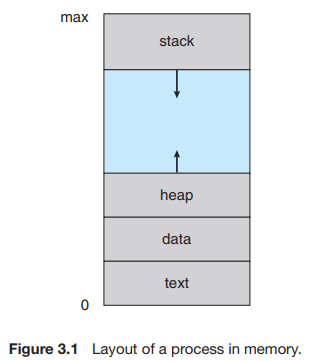
1. **Process Concept**
2. The Process

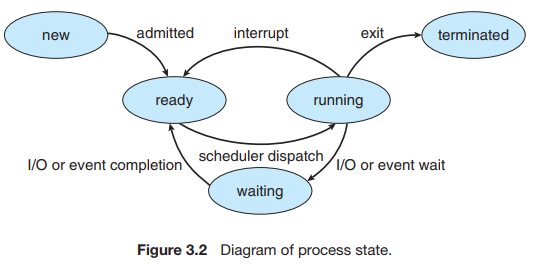
* A process is a program in execution
* Status of the current activity of a process: program counter + contents of the processor’s registers
* Memory layout:
* Text section: executable code
* Data section: global variables
* Heap section: memory that is dynamically allocated during execution
* Stack section: temporary data storage when invoking functions



* Activation record:
* Contains function parameters, local variables, etc.
* Is pushed onto the stack when a function is called
* Is popped from the stack when the control is returned from the function

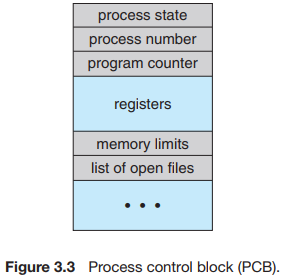
1. Process State

* As a process executes, it changes state:
* New: The process is being created
* Running: Instructions are being executed
* Waiting: The process is waiting for some event to occur (e.g: I/O completion)
* Ready: The process is waiting to be assigned to a processor
* Terminated: The process has finished execution



1. Process Control Block

* Represents each process in the OS.
* Serves as the repository for all the data needed to start/restart the process, along with some accounting data
* Contains many pieces of information associated with a specific process



* Process state: new, ready, running, waiting, halted, etc.
* Program counter: indicates the address of the next instruction to be executed for this process.
* CPU registers: vary in number and type, depending on the computer architecture. They include accumulators, index registers, stack pointers, general-purpose registers and any condition-code information. Along with the program counter, they must be saved when an interrupt occurs -> allow the process to be continued correctly afterward.
* CPU-scheduling information: includes a process priority, pointers to scheduling queues, and any other scheduling parameters.
* Memory management information: may include such items as the value of the base and limit registers and the page tables, or the segment tables, depending on the system used by the OS.
* Accounting information: includes the amount of CPU and real time used, time limits, account numbers, job or process numbers, etc.
* I/O status information: includes the list of I/O devices allocated to the process, a list of open files, etc.

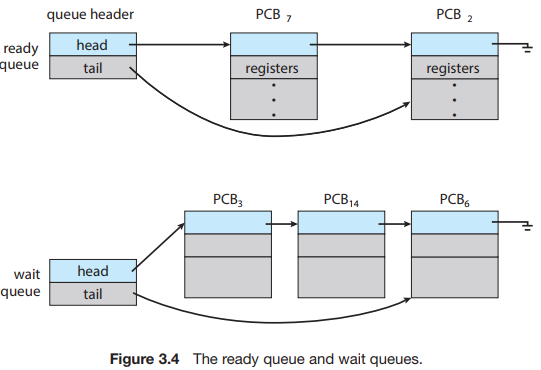
1. Threads

* the smallest sequence of programmed instructions that can be managed independently by a scheduler, which is typically a part of the operating system.
* Most modern operating systems have extended the process concept to allow a process to have multiple threads of execution and thus to perform more than one task at a time.

