Lecture 4

Processes

Lecture Information

Ceng328 *Operating Systems* at March 9, 2010

Dr. Cem Özdoğan
Computer Engineering Department
Çankaya University

Contents

Process Management

- What is a process? The most central concept in any OS is the **process**: an abstraction of a running program.

Process Management

- What is a process? The most central concept in any OS is the **process**: an abstraction of a running program.
- A process can be thought of as a program in execution. A unit of execution characterized by a single, sequential thread of execution.

Process Management

- What is a process? The most central concept in any OS is the **process**: an abstraction of a running program.
- A process can be thought of as a program in execution. A unit of execution characterized by a single, sequential thread of execution.
- The resources are allocated to the process either when it is created or while it is executing.

Process Management

- What is a process? The most central concept in any OS is the **process**: an abstraction of a running program.
- A process can be thought of as a program in execution. A unit of execution characterized by a single, sequential thread of execution.
- The resources are allocated to the process either when it is created or while it is executing.
- The OS is responsible for the following activities in connection with process and thread management:

Process Management

- What is a process? The most central concept in any OS is the **process**: an abstraction of a running program.
- A process can be thought of as a program in execution. A unit of execution characterized by a single, sequential thread of execution.
- The resources are allocated to the process either when it is created or while it is executing.
- 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;

Process Management

- What is a process? The most central concept in any OS is the **process**: an abstraction of a running program.
- A process can be thought of as a program in execution. A unit of execution characterized by a single, sequential thread of execution.
- The resources are allocated to the process either when it is created or while it is executing.
- 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;

Process Management

- What is a process? The most central concept in any OS is the **process**: an abstraction of a running program.
- A process can be thought of as a program in execution. A unit of execution characterized by a single, sequential thread of execution.
- The resources are allocated to the process either when it is created or while it is executing.
- 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.

The Process I

- A process is more than the program code, which is sometimes known as the **text section**.

The Process I

- 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.

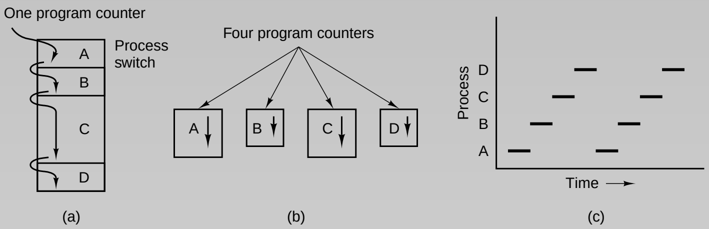

Figure: (a) Multiprogramming of four programs. (b) Conceptual model of four independent, sequential processes. (c) Only one program is active at once.

The Process II

- A process generally also includes

The Process II

- A process generally also includes
  - a **data section**, which contains global variables,

The Process II

- 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),

The Process II

- 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 Process II

- 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.

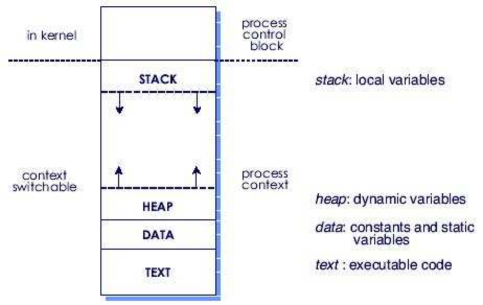

Figure: Process in memory.

The Process III

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

The Process III

- 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.

The Process III

- 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 <u>two separate execution sequences</u>.
  - For instance, several users may be running different copies of the mail program,

The Process III

- 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.

The Process III

- 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.

Process State I

- As a process executes, it changes **state**.

Process State I

- As a process executes, it changes **state**.
- The state of a process is defined in part by the current activity of that process.

Process State I

- 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:

Process State I

- 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.

Process State I

- 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.

Process State I

- 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).

Process State I

- 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.

Process State I

- 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.

Process State I

- 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.

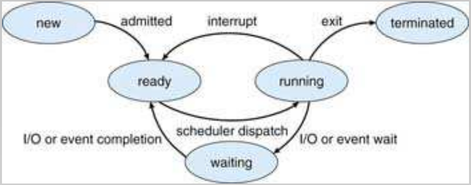

Figure: Diagram of process state.

