Part 5

PROCESSES AND THREADS

Introduction to UNIX

- **UNIX** is a powerful computer operating system.

- **UNIX**, like other operating systems, is a layer between the hardware and the applications that run on the computer.

- **UNIX** provides a standard set of libraries and applications that developers and users can use. This standard interface allows application portability and facilitates user familiarity with the interface.

Unix Implementations

- **SunOS/Solaris:** Sun Microsystems, SVR4
- **Digital Unix**: Digital/Compaq
- **HP-UX:** Hewlett Packard, SVR2
- **Irix:** Silicon Graphics, SVR4
- **UNICOS:** Cray
- **AIX:** IBM, similar to SVR4
- **NetBSD, FreeBSD:** UC Berkeley
- SCO-Unix, Xenix, ...
- **Linux:** Linus Torvalds, PC based

UNIX Philosophy

- Simple, commands based (academically designed, programmer oriented).
  - The original UNIX systems were very small. Same applies to Linux Kernel.
  - Each command attempted to do one thing well.

- Commands connected through pipes.
  - To turn the simple commands into a powerful toolset, UNIX enables the user to use the output of one command as the input to another.
  - This connection is called a pipe. Now windows OS adopted as well.

UNIX Philosophy (Cont'd)

- A (mostly) common single letter reference for parameter settings to command.
  - Each command has actions that can be controlled with options, which are specified by a hyphen followed by a single letter option.

- No file types.
  - UNIX pays no attention to the contents of a file (except when you try to run a file as a command). Windows did better.

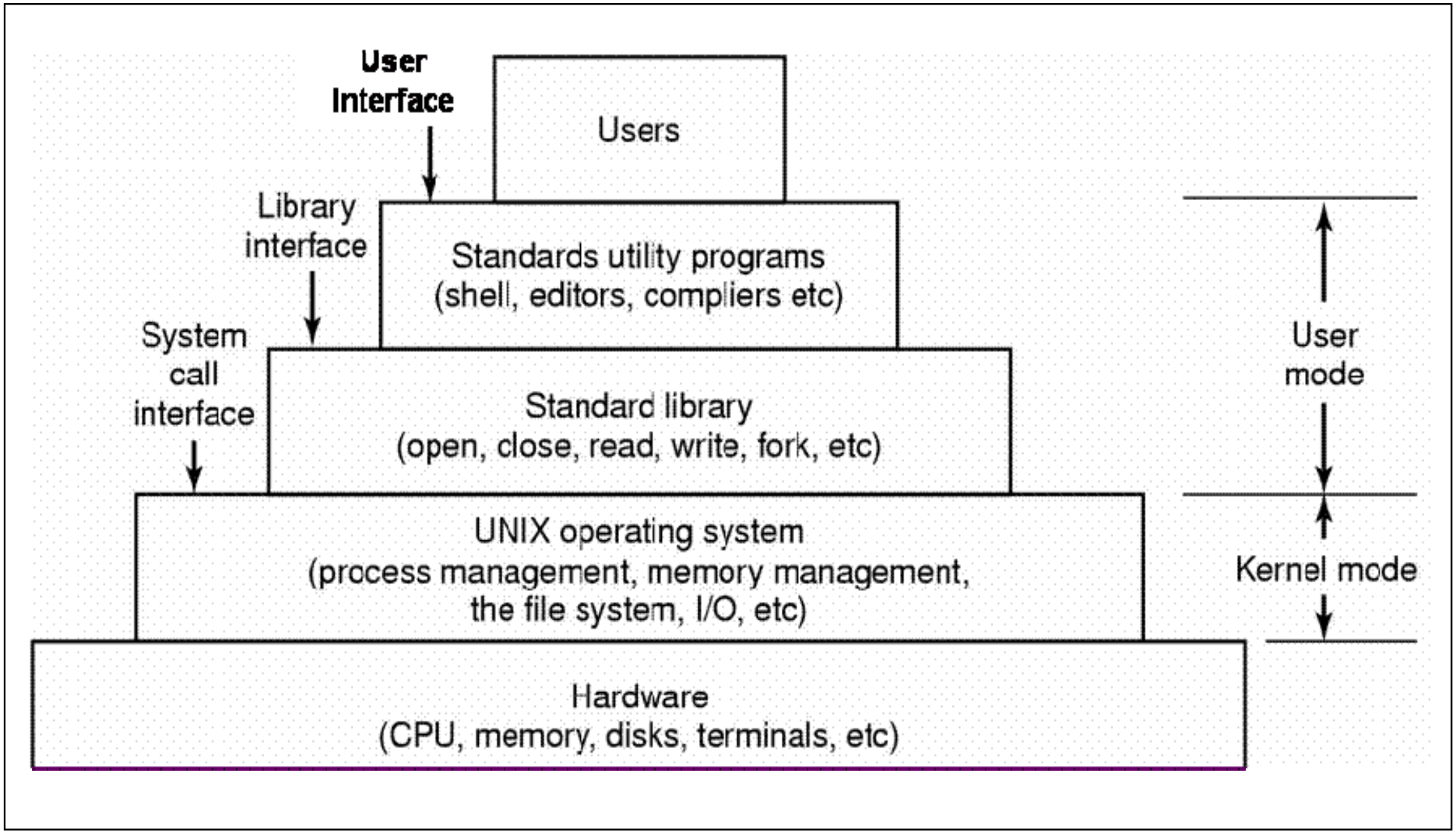
Linux

- Initial version written by Linus Torvalds, a Finnish CS student . He posted an early version of Linux on the Internet in 1991.
- With the effort of thousands of open source activists, it is now one of the most challenging operating systems.
- Linux is a multi-user, multitasking system with a full set of UNIX compatible tools.
- Linux has developed into a full-scale OS which include much of the UNIX functionality for PCs. Now, it can also run on a variety of platforms.
- Many Versions of Linux - Redhat, Fedroa, Debian, SuSE and MandrakeSoft.

Unix Features

- Portability (much better than previous systems)
- Multi-process architecture (multitasking)
- Multi-user capability
- Ability to initiate asynchronous processes
- A hierarchical file system
- Everything is a file (device independent I/O operations)
- User interface: Shell; selectable per user basis
- Designed by programmers for programmers.

The Layers of a UNIX system



Kernel

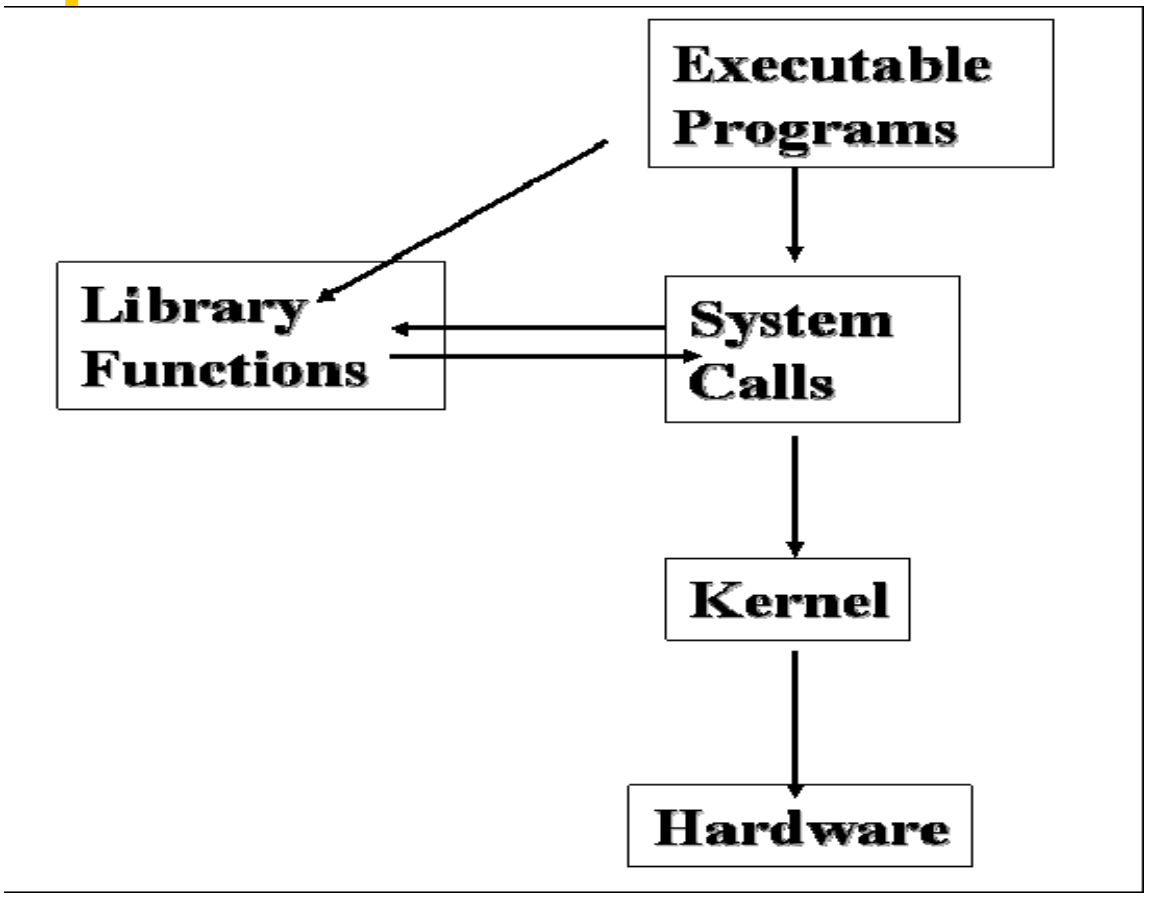
- Part of UNIX OS that contains code for:
  - Controlling execution of processes (creation, termination, suspension, communication),
  - Scheduling processes fairly for execution on the CPU,
  - Allocating main memory for exec of processes,
  - Allocating secondary memory for efficient storage and retrieval of user data,
  - Handling peripherals such as terminals, tape drives, disk drives and network devices.

Kernel (Cont'd)

- Kernel loaded into memory and runs until the system is turned off or crashes.
- Mostly written in C with some assembly language written for efficiency reasons.
- User programs make use of kernel services via the system call interface.
- Provides its services transparently.

