

Chapter 3 – Processes

Contemporary computer systems allow multiple programs to be loaded into memory and executed concurrently. A process is a program in execution, it is the unit of work in a modern computing system. A system therefore consists of a collection of processes – some executing user code, others executing OS code. All these processes can execute concurrently with the CPU(s) multiplexed among them.

In this chapter we discuss what **processes** are, how they are represented in an OS and how they work.

Contents

3.1 Process Concept

3.1.1 The Process

3.1.2 Process State

3.1.3 Process Control Block

3.1.4 Threads

3.2 Process Scheduling

3.2.1 Scheduling Queues

3.2.2 CPU Scheduling

3.2.3 Context Switch

3.3 Operations on Processes

3.3.1 Process Creation

3.3.2 Process Termination

3.4 Interprocess Communication (IPC)

3.5 IPC in Shared-Memory Systems

3.6 IPC in Message Passing Systems

3.7 Examples of IPC Systems

3.8 Communication in Client-Server Systems

3.8.1 Sockets

3.8.2 Remote Procedure Calls

Chapter Objectives

- Identify the separate components of a process and illustrate how they are represented and scheduled in an operating system.
- Describe how processes are created and terminated in an operating system, including developing programs using the appropriate system calls that perform these operations.
- Describe and contrast interprocess communication using shared memory and message passing.
- Become Familiar with the programs that uses pipes and POSIX shared memory to perform interprocess communication.
- Describe client-server communication using sockets and remote procedure calls.

3.1 Process Concept

3.1.1 The Process

A **process** is a program in execution. A program is a **passive** entity, such as a file containing a list of instructions (often called an executable file) stored on the disk. 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. Several processes can be created from one program, for example when multiple users executing the same program, these are all different processes.

Process execution must progress in sequential fashion, that is, the instructions within it are executed in their order. The status of the current activity of a process is represented by the value of the program counter and the contents of a processor's registers. The memory layout of a process is typically divided into multiple **sections**. These sections include:

- **Text section** – the program code
- **Data section** – global variables
- **Heap section** – memory that is dynamically allocated during program run time
- **Stack section** – temporary data storage when invoking functions (such as function parameters, return addresses, and local variables)

The sizes of the text and data sections are fixed, as these do not change during program run time. However the stack and heap sections can shrink and grow dynamically during program execution.