Process State II

- Instead of thinking about interrupts,

Process State II

- 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.

Process State II

- 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.

Process State II

- 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.

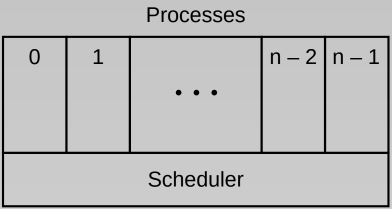

Figure: The lowest layer of a process-structured OS handles interrupts and scheduling. Above that layer are sequential processes.

Process Control Block I

- The OS must know specific information about processes in order to manage, control them and also to implement the process model.

Process Control Block I

- 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.

Process Control Block I

- 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. Keeps the information; everything about the process.

Figure: Process control block (PCB).

Process Control Block II

- Such information is usually grouped into two categories: *Process State Information* and *Process Control Information*. Including these:

Process Control Block II

- Such information is usually grouped into two categories: *Process State Information* and *Process Control Information*. Including these:
  - **Process state**.

Process Control Block II

- Such information is usually grouped into two categories: *Process State Information* and *Process Control Information*. Including these:
  - **Process state**.
  - **Program counter**.

Process Control Block II

- Such information is usually grouped into two categories: *Process State Information* and *Process Control Information*. Including these:
  - **Process state**.
  - **Program counter**.
  - **CPU registers**.

Process Control Block II

- Such information is usually grouped into two categories: *Process State Information* and *Process Control Information*. Including these:
  - **Process state**.
  - **Program counter**.
  - **CPU registers**.
  - **CPU-scheduling information**.

Process Control Block II

- Such information is usually grouped into two categories: *Process State Information* and *Process Control Information*. Including these:
  - **Process state**.
  - **Program counter**.
  - **CPU registers**.
  - **CPU-scheduling information**.
  - **Memory-management information**.

Process Control Block II

- Such information is usually grouped into two categories: *Process State Information* and *Process Control Information*. Including these:
  - **Process state**.
  - **Program counter**.
  - **CPU registers**.
  - **CPU-scheduling information**.
  - **Memory-management information**.
  - **Accounting information**.

Process Control Block II

- Such information is usually grouped into two categories: *Process State Information* and *Process Control Information*. Including these:
    - **Process state**.
    - **Program counter**.
    - **CPU registers**.
    - **CPU-scheduling information**.
    - **Memory-management information**.
    - **Accounting information**.
    - **I/O status information**.

Process Control Block III

- Figure 6 shows some of the more important fields in a typical system.

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

Process Control Block III

- Figure 6 shows some of the more important fields in a typical system.

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

- The fields in the first column relate to process management. The other two columns relate to memory management and file management, respectively.

Process Control Block IV

- 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).

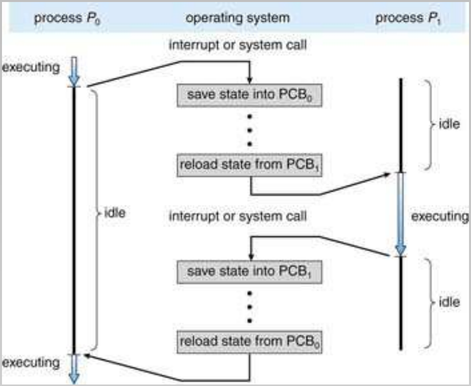

Figure: Diagram showing CPU switch from process to process.

Process Scheduling

- The objective of multiprogramming is to have some process running at all times, to maximize CPU utilization.

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.

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.

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.

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.

Process Scheduling

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

Scheduling Queues I

- As processes enter the system, they are put into a **job queue**, which consists of all processes in the system.

Scheduling Queues I

- 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**.

Scheduling Queues I

- 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.

Scheduling Queues I

- 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.

Scheduling Queues I

- 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.

Scheduling Queues I

- 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.

Scheduling Queues II

- 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).

