OS (Process Management)

★★Process

A process can be thought of as a program in execution. A process will need certain resources-such as CPU time, memory, files, and I/O devices-to accomplish its task. These resources are allocated to the process either when it is created or while it is executing.

[We emphasize that a program by itself is not a process; a program is a *passive* entity, such as the contents of a file stored on disk, whereas a process is an *active* entity, with a program counter specifying the next instruction to execute and a set of associated resources.**]**

★★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. Each 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 or reception of a signal).
- **Ready:** The process is waiting to be assigned to a processor.
- **Terminated:** The process has finished execution.

These state names are arbitrary, and they vary across operating systems. The states that they represent are found on all systems, however. Only one process can be *running* on any processor at any instant, although many processes may be *ready* and *waiting.* The state diagram corresponding to these states is presented in Figure.

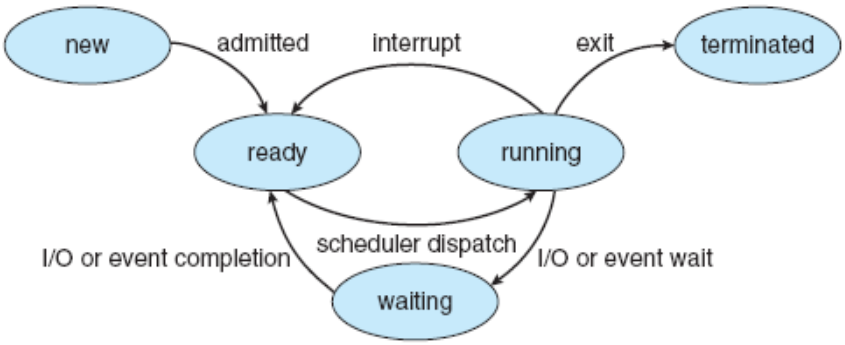


Figure : Diagram of process state.

★★Process Control Block

Each process is represented in the operating system by a **process control block** (PCB)-also called a task control block. A PCB is shown in Figure 4.2. It contains many pieces of information associated with a specific process, including these:

- **Process state:** The state may be new, ready, running, waiting, halted, and So on.
- **Program counter:** The counter indicates the address of the next instruction to be executed for this process.
- **CPU registers:** The registers vary in number and type, depending on the computer architecture. They include accumulators, index registers, stack pointers, and general-purpose registers, plus any condition-code information.
- **CPU-scheduling information:** This information includes a process priority, pointers to scheduling queues, and any other scheduling parameters.
- **Memory-management information:** This information may include such information as the value of the base and limit registers, the page tables, or the segment tables, depending on the memory system used by the operating system.
- **Accounting information:** This information includes the amount of CPU and real time used, time limits, account numbers, job or process numbers, and so on.
- **I/O status information:** The information includes the list of I/O devices allocated to this process, a list of open files, and so on.

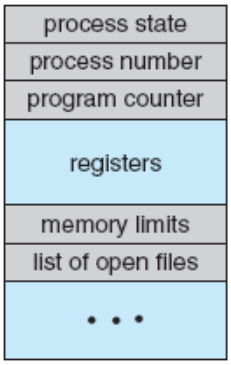


Figure 4.2: Process control block (PCB).

The PCB simply serves as the repository for any information that may vary from process to process.

★★Scheduling [Job and Process scheduling]

Scheduling refers to a set of policies and mechanism into operating system that govern the order in which the work to be done by a computer system. This is **important** because a CPU can only handle one task at a time, but there are usually many tasks that need to be processed. The following are different purposes of a CPU scheduling process.

- Maximize the CPU utilization
- Minimize the response and waiting time of the process.

★★Scheduling Queues

Job queue: As processes enter the system, they are put into a **job queue.** This queue consists of all processes in the system.

Ready queue: The processes that are residing in main memory and are ready and waiting to execute are kept on a list called the **ready queue.** This queue is generally stored as a linked list. A ready-queue header contains pointers to the first and final PCBs in the list.

Device queue: The list of processes waiting for a particular I/O device is called a device queue.

A common representation of process scheduling is a **queuing diagram,** such as that in Figure 4.5. Each rectangular box represents a queue. Two types of queues are present: the ready queue and a set of device queues. The circles represent the resources that serve the queues, and the arrows indicate the flow of processes in the system.

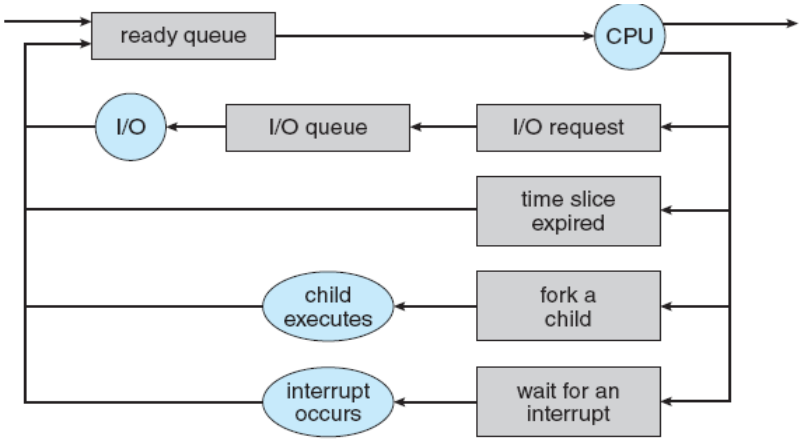


Figure 4.5 Queueing-diagram representation of process scheduling.

A new process is initially put in the ready queue. It waits in the ready queue until it is selected for the CPU.

Once the process is assigned to 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 subprocess and wait for its termination.
- The process could be removed forcibly from the CPU, as a result of an interrupt, and be put back in the ready queue.

After running on the CPU, it waits for an I/O operation by moving to an I/O queue. Eventually, it is served by the I/O device and returns to the ready queue. A process continue this CPU-I/O cycle until it finishes; then it exits from the system.

★★Types of Schedulers

In general, there are three different types of schedulers, which may coexist in a complex operating system : long-term, medium-term, and short-term schedulers.

Fig. 3.4 shows the possible traversal paths of jobs and programs through the components and queues, depicted by rectangles, of a computer system. The primary places of action of the three types of schedulers are marked with down-arrows. As shown in Fig. 3.4, a submitted batch job joins the batch queue while waiting to be processed by the long-term scheduler. Once scheduled for execution, processes spawned by the job enter the ready queue to await processor allocation by the short-term scheduler. After becoming suspended, the running process may be removed from memory and swapped out to secondary storage. Such processes are subsequently admitted to main memory by the medium-term scheduler in order to be considered for execution by the short-term scheduler.

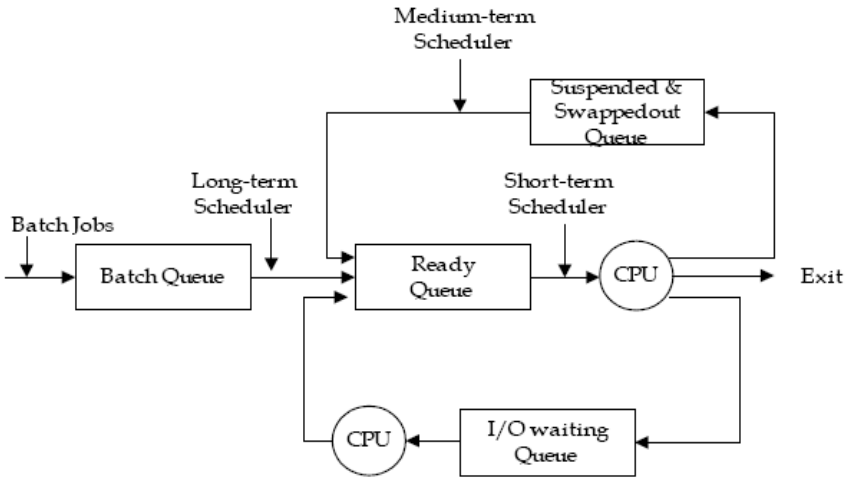


Fig. 3.4 : Schedulers.

Long-term scheduler

- The long-term scheduler (or job scheduler) works with the batch queue and determines which jobs are admitted to the system for processing.
- The primary objective of the long-term scheduler is to provide a balanced mix of jobs, such as CPU bound and I/O bound, to the short-term scheduler.

Short-term scheduler

The short-term scheduler (or CPU scheduler) selects from among the jobs in memory which are ready to execute and allocates the CPU to one of them.

Medium-term scheduler

- Medium-term scheduler can sometimes be advantageous to remove processes from memory and thus reduce the degree of multiprogramming.
- At some later time, the process can be reintroduced into memory and continued where it left off. This scheme is often called swapping.

Dispatcher

The dispatcher is the module that actually gives control of the CPU to the process selected by the short-term scheduler. This function involves –

- loading the registers of the process
- switching to user mode
- And jumping to the proper location in the user program to restart it.

★★Distinction between long-term and short-term schedulers

The primary distinction between these two schedulers is the frequency of their execution. The short-term scheduler must select a new process for the CPU quite often. A process may execute only a few milliseconds before waiting for an I/O request. Often the short-term scheduler executes at least once every 10 milliseconds. Because of the short duration between executions, the short-term scheduler must be very fast.

The long-term scheduler, on the other hand, executes much less frequently. It may be minutes between the arrival of new jobs in the system. The long-term scheduler controls the degree of multiprogramming (the number of processes in memory).

★★Context Switch

- Switching the CPU to another process requires saving the state of the old process and loading the saved state for the new process. This task is known as a context switch.

- The context of a process is represented in the PCB of a process; it includes the value of the CPU registers, the process state and memory-management information. When a context switch occurs, the kernel saves the context of the old process in its PCB and loads the saved context of the new process scheduled to run.

- Context-switch time is pure overhead, because the system does no useful work while switching.

- Its speed varies from machine to machine, depending on the memory speed, the number of registers that must be copied, and the existence of special instructions.

★★Operations on Processes

The processes in the system can execute concurrently, and they must be created and deleted dynamically. Thus, the operating system must provide a mechanism (or facility) for process creation and termination.

☆☆Process Creation

A process may create several new processes, via a **create-process** system call, during the course of execution. The creating process is called a **parent** process, whereas 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 (Figure 4.7).

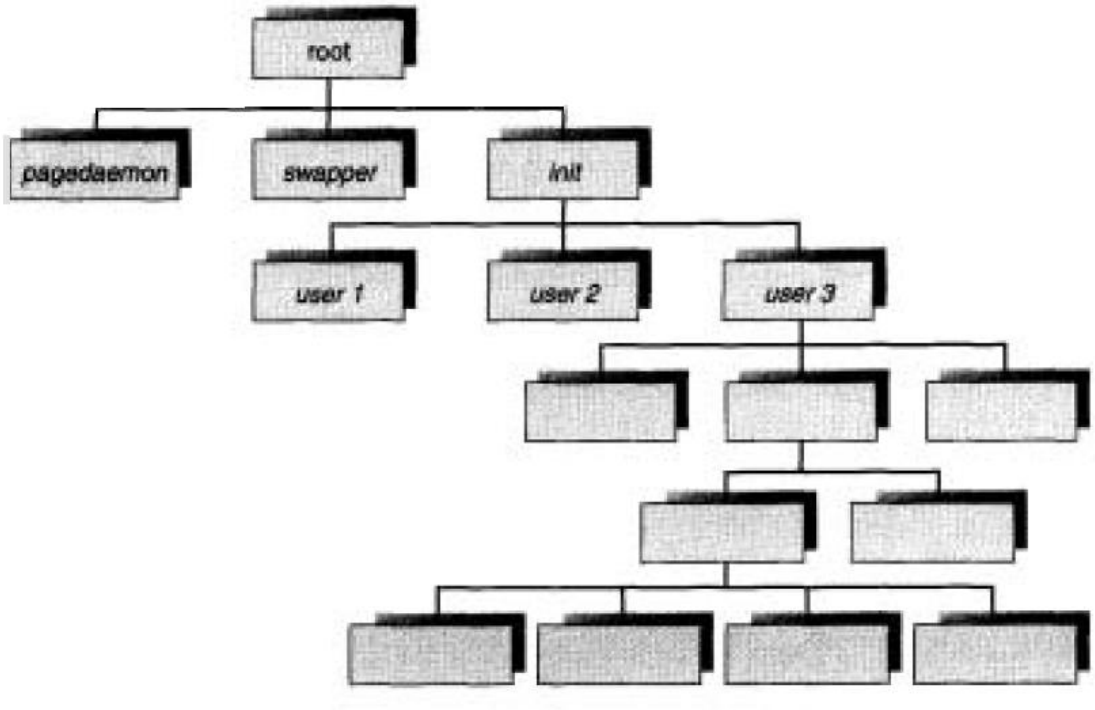


Fig 4.7: A tree of processes in a typical UNIX System

When a process creates a new process, two possibilities exist in terms of execution:

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 possibilities in terms of the address space of the new process:

1. The child process is a duplicate of the parent process.

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

☆☆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. At that point, the process may return data (output) to its parent process (via the **wait** system call). All the resources of the process-including physical and virtual memory, open files, and 1/0 buffers-are deallocated by the operating system.

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. This requires the parent to 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.

★★Cooperating Processes

The concurrent processes executing in the operating system may be either independent processes or cooperating processes. A process is independent if it cannot affect or be affected by the other processes executing in the system. On the other hand, a process is cooperating if it can affect or be affected by the other processes executing in the system.

We may want to provide an environment that allows process cooperation for several reasons:

Information sharing: Since several users may be interested in the same piece of information (for instance, a shared file), we must provide an environment to allow concurrent access to these types of resources.

Computation speedup: If we want a particular task to run faster, we must break it into subtasks, each of which will be executing in parallel with the others.

Modularity: We may want to construct the system in a modular fashion, dividing the system functions into separate processes or threads.

Convenience: Even an individual user may have many tasks on which to work at one time. For instance, a user may be editing, printing, and compiling in parallel.

★★Interprocess Communication

IPC (Interprocess Communication) provides a mechanism to allow processes to communicate and to synchronize their actions without sharing the same address space. IPC is particularly useful in a distributed environment where the communicating processes may reside on different computers connected with a network. An example is a **chat** program used on the World Wide Web.

IPC is best provided **by a message-passing system**, and message systems can be defined in many ways.

★★Message-Passing System

- The function of a message system is to allow processes to communicate with one another without the need to resort to shared data.

- Communication among the user processes is accomplished through the passing of messages. An IPC facility provides at least the two operations: send (message) and receive (message).

- Messages sent by a process can be of either fixed or variable size. If only fixed-sized messages can be sent, the system-level implementation is straightforward. On the other hand, variable-sized messages require a more complex system-level implementation, but the programming task becomes simpler.

- If processes *P* and Q want to communicate, they must send messages to and receive messages from each other; a **communication link** must exist between them. Here are several methods for logically implementing a link and the send/receive operations:
  - Direct or indirect communication
  - Symmetric or asymmetric communication
  - Automatic or explicit buffering
  - Send by copy or send by reference
  - Fixed-sized or variable-sized messages

In following section, we look at different issues when designing message-passing systems.

☆☆**Naming**

Processes that want to communicate must have a way to refer to each other. They can use either direct or indirect communication.

(1) Direct Communication

With direct communication, each process that wants to communicate **must explicitly name the recipient or sender** of the communication. In this scheme, 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.**

A communication link in this scheme has the following properties:

- 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.
- Exactly one link exists between each pair of processes.

This scheme exhibits symmetry in addressing; that is, both the sender and the receiver processes must name the other to communicate. A variant of this scheme employs asymmetry in addressing. Only the sender names the recipient; the recipient is not required to name the sender.

(2) Indirect Communication

With indirect communication, the messages are sent to and received from **mailboxes, or ports**. A mailbox can be viewed abstractly as an object into which messages can be placed by processes and from which messages can be removed. Each mailbox has a unique identification. In this scheme, a process can communicate with some other process via a number of different mailboxes. Two processes can communicate only if they share a mailbox. The **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.**

In this scheme, a communication link has the following properties:

- 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.
- A number of different links may exist between each pair of communicating processes, with each link corresponding to one mailbox.

☆☆**Synchronization**

Communication between processes takes place by calls to send and receive primitives. There are different design options for implementing each primitive. Message passing may be either blocking or nonblocking-also known as synchronous and asynchronous.

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

Different combinations of send and receive are possible. When both the send and receive are blocking, we have a rendezvous between the sender and the receiver.

☆☆**Buffering**

Whether the communication is direct or indirect, messages exchanged by communicating processes reside in a temporary queue. Basically, such a queue can be implemented in **three ways**:

- **Zero capacity:** The queue has maximum length 0; thus, the link cannot have any messages waiting in it. In this case, the sender must block until the recipient receives the message.
- **Bounded capacity:** The queue has finite length n; thus, at most n messages can reside in it. If the queue is not full when a new message is sent, the latter is placed in the queue (either the message is copied or a pointer to the message is kept), and the sender can continue execution without waiting. The link has a finite capacity, however. If the link is full, the sender must block until **space** is available in the queue.
- **Unbounded capacity:** The queue has potentially infinite length; thus, any number of messages can wait in it. The sender never blocks.

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

☆☆**An Example: Windows 2000**

Windows 2000 provides support for multiple operating environments or subsystems, with which application programs communicate via a message-passing mechanism. The application programs can be considered to be clients of the Windows 2000 subsystem server.

The message-passing facility in Windows 2000 is called the **local procedure call (LPC)** facility. The LPC in Windows 2000 communicates between two processes that are on the same machine. Windows 2000 uses a port object to establish and maintain a connection between two processes. Every client that calls a subsystem needs a communication channel, which is provided by a port object and is never inherited. Windows 2000 uses two types of ports: connection ports and communication ports. They are really the same but are given different names according to how they are used. Connection ports are named objects, which are visible to all processes; they give applications a way to set up a communication channel. This communication works as follows:

- The client opens a handle (resource) 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.

Windows 2000 uses two types of message-passing techniques over a port that the client specifies when it establishes the channel.

- The simplest, which is used for small messages, uses the port's message queue as intermediate storage and copies the message from one process to the other. Under this method, messages of up to 256 bytes can be sent.

- If a client needs to send a larger message, it passes the message through a section object (or shared memory). The client has to decide, when it sets up the channel, whether or not it will need to send a large message. If the client determines that it does want to send large messages, it asks for a section object to be created.

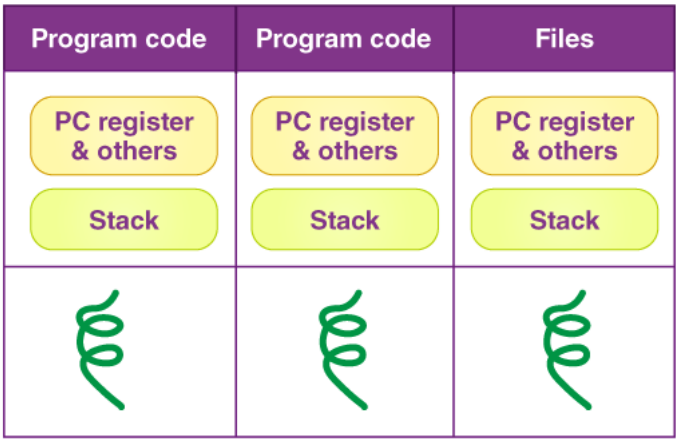
In both cases, a callback mechanism can be used when either the client or the server cannot respond immediately to a request. The callback mechanism allows them to perform asynchronous message handling.