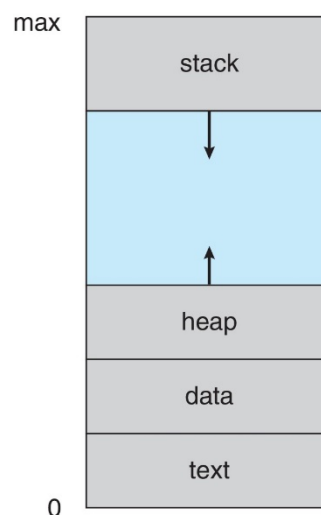


Figure 1 Layout of a process in memory

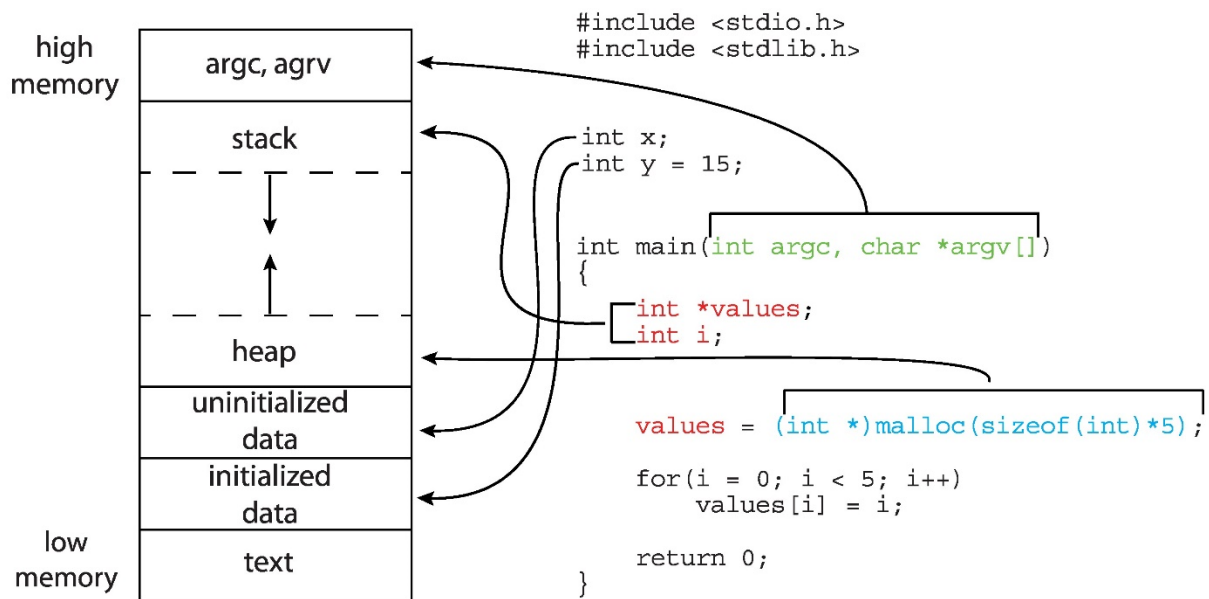


Figure 2 Memory layout of a C program

3.1.2 Process State

As a process executes, it changes state. The state of a process is defined in part 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 I/O completion).
- **Ready**: The process is waiting to be assigned to a processor.
- **Terminated**: The process has finished execution.

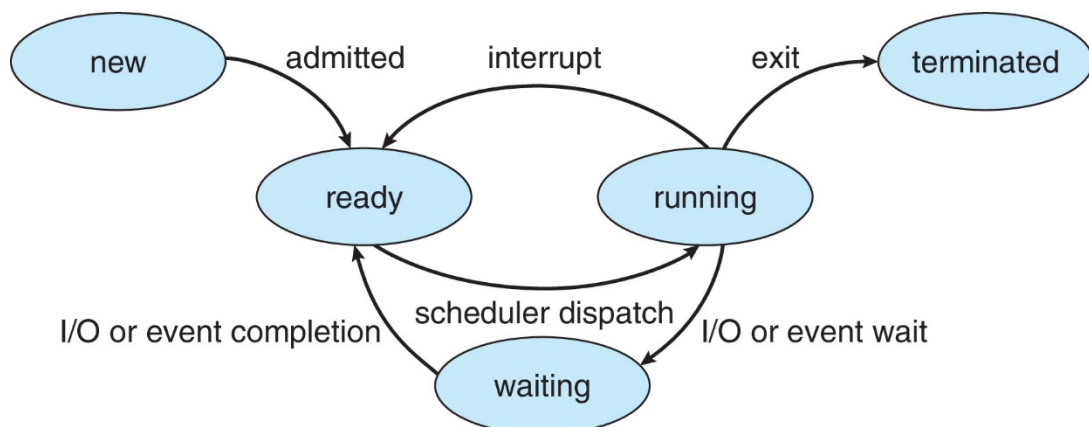


Figure 3 State transition diagram of a process.

Only **one** process can be **running** on any processor core at any instant while many other processes may be **ready** and **waiting**.

3.1.3 Process Control Block

Each process is represented in the operating system by a **process control block (PCB)**, also called a **task control block**. It contains many pieces of information associated with a specific process including these:

- **Process state** – The state may be new, ready, running, waiting, halted etc.
- **Program counter** – The counter indicates the address of the next instruction to be executed for this process.
- **CPU registers** – The registers include accumulators, index registers, stack pointers, general purpose registers etc. Along with the counter, the contents of these registers must be saved when an interrupt occurs, to allow the process to be resumed properly.
- **CPU scheduling information** – This information includes process priorities, pointers to scheduling queues and any other scheduling parameters.
- **Memory-management information** – This includes information of memory allocated to the process, such as the value of the base and limit registers and the page tables, or the segment tables etc.
- **Accounting information** – This includes the amount of CPU and clock time elapsed since start, time limits, account numbers, process numbers and so on.
- **I/O status information** – This includes the list of I/O devices allocated to process, list of open files etc.

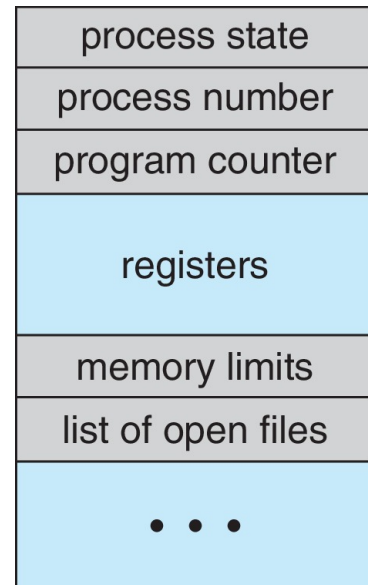


Figure 4 Process control block (PCB)

In brief, the PCB serves as the repository for all the data needed by a process, as well as accounting data.

3.1.4 Threads

The process model discussed so far implies that 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 have extended this concept to allow a process to have multiple threads of execution and thus to perform more than one task at a time. This feature is especially beneficial on multicore system where multiple threads can run in parallel. On systems that support threads, the PCB is extended to include information of each thread, some changes throughout the system are also needed.

3.2 Process Scheduling

The objective of multiprogramming is to have some process running at all times so as to maximize CPU utilization. The objective of time sharing is to switch a CPU core among processes frequently that users can interact with each program while it is running. To meet these objectives, the process scheduler selects an available process to execute on a core. Each CPU core can run one process at a time, where a multicore system can run multiple processes at one time. If there are more processes than cores, excess process will have to wait until a core is free.

3.2.1 Scheduling Queues

As a new process enters the system, it is initially put into a **ready queue**. It waits there until it is selected for execution on a CPU core. This queue is generally a linked list; a ready queue header contains pointers to the first PCB in the list and each PCB includes a pointer to the next PCB in the ready queue.

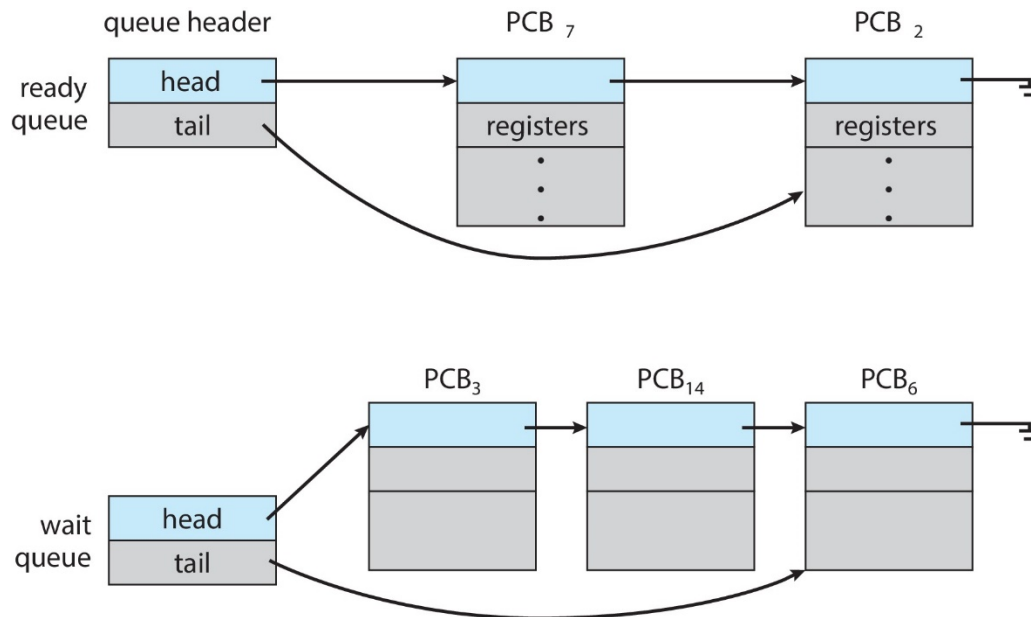


Figure 5 The ready queue and the wait queues

The system also includes other queues. When a process is given a CPU core, it executes for a while and eventually terminates, is interrupted or waits for a particular event to occur (such as the completion of I/O request). Processes that are waiting for a certain event to occur are placed in a **wait queue**.

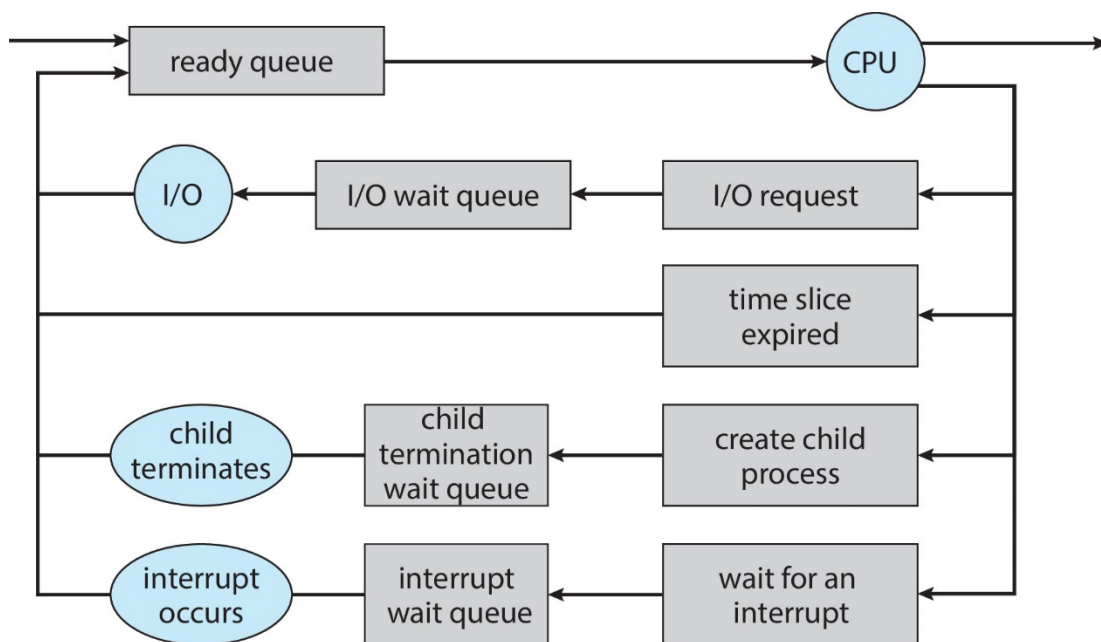


Figure 6 Representation of process scheduling

A process can be moved back and forth between ready and wait queues until it terminates. After termination it is removed from all the queues and has its PCB and resources deallocated.

3.2.2 CPU Scheduling

A process migrates among the ready queue and various wait queues throughout its lifetime. The role of the **CPU scheduler** is to select from among the processes that are in the ready queue and allocate a CPU core to one of them, in a way so that a new process gets selected frequently.

3.2.3 Context Switch

We know that interrupts cause the operating system pause its current task and to run a kernel routine. When the CPU switches from one process to another, the system needs to save the state (context) of the old process (so that it can resume later) and load the saved state for the new process. This task is known as a **context switch**. Context of a process is represented in its PCB. It includes the value of the CPU registers, the process state and memory management information etc.

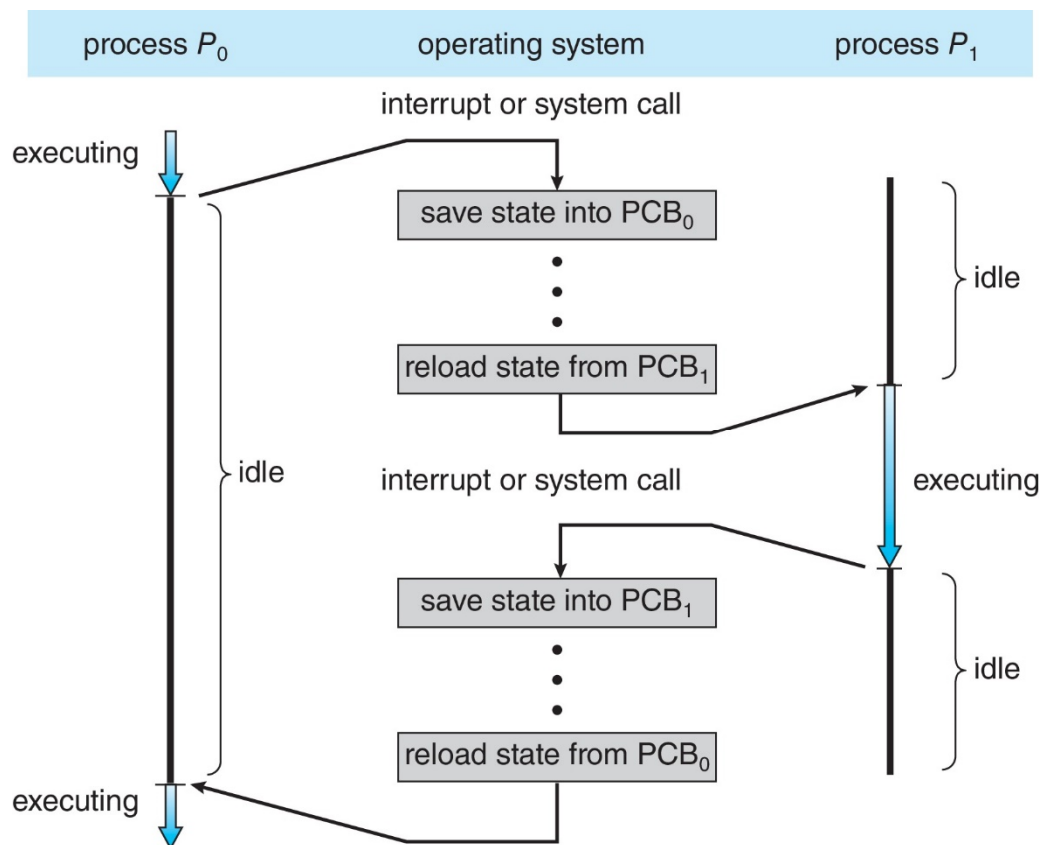


Figure 7 Context switch from process to process

Context-switch time is purely overhead as the system does no useful work while switching. The more complex the OS and the PCB, the longer time is needed for the context switch. It is also highly dependent of the hardware support.

3.3 Operations on Processes

The processes in most systems execute concurrently and they may be created and deleted dynamically. In this section we explore the mechanisms involved in process creation and deletion.

3.3.1 Process Creation

During the course of action, a process may create several new processes. The creating process is a **parent** process and the new processes are called the **children** of that process. Each of these new processes may in turn create other children, forming a tree of processes. Most OS identify processes according to a **unique process identifier (pid)** – a unique integer that can be used as an index to access various attributes of a process within the kernel.

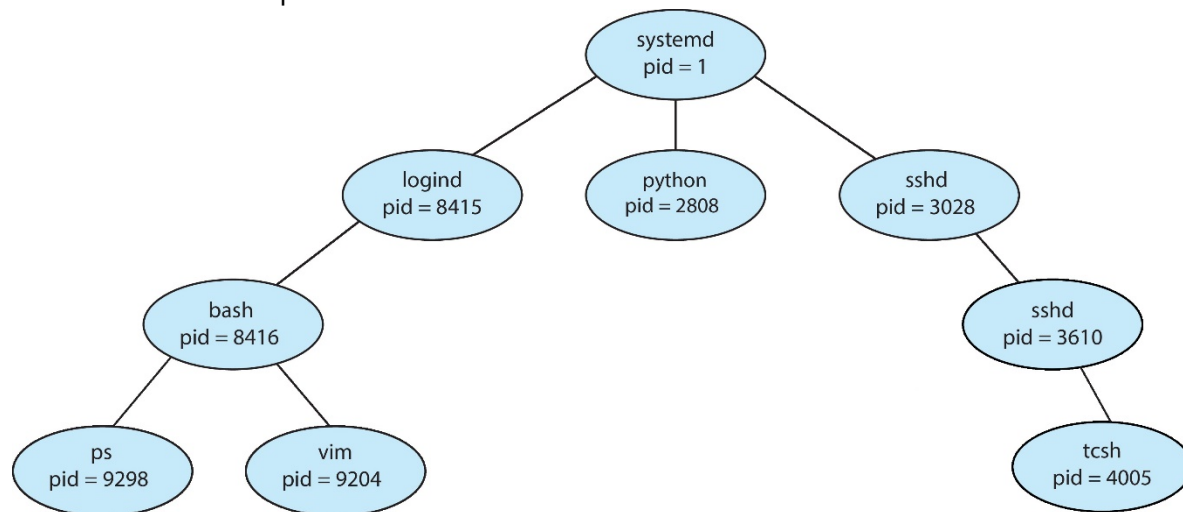


Figure 8 A tree of processes on a typical Linux system.

In general, parent and children may share all resources. Child processes may share subset of parent's resources or parent and child may share no resources at all.

When a process creates a new process, parent and children may execute concurrently or the parent may wait until the children terminate.

There are also two address-space possibilities for the new process: it may be a duplicate of the parent process (have the same program and data) or it may have a new program loaded into it.

