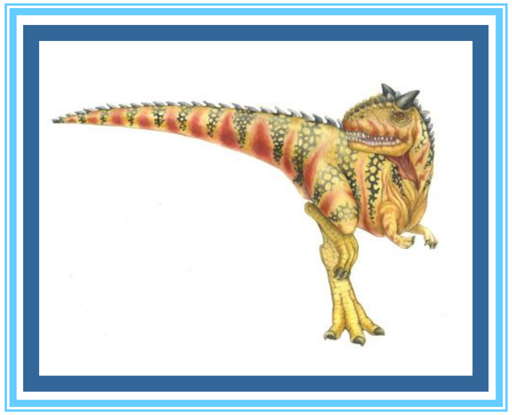
Chapter 4: Threads



Chapter 4: Threads

- Overview
- Multicore Programming
- Multithreading Models
- Thread Libraries
- Threading Issues

Objectives

- To introduce the notion of a **thread—a fundamental unit of CPU utilization** that forms the basis of multithreaded computer systems
- To discuss the APIs for the **Pthreads, Windows, and Java thread libraries**
- To examine issues related to **multithreaded programming**
- To cover operating **system support for threads** in Linux

Overview

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

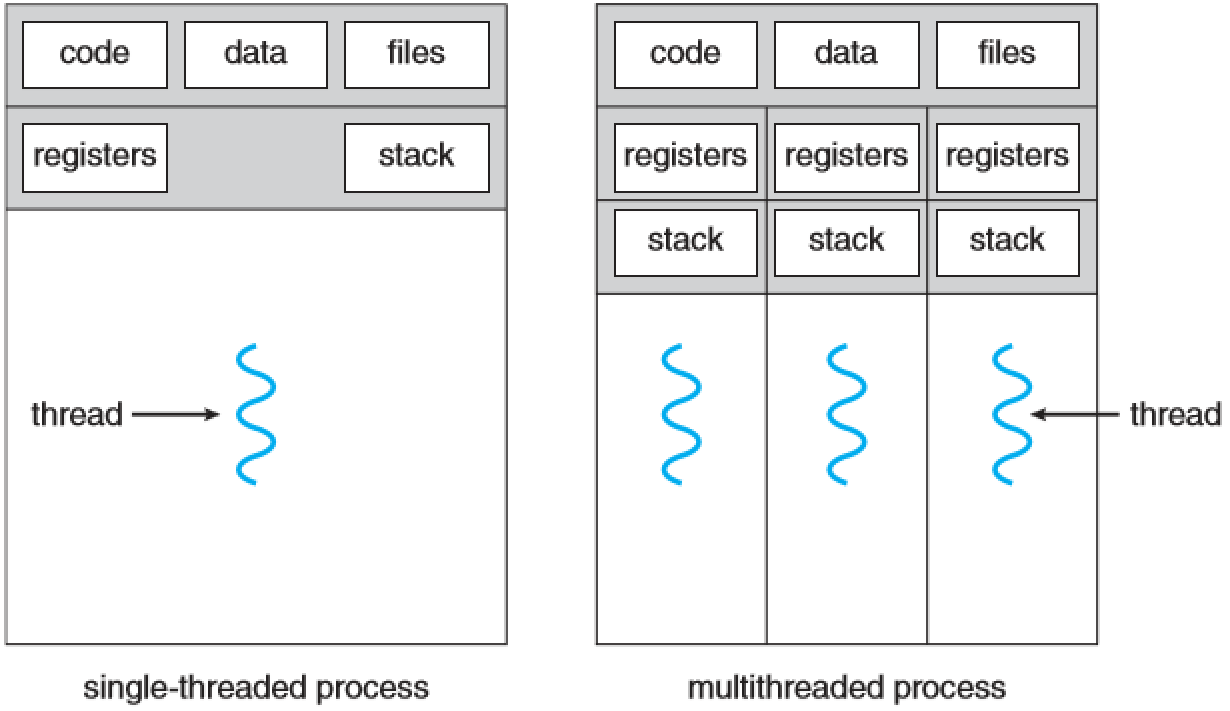
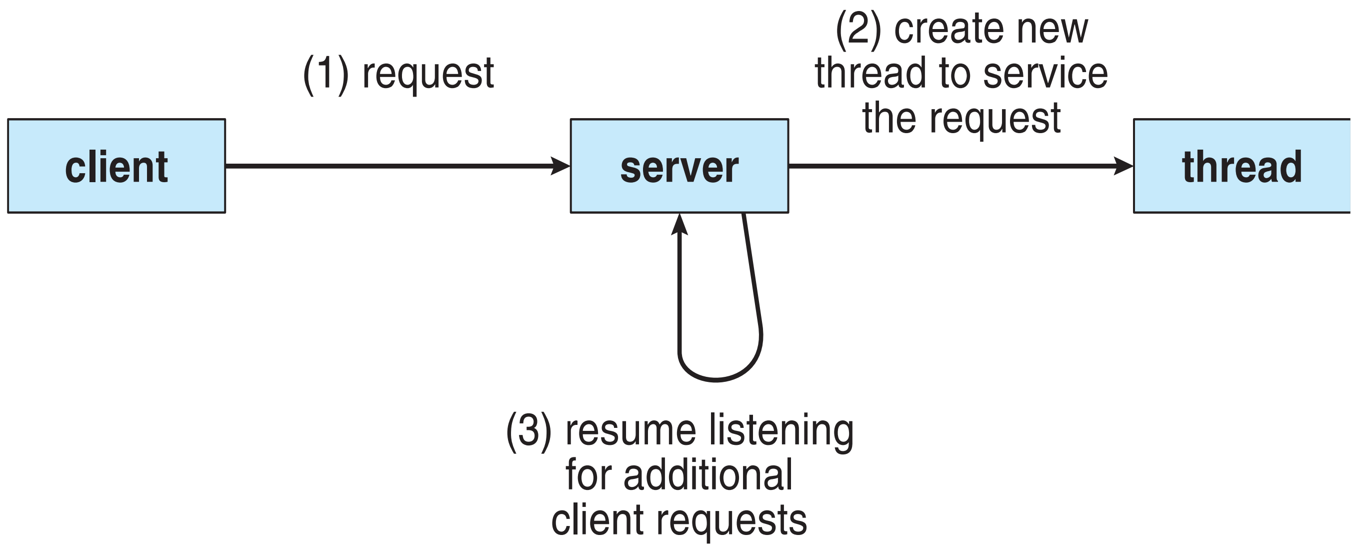


Figure 4.1 Single-threaded and multithreaded processes.

Motivation

- Most **modern applications** are multithreaded
- Threads run within application
- **Multiple tasks** with the application can be implemented by separate threads
  - Update display
  - Fetch data
  - Spell checking
  - Answer a network request
- Process creation is heavy-weight (time consuming and resource intensive) while thread creation is light-weight (because threads share the code, data and OS resources)
- If the web-server process is multithreaded, the server will create a separate thread that listens for client requests.
- When a request is made, rather than creating another process, the server creates a new thread to service the request and resume listening for additional requests.

Multithreaded Server Architecture


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

- If the web server ran as a traditional single-threaded process, it would be able to service only **one client at a time, and a client might have to wait a very long time** for its request to be serviced.

- Threads can simplify code, increase efficiency
- Kernels are generally multithreaded (Several threads operate in the kernel, and each thread performs a specific task, such as managing devices, managing memory, or interrupt handling)

Benefits

- **Responsiveness –** may allow continued execution if part of process is blocked, especially important for user interfaces
  - For instance, consider what happens when a user clicks a button that results in the performance of a time-consuming operation.
  - A single-threaded application would be unresponsive to the user until the operation had completed.
  - In contrast, if the time-consuming operation is performed in a separate thread, the application remains responsive to the user.
- **Resource Sharing –**
  - **Processes can only share resources through techniques such as shared memory and message passing.**
  - **must be explicitly arranged by the programmer**
  - threads share resources of process, easier than shared memory or message passing

- **Economy –** cheaper than process creation, thread switching lower overhead than context switching
  - because threads share the resources of the process to which they belong
- **Scalability –** process can take advantage of multiprocessor architectures
  - A single-threaded process can run on only one processor, regardless how many are available.

Multicore Programming

- **Multicore** or **multiprocessor** systems putting pressure on programmers, (to make better use of the multiple computing cores)
- challenges include:
  - **Dividing activities (in such a way that allow parallel execution of these activities)**
  - **Balance (programmers must also ensure that the tasks perform equal work of equal value)**
  - **Data splitting (the data accessed and manipulated by the tasks must be divided to run on separate cores)**
  - **Data dependency (when one task depends on data from another, programmers must ensure that the execution of the tasks is synchronized (more on this in chap 5))**
  - **Testing and debugging (since many different execution paths are possible, testing and debugging such concurrent programs is inherently more difficult)**

- ***Parallelism*** implies a system can perform more than one task **simultaneously** (see Figure 4.4)
- ***Concurrency*** allows **more than one task to make progress**
  - Single processor / core, **scheduler** providing concurrency (see Figure 4.3)

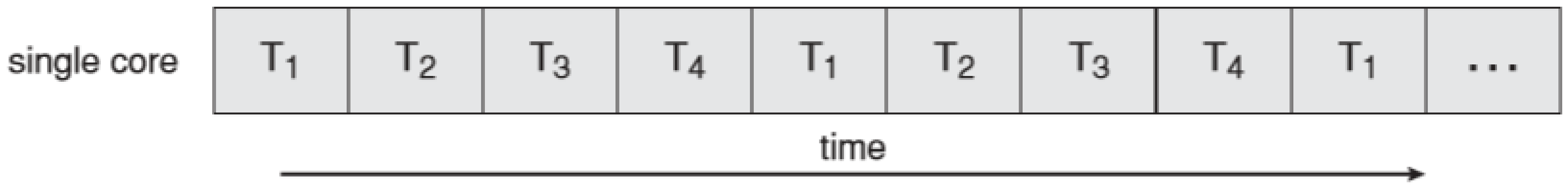


Figure 4.3 Concurrent execution on a single-core system.

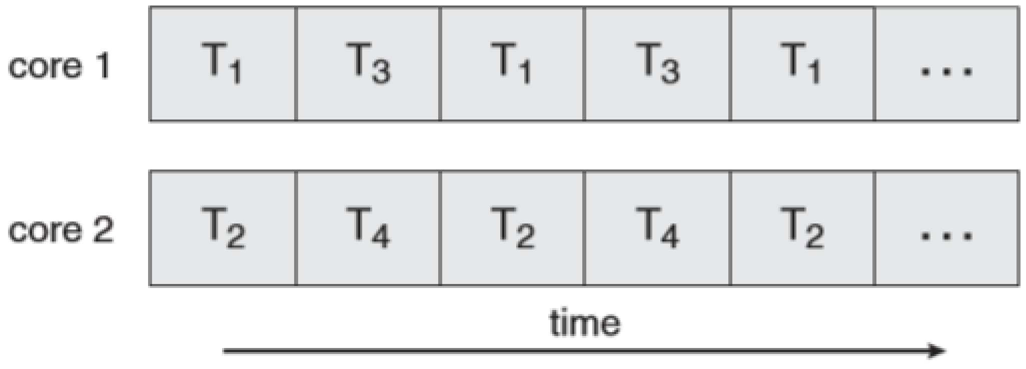

core 1
T_1 T_3 T_1 T_3 T_1 . . .
core 2
T_2 T_4 T_2 T_4 T_2 . . .
time

Figure 4.4 Parallel execution on a multicore system.

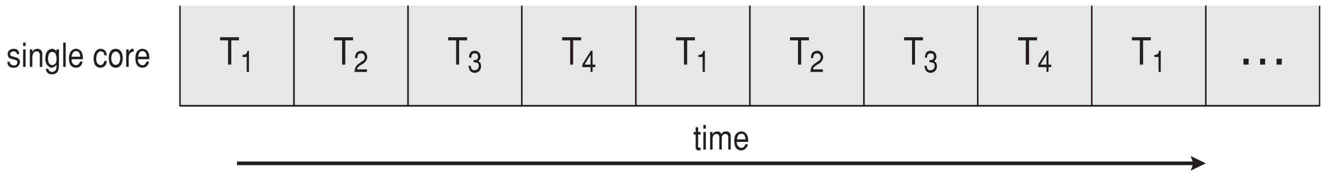
Multicore Programming (Cont.)

- Types of parallelism
  - **Data parallelism** – distributes subsets of the same data across multiple cores, same operation on each
    - **Consider, for example, summing the contents of an array of size N. On a single-core system, one thread would simply sum the elements[0] ...[N−1].**
    - **On a dual-core system, however, thread A, running on core 0, could sum the elements [0] ...[N/2−1] while thread B, running on core 1, could sum the elements [N/2]...[N−1]. The two threads would be running in parallel on separate computing cores.**
  - **Task parallelism** – distributing threads across cores, each thread performing unique operation

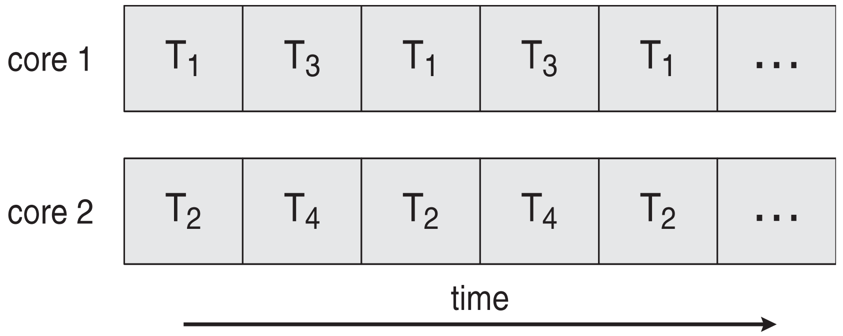
- As # of threads grows, so does **architectural support for threading (for fast context switching)**
  - CPUs have cores as well as ***hardware threads***
  - Consider Oracle SPARC T4 with 8 cores, and 8 hardware threads per core

Concurrency vs. Parallelism

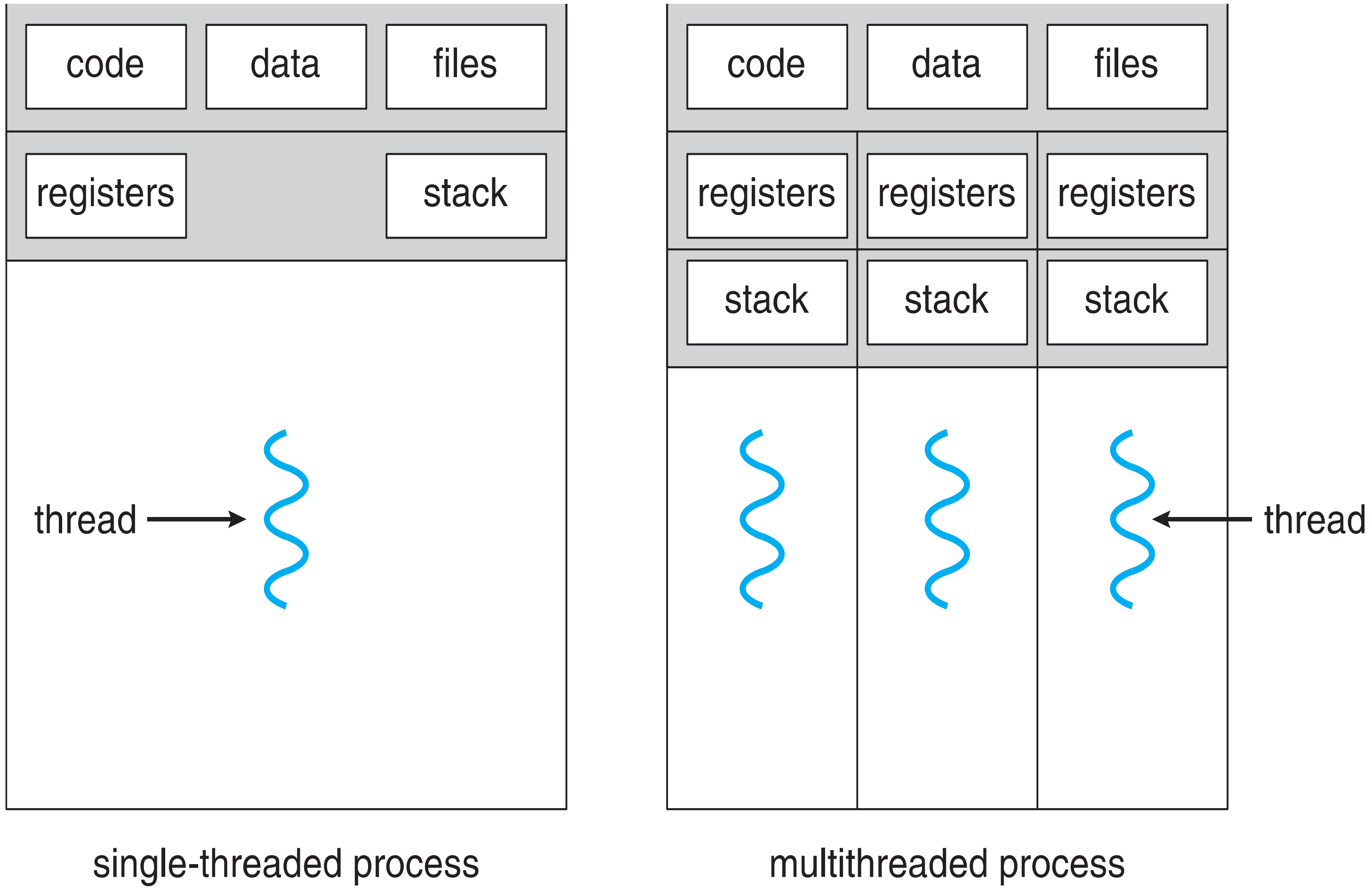
- **Concurrent execution on single-core system:**



- **Parallelism on a multi-core system:**



Single and Multithreaded Processes



Amdahl's Law

- Identifies performance gains from adding additional cores to an application that has both serial and parallel components
- *S* is serial portion (things that must be done sequentially)
- *N* processing cores

$$speedup \leq \frac{1}{S + \frac{(1-S)}{N}}$$

- That is, if application is 75% parallel / 25% serial, moving from 1 to 2 cores results in speedup of 1.6 times
- As *N* approaches infinity, speedup approaches 1 / *S*

Serial portion of an application has disproportionate effect on performance gained by adding additional cores

- But does the law take into account contemporary multicore systems?

User Threads and Kernel Threads

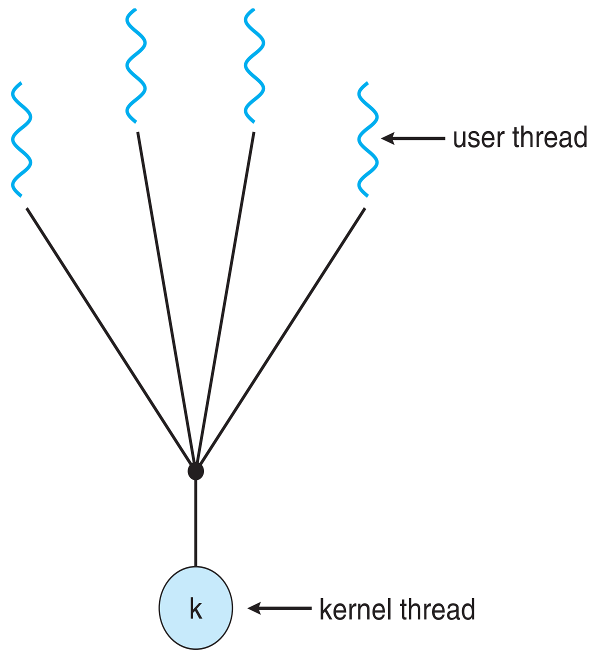
- **User threads** - management done by user-level threads library (without kernel support)
- Three primary thread libraries:
  - POSIX **Pthreads**
  - Windows threads
  - Java threads
- **Kernel threads** - Supported by the Kernel (supported and managed directly by the operating system)
- Examples – virtually all general purpose operating systems support kernel threads, including:
  - Windows
  - Solaris
  - Linux
  - Tru64 UNIX
  - Mac OS X

Multithreading Models

- Ultimately, a relationship must exist between user threads and kernel threads.
- Many-to-One
- One-to-One
- Many-to-Many

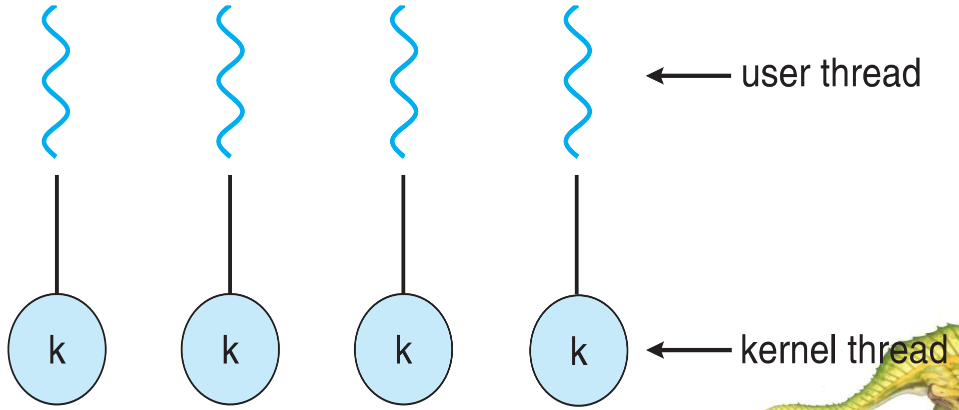
Many-to-One

- Many user-level threads mapped to single kernel thread
- One thread blocking causes all to block
- Multiple threads are **unable to run in parallel on muticore** system because only one may be in kernel at a time
- i.e. the kernel can schedule only one thread at a time
- Few systems currently use this model (because of its inability to take advantage of multiple processing cores)
- Examples:
  - **Solaris Green Threads**
  - **GNU Portable Threads**



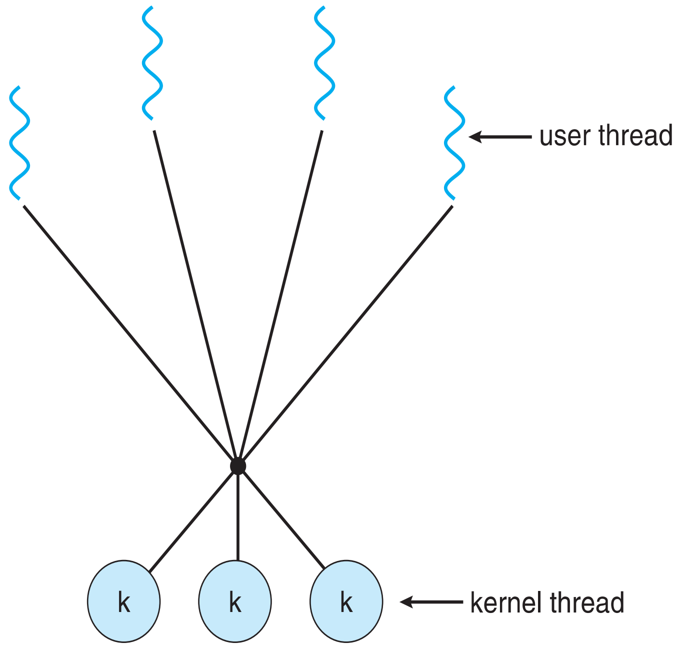
One-to-One

- Each user-level thread maps to kernel thread
- Creating a user-level thread creates a kernel thread
- It also allows multiple threads to run in parallel on multiprocessors
- More concurrency than many-to-one
- The only drawback to this model is that creating a user thread requires creating the corresponding kernel thread
- Number of threads per process sometimes restricted due to overhead
- Examples
  - Windows
  - Linux
  - Solaris 9 and later



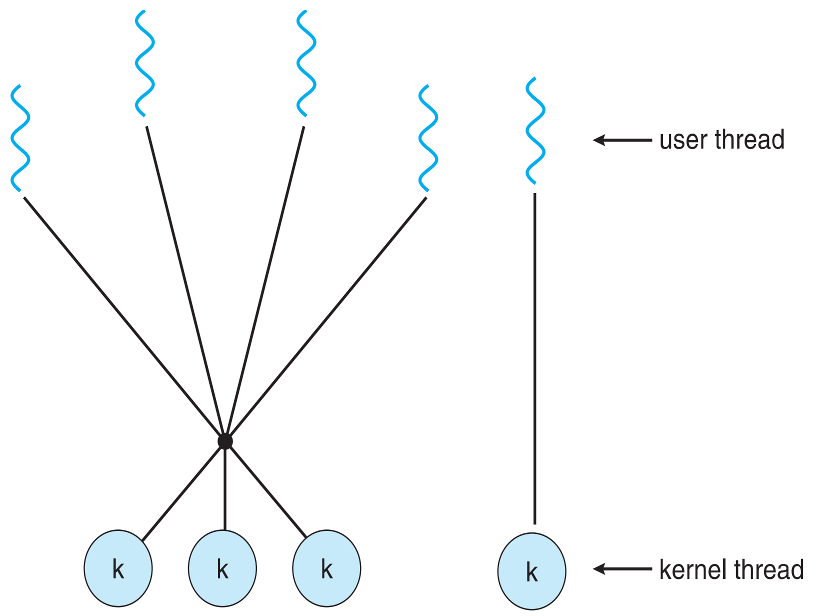
Many-to-Many Model

- Allows many user level threads to be mapped to a smaller or equal number of kernel threads
- Allows the operating system to create a sufficient number of kernel threads
- Solaris prior to version 9
- Windows with the *ThreadFiber* package



Two-level Model

- Similar to M:M, except that it allows a user thread to be **bound** to kernel thread
- Examples
  - IRIX
  - HP-UX
  - Tru64 UNIX
  - Solaris 8 and earlier



Thread Libraries

- **Thread library** provides programmer with API for creating and managing threads
- Two primary ways of implementing
  - Library entirely in user space
    - All code and data structures for the library exist in user space.
    - This means that invoking a function in the library results in a local function call in user space and **not a system call.**
  - Kernel-level library **supported directly by the OS**
    - Invoking a function in the API for the library typically results in **a system call to the kernel**.

- Three main thread libraries are in use today:
  - POSIX Pthreads
  - Win32 threads
  - Java threads.

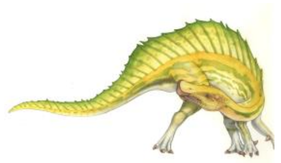
Asynchronous threading and Synchronous threading

- Two general strategies for creating multiple threads:
- Asynchronous threading,
  - once the parent creates a child thread, **the parent resumes its execution**
  - the parent and child **execute concurrently**
  - The parent thread **need not know when its child terminates**
  - Typically **little data sharing** between threads

- Synchronous threading occurs when the parent thread creates one or more children and
  - then must wait for all of its children to terminate before it resumes
  - the so-called **fork-join strategy**.
  - Once each thread has finished its work**, it terminates and joins with its parent**
  - Only after all of the children have joined can **the parent resume execution**
  - Typically, synchronous threading **involves significant data sharing** among threads
  - the parent thread may **combine the results** calculated by its various children**.**
- **All of the following examples use synchronous threading**.

Pthreads

- May be provided **either as user-level or kernel-level**
- The threads extension of the POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
  - **POSIX stands for Portable Operating System Interface**, is a family of standards specified by the IEEE Computer Society for maintaining compatibility between operating systems.
- ***Specification***, not ***implementation***
- 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)
- **Any data declared globally**—that is, declared outside of any function—are **shared among all threads** belonging to the same process.



Example

- As an illustrative example, we design a multi threaded program that performs the summation of a non-negative integer in a separate thread

- For example, if N were 5, this function would represent the summation of integers from 0 to 5, which is 15.
- When this program begins, a single thread of control begins in main().
- After some initialization, **main() creates a second thread** that begins control in the **runner() function**.
- **Both threads share the global data sum**.

Pthreads Example

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

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

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;
   }
```

Pthreads Example (Cont.)

```
    /* 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);
}
```

Pthreads Code for Joining 10 Threads

```
#define NUM_THREADS 10

/* an array of threads to be joined upon */
pthread_t workers[NUM_THREADS];

for (int i = 0; i < NUM_THREADS; i++)
   pthread_join(workers[i], NULL);
```

Java Threads

- Java threads are managed by the JVM
- Typically implemented using the threads model provided by underlying OS
- Because **Java has no notion of global data**, access to **shared data must be explicitly arranged** between threads.
- Java threads may be created by:

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

  - Extending Thread class **or**
  - Implementing the Runnable interface

Creating Threads in Java

- There are two techniques for creating threads in a Java program.
  - One approach is to create a new class that is derived from the Thread class and to override its run() method.
  - An alternative—and more commonly used— technique is to define a class that implements the Runnable interface.
    - The code implementing the run() method is what runs as a separate thread.
    - Thread creation is performed by creating an object instance of the Thread class and passing the constructor a Runnable object.
    - The start() method creates the new thread
    - start() allocates memory and initializes a new thread in the JVM.
    - It also calls the run() method, making the thread eligible to be run by the JVM.
    - The join() method in Java is equivalent to pthread join() and WaitForSingleObject()

Sharing Data Between Threads in Java

- If two or more threads are to share data in a Java program, the sharing occurs by passing references to the shared object to the appropriate threads.
- In our example the main thread and the summation thread share the object instance of the Sum class.
- This shared object is referenced through the appropriate getSum() and setSum() methods

Java Multithreaded Program

```
class Sum
{
  private int sum;

  public int getSum() {
   return sum;
  }

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

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

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

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

Java Multithreaded Program (Cont.)

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

Extra Example on Threads in Java

Threads

2. Creating threads

(1) Inheriting from the `Thread` class

- The general approach is
  - (1) **Define a class** by **extending** the `Thread` class and **overriding** the `run` method.
    - In the run method, you should write the code that you wish to run when this particular thread has started.
  - (2) **Create an instance** of the above class
  - (3) **Start running the instance** using the `start` method that is defined in

```
public class WhereAmI extends Thread {
    int n;
// constructor
    public WhereAmI(int number) {
        n = number;
    }
// override the run method
    public void run() {
        for (int i = 0; i < 100; i++)
            System.out.println("I'm in thread " + n);

    }
}
```

```
public class ThreadTester {
    public static void main(String[] args) {
        // create the threads
        WhereAmI place1 = new WhereAmI(1);
        WhereAmI place2 = new WhereAmI(2);
        WhereAmI place3 = new WhereAmI(3);
        // start the threads
        place1.start();
        place2.start();
        place3.start();
    }
}
```

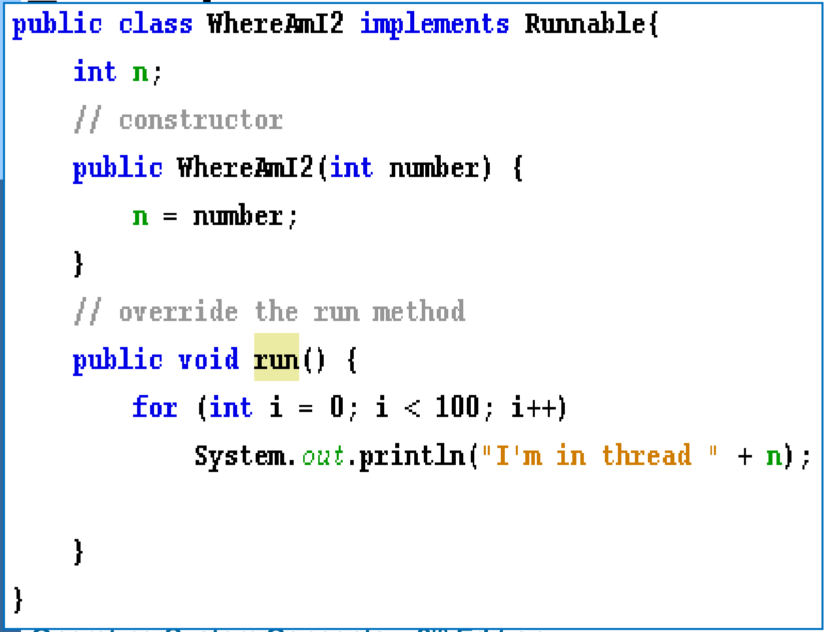
The output

```
I'm in thread 3
I'm in thread 1
I'm in thread 3
I'm in thread 1
I'm in thread 3
I'm in thread 1
I'm in thread 3
I'm in thread 2
I'm in thread 3
I'm in thread 2
I'm in thread 3
I'm in thread 2
```

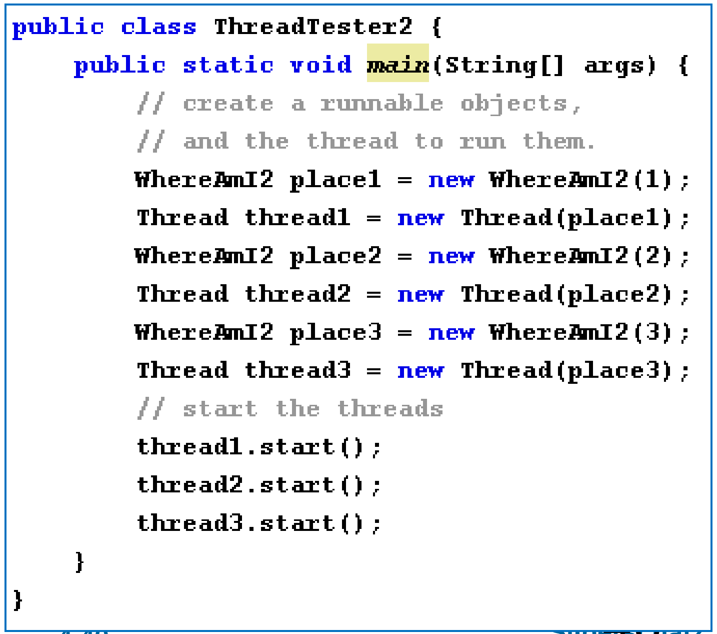
2. Creating threads

(2) Implementing the `Runnable` interface

- The general approach is
  - **(1)** **Define a class** that **implements** `Runnable` and **overriding** the `run` method.
  - **(2)** **Create an instance** of the above class.
  - **(3)** **Create a thread** that runs this instance.
  - **(4)** **Start running the instance** using the `start` method.

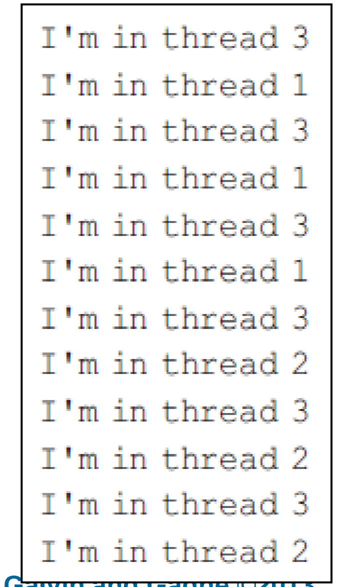


```java
public class WhereAmI2 implements Runnable{
    int n;
    // constructor
    public WhereAmI2(int number) {
        n = number;
    }
    // override the run method
    public void run() {
        for (int i = 0; i < 100; i++)
            System.out.println("I'm in thread " + n);

    }
}
```



```java
public class ThreadTester2 {
    public static void main(String[] args) {
        // create a runnable objects,
        // and the thread to run them.
        WhereAmI2 place1 = new WhereAmI2(1);
        Thread thread1 = new Thread(place1);
        WhereAmI2 place2 = new WhereAmI2(2);
        Thread thread2 = new Thread(place2);
        WhereAmI2 place3 = new WhereAmI2(3);
        Thread thread3 = new Thread(place3);
        // start the threads
        thread1.start();
        thread2.start();
        thread3.start();
    }
}
```

The output



```
I'm in thread 3
I'm in thread 1
I'm in thread 3
I'm in thread 1
I'm in thread 3
I'm in thread 1
I'm in thread 3
I'm in thread 2
I'm in thread 3
I'm in thread 2
I'm in thread 3
I'm in thread 2
```

Threading Issues

- In this section, we discuss some of the issues to consider in designing multithreaded programs

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

Semantics of fork() and exec()

- Does **`fork() (when invoked by a thread)`** duplicate only the calling thread or all threads?
  - Some UNIXes have two versions of fork
- **`exec()`** usually works as normal – replace the running process including all threads.
  - That is, if a thread invokes the exec() system call, the program specified in the parameter to exec() will replace the entire process—including all threads.

Signal Handling

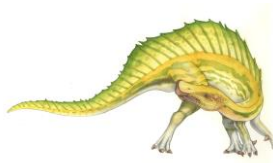
- **Signals** are used in UNIX systems to notify a process that a particular event has occurred.
- A **signal handler** is used to process signals
  1. Signal is generated by particular event
  2. Signal is delivered to a process
  3. Signal is handled by one of two signal handlers:
     1. default
     2. user-defined
- Every signal has **default handler** that kernel runs when handling signal
  - **User-defined signal handler** can override default
  - For single-threaded, signal delivered to process

Synchronous Signals

- A signal may be received either synchronously or asynchronously
- Examples of synchronous signal include
  - illegal memory access and
  - division by 0.
- If a running program performs either of these actions, a signal is generated.
- Synchronous signals are delivered to the same process that performed the operation that caused the signal (that is the reason they are considered synchronous).

Asynchronous Signals

- When a signal is generated by an event external to a running process, that process receives the signal asynchronously.
- Examples of such signals include terminating a process with specific keystrokes (such as **<control><C>)** and having a timer expire.
- Typically, an asynchronous signal is sent to another process.



Signal Handling (Cont.)

- Handling signals in single-threaded programs is straightforward: signals are always delivered to a process.
- However, delivering signals is more complicated in multithreaded programs
- Where should a signal be delivered for multi-threaded?
  - Deliver the signal to the thread to which the signal applies
  - Deliver the signal to every thread in the process
  - Deliver the signal to certain threads in the process
  - Assign a specific thread to receive all signals for the process

Thread Cancellation

- Terminating a thread before it has finished
- For example, if multiple threads are concurrently searching through a database and one thread returns the result, the remaining threads might be canceled.
- Another situation might occur when a user presses a button on a web browser that stops a web page from loading any further.
  - Often, a web page loads using several threads—each image is loaded in a separate thread.
  - When a user presses the stop button on the browser, all threads loading the page are canceled.

Thread Cancellation

- Thread to be canceled is **target thread**
- Two general approaches:
  - **Asynchronous cancellation** terminates the target thread immediately (may not free a necessary system-wide resource)
  - **Deferred cancellation** allows the target thread to periodically check if it should be cancelled, allowing it an opportunity to terminate itself in an orderly fashion.
- Pthread code to create and cancel a thread:

```
pthread_t tid;

/* create the thread */
pthread_create(&tid, 0, worker, NULL);

. . .

/* cancel the thread */
pthread_cancel(tid);
```

Thread Cancellation (Cont.)

- Invoking thread cancellation requests cancellation, but actual cancellation depends on thread state
- A thread cannot be canceled if cancellation is disabled

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

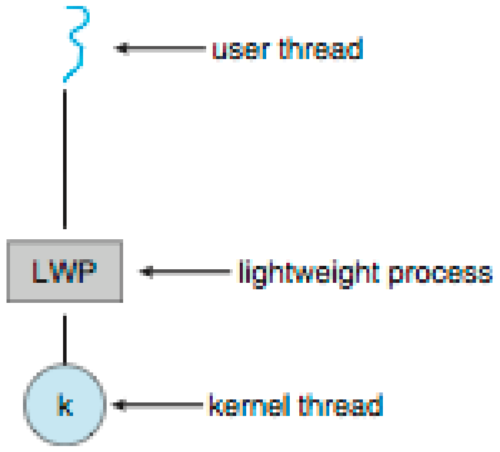
- If thread has cancellation disabled, cancellation remains pending until thread enables it
- Default type is deferred
  - Cancellation only occurs when thread reaches **cancellation point**
    - I.e. **`pthread_testcancel()`**
    - Then **cleanup handler** is invoked
- On Linux systems, thread cancellation is handled through signals

Thread-Local Storage

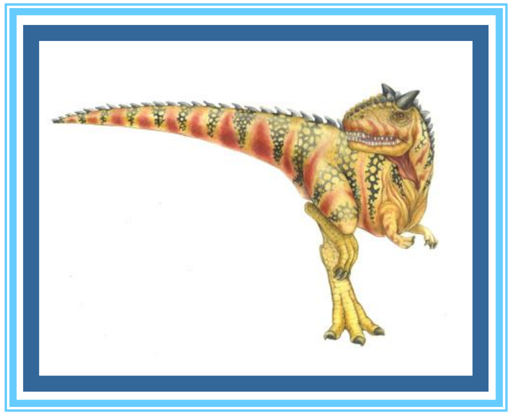
- **Thread-local storage** (**TLS**) allows each thread to have its own copy of data
- For example, in a transaction-processing system, we might service each transaction in a separate thread. Furthermore, each transaction might be assigned a unique identifier. To associate each thread with its unique identifier, we could use thread-local storage.
- Useful when you do not have control over the thread creation process (i.e., when using a thread pool)
- Different from local variables
  - Local variables visible only during single function invocation
  - TLS visible across function invocations
- Similar to `static` data
  - TLS is unique to each thread
- Most thread libraries—including Windows and Pthreads—provide some form of support for thread-local storage; Java providessupport as well

Scheduler Activations

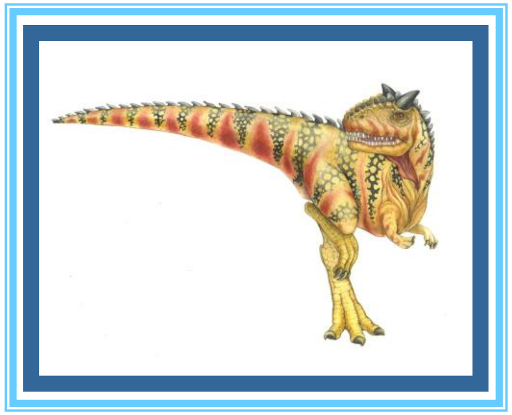
- Both M:M and Two-level models require communication to maintain the appropriate number of kernel threads allocated to the application
- Typically use an intermediate data structure between user and kernel threads – **lightweight process** (**LWP**)
  - Appears to be a virtual processor on which process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



End of Chapter 4



Chapter 5: Process Synchronization



Chapter 5: Process Synchronization

- Background
- The Critical-Section Problem
- Peterson's Solution
- Synchronization Hardware
- Mutex Locks
- Semaphores
- Classic Problems of Synchronization

Objectives

- To present the concept of process synchronization.
- To introduce the critical-section problem, whose solutions can be used to ensure the consistency of shared data
- To present both software and hardware solutions of the critical-section problem
- To examine several classical process-synchronization problems

Background

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

Producer

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

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

Consumer

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

Race Condition

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

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

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

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

- Consider this execution interleaving with “count = 5” initially:

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

Race Condition

- We would arrive at this incorrect state because we allowed both processes **to manipulate the variable counter concurrently**.
- We have several processes access and manipulate the same data concurrently and
- **the outcome** of the execution **depends on the particular order** in which the access takes place,
- This is called **a race condition**.
- We need to ensure that **only one process at a time can manipulate** the variable counter.
- The processes **need to be synchronized** in some way.