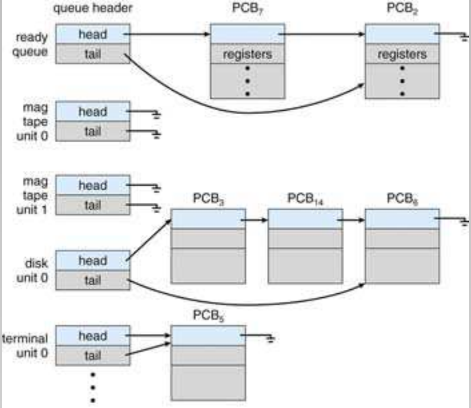

Figure: The ready queue and various I/O device queues.

Scheduling Queues III

- A common representation for a discussion of process scheduling is a queuing diagram, such as that in Fig. 9.

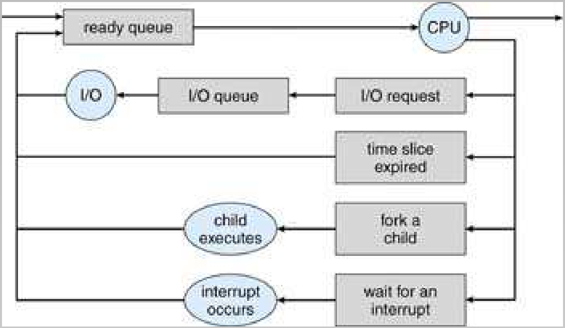

Figure: Queueing-diagram representation of process scheduling.

Scheduling Queues III

- A common representation for a discussion of process scheduling is a queuing diagram, such as that in Fig. 9.

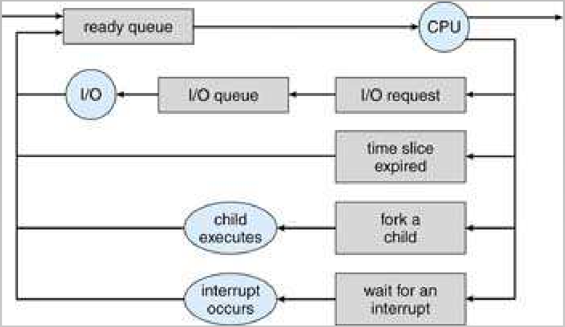

Figure: Queueing-diagram representation of process scheduling.

- Each rectangular box represents a queue. Two types of queues are present: the ready queue and a set of device queues.

Scheduling Queues III

- A common representation for a discussion of process scheduling is a queuing diagram, such as that in Fig. 9.

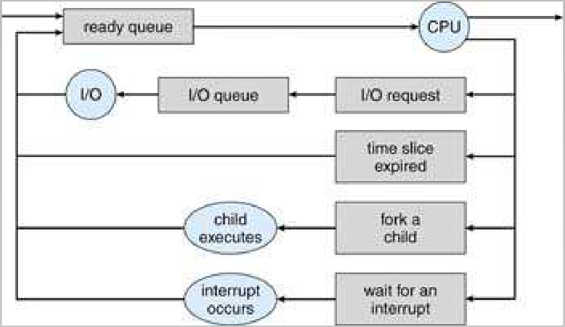

Figure: Queueing-diagram representation of process scheduling.

- 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.

Scheduling Queues IV

- A new process is initially put in the ready queue. It waits there until it is selected for execution, or is **dispatched**.

Scheduling Queues IV

- 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:

Scheduling Queues IV

- 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.

Scheduling Queues IV

- 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.

Scheduling Queues IV

- 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.

Scheduling Queues IV

- 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.

Schedulers I

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

Schedulers I

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

Schedulers I

- A process migrates among the various scheduling queues throughout its lifetime.
- The OS must select, for scheduling purposes, processes from these queues in some fashion. The selection process is carried out by the appropriate **scheduler**.
- The **long-term scheduler**, or **job scheduler**, selects processes from this pool and loads them into memory for execution.

Schedulers I

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

Schedulers I

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

Schedulers II

- 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.

Schedulers II

- 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.

Schedulers II

- 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.
- 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.

Schedulers II

- 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.
- 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.

Schedulers II

- 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.
- 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.

Schedulers II

- 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.
- 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.

Schedulers II

- 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.
- 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.

