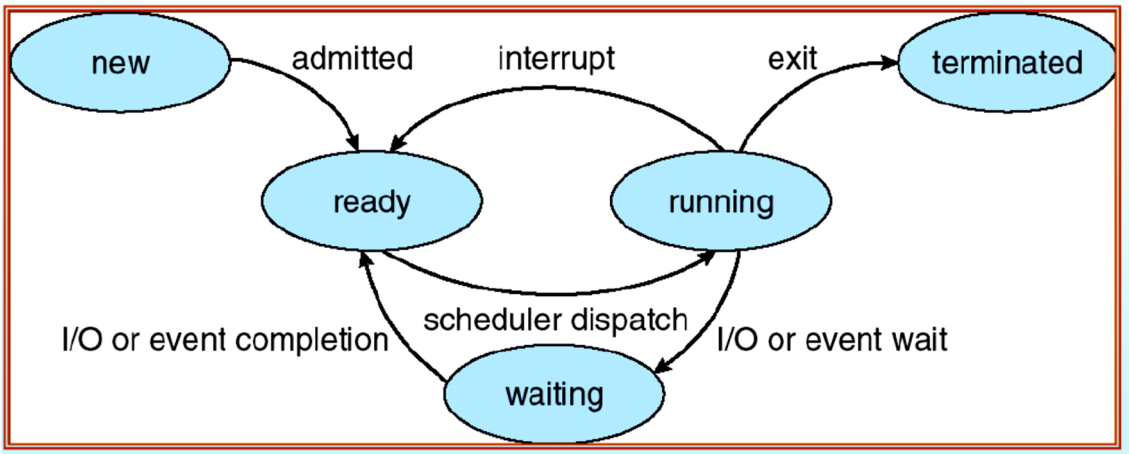
Chapter 5
Process

This chapter will discuss the following concepts:

5.1 Process Concept
5.2 Process Scheduling
5.3 Operations on Processes
5.4 Inter-process Communication



5.1 Process Concept

A question that arises in discussing operating systems involves what to call all the CPU activities. A batch system executes jobs, whereas a time-shared system has user programs, or tasks. Even on a single-user system such as Microsoft Windows, a user may be able to run several programs at one time: a word processor, a web browser, and an e-mail package. Even if the user can execute only one program at a time, the operating system may need to support its own internal programmed activities, such as memory management. In many respects, all these activities are similar, so we call all of them processes. The terms job and process are used almost interchangeably in this text.

5.1.1 The Process

Informally, as mentioned earlier, a process is a **program in execution**. A process is more than the program code, which is sometimes known as the **text section**. It also includes the current activity, as represented by the value of the **program counter** and the contents of the **processor's registers**. A process generally also includes the process **stack**, which contains temporary data, 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. The structure of a process in memory is shown in Figure 5.1.

We emphasize that a program by itself is not a process; a program is a **passive entity**, such as a file containing a list of instructions stored on disk (often called an **executable file**), whereas 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.

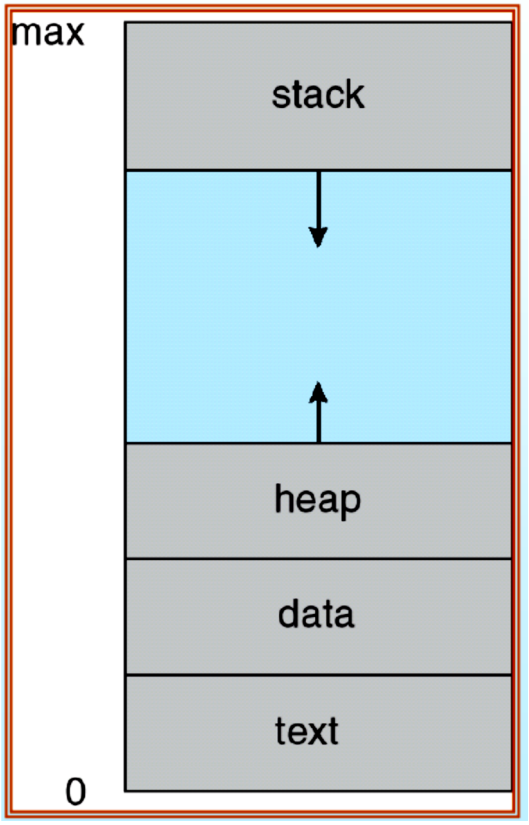


Figure 5.1 Process in memory.