Critical Section Problem

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

Critical Section

- General structure of process P_i

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

Algorithm for Process P_i

```
do {

    while (turn == j);

                critical section

    turn = j;

                remainder section

 } while (true);
```

Solution to Critical-Section Problem

A solution to the critical-section problem **must satisfy the following three requirements:**

1. **Mutual Exclusion** - If process ***P_i*** is executing in its critical section, then no other processes can be executing in their critical sections
2. **Progress** - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next **cannot be postponed indefinitely** **(i.e. the selection process must take a finite time)**
3. **Bounded Waiting** - A bound must exist **on the number of times that other processes are allowed to enter their critical sections** after a process has made a request to enter its critical section and before that request is granted
   - Assume that each process executes at a nonzero speed
   - No assumption concerning **relative speed** of the ***n*** processes

Critical-Section Handling in OS

- the code implementing an operating system (kernel code) is subject to several possible race conditions
- For example
  - a kernel data structure that maintains a list of all open files in the system (.If two processes were to open files simultaneously, the separate updates to this list could result in a race condition)
  - structures for maintaining memory allocation,
  - for maintaining process lists, and
  - for interrupt handling

Critical-Section Handling in OS

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

- **Preemptive** – allows preemption of process when running in kernel mode
- **Non-preemptive** – runs until exits kernel mode, blocks, or voluntarily yields CPU
  - Essentially free of race conditions in kernel mode (as only one process is active in the kernel at a time)

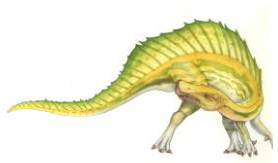
- Preemptive kernels, so they must be carefully designed to ensure that shared kernel data are free from race conditions.
- Which could be difficult
- A pre-emptive kernel **may be more responsive**, since there is less risk that a kernel-mode process will run for an arbitrarily long period before relinquishing the process or to waiting processes

Peterson's Solution

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

Algorithm for Process P_i

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



Peterson's Solution (Cont.)

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

     P_i enters CS only if:

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

- Because of the way **modern computer architectures** perform basic machine-language instructions, such as load and store, **there are no guarantees that Peterson's solution will work correctly on such architectures.**

- However, the solution **provides a good algorithmic description** of solving the critical-section problem

Synchronization Hardware

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

Solution to Critical-section Problem Using Locks

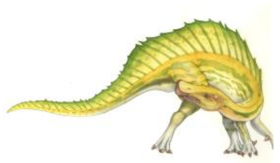
```
do {
      acquire lock
                  critical section
      release lock
                  remainder section
} while (TRUE);
```

test_and_set Instruction

Definition:

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

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



Solution using test_and_set()

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

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

compare_and_swap Instruction

Definition:

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

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

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

Solution using compare_and_swap

- Shared integer "lock" initialized to 0;
- Solution:

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

- Although these algorithms satisfy the mutual-exclusion requirement, they **do not satisfy the bounded-waiting requirement**.
- Next we present another algorithm using the test and set() instruction **that satisfies all the critical-section requirements**

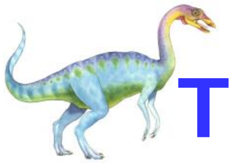
- The common data structures are

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

- These data structures are initialized to false

Bounded-waiting Mutual Exclusion with test_and_set

```
do {
   waiting[i] = true;
   key = true;
   while (waiting[i] && key)
      key = test_and_set(&lock);
   waiting[i] = false;
   /* critical section */
   j = (i + 1) % n;
   while ((j != i) && !waiting[j])
      j = (j + 1) % n;
   if (j == i)
      lock = false;
   else
      waiting[j] = false;
   /* remainder section */
} while (true);
```

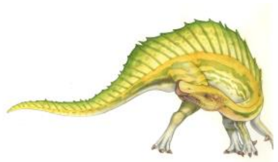


The Mutual Exclusion Requirement

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

The bounded-waiting Requirement

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



Mutex Locks

- Previous solutions are complicated and generally inaccessible to application programmers
- Instead, OS designers build software tools to solve critical section problem
- Simplest is mutex lock
- the term mutex is short for mutual exclusion
- Protect a critical section by first **`acquire()`** a lock then **`release()`** the lock
  - Boolean variable indicating if lock is available or not
- Calls to **`acquire()`** and **`release()`** must be atomic
  - Usually implemented via hardware atomic instructions
- If the lock is available, a call to acquire()succeeds, and the lock is then considered unavailable

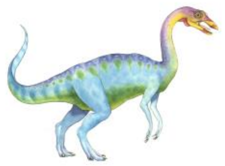
acquire() and release()

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

- But this solution requires **busy waiting**
  - **Because while a process is in its critical section, any other process that tries to enter its critical section must loop continuously in the call to acquire()**
  - This lock therefore called a **spinlock**

  - **Busy waiting wastes CPU cycles that some other process might be able to use productively.**

  - **Spinlocks do have an advantage, however, in that no context switch is required when a process must wait on a lock, and a context switch may take considerable time.**
  - **Thus, when locks are expected to be held for short times, spin locks are useful.**
  - **They are often employed on multiprocessor systems where one thread can “spin” on one processor while another thread performs its critical section on another processor**



Semaphore

- Synchronization tool that provides more sophisticated ways (than Mutex locks) for process to synchronize their activities.
- Semaphore ***S*** – integer variable
- Can only be accessed via two indivisible (atomic) operations
  - **`wait()`** and **`signal()`**
    - Originally called **`P()`** and **`V()`**
- Definition of the **`wait() operation`**

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

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

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

Semaphore Usage

- **Counting semaphore** – integer value can range over an unrestricted domain
- **Binary semaphore** – integer value can range only between 0 and 1
  - Same as a **mutex lock**
  - **In fact, on systems that do not provide mutex locks, binary semaphores can be used instead for providing mutual exclusion.**
- Counting semaphores can be used to control access to a given resource consisting of a finite number of instances (e.g. 3 printers).
  - The semaphore **is initialized to the number of resources available.**
  - Each process that wishes to use a resource performs a wait() operation on the semaphore **(thereby decrementing the count).**
  - When a process releases a resource, it performs a signal() operation (incrementing the count).
  - When the count for the semaphore goes to 0, all resources are being used.
  - After that, processes that wish to use a resource will block until the count becomes greater than 0.

- Can solve various synchronization problems
- Consider ***P*₁** and ***P*₂** that require ***S*₁** to happen before ***S*₂**

  Create a semaphore "**`synch`**" initialized to 0

```
P1:
    $S_1$;
    signal(synch);
P2:
    wait(synch);
    $S_2$;
```

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

Semaphore Implementation

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

Semaphore Implementation with no Busy waiting

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

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

Implementation with no Busy waiting (Cont.)

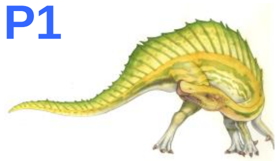
```
wait(semaphore *S) {
   S->value--;  // equivalent to(*S).value--
   if (S->value < 0) {
      add this process to S->list;
      block(); // suspends the process that invokes it
   }
}

signal(semaphore *S) {
   S->value++;
   if (S->value <= 0) {
      remove a process P from S->list;
      wakeup(P);
         // resumes the execution of a blocked process P.
   }
}
```

Deadlock and Starvation

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

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

- **Suppose that P0 executes wait(S) and then P1 executes wait(Q).**
- **WhenP0 executes wait(Q), it must wait until P1 executes signal(Q).**
- **Similarly, when P1 executes wait(S), it must wait untilP0 executes signal(S).**
- **Since these signal() operations cannot be executed, P0 and P1 are deadlocked.**



- **Deadlock**
  - **We say that a set of processes is in a deadlocked state when every process in the set is waiting for an event that can be caused only by another process in the set.**
- **Starvation** – **indefinite blocking**
  - A process may never be removed from the semaphore queue in which it is suspended
- **Priority Inversion** – Scheduling problem when lower-priority process holds a lock needed by higher-priority process
  - Solved via **priority-inheritance protocol**

Classical Problems of Synchronization

- Classical problems used to test newly-proposed synchronization schemes
  - Bounded-Buffer Problem
  - Readers and Writers Problem
  - Dining-Philosophers Problem
- In our solutions to the problems, **we use semaphores for synchronization,** since that is the traditional way to present such solutions.

Bounded-Buffer Problem

- ***In our problem, the producer and consumer processes share the following data structures***
- ***n*** buffers, each can hold one item
- Semaphore **`mutex`** initialized to the value 1
  - Provides mutual exclusion for accesses to the buffer pool
- Semaphore **`full`** initialized to the value 0
  - Counts the number of full buffers
- Semaphore **`empty`** initialized to the value n
  - Counts the number of empty buffers

Bounded Buffer Problem (Cont.)

- the producer process

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

- the consumer process

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

Readers-Writers Problem

- A data set is shared among a number of concurrent processes
  - Readers – only read the data set; they do ***not*** perform any updates
  - Writers – can both read and write
- Obviously, if two readers access the shared data simultaneously, no adverse effects will result.
- However, if a writer and some other process (either a reader or a writer) access the database simultaneously, chaos may ensue.
- Problem – allow multiple readers to read at the same time
  - Only one single writer can access the shared data at the same time
- **Several variations** of how readers and writers are considered – all involve some form of priorities

- The first readers–writers problem, requires that no reader be kept waiting unless **a writer has already obtained permission** to use the shared object.
- In other words, **no reader should wait** for other readers to finish simply **because a writer is waiting**
- **Note here writers may starve**
- **Solution to the first readers–writers problem**
- **The reader processes share** the following data structures:
- Shared Data
  - Data set
  - Semaphore **`rw_mutex`** initialized to 1
    - common to both reader and writer processes;
    - a mutual exclusion semaphore for the writers;
    - also used by the first or last reader that enters or exits the critical section

- Integer **`read_count`** initialized to 0
    - keeps track of how many processes are currently reading the object
- Semaphore **`mutex`** initialized to 1
    - used to ensure mutual exclusion when the variable read count is updated.

Readers-Writers Problem (Cont.)

- The structure of a writer process

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

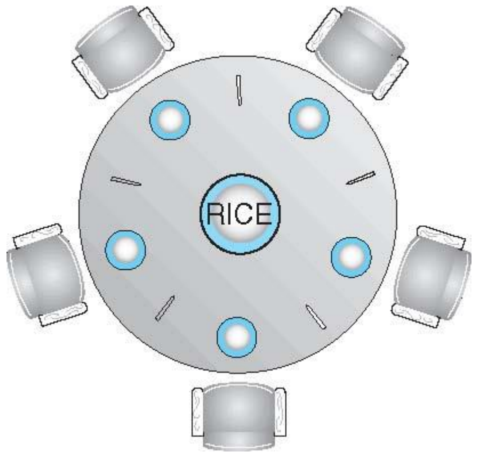
- The structure of a reader process

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

Readers-Writers Problem Variations

- ***First*** variation – no reader kept waiting unless writer has permission to use shared object
- ***Second*** variation – once writer is ready, it performs the write ASAP
- Both may have starvation leading to even more variations
- Problem is solved on some systems by kernel providing reader-writer locks

Dining-Philosophers Problem



- Consider five philosophers who spend their lives thinking and eating. The philosophers share a circular table surrounded by five chairs, each belonging to one philosopher. In the center of the table is a bowl of rice, and the table is laid with five single chopsticks (Figure 5.13). When a philosopher thinks, she does not interact with her colleagues. From time to time, a philosopher gets hungry and tries to pick up the two chopsticks that are closest to her (the chopsticks that are between her and her left and right neighbors). A philosopher may pick up only one chop stick at a time. Obviously, she cannot pick up a chopstick that is already in the hand of a neighbor. When a hungry philosopher has both her chopsticks at the same time, she eats without releasing the chopsticks. When she is finished eating, she puts down both chopsticks and starts thinking a gain.

- Philosophers spend their lives alternating thinking and eating
- Don’t interact with their neighbors, occasionally try to pick up 2 chopsticks (one at a time) to eat from bowl
  - Need both to eat, then release both when done
- It is a simple representation of the need to allocate several resources among several processes in a deadlock-free and starvation-free manner.
- A simple solution
  - represent each chopstick with a semaphore.
  - A philosopher tries to grab a chopstick by executing a wait() operation on that semaphore.
  - She releases her chopsticks by executing the signal() operation on the appropriate semaphores.
- In the case of 5 philosophers
  - Shared data
    - Bowl of rice (data set)
    - Semaphore chopstick [5] initialized to 1

Dining-Philosophers Problem Algorithm

- The structure of Philosopher *i*:

```
do {
     wait (chopstick[i] );
      wait (chopStick[ (i + 1) % 5] );

                  //  eat

      signal (chopstick[i] );
      signal (chopstick[ (i + 1) % 5] );

                   //  think

} while (TRUE);
```

- What is the problem with this algorithm?

Answer:

- It could create a deadlock.
  - Suppose that all five philosophers become hungry at the same time and each grabs her left chopstick.
  - All the elements of chopstick will now be equal to 0. When each philosopher tries to grab her right chopstick, she will be delayed forever.
- Several possible remedies to the deadlock problem (see next slide)

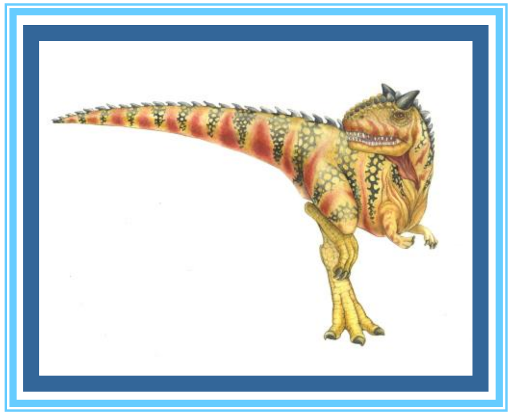
Dining-Philosophers Problem Algorithm (Cont.)

- Deadlock handling
  - Allow at most 4 philosophers to be sitting simultaneously at the table.
  - Allow a philosopher to pick up the forks only if both are available (picking must be done in a critical section.
  - Use an asymmetric solution -- an odd-numbered philosopher picks up first the left chopstick and then the right chopstick. Even-numbered philosopher picks up first the right chopstick and then the left chopstick.
- Monitors present a solution to the dining philosophers problem that ensures freedom of deadlocks

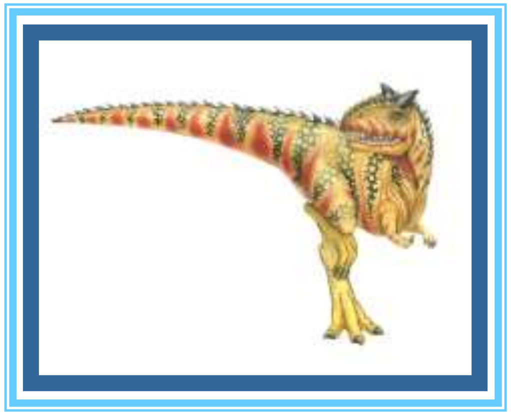
Problems with Semaphores

- Incorrect use of semaphore operations (caused by an honest programming error or an uncooperative programmer):
  - signal (mutex) …. wait (mutex)
    - Several processes may be executing in their critical sections simultaneously, violating the mutual-exclusion requirement
  - wait (mutex) … wait (mutex)
    - In this case, a deadlock will occur.
  - Omitting of wait (mutex) or signal (mutex) (or both)
    - In this case, either mutual exclusion is violated or a deadlock will occur.
- Deadlock and starvation are possible.

End of Chapter 5

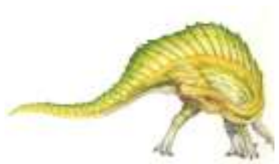


Chapter 6: CPU Scheduling



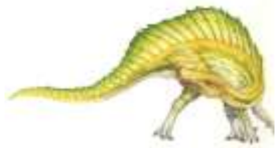
Chapter 6: CPU Scheduling

- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms
- Thread Scheduling



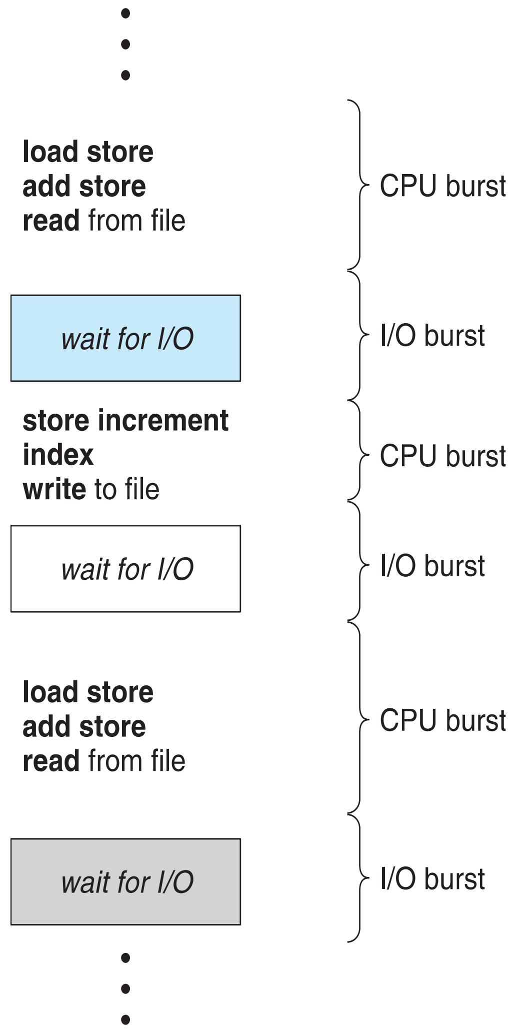
Objectives

- To introduce CPU scheduling, which is the basis for multiprogrammed operating systems
- To describe various CPU-scheduling algorithms
- To discuss evaluation criteria for selecting a CPU-scheduling algorithm for a particular system

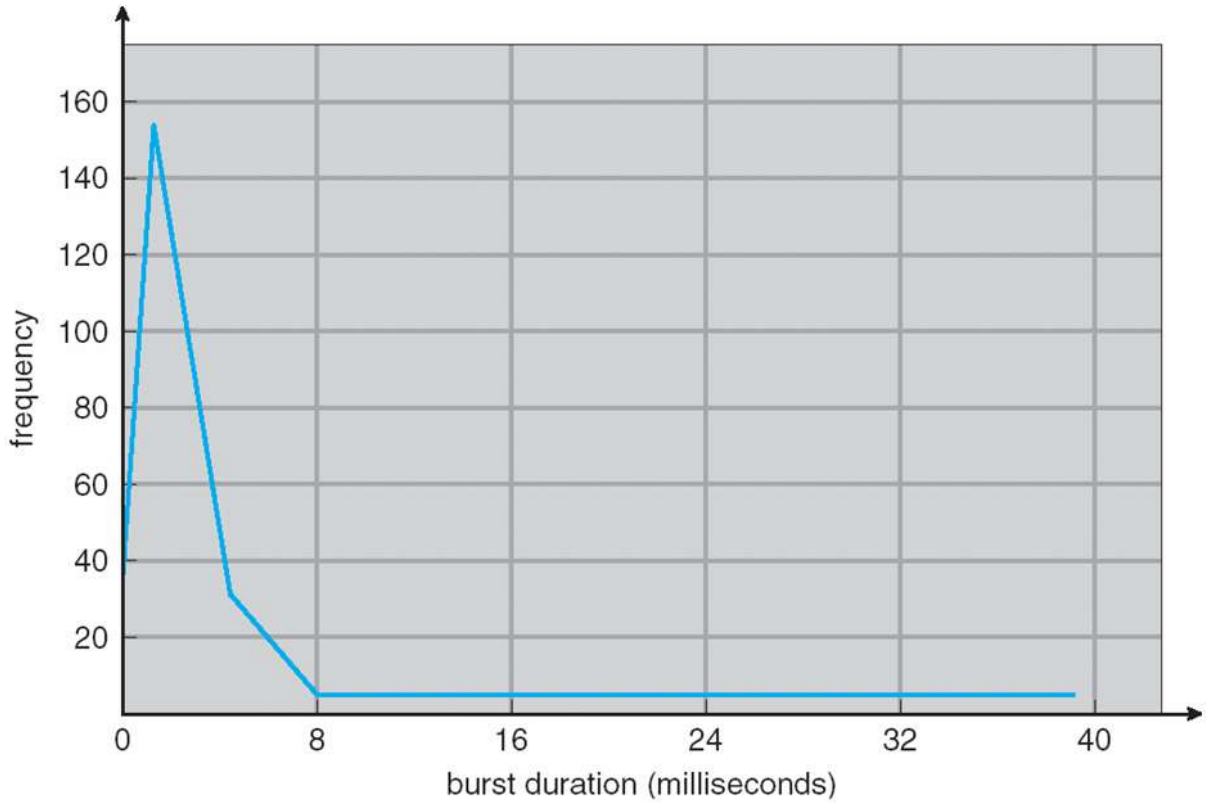


Basic Concepts

- Maximum CPU utilization obtained with multiprogramming
- CPU–I/O Burst Cycle – Process execution consists of a **cycle** of CPU execution and I/O wait
- **CPU burst** followed by **I/O burst**
- CPU burst distribution is of main concern
- The following figure shows that we usually have a large number of short CPU bursts and a small number of long CPU bursts.

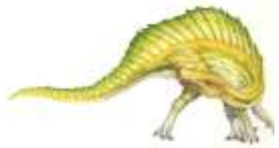


Histogram of CPU-burst Times



CPU Scheduler

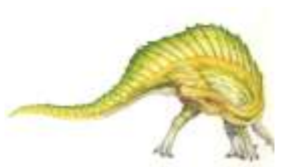
- **Short-term scheduler** selects from among the processes in ready queue, and allocates the CPU to one of them
  - Queue may be ordered in various ways
  - A ready queue can be implemented as a FIFO queue, a priority queue, a tree, or simply an unordered linked list
  - The records in the queues are generally process control blocks (PCBs) of the processes
- CPU scheduling decisions may take place when a process:
  1. Switches from running to waiting state (no choice in terms of scheduling)
  2. Switches from running to ready state (there is a choice)
  3. Switches from waiting to ready (there is a choice)
  4. Terminates (no choice in terms of scheduling)



- Scheduling under 1 and 4 is **nonpreemptive (or cooperative)**
- All other scheduling is **preemptive**
- **Nonpreemptive scheduling: Once the CPU has been allocated to a process, the process keeps the CPU until it releases the CPU either by terminating or by switching to the waiting state.**
- **Unfortunately, preemptive scheduling can result in race conditions when data are shared among several processes.**
- **Preemptive Scheduling needs to**
  - Consider access to shared data (the race condition)
  - Consider preemption while in kernel mode (might also occur while accessing shared kernel data)
  - Consider interrupts occurring during crucial OS activities (the sections of code affected by interrupts must be guarded from simultaneous use; otherwise input might be lost or output overwritten; these sections of code are not accessed concurrently by several processes, they disable interrupts at entry and re-enable interrupts at exit.)

Dispatcher

- Dispatcher module gives control of the CPU to the process selected by the short-term scheduler; this involves:
  - switching context
  - switching to user mode
  - jumping to the proper location in the user program to restart that program
- **Dispatch latency** – time it takes for the dispatcher to stop one process and start another running
- The dispatcher should be as fast as possible, since it is invoked during every process switch



Scheduling Criteria

- **CPU utilization** – keep the CPU as busy as possible
- **Throughput** – # of processes that complete their execution per time unit
- **Turnaround time** – amount of time to execute a particular process (The interval from the time of submission of a process to the time of completion is the turnaround time.)
- **Waiting time** – amount of time a process has been waiting in the ready queue
- **Response time** – amount of time it takes from when a request was submitted until the first response is produced, This is the time it takes to start responding, not the time it takes to output the response (for time-sharing environment). In an interactive system this is more important than the turn around time which is limited by the speed of the output device.

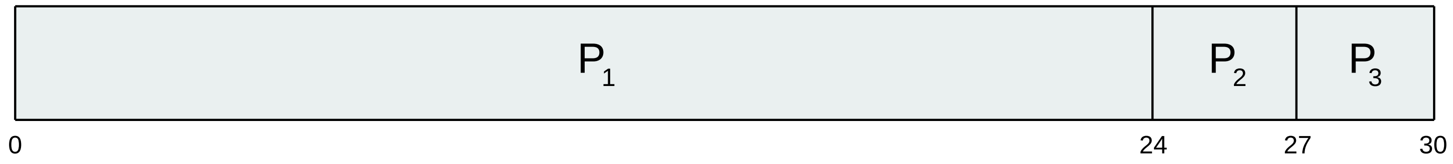
Scheduling Algorithm Optimization Criteria

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

First- Come, First-Served (FCFS) Scheduling

Process	Burst Time
P_1	24
P_2	3
P_3	3

- Suppose that the processes arrive in the order: P_1 , P_2 , P_3
  The Gantt Chart for the schedule is:



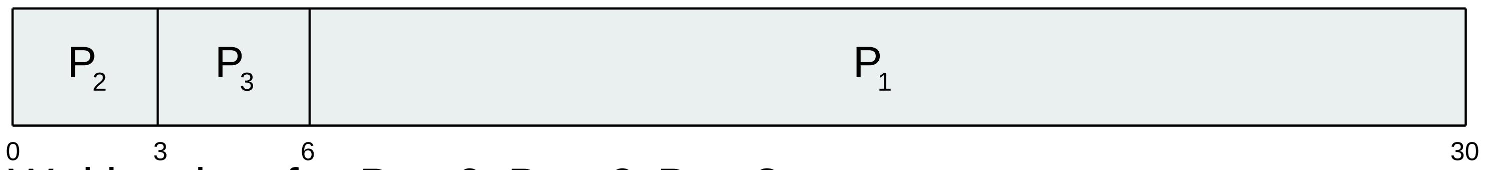
- Waiting time for $P_1 = 0$; $P_2 = 24$; $P_3 = 27$
- Average waiting time: $(0 + 24 + 27)/3 = 17$

FCFS Scheduling (Cont.)

Suppose that the processes arrive in the order:

P_2 , P_3 , P_1

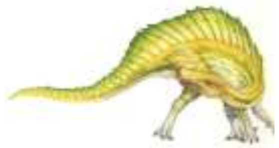
- The Gantt chart for the schedule is:



- Waiting time for $P_1 = 6$; $P_2 = 0$; $P_3 = 3$
- Average waiting time: (6 + 0 + 3)/3 = 3 (a substantial reduction)
- Much better than previous case
- Thus, the average waiting time under an FCFS policy is generally not minimal and may vary substantially if the processes' CPU burst times vary greatly
- **Convoy effect** - short process behind long process
  - Consider one CPU-bound and many I/O-bound processes
  - This effect results in lower CPU and device utilization than might be possible if the shorter processes were allowed to go first. (this is bad of course)

Shortest-Job-First (SJF) Scheduling

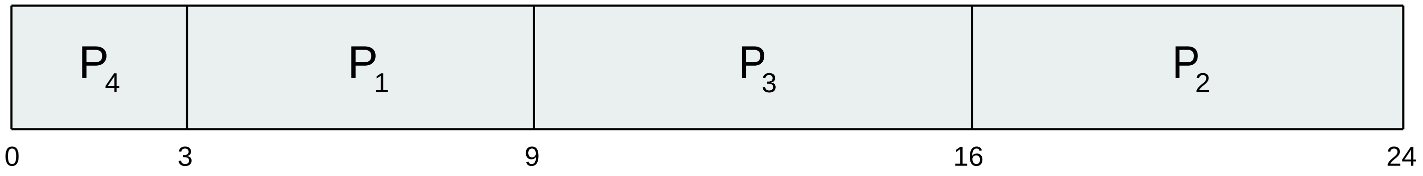
- Associate with each process the length of its next CPU burst
  - Use these lengths to schedule the process with the shortest time
- SJF is optimal – gives minimum average waiting time for a given set of processes
  - The difficulty is knowing the length of the next CPU request
  - Could ask the user



Example of SJF

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

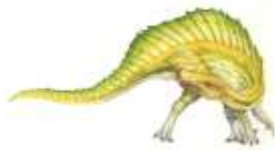
- SJF scheduling chart



- Average waiting time = (3 + 16 + 9 + 0) / 4 = 7
- By comparison, if we were using the FCFS scheduling scheme, the average waiting time would be 10.25 milliseconds.
- Notice that the FCFS scheduling algorithm is non-pre-emptive

- The SJF scheduling algorithm **is provably optimal**, in that it gives the minimum average waiting time for a given set of processes.
- Moving a short process before along one decreases the waiting time of the short process more than it increases the waiting time of the long process.
- Consequently, the average waiting time decreases.

- Although the SJF algorithm is optimal, it cannot be implemented at the level of short-term CPU scheduling. With short-term scheduling, **there is no way to know the length of the next CPU burst.**



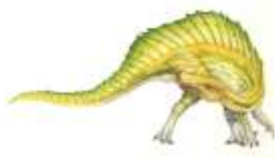
Determining Length of Next CPU Burst

- With short-term scheduling, there is no way to know the length of the next CPU burst
- Can only estimate the length – should be similar to the previous ones
  - Then pick process with shortest predicted next CPU burst
- Can be done by using the length of previous CPU bursts, using exponential averaging

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

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

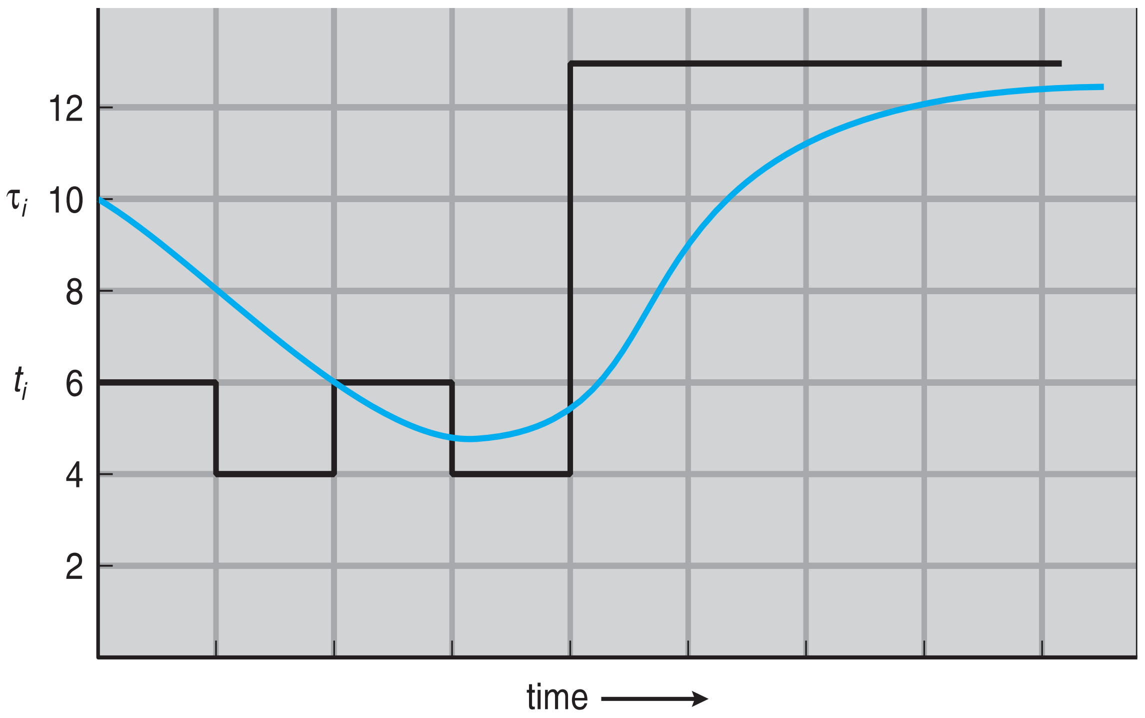
- Commonly, α set to ½ (to give recent and past history equal weights)
- Preemptive version called **shortest-remaining-time-first**



- Assuming alpha is ½ and T_0=10

Prediction of the Length of the Next CPU Burst

- Assuming alpha is ½ and T_0=10



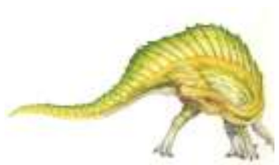
CPU burst (t_i)		6	4	6	4	13	13	13	…
"guess" (τ_i)	10	8	6	6	5	9	11	12	…

Examples of Exponential Averaging

- α =0
  - $\tau_{n+1} = \tau_n$
  - Recent history does not count
- α =1
  - $\tau_{n+1} = \alpha\ t_n$
  - Only the actual last CPU burst counts
- If we expand the formula by substituting for T_n, 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 - α) are less than or equal to 1, each successive term has less weight than its predecessor

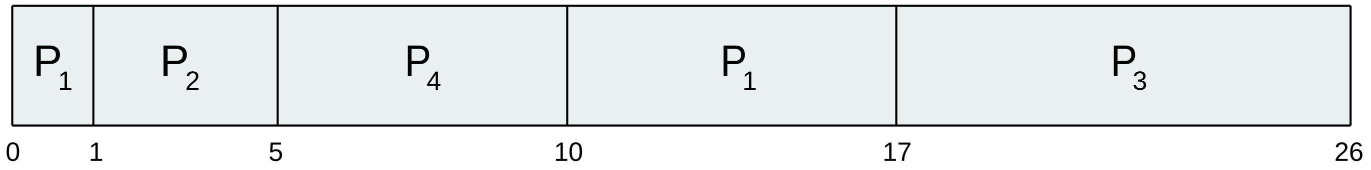


Example of Shortest-remaining-time-first

- Now we add the concepts of varying arrival times and preemption to the analysis

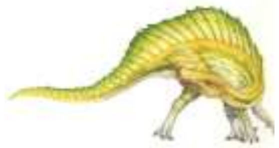
Process	*Arrival* Time	Burst Time
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

- *Preemptive* SJF Gantt Chart



- Average waiting time = [(10-1)+(1-1)+(17-2)+5-3)]/4 = 26/4 = 6.5 msec

- The next CPU burst of a newly arrived process may be shorter than what is left of the currently executing process.
- A preemptive SJF algorithm will preempt the currently executing process,
- whereas a nonpreemptive SJF algorithm will allow the currently running process to finish its CPU burst.
- Preemptive SJF scheduling is sometimes called shortest-remaining-time-first scheduling.



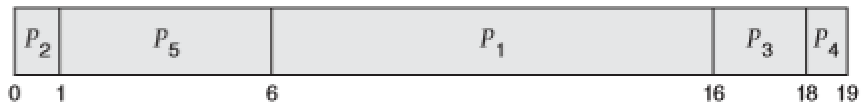
Priority Scheduling

- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority (smallest integer ≡ highest priority)
  - Preemptive
  - Nonpreemptive
- SJF is priority scheduling where priority is the inverse of predicted next CPU burst time
- Problem ≡ **Starvation** – low priority processes may never execute
- Solution ≡ **Aging** – as time progresses increase the priority of the process

Example of Priority Scheduling

Process	Burst Time	Priority
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

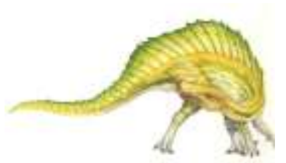
- Priority scheduling Gantt Chart



- Average waiting time = 8.2 msec

Round Robin (RR)

- Each process gets a small unit of CPU time (**time quantum** *q*), usually 10-100 milliseconds.
- After this time has elapsed, **the process is preempted** and **added to the end of the ready queue**.
- If there are *n* processes in the ready queue and the time quantum is *q*, then each process gets 1/*n* of the CPU time in chunks of at most *q* time units at once (because the process may release the CPU before its time slice is over).
- No process waits more than (*n*-1)*q* time units until its next time quantum
- For example, with five processes and a time quantum of 20 milliseconds, each process will get up to 20 milliseconds every 100 milliseconds

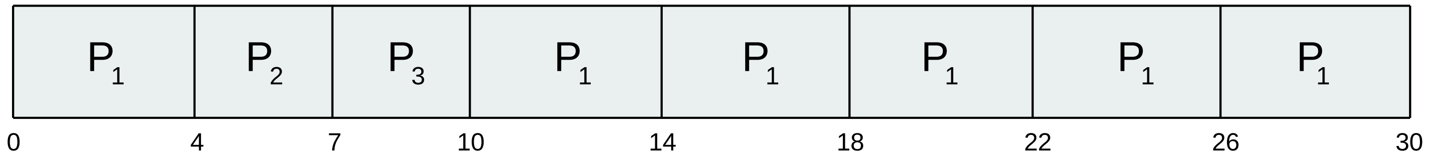


- The performance of the RR algorithm depends heavily on the size of the time quantum.
  - At one extreme, if the time quantum is **extremely large**, **the RR policy s the same as the FCFS policy.**
  - In contrast, if the time quantum is **extremely small** (say, 1 millisecond), the RR approach can result in **a large number of context switches**.
- Timer interrupts every quantum to schedule next process
- Performance
  - q large $\Rightarrow$ FIFO
  - q small $\Rightarrow$ ***q*** **must be large with respect to context switch, otherwise overhead is too high**
  - If the context-switch time is approximately 10 percent of the time quantum, then about 10 percent of the CPU time will be spent in context switching.

Example of RR with Time Quantum = 4

Process	Burst Time
P_1	24
P_2	3
P_3	3

- The Gantt chart is:

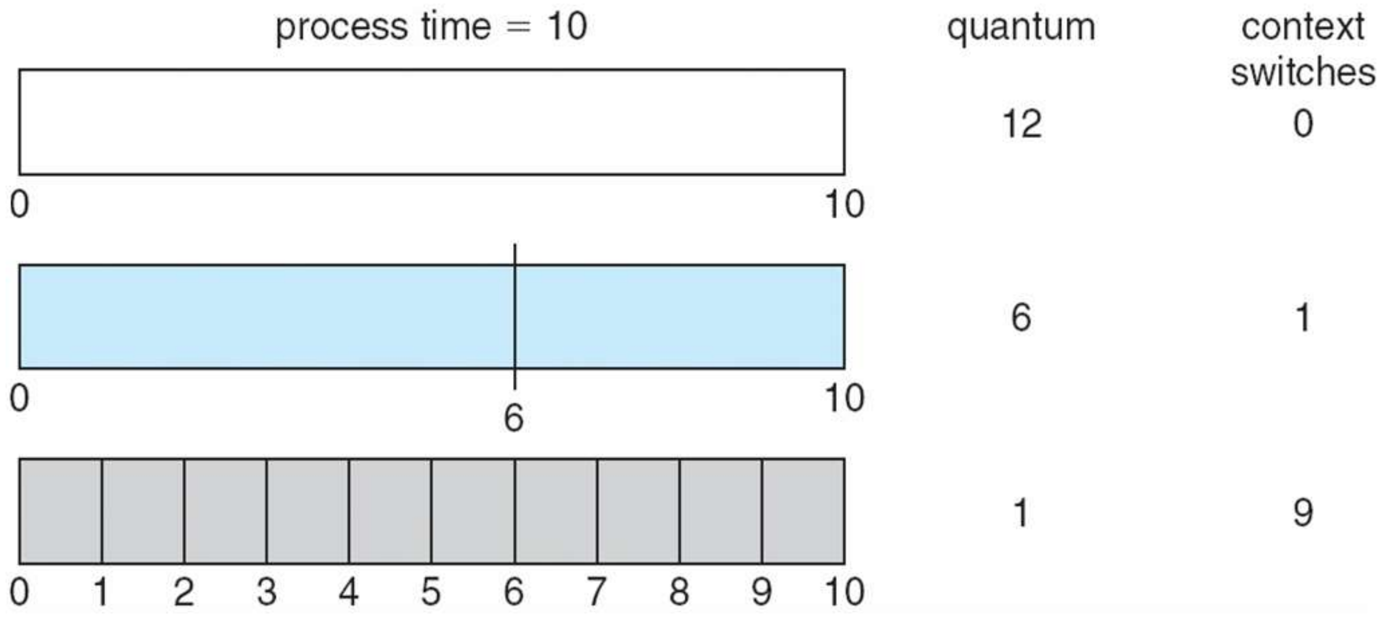


- **the average waiting time**
  - P1 waits for 6 milliseconds(10-4),
  - P2 waits for 4 milliseconds,
  - and P3 waits for 7 milliseconds.
- Thus, the average waiting time is 17/3 = 5.66 milliseconds.

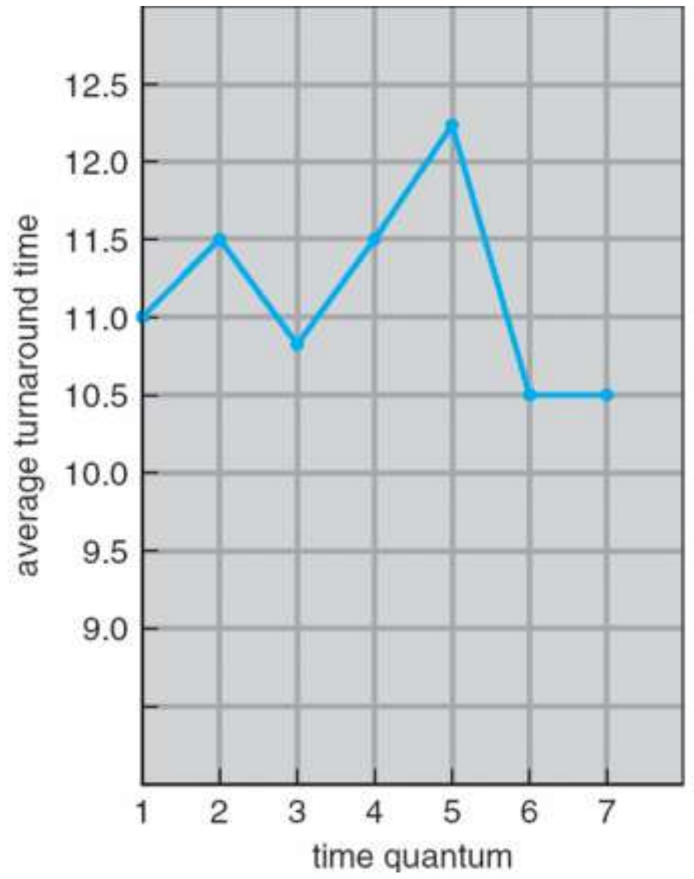
- Typically, **higher average turnaround** than SJF, but better ***response***

- q should be large compared to context switch time
- In modern machines, q is usually 10ms to 100ms, context switch < 10 ms

Time Quantum and Context Switch Time



Turnaround Time Varies With The Time Quantum

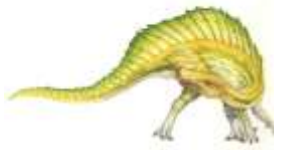


process	time
P_1	6
P_2	3
P_3	1
P_4	7

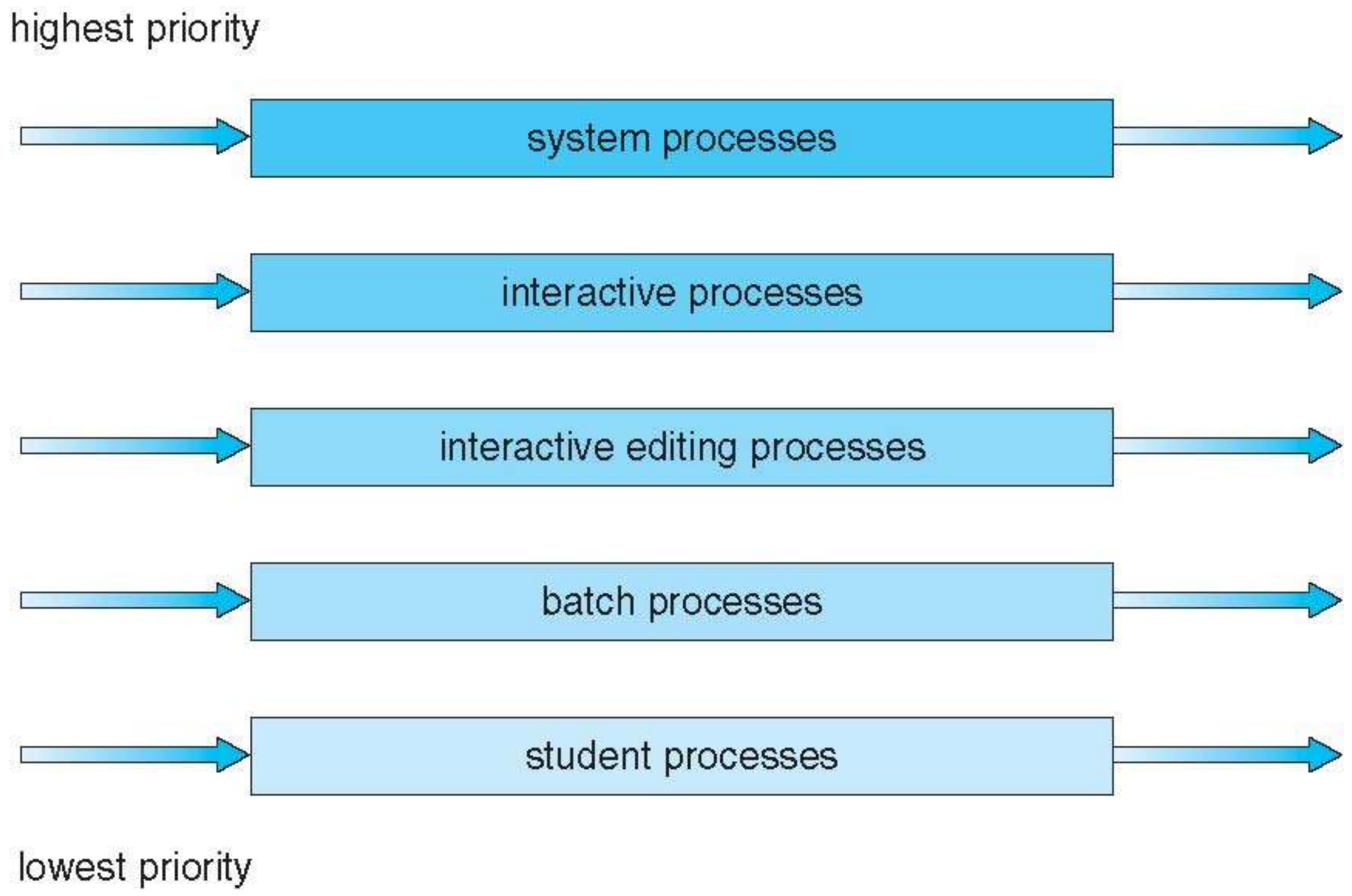
80% of CPU bursts should be shorter than q

Multilevel Queue

- Ready queue is partitioned into separate queues, eg:
  - **foreground** (interactive)
  - **background** (batch)
- Process permanently in a given queue
- Each queue has its own scheduling algorithm:
  - foreground – RR
  - background – FCFS
- Scheduling must be done between the queues:
  - Fixed priority scheduling; (i.e., serve all from foreground then from background). Possibility of starvation.
  - Time slice – each queue gets a certain amount of CPU time which it can schedule amongst its processes; i.e., 80% to foreground in RR
  - 20% to background in FCFS

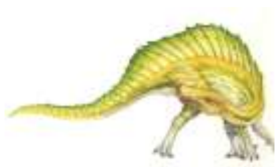


Multilevel Queue Scheduling



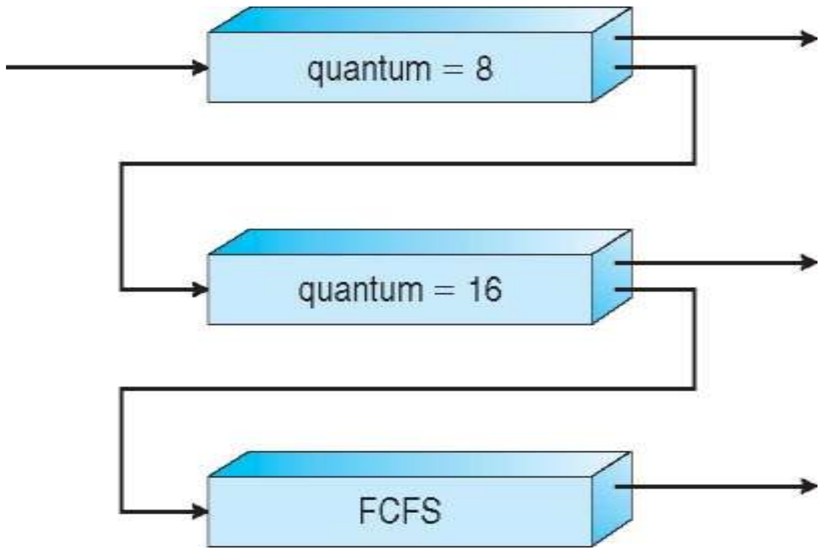
Multilevel Feedback Queue

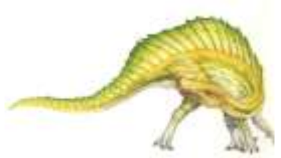
- A process can move between the various queues; aging can be implemented this way
- Multilevel-feedback-queue scheduler defined by the following parameters:
  - number of queues
  - scheduling algorithms for each queue
  - method used to determine when to upgrade a process
  - method used to determine when to demote a process
  - method used to determine which queue a process will enter when that process needs service



Example of Multilevel Feedback Queue

- Three queues:
  - Q_0 – RR with time quantum 8 milliseconds
  - Q_1 – RR time quantum 16 milliseconds
  - Q_2 – FCFS
- Scheduling
  - A new job enters queue Q_0 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





Thread Scheduling

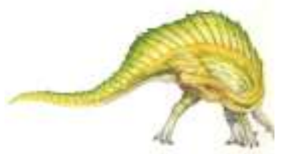
- Distinction between user-level and kernel-level threads
- When threads supported, threads scheduled, not processes
- Many-to-one and many-to-many models, thread library schedules user-level threads to run on LWP
  - Known as **process-contention scope** (**PCS**) since scheduling competition is within the process
  - Typically done via priority set by programmer
- Kernel thread scheduled onto available CPU is **system-contention scope** (**SCS**) – competition among all threads in system

Pthread Scheduling

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

Pthread Scheduling API

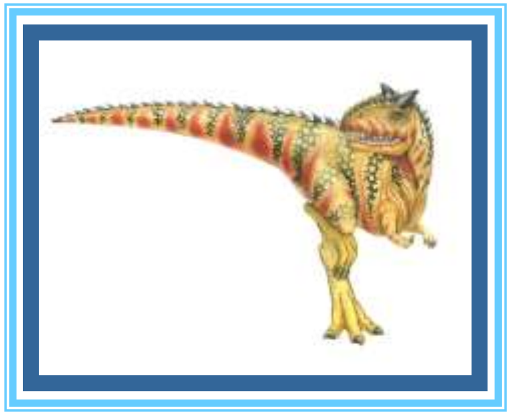
```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5
int main(int argc, char *argv[]) {
   int i, scope;
   pthread_t tid[NUM THREADS];
   pthread_attr_t attr;
   /* get the default attributes */
   pthread_attr_init(&attr);
   /* first inquire on the current scope */
   if (pthread_attr_getscope(&attr, &scope) != 0)
      fprintf(stderr, "Unable to get scheduling scope\n");
   else {
      if (scope == PTHREAD_SCOPE_PROCESS)
         printf("PTHREAD_SCOPE_PROCESS");
      else if (scope == PTHREAD_SCOPE_SYSTEM)
         printf("PTHREAD_SCOPE_SYSTEM");
      else
         fprintf(stderr, "Illegal scope value.\n");
   }
```



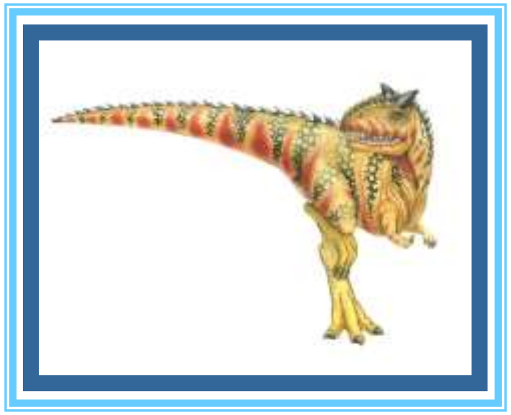
Pthread Scheduling API

```
    /* 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);
}
```

End of Chapter 6

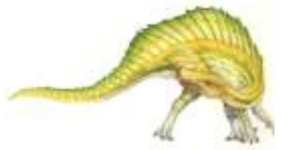


Chapter 8: Main Memory



Chapter 8: Memory Management

- Background
- Swapping
- Contiguous Memory Allocation
- Segmentation
- Paging

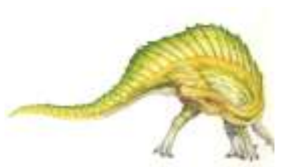


Objectives

- To provide a detailed description of various ways of **organizing memory hardware**
- To discuss various memory-management techniques, **including paging and segmentation**

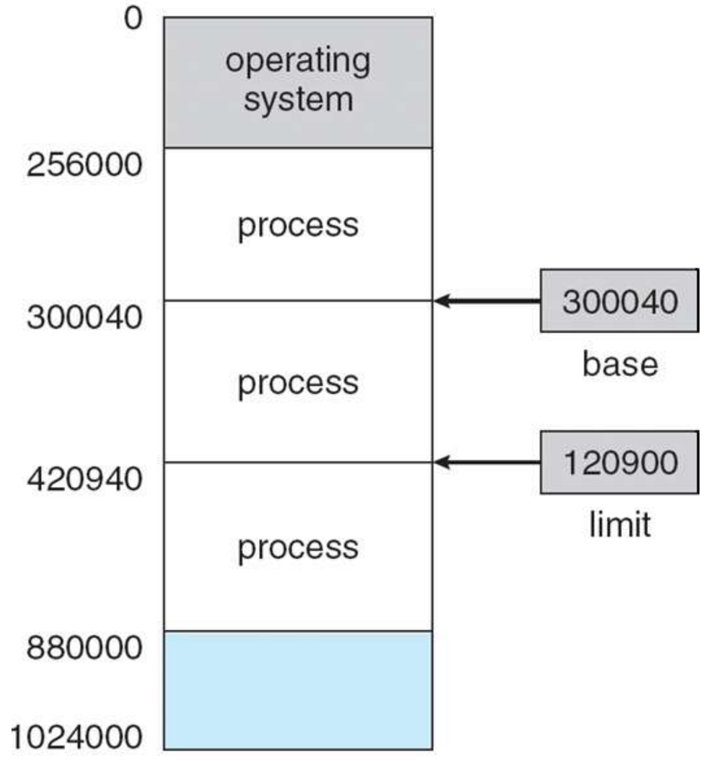
Background

- **Program must be brought (from disk) into memory** and placed within a process for it to be run
- **Main memory** and **registers** are only storage **CPU can access directly**
- Memory unit only sees a stream of addresses + read requests, or address + data and write requests
- i.e memory unit does not know how they are generated (by the instruction counter, indexing, indirection, literal addresses, and so on) or what they are for (instructions or data).
- Register access in one CPU clock (or less)
- Main memory can take many cycles, causing a **stall ➔a cache is needed**
- **Cache** sits between main memory and CPU registers
- Protection of memory required to ensure correct operation

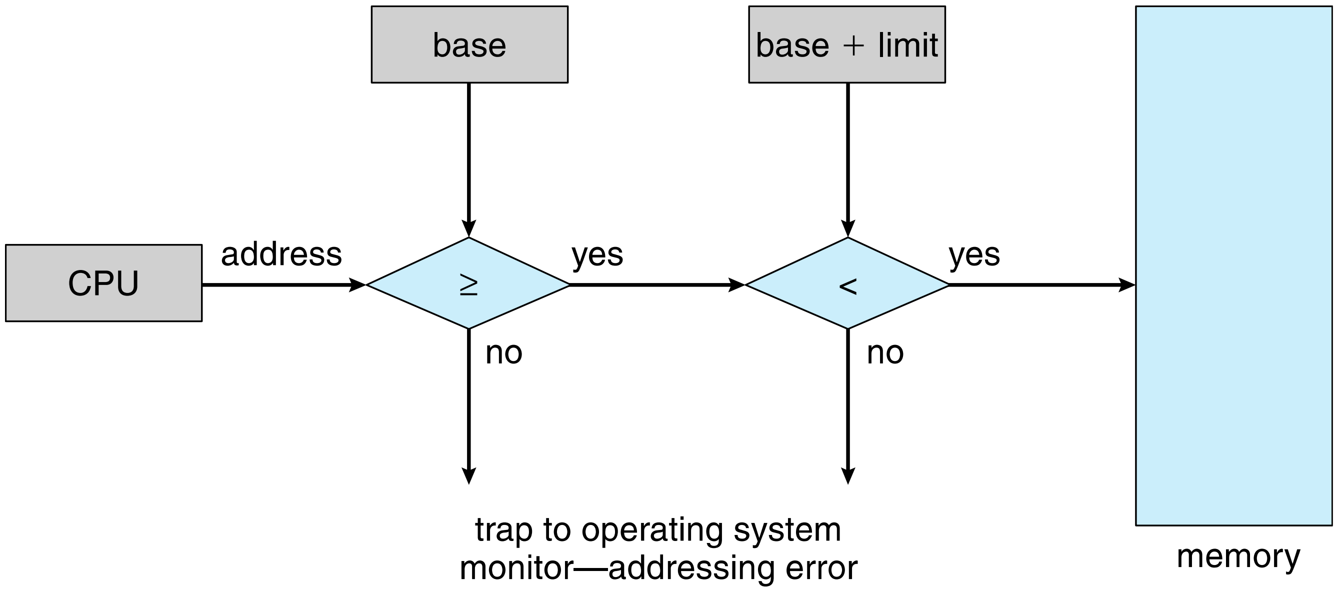


Base and Limit Registers

- A pair of **base** and **limit registers** define the logical address space
- CPU must check every memory access generated in user mode to be sure it is between base and limit for that user

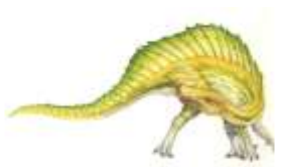


Hardware Address Protection



Address Binding

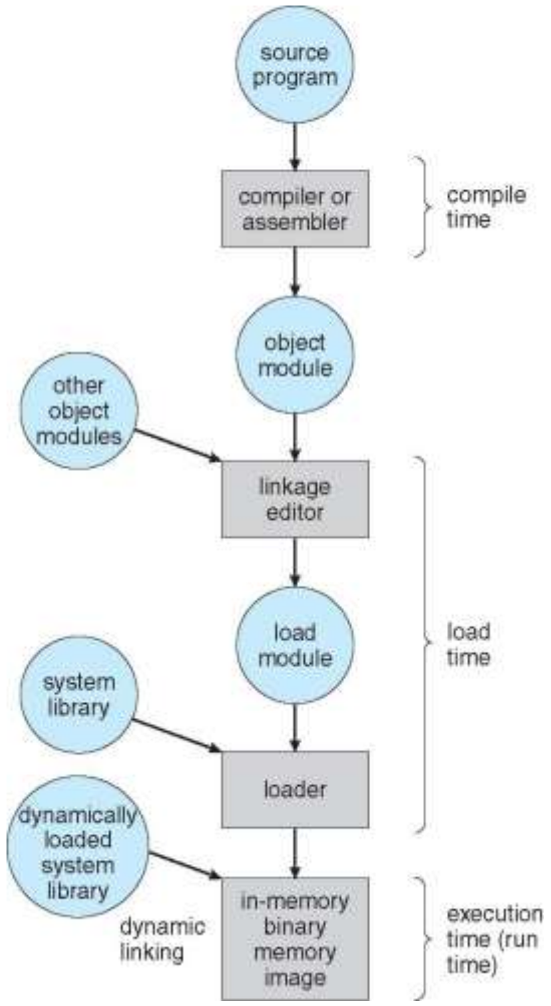
- Programs on disk, ready to be brought into memory to execute form an **input queue (contains the processes on the disk waiting to be executed)**
  - Without support, must be loaded into address 0000
- Inconvenient to have first user process physical address always at 0000
  - How can it not be?
- Further, addresses represented in different ways at different stages of a program's life
  - Source code addresses usually symbolic
  - Compiled code addresses **bind** to relocatable addresses
    - i.e. "14 bytes from beginning of this module"
  - Linker or loader will bind relocatable addresses to absolute addresses
    - i.e. 74014
  - Each binding maps one address space to another



Binding of Instructions and Data to Memory

- Address binding of instructions and data to memory addresses can happen at three different stages
  - **Compile time**: If memory location known a priori, **absolute code** can be generated; must recompile code if starting location changes
  - **Load time**: Must generate **relocatable code** if memory location is not known at compile time
  - **Execution time**: Binding delayed until run time **if the process can be moved during its execution** from one memory segment to another
    - Need hardware support for address maps (e.g., base and limit registers)

Multistep Processing of a User Program

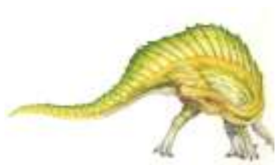


Logical vs. Physical Address Space

- The concept of a logical address space that is bound to a separate **physical address space** is central to proper memory management
  - **Logical address** – generated by the CPU; also referred to as **virtual address**
  - **Physical address** – address seen by the memory unit
- Logical and physical addresses are **the same in compile-time and load-time address-binding schemes;**
- logical (virtual) and physical addresses differ in execution-time address-binding scheme
- **Logical address space** is the set of all logical addresses generated by a program
- **Physical address space** is the set of all physical addresses generated by a program

Memory-Management Unit (MMU)

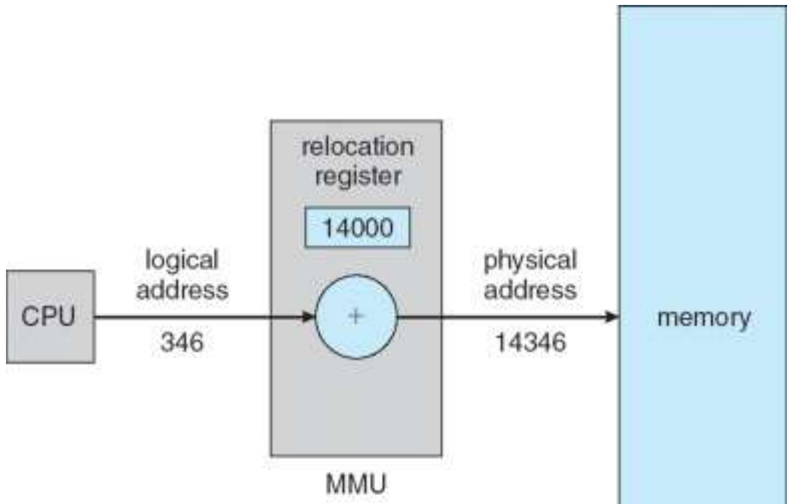
- Hardware device that at run time **maps virtual to physical address**
- Many methods possible, covered in the rest of this chapter
- To start, consider simple scheme where the value in the relocation register is added to every address generated by a user process at the time it is sent to memory
  - Base register now called **relocation register**
  - MS-DOS on Intel 80x86 used 4 relocation registers
- For example, if the base is at14000, then an attempt by the user to address location 0 is dynamically relocated to location14000; an access to location 346 is mapped to location 14346.



- The user program deals with *logical* addresses; it never sees the *real* physical addresses
  - Execution-time binding occurs when reference is made to location in memory
  - Logical address bound to physical addresses

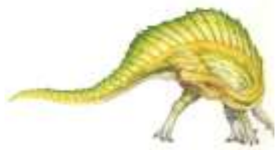
Dynamic relocation using a relocation register

- Routine is not loaded until it is called (dynamic loading)
- Better memory-space utilization; **unused routine is never loaded**
- All routines **kept on disk** in relocatable load format
- Useful when large amounts of code are needed to handle **infrequently occurring cases**
- No special support from the operating system is required
  - Implemented through program design
  - OS can help by providing libraries to implement dynamic loading



Dynamic Linking

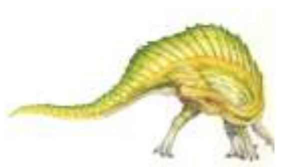
- Dynamically linked libraries are **system libraries (such as language subroutine libraries)** that are linked to user programs when the programs are run
- **Static linking** – system libraries and program code combined by the loader into the binary program image
- Dynamic linking –linking postponed until execution time
- Small piece of code, **stub**, used to locate the appropriate memory-resident library routine or to load the library if the routine is not present
- Stub replaces itself with the address of the routine, and executes the routine
- Thus, the next time that particular code segment is reached, the library routine is executed directly, incurring no cost for dynamic linking
- Under this scheme, all processes that use a language library execute only one copy of the library code **(shared libraries)**.



- Operating system checks if routine is in processes' memory address
  - If not in address space, add to address space
- Dynamic linking is particularly useful for libraries
- System also known as **shared libraries**
- A library may be **replaced by a new version**, and all programs that reference the library will automatically use the new version.
- Consider applicability to patching system libraries
  - Versioning may be needed

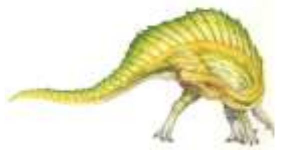
Swapping

- A process can be **swapped** temporarily out of memory to a backing store, and then brought back into memory for continued execution
  - **Total physical memory space of processes can exceed physical memory**
- **Backing store** – fast disk large enough to accommodate copies of all memory images for all users; must provide direct access to these memory images
- **Roll out, roll in** – swapping variant used for priority-based scheduling algorithms; lower-priority process is swapped out so higher-priority process can be loaded and executed
- Major part of swap time is transfer time; total transfer time is directly proportional to the amount of memory swapped
- System maintains a **ready queue** of ready-to-run processes which have memory images on disk

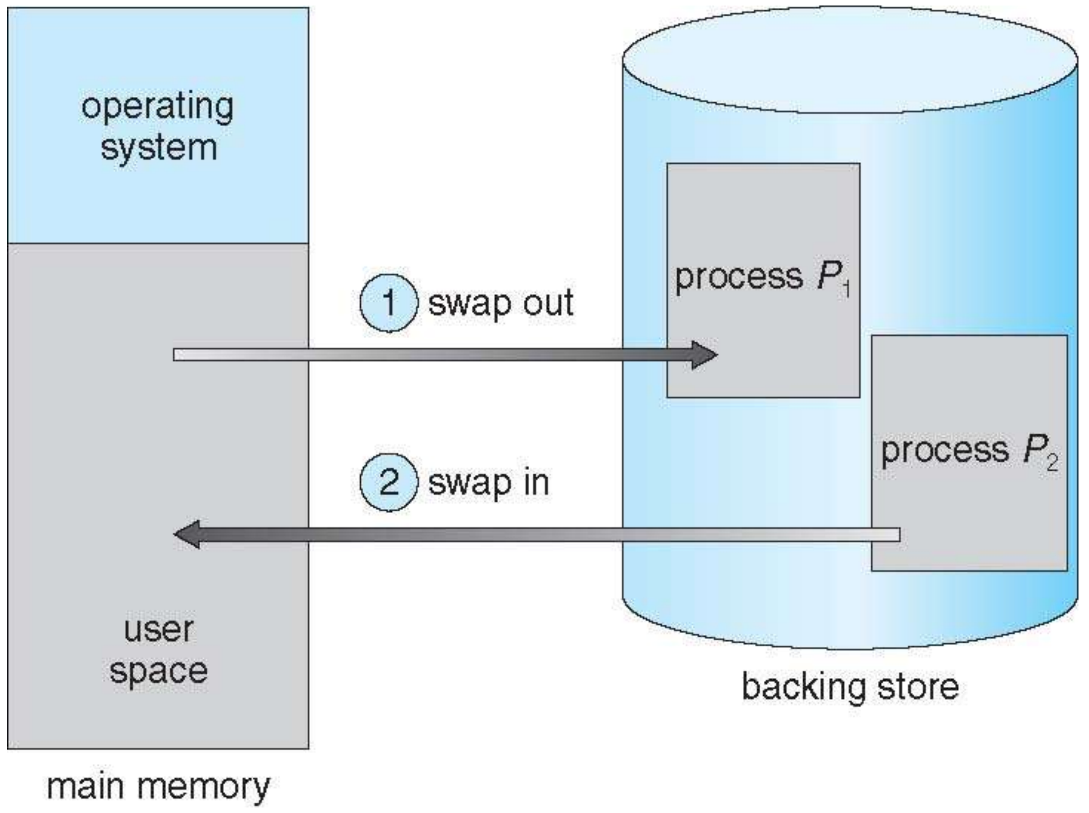


Swapping (Cont.)

- Does the swapped out process need to swap back in to same physical addresses?
- Depends on address binding method
  - Plus consider pending I/O to / from process memory space
- Modified versions of swapping are found on many systems (i.e., UNIX, Linux, and Windows)
  - Swapping normally disabled
  - Started if more than threshold amount of memory allocated
  - Disabled again once memory demand reduced below threshold



Schematic View of Swapping



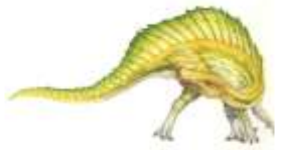
Context Switch Time including Swapping

- If next processes to be put on CPU is not in memory, need to swap out a process and swap in target process
- Context switch time can then be very high
- 100MB process swapping to hard disk with transfer rate of 50MB/sec
  - Swap out time of 2000 ms
  - Plus swap in of same sized process
  - Total context switch swapping component time of 4000ms (4 seconds)
- If we have a computer system with 4 GB of main memory and a resident operating system taking 1 GB, the maximum size of the user process is 3 GB.
- However, many user processes may be much smaller than this—say, 100 MB.
- A 100-MB process could be swapped out in 2 seconds, compared with the 60 seconds required for swapping 3 GB.

- Clearly, it would be useful to know exactly how much memory a user process is using, not simply how much it might be using.
- Then we would need to swap only what is actually used, reducing swap time. For this method to be effective, the user must keep the system informed of any changes in memory requirements.
- Thus, a process with dynamic memory requirements will need to issue system calls (request memory() and release memory()) to inform the operating system of its changing memory needs.

Context Switch Time and Swapping (Cont.)

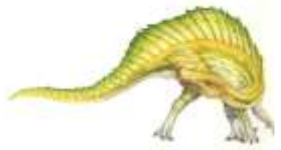
- Other constraints as well on swapping
- If we want to swap a process, we must be sure that it is completely idle
- A process may be waiting for an I/O operation when we want to swap that process to free up memory. However, if the I/O is asynchronously accessing the user memory for I/O buffers, then the process cannot be swapped. Assume that the I/O operation is queued because the device is busy. If we were to swap out process P1 and swap in process P2, the I/O operation might then attempt to use memory that now belongs to process P2.
- Two solutions
  - Pending I/O – can't swap out as I/O would occur to wrong process
  - Or always transfer I/O to kernel space, then to I/O device
    - Known as **double buffering**, adds overhead



- Standard swapping not used in modern operating systems
    - But there is a modified version that is common
        - **Swap only when free memory extremely low**

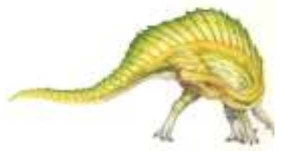
Swapping on Mobile Systems

- Not typically supported (mobile systems typically do not support swapping in any form)
  - Flash memory based
    - Small amount of space (not large enough for swaping)
    - Limited number of write cycles
    - Poor throughput between flash memory and CPU on mobile platform
- Instead use other methods to free memory if low
  - iOS ***asks*** apps to voluntarily relinquish allocated memory
    - Read-only data thrown out and reloaded from flash if needed
    - Failure to free can result in termination
  - Android terminates apps if low free memory, but first writes **application state** to flash for fast restart
  - Both OSes support paging as discussed below



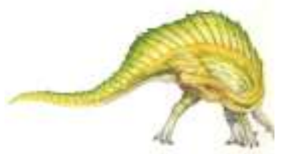
Contiguous Allocation

- Main memory must support both OS and user processes
- Limited resource, must allocate efficiently
- Contiguous allocation is one early method
- Main memory usually into two **partitions**:
  - Resident operating system, usually held in low memory with interrupt vector
  - User processes then held in high memory
  - Each process contained in single contiguous section of memory

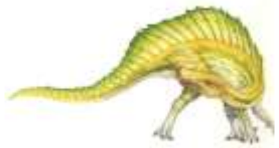


Contiguous Allocation (Cont.)

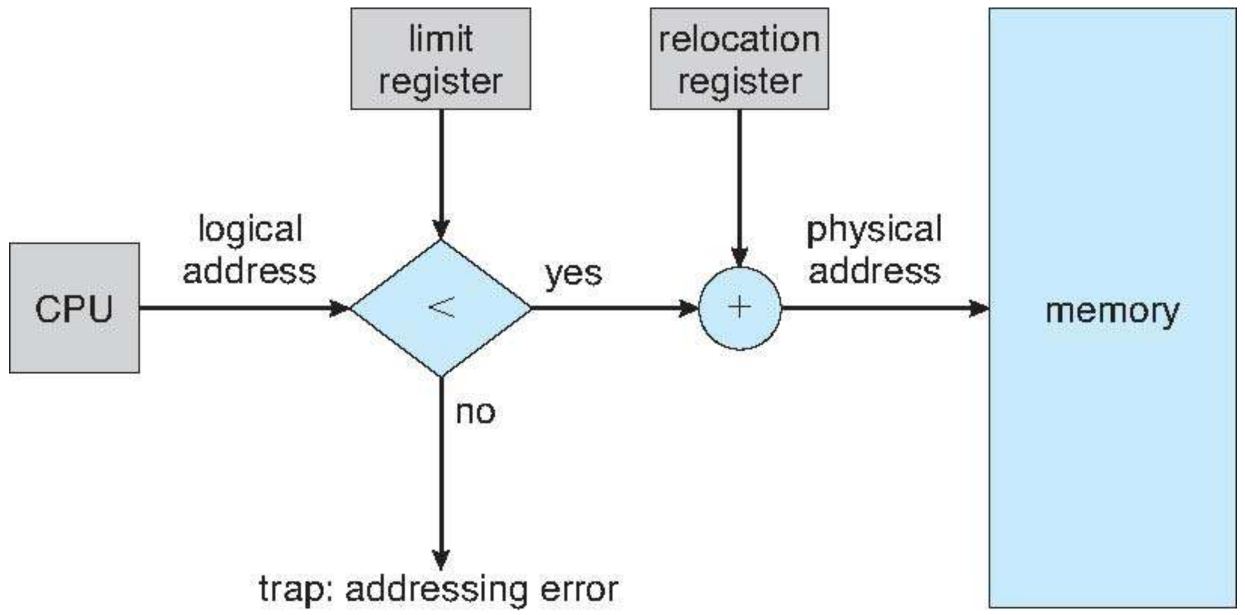
- Relocation registers **used to protect user processes from each other**, and **from changing operating-system code and data**
  - Base register contains value of smallest physical address
  - Limit register contains range of logical addresses – each logical address must be less than the limit register
  - MMU maps logical address *dynamically*
  - Can then allow actions such as kernel code being **transient** and kernel changing size



- This allows the operating system’s size to change dynamically.
- For example, the operating system contains code and buffer space for device drivers. If a device driver (or other operating-system service) is not commonly used, we donot want to keep the code and data in memory,as we might be able to use that space for other purposes.
- Such code is sometimes called **transient operating-system code**;
- it comes and goes as needed

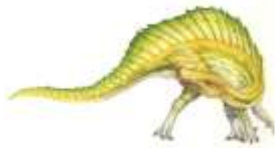


Hardware Support for Relocation and Limit Registers



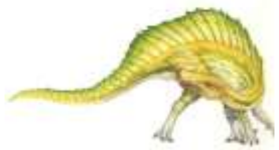
Memory Allocation

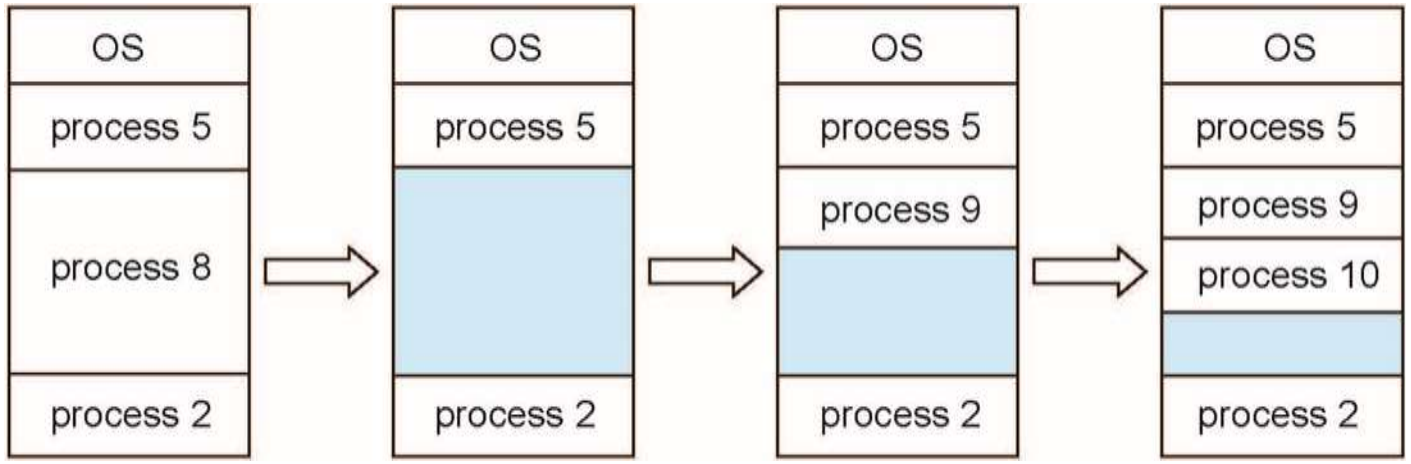
- One of the simplest methods for allocating memory is to divide memory into several **fixed-sized partitions**.
- Each partition may contain exactly one process.
- Thus, the **degree of multiprogramming is bound by** the number of partitions.
- In this multiple partition method, when a partition is free, a process is selected from the input queue and is loaded into the free partition.
- When the process terminates, the partition becomes available for another process.
- no longer in use



Multiple-partition allocation

- **Variable-partition**
  - Initially, all memory is available for user processes and is considered **one large block of available memory**, a hole.
  - Eventually, as you will see, memory contains a set of holes of various sizes.
  - **Variable-partition** sizes for efficiency (sized to a given process' needs)
  - **Hole** – block of available memory; holes of various size are scattered throughout memory
  - When a process arrives, it is allocated memory from **a hole large enough to accommodate it**
  - Process exiting frees its partition, **adjacent free partitions combined**
  - Operating system maintains information about:
    a) allocated partitions b) free partitions (holes)



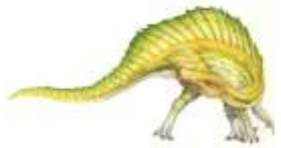

OS
process 5
process 8
process 2
OS
process 5
process 2
OS
process 5
process 9
process 2
OS
process 5
process 9
process 10
process 2

Dynamic Storage-Allocation Problem

How to satisfy a request of size ***n*** from a list of free holes?

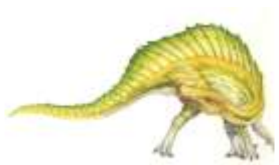
- **First-fit**: Allocate the ***first*** hole that is big enough
- **Best-fit**: Allocate the ***smallest*** hole that is big enough; must search entire list, unless ordered by size
  - Produces the **smallest leftover hole**
- **Worst-fit**: Allocate the ***largest*** hole; must also search entire list
  - Produces **the largest leftover hole**

First-fit and best-fit better than worst-fit in terms of speed and storage utilization



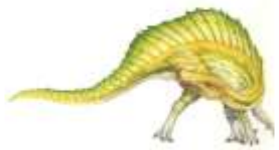
Fragmentation

- **External Fragmentation** – total memory space exists to satisfy a request, but it is not contiguous **(broken into little pieces)**
- **Internal Fragmentation** – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used
- First fit analysis reveals that given *N* blocks allocated, another 0.5 *N* blocks lost to fragmentation
  - i.e. 1/3 may be unusable -> this is known as **50-percent rule**



Fragmentation (Cont.)

- Reduce external fragmentation by **compaction**
  - **Shuffle memory contents** to place all free memory together in one large block
  - Compaction is possible ***only*** **if relocation is dynamic**, and is **done at execution time**
  - I/O problem
    - Latch job in memory while it is involved in I/O
    - Do I/O only into OS buffers
- Now consider that backing store has same fragmentation problems

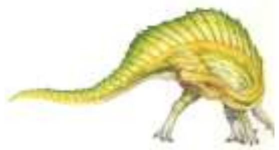


- Another possible solution to the external-fragmentation problem is to **permit the logical address space of the processes to be noncontiguous**,
- thus allowing a process to be **allocated physical memory wherever such memory is available**.
- Two complementary techniques achieve this solution: **segmentation and paging**.

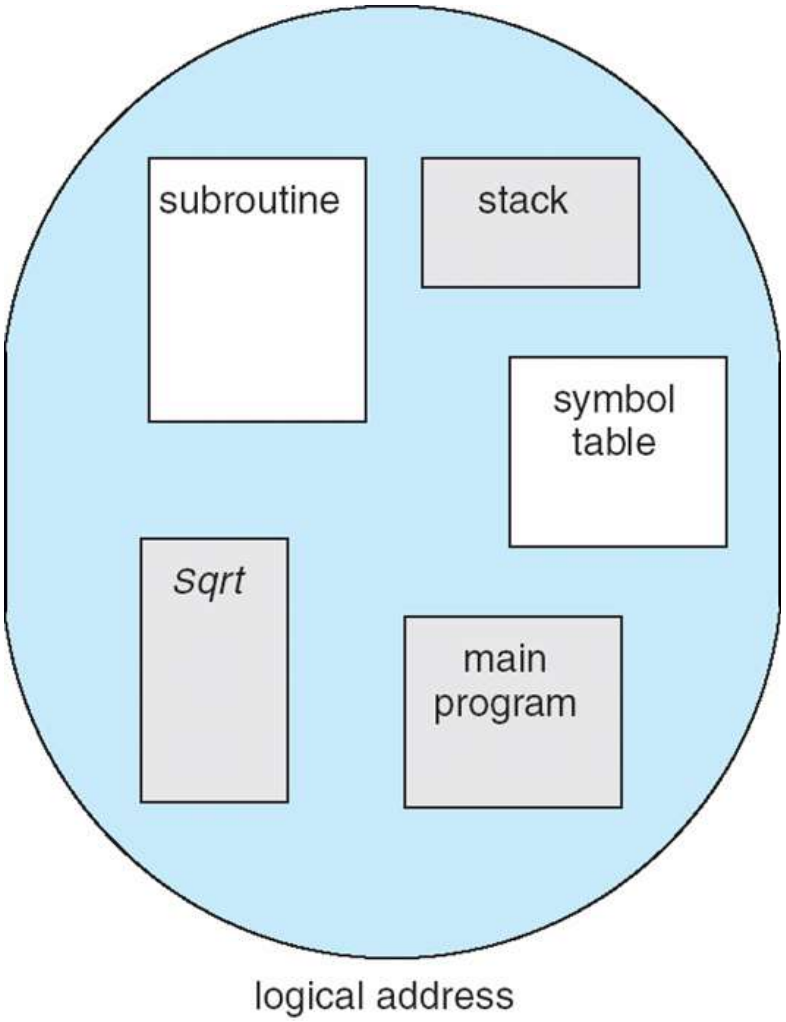
Segmentation

- Memory-management scheme that **supports user view of memory**
- **A program is a collection of segments**
  - **A segment is a logical unit** such as:

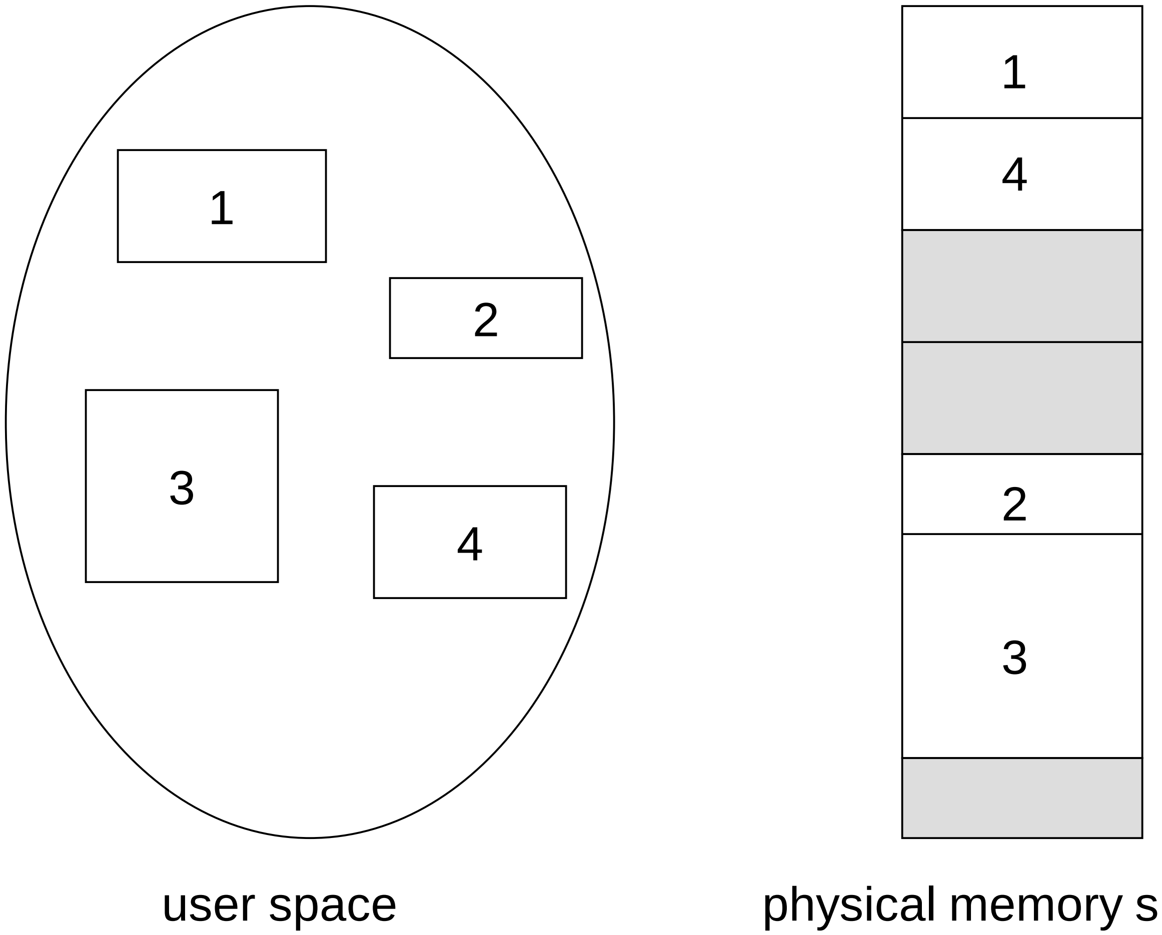
    main program
    procedure
    function
    method
    object
    local variables, global variables
    common block
    stack
    symbol table
    arrays



User's View of a Program



Logical View of Segmentation



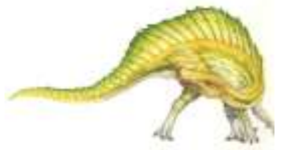
Segmentation Architecture

- Logical address consists of a two tuple:

  <segment-number, offset>,

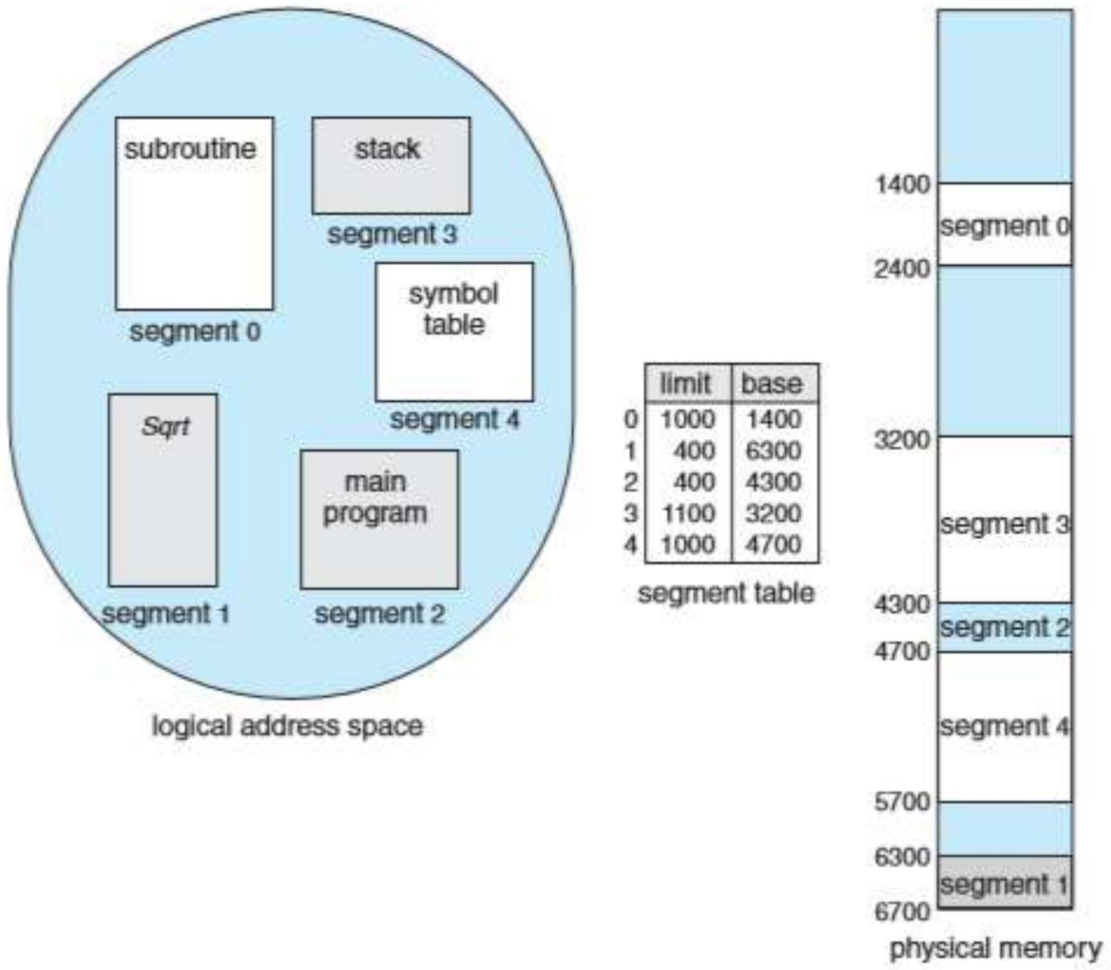
- **Segment table** – maps two-dimensional physical addresses; each table entry has:
  - **base** – contains the starting physical address where the segments reside in memory
  - **limit** – specifies the length of the segment
- **Segment-table base register (STBR)** points to the segment table's location in memory
- **Segment-table length register (STLR)** indicates number of segments used by a program;

  segment number ***s*** is legal if ***s*** < **STLR**



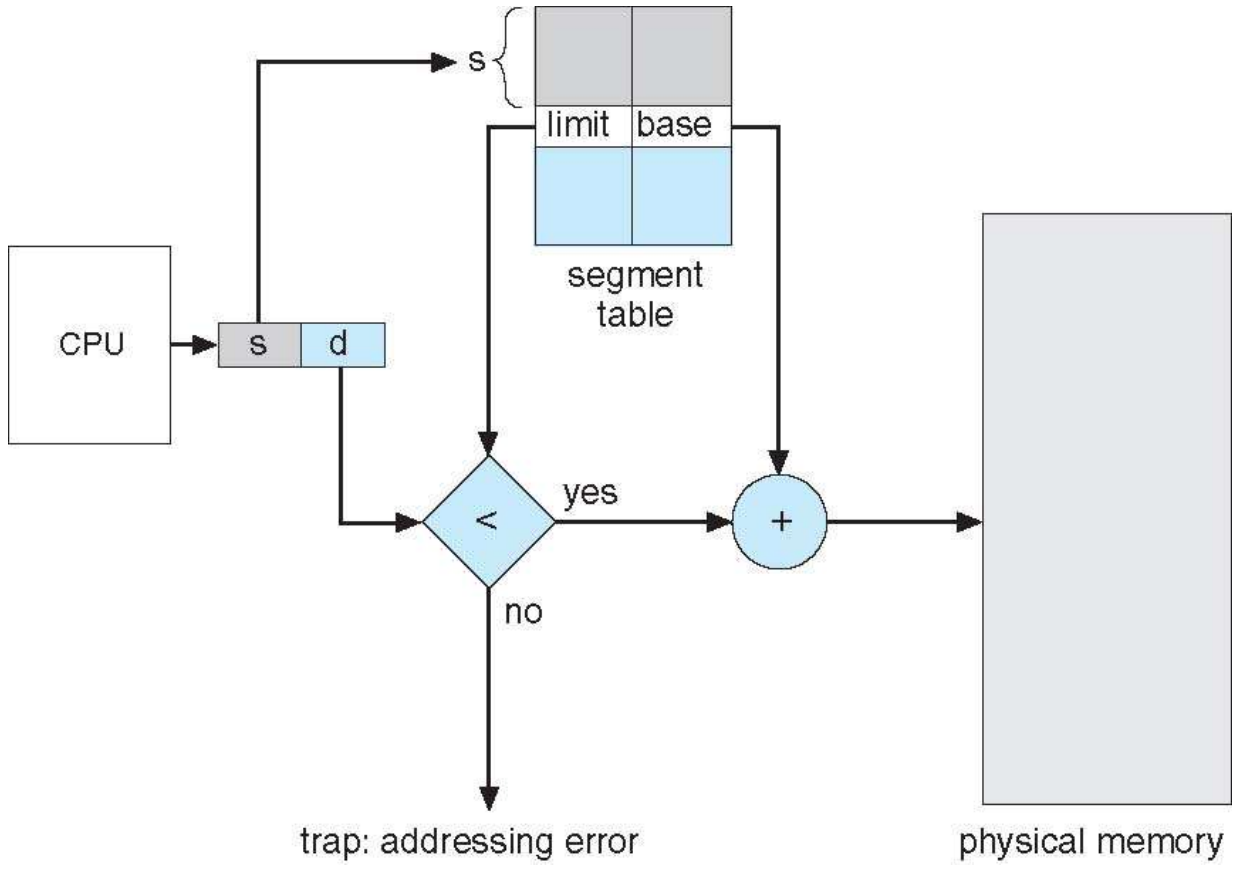
Segmentation Architecture (Cont.)

- Protection
  - With each entry in segment table associate:
    - validation bit = 0 $\Rightarrow$ illegal segment
    - read/write/execute privileges
- Protection bits associated with segments; code sharing occurs at segment level
- Since segments vary in length, memory allocation is a dynamic storage-allocation problem
- A segmentation example is shown in the following diagram

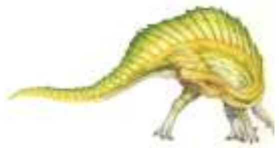


	limit	base
0	1000	1400
1	400	6300
2	400	4300
3	1100	3200
4	1000	4700

Segmentation Hardware

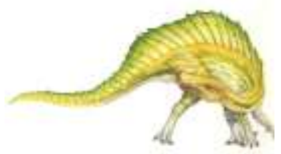


- Segmentation does not avoid external fragmentation and the need for compaction



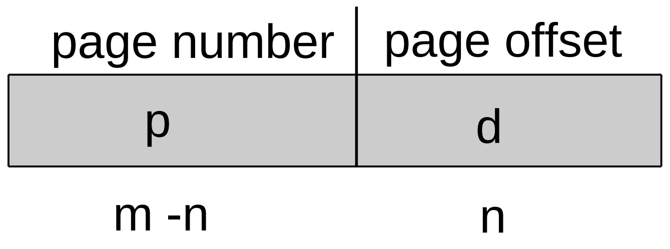
Paging

- Physical address space of a process can be noncontiguous; process is allocated physical memory whenever the latter is available
  - **Avoids external fragmentation**
  - **Avoids problem of varying sized memory chunks**
- Divide physical memory into fixed-sized blocks called **frames**
  - Size is power of 2, between 512 bytes and 16 Mbytes
- **Divide logical memory into blocks of same size called pages**
- Keep track of all free frames
- To run a program of size ***N*** pages, need to find ***N*** free frames and load program
- Set up a **page table** to **translate logical to physical addresses**
- **Backing store likewise split into pages**
- **Still have Internal fragmentation**



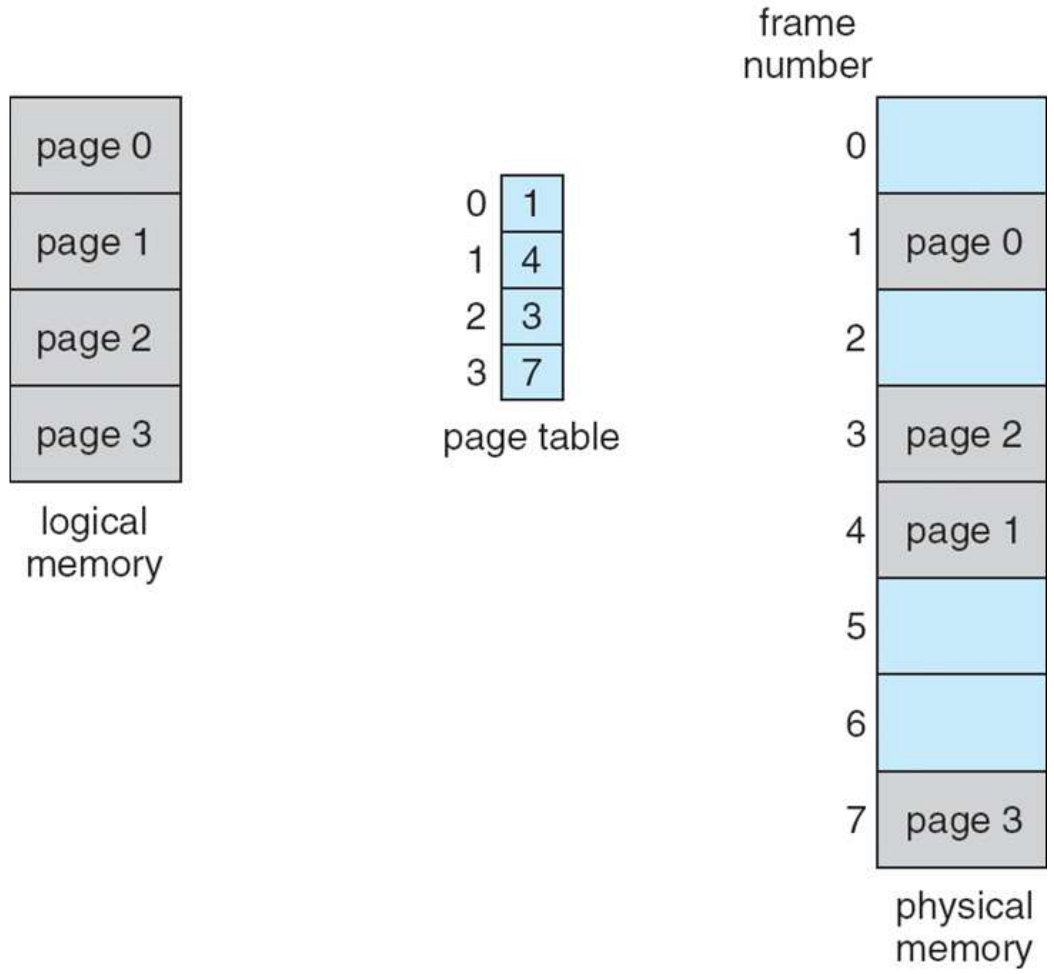
Address Translation Scheme

- Address generated by CPU is divided into:
  - **Page number** (***p***) – used as an index into a **page table** which contains base address of each page in physical memory
  - **Page offset** (***d***) – combined with base address to define the physical memory address that is sent to the memory unit



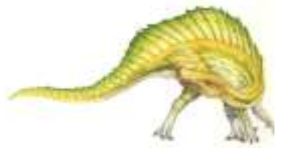
  - For given logical address space 2^m and page size 2^n
  - where p is an index into the page table and d is the displacement within the page.
  - That is why we use a page size of power of 2.

Using a page size of 4 bytes and a physical memory of 32 bytes (8 pages),

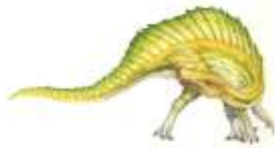


- Logical address 0 is page 0, offset 0.
- Indexing into the page table, we find that page 0 is in frame 5.
- Thus, logical address 0 maps to physical address 20 [= (5 × 4) + 0].
- Logical address3 (page 0, offset3) maps to physical address 23

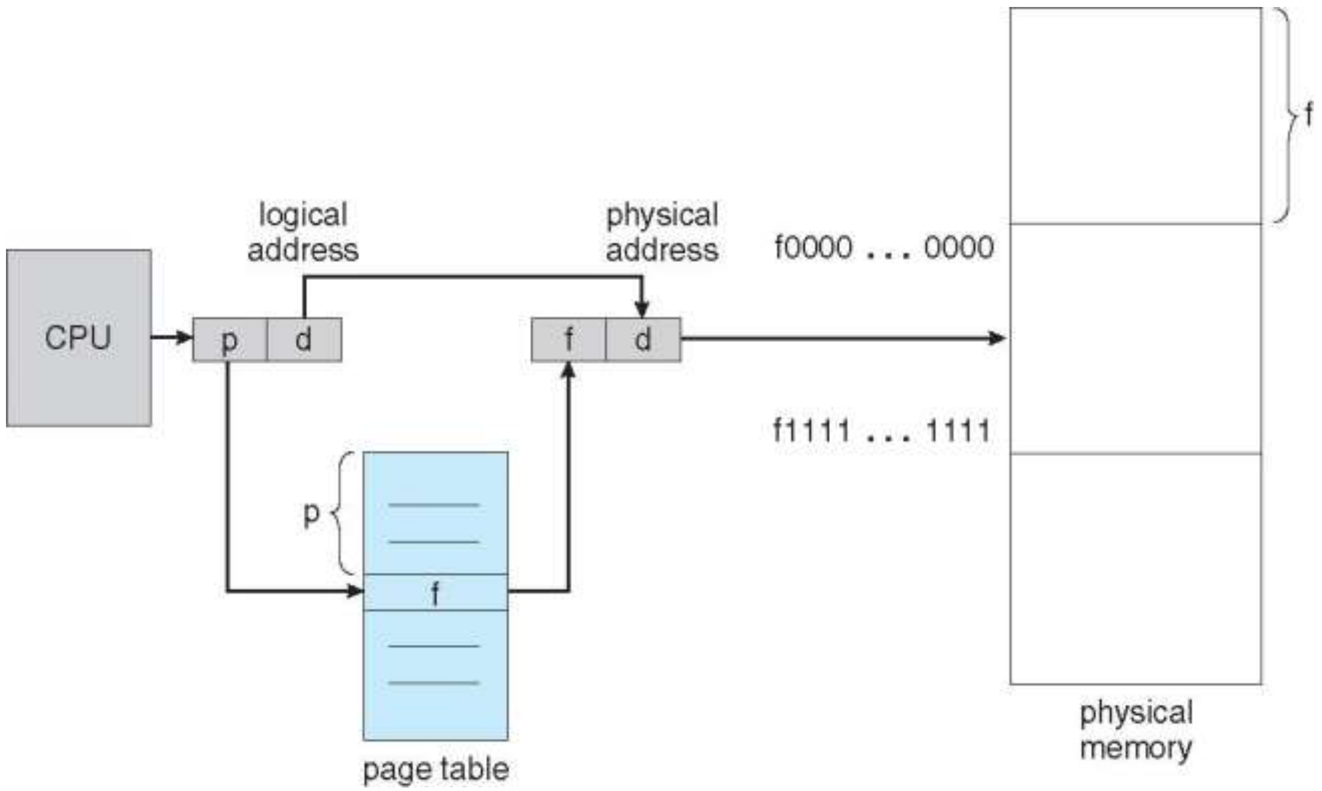
  [= (5 × 4) + 3].
- Logical address 4 is page 1, offset 0; according to the page table, page 1 is mapped to frame 6.
- Thus, logical address 4 maps to physical address 24 [= (6 × 4) + 0].



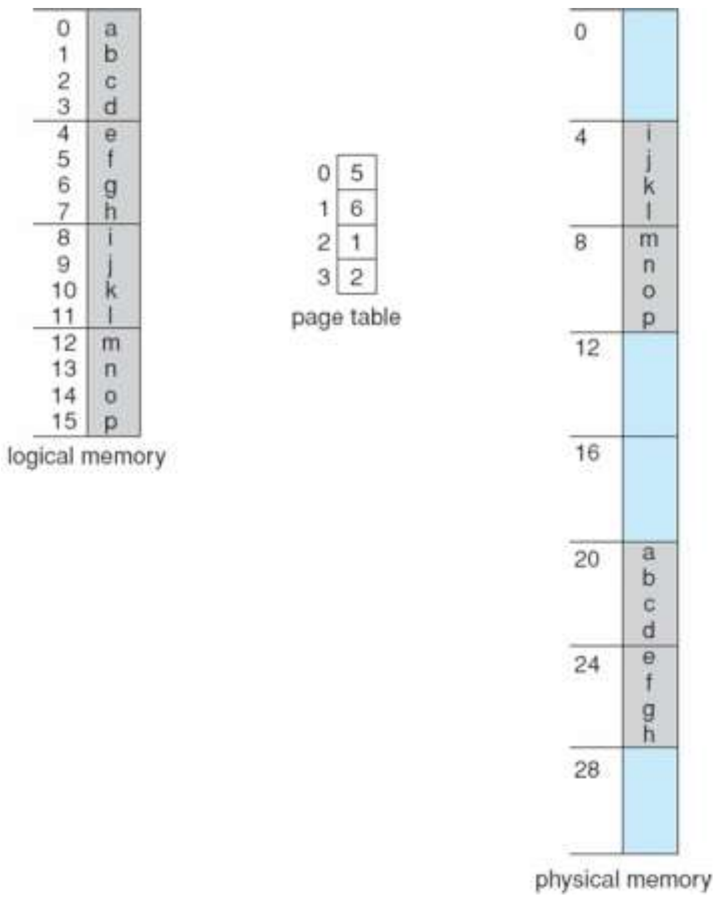
- When we use a paging scheme,
- we have no external fragmentation: any free frame can be allocated to a process that needs it.
- However, we may have some internal fragmentation.



Paging Hardware



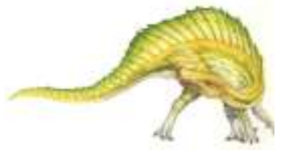
Paging Example



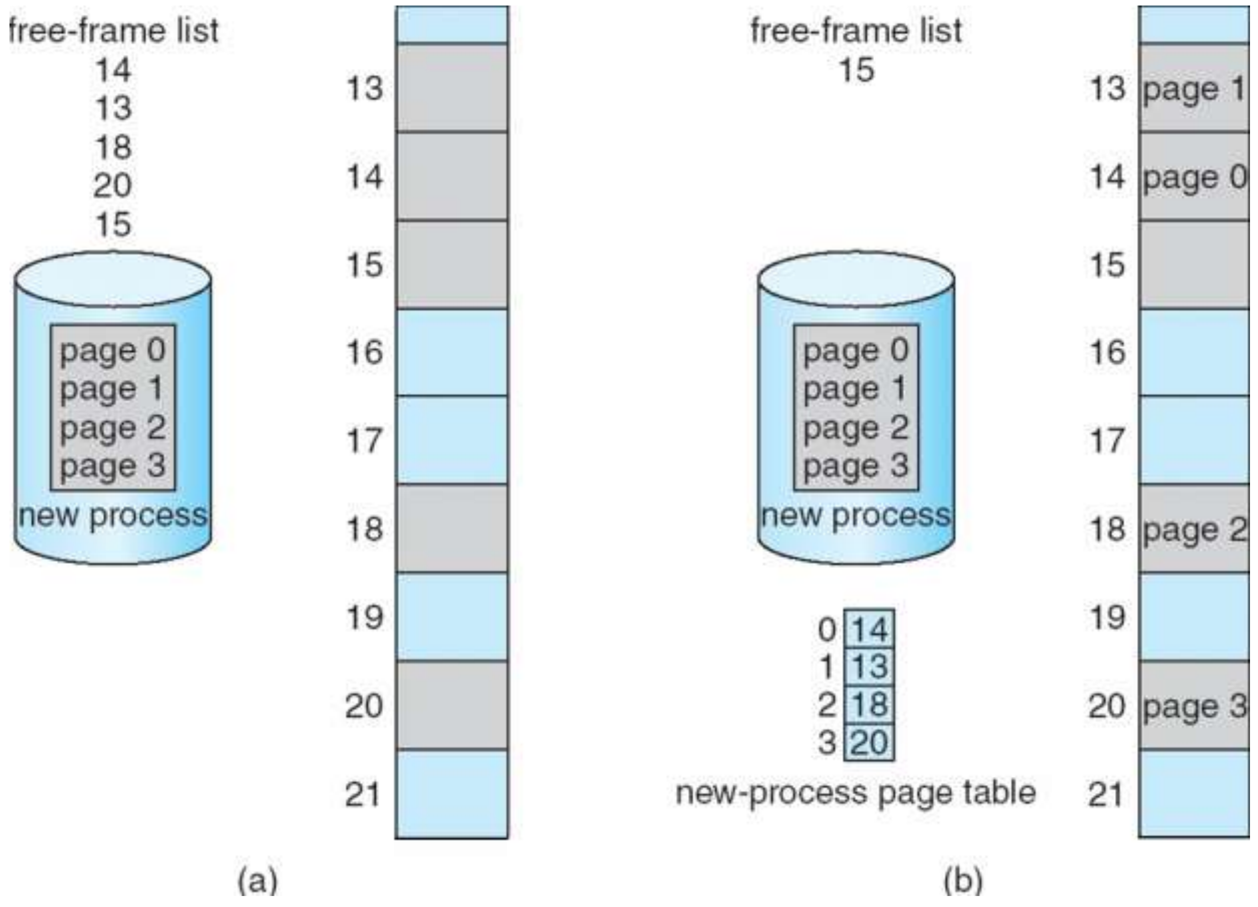
n=2 and *m*=4   32-byte memory and 4-byte pages

Paging (Cont.)

- Calculating internal fragmentation
  - Page size = 2,048 bytes
  - Process size = 72,766 bytes
  - 35 pages + 1,086 bytes
  - Internal fragmentation of 2,048 - 1,086 = 962 bytes
  - Worst case fragmentation = 1 frame – 1 byte
  - On average fragmentation = 1 / 2 frame size
  - So small frame sizes desirable?
  - But each page table entry takes memory to track
  - Page sizes growing over time
    - Solaris supports two page sizes – 8 KB and 4 MB
- Process view and physical memory now very different
- By implementation process can only access its own memory



Free Frames

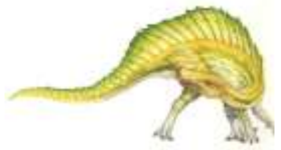


Before allocation

After allocation

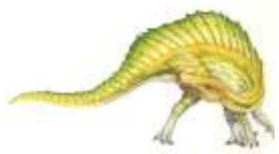
Implementation of Page Table

- Page table is kept in main memory
- **Page-table base register** (**PTBR**) points to the page table
- **Page-table length register** (**PTLR**) indicates size of the page table
- In this scheme every data/instruction access requires two memory accesses
  - One for the page table and one for the data / instruction
- The two memory access problem can be solved by the use of a special fast-lookup hardware cache called **associative memory** or **translation look-aside buffers** (**TLBs**)



Implementation of Page Table (Cont.)

- Some TLBs store **address-space identifiers** (**ASIDs**) in each TLB entry – uniquely identifies each process to provide address-space protection for that process
  - Otherwise need to flush at every context switch
- TLBs typically small (64 to 1,024 entries)
- On a TLB miss, value is loaded into the TLB for faster access next time
  - Replacement policies must be considered
  - Some entries can be **wired down** for permanent fast access



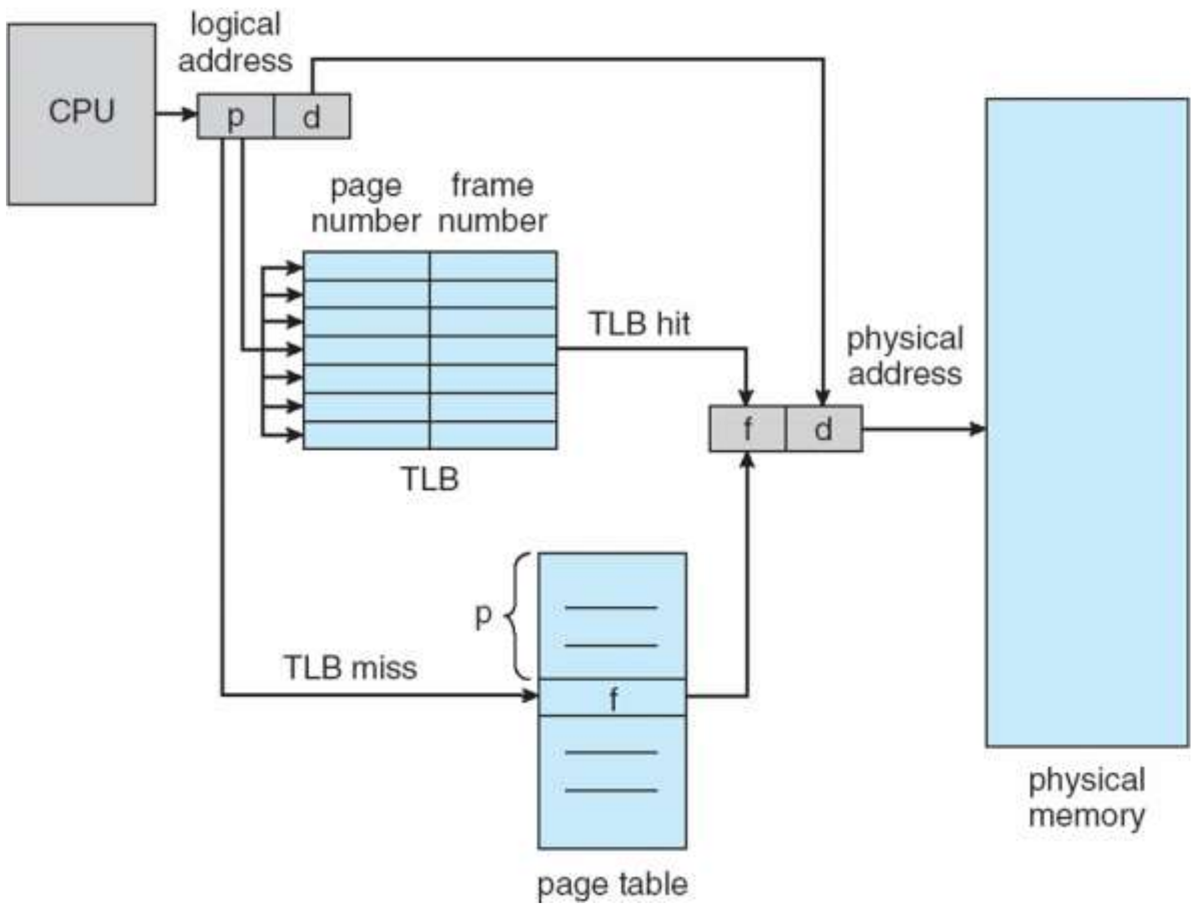
Associative Memory

- Associative memory – parallel search

Page #	Frame #

- Address translation (p, d)
  - If p is in associative register, get frame # out
  - Otherwise get frame # from page table in memory

Paging Hardware With TLB



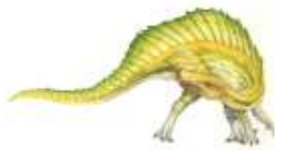
Effective Access Time

- Associative Lookup = ε time unit
  - Can be < 10% of memory access time
- Hit ratio = α
  - Hit ratio – percentage of times that a page number is found in the associative registers; ratio related to number of associative registers
- Consider α = 80%, ε = 20ns for TLB search, 100ns for memory access
- **Effective Access Time** (**EAT**)

$$\text{EAT} = (1\text{xMA} + \varepsilon)\,\alpha + (2\text{xMA} + \varepsilon)(1 - \alpha)$$

$$= 2 + \varepsilon - \alpha \qquad\qquad \text{(MA: Memory Access)}$$

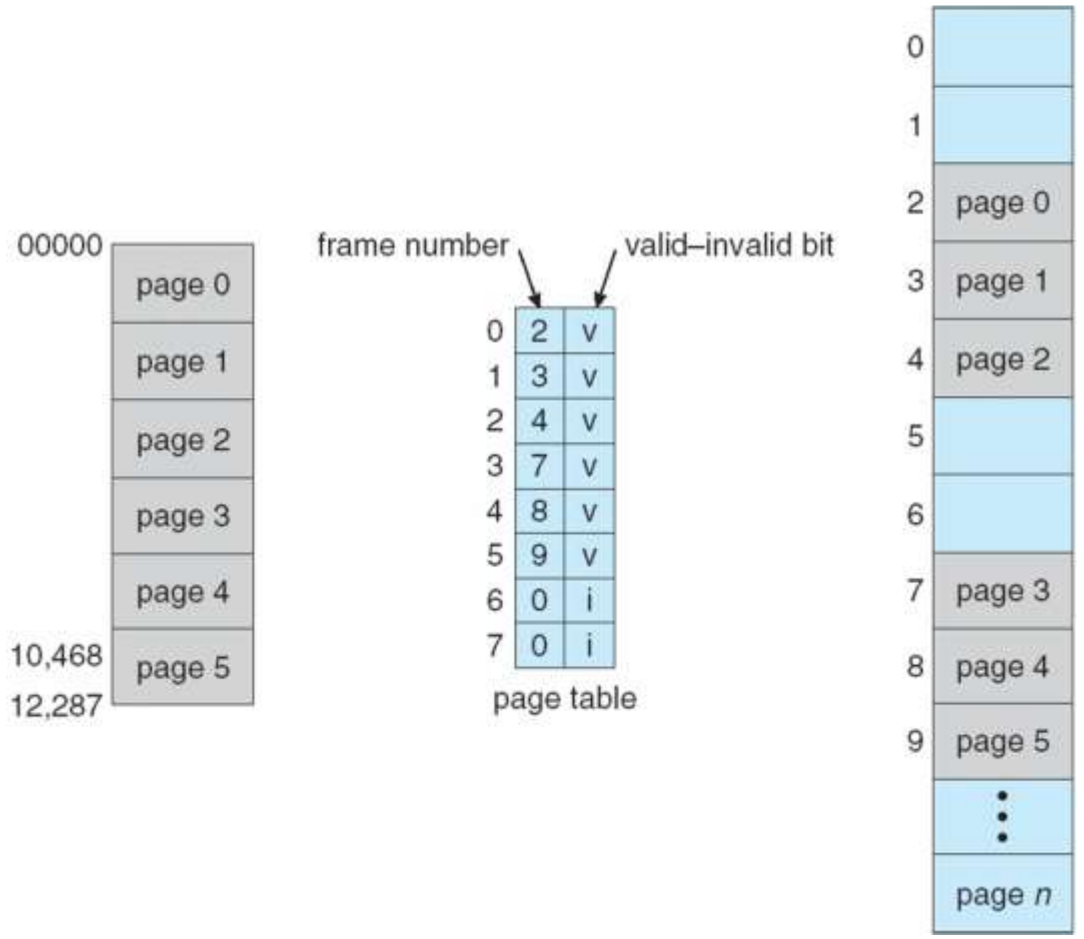
- Consider α = 80%, ε = 20ns for TLB search, 100ns for memory access
  - EAT = (1x100+20) x 0.8 + (2x100+20) x 0.20 = 96 + 44 = 140 ns
- Consider more realistic hit ratio -> α = 99%, ε = 20ns for TLB search, 100ns for memory access
  - EAT = (1x100+20) x 0.99 + (2x100+20) x 0.01 = 118.8 + 2.20 = 121 ns



Memory Protection

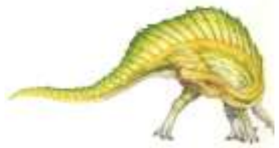
- Memory protection implemented by associating protection bit with each frame to indicate if read-only or read-write access is allowed
  - Can also add more bits to indicate page execute-only, and so on
- **Valid-invalid** bit attached to each entry in the page table:
  - "valid" indicates that the associated page is in the process' logical address space, and is thus a legal page
  - "invalid" indicates that the page is not in the process' logical address space
  - Or use **page-table length register** (**PTLR**)
- Any violations result in a trap to the kernel

Valid (v) or Invalid (i) Bit In A Page Table

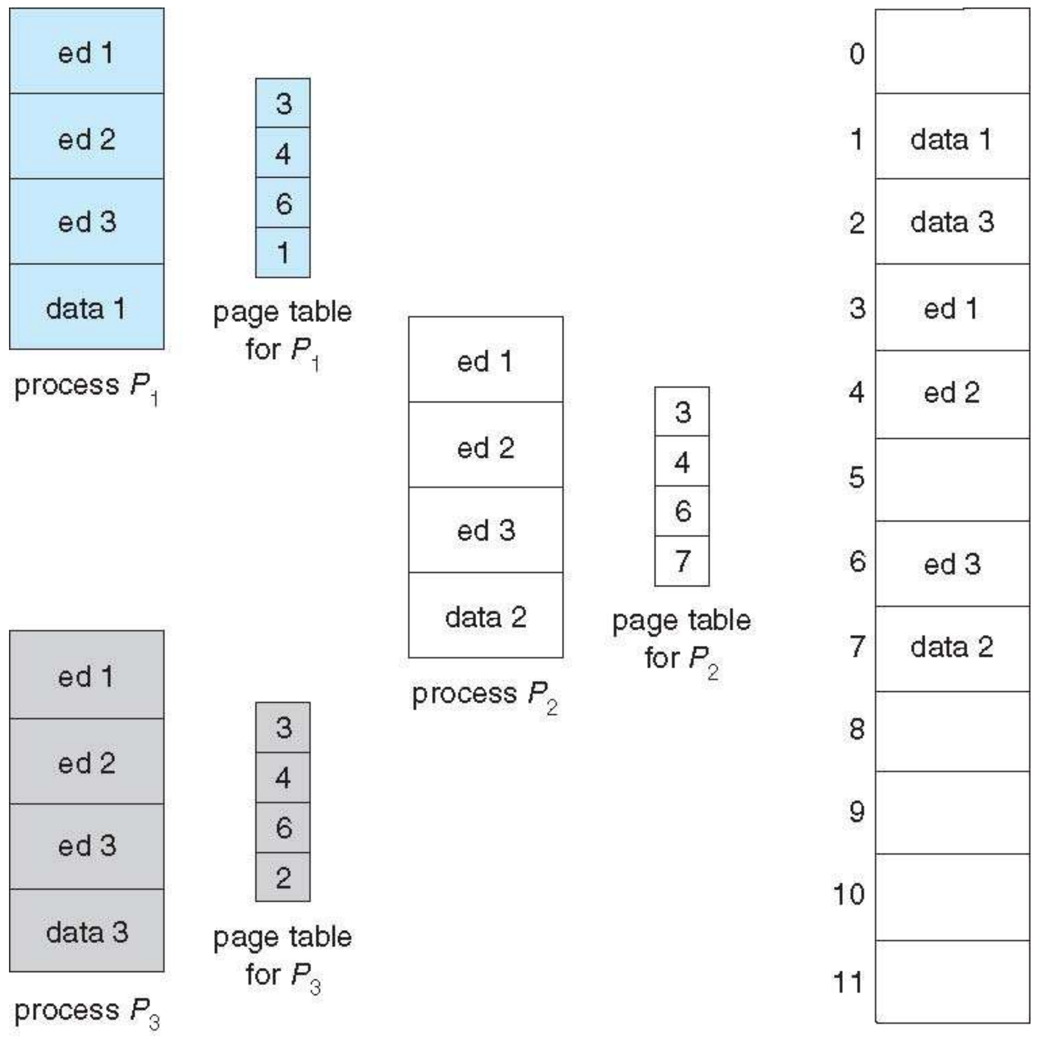


Shared Pages

- **Shared code**
  - One copy of read-only (**reentrant**) code shared among processes (i.e., text editors, compilers, window systems)
  - Similar to multiple threads sharing the same process space
  - Also useful for interprocess communication if sharing of read-write pages is allowed
- **Private code and data**
  - Each process keeps a separate copy of the code and data
  - The pages for the private code and data can appear anywhere in the logical address space



Shared Pages Example



End of Chapter 8