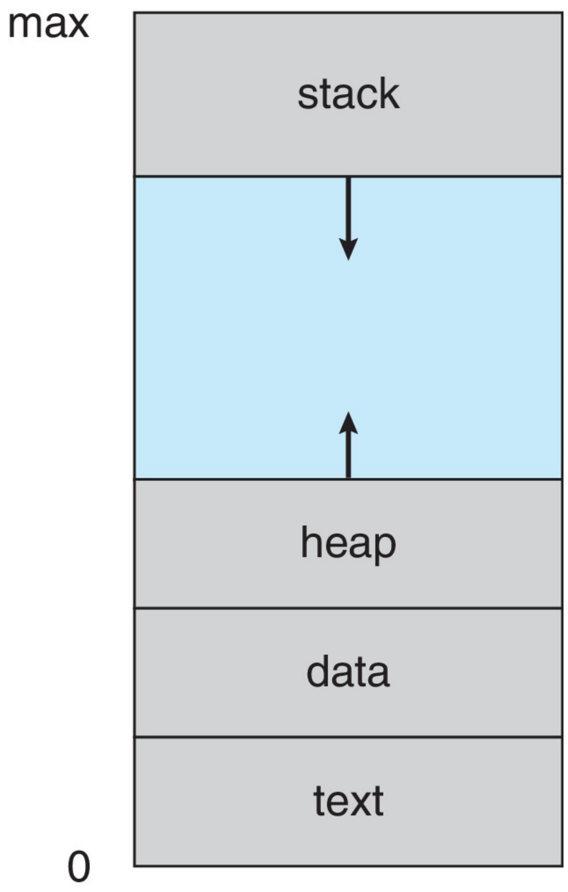
Chapter 3: Processes

Chapter 3: Processes

3.1 Process Concept



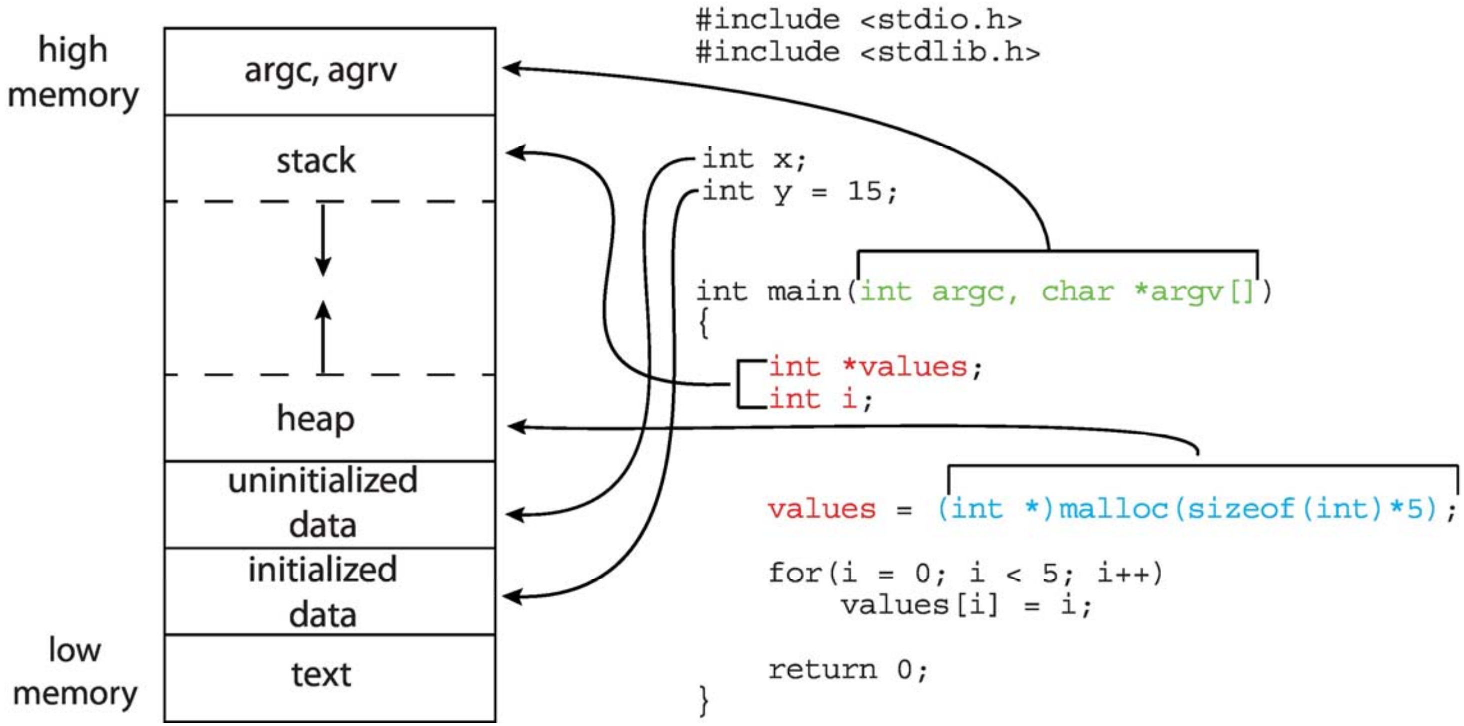
An operating system executes a variety of programs that run as a process

- Program is ***passive*** entity stored on disk (**executable file**
- **Process** is ***active*** – a program in execution; process execution must progress in sequential fashion