5.1.2 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. 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.

The state diagram corresponding to these states is presented in Figure 5.2.

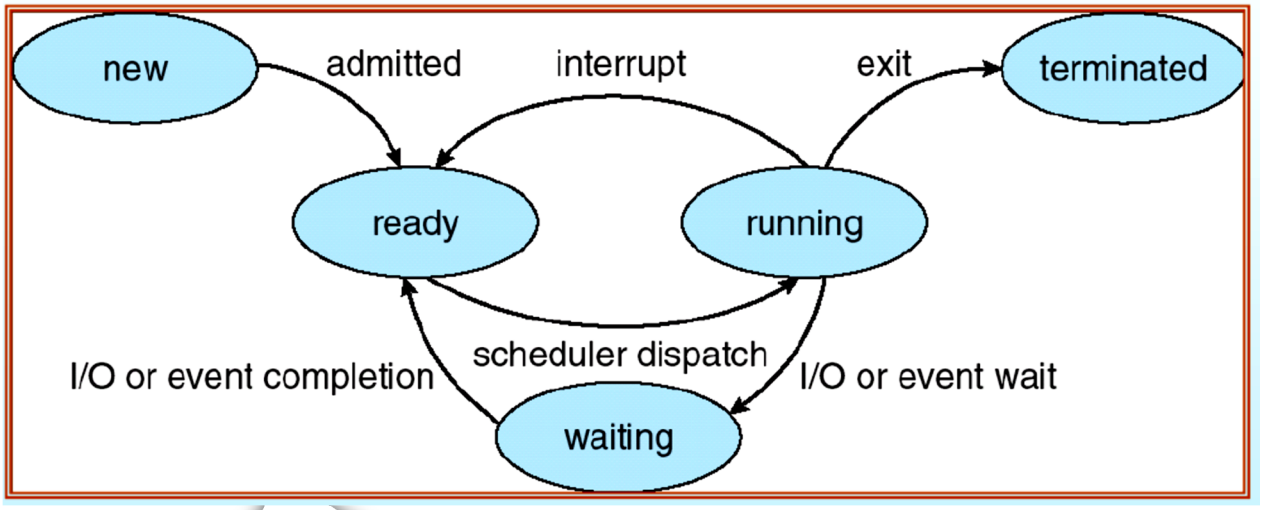


Figure 5.2 Diagram of process state.

5.1.3 Process Control Block

Each process is represented in the operating system by a **process control block** (**PCB**). A PCB is shown in Figure 5.3. 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. (Figure 5.4).
- **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:** This information includes the list of I/O devices allocated to the process, a list of open files, and so on.

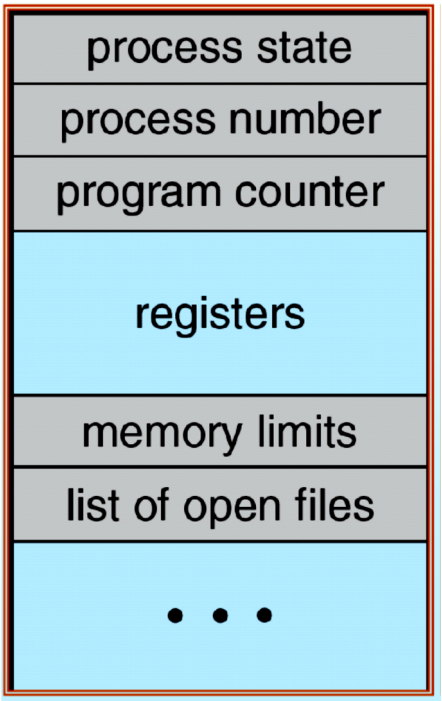


Figure 5.3 Process control block (PCB).

