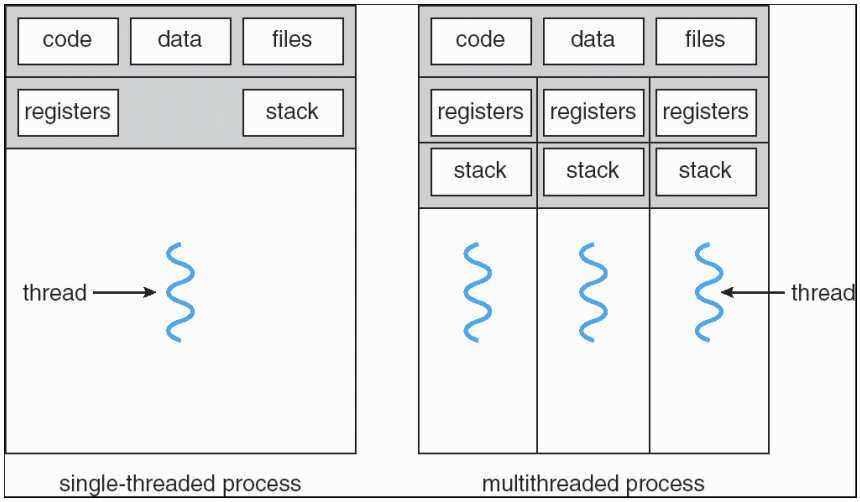
Overview

- A thread is a **flow of control within** a **process**
- A **multithreaded process** contains several different flows of control within the same address space
- A traditional (heavyweight) process has one thread of control
- A thread / lightweight process (LWP) = a unit of **CPU utilization**
- It comprises a thread ID, program counter, register set, & stack
- It shares with other threads belonging to the same process its code section, data section, and other OS resources
- If a process has multiple threads of control, it can perform more than one task at a time
- Look at fig 4.1 p.153 TB



- User-level threads are threads that are visible to a programmer and are unknown to the kernel
- OS kernel supports and manages kernel-level threads

Motivation

- It is more efficient to have multithreading than many processes
- RPC servers are typically multithreaded
    - When a server receives a message, it services it with a separate thread
    - This lets the server service several concurrent requests

Benefits

- **Responsiveness**:
    - A program can continue running even if part of it is busy
- **Resource sharing**:
    - Threads share the memory and resources of their process
- **Economy**:
    - Allocating memory and resources for processes is costly (time)
- **Scalability**:
    - Utilization of multiprocessor architectures
    - Each thread runs on a separate CPU, increasing concurrency / parallelism

Multicore Programming

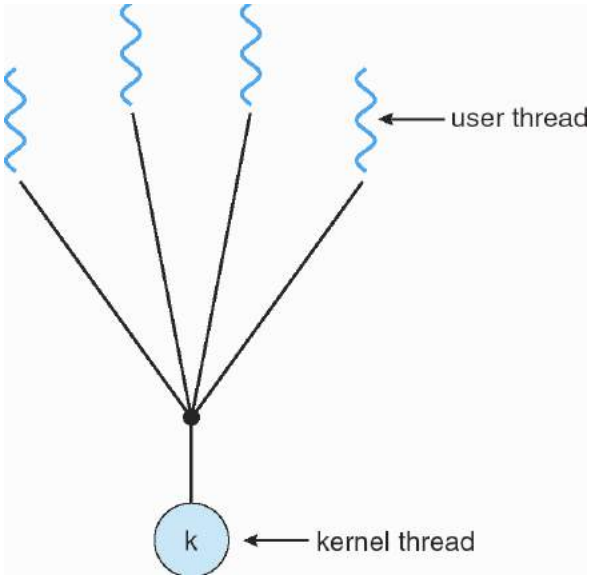
- p.156 - 157 TB
- Provides a mechanism for more efficient use of multiple cores and improved concurrency
- On a system with multiple cores the processes run concurrently since the system can assign a separate thread to each core
- Five areas that present challenges in programming multicore systems:
    - **Dividing activities:**
        - Areas of applications to be divided into different tasks
    - **Balance:**
        - Tasks must perform equal work of equal value, else CPU time is wasted
    - **Data splitting:**
        - Data accessed and manipulated must be divided to run on separate cores
    - **Data dependency**:
        - If data between cores depends on each other, execution must be synchronized
    - **Testing and debugging**:
        - More difficult to test and debug than single-threaded execution

Multithreading Models

User threads (Many-to-One)	**Kernel threads (One-to-One)**
Implemented by a thread library at the user level	Supported directly by the OS
The library provides support for thread creation, scheduling, and management with no support from the OS kernel	The kernel performs thread creation, scheduling, and management in kernel space
Faster to create & manage because the kernel is unaware of user threads and doesn't intervene	Slower to create & manage than user threads because thread management is done by the OS
Disadvantage: If the kernel is single- threaded, then any user-level thread performing a blocking system call will cause the entire process to block, even if other threads are available to run within the application	Since the kernel is managing the threads, if a thread performs a blocking system call, the kernel can schedule another thread in the application for execution. In a multiprocessor environment, the kernel can schedule threads on different processors.

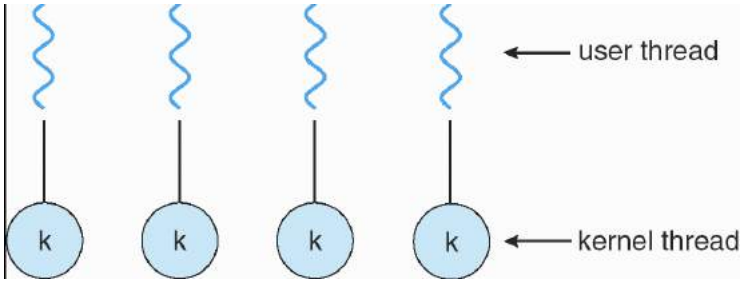
Many-to-One Model

- Many user-level threads are mapped to one kernel thread
- Thread management is done in user space, so it is **efficient**
- The entire process will block if a thread makes a blocking call
- Multiple threads can’t run in parallel on multiprocessors



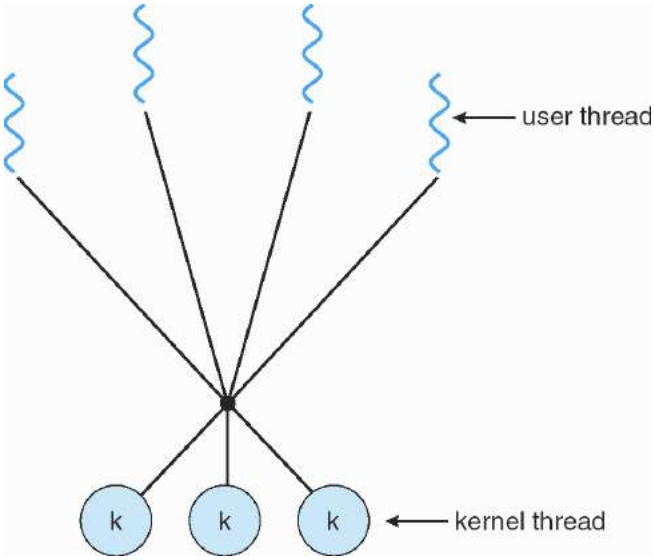
One-to-One Model

- Each user thread is mapped to a kernel thread
- **More concurrency** than the many-to-one model because another thread can run when a thread makes a blocking system call
- Multiple threads can run in parallel on multiprocessors
- Disadvantage: creating a user thread requires creating the corresponding kernel thread (This **overhead** burdens performance)