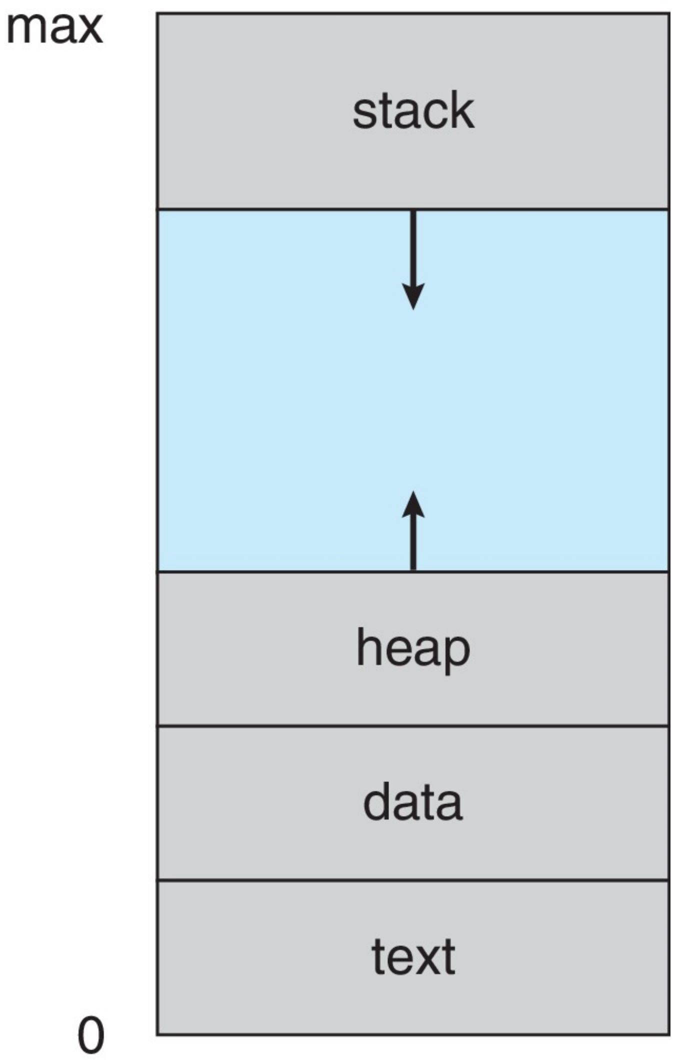
Multiple parts

- The program code, also called **text section**
- Current activity including **program counter**, processor registers
- **Stack** containing temporary data
  - Function parameters, return addresses, local variables
- **Data section** containing global variables
- **Heap** containing memory dynamically allocated during run time

Memory Layout of a C Program



Process in Memory

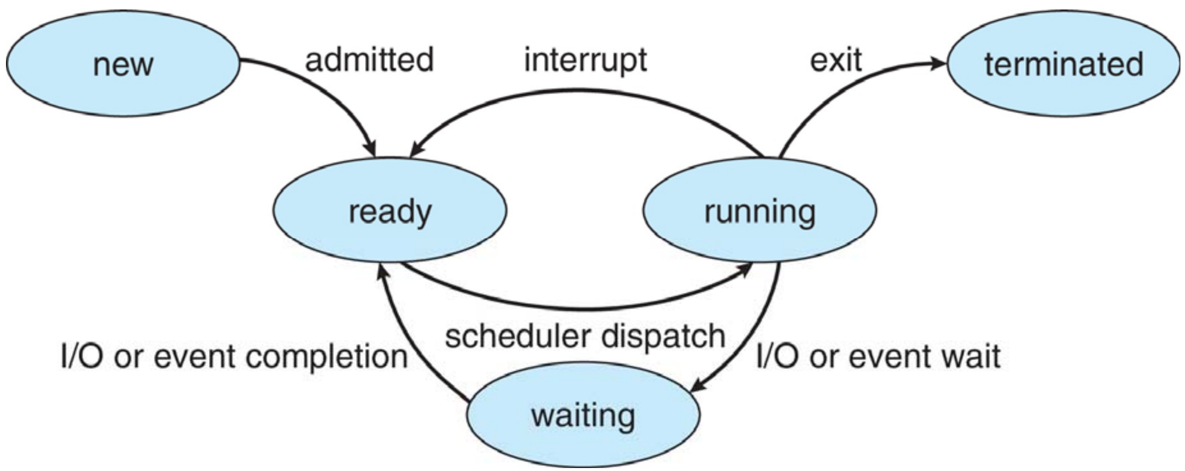



```
int          dataarea = 9999;

void output(int a, b, c)
{
        int          x, y, z;
        char         *p;
        static char buf[65536];

        x = a * b * c;
        y = a + b + c;
        z = dataarea + y;
        printf("%d, %d, %d\n", x, y, z);
        p = malloc(c);
}

int main(int argc, char *argv[])
{
        int          a, b, c;

        scanf("%d, %d, %d", &a, &b, &c);
        output(a, b, c);
        return 0;
}
```

Process State

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

- **New**: The process is being created
- **Running**: Instructions are being executed
- **Waiting**: The process is waiting for some event to occur
- **Ready**: The process is waiting to be assigned to a processor
- **Terminated**: The process has finished execution



Process Control Block (PCB)

Information associated with each process

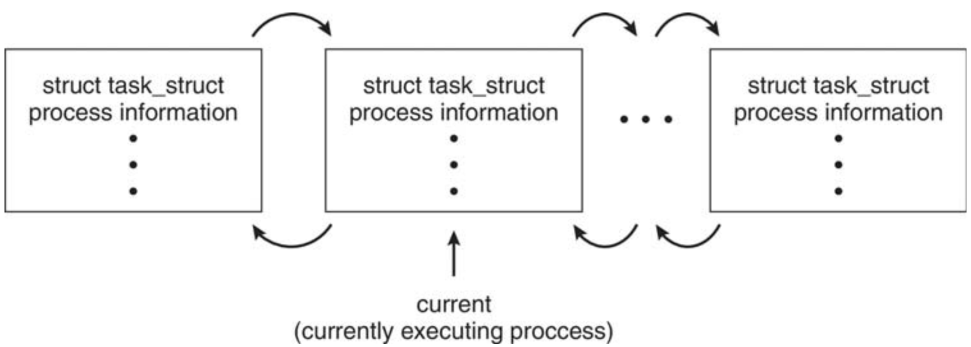
(also called **task control block**)