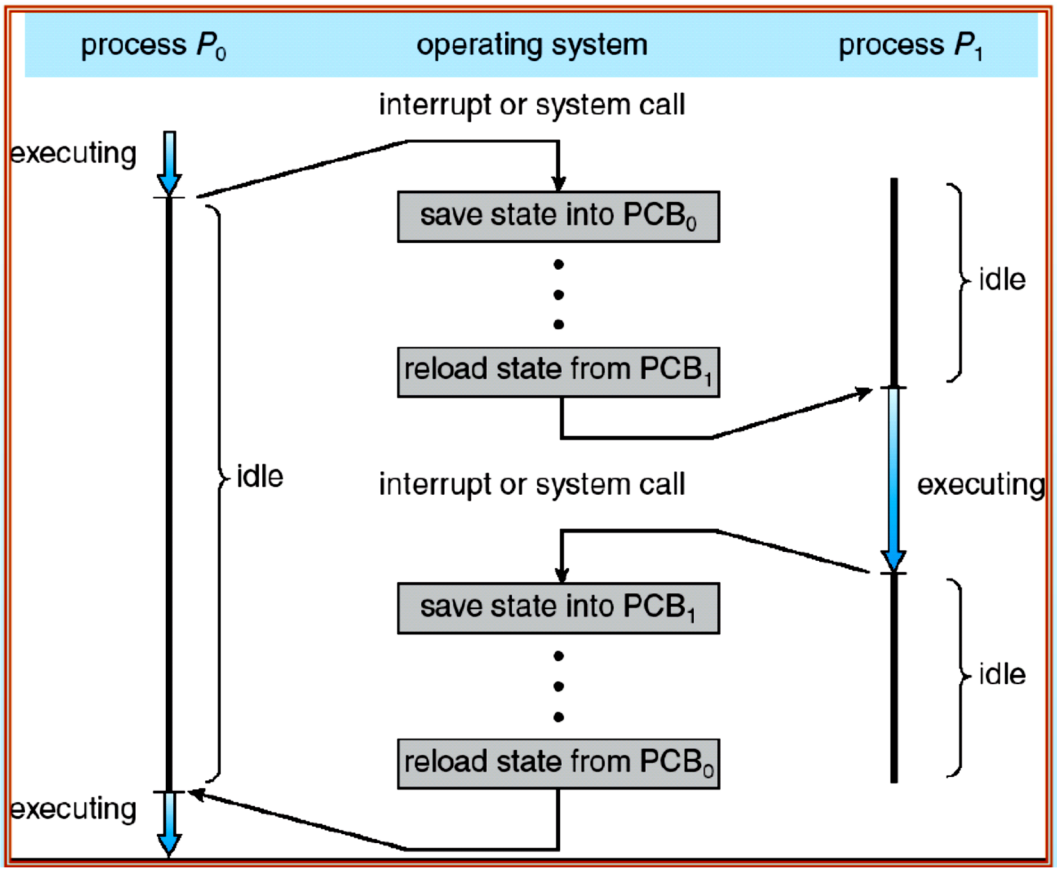


Figure 5.4 Diagram showing CPU switch from process to process.

5.1.4 Threads

A process is a program that performs a single **thread** of execution. For example, when a process is running a word-processor program, a single thread of instructions is being executed. This single thread of control allows the process to perform only one task at one time. The user cannot simultaneously type in characters and run the spell checker within the same process.

5.2 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. For a single-processor system, there will never be more than one running process. If there are more processes, the rest will have to wait until the CPU is free and can be rescheduled.

5.2.1 Scheduling Queues

As processes enter the system, they are put into a **job queue**, which consists of all processes in the system. The processes that are residing in main memory and are ready and waiting to execute are kept on a list called the **ready queue**. The system also includes other queues.

Suppose the process makes an I/O request to a shared device, such as a disk. Since there are many processes in the system, the disk may be busy with the I/O request of some other process. The process therefore may have to wait for the disk. The list of processes waiting for a particular I/O device is called a **device queue**. Each device has its own device queue (Figure 5.5).

