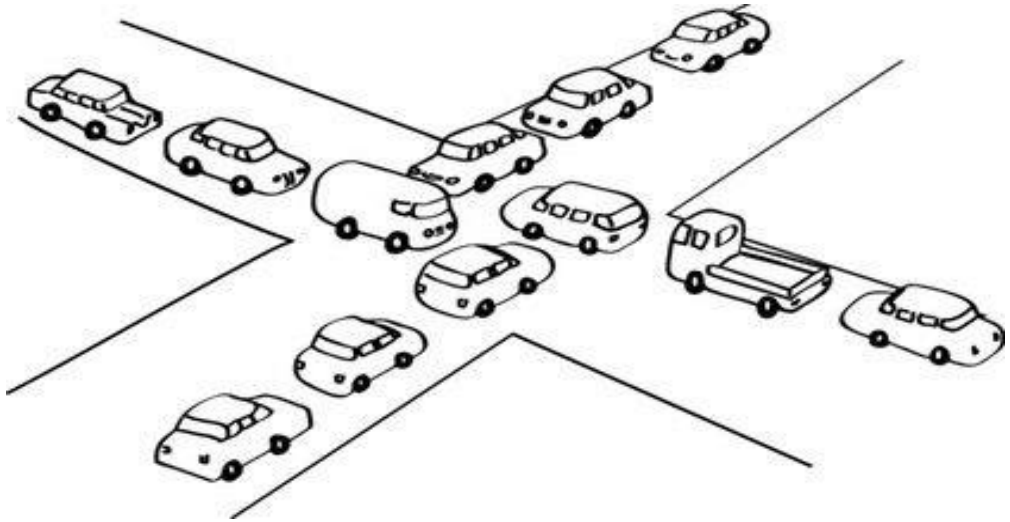


Deadlocks - Overview

- A set of processes is deadlocked when every process in the set is waiting for a resource that is currently allocated to another process in the set (and which can only be released when that other waiting process makes progress.)







Deadlocks - Overview

- A deadlock is a situation in which two computer programs sharing the same resource are effectively preventing each other from accessing the resource, resulting in both programs ceasing to function. The earliest computer operating systems ran only one program at a time.
- For the purposes of deadlock discussion, a system can be modeled as a collection of limited resources, which can be partitioned into different categories, to be allocated to a number of processes, each having different needs.
- Resource categories may include memory, printers, CPUs, open files, tape drives, CD-ROMS, etc.
- By definition, all the resources within a category are equivalent, and a request of this category can be equally satisfied by any one of the resources in that category.
- If this is not the case (i.e. if there is some difference between the resources within a category), then that category needs to be further divided into separate categories. For example, "printers" may need to be separated into "laser printers" and "color inkjet printers".
- Some categories may have a single resource.

Deadlocks - Overview

- In normal operation a process must request a resource before using it, and release it when it is done, in the following sequence:
  - Request - If the request cannot be immediately granted, then the process must wait until the resource(s) it needs become available. For example the system calls open( ), malloc( ), new( ), and request( ).
  - Use - The process uses the resource, e.g. prints to the printer or reads from the file.
  - Release - The process relinquishes the resource. so that it becomes available for other processes. For example, close( ), free( ), delete( ), and release( ).
- For all kernel-managed resources, the kernel keeps track of what resources are free and which are allocated, to which process they are allocated, and a queue of processes waiting for this resource to become available.
- Application-managed resources can be controlled using mutexes or wait( ) and signal( ) calls, (i.e. binary or counting semaphores.)
- A set of processes is deadlocked when every process in the set is waiting for a resource that is currently allocated to another process in the set (and which can only be released when that other waiting process makes progress.)

Deadlocks Characterization

- There are four conditions that are necessary to achieve deadlock:
  - **Mutual Exclusion** - At least one resource must be held in a non-sharable mode; If any other process requests this resource, then that process must wait for the resource to be released.
  - **Hold and Wait** - A process must be simultaneously holding at least one resource and waiting for at least one resource that is currently being held by some other process.
  - **No preemption** - Once a process is holding a resource (i.e. once its request has been granted), then that resource cannot be taken away from that process until the process voluntarily releases it.
  - **Circular Wait** - A set of processes { P0, P1, P2, . . ., PN } must exist such that every P[i] is waiting for P[(i + 1) % (N + 1)]. (Note that this condition implies the hold-and-wait condition, but it is easier to deal with the conditions if the four are considered separately.)