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

Process Representation in Linux

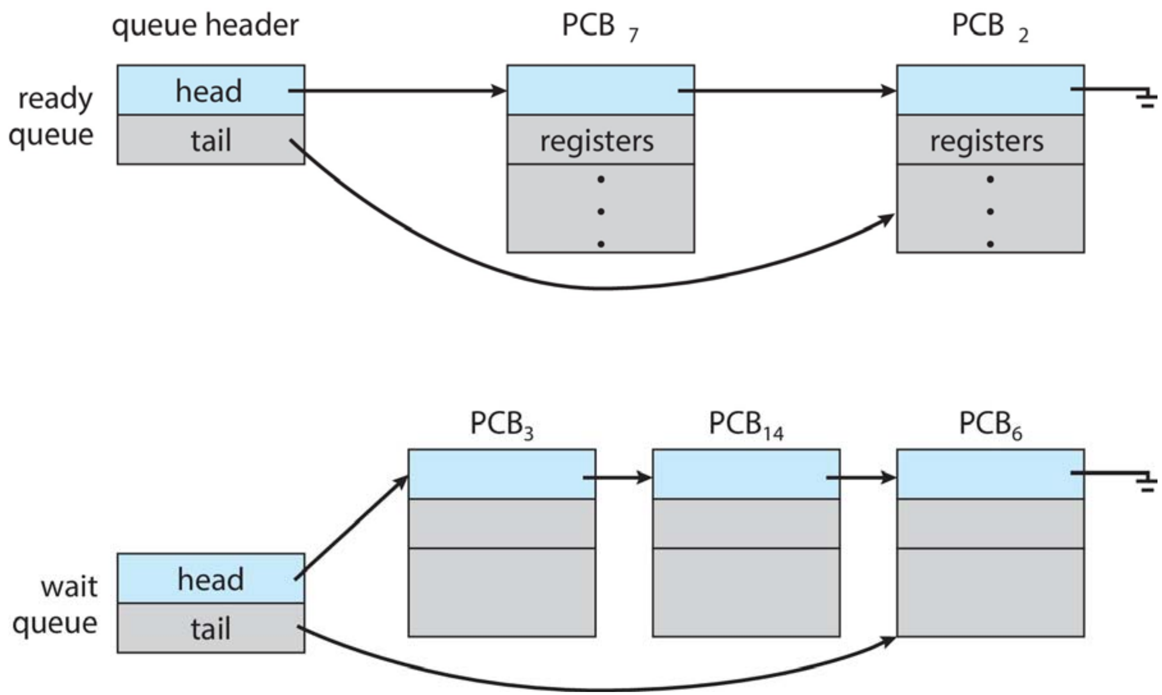
Represented by the C structure `task_struct`

```
pid t_pid;                    /* process identifier */
long state;                   /* state of the process */
unsigned int time_slice    /* scheduling information */
struct task_struct *parent;/* this process's parent */
struct list_head children; /* this process's children */
struct files_struct *files;/* list of open files */
struct mm_struct *mm;      /* address space of this process */
```



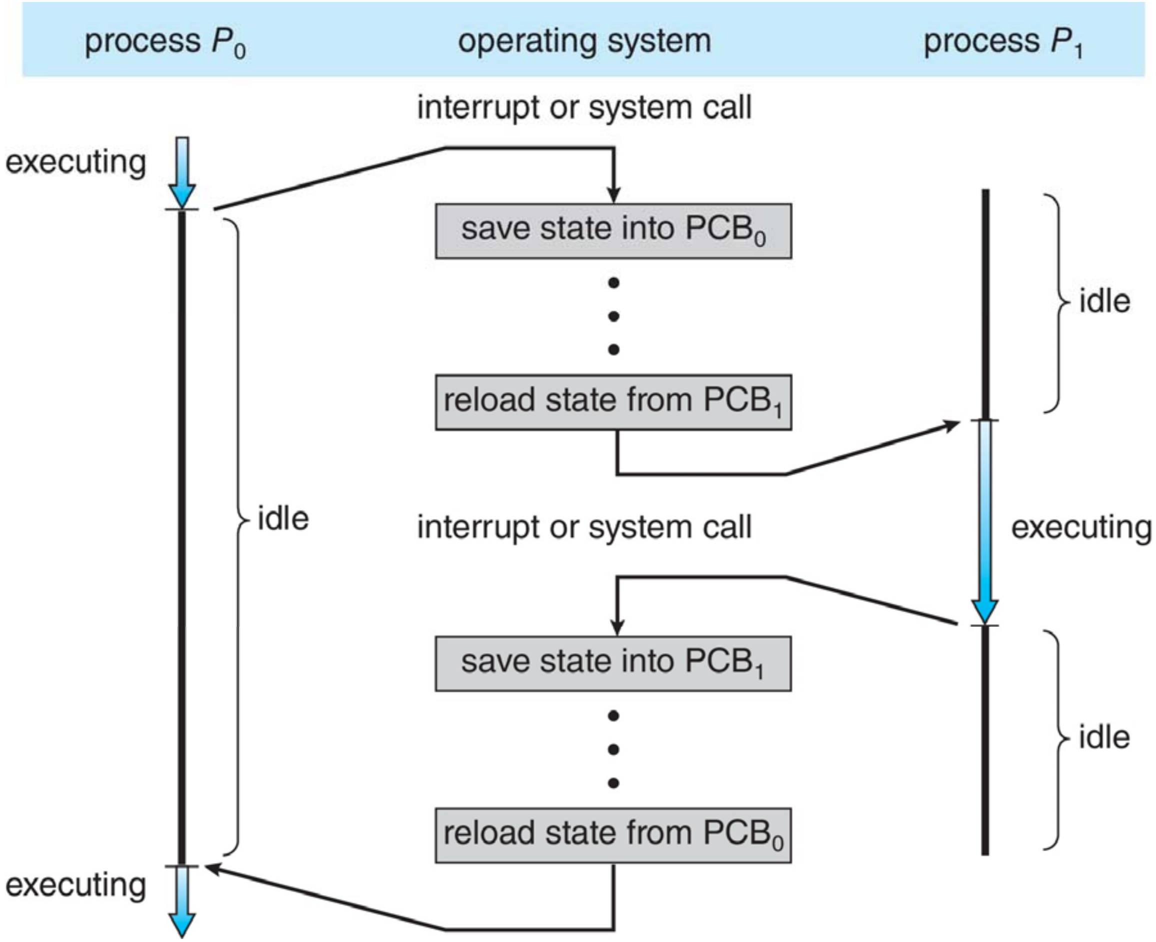
3.2 Process Scheduling

- Maximize CPU use, quickly switch processes onto CPU core
- **Process scheduler** selects among available processes for next execution on CPU core
- Maintains **scheduling queues** of processes
  - **Ready queue** – set of all processes residing in main memory, ready and waiting to execute
  - **Wait queues** – set of processes waiting for an event (i.e. I/O)
  - Processes migrate among the various queues



CPU Switch From Process to Process

context switch occurs when the CPU switches from one process to another
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



