1 Process Management

- What is a process? The most central concept in any OS is the **process**: an abstraction of a running program.
- At the very least, we are recognizing that some program code is resident in memory and the CPU is fetching the instructions in this code and executing them (including the current values of the program counter, registers, and variables).
- A process can be thought of as a program in execution. A unit of execution characterized by a single, sequential thread of 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.
- In a multiprogramming system, the CPU also switches from program to program, running each for tens or hundreds of milliseconds. While, strictly speaking, at any instant of time, the CPU is running only one program, in the course of 1 second, it may work on several programs, thus giving the users the illusion of parallelism.
- The OS is responsible for the following activities in connection with process and thread management:
  - the creation and deletion of both user and system processes;
  - the scheduling of processes;
  - the provision of mechanisms for synchronization, communication, and deadlock handling for processes.

1.1 Process Concept

- A batch system executes *jobs*, whereas a time-shared system has *user programs*, or *tasks*.
- In many respects, all these activities are similar, so we call all of them ***processes***. The terms *job* and *process* are used almost interchangeably.

1.1.1 The Process

- 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.

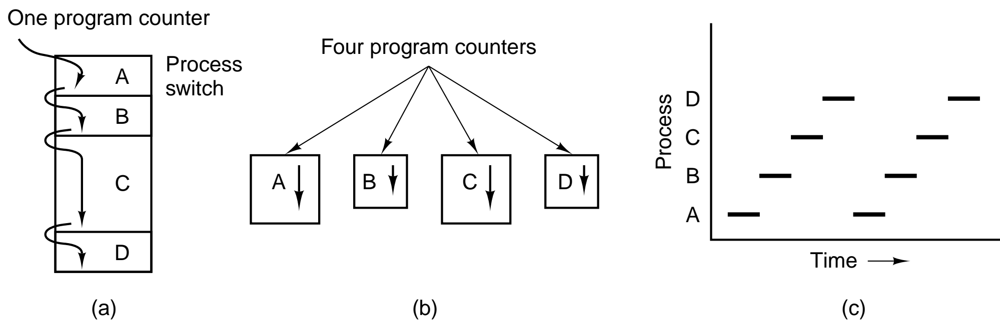

Figure 1: (a) Multiprogramming of four programs. (b) Conceptual model of four independent, sequential processes. (c) Only one program is active at once.

- In Fig. 1a we see a computer multiprogramming four programs in memory.
- In Fig. 1b we see four processes, each with its own flow of control (i.e., its own logical program counter), and each one running independently of the other ones. Of course, there is only one physical program counter, so when each process runs, its logical program counter is loaded into the real program counter. When it is finished for the time being, the physical program counter is saved in the process'logical program counter in memory.
- In Fig. 1c we see that viewed over a long enough time interval, all the processes have made progress, but at any given instant only one process is actually running.

- A process generally also includes
  - a **data section**, which contains global variables,
  - the process **stack**, which contains temporary data (such as function parameters, return addresses, and local variables),

- a **heap**, which is memory that is dynamically allocated during process run time.

- The structure of a process in memory is shown in Fig. 2.

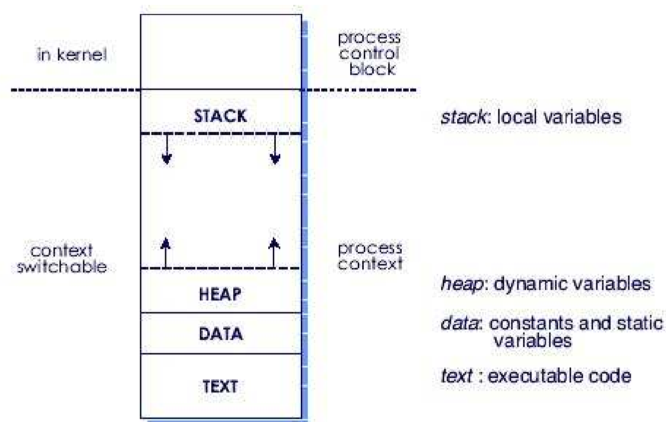

Figure 2: Process in memory.

- A program becomes a process when an executable file is loaded into memory.

- Although two processes may be associated with the same program, they are nevertheless considered two separate execution sequences.

  - For instance, several users may be running different copies of the mail program,

  - or the same user may invoke many copies of the web browser program.

- Each of these is a separate process; and although the text sections are equivalent, the data, heap, and stack sections vary.

1.1.2 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 Fig. 3.

