**Process Concept**

A process, which is a program in execution is the unit of work in a modem time-sharing system.

**3.1 Process Concept**

A batch system executes jobs, whereas a time-shared system has user programs, or tasks.

**3.1.1 The Process**

* A process is more than the program code, which is sometimes known as the text section.
* It also includes:
* The current activity, represented by **program counter** and the contents of the processor's registers.
* **process stack**, which contains temporary data (such as function parameters, return addresses, and local variables)
* **Data section**, which contains global variables.
* A **heap**, which is memory that is dynamically allocated during process run time.
* A program is a **passive entity** not a process, such as a file containing a list of instructions stored on disk (often called an executable file).
* In contrast, a process is an **active entity**, with a program counter specifying the next instruction to execute and a set of associated resources.
* A program **becomes a process** when an executable file is loaded into memory.
* **Two common techniques for loading executable files:** are double-clicking an icon representing it and entering the name on the command line (as in prog. exe or a. out). **Two separate execution sequences.**
* A process itself can be an execution environment for other code. EX: JVM(JAVA).

**3.1.2 Process State**

As a process executes, it changes state.

By the current activity of that process; a process may be in one of the following states:

* New. The process is being created.
* Running. Instructions are being executed.
* Waiting. The process is waiting for some event to occur (such as an 1/0 completion or reception of a signal).
* Ready. The process is waiting to be assigned to a processor.
* Terminated. The process has finished execution.

Only one process can be running on any processor at any instant. Many processes may be **ready** and **waiting.**

**3.1.3 Process Control Block**

Each process is represented in the OS by **a process control block** (**PCB)** - also called a task control block.

A PCB contains many pieces of information associated with a specific process, including these:

**• Process state:** new, ready, running, waiting, halted, and so on.

**• Program counter.** Indicates the address of the next instruction to be executed for this process.

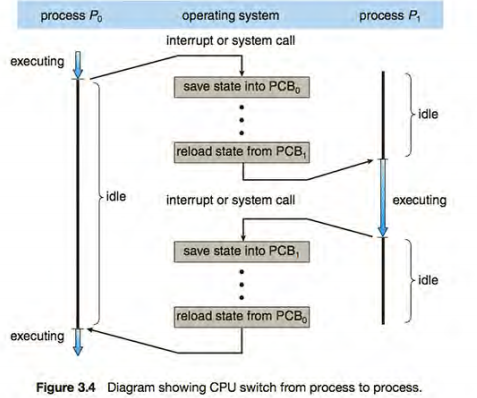
**• CPU registers** vary in number and type, They include accumulators, index registers, stack pointers, and general-purpose registers, plus any condition-code information. Along with the program counter, this state information must be saved when an **interrupt** occurs, to allow the process to be continued correctly afterward (Figure 3.4).

**• CPU-scheduling information:** includes a process priority, pointers to scheduling queues, and any other scheduling parameters.

**• Memory-management information:** include such items as the value of the base and limit registers and the page tables, or the segment tables, depending on the memory system used by the operating system .

**• Accounting information:** includes the amount of CPU and real time used, time limits, and so on.

**• 110 status information:** includes the list of I/O devices allocated to the process, a list of open files, and so on.



**3.1.4 Threads**

* A process is a program that performs a single **thread** of execution.
* This single thread of control allows the process to perform only one task at a time.
* Most modern operating systems allow a process to have multiple threads.

**3.2 Process Scheduling**

* The objective of **multiprogramming** is to have some process running at all times, to maximize CPU utilization.
* The objective of **time sharing** is to switch the CPU among processes so frequently that users can interact with each program while it is running.
* To meet these objectives, **the process scheduler** selects an available process (possibly from a set of several available processes) for program execution on the CPU.

**3.2.1 Scheduling Queues**

* As processes enter the system, they are put into a **job queue**, which consists of all processes in the system.
* The processes in main memory (ready and waiting to execute) are kept on a **linked list** called the **ready queue**.
* **A ready-queue header** contains **pointers** to the first and final PCBs in the list.
* Each PCB includes a pointer field that points to the next PCB in the ready queue.
* The list of processes waiting for a particular I/O device is called a **device queue**. Each device has its own device queue.

A new process is **initially** put in the ready queue. It waits there until it is selected for execution, or **dispatched**. Once the process is allocated the CPU and is executing\_ one of several events could occur:

• The process could issue an **I/O request** and then be placed in an I/O queue.