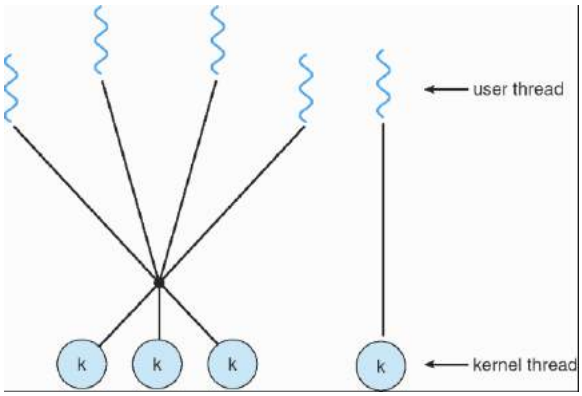


Many-to-Many Model

- Many user-level threads are multiplexed to a smaller / equal number of kernel threads
- Developers can create as many user threads as necessary
- The kernel threads can run in parallel on a multiprocessor
- When a thread performs a blocking system call, the kernel can schedule another thread for execution



- A variation on the Many-to-Many Model is the two level-model:
    - Similar to M:M, except that it allows a user thread to be **bound** to kernel thread



Thread Libraries

- Thread libraries provide the application programmer with an API for creating and managing threads
- Three main thread libraries in use today:
    - POSIX Pthreads
    - Win32 threads
    - Java threads

Pthreads

- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
- API specifies behavior of the thread library, implementation is up to development of the library
- Common in UNIX operating systems (Solaris, Linux, Mac OS X)

Win32 Threads

Java Threads

- Java threads are managed by the JVM
- Java threads may be created by:
    1. Implementing the Runnable interface

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

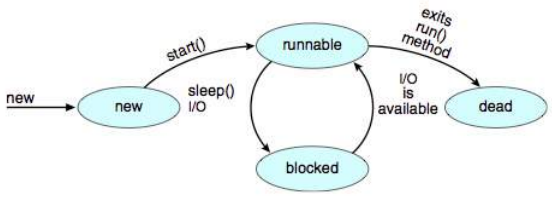
- Sample program:

```
class MutableInteger
{
  private int value;
  public int getValue() {
   return value;
  }
  public void setValue(int value) {
   this.value = value;
  }
}

class Summation implements Runnable
{
  private int upper;
  private MutableInteger sumValue;
  public Summation(int upper, MutableInteger sumValue) {
   this.upper = upper;
   this.sumValue = sumValue;
  }
  public void run() {
   int sum = 0;
   for (int i = 0; i <= upper; i++)
      sum += i;
   sumValue.setValue(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
      MutableInteger sum = new MutableInteger();
      int upper = Integer.parseInt(args[0]);
      Thread thrd = new Thread(new Summation(upper, sum));
      thrd.start();
      try {
         thrd.join();
         System.out.println
                ("The sum of "+upper+" is "+sum.getValue());
      } catch (InterruptedException ie) { }
     }
    }
    else
     System.err.println("Usage: Summation <integer value>");
    }
}
```

- Java thread states:



Threading Issues

- Here we discuss issues to consider with multithreaded programs

The fork() and exec() System Calls

- fork() system call: used to create a **separate**, **duplicate** process
  - Some versions of fork() duplicate all threads
    - If exec() won't be called afterwards
  - Other versions duplicate only the thread that invoked fork()
    - If exec() is called immediately after forking
- exec() system call: the parameter used will **replace** the process
  - All threads will also be replaced

Cancellation

- **Thread cancellation** is the task of terminating a thread before it has completed.
- Target thread = the thread that is to be canceled
- Cancellation of target threads occur in two different scenarios:

Asynchronous cancellation	**Deferred cancellation**
One thread immediately terminates the target thread	The target thread can periodically check if it should terminate
Canceling a thread may not free a necessary system- wide resource	Cancellation occurs only when the target thread checks if it should be canceled. (Cancellation points)

- Deferred cancellation in Java
  - Interrupting a thread

```
Thread thrd = new Thread(new InterruptibleThread());
thrd.start();
. . .
thrd.interrupt();
```

- Deferred cancellation in Java
  - Checking interruption status

```
class InterruptibleThread implements Runnable
{
   /**
    * This thread will continue to run as long
    * as it is not interrupted.
    */
   public void run() {
      while (true) {
         /**
          * do some work for awhile
          * . . .
          */

         if (Thread.currentThread().isInterrupted()) {
            System.out.println("I'm interrupted!");
            break;
         }
      }
      // clean up and terminate
   }
}
```

Signal Handling

- A **signal is used in UNIX to** notify a process that a particular event has occurred
- All signals follow this pattern:
    - A signal is generated by the occurrence of a certain event
    - A generated signal is delivered to a process
    - Once delivered, the signal must be handled
- A **signal handler** is used to process signals
    - Signal is generated by particular event
    - Signal is delivered to a process
    - Signal is handled
- Delivering signals in multithreaded programs, the **options** are:
    - 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
- Synchronous signals are delivered to the *same* process that performed the operation causing the signal (E.g. / by 0)
- Asynchronous signals are generated by an event external to a running process (E.g. user terminating a process with <ctrl><c>)
- Every signal must be handled by one of two possible handlers:
    - A default signal handler
        - Run by the kernel when handling the signal
    - A user-defined signal handler
        - Overrides the default signal handler

Single-threaded programs	Multithreaded programs
Straightforward signal handling	Complicated signal handling
Signals are always delivered to a process	Which thread should the signal be delivered to?

- The method for delivering a signal depends on the signal type:
    - **Synchronous** signals need to be delivered to the thread that generated the signal, not to other threads in the process
    - It is not clear what to do with **asynchronous** signals
        - Signals need to be handled only once, so they're usually delivered to the 1st thread not blocking them

Thread Pools

- The idea is to create a number of threads at process startup and place them into a pool, where they sit and wait for work
    - When a server receives a request, it awakens a thread from this pool
    - If one is available the request is passed to it for service

- Once the service is completed, the thread returns to the pool and wait for more work

- **Benefits** of thread pools:
  - It is **faster** to service a request with an existing thread
  - A thread pool **limits the number of threads** that exist
- Potential **problems** with a multithreaded server:
  - It takes **time** to create a thread before servicing a request
  - Unlimited threads could **exhaust** system resources (CPU time)
- Thread pools are a **solution to these problems**:
  - At process startup, several threads are created and placed into a pool, where they sit and wait for work
  - When a server receives a request, it awakens a thread from this pool, passing it the request to service
  - When the thread finishes its service it returns to the pool

Thread-Specific Data

- Threads belonging to a process **share the data** of the process
- Sometimes, each thread might need its own copy of certain data
  - E.g. Transactions in different threads may each be assigned a unique identifier
- Thread-specific data in Java

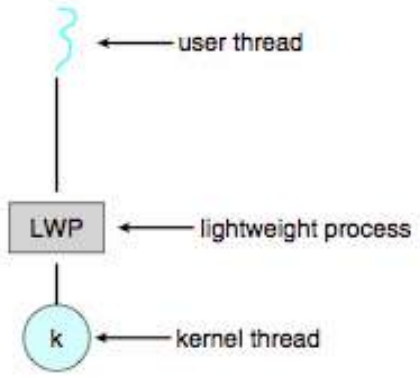
```
class Service
{
   private static ThreadLocal errorCode =
     new ThreadLocal();

   public static void transaction() {
      try {
         /**
          * some operation where an error may occur
          . . .
          */
      }
      catch (Exception e) {
          errorCode.set(e);
      }
   }

   /**
    * get the error code for this transaction
    */
   public static Object getErrorCode() {
      return errorCode.get();
   }
}
```

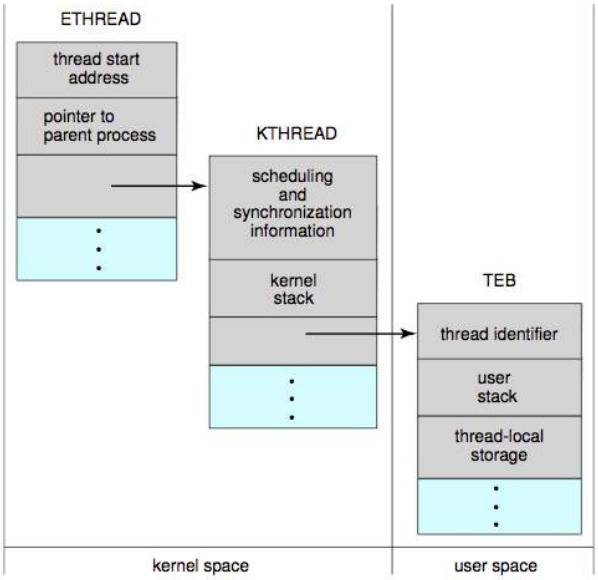
Scheduler Activations

- Both M:M and Two-level models require communication to maintain the appropriate number of kernel threads allocated to the application
- Scheduler activations provide **upcalls** - a **communication mechanism from the kernel to the thread library**
- This communication allows an application to maintain the correct number kernel threads