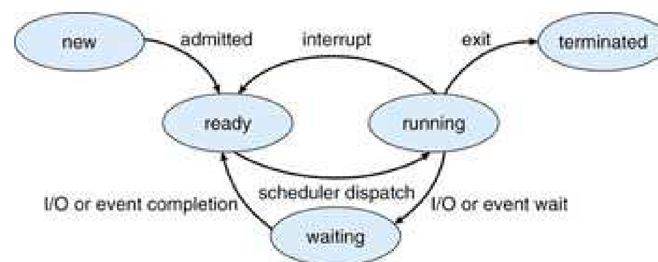

Figure 3: Diagram of process state.

- Using the process model, it becomes much easier to think about what is going on inside the system.
  - Some of the processes run programs that carry out commands typed in by a user.
  - Other processes are part of the system and handle tasks such as carrying out requests for file services or managing the details of running a disk or a tape drive.
  - When a disk interrupt occurs, the system makes a decision to stop running the current process and run the disk process, which was blocked waiting for that interrupt.

- Instead of thinking about interrupts, we can think about user processes, disk processes, terminal processes, and so on, which block when they are waiting for something to happen. When the disk has been read or the character typed, the process waiting for it is unblocked and is eligible to run again. This view gives rise to the model shown in Fig. 4.

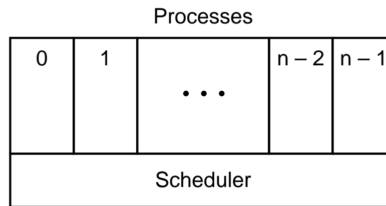

Figure 4: The lowest layer of a process-structured OS handles interrupts and scheduling. Above that layer are sequential processes.

1.1.3 Process Control Block

- The OS must know specific information about processes in order to manage, control them and also to implement the process model, the OS maintains a table (an array of structures), called the **process table**, with one entry per process.
- These entries are called **process control blocks** (PCB) - also called a task control block.
- This entry contains information about the process'state, its program counter, stack pointer, memory allocation, the status of its open files, its accounting and scheduling information, and everything else about the process that must be saved when the process is switched from running to ready or blocked state so that it can be restarted later as if it had never been stopped. A PCB is shown in Fig. 5.

Figure 5: Process control block (PCB).

- Such information is usually grouped into two categories: *Process State Information* and *Process Control Information*. 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 OS.
  - **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 6 shows some of the more important fields in a typical system. The fields in the first column relate to process management. The other two columns relate to memory management and file management, respectively.

Process management	Memory management	File management
Registers	Pointer to text segment info	Root directory
Program counter	Pointer to data segment info	Working directory
Program status word	Pointer to stack segment info	File descriptors
Stack pointer		User ID
Process state		Group ID
Priority		
Scheduling parameters		
Process ID		
Parent process		
Process group		
Signals		
Time when process started		
CPU time used		
Children's CPU time		
Time of next alarm		

Figure 6: Some of the fields of a typical process table entry.

- Along with the program counter, this state information must be saved when an interrupt occurs, to allow the process to be continued correctly afterward (see Fig. 7).

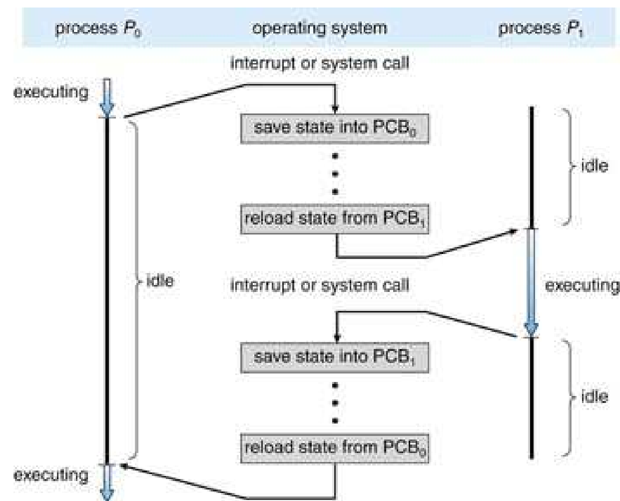

Figure 7: Diagram showing CPU switch from process to process.

1.2 Process Scheduling

- The objective of multiprogramming is to have some process running at all times, to maximize CPU utilization.
- With the CPU switching back and forth among the processes, the rate at which a process performs its computation will not be uniform and probably not even reproducible if the same processes are run again.
- 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.

1.2.1 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**.
- This queue is generally stored as a linked list. 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.
- 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 (see Fig. 8).
- A common representation for a discussion of process scheduling is a queuing diagram, such as that in Fig. 9.
  - Each rectangular box represents a queue. Two types of queues are present: the ready queue and a set of device queues.