* ***Handle:*** *In computer programming, a handle is an abstract reference to a resource that is used when application software references blocks of memory or objects that are managed by another system like a database or an operating system.*

★★ Thread in Operating Systems

A **thread** refers to a single sequential flow of activities being executed in a process; it is also known as the thread of execution or the thread of control.

Now, thread execution is possible within any OS' s process. Apart from that, a process can have several threads. A distinct program counter, a stack of activation records as well as control blocks are used by each thread of the same process. Following figure shows three threads of same process. Thread is frequently described as a light technique.



Three threads of same process

The **procedure** can be easily broken down into numerous **different threads**. Multiple tabs in a browser, **for example, can be considered threads**.

☆☆Why Do We Need Thread?

- Creating a new thread in a current process requires significantly less time than creating a new process.
- Threads can share common data without needing to communicate with each other.
- When working with threads, context switching is faster.
- Terminating a thread requires less time than terminating a process.

☆☆**Types of Threads**

1. Kernel-Level Thread

The operating system is recognized by the kernel thread. Each thread and process in the kernel-level thread has its own thread control block as well as process control block in the system. The operating system implements the kernel-level thread. The kernel is aware of all threads and controls them. The kernel-level thread provides a system call for user-space thread creation and management. Kernel threads are more complex to build than user threads. The kernel thread's context switch time is longer. The execution of the Banky thread can continue in case a kernel thread performs a blocking operation. **Solaris, for example.**

Pros

- All threads are completely aware of the kernel-level thread.
- The scheduler may decide to devote extra CPU time to threads with large numerical values.
- Applications that happen to block the frequency should use the kernel-level thread.

Cons

- All threads are managed and scheduled by the kernel thread.
- Kernel threads are more complex to build than user threads.
- Kernel-level threads are slower than user-level threads.

*[*A **kernel** is the core part of an operating system. It acts as a bridge between software applications and the hardware of a computer.]*

2. User-Level Thread

The user-level thread is ignored by the operating system. User threads are simple to implement and are done so by the user. The entire process is blocked if a user executes a user-level operation of thread blocking. The kernel-level thread is completely unaware of the user-level thread. User-level threads are managed as single-threaded processes by the kernel-level thread.

Threads in Java, POSIX, and other languages are examples.

Pros

- User threads are easier to implement than kernel threads.
- Threads at the user level can be used in operating systems that do not allow threads at the kernel level.
- It is more effective and efficient.

- Context switching takes less time than kernel threads.
- It does not necessitate any changes to the operating system.
- The representation of user-level threads is relatively straightforward. The user-level process's address space contains the register, stack, PC, and mini thread control blocks.
- Threads may be easily created, switched, and synchronised without the need for process interaction.

Cons

- Threads at the user level are not coordinated with the kernel.
- The entire operation is halted if a thread creates a page fault.

☆☆**Benefits of Threads**

- **Enhanced system throughput:** The number of jobs completed per unit time increases when the process is divided into numerous threads, and each thread is viewed as a job. As a result, the system's throughput likewise increases.
- **Effective use of a Multiprocessor system:** We can schedule multiple threads in multiple processors when we have many threads in a single process.
- **Faster context switch:** The thread context switching time is shorter than the process context switching time. The process context switch adds to the CPU's workload.
- **Responsiveness:** When a process is divided into many threads, and each of them completes its execution, then the process can be responded to as quickly as possible.
- **Communication:** Multiple-thread communication is straightforward because the threads use the same address space, while communication between two processes is limited to a few exclusive communication mechanisms.
- **Resource sharing:** Code, data, and files, for example, can be shared among all threads in a process. Note that threads cannot share the stack or register. Each thread has its own stack and register.