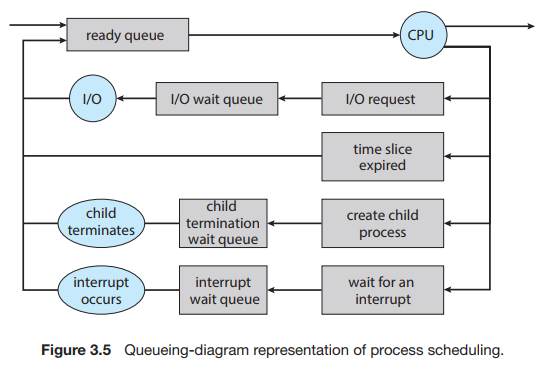
**II. Process Scheduling**

* Objective of multiprogramming: have some process running at all times -> maximize CPU utilization
* Objective of time sharing: to switch a CPU core among processes so frequently that users can interact with each program while it is running

→ Process scheduler: select an available process for program execution on a core



1. Scheduling Queues



* This process could issue an I/O request and then be placed in an I/O wait queue
* The process could create a new child process and then be placed in a wait queue while it awaits the child’s termination
* The process could be removed forcibly from the core, as a result of an interrupt or having its time slice expire, and be put back in the ready queue

1. CPU Scheduling

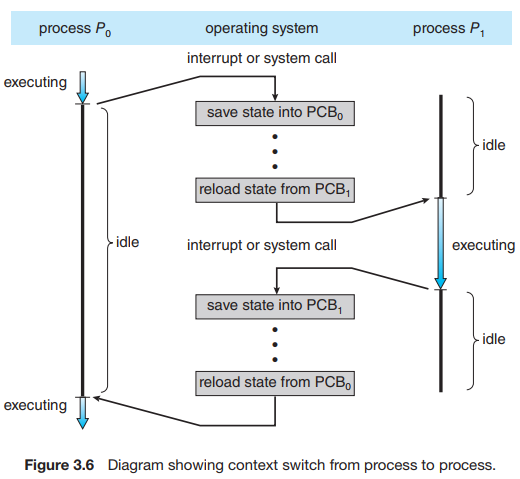
* CPU scheduler: select from among the processes that are in the ready queue and allocate a CPU core to one of them.
* Swapping: sometimes it can be advantageous to remove a process from memory -> reduce the degree of multiprogramming. Later the process can be reintroduced into memory, and its execution can be continued where it left off

1. Context Switch

* When interruption occurs -> causes OS to change a CPU core from its current task and to run a kernel routine

→ Needs to save the current context of the process -> restore later

* Context switch: state save and state restore



**III. Operations on Processes**

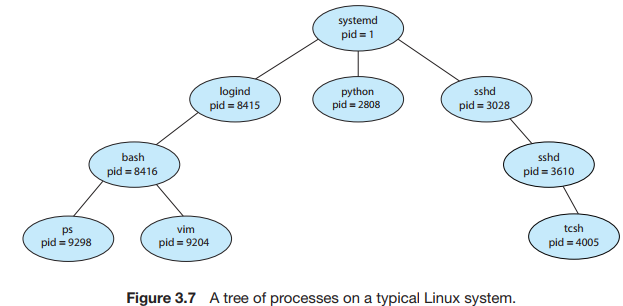
1. Process Creation

* During execution, a process may create new processes -> Tree of processes
* Process identifie (pid): provides a unique value for each process in the system, can be used as an index to access various attributes of a process
* Child processes will need certain resources (CPU time, memory, files, I/O devices): can be obtained directly from the OS or constrained to a subset of the parent process.

→ Need to prevent any process from overloading the system by creating too many child processes

→ Restricting a child process to a subset of the parent’s resources

* When a process creates a new process, 2 possibilities for execution:
* Continues to execute concurrently with its children
* Waits until some or all of its children have terminated
* 2 address-space possibilities for the new process:
* The child is a duplicate of the parent (same program same data)
* The child has a new program loaded into it



1. Process Termination

* Terminates when it finishes executing its final statement and asks the OS to delete it -> The process may return a status value (typically an int) to its waiting parent process.
* Other circumstances: Can cause the extermination of another process via an appropriate system call.