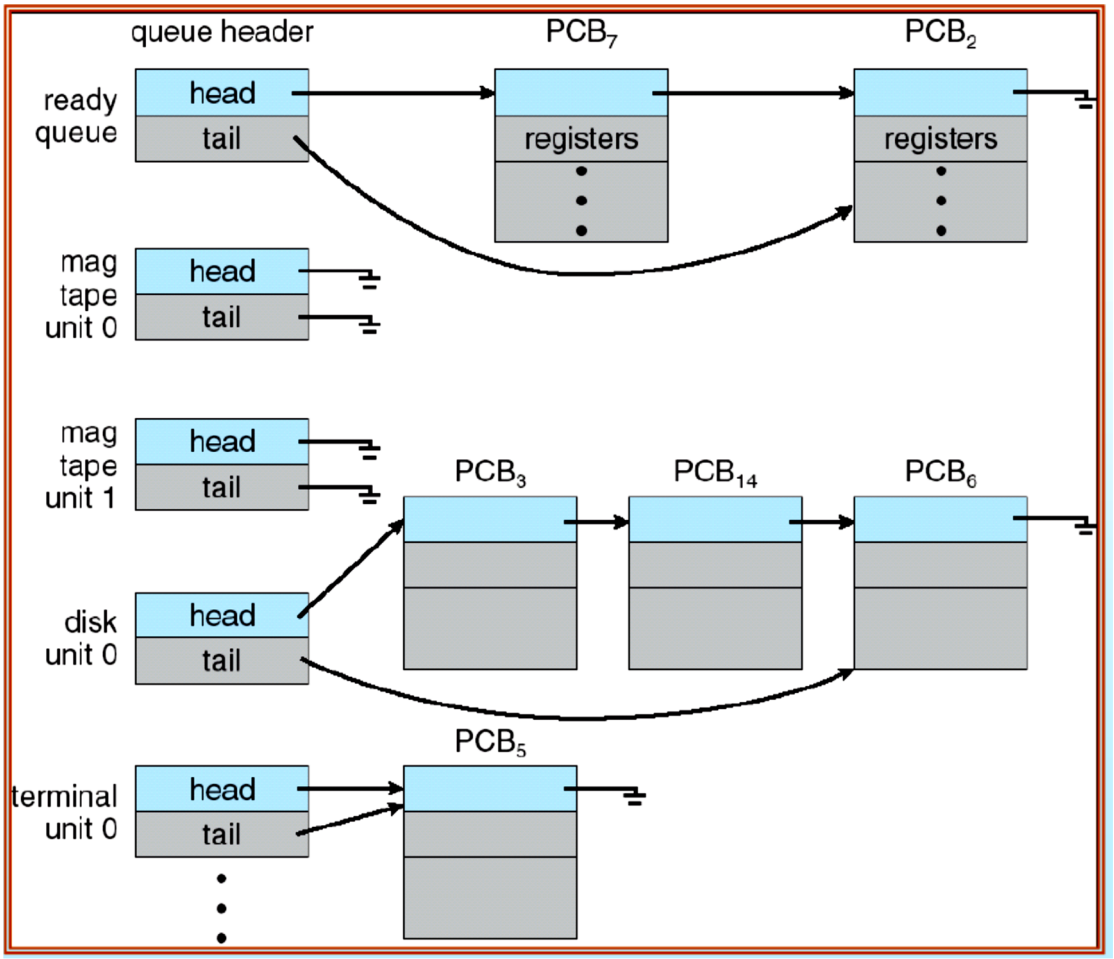


Figure 5.5 the ready queue and various I/O device queues.

A new process is initially put in the ready queue. It waits there until it is selected for execution, or is **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 sub-process and wait for the sub-process's termination.
- The process could be removed from the CPU, as a result of an interrupt, and be put back in the ready queue.

5.2.2 Schedulers

A process migrates among the various scheduling queues throughout its lifetime. The operating system must select, for scheduling purposes, processes from these queues in some fashion. The selection process is carried out by the appropriate **scheduler**.

Often, in a batch system, more processes are submitted than can be executed immediately. These processes are spooled to a mass-storage device (typically a disk), where they are kept for later execution. 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.