Operating-System Examples

- **Windows XP threads**
  - Implements the one-to-one mapping
  - Each thread contains
    1. A thread id
    2. Register set
    3. Separate user and kernel stacks
    4. Private data storage area
  - The register set, stacks, and private storage area are known as the **context** of the threads



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

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

Summary

Chapter 5: Process (CPU) Scheduling

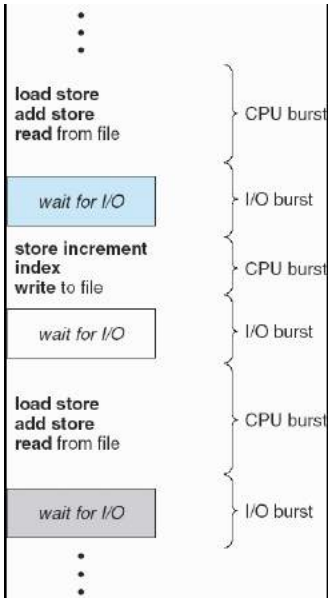
- Here we look at basic CPU-scheduling concepts and present several CPU-scheduling algorithms.
- We also consider the problem of selecting an algorithm for a particular system.
- **Objectives**:
    - To introduce CPU scheduling, which is the basis for multi-programmed operating systems.
    - To describe various CPU-scheduling algorithms.
    - To discuss evaluation criteria for selecting a CPU-scheduling algorithm for a particular system.
- The terms process scheduling and thread scheduling are often used interchangeably

Basic Concepts

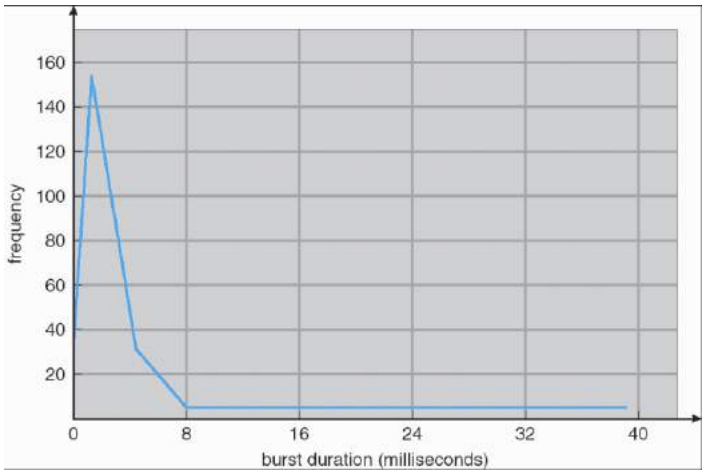
- CPU scheduling is the task of selecting a waiting process from the ready queue and allocating the CPU to it
    - The CPU is allocated to the selected process by the dispatcher
- In a uni-processor system, only one process may run at a time; any other process must wait until the CPU is rescheduled
- The objective of multiprogramming is to have some process running at all times, in order to maximize CPU utilization
- CPU–I/O Burst Cycle – Process execution consists of a *cycle* of CPU execution and I/O wait
- CPU burst distribution

CPU-I/O Burst Cycle

- Process execution comprises a cycle of CPU execution & I/O wait
- Process execution begins with a CPU burst, followed by an I/O burst, then another CPU burst, etc…
- Finally, a CPU burst ends with a request to terminate execution



Histogram of CPU-burst times:



- An I/O-bound program typically has many short CPU bursts
- A CPU-bound program might have a few long CPU bursts
- These are important points to keep in mind for the selection of an appropriate CPU-scheduling algorithm

CPU Scheduler

- Selects from among the processes in memory that are ready to execute, and allocates the CPU to one of them
- The short-term scheduler selects a process in the ready queue when the CPU becomes idle
- The ready queue could be a FIFO / priority queue, tree, list…
- The records in the queues are generally process control blocks (PCBs) of the processes

Preemptive Scheduling

- Circumstances under which CPU scheduling decisions take place:
    - When a process switches from the running state to the waiting state (E.g. I/O request) (1)
    - When a process switches from the running state to the ready state (E.g. when an interrupt occurs) (2)
    - When a process switches from the waiting state to the ready state (E.g. completion of I/O) (3)

    - When a process terminates (4)
- Non-preemptive/cooperative scheduling
    - Processes are allowed to **run to completion**
    - When scheduling takes place under circumstances 1 & 4
    - There is no choice in terms of scheduling
- Preemptive scheduling
    - Processes that are runnable may be **temporarily suspended**
    - There is a scheduling choice in circumstances 2 & 3
    - Problem: if one process is busy updating data and it is preempted for the second process to run, if the second process reads that data, it could be inconsistent

Dispatcher

- A component involved in the CPU scheduling function
- The dispatcher is the module that **gives control** of the CPU to the process selected by the short-term scheduler
- This function involves:
    - Switching context
    - Switching user mode
    - Jumping to the proper location in the user program to restart that program
- The dispatcher should be as fast as possible, given that it is invoked during every process switch
- Dispatch latency = the time it takes for the dispatcher to stop one process and start another running

Scheduling Criteria

- Different CPU-scheduling algorithms have different properties and the choice of a particular algorithm may favor one class of process over another
- **Criteria** to compare CPU-scheduling algorithms:
    - **CPU utilization**
        - CPU utilization should range from 40% - 90%
    - **Throughput**
        - The number of processes completed per time unit
    - **Turnaround time**
        - The time interval from process submission to completion
        - Formula: Time of completion – Time of submission
        - Formula: CPU burst time + Waiting time (includes I/O)
    - **Waiting time**
        - The sum of the periods spent waiting in the ready queue
        - Formula: Turnaround time – CPU burst time
    - **Response time**
        - The amount of time it takes to start responding, but not the time it takes to output that response
- We want to maximize CPU utilization, and minimize turnaround, waiting & response time

Scheduling Algorithms

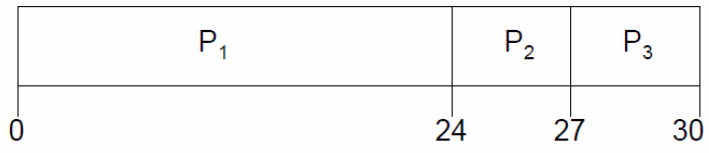
- CPU scheduling deals with the problem of deciding which of the processes in the ready queue is to be allocated the CPU
- There are many different CPU-scheduling algorithms. Here we describe several of them.

First-Come, First-Served (FCFS) Scheduling

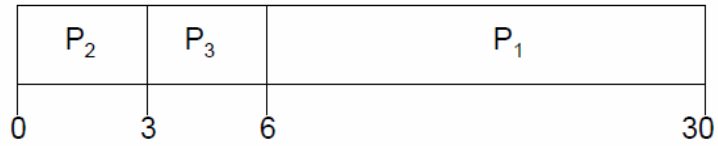
- The process that requests the CPU first is allocated the CPU 1st
- The PCB of a process is linked onto the tail of the ready queue
- When the CPU is free, it gets the process at the queue's head
- The average waiting time is generally **not minimal**
- Convoy effect = when processes wait for a big one to get off
- Non-preemptive (a process keeps the CPU until it releases it)
- Not good for time-sharing systems, where each user needs to get a share of the CPU at regular intervals
- **Example**:

Process	**Burst Time**
P1	24
P2	3
P3	3

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



  - Waiting time for *P1* = 0; *P2* = 24; *P3* = 27
  - Average waiting time: (0 + 24 + 27)/3 = 17
- Suppose that the processes arrive in the order *P2*, *P3*, *P1*
  - The Gantt chart for the schedule is:



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

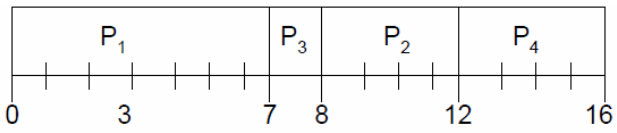
Shortest-Job-First (SJF) Scheduling

- The CPU is assigned the process with the shortest next CPU burst
- If two processes have the same length, FCFS scheduling is used

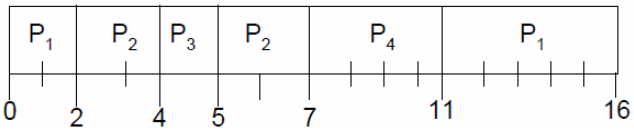
- The difficulty is knowing the length of the next CPU request
- For **long-term scheduling** in a **batch** system, we can use the process time limit specified by the user, as the 'length'
- SJF can't be implemented at the level of short-term scheduling, because there is no way to know the length of the next CPU burst
- We can, however, try to predict the length of the next CPU burst
- The SJF algorithm may be either **preemptive** or **non-preemptive**
    - **Preemptive SJF algorithm**:
        - If the new process has a shorter next CPU burst than what is left of the executing process, that process is preempted
        - aka Shortest-Remaining-Time-First (SRTF) scheduling
    - **Non-preemptive SJF algorithm**:
        - The current process is allowed to finish its CPU burst
- SJF has the **minimum average waiting time** for a set of processes
- **Example**:

Process	Arrival Time	Burst Time
P1	0.0	7
P2	2.0	4
P3	4.0	1
P4	5.0	4

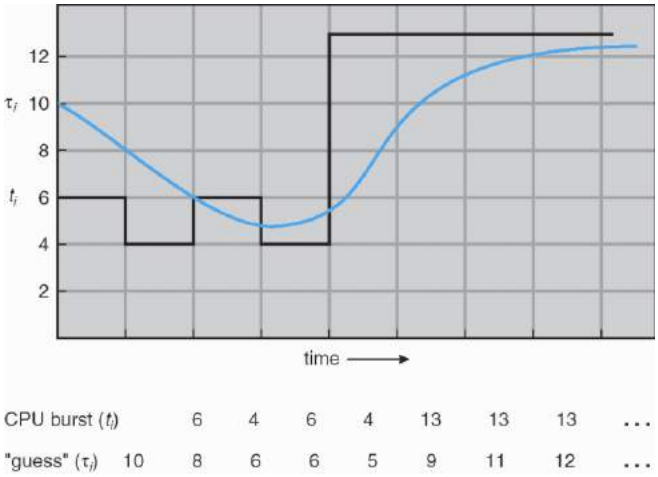
    - **SJF (non-preemptive)**



        - Average waiting time = (0 + 6 + 3 + 7)/4 = 4
    - **SJF (preemptive)**



        - Average waiting time = (9 + 1 + 0 +2)/4 = 3
- **Determining the length of the next CPU burst**:
    - Can only estimate the length
    - Can be done by using the length of previous CPU bursts, using exponential averaging
    - Formula on p.191 top



- **Examples of exponential averaging**:

$\alpha = 0$

- $\tau_{n+1} = \tau_n$
- Recent history does not count

$\alpha = 1$

- $\tau_{n+1} = \alpha\, t_n$
- Only the actual last CPU burst counts

If we expand the formula, we get:

$$\tau_{n+1} = \alpha\, t_n + (1 - \alpha)\alpha\, t_{n-1} + \ldots + (1 - \alpha)^j \alpha\, t_{n-j} + \ldots + (1 - \alpha)^{n+1} \tau_0$$

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

Priority Scheduling

- Each process gets a priority (Highest priority = executed first)
  - **Preemptive** priority scheduling
    - The CPU is preempted if the priority of the newly arrived process is higher than the priority of the current one
  - **Non-preemptive** priority scheduling
    - The new process is put at the **head** of the ready queue
- Equal-priority processes are scheduled in FCFS order
  - **Internally-defined** priorities
    - Use some measurable quantity to compute the priority
    - E.g. time limits, memory requirements, no. of open files…
  - **Externally-defined** priorities
    - Set by criteria that are external to the OS
    - E.g. the importance of a process, political factors…
- **Problem**:
  - Indefinite blocking (starvation), where low-priority processes are left waiting indefinitely for the CPU

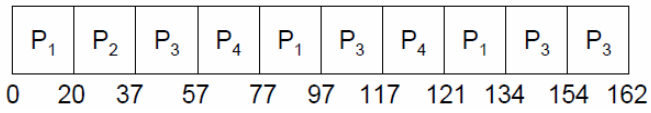
- **Solution**:
  - Aging (a technique of gradually increasing the priority of processes that wait in the system for a long time)

Round-Robin Scheduling

- Designed especially for time-sharing systems
- Like FCFS scheduling, but with preemption
- A time quantum / time slice is defined (generally 10 – 100 ms)
- The ready queue is treated as a circular queue
- The CPU scheduler goes around the ready queue, allocating the CPU to each process for a time interval of up to 1 time quantum
- The ready queue is kept as a FIFO queue of processes
- The CPU scheduler
  - picks the 1st process from the ready queue
  - sets a timer to interrupt after 1 time quantum, and
  - dispatches the process
- One of two things will then happen:
  - The process may have a CPU burst of less than 1 time quantum, and will release the CPU voluntarily
  - If the process has a CPU burst longer than 1 time quantum, the timer will go off and cause an interrupt to the OS. The process will then be put at the **tail** of the ready queue
- If there are n processes in the ready queue and the time quantum is q, then each process gets $1/n$ of the CPU time in chunks of at most q time units at once. No process waits more than $(n-1)q$ time units
- RR Performance depends heavily on the size of the time quantum
  - q large ⇒ FIFO
  - q small ⇒ q must be large with respect to context switch, otherwise overhead is too high
- We want the time quantum to be large with respect to the context-switch time
- **Example of RR with time Quantum = 20:**

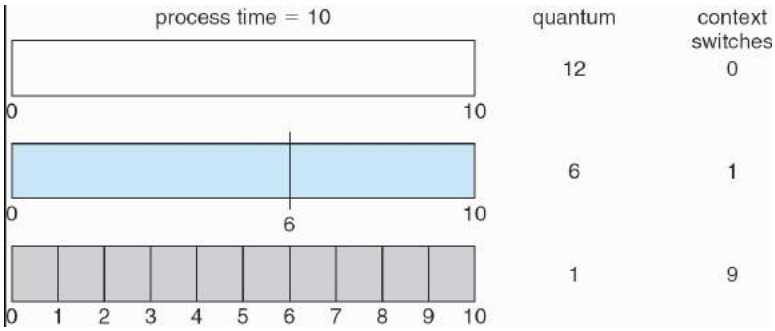
Process	**Burst Time**
P1	53
P2	17
P3	68
P4	24

  - The Gantt chart is:

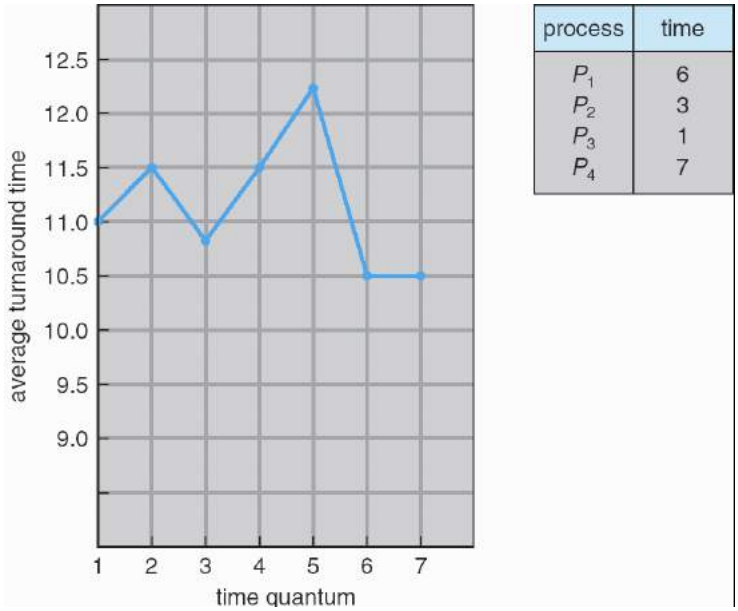


- Typically, **higher average turnaround** than SJF, but better *response*
- In software we need to consider the effect of context switching on the performance of RR scheduling
  - The larger the time quantum for a specific process time, the less time is spend on context switching

- The smaller the time quantum, more overhead is added for the purpose of context-switching
- **Example**: (This is on a per case situation)



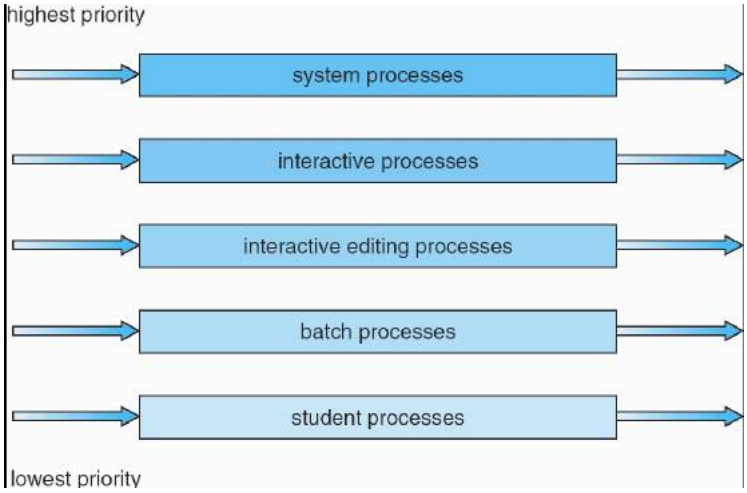
- Turnaround time also depends on the size of the time quantum:



process	time
P_1	6
P_2	3
P_3	1
P_4	7

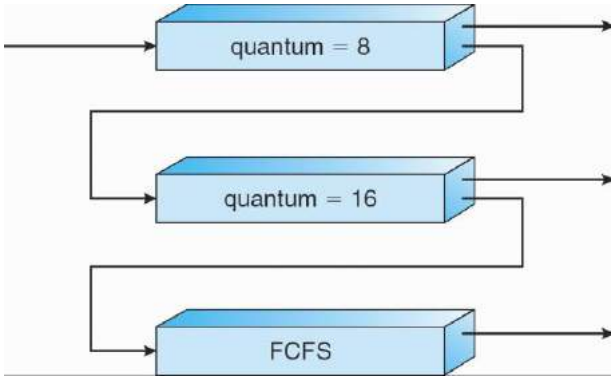
Multilevel Queue Scheduling

- For when processes can easily be classified into separate groups
- E.g. a common division is made between foreground (interactive) and background (batch) processes
- The ready queue is partitioned into several separate queues
- The processes are **permanently** assigned to one queue, based on a property like memory size, process priority, process type…
- Each queue has its own scheduling algorithm
- There must also be **scheduling among the queues**, which is commonly implemented as fixed-priority preemptive scheduling
    - Fixed priority scheduling; (i.e., serve all from foreground then from background). Possibility of starvation.
    - Time slice -each queue gets a certain amount of CPU time which it can schedule amongst its processes; i.e., 80% to foreground in RR
    - 20% to background in FCFS



Multilevel Feedback Queue Scheduling

- Processes may **move** between queues
- Processes with different CPU-burst characteristics are separated
- If a process uses too much CPU time, it will be moved to a lower-priority queue
- If a process waits too long in a lower-priority queue, it may be moved to a higher-priority queue (Aging prevents starvation)
- In general, a multilevel feedback queue scheduler 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
- This is the most general, but complex scheme
- **Example** of Multilevel Feedback Queue:
    - **Three queues**:
        - *Q*0 - RR with time quantum 8 milliseconds
        - *Q*1 - RR time quantum 16 milliseconds
        - *Q*2 - FCFS
    - **Scheduling**
        - A new job enters queue *Q0* which is served FCFS
            - When it gains CPU, job receives 8 milliseconds
            - If it does not finish in 8 milliseconds, job is moved to queue *Q*1
        - At *Q*1 job is again served FCFS and receives 16 additional milliseconds
            - If it still does not complete, it is preempted and moved to queue *Q*2
- Multilevel feedback queues:



Thread Scheduling

- p.199 TB
- On operating systems that support them, it is kernel-level threads - not processes - that are being scheduled by the operating system
- **Local Scheduling**
  - How the **threads library** decides which thread to put onto an available LWP
- **Global Scheduling**
  - How the **kernel** decides which kernel thread to run next

Contention Scope

- **Process-Contention scope**:
  - On systems implementing the many-to-one and many-to-many models, the thread library schedules user-level threads to run on an available LWP
- **System-Contention scope**:
  - The process of deciding which kernel thread to schedule on the CPU

Pthread Scheduling

Sample of thread creation with Pthreads:

```
int main(int argc, char *argv[])
{
   int i, scope;
   pthread_t tid[NUM_THREADS];
   pthread_attr_t attr;

   /* get the default attributes */
   pthread_attr_init(&attr);

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

   /* set the scheduling algorithm to PCS or SCS */
   pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);

   /* create the threads */
   for (i = 0; i < NUM_THREADS; i++)
       pthread_create(&tid[i],&attr,runner,NULL);

   /* now join on each thread */
   for (i = 0; i < NUM_THREADS; i++)
       pthread_join(tid[i], NULL);
}

/* Each thread will begin control in this function */
void *runner(void *param)
{
   /* do some work ... */

   pthread_exit(0);
}
```

Multiple-Processor Scheduling

- CPU scheduling more complex when multiple CPUs are available
- *Homogeneous processors* within a multiprocessor
    - Typically each processor maintains its own private queue of processes (or threads) all of which are available to run
- *Load sharing*
- *Asymmetric multiprocessing*
    - Only one processor accesses the system data structures, alleviating the need for data sharing

Approaches to Multiple-Processor Scheduling

- We assume homogeneous processors (identical in functionality) and uniform memory access (UMA)
- If several identical processors are available, then load sharing can occur, with a **common ready queue**
- Processes in the queue are scheduled to any available processor
- One of two scheduling approaches may be used:
    - Each processor is **self-scheduling**, and selects a process from the common ready queue to execute
    - One processor is appointed as scheduler for the other processors, creating a **master-slave** structure
- Some systems carry the master-slave structure further by having all scheduling decisions, I/O processing, and other system activities handled by one single processor – the **master server**

- This **asymmetric multiprocessing** is simpler than **symmetric multiprocessing (SMP)**, because only one processor accesses the system data structures, alleviating the need for data sharing
- It isn't as efficient, because I/O processes may bottleneck on the one CPU that is performing all of the operations
- Typically, asymmetric multiprocessing is implemented 1st within an OS, and then upgraded to symmetric as the system evolves

Processor Affinity

- Processor affinity:
    - Migration of processes to another processor is avoided because of the cost of invalidating the process and repopulating the processor cache
- **Soft** affinity:
    - When an OS try to keep a process on one processor because of policy, but cannot guarantee it will happen
- **Hard** affinity:
    - When an OS have the ability to allow a process to specify that it is not to migrate to other processors

Load Balancing

- Load balancing attempts to keep the workload evenly distributed across all processors in an SMP system
- Two **migration** approaches:
    - **Push** migration
        - A specific task checks the load on each processor and if it finds an imbalance it evenly distributes the load to less-busy processors
    - **Pull** migration
        - A idle processor pulls a waiting task from a busy processor

Multicore Processors

- Complicated scheduling issue

Virtualization and Scheduling

Operating System Examples

Algorithm Evaluation

- p.213 TB
- Deterministic modeling:
    - Takes a particular predetermined workload and defines the performance of each algorithm for that workload
- Queuing models
- Implementation

Deterministic Modeling

- A method that takes a particular predetermined workload and defines the performance of each algorithm for that workload
- Simple; fast; exact numbers, so algorithms can be compared

- However, it requires exact numbers for input, and its answers apply to only those cases
- The main uses of deterministic modeling are in describing scheduling algorithms and providing examples
- Good if you're running the same programs over and over again
- Over many examples, deterministic modeling may indicate trends
- In general, deterministic modeling is too specific, and requires too much exact knowledge to be useful

Queuing Models

- You can determine the distribution of CPU and I/O bursts
- A formula describes the probability of a particular CPU burst
- The computer system is described as a network of servers
- Each server has a queue of waiting processes
- Knowing arrival & service rates, we can compute utilization, average queue length, wait time… (= queuing-network analysis)
- Limitations of queuing analysis:
    - The algorithms that can be handled are limited
    - The math of complicated algorithms can be hard to work with
    - It is necessary to make assumptions that may not be accurate
    - As a result, the computed results might not be accurate

Simulations

- Involve programming a model of the computer system
- Software data structures represent the major system components
- The simulator has a variable representing a clock
- As this variable's value is increased, the simulator modifies the system state to reflect the activities of the devices, the process, and the scheduler
- As the simulation executes, statistics that indicate algorithm performance are gathered and printed
- A random-number generator is programmed to generate processes, CPU-burst times… according to probability distributions
- The distributions may be defined mathematically or empirically
- If the distribution is to be defined empirically, measurements of the actual system under study are taken
- The results are used to define the actual distribution of events in the real system, and this distribution can then be used to drive the simulation
- Trace tapes can be used to record the sequence of actual events
- **Disadvantages**:
    - Simulations can be costly, requiring hours of computer time
    - Traced tapes can require large amounts of storage space
    - The design, coding, and debugging of the simulator can be a major task

Implementation

- The only completely accurate way to evaluate a scheduling algorithm is to code it, put it in the OS, and see how it works
- The major difficulty is the cost of this approach

- The environment in which the algorithm is used will change

Summary

PART THREE: PROCESS COORDINATION

Chapter 6: Synchronization

- Co-operating process = one that can affect / be affected by other processes.
- Co-operating processes may either directly share a logical address space (i.e. code & data) , or share data through files or messages through threads (ch4).
- Concurrent access to shared data can result in inconsistencies
- **Objectives**:
    1. To introduce the critical-section problem, whose solutions can be used to ensure the consistency of shared data
    2. To present both software and hardware solutions of the critical- section problem
    3. To introduce the concept of an atomic transaction and describe mechanisms to ensure atomicity

Background

- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- Suppose that we wanted to provide a solution to the **consumer-producer** problem that fills *all* the buffers. We can do so by having an integer *count* that keeps track of the number of full buffers. Initially, count is set to 0. It is incremented by the producer after it produces a new buffer and is decremented by the consumer after it consumes a buffer
    - **Producer**:

```
while (count == BUFFER_SIZE)
   ; // do nothing

// add an item to the buffer
++count;
buffer[in] = item;
in = (in + 1) % BUFFER_SIZE;
```

    - **Consumer**:

```
while (count == 0)
   ; // do nothing

// remove an item from the buffer
--count;
item = buffer[out];
out = (out + 1) % BUFFER_SIZE;
```

- **Race condition**:
    - When the outcome of the execution depends on the particular **order** in which data access takes place
- **Example**:
    - count++ could be implemented as

        register1 = count

register1 = register1 + 1

count = register1

- count-- could be implemented as

  register2 = count

  register2 = register2 -1

  count = register2

- Consider this execution interleaving with "count = 5" initially:

  S0: producer execute register1 = count{register1 = 5}

  S1: producer execute register1 = register1 + 1 {register1 = 6}

  S2: consumer execute register2 = count{register2 = 5}

  S3: consumer execute register2 = register2 -1{register2 = 4}

  S4: producer execute count = register1{count = 6 }

  S5: consumer execute count = register2{count = 4}

The Critical-Section Problem

- Critical section = a segment of code in which a process may be changing common variables, updating a table, writing a file, etc
  - **Entry section**
    - Requests permission to enter the critical section
  - **Critical section**
    - Mutually exclusive in time (no other process can execute in its critical section)
  - **Exit section**
    - Follows the critical section
  - **Remainder section**
- ***A solution to the critical-section problem must satisfy***:
  - **Mutual exclusion**
    - Only one process can be in its critical section
  - **Progress**
    - Only processes that are not in their remainder section can enter their critical section, and the selection of a process cannot be postponed indefinitely
  - **Bounded waiting**
    - There must be a bound 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 the request is granted

- **Structure of a typical process**:

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

Peterson's Solution

- This is an example of a software solution that can be used to prevent race conditions
- Two process solution
- Assume that the LOAD and STORE 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 Pi is ready!
- Algorithm for process Pi:

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

- To prove that this solution is correct we show that:
  - **Mutual exclusion** is preserved
  - The **progress requirement** is satisfied
  - The **bounded-waiting requirement** is met

Synchronization Hardware

- Hardware can also be used to solve the critical-section problem
- If in a uni-processor environment interrupts were disabled, no unexpected modifications would be made to shared variables
- **Disabling interrupts** in a multi-processor environment **isn't feasible**, so many machines provide special hardware instructions

- Instead, we can generally state that any solution to the critical-section problem requires a simple tool, a **lock**
    - Race conditions are prevented by requiring that critical regions be protected by locks

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

- Modern machines provide special atomic hardware instructions
    - **Atomic = non-interruptible**
    - Either test memory word and set value
    - Or swap contents of two memory words
- These instructions allow us either to test & modify the content of a word, or to swap the contents of two words, atomically
    - TestAndSet

```
boolean TestAndSet( boolean *target ) {
  boolean rv = *target;
  *target = true;
  return rv;
}
```

    - NB characteristic: this instruction is executed atomically, so if two TestAndSet instructions are executed simultaneously (on different CPUs), they will be executed sequentially
    - TestAndSet with **mutual exclusion**

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

    - lock is initialized to false
    - Swap

```
void swap( boolean *a, boolean *b ) {
  boolean temp = *a;
  *a = *b;
  *b = temp;
```

```
}
```

- Swap with mutual-exclusion

```
do{
  key = true;
  while(key == true )
    swap(& lock, &key );
  // critical section
  lock = false;
  // remainder section
} while(true);
```

- lock is initialized to false
- Bounded-waiting mutual exclusion with TestAndSet

```
do{
  waiting[i] = true;
  key = true;
  while( waiting[i] && key }
    key = TestAndSet( &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);
```

- Common data structures are

```
boolean waiting[n];
boolean lock;
```

- Data structures initialized to false

- To **prove** that the mutual-exclusion requirements is met:
  - note that Pi can enter its critical section only if either waiting[i] == false or key == false
  - key can become false only if the TestAndSet() is executed
    - first process to execute TestAndSet() will find key == false, all others must wait

    - waiting[i] can become false only if another process leaves its critical section
        - only one waiting[i] is set to false
- To **prove** the Progress requirement is met:
    - The mutual exclusion arguments apply, since they let a process that is waiting to enter its critical section proceed
- To **prove** the Bounded waiting requirement is met:
    - When a process leaves its critical section, it scans the waiting array in the cyclic ordering (i+1, i+2…, n-1, 0…, i-1) and 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

Semaphores

- Semaphore = a **synchronization tool** used to control **access to shared variables** so that only one process may at any point in time **change** the value of the shared variable
- A semaphore S is an integer variable that is accessed only through two standard atomic operations: wait and signal

```
wait(s){
  while(s<=0) ;
   ;//no-op
s--;
 }
 signal(s){
s++;
 }
```

- Modifications to the integer value of the semaphore in the wait and signal operations must be executed indivisibly (atomic)

Usage

- Counting semaphores can allow n processes to access (e.g. a database) by initializing the semaphore to n
- Binary semaphores (with values 0 & 1) are simpler to implement
- N processes share a semaphore, mutex (*mut*ual *ex*clusion), initialized to 1
- Each process is organized as follows:

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

- Example on p.235 mid

Implementation

- p.235 - p.238

- Disadvantage of these mutual-exclusion solutions: they all require **busy waiting** (i.e. processes trying to enter their critical sections must **loop continuously** in the entry code)
- This **wastes CPU cycles** that another process might be able to use
- This type of semaphore is also called a **spinlock** (because the process 'spins' while waiting for the lock)
- Advantage of a spinlock: **no context switch** is required when a process must wait on a lock (Useful for short periods)
- To overcome the need for busy waiting, we can modify the definition of the wait and signal semaphore operations so that rather than busy waiting, the process can **block itself**:
    - The process is placed into a waiting queue
    - The state of the process is switched to the waiting state
    - Control is transferred the CPU scheduler
    - The CPU scheduler selects another process to execute
- The critical aspect of semaphores is that they must be executed **atomically**, i.e. wait & signal operations can't execute together
- This (critical-section) problem can be solved in two ways:
    - In a uni-processor environment
        - Inhibit interrupts when the wait and signal operations execute
        - Only the current process executes, until interrupts are re-enabled and the scheduler regains control
    - In a multiprocessor environment
        - Inhibiting interrupts doesn't work
        - Use the hardware / software solutions described above

Deadlocks and Starvation

- Deadlock state = when every process in a set is waiting for an event that can be caused only by another process in the set
- Implementing a semaphore with a waiting queue may result in two processes each waiting for the other one to **signal**
- **Resource acquisition** and **release** are the events concerned here
- **Starvation (indefinite blocking)** = when processes wait indefinitely within the semaphore

Priority Inversion

- Priority inversion = when a high-priority process needs data currently being accessed by a lower-priority one

Classic Problem of Synchronization

Bounded-Buffer Problem

- There is a pool of n buffers, each capable of holding one item
- The mutex semaphore provides mutual exclusion for access to the buffer pool and is initialized to 1
- The empty & full semaphores count the no of empty & full buffers
- Symmetry: The producer produces full buffers for the consumer / the consumer produces empty buffers for the producer
- p.240 TB

The Readers-Writers Problem

- p.241 TB
- 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
- Many readers can access shared data without problems
- Writers need **exclusive use** to shared objects
- First readers-writers problem:
    - Readers don't wait, unless a writer has permission
    - Problem: writers may starve if new readers keep appearing because the readers are granted shared access each time
- Second readers-writers problem:
    - If a writer is ready, no new readers may start reading
    - Problem: readers may starve
- Used to provide reader-writer locks on some systems
- The mode of lock needs to be specified
    - read access
    - write access
- Reader-writer locks most useful in following situations:
    - In applications where it is easy to identify which processes only read shared data and which processes only write shared data
    - In applications that have more readers than writers. This is because reader-writer locks generally require more overhead to establish than semaphores or mutual-exclusion locks. The increased concurrency of allowing multiple readers compensates for the overhead involved in setting up the reader-writer lock

The Dining-Philosophers Problem

- p.242 TB
- There are 5 philosophers with 5 chopsticks (semaphores)
- A philosopher is either eating (with two chopsticks) or thinking
- A philosopher tries to grab a chopstick by executing a wait operation on that semaphore, and releases the chopsticks by executing the signal operation on the appropriate semaphores
- The shared data are: semaphore chopstick[5]; where all the elements of chopstick are initialized to 1
- This solution guarantees that no two neighbors are eating simultaneously, but a deadlock will occur if all 5 philosophers become hungry simultaneously and grab their left chopstick
- Some remedies to the deadlock problem:
    - Allow **at most four** philosophers to be sitting simultaneously at the table
    - Allow a philosopher to pick up chopsticks **only if both are available** (He must pick them up in a critical section)

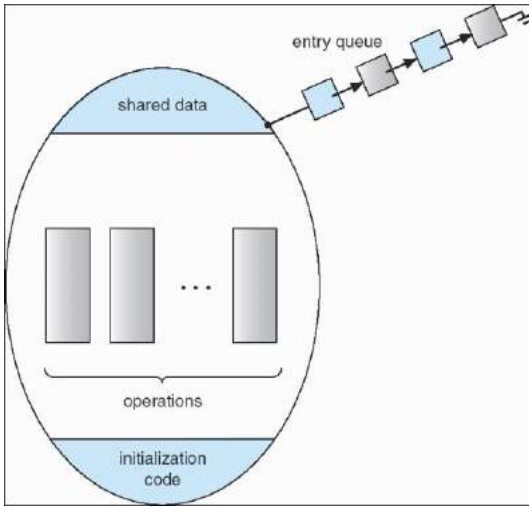
    - Use an **asymmetric solution**: An odd philosopher picks up his left chopstick first, and an even one the right one first
- A deadlock-free solution doesn't necessarily eliminate the possibility of starvation
- Monitors is a solution to the dining-philosophers problem

Monitors

- p.244-245 TB
- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- Only one process may be active within the monitor at a time
- Monitors are needed because if many programmers are using a semaphore and one programmer forgets to signal after the program has left the critical section, then the entire synchronization mechanism will end up in a deadlock
- Definition: A collection of procedures, variables, and data structures that are all grouped together in a module / package

Usage

- p.245-246 TB
- A monitor type is presents a set of programmer-defined operations that are provided to ensure mutual exclusion within the monitor
- Three distinguishing characteristics of a monitor:
    - It **encapsulates** its permanent variables
    - Procedures execute in **mutual exclusion**
    - Synchronization is provided via **condition variables**
- The monitor shouldn't access any non-local variables, and local variables shouldn't be accessible from outside the monitor
- Any process may call the monitor, but **only one** process at any point in time may be **executing inside** the monitor



Dining-Philosophers Solution Using Monitors

- p.248 - p.249

Implementing a Monitor Using Semaphores

Resuming Processes within a Monitor

Synchronization Examples

Atomic Transactions

System Model

- Assures that operations happen as a single logical unit of work, in its entirety, or not at all
- Related to field of database systems
- Challenge is assuring atomicity despite computer system failures
- Transaction - collection of instructions or operations that performs single logical function
    - Here we are concerned with changes to stable storage - disk
    - Transaction is series of read and write operations
    - Terminated by commit(transaction successful) or abort(transaction failed) operation
    - Aborted transaction must be rolled back to undo any changes it performed
- To determine how the system should ensure atomicity, we need first to identify the properties of devices used for storing the various data accessed by transactions
    - Volatile storage - information stored here does not survive system crashes
        - Example: main memory, cache
    - Nonvolatile storage -Information usually survives crashes
        - Example: disk and tape
    - Stable storage - Information never lost
        - Not actually possible, so approximated via replication or RAID to devices with independent failure modes
- Goal is to assure transaction atomicity where failures cause loss of information on volatile storage

Log-Based Recovery

- Write-ahead logging: Each log record describes a single operation of a transaction write and has these fields:
    - Transaction name (The unique name of the transaction)
    - Data item name (The unique name of the item written)
    - Old value (The value of the data prior to the write)
    - New value (The value that the data will have afterwards)
- Prior to a write being executed, the log records must be written onto stable storage
- Performance penalty: Two physical writes are required for each logical write requested, and more storage is needed
- Two procedures used by the recovery algorithm:
    - undo – restores the data to the old values
    - redo – sets the data to the new values
- The set of data updated by the transaction and their respective old and new values can be found in the log

Checkpoints

- When a failure occurs, we must consult the log to determine those transactions that need to be redone / undone
- Drawbacks to searching the entire log:
    - The search is time-consuming
    - Redoing data modifications causes recovery to take longer
- To reduce this overhead, the system performs checkpoints:
    - Output all log records onto stable storage
    - Output all modified data to the stable storage
    - Output a log record <checkpoint> onto stable storage
- The presence of a <checkpoint> record in the log allows streamlined recovery, since you search for the last checkpoint

Concurrent Atomic Transactions

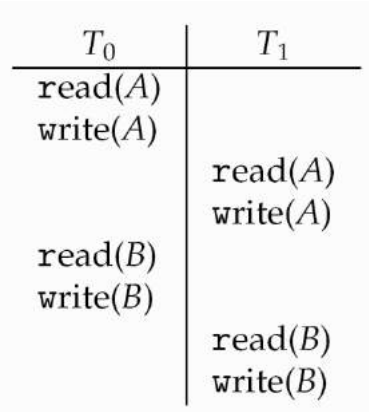
- Serializability = when transactions are executed serially
    - Can be maintained by executing each transaction within a critical section
- All transactions could share a semaphore mutex, initialized to 1
    - When the transaction starts, it first executes wait
    - After the transaction commits / aborts, it executes signal
- This scheme ensures **atomicity** of all concurrently executing transactions, but is still too restrictive
- Concurrency-control algorithms to ensure serializability

Serializability

- **Serial schedule**: each transaction executes atomically
- **Example**:
    - Consider two data items A and B
    - Consider Transactions T0 and T1
    - Execute T0, T1atomically
    - Execution sequence called schedule
    - Atomically executed transaction order called serial schedule
    - For N transactions, there are N! valid serial schedules
- Schedule 1: T0 then T1

T_0	T_1
read(A)	
write(A)	
read(B)	
write(B)	
	read(A)
	write(A)
	read(B)
	write(B)

- **Non-serial schedule**: transactions overlap execution
  - Resulting execution not necessarily incorrect
- Consider schedule S, operations Oi, Oj
  - Conflict if access same data item, with at least one write
- If Oi, Oj consecutive and operations of different transactions & Oi and Oj don't conflict
  - Then S' with swapped order Oj Oi equivalent to S
- If S can become S' via swapping non conflicting operations
  - S is conflict serializable



T_0	T_1
read(A)	
write(A)	
	read(A)
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)

Locking Protocol

- Ensure serializability by associating lock with each data item
  - Follow locking protocol for access control
- Locks
  - **Shared**
    - It has shared-mode lock (S) on item Q, It can read Q but not write Q
  - **Exclusive**
    - Ti has exclusive-mode lock (X) on Q, Tican read and write Q
- Require every transaction on item Q acquire appropriate lock
- If lock already held, new request may have to wait
  - Similar to readers-writers algorithm

Timestamp-Based Protocols

- Select order among transactions in advance -timestamp-ordering
- Transaction Ti associated with timestamp TS(Ti) before Ti starts
  - TS(Ti) < TS(Tj) if Ti entered system before Tj
  - TS can be generated from system clock or as logical counter incremented at each entry of transaction
- Timestamps determine serializability order
  - If TS(Ti) < TS(Tj), system must ensure produced schedule equivalent to serial schedule where It appears before Tj

T_2	T_3
read(B)	
	read(B)
	write(B)
read(A)	
	read(A)
	write(A)

Summary

Chapter 7: Deadlocks

- **Objectives:**
  - To develop a description of deadlocks, which prevents sets of concurrent processes from completing their tasks
  - To present a number of different methods for preventing or avoiding deadlocks in a computer system

System Model

- Computer resources are partitioned into several types (e.g. memory space, CPU cycles, files, I/O devices…)
- Each type consists of some number of identical instances (e.g. if you have two CPUs, 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 will satisfy the request
- A process may utilize a resource in only this sequence: (p.284 TB)
  - **Request**:
    - The process requests the resource
    - If the request cannot be granted immediately (for example, if the resource is being used by other process), then the requesting process must **wait** it can acquire the resource
  - **Use**:
    - The process can operate on the resource (for example, if the resource is a printer, the process can print on the printer)
  - **Release**:
    - The process releases the resource

Deadlock Characterization

- A deadlocked state occurs when two or more processes are waiting indefinitely for an event that can be caused only by one of the waiting processes

Necessary Conditions

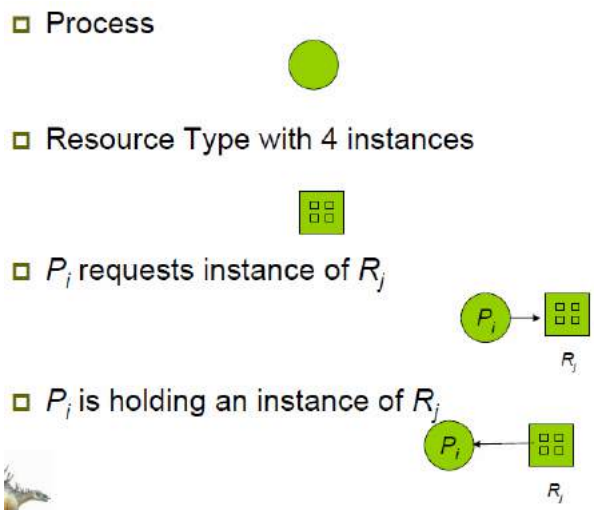
- A deadlock situation can arise if **all** these situations hold **simultaneously**:
  - **Mutual exclusion**
    - At least one resource must be held in a non-sharable mode
  - **Hold and wait**
    - A process must hold at least one resource and be waiting

- **No preemption**
  - A resource can be released only voluntarily by a process
- **Circular wait**
  - In a set of waiting processes, all are waiting for a resource held by another process in the set

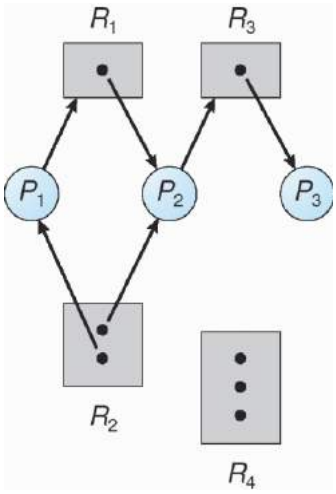
- All four conditions must hold for a deadlock to occur

Resource-Allocation Graph

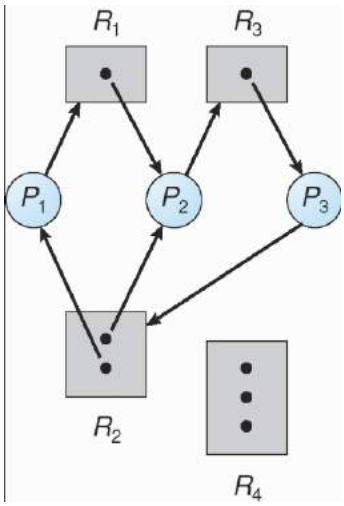
- p.287 - p.289 TB
- Deadlocks can be described more precisely in terms of a directed graph called a system resource-allocation graph
- It consists of the following parts:
  - A set of vertices *V* and a set of edges *E*
    - V is partitioned into two types:
      - *P*= {*P*1, *P*2, …, *Pn*}, the set consisting of all the processes in the system
      - *R*= {*R*1, *R*2, …, *Rm*}, the set consisting of all resource types in the system
    - **request edge** - directed edge *P*1 →*Rj*
      - From a process to a resource
    - **assignment edge** - directed edge *Rj*→*Pi*
      - From a resource to a process



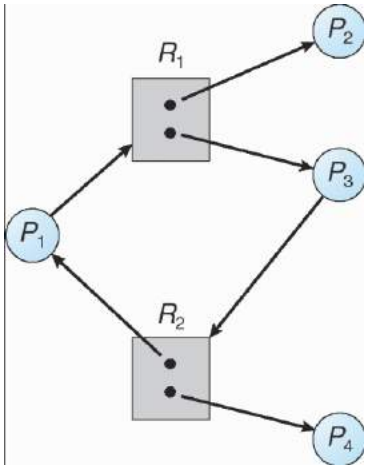
- **Example** of a Resource Allocation Graph:



- If a resource-allocation graph doesn't have a cycle, then the system is not in a deadlock state
- If there is a cycle, then the system may / may not be in a deadlock state
- Resource Allocation Graph with a deadlock:



- Graph with a Cycle but no deadlock:



- **Basic facts about Resource Allocation Graphs**:
    - If graph contains no cycles ⇒ no deadlock