→ Usually, can be invoked only by the parent of the process that is to be

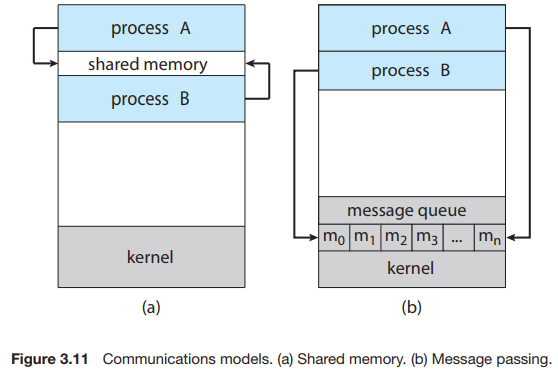
terminated.

* A parent may terminate the execution of one of its children:
* 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.

→ Last scenario: Cascading Termination

**IV. Interprocess Communication**

* Process can be either independent or cooperating
* Independent: it does not share data with any other processes executing in the system.
* Cooperating: it can affect or be affected by other processes executing in the system.
* Reasons for providing an environment that allows cooperation:
* Information sharing: several applications may be interested in the same piece of information.
* Computation speedup: a task runs faster -> break it into subtasks -> each will be executed in parallel with the others.
* Modularity: we may want to construct the system in a modular fashion, dividing the system functions into separate processes or threads
* Require Interprocess Communication (IPC) that will allow exchanging data:
* Shared memory: a region of memory that is shared by the cooperating processes is established. Information exchange can be done by reading and writing data to the shared region.
* Message passing: communication takes place by means of messages exchanged between the cooperating processes.



* Message passing:
* Useful for exchanging smaller amounts of data -> no conflicts need be avoided
* Easier to implement in a distributed system
* Shared memory:
* Can be faster: system calls are required only to establish shared region

→ Once done, all accesses are treated as routine memory accesses

→ No assistance from the kernel is required

**V. IPC in Shared-Memory Systems**

* Shared memory system: a region of shared memory -> resides in the address space of the process creating the shared memory segment -> If want to communicate -> must attach the segment to their address space.
* Problem: normally, the OS tries to prevent one process from accessing another process’s memory -> Shared memory requires those processes agree to remove this restriction

→ Producer-consumer problem: To run concurrently, we must have an available buffer of items that can be filled by the producer and emptied by the consumer.

→ The buffer will reside in the shared memory region.

→ They need to be synchronized: consumer doesn't try to consume an item that has not yet been produced.

* 2 types of buffers can be used:
* Unbounded buffer: 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.
* Bounded buffer: assumes a fixed buffer size. The consumer must wait if the buffer is empty, while the producer must wait if the buffer is full.

**VI. IPC in Message-Passing Systems**

* Message passing provides a mechanism to allow processes to communicate and to synchronize their actions without sharing the same address space

→ Useful in distributed systems, where the processes may reside on different computers connected to the same network.

* Messages can be either fixed or variable in size
* If only fixed-size: straight-foward system-level implementation, but makes programming more difficult
* Conversely for variable-sized messages
* Communication link: for send() / receive() operations
* Direct or indirect
* Synchronous or asynchronous
* Automatic or explicit buffering

1. Naming

* Processes need a way to refer to each other -> use either direct or indirect communication
* Direct communication: explicitly name the recipient or sender of the communication
* Communication link has the following properties:
* A link is established automatically between every pair of processes that want to communicate.
* A link is associated with exactly two processes.
* Between each pair of processes, there exists exactly one link.

→ Symmetry in addressing: both sender and receiver must name the other to communicate

→ Asymmetry: only the sender names the recipient

* Disadvantages: the limited modularity of the resulting process definitions
* Indirect communication: the messages are sent/received from mailboxes/ports
* Mailbox:
* Viewed as a place to place/remove messages by processes
* Has a unique identification
* If owned by a process -> unique owner -> no confusion about receiver of the message in the mailbox

→ Process terminates -> mailbox disappears

* If owned by OS: has an existence of its own -> independent -> requires a mechanism by OS that allow processes to:

++ Create new mailboxes.

++ Send/receive messages through the mailbox.

++ Delete a mailbox.

* Communication link:
* A link is established between a pair of processes only if both members of the pair have a shared mailbox
* A link may be associated with more than two processes
* Between each pair of communicating processes, a number of different links may exist, with each link corresponding to one mailbox

1. Synchronization

* Message passing may be either blocking or nonblocking -> synchronous/asynchronous
* Concepts:
* Blocking send: the sending process is blocked until the message is received by the receiving process or by the mailbox.
* Nonblocking send: the sending process sends the message and resumes operation.
* Blocking receive: the receiver blocks until a message is available.
* Nonblocking receive: the receiver retrieves either a valid message or a null.

1. Buffering

* Messages exchanged by communication processes reside in a temporary queue.
* Zero capacity: the queue has a maximum length of zero -> the link cannot have any messages waiting in it -> the sender must block until the recipient receives the message.
* Bounded capacity:
* The queue has finite length n -> at most n messages can reside in it.
* If the queue is not full when a new message is sent -> message is placed in the queue -> the sender can continue execution without waiting.
* If full -> the sender must block until space is available in the queue.
* Unbounded capacity: queue’s length is potentially infinite -> any number of messages can wait in it -> the sender never blocks.

**VII. Examples of IPC Systems**

1. POSIX Shared Memory

* Several IPC mechanisms are available for POSIX systems, including shared memory and message passing. Here, we explore the POSIX API for shared memory.
* POSIX shared memory is organized using memory-mapped files, which associate the region of shared memory with a file. A process must first create a shared-memory object using the shm open() system call, as follows:

fd = shm open(name, O CREAT | O RDWR, 0666);

* The first parameter specifies the name of the shared-memory object. Processes that wish to access this shared memory must refer to the object by this name. The subsequent parameters specify that the shared-memory object is to be created if it does not yet exist (O CREAT) and that the object is open for reading and writing (O RDWR). The last parameter establishes the file-access permissions of the shared-memory object. A successful call to shm open() returns an integer file descriptor for the shared-memory object.
* Once the object is established, the ftruncate() function is used to configure the size of the object in bytes. The call

ftruncate(fd, 4096);

→ sets the size of the object to 4,096 bytes.

* Finally, the mmap() function establishes a memory-mapped file containing the shared-memory object. It also returns a pointer to the memory-mapped file that is used for accessing the shared-memory object.

1. Mach Message Passing

* The Mach kernel supports the creation and destruction of multiple tasks, which are similar to processes but have multiple threads of control and fewer associated resources. Most communication in Mach—including all intertask communication—is carried out by messages. Messages are sent to, and received from, mailboxes, which are called ports in Mach.
* Associated with each port is a collection of port rights that identify the capabilities necessary for a task to interact with the port.
* Each task also has access to a bootstrap port, which allows a task to register a port it has created with a system-wide bootstrap server.
* Mach messages contain the following two fields:
* A fixed-size message header containing metadata about the message, including the size of the message as well as source and destination ports. Commonly, the sending thread expects a reply, so the port name of the source is passed on to the receiving task, which can use it as a “return address” in sending a reply.
* A variable-sized body containing data.
* Messages may be either simple or complex. A simple message contains ordinary, unstructured user data that are not interpreted by the kernel. A complex message may contain pointers to memory locations containing data (known as “out-of-line” data) or may also be used for transferring port rights to another task.

1. Windows

* The message-passing facility in Windows is called the advanced local procedure call (ALPC) facility. It is used for communication between two processes on the same machine. It is similar to the standard remote procedure call (RPC) mechanism that is widely used, but it is optimized for and specific to Windows. Like Mach, Windows uses a port object to establish and maintain a connection between two processes. Windows uses two types of ports: connection ports and communication ports.
* When an ALPC channel is created, one of three message-passing techniques is chosen:
* For small messages (up to 256 bytes), the port’s message queue is used as intermediate storage, and the messages are copied from one process to the other.
* Larger messages must be passed through a section object, which is a region of shared memory associated with the channel.
* When the amount of data is too large to fit into a section object, an API is available that allows server processes to read and write directly into the address space of a client.

1. Pipes

* A pipe acts as a conduit allowing two processes to communicate. Pipes were one of the first IPC mechanisms in early UNIX systems. In implementing a pipe, four issues must be considered:
* Does the pipe allow bidirectional communication, or is communication unidirectional?
* If two-way communication is allowed, is it half duplex (data can travel only one way at a time) or full duplex (data can travel in both directions at the same time)?
* Must a relationship (such as parent–child) exist between the communicating processes?
* Can the pipes communicate over a network, or must the communicating processes reside on the same machine?
* Ordinary Pipes
* Ordinary pipes allow two processes to communicate in standard producer– consumer fashion: the producer writes to one end of the pipe (the write end) and the consumer reads from the other end (the read end).
* Ordinary pipes on Windows systems are termed anonymous pipes, and they behave similarly to their UNIX counterparts: they are unidirectional and employ parent–child relationships between the communicating processes
* Named Pipes
* Named pipes provide a much more powerful communication tool.
* Named pipes are referred to as FIFOs in UNIX systems.
* Named pipes on Windows systems provide a richer communication mechanism than their UNIX counterparts.

**VIII. Communication in Client-Server Systems**

1. Sockets

* A socket is defined as an endpoint for communication.
* Servers implementing specific services (such as SSH, FTP, and HTTP) listen to well-known ports (an SSH server listens to port 22; an FTP server listens to port 21; and a web, or HTTP, server listens to port 80). All ports below 1024 are considered well known and are used to implement standard services.
* Connection-oriented (TCP) sockets are implemented with the Socket class.
* Connectionless (UDP) sockets use the DatagramSocket class.
* The IP address 127.0.0.1 is a special IP address known as the loopback.

1. Remote Procedure Calls

* A port in this context is simply a number included at the start of a message packet.
* The RPC system hides the details that allow communication to take place by providing a stub on the client side. This stub locates the port on the server and marshals the parameters.
* On Windows systems, stub code is compiled from a specification written in the Microsoft Interface Definitio Language (MIDL), which is used for defining the interfaces between client and server programs.
* Consider the representation of 32-bit integers. Some systems (known as big-endian) store the most significant byte first, while other systems (known as little-endian) store the least significant byte first.
* To resolve differences like this, many RPC systems define a machine-independent representation of data. One such representation is known as external data representation (XDR).

**IX. Summary**

* A process is a program in execution, and the status of the current activity of a process is represented by the program counter, as well as other registers.
* The layout of a process in memory is represented by four different sections: (1) text, (2) data, (3) heap, and (4) stack.
* As a process executes, it changes state. There are four general states of a process: (1) ready, (2) running, (3) waiting, and (4) terminated.
* A process control block (PCB) is the kernel data structure that represents a process in an operating system.
* The role of the process scheduler is to select an available process to run on a CPU.
* An operating system performs a context switch when it switches from running one process to running another.
* The fork() and CreateProcess() system calls are used to create processes on UNIX and Windows systems, respectively.
* When shared memory is used for communication between processes, two (or more) processes share the same region of memory. POSIX provides an API for shared memory.
* Two processes may communicate by exchanging messages with one another using message passing. The Mach operating system uses message passing as its primary form of interprocess communication. Windows provides a form of message passing as well.
* A pipe provides a conduit for two processes to communicate. There are two forms of pipes, ordinary and named. Ordinary pipes are designed for communication between processes that have a parent–child relationship. Named pipes are more general and allow several processes to communicate.
* UNIX systems provide ordinary pipes through the pipe() system call. Ordinary pipes have a read end and a write end. A parent process can, for example, send data to the pipe using its write end, and the child process can read it from its read end. Named pipes in UNIX are termed FIFOs.
* Windows systems also provide two forms of pipes—anonymous and named pipes. Anonymous pipes are similar to UNIX ordinary pipes. They are unidirectional and employ parent–child relationships between the communicating processes. Named pipes offer a richer form of interprocess communication than the UNIX counterpart, