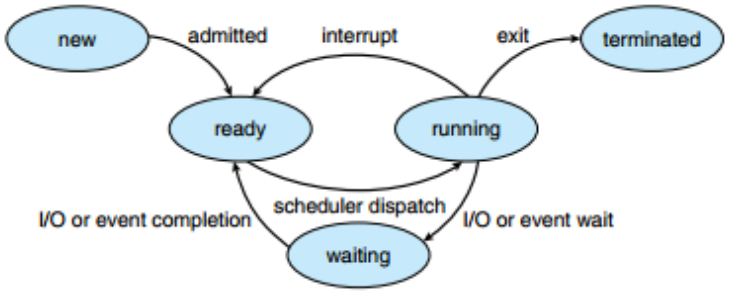
Process

1 Introduction

A process is a program in execution. Program is a passive activity whereas process is an active entity. A program becomes a process when an executable file is **loaded into memory**.

A process is more than the program code (**text section**). It also includes the current activity represented by the program counter and contents of the processor's registers. A process also includes the process **stack**, which contains temporary data (such as function parameters, return addresses, and local variables), and a **data section**, which contains global variables. A process may also include a **heap**, which is memory that is dynamically allocated during process run time.

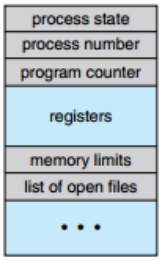


- **New**: The process is being created.
- **Running**: Instructions are being executed
- **Waiting**: It is waiting for some event to occur (such as I/O completion or reception of signal).
- **Ready**: The process is waiting to be assigned to a processor.
- **Terminated**: The process has finished execution.

Process Control Block

Each process is represented in the operating system by a PCB—also called a **task control block**.

- **Process state** indicates state of process.
- **Program counter** indicates the address of the next instruction to be executed for this process.
- **CPU registers** vary in number and type depending on architecture and include accumulators, index registers, stack pointers, and general-purpose registers, plus condition-code information.
- **CPU-scheduling information** This information includes a process priority, pointers to scheduling queues, and any other scheduling parameters.
- **Memory-management information** includes the value of the base and limit registers, the page tables, or the segment tables (depends on system organization)
- **Accounting information** includes the amount of CPU and real time used, time limits.
- **I/O status information** includes list of I/O devices allocated to process, list of open files, etc.



When an interrupt occurs, the system needs to **save** the current context of the process running on the CPU, be it in kernel or user mode, so that it can **restore** that context when its processing is done, essentially suspending the process and then resuming it. This is known as **context switch**.

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.

Process Creation

A process can create another process by using *fork()*. The creating process is called a **parent process**, and the new processes are called the **children** of that process. Each process is identified by a **process identifier**. On UNIX, we can obtain a listing of processes by using the **ps** command.

Typically, the *exec()* system call is used after a *fork()* system call to replace the process's memory space with a binary file that it executes. A parent may terminate the execution of it children if it exceeds resource usage, task assigned is not required, or if parent is exiting and OS doesn't allow **orphan** process.

In Solaris, the process at the top of the tree is the sched process, with pid of 0. The sched process creates several children processes—including pageout and fsflush. These processes are responsible for managing memory and file systems. The sched process also creates the init process, which serves as the root parent process for all user processes. inetd is responsible for networking services such as telnet and ftp; dtlogin is the process representing a user login screen. When a user logs in, dtlogin creates an X-windows session (Xsession), which in turns creates the sdt-shel process. C-shell is a user's command-line shell.

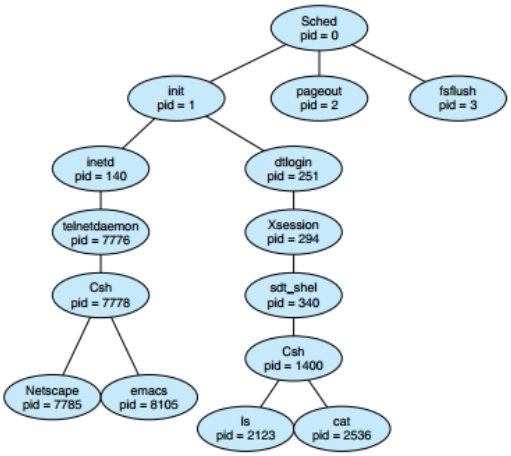
Zombie and Orphan Process

Zombie Process: A process which has finished the execution but still has entry in the process table to report to its parent process is known as a zombie process. A child process always first becomes a zombie before being removed from the process table. The parent process reads the exit status which reaps off the child process entry from the process table.

At the termination of the child, a 'SIGCHLD' signal is generated which is delivered by the kernel to the parent which reaps the status of the child from the process table. Even though, the child is terminated, there is an entry in the process table corresponding to the child where the status is stored. When parent collects the status, this entry is deleted. Thus, all the traces of the child process are removed from the system.

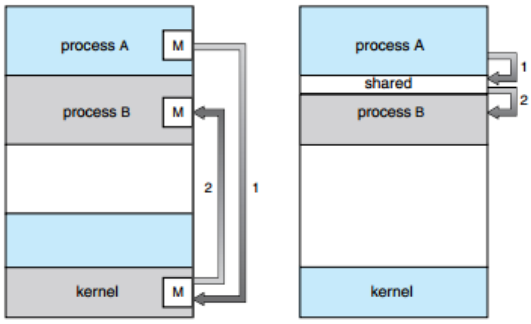
By ignoring the signal, we can avoid the creation of zombie process but we will not know about the exit status of the child.

Orphan Process: A process whose parent process no more exists i.e. either finished or terminated without waiting for its child process to terminate is called an orphan process.



2 Inter-process Communication

- **Information sharing** Several processes may wish to use same piece of information.
- **Computation speedup**. Break task into sub-tasks and parallelize them.
- **Modularity** To construct a modular system dividing system functions into separate processes.
- **Convenience**. Allows user to run multiple linked processes.



Shared-Memory Systems

In the shared-memory model, a region of memory that is shared by cooperating processes is established. Processes exchange information by reading and writing data to the shared region. It allows maximum speed and convenience of communication. System calls are required only to establish shared-memory regions. Once shared memory is established, all accesses are treated as routine memory accesses, and no assistance from the kernel is required.

Message-Passing Systems

They are implemented using system calls and thus require the more time-consuming task of kernel intervention. Message passing is useful for exchanging smaller amounts of data, because no conflicts need be avoided. Message passing is easier to implement than shared memory for **inter-computer communication**.

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

int main()
{
/* the identifier for the shared memory segment */
int segment_id;
/* a pointer to the shared memory segment */
char *shared_memory;
/* the size (in bytes) of the shared memory segment */
const int size = 4096;

    /* allocate a shared memory segment */
    segment_id = shmget(IPC_PRIVATE, size, S_IRUSR | S_IWUSR);

    /* attach the shared memory segment */
    shared_memory = (char *) shmat(segment_id, NULL, 0);

    /* write a message to the shared memory segment */
    sprintf(shared_memory, "Hi there!");

    /* now print out the string from shared memory */
    printf("*%s\n", shared_memory);

    /* now detach the shared memory segment */
    shmdt(shared_memory);

    /* now remove the shared memory segment */
    shmctl(segment_id, IPC_RMID, NULL);

    return 0;
}
```

Naming

A communication link must exist between processes that want to communicate. Under **direct communication**, each process that wants to communicate must explicitly name the recipient or sender of the communication. With **indirect communication**, the messages are sent to and received from **mailboxes**, or **ports**.

If the mailbox is owned by a process (part of the address space of process), then owner (only receives messages) and the user (only sends messages) can be distinguished for this mailbox. A mailbox that is owned by the OS has an existence of its own and is not attached to any particular process. When a process creates a new mailbox, it is that mailbox's owner by default.

Ownership and receiving privilege may be passed to other processes through appropriate system calls.

Synchronization

Message passing may be **blocking** or **non-blocking**, also known as synchronous and asynchronous.

- **Blocking send**. The sending process is blocked until the message is received.
- **Non-Blocking send**. The sending process sends the message and resumes operation.
- **Blocking receive**. The receiver blocks until a message is available.
- **Non-Blocking receive**. The receiver retrieves either a valid message or a null.

Buffering

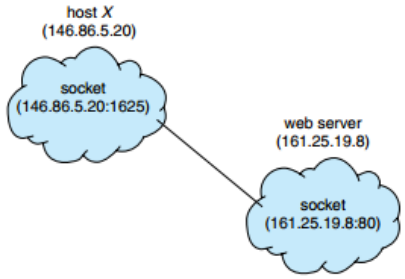
- **Zero Capacity**. The link cant have messages waiting in it. Sender must block till recipient receives.
- **Bounded Capacity**. Sender can send till the link is full, else it must block.
- **Infinite Capacity**. Sender never blocks.

The zero-capacity case referred to as a message system with **no buffering**; the other cases are referred to as systems with **automatic buffering**.

3 Communication in Client-Server Systems

Sockets

A **socket** is defined as an endpoint for communication that is identified by an IP address concatenated with a port number. Sockets mostly use a client–server architecture. The server waits for incoming client requests by listening to a specified port. Once a request is received, the server accepts a connection from the client socket to complete the connection. All ports below 1024 are considered *well known.*



When a client process initiates a request for a connection, it is assigned a port by its host computer. All connections must be **unique**. Therefore, if another process also on host X wished to establish another connection with the same Web server, it would be assigned a port number greater than 1024 and not equal to the previous one.

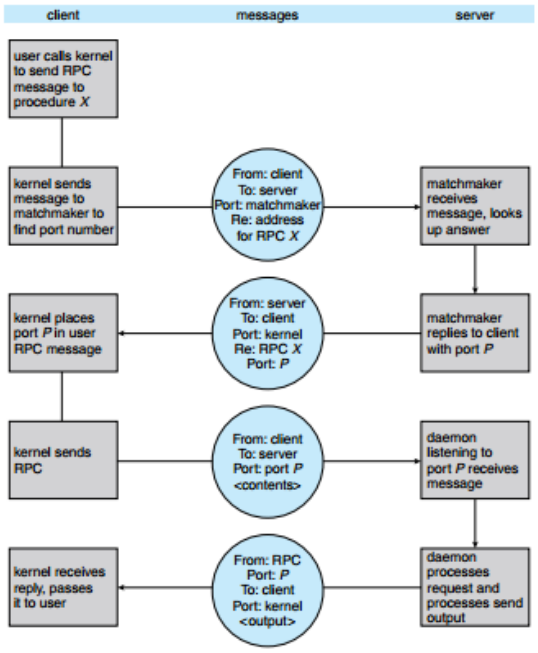
Communication using sockets is a low-level form of communication between distributed processes. One reason is that sockets allow only an **unstructured** stream of bytes to be exchanged between the communicating threads. It is the responsibility of the client or server application to impose a structure on the data.

Remote Procedure Call

It is similar in many respects to the IPC mechanism but because we are dealing with an environment in which the processes are executing on separate systems, we must use a message-based communication scheme to provide remote service. In contrast to the IPC facility, the messages exchanged in RPC communication are well structured and are thus no longer just packets of data. Each message is addressed to an RPC daemon listening to a port on the remote system, and each contains an identifier of the function to execute and the parameters to pass to that function. The function is then executed as requested, and any output is sent back to the requester in a separate message.

The semantics of RPCs allow 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. Typically, a separate stub exists for each separate remote procedure. When the client invokes a remote procedure, the RPC system calls the appropriate stub, passing it the parameters provided to the remote procedure. This stub locates the port on the server and marshals the parameters. **Parameter marshalling** involves packaging the parameters into a form that can be transmitted over a network. The stub then transmits a message to the server using message passing. A similar stub on the server side receives this message and invokes the procedure on the server.

To resolve differences like data representation on client and server, many RPC systems define a machine-independent representation of data such as **external data representation (XDR)**.



RPCs can fail and the OS must ensure that it has been executed exactly once. "At most once" can be implemented by attaching a timestamp to each message. The server must keep a history of all the timestamps of messages it has already processed. Incoming messages that have a timestamp already in the history are ignored. The client can then send a message one or more times and be assured that it only executes once. For "exactly once," we must also acknowledge to the client that the RPC call was received and executed. The client must resend each RPC call periodically until it receives the ACK for that call.