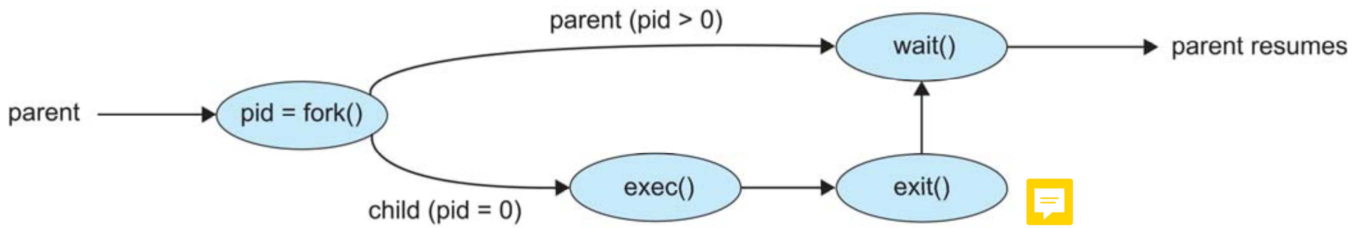
3.3 Operations on Processes

System must provide mechanisms for:

- process creation
- process termination

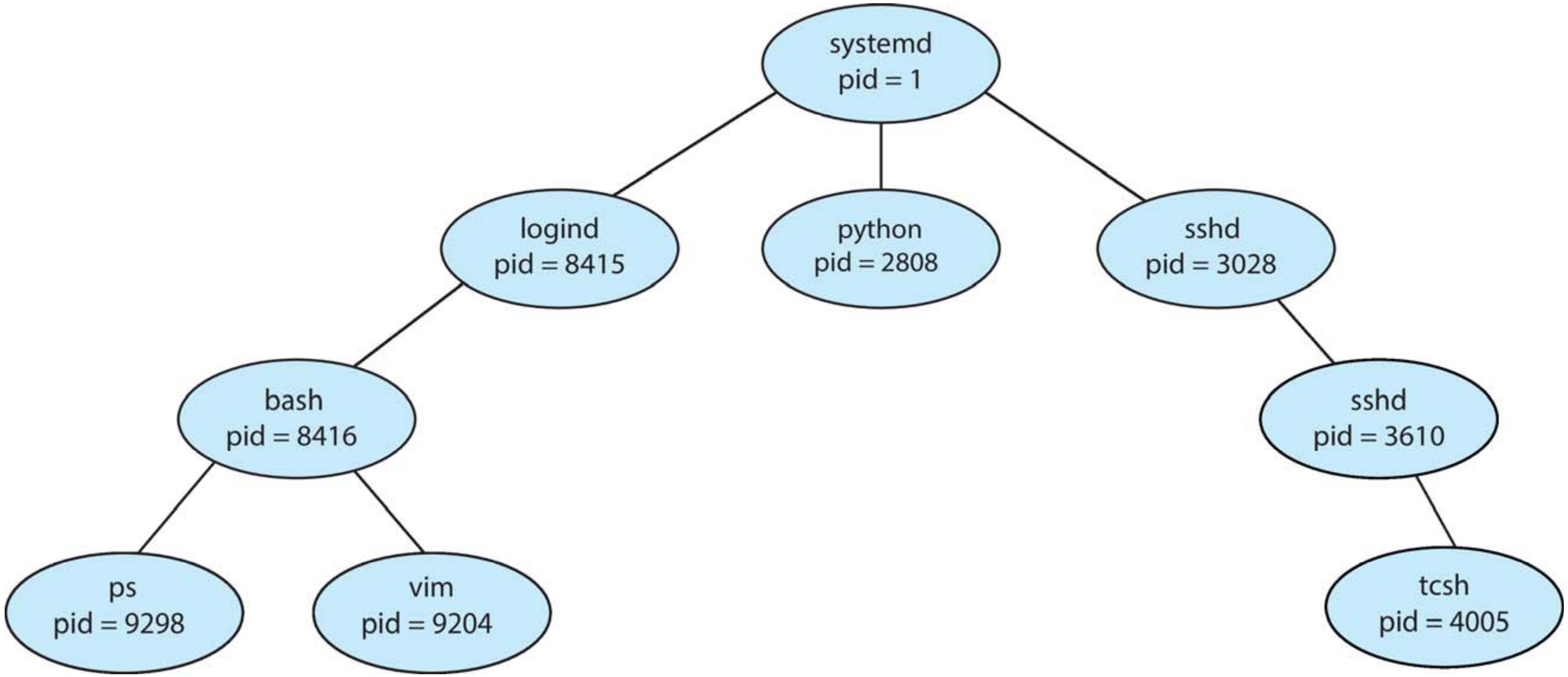
Process Creation

- **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**)
- Resource sharing options
  - Parent and children share all resources
  - Children share subset of parent’s resources
  - Parent and child share no resources



- Execution options
  - Parent and children execute concurrently
  - Parent waits until children terminate
- Address space
  - Child duplicate of parent
  - Child has a program loaded into it
- UNIX examples
  - **`fork()`** system call creates new process
  - **`exec()`** system call used after a **`fork()`** to replace the process’ memory space with a new program
  - Parent process calls **`wait()`** for the child to terminate

A Tree of Processes in Linux



C Program Forking Separate Process

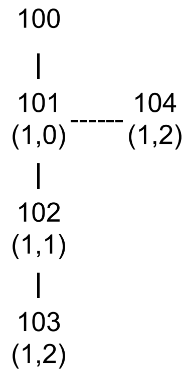
```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
pid_t pid;

    /* fork a child process */
    pid = fork();

    if (pid < 0) { /* error occurred */
       fprintf(stderr, "Fork Failed");
       return 1;
    }
    else if (pid == 0) { /* child process */
       execlp("/bin/ls","ls",NULL);
    }
    else { /* parent process */
       /* parent will wait for the child to complete */
       wait(NULL);
       printf("Child Complete");
    }

    return 0;
}
```

C Program Nest-Forking Process

```
int depth = 0;
void mylog(int l1, int l2)
{
    depth ++;
    printf("#%03d(%d,%d): pid=%d, ppid=%d\n",
        depth, l1, l2, getpid(), getppid());
    fflush(stdout);
}

void main_fork(int n1, int n2)
{
    int     l1, l2;
    pid_t       pid;

    for(l1 = 1; l1 <= n1; l1++) {
        pid = fork();
        if(pid > 0) {    /* parent */
            wait(NULL);
        } else if(pid == 0) { /* child */
            mylog(l1, 0);
            for(l2 = 1; l2 <= n2; l2++) {
                pid = fork();
                if(pid > 0) { /* parent */
                    wait(NULL);
                } else if(pid == 0) { /* child */
                    mylog(l1, l2);
                }
            }
        }
    }
}
```

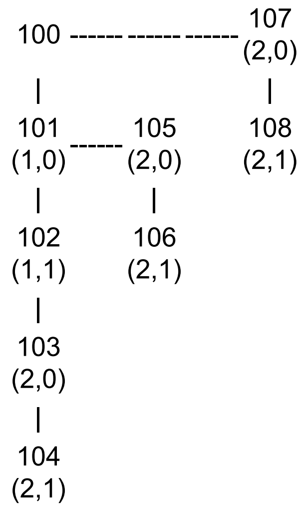
<u>./a.out 1 2</u>

```
#001(1,0): pid=101, ppid=100
#002(1,1): pid=102, ppid=101
#003(1,2): pid=103, ppid=102
#002(1,2): pid=104, ppid=101
```



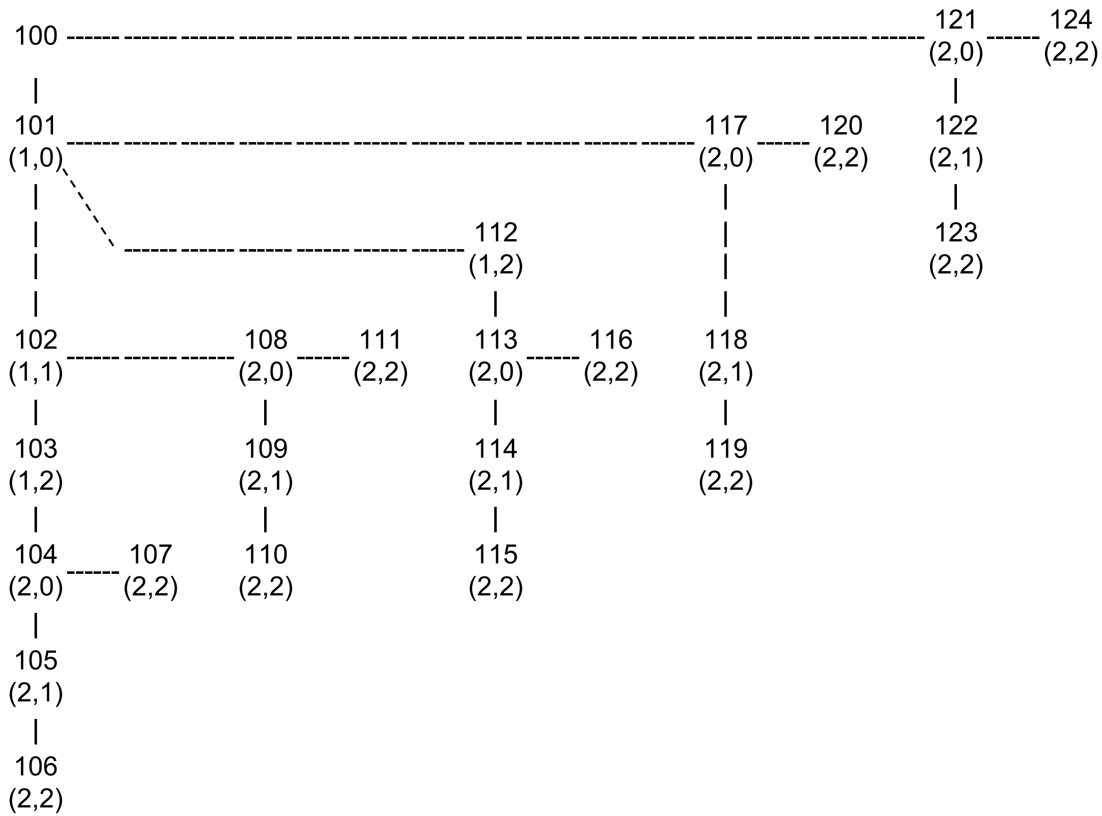
<u>./a.out 2 1</u>

```
#001(1,0): pid=101, ppid=100
#002(1,1): pid=102, ppid=101
#003(2,0): pid=103, ppid=102
#004(2,1): pid=104, ppid=103
#002(2,0): pid=105, ppid=101
#003(2,1): pid=106, ppid=105
#001(2,0): pid=107, ppid=100
#002(2,1): pid=108, ppid=107
```



C Program Nest-Forking Process

```
./a.out 2 2
#001(1,0): pid=101, ppid=100
#002(1,1): pid=102, ppid=101
#003(1,2): pid=103, ppid=102
#004(2,0): pid=104, ppid=103
#005(2,1): pid=105, ppid=104
#006(2,2): pid=106, ppid=105
#005(2,2): pid=107, ppid=104
#003(2,0): pid=108, ppid=102
#004(2,1): pid=109, ppid=108
#005(2,2): pid=110, ppid=109
#004(2,2): pid=111, ppid=108
#002(1,2): pid=112, ppid=101
#003(2,0): pid=113, ppid=112
#004(2,1): pid=114, ppid=113
#005(2,2): pid=115, ppid=114
#004(2,2): pid=116, ppid=113
#002(2,0): pid=117, ppid=101
#003(2,1): pid=118, ppid=117
#004(2,2): pid=119, ppid=118
#003(2,2): pid=120, ppid=117
#001(2,0): pid=121, ppid=100
#002(2,1): pid=122, ppid=121
#003(2,2): pid=123, ppid=122
#002(2,2): pid=124, ppid=121
```



Process Termination

- Process executes last statement and then asks the operating system to delete it using the **`exit()`** system call
  - Returns status data from child to parent (via **`wait()`**)
  - Process' resources are deallocated by operating system
- Parent may terminate the execution of children processes using the **`abort()`** system call. Some reasons for doing so:
  - Child has exceeded allocated resources
  - Task assigned to child is no longer required
  - The parent is exiting and the operating systems does not allow a child to continue if its parent terminates
- Some operating systems do not allow child to exists if its parent has terminated. If a process terminates, then all its children must also be terminated.
  - **cascading termination.** All children, grandchildren, etc. are terminated.
  - The termination is initiated by the operating system.
- The parent process may wait for termination of a child process by using the **`wait()`** system call. The call returns status information and the pid of the terminated process

```
pid = wait(&status);
```