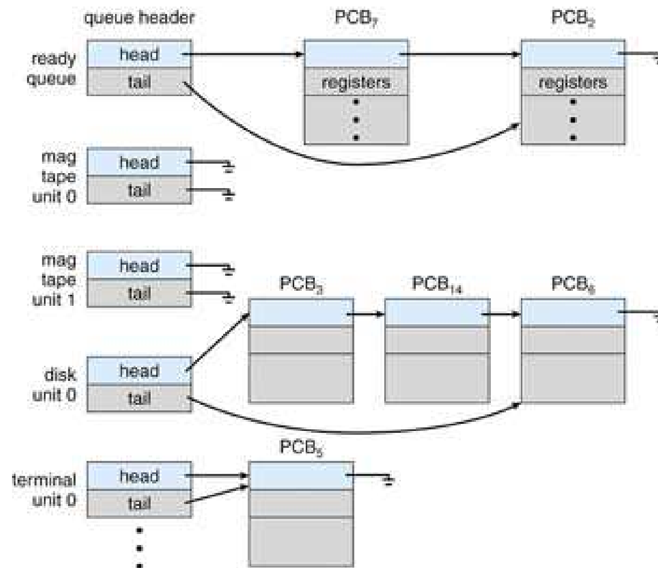

Figure 8: The ready queue and various I/O device queues.

  - The circles represent the resources that serve the queues, and the arrows indicate the flow of processes in the system.

- 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 subprocess and wait for the subprocess'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.
- A process continues this cycle until it terminates, at which time it is removed from all queues and has its PCB and resources deallocated.

1.2.2 Schedulers

- A process migrates among the various scheduling queues throughout its lifetime.

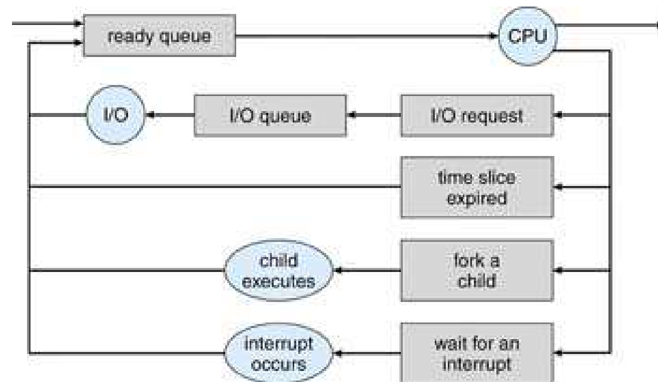

Figure 9: Queueing-diagram representation of process scheduling.

- The OS 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.
- The long-term scheduler controls the **degree of multiprogramming** (the number of processes in memory). If the degree of multiprogramming is stable, then the average rate of process creation must be equal to the average departure rate of processes leaving the system.
- Thus, the long-term scheduler may need to be invoked only when a process leaves the system. Because of the longer interval between executions, the long-term scheduler can afford to take more time to decide which process should be selected for execution.
- It is important that the long-term scheduler make a careful selection. In general, most processes can be described as either I/O bound or CPU bound.

- An I/O-bound process is one that spends more of its time doing I/O than it spends doing computations.
- A CPU-bound process, in contrast, generates I/O requests infrequently, using more of its time doing computations.

- 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 systems often have no long-term scheduler but simply put every new process in memory for the short-term scheduler.

- The stability of these systems depends either on a physical limitation (such as the number of available terminals) or on the self-adjusting nature of human users.

- Some OSs, such as time-sharing systems, may introduce an additional, intermediate level of scheduling. This **medium-term scheduler** is diagrammed in Fig. 10.

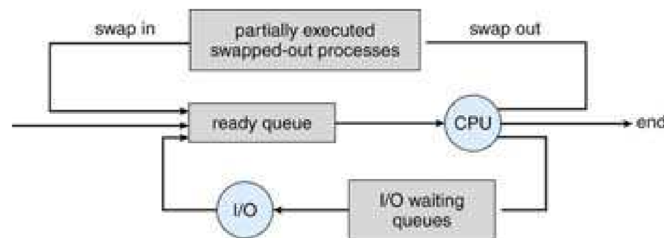

Figure 10: Addition of medium-term scheduling to the queuing diagram.

- 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**.

1.2.3 Context Switch

- When an interrupt occurs, the system needs to save the current **context** of the process currently 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.
- 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.
  - process table keeps track of processes,
  - context information stored in PCB,
  - process suspended: register contents stored in PCB,
  - process resumed: PCB contents loaded into registers
- Context-switch time is pure overhead, because the system does no useful work while switching. Context switching can be critical to performance.

1.2.4 Modelling Multiprogramming

- When multiprogramming is used, the CPU utilization can be improved. Crudely put, if the average process computes only 20% of the time it is sitting in memory at once, the CPU should be busy all the time.
- Unrealistically optimistic, assumes that all five processes will never be waiting for I/O at the same time.
- Suppose that a process spend a fraction p of its time waiting for I/O to complete. With n processes in memory at once, the probability that all n processes are waiting for I/O is p^n. The CPU utilization is then given by the formula:

$$CPU\ utilization = 1 - p^n$$

- Figure 11 shows the CPU utilization as a function of n, which is called the **degree of multiprogramming**.

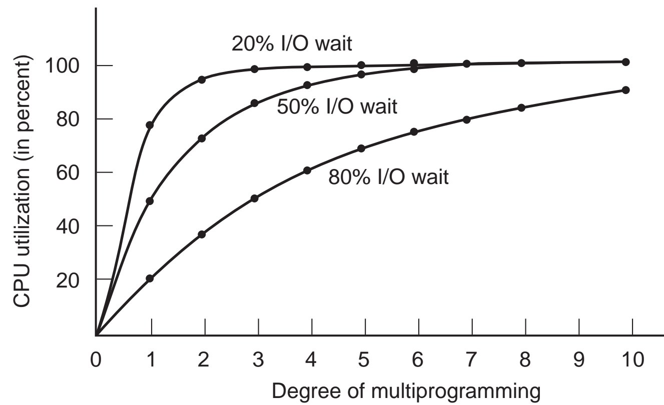

Figure 11: CPU utilization as a function of the number of processes in memory.

- For the sake of complete accuracy, it should be pointed out that the probabilistic model is only an approximation. Context switching overhead is ignored.

1.3 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.

1.3.1 Process Creation

- There are four principal events that cause processes to be created:

  1. *System initialization.* When an OS is booted, typically several processes are created.
  2. *Execution of a process creation system call by a running process.* Often a running process will issue system calls to create one or more new processes to help it do its job. Creating new processes is particularly useful when the work to be done can easily be formulated in terms of several related, but otherwise independent interacting processes.

3. *A user request to create a new process.* In interactive systems, users can start a program by typing a command or (double) clicking an icon.
4. *Initiation of a batch job.* Here users can submit batch jobs to the system (possibly remotely). When the OS decides that it has the resources to run another job, it creates a new process and runs the next job from the input queue in it.

- In all these cases, a new process is created by having an existing process execute a process creation system call (in UNIX,$fork()$).
- 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 OSs (including UNIX and the Windows family of OSs) identify processes according to a unique process identifier (or pid), which is typically an integer number.
- In general, a process will need certain resources (CPU time, memory, files, I/0 devices) to accomplish its task. When a process creates a subprocess, that subprocess may be able to obtain its resources directly from the OS, 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.
- Restricting a child process to a subset of the parent's resources prevents any process from overloading the system by creating too many subprocesses.
- When a process creates a new process, two possibilities exist in terms of execution:
  1. The parent continues to execute concurrently with its children, competing equally for the CPU.
  2. The parent waits until some or all of its children have terminated (on UNIX, see the man pages for {wait, waitpid, wait4, wait3}.).
- 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 (it has the same program and data as the parent, an exact clone). The two processes, the *parent* and the *child*, have the same memory image, the same environment strings, and the same open files.
2. The child process has a new program loaded into it.

- The C program shown below illustrates the UNIX system calls previously described.

```
#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");
  exit(-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");
  exit(0);
 }
}
```

  - We now have two different processes running a copy of the same program.
  - The value of pid for the child process is zero; that for the parent is an integer value greater than zero.
  - The child process overlays its address space with the UNIX command */bin/ls* (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.
– This is also illustrated in Fig. 12.

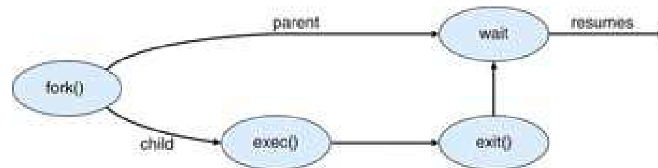

Figure 12: Process creation.