Talking to Kernel

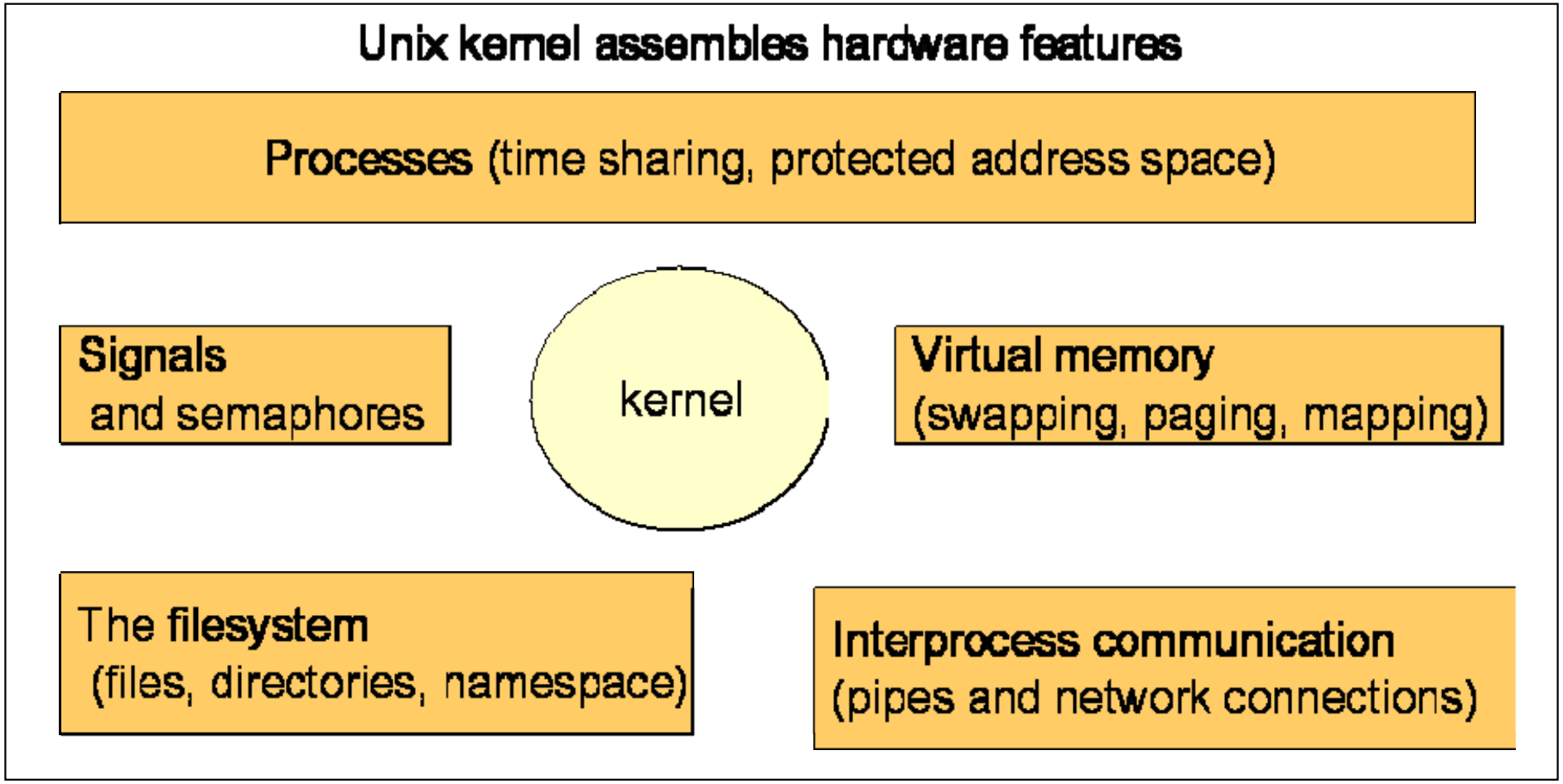
- Processes access kernel facilities via **system calls.**
- Peripherals communicate with the kernel via **hardware interrupts**.

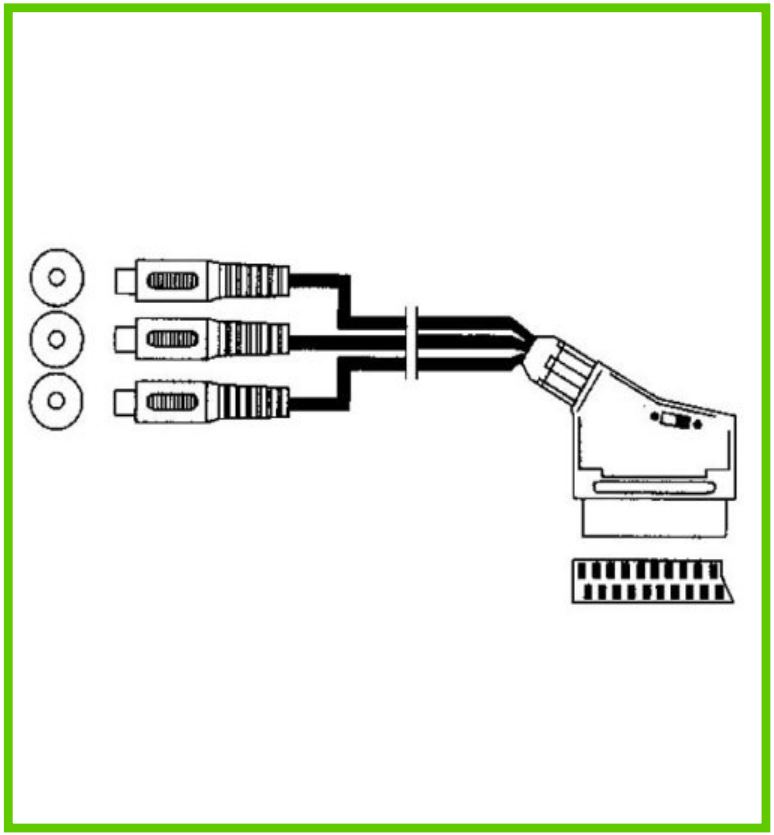


Kernel Subsystems

- **File (I/O) system**
  - Directory hierarchy, regular files, peripherals
  - Multiple file systems
- **Process management**
  - How processes share CPU, memory and signals
- **Inter-process Communication**
- **Memory management**
- **Signals:**
  - Signals are widely used for one process to send information (alert, notation) to another process.
- **Semaphores**:
  - In the Kernel, semaphore is also used with wait queues to schedule the processes. This semaphore is not open for users.

Kernel Subsystems (Cont'd)

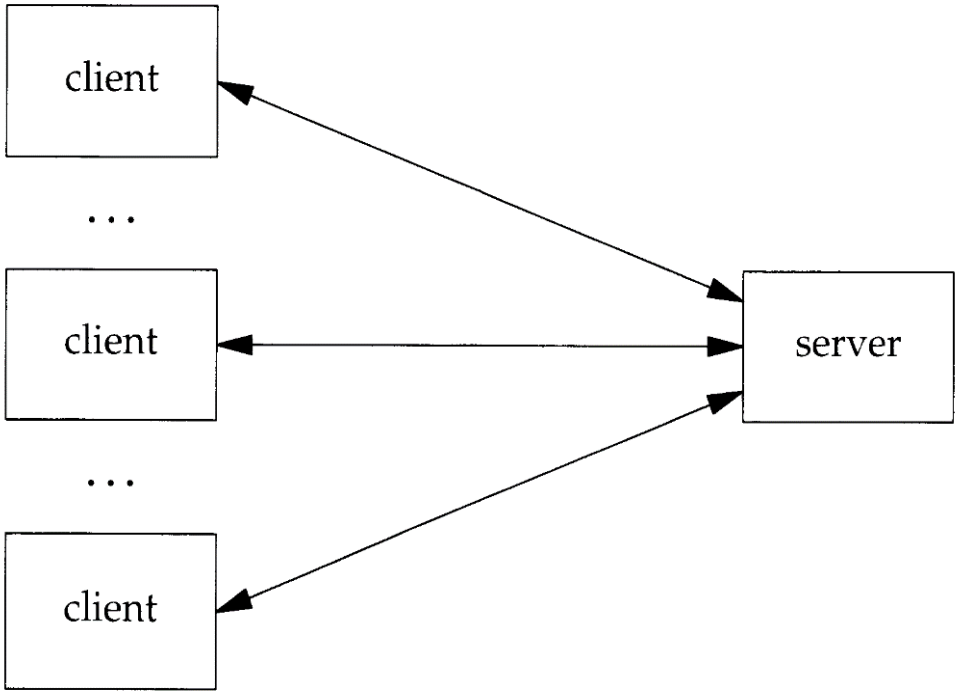




Sockets

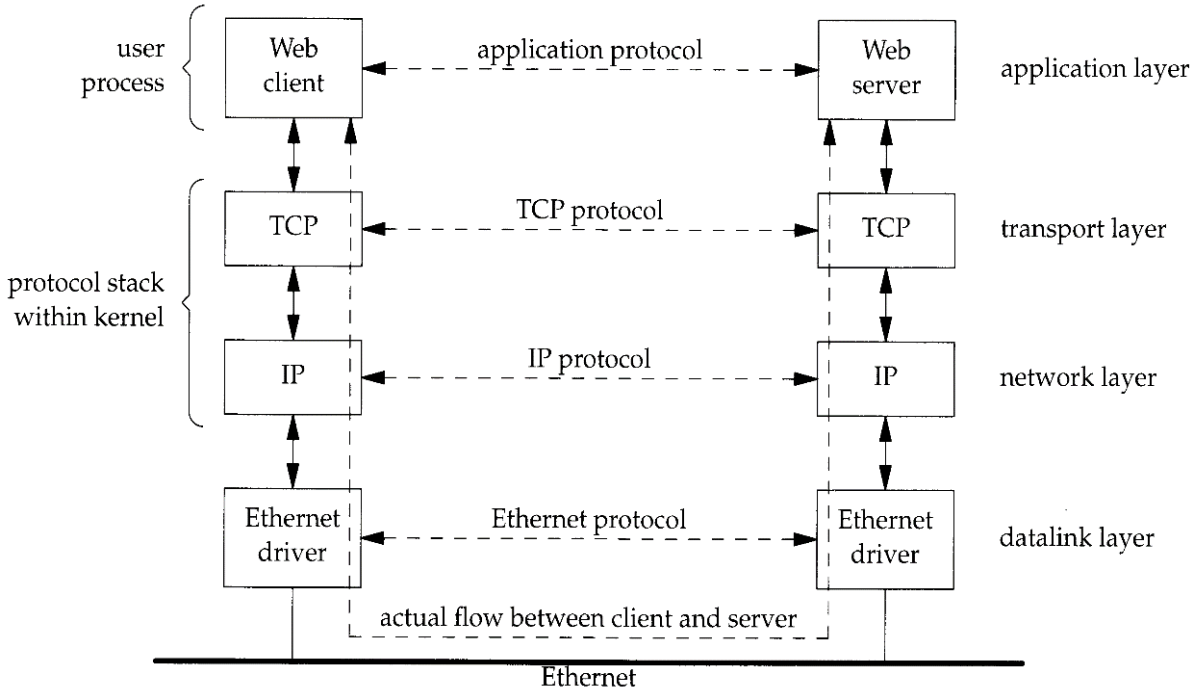
Sockets

- Server handling multiple clients at the same time



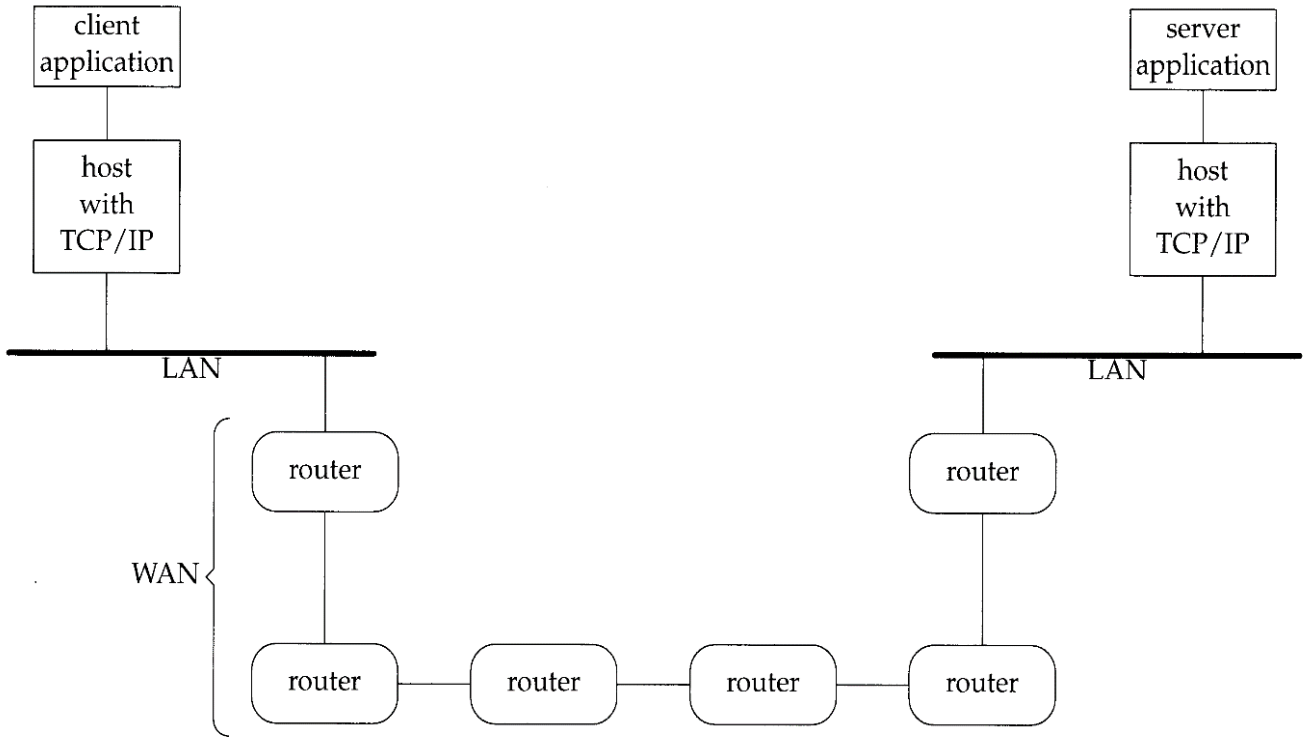
Sockets

- Client and sever on the same Ethernet communicating with TCP



Sockets

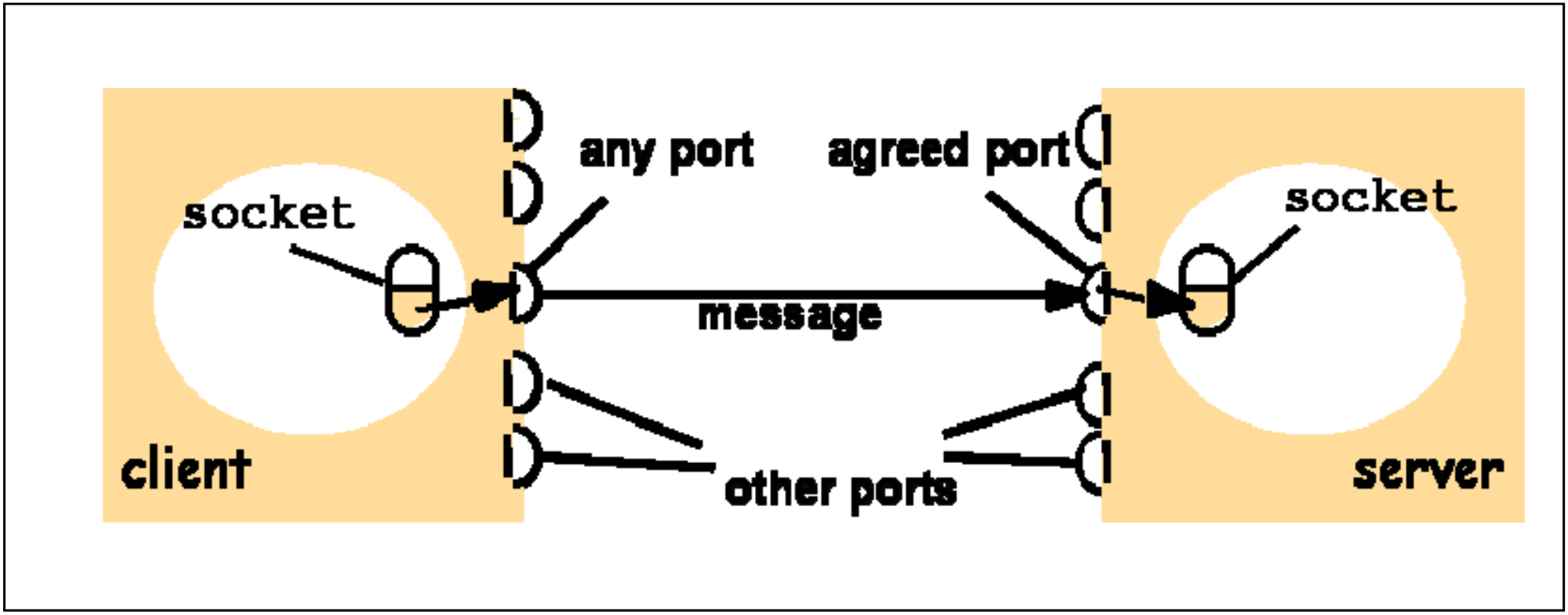
- Client and sever on different LANs connected through a WAN



Sockets

- Socket is an abstraction for an end point of communication that can be manipulated with a file descriptor.
- It is an abstract object from which messages are sent and received.
- Sockets are created within a communication domain just as files are created within a file system.

Sockets (Cont'd)



Sockets (Cont'd)

- Unlike pipes, sockets are initially proposed bi-directional. (some type of pipes are later enhanced to bi-directional)
- Sockets are now widely adopted by most major operating systems.
  - Sockets system calls were tightly-coupled with BSD networking architecture in the OS kernel.
  - Recent versions of sockets (Sun OS and SVR4) are implemented as library functions in user-space to improve flexibility.
  - Sockets were originally developed for TCP/IP protocols. Later generalized to include other protocol families as well.

IPC Using Sockets

- Inter-process communication via sockets is based on client-server model:

- The steps involved in establishing a socket:
  - **Server side:**
    - ✓ Create a socket.
    - ✓ Bind the socket to an address.
    - ✓ Listen for connections.
    - ✓ Accept a connection.
    - ✓ Send and receive data.
  - **Client side:**
    - ✓ Create a socket.
    - ✓ Connect the socket to the address of the server.
    - ✓ Send and receive data.

Socket Types

- Two processes can communicate with each other only if their sockets are of the same type and in the same domain.
- There are several types of sockets currently available:
  - **Stream socket**
    - ✓ **Supports delivery of bi-directional, reliable, sequenced, and unduplicated of byte-stream data.**
    - ✓ **Metaphor: A network pipe.**
  - **Datagram socket**
    - ✓ **Supports delivery of unreliable, bi-directional and un-sequenced datagram.**
    - ✓ **Metaphor: sending a letter.**
  - **Raw socket**
    - ✓ **Allows user-defined protocols that interface with IP.**

Socket Types (Cont'd)

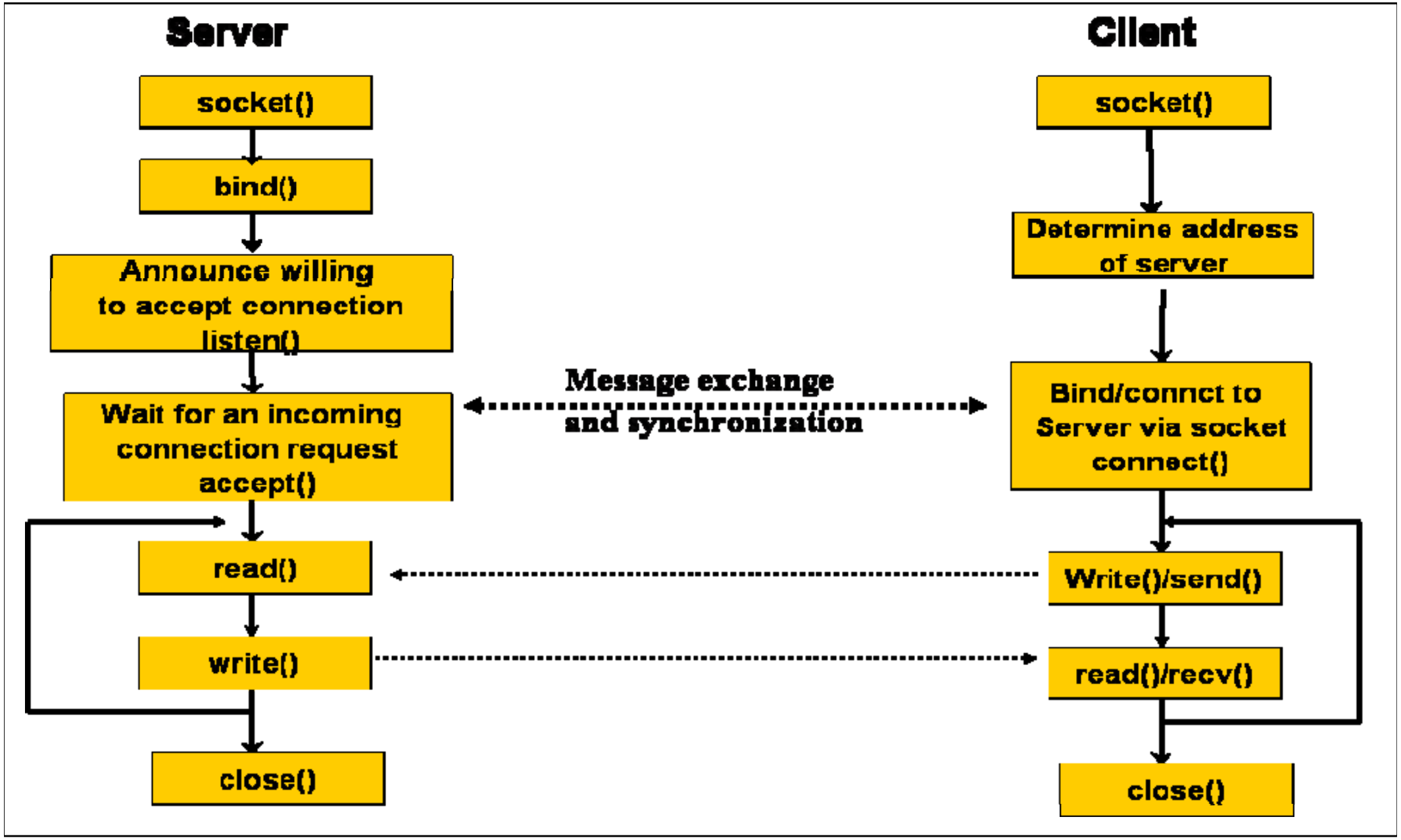
- **Sequenced packet stream socket**
  - ✓ **Supports reliable bi-directional delivery of record-oriented data.**
  - ✓ **Metaphor: sending a letter.**
- **Reliably delivered message socket**
  - ✓ **Supports reliable datagram.**
  - ✓ **Metaphor: sending a registered letter.**

Socket Examples

C code for two simple client and server are provided:

- **Connection-oriented**
  - ✓ **TCP Server-Client**
- **Connectionless**
  - ✓ **UDP Server-Client**

Connection-oriented Sockets



Basic Socket Calls

- **socket()**
  - ✓ Creates and opens a socket and returns a descriptor.
  - ✓ **int socket (int domain/family, int type, int protocol);**
- **bind()**
  - ✓ Associates a local address (eg an IP address, address family, and port number) to an unnamed socket.
  - ✓ **int bind (int s, struct sockaddr *addr, int addrlen);**
- **close()**
  - ✓ Closes a socket.
  - ✓ **int close (int s);**
- **shutdown()**
  - ✓ Shutdowns part or all of full-duplex connection.
  - ✓ **int shutdown (int s, int how);**

Basic Socket Calls (Cont'd)

- **getsockname()**
  - Returns address info describing the local socket.
  - **int getsockname(int s,struct sockaddr *addr,int *addrlenptr);**
- **getpeername()**
  - Returns address info describing the connected peer socket.
  - **int getpeername (int s, struct sockaddr *addr, int *addrlenptr);**
- **write()** (byte level based)
  - Sends a message to a socket.
  - **int write (int s, char *msg, int len);**
- **send()** (stream based)
  - Sends a message to a socket.
  - **int send (int s, char *msg, int len, int flags);**

Example: Server daytimetcpsrv.c

```
1 #include        <sys/socket.h>  /* basic socket definitions */
2 #include        <sys/types.h>   /* basic system data types */
3 #include        <arpa/inet.h>   /* inet(3) functions */
4 #include        <stdio.h>
5 #include        <stdlib.h>
6 #include        <time.h>
7
8 #define MAXLINE         4096    /* max text line length */
9
10 int main(int argc, char **argv)
11 {
12          int                     listenfd, connfd;
13          struct sockaddr_in      servaddr;
14          char                    buff[MAXLINE];
15          time_t                  ticks;
16
17          listenfd = socket(AF_INET, SOCK_STREAM, 0);
18
```

cc –o daytimetcpsrv daytimetcpsrv.c

Example: Server daytimetcpsrv.c

```
19          bzero(&servaddr, sizeof(servaddr));
20          servaddr.sin_family      = AF_INET;
21          servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
22          servaddr.sin_port        = htons(6060); /* daytime server
    */
23
24          bind(listenfd, (struct sockaddr *) &servaddr,
    sizeof(servaddr));
25
26          listen(listenfd, 1024);
27
28          for ( ; ; ) {
29                  connfd = accept(listenfd, (struct sockaddr *) NULL,
    NULL);
30
31                  ticks = time(NULL);
32                  sprintf(buff, "%.24s\r\n", ctime(&ticks));
33                  write(connfd, buff, strlen(buff));
34
35                  close(connfd);
36          }
37  }
```

cc –o daytimetcpsrv daytimetcpsrv.c

Example: Client daytimetcpcli.c

```
1 #include        <sys/socket.h>  /* basic socket definitions */
2 #include        <sys/types.h>   /* basic system data types */
3 #include        <arpa/inet.h>   /* inet(3) functions */
4 #include        <stdio.h>
5 #include        <stdlib.h>
6
7 #define MAXLINE         4096    /* max text line length */
8
9 int main(int argc, char **argv)
10 {
11          int                     sockfd, n;
12          char                    recvline[MAXLINE + 1];
13          struct sockaddr_in      servaddr;
14
15          if (argc != 2){
16                  printf("usage: a.out <IPaddress>");
17                  exit(0);
18          }
19
```

cc –o daytimetcpcli daytimetcpcli.c

Example: Client daytimetcpcli.c

(Cont'd)

```
20          sockfd = socket(AF_INET, SOCK_STREAM, 0);
21
22          bzero(&servaddr, sizeof(servaddr));
23          servaddr.sin_family = AF_INET;
24          servaddr.sin_port   = htons(6060);      /* daytime
   server */
25          inet_pton(AF_INET, argv[1], &servaddr.sin_addr);
26
27          connect(sockfd, (struct sockaddr *) &servaddr,
   sizeof(servaddr));
28
29          while ( (n = read(sockfd, recvline, MAXLINE)) > 0) {
30                  recvline[n] = 0;        /* null terminate */
31                  fputs(recvline, stdout);
32          }
33
34          exit(0);
35  }
```

cc –o daytimetcpcli daytimetcpcli.c

Example: Server daytimetcpsrvport.c

```
1 #include        <sys/socket.h>  /* basic socket definitions */
2 #include        <sys/types.h>   /* basic system data types */
3 #include        <arpa/inet.h>   /* inet(3) functions */
4 #include        <stdio.h>
5 #include        <stdlib.h>
6 #include        <time.h>
7
8 #define MAXLINE         4096    /* max text line length */
9
10 int main(int argc, char **argv)
11 {
12          int                     listenfd, connfd;
13          socklen_t               clilen;
14          struct sockaddr_in      servaddr, cliaddr;
15          char                    buff[MAXLINE];
16          time_t                  ticks;
17
18          listenfd = socket(AF_INET, SOCK_STREAM, 0);
19          bzero(&servaddr, sizeof(servaddr));
```

cc –o daytimetcpsrvport daytimetcpsrvport.c

Example: Server daytimetcpsrvport.c

```
20          servaddr.sin_family      = AF_INET;
21          servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
22          servaddr.sin_port        = htons(6060); /* daytime server
    */
23          bind(listenfd, (struct sockaddr *) &servaddr,
    sizeof(servaddr));
24          listen(listenfd, 1024);
25
26          for ( ; ; ) {
27                  clilen = sizeof(cliaddr);
28                  connfd = accept(listenfd, (struct sockaddr *)
    &cliaddr, &clilen);
29                  printf("Connection from client %s,at port %d\n“,
    inet_ntoa(cliaddr.sin_addr),ntohs(cliaddr.sin_port)); /* Different
    client port number will be shown at each connection */
30
31                  ticks = time(NULL);
32                  sprintf(buff, "%.24s\r\n", ctime(&ticks));
33                  write(connfd, buff, strlen(buff));
34
35                  close(connfd);
36          }
37  }
```

cc –o daytimetcpsrvport daytimetcpsrvport.c

Example: Server interactivetcpsrv.c

```
1 /* A simple server in the internet domain using TCP
2  The port number is passed as an argument */
3 #include <stdio.h>
4 #include <sys/types.h>
5 #include <sys/socket.h>
6 #include <netinet/in.h>
7
8 void error(char *msg) {
9       perror(msg);
10      exit(1);
11 }
12 int main(int argc, char *argv[]) {
13      int sockfd, newsockfd, portno, clilen;
14      char buffer[256];
15      struct sockaddr_in serv_addr, cli_addr;
16      int n;
17      if (argc < 2) {
18              fprintf(stderr,"ERROR, no port provided\n");
19              exit(1);
20       }
21
```

cc –o interactivetcpsrv interactivetcpsrv.c

Example: Server interactivetcpsrv.c

```
22      sockfd = socket(AF_INET, SOCK_STREAM, 0);
23      if (sockfd < 0)
24              error("ERROR opening socket");
25
26      bzero((char *) &serv_addr, sizeof(serv_addr));
27      portno = atoi(argv[1]);
28      serv_addr.sin_family = AF_INET;
29      serv_addr.sin_addr.s_addr = INADDR_ANY;
30      serv_addr.sin_port = htons(portno);
31      if (bind(sockfd, (struct sockaddr *) &serv_addr,
   sizeof(serv_addr)) < 0)
              error("ERROR on binding");
32
33      listen(sockfd,5);    /*maximum length the queue of pending
   connections may grow to*/
34      clilen = sizeof(cli_addr);
35      newsockfd = accept(sockfd, (struct sockaddr *) &cli_addr,
   &clilen);
   /*   you get client address by accepting the connection the
   newsocketfd is obtained and previous one can listen to other
   connections*/
36
```

Example: Server interactivetcpsrv.c (Cont'd)

```
37      if (newsockfd < 0)
38              error("ERROR on accept");
39
40      bzero(buffer,256);
41      n = read(newsockfd,buffer,255);
42      if (n < 0)
43              error("ERROR reading from socket");
44      printf("Here is the message from %s: %s\n",
   inet_ntoa(cli_addr.sin_addr), buffer);
45      n = write(newsockfd,"I got your message",18);
46      if (n < 0)
47              error("ERROR writing to socket");
48
49      return 0;
50 }
```

Example: Client interactivetcpcli.c

```
1 #include <stdio.h>
2 #include <sys/types.h>
3 #include <sys/socket.h>
4 #include <netinet/in.h>
5 #include <netdb.h>
6
7 void error(char *msg) {
8       perror(msg);
9       exit(0);
10 }
11
12 int main(int argc, char *argv[]) {
13      int sockfd, portno, n;
14      struct sockaddr_in serv_addr;
15      struct hostent *server;
16      char buffer[256];
17
18      if (argc < 3) {
19              fprintf(stderr,"usage %s hostname port\n", argv[0]);
20              exit(0);
21      }
```

cc –o interactivetcpsrv interactivetcpsrv.c

Example: Client interactivetcpcli.c (Cont'd)

```
22      portno = atoi(argv[2]);
23      sockfd = socket(AF_INET, SOCK_STREAM, 0);
24      if (sockfd < 0)
25              error("ERROR opening socket");
26
27      server = gethostbyname(argv[1]);
28      if (server == NULL) {
29              fprintf(stderr,"ERROR, no such host\n");
30              exit(0);
31      }
32
33      bzero((char *) &serv_addr, sizeof(serv_addr));
34      serv_addr.sin_family = AF_INET;
35      bcopy((char *)server->h_addr, (char
   *)&serv_addr.sin_addr.s_addr, server->h_length);
36      serv_addr.sin_port = htons(portno);
37      if (connect(sockfd,(struct sockaddr
   *)&serv_addr,sizeof(serv_addr)) < 0)
38              error("ERROR connecting");
39
```

Example: Client interactivetcpcli.c (Cont'd)

```
40      printf("Please enter the message: ");
41      bzero(buffer,256);
42      fgets(buffer,255,stdin);
43      n = write(sockfd,buffer,strlen(buffer));
44      if (n < 0)
45              error("ERROR writing to socket");
46      bzero(buffer,256);
47      n = read(sockfd,buffer,255);
48      if (n < 0)
49              error("ERROR reading from socket");
50      printf("%s\n",buffer);
51      return 0;
52 }
```

How to run the codes?

- Compile **interactivetcpsrv.c** and **interactivetcpcli.c** separately into two executables called server and client.
- They probably won't require any special compiling flags, but on some Solaris systems you may need to link to the socket library by appending **-lsocket** to your compile command.
- Ideally, you should run the client and the server on separate hosts on the Internet. However, you can simulate this on a single machine by running the server in one window and the client in another.
- Start the server first.
- When you run the server, you need to pass the port number in as an argument. You can choose any number between 2000 and 65535. If this port is already in use on that machine, the server will tell you this and exit. If this happens, just choose another port and try again.

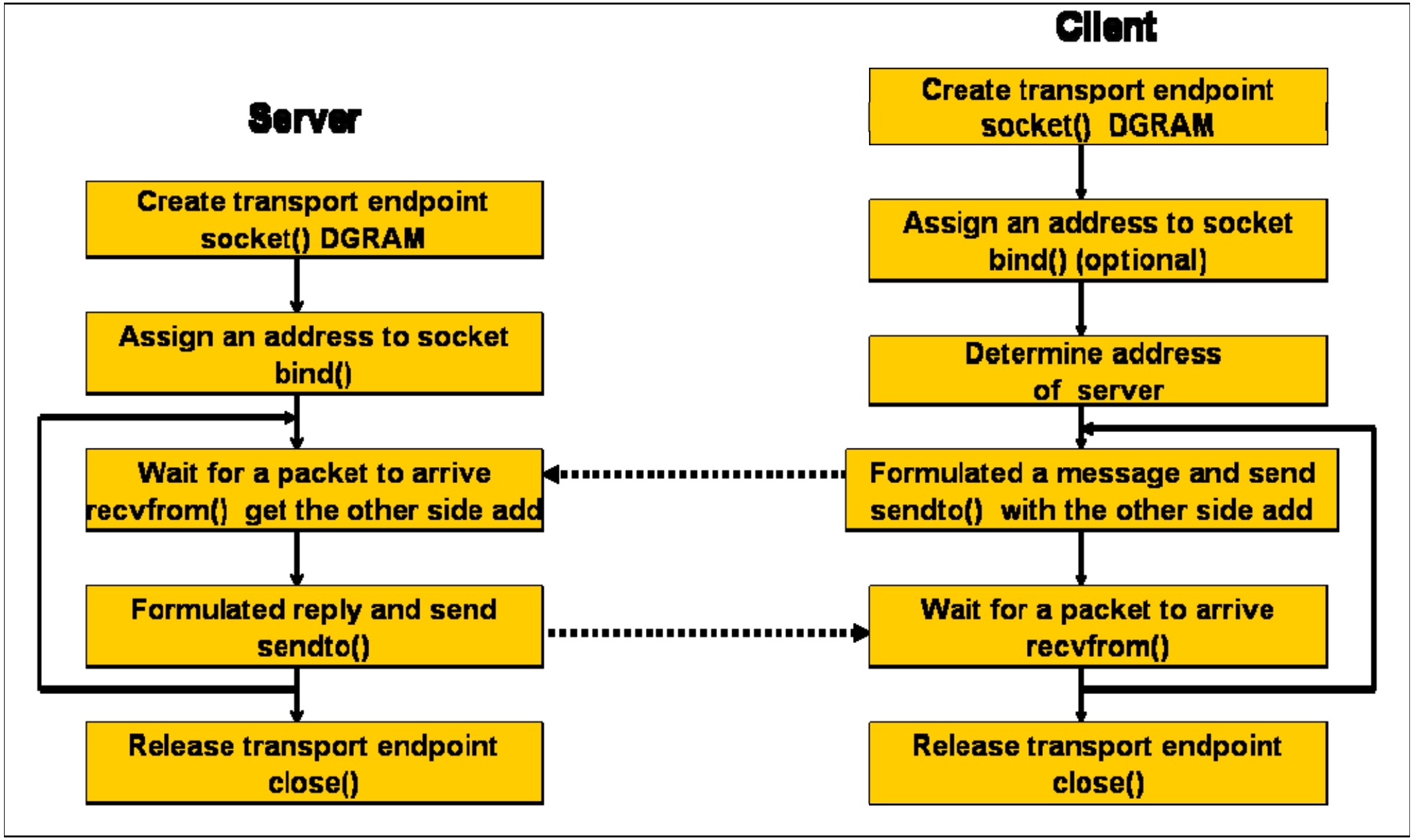
How to run the codes? (Cont'd)

- If the port is available, the server will block until it receives a connection from the client.
  - Here is a typical command line:

    interactivetcpsrv 51717
- You need to pass the name of the host on which the server is running and the port number on which the server is listening as arguments when running the client:
  - Here is the command line to connect to the server (e.g. ICIS-Linux):

    interactivetcpcli 155.69.221.20 51717
  - If you running server ad client in the same machine, you can use the keyword **localhost** as the first argument to the client.
- The client will prompt you to enter a message. If everything works correctly, the server will display your message on **stdout**, send an acknowledgement message to the client and terminate. The client will print the acknowledgement message from the server and then terminate.

Connectionless Sockets



Basic Socket Calls (Cont'd)

- **read()** (byte level based)
  - Receives a message from a socket:
  - **int read (int s, char *buf, int len);**
- **recv()** (stream based)
  - Receives a message from a socket:
  - **int recv (int s, char *buf, int len, int flags);**
- **sendto()**
  - Sends a datagram message to an UDP socket. (in case of TCP, sockaddr is omitted)
  - **int sendto (int s, char *msg, int len, int flags, struct sockaddr *addr, int addrlen);**
- **recvfrom()**
  - Receives a datagram message from an UDP socket.
  - **int recvfrom (int s, char *buf, int len, int flags, struct sockaddr *addr, int *addrlenptr);**

Example: Server interactiveudpsrv.c

```
1 /* Creates a datagram server. The port  number is passed as an
   argument.  This server runs forever */
2 #include <sys/types.h>
3 #include <sys/socket.h>
4 #include <netinet/in.h>
5 #include <netdb.h>
6 #include <stdio.h>
7
8 void error(char *msg) {
9       perror(msg);
10      exit(0);
11 }
12 int main(int argc, char *argv[]) {
13      int sock, length, fromlen, n;
14      struct sockaddr_in server;
15      struct sockaddr_in from;
16      char buf[1024];
17      if (argc < 2) {
18              fprintf(stderr, "ERROR, no port provided\n");
19              exit(0);
20      }
```

Example: Server interactiveudpsrv.c (Cont'd)

```
21      sock=socket(AF_INET, SOCK_DGRAM, 0);
22      if (sock < 0)
23              error("Opening socket");
24
25      length = sizeof(server);
26      bzero(&server,length);
27      server.sin_family=AF_INET;
28
29      /* Automatically get the server ip */
30      server.sin_addr.s_addr=INADDR_ANY;
31      server.sin_port=htons(atoi(argv[1]));
32      /*  the socketaddr_in  has the three member: protocol
   family,
        host address and port number */
33
34      if (bind(sock,(struct sockaddr *)&server,length)<0)
35              error("binding");   /* convert the data type to
   sockaddr */
36
37       fromlen = sizeof(struct sockaddr_in);
38
```

Example: Server interactiveudpsrv.c (Cont'd)

```
39      while (1) {
40              n = recvfrom(sock,buf,1024,0,(struct sockaddr *)
   &from,&fromlen);
41              if (n < 0) error("recvfrom");
42              write(1,"Received a datagram: ",21);     /* 1, the
   output */
43              write(1,buf,n);
44
45              /* use the other side address*/
46              n = sendto(sock,"Got your message\n",17, 0,(struct
   sockaddr *) &from,fromlen);
47
48              if (n  < 0)
49                      error("sendto");
50      }
51 }
```

Example: Client interactiveudpcli.c

```
1 /* UDP client in the internet domain */
2 #include <sys/types.h>
3 #include <sys/socket.h>
4 #include <netinet/in.h>
5 #include <arpa/inet.h>
6 #include <netdb.h>
7 #include <stdio.h>
8
9 void error(char *);
10
11 int main(int argc, char *argv[]) {
12      int sock, length, n;
13      struct sockaddr_in server, from;
14      struct hostent *hp;
15      char buffer[256];
16      if (argc != 3) {
17              printf("Usage: server port\n");
18              exit(1);
19      }
20
```

Example: Client interactiveudpcli.c (Cont'd)

```
21      sock= socket(AF_INET, SOCK_DGRAM, 0);  /* UDP, DGRAM*/
22      if (sock < 0)
23              error("socket");
24
25      server.sin_family = AF_INET;
26      hp = gethostbyname(argv[1]);
27      if (hp==0)
28              error("Unknown host");
29
30      bcopy((char *)hp->h_addr, (char *)&server.sin_addr, hp-
   >h_length);
31      server.sin_port = htons(atoi(argv[2]));
32      length=sizeof(struct sockaddr_in);
33      printf("Please enter the message: ");
34      bzero(buffer,256);
35      fgets(buffer,255,stdin);
36      n=sendto(sock,buffer,strlen(buffer),0,(struct sockaddr*)
   &server,length);
37      if (n < 0)
38              error("Sendto");
39      n = recvfrom(sock,buffer,256,0,(struct sockaddr*) &from,
   &length);
```

Example: Client interactiveudpcli.c (Cont'd)

```
40      if (n < 0)
41              error("recvfrom");
42      write(1,"Got an ack: ",12);
43      write(1,buffer,n);
44 }
45
46 void error(char *msg) {
47      perror(msg);
48      exit(0);
49 }
```

Unix Processes and Threads

Unix Process

- While a program is running, it is known as a process.
- A Unix process is an instance of an executing program: it has a separate existence from all the other processes on the system.
  - **Program**
    - ✓A program is a sequence of instructions.
  - **Process**
    - ✓A process is a dynamic invocation of a program along with the resources required to run.