- Terminates but no parent waiting (did not invoke **`wait()`**), process is a **zombie**
- If parent terminated without invoking **`wait()`** , process is an **orphan**

3.4 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

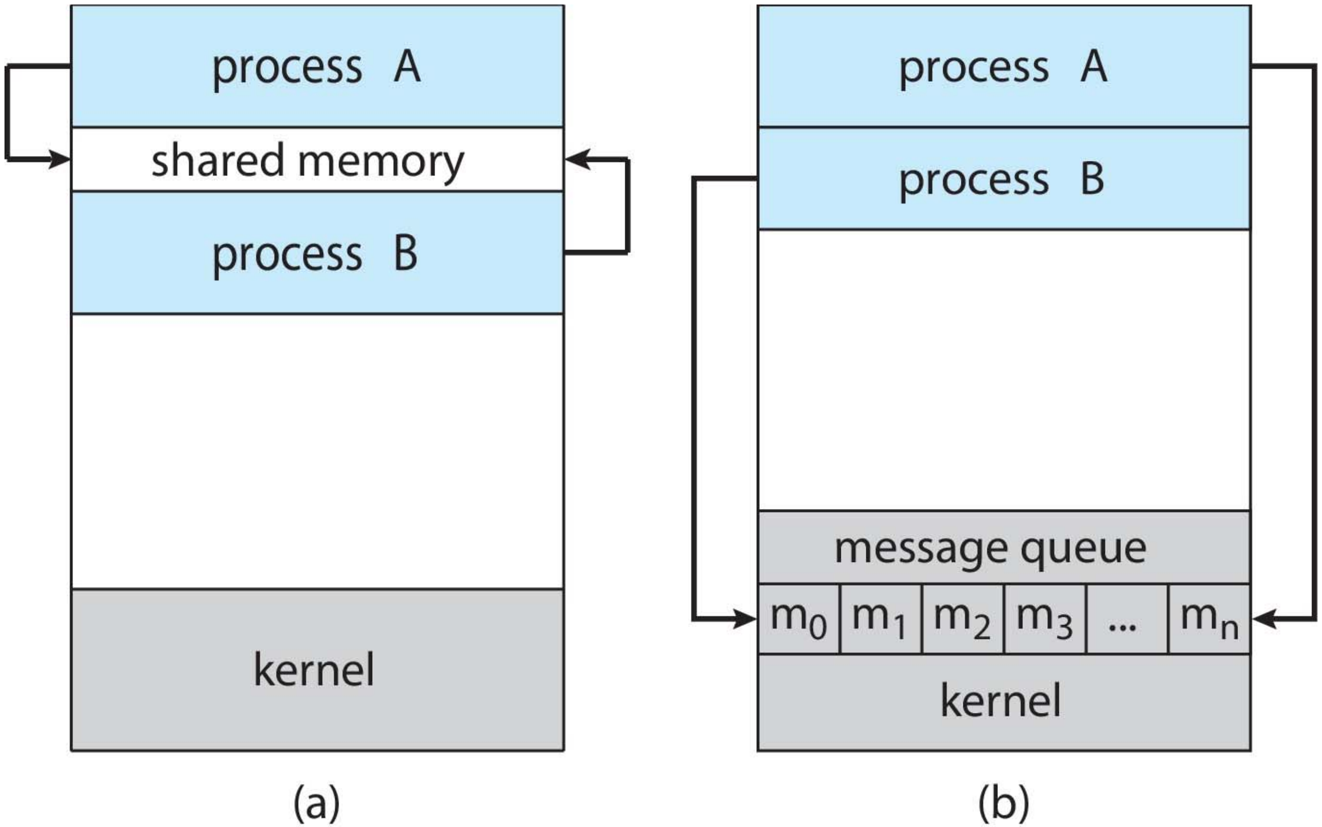
Cooperating processes need **interprocess communication** (**IPC**)

Two models of IPC

- **Shared memory**
- **Message passing**

Communications Models

(a) Shared memory. (b) 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

3.5 Interprocess Communication – Shared Memory

An area of memory shared among the processes that wish to communicate

The communication is under the control of the users processes not the operating system

Major issues is to provide mechanism that will allow the user processes to synchronize their actions when they access shared memory

Synchronization is discussed in great details in Chapters 6 & 7

Bounded-Buffer – Shared-Memory Solution

Shared data

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

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

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

Producer-Consumer Process – Shared Memory

Producer Process

```
item next_produced;

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

Consumer Process

```
item next_consumed;

while (true) {
        while (in == out)
                ; /* do nothing */
        next_consumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;

        /* consume the item in next consumed */
}
```

3.7 Interprocess Communication – Message Passing

Mechanism for processes to communicate and to synchronize their actions

Message system – processes communicate with each other without resorting to shared variables

IPC facility provides two operations:

- **`send`**(*message*)
- **`receive`**(*message*)

The *message* size is either fixed or variable

Message Passing (Cont.)

If processes *P* and *Q* wish to communicate, they need to:
- Establish a ***communication link*** between them
- Exchange messages via send/receive

Implementation issues:
- How are links established?
- Can a link be associated with more than two processes?
- How many links can there be between every pair of communicating processes?
- What is the capacity of a link?
- Is the size of a message that the link can accommodate fixed or variable?
- Is a link unidirectional or bi-directional?

Implementation of communication link
- Physical:
  - Shared memory
  - Hardware bus
  - Network
- Logical:
  - Direct or indirect
  - Synchronous or asynchronous
  - Automatic or explicit buffering

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

Properties of communication link

- Links are established automatically
- A link is associated with exactly one pair of communicating processes
- Between each pair there exists exactly one link
- The link may be unidirectional, but is usually bi-directional

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
- Properties of communication link
  - Link established only if processes share a common mailbox
  - A link may be associated with many processes
  - Each pair of processes may share several communication links
  - Link may be unidirectional or bi-directional
- Operations
  - create a new mailbox (port)
  - send and receive messages through mailbox
  - destroy a mailbox
- Primitives are defined as:
- **`send`**(*A, message*) – send a message to mailbox A
- **`receive`**(*A, message*) – receive a message from mailbox A

Synchronization

Message passing may be either blocking or non-blocking

Blocking is considered **synchronous**

- **Blocking send** -- the sender is blocked until the message is received
- **Blocking receive** -- the receiver is blocked until a message is available

Non-blocking is considered **asynchronous**

- **Non-blocking send** -- the sender sends the message and continue
- **Non-blocking receive** -- the receiver receives:
  - A valid message, or
  - Null message

Different combinations possible

- If both send and receive are blocking, we have a **rendezvous**

Producer-Consumer – ~~Shared Memory~~

Producer – ~~Shared Memory~~

```
message next_produced;

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

         send(next_produced);
}
```

Consumer – ~~Shared Memory~~

```
message next_consumed;

while (true) {
        receive(next_consumed)

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

Buffering

Queue of messages attached to the link.

Implemented in one of three ways

1. Zero capacity – no messages are queued on a link.
   Sender must wait for receiver (rendezvous)
2. Bounded capacity – finite length of *n* messages
   Sender must wait if link full
3. Unbounded capacity – infinite length
   Sender never waits

3.7 Examples of IPC Systems - POSIX

POSIX Shared Memory

- Process first creates shared memory segment
  `shm_fd = shm_open(name, O CREAT | O RDWR, 0666);`
- Also used to open an existing segment
- Set the size of the object

  `ftruncate(shm_fd, 4096);`
- Use **`mmap()`** to memory-map a file pointer to the shared memory object
- Reading and writing to shared memory is done by using the pointer returned by **`mmap()`**.

IPC POSIX Producer-Consumer

Producer

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>

int main()
{
/* the size (in bytes) of shared memory object */
const int SIZE = 4096;
/* name of the shared memory object */
const char *name = "OS";
/* strings written to shared memory */
const char *message_0 = "Hello";
const char *message_1 = "World!";

/* shared memory file descriptor */
int shm_fd;
/* pointer to shared memory obect */
void *ptr;

   /* create the shared memory object */
   shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);

   /* configure the size of the shared memory object */
   ftruncate(shm_fd, SIZE);

   /* memory map the shared memory object */
   ptr = mmap(0, SIZE, PROT_WRITE, MAP_SHARED, shm_fd, 0);

   /* write to the shared memory object */
   sprintf(ptr,"%s",message_0);
   ptr += strlen(message_0);
   sprintf(ptr,"%s",message_1);
   ptr += strlen(message_1);

   return 0;
}
```

Consumer

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>

int main()
{
/* the size (in bytes) of shared memory object */
const int SIZE = 4096;
/* name of the shared memory object */
const char *name = "OS";
/* shared memory file descriptor */
int shm_fd;
/* pointer to shared memory obect */
void *ptr;

   /* open the shared memory object */
   shm_fd = shm_open(name, O_RDONLY, 0666);

   /* memory map the shared memory object */
   ptr = mmap(0, SIZE, PROT_READ, MAP_SHARED, shm_fd, 0);

   /* read from the shared memory object */
   printf("%s",(char *)ptr);

   /* remove the shared memory object */
   shm_unlink(name);

   return 0;
}
```

Pipes

Acts as a conduit allowing two processes to communicate

Issues:

- Is communication unidirectional or bidirectional?
- In the case of two-way communication, is it half or full-duplex?
- Must there exist a relationship (i.e., ***parent-child***) between the communicating processes?
- Can the pipes be used over a network?

Ordinary pipes – cannot be accessed from outside the process that created it. Typically, a parent process creates a pipe and uses it to communicate with a child process that it created

Named pipes – can be accessed without a parent-child relationship

Ordinary Pipes

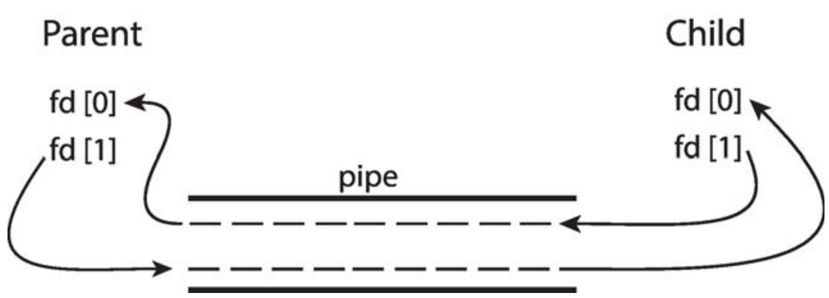
Ordinary Pipes allow communication in standard producer-consumer style

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




```
int main(void) {
    FILE * fp;
    if ((fp = popen("ls -l", "r")) == NULL) {
        perror("open failed!");
        return -1;
    }
    char but[256];
    while (fgets(but, 255, fp) != NULL)
        printf("%s", buf);
    pclose(fp);
    return 0;
}
```

Windows calls these **anonymous pipes**

Named Pipes

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

fifoserver.c

```
#define FIFO_FILE       "MYFIFO"

int main(void) {
        FILE *fp;
        char readbuf[80];

        /* Create the FIFO if it does not exist */
        umask(0);
        mknod(FIFO_FILE, S_IFIFO|0666, 0);

        while(1) {
                fp = fopen(FIFO_FILE, "r");
                fgets(readbuf, 80, fp);
                printf("Received string: %s\n", readbuf);
                fclose(fp);
        }
        return(0);
}
```

fifoclient.c

```
#define FIFO_FILE       "MYFIFO"

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

        if ( argc != 2 ) {
                printf("USAGE: fifoclient [string]\n");
                exit(1);
        }
        if((fp = fopen(FIFO_FILE, "w")) == NULL) {
                perror("fopen");
                exit(1);
        }
        fputs(argv[1], fp);

        fclose(fp);
        return(0);
}
```