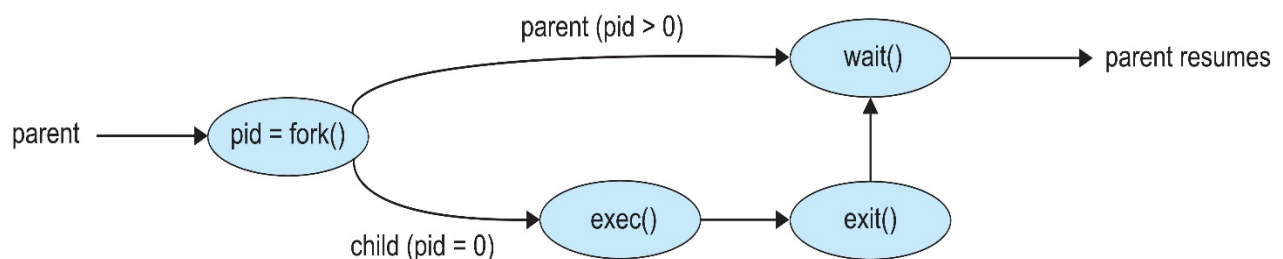


Figure 9 Process creation using the `fork()` system call

```

#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");
    }

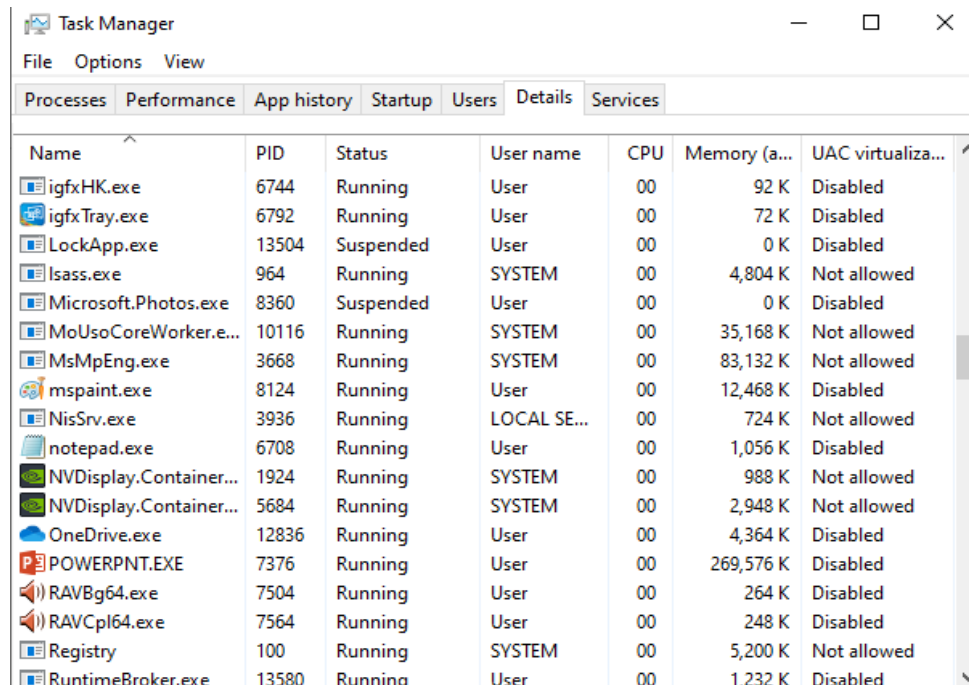
    return 0;
}

```

Figure 10 Process creation using the UNIX `fork()` system call

3.3.2 Process Termination

A process terminates when it finishes executing its final statement and then asks the operating system to delete it using the `exit()` system call. At that point the process (child) may return a status value to its waiting parent processes via the `wait()` system call. All the resources of the process including physical and virtual memory, open files and I/O buffers are deallocated and reclaimed by the operating system. Some operating systems do not allow the child process to exist if its parent has terminated. If a process terminates, then all its children must also be terminated.



The image shows a screenshot of the Windows Task Manager application. The 'Processes' tab is selected, displaying a list of running processes. The columns shown are Name, PID, Status, User name, CPU, Memory (a...), and UAC virtualiza... (partially visible). The list includes various system and user processes, such as igfxHK.exe, igfxTray.exe, LockApp.exe, lsass.exe, Microsoft.Photos.exe, MoUsCoreWorker.e..., MsMpEng.exe, mspaint.exe, NisSrv.exe, notepad.exe, NVDIsplay.Container..., OneDrive.exe, POWERPNT.EXE, RAVBg64.exe, RAVCpl64.exe, Registry, and RuntimeBroker.exe.

Name	PID	Status	User name	CPU	Memory (a...)	UAC virtualiza...
igfxHK.exe	6744	Running	User	00	92 K	Disabled
igfxTray.exe	6792	Running	User	00	72 K	Disabled
LockApp.exe	13504	Suspended	User	00	0 K	Disabled
lsass.exe	964	Running	SYSTEM	00	4,804 K	Not allowed
Microsoft.Photos.exe	8360	Suspended	User	00	0 K	Disabled
MoUsCoreWorker.e...	10116	Running	SYSTEM	00	35,168 K	Not allowed
MsMpEng.exe	3668	Running	SYSTEM	00	83,132 K	Not allowed
mspaint.exe	8124	Running	User	00	12,468 K	Disabled
NisSrv.exe	3936	Running	LOCAL SE...	00	724 K	Not allowed
notepad.exe	6708	Running	User	00	1,056 K	Disabled
NVDIsplay.Container...	1924	Running	SYSTEM	00	988 K	Not allowed
NVDIsplay.Container...	5684	Running	SYSTEM	00	2,948 K	Not allowed
OneDrive.exe	12836	Running	User	00	4,364 K	Disabled
POWERPNT.EXE	7376	Running	User	00	269,576 K	Disabled
RAVBg64.exe	7504	Running	User	00	264 K	Disabled
RAVCpl64.exe	7564	Running	User	00	248 K	Disabled
Registry	100	Running	SYSTEM	00	5,200 K	Not allowed
RuntimeBroker.exe	13580	Running	User	00	1,232 K	Disabled