Some operating systems, such as time-sharing systems, may introduce an additional, intermediate level of scheduling. This is called **medium-term scheduler**. The key idea behind a medium-term scheduler is that sometimes it can be advantageous to remove processes from memory (and from active contention for the CPU) and thus reduce the degree of multiprogramming. Later, the process can be reintroduced into memory, and its execution can be continued where it left off. This scheme is called **swapping**.

5.2.3 Context Switch

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**. 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 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** (such as a single instruction to **load** or **store** all registers).

5.3 Operations on Processes

The processes in most systems can execute concurrently, and they may be created and deleted dynamically. Thus, these systems must provide a mechanism for process **creation** and **termination**.

5.3.1 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, 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.

Most operating systems identify processes according to a unique **process identifier** (or **pid**), which is typically an integer number. Figure 5.6 illustrates a typical process tree for the Solaris operating system, showing the name of each process and its **pid**. 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, fsflush and init. The init process serves as the root parent process for all user processes.

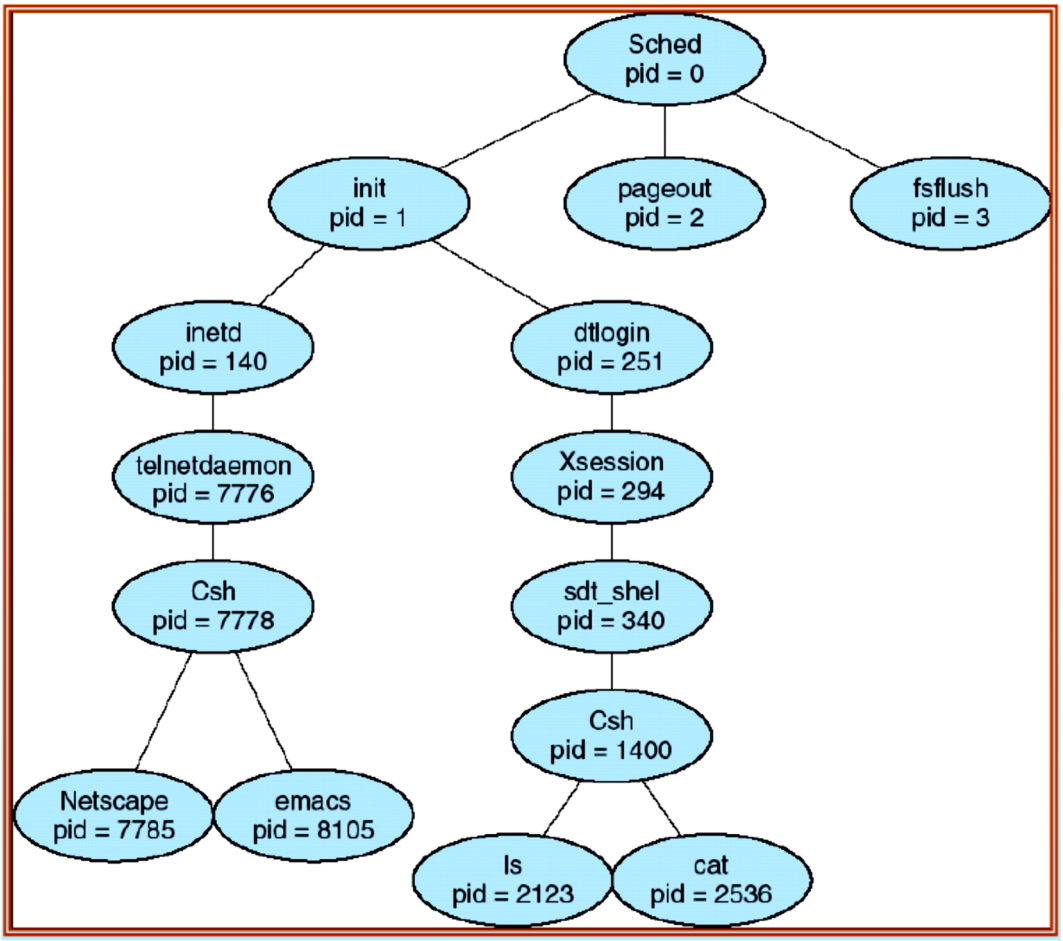


Figure 5.6 A tree of processes on a typical Solaris system.