• The process could create a new **child** process and wait for the child's termination.

• The process could be removed forcibly from the CPU, as a result of an **interrupt**, and be put back in the ready queue.

**3.2.2 Schedulers**

* The selected process by the operating system is carried out by the appropriate scheduler.
* The **long-term scheduler**, **or job scheduler**, selects processes from this pool and loads them into memory for execution.
* The **short-term scheduler, or CPU scheduler**, selects from among the processes that are ready to execute and allocates the CPU to one of them.

|  |  |
| --- | --- |
| **short-term scheduler** | **long-term scheduler** |
| * Select a new process for the CPU frequently. * A process may execute for only a few milliseconds before waiting for an I/O request. | * make the degree of multiprogramming (the number of processes in memory stable , then the average rate of process creation must be equal to the average departure rate of processes leaving the system. * It is important that the long-term scheduler make a careful selection. |
| Often, the short-term scheduler executes at least **once every 100 milliseconds**. Because of the short time between executions, it’s fast. If it takes 10 milliseconds to decide to execute a process for 100 milliseconds, then 10/(100 + 10) = 9 percent of the CPU is being used  (Wasted) simply for scheduling the work. | The long-term scheduler executes much less frequently; minutes may separate the creation of one new process and the next. |
| If all processes are I/O bound, the ready queue will almost always be empty, and the short-term scheduler will have little to do. | It is important that the long-term scheduler select a good process mix **of I/O bound** and **CPU-bound processes.** |
|  | On some systems, the long-term scheduler may be absent or minimal. For example, time-sharing systems such as UNIX and Microsoft Windows (no short-term). |

* Some operating systems, such as time-sharing systems, may introduce an additional, **intermediate** level of scheduling. **This medium-term scheduler** can be advantageous to **remove** a process from memory .Later, the process can be **reintroduced** into memory, and its execution can be continued **where** it left off. This scheme is called **swapping**.
* **I/O-bound** process: spends more of its time doing I/O than it spends doing computations.
* **A CPU-bound** process, in contrast, using more of its time doing computations.

**3.3 Operations on Processes**

The processes in most systems can execute concurrently, and they may be created and deleted dynamically.

**3.3.1 Process Creation**

* The creating process is called a **parent** process, and the new processes are called the **children** of that process.
* Each of these new processes may in turn create other processes, forming a **tree** of processes.
* Process identifier (or **pid**), is an integer number provides a **unique value for each process in the system**, and it can be **used as an index to access various attributes of a process** within the kernel.

|  |  |
| --- | --- |
| The **init** process (has a pid of 1) | Serves as the root parent process for all user processes.  \*\*\*Once the system has booted, the **init** process can also create various user processes, such as **a web or print server**, an ssh server, and the like. |
| The **kthreadd** process | For creating additional processes that perform tasks on behalf of the kernel. |
| The **sshd** process | For managing clients that connect to the system by using ssh (which is short for secure shell). |
| The login process | For managing clients that directly log onto the system. |

IN GENERAL,

1-when a process creates a child process, that child process will need certain resources (CPU time, memory, files, I/O devices) to accomplish its task.

2- Resources are obtained directly from the OS, or it may be constrained to a subset of the resources of the parent process.

3- In addition to supplying resources, the parent process may pass along initialization data (input) to the child process.

* When a process **creates** a new process**, two possibilities for execution exist:**

1. The parent continues to execute concurrently with its children.

2. The parent waits until some or all of its children have terminated.

There are also **two address-space possibilities** for the new process:

1. The child process is a duplicate of the parent process (it has the same program and data as the parent).

2. The child process has a **new** program loaded into it.

In UNIX, each process is identified by its process identifier.

* A new process is **created** by the fork() system call. The new process consists of a **copy** of the address space of the original process.
* This mechanism allows the parent process **to communicate easily with its child process**.
* Both processes (the parent and the child) **continue execution** at the instruction after the fork(), with one difference:
* The return code for the fork() is zero for the new (child) process, (nonzero) process identifier to the parent.
* After a fork 0 system call, one of the two processes typically **uses the** exec () system call to replace the process's memory space with a new program.
* exec () loads a binary file into memory (destroying the memory image of the program containing the exec () system call) and starts its execution.
* The parent can then create more children; or, if it has nothing else to do while the child runs, it can issue a **wait ()** system call to **move itself off the ready queue** until the termination of the child.
* Because the call to exec () overlays the process's address space with a new program, the call to exec () **does not return control unless an error occurs**.

The C program we now have two different processes running copies of the same program.

The only difference is that the value of **pid** for the child process is **zero**, while that for the parent is an integer value greater than zero (in fact, it is the actual pid of the child process).

* The child process **inherits** privileges and scheduling attributes from the parent, as well certain resources, such as open files.
* The child process then **overlays** its address space with the UNIX command **/bin/Is** (used to get a directory listing) using the **execlp ()** system call.
* (Execlp () is a version of the exec () system call).
* The parent **waits** for the child process to complete with the **wait ()** system call.
* When the child process completes (by either implicitly or explicitly invoking **exit()**), the parent process **resumes** from the call to wait (), where it completes using the exit() system call.
* Lookp117

**3.3.2 Process Termination**

* A process terminates when it **finishes executing its final statement** and asks the operating system to delete it by using the **exit ()** system call.
* The process may return **a status value** (typically an integer) to its parent process (via the wait() system call).
* All the resources of the process are deallocated by the OS.
* A process can cause the termination of another process via an appropriate system call (for example, TerminateProcess () in Windows).

Note that a parent needs to know the **identities** of its children if it is *to terminate* them.

* **A parent may terminate the execution of one of its children for a variety of reasons, such as these**:

• The child has exceeded its usage of some of the resources that it has been allocated. (To determine whether this has occurred, the parent must have a mechanism to inspect the state of its children.)

• The task assigned to the child is no longer required.

• The parent is exiting, and the operating system does not allow a child to continue if its parent terminates.

In some systems, if a process terminates, then all its children must also be terminated. This phenomenon, referred to as **cascading termination**.

In Linux and UNIX systems,

* we can terminate a process by using the exit() system call, providing an exit status as a parameter:

I\* exit with status 1 \*I

exit (l);

**exit () may be called either directly (as shown above) or indirectly (by a return statement in main ())**

* A **parent** process may wait for the termination of a child process by using the **wait ()** system call.
* The wait () system call is passed a parameter that allows the parent to obtain the exit status of the child.
* returns the process identifier of the terminated child so that the parent can tell which of its children has terminated:

pid\_t pid;

int status;

pid = wait (&status);

* When a process terminates**, its resources are deallocated** by the operating system.
* However, its entry in the process table must remain there until the parent calls wait(), **because** the process table contains the process's exit status.
* A process that has terminated, but whose parent has **not** yet called wait (), is known as a **zombie** process.
* All processes **transition** to this state when they terminate, but generally they exist as zombies only **briefly**.
* Once the parent calls wait (), the **process identifier** of the zombie process **and its entry** in the process table are **released**.
* **If a parent did not invoke wait ()** and instead terminated, thereby leaving its child processes as **orphans**.

**1-**Linux and UNIX address this scenario by assigning the **init** process as the new parent to orphan processes.

2-The init process periodically invokes wait (), thereby allowing the exit status of any orphaned process to be collected and releasing the orphan's process identifier and process-table entry.

**3.2.3 Context Switch**

* On general-purpose system: **interrupts** cause the operating system to change a CPU from its current task and to run a kernel routine.
* When an interrupt occurs, the system needs to **save the current context** of the process running on the CPU so that it can restore that context when its processing is done, essentially suspending the process and then resuming it.
* The context is **represented in the PCB** of the process.
* It includes the **value of the CPU registers**, **the process state** and **memory-management information.**
* **We perform a state save of the current state of the CPU, be it in kernel or user mode, and then a state restore to resume operations.**
* **Switching the CPU to another process** requires performing a state save of the **current** process and a state restore of a **different** process. This task is known as a **context switch**.

**3.4 interprocess Communication**

* Processes executing **concurrently** in the operating system may be either **independent** processes or cooperating (affect or be affected by the other processes) processes.
* **Cooperating processes** require an **interprocess communication (IPC)** mechanism that will allow them to exchange data and information.

There are two fundamental models of interprocess communication: **shared memory** and **message** **passing**.



|  |  |
| --- | --- |
| shared-memory model | Message passing model |
| A region of memory that is shared by cooperating processes is established. Processes can then exchange information by reading and writing data to the shared region. | In the message-passing model, communication takes place by means of messages exchanged between the cooperating processes. |
| faster than message passing | \*Slower because it’s implemented using system calls and thus require the more time-consuming task of kernel intervention.  \*Useful for exchanging smaller amounts of data, because no conflicts need be avoided. |
| A shared-memory region resides in the address space of the process creating the shared-memory segment.  Other processes that wish to communicate using this shared-memory segment must attach it to their address space. | Provides a **mechanism** to allow processes to communicate and to synchronize their actions without sharing the same address space.  \*useful (easier to implement) in **a distributed environment,** where the communicating processes may reside on different computers connected by a network. |
| System calls are required only to establish shared memory regions.  Then: all accesses are treated as routine memory accesses, and no assistance from the kernel is required. | A message-passing facility provides at least two operations:  send(message) receive(message) |
| For example, a web server produces (that is, provides) HTML files and images, which are consumed (that is, read) by the client web browser requesting the resource. | We generally think of a server as a producer and a client as a consumer. For example, a web server produces (that is, provides) HTML files and images, which are consumed (that is, read) by the client web browser requesting the resource. |
| Shared memory suffers from cache coherency issues, which arise because shared data migrate among the several caches. | Research indicates that it provides better performance than shared memory on systems with several cores |

**If processes *P* and *Q* want to communicate:** a ***communication link*** must exist between them.

Several methods for logically implementing a link and the send()/receive() operations:

•Direct or indirect communication (Naming)

• Synchronous or asynchronous communication

• Automatic or explicit buffering

**1- Direct or indirect communication (Naming)**

|  |  |  |
| --- | --- | --- |
| Direct: Each process that wants to communicate must explicitly name the recipient or sender of the communication.(symmetry in addressing) | *Direct: asymmetry* in addressing. Here, only the sender names the recipient; the recipient is not required to name the sender. | Indirect: the messages are sent to and received from  *mailboxes*, or *ports*  *A mailbox*: like an object into which messages can be placed by processes and from which messages can be removed. |
| the send() and receive() primitives are defined as:  • Send (P, message)—Send a message to process P.  • receive(Q, message)—Receive a message from process Q. | the send() and receive() primitives are defined as follows:  • Send (P, message)—Send a message to process P.  • receive (id, message)—Receive a message from any process. The variable **id** is set to the name of the process with which communication has taken place. | send() and receive() primitives are defined as follows:  • send(A, message)—Send a message to mailbox A.  • receive(A, message)—Receive a message from mailbox A. |
| * Each mailbox has a unique ID. Example, POSIX message queues use an integer value to identify a mailbox. * A process can communicate with another process via a number of different mailboxes, but two processes can communicate only if they have a **shared** mailbox. |
| •A link is established automatically between every pair of processes that want to communicate. The processes need to know only each other’s identity to communicate.  • A link is associated with exactly two processes.  • Between each pair of processes, there exists exactly one link. |  | • A **link** is established between a pair of processes only if both members of the pair have a shared mailbox.  • A link may be associated with more than two processes.  • Between each pair of communicating processes, a number of different links may exist, with each link corresponding to one mailbox. |
| The disadvantage in both of these schemes (symmetric and asymmetric)is the limited modularity of the resulting process definitions (Changing the identifier of a process may necessitate examining all other process definitions). | |  |

**2-Synchronization**

Communication between processes takes place through calls to send() and receive() primitives.

There are different design options for implementing each primitive:

• **Blocking send**.

• **Nonblocking send**.

• **Blocking receive**.

• **Nonblocking receive**.

|  |  |  |
| --- | --- | --- |
|  | Sending process | Receiving process |
| Blocking(synchronous) | Blocked | Waiting the massage to recive |
| Waiting the message to be available | blocked |
| rendezvous | blocked | Blocked(both) |
| Nonblocking (asynchronous). | Sends the message and waits |  |
|  | Retrieves either a valid or null message |

**3-Buffering**

Messages exchanged by communicating processes reside in a temporary queue whether communication is direct or indirect.

Such queues can be implemented in three ways:

|  |  |  |  |
| --- | --- | --- | --- |
|  | Zero capacity. | Bounded capacity. | Unbounded capacity. |
| Queue length | queue has a maximum length of zero | queue has finite length *n;* at most *n* messages can reside in it | queue’s length is infinite |
| link | The link cannot have any messages waiting in it. | A: Queue is not full: any new message is placed in it when arrived.  B:the link is full: | Any number of messages can wait in the queue. |
| sender | The sender must block until the recipient receives the message. | A:sender can continue execution without waiting  B: the sender must block until space is available in the queue. | The sender never blocks |

3.5 Examples of IPC Systems

**3.5.1 An Example: POSIX Shared Memory**

Here, we explore the POSIX API for shared memory.



**POSIX shared memory is**

* **Organized** using memory-mapped files, which associate the region of shared memory with a file.
* A process must first create a shared-memory object using the shm\_open() system call, as follows:

shm\_fd = shm\_open (name, **0\_CREAT | 0\_RDRW**, **0666**);

* The first parameter **specifies the name of the shared-memory object**.
* The Subsequent parameters **specify that the shared-memory object is to be created**

**\*if it does not yet exist (O\_CREAT) and that the object is open for reading and writing (O\_RDRW).**

* The last parameter **establishes the directory permissions of the shared-memory object.**
* A successful call to **shm\_open ()** **returns an integer file descriptor** for the shared-memory object.
* Once the object is established, the **ftruncate ()** function is used to **configure the size of the object in bytes.** The call  **ftruncate (shm\_fd, 4096);** **sets the size of the object to 4,096 bytes.**
* Finally**, the mmap ()** function **establishes a memory-mapped file containing** the shared-memory object.
* It also returns a pointer to the memory-mapped file that is used for accessing the shared-memory object.

**3.5.2 An Example: Mach**

* Most communication in Mach including all intertask information-is carried out by **messages**.
* Messages are sent to and received from mailboxes, called **ports** in Mach.
* Even system calls are made by messages. When a task is created, two special mailboxes-the Kernel mailbox and the Notify mailbox-are also created.
* Kernel mailbox to communicate with the task and sends notification of event occurrences to the Notify port.
* Only three system calls are needed for message transfer.
* **1- The** **msg\_send ()** call sends a message to a mailbox.

**2-**A message is received via **msg\_receive ().**

**3-Remote procedure calls (RPCs)** are executed via **msg\_rpc (),** which sends a message and waits for exactly one return message from the sender.

* The **port\_allocate ()** system **call creates a new mailbox** and **allocates space for its queue of messages.**
* **For more information and windows example look at textbook p131**

**3.6 Communication in Client-Server Systems**

Shared memory and message passing. These techniques can be used for communication in client-server systems. In this section, we explore three other strategies for communication in client-server systems: sockets, remote procedure calls (RPCs), and pipes.

**3.6.1 Sockets**

* A **socket** is defined as **an endpoint for communication.**
* A pair of processes communicating over a network employs a pair of sockets.
* A **socket** is identified by an IP address concatenated) متسلسلة with a port number.
* Sockets use a **client-server** architecture.
* Servers implementing specific services (such as telnet, FTP, and HTIP) listen to well-known ports (a telnet server listens to port 23; **an FTP server** listens to port 21; and a **web, or HTIP**, server listens to port 80).
* All ports below 1024 are considered ***well known***; we can use them to implement standard services.
* When a client process initiates a request, it is assigned a port by its host computer. This port has some arbitrary number greater than 1024.
* All connections must be unique.
* We will illustrate sockets using Java, as it provides a much **easier interface** to sockets and has a rich library for networking utilities.
* Java provides three different types of sockets.
* **Connection-oriented (TCP) sockets** are implemented with the **Socket** class. **Connectionless (UDP)** sockets use the **DatagramSocke**t class. Finally, the **MulticastSocket** class is a **subclass** of the **DatagramSocket** class: allows data to be sent to multiple recipients.

Our example describes a date server that uses Connection-oriented TCP sockets.

The operation allows **clients** to request the current date and time from the server.

The **server** listens to port 6013.

When a connection is received, the server returns the date and time to the client.

The server creates **a ServerSocket** that specifies that it will listen to port 6013.

The server then begins listening to the port with the accept() method.

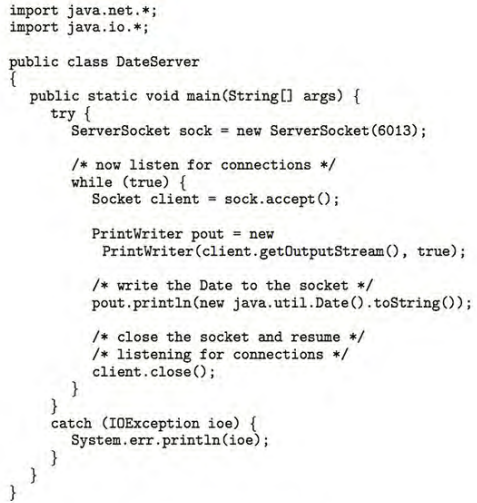
The server blocks on the **accept ()** method w When a connection request is received, accept () **returns** a socket that the server can use to communicate with the client. aiting for a client to request a connection.

The server first establishes a **PrintWriter** object that it will use to communicate with the client.

A **PrintWriter** object allows the server to write to the socket using the routine print() and **println()** methods for output.

The server process **sends the date to the client**, calling the method println ().

Once it has written the date to the socket, the server closes the socket to the client and resumes listening for more requests.



**3.6.2 Remote Procedure Calls**

The RPC was designed as a way to **abstract the procedure-call mechanism for use between systems with network connections.**

* **In contrast to IPC messages, the messages exchanged in RPC communication are well structured and are thus no longer just packets of data.**
* **Each message is**

**1-addressed to an RPC daemon listening to a port on the remote system.**

**2-contains an identifier specifying the function to execute and the parameters to pass to that function.**

**3-The function is then executed as requested, and any output is sent back to the requester in a separate message.**

* **A *port* is simply a number included at the** start **of a message packet.**
* **A system normally has one network address, many ports within that address to differentiate the many network services it supports.**
* **The semantics of RPCS allows a client to invoke a procedure on a remote host as it would invoke a procedure locally.**
* **The RPC system hides the details that allow communication to take place by providing a stub on the client side.**
* **This stub locates the port on the server and marshals the parameters.**
* **Marshalling involves packaging the parameters so that can be transmitted over a network.**
* **The stub then transmits a message to the server using message passing.**
* **On Windows systems, stub code is compiled from a specification written in the Microsoft Interface Definition Language (MIDL), which is used for defining the interfaces between client and server programs.**
* **Issues that must be dealt with:**

**1-differences in data representation on the client and server machines:**

* **Some systems (known as** big-endian**) store the most significant byte first, while other systems (known as** little-endian**) store the least significant byte first.**

**To resolve differences like this, many RPC systems define a machine-independent representation of data (known as external data representation (XDR)).**

**2-** **Another important issue involves the semantics of a call.**

* **Local procedure calls fail only under extreme circumstances, RPCs can fail, or be duplicated and executed more than once, as a result of common network errors.**
* **One way to address this problem is for the OS to ensure that messages are acted on exactly once, rather than at most once.**

**3-** **Yet another important issue concerns the communication between a server and a client.**

* **Two approaches are common. First, the binding information may be predetermined, in the form of fixed port addresses.**
* **Second, binding can be done dynamically by a rendezvous mechanism. Typically, an operating system provides a rendezvous (also called a matchmaker) daemon on a fixed RPC port.**

**3.6.3 Pipes**

A **pipe** acts as a conduit allowing two processes to communicate.

* Pipes were one of the first/simpler IPC mechanisms in early UNIX systems.

In implementing a pipe, four **issues** must be considered:

1. Does the pipe **allow bidirectional/unidirectional** communication?

2. If two-way communication is allowed, is it **half duplex** (data travel only one way at a time) or **full duplex (**in both directions at the same time)?

3. Must a **relationship** (such as parent-child) exist between the communicating processes?

4. Can the pipes **communicate over a network**, or must the communicating processes reside on the same machine?

**3.6.3.1 Ordinary Pipes**

Allow two processes to communicate in standard **producer-consumer fashion**.

* The producer **writes** to one end of the pipe (the write-end) and the consumer **reads** from the other end (the read-end).
* Ordinary pipes are unidirectional(one-way).
* If two-way communication is required, two pipes must be used, with each pipe sending data in a different direction.
* on both UNIX and Windows

One process writes the message Greetings to the pipe, while the other process reads.

* On UNIX systems, ordinary pipes are constructed using the function

\*pipe (int fd []): fd [OJ is the read-end of the pipe, and fd [1] is the write-end.

\*UNIX **treats a pipe as a special type of file**. Thus, pipes can be accessed using ordinary read() and write() system calls.

\*An ordinary pipe cannot be accessed from outside the process that created it.