Unix Process (Cont'd)

- When a UNIX process is created, it is assigned some standard file descriptors (file pointers):
  - **STDIN**: standard input (Typically pointing to the keyboard)
  - **STDOUT**: standard output (Typically pointing to the terminal)
  - **STDERR**: Standard error (Typically pointing to the terminal)

- It is assumed that a standard UNIX process reads its input from the STDIN, writes its output to the STDOUT and writes its error messages to the STDERR.

Process Environment

- When a program is executed on a UNIX system, the system creates a special environment for that program.

- This environment contains everything needed for the system to run the program as if no other programs were running on the system.

- A Unix Process consists of:
  - an **address space,**
  - a context (machine state and stack), (other than threads)
  - ancillary information (credentials, open files, etc.).

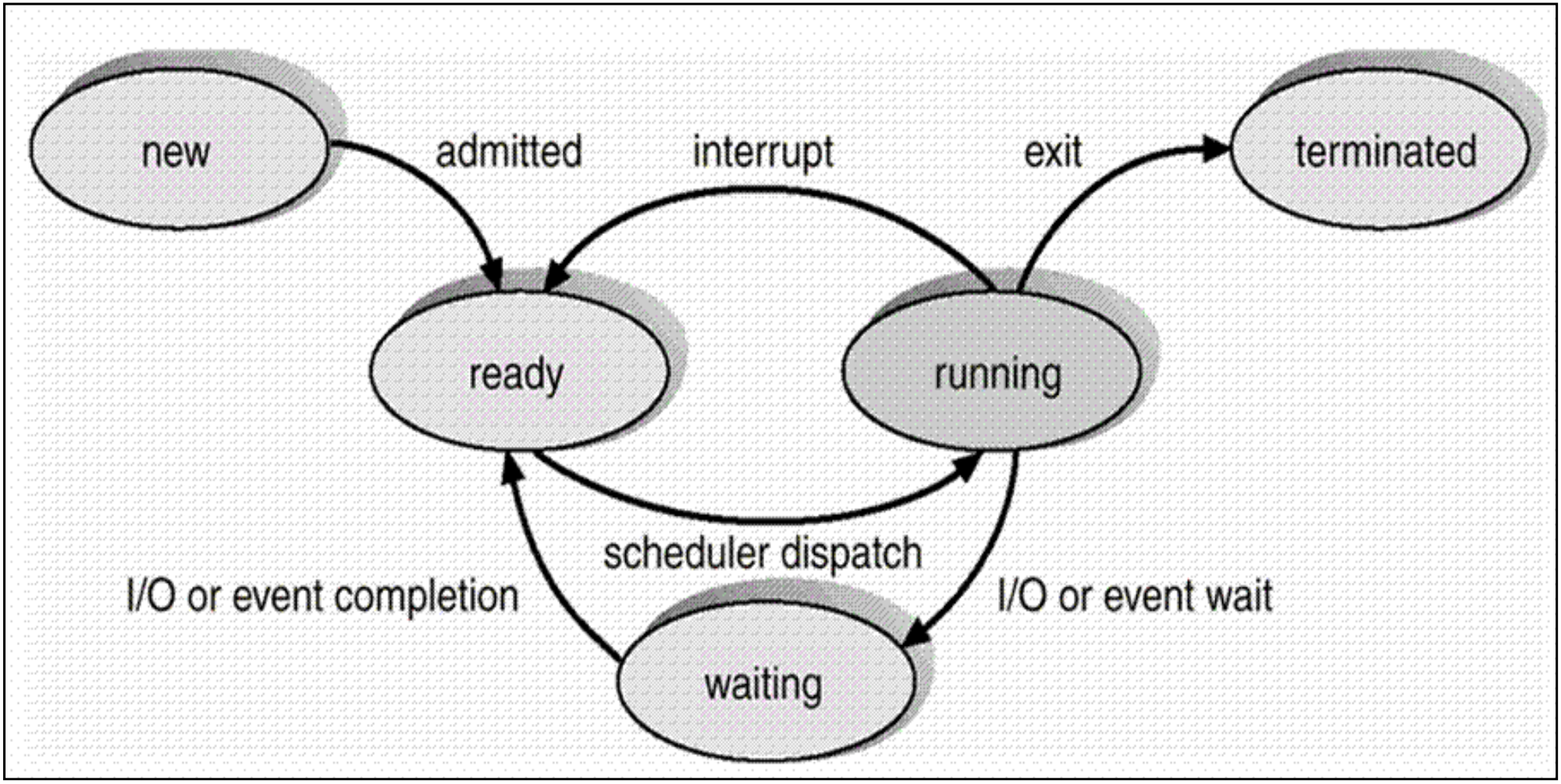
Process Types and Attributes

- **Process Types:** A process may be:
  - **Interactive:** actively using a terminal,
  - **Batch:** running separate from any terminal, and possibly at a different time,
  - **Daemon:** a system process, running all the time.
- **Process attributes:** Every process has several attributes. Among them, it has:
  - a unique number known as its **process-id** or **pid**,
  - the **pid** of its parent process: the process which started it, usually a shell,
  - the **user-id** and **group-id** of the user who started the process
  - the terminal (if any), to which the process is attached.

Process States

- A process could have different states including:
  - **User running:** Process executes in user mode,
  - **Kernel running:** Process executes in kernel mode,
  - **Ready to run**: process is waiting to be scheduled,
  - **Asleep**: waiting for an event,
  - **Swapped**: ready to run but requires swapping in,
  - **Preempted**: Process is returning from kernel to user-mode but the system has scheduled another process instead,
  - **Created**: Process is newly created and not ready to run,
  - **Zombie**: Process no longer exists, but it leaves a record for its parent process to collect.

Process Life Cycle



Daemon Process

- A daemon is a process that detaches itself from the terminal and runs, disconnected, in the background, waiting for requests and responding to them.
- Many system functions are commonly performed by daemons, including the **sendmail** daemon, which handles mail.
- They run in background and they don’t have a controlling terminal.
- Unix systems have numerous daemons that perform day-to-day activities.

Process Creation

- Possible mechanisms for creating new processes using an existing one:
  - **Synchronous**:
    - ✓The new process must complete execution before the old one can resume.
  - **Asynchronous**:
    - ✓The two process may be run in pseudo-parallel.
  - **Parent-Child relationship:**
    - ✓When a new process is created in Unix, it uses the old one as "**parent**".

Process Creation (Cont'd)

- **fork()** is the only way to create a new process by UNIX kernel.
- Child gets identical copy of parent's process space.
- Resources opened by parent are shared by the child. The child also inherits the resource limits.
- Usually used to:
  - execute another program using the exec() system primitives in the current context.
  - spawn a child to handle a new client's request.

fork()

- The function calls to **fork()** create child processes.

```
#include <sys/types.h>
#include <unistd.h>
pid_t  fork(void)
```

  - Returns: 0 in child, process ID of child in parent, -1 on error.
- This splits the current process into two almost identical copies.
  - A copy is made of the user and system stacks, of the allocated data space and of the registers.
  - The major difference is that one has the PID of the parent, the other has a new PID.
  - The fork returns a value that is a PID.
  - The PID is zero if you are the child, or the PID of the child if you are the parent.

Example forkDemo.c

```
1 #include <stdio.h>
2 #include <unistd.h>
3 #include <sys/wait.h>
4 #include <sys/types.h>
5
6 int main(int argc, char *argv[])  {
7         int pid, stat, option;
8         pid_t PID;
9
10        printf("Forking for a new Child Process.\n");
11        pid = fork(); /* fork another process */
12        if (pid < 0) {  /* Error occured */
13                  fprintf(stderr, "Fork has faid\n");
14        exit (-1);
15        }
16        else if (pid == 0) {  /* Child process */
17                  printf("Inside the Child process.\n");
18                  execlp("ls","ls","-la",NULL);
19        }
20        else      {         /* Parent Process */
21                  printf("The parent Process is waiting for Child to complete.\n");
          /* parent waits for children */
22                  while ((PID = waitpid(-1,&stat,option)) > 0)
23                            printf("Child %d is terminated\n",PID);
24                  printf("Back to the Parent Process.\n");
25                  exit(0);
26        }
27 }
```

Compile: cc forkDemo.c –o forkDemo

wait(), wiatpid()

```
#include <sys/types.h>
#include <sys/wait.h>

pid_t wait(int *status)
pid_t waitpid(pid_t pid,int *status, int option)
```

- The **wait()** function suspends execution of the current process until a child has exited, or until a signal is delivered whose action is to terminate the current process or to call a signal handling function.
- If a child has already exited by the time of the call (a so-called "zombie" process), the function returns immediately.
- The **waitpid()** function suspends execution of the current process until a child as specified by the pid argument has exited.

wait(), wiatpid() (Cont'd)

- The value of pid can be one of:
  - **<-1:** to wait for any child process whose process **group ID** is equal to the absolute value of pid.
  - **-1 :** to wait for **any child** process; this is the same behavior which wait exhibits.
  - **0 :** to wait for **any child** process whose process **group ID** is equal to that of the calling process.
  - **>0 :** to wait for the child whose **process ID** is equal to the value of pid.

wait(), wiatpid() (Cont'd)

- The value of status can be checked for "reason" child terminated.
  - WIFSTOPPED is true if child was stopped (usu. by ctrl-Z).
  - WIFSIGNALED (did child fail to catch a signal?),
  - WIFTERMINATED (did child terminate normally?).

- The value of options can modify the behavior of **waitpid()** :
  - **WNOHANG:** if specified, waitpid() will not block if no child terminated.
  - **WUNTRACED**: if specified, previously stopped processes will be reported.

Example processDemo.c

```
1 #include <stdio.h>
2 #include <unistd.h>
3 #include <sys/wait.h>
4 #include <sys/types.h>
5
6 void Child1();
7 void Child2();
8
9 int main(){
10        pid_t child1, child2;
11        pid_t PID;
12        int  stat, option;
13
14        printf("I am now in the Parent process.\n\n");
15        /* start forking children */
16        if ((child1 = fork()) == 0) {           /* first child */
17                  Child1();
18                  exit(0);
19        }
20        if ((child2 = fork()) == 0)  {    /* second child */
21                  Child2();
22                  exit(0);
23        }
24        /* parent waits for children */
25        while ((PID = wait(&stat)) > 0)
26                  printf("Child %d is terminated\n",PID);
27
28        printf("\nNow I am back to the Parent Process.\n");
29        return 0;
30 }
```

Example processDemo.c (Cont'd)

```
31
32 void Child1() { /* Waste time */
33         int i, j, x=0;
34
35         for (i = 0;  i < 4; i++) {
36                   printf("I am now in Child 1 Process\n");
37                   for (j = 0; j < 10000; j++)
38                             x = x + i;
39         }
40 }
41
42 void Child2() { /* Waste time. The code is almost the same as Child1() */
43         int i, j, x=0;
44
45         for (i = 0;  i < 4; i++) {
46                   printf("I am now in Child 2 Process\n");
47                   for (j = 0; j < 10000; j++)
48                             x = x + i;
49         }
50 }
```

Compile: cc processDemo.c –o processDemo

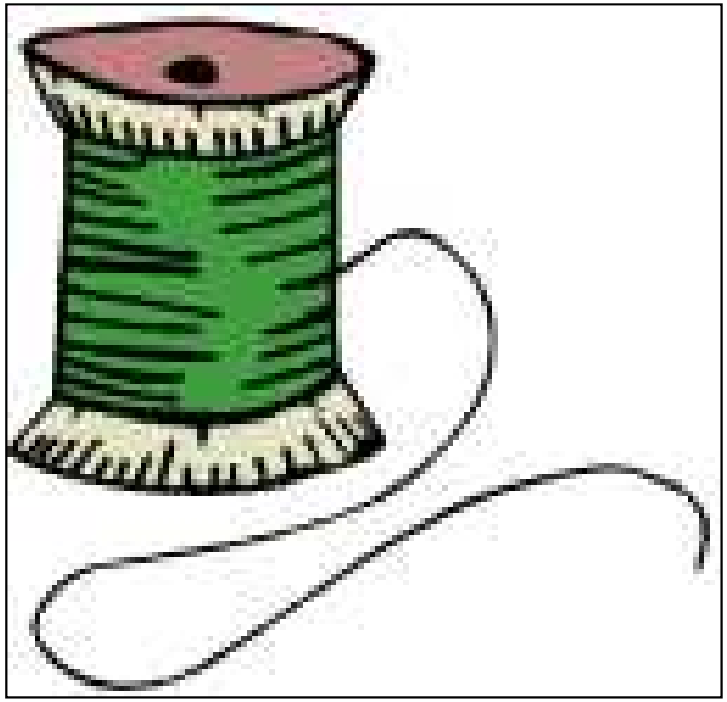
Process Scheduling

- A CPU is a resource that must be shared by all processes.
- The part of kernel that apportions CPU time between processes is called the ***scheduler***.

- A scheduler must compromise between certain desirable things:
  - Make sure each process gets a fair share of the CPU,
  - Keep the CPU busy 100% of the time,
  - Minimize response delays,
  - Be consistent so that behaviour is expected.

Concurrent Server

```
1 pid_t         pid;
2 int           listenfd, connfd;
3
4 listenfd = socket (...);
5
6 bind(listenfd, ...);
7 listen(listenfd, ...);
8
9 for (; ;){
10      connfd=accept(listenfd, ...)
11
12       if ((pid = fork()) ==0){
13              close(listenfd);        // child closes listening
   socket
14              doit(connfd);           // process the request
15              close(connfd);          // done with this client
16              exit(0);                // child terminates
17      }
18
19      close(connfd);                  // parent closes connected
   socket
20 }
```



Threads In UNIX

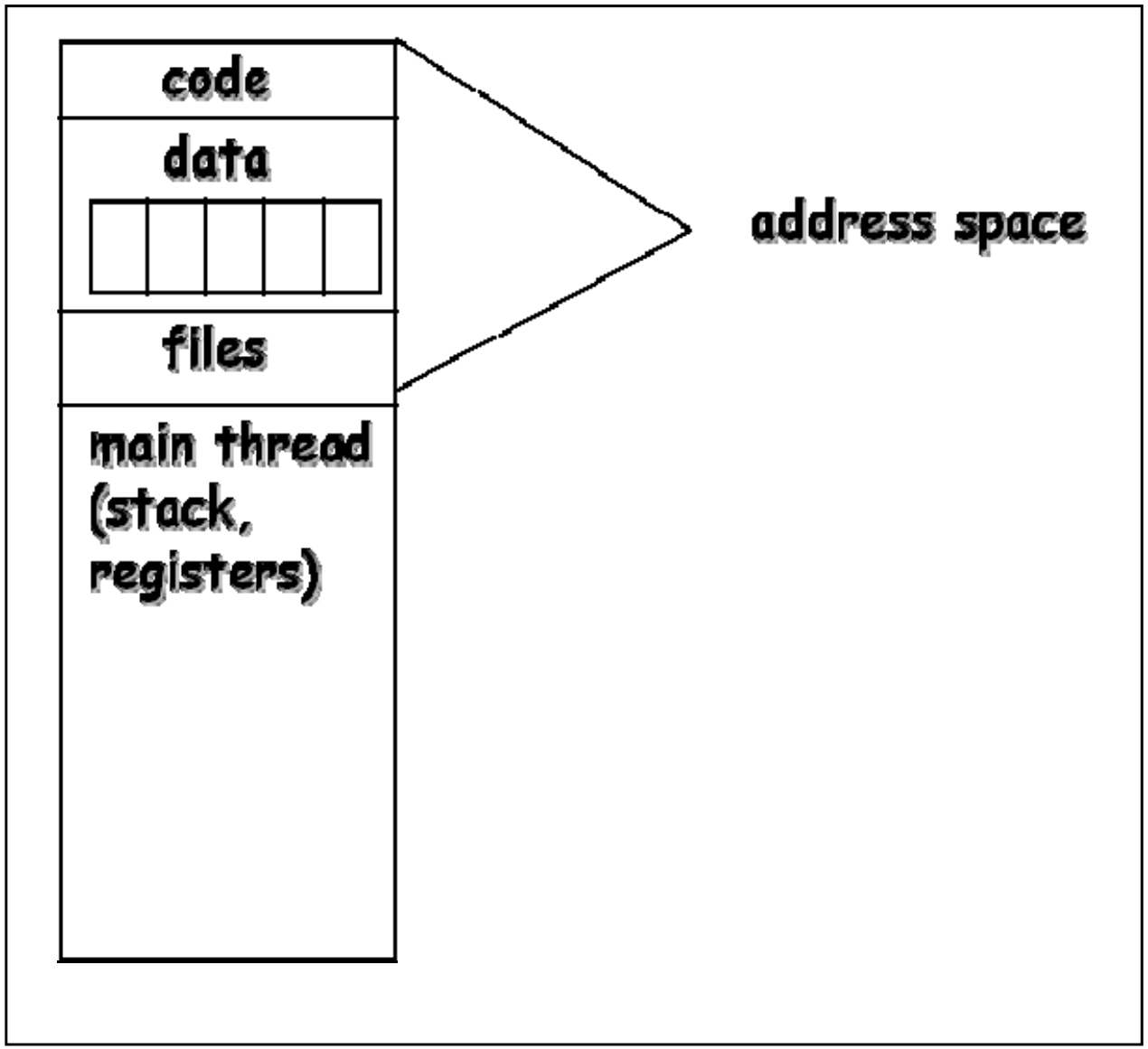
Threads

- Problem with traditional process forking:
  - fork() is expensive in terms of system resources.
- Solution: Threads (**lightweight process**)
- Share same global memory within the process.
  - problem: synchronization

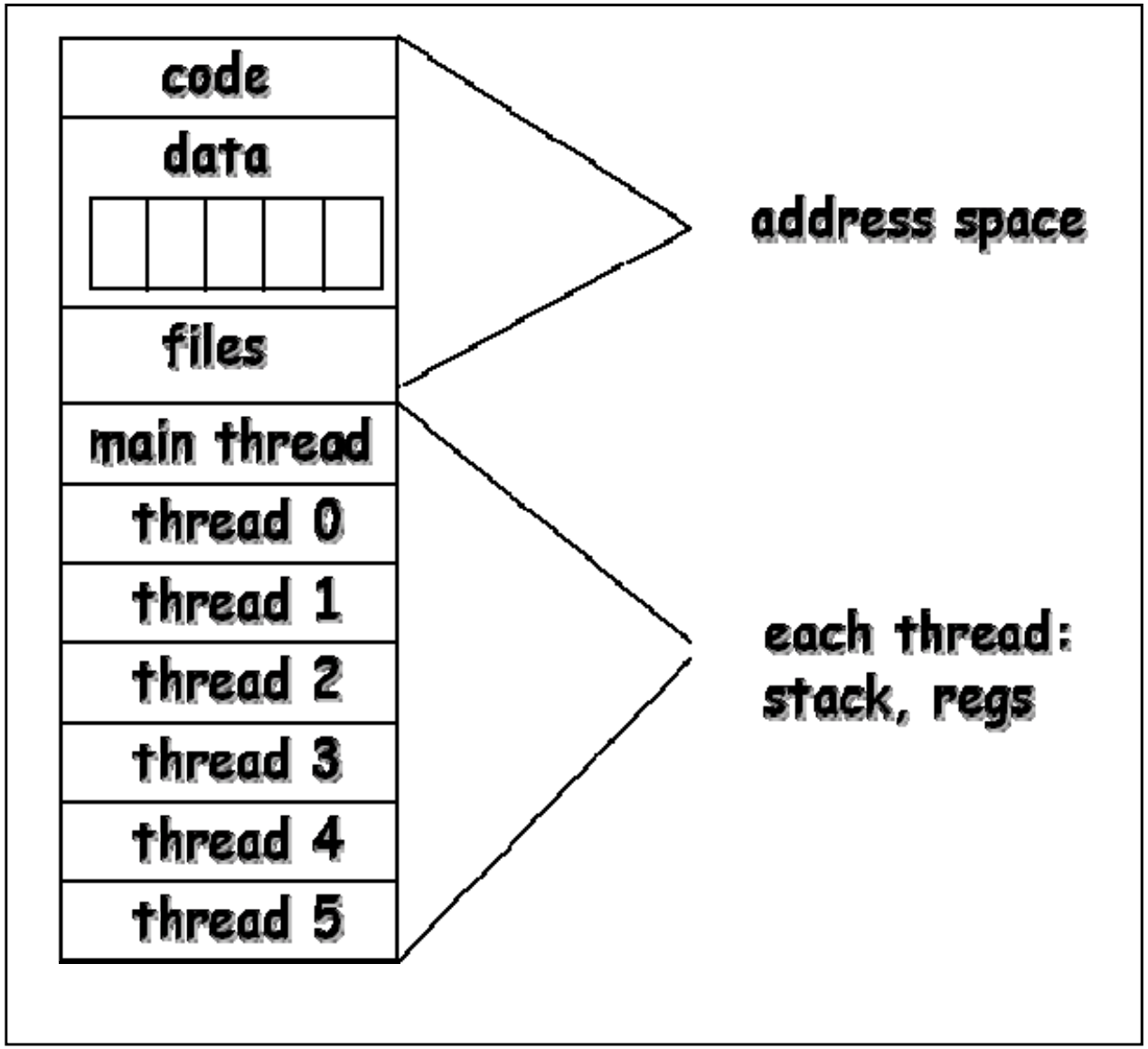
Threads (Cont'd)

- All threads within the same process share:
  - Process instructions
  - Most part of the data
  - Open files
  - Signal handlers
  - User and group ID
- Each thread has
  - Thread ID
  - Set of registers, including program counter and stack pointer
  - Stack (for local variables and program counters)
  - Priority
  - Signal mask

Single Thread Process



Multiple Threads



Thread Functions (1)

- Creating a Thread:

```
#include <pthread.h>
int pthread_create(pthread_t *thread,
                   const pthread_attr_t attr,
                   void *(*func)(void *),
                   void *arg);
```

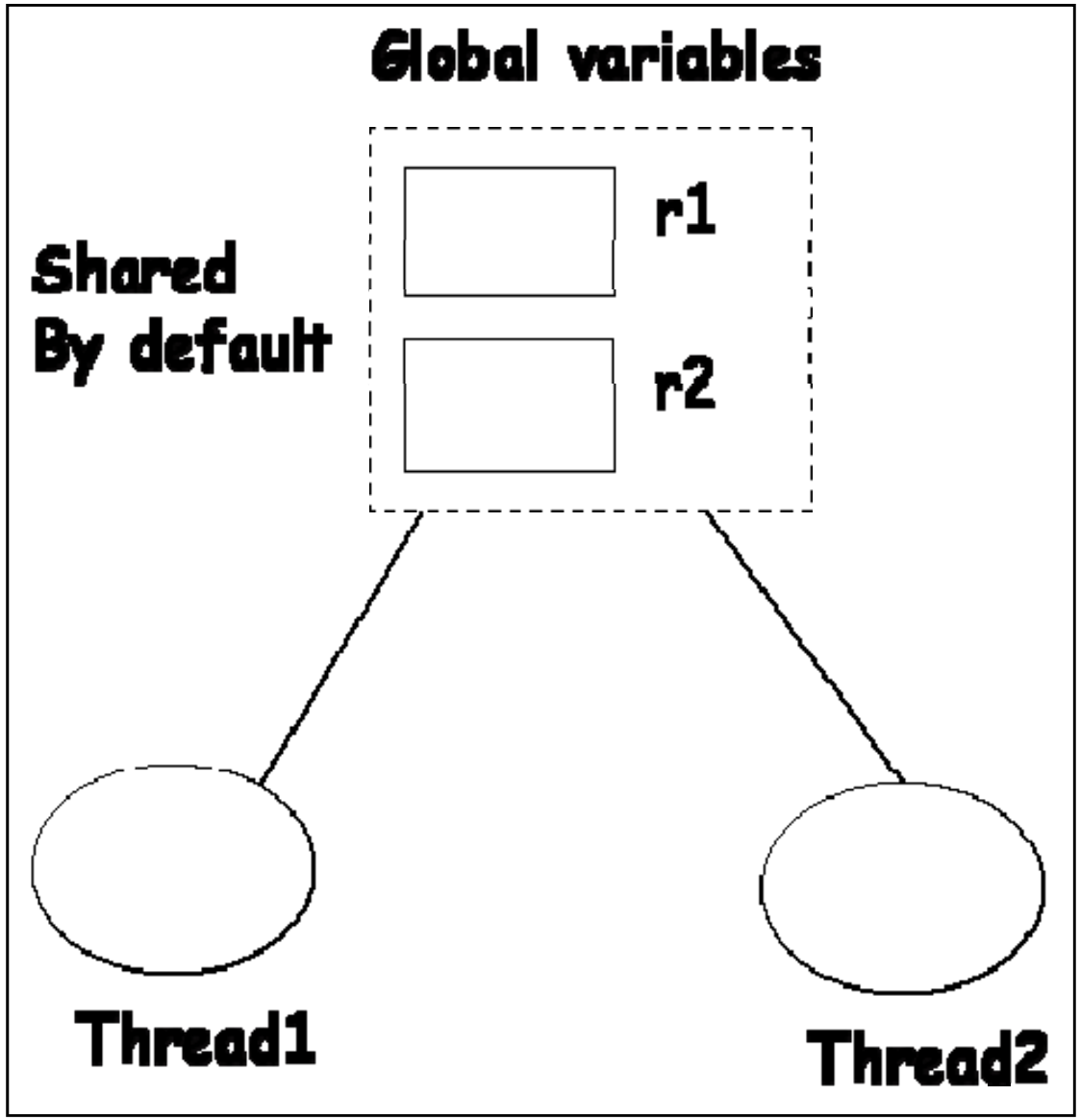
  - The pthread_create() function creates a thread with the specified attributes and starts executing func() and pass it arg.
  - If it completes successfully, the pthread handle is stored in the contents of the location referred to by **thread**.
  - The thread attributes object containing the attributes to be associated with the newly created thread. If NULL the default thread attributes are used.

Thread Functions (2)

```
int pthread_join(pthread_t thread,
                 void **value_ptr);
```

- Makes the calling thread wait for the specified thread to terminate.
- It is like wait() function for processes.
- value_ptr is assigned its return value (or PTHREAD_CANCELLED).

Example threadDemo.c



Example threadDemo.c (Cont'd)

```
1 #include <stdio.h>
2 #include </usr/include/pthread.h>
3
4 void Thread1(int *pnum);
5 void Thread2(int *pnum);
6
7 int r1 = 0, r2 = 0; /* Global variables */
8
9 int main() {
10        int temp;
11        pthread_t thread1, thread2;
12
13        printf("Thread1: Creating a new thread.\n");
14        temp=pthread_create(&thread1, NULL, (void *) Thread1, (void *) &r1);
15
16        printf("Thread2: Creating a new thread.\n");
17        temp=pthread_create(&thread2, NULL, (void *) Thread2,(void *) &r2);
18
19        temp=pthread_join(thread1, NULL); /* Waiting for the two Threads to
    terminate */
20        temp=pthread_join(thread2, NULL);
21
22        printf("Final Values: %d, %d\n",r1, r2);
23        return 0;
24}
```

Example threadDemo.c (Cont'd)

```
26 void Thread1(int *pnum) {/* Thread1 function*/
27       int i, j, x;
28       x=0;
29       for (i = 0;  i < 4; i++) {
30                printf("I am in Thread1 now :-) \n");
31                sleep(2);
32                for (j = 0; j < 10000; j++)
33                         x = x + i;
34                (*pnum)++;
35       }
36 }
37
38 void Thread2(int *pnum) { /* Thread2 function */
39       int i, j, x;
40       x=0;
41       for (i = 0;  i < 4; i++) {
42                printf("I am in Thread2 now :-)\n");
43                sleep(2);
44                for (j = 0; j < 10000; j++)
45                         x = x + i;
46                (*pnum)++;
47       }
48 }
```

Compile: cc –lpthread threadDemo.c –o threadDemo

Example threadMatrix.c

- Parallel Matrix Multiplication:
  - Create MATSIZE (e.g. 4) threads: one for each column of the results[] array
  - each column will be calculated in parallel.
  - The parallelism could be increased by creating MATSIZE*MATSIZE threads: one for each element of the results[] array.

$$\begin{pmatrix} 9 & 8 & 7 & 6 \\ 6 & 5 & 4 & 3 \\ 3 & 2 & 1 & 0 \\ 0 & -1 & -2 & -3 \end{pmatrix} * \begin{pmatrix} 1 & 2 & 3 & 4 \\ 4 & 5 & 6 & 7 \\ 7 & 8 & 9 & 10 \\ 10 & 11 & 12 & 13 \end{pmatrix} = \begin{pmatrix} 150 & 180 & 210 & 240 \\ 84 & 102 & 120 & 138 \\ 18 & 24 & 30 & 36 \\ -48 & -54 & -60 & -66 \end{pmatrix}$$

Example threadMatrix.c

```
1 #include <stdio.h>
2 #include <pthread.h>
3
4 #define MATSIZE 4
5 void *matMult(void *);
6 void printMult(void);
7
8 /* global and shared data */
9 int mat1[MATSIZE][MATSIZE] = {{9,8,7,6},{6,5,4,3},{3,2,1,0},{0,-1,-2,-3}};
10 int mat2[MATSIZE][MATSIZE] = {{1,2,3,4},{4,5,6,7},{7,8,9,10},{10,11,12,13}};
11 int result[MATSIZE][MATSIZE];
12
13 int main() {
14        pthread_t thr[MATSIZE];
15        int i;
16
17        for(i=0; i < MATSIZE; i++)
18                  pthread_create(&thr[i], NULL, matMult, (void *)i);
19
20        for(i=0; i < MATSIZE; i++)
21                  pthread_join(thr[i], NULL);
22
23        printMult();
24        return 0;
25 }
```

Example threadMatrix.c (Cont'd)

```
26 void *matMult(void *colv) {
27         int i, j;
28         int col = (int)colv;
29
30         for(i=0; i < MATSIZE; i++) {
31                   result[i][col] = 0;
32                   for(j=0; j < MATSIZE; j++)
33                             result[i][col] += mat1[i][j] * mat2[j][col];
34         }
35 }
36
37 void printMult(void) {  /* Prints the result */
38         int i, j;
39
40         for(i=0; i < MATSIZE; i++) {
41                   printf("|");
42                   for(j=0; j < MATSIZE; j++)
43                             printf("%3d", mat1[i][j]);
44                   printf("|%c|", (i==MATSIZE/2 ? '*' : ' ' ));
45                   for(j=0; j < MATSIZE; j++)
46                             printf("%3d", mat2[i][j]);
47                   printf("|%c|", (i==MATSIZE/2 ? '=' : ' ' ));
48                   for(j=0; j < MATSIZE; j++)
49                             printf("%4d", result[i][j]);
50                   printf("|\n");
51         }
52 }
```