Schedulers II

- 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.
- 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.

Schedulers III

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

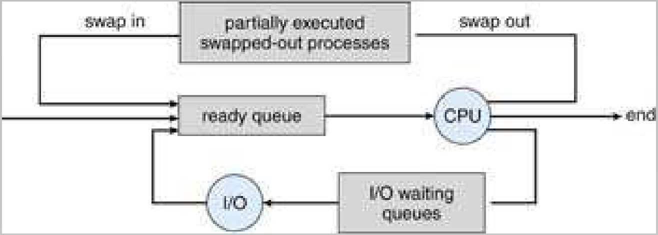

Figure: Addition of medium-term scheduling to the queuing diagram.

Schedulers III

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

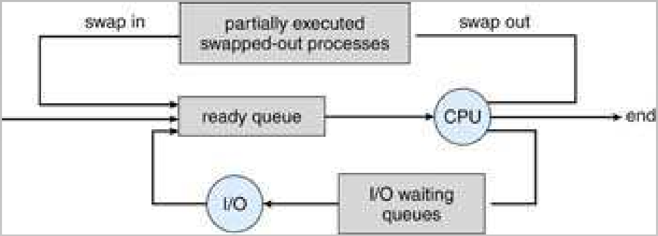

Figure: Addition of medium-term scheduling to the queuing diagram.

- 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.

Schedulers III

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

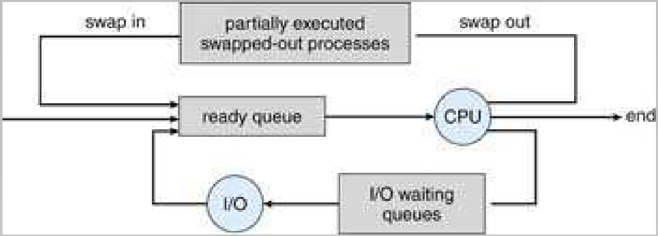

Figure: Addition of medium-term scheduling to the queuing diagram.

- 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**.

Context Switch

- When an interrupt occurs, the system needs to save the current **context** of the process currently running on the CPU.

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.

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.
- Switching the CPU to another process requires performing a state save of the current process and a state restore of a different process.

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.
- 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

- 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.
- 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 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.
- 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,

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.
- 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,

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.
- 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

- 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.
- 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 <u>overhead</u>, because the system does <u>no useful work while switching</u>.

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.
- 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.

Modelling Multiprogramming I

- 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.

Modelling Multiprogramming I

- 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.

Modelling Multiprogramming I

- 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:

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

Modelling Multiprogramming II

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

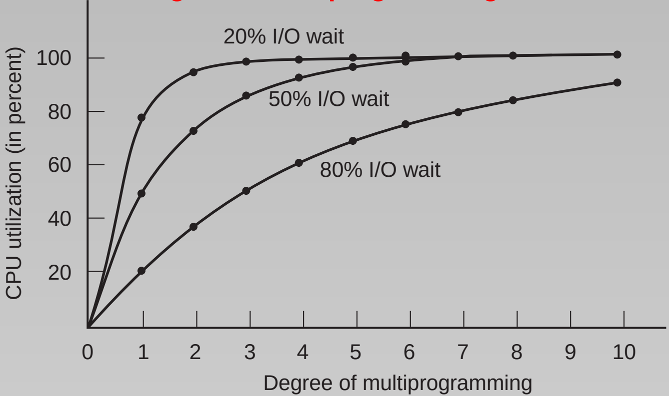

Figure: CPU utilization as a function of the number of processes in memory.

Modelling Multiprogramming II

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

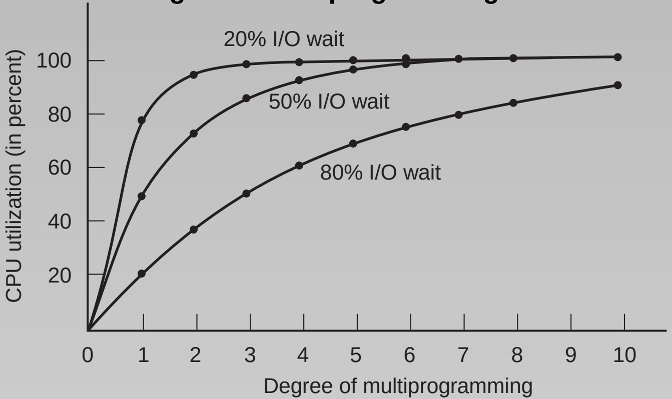

Figure: CPU utilization as a function of the number of processes in memory.