Figure 11 Some Windows10 processes

3.4 Interprocess Communication (IPC)

Processes executing concurrently in the operating system may be independent or cooperating. A process is **independent** if it does not share data with any other processes executing in the system. A **cooperating** process can share data, affect or be affected by other processes executing in the system. There are several reasons of providing environment for process cooperation:

- **Information sharing** – Several applications may need the same piece of information, so it should be concurrently accessible.
- **Computation speedup** – A large task can be broken into smaller tasks to allow parallel execution, where those smaller components need to cooperate with each other for proper execution.
- **Modularity** – To construct a modular system, that is to be able to divide the system functions into separate processes or threads, cooperation among them is must.

Cooperating processes require an **interprocess communication (IPC)** mechanism that will allow them to exchange data. There are two fundamental models of IPC: **Shared memory** and **Message passing**.

3.5 IPC in Shared-Memory Systems

Interprocess communication using shared memory requires communicating process to establish a region of shared memory. Typically, the shared region resides in the address space of the process creating this. Other processes that wish to communicate using this shared memory segment must attach it to their address space. The processes can then exchange information by reading and writing data in the shared areas. The form of the data and the location are determined and controlled by these processes. Proper mechanism is needed for the processes to communicate and to synchronize their actions.

3.6 IPC in Message Passing Systems

Another way the interprocess communication can take place is by exchanging messages between the cooperating processes. It allows the processes to communicate and to synchronize their actions without sharing the same address space. The messages sent by a process can be either fixed or variable in size. If the processes P and Q wish to communicate, they need to:

- Establish a communication link between them
- Exchange messages via send/receive

The communication link between processes can be established in several ways. The link can be direct or indirect, synchronous or asynchronous, unidirectional or bi-directional and can accommodate fixed or variable length messages and so on.

3.7 Examples of IPC Systems

The POSIX API facilitates communication by shared memory, the Mach OS has message passing system. Windows IPC also uses shared memory and early UNIX systems uses pipes (a simple communication channel) as the IPC mechanism.

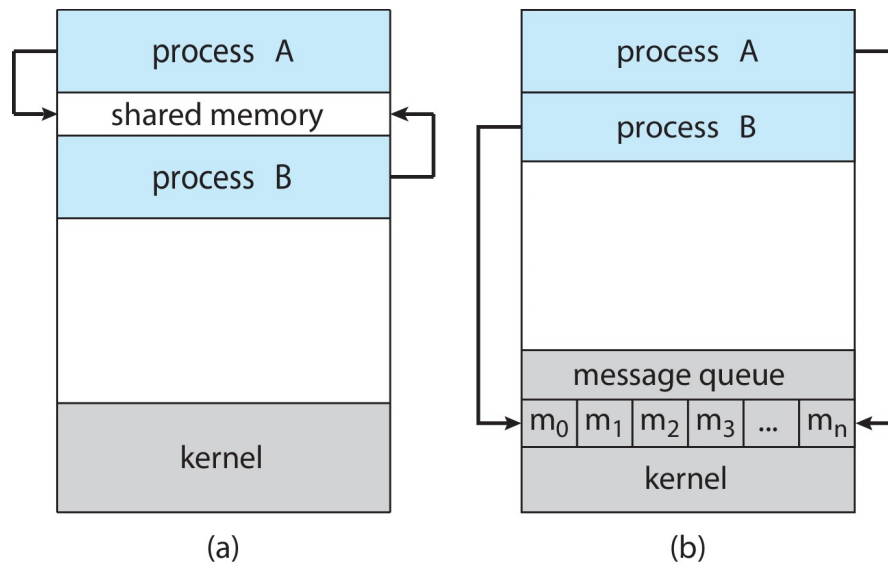


Figure 12 Communications models. (a) Shared memory, (b) Message passing.

3.8 Communication in Client-Server Systems

In this section we explore two other strategies for communication in client-server systems: **sockets** and **remote procedure calls (RPCs)**.

3.8.1 Sockets

A socket is defined as an endpoint for communication. A pair of processes communicating over a network employ a pair of sockets – one for each process. A socket is identified by an IP address concatenated with a port number. In general sockets use a client-server architecture. The server waits for the 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.

