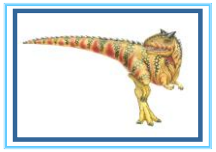
Chapter 3: Processes



Chapter 3: Processes

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

Objectives

- To introduce the notion of a process -- a program in execution, which forms the basis of all computation
- To describe the various features of processes, including scheduling, creation and termination, and communication
- To describe communication in client-server systems

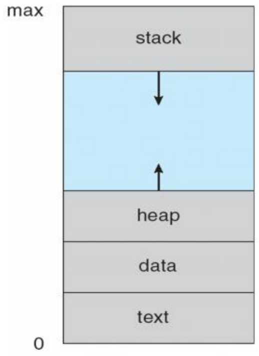
Process Concept

- An operating system executes a variety of programs:
  - Batch system – jobs
  - Time-shared systems – user programs or tasks
- Textbook uses the terms *job* and *process* almost interchangeably
- Process – a program in execution; process execution must progress in sequential fashion
- A process includes:
  - program counter
  - stack
  - data section

The Process

- Multiple parts
  - The program code, also called **text section**
  - Current activity including **program counter**, processor registers
  - **Stack** containing temporary data
    - Function parameters, return addresses, local variables
  - **Data section** containing global variables
  - **Heap** containing memory dynamically allocated during run time
- Program is passive entity, process is active
  - Program becomes process when executable file loaded into memory
- Execution of program started via GUI mouse clicks, command line entry of its name, etc
- One program can be several processes
  - Consider multiple users executing the same program

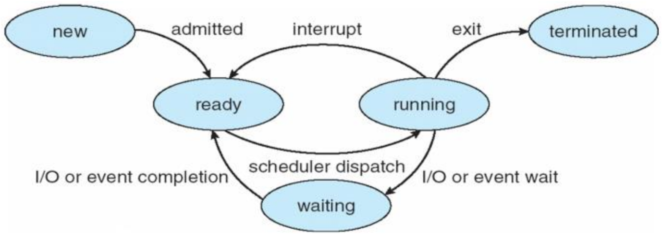
Process in Memory



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

Diagram of Process State

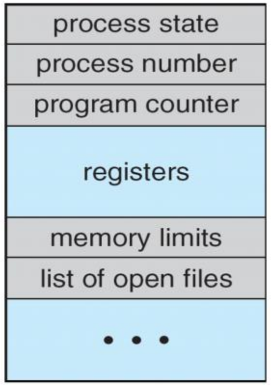


Process Control Block (PCB)

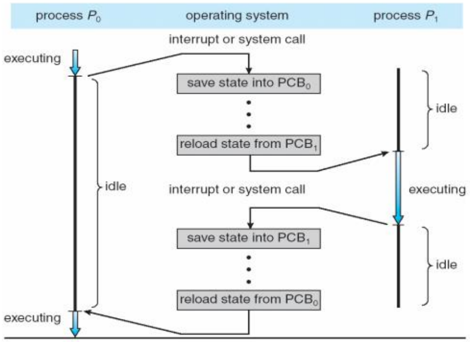
Information associated with each process

- Process state
- Program counter
- CPU registers
- CPU scheduling information
- Memory-management information
- Accounting information
- I/O status information

Process Control Block (PCB)



CPU Switch From Process to Process



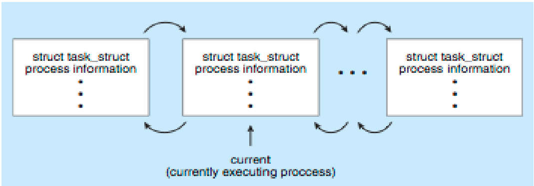
Process Scheduling

- Maximize CPU use, quickly switch processes onto CPU for time sharing
- **Process scheduler** selects among available processes for next execution on CPU
- Maintains **scheduling queues** of processes
  - **Job queue** – set of all processes in the system
  - **Ready queue** – set of all processes residing in main memory, ready and waiting to execute
  - **Device queues** – set of processes waiting for an I/O device
  - Processes migrate among the various queues

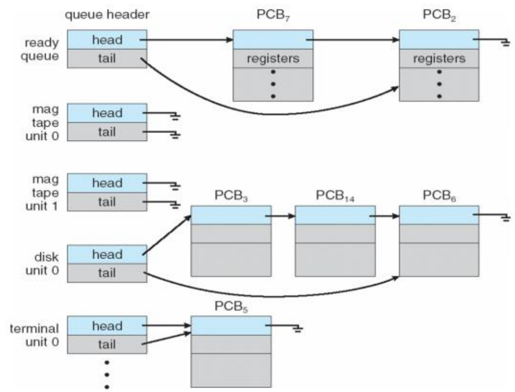
Process Representation in Linux

- Represented by the C structure `task_struct`

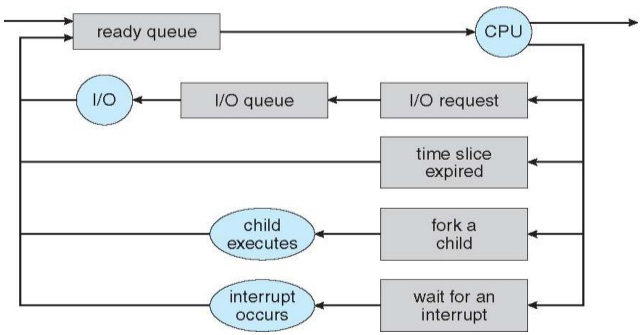
```
pid t pid; /* process identifier */
long state; /* state of the process */
unsigned int time slice /* scheduling information */
struct task struct *parent; /* this process's parent */
struct list head children; /* this process's children */
struct files struct *files; /* list of open files */
struct mm struct *mm; /* address space of this pro */
```



Ready Queue And Various I/O Device Queues



Representation of Process Scheduling



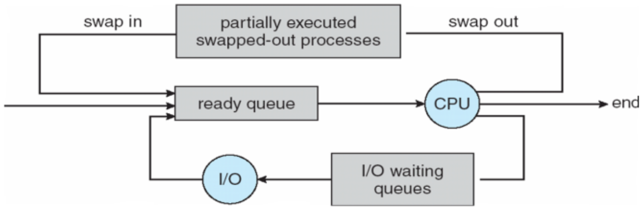
Schedulers

- **Long-term scheduler** (or job scheduler) – selects which processes should be brought into the ready queue
- **Short-term scheduler** (or CPU scheduler) – selects which process should be executed next and allocates CPU
  - Sometimes the only scheduler in a system

Schedulers (Cont.)

- Short-term scheduler is invoked very frequently (milliseconds) ⇒ (must be fast)
- Long-term scheduler is invoked very infrequently (seconds, minutes) ⇒ (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

Addition of Medium Term Scheduling



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 overhead; the system does no useful work while switching
  - The more complex the OS and the PCB -> longer the context switch
- Time dependent on hardware support
  - Some hardware provides multiple sets of registers per CPU -> multiple contexts loaded at once

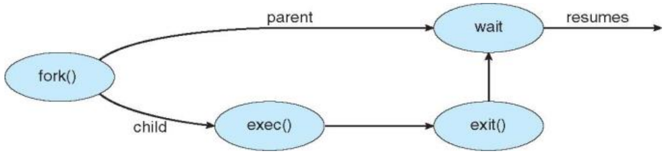
Process Creation

- **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
  - Parent and children share all resources
  - Children share subset of parent's resources
  - Parent and child share no resources
- Execution
  - Parent and children execute concurrently
  - Parent waits until children terminate

Process Creation (Cont.)

- 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

Process Creation



C Program Forking Separate Process

```
#include <sys/types.h>
#include <studio.h>
#include <unistd.h>
int main()  {
     pid_t  pid;
     /* fork another 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 */
          wait (NULL);
          printf ("Child Complete");
     }
     return 0;
}
```


A Simple fork() Example

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

int main ( void )  {

        printf("Message before fork\n");

        fork();

        printf("Message after fork\n");

        return 0;
}
```

- a simple fork example
- Message after fork is printed **twice** !!

```
File  Edit  View  Terminal  Help
lucid@ubuntu:~/Downloads$ ./Fork1
Message before fork
Message after fork
lucid@ubuntu:~/Downloads$ Message after fork

lucid@ubuntu:~/Downloads$
```

- **fork1.c**

Self Identification

```c
#include <stdio.h>
#include <unistd.h>

int main ( void )  {

        int forkResult;

        printf("process id : %i\n",getpid());
        forkResult = fork();
        printf("process id : %i - result : %d\n",
                getpid(), forkResult);

        return 0;
}
```

- for the parent process fork returns **child's pid**
- for the child process fork returns **0**

```
File  Edit  View  Terminal  Help
lucid@ubuntu:~/Downloads$ ./Fork2
process id : 2682
process id : 2682 - result : 2683
lucid@ubuntu:~/Downloads$ process id : 2683 - result : 0
```

- **fork2.c**

Process Differentiation by source code

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main ( void )  {

        printf("(%i) Parent does something...\n", getpid());

        if(fork())  {  // Parent
                printf("(%i) Parent do completely different stuff\n",getpid());
        } else {       // Child
                printf("(%i) Child can do some stuff\n",getpid());
        }


        exit(0);
}
```

```
File  Edit  View  Terminal  Help
lucid@ubuntu:~/Downloads$ ./Fork3
(2767) Parent does something...
(2767) Parent do completely different stuff
lucid@ubuntu:~/Downloads$ (2768) Child can do some stuff
```

- **fork3.c**

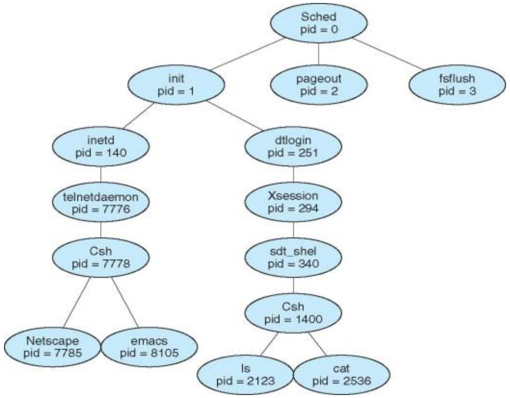
A Simple exec() Example

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main ( void )  {

        printf("Parent does stuff and then calls fork...\n");

        if(fork())  { // Parent
                printf("... parent do something completely different\n");
        } else  {     // Child
                printf("Child runs an executable...\n");
                execl("/bin/ls","/bin/ls","-l","/etc/apache2/conf.d/",NULL);
        }

        exit(0);
}
```

```
lucid@ubuntu:~/Downloads$ ./Exec
Parent does stuff and then calls fork...
... parent do something completely different
lucid@ubuntu:~/Downloads$ Child runs an executable...
/bin/ls: cannot access /etc/apache2/conf.d/: No such file or directory
```

- **exec.c**

A Tree of Processes on Solaris

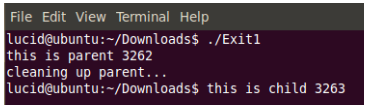


Process Termination

- Process executes last statement and asks the operating system to delete it (**exit**)
  - Output data from child to parent (via **wait**)
  - Process' resources are deallocated by operating system
- Parent may terminate execution of children processes (**abort**)
  - Child has exceeded allocated resources
  - Task assigned to child is no longer required
  - If parent is exiting
    - Some operating systems do not allow child to continue if its parent terminates
      - All children terminated - **cascading termination**

atexit() example

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

void parentCleaner ( void );

int main ( void )  {

        if(fork())  { // parent process
                atexit(parentCleaner);
                printf("this is parent %i\n",getpid());
        } else  {     // child process
                printf("this is child %i\n",getpid());
        }
        exit(0);
}

void parentCleaner ( void )  {

        printf("cleaning up parent...\n");

}
```



- registers a function to clean up resource at process termination

- **atexit.c**

Zombie example
non-terminating parent

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main ( void ) {

        if(fork())  {  // Parent
                printf("Running parent, pid : %i\n",getpid());
                while(1);
        } else  {    // Child
                printf("Terminating child, pid : %i\n", getpid());
                exit(0);
        }

        exit (0);
}
```

```
lucid@ubuntu:~/Downloads$ ps -ef | grep Zombie
lucid     3380  2182 71 03:42 pts/0    00:00:21 ./Zombie1
lucid     3381  3380  0 03:42 pts/0    00:00:00 [Zombie1] <defunct>
lucid     3402  3382  0 03:43 pts/1    00:00:00 grep --color=auto Zombie
lucid@ubuntu:~/Downloads$
```

- zombie1.c

Zombie example
non-terminating child

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main ( void ) {

        if(fork())  {  // Parent
                printf("Running parent, pid : %i\n",getpid());
                exit(0);
        } else  {    // Child
                printf("Terminating child, pid : %i\n", getpid());
                while(1);
        }

        exit (0);
}
```

```
lucid@ubuntu:~/Downloads$ ps -ef | grep Zombie
lucid     3467     1 77 03:45 pts/0    00:00:29 ./Zombie2
lucid     3473  3382  0 03:46 pts/1    00:00:00 grep --color=auto Zombie
lucid@ubuntu:~/Downloads$
```

- zombie2.c

wait() Example

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

#define numOfChilds 5
int main ( void )  {

        int i;
        int child_status;
        pid_t pid[numOfChilds];
        pid_t wpid;

        for (i = 0; i < numOfChilds; i++)  {
                if ((pid[i] = fork()) == 0)  {
                        exit(100+i);            // create & exit child
                }
        }

        for (i = 0; i < numOfChilds; i++) {
                wpid = wait(&child_status);     // wait for child
                if (WIFEXITED(child_status))  { // check exit status
                        printf("Child %d terminated with exit status %d\n",
                                wpid, WEXITSTATUS(child_status));
                } else {
                        printf("Child %d terminate abnormally\n", wpid);
                }
        }
        exit(0);
}
```

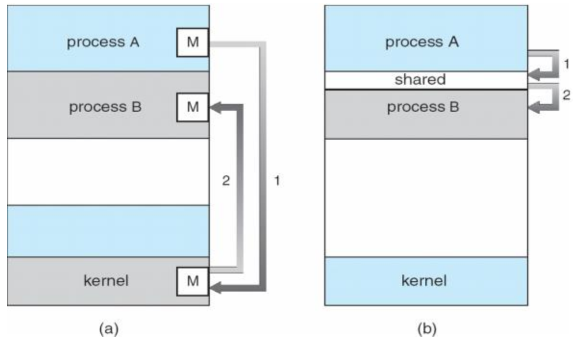
```
lucid@ubuntu:~/Downloads$ ./Wait1
Child 3630 terminated with exit status 100
Child 3631 terminated with exit status 101
Child 3633 terminated with exit status 103
Child 3634 terminated with exit status 104
Child 3632 terminated with exit status 102
lucid@ubuntu:~/Downloads$
```

- **wait1.c**

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



Cooperating Processes

- **Independent** process cannot affect or be affected by the execution of another process
- **Cooperating** process can affect or be affected by the execution of another process
- Advantages of process cooperation
  - Information sharing
  - Computation speed-up
  - Modularity
  - Convenience

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;
```

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

Bounded-Buffer – Producer

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


Bounded Buffer – Consumer

```
while (true) {
    while (in == out)
        ; // do nothing -- nothing to consume

    // remove an item from the buffer
    item = buffer[out];
    out = (out + 1) % BUFFER SIZE;
return item;
}
```

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*) – 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
- Implementation of communication link
  - physical (e.g., shared memory, hardware bus)
  - logical (e.g., logical properties)

Implementation Questions

- 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?

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

Indirect Communication

- Operations
  - create a new mailbox
  - 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

Indirect Communication

- Mailbox sharing
  - P_1, P_2, and P_3 share mailbox A
  - P_1, sends; P_2 and P_3 receive
  - Who gets the message?
- Solutions
  - Allow a link to be associated with at most two processes
  - Allow only one process at a time to execute a receive operation
  - Allow the system to select arbitrarily the receiver. Sender is notified who the receiver was.

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

Buffering

- Queue of messages attached to the link; implemented in one of three ways
  1. Zero capacity – 0 messages
     Sender must wait for receiver (rendezvous)
  2. Bounded capacity – finite length of *n* messages
     Sender must wait if link full
  3. Unbounded capacity – infinite length
     Sender never waits

Examples of IPC Systems - POSIX

- POSIX 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

```
shared memory = (char *) shmat(id, NULL, 0);
```

  - Now the process could write to the shared memory

```
sprintf(shared memory, "Writing to shared memory");
```

  - When done a process can detach the shared memory from its address space

```
shmdt(shared memory);
```


Examples of IPC Systems - Mach

- Mach communication is message based
  - Even system calls are messages
  - Each task gets two mailboxes at creation- Kernel and Notify
  - Only three system calls needed for message transfer

```
msg_send(), msg_receive(), msg_rpc()
```

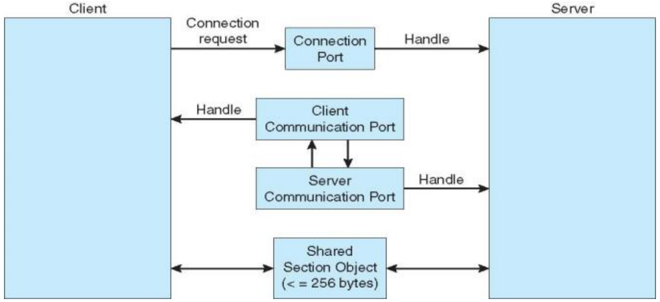
  - Mailboxes needed for commuication, created via

```
port_allocate()
```

Examples of IPC Systems – Windows XP

- Message-passing centric via **local procedure call (LPC)** facility
  - Only works between processes on the same system
  - Uses ports (like mailboxes) to establish and maintain communication channels
  - Communication works as follows:
    - The client opens a handle to the subsystem's connection port object.
    - The client sends a connection request.
    - The server creates two private communication ports and returns the handle to one of them to the client.
    - The client and server use the corresponding port handle to send messages or callbacks and to listen for replies.

Local Procedure Calls in Windows XP



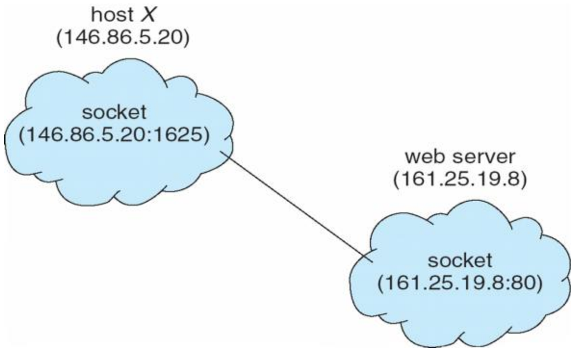
Communications in Client-Server Systems

- Sockets
- Remote Procedure Calls
- Pipes
- Remote Method Invocation (Java)

Sockets

- A **socket** is defined as an *endpoint for communication*
- Concatenation of IP address and port
- The socket **161.25.19.8:1625** refers to port **1625** on host **161.25.19.8**
- Communication consists between a pair of sockets

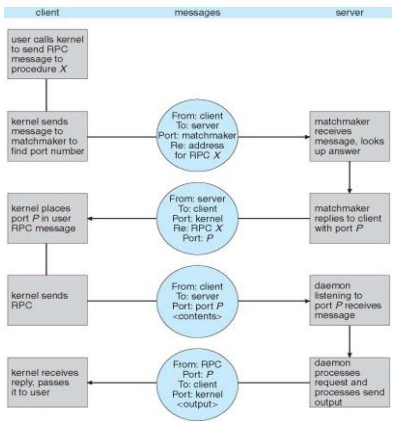
Socket Communication



Remote Procedure Calls

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

Execution of RPC



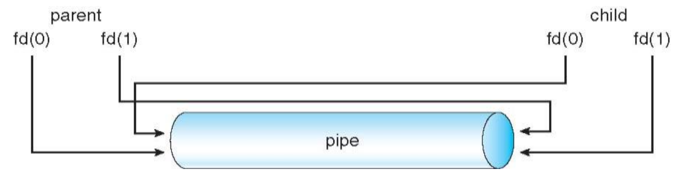
Pipes

- 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

- **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

Ordinary 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

End of Chapter 3