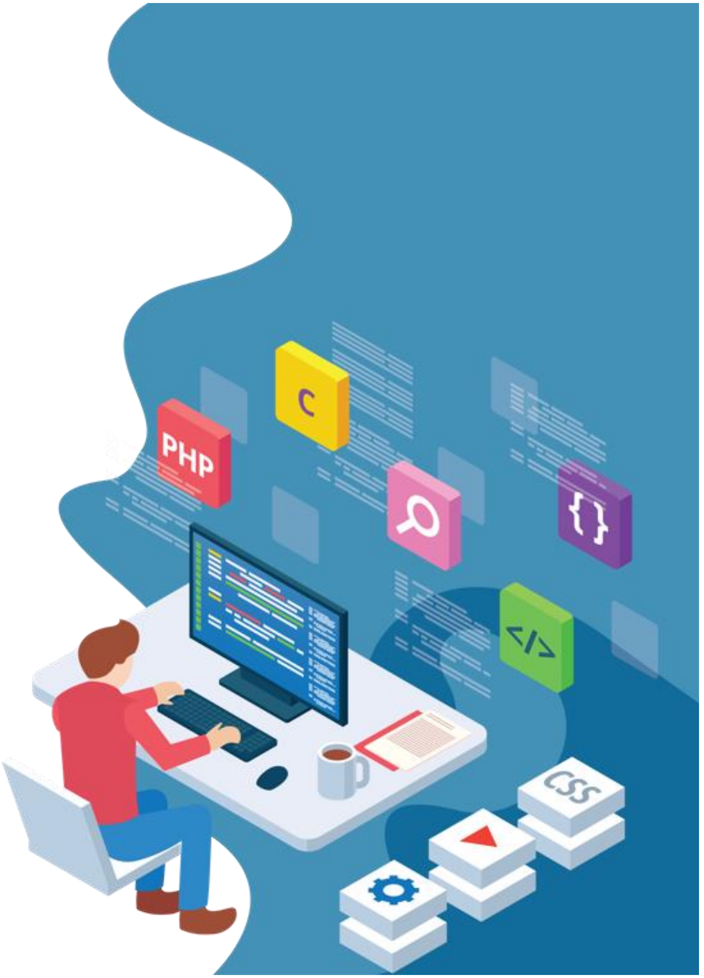
NATIONAL ECONOMICS UNIVERSITY
SCHOOL OF INFORMATION TECHNOLOGY AND DIGITAL ECONOMICS

CHAPTER 2

PROCESS MANAGEMENT

OUTLINE

- Process
- Interprocess Communication
- Thread
- Scheduling Algorithms
- Process Synchonization
- Deadlocks



OBJECTIVES

- Identify the separate components of a process and illustrate how they are represented and scheduled in an operating system.
- Describe how processes are created and terminated in an operating system, including developing programs using the appropriate system calls that perform these operations.
- Identify the basic components of a thread, and contrast threads and processes
- Describe the benefits and challenges of designng multithreaded applications
- Describe various CPU scheduling algorithms
- Describe the critical-section problem and illustrate a race condition
- Illustrate solutions to the critical-section problem using "busy waiting" solutions and « sleep and wakeup » solutions.

OUTLINE

- Process
- Interprocess Communication
- Thread
- Scheduling Algorithms
- Process Synchonization
- Deadlocks

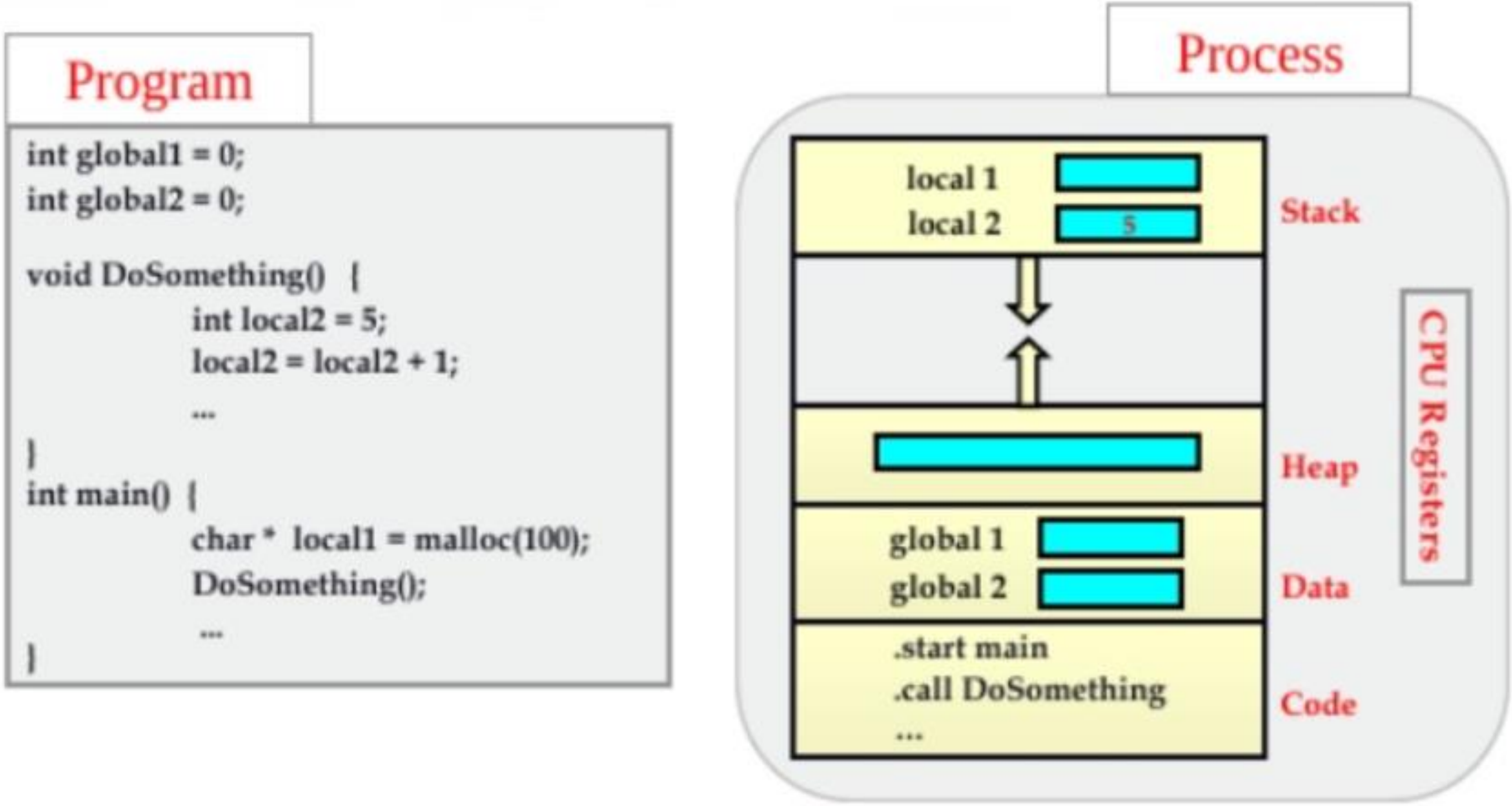
PROCESS CONCEPT

- An operating system executes a variety of programs that run as a process.
- **Process** – a program in execution; process execution must progress in sequential fashion. No parallel execution of instructions of a single process
- Multiple parts
  - The program code, also called **text section**
  - Current activity including **program counter**, processor registers
  - **Stack** containing temporary data
    - Function parameters, return addresses, local variables
  - **Data section** containing global variables
  - **Heap** containing memory dynamically allocated during run time

PROCESS CONCEPT (CONT.)

- Execution of program started via GUI mouse clicks, command line entry of its name, etc.
- One program can be several processes
  - Consider multiple users executing the same program

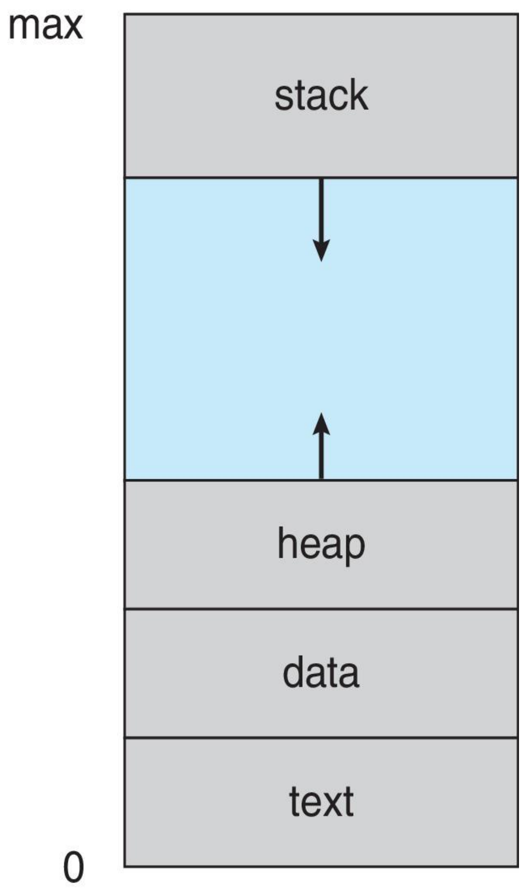
PROGRAM VS PROCESS



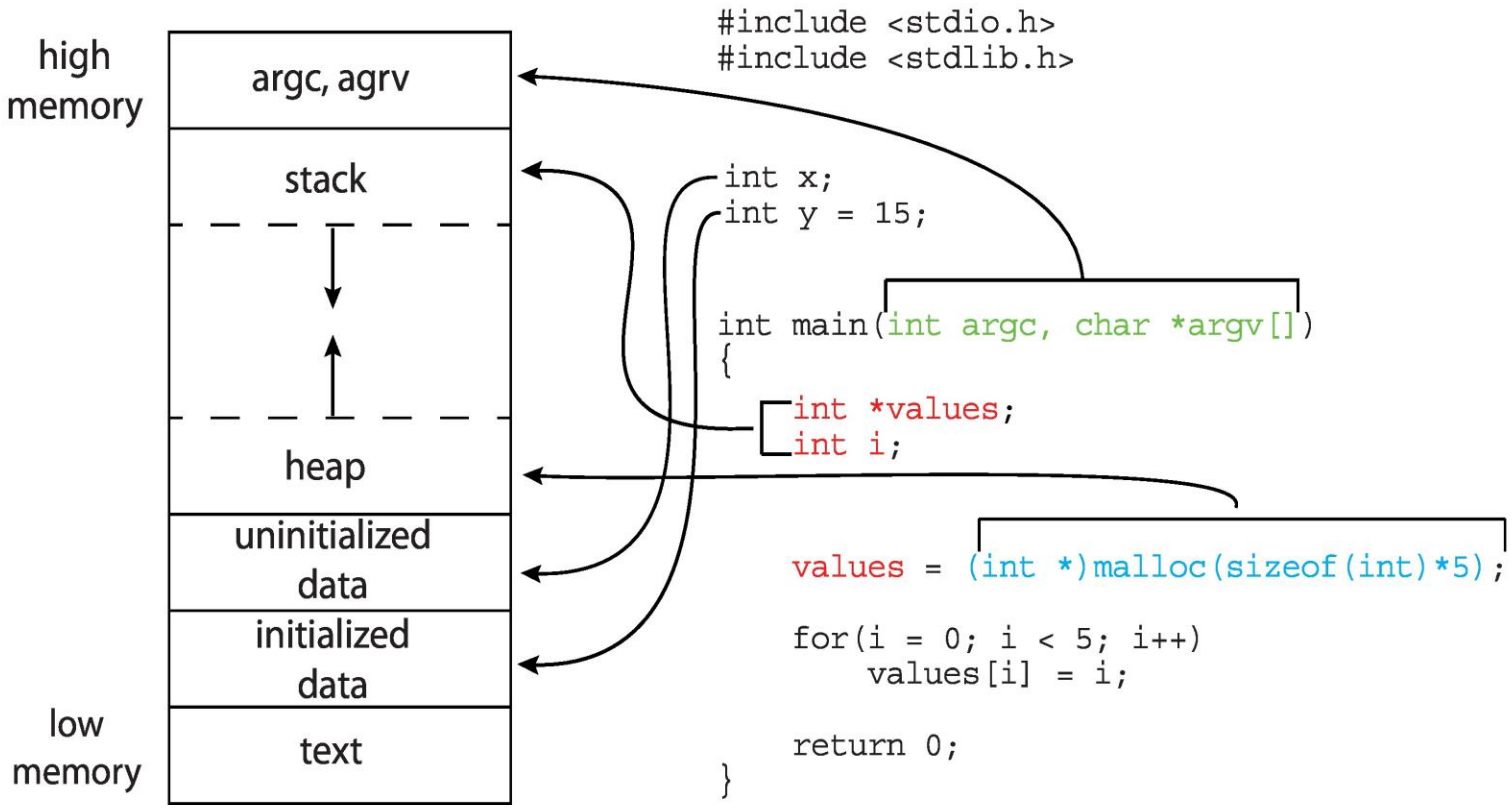
PROGRAM VS PROCESS

- Program is a passive entity, a process is an active one.
- A program becomes a process when an executable file is loaded into memory.
- Two processes may be associated with the same program.
- Process itself can be an execution environment for other code.

PROCESS IN MEMORY



MEMORY LAYOUT OF A C PROGRAM



PROCESS STATE

- As a process executes, it changes *state*
  - new: The process is being created.
  - ready: The process is waiting to be assigned to a process.
  - running: Instructions are being executed.
  - waiting: The process is waiting for some event to occur.
  - terminated: The process has finished execution.

PROCESS STATES EXAMPLES

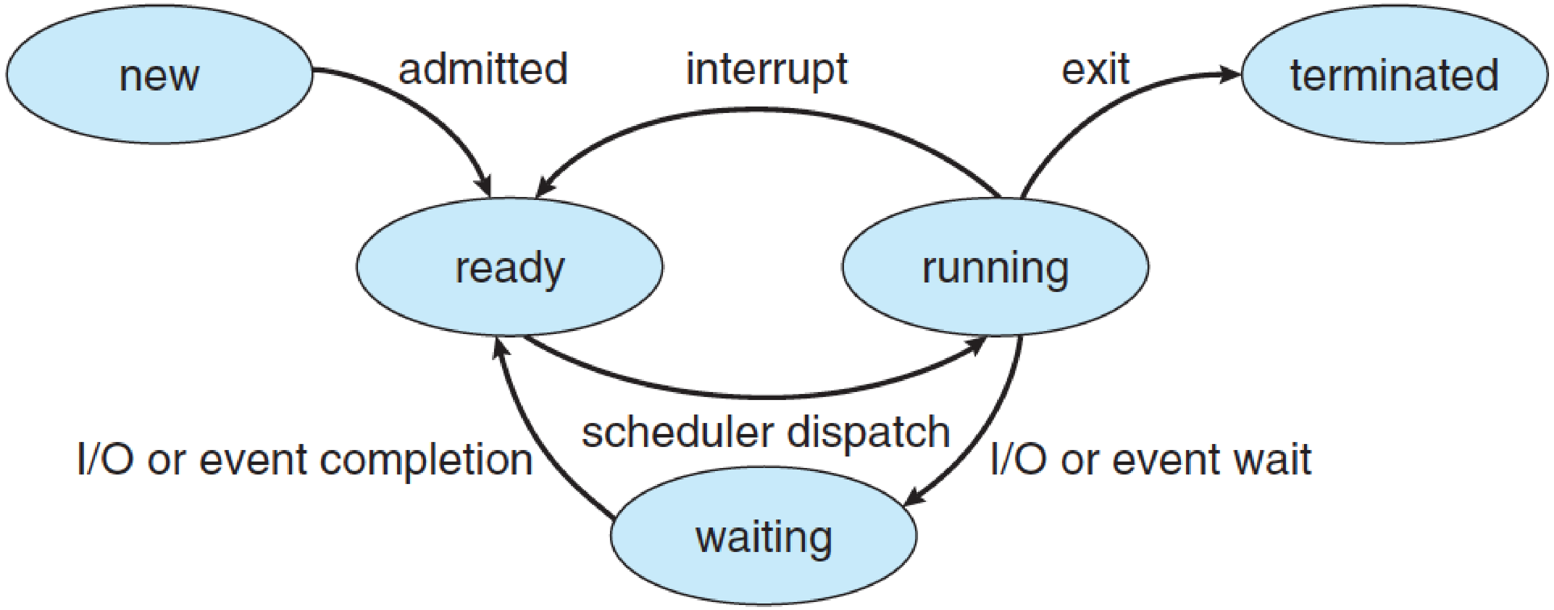
```
void main()
   {
      printf ("Hi");
      exit(0);
   }
```

1.New
2.Ready
3.Running
4.Waiting
5.Ready
6.Running
7.Terminal

```
void main()
 {
     printf ("Hello");
     printf ("Nice to see you");
     exit(0);
  }
```

- How many Process state?

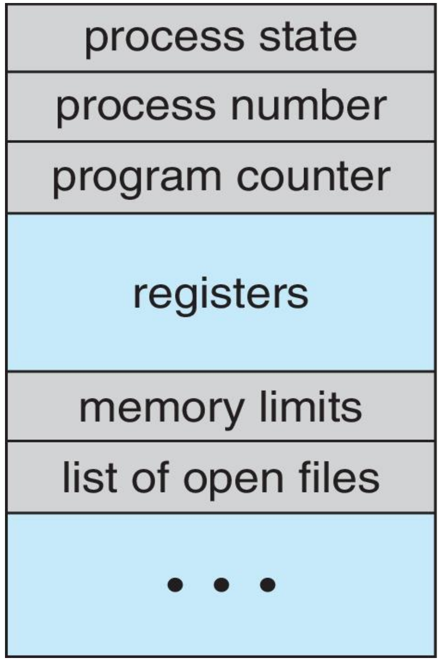
DIAGRAM OF PROCESS STATE



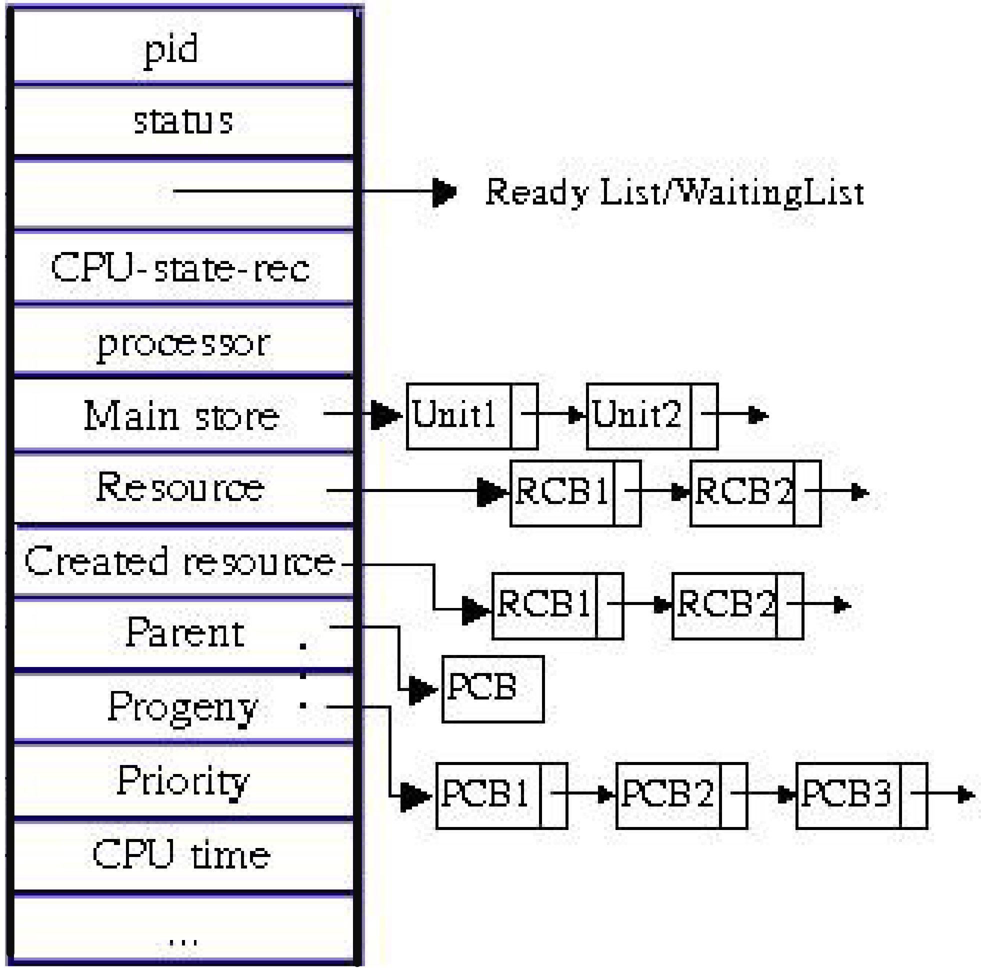
PROCESS CONTROL BLOCK (PCB)