3.8 Communications in Client-Server Systems

Sockets

Remote Procedure Calls

Sockets

A **socket** is defined as an endpoint for communication

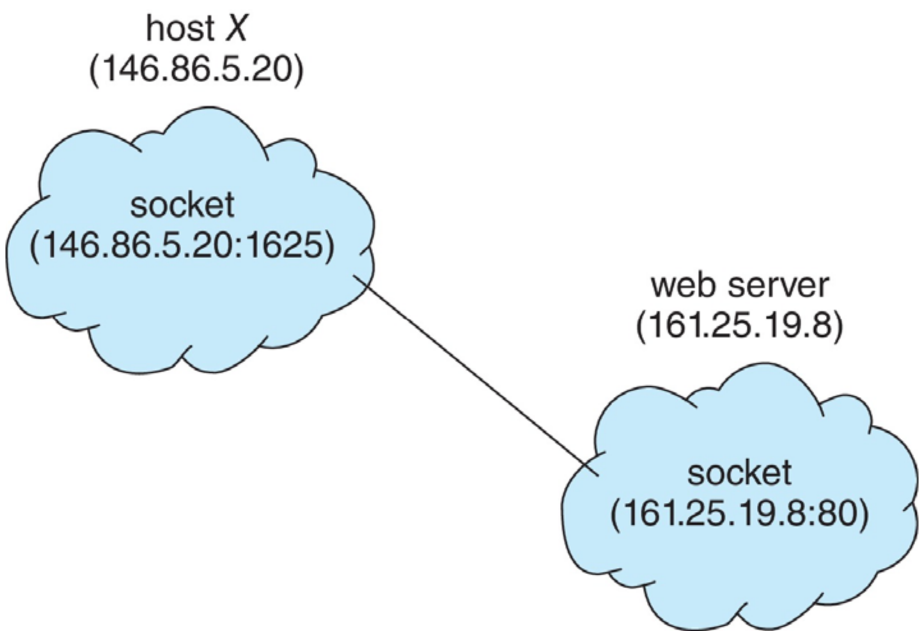
Concatenation of IP address and **port** – a number included at start of message packet to differentiate network services on a host

The socket **161.25.19.8:1625** refers to port **1625** on host **161.25.19.8**

Communication consists between a pair of sockets

All ports below 1024 are ***well known***, used for standard services

Special IP address 127.0.0.1 (**loopback**) to refer to system on which process is running



Remote Procedure Calls

Remote procedure call (RPC) abstracts procedure calls between processes on networked systems

- Again uses ports for service differentiation

Stubs – client-side proxy for the actual procedure on the server

The client-side stub locates the server and **marshalls** the parameters

The server-side stub receives this message, unpacks the marshalled parameters, and performs the procedure on the server

On Windows, stub code compile from specification written in **Microsoft Interface Definition Language** (**MIDL**)

Data representation handled via **External Data Representation** (**XDL**) format to account for different architectures

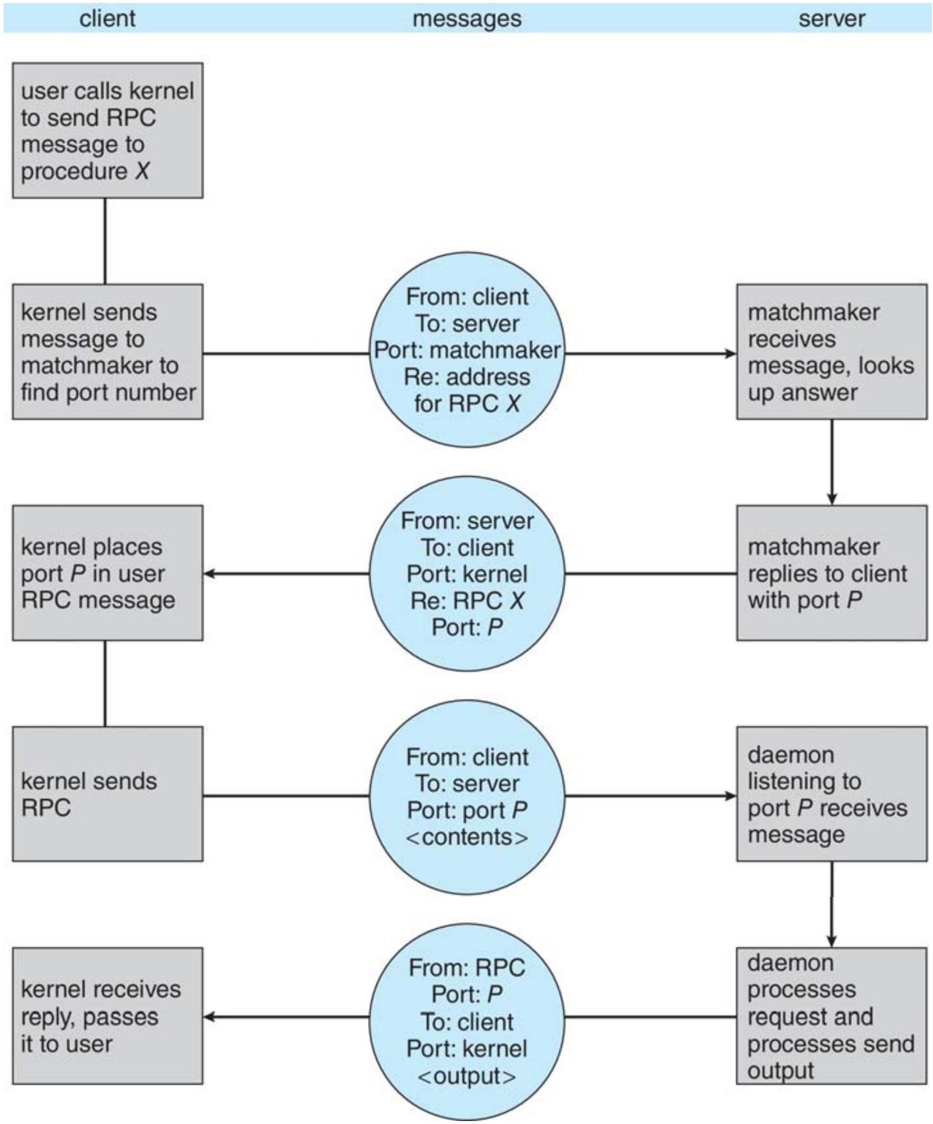
- **Big-endian** and **little-endian**

Remote communication has more failure scenarios than local

- Messages can be delivered ***exactly once*** rather than ***at most once***

OS typically provides a rendezvous (or **matchmaker**) service to connect client and server

Execution of RPC



Chapter 3: Processes

- Process Concept
- Process Scheduling
- Operations on Processes
- Interprocess Communication
- IPC in Shared-Memory Systems
- IPC in Message-Passing Systems
- Examples of IPC Systems
- Communication in Client-Server Systems