Java provides **three** different types of sockets:

- **Connection-oriented (TCP)** sockets, implemented with the *Socket* class.
- **Connectionless (UDP)**, implemented using *DatagramSocket* class
- *MulticastSocket* class– data can be sent to multiple recipients

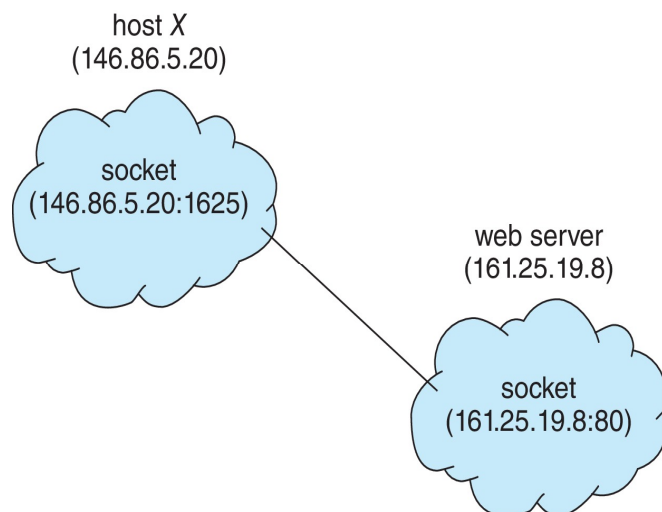


Figure 13 Communication using sockets.

```

import java.net.*;
import java.io.*;

public class DateServer
{
    public static void main(String[] args) {
        try {
            ServerSocket sock = new ServerSocket(6013);

            /* now listen for connections */
            while (true) {
                Socket client = sock.accept();

                PrintWriter pout = new
                    PrintWriter(client.getOutputStream(), true);

                /* write the Date to the socket */
                pout.println(new java.util.Date().toString());

                /* close the socket and resume */
                /* listening for connections */
                client.close();
            }
        } catch (IOException ioe) {
            System.err.println(ioe);
        }
    }
}

```

```

import java.net.*;
import java.io.*;

public class DateClient
{
    public static void main(String[] args) {
        try {
            /* make connection to server socket */
            Socket sock = new Socket("127.0.0.1", 6013);

            InputStream in = sock.getInputStream();
            BufferedReader bin = new
                BufferedReader(new InputStreamReader(in));

            /* read the date from the socket */
            String line;
            while ( (line = bin.readLine()) != null)
                System.out.println(line);

            /* close the socket connection */
            sock.close();
        } catch (IOException ioe) {
            System.err.println(ioe);
        }
    }
}

```

Figure 14 Sockets in java, Server and the Client

3.8.2 Remote Procedure Calls

Remote procedure call (RPC) abstracts the procedure calls between processes on networked systems. It is similar to the IPC mechanism discussed earlier. When a client has a message request, the RPC translates it and sends to the server. This request may be a procedure or a function call to a remote server. When the server receives the request, it sends the required response back to the client.

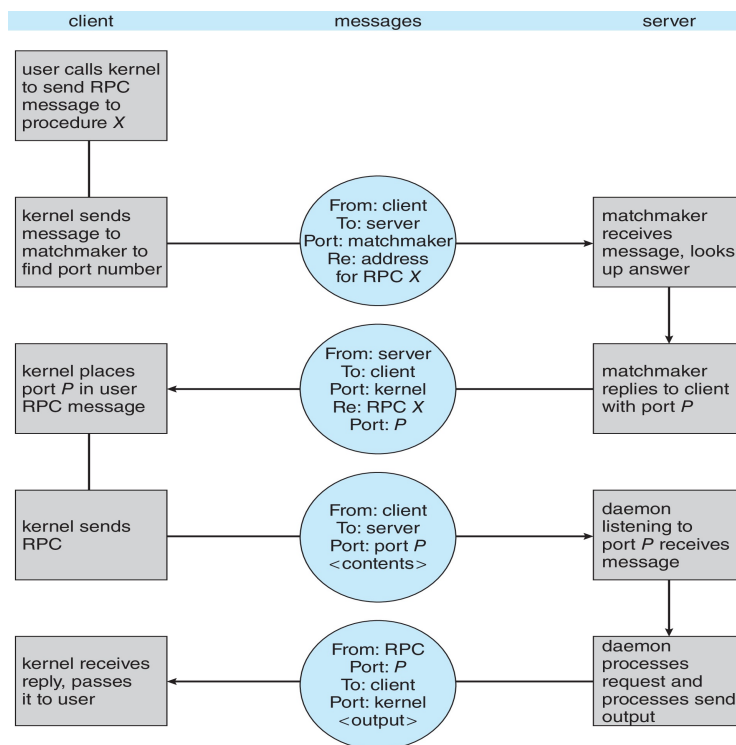


Figure 15 Execution of a remote procedure call (RPC)