1.3.2 Process Termination

- *Normal exit (voluntary)*: A process terminates when it finishes executing its final statement and asks the OS 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 (via the $wait()$ system call). All the resources of the process -including physical and virtual memory, open files, and I/O buffers- are deallocated by the OS.
- *Abnormal termination*: programming errors, run time errors, I/O, user intervention.
  - *Error exit (voluntary)*: An error caused by the process, often due to a program bug (executing an illegal instruction, referencing non-existent memory, or dividing by zero). In some systems (e.g. UNIX), a process can tell the OS that it wishes to handle certain errors itself, in which case the process is signaled (interrupted) instead of terminated when one of the errors occurs.
  - *Fatal error (involuntary)*: i.e.; no such file exists during the compilation.
  - *Killed by another process (involuntary)*: A process can cause the termination of another process via an appropriate system call (for example, $TerminateProcess()$ in Win32). Usually, such a system call can be invoked only by the parent of the process that is to be terminated.
- 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 OS does not allow a child to continue if its parent terminates (cascading termination).

- To illustrate process execution and termination, consider that, in UNIX, we can terminate a process by using the *exit*() system call; its parent process may wait for the termination of a child process by using the *wait*() system call.
- The *wait*() system call returns the process identifier of a terminated child so that the parent can tell which of its possibly many children has terminated.
- If the parent terminates, then the child will become a *zombie* process and may be listed as such in the process status list!. This is not always true since all its children could have been assigned as their new parent the *init* process. Thus, the children still have a parent to collect their status and execution statistics.

1.4 Interprocess Communication

- Processes executing concurrently in the OS 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. Notice that such a speedup can be achieved only if the computer has multiple processing elements (such as CPUs or I/O channels).
- **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 **interprocess communication** (IPC) mechanism that will allow them to exchange data and information.

- There are two fundamental models of interprocess communication:
  - **Shared Memory**. 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.
  - **Message Passing**. Communication takes place by means of messages exchanged between the cooperating processes. The two communications models are contrasted in Fig. 13.

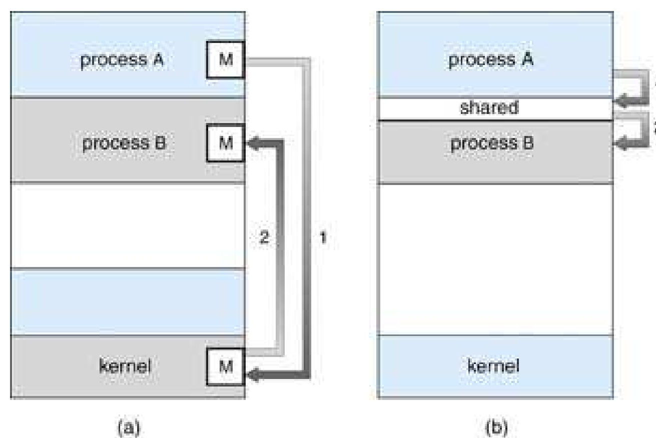

Figure 13: Communications models. (a) Message passing. (b) Shared memory.

- Both of the models just discussed are common in OSs, and many systems implement both.
- Message passing is useful for exchanging smaller amounts of data, because no conflicts need be avoided. Message passing is also easier to implement than is shared memory for intercomputer communication.
- Shared memory allows maximum speed and convenience of communication, as it can be done at memory speeds when within a computer. Shared memory is faster than message passing, as message-passing systems are typically implemented using system calls and thus require the more time-consuming task of kernel intervention.
- In contrast, in shared-memory systems, 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.

1.4.1 Shared-Memory Systems

- Interprocess communication using shared memory requires communicating processes to establish a region of shared memory. Typically, 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.
- Recall that, normally, the OS tries to prevent one process from accessing another process's memory. Shared memory requires that two or more processes agree to remove this restriction. They can then exchange information by reading and writing data in the shared areas.
- The form of the data and the location are determined by these processes and are not under the OS's control. The processes are also responsible for ensuring that they are not writing to the same location simultaneously.
- To illustrate the concept of cooperating processes, let's consider the **producer-consumer problem**, which is a common paradigm for cooperating processes. A producer process produces information that is consumed by a consumer process.

- One solution to the producer-consumer problem uses shared memory. To allow producer and consumer processes to run concurrently, we must have available a buffer of items that can be filled by the producer and emptied by the consumer.

- This buffer will reside in a region of memory that is shared by the producer and consumer processes. A producer can produce one item while the consumer is consuming another item.

- The producer and consumer must be synchronized, so that the consumer does not try to consume an item that has not yet been produced.

- Two types of buffers can be used.

  1. The **unbounded buffer** places no practical limit on the size of the buffer. The consumer may have to wait for new items, but the producer can always produce new items.
  2. The **bounded buffer** assumes a fixed buffer size. In this case, the consumer must wait if the buffer is empty, and the producer must wait if the buffer is full.