CS 471 – Operating Systems

Lectures

1) Introduction

2) Operating System Structures

3) Processes

4) Threads

5) CPU Scheduling

6) Process Synchronization

7) Deadlocks

8) Main Memory

9) Virtual Memory

10) File-System Interface

11) File-System Implementation

12) Mass-Storage Structure

13) I/O Systems

14) Protection

15) Security

Operating-System Structures

★ A View of Operating System Services

- **Communications** –Processes may exchange information, on the ***same*** computer or ***between*** computers over a network.
  Communications may be via ***shared memory*** or through ***message passing*** (packets moved by the OS)

- **Error detection** – OS needs to be constantly aware of possible errors
  - May occur in the CPU and memory hardware, in I/O devices, in user program
  - For each type of error, OS should take the appropriate action to ensure correct and consistent computing
  - Debugging facilities can greatly enhance the user's and programmer's abilities to efficiently use the system

- **Resource allocation -** When multiple users or multiple jobs running concurrently, resources must be allocated to each of them.
  Many types of resources - Some (such as CPU cycles, main memory, and file storage) may have special allocation code,
  others (such as I/O devices) may have general request and release code

- **Accounting -** To keep track of which users use how much and what kinds of computer resources

- **Protection and security -** The owners of information stored in a multiuser or networked computer system may want to control use of that information, concurrent processes should not interfere with each other
  - **Protection** involves ensuring that all access to system resources is controlled
  - **Security** from outsiders requires user authentication, extends to defending external I/O devices from invalid access attempts.
  - For a system to be protected and secure, precautions must be instituted throughout it. **A chain is only as strong as its weakest link**.

★ User Operating System Interface

- **Command Line Interface** (**CLI**) or command interpreter allows direct command entry
  - Sometimes implemented in kernel, sometimes by systems program
  - Sometimes multiple flavors implemented – **shells**
  - Primarily fetches a command from user and executes it
    - Sometimes commands built-in, sometimes just names of programs
      If the latter, adding new commands doesn't require shell modification.

- User-friendly desktop **Graphical User Interface** (**GUI**) metaphor interface
  - Usually mouse, keyboard, and monitor
  - Icons represent files, programs, actions, etc
  - Various mouse buttons over objects in the interface cause various actions
    (Provide information, options, execute function, open directory (folder)
  - Invented at Xerox PARC

- Many systems now include both **CLI & GUI** interfaces
  - Microsoft Windows is GUI with CLI "command" shell
  - Apple Mac OS has GUI interface with UNIX kernel underneath and shells available
  - Solaris is CLI with optional GUI interfaces (Java Desktop, KDE)

★ System Calls

- Programming interface to the services provided by the OS
- Typically written in a high-level language (C or C++)
- Mostly accessed by programs via a high-level Application Program Interface (**API**)
- Three most common APIs are
  - Win32 API for Windows,
  - POSIX API (all versions of UNIX, Linux, and Mac OS), and
  - Java API for the Java virtual machine (JVM)

Example of System Calls:

Examples:

Example 1: copy

Description: Open an **existing** input file and copy it to **non-existing** output file.

C-version

http://www.cs.odu.edu/~cs471w/code/misc/copy.c

```
int
main(int argc, char **argv)
{
  int             n, in, out;
  char            buf[1024];
  char            InputFile[1024];
  char            OutputFile[1024];
  struct stat     st;

  printf("enter Input File Name: ");
  scanf("%s", InputFile);
  if (stat(InputFile, &st) != 0) {
       perror("InputFile does not exit");
       exit(1);
  }
  printf("enter Outut File Name: ");
  scanf("%s", OutputFile);
  if (stat(OutputFile, &st) == 0) {
       perror("OutputFile exists");
       exit(1);
  }

  if ((in = open(InputFile, O_RDONLY)) < 0) {
       perror(InputFile);
       exit(1);
  }

  if ((out = open(OutputFile, O_CREAT | O_WRONLY, 0755)) < 0) {
       perror(OutputFile);
       exit(1);
  }

  while ((n = read(in, buf, sizeof(buf))) > 0)
       write(out, buf, n);
```

```
  close(out);
  close(in);
  printf("copied %s to %s\n", InputFile, OutputFile);
  exit(0);
}
```

To execute:

% copy

NOTE: **0755** means file permission should be: **rwx r-x r-x**
However, the shell varaible ***umask*** may change the final permission.
For example in ***/home/cs771w/.cshrc*** the value of umask **0027: rwx r-x ---**
The file permission should not exceed the umask value and
therefore the file created will have permission: **rwx r-x ---**

❖ Java-version

http://www.cs.odu.edu/~cs471w/code/misc/jcopy.java

```
public class jcopy {
   public static void main(String[] args) throws IOException {

     BufferedReader stdin = new BufferedReader( new InputStreamReader(System.in));

     System.out.println("enter Input File Name: ");
     String InputFile = stdin.readLine();
     File infile = new File(InputFile);
     InputStream fin = new FileInputStream(infile);

     System.out.println("enter Outut File Name: ");
     String OutputFile = stdin.readLine();
     File outfile = new File(OutputFile);
     OutputStream fout = new FileOutputStream(outfile);

     int len;
     byte[] buf = new byte[1024];
     while ((len = fin.read(buf)) > 0) {
         fout.write(buf, 0, len);
     }
     fin.close();
     fout.close();
     System.out.println("copied " + InputFile + " to " +  OutputFile);
   }
}
```

To execute:

% java jcopy

❖ Python-version

http://www.cs.odu.edu/~cs471w/code/misc/copy.py

```
#! /usr/bin/python
import sys
print ("enter Input File Name: ")
fname = sys.stdin.readline()
InputFile=fname.strip('\n')
print(InputFile)
print ("enter Output File Name: ")
fname = sys.stdin.readline()
OutputFile=fname.strip('\n')
```

```
print(OutputFile)
try:
     fin = open(InputFile,'r')
     fout = open(OutputFile,'w')
     for line in fin:
          fout.write(line)
     fin.close()
     fout.close()
except:
     print ('Something went wrong.')
```

To execute:

% copy.py

➔ Example 2: save:

Description: save 2nd argument (a string) into 1st argument file.
If the file exists it is truncated, otherwise it is created.
It then displays the content of the file to tty (standard output or file descriptor 1).

❖ C-version:

http://www.cs.odu.edu/~cs471w/code/misc/save.c

```
int
main(int argc, char *argv[])
{
    int sfd;
    int n;
    char buf[1024];
    int FileLength;
    char *FileName = argv[1];
    char *Message = argv[2];

    if ((sfd = open(FileName, O_RDWR | O_TRUNC| O_CREAT, 0600)) < 0) {
        perror(FileName);
        exit(1);
    }

    write(sfd, Message, strlen(Message));
    lseek (sfd,0, SEEK_SET);
    printf("\nFile '%s' contains:\n", FileName);
    fflush(stdout);
    n=read(sfd, buf, sizeof(buf));
    write(1,buf, n);
    FileLength = lseek(sfd, 0, SEEK_END);
    printf("\n\nFile '%s' length: '%d'\n", FileName, FileLength);
}
```

To execute:

% save savefile "how are you?"

❖ Java-version

http://www.cs.odu.edu/~cs471w/code/misc/jsave.java

```
public class jsave {
   public static void main(String[] args) throws IOException {
```

```
      File FileName = new File(args[0]);
      String Message = args[1];

      OutputStream fout = new FileOutputStream(FileName);
      fout.write(Message.getBytes());

     System.out.println("File " + FileName + " contains:");

     BufferedReader fin = new BufferedReader( new FileReader(FileName));
     String str;
     while((str = fin.readLine())!= null)
            System.out.println(str);
      long length = FileName.length();
      System.out.println( FileName + " lenght is: " + length);

   }
}
```

To execute:

% java jsave savefile "how are you?"

❖ Python-version

http://www.cs.odu.edu/~cs471w/code/misc/save.py

```
#! /usr/bin/env python

import sys
fname = sys.argv[1]
msg = sys.argv[2]
try:
     fout = open(fname,'w')
     fout.write(msg)
        fout.close()
     fin = open(fname,'r')
        for line in fin:
                print(line)
     fin.seek(0,2) # move the cursor to the end of the file
     size = fin.tell()
     print(size)
except:
     print ('Something went wrong.')
```

To execute:

% save.py savefile "how are you?"

API – System Call – OS Relationship

Standard C Library Example:

C program invoking **printf()** library call, which calls **write()** system call

System Call Parameter Passing

Three general methods used to pass parameters to the OS

1. Simplest: pass the parameters in **registers**
2. Parameters stored in a **block**, or table, in memory, and ***address*** of block passed as a parameter in a register. This approach taken by Linux and Solaris
3. Parameters placed, or ***pushed***, onto the **stack** by the program and ***popped*** off the stack by the operating system

Block and stack methods do not limit the number or length of parameters being passed

Types of System Calls:

- ✓ Process control
- ✓ File management
- ✓ Device management
- ✓ Information maintenance
- ✓ Communications
- ✓ Protection

Examples of Windows and Unix System Calls

	Windows	Unix
Process Control	CreateProcess()	fork()
	ExitProcess()	exit()
	WaitForSingleObject()	wait()
File Manipulation	CreateFile()	open()
	ReadFile()	read()
	WriteFile()	write()
	CloseHandle()	close()
Device Manipulation	SetConsoleMode()	ioctl()
	ReadConsole()	read()
	WriteConsole()	write()
Information Maintenance	GetCurrentProcessID()	getpid()
	SetTimer()	alarm()
	Sleep()	sleep()
Communication	CreatePipe()	pipe()
	CreateFileMapping()	shmget()
	MapViewOfFile()	mmap()
Protection	SetFileSecurity()	chmod()
	InitlializeSecurityDescriptor()	umask()
	SetSecurityDescriptorGroup()	chown()

★ System Programs

- System programs provide a convenient environment for program development and execution.
  They can be divided into:
  - **File** management
  - **Status** information
  - **Programming language** support, loading and execution.
  - **Communications**
  - **Application programs**
- Most users' view of operating system is defined by system programs, not the actual system calls
- Provide a convenient environment for program development and execution
  Some of them are simply user interfaces to system calls; others are considerably more complex
- **File management** - create, delete, copy, rename, print, dump, list, and generally manipulate files and directories

- **Status information**
  - **System info** - date, time, amount of available memory, disk space, number of users
  - Others provide detailed performance, logging, and debugging information
  - Typically, these programs format and print the output to the terminal or other output devices

★ Operating System Design and Implementation

- **User goals** – operating system should be **convenient** to use, **easy** to learn, **reliable**, **safe**, and **fast**
- **System goals** – operating system should be easy to design, implement, maintain, flexible, reliable, error-free, and efficient
- Important principle to separate:
  Policy: *What will be done*?
  Mechanism: *How to do it*?
  The separation of policy from mechanism is a very important principle,
  it allows maximum flexibility if policy decisions are to be changed later

Traditional UNIX System Structure:

The UNIX OS consists of two separable parts

- Systems programs
- The kernel

  - Consists of everything below the system-call interface and above the physical hardware

  - Provides the file system, CPU scheduling, memory management, and other operating-system functions;

★ Virtual Machines

- A virtual machine provides an interface *identical* to the underlying bare hardware
- The operating system host creates the illusion that a process has its own processor and (virtual memory)
- Each guest provided with a (virtual) copy of underlying computer

VMware Architecture:

The Java Virtual Machine:

-

★ Operating-System Debugging

- Debugging is finding and fixing errors, or bugs
- OSs generate log files containing error information
- Failure of an application can generate **core dump** file capturing memory of the process
- Operating system failure can generate crash dump file containing kernel memory
- ***Kernighan's Law***: "**Debugging** *is* **twice as hard** *as* **writing the code** *in the first place. Therefore, if you write the code as* **cleverly** *as possible, you are, by definition,* **not smart** *enough to debug it*."

★ Operating System Generation

- Operating systems are designed to run on any class of machines; the system must be **configured** for each specific computer site
- **SYSGEN** program obtains information for specific configuration of the hardware system
- **Booting** – starting a computer by loading the kernel
- **Bootstrap program** – code stored in ROM that is able to locate the kernel, load it into memory, and start its execution

System Boot:

Operating system must be made available to hardware to start

- Small piece of code – **bootstrap loader**, locates the kernel, loads it into memory, and starts it
- Sometimes two-step process where **boot block** at fixed location loads bootstrap loader
- When power initialized on system, execution starts at a fixed memory location

Procesess

❖Process in Memory

As a process executes, it changes ***state***

- **new**: The process is being created
- **ready**: The process is waiting to be assigned to a processor
- **running**: Instructions are being executed
- **waiting**: The process is waiting for some event to occur
- **terminated**: The process has finished execution

Diagram of Process State

-

Process Control Block (PCB

Information associated with each process

CPU Switch from Process to Process

Process Scheduling Queues

- **Job queue** – set of all processes in the system
- **Ready queue** – set of all processes residing in main memory, ready and waiting to execute
- **Device queues** – set of processes waiting for an I/O device

Processes migrate among the various queues

Representation of Process Scheduling

Schedulers

- **Long-term scheduler** (or job scheduler) – selects which processes should be in the ready queue
- **Short-term scheduler** (or CPU scheduler) – selects which process should be allocated the CPU
  - Short-term scheduler is invoked very frequently (milliseconds) Þ (must be fast)
  - Long-term scheduler is invoked very infrequently (seconds, minutes) Þ (may be slow)
  - The long-term scheduler controls the *degree of multiprogramming*
  - Processes can be described as either:
    - **I/O-bound process** – spends more time doing I/O than computations, many short CPU bursts
    - **CPU-bound process** – spends more time doing computations; few very long CPU bursts

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 an overhead; the system does no useful work while switching

❖Process Creation

- **Parent** process create **children** processes, which, in turn create other processes, forming a **tree** of processes
- Generally, process identified and managed via **a process identifier** (**pid**)

- **Execution**
  - Parent and children **execute concurrently**
  - Parent may **waits** until children terminate
- **Address space**
  - Child **duplicate** of parent
  - Child may **load new program**.
- **UNIX examples**
  - ***fork*** system call creates new process
  - ***exec*** system call used after a fork to replace the process' memory with a new program

EXAMPLE:

newproc-posix.c

```
int main()
{
    pid_t  pid;
    pid = fork();
    if (pid == 0) { /* child process */
        execlp("/bin/ls", "ls", NULL);
    }
```

```
        else { /* parent process */
             wait (NULL);
             printf ("Child Complete");
             exit(0);
        }
    }
```

★ To execute:

```
% newproc-posix
```

EXAMPLE:

newproc.py

```
#! /usr/bin/env python
import sys
import os

childpid = os.fork()
if childpid == 0:
    print("I am child")
    os.execlp("/bin/date","date");
    os.exit(0)
else:
    os.wait()
    print("I am the parent of: ", childpid)
```

★ To execute:

```
% newproc.py
```

❖ 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**)
- Examples of IPC
  - ✓ Shared memory
  - ✓ Message passing

Producer-Consumer Problem

Paradigm for cooperating processes,
producer process produces information that is consumed by a **consumer** process
- **unbounded-buffer** places no practical limit on the size of the buffer
- **bounded-buffer** assumes that there is a fixed buffer size

Bounded-Buffer – Shared-Memory Solution

Shared data:

```
#define BUFFER_SIZE 10

typedef struct {
   . . .
} item;

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

Producer:

```
while (true) {
   while ((( (in + 1) % BUFFER_SIZE)  == out)
      ;   /* do nothing -- no free buffers */
   /* Produce an item */
    buffer[in] = item;
    in = (in + 1) % BUFFER_SIZE;
}
```

Consumer:

```
while (true) {
    while (in == out)
      ; /* do nothing -- nothing to consume */
     /* Consume an item */
     item = buffer[out];
     out = (out + 1) % BUFFER_SIZE;
}
```

✓ **This solution is correct, but can only use BUFFER_SIZE - 1 elements**

Interprocess Communication – Message Passing

- IPC facility provides two operations:
  ✓ ***send*** *(message)* – message size fixed or variable
  ✓ ***receive*** *(message)*

- If ***P*** and ***Q*** wish to communicate, they need to:
  ✓ establish a ***communication link*** between them
  ✓ ***exchange*** messages via send/receive

Direct Communication

Processes must name each other explicitly:
- ***send (P****, message)* – send a message to process ***P***

- ***receive (Q, message)*** – receive a message from process ***Q***

Indirect Communication

Messages are directed and received from **mailboxes** (also referred to as ports)

- Each mailbox has a unique id
- Processes can communicate only if they share a mailbox

Synchronization

Message passing may be either ***blocking*** or ***non-blocking***

- **Blocking** is considered ***synchronous***
  - ✓ **Blocking send** has the sender block until the message is received
  - ✓ **Blocking receive** has the receiver block until a message is available
- **Non-blocking** is considered ***asynchronous***
  - ✓ **Non-blocking** send has the sender send the message and continue
  - ✓ **Non-blocking** receive has the receiver receive a valid message or null

---

❖ Shared Memory

- Process first creates shared memory segment:
  segment id = ***shmget*** *(IPC_PRIVATE, size, S_IRUSR | S_IWUSR);*
- Process wanting access to that shared memory must ***attach*** to it:
  sharedmemory = *(char *)* ***shmat*** *(**id**, NULL, 0);*
- The process could write/read to/from the shared memory:
  ***sprintf (sharedmemory**, "Writing to shared memory");*

  ***printf("%s", sharedmemory)**;*
- When done a process can ***detach*** the shared memory from its address space:
  shmdt *(**sharedmemory**);*

EXAMPLES:

shm-fork-posix.c

```
int  main()
{
     pid_t           pid;
     int             segment_id;
     char           *shared_memory;
     const int       segment_size = 4096;


     segment_id = shmget(IPC_PRIVATE, segment_size, S_IRUSR | S_IWUSR);
     shared_memory = (char *) shmat(segment_id, NULL, 0);

     pid = fork();

     if (pid == 0) {          /* child process */
```

```
                sprintf(shared_memory, "I love my parents");

            } else {                /* parent process */
                  sleep(1);
                  printf("Parent Got: %s\n", shared_memory);
                if (shmdt(shared_memory) == -1) {
                        fprintf(stderr, "Unable to detach\n");
                }
                shmctl(segment_id, IPC_RMID, NULL);
                return 0;
            }
        }
    }
```

★ To execute:

```
% shm-fork-posix
```

***Shared/NonShared* Example:**

shmfork.c:

```
int
main()
{
   int             segment_id = shmget(IPC_PRIVATE, sizeof(int), S_IRUSR | S_IWUSR);
   int            *sharedVar = (int *) shmat(segment_id, NULL, 0);
   int            *nonsharedVar = (int *) malloc(sizeof(int));


    sharedVar[0] = 0;
    nonsharedVar[0] = 0;

   if (fork() == 0) {
      sharedVar[0]=getpid();
      nonsharedVar[0]=getpid();
      printf("\nChild  : shared %d, nonshared %d\n", sharedVar[0], nonsharedVar[0]);
      exit(0);
   }
   wait(NULL);
   printf("Parent : shared %d, nonshared %d\n\n", sharedVar[0], nonsharedVar[0]);
}
```

★ To execute:

```
% shmfork
```

❖Client-Server Communication

Sockets

- A socket is defined as an ***endpoint*** for communication
- Is ***bind*** to of **IP address (e.g., 128.82.4.98)** and **port (e.g., 10117).**
- Communication is between a ***pair*** of sockets

EXAMPLE:

DateServer.java

```
public class DateServer
{
    public static void main(String[] args)  {
     try {
         ServerSocket sock = new ServerSocket(10117);
         // now listen for connections
         while (true) {
             Socket client = sock.accept();
             PrintWriter pout = new PrintWriter(client.getOutputStream(), true);

             // write the Date to the socket
             pout.println(new java.util.Date().toString());

             // close the socket and resume listening for more connections
             client.close();
         }
     }
     catch (IOException ioe) {
             System.err.println(ioe);
     }
    }
}
```

DateClient.java

```
public class DateClient
{
    public static void main(String[] args)  {
     try {
         // this could be changed to an IP name or address other than the localhost
         Socket sock = new Socket("127.0.0.1",10117);
         InputStream in = sock.getInputStream();
         BufferedReader bin = new BufferedReader(new InputStreamReader(in));
         String line;
         while( (line = bin.readLine()) != null)
             System.out.println(line);
         sock.close();
     }
     catch (IOException ioe) {
             System.err.println(ioe);
     }
}
```

★ To execute:

```
% java   DateServer
```

In another window at the **same host:**

```
% java   DateClient
```

DateServer.c

```
int main(void)
{
      int             len, listenfd, connfd;
      struct sockaddr_in servaddr, cliaddr;
      char            buff[512];
      time_t          ticks;

      listenfd = socket (AF_INET, SOCK_STREAM, 0);

      bzero(&servaddr, sizeof(servaddr));
      servaddr.sin_family = AF_INET;
      servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
      servaddr.sin_port = htons(10117);

      bind (listenfd, (SA *) & servaddr, sizeof(servaddr));

      listen(listenfd, 0);

      for (;;) {
              len = sizeof(cliaddr);
              connfd = accept (listenfd, (SA *) & cliaddr, &len);

              ticks = time(NULL);
              snprintf(buff, sizeof(buff), "%.24s\r\n", ctime(&ticks));
              write (connfd, buff, strlen(buff));
              close (connfd);
      }
}
```

DateClient.c

```
int main(int argc, char **argv)
{
      int             sockfd, n;
      char            recvline[512];
      struct sockaddr_in servaddr;
      struct hostent *hp, *gethostbyname();

      sockfd = socket (AF_INET, SOCK_STREAM, 0);

      bzero(&servaddr, sizeof(servaddr));
      servaddr.sin_family = AF_INET;

      hp = gethostbyname( "localhost" );
      bcopy ( hp->h_addr, &(servaddr.sin_addr.s_addr), hp->h_length);
      servaddr.sin_port = htons(10117);

      if (connect (sockfd, (SA *) & servaddr, sizeof(servaddr)) < 0) {
              perror("connect error");
              exit(-1);
      }
      read (sockfd, recvline, 512);
      printf("%s", recvline);
```

```
        exit(0);
}
```

★ To execute:

```
% DateServer
```

In another window at the **same host:**

```
% DateClient
```

DateServer.py

```
#! /usr/bin/env python
import socket
import sys
import os
import datetime

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

host = ''
port = 10117
s.bind((host, port))
s.listen(1)

while (1):
  conn, addr = s.accept()
  print 'client is at', addr
  #today = datetime.date.today()
  date= datetime.datetime.now().strftime("%A %B %d %H:%M:%S EST %Y \n")
       conn.send(date)
  conn.close()
```

DateClient.py

```
#! /usr/bin/env python
import socket
import sys

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

host = "localhost"
port = 10117
s.connect((host, port))
date = s.recv(1000000)
print (date)
s.close()
```

★ To execute:

```
% DateServer.py
```

In another window at the **same host:**

```
% DateClient.py
```

Mix & Match:

You may use any **c/java/python** ***server*** talk to any **c/java/python** ***client***.

Also Try:

```
% telnet 127.0.0.1 10117
With all 3 servers.
```

i.e., there are 4x3 = 12 possible combinations.

❖Pipes

- **Ordinary Pipes** allow communication in standard producer-consumer style
- Producer writes to one end (the ***write-end*** of the pipe)
- Consumer reads from the other end (the ***read-end*** of the pipe)
- Ordinary pipes are therefore ***unidirectional***
- Require **parent-child** relationship between communicating processes

EXAMPLES:

unix_pipe.c

-

```
#define BUFFER_SIZE 25
#define READ_END        0
#define WRITE_END       1

int main(void)
{
    char write_msg[BUFFER_SIZE] = "Greetings";
    char read_msg[BUFFER_SIZE];
    pid_t pid;
    int fd[2];

    /** create the pipe */
    pipe(fd)
    /** now fork a child process */
    pid = fork();

    if (pid > 0) {  /* parent process */
            /* close the unused end of the pipe */
            close(fd[READ_END]);
            /* write to the pipe */
            write(fd[WRITE_END], write_msg, strlen(write_msg)+1);
            /* close the write end of the pipe */
            close(fd[WRITE_END]);
    }
    else { /* child process */
            /* close the unused end of the pipe */
            close(fd[WRITE_END]);
            /* read from the pipe */
            read(fd[READ_END], read_msg, BUFFER_SIZE);
            printf("child read %s\n",read_msg);
```

```
                /* close the write end of the pipe */
                close(fd[READ_END]);
        }
    }
```

★ To execute:

```
% unix_pipe
```

pipefork.c:

```
main()
{
   char buf[10];
   int pfd[2];
   pipe(pfd);
   int n;

   if (fork() == 0) {
      write(pfd[1], "child\n", 6);
      n=read(pfd[0], buf, 10);
      write(1, buf, n);
      exit(0);
   }
      n=read(pfd[0], buf, 10);
      write(1, buf, n);
      write(pfd[1], "parent\n", 7);
      wait(NULL);
}
```

★ To execute:

```
% pipefork
```

pipe.py:

```
#!/usr/bin/python

import os, sys

# file descriptors r, w for reading and writing
r, w = os.pipe()

processid = os.fork()
if processid: #parent
    os.close(w)
    r = os.fdopen(r)
    str = r.read()
    print "Parent Got: ", str
    sys.exit(0)
else:
    os.close(r)
    w = os.fdopen(w, 'w')
    w.write("How are you my dear Parent.." )
    w.close()
    sys.exit(0)
```

★ To execute:

```
% pipe.py
```

❖Shared Files

-

File Sharing between processes

EXAMPLES:

sharedFile1.c (single open)

```
int
main(int argc, char *argv[])
{
  int             n;
  int             i;
  int             sfd;
  pid_t           pid;
  char            buf[1024];

  sfd = open("TempFile", O_RDWR | O_TRUNC | O_CREAT, 0600);

  pid = fork;

  if (pid == 0) {
       sleep(1);
       lseek(sfd, 0, SEEK_SET);
       n = read(sfd, buf, sizeof(buf));
       write(1, buf, n);
       exit(0);
  } else { /* Parent */
       write(sfd, "Hi\n", 3);
       sleep(2);
       exit(0);
  }
}
```

★ To execute:

```
% sharedFile1
```

sharedFile2.c (two opens)

```
int
main(int argc, char *argv[])
{
  int             n;
  int             sfd;
  int             sfd2;
  pid_t           pid;
  char            buf[1024];

  pid = fork();

  if (pid == 0) { /* child */
       sleep(1);
       sfd2 = open("TempFile", O_RDWR;
       n = read(sfd2, buf, sizeof(buf));
       write(1, buf, n);
       exit(0);
  } else { /* parent */
       sfd = open("TempFile", O_RDWR | O_TRUNC | O_CREAT, 0600) ;
       write(sfd, "Hi\n", 3);
       sleep(2);
       exit(0);
  }
}
```

★ To execute:

```
% sharedFile2
```

❖ Echo Server (Serving *single* client)

EchoServer.c

```
int
main(int argc, char* argv[])
{
    int             len, listenfd, connfd;
    struct sockaddr_in servaddr, cliaddr;
    char            buff[512];
    int             nread;

    listenfd = socket(AF_INET, SOCK_STREAM, 0);

    bzero(&servaddr, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
    servaddr.sin_port = htons(atoi(argv[1]));

    bind(listenfd, (SA *) & servaddr, sizeof(servaddr));

    listen(listenfd, 0);

    len = sizeof(cliaddr);
    connfd = accept(listenfd, (SA *) & cliaddr, &len);
```

```
    EchoToClient(connfd);
}

void
EchoToClient(int sockfd)
{

    int             nread;
    char            buffer[512];
    for (;;) {
         nread = read(sockfd, buffer, 512);
         if (nread == 0)
             break;
         write(sockfd, buffer, nread);
    }
}
```

EchoClient.c

```
int
main(int argc, char **argv)
{
    int             sockfd;
    struct sockaddr_in servaddr;
    struct hostent *hp, *gethostbyname();
    char            buffer[512];
    int             nread;

    sockfd = socket(AF_INET, SOCK_STREAM, 0);

    bzero(&servaddr, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    hp = gethostbyname(argv[1]);
    bcopy(hp->h_addr, &(servaddr.sin_addr.s_addr), hp->h_length);
    servaddr.sin_port = htons(atoi(argv[2]));

    if (connect(sockfd, (SA *) & servaddr, sizeof(servaddr)) < 0) {
         perror("connect error");
         exit(-1);
    }
    for (;;) {
         nread = read(0, buffer, 512);
         if (nread == 0)
             exit(0);
         write(sockfd, buffer, nread);

         nread = read(sockfd, buffer, 512);
         write(1, buffer, nread);
    }
}
```

★ To execute:

```
 sirius % EchoServer   10123
```

In another window for the ***same host sirius***

```
 sirius % EchoClient    localhost     10123
```

OR login to ***another host***, e.g., ***artia***

```
 atria % EchoClient    sirius     10123
```

❖Concurrency using fork

Echo Server (serving *multiple* clients simultaniusly)

EchoServerFork.c

```
int
main(int argc, char *argv[])
{
    int             len, listenfd, connfd;
    struct sockaddr_in servaddr, cliaddr;
    char            buff[512];
    int             nread;

    listenfd = socket(AF_INET, SOCK_STREAM, 0);

    bzero(&servaddr, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
    servaddr.sin_port = htons(atoi[argv[1]]);

    bind(listenfd, (SA *) & servaddr, sizeof(servaddr));

    listen(listenfd, 0);

    for (;;) {
         len = sizeof(cliaddr);
         connfd = accept(listenfd, (SA *) & cliaddr, &len);
         if (fork() == 0)
             EchoToClient(connfd);
    }
}

void
EchoToClient(int sockfd)
{
    int             nread;
    char            buffer[512];
    for (;;) {
         nread = read(sockfd, buffer, 512);
         if (nread == 0)
             exit(0);
         write(sockfd, buffer, nread);
    }
}
```

EchoClientFork.c

```
int
main(int argc, char **argv)
{
    int             sockfd;
    int             pid;
    struct sockaddr_in servaddr;
    struct hostent *hp, *gethostbyname();

    sockfd = socket(AF_INET, SOCK_STREAM, 0);

    bzero(&servaddr, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    hp = gethostbyname(argv[1]);
    bcopy(hp->h_addr, &(servaddr.sin_addr.s_addr), hp->h_length);
```

```
    servaddr.sin_port = htons(atoi(argv[2]));

    if (connect(sockfd, (SA *) & servaddr, sizeof(servaddr)) < 0) {
         perror("connect error");
         exit(-1);
    }
     pid = fork();
    if (pid == 0) { /* child */
         ReadFromKeyBoard(sockfd);
         kill(getppid(), 9);  /* kill parent */
    } else        { /* parent */
         ReadFromSocket(sockfd);
         kill(pid, 9); /* kill child */
    }
}
void
ReadFromKeyBoard(int sockfd)
{
    int             nread;
    char            buffer[512];
    for (;;) {
         nread = read(0, buffer, 512);
         if (nread == 0)
             break;
         write(sockfd, buffer, nread);
    }
}

void
ReadFromSocket(int sockfd)
{
    int             nread;
    char            buffer[512];
    for (;;) {
         nread = read(sockfd, buffer, 512);
         if (nread == 0)
             break;
         write(1, buffer, nread);
    }
}
```

★ **To execute:**

cirius % EchoServerFork 10123

In another window for the same host (vega)

% EchoClientFork localhost 10123

In another window for the another host (antares)

atria % EchoClientFork cirius 10123

Repeat for any number of **other windows.**

❖ **Be Clever!**

Echo Client Simple Version: EchoClientForkShort.c

```
.............
        pid = fork();
        if (pid == 0) {
             ReadFrom(0, sockfd);
             kill(getppid(), 9);
        } else {
             ReadFrom(sockfd, 1);
             kill(pid, 9);
        }
 ........

 void ReadFrom(int fd1, int fd2)
 {

        int              nread;
        char             buffer[512];
        for (;;) {
             nread = read(fd1, buffer, 512);
             if (nread == 0)
                   break;
             write(fd2, buffer, nread);
        }

 }
```

❖ Be Smart!

Echo Client Optimal Version (remember ***Kernighan's Law*** for Debugging!):

Note that writing to 0 is the same as writing to 1!

EchoClientForkOptimum.c

```
      ........
      ........

       dup2(sockfd, 1);
       close(sockfd);
       pid = fork();
       if (pid == 0) {
            ReadFrom(0);
            kill(getppid(), 9);
       } else {
            ReadFrom(1);
            kill(pid, 9);
       }
      }


      void ReadFrom(int fd)
      {

       int              nread;
       char             buffer[512];
       for (;;) {
            nread = read(fd, buffer, 512);
            if (nread == 0)
                  break;
            write((fd + 1) % 2, buffer, nread);
       }
 }
```

❖ Python Echo Server & Client

EchoServerFork.py

```
#! /usr/bin/env python
import socket
import sys
import os
import datetime

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

host = ''
port = int(sys.argv[1])
s.bind((host, port))
s.listen(1)

while (1):
conn, addr = s.accept()
print 'client is at', addr
childpid = os.fork()
if childpid == 0:
              print("child, serving echo client")
       while (1):
                     data = conn.recv(1000000)
              if not data:           # if end of data, leave loop
                     print("child, echo client is done")
                     conn.close()
                     exit(0)

                     conn.send(data)
else:
              print("I am the parent of: ", childpid)
       conn.close()
```

EchoClient.py

```
#! /usr/bin/env python
import socket
import sys

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

host = sys.argv[1] # server address
port = int(sys.argv[2]) # server port
s.connect((host, port))

while(1):
    print 'type a line (type CTRL-D when done)'
    line = sys.stdin.readline()    # keyboard input, ctrl-d to end
    if not line:                   # if end of data, leave loop
       break
    s.send(line)
    data = s.recv(1000000)         # read up to 1000000 bytes
    print 'received', len(data), 'bytes: ', data

s.close()
```

★ To execute:

sirius % EchoServerFork.py 10123

In another window for the **same host**

% EchoClient.py localhost 10123
OR
atria % EchoClient.py sirius 10123

Threads

Single and Multithreaded Processes

- **Benefits**: Responsiveness & Resource Sharing

Multithreaded Server Architecture

Pthreads

- ✓ A POSIX standard API for thread creation and synchronization
- ✓ Common in UNIX operating systems (Solaris, Linux, Mac OS X)

EXAMPLE:

↓thrd2-posix.c:

```
int             sum;      /* this data is shared by the thread(s) */
```

```
int             sum2;     /* this data is shared by the thread(s) */

void           *runner(void *param);   /* the thread */
void           *runner2(void *param);  /* the thread */

int main(int argc, char *argv[])
{
    pthread_t       tid;  /* the thread identifier */
    pthread_t       tid2; /* the thread identifier */
    pthread_attr_t  attr; /* set of attributes for the thread */

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

    /* create the thread */
    pthread_create(&tid, &attr, runner, argv[1]);
    /* create the  2nd thread */
    pthread_create(&tid2, &attr, runner2, argv[1]);

    /* now wait for the thread to exit */
    pthread_join(tid, NULL);
    pthread_join(tid2, NULL);

    printf("sum = %d\n", sum);
    printf("sum2 = %d\n", sum2);
}


void  *runner(void *param)
{
    int             i, upper = atoi(param);
    sum = 0;

    if (upper >= 0) {
         for (i = 0; i <= upper; i++) {
             sum += i;
         }
    }
    pthread_exit(0);
}

void  *runner2(void *param)
{
    int             i, upper = atoi(param);
    sum2 = 0;

    if (upper >= 0) {
         for (i = 0; i <= upper; i++) {
             sum2 += pow(2, i);
         }
    }
    pthread_exit(0);
}
```

To execute:

```
% thrd2-posix  1
sum = 1
sum2 = 3

% thrd2-posix  20
sum = 210
```

```
sum2 = 2097151
```

Java Threads

- ✓ Java threads are managed by the JVM
- ✓ Typically implemented using the threads model provided by underlying OS

EXAMPLE:

-

↓SimpleThread.java:

```
public class SimpleThread extends Thread {
  private int countDown = 50;
  private static int threadCount = 0;
  private int threadNumber = ++threadCount;

  public SimpleThread() {
    System.out.println("Making " + threadNumber);
  }
  public void run() {
    while(true) {
      System.out.println("Thread " +
        threadNumber + "(" + countDown + ")");
      if(--countDown == 0) return;
    }
  }

  public static void main(String[] args) {
    for(int i = 0; i < 5; i++)
      new SimpleThread().start();
    System.out.println("All Threads Started");
  }
}
```

To execute:

```
% java SimpleThread
```

Python Threads

EXAMPLE:

-

SimpleThread.py:

```
#! /usr/bin/python
import threading
import datetime

class ThreadClass(threading.Thread):
    def run(self):
         now = datetime.datetime.now()
         print ("%s says Hello World at time: %s\n" %
         (self.getName(), now))
```

```
for i in range(2):
    t = ThreadClass()
    t.start()
```

To execute:

```
% SimpleThread.py
```

Threading Issues

· Semantics of fork() and exec()

Does **fork()** duplicate only the calling thread or all threads?

Answer: **No, it does not duplicate any of the threads.**

↓Example: thread-fork.c

```
main(int argc, char *argv[])
{
   pthread_create(&tid, &attr, runner1, 0);
   if ( fork() == 0){
       runner2();
       exit(0);
   }
   else {
      pthread_join(tid, NULL);
      printf("thread done\n");
      wait(NULL);
      printf("child  process done\n");
   }
}
void *runner1()
{
   int i;
   for(i=0;i<=7;i++) {
      printf("HI %d\n", i);
      sleep(i);
   }
}
void *runner2()
{
   int i;
   for(i=0;i<=7;i++) {
      printf("HI %c\n", i+'a');
      sleep(i);
   }
}
```

To execute:

```
% thread-fork
  HI 0
  HI 1
  HI a
  HI b
```

```
HI 2
HI c
HI d
HI 3
HI e
HI 4
HI f
HI 5
Etc...
```

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
- Signal is generated by particular event
- Signal is delivered to a process

Options:
- Deliver the signal to the thread to which the signal applies
- Deliver the signal to every thread in the process
- Deliver the signal to certain threads in the process
- Assign a specific thread to receive all signals for the process

In UNIX:
- The signal is delivered to ***EVERY*** **process** in the program
- The parent process of all threads gets the signal and ***NONE*** **of the threads** gets it.

↓Example 1: basic-signal.c

```
int
main(int argc, char *argv[])
{
   int i;
   signal(SIGINT, SIGhandler);

   printf("\nParent Process, PID: %d\n", getpid());
   for(;;){
      printf("HI %c\n", (i++) %26 +'A');
      sleep(1);
   }
}

void SIGhandler(int sig)
{
    signal(sig, SIGhandler);
    psignal(sig, "\nReceived SIGhandler1 signal");
}
```

To execute:

```
% basic-signal

 Type: CTRL-C
OR
 Type: CTRL-\ to exit
```

↓Example 2: fork-signal.c

```
main(int argc, char *argv[])
{
   signal(SIGINT, SIGhandler);
   if (fork() == 0){
            printf("\nChild Process1, PID: %d\n", getpid());
            for (;;) pause();
   }
   if (fork() == 0){
        signal(SIGINT, SIG_IGN);
            printf("\nChild Process2, PID: %d\n", getpid());
            for (;;) pause();
   }
   printf("\nParent Process, PID: %d\n", getpid());
   for (;;) pause();
}
void SIGhandler(int sig)
{
    printf("\nProcess, PID: %d\n", getpid());
    signal(sig, SIGhandler);
    psignal(sig, "\nReceived SIGhandler signal");
}
```

To execute:

```
%   fork-signal

 Type: CTRL-C
OR
 Type: CTRL-\ to exit
```

↓Example 3: thread-signal.c

```
int
main(int argc, char *argv[])
{

   pthread_t       tid1;
   pthread_t       tid2;
   pthread_attr_t  attr;
   int i;

   signal(SIGINT, SIGhandler0);
   pthread_attr_init(&attr);
   pthread_create(&tid1, &attr, runner1, 0);
   pthread_create(&tid2, &attr, runner2, 0);

   for(;;){
      printf("Main Thread %c%c\n", (i++)%26 +'a', (i)%26 +'A');
      sleep(1);
   }
}

void *runner1()
{
   int i=0;
   signal(SIGINT, SIGhandler1);

   for(;;){
      printf("thread 1 %c\n", (i++)%26 +'a');
      sleep(1);
```

```
      }
   }

   void *runner2()
   {
      int i=0;
      signal(SIGINT, SIGhandler2);

      for(;;){
         printf("THREAD2  %c\n", (i++)%26 +'A');
         sleep(1);
      }
   }

   void SIGhandler0(int sig)
   {
       signal(sig, SIGhandler0);
       psignal(sig, "\nReceived SIGhandler0 signal");
   }
   void SIGhandler1(int sig)
   {
       signal(sig, SIGhandler1);
       psignal(sig, "\nReceived SIGhandler1 signal");
   }
   void SIGhandler2(int sig)
   {
       signal(sig, SIGhandler2);
       psignal(sig, "\nReceived SIGhandler2 signal");
   }
```

To execute:

```
   %  thread-signal

    Type: CTRL-C
OR
    Type: CTRL-\ to exit
```

↓Example 4: fork-thread-signal.c

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

      pthread_attr_init(&attr);

      signal(SIGINT, SIG_IGN);
      if ( fork() == 0){
          childrunner();
          exit(0);
      }
      else {
         pthread_create(&tid, &attr, threadrunner, 0);
         pthread_join(tid, NULL);
         printf("thread done\n");
         wait(NULL);
         printf("child  process done\n");
      }
   }
```

```
void childrunner()
{
   int i=0;
   signal(SIGINT, SIGhandlerChild);
   for(;;){
      printf("CHILD %c\n", (i++)%26 + 'A');
      sleep(i);
   }
}
void *threadrunner()
{
   int i=0;
   signal(SIGINT, SIGhandlerThread);
   for(;;){
      printf("thread %c\n", (i++)%26 +'a');
      sleep(i);
   }
}
void SIGhandlerChild(int sig)
{
    signal(sig, SIGhandlerChild);
    psignal(sig, "Received  SIGhandlerChild signal");
    printf("Child PID: %d\n", getpid());
}
void SIGhandlerThread(int sig)
{
    signal(sig, SIGhandlerThread);
    psignal(sig, "Received  SIGhandlerThread signal");
    printf("Thread PID: %d\n", getpid());
}
```

To execute:

```
%   fork-thread-signal

 Type: CTRL-C
OR
 Type: CTRL-\ to exit
```

↓Example 5: signal.py

```
#! /usr/bin/python
import os
import signal
import time
import sys

def INThandle (signum, frame ):
        print ("Got SIGINT, PID: ", os.getpid())
        print( "SIGINT number:", signum)
        pass

signal.signal( signal.SIGINT, INThandle )
childpid = os.fork()
if childpid == 0:
        print("I am child, my PID: ", os.getpid())
        signal.pause()
        print("Child: got SIGINT signal")
else:
        print("I am the parent, my PID: ", os.getpid())
```

```
print("My child PID: ", childpid)
signal.pause()
print("Parent: got SIGINT signal")
```

To execute:

```
%  signal.py

 Type: CTRL-C
OR
 Type: CTRL-\ to exit
```

Thread Pools

Create a number of threads in a pool where they await work

Advantages:

Usually slightly **faster to service** a request with an existing thread than create a new thread
Allows the number of threads in the application(s) to be **bound to the size** of the pool

---

EXAMPLES of Threaded Servers:

NOTE:
If the default port 10117 is used, use any other port as an argument to the server,
For example:
% EchoServerThread 10321

↓EchoServerThread.c:

```
Int main(void)
{
    int             len, listenfd, connfd;
    struct sockaddr_in servaddr, cliaddr;
    char            buff[512];
    int             nread;
    pthread_t       tid;
    pthread_attr_t  attr;

    if ( (listenfd = socket(AF_INET, SOCK_STREAM, 0)) == -1 ) {
         printf ("error\n");
         exit(-1);
    }

    bzero(&servaddr, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
    servaddr.sin_port = htons(10117);

    if ( bind(listenfd, (SA *) & servaddr, sizeof(servaddr)) < 0) {
            printf("can not bind\n");
```

```
            exit(-1);
        }

    listen(listenfd, 0);

    pthread_attr_init(&attr);

    for (;;) {
         len = sizeof(cliaddr);
         connfd = accept(listenfd, (SA *) & cliaddr, &len);
        printf("connection from: %s:%d\n", inet_ntoa(cliaddr.sin_addr),
        ntohs(cliaddr.sin_port));
         pthread_create(&tid, &attr, &EchoToClient, (void *) connfd);
        }
}

void  *EchoToClient(void *sockfd)
{
    int             nread;
    char            buffer[512];
    for (;;) {
         nread = read((int) sockfd, buffer, 512);
         if (nread == 0)
             pthread_exit(0);
             // exit(0);
         write((int) sockfd, buffer, nread);
    }
}
```

To execute:

```
% EchoServerThread

✓In another window for the same host:
  % EchoClientFork    localhost     10117

✓In another window for the same host:
  % EchoClientFork    localhost     10117

✓Repeat for any number of other windows.
```

↓EchoServerThreadJava:

```
class EchoToClient extends Thread {
  private Socket socket;
  private BufferedReader in;
  private PrintWriter out;

  public EchoToClient(Socket s) throws IOException {
    socket = s;
    in = new BufferedReader( new InputStreamReader( socket.getInputStream()));
   out = new PrintWriter( new BufferedWriter(
              new OutputStreamWriter(socket.getOutputStream())), true);
    start();
  }
  public void run()  {
    try {
      while (true) {
        String str = in.readLine();
        if (str.length() == 0 )
```

```
                System.exit(0);
            else
            out.println(str);
          }
        } catch(IOException e) {
        }
      }
    }

    public class EchoServerThreadJava {
      public static void main(String[] args)
          throws IOException {
        ServerSocket s = new ServerSocket(10117);
        try {
          while(true) {
            Socket socket = s.accept();
            try {
              new EchoToClient (socket);
            } catch(IOException e) {

            }
          }
        } catch(IOException e) {
        }
      }
    }
```

To execute:

```
% java EchoServerThreadJava
```

✓ In another window for the same host:

```
% EchoClientFork    localhost     10117
```

✓ In another window for the same host:

```
% EchoClientFork    localhost     10117
```

✓ Repeat for any number of other windows.

↓EchoServerThread.py:

```
#! /usr/bin/env python
import socket
import sys
import os
import threading

def EchoThread(conn,addr):
    while (1):
         data = conn.recv(1000000)
         if not data:
             break
         print 'received', len(data), 'bytes: ', data
         conn.send(data)
    conn.close()

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
host = ''
port = 10117
s.bind((host, port))
s.listen(1)
```

```
while (1):
    conn, addr = s.accept()
    print 'client is at', addr
    threading.Thread(target=EchoThread,args=(conn,addr)).start()
```

To execute:

```
% EchoServerThread.py
```

✓ In another window for the same host:

```
% EchoClientFork    localhost     10117
```

✓ In another window for the same host:

```
% EchoClientFork    localhost     10117
```

✓ Repeat for any number of other windows.

CPU Scheduling

★ Basic Concepts

CPU–I/O Burst Cycle:

Process execution consists of a cycle of **CPU execution** & **I/O wait**

CPU Scheduler

- Selects from among the processes in memory that are ***ready*** to execute, and

allocates the CPU to one of them

- CPU scheduling decisions may take place when a process:
  - Switches from **running** *to* **waiting** state
  - Switches from **running** *to* **ready** state
  - Switches from **waiting** *to* **ready**
  - **Terminates**

Dispatcher

- Dispatcher module gives control of the CPU to the process selected by the scheduler;
- Dispatch **latency** – time it takes for the dispatcher to **stop** one process and **start** another running

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
- **Waiting time** – amount of time a process has been waiting in the ready queue
- **Response time** – amount of time from when a request was submitted until the **first response** is produced.

Scheduling Algorithm Optimization Criteria

- Maximize **CPU utilization**
- Maximize **throughput**
- Minimize **turnaround time**
- Minimize **waiting time**
- Minimize **response time**

★ First-Come, First-Served (FCFS) Scheduling

Process	Burst Time
P_1	24
P_2	3
P_3	3

The ***Gantt Chart*** for the schedule is:

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

★ 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

Example of SJF

Process	Burst Time
P1	6
P2	8
P3	7
P4	3

SJF scheduling Gantt chart

Average waiting time = (3 + 16 + 9 + 0) / 4 = 7

Determining Length of Next CPU Burst

To estimate the length of next CPU burst using the length of previous CPU bursts

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

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

Prediction of the Length of the Next CPU Burst (α = 0.5)

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

★ Priority Scheduling

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

★ Round Robin (RR)

- Each process gets a small unit of CPU time (***time quantum***), 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.
No process waits more than **(*n*-1)*q*** time units.

- Performance
  - ***q*** large Þ **FIFO**
  - ***q*** small Þ *q* must be large with respect to **context switch**, otherwise overhead is too high

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

★ Multilevel Queue

- **Ready queue** is partitioned into separate queues:
  - **foreground** (interactive)
  - **background** (batch)
- 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;
    E.g., **80%** to foreground in RR and **20%** to background in FCFS

Multilevel Queue Scheduling

Example of Multilevel Feedback Queue

★ Multiple-Processor Scheduling

- CPU scheduling more complex when multiple CPUs are available
- **Symmetric multiprocessing (SMP)** – each processor is self-scheduling,

all processes in common ready queue, **or** each has its own private queue of ready processes

Process Synchronization

Basic Concepts

Objectives

- To introduce the critical-section problem, to ensure the ***consistency of shared data***
- To introduce the concept of an ***atomic transaction*** and describe mechanisms to ensure atomicity

Background

- Concurrent access to shared data may result in data inconsistency
- For Example:

  To provide a solution to the consumer-producer problem that **fills all** the buffers:
  We can do so by having an integer **count** that keeps track of the number of full buffers:
  - Initially, **count := 0**.
  - It is **incremented** by the **producer** after it produces a new buffer and
  - It is **decremented** by the **consumer** after it consumes a buffer.

Producer

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

Consumer

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

Race Condition

- count++ could be implemented as

```
register1 = count
register1 = register1 + 1
count = register1
```

- count-- could be implemented as

register2 = count
register2 = register2 - 1
count = register2

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

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

Mutual Exclusion

If process P_i is executing in its critical section, then ***no other processes*** can be executing in their critical sections

Peterson's Solution

- **Two process** solution
- The two processes **share two variables**:
  - ✓ int **turn**;
  - ✓ boolean **ready[2];**
- The variable **turn** indicates whose turn is to enter the critical section.
- The **ready** array is used to indicate if a process is ready to enter the critical section.
  ready[i] = true implies that process P_i is ready!

Algorithm for Process P_i

```
do {
        ready[i] = TRUE;
        turn = j;
        while (ready[j] && turn == j)  ;

                critical section

        ready[i] = FALSE;

                remainder section

} while (TRUE);
```

Examples:

- **twothreads.c**

```
int
main(int argc, char *argv[])
{
```

```
   pthread_t       tid0;
   pthread_t       tid1;
   pthread_attr_t  attr;
   int             i;

   pthread_attr_init(&attr);

   pthread_create(&tid0, &attr, runner0, 0);
   pthread_create(&tid1, &attr, runner1, 0);

   pthread_join(tid0, NULL);
   printf("thread 0 done\n");
   pthread_join(tid1, NULL);
   printf("thread 1 done\n");
}

void  *runner0()
{
   int             i;
   printf("Thread 0 Started\n");
   for (;;) {
      for (i = 1; i <= 3; i++) {
         printf("HI %d\n", i);
         sleep(1);
      }
   }
}

void *runner1()
{
   int             i;
   printf("Thread 1 Stareted\n");
   for (;;) {
      for (i = 0; i <= 2; i++) {
         printf("HI %c\n", i + 'a');
         sleep(1);
      }
   }
}
```

To execute:

% twothreads

peterson.c

```
int             turn;
int             ready[2];

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

   pthread_t       tid0;
   pthread_t       tid1;
   pthread_attr_t  attr;
   int             i;

   pthread_attr_init(&attr);

   pthread_create(&tid0, &attr, runner0, 0);
```

```
   pthread_create(&tid1, &attr, runner1, 0);

   pthread_join(tid0, NULL);
   printf("thread 0 done\n");
   pthread_join(tid1, NULL);
   printf("thread 1 done\n");
}

void *runner0()
{
   int             i;
   printf("Thread 0 Started\n");
   for (;;) {
      ready[0] = TRUE;
      turn = 1;
      while (ready[1] && turn == 1);
      for (i = 1; i <= 3; i++) {
         printf("HI %d\n", i);
         sleep(1);
      }
      printf("\n");
      ready[0] = FALSE;
   }
}

void *runner1()
{
   int             i;
   printf("Thread 1 Stareted\n");
   for (;;) {
      ready[1] = TRUE;
      turn = 0;
      while (ready[0] && turn == 0);
      for (i = 0; i <= 2; i++) {
         printf("HI %c\n", i + 'a');
         sleep(1);
      }
      printf("\n");
      ready[1] = FALSE;
   }
}
```

To execute:

% peterson

Synchronization Hardware

- Many systems provide hardware support for critical section code
- Modern machines provide special **atomic hardware instructions** (non-interruptible).

Examples:

✓ **TestAndSet** memory value.

✓ **Swap** contents of two memory words.

Solution to Critical-section Problem Using Locks

do {

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

✓ TestAndSet Instruction

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

Solution using TestAndSet

Shared boolean variable **lock := FALSE;**

.

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

-

-

✓ Swap Instruction

```
void Swap (boolean *a, boolean *b)
{
    boolean temp = *a;
    *a = *b;
    *b = temp:
}
```

Solution using Swap

Shared Boolean variable **lock := FALSE**;
Each process has a **local** Boolean variable **key**

```
do {
    key = TRUE;
    while ( key == TRUE) Swap (&lock, &key );
        critical section
    lock = FALSE;
```

remainder section

} while (TRUE);

Semaphore

Semaphore ***S*** – integer variable
Two standard operations modify ***S***: **wait (S) & signal (S)**

Originally called (by the late Dijkstra): ***P & V***

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

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

Mutual Exclusion

Semaphore **mutex** = 1;

do {

wait (mutex);
Critical Section
signal (mutex);

Remainder Section

} while (TRUE);

Semaphore implementation must guarantee that **no two processes** can execute **wait ()** and **signal ()** on the same semaphore at the same time.

Example:

ThreadMutex.c

```
void            *functionC();
pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
pthread_t       thread1, thread2;
int             counter = 0;
int             lockone = 0;
int             lockall = 0;
int             nolock = 0;
```

```
main(int argc, char *argv[])
{
  int             rc1, rc2;
  int             mode;
  int             index = 1;


  if (argc == 1) {
       printf("Usage is ThreadMutex [0|1|2]\n");
       printf("0 - no lock\n1 - lock one by one\n2 - lock all\n");
       exit(0);
  }
  mode = atoi(argv[1]);
  if (mode == 0)        nolock = 1;
  else if (mode == 1)   lockone = 1;
  else if (mode == 2)   lockall = 1;

  pthread_create(&thread1, NULL, &functionC, (void *) index++);
  pthread_create(&thread2, NULL, &functionC, (void *) index++);

  pthread_join(thread1, NULL);
  pthread_join(thread2, NULL);
  printf("Counter value: %d\n", counter);

  exit(0);
}

void *functionC( void *arg)
{
  int             i;
  int myindex = (int) arg;
  if (lockall) {

       pthread_mutex_lock(&mutex1);

      for (i = 1; i <= 1000000; i++) {

          counter++;
          if (i % 100000 == 0)
             printf(" ThreadIndex (%d), ThreadID (%u): i is: %d and counter is: %d \n",
                      myindex , pthread_self(), i, counter);
       }

       pthread_mutex_unlock(&mutex1);
  }
  else
  if (lockone) {
       for (i = 1; i <= 1000000; i++) {

          pthread_mutex_lock(&mutex1);

          counter++;
          if (i % 100000 == 0)
             printf(" ThreadIndex (%d), ThreadID (%u): i is: %d and counter is: %d \n",
                      myindex , pthread_self(), i, counter);

          pthread_mutex_unlock(&mutex1);
       }
  } else
  if (nolock) {
       for (i = 1; i <= 1000000; i++) {
```

```
            counter++;
            if (i % 100000 == 0)
               printf(" ThreadIndex (%d), ThreadID (%u): i is: %d and counter is: %d \n",
                        myindex , pthread_self(), i, counter);
        }
    }
}
```

To execute:

% ThreadMutex 0 (or 1 or 2)

Semaphore Implementation with *no busy waiting*

- With each semaphore there is an associated **waiting queue**.
- Each semaphore has two data items:
  - ✓ **value** (of type integer)
  - ✓ **pointer** to next process in the waiting queue.
- 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.