Methods of Handling Deadlocks

- There are three ways of handling deadlocks:
  - Deadlock prevention or avoidance - Do not allow the system to get into a deadlocked state.
  - Deadlock detection and recovery - Abort a process or preempt some resources when deadlocks are detected.
  - Ignore the problem all together - If deadlocks only occur once a year or so, it may be better to simply let them happen and reboot as necessary than to incur the constant overhead and system performance penalties associated with deadlock prevention or detection. This is the approach that both Windows and UNIX take.
- In order to avoid deadlocks, the system must have additional information about all processes. In particular, the system must know what resources a process will or may request in the future. (Ranging from a simple worst-case maximum to a complete resource request and release plan for each process, depending on the particular algorithm.)
- Deadlock detection is fairly straightforward, but deadlock recovery requires either aborting processes or preempting resources, neither of which is an attractive alternative.
- If deadlocks are neither prevented nor detected, then when a deadlock occurs the system will gradually slow down, as more and more processes become stuck waiting for resources currently held by the deadlock and by other waiting processes. Unfortunately this slowdown can be indistinguishable from a general system slowdown when a real-time process has heavy computing needs.

Deadlock Prevention

- Deadlocks can be prevented by preventing at least one of the four required conditions:

1. **Mutual Exclusion**
   - Shared resources such as read-only files do not lead to deadlocks.
   - Unfortunately some resources, such as printers and tape drives, require exclusive access by a single process.
2. **Hold and Wait**
   - To prevent this condition processes must be prevented from holding one or more resources while simultaneously waiting for one or more others. There are several possibilities for this:
     - Require that all processes request all resources at one time. This can be wasteful of system resources if a process needs one resource early in its execution and doesn't need some other resource until much later.
     - Require that processes holding resources must release them before requesting new resources, and then re-acquire the released resources along with the new ones in a single new request. This can be a problem if a process has partially completed an operation using a resource and then fails to get it re-allocated after releasing it.
     - Either of the methods described above can lead to starvation if a process requires one or more popular resources.

Deadlock Prevention

3. No Preemption

- Preemption of process resource allocations can prevent this condition of deadlocks, when it is possible.
  - One approach is that if a process is forced to wait when requesting a new resource, then all other resources previously held by this process are implicitly released, (preempted), forcing this process to re-acquire the old resources along with the new resources in a single request, similar to the previous discussion.
  - Another approach is that when a resource is requested and not available, then the system looks to see what other processes currently have those resources *and* are themselves blocked waiting for some other resource. If such a process is found, then some of their resources may get preempted and added to the list of resources for which the process is waiting.
  - Either of these approaches may be applicable for resources whose states are easily saved and restored, such as registers and memory, but are generally not applicable to other devices such as printers and tape drives.

4. Circular Wait

- One way to avoid circular wait is to number all resources, and to require that processes request resources only in strictly increasing (or decreasing) order.
- In other words, in order to request resource Rj, a process must first release all Ri such that i >= j.
- One big challenge in this scheme is determining the relative ordering of the different resources

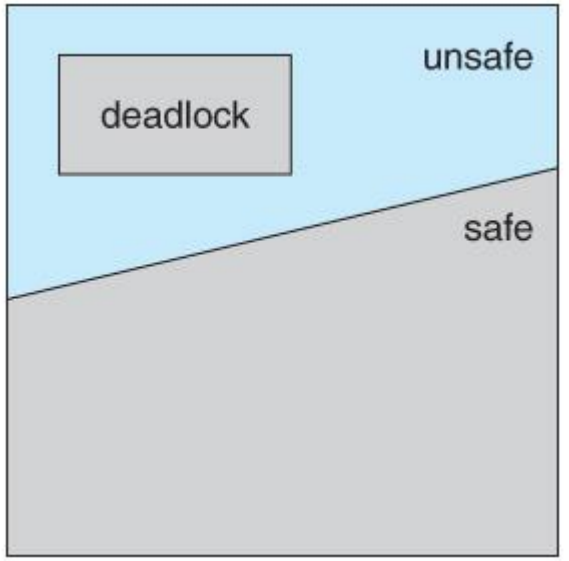
Deadlock Avoidance

- The general idea behind deadlock avoidance is to prevent deadlocks from ever happening, by preventing at least one of the aforementioned conditions.
- This requires more information about each process, AND tends to lead to low device utilization. (I.e. it is a conservative approach.)
- In some algorithms the scheduler only needs to know the *maximum* number of each resource that a process might potentially use. In more complex algorithms the scheduler can also take advantage of the *schedule* of exactly what resources may be needed in what order.
- When a scheduler sees that starting a process or granting resource requests may lead to future deadlocks, then that process is just not started or the request is not granted.
- A resource allocation ***state*** is defined by the number of available and allocated resources, and the maximum requirements of all processes in the system.