Information associated with each process:

- Process state – running, waiting, etc.
- Program counter – location of instruction to next execute
- CPU registers – contents of all process-centric registers
- CPU scheduling information- priorities, scheduling queue pointers
- Memory-management information – memory allocated to the process
- Accounting information – CPU used, clock time elapsed since start, time limits
- I/O status information – I/O devices allocated to process, list of open files



PCB STRUCTURE



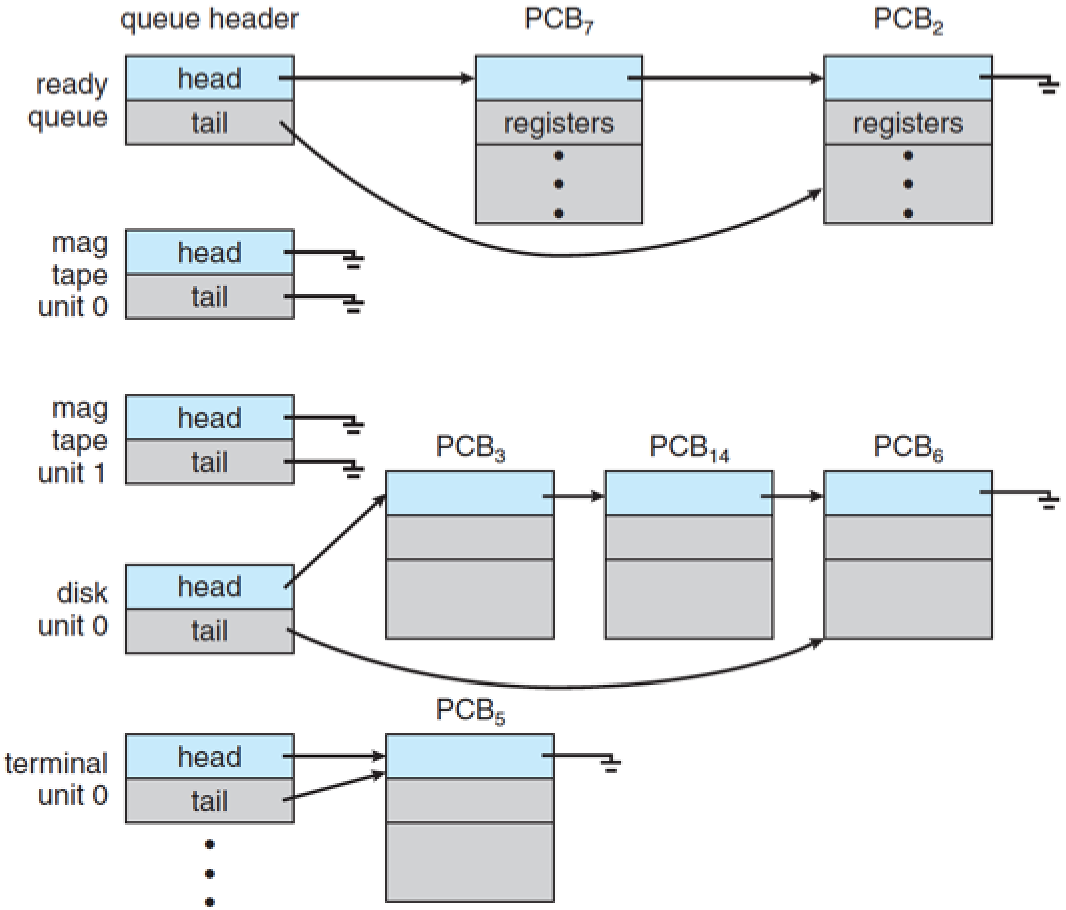
PROCESS SCHEDULING

- **Process scheduler** selects among available processes for next execution on CPU core
- Goal -- Maximize CPU use, quickly switch processes onto CPU core
- Main **scheduling queues** of processes
  - Job queue – set of all processes in the system.
  - **Ready queue** – set of all processes residing in main memory, ready and waiting to execute
  - **Wait queues** – set of processes waiting for an event (i.e., I/O)
  - Device queues – set of processes waiting for an I/O device.
  - Processes migrate among the various queues
  - Select a process in the queue, in the ready state, with the highest priority

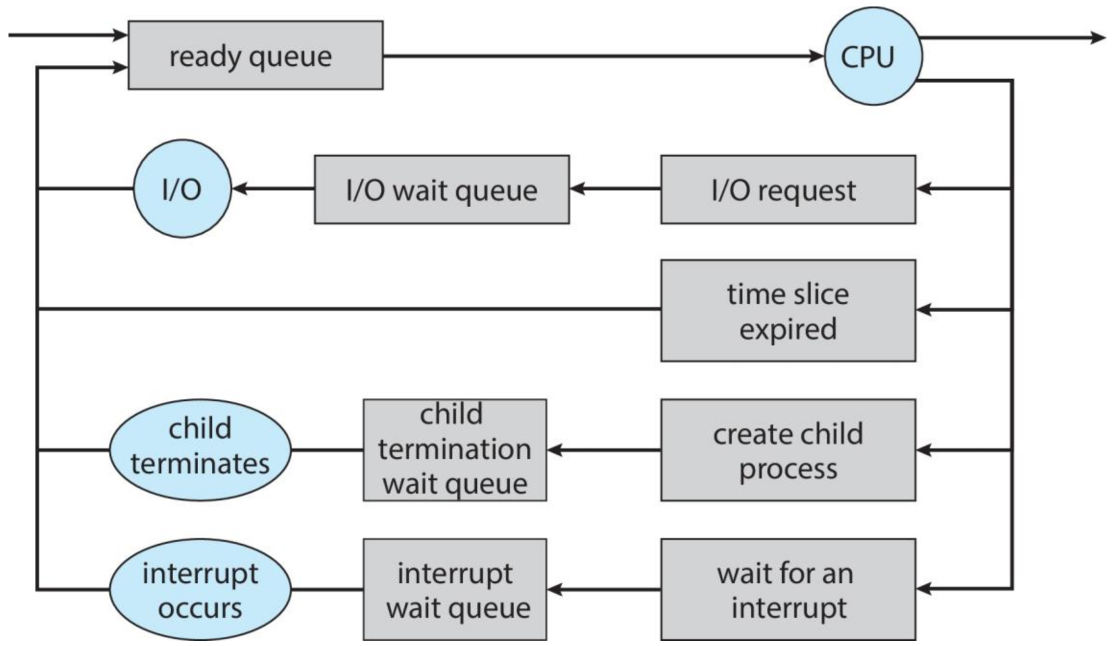
PROCESS SCHEDULING

- The goal of **multiprogramming** is to have multiple processes running at the same time to maximize CPU usage.
- The goal of **time-sharing** is to switch the CPU between processes as often as possible so that the user can interact with each program while it is running.
- If multiple processes exist, they must wait until the CPU is idle and redistributed.

READY AND WAIT QUEUES



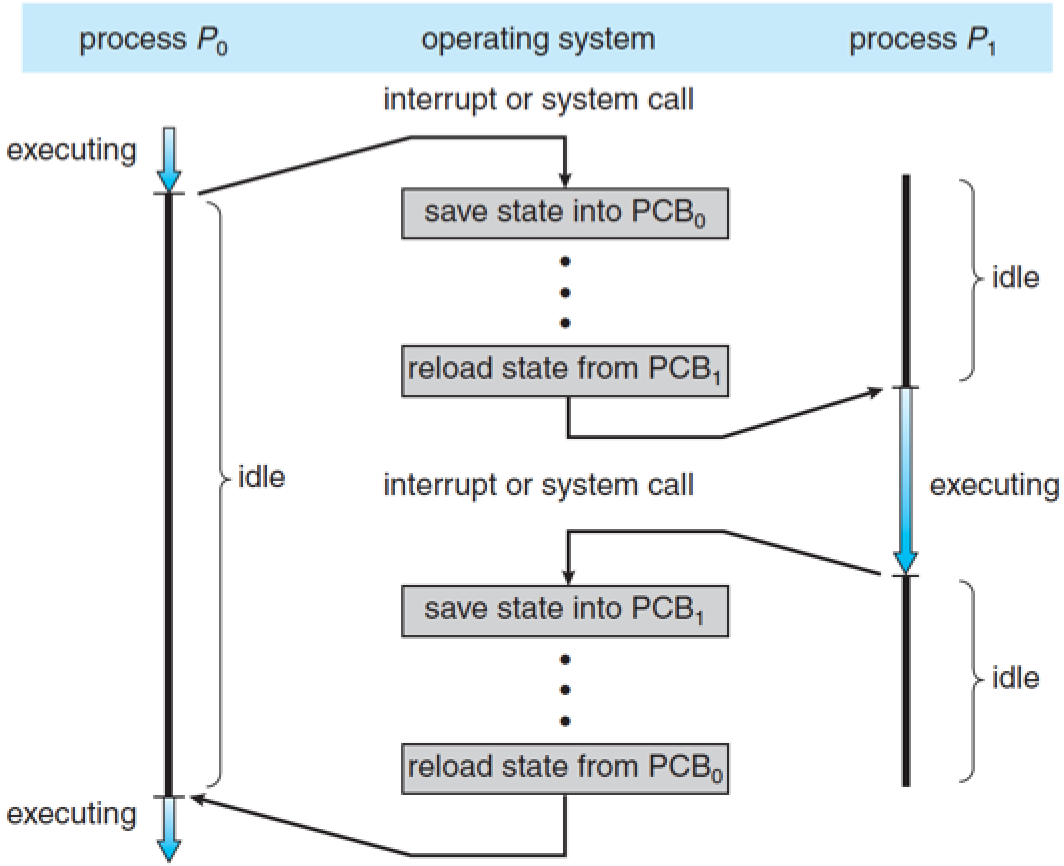
REPRESENTATION OF PROCESS SCHEDULING



PROCESS SCHEDULING QUEUES

- A new process is initially put in the ready queue. It waits there until it is selected for execution. Once the process is allocated the CPU and is executing, one of several events could occur:
  - The process could issue an I/O request and then be placed in an I/O queue.
  - The process could create a new child process and wait for the child's termination.
  - The process could be removed forcibly from the CPU, as a result of an interrupt, and be put back in the ready queue.

CPU SWITCH FROM PROCESS TO PROCESS



CONTEXT SWITCH

- When CPU switches to another process, the system must **save the state** of the old process and load the **saved state** for the new process via a **context switch**
- **Context** of a process represented in the PCB
- Context-switch time is pure overhead; the system does no useful work while switching
  - The more complex the OS and the PCB ➔ the longer the context switch
- Time dependent on hardware support
  - Some hardware provides multiple sets of registers per CPU ➔ multiple contexts loaded at once
- It includes the value of the CPU registers, the process state, and memory-management information.

PROPERTIES OF PROCESS

- ***I/O-bound process***
- ***CPU-bound process***
- Batch processing (via quantum time)
- Process priority
- CPU time used by the process
- Remaining time the process takes to complete the task

MULTITASKING IN MOBILE SYSTEMS

- Some mobile systems (e.g., early version of iOS) allow only one process to run, others suspended
- Due to screen real estate, user interface limits iOS provides for a
  - Single **foreground** process- controlled via user interface
  - Multiple **background** processes– in memory, running, but not on the display, and with limits
  - Limits include single, short task, receiving notification of events, specific long-running tasks like audio playback
- Android runs foreground and background, with fewer limits
  - Background process uses a **service** to perform tasks
  - Service can keep running even if background process is suspended
  - Service has no user interface, small memory use

OPERATIONS ON PROCESSES

- Create new Process
- Destroy or terminal Process
- Suspend Process
- Resume Process
- Change the Process priority

PROCESS CREATION

OS activities when Process Creation:

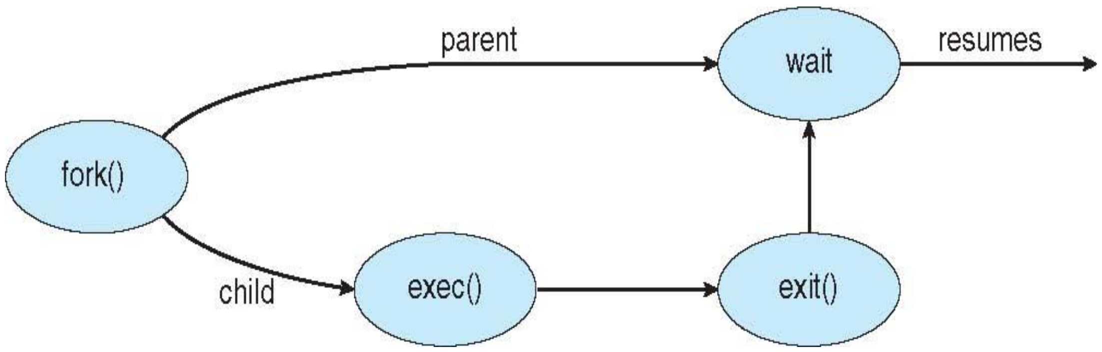
- Identifier for the newly process
- Put the process on the system's management list
- Determine the priority for the process
- Create PCB for Process
- Allocate initial resources to the process

PROCESS CREATION (CONT.)

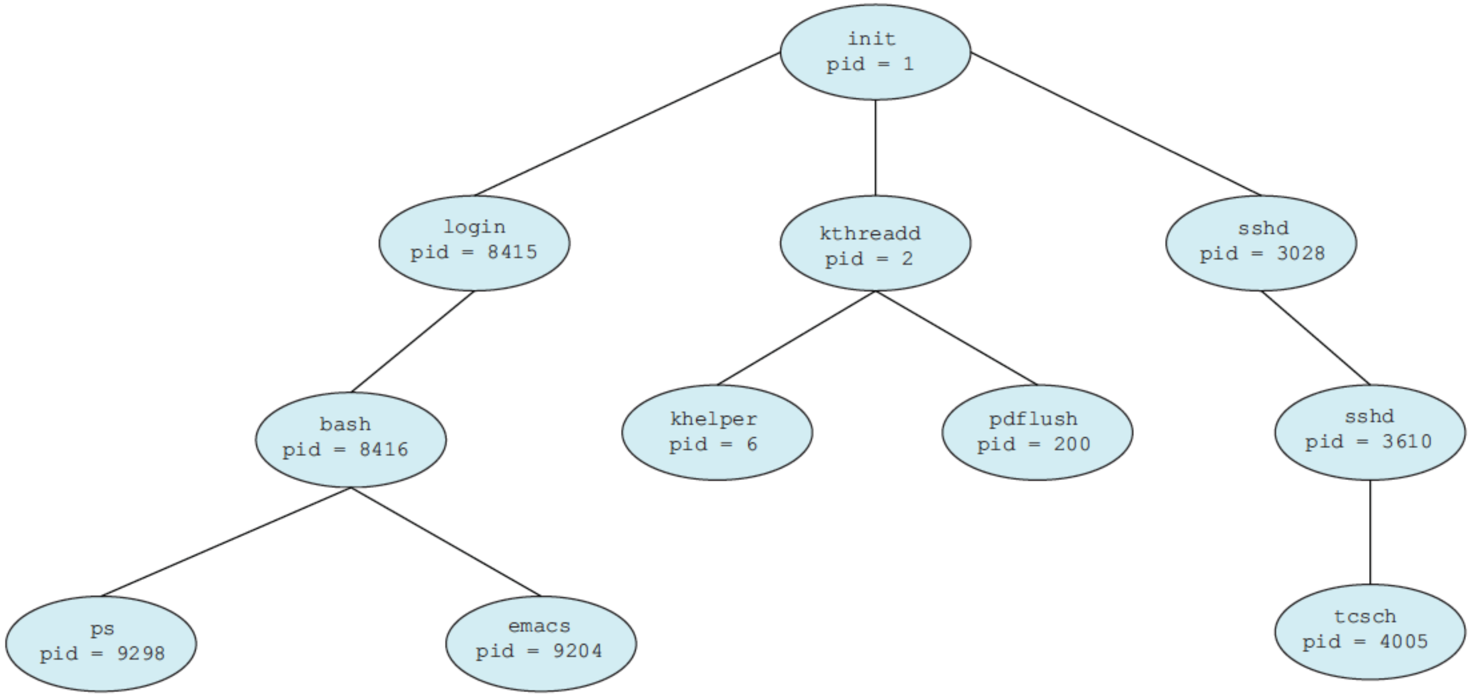
- Parent process creates children processes, which, in turn create other processes, forming a tree of processes.
- Generally, process identified and managed via a process identifier (pid)
- Resource sharing options
  - Parent and children share all resources.
  - Children share subset of parent's resources.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

PROCESS CREATION (CONT.)

- Address space
  - Child duplicate of parent (recursive)
  - Child has a program loaded into it (sub function)
- UNIX examples
  - **`fork()`** system call creates new process
  - **`exec()`** system call used after a **`fork()`** to replace the process' memory space with a new program
  - Parent process calls **`wait()`** waiting for the child to terminate



A TREE OF PROCESSES ON A TYPICAL UNIX SYSTEM



UNIX PROCESS CREATION

```
#include <stdio.h>
#include <unistd.h>
int main()
{
        int ret;
        Printf("before fork \n");
        ret=fork();
        //Creating a child process. This will now throw 2 return values. One>0 and other==0. >0 is parent, equal to 0 is child.
        If(ret>0)
        {
        printf(" \n A Parent ");
        printf(" \n My PID is %d", getpid());
        }
        if (ret==0)
        printf  (" \n A CHILD MAN ");
        printf(" \n My PID is %d", getpid());
        printf(" \n My Parent PID is %d", getppid());
        }
        Printf("common Work \n");
}
```

UNIX PROCESS CREATION

```c
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
int main() {
     pid_t pid;
     pid = fork(); /* fork a child process */
     if (pid < 0) { /* error occurred */
             fprintf(stderr, "Fork Failed");
             return 1;
     }
     else if (pid == 0) { /* child process */
             execlp("/bin/ls","ls",NULL);
     }
     else {  /* parent process */
             /* parent will wait for the child to complete */
             wait(NULL);
             printf("Child Complete");
     }
     return 0;
}
```

EXAMPLE

```
void main()
  {
        printf ("Hi");
        fork();
        printf ("Hello");
        fork();
        printf ("Bye");
   }
```

```
void main()
  {
        pid_t pid;
        printf ("Hi");
        pid=fork();
        If(pid==0)
        {
        fork();
        printf ("Hello");
        }else
        printf ("Bye");
   }
```

WINDOWS PROCESS CREATION VIA WINDOWS API

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

int main(VOID)
{
STARTUPINFO si;
PROCESS_INFORMATION pi;

   /* allocate memory */
   ZeroMemory(&si, sizeof(si));
   si.cb = sizeof(si);
   ZeroMemory(&pi, sizeof(pi));

   /* create child process */
   if (!CreateProcess(NULL, /* use command line */
    "C:\\WINDOWS\\system32\\mspaint.exe", /* command */
    NULL, /* don't inherit process handle */
    NULL, /* don't inherit thread handle */
    FALSE, /* disable handle inheritance */
    0, /* no creation flags */
    NULL, /* use parent's environment block */
    NULL, /* use parent's existing directory */
    &si,
    &pi))
   {
     fprintf(stderr, "Create Process Failed");
     return -1;
   }
   /* parent will wait for the child to complete */
   WaitForSingleObject(pi.hProcess, INFINITE);
   printf("Child Complete");

   /* close handles */
   CloseHandle(pi.hProcess);
   CloseHandle(pi.hThread);
}
```

PROCESS CREATION (CONT.)

- On UNIX systems, we can obtain a listing of processes by using the **ps** command. For example, the command **ps –el** will list complete information for all processes currently active in the system
- **pstree** is a UNIX command that shows the running processes
- **Gnome-system-monitor**: Show process manage by GUI

UNISTD C LIBRARY

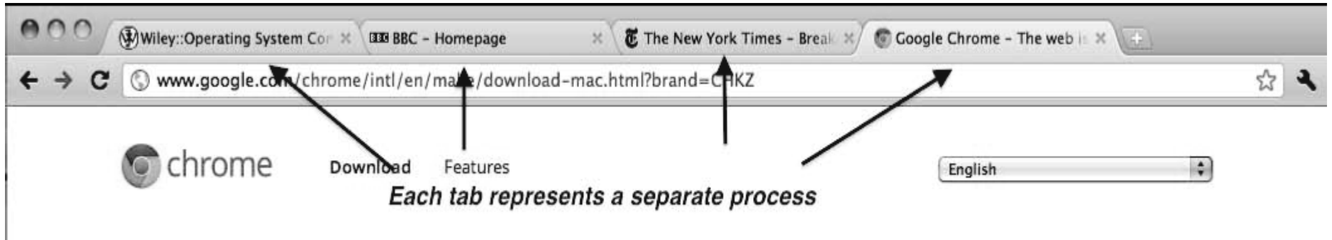
- getpid()
- getppid()
- fork()
- wait()

PROCESS TERMINATION

- Process executes last statement and asks the operating system to decide it (**exit**).
  - Output data from child to parent (via **wait**).
  - Remove PCB of process
  - Remove the process from all system management lists
  - Process' resources are deallocated by operating system (reallocate).
- Parent may terminate execution of children processes (**abort**).
  - Child has exceeded allocated resources.
  - Task assigned to child is no longer required.
  - Parent is exiting. Operating system does not allow child to continue if its parent terminates.

MULTIPROCESS ARCHITECTURE – CHROME BROWSER

- Many web browsers ran as single process (some still do)
  - If one web site causes trouble, entire browser can hang or crash
- Google Chrome Browser is multiprocess with 3 different types of processes:
  - **Browser** process manages user interface, disk and network I/O
  - **Renderer** process renders web pages, deals with HTML, Javascript. A new renderer created for each website opened
    - Runs in **sandbox** restricting disk and network I/O, minimizing effect of security exploits
  - **Plug-in** process for each type of plug-in



OUTLINE

- Process
- Interprocess Communication
- Thread
- Scheduling Algorithms
- Process Synchonization
- Deadlocks

COOPERATING PROCESSES - INTERPROCESS COMMUNICATION

- Processes within a system may be ***independent*** or ***cooperating***
- Cooperating process can affect or be affected by other processes, including sharing data
- Reasons for cooperating processes:
  - Information sharing
  - Computation speedup
  - Modularity
  - Convenience
- Cooperating processes need **interprocess communication** (**IPC**)
- Main models of IPC
  - **Shared memory**
  - **Message passing**

IPC MECHANISMS

- Signal
- Pipe
- Message passing
- Shared memory
- Sockets

SIGNAL

- Software mechanism similar to hardware interrupts that affect processes
- A signal used to notify the process of an event occurring
- There are many signals defined, each of which has a meaning corresponding to a particular event (**End Task, Close all** on the taskbar)
- Each process has a table representing different signals. For each signal, there will be a corresponding **signal handler** that regulates the processing of the process when receiving the corresponding signal.

SIGNAL (CONT.)

- Signals are sent by:
  - Hardware (eg errors due to arithmetic operations)
  - OS sends to a process
  - A process sends to another process (e.g. parent requesting a child process to terminate)
  - User (eg press Alt-F4 to interrupt process)
- When a process receives a signal, it can behave in one of the following ways:
  - Ignore signal;
  - Signal Processing by default;
  - Receive the signal and process it in a special way of the process

ORDINARY PIPES

- An unidirectional communication channel between two processes: the output of one process is passed as input to the other in the form of a stream of bytes.
- When a pipe is established between two processes
  - A process writes data to pipe
  - The other process will read data from the pipe
- The order of data passed through the pipe is preserved according to the FIFO principle
- Ordinary pipes – cannot be accessed from outside the process that created it. Typically, a parent process creates a pipe and uses it to communicate with a child process that it created.
- Named pipes – can be accessed without a parent-child relationship.

NAMED PIPES

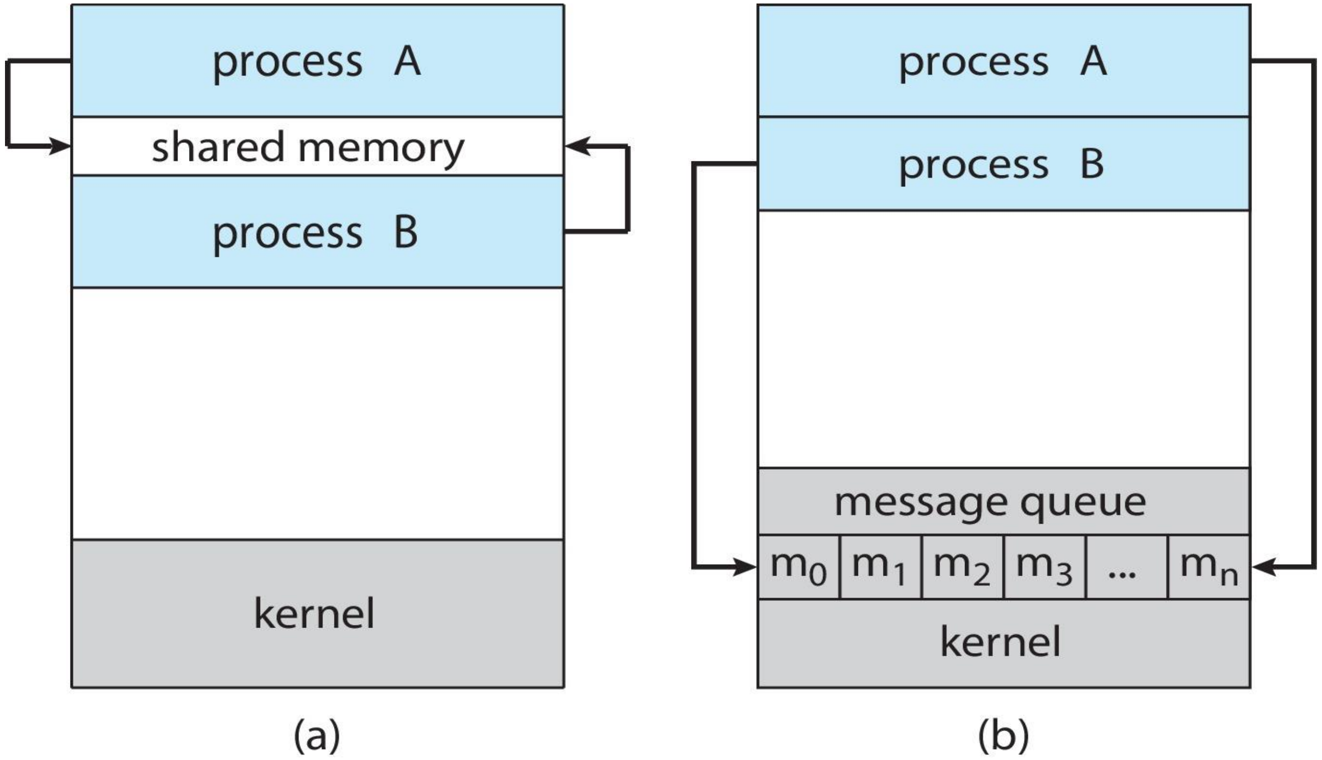
- Named Pipes are more powerful than ordinary pipes
- Communication is bidirectional
- No parent-child relationship is necessary between the communicating processes
- Several processes can use the named pipe for communication
- Provided on both UNIX and Windows systems

COOPERATING PROCESSES - COMMUNICATIONS

- There are two fundamental models of interprocess communication:
  - shared memory
  - message passing

(a) Shared memory.

(b) Message passing.



PRODUCER-CONSUMER PROBLEM

- Paradigm for cooperating processes:
  - *producer* process produces information that is consumed by a *consumer* process
- Two variations:
  - **unbounded-buffer** places no practical limit on the size of the buffer:
    - Producer never waits
    - Consumer waits if there is no buffer to consume
  - **bounded-buffer** assumes that there is a fixed buffer size
    - Producer must wait if all buffers are full
    - Consumer waits if there is no buffer to consume

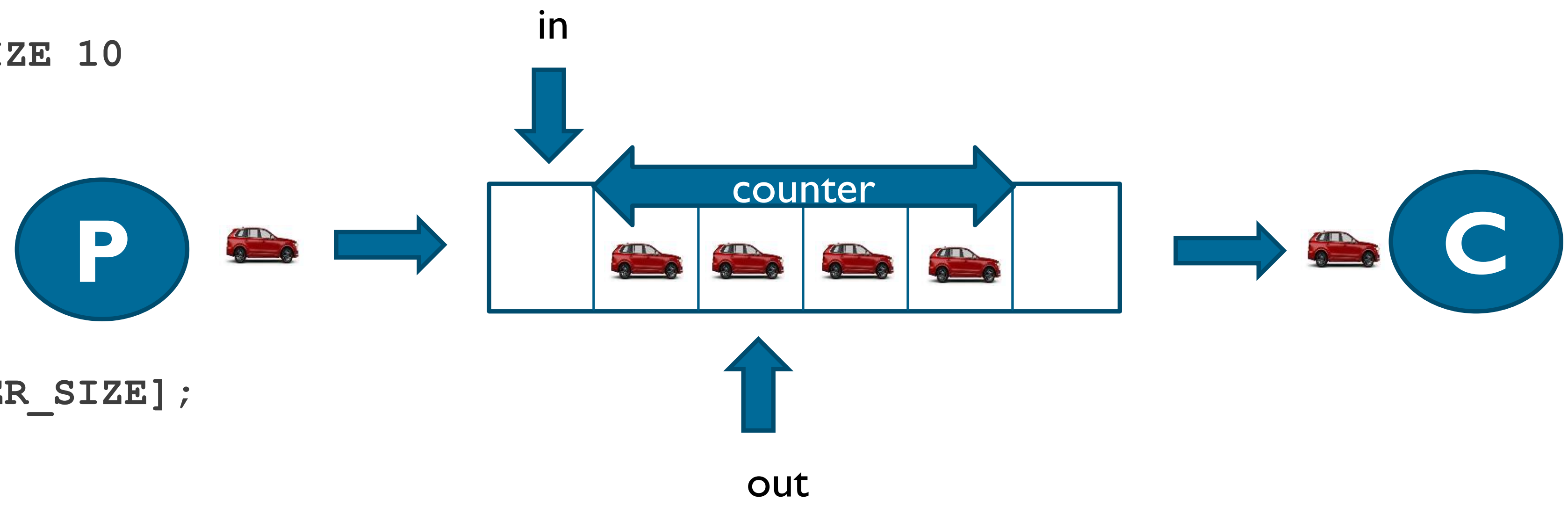
IPC – SHARED MEMORY

- An area of memory shared among the processes that wish to communicate
- The communication is under the control of the users processes not the operating system.
- Major issues is to provide mechanism that will allow the user processes to synchronize their actions when they access shared memory.

BOUNDED-BUFFER – SHARED-MEMORY SOLUTION

- Shared data

```
#define BUFFER_SIZE 10
typedef struct {
  . . .
} item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```



- Solution is correct, but can only use **BUFFER_SIZE-1** elements

WHAT ABOUT FILLING ALL THE BUFFERS?

- 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.
- The integer `counter` is incremented by the producer after it produces a new buffer.
- The integer `counter` is and is decremented by the consumer after it consumes a buffer.

PRODUCER PROCESS – SHARED MEMORY

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

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

IPC – MESSAGE PASSING

- Mechanism for processes to communicate and to synchronize their actions.
- Message system – processes communicate with each other without resorting to shared variables.
- IPC facility provides two operations:
  - **send**(*message*) – message size fixed or variable
  - **receive**(*message*)

IMPLEMENTATION QUESTIONS

- If P and Q wish to communicate, they need to:
  - establish a communication link between them
  - exchange messages via send/receive
- There are many ways to implement association between two processes and implement corresponding send /receive actions: direct or indirect communication, synchronous or asynchronous communication
- The unit of information exchanged is the message, messages can have a structure

PRODUCER-CONSUMER: MESSAGE PASSING

- Producer

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

 send(next_produced);
}
```

- Consumer

```
message next_consumed;
while (true) {
receive(next_consumed)

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

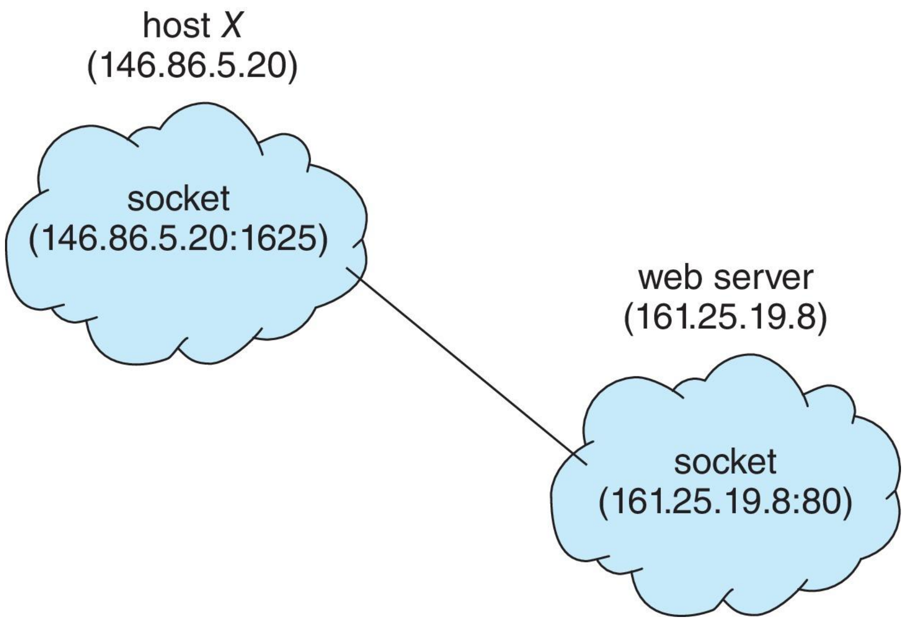
BUFFERING

- Queue of messages attached to the link; implemented in one of three ways.
  1. Zero capacity (No buffering) – 0 messages
     Sender must wait for receiver (rendezvous).
  2. Bounded capacity – finite length of *n* messages
     Sender must wait if link full.
  3. Unbounded capacity – infinite length
     Sender never waits.

SOCKETS

- A **socket** is defined as an endpoint for communication
- Concatenation of IP address and **port** – a number included at start of message packet to differentiate network services on a host
- The socket **161.25.19.8:1625** refers to port **1625** on host **161.25.19.8**
- Communication consists between a pair of sockets
- All ports below 1024 are ***well known***, used for standard services

SOCKET COMMUNICATION



OUTLINE

- Process
- Interprocess Communication
- Thread
- Scheduling Algorithms
- Process Synchonization
- Deadlocks

TDM-GCC 4.9.2 64-bit Release

(globals)

Project Classes Debug

thread windows.cpp

```
#include <Windows.h>
#include <process.h>
#include <stdio.h>
int sum1=0,sum2=0;
volatile int i;
void mythread1(void* data)
{
    printf("mythread1 ID %d \n", GetCurrentThreadId());
    for (i = 0; i < 10; i++)
        if (i%2==0){

    sum1=sum1+i;}
        printf("sum= %d \n",sum1);
}

void mythread2(void* data)
{
    for (i = 0; i < 10; i++)
    if (i%2!=0){
    sum2=sum2+i;
        }
    printf("sum= %d \n",sum2);
    printf("mythread2 ID %d \n", GetCurrentThreadId());
}

int main(int argc, char* argv[])
{
    HANDLE myhandle1, myhandle2;
    myhandle1 = (HANDLE)_beginthread(&mythread1, 0, 0);
    myhandle2 = (HANDLE)_beginthread(&mythread2, 0, 0);
    WaitForSingleObject(myhandle1, INFINITE);
    WaitForSingleObject(myhandle2, INFINITE);
```

Compiler Resources Compile Log Debug Find Results

Line: 1 Col: 1 Sel: 0 Lines: 35 Length: 750 Insert Done parsing in 6.453 seconds

File Edit Search View Project Execute Tools AStyle Window Help

TDM-GCC 4.9.2 64-bit Release

(globals)

Project | Classes | Debug

thread windows.cpp | mutexlock.cpp

```cpp
#include <unistd.h>
#include <sys/types.h>
#include <pthread.h>
#define NUM_LOOPS 200
long long sum=0;
pthread_mutex_t mutex =PTHREAD_MUTEX_INITIALIZER;

void* counting_thread(void *arg)
{
    int offset=*(int *) arg;
    for(int i=0; i<NUM_LOOPS;i++){
    //start critical section
//  pthread_mutex_lock(&mutex);
    sum +=offset;
    //end critical section
//  pthread_mutex_unlock(&mutex);
    }
    pthread_exit(NULL);

}
int main()
{
    pthread_t id1;
    int offset1=1;
    pthread_create(&id1,NULL, counting_thread,&offset1);
    pthread_t id2;
      //  offset1=-1;
    int offset2=-1;
    pthread_create(&id2,NULL, counting_thread,&offset2);
//  pthread_create(&id2,NULL, counting_thread,&offset1);

// wait for thread finish
```

Compiler | Resources | Compile Log | Debug | Find Results

Line: 1 Col: 1 Sel: 0 Lines: 39 Length: 849 Insert Done parsing in 0.328 seconds

THREADS

- 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 while thread creation is light-weight
- Can simplify code, increase efficiency
- Kernels are generally multithreaded

THREADS (CONT.)

- A *thread* (or *lightweight process*) is a basic unit of CPU utilization; it consists of:
  - program counter
  - register set
  - stack space
- A thread shares with its peer threads its:
  - code section
  - data section
  - operating-system resources

  collectively know as a *task*.
- A traditional or *heavyweight* process is equal to a task with one thread

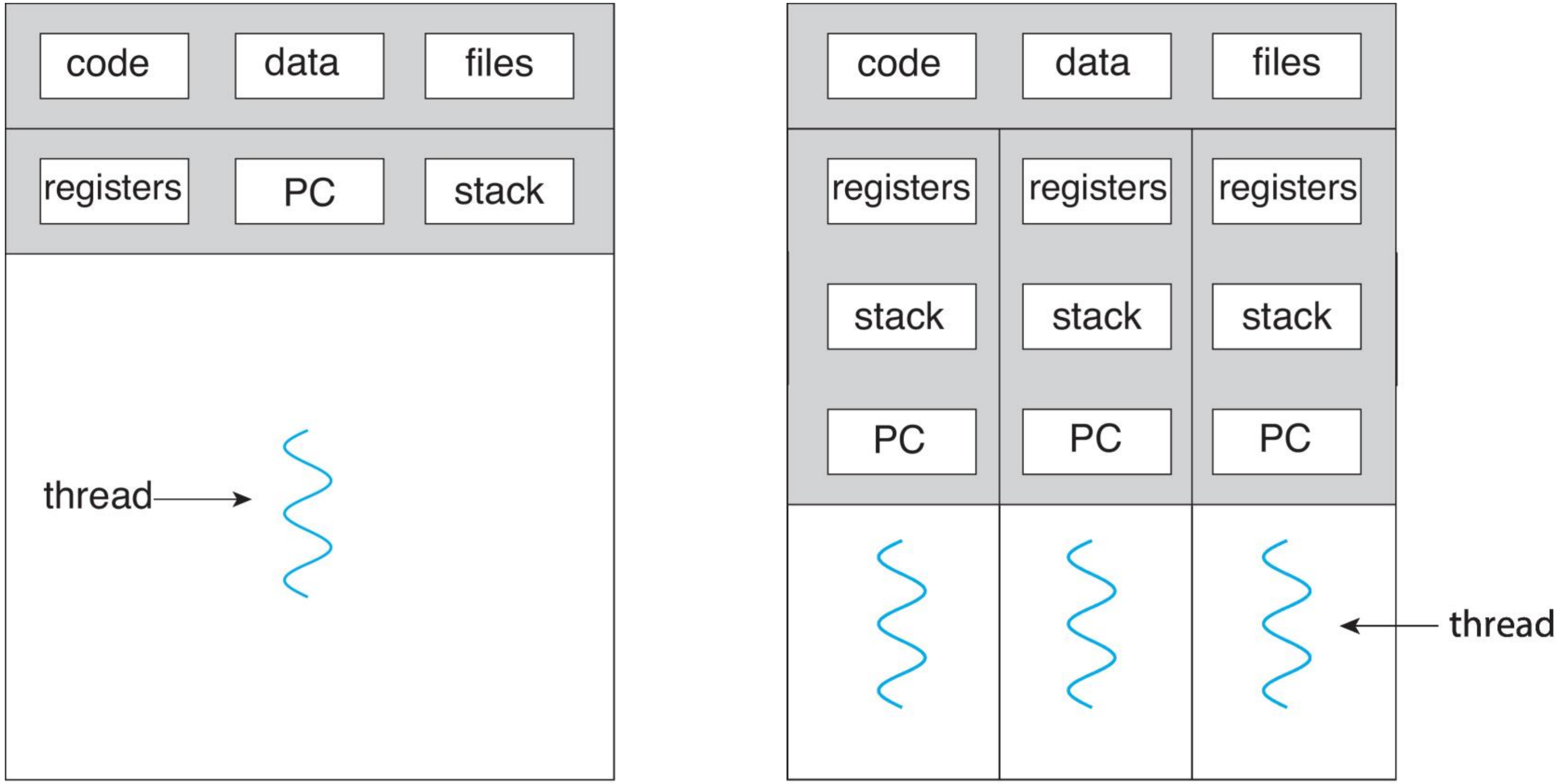
THREADS (CONT.)

- Each thread can interact with a specific part of the system, such as disks, network I/O, or users.
- Threads are scheduled for execution because some threads can wait for some event to happen or wait for some work to finish from another thread.
- Threads includes:
  - Thread ID (thread ID)
  - Program counter (PC)
  - Register set
  - Stacks
- Threads in a process share code, data, and other system resources such as open files and signals.

THREADS (CONT.)

- Most software applications that run on modern computers are multithreaded
  - A web browser might have one thread display images or text while another thread retrieves data from the network, for example.
  - A word processor may have a thread for displaying graphics, another thread for responding to keystrokes from the user, and a third thread for performing spelling and grammar checking in the background
- Benefits:
  - Responsiveness
  - Resource Sharing
  - Economy
  - Scalability

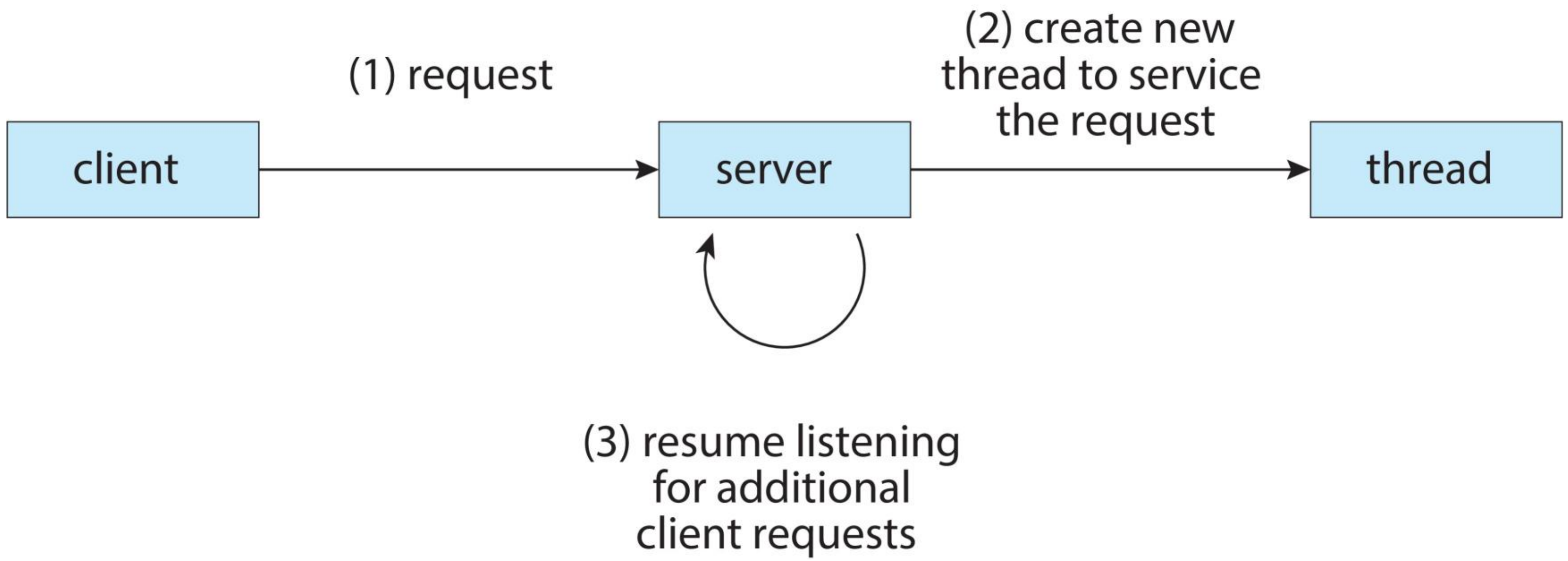
SINGLE AND MULTITHREADED PROCESSES



single-threaded process

multithreaded process

MULTITHREADED SERVER ARCHITECTURE

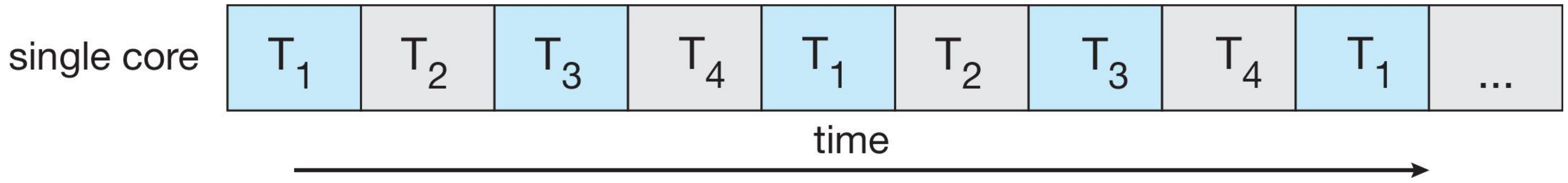


MULTICORE PROGRAMMING

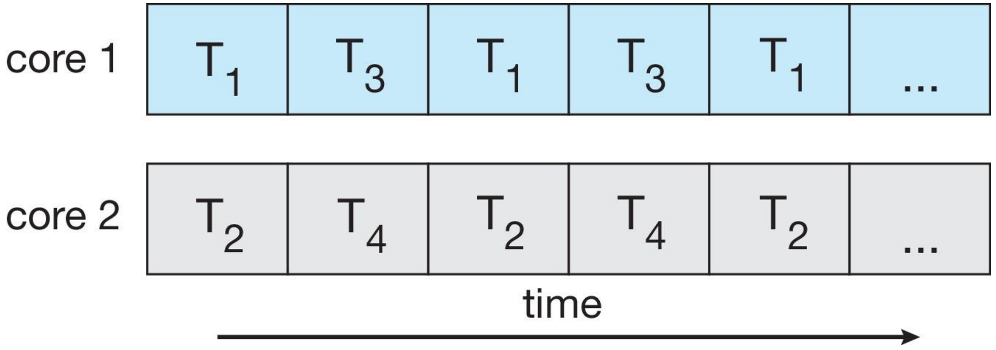
- **Multicore** or **multiprocessor** systems putting pressure on programmers, challenges include:
  - **Dividing activities**
  - **Balance**
  - **Data splitting**
  - **Data dependency**
  - **Testing and debugging**
- ***Parallelism*** implies a system can perform more than one task simultaneously
- ***Concurrency*** supports more than one task making progress
  - Single processor / core, scheduler providing concurrency

CONCURRENCY VS. PARALLELISM

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



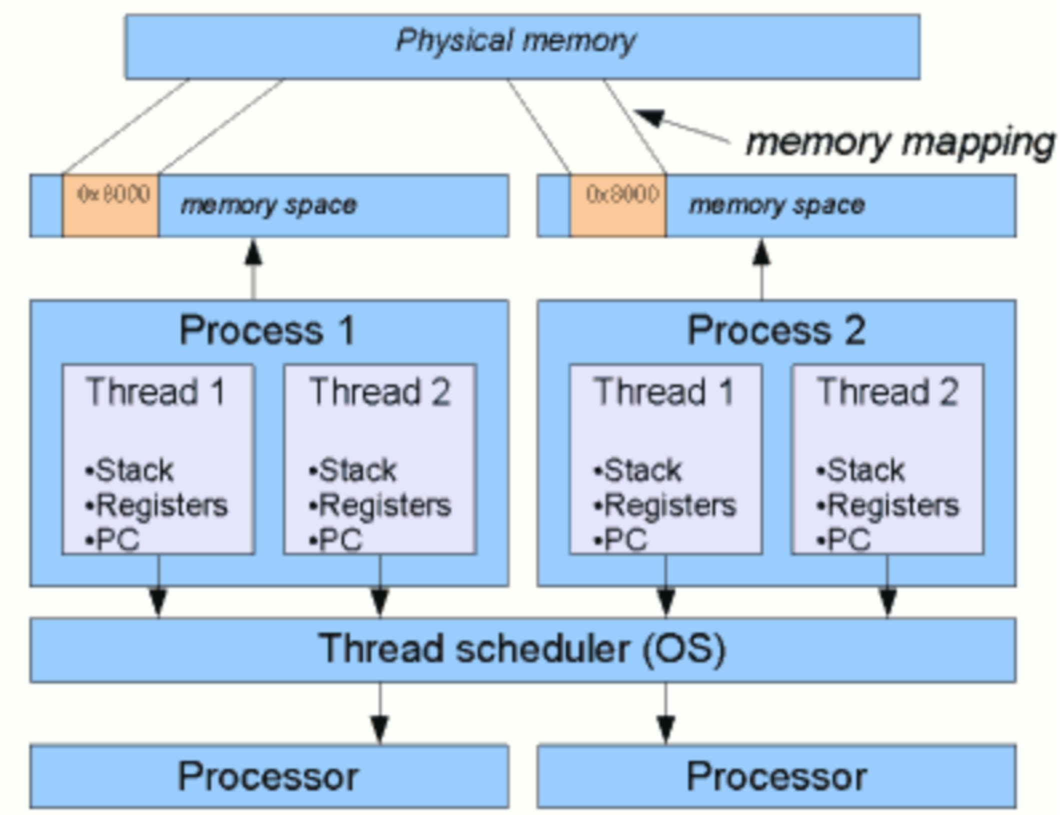
- **Parallelism on a multi-core system:**



PROCESS VS THREAD

Process	Thread
Process is considered heavy weight	Thread is considered light weight
Unit of Resource Allocation and of protection	Unit of CPU utilization
Process creation is very costly in terms of resources	Thread creation is very economical
Program executing as process are relatively slow	Programs executing using thread are comparatively faster
Process cannot access the memory area belonging to another process	Thread can access the memory area belonging to another thread within the same process
Process switching is time consuming	Thread switching is faster
One Process can contain several threads	One thread can belong to exactly one process

THREADS (CONT.)



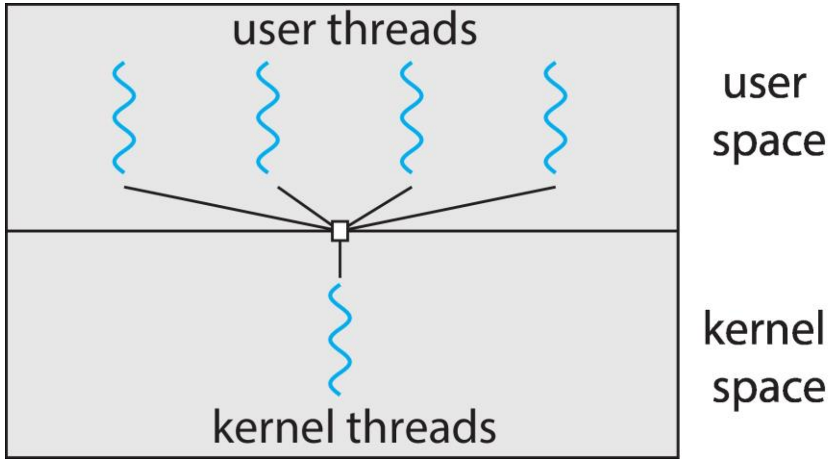
USER THREADS AND KERNEL THREADS

- **User threads** - management done by user-level threads library
- Three primary thread libraries:
  - POSIX **Pthreads**
  - Windows threads
  - Java threads
- **Kernel threads** - Supported by the Kernel
- Examples – virtually all general purpose operating systems, including:
  - Windows
  - Solaris
  - Linux
  - Tru64 UNIX
  - Mac OS X

MULTITHREADING MODELS

Many-to-one model

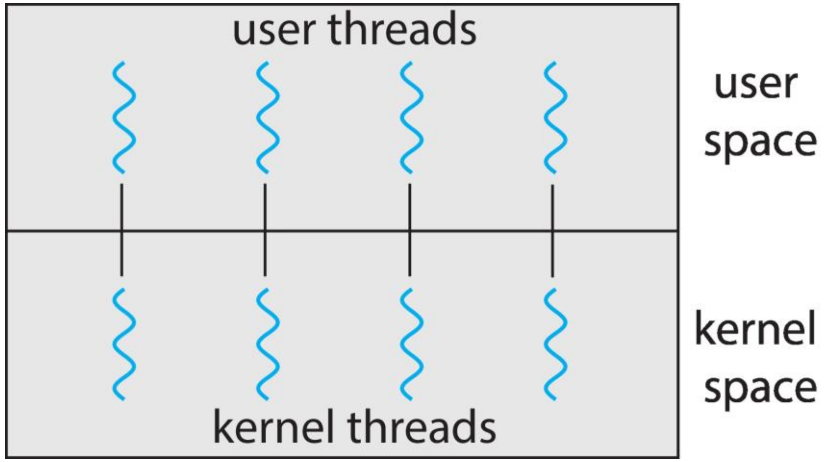
- Many user-level threads mapped to single kernel thread
- One thread blocking causes all to block
- Multiple threads may not run in parallel on muticore system because only one may be in kernel at a time
- Few systems currently use this model
- Examples:
  - **Solaris Green Threads**
  - **GNU Portable Threads**



MULTITHREADING MODELS

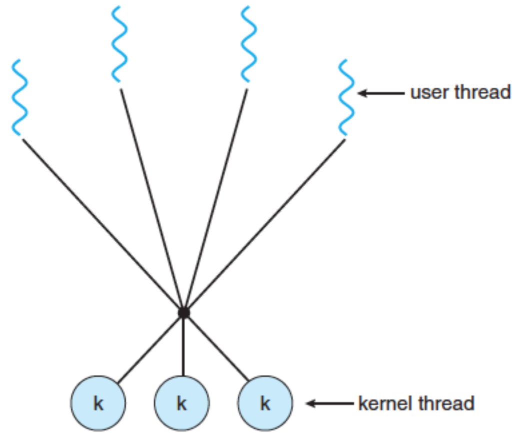
One-to-one model.

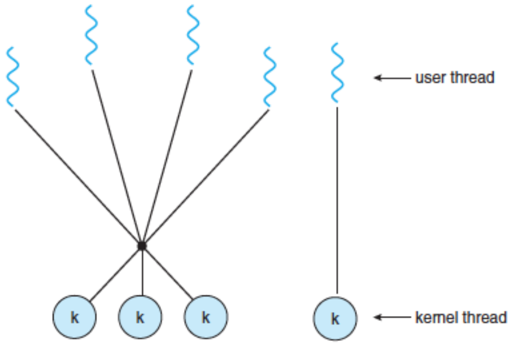
- Each user-level thread maps to kernel thread
- Creating a user-level thread creates a kernel thread
- More concurrency than many-to-one
- Number of threads per process sometimes restricted due to overhead
- Examples
  - Windows
  - Linux



MULTITHREADING MODELS

- Many-to-many model.





THREAD LIBRARIES

- A thread library provides the programmer with an API for creating and managing threads.
- Three main thread libraries are in use today:
  - POSIX Pthreads
  - Windows
  - Java.

THREAD LIBRARIES

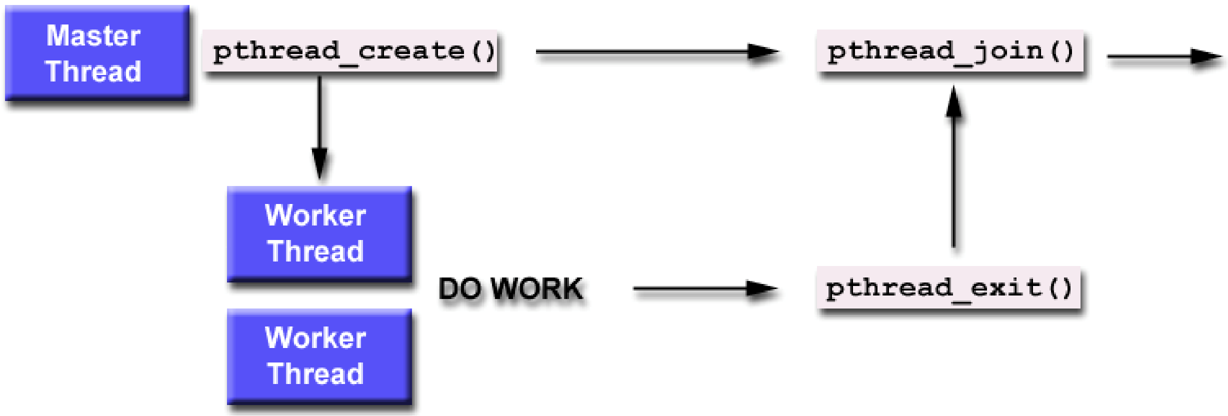
- For example, a multithreaded program that performs the summation of a non-negative integer in a separate thread:

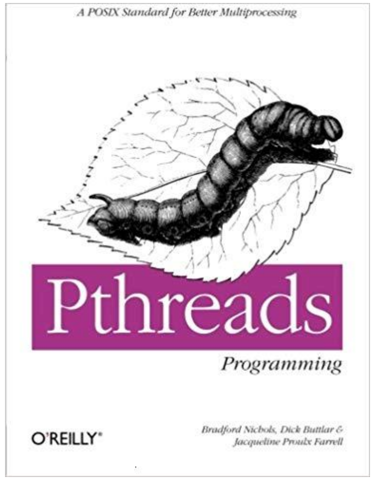
$$sum = \sum_{i=0}^{N} i$$

- Two strategies for creating multiple threads:
  - Asynchronous threading
  - Synchronous threading.

PTHREADS

- May be provided either as user-level or kernel-level
- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
- ***Specification***, not ***implementation***
- API specifies behavior of the thread library, implementation is up to development of the library
- Common in UNIX operating systems (Linux & Mac OS X)





PTHREADS EXAMPLE

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

#include <stdlib.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 */

  /* set the default attributes of the thread */
  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);
}
```

```
#include <pthread.h>
#include <stdio.h>
int sum=0; /* this data is shared by the thread(s) */
void* runner(void *param); /* threads call this function */
int amout1=5; amout2=8;
int main(int argc, char *argv[]) {
	pthread_t tid1; /* the thread identifier */
	pthread_t tid2; /* the thread identifier */
	pthread_attr_t attr; /* set of thread attributes */
	pthread_attr_init(&attr); /* get the default attributes */
	pthread_create(&tid1,&attr,runner,&amout1); /* create the thread */
	pthread_create(&tid2,&attr,runner,&amout2); /* create the thread */
	pthread_join(tid1,NULL); /* wait for the thread to exit */
	pthread_join(tid2,NULL); /* wait for the thread to exit */
	printf("sum = %d\n",sum);
}
void* runner(void* arg) { /* Thread will begin control in this function */
	int *param_ptr=(int *) arg;
	int param =*param_ptr;
	for (int i = 1; i <= param; i++)
	sum += i;
	pthread_exit(0);
}
```

WINDOWS THREADS

- The technique for creating threads using the Windows thread library is similar to the Pthreads technique in several ways.
- Threads are created in the Windows API using the CreateThread() function, and - just as in Pthreads a set of attributes for the thread is passed to this function
- Once the summation thread is created, the parent must wait for it to complete before outputting the value of Sum, as the value is set by the summation thread
- Windows API using the WaitForSingleObject() function to wait for the summation thread

```c
#include <Windows.h>
#include <process.h>
#include <stdio.h>
int sum1=0,sum2=0;
int i;
void mythread1(void* data)
{
        printf("mythread1 ID %d \n", GetCurrentThreadId());
        for (i = 0; i < 10; i++)
                if (i%2==0){
                sum1=sum1+i;}
                printf("sum= %d \n",sum1);
}
void mythread2(void* data)
{
        for (i = 0; i < 10; i++)
        if (i%2!=0){
        sum2=sum2+i;}
    printf("sum= %d \n",sum2);
        printf("mythread2 ID %d \n", GetCurrentThreadId());
}
```

```c
int main(int argc, char* argv[])
{
   HANDLE myhandle1, myhandle2;
   myhandle1 = (HANDLE)_beginthread(&mythread1, 0, 0);
   myhandle2 = (HANDLE)_beginthread(&mythread2, 0, 0);
    WaitForSingleObject(myhandle1, INFINITE);// join
    WaitForSingleObject(myhandle2, INFINITE);// join

       return 0;
}
```

JAVA THREADS

- Threads are the fundamental model of program execution in a Java program, and the Java language and its API provide a rich set of features for the creation and management of thread
- All Java programs comprise at least a single thread of control—even a simple Java program consisting of only a main() method runs as a single thread in the JVM
- Java threads may be created by:
  - Extending Thread class
  - Implementing the Runnable interface

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

  - Standard practice is to implement Runnable interface

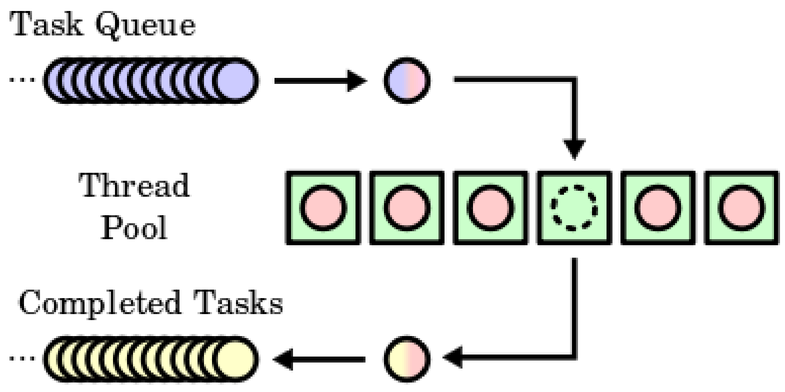
```java
public class Main {
public static void main(String[] args) {
       System.out.println("Start");
       Thread t1 = new Thread (new Runnable(){
          @Override
          public void run(){
             for (int i=0;i<10;i++){
                System.out.println("Thread 1>"+i);
             }
          }
       });
        Thread t2 = new Thread (new Runnable(){
          @Override
          public void run(){
             for (int i=0;i<10;i++){
                System.out.println("Thread 2>"+i);
             }
          }
       });
       t1.start();
       t2.start();
  System.out.println("Finish");
}}
```

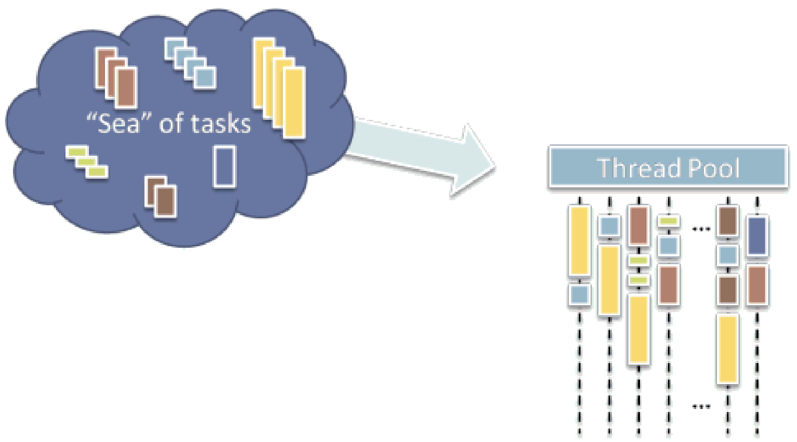
IMPLICIT THREADING

- **Implicit Threading:** strategy for designing multithreaded programs that can take advantage of multicore processors
  - Thread Pools
  - OpenMP (**Open** **M**ultiple **P**rocessing)
  - GDC (**G**rand **C**entral **D**ispatch)

THREAD POOLS

- The general idea behind a thread pool is to create a number of threads at process startup and place them into a pool, where they sit and wait for work





THREAD POOLS

- Thread pools offer these benefits:
  - Servicing a request with an **existing thread** is faster than waiting to create a thread.
  - A thread pool **limits the number of threads** that exist at any one point. This is particularly important on systems that cannot support a large number of concurrent threads.
  - Separating the task to be performed from the mechanics of creating the task allows us to use different **strategies for running the task**. For example, the task could be scheduled to execute after a time delay or to execute periodically

THEAD POOLS

- The Windows API provides several functions related to thread pools:
  - QueueUserWorkItem(&poolFunction, context, flags):
    - **&poolFunction**: A pointer to the defined callback function
    - **Context**: A single parameter value to be passed to the thread function.
    - The **flags** that control execution. This parameter can be one or more of the following values.

OPENMP

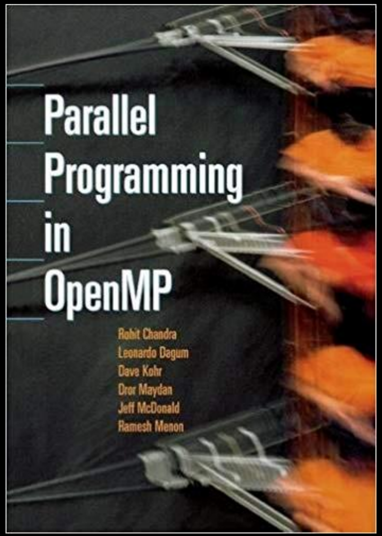
- OpenMP is a set of **compiler directives** as well as an API for programs written in C, C++, or FORTRAN that provides support for parallel programming in shared-memory environments
- OpenMP identifies **parallel regions** as blocks of code that may run in parallel
- Identifies **parallel regions** – blocks of code that can run in parallel

```
#pragma omp parallel
```

Create as many threads as there are cores

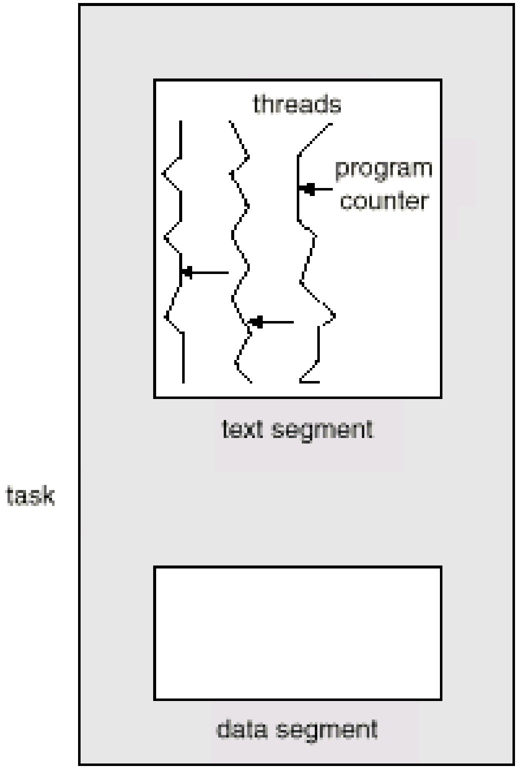
```
#include <omp.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
   /* sequential code */

   #pragma omp parallel
   {
      printf("I am a parallel region.");
   }

   /* sequential code */

   return 0;
}
```

```c
#include <omp.h>
#include <stdio.h>
int main(int argc, char *argv[]){
        /* sequential code */
        #pragma omp parallel
        {
                printf("I am a parallel region.");
        }
        /* sequential code */
        return 0;
}
```




```c
#pragma omp parallel for
for (i = 0; i < N; i++) {
    c[i] = a[i] + b[i];
}
```

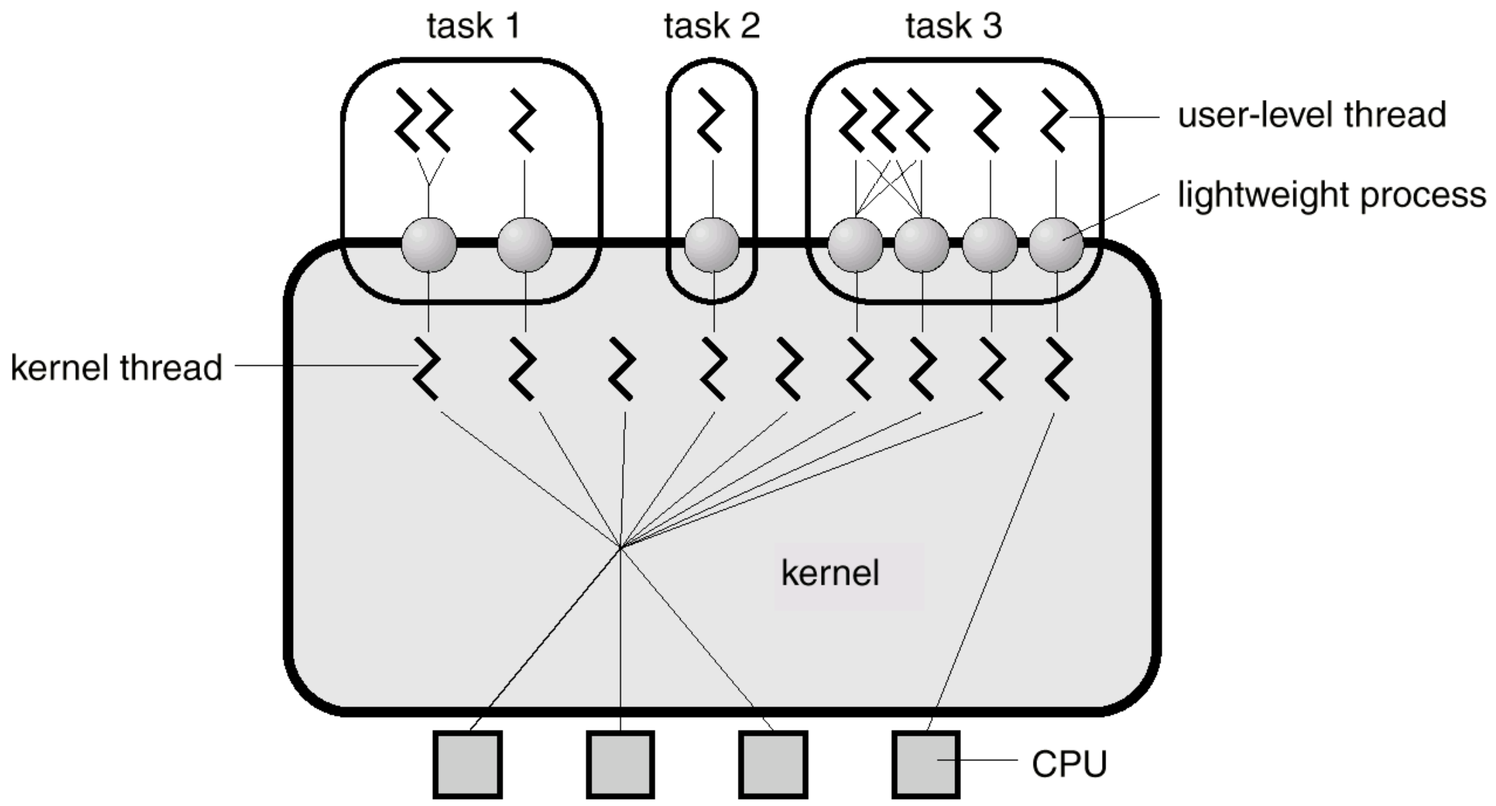
MULTIPLE THREADS WITHIN A TASK



THREADS SUPPORT IN SOLARIS 2

- Solaris 2 is a version of UNIX with support for threads at the kernel and user levels, symmetric multiprocessing, and real-time scheduling.
- LWP – intermediate level between user-level threads and kernel-level threads.
- Resource needs of thread types:
  - Kernel thread: small data structure and a stack; thread switching does not require changing memory access information – relatively fast.
  - LWP: PCB with register data, accounting and memory information,; switching between LWPs is relatively slow.
  - User-level thread: only ned stack and program counter; no kernel involvement means fast switching. Kernel only sees the LWPs that support user-level threads.

SOLARIS 2 THREADS

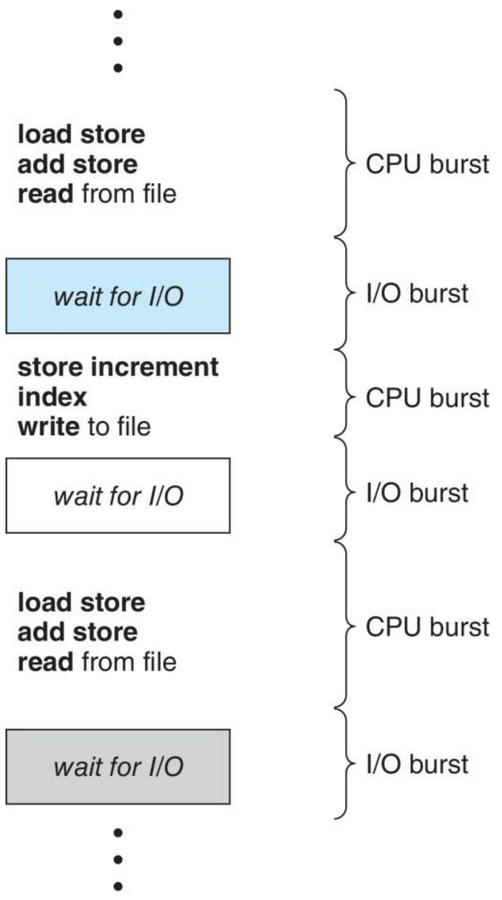


OUTLINE

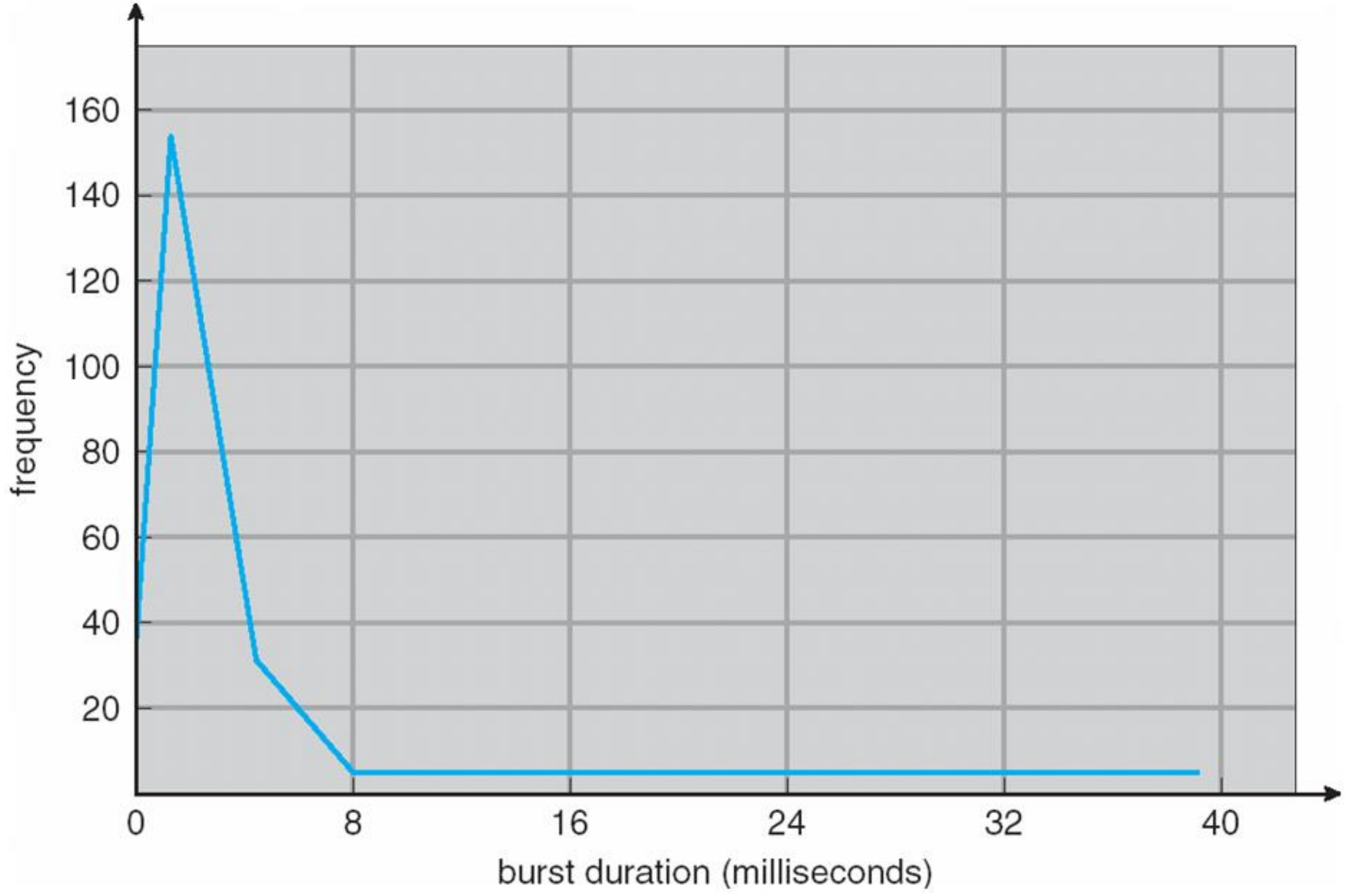
- Overview
- Interprocess Communication
- Thread
- Scheduling Algorithms
- Process Synchonization
- Deadlocks

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



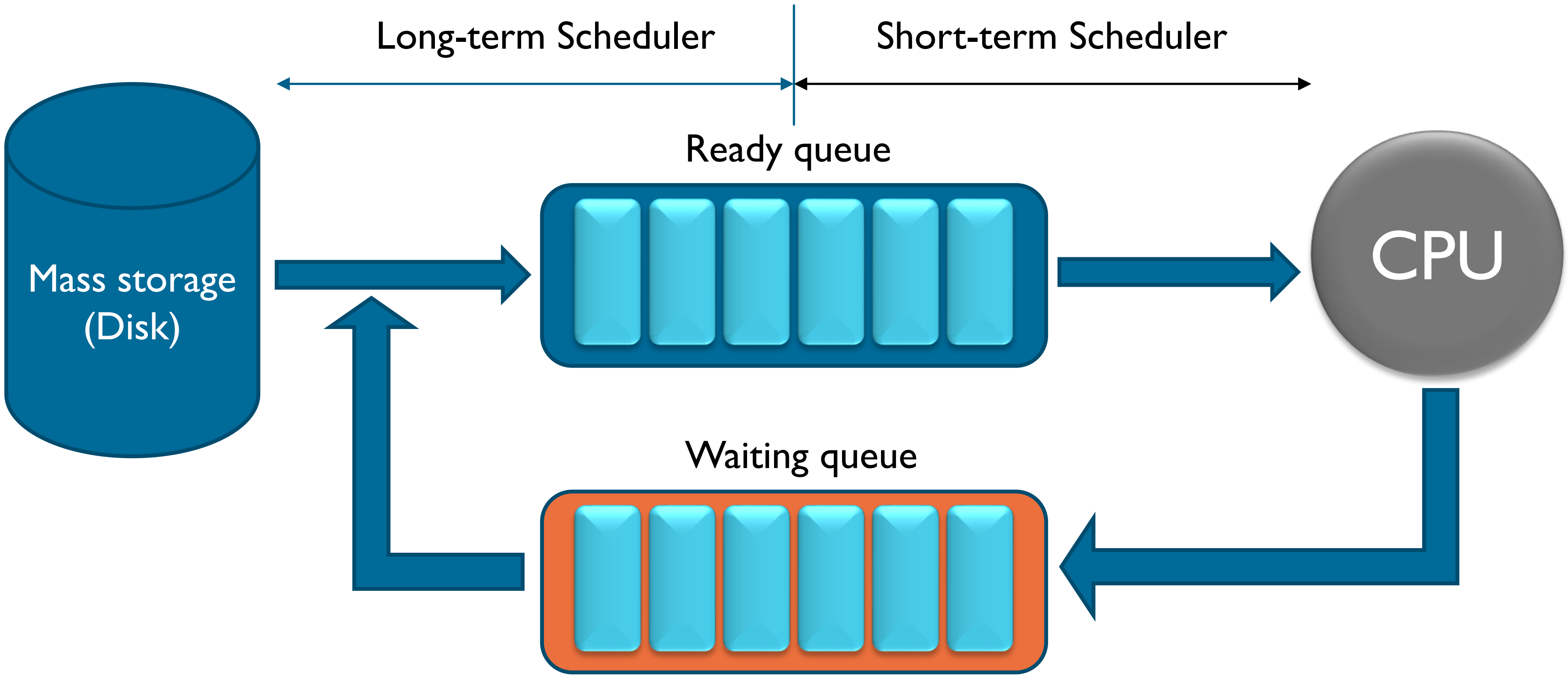
HISTOGRAM OF CPU-BURST TIMES



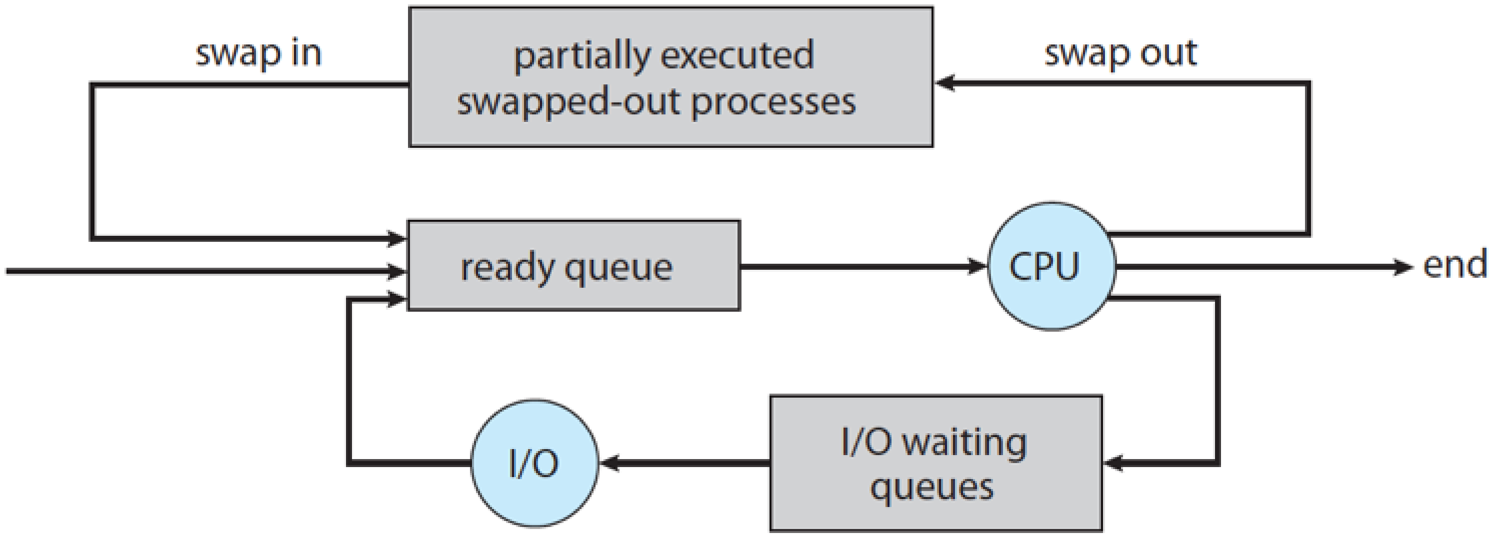
SCHEDULERS

- Long-term scheduler (or job scheduler) – selects which processes should be brought into the ready queue $\Rightarrow$ controls the degree of multiprogramming $\Rightarrow$ (may be slow)
- Medium-term: Select which process should be (swap in) and swap out
- Short-term scheduler (or CPU scheduler) – selects which process should be executed next and allocates CPU. $\Rightarrow$ (must be fast).

LONG-TERM SCHEDULER VS SHORT-TERM SCHEDULER



ADDITION OF MEDIUM TERM SCHEDULING

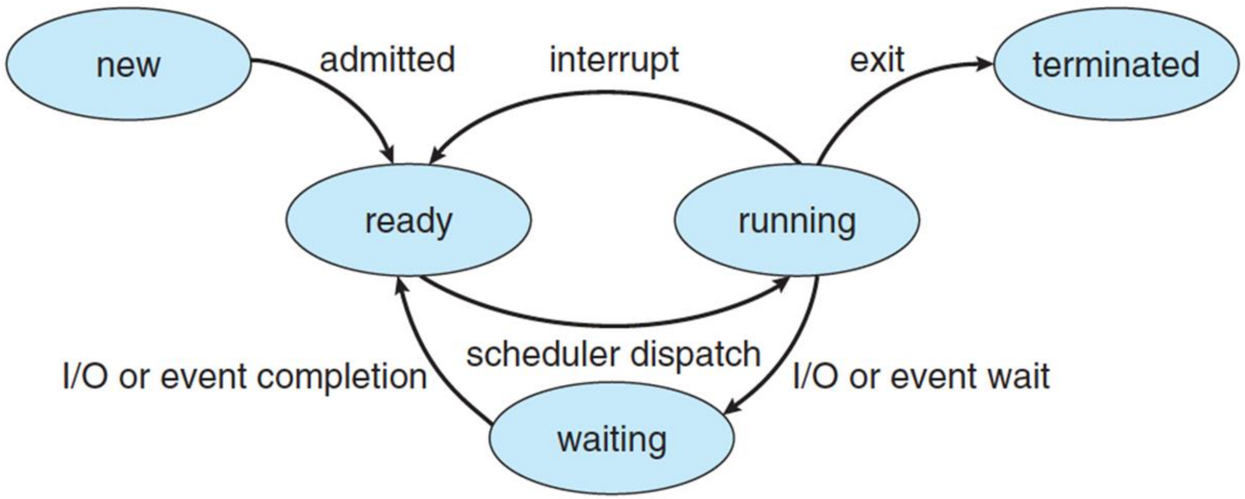


SCHEDULING ALGORITHMS

- Whenever the CPU becomes idle, the operating system must select one of the processes in the ready queue to be executed.
- The selection process is carried out by the **short-term scheduler**, or **CPU scheduler**

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
- CPU scheduling decisions may take place when a process:
  1. Switches from running to waiting state
  2. Switches from running to ready state
  3. Switches from waiting to ready
  4. Terminates
  5. New process to ready sate
- Scheduling under 1 and 4 is **nonpreemptive**
- All other scheduling is **preemptive**
  - Consider access to shared data
  - Consider preemption while in kernel mode
  - Consider interrupts occurring during crucial OS activities

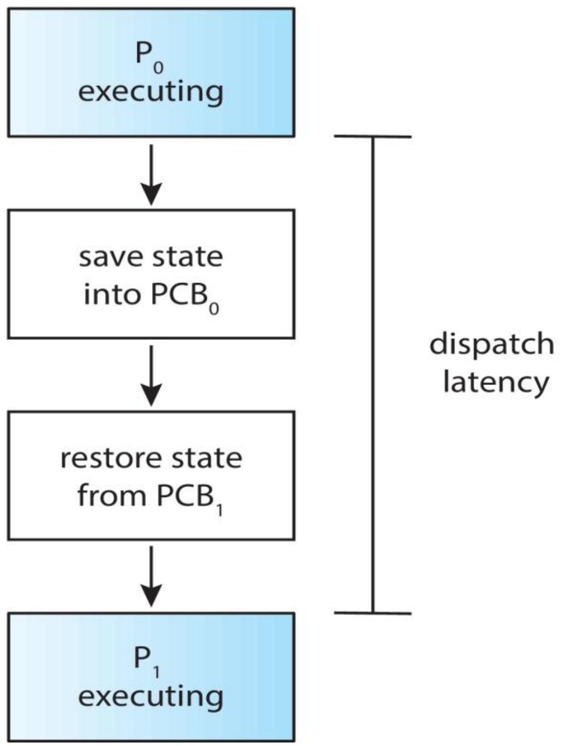


PREEMPTIVE VS NON-PREEMPTIVE SCHEDULING

- Non-preemptive
  - The running process keeps the CPU until it voluntarily gives up the CPU
    - process exits
    - switches to blocked state
- Preemptive:
  - The running process can be interrupted and must release the CPU (can be forced to give up CPU)

DISPATCHER

- Dispatcher module gives control of the CPU to the process selected by the CPU 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



SCHEDULING CRITERIA

- **CPU utilization:**In a real system, it should range from 40 percent (for a lightly loaded system) to 90 percent (for a heavily loaded system).
- **Throughput**: the number of processes that are completed per time unit
- **Turnaround time**:The interval from the time of submission of a process to the time of completion. = waiting to get into memory + waiting in the ready queue + executing on the CPU + doing I/O.
- **Waiting time**. is the sum of the periods spent waiting in the ready queue.
- **Response time**. the time from the submission of a request until the first response is produced

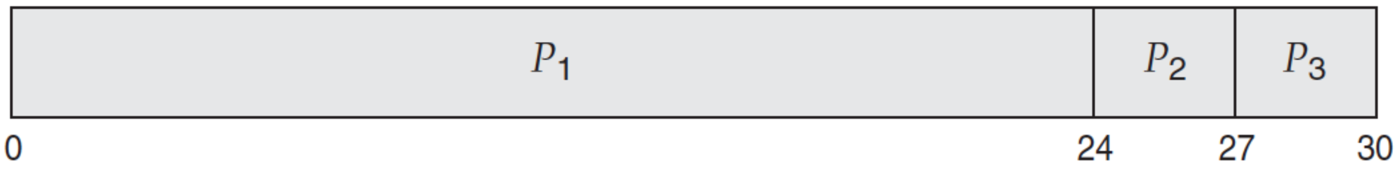
It is desirable to maximize CPU utilization and throughput and to minimize turnaround time, waiting time, and ***response time****.*

SCHEDULING ALGORITHMS: FIRST-COME, FIRST-SERVED

1. First-Come, First-Served 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

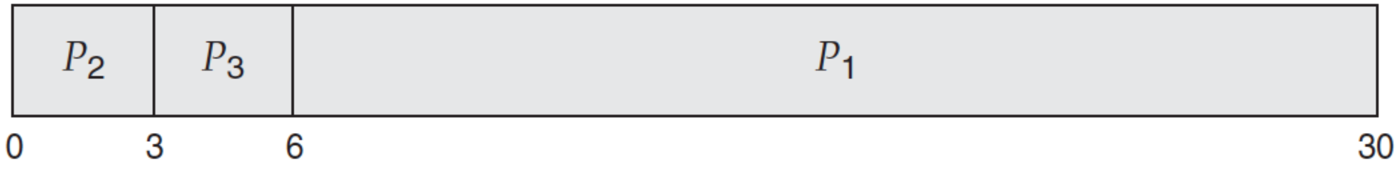
SCHEDULING ALGORITHMS: FIRST-COME, FIRST-SERVED(CONT)

1. First-Come, First-Served Scheduling (FCFS)
   - **Non-preemptive**
   - There is a convoy effect as all the other processes wait for the one big process to get off the CPU

SCHEDULING ALGORITHMS: FIRST-COME, FIRST-SERVED (CONT)

1. First-Come, First-Served Scheduling (FCFS)

- If the processes arrive in the order P2, P3, P1, however, the results will be as shown in the following Gantt chart:



- Waiting time for $P_1 = 6; P_2 = 0; P_3 = 3$
- Average waiting time: $(6 + 0 + 3)/3 = 3$

SCHEDULING ALGORITHMS: FIRST-COME, FIRST-SERVED(CONT)

Process	Arrival time	Bust time
P1	0	11
P2	3	7
P3	5	4
P4	5	2
P5	2	8

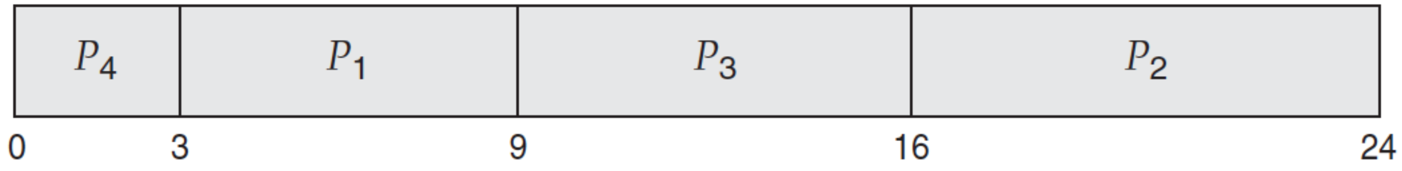
p1	p5	p2	p3	p4
0	11	19	26	30 32

SCHEDULING ALGORITHMS: SHORTEST-JOB-FIRST (SJF)

- 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
  - Estimate

SCHEDULING ALGORITHMS: SHORTEST-JOB-FIRST(CONT)

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3



- Waiting time = (3+16+9+0)/4 = 7

SHORTEST-JOB-FIRST(NON PREEMPTIVE) WITH ARRIVAL TIME

Process	Arrival time	Bust time
P1	0	11
P2	3	7
P3	5	4
P4	5	2
P5	2	8

- Average waiting time?
- Average turnaround time?

SHORTEST-JOB-FIRST(NON PREEMPTIVE) WITH ARRIVAL TIME

Process	Arrival time	Bust time
P1	0	11
P2	3	7
P3	5	4
P4	5	2
P5	2	8

- Gantt chart:

P1				
0 11				

T_{p1}	T_{p2}	T_{p3}	T_{p4}	T_{p5}
0	17-3=14	13-5=8	11-5=6	24 -2 =22

- Average waiting time:

$$\bar{T} = \frac{T_{p1}+T_{p2}+T_{p3}+Tp_4+Tp_5}{5} = \frac{0+14+8+6+22}{5} = \frac{50}{5} = 10$$

- Average turnaround time

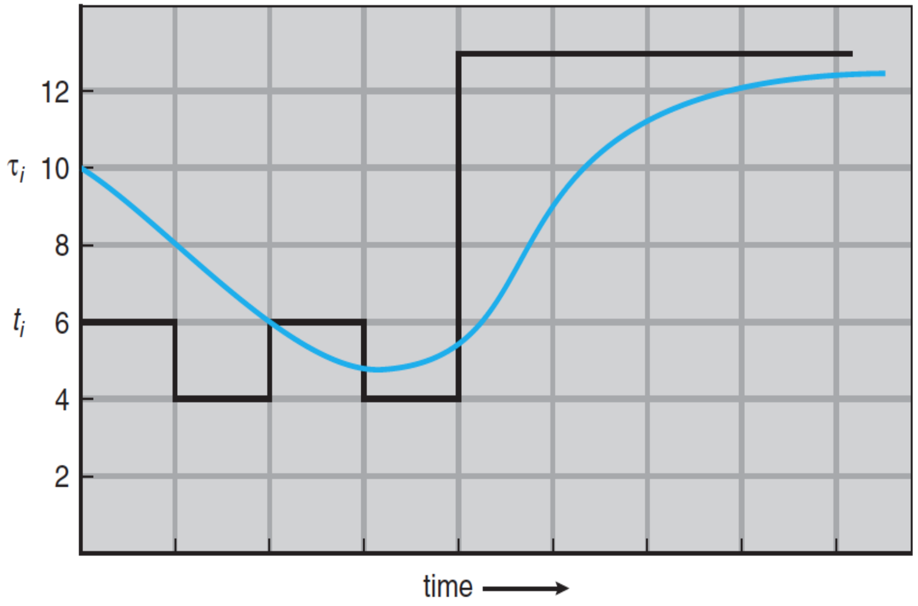
$$\overline{T} + \overline{T_{bust}} = 10 + \frac{11+7+4+2+8}{5} = 10 + \frac{32}{5} = 16.4$$

SCHEDULING ALGORITHMS: SHORTEST-JOB-FIRST(CONT)

- **Non-preemptive** scheduling
- The real difficulty with the SJF algorithm is knowing the length of the next CPU request.
- For long-term (job) scheduling in a batch system, we can use the process time limit that a user specifies when he submits the job
- SJF scheduling is used frequently in long-term scheduling.
- One approach to this problem is to try to approximate SJF scheduling. We may not know the length of the next CPU burst, but we may be able to **predict** its value.

SCHEDULING ALGORITHMS: SHORTEST-JOB-FIRST(CONT)

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



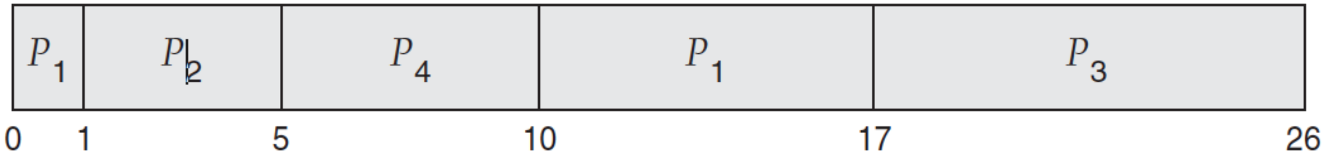
CPU burst (t_i)		6	4	6	4	13	13	13	…
"guess" (τ_i)	10	8	6	6	5	9	11	12	…

EXAMPLE OF SHORTEST-REMAINING-TIME-FIRST (SJF PREEMPTIVE)

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



Waiting time: [(10 − 1) + (1 − 1) + (17 − 2) + (5 − 3)]/4 = 26/4 = 6.5

SHORTEST-REMAINING-TIME-FIRST (SJF PREEMPTIVE

Process	Arrival time	Bust time
P1	0	11
P2	3	7
P3	5	4
P4	5	2
P5	2	8

- Average waiting time?
- Average turnaround time?

SHORTEST-REMAINING-TIME-FIRST (SJF PREEMPTIVE)

Process	Arrival time	Bust time
P1	0	11
P2	3	7
P3	5	4
P4	5	2
P5	2	8

Remaining time

P1	P2	P3	P4	P5

ready queue

Running queue

- Gantt

P1							
0 2							

P1	P5	P4	P3	P5	P2	P1
0	2	5	7	11	16	23 32

T_{p1}	T_{p2}	T_{p3}	T_{p4}	T_{p5}
0+(23-2)=21	16-3=13	7-5=2	5-5=0	2+(11-5)-2=6

The average waiting time: 8.4 milliseconds

The Average turnaround time: 8.4 + 6.4= 14.8 milliseconds.

SHORTEST-REMAINING-TIME-FIRST (SJF PREEMPTIVE

Process	Arrival time	Bust time
P1	0	11
P2	3	7
P3	5	4
P4	5	2
P5	2	8

- Average waiting time?
- Average turnaround time?

SCHEDULING ALGORITHMS: ROUND-ROBIN

- **Preemption** scheduling
- The round-robin (RR) scheduling algorithm is designed especially for timesharing systems.
- Each process gets a small unit of time, called a time quantum (q) or time slice, is defined. A time quantum is generally from 10 to 100 milliseconds in length.
- After this time has elapsed, the process is preempted and added to the end of the ready queue.
- The ready queue is treated as a circular queue.
- Timer interrupts every quantum to schedule next process

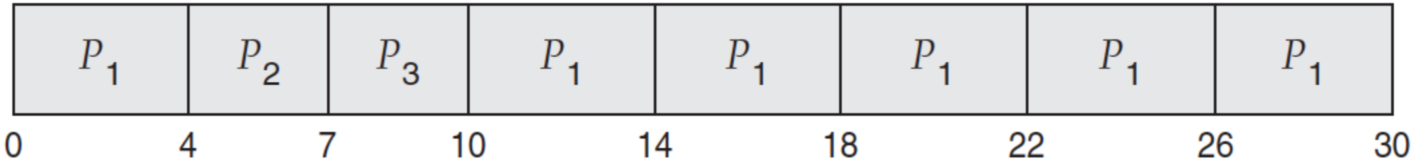
- Performance
  - q large $\Rightarrow$ FIFO
  - q small $\Rightarrow$ q must be large with respect to context switch, otherwise overhead is too high

SCHEDULING ALGORITHMS: ROUND-ROBIN(CONT)

- In the RR scheduling algorithm, no process is allocated the CPU for more than 1 time quantum in a row (unless it is the only runnable process)
- 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
  - if the time quantum is extremely small, the RR approach can result in a large number of context switches

SCHEDULING ALGORITHMS: ROUND-ROBIN(CONT)

Process	Burst Time
P_1	24
P_2	3
P_3	3



- q should be large compared to context switch time
- q usually 10ms to 100ms, context switch < 10 microseconds

ROUND ROBIN WITH ARRIVAL TIME

Process	Arrival time	Bust time
P1	0	11
P2	3	7
P3	5	4
P4	5	2
P5	2	8

Play

Remaining time

P1	P2	P3	P4	P5

P4 P3

ready queue

P2 P5 P1

Running Process

- Gantt chart
- Q=3

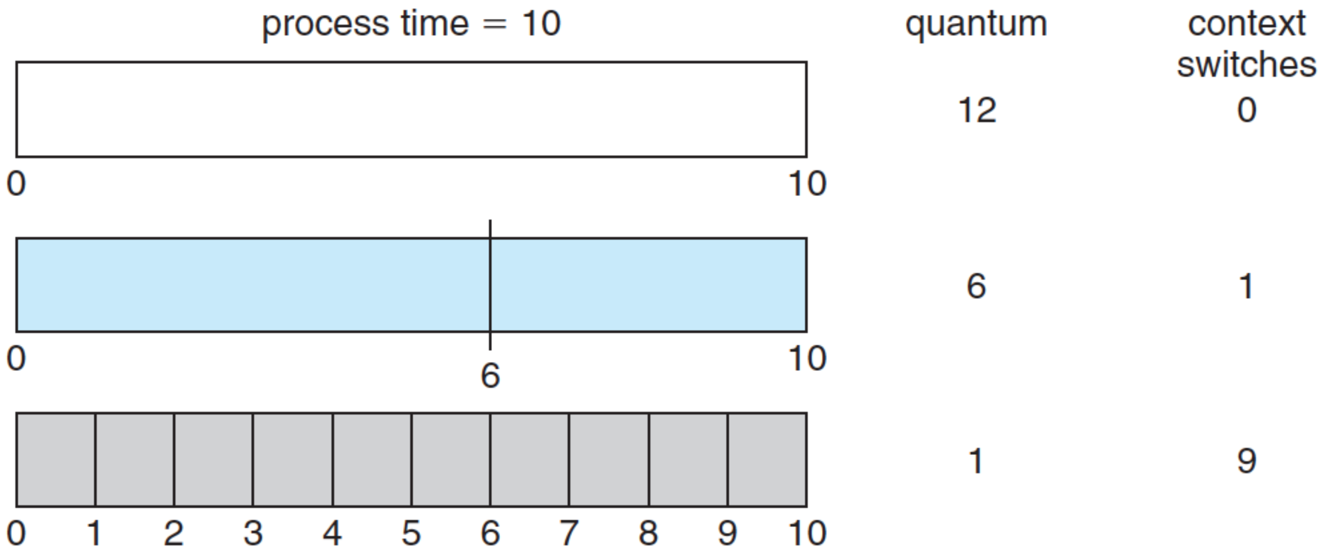
P1													
0	3												

T_{p1}	T_{p2}	T_{p3}	T_{p4}	T_{p5}
$T_{cp1}=0+(9-3)+(23-12)+(30-26)=21$	$T_{cp2}=6+(20-9)+(29-23)-3=20$	$T_{cp3}=12+(26-15)-5=18$	$T_{cp4}=15-5=10$	$T_{cp5}=3+(17-6)+(27-20)-2=19$

The average waiting time: 17.6 milliseconds

The Average turnaround time: 17.6 + 32/5= 24 milliseconds.

SCHEDULING ALGORITHMS: ROUND-ROBIN(CONT)



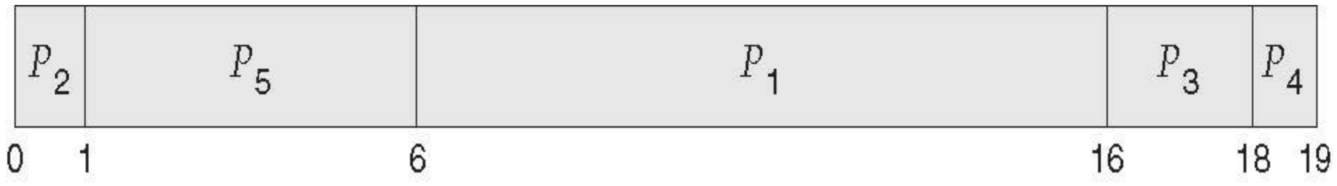
SCHEDULING ALGORITHMS: PRIORITY

- Priorities are generally indicated by some fixed range of numbers, such as 0 to 7 or 0 to 4,095.
- Some systems use low numbers to represent low priority; others use low numbers for high priority
- In this text, we assume that low numbers represent high priority
  - Preemptive
  - Nonpreemptive

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

SCHEDULING ALGORITHMS: PRIORITY(CONT)

- Priorities can be defined either internally or externally.
- Internally defined priorities use some measurable quantity or quantities to compute the priority of a process.
  - time limits
  - memory requirements
  - the number of open files
  - ratio of average I/O burst to average CPU burst
- External priorities are set by criteria outside the operating system
  - the importance of the process
  - the type and amount of funds being paid for computer use
  - the department sponsoring the work
  - often political, factors

SCHEDULING ALGORITHMS: PRIORITY(CONT)

- A major problem with priority scheduling algorithms is **indefinite blocking**, or **starvation**.
- Asolution to the problem of indefinite blockage of low-priority processes is **aging**
- For example, if priorities range from 127 (low) to 0 (high), we could increase the priority of a waiting process by 1 every 15 minutes.
- Priority scheduling can be either **preemptive (4a)** or **nonpreemptive**. (4b)

PRIORITY WITH ARRIVAL TIME PREEMPTIVE

Process	Arrival time	Bust time	Priority
P1	0	11	2
P2	3	7	3
P3	5	4	1
P4	5	2	3
P5	2	8	2

The average waiting time is 12.8 milliseconds
The Average turnaround time is 19.2 milliseconds.

PRIORITY WITH ARRIVAL TIME PREEMPTIVE

Process	Arrival time	Bust time	Priority
P1	0	11	2
P2	3	7	3
P3	5	4	1
P4	5	2	3
P5	2	8	2

ready queue

Gantt chart

P1	P3	P5	P1	P2	P4
0	5	9	17	23	30 32

Tp1	Tp2	Tp3	Tp4	Tp5
0+(17-2)=12	23-3=20	5-5=0	30-5=25	9-2=7

The average waiting time is 12.8 milliseconds

The average turnaround time: is 12.8 + 32/5= 19.2 milliseconds.

PRIORITY WITH ARRIVAL TIME – NON PREEMPTIVE

Process	Arrival time	Bust time	Priority
P1	0	11	2
P2	3	7	3
P3	5	4	1
P4	5	2	3
P5	2	8	2

PRIORITY WITH ARRIVAL TIME – NON PREEMPTIVE

Process	Arrival time	Bust time	Priority
P1	0	11	2
P2	3	7	3
P3	5	4	1
P4	5	2	3
P5	2	8	2

- Gantt chart

ready queue

P1	P3	P5	P2	P4	
0	11	15	23	30	32

Tp1	Tp2	Tp3	Tp4	Tp5
0-0=0	23-3=20	11-5=6	30-5=25	15-2=13

PRIORITY SCHEDULING W/ ROUND-ROBIN

Process	Burst Time	Priority
P_1	4	3
P_2	5	2
P_3	8	2
P_4	7	1
P_5	3	3

- Run the process with the highest priority. Processes with the same priority run round-robin
- Gantt Chart with time quantum = 2

P_4	P_2	P_3	P_2	P_3	P_2	P_3	P_1	P_5	P_1	P_5
0–7	7–9	9–11	11–13	13–15	15–16	16–20	20–22	22–24	24–26	26–27

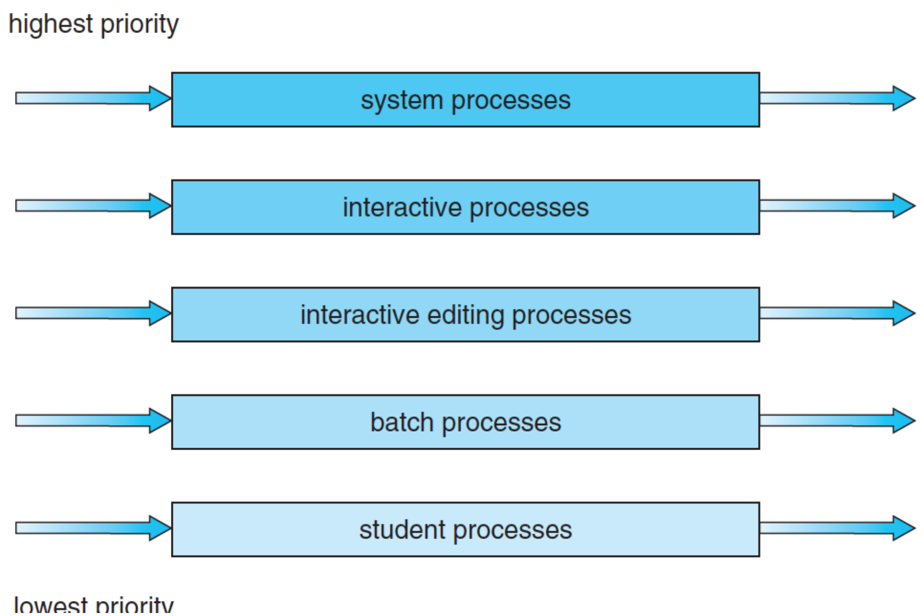
SCHEDULING ALGORITHMS: MULTILEVEL QUEUE

- 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

SCHEDULING ALGORITHMS (CONT)

Multilevel Queue Scheduling

- partitions the ready queue into several separate queues

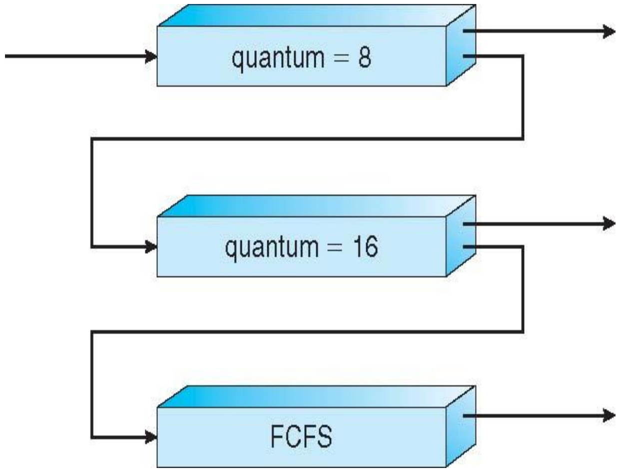


SCHEDULING ALGORITHMS: MULTILEVEL FEEDBACK QUEUE

- 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



SCHEDULING ALGORITHMS: LOTTERY SCHEDULING WORKS

- Scheduling works by assigning processes lottery tickets
- Whenever a scheduling decision has to be made, a lottery ticket is chosen at random.
- More tickets – higher probility of winning
- Solves starvation

OUTLINE

- Overview
- Interprocess Communication
- Thread
- Scheduling Algorithms
- Process Synchonization
- Deadlocks

PROCESS SYNCHRONIZATION

- Concurrent access to shared data may result in data inconsistency.
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes.
- Shared-memory solution to bounded-buffer problem allows at most $n - 1$ items in buffer at the same time. A solution, where all N buffers are used is not simple.
  - Suppose that we modify the producer-consumer code by adding a variable *counter*, initialized to 0 and incremented each time a new item is added to the buffer
- “Busy wating” solutions
- “Sleep and wakeup” solutions

BOUNDED-BUFFER

- Shared data **type** *item* = … ;

  var *buffer* **array** [0..*n*-1] **of** item;

  in, *out*: 0..*n*-1;

  counter: 0..*n*;

  in, *out*, *counter* := 0;

- Producer process

  repeat

  …

  produce an item in *nextp*

  …

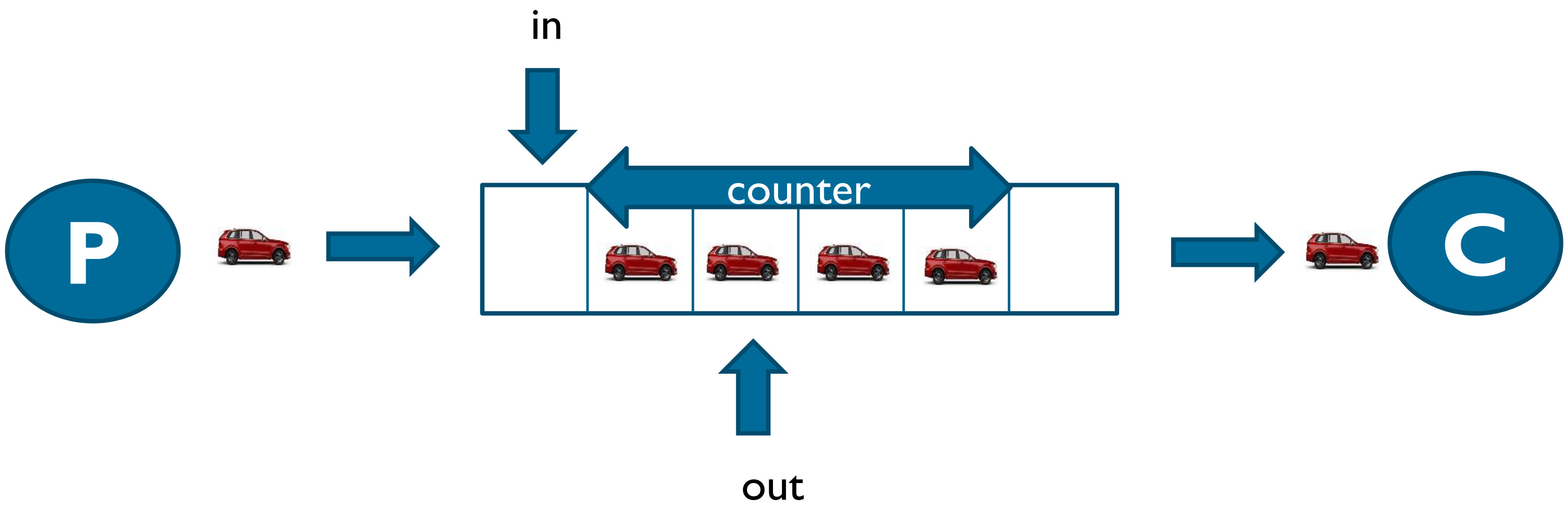
  while *counter* = n **do** no-op;

  buffer [*in*] := *nextp*;

  in := *in* + 1 **mod** *n*;

  counter := *counter* +1;

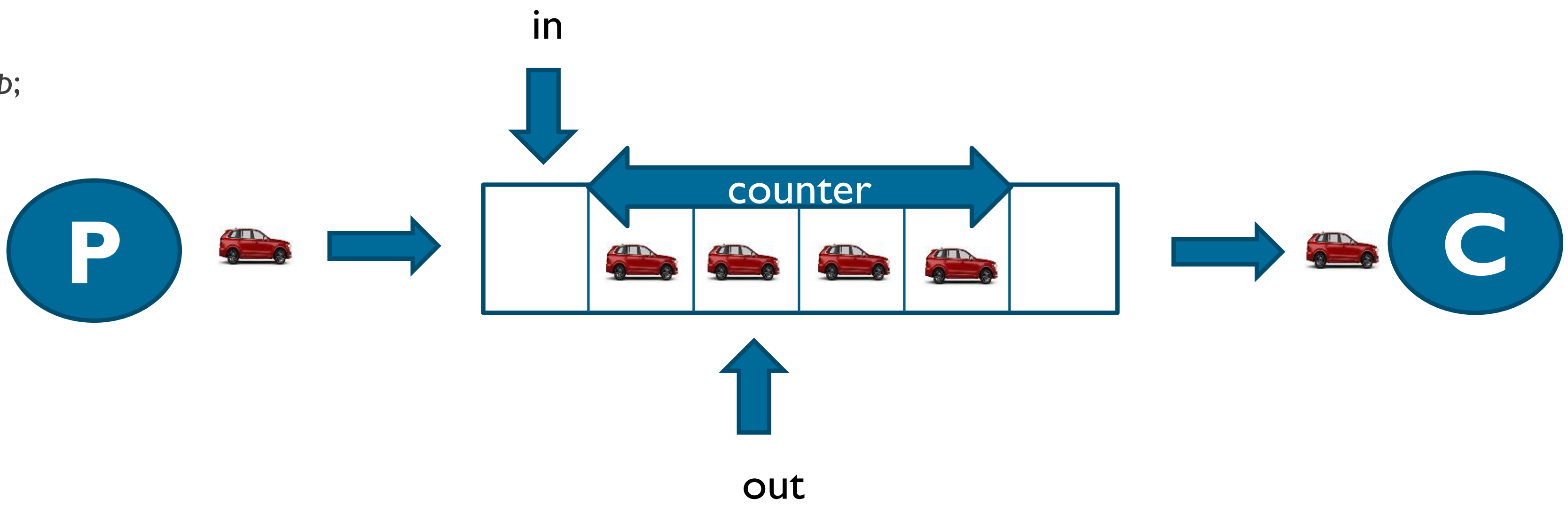
  until false;



BOUNDED-BUFFER (CONT.)

- Consumer process

repeat
    while *counter* = 0 **do** *no-op*;
    nextc := *buffer* [*out*];
    out := *out* + 1 **mod** *n*;
    counter := *counter* – 1;
      …
    consume the item in *nextc*
      …
until *false*;



- The statements:
  - *counter* := *counter* + 1;
  - *counter* := *counter* - 1;

must be executed *atomically*.

RACE CONDITION

- `counter++` could be implemented as

```
register1 = counter (load)
register1 = register1 + 1 (Inc)
counter = register1 (Store)
```

- `counter--` could be implemented as

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

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

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

THE CRITICAL-SECTION PROBLEM

- 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**
- Structure of process P_i

repeat

entry section

critical section

exit section

reminder section

until *false*;

CRITICAL SECTION

- General structure of process P_i

```
while (true) {
        entry section

            critical section

        exit section

            remainder section

}
```

SOLUTION TO CRITICAL-SECTION PROBLEM

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

"BUSY – WAITING" SOLUTION

While (unpermission) do nothing();

Critical section;

Give up permission with CS

- Synchronization Software
  - Lock
  - Peterson's Solution
- Synchronization Hardware
  - non-interruptible
  - Test&Set Instruction

INITIAL ATTEMPTS TO SOLVE PROBLEM

- Only 2 processes, P_0 and P_1
- General structure of process P_i (other process P_j)

repeat

entry section

critical section

exit section

reminder section

until *false*;

- Processes may share some common variables to synchronize their actions.

ALGORITHM 1

```
while(true){
    while lock ==1 do no-op;          /* waiting lock=0*/
        //lock =0
        lock=1; //set lock =1 to prevent other process (lock)
        critical section
        lock=0; // finish CS the release resource (unlock)
        reminder section
    }
```

ALGORITHM 2

- Process P_i

```
do {
      while turn ==j do no-op;          /* waiting turn #j*/
            critical section
      turn = j;
            reminder section
} while (1);
```

- Satisfies mutual exclusion, but not progress and bounded waiting because of strict alternation

ALGORITHM FOR PROCESS P_I (CONT.)

- Process P0:

while (true) {

while (turn != 0); //waiting for turn=0

critical section

turn = 1; //set turn =1 for P1

remainder section

}

- Process P1:

while (true) {

while (turn != 1); //waiting for turn=1

critical section

turn = 0; //set turn =0 for P0

remainder section

- }

- Satisfies mutual exclusion, but not progress and bounded waiting because of strict alternation

PETERSON'S SOLUTION

- 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 interesse[2]`
- The variable `turn` indicates whose turn it is to enter the critical section
- The `interess` array is used to indicate if a process is ready to enter the critical section.
  - `interesse[2] =` *`true`* implies that process $\mathbf{P_i}$ is ready!

PETERSON SOLUTIONS

- Combined shared variables of algorithms 1 and 2.
- Process P_i

  do {

  ```
  flag [i] := true;                /* Pi ready to enter its critical section */
  turn := j;                       ; /* Pi give away to Pj */
  while (flag [j] and turn == j) do no-op;

      critical section

  flag [i] := false;

      remainder section

  } while (1);
  ```

- Meets all three requirements; solves the critical-section problem for two processes.

PETERSON SOLUTIONS FOR PROCESS P_I (CONT.)

- Process P0:

```
While (true) {
    /*  0 wants in  */
    interesse[0] = true;
      /* 0 gives a chance to 1  */
     turn = 1;
     while (interesse[1] && turn == 1);//wait
    critical section   /*0 no longer wants in */
    interesse[0] = false;
      remainder section;
  }
```

- Process P1:

```
While (true) {
    /*  1 wants in  */
    interesse [1] = true;
      /* 1 gives a chance to 0  */
     turn = 0;
     while (interesse [0] && turn == 0); ;//wait
    critical section
    /*1 no longer wants in */
    interesse[1] = false;
      remainder section;
  }
```

CORRECTNESS OF PETERSON'S SOLUTION

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

       $\mathbf{P_i}$ enters CS only if:

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

MUTEX LOCKS

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

SOLUTION TO CS PROBLEM USING MUTEX LOCKS

```
while (true) {
    acquire lock

        critical section

    release lock

remainder section
}
```

SYNCHRONIZATION HARDWARE

- Allows the process to disable all interrupts before entering the critical section, and to recover interrupts when leaving the critical section.
- At that time, the clock interrupt also does not occur, so the system cannot pause the operation of the process that is processing to allocate CPU for another process, so that the current process can rest assured to operate on the critical segment. without fear of being disputed by any other process

SYNCHRONIZATION HARDWARE

- Test and modify the content of a word atomically.

  function *Test-and-Set* (**var** target: *boolean*): *boolean*;

  Begin

  Test-and-Set := *target*;
  target := *true*;

  End

MUTUAL EXCLUSION WITH TEST-AND-SET

- Shared data: **var** *lock*: *boolean* (*initially false*)
- Process P_i

```
while(true)
    while Test-and-Set (lock) do no-op;
        critical section
    lock := false;
        remainder section
```

“SLEEP & WAKE UP” SOLUTIONS

If (not have permission) Sleep();

Critical section;

Wakeup (somebody);

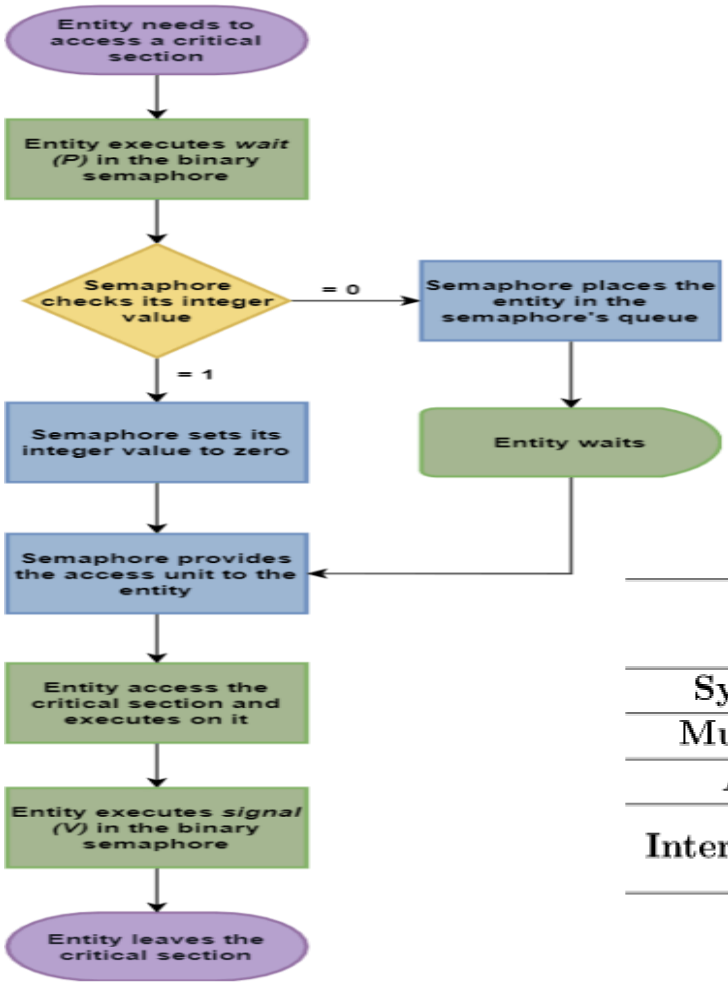
```
while(true){
    if(busy){
        ++blocked;
        sleep();}
    else busy =1;//set busy =1 to prevent
other
        critical section
    busy =0;// set busy=0
    if(blocked)//check any waited process?
    {     wakeup(process);// wakup other
        blocked --;}
        remainder section
            }
```

SEMAPHORE AS GENERAL SYNCHRONIZATION TOOL

- **Counting** semaphore – integer value can range over an unrestricted domain
- **Binary** semaphore – integer value can range only between 0 and 1; can be simpler to implement
  - Also known as **mutex locks**
- Can implement a counting semaphore S as a binary semaphore
- Provides mutual exclusion

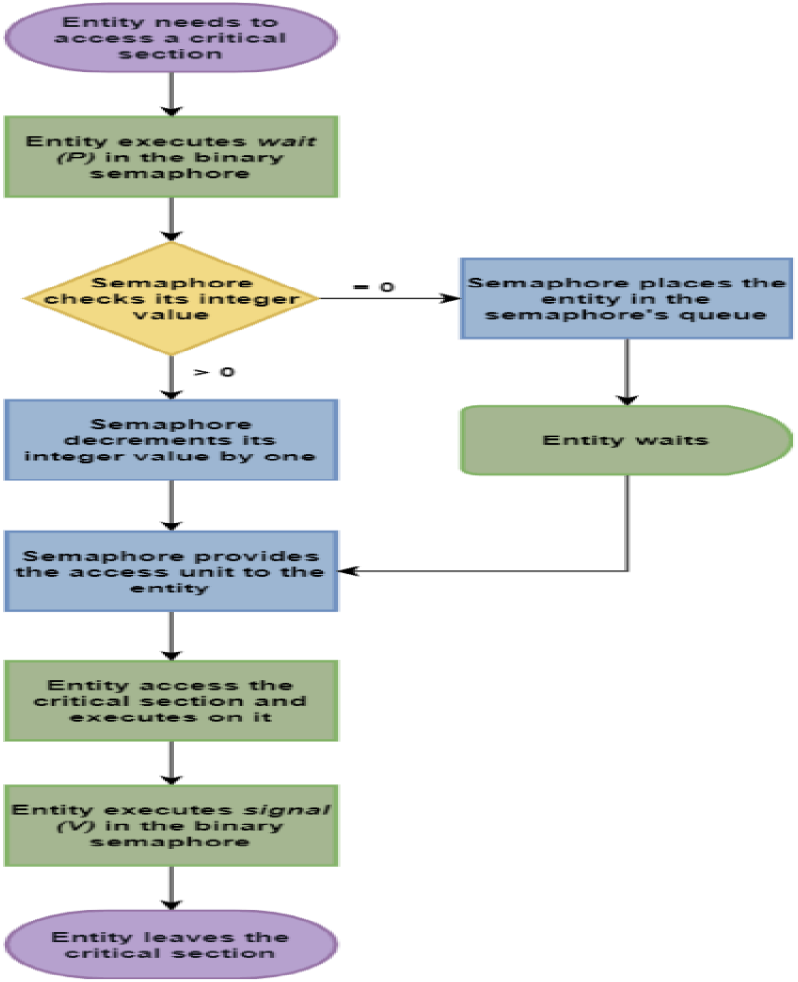
```
Semaphore mutex;    //  initialized to 1
 do {
   wait (mutex);
    // Critical Section
   signal (mutex);
 // remainder section
 } while (TRUE);
```

BINARY SEMAPHORE - **COUNTING** SEMAPHORE



Binary semaphore

	Binary Semaphore	Counting Semaphore
Synchronization	Yes	Yes
Mutual Exclusion	Yes	No
Access Units	One	Multiple
Internal Value Range	[0;1]	[0;n] (n >1)



Counting semaphore

SEMAPHORE IMPLEMENTATION

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

SEMAPHORE IMPLEMENTATION WITH NO BUSY WAITING

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

SEMAPHORE IMPLEMENTATION WITH WITH SLEEP AND WAKUP

- Semaphore operations now defined as

```
typedef struct{
int value;
struct process *Ptr;
} Semaphorre
```

```
void wait(Semaphore S){
        S.value --;
        if (S.value < 0)
            {
            add this process to S.L;
            block();
            }
        }
```

```
void signal(Semaphore S) {
        S.value ++;
        if (S.value ≤ 0)
            {
            remove a process P from S.L;
            wakeup(P);
            }
        }
```

SEMAPHORE IMPLEMENTATION WITH SLEEP AND WAKUP

P(S)

```
void wait(Semaphore S){
        S.value --;
        if (S.value < 0)
            {
            add this process to S.L;
            block();
            }
}
```

PLAY

V(S)

```
void signal(Semaphore S) {
 S.value ++;
 if (S.value ≤ 0)
      {
      remove a process P from S.L;
      wakeup(P);
      }
}
```

waiting | Ready

Block	Block	Block	
P4	P3	P2	P1
wait(S)	wait(S)	wait(S)	wait(S)

S.value=-1

Shared Data

Signal(S)

Critical Section

PROBLEMS WITH SEMAPHORES

- Incorrect use of semaphore operations:
  - **`signal (mutex)`** …. **`wait (mutex)`**
  - **`wait (mutex)`** … **`wait (mutex)`**
  - Omitting of **`wait (mutex)`** and/or **`signal (mutex)`**
- These – and others – are examples of what can occur when semaphores and other synchronization tools are used incorrectly.

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

P_0 P_1

wait(S); *wait*(Q);

wait(Q); *wait*(S);

⋮ ⋮

signal(S); *signal*(Q);

signal(Q) *signal*(S);

- Starvation – indefinite blocking. A process may never be removed from the semaphore queue in which it is suspended.

CLASSICAL PROBLEMS OF SYNCHRONIZATION

Classical problems used to test newly-proposed synchronization schemes

- Bounded-Buffer Problem
- Readers and Writers Problem
- Dining-Philosophers Problem

BOUNDED-BUFFER PROBLEM

- *N* buffers, each can hold one item
- Semaphore mutex initialized to the value 1
- Semaphore full initialized to the value 0
- Semaphore empty initialized to the value N

SEMAPHORE WITH PRODUCER-CONSUMER PROBLEM

- Producer

```
do  {

    //   produce an item in nextp
              wait (empty);// check vacancies
              wait (mutex);
    //  add the item to the  buffer
               signal (mutex);
               signal (full);
       } while (TRUE);
```

- Consumer

```
do {
            wait (full);
            wait (mutex);
        //  remove an item from  buffer to nextc
            signal (mutex);
            signal (empty);

         //  consume the item in nextc
      } 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
- Problem – allow multiple readers to read at the same time
  - Only one single writer can access the shared data at the same time
- Several variations of how readers and writers are considered – all involve some form of priorities
- Shared Data
  - Data set
  - Semaphore **`rw_mutex`** initialized to 1
  - Semaphore **`mutex`** initialized to 1
  - Integer **`read_count`** initialized to 0

SEMAPHORE WITH **READERS-WRITERS** PROBLEM

- writer

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

- reader

```
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);
```

OUTLINE

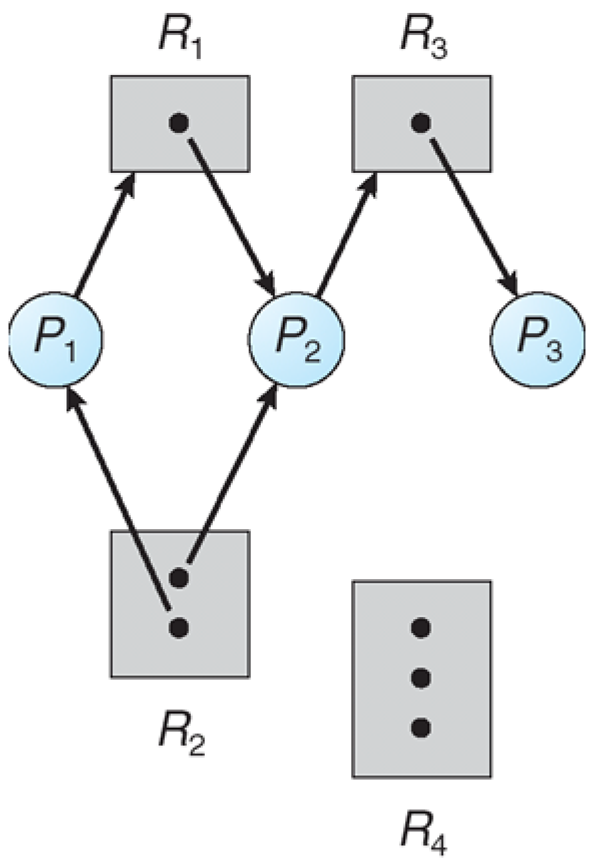
- Overview
- Interprocess Communication
- Thread
- Scheduling Algorithms
- Process Synchonization
- Deadlocks

DEADLOCK CHARACTERIZATION

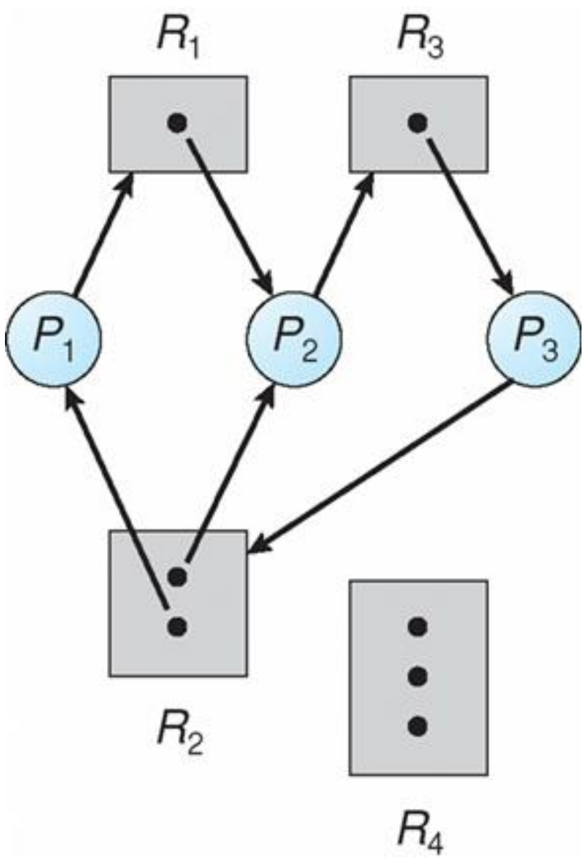
Deadlock can arise if four conditions hold simultaneously.

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

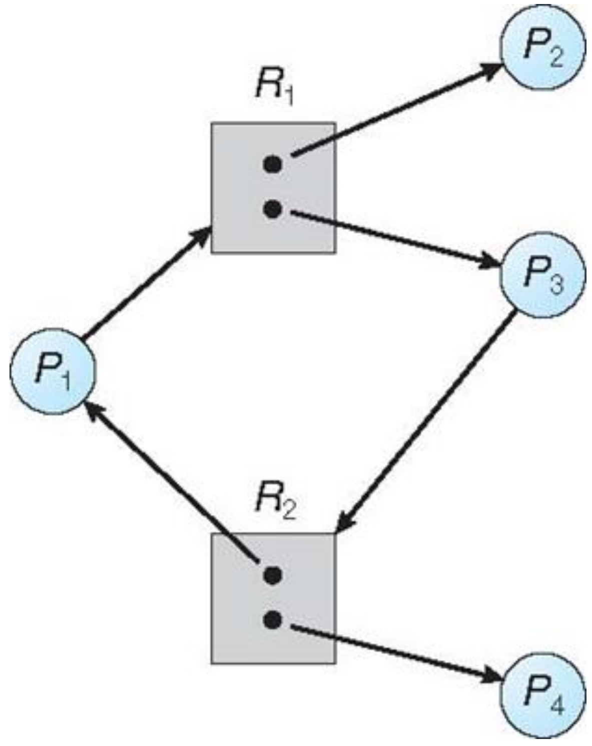
EXAMPLE OF A RESOURCE ALLOCATION GRAPH



Example of a Resource Allocation Graph



Graph With a Deadlock



Graph With A Cycle But No Deadlock

METHODS FOR HANDLING DEADLOCKS

- Ensure that the system will ***never*** enter a deadlock state:
  - Deadlock prevention
  - Deadlock avoidence
- Allow the system to enter a deadlock state and then recover
- Ignore the problem and pretend that deadlocks never occur in the system; used by most operating systems, including UNIX

DEADLOCK PREVENTION

- **Mutual Exclusion** – not required for sharable resources (e.g., read-only files); must hold for non-sharable resources
- **Hold and Wait** – must guarantee that whenever a process requests a resource, it does not hold any other resources
  - Require process to request and be allocated all its resources before it begins execution, or allow process to request resources only when the process has none allocated to it.
  - Low resource utilization; starvation possible

DEADLOCK PREVENTION

- **No Preemption –**
  - If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released
  - Preempted resources are added to the list of resources for which the process is waiting
  - Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting
- **Circular Wait** – impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration

DEADLOCK AVOIDANCE

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

SAFE STATE

- When a process requests an available resource, system must decide if immediate allocation leaves the system in a safe state
- System is in **safe state** if there exists a sequence <P_1, P_2, …, P_n> of ALL the processes in the systems such that for each P_i, the resources that P_i can still request can be satisfied by currently available resources + resources held by all the P_j, with j <i
- That is:
  - If P_i resource needs are not immediately available, then P_i can wait until all P_j have finished
  - When P_j is finished, P_i can obtain needed resources, execute, return allocated resources, and terminate
  - When P_i terminates, P_{i+1} can obtain its needed resources, and so on

BANKER'S ALGORITHM

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

DATA STRUCTURES FOR THE BANKER'S ALGORITHM

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

SAFETY ALGORITHM

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

Work* = *Available

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

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

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

(b) ***Need***$_i$ **<= *Work***

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

3. ***Work* = *Work* + *Allocation***$_i$
Finish*[*i*] = *true
go to step 2

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

RESOURCE-REQUEST ALGORITHM FOR PROCESS P_I

- $\boldsymbol{Request_i}$ = request vector for process $\boldsymbol{P_i}$. If $\boldsymbol{Request_i\,[j] = k}$ then process $\boldsymbol{P_i}$ wants $\boldsymbol{k}$ instances of resource type $\boldsymbol{R_j}$

1. If $\boldsymbol{Request_i \le Need_i}$ go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim

2. If $\boldsymbol{Request_i \le Available}$, go to step 3. Otherwise $\boldsymbol{P_i}$ must wait, since resources are not available

3. Pretend to allocate requested resources to $\boldsymbol{P_i}$ by modifying the state as follows:

$\boldsymbol{Available = Available - Request_i;}$

$\boldsymbol{Allocation_i = Allocation_i + Request_i;}$

$\boldsymbol{Need_i = Need_i - Request_i;}$

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

EXAMPLE OF BANKER'S ALGORITHM

- 5 processes P_0 through P_4;

  3 resource types:

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

- Snapshot at time T_0:

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

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

EXAMPLE OF BANKER'S ALGORITHM

- 5 processes P_0 through P_4;

  3 resource types:

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

- Snapshot at time T_0:

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

EXAMPLE (CONT.)

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

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

EXAMPLE: P_1 REQUEST (1,0,2)

- Check that Request <= Available (that is, (1,0,2) <= (3,3,2) -> true

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

- Executing safety algorithm shows that sequence $\langle \boldsymbol{P_1, P_3, P_4, P_0, P_2} \rangle$ satisfies safety requirement
- Can request for (3,3,0) by $\boldsymbol{P_4}$ be granted?
- Can request for (0,2,0) by $\boldsymbol{P_0}$ be granted?

RECOVERY FROM DEADLOCK: PROCESS TERMINATION

- Abort all deadlocked processes
- Abort one process at a time until the deadlock cycle is eliminated
- In which order should we choose to abort?
  - 1.Priority of the process
  - 2.How long process has computed, and how much longer to completion
  - 3.Resources the process has used
  - 4.Resources process needs to complete
  - 5.How many processes will need to be terminated
  - 6. Is process interactive or batch?

RECOVERY FROM DEADLOCK: RESOURCE PREEMPTION

- **Selecting a victim** – minimize cost
- **Rollback** – return to some safe state, restart process for that state
- **Starvation** – same process may always be picked as victim, include number of rollback in cost factor