Implementation of wait:

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

Implementation of signal:

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

P_0	P_1
wait (S);	**wait (Q);**
wait (Q);	**wait (S);**

.	.
.	.
.	.
signal (S);	signal (Q);
signal (Q);	signal (S);

Examples:

EX 1: waitForAll.c (use lock and unclock mutex to protect count).

```
sem_t           FinishSemaphore;
pthread_mutex_t CountMutex;

int MAXThreads;
int RandTime;
int             GoneCount;

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

    int i;
    pthread_t       One_thread;

    MAXThreads =atoi(argv[1]);
    RandTime = atoi(argv[2]);
    srand((unsigned) time(0) + getpid());
    sem_init(&FinishSemaphore, 0, 0);
    pthread_mutex_init(&CountMutex, NULL);

    for (i=1; i <= MAXThreads; i++) {
        pthread_create(&One_thread, NULL, OneThread, (void *) i);
    }
    sem_wait(&FinishSemaphore);
    printf("ALL Threads Are Gone\n");
    exit(0);
}

void *OneThread (void *arg)
{
    int             Index = (int) arg;
    int sleepTime;

    sleepTime = getRand(RandTime);
    sleep(sleepTime);
    pthread_mutex_lock(&CountMutex);
          GoneCount++;
    pthread_mutex_unlock(&CountMutex);
    printf("\n<<< Thread%d is Gone after sleeping %d seconds\n", Index, sleepTime);
    if (GoneCount == MAXThreads)
        sem_post(&FinishSemaphore);
}

int getRand(int N) {
    int             RVal;
    RVal = (rand() % (N));
    if (RVal == 0) RVal++;
    return (RVal);
}
```

To execute:

% waitForAll 10 10

Ex 2: wait4all.c (use wait and post semaphore to protect count).

```
sem_t           FinishSemaphore;
sem_t           CountSemaphore;

int MAXThreads;
int RandTime;
int             GoneCount;
main(int argc, char *argv[])
{

    int i;
    pthread_t       One_thread;

    MAXThreads =atoi(argv[1]);
    RandTime = atoi(argv[2]);
    srand((unsigned) time(0) + getpid());
    sem_init(&FinishSemaphore, 0, 0);
    sem_init(&CountSemaphore, 0, 1);

    for (i=1; i <= MAXThreads; i++) {
        pthread_create(&One_thread, NULL, OneThread, (void *) i);
    }
    sem_wait(&FinishSemaphore);
    printf("ALL %d Threads Are Gone\n", GoneCount);
    exit(0);
}

void *OneThread (void *arg)
{
    int             Index = (int) arg;
    int sleepTime;

    sleepTime = getRand(RandTime);
    sleep(sleepTime);
    sem_wait(&CountSemaphore);
         GoneCount++;
    sem_post(&CountSemaphore);
    printf("\n<<< Thread%d is Gone after sleeping %d seconds\n", Index, sleepTime);
    fflush(stdout);
    if (GoneCount == MAXThreads)
        sem_post(&FinishSemaphore);
}
```

To execute:

% wait4all 10 10

EX 3: wait4allfork.c (fork, shared memory, wait and post semaphore).

```
sem_t           *FinishSemaphore;
sem_t           *CountSemaphore;
int             *GoneCount;

int MAXChildren;
```

```
int RandTime;

int             segment_id1;
int             segment_id2;
int             segment_id3;


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

    int i;
    pthread_t       One_thread;

    signal(SIGCHLD, SIGhandler);

    segment_id1 = shmget(IPC_PRIVATE, sizeof(sem_t), S_IRUSR | S_IWUSR);
    FinishSemaphore = (sem_t *) shmat(segment_id1, NULL, 0);

    segment_id2 = shmget(IPC_PRIVATE, sizeof(sem_t), S_IRUSR | S_IWUSR);
    CountSemaphore = (sem_t *) shmat(segment_id2, NULL, 0);

    segment_id3 = shmget(IPC_PRIVATE, sizeof(int), S_IRUSR | S_IWUSR);
    GoneCount = (int *) shmat(segment_id3, NULL, 0);

    MAXChildren =atoi(argv[1]);
    RandTime = atoi(argv[2]);

    sem_init(FinishSemaphore, 1, 0);
    sem_init(CountSemaphore, 1, 1);

    for (i=1; i <= MAXChildren; i++) {
        if (fork()==0) {
            OneChild(i);
        }
    }
    sem_wait(FinishSemaphore);
    printf("ALL %d Children Are Gone\n", GoneCount[0]);
    exit(0);
}

OneChild (int arg)
{
    int             Index = arg;
    int sleepTime;


    srand((unsigned) time(0) + getpid());

    sleepTime = getRand(RandTime);

    sem_wait(CountSemaphore);
        ++GoneCount[0];
    sem_post(CountSemaphore);
    printf("\n<<< Child%d is Gone after sleeping %d seconds, GoneCount %d\n",
            Index, sleepTime, GoneCount[0]);
    fflush(stdout);
    if (GoneCount[0] == MAXChildren) {
        printf("LAST Child to exit\n");
        sem_post(FinishSemaphore);
    }
    exit(0);
}
```

```
void SIGhandler(int sig)
{
    signal(sig, SIGhandler);
}
```

To execute:

% wait4allfork 10 10

➔ Classical Problems of Synchronization

I. Bounded-Buffer
II. Readers-Writers
III. Dinning-Philosophers

I. 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**.

The producer process

```
do {

    produce an item

    wait (empty);
    wait (mutex);
        add the item to the buffer
    signal (mutex);
    signal (full);

} while (TRUE);
```

The consumer process

```
do {

    wait (full);
    wait (mutex);
        remove an item from buffer
    signal (mutex);
```

signal (empty);

consume the item

} while (TRUE);

Example:

ProducerConsumerSemaphore.c

```
#define BUFFER_SIZE 10

/* The mutex lock */
pthread_mutex_t mutex;

/* the semaphores */
sem_t           full;
sem_t           empty;


int             b[BUFFER_SIZE];
int             psleep, csleep;

main()
{
    pthread_t       producer_thread;
    pthread_t       consumer_thread;
    void           *producer();
    void           *consumer();

    psleep = atoi(argv[1]);
    csleep = atoi(argv[2]);

   /* Create the mutex lock */
   pthread_mutex_init(&mutex, NULL);

   /* Create the full semaphore and initialize to 0 */
   sem_init(&full, 0, 0);

   /* Create the empty semaphore and initialize to BUFFER_SIZE */
   sem_init(&empty, 0, BUFFER_SIZE);

    pthread_create(&consumer_thread, NULL, consumer, NULL);
    pthread_create(&producer_thread, NULL, producer, NULL);
    pthread_join(consumer_thread, NULL);
}

int in = 0;
void add_buffer(int i)
{
     b[in] = i;
     in = (in + 1) % BUFFER_SIZE;
}

int out = 0;
int get_buffer()
{
     int value;
     value = b[out];
     out = (out + 1) % BUFFER_SIZE;
     return value;
```

```
}
```

producer()

```
{
    int i = 0;
    int ranvalue;
    for (;;) {
       ranvalue = 1 + (rand() % psleep);
       sleep(ranvalue);

       sem_wait(&empty);      //acquire the empty lock

         pthread_mutex_lock(&mutex);     //acquire the mutex lock

             add_buffer(i);                // CRITICAL SECTION
             printf(">>put %d\n", i++);

         pthread_mutex_unlock(&mutex);   //release the mutex lock

       sem_post(&full);       //signal full


    }
    pthread_exit(NULL);
}
```

consumer()

```
{
int ranvalue;
    int  v;
    for (;;) {

       sem_wait(&full);                //aquire the full lock

           pthread_mutex_lock(&mutex);     //aquire the mutex lock

               v = get_buffer();               //CRITICAL SECTION
               printf("<<got %d\n", v);

           pthread_mutex_unlock(&mutex);   //release the mutex lock

       sem_post(&empty);               //signal empty

       ranvalue = 1 + (rand() % csleep);
       sleep(ranvalue);
    }
    pthread_exit(NULL);
}
```

To execute:

```
% ProducerConsumerSemaphore   1   1
% ProducerConsumerSemaphore   5   5
% ProducerConsumerSemaphore   5   4
% ProducerConsumerSemaphore   4   5
% ProducerConsumerSemaphore   1   10
% ProducerConsumerSemaphore   10   1
```

II. 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
- Allow **multiple readers** to read at the same time.
- Only one **single writer** can access the shared data at the same time
- Shared Data
  - ✓ **Data** set
  - ✓ Semaphore **mutex** initialized to **1**
  - ✓ Semaphore **wrt** initialized to **1**
  - ✓ Integer **readcount** initialized to **0**

The Writer process

```
do {

        wait (wrt)

            writing is performed

        signal (wrt) ;

} while (TRUE);
```

The Reader process

```
do {

        wait (mutex) ;

            readcount ++ ;
            if (readcount == 1)       //First Reader, keep writers out
                    wait (wrt) ;

        signal (mutex)

            reading is performed

            wait (mutex) ;
                readcount  - - ;
                if (readcount  == 0)     //Last Reader, allow in writers
                        signal (wrt) ;
            signal (mutex) ;
```

} while (TRUE);

➔ Example:

WritersReaders.c

```
% WritersReaders  W R
  where
          W - Number of Writers' threads
          R -  Number of Readers threads
```

- **The Shared Buffer:**
  (Buffer[0] ... Buffer[9])

- **The Writer Thread**:
  Sleeps a random time T in the range 0-9.
  If Buffer [T] = 9 then exits,
  Otherwise *increments* Buffer [T].

- **The Reader Thread**:
  Sleeps a random time T in the range 0-9.
  Prints the contents of the Buffer.
  If Buffer [T] = 9 then exits.

```
#define BUFFER_SIZE 10
int             Buffer[BUFFER_SIZE];
pthread_mutex_t mutex;
pthread_mutex_t wrt;
int             readcount = 0;
int             Nwriters = 0;
int             Nreaders = 0;

main(int argc, char *argv[])
{
    pthread_t       writer_thread;
    pthread_t       reader_thread;
    void           *writer();
    void           *reader();
    int             i;


    Nwriters = atoi(argv[1]);
    Nreaders = atoi(argv[2]);

    pthread_mutex_init(&mutex, NULL);
    pthread_mutex_init(&wrt, NULL);

    for (i = 1; i <= Nwriters; i++) {
        pthread_create(&writer_thread, NULL, writer, (void *) i);
    }
    for (i = 1; i <= Nreaders; i++) {
        pthread_create(&reader_thread, NULL, reader, (void *) i);
    }
    pause();
}

void           *
```

```
writer(void *ID)
{
    int             ranvalue;
    int             threadID = (int) ID;
    int             i;
    printf("STARING WRITER THREAD %d\n", threadID);

    srand((unsigned) time(0) + threadID);
    for (;;) {
        ranvalue = (rand() % 10);
        sleep(ranvalue);

        pthread_mutex_lock(&wrt);
        if (Buffer[ranvalue] == 9) {
            printf("Writer %d: ...EXIT... since Buffer[%d] = %d", threadID, ranvalue,
Buffer[ranvalue]);
            Nwriters--;
            printf(" --- NUMBER of Remaining WRITERS  ---> %d\n", Nwriters);

            if (Nwriters == 0 && Nreaders == 0) {
                printf("\n\nLAST thread (Writer %d) -- FINAL Buffer Content: ", threadID);

                for (i = 0; i <= 9; i++)
                    printf("%d ", Buffer[i]);


                pthread_mutex_unlock(&wrt);
                exit(0);
            } else {
                pthread_mutex_unlock(&wrt);
                pthread_exit(NULL);
            }
        } else {
            ++Buffer[ranvalue];
            pthread_mutex_unlock(&wrt);
        }
    }
}

void           *
reader( void *ID)
{
    int             ranvalue;
    int             threadID = (int) ID;
    int             i;

    srand((unsigned) time(0) + threadID);
    printf("STARTING READER THREAD %d\n", threadID);

    for (;;) {
        ranvalue = (rand() % 10);
        sleep(ranvalue);

        pthread_mutex_lock(&mutex);
        readcount++;
        if (readcount == 1)
            pthread_mutex_lock(&wrt);
        pthread_mutex_unlock(&mutex);
        DisplayBuffer(threadID);

        if (Buffer[ranvalue] == 9) {
            printf("...EXIT... since Buffer[%d] = %d", ranvalue, Buffer[ranvalue]);
```

```
            pthread_mutex_lock(&mutex);
            readcount--;
            if (readcount == 0)
                pthread_mutex_unlock(&wrt);

            Nreaders--;
            pthread_mutex_unlock(&mutex);
            printf(" ---  NUMBER of Remaining READERS ---> %d\n", Nreaders);

            if (Nwriters == 0 && Nreaders == 0) {
                printf("\n\nLAST thread (Reader %d) -- FINAL Buffer Content: ", threadID);
                DisplayBuffer(threadID);
                exit(0);
            } else {
                pthread_exit(NULL);
            }
        } else {
            pthread_mutex_lock(&mutex);
            readcount--;
            if (readcount == 0)
                pthread_mutex_unlock(&wrt);
            pthread_mutex_unlock(&mutex);
        }
    }
}

DisplayBuffer(int threadID)
{
    char            str[100];
    char            str2[100];
    int             i;
    bzero(str, 1);
    bzero(str2, 1);
    sprintf(str, "Reader %d:( ", threadID);
    for (i = 0; i <= 9; i++) {
        sprintf(str2, "%d ", Buffer[i]);
        strcat(str, str2);
    }
    printf("%s) ", str);
    fflush(stdout);
}
```

To execute:

```
% WritersReaders  20   10
```

III. Dining-Philosophers Problem

Shared data

- Bowl of rice (data set)
- Semaphore **chopstick [5]** initialized to **1**

Philosopher i :

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

            Eat

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

            Think

} while (TRUE);
```

Examples:

DinningPhilosopher.c

```
#define   N   5
pthread_mutex_t   mymutex[N];

void            Pickup(int);
void            Putdown(int);

philosopher(void *arg)
{
ll
k,  int             sleeptime;
    int             i = (int *)arg;
    srand((unsigned) time(0));

    while (1) {
         sleeptime = 1 + rand() % N;
```

```
        printf("\nPhilosopher  %d is THINKING for %d seconds\n", i, sleeptime);
        sleep(sleeptime);

        Pickup(i);
        sleeptime = 1 + rand() % N;
        printf("\n   >> Philosopher  %d is EATING for %d seconds\n", i, sleeptime);
        sleep(sleeptime);
        Putdown(i);
    }
}

Pickup(int i)
{
        pthread_mutex_lock(&mymutex[(i) % N]);
usleep(slowdown);
        pthread_mutex_lock(&mymutex[(i + 1) % N]);
}

Putdown(int i)
{
        pthread_mutex_unlock(&mymutex[(i) % N]);
        pthread_mutex_unlock(&mymutex[(i + 1) % N]);
}

main(int argc, char *argv[])
{
    int             rc, i;
    int             status;

    pthread_t       threads[N];
    pthread_attr_t  attr;

   if (argc > 1) slowdown = atoi(argv[1]);
    for (i = 0; i < N; i++) {
        pthread_mutex_init(&mymutex[i], NULL);
    }
    pthread_attr_init(&attr);
    pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
    for (i = 0; i < N; i++) {
        pthread_create(&threads[i], NULL, philosopher, (void *) i);
    }
    pthread_exit(NULL);
}
```

To execute:

```
% DinningPhilosopher
```

To casuse deadlock:

```
% DinningPhilosopher 5
```
// slowdown picking the 2nd fork

Atomic Transactions

- A collection of instructions or read /write operations that performs ***single logical function***.
- Terminated by ***commit*** (transaction successful) or ***abort*** (transaction failed) operation
- **Aborted** transaction must be rolled back to ***undo*** any changes it performed

Types of Storage Media

- **Volatile storage** – information stored does not survive system crashes
  Example: main memory, cache
- **Nonvolatile storage** – Information usually survives crashes
  Example: disk and tape
- **Stable storage** – Information never lost
  Not actually possible, so approximated via **replication** to devices with independent failure modes

Log-Based Recovery

- Record to stable storage information about all modifications by a transaction
- Each record describes single transaction write operation including:
  - ✓ Transaction name
  - ✓ Data item name
  - ✓ Old value
  - ✓ New value
- **<T_i starts>** written to log when transaction T_i starts
- **<T_i commits>** written when T_i commits

Log-Based Recovery Algorithm

- Using the log, system can handle any volatile memory errors
  - ✓ **Undo(T_i)** restores value of all data updated by T_i
  - ✓ **Redo(T_i)** sets values of all data in transaction T_i to new values
- Undo(T_i) and redo(T_i) must be ***idempotent***
  i.e., multiple executions must have the same result as one execution
- If system fails, restore state of all updated data via log
  - ✓ If log contains **<T_i starts>** without **<T_i commits>**, ***Undo(T_i)***
  - ✓ If log contains **<T_i starts>** and **<T_i commits>**, ***Redo(T_i)***

Checkpoints

- Checkpoints shorten log and recovery time.
- **Checkpoint scheme**:
  - ✓ Output all log records currently in volatile storage to stable storage
  - ✓ Output all modified data from volatile to stable storage
  - ✓ Output a log record ***<checkpoint>*** to the log on stable storage
- **Recovery** only includes Ti, such that Ti started executing before the most recent checkpoint.

Concurrent Transactions

Transaction execution must be equivalent to serial execution – **serializability**
Concurrency-control algorithms provide serializability

Consider two data items ***A, B*** and Transactions T_0, T_1

Schedule 1: ***T_0* then *T_1***

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

Schedule 2: **Concurrent *Serializable* Schedule**

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

Locking Protocol to ensure serializabilty

- Ensure serializability by associating **lock** with each data item
- **Locks**
  - ✓ **Shared**: T_i has share lock on item Q, then T_i can: **read** Q **but not write** Q
  - ✓ **Exclusive**: T_i has exclusive lock on item Q, then T_i can: **read** Q **and write** Q
- Require every transaction on item Q acquire appropriate lock using:

Two-phase Locking Protocol:

Each transaction issues lock and unlock requests in ***two phases***
✓ ***Growing*** – obtaining locks
✓ ***Shrinking*** – releasing locks

Deadlocks

➔Definition & Basics:

Deadlock:

A set of blocked processes each ***holding*** a resource ***& waiting*** to acquire a resource held by another process in the set.

Ex1

System has 2 disk drives
Processes P1 and P2 each hold one disk drive and each needs another one

Ex2

Semaphores **A** and **B**, initialized to 1

P0	**P1**
wait (A);	wait(B)
wait (B);	wait(A)

Deadlock can arise if *4 conditions* hold simultaneously:

1. ***Mutual exclusion:*** only one process at a time can use a resource
2. ***Hold and Wait:*** holding at least one resource and is waiting to acquire additional resources held by others
3. ***No preemption:*** a resource can be released only voluntarily by the process holding it.
4. ***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.

➔Resource-Allocation Graph

A set of **vertices** ***V*** and a set of **edges** ***E:***

- ***V*** is partitioned into two types:
  - ***P*** = $\{P_1, P_2, \ldots, P_n\}$, the set consisting of all the ***Processes*** in the system
  - ***R*** = $\{R_1, R_2, \ldots, R_m\}$, the set consisting of all ***Resource*** types in the system

- ***E*** is partitioned into two types:
  - ***request*** edge – directed edge P_i ® R_j
  - ***assignment*** edge – directed edge R_j ® P_i

Resource Allocation Graph with *No Deadlock (no cycle)*

Resource Allocation Graph with *Deadlock (cycle)*

Resource Allocation Graph with *No Deadlock (cycle)*

Basic Facts

- If graph contains **no cycles Þ no deadlock**
- If graph contains a cycle Þ
  - if only **one** instance per resource type, then **deadlock**
  - if **several** instances per resource type, ***possibility* of deadlock**

➜Deadlock Prevention

Restrain the ways request can be made.
Any of the following polices will ***prevent*** deadlock:

1. **Hold and Wait** –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.

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

➔Deadlock Avoidance

Requires that the system has ***a priori information*** available.

- Simplest model requires that each process declare the ***maximum number*** of
  resources of each type that it may need.
- The deadlock-avoidance algorithm dynamically examines ***resource-allocation state*** to
  ensure there is no ***circular-wait***.

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 $<P1, P2, \ldots, Pn>$ of ALL processes
  in the systems such that:
  For each ***Pi*** , the resources that ***Pi*** can still request can be satisfied by currently available resources
  + resources held by all the ***Pj*** , with ***j < i***
- If a system is in **safe state Þ no deadlocks**
- If a system is in **unsafe state Þ possibility of deadlock**
- **Avoidance Þ** ensure that a system will **never enter** an **unsafe state**.

Safe, Unsafe & Deadlock State

➔Avoidance algorithms

-

❖ Resource-Allocation Graph Scheme:

(Used for ***Single*** instance of resource types)

Resources must be ***claimed a priori*** in the system.

- ***Claim edge*** P_i ® R_j indicated that process P_i **may request** resource R_j; represented by a **dashed** line
- Claim edge converts to ***Request edge*** when a process requests a resource
- Request edge converted to an ***Assignment edge*** when the resource is **allocated** to the process
- When a resource is released by a process, assignment edge reconverts to a ***Claim edge***

Resource-Allocation Graph

-

-

Unsafe State In Resource-Allocation Graph

-

Resource-Allocation Graph Algorithm

Suppose that process P_i requests a resource R_j

The request can be granted ***only if*** :

converting the request edge to an assignment edge

does not result in the formation of a ***cycle*** in the resource allocation graph

-

- **Banker's Algorithm (by Dijkstra)**

  (Used for ***Multiple*** instances of a resource type)

  Each process must a priori claim ***maximum*** use.

○ Data Structures

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 x m* **matrix**.
If ***Max* [*i,j*] = *k***, then process P_i may request ***at most k*** instances of resource type R_j

✓ **Allocation**: *n* x *m* **matrix**.
If **Allocation [*i,j*] = *k*** then P_i is ***currently allocated k*** instances of R_j

✓ **Need**: *n* x *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 index *i* such that:
   Finish [*i*] = *false* AND $Need_i$ £ *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, otherwise it is **unsafe**

The Algorithm requires $m \times n^2$ operations to detect whether the system is in deadlocked state

○ Resource-Request for a Process

Request = request vector for process P_i

If $Request_i[j] = k$ then process P_i wants k instances of resource type R_j

- **If** $Request_i > Need_i$
  Raise error condition, since process has exceeded its maximum claim.

- **If** $Request_i > Available$
  P_i ***must wait***, since resources are not available

- **If** $Request_i <= Available$
  Pretend to allocate requested resources to P_i by modifying the state as follows:

$$Available = Available - Request_i$$
$$Allocation_i = Allocation_i + Request_i$$
$$Need_i = Need_i - Request_i$$

- *If state is* ***safe*** Þ *the resources are* ***allocated*** *to* ***Pi***
- *If state is* ***unsafe*** Þ ***Pi*** *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 B C
  10 5 7

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 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

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

Example:

P_1 Request (1,0,2)

- Check that **Request £ Available** that is**, (1,0,2) £ (3,3,2) Þ 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 1	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 < P_1, P_3, P_4, P_0, P_2> **satisfies safety** requirement

↓Deadlock Detection

Allow system to enter deadlock state then: **Detect & Recover**.

- **Single Instance of Each Resource Type**
  - Maintain ***wait-for*** graph
    - Nodes are processes
    - P_i ® P_j if P_i is waiting for P_j

  - **Periodically** invoke an algorithm that searches for a **cycle** in the graph.
  - If there is a **cycle**, there exists a **deadlock**
  - An algorithm to detect a cycle in a graph requires an order of n^2 operations,
    where n is the number of vertices in the graph

Resource-Allocation Graph **&** ***Wait-for Graph***

-

Resource-Allocation Graph Corresponding wait-for graph

- ## Several Instances of a Resource Type
  - ✓ **Available**: A vector of length *m* indicates the number of available resources of each type.
  - ✓ **Allocation**: An *n* x *m* matrix defines the number of resources of each type currently allocated to each process.
  - ✓ **Request**: An *n* x *m* matrix indicates the current request of each process.
    If ***Request*** **[*i*$_j$] = *k***, then process ***P*$_i$** is requesting ***k*** more instances of resource type ***R*$_j$.**

Detection Algorithm

1. Let ***Work*** and ***Finish*** be vectors of length ***m*** and ***n***, respectively
   Initialize:
   Work = ***Available***
   For *i* = 1,2, …, *n*: <u>if</u> ***Allocation*$_i$** ^{1} 0, <u>then</u> ***Finish*[i]** = false; <u>else</u>
   ***Finish*[i]** = *true*

2. Find an index ***i*** such that both:
   Finish*[*i*] == *false **AND** ***Request*$_i$ £ *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*] == false**, for some *i*, 1 £ *i* £ *n*, then the system is in **deadlock state**.
   and P_i is **deadlocked**

The Algorithm requires *m* x *n*^{2} operations to detect whether the system is in deadlocked state

Example of Detection Algorithm

- Five processes P_0 through P_4;
- Three resource types
  A B C
  7 2 6

Snapshot at time T_0:

	Allocation	***Request***	***Available***
	A B C	***A B C***	***A B C***
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 0	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

System is **not deadlocked**, sequence **<P_0, P_2, P_3, P_1, P_4>** will result in ***Finish*[*i*] = true** for all ***i***

***P*$_2$** requests an additional **1** instance of type *C*

	Request
	A B C
P_0	0 0 0
P_1	2 0 2
P_2	0 0 1
P_3	1 0 0
P_4	0 0 2

- State of system?

- Can reclaim resources held by process P_0,
  but insufficient resources to fulfill other processes requests

- **Deadlock exists**, consisting of processes P_1, P_2, P_3, **and** P_4

➔Recovery from Deadlock:

Options:

- **Abort all** deadlocked processes
- **Abort one** process at a time until the deadlock cycle is eliminated
- In **which order** should we choose to abort?
  - **Priority** of the process
  - **How long** process has computed, and **how much longer** to completion
  - **Resources** the process has used
  - Resources process **needs** to complete
  - **How many processes** will need to be terminated
  - Is process **interactive** or **batch**?
- Selecting a **victim** to ***minimiz*e *cost***
- **Rollback** – return to some safe state, restart process from that state
- **Starvation** – **same process** may always be picked as victim, include number of rollback in cost factor

Main Memory

❖ Basics:

- Program must be brought (from disk) into Main memory to run
- Main memory and registers are the only storage the CPU can access directly
- Register accessed in one CPU clock cycle (or less)
- Main memory can take many cycles
- Cache sits between main memory and CPU registers
- Protection of memory required to ensure correct operation

Base and Limit Registers

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

Logical address space bound to a separate **physical address space** is central concept in memory management

- **Logical address** – generated by the CPU; also referred to as **virtual address**
- **Physical address** – address seen by the memory unit

Memory-Management Unit (MMU)

- Hardware device that maps **virtual** to **physical** address
- The value in the **relocation register** is added to every address generated by a user process
- The user program deals with ***logical*** addresses; it never sees the real ***physical*** addresses

Dynamic Loading

- Routine is not loaded **until it is called**
- Better memory-space utilization; unused routine is never loaded
- Useful when large amounts of code are needed to handle **infrequently occurring cases**

Swapping

- A process can be swapped temporarily out of memory to a **backing store**, and then brought back for continued execution
- **Backing store** – **fast disk** large enough to accommodate copies of all memory images for all users;
- Major part of swap time is **transfer time**; directly proportional to the amount of memory swapped

Hardware Support for Relocation and Limit Registers

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**

❖ Multiple-partition:

Multiple-partition allocation

- **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
- Operating system maintains information about:
  - allocated partitions
  - free partitions (hole)

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

Fragmentation

- **External Fragmentation** – total memory space exists to satisfy a request, but it is not contiguous
- 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**

❖ Paging:

- Logical address space of a process can be **noncontiguous**; process is allocated physical memory

whenever available

- Divide **physical memory** into **fixed-sized blocks** called ***frames*** (size is power of 2, between 512 bytes and 8,192 bytes)
- 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
- Produce **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

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 (PRLR)** 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)**

Effective Access Time

- Associative Lookup = e time unit
- Assume memory cycle time is 1 microsecond
- Hit ratio = a percentage of times that a page number is found in the associative registers;
- Effective Access Time (EAT):

$$EAT = (1 + e)\,a + (2 + e)(1 - a)$$
$$= 2 + e - a$$

For example: If $a = .9$ and $e = .1$ then EAT = 1.2

Shared Pages

- **Shared code**
  - One copy of read-only (reentrant) code shared among processes (i.e., text editors, compilers)
  - Shared code must appear in same location in the logical address space of all processes
- **Private code and data**
  - Each process keeps a separate copy of the private code and data
  - The pages for the private code and data can appear anywhere in the logical address space

Shared Pages Example

❖ Structure of the Page Table:

- Hierarchical Paging
- Hashed Page Tables
- Inverted Page Tables

Two-Level Page-Table Scheme

Two-Level Paging Example

- A logical address (on 32-bit machine with 1K page size) is divided into:
  - a page number consisting of 22 bits
  - a page offset consisting of 10 bits
- Since the page table is paged, the page number is further divided into:
  - a 12-bit page number
  - a 10-bit page offset
- Thus, a logical address is shown above, where p_i is an index into the outer page table, and p_2 is the displacement within the page of the outer page table

Hashed Page Tables

- Common in address spaces > 32 bits
- The virtual page number is hashed into a page table
  - This page table contains a chain of elements hashing to the same location
- Virtual page numbers are compared in this chain searching for a match
  - If a match is found, the corresponding physical frame is extracted

❖ Segmentation:

Memory-management scheme that supports user view of memory
A program is a collection of **segments**

Example of Segmentation

Virtual Memory

Objectives

- To describe the benefits of a virtual memory system
- To explain the concepts of ***demand paging*** **&** ***page-replacement*** algorithms, and allocation of page frames
- To discuss the principle of the ***working-set*** model

Background

Virtual memory – separation of user **logical** memory from **physical** memory:

- Only **part** of the program needs to be in memory for execution
- Logical address space can therefore be much **larger** than physical address space
- Allows address spaces to be **shared** by several processes

Virtual Memory **Larger** than *Physical* Memory

Shared Library Using Virtual Memory

Demand Paging

Bring a page into memory only when it is **needed**

Valid-Invalid Bit

- With each page table entry a valid–invalid bit is associated (v => in-memory, i => not-in-memory)
- Initially valid–invalid bit is set to i on all entries
- Example of a page table snapshot:

- During address translation, if valid–invalid bit in page table entry is i => page fault

Page Table When Some Pages Are Not in Main Memory

Page Fault

- If there is a reference to a page, first reference to that page will trap to operating system:

  page fault

- Get empty frame
- Swap page into frame
- Reset page table
- Set validation bit = **v**
- Restart the instruction that caused the page fault

Steps in Handling a Page Fault

Performance of Demand Paging

- Page Fault Rate 0 £ *p* £ 1.0
  - if *p* = 0 no page faults
  - if *p* = 1, every reference is a fault
- Effective Access Time (EAT)

  EAT = (1 – *p*) x memory access + *p* x (page fault ***overhead***)

  overhead = swap page out+ swap page in+ restart overhead.

Example

- Memory access time = 200 nanoseconds
- Average overhead time = 8 milliseconds
- EAT = (1 – p) x 200 (nanoseconds) + p x 8 (milliseconds)
- If one access out of 1,000 causes a page fault, then

  EAT = 8200 nanoseconds.
- This is a slowdown by a factor of 40!!

Copy-on-Write

- Copy-on-Write (COW) allows both **parent** and **child** processes to initially **share** the same pages in memory

  If either process modifies a shared page, only then is the page copied
- COW allows more efficient process creation as only **modified** pages are copied

Before **Process 1 Modifies Page C**

After **Process 1 Modifies Page C**

What happens if there is no free frame?

Page replacement – find some page in memory, but not really in use, swap it out
Performance – want an algorithm which will result in **minimum number of page faults**

Basic Page Replacement

- Find the location of the desired page on disk
- Find a free frame:
  - If there is a free frame, use it
  - If there is no free frame, use a page replacement algorithm to select a ***victim*** frame
- Bring the desired page into the (newly) free frame; update the page and frame tables
- Restart the process

Page Replacement Algorithms

- ✓ We want to have the **lowest page-fault rate**.
- ✓ **Evaluate algorithms** by running it on a particular string of memory references and Compute the **number of page faults** on that string.

Algorithms:

- **FIFO**: First-In-First-Out
- **Optimal** (replace the page that will ***not be used for longest time*** in the future).
- **LRU**: Least Recently Used.
- **LFU**: Least Frequently Used,
- **MFU**: Most Frequently Used.

First-In-First-Out (FIFO) Algorithm

- Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- 3 frames (3 pages can be in memory at a time per process)

1	1	4	5	
2	2	1	3	9 page faults
3	3	2	4	

- 4 frames

1	1	5	4	
2	2	1	5	10 page faults
3	3	2		
4	4	3		

- Belady's Anomaly: more frames => more page faults

FIFO Page Replacement Example (# of page faults = 15)

FIFO Illustrating Belady's Anomaly

Optimal Algorithm

- Replace page that will not be used for longest period of time
- 4 frames example

1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

- How do you know this?
- Used for measuring how well your algorithm performs

Optimal Page Replacement Example (# of page faults = 9)

Least Recently Used (LRU) Algorithm

- Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

1	1	1	1	5
2	2	2	2	2
3	5	5	4	4
4	4	3	3	3

- Counterimplementation
  - Every page entry has a counter; every time page is referenced through this entry, copy the clock into the counter
  - When a page needs to be changed, look at the counters to determine which are to change

LRU Page Replacement Example (# of page faults = 12)

reference string

7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

7	7	7	2		2		4	4	4	0		1		1		1		
	0	0	0		0		0	0	3	3		3		0		0		
		1	1		3		3	2	2	2		2		2		7		

page frames

Stack implementation of LRU – keep a stack of page numbers:

- Page referenced: move it to the top
- No search for replacement, just replace the page at the bottom of the stack.

Counting Algorithms

Keep a counter of the number of references that have been made to each page.

- **LFU Algorithm**: replaces page with **least** count
- **MFU Algorithm**: replaces page with **most** count.
  Based on the argument that:
  the page with the smallest count was probably just brought in and has yet to be used.

Frame Allocation

- Each process needs a **minimum** number of pages
- Two major allocation schemes
  - **fixed** equal allocation
  - **priority** allocation

Fixed Allocation

- Equal allocation – For example, if there are 100 frames and 5 processes, give each process 20 frames.
- Proportional allocation – Allocate according to the size of process
  - s_i = size of process p_i
  - $S = \sum s_i$
  - m = total number of frames
  - a_i = allocation for $p_i = \frac{s_i}{S} \times m$

$$m = 64$$
$$s_1 = 10$$
$$s_2 = 127$$
$$a_1 = \frac{10}{137} \times 64 \approx 5$$
$$a_2 = \frac{127}{137} \times 64 \approx 59$$

Priority Allocation

- Use a proportional allocation scheme using priorities rather than size
- If process P_i generates a page fault,
  select for replacement a frame from a process with lower priority number

Global vs. Local Allocation

- **Global** replacement – process selects a replacement frame from the set of all frames; one process can take a frame **from another**
- **Local** replacement – each process selects from only **its own** set of allocated frames

Thrashing

- If a process does not have “enough” pages, the page-fault rate is very high. This leads to:
  - low CPU utilization
  - operating system thinks that it needs to increase the degree of multiprogramming
  - another process added to the system

- **Thrashing** º a process is busy swapping pages in and out

Demand Paging and Thrashing

-

- *Why does demand paging work?*
  Locality model
  - Process migrates from one locality to another
  - Localities may overlap

- *Why does thrashing occur?*
  S size of locality > total memory size

Working-Set Model

-

- **D** º working-set window º a fixed number of page references
  Example: 10,000 instruction
- WSS_i (working set of Process P_i) =
  total **number of pages referenced** in the most recent D (varies in time)
  - if D too small will not encompass entire locality
  - if D too large will encompass several localities
  - if D = ¥ Þ will encompass entire program
- D = S WSS_i º total demand frames

- if $D > m$ Þ Thrashing
- Policy if $D >$ m, then suspend one of the processes

Page-Fault Frequency Scheme

- Establish "**acceptable**" ***page-fault rate***
  - ✓ If actual rate too low, process loses frames
  - ✓ If actual rate too high, process gains frames

Other Issues

➢ Prepaging

- To reduce the large number of page faults that occurs at process startup
- Prepage all or some of the pages a process will need, before they are referenced

➢ Page Size

Page size selection must take into consideration:

- fragmentation
- table size
- I/O overhead
- locality

➢ Program structure

Int [128,128] data;
Each row is stored in one page

Program 1:

```
for (j = 0; j <128; j++)
  for (i = 0; i < 128; i++)
      data[i,j] = 0;
```

128 x 128 = 16,384 page faults

Program 2:

```
for (i = 0; i < 128; i++)
  for (j = 0; j < 128; j++)
      data[i,j] = 0;
```

128 page faults

➢ I/O Interlock

Pages must sometimes be locked into memory
Consider I/O - Pages that are used for copying a file from a device must be **locked** from being selected for eviction by a page replacement algorithm

buffer
disk drive

File System Interface

File Attributes

- **Name** – only information kept in human-readable form
- **Identifier** – unique tag (number) identifies file within file system
- **Type** – needed for systems that support different types
- **Location** – pointer to file location on device
- **Size** – current file size
- **Protection** – controls who can do reading, writing, executing
- **Time, date & user identification**

Information about files are kept in the directory structure, which is maintained on the disk

File Operations

- **Create**
- **Write**
- **Read**
- **Reposition within file**
- **Delete**
- **Truncate**

File Locking

- **Mandatory** – access is **denied** depending on locks held.
- **Advisory** – processes can find status of locks and decide what to do.

Lock and Unlock Functions:

```
read_lock(int fd)
{
      fcntl(fd, F_SETLKW, file_lock(F_RDLCK, SEEK_SET));
}

write_lock(int fd)
{
      fcntl(fd, F_SETLKW, file_lock(F_WRLCK, SEEK_SET);
}

unlock(int fd)
{
      fcntl(fd, F_SETLKW, file_lock(F_UNLCK, SEEK_SET);
}

struct flock   *
file_lock(short type, short whence)
{
```

```
        static struct flock ret;
        ret.l_type = type;
        ret.l_start = 0;
        ret.l_whence = whence;
        ret.l_len = 0;  /* means until end of file */
        ret.l_pid = getpid();
        return &ret;
}
```

Example: [LockingExampleRead.c]

```
int
main(int argc, char **argv)
{
        int             fd;
        fd = open(argv[1], O_RDONLY);
        read_lock(fd);

    printf("obtained read lock\n");
        sleep(30);

        unlock(fd);
}
```

Example: [LockingExampleWrite.c]

```
int
main(int argc, char **argv)
{
        int             fd;
        fd = open(argv[1], O_WRONLY);
        write_lock(fd);

    printf("obtained write lock\n");
        sleep(30);


        unlock(fd);
}
```

Example usage: On 4 windows:

```
% LockingExampleRead lockfile.txt        % LockingExampleRead lockfile.txt

% LockingExampleWrite lockfile.txt       % LockingExampleWrite lockfile.txt
```

NOTE: **When a process is terminated, all file locks are released.**

Example: [wait4allforkFileLock.c]

File Types – Name, Extension

file type	usual extension	function
executable	exe, com, bin or none	ready-to-run machine-language program
object	obj, o	compiled, machine language, not linked
source code	c, cc, java, pas, asm, a	source code in various languages
batch	bat, sh	commands to the command interpreter
text	txt, doc	textual data, documents
word processor	wp, tex, rtf, doc	various word-processor formats
library	lib, a, so, dll	libraries of routines for programmers
print or view	ps, pdf, jpg	ASCII or binary file in a format for printing or viewing
archive	arc, zip, tar	related files grouped into one file, sometimes compressed, for archiving or storage
multimedia	mpeg, mov, rm, mp3, avi	binary file containing audio or A/V information

Access Methods

Sequential Access

read next
write next
reset

Direct Access

read *n*
write *n*
position to *n*
read next
write next

n = relative block number

Example: fileseek.c

```
int
main(int argc, char **argv)
{
  int  fd, length;
  off_t FileLength, position;
  char buf[1024];
  fd = open(argv[1], O_RDONLY);
  position = atoi(argv[2]);
  length = atoi(argv[3]);
  FileLength = lseek(fd, 0, SEEK_END);
  printf("File Length %d\n", FileLength);
  if ((position+length) > FileLength){
```

```
    printf("exceeds file limit\n");
    exit(0);
  }
  lseek(fd, position, SEEK_SET);
  read(fd, buf, length);
  write(1, buf, length);
}
```

Directory Structure

- A collection of nodes containing information about all files

Both the directory structure and the files reside on disk
Backups of these two structures are kept on tapes

Typical File-system Organization

Operations Performed on Directory

- Search for a file
- Create a file
- Delete a file
- List a directory

- Rename a file
- Traverse the file system

Single-Level Directory

Two-Level Directory

Tree-Structured Directories

<u>Acyclic-Graph Directories</u>

General Graph Directory

Example:

```
>touch  testfile1
>ls  -l
-rw-------  1 cs471w cs471w    0 2011-04-08 11:46 testfile1

>ln   testfile1   testfile1-link1

>ls   -l
-rw-------  2 cs471w cs471w    0 2011-04-08 11:46 testfile1
-rw-------  2 cs471w cs471w    0 2011-04-08 11:46 testfile1-link1

>ln testfile1  testfile1-link2

>ls -l
-rw-------  3 cs471w cs471w    0 2011-04-08 11:46 testfile1
-rw-------  3 cs471w cs471w    0 2011-04-08 11:46 testfile1-link1
-rw-------  3 cs471w cs471w    0 2011-04-08 11:46 testfile1-link2

>ln -s /home/cs471w symbolic-link

>ls -l
lrwxrwxrwx  1 cs471w cs471w   12 2011-04-08 11:50 symbolic-link -> /home/cs471w
```

File System Mounting:

File Sharing

- Sharing of files on multi-user systems is desirable
- Sharing may be done through a **protection** scheme
- On distributed systems, files may be shared across a **network**
- Network File System (**NFS**) is a common distributed file-sharing method

File Sharing – Multiple Users

- **User IDs** identify users, allowing permissions and protections to be per-user
- **Group IDs** allow users to be in groups, permitting group access rights

File Sharing – Remote File Systems

- Uses networking to allow file system access between systems
  - Manually via programs like **FTP**
  - Automatically, seamlessly using **distributed file systems**
  - Semi automatically via the **world wide web**

- **Client-server** model allows clients to mount remote file systems from servers
  - Server can serve multiple clients
  - **NFS** is standard UNIX client-server file sharing protocol
  - Standard operating **system file calls** are translated into **remote calls**

- Remote file systems add new **failure modes**, due to network failure, server failure

- Unix file system (UFS) implements:
  Writes to an open file is **visible immediately** to other users of the same open file
  Sharing file pointer to allow multiple users to read and write **concurrently**

Protection

- File owner/creator should be able to control:
  - what can be done
  - by whom
- Types of access
  - **Read**
  - **Write**
  - **Execute**
  - **Append**
  - **Delete**
  - **List**

Access Lists and Groups

- Mode of access: read, write, execute
- Three classes of users

			RWX
a) owner access	7	=>	1 1 1
			RWX
b) group access	6	=>	1 1 0
			RWX
c) public access	1	=>	0 0 1

- Ask manager to create a group (unique name), say G, and add some users to the group.
- For a particular file (say *game*) or subdirectory, define an appropriate access.

Attach a group to a file

chgrp G game

Example: **getfacl & setfacl**

```
% cd
% pwd
/home/wahab
% hostname
atria
% uname
Linux
% cd /tmp
% touch testfile
% /bin/ls -lt testfile
-rw-r----- 1 wahab faculty 0 Apr 15 12:01 testfile

% getfacl testfile
# file: testfile
# owner: wahab
# group: faculty
user::rw-
group::r--
other::---

% setfacl -m u:cs471w:rwx

% getfacl testfile
# file: testfile
# owner: wahab
# group: faculty
user::rw-
user:cs471w:rwx
group::r--
mask::rwx
other::---

/bin/ls -lt testfile
-rw-rwx---+ 1 wahab faculty 0 Apr 15 12:01 testfile
```

Example: **groups & chgrp**

```
 % groups
faculty cs778 cs779 cs471w tcpdump cs476

% chgrp cs471w testfile
/bin/ls -lt testfile
-rw-rwx---+ 1 wahab cs471w 0 Apr 15 12:01 testfile
```

```
% getfacl testfile
# file: testfile
# owner: wahab
# group: cs471w
user::rw-
user:cs471w:rwx
group::r--
mask::rwx
other::---
```

Example: listfiles

```
int
main(int argc, char **argv)
{
   DIR            *dp;
   char           *dirname;
   struct stat     st;
   struct dirent  *d;
   char            filename[1024];

   dirname = argv[1];

   if ((dp = opendir(dirname)) == NULL) {
      perror(dirname);
      exit(-1);
   }

   while ((d = readdir(dp)) != NULL) {
      sprintf(filename, "%s/%s", dirname, d->d_name);
      if (lstat(filename, &st) < 0) {
         perror(filename);
         continue;
      }
      printf("%u", st.st_ino);
      printf("%5d ", st.st_blocks);
      printf("%3d ", st.st_nlink);
      printf("%5d/%-5d ", st.st_uid, st.st_gid);
      printf("%.12s ", ctime(&st.st_mtime) + 4);
      printf("%s\n", filename);
   }
}
```

File System Implementation

- **Boot control block** contains info needed by system to boot OS from that volume
- **Volume control block** contains volume details
- **Directory structure** organizes the files
- **Per-file File Control Block (FCB)** contains many details about the file

A Typical File Control Block

file permissions
file dates (create, access, write)
file owner, group, ACL
file size
file data blocks or pointers to file data blocks

File Blocks Allocation Methods

An allocation method refers to how disk blocks are allocated for files:

1. Contiguous allocation
2. Linked allocation
3. Indexed allocation

1. Contiguous Allocation

directory

file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2

2. Linked Allocation

3. Indexed Allocation

UNIX File system

Free Space Management

Bit vector of (n blocks):
Bit [*i*] = 0 means **block *i*** is ***occupied***
Bit [*i*] = 1 means **block *i*** is ***free.***

First Free block: the position of the first 1 bit:

Is Calculated as follows: (number of 0-valued words) * (number of bits per word) + offset of first 1 bit.

Linked Free Space List

Linked list (free list)

- Cannot get contiguous space easily
- No waste of space

Recovery

- Use system programs to back up data from disk to another storage device (magnetic tape, other magnetic disk, optical)
- Recover lost file or disk by restoring data from backup

Log Structured File Systems

- Log structured (or journaling) file systems: Record each update to the file system as a **transaction**
- All transactions are **written** to a **log**
  A transaction is considered committed once it is written to the log. However, the file system may not yet be updated.
- The transactions in the log are **asynchronously** written to the file system
  When the file system is modified, the transaction is removed from the log.

The Sun Network File System (NFS)

- Interconnected workstations viewed as a set of independent machines with independent file systems, which allows **sharing among these file systems** in a transparent manner
- A remote directory is **mounted** over a local file system directory.
  The mounted directory looks like an integral subtree of the local file system, replacing the subtree descending from the local directory. Files in the remote directory can then be accessed in a transparent manner.
- Subject to **access-rights** accreditation, potentially any file system (or directory within a file system), can be mounted remotely on top of any local directory
- NFS is designed to operate in a **heterogeneous environment** of different machines, operating systems, and network architectures; the NFS specifications independent of these media. This independence is achieved through the use of **RPC** primitives built on top of an External Data Representation (**XDR**) protocol used between two implementation-independent interfaces

Three Independent File Systems

Mounting in NFS

NFS Protocol

- Provides a set of remote procedure calls for **remote file operations**. The procedures support the following operations:
  - ✓ searching for a file within a directory
  - ✓ reading a set of directory entries
  - ✓ manipulating links and directories
  - ✓ accessing file attributes
  - ✓ reading and writing files
- NFS servers are **stateless**; each request has to provide a full set of arguments
- Modified data must be committed to the server's disk before results are returned to the client.
- The NFS protocol does **not provide concurrency-control** mechanisms

Schematic View of NFS Architecture

Mass-Storage Systems

Overview of Mass Storage Structure

Magnetic disks provide bulk of secondary storage of modern computers:

- Drives **rotate** at 60 to 200 times per second
- **Transfer rate** is the rate at which data flow between drive and computer
- **Positioning time** (random-access time) is time to move disk arm to desired cylinder (**seek time**)
  and time for desired sector to rotate under the disk head (**rotational latency**)
- **Head crash** results from disk head making contact with the disk surface
- Disks can be **removable**
- Disk drives are addressed as large **1-dimensional arrays** of **logical blocks**,

where the logical block is the **smallest unit of transfer**

- The 1-dimensional array of logical blocks is mapped into the **sectors** of the disk sequentially.
  Sector 0 is the first sector of the **first track** on the **outermost cylinder.**
  Mapping proceeds in order through that track, then the rest of the tracks in that cylinder and
  then through the rest of the cylinders from **outermost to innermost.**

Disk Scheduling Algorithms

The operating system is responsible for using hardware efficiently.
For the disk drives, this means having a fast ***access time*** & ***disk bandwidth***.

✓ **Access time** has two major components:
Seek time is the time for the disk to **move the heads** to the cylinder containing the desired sector
Rotational latency time waiting for the disk to rotate the desired sector to the disk head

We like to ***minimize seek time***.

✓ **Disk bandwidth** is the total number of bytes transferred ***divided*** by the total time between the first request for service and the completion of the last transfer.

Several algorithms exist to schedule the servicing of disk I/O requests.
We illustrate them with a **Request Queue** (cylinder range 0-199):

98, 183, 37, 122, 14, 124, 65, 67

Head pointer: cylinder 53

➔FCFS

Illustration shows total head movement of **640** cylinders

➔SSTF

Selects the request with the **minimum seek time** from the current head position
SSTF scheduling may cause **starvation** of some requests

Illustration shows total head movement of **236** cylinders

➔SCAN

The disk arm **starts at one end** of the disk, and moves toward the other end,
servicing requests until it gets to the other end of the disk,
where the head movement is **reversed** and servicing continues.

SCAN algorithm sometimes called the **elevator** algorithm.

Illustration shows total head movement of **208** cylinders

➔C-SCAN

Provides a more uniform wait time than SCAN

- ✓ The head moves from one end of the disk to the other, servicing requests as it goes
- ✓ When it reaches the other end, however, it immediately **returns to the beginning of the disk**,
  without servicing any requests on the return trip.

Treats the cylinders as a **C**ircular list that wraps around from the last cylinder to the first one

➔C-LOOK

- ✓ Version of C-SCAN
- ✓ Arm only goes as far as the **last request** in each direction, then reverses direction immediately, without first going all the way to the end of the disk.

Selecting a Disk-Scheduling Algorithm

- ✓ **SSTF** is common and has a natural appeal
- ✓ **SCAN** and **C-SCAN** perform better for systems that place a heavy load

on the disk

Network-Attached Storage

- Network-attached storage (NAS) is storage made available over a **network**
  rather than over a local connection (such as a bus)
- Implemented via **remote procedure calls** (RPCs) between host and storage

Storage Area Network

RAID (Redundent Arrays of Independent Disks) Structure

- multiple disk drives provides reliability via redundancy
  Increases the mean time to failure
- improve performance and reliability of the storage system by storing redundant data
- Mirroring or shadowing keeps duplicate of each disk
- small number of hot-spare disks are left unallocated,
  automatically replacing a failed disk and having data rebuilt onto them

Stable-Storage Implementation

To implement stable storage:

- ✓ **Replicate** information on more than one nonvolatile storage media with independent failure modes
- ✓ **Update** information in a controlled manner to ensure that we can recover the stable data after any failure during data transfer or recovery.

Tertiary Storage Devices

- **Low cost** is the defining characteristic of tertiary storage
- Generally, tertiary storage is built using **removable** media
- Common examples of removable media are **floppy disks** and **CD-ROMs**; other types are available

Removable Disks

- **Floppy disk** — thin flexible disk coated with magnetic material, enclosed in a protective plastic case
  Most floppies hold about 1 MB;
- **Removable magnetic disks** can be nearly as fast as hard disks,
  but they are at a greater risk of damage from exposure
- **Magneto-optic disk** records data on a rigid platter coated with magnetic

material
The magneto-optic head flies much farther from the disk surface than a magnetic disk head,
and the magnetic material is covered with a protective layer of plastic or glass; resistant to head crashes

- **Optical disks** do not use magnetism; they employ special materials that are altered by laser light

WORM Disks

- The data on **read-write** disks can be modified over and over
- WORM ("**Write Once, Read Many** Times") disks can be written only once
- Thin aluminum film sandwiched between two glass or plastic platters
- To write a bit, the drive uses a laser light to burn a small hole through the aluminum;
  information can be destroyed by not altered
- Very durable and reliable
- **Read-only** disks, such as **CD-ROM** and **DVD**, come from the factory with the data pre-recorded

Magnetic tape

- Was early secondary-storage medium
- Relatively **permanent** and holds **large** quantities of data
- Access time **slow**
- Random access ~1000 times slower than disk
- Mainly used for **backup**, storage of infrequently-used data, transfer medium between systems
- Once data under head, transfer rates comparable to disk
- 20-200GB typical storage
- Compared to a disk, a tape is less expensive and holds more data, but **random access** is much slower

- Large tape installations typically use **robotic tape changers** that move tapes between tape drives and storage slots in a tape library

Tape Drives Application Interface

- The basic operations for a tape drive differ from those of a disk drive
- **locate()** positions the tape to a specific logical block,
- The **read_position()** returns the logical block number where the tape head is
- The **space()** operation enables relative motion
- Tape drives are "**append-only**" devices; updating a block in the middle of the tape also effectively erases everything beyond that block
- An **EOT mark** is placed after a block that is written

Hierarchical Storage Management (HSM)

- A **hierarchical storage system** extends the storage hierarchy beyond primary memory and secondary storage to incorporate tertiary storage — usually implemented as a **jukebox of tapes** or **removable disks**
- Small and frequently used files remain on disk
- Large, old, inactive files are archived to the jukebox
- HSM are found in supercomputing centers that have enormous volumes of data.

Speed

- Two aspects of speed in tertiary storage are **bandwidth** and **latency**
  Sustained bandwidth – average data rate during a large transfer;
  Effective bandwidth – average over the entire I/O time, including seek() or locate(), and cartridge switching
  Access latency – amount of time needed to locate data
- Access time for a disk – move the arm to the selected cylinder and wait

for the rotational latency;
< 35 milliseconds

- Access on tape requires winding the tape reels until the selected block reaches the tape head;
  tens or hundreds of seconds
- Generally random access within a tape cartridge is about a **thousand times slower** than random access on disk

Reliability

- A **fixed disk** drive is likely to be more reliable than a **removable disk** or tape drive
- An **optical cartridge** is likely to be more reliable than a **magnetic disk or tape**
- A **head crash** in a fixed hard disk generally destroys the data, whereas the **failure** of a tape drive or optical disk drive often leaves the data cartridge unharmed

Cost

-

- **Main memory** is much more expensive than **disk storage**
- The cost per megabyte of hard disk storage is competitive with magnetic tape.

I/O Systems

I/O Hardware

- Incredible **variety** of I/O devices
- Common concepts
  - Port
  - Bus
  - Controller
- I/O instructions control devices
- Devices have addresses, used by
  - Direct I/O instructions
  - Memory-mapped I/O

A Typical PC Bus Structure

Device I/O Port Locations on PCs (partial)

I/O address range (hexadecimal)	device
000–00F	DMA controller
020–021	interrupt controller
040–043	timer
200–20F	game controller
2F8–2FF	serial port (secondary)
320–32F	hard-disk controller
378–37F	parallel port
3D0–3DF	graphics controller
3F0–3F7	diskette-drive controller
3F8–3FF	serial port (primary)

Polling

- **Determines state of device**
  - **ready**
  - **busy**
  - **Error**
- **Busy-wait cycle to wait for I/O from device**

Interrupts

- CPU Interrupt- triggered by I/O device
- Interrupt handler receives interrupts
- Maskable to ignore or delay some interrupts
- Interrupt vector to dispatch interrupt to correct handler
  - Based on priority
  - Some nonmaskable
- Interrupt mechanism also used for **exceptions**

Direct Memory Access

- Used to avoid programmed I/O for large data movement
- Requires DMA controller
- Bypasses CPU to transfer data directly between I/O device and memory

Application I/O Interface

- **I/O system calls** encapsulate device behaviors in generic classes
- Device-driver layer hides differences among I/O controllers from kernel
- Devices vary in many dimensions:
  - Character-stream or block
  - Sequential or random-access
  - Sharable or dedicated
  - Speed of operation
  - read-write, read only, or write only

aspect	variation	example
data-transfer mode	character block	terminal disk
access method	sequential random	modem CD-ROM
transfer schedule	synchronous asynchronous	tape keyboard
sharing	dedicated sharable	tape keyboard
device speed	latency seek time transfer rate delay between operations	
I/O direction	read only write only read–write	CD-ROM graphics controller disk

A Kernel I/O Structure

Block and Character Devices

- **Block devices** include disk drives
  - Commands include ***read(), write(), seek()***
  - Raw I/O or file-system access
  - Memory-mapped file access possible
- **Character devices** include keyboards, mice, serial ports
  - Commands include ***get(), put()***
- **Libraries** layered on top allow **line editing.**

Network Devices

- Varying enough from block and character to have own interface
- Unix and Windows include **socket interface**
  Separates network protocol from network operation.

❖Clocks and Timers

-

- Provide current time, timer
- Programmable interval timer used for timings, periodic **interrupts.**

✓Example1: timeout.c