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

Process Creation I

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

Process Creation I

- There are four principal events that cause processes to be created:
  1. *System initialization.*

Process Creation I

- There are four principal events that cause processes to be created:
  1. *System initialization.*
  2. *Execution of a process creation system call by a running process.*

Process Creation I

- There are four principal events that cause processes to be created:
  1. *System initialization.*
  2. *Execution of a process creation system call by a running process.*
  3. *A user request to create a new process.*

Process Creation I

- There are four principal events that cause processes to be created:
  1. *System initialization.*
  2. *Execution of a process creation system call by a running process.*
  3. *A user request to create a new process.*
  4. *Initiation of a batch job.*

Process Creation I

- There are four principal events that cause processes to be created:
  1. *System initialization.*
  2. *Execution of a process creation system call by a running process.*
  3. *A user request to create a new process.*
  4. *Initiation of a batch job.*
- In all these cases, a new process is created by having an existing process execute a process creation system call (in UNIX,*fork*()).

Process Creation I

- There are four principal events that cause processes to be created:
  1. *System initialization.*
  2. *Execution of a process creation system call by a running process.*
  3. *A user request to create a new process.*
  4. *Initiation of a batch job.*
- 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.

Process Creation I

- There are four principal events that cause processes to be created:
  1. *System initialization.*
  2. *Execution of a process creation system call by a running process.*
  3. *A user request to create a new process.*
  4. *Initiation of a batch job.*
- 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.

Process Creation I

- There are four principal events that cause processes to be created:
  1. *System initialization.*
  2. *Execution of a process creation system call by a running process.*
  3. *A user request to create a new process.*
  4. *Initiation of a batch job.*
- 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.

Process Creation II

- When a process creates a subprocess, that subprocess may be able to obtain its resources directly from the OS,

Process Creation II

- 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.

Process Creation II

- 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,

Process Creation II

- 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.

Process Creation II

- 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.

Process Creation III

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

Process Creation III

- 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.

Process Creation III

- 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}.).

Process Creation III

- 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:

Process Creation III

- 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.

Process Creation III

- 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.

Process Creation IV

The C program shown below illustrates the UNIX system calls.

```
#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);
 }
}
```

Process Creation V

We now have two different processes running a copy of the same program. This is also illustrated in Fig. 12.

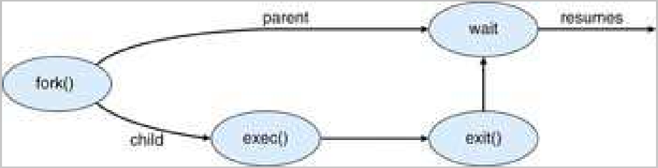

Figure: Process creation.

Process Termination I

- *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.

Process Termination I

- *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).

Process Termination I

- *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).
    - Releasing all the resources.

Process Termination I

- *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).
  - Releasing all the resources.
- *Abnormal termination*: programming errors, run time errors, I/O, user intervention.

Process Termination I

- *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).
  - Releasing all the resources.
- *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).

Process Termination I

- *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).
  - Releasing all the resources.
- *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).
  - *Fatal error (involuntary)*: i.e.; no such file exists during the compilation.

Process Termination I

- *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).
    - Releasing all the resources.
- *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).
    - *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).

Process Termination II

- A parent may terminate the execution of one of its children for a variety of reasons, such as these:

Process Termination II

- 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.

Process Termination II

- 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.

Process Termination II

- 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 OS does not allow a child to continue if its parent terminates (cascading termination).

Process Termination II

- 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 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.

Process Termination II

- 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 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.
- 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.

Process Termination II

- 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 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.
- 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!.

Process Termination II

- 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 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.
- 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.