Deadlock Avoidance

Safe State

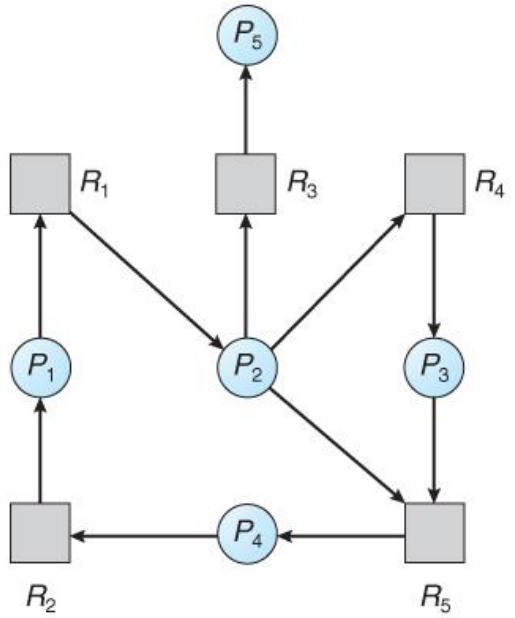
- A state is ***safe*** if the system can allocate all resources requested by all processes (up to their stated maximums) without entering a deadlock state.
- More formally, a state is safe if there exists a ***safe sequence*** of processes { P0, P1, P2, ..., PN } such that all of the resource requests for Pi can be granted using the resources currently allocated to Pi and all processes Pj where j < i. (I.e. if all the processes prior to Pi finish and free up their resources, then Pi will be able to finish also, using the resources that they have freed up.)
- If a safe sequence does not exist, then the system is in an unsafe state, which ***MAY*** lead to deadlock. (All safe states are deadlock free, but not all unsafe states lead to deadlocks.)



Safe, unsafe, and deadlocked state spaces

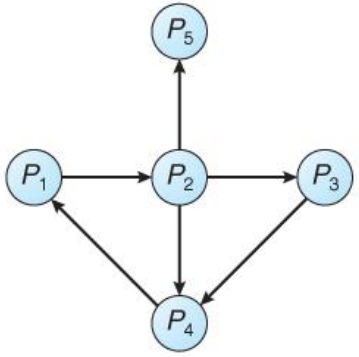
Deadlock Detection

- If deadlocks are not avoided, then another approach is to detect when they have occurred and recover somehow.
- In addition to the performance hit of constantly checking for deadlocks, a policy / algorithm must be in place for recovering from deadlocks, and there is potential for lost work when processes must be aborted or have their resources preempted.
- **Single Instance of Each Resource Type**
  - If each resource category has a single instance, then we can use a variation of the resource-allocation graph known as a ***wait-for graph***.
  - A wait-for graph can be constructed from a resource-allocation graph by eliminating the resources and collapsing the associated edges, as shown in the figure below.
  - An arc from Pi to Pj in a wait-for graph indicates that process Pi is waiting for a resource that process Pj is currently holding.
  - As before, cycles in the wait-for graph indicate deadlocks.
  - This algorithm must maintain the wait-for graph, and periodically search it for cycles.



(a)

(a) Resource allocation graph



(b)

(b) Corresponding Wait for Graph

Recovery From Deadlock

- There are three basic approaches to recovery from deadlock:
  - Inform the system operator, and allow him/her to take manual intervention.
  - Terminate one or more processes involved in the deadlock
  - Preempt resources.
- **Process Termination**
- Two basic approaches, both of which recover resources allocated to terminated processes:
  - Terminate all processes involved in the deadlock. This definitely solves the deadlock, but at the expense of terminating more processes than would be absolutely necessary.
  - Terminate processes one by one until the deadlock is broken. This is more conservative, but requires doing deadlock detection after each step.
- In the latter case there are many factors that can go into deciding which processes to terminate next:
  - Process priorities.
  - How long the process has been running, and how close it is to finishing.
  - How many and what type of resources is the process holding. (Are they easy to preempt and restore?)
  - How many more resources does the process need to complete.
  - How many processes will need to be terminated
  - Whether the process is interactive or batch.
  - (Whether or not the process has made non-restorable changes to any resource.)

Recovery From Deadlock

- **Resource Preemption**
- When preempting resources to relieve deadlock, there are three important issues to be addressed:
  - **Selecting a victim -** Deciding which resources to preempt from which processes involves many of the same decision criteria outlined above.
  - **Rollback -** Ideally one would like to roll back a preempted process to a safe state prior to the point at which that resource was originally allocated to the process. Unfortunately it can be difficult or impossible to determine what such a safe state is, and so the only safe rollback is to roll back all the way back to the beginning. (I.e. abort the process and make it start over.)
  - **Starvation** - How do you guarantee that a process won't starve because its resources are constantly being preempted? One option would be to use a priority system, and increase the priority of a process every time its resources get preempted. Eventually it should get a high enough priority that it won't get preempted any more.

धन्यवाद