```
main(int argc, char *argv[])
{
        char            buf[BUFSIZ];
        int             timeout;

        timeout = atoi(argv[1]);
        while (1) {
                signal(SIGALRM, alarmhandler);
                alarm(timeout);
                printf("Enter a string (withen %d seconds): ", timeout);
                if ( gets(buf)!= NULL) {
                     alarm(0);
```

```
                        printf("got %s\n", buf);
                }
        }
}

alarmhandler(int sig)
{
        printf("Timeout, Hurry Up!\n");
}
```

Usage Example:

```
% timeout 3
```

✓ Example2: ChatClientAlarm.c

```
int main(int argc, char **argv)
{
        int             sockfd;
        struct sockaddr_in servaddr;
        struct hostent *hp, *gethostbyname();
        char            buffer[512];
        int             nread;

        sockfd = socket(AF_INET, SOCK_STREAM, 0);       bzero(&servaddr,
sizeof(servaddr));
        hp = gethostbyname(argv[1]);
        bcopy(hp->h_addr, &(servaddr.sin_addr.s_addr), hp->h_length);
servaddr.sin_family = AF_INET;
        servaddr.sin_port = htons(atoi(argv[2]));

        if (connect(sockfd, (SA *) & servaddr, sizeof(servaddr)) < 0) {
                perror("connect error");
                exit(-1);
        }
      AlarmChat (sockfd);
}
AlarmChat(int sockfd)
{
        int             nread;
        char            buffer[512];
        siginterrupt(SIGALRM, 1);


        signal(SIGALRM, handle_ALARM);
        alarm(1);
        for (;;) {
                nread = read(0, buffer, 512);
                if (nread == 0)
                        exit(0);
                write(sockfd, buffer, nread);
                nread = read(sockfd, buffer, 512);
                if (nread == 0)
                        exit(0);
```

```
                write(1, buffer, nread);
        }
}

handle_ALARM(int signo)
{
        signal(SIGALRM, handle_ALARM);
        alarm(1);

}
```

Usage Example:

```
% ChatServerSelect 10990 &

% ChatClientAlarm localhost 10990 1
```

❖ Blocking and Nonblocking I/O

- **Blocking** - process suspended until I/O completed
- **Nonblocking** - Returns quickly with count of bytes read or written
- **Asynchronous** - I/O subsystem signals process when I/O completed

➢ Blocking:

- ✓ Example 3: block.c

```
int main(int argc, char *argv[])
{
        char  buffer[512];
        int nread;
        while (1) {
                 printf("Enter a string: ");
                nread = read(0, buffer, 512);
                if (nread > 0) {
                        printf("\ngot: ");
                          write(1, buffer, nread);
                }
            }
}
```

Usage Example:

```
% block
```

➢ Nonblocking:

✓ Example 4:nonblockRead.c

```
int
main(int argc, char *argv[])
{
  char            buffer[512];
  int             sleeptime;
  int             nread;
  int             val;

  sleeptime = atoi(argv[1]);

  val = fcntl(0, F_GETFL, 0);
  fcntl(0, F_SETFL, val | O_NONBLOCK);

  while (1) {
    printf("Enter a string: ");
    nread = read(0, buffer, 512);
    if (nread == -1) {
      if (errno == EWOULDBLOCK) {
        printf("WouldBlock, sleeping: %d\n", sleeptime);
        sleep(sleeptime);
      } else {
        perror("read error");
        exit(-1);
      }
    } else if (nread == 0) {
      printf("EOF\n");
      exit(0);
    } else {
      printf("\nGot: ");
      write(1, buffer, nread);
    }
  }
}
```

Usage Example:

```
% nonblockRead 3
```

✓ Example 5: nonblockWrite.c

```
-
char            buffer[128];
void            fillbuf();
int
main(int argc, char *argv[])
{
   int             sleeptime;
   int             nwrite;
   int             val;
   int             Total = 0;

   sleeptime = atoi(argv[1]);
   fillbuf();

   val = fcntl(1, F_GETFL, 0);
```

```
   fcntl(1, F_SETFL, val | O_NONBLOCK);

   while (1) {
      nwrite = write(1, buffer, sizeof(buffer));
      printf("..%d..", nwrite);
      if (nwrite)
         Total += nwrite;
      if (nwrite == -1) {
         if (errno == EWOULDBLOCK) {
            sleep(sleeptime);
            printf("\nwrote %d, slept %d\n", Total, sleeptime);
            Total = 0;
            sleep(sleeptime);
         }
      }
   }
}

fillbuf()
{
   int             i;
   for (i = 0; i <= sizeof(buffer); i++)
      buffer[i] = NULL;
}
```

Usage Example:

```
% nonblockWrite 3
```

✓ Example 6: ChatClientNonBlock.c

```
int main(int argc, char **argv)
{
        int             sockfd;
        struct sockaddr_in servaddr;
        struct hostent *hp, *gethostbyname();
        char            buffer[512];
        int             nread;

        sockfd = socket(AF_INET, SOCK_STREAM, 0);

        bzero(&servaddr, sizeof(servaddr));
        hp = gethostbyname(argv[1]);
        bcopy(hp->h_addr, &(servaddr.sin_addr.s_addr), hp->h_length);

        servaddr.sin_family = AF_INET;
        servaddr.sin_port = htons(atoi(argv[2]));

        if (connect(sockfd, (SA *) & servaddr, sizeof(servaddr)) < 0) {
                perror("connect error");
                exit(-1);
        }
   ChatNonBlock (sockfd);
}
ChatNonBlock(int sockfd)
{
```

```
        int             nread;
        char            buffer[512];
        int val;
        val = fcntl (sockfd, F_GETFL, 0);
        fcntl (sockfd, F_SETFL, val | O_NONBLOCK);
        val = fcntl (0, F_GETFL, 0);
        fcntl (0, F_SETFL, val | O_NONBLOCK);

        for (;;) {
                nread = read(0, buffer, 512);
                if (nread == 0)
                        exit(0);
                write(sockfd, buffer, nread);

                nread = read(sockfd, buffer, 512);
                if (nread == 0)
                        exit(0);
                write(1, buffer, nread);
                sleep(1);

        }
}
```

Usage Example:

```
% ChatServerSelect 10990 &
% ChatClientNonBlock localhost 10990
```

❖ Asynchronous

✓ Example 7: ChatClientSelect.c

```
int main(int argc, char **argv)
{
   int             sockfd;
   int             pid;
   struct sockaddr_in servaddr;
   struct hostent *hp, *gethostbyname();

   sockfd = socket(AF_INET, SOCK_STREAM, 0);

   bzero(&servaddr, sizeof(servaddr));
   hp = gethostbyname(argv[1]);
   bcopy(hp->h_addr, &(servaddr.sin_addr.s_addr), hp->h_length);

   servaddr.sin_family = AF_INET;
   servaddr.sin_port = htons(atoi(argv[2]));

   if (connect(sockfd, (SA *) & servaddr, sizeof(servaddr)) < 0) {
      perror("connect error");
      exit(-1);
   }
   AsynchIO(0, sockfd);
```

```
}

AsynchIO(int fd1, int fd2)
{

   fd_set          readset;
   char            buffer[512];
   int             nread;
   int             i;

   for (;;) {

      FD_SET(fd1, &readset);
      FD_SET(fd2, &readset);
      bzero(buffer, sizeof(buffer));

      select(MAXFD, &readset, NULL, NULL, NULL);

      if (FD_ISSET(fd1, &readset)) {
         nread = read(fd1, buffer, 512);
         if (nread == 0) exit(0);
         if (write(fd2, buffer, nread) == -1) exit(0);
      }
      if (FD_ISSET(fd2, &readset)) {
         nread = read(fd2, buffer, 512);
         if (nread == 0) exit(0);
         if (write(fd1, buffer, nread) == -1) exit(0);
      }
   }

}
```

Usage Example:

```
 % ChatServerSelect 10990 &
% ChatClientSelect localhost 10990
```

✓ Example 8: ChatServerSelect.c

```
int main(int argc, char **argv)
{
   int             i;
   strcpy(ServerPort, argv[1]);

   TCPServer.sin_family = AF_INET;
   TCPServer.sin_addr.s_addr = htonl(INADDR_ANY);
   TCPServer.sin_port = htons(atoi(ServerPort));

   for (i = 0; i <= MAXTCP; i++) connfd[i] = -1;

   TCPSocket = socket(AF_INET, SOCK_STREAM, 0);
   if (bind(TCPSocket, (SA *) & TCPServer, sizeof(TCPServer)) < 0) {
      close(TCPSocket);
```

```
        perror("binding TCPSocket socket");
        exit(-1);
    }
    listen(TCPSocket, 4);

    FD_ZERO(&allset);
    FD_SET(TCPSocket, &allset);

    for (;;) {
        readset = allset;
        select(MAXCLIENTS, &readset, NULL, NULL, NULL);

        if (FD_ISSET(TCPSocket, &readset)) {
            for (i = 0; i <= MAXTCP; i++) {
                if (connfd[i] == -1) {
                    connfd[i] = accept(TCPSocket, 0, 0);
                    FD_SET(connfd[i], &allset);
                    break;
                }
            }
        }
        for (i = 0; i <= MAXTCP; i++) {
            if ((connfd[i]) == -1) continue;
            if (FD_ISSET(connfd[i], &readset))
                handleTCP(i);
        }
    }

}

handleTCP(int index)
{

    int             nread;
    char            buffer[BUFSIZE];
    int             i;
    int             myIndex = index;
    bzero(buffer, sizeof(buffer));
    if ((nread = read(connfd[myIndex], buffer, sizeof(buffer))) < 0) {
        if (errno == EINTR) return;
        else perror("read connfd");
    }
    if (nread == 0) {
        connfd[myIndex] = -1;
        return;
    }
    for (i = 0; i <= MAXTCP; i++)
        if (connfd[i] != -1)
            write(connfd[i], buffer, nread);
}
```

❖ Error Handling

System calls return an error number or code when I/O request fails.

In Unix we use the C-lib function *perror* to translate the error number into text message.

✓ Example 9: error.c

```
main(int argc, char *argv[])
{
        int             fd;
        char           *fname;
        fname = argv[1];
        if ((fd = open(fname, O_RDWR)) == -1) {
          printf("errno: %d\n", errno);
                perror(fname);
                exit(-1);
        } else
         printf( "fd is %d\n", fd);
         printf("opened %s\n", fname);
}
```

Usage Example:

```
% error /tmp/testfile
```

I/O Protection

User process may accidentally or purposefully attempt to disrupt normal operation via illegal I/O instructions

- All I/O instructions defined to be privileged
- I/O must be performed via **system calls**

Kernel Data Structures

Kernel keeps state info for I/O components, including **open file tables**, **network connections**, **and character device state**

UNIX I/O Kernel Structure

Example 10: fdTypeTest.c

Usage Example:

```
% fdTypeTest 101990 &
% telnet localhost 10990
```

Deadlocks

References:

1. Abraham Silberschatz, Greg Gagne, and Peter Baer Galvin, "Operating System Concepts, Ninth Edition ", Chapter 7

7.1 System Model

- For the purposes of deadlock discussion, a system can be modeled as a collection of limited resources, which can be partitioned into different categories, to be allocated to a number of processes, each having different needs.
- Resource categories may include memory, printers, CPUs, open files, tape drives, CD-ROMS, etc.
- By definition, all the resources within a category are equivalent, and a request of this category can be equally satisfied by any one of the resources in that category. If this is not the case (i.e. if there is some difference between the resources within a category), then that category needs to be further divided into separate categories. For example, "printers" may need to be separated into "laser printers" and "color inkjet printers".
- Some categories may have a single resource.
- In normal operation a process must request a resource before using it, and release it when it is done, in the following sequence:
    1. Request - If the request cannot be immediately granted, then the process must wait until the resource(s) it needs become available. For example the system calls open( ), malloc( ), new( ), and request( ).
    2. Use - The process uses the resource, e.g. prints to the printer or reads from the file.
    3. Release - The process relinquishes the resource. so that it becomes available for other processes. For example, close( ), free( ), delete( ), and release( ).
- For all kernel-managed resources, the kernel keeps track of what resources are free and which are allocated, to which process they are allocated, and a queue of processes waiting for this resource to become available. Application-managed resources can be controlled using mutexes or wait( ) and signal( ) calls, (i.e. binary or counting semaphores.)
- A set of processes is deadlocked when every process in the set is waiting for a resource that is currently allocated to another process in the set (and which can only be released when that other waiting process makes progress.)

7.2 Deadlock Characterization

DEADLOCK WITH MUTEX LOCKS

Let's see how deadlock can occur in a multithreaded Pthread program using mutex locks. The `pthread_mutex_init()` function initializes an unlocked mutex. Mutex locks are acquired and released using `pthread_mutex_lock()` and `pthread_mutex_unlock()`, respectively. If a thread attempts to acquire a locked mutex, the call to `pthread_mutex_lock()` blocks the thread until the owner of the mutex lock invokes `pthread_mutex_unlock()`.

Two mutex locks are created in the following code example:

```
/* Create and initialize the mutex locks */
pthread_mutex_t first_mutex;
pthread_mutex_t second_mutex;

pthread_mutex_init(&first_mutex,NULL);
pthread_mutex_init(&second_mutex,NULL);
```

Next, two threads—thread_one and thread_two—are created, and both these threads have access to both mutex locks. thread_one and thread_two run in the functions do_work_one() and do_work_two(), respectively, as shown below:

```
/* thread_one runs in this function */
void *do_work_one(void *param)
{
   pthread_mutex_lock(&first_mutex);
   pthread_mutex_lock(&second_mutex);
   /**
    * Do some work
    */
   pthread_mutex_unlock(&second_mutex);
   pthread_mutex_unlock(&first_mutex);

   pthread_exit(0);
}

/* thread_two runs in this function */
void *do_work_two(void *param)
{
   pthread_mutex_lock(&second_mutex);
   pthread_mutex_lock(&first_mutex);
   /**
    * Do some work
    */
   pthread_mutex_unlock(&first_mutex);
   pthread_mutex_unlock(&second_mutex);