Interprocess Communication I

- Processes executing concurrently in the OS may be either independent processes or cooperating processes.

Interprocess Communication I

- 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.

Interprocess Communication I

- 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.

Interprocess Communication I

- 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:

Interprocess Communication I

- 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**.

Interprocess Communication I

- 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**.
  - **Computation speedup**.

Interprocess Communication I

- 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**.
  - **Computation speedup**.
  - **Modularity**.

Interprocess Communication I

- 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**.
    - **Computation speedup**.
    - **Modularity**.
    - **Convenience**.

Interprocess Communication II

- Cooperating processes require an **interprocess communication** (IPC) mechanism that will allow them to exchange data and information.

Interprocess Communication II

- 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:

Interprocess Communication II

- 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**.

Interprocess Communication II

- 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**.
  - **Message Passing**.

Interprocess Communication II

- 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**.
  - **Message Passing**.
- Message passing is useful for exchanging smaller amounts of data, because no conflicts need be avoided.

Interprocess Communication II

- 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**.
  - **Message Passing**.
- 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.

Interprocess Communication II

- 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**.
  - **Message Passing**.
- 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.

Interprocess Communication II

- 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**.
  - **Message Passing**.
- 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.
- In shared-memory systems, system calls are required only to establish shared-memory regions (no assistance from the kernel).

Interprocess Communication III

The two communications models are contrasted in Fig. 13.

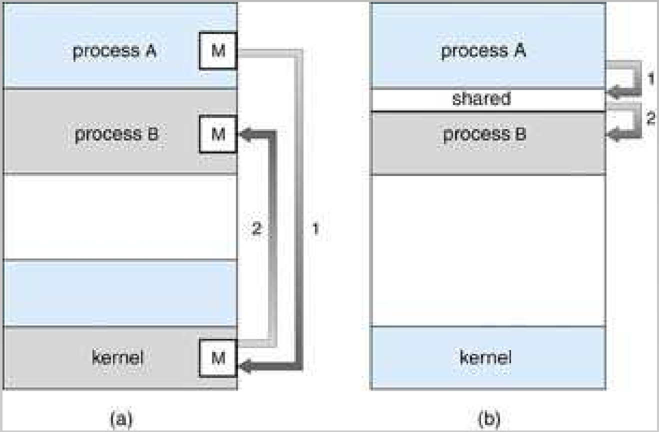

Figure: Communications models. (a) Message passing. (b) Shared memory.

Shared-Memory Systems I

- Typically, a shared-memory region *resides in the address space of the process creating the shared-memory segment*.

Shared-Memory Systems I

- 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.

Shared-Memory Systems I

- 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 Systems I

- 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.

Shared-Memory Systems I

- 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.
- The form of the data and the location are determined by these processes and are not under the OS's control.

Shared-Memory Systems I

- 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.
- 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.

Shared-Memory Systems II

- To illustrate the concept of cooperating processes, let's consider the **producer-consumer problem**.

Shared-Memory Systems II

- To illustrate the concept of cooperating processes, let's consider the **producer-consumer problem**.
- One solution to the producer-consumer problem uses shared memory.

Shared-Memory Systems II

- To illustrate the concept of cooperating processes, let's consider the **producer-consumer problem**.
- 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.

Shared-Memory Systems II

- To illustrate the concept of cooperating processes, let's consider the **producer-consumer problem**.
- 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.

Shared-Memory Systems II

- To illustrate the concept of cooperating processes, let's consider the **producer-consumer problem**.
- 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.
- The producer and consumer must be **synchronized**.

Shared-Memory Systems II

- To illustrate the concept of cooperating processes, let's consider the **producer-consumer problem**.
- 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.
- The producer and consumer must be **synchronized**.
- Two types of buffers can be used.

Shared-Memory Systems II

- To illustrate the concept of cooperating processes, let's consider the **producer-consumer problem**.
- 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.
- The producer and consumer must be **synchronized**.
- 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.

Shared-Memory Systems II

- To illustrate the concept of cooperating processes, let's consider the **producer-consumer problem**.
- 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.
- The producer and consumer must be **synchronized.**
- 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.