In general, a process will need certain resources (CPU time, memory, files, and I/O devices) to accomplish its task. When a process creates a sub-process, that sub-process may be able to obtain its resources directly from the operating system, or it may be constrained to a subset of the resources of the parent process. The parent may have to partition its resources among its children, or it may be able to share some resources (such as memory or files) among several of its children. 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.

5.3.2 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. At that point, the process may return a status value (typically an integer) to its parent process. All the resources of the process—including physical and virtual memory, open files and I/O buffers—are de-allocated 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.
- 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.

5.4 Inter-process Communication

Processes executing concurrently 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. Any process that does not share data with any other process is independent.

A process is **cooperating** if it can affect or be affected by the other processes executing in the system. Clearly, any process that shares data with other processes is a cooperating process. There are several reasons for providing an environment that allows process cooperation:

- **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 such information.
- **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 work on many tasks at the same time. For instance, a user may be editing, printing, and compiling in parallel.

Cooperating processes require an **inter-process communication** (**IPC**) mechanism that will allow them to exchange data and information. There are two fundamental models of inter-process communication: (1) **shared memory** and (2) **message passing**.

In the **shared memory 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. The two communications models are contrasted in Figure 5.7.

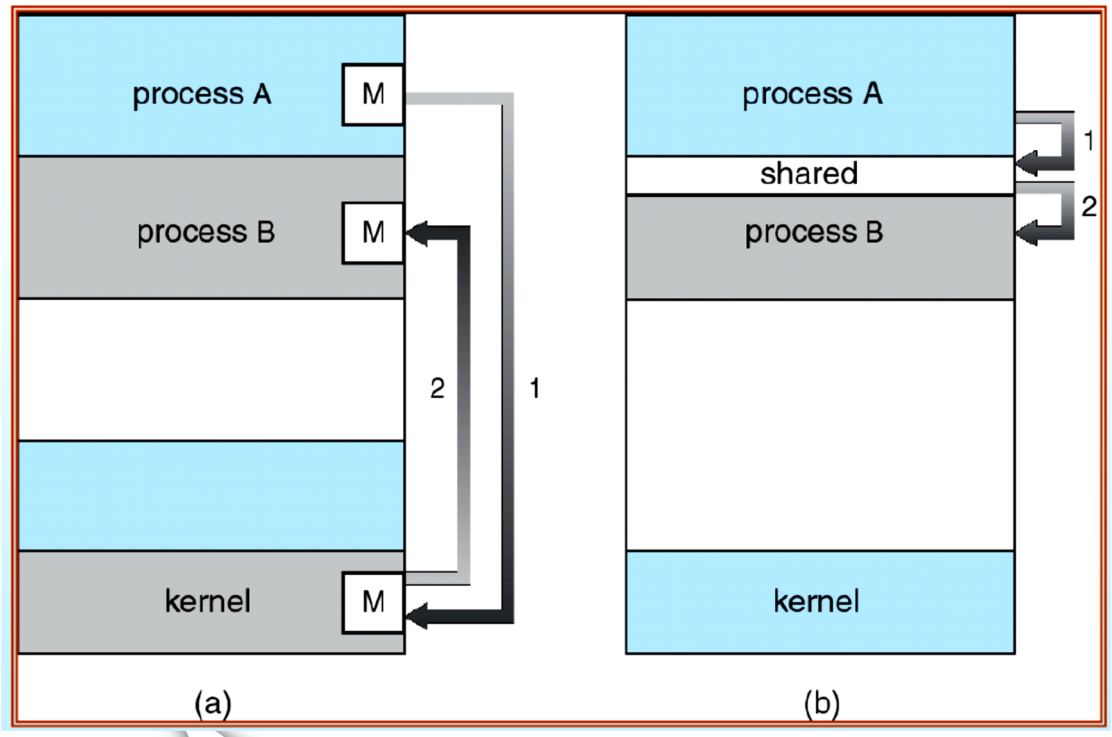


Figure 5.7 Communications models, (a) Message passing, (b) Shared memory.

RTH

End of chapter 5