   pthread_exit(0);
}
```

In this example, `thread_one` attempts to acquire the mutex locks in the order (1) `first_mutex`, (2) `second_mutex`, while `thread_two` attempts to acquire the mutex locks in the order (1) `second_mutex`, (2) `first_mutex`. Deadlock is possible if `thread_one` acquires `first_mutex` while `thread_two` acquires `second_mutex`.

Note that, even though deadlock is possible, it will not occur if `thread_one` can acquire and release the mutex locks for `first_mutex` and `second_mutex` before `thread_two` attempts to acquire the locks. And, of course, the order in which the threads run depends on how they are scheduled by the CPU scheduler. This example illustrates a problem with handling deadlocks: it is difficult to identify and test for deadlocks that may occur only under certain scheduling circumstances.

New Sidebar in Ninth Edition

7.2.1 Necessary Conditions

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

7.2.2 Resource-Allocation Graph

- In some cases deadlocks can be understood more clearly through the use of **Resource-Allocation Graphs**, having the following properties:
  - A set of resource categories, { R1, R2, R3, . . ., RN }, which appear as square nodes on the graph. Dots inside the resource nodes indicate specific instances of the resource. (E.g. two dots might represent two laser printers.)
  - A set of processes, { P1, P2, P3, . . ., PN }
  - **Request Edges -** A set of directed arcs from Pi to Rj, indicating that process Pi has requested Rj, and is currently waiting for that resource to become available.
  - **Assignment Edges -** A set of directed arcs from Rj to Pi indicating that resource Rj has been allocated to process Pi, and that Pi is currently holding resource Rj.
  - Note that a **request edge** can be converted into an **assignment edge** by reversing the direction of the arc when the request is granted. (However note also that request edges point to the category box, whereas assignment edges emanate from a particular instance dot within the box.)
  - For example:

Figure 7.1 - Resource allocation graph

- If a resource-allocation graph contains no cycles, then the system is not deadlocked. (When looking for cycles, remember that these are ***directed*** graphs.) See the example in Figure 7.2 above.
- If a resource-allocation graph does contain cycles **AND** each resource category contains only a single instance, then a deadlock exists.
- If a resource category contains more than one instance, then the presence of a cycle in the resource-allocation graph indicates the *possibility* of a deadlock, but does not guarantee one. Consider, for example, Figures 7.3 and 7.4 below:

Figure 7.2 - Resource allocation graph with a deadlock

Figure 7.3 - Resource allocation graph with a cycle but no deadlock

7.3 Methods for Handling Deadlocks

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

7.4 Deadlock Prevention

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

7.4.1 Mutual Exclusion

- Shared resources such as read-only files do not lead to deadlocks.
- Unfortunately some resources, such as printers and tape drives, require exclusive access by a single process.

7.4.2 Hold and Wait

- To prevent this condition processes must be prevented from holding one or more resources while simultaneously waiting for one or more others. There are several possibilities for this:
    - Require that all processes request all resources at one time. This can be wasteful of system resources if a process needs one resource early in its execution and doesn't need some other resource until much later.
    - Require that processes holding resources must release them before requesting new resources, and then re-acquire the released resources along with the new ones in a single new request. This can be a problem if a process has partially completed an operation using a resource and then fails to get it re-allocated after releasing it.
    - Either of the methods described above can lead to starvation if a process requires one or more popular resources.

7.4.3 No Preemption

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

7.4.4 Circular Wait

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

7.5 Deadlock Avoidance

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

7.5.1 Safe State

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

Figure 7.6 - Safe, unsafe, and deadlocked state spaces.

- For example, consider a system with 12 tape drives, allocated as follows. Is this a safe state? What is the safe sequence?

	Maximum Needs	Current Allocation
P0	10	5
P1	4	2
P2	9	2

- What happens to the above table if process P2 requests and is granted one more tape drive?
- Key to the safe state approach is that when a request is made for resources, the request is granted only if the resulting allocation state is a safe one.

7.5.2 Resource-Allocation Graph Algorithm

- If resource categories have only single instances of their resources, then deadlock states can be detected by cycles in the resource-allocation graphs.
- In this case, unsafe states can be recognized and avoided by augmenting the resource-allocation graph with ***claim edges***, noted by dashed lines, which point from a process to a resource that it may request in the future.
- In order for this technique to work, all claim edges must be added to the graph for any particular process before that process is allowed to request any resources. (Alternatively, processes may only make requests for resources for which they have already established claim edges, and claim edges cannot be added to any process that is currently holding resources.)

- When a process makes a request, the claim edge Pi->Rj is converted to a request edge. Similarly when a resource is released, the assignment reverts back to a claim edge.
- This approach works by denying requests that would produce cycles in the resource-allocation graph, taking claim edges into effect.
- Consider for example what happens when process P2 requests resource R2:

Figure 7.7 - Resource allocation graph for deadlock avoidance

- The resulting resource-allocation graph would have a cycle in it, and so the request cannot be granted.

Figure 7.8 - An unsafe state in a resource allocation graph

7.5.3 Banker's Algorithm

- For resource categories that contain more than one instance the resource-allocation graph method does not work, and more complex (and less efficient) methods must be chosen.
- The Banker's Algorithm gets its name because it is a method that bankers could use to assure that when they lend out resources they will still be able to satisfy all their clients. (A banker won't loan out a little money to start building a house unless they are assured that they will later be able to loan out the rest of the money to finish the house.)
- When a process starts up, it must state in advance the maximum allocation of resources it may request, up to the amount available on the system.
- When a request is made, the scheduler determines whether granting the request would leave the system in a safe state. If not, then the process must wait until the request can be granted safely.

- The banker's algorithm relies on several key data structures: (where n is the number of processes and m is the number of resource categories.)
  - Available[m] indicates how many resources are currently available of each type.
  - Max[n][m] indicates the maximum demand of each process of each resource.
  - Allocation[n][m] indicates the number of each resource category allocated to each process.
  - Need[n][m] indicates the remaining resources needed of each type for each process. (Note that Need[i][j] = Max[i][j] - Allocation[i][j] for all i, j.)
- For simplification of discussions, we make the following notations / observations:
  - One row of the Need vector, Need[i], can be treated as a vector corresponding to the needs of process i, and similarly for Allocation and Max.
  - A vector X is considered to be <= a vector Y if X[i] <= Y[i] for all i.

7.5.3.1 Safety Algorithm

- In order to apply the Banker's algorithm, we first need an algorithm for determining whether or not a particular state is safe.
- This algorithm determines if the current state of a system is safe, according to the following steps:
  1. Let Work and Finish be vectors of length m and n respectively.
     - Work is a working copy of the available resources, which will be modified during the analysis.
     - Finish is a vector of booleans indicating whether a particular process can finish. (or has finished so far in the analysis.)
     - Initialize Work to Available, and Finish to false for all elements.
  2. Find an i such that both (A) Finish[i] == false, and (B) Need[i] < Work. This process has not finished, but could with the given available working set. If no such i exists, go to step 4.
  3. Set Work = Work + Allocation[i], and set Finish[i] to true. This corresponds to process i finishing up and releasing its resources back into the work pool. Then loop back to step 2.
  4. If finish[i] == true for all i, then the state is a safe state, because a safe sequence has been found.
- (JTB's Modification:
  1. In step 1. instead of making Finish an array of booleans initialized to false, make it an array of ints initialized to 0. Also initialize an int s = 0 as a step counter.
  2. In step 2, look for Finish[i] == 0.
  3. In step 3, set Finish[i] to ++s. S is counting the number of finished processes.
  4. For step 4, the test can be either Finish[i] > 0 for all i, or s >= n. The benefit of this method is that if a safe state exists, then Finish[ ] indicates one safe sequence (of possibly many.))

7.5.3.2 Resource-Request Algorithm (The Bankers Algorithm)

- Now that we have a tool for determining if a particular state is safe or not, we are now ready to look at the Banker's algorithm itself.
- This algorithm determines if a new request is safe, and grants it only if it is safe to do so.
- When a request is made (that does not exceed currently available resources), pretend it has been granted, and then see if the resulting state is a safe one. If so, grant the request, and if not, deny the request, as follows:

1. Let Request[n][m] indicate the number of resources of each type currently requested by processes. If Request[i] > Need[i] for any process i, raise an error condition.
2. If Request[i] > Available for any process i, then that process must wait for resources to become available. Otherwise the process can continue to step 3.
3. Check to see if the request can be granted safely, by pretending it has been granted and then seeing if the resulting state is safe. If so, grant the request, and if not, then the process must wait until its request can be granted safely.The procedure for granting a request (or pretending to for testing purposes) is:
   - Available = Available - Request
   - Allocation = Allocation + Request
   - Need = Need - Request

7.5.3.3 An Illustrative Example

- Consider the following situation:

	Allocation	Max	Available	Need
	A B C	A B C	A B C	A B C
P_0	0 1 0	7 5 3	3 3 2	7 4 3
P_1	2 0 0	3 2 2		1 2 2
P_2	3 0 2	9 0 2		6 0 0
P_3	2 1 1	2 2 2		0 1 1
P_4	0 0 2	4 3 3		4 3 1

- And now consider what happens if process P1 requests 1 instance of A and 2 instances of C. (Request[1] = (1, 0, 2))

	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	

- What about requests of (3, 3,0) by P4? or (0, 2, 0) by P0? Can these be safely granted? Why or why not?

7.6 Deadlock Detection

- If deadlocks are not avoided, then another approach is to detect when they have occurred and recover somehow.
- In addition to the performance hit of constantly checking for deadlocks, a policy / algorithm must be in place for recovering from deadlocks, and there is potential for lost work when processes must be aborted or have their resources preempted.

7.6.1 Single Instance of Each Resource Type

- If each resource category has a single instance, then we can use a variation of the resource-allocation graph known as a ***wait-for graph***.
- A wait-for graph can be constructed from a resource-allocation graph by eliminating the resources and collapsing the associated edges, as shown in the figure below.
- An arc from Pi to Pj in a wait-for graph indicates that process Pi is waiting for a resource that process Pj is currently holding.

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

- As before, cycles in the wait-for graph indicate deadlocks.
- This algorithm must maintain the wait-for graph, and periodically search it for cycles.

7.6.2 Several Instances of a Resource Type

- The detection algorithm outlined here is essentially the same as the Banker's algorithm, with two subtle differences:
  - In step 1, the Banker's Algorithm sets Finish[i] to false for all i. The algorithm presented here sets Finish[i] to false only if Allocation[i] is not zero. If the currently allocated resources for this process are zero, the algorithm sets Finish[i] to true. This is essentially assuming that IF all of the other processes can finish, then this process can finish also. Furthermore, this algorithm is specifically looking for which processes are

involved in a deadlock situation, and a process that does not have any resources allocated cannot be involved in a deadlock, and so can be removed from any further consideration.
  - Steps 2 and 3 are unchanged
  - In step 4, the basic Banker's Algorithm says that if Finish[i] == true for all i, that there is no deadlock. This algorithm is more specific, by stating that if Finish[i] == false for any process Pi, then that process is specifically involved in the deadlock which has been detected.
- (Note: An alternative method was presented above, in which Finish held integers instead of booleans. This vector would be initialized to all zeros, and then filled with increasing integers as processes are detected which can finish. If any processes are left at zero when the algorithm completes, then there is a deadlock, and if not, then the integers in finish describe a safe sequence. To modify this algorithm to match this section of the text, processes with allocation = zero could be filled in with N, N - 1, N - 2, etc. in step 1, and any processes left with Finish = 0 in step 4 are the deadlocked processes.)
- Consider, for example, the following state, and determine if it is currently deadlocked:

	Allocation	*Request*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 0	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

- Now suppose that process P2 makes a request for an additional instance of type C, yielding the state shown below. Is the system now deadlocked?

	Allocation	*Request*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 1	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

7.6.3 Detection-Algorithm Usage

- When should the deadlock detection be done? Frequently, or infrequently?
- The answer may depend on how frequently deadlocks are expected to occur, as well as the possible consequences of not catching them immediately. (If deadlocks are not removed immediately when they occur, then more and more processes can "back up" behind the deadlock, making the eventual task of unblocking the system more difficult and possibly damaging to more processes.)

- There are two obvious approaches, each with trade-offs:
  1. Do deadlock detection after every resource allocation which cannot be immediately granted. This has the advantage of detecting the deadlock right away, while the minimum number of processes are involved in the deadlock. (One might consider that the process whose request triggered the deadlock condition is the "cause" of the deadlock, but realistically all of the processes in the cycle are equally responsible for the resulting deadlock.) The down side of this approach is the extensive overhead and performance hit caused by checking for deadlocks so frequently.
  2. Do deadlock detection only when there is some clue that a deadlock may have occurred, such as when CPU utilization reduces to 40% or some other magic number. The advantage is that deadlock detection is done much less frequently, but the down side is that it becomes impossible to detect the processes involved in the original deadlock, and so deadlock recovery can be more complicated and damaging to more processes.
  3. (As I write this, a third alternative comes to mind: Keep a historical log of resource allocations, since that last known time of no deadlocks. Do deadlock checks periodically (once an hour or when CPU usage is low?), and then use the historical log to trace through and determine when the deadlock occurred and what processes caused the initial deadlock. Unfortunately I'm not certain that breaking the original deadlock would then free up the resulting log jam.)

7.7 Recovery From Deadlock

- There are three basic approaches to recovery from deadlock:
  1. Inform the system operator, and allow him/her to take manual intervention.
  2. Terminate one or more processes involved in the deadlock
  3. Preempt resources.

7.7.1 Process Termination

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

7.7.2 Resource Preemption

- When preempting resources to relieve deadlock, there are three important issues to be addressed:
  1. **Selecting a victim** - Deciding which resources to preempt from which processes involves many of the same decision criteria outlined above.
  2. **Rollback** - Ideally one would like to roll back a preempted process to a safe state prior to the point at which that resource was originally allocated to the process. Unfortunately

it can be difficult or impossible to determine what such a safe state is, and so the only safe rollback is to roll back all the way back to the beginning. (I.e. abort the process and make it start over.)

3. **Starvation** - How do you guarantee that a process won't starve because its resources are constantly being preempted? One option would be to use a priority system, and increase the priority of a process every time its resources get preempted. Eventually it should get a high enough priority that it won't get preempted any more.

7.8 Summary

Protection

Goals of Protection

- Operating system consists of a collection of **objects**: hardware or software
- Each object has a **unique name** and can be accessed through a **well-defined set of operations**
- **Protection problem** - ensure that **each object** is **accessed correctly** and only by those processes that are allowed to do so

✓ **Protection guiding principle** – ***principle of least privilege***:
*Programs, users and systems should be given **just enough** privileges to perform their tasks*

Domain Structure

- **Access-right** = <***object-name***, ***rights-set***>
  where ***rights-set*** is a subset of all valid operations that can be performed on the object.

- **Domain** = set of access-rights

Domain Implementation in UNIX

- Domain = user-id
- Domain switch accomplished via file system
  - ✓ Each file has associated with it a domain bit (setuid bit)
  - ✓ When file is executed and ***setuid*** is on, then user-id is set to owner of the file being executed.
  - ✓ When execution completes user-id is reset

Examples:

setuid (set user id):

Allows a user to be another user during the execution of a program.

For example:

[antares] ~> cd /usr/bin
[antares] /usr/bin> ls -l passwd

```
-r-sr-sr-x   1 root     sys        27228 Aug 16
2007 passwd
```

[antares] /usr/bin> ls -l | grep sr

```
-r-sr-xr-x   1 root     bin        31408 Jan 22
2005 login
-r-sr-sr-x   1 root     sys        27228 Aug 16
2007 passwd
.........
```

Run *passwd* in one window:

```
[antares] ~> passwd
```

Run in another window:

```
[antares] ~> ps -a -o user,ruser,comm

USER   RUSER   COMMAND
root   cs471w passwd
```

How to setuid of a program:

```
[antares] ~>  chmod  u+s  program
```

Example:

```
% cd /home/cs471w/public_html/code/final-src/chap14
% setuidf
```

Access Matrix

- View protection as a matrix (*access matrix*)
- Rows represent **domains**
- Columns represent **objects**
- ***Access(i, j)*** is the set of operations that a process executing in ***Domain i*** can invoke on ***Object j***

domain \ object	F_1	F_2	F_3	printer
D_1	read		read	
D_2				print
D_3		read	execute	
D_4	read write		read write	

Use of Access Matrix

- If a process in Domain Di tries to do ***"op"*** on object Oj then ***"op"*** must be in the access matrix
- **Access matrix design**: separate ***Mechanism*** from ***Policy:***
  Mechanism
  It ensures that the matrix is only manipulated by authorized agents

and rules are strictly enforced

Policy

User dictates policy:

Who can access what object and in what mode

Implementation of Access Matrix

- Each column = **Access-control list** for one object
  Defines who can perform what operation.

  Domain 1 = Read, Write
  Domain 2 = Read
  Domain 3 = Read

M

- Each Row = **Capability List** (like a key)
  Fore each domain, what operations allowed on what objects.

  Object 1 – Read
  Object 4 – Read, Write, Execute
  Object 5 – Read, Write, Delete, Copy

Access Matrix With Domains as Objects

transfer *– switch from domain* D_i *to* D_j

domain \ object	F_1	F_2	F_3	laser printer	D_1	D_2	D_3	D_4
D_1	read		read			switch		
D_2				print			switch	switch
D_3		read	execute					
D_4	read write		read write		switch			

Security

Objectives

- To discuss security **threats** and **attacks**
- To explain the fundamentals of **encryption** & **authentication**.
- To examine the **uses of cryptography** in computing
- To describe the various **countermeasures to security attacks**

Security Violations

- **Categories**
  - ✓ **Breach of confidentiality**
  - ✓ **Breach of integrity**
  - ✓ **Breach of availability**
  - ✓ **Theft of service**
  - ✓ **Denial of service**

- **Methods**
  - ✓ **Masquerading (breach authentication)**
  - ✓ **Replay attack**
  - ✓ **Message modification**
  - ✓ **Man-in-the-middle attack**

Standard Security Attacks

Security Measure Levels

- Security must occur at four levels to be effective:
  - ✓ Physical
  - ✓ Human: Avoid social engineering, phishing.
  - ✓ Operating System
  - ✓ Network
- Security is as weak as the weakest link in the chain

Program Threats

Trojan Horse

Code segment that misuses its environment:
Spyware, pop-up browser windows, covert channels

Trap Door

Specific user identifier or password that circumvents normal security procedures

Logic Bomb

Program that initiates a security incident under certain circumstances

Stack & Buffer Overflow

Exploits a bug in a program (overflow either the stack or memory buffers)

C Program with Buffer-overflow Condition

Bufferoverflow.c: strncpy vs. strcpy

```
#define MemSize 10
int
main(int argc, char *argv[])
{
     char            buf1[10];
     char            buf2[10];
     char           *buf3 = malloc(MemSize);
     char           *buf4 = malloc(MemSize);

        strncpy(buf1, argv[1], sizeof(buf1));
        printf("stack buf1 ncpy: %s\n", buf1);

        strcpy(buf2, argv[1]);
        printf("stack buf2: %s\n", buf2);

        strncpy(buf3, argv[1], MemSize);
        printf("heap buf3 ncpy:%s\n", buf3);

        strcpy(buf4, argv[1]);
        printf("heap buf4: %s\n", buf4);

}
```

To Run:

```
% bufferoverflow 0123456789abdelwahab
stack buf1 ncpy: 0123456789
stack buf2: 0123456789abdelwahab
heap buf3 ncpy:0123456789
heap buf4: 0123456789abdelwahab

% bufferoverflow 0123456789abdelwahab0123456789
stack buf1 ncpy: 0123456789
```

```
stack buf2: 0123456789abdelwahab0123456789
Segmentation fault (core dumped)
```

System and Network Threats

Internet worm

Exploited UNIX networking features (remote access) and bugs in ***finger*** and ***sendmail*** programs

Port scanning

Automated attempt to connect to a range of ports on one or a range of IP addresses

Denial of Service

- ✓ Overload the targeted computer to prevent it from doing any useful work
- ✓ Distributed denial-of-service (**DDOS**) come from multiple sites at once

Secure Communication over Insecure Medium

Symmetric Encryption

plaintext >>>>>>>>>> (encryption) ciphertext

|

key

|

ciphertext >>>>>>>>> (decryption) plaintext

- ✓ Same key used to encrypt and decrypt
- ✓ **DES** is most commonly used symmetric block-encryption algorithm (created by US Govt)
- ✓ **Triple-DES** considered more secure
- ✓ Advanced Encryption Standard (**AES**)

Example: XOR encryption/decryption

```
Message M = 7 (111)
Key  K = 4 (100)

C = E(K, M) = M XOR K = 111 XOR 100 = 011 = 3
M = D(K, C) = C XOR K = 011 XOR 100 = 111 = 7
```

Asymmetric Encryption

Each individual has two keys:

✓ ***private key (not revealed to anyone)***

✓ ***public key (make it known to everyone)***

Example:

-

Public Key (5, 91) &

Private Key (29, 91)

M = 69.

C = 69 ** 5 mod 91 = 62
M = 62 ** 29 mod 91 = 69

Authentication – Hash Functions

Also known as: message digest, fingerprint, one-way function

The hash of a message m, *h=H(m)* has the following properties:

- ✓ Given *m*, it is easy to compute *h*.
- ✓ Given *h*, it is hard to compute *m*.
- ✓ Given *m*, it is hard to find another *m'* such that *H(m) = H(m')*.

- ✓ It is hard to find *m1* and *m2* such that *H(m1) = H(m2)*.

Common message-digest functions include **MD5**, which produces a **128-bit hash**,
and **SHA-1**, which outputs a **160-bit hash**

Authentication

MAC message-authentication code

Send (m,h) where h = H(m|K) >>>>> Got (m,h) is OK if h = H (m|K)

K is the shared secret

Digital Signature

```
                (signing)
plaintext       >>>>>>>>>>        ciphertext
                    |
                private key

                Public  key
                    |
ciphertext      >>>>>>>>>>        plaintext
                (verification)
```

Key Distribution

Digital Certificates: Proof of who owns a public key

- ✓ It is a Public key digitally signed by a trusted party
- ✓ Trusted party receives proof of identification from entity and certifies that public key belongs to entity
- ✓ Certificate authority (**CA**) are trusted party – their public keys included with web browser distributions

Example – SSL

-

- ✓ SSL – Secure Socket Layer.
- ✓ Used between web servers and browsers for secure communication (credit card numbers)
- ✓ The server is verified with a **certificate** assuring the client that it is talking to the correct server
- ✓ Asymmetric cryptography used to establish a secure **session key** (symmetric encryption)
  for bulk of communication during session

User Authentication

- User identity most often established through **passwords**,
- Passwords must be kept secret
  - ✓ **Frequent** change of passwords
  - ✓ Use of "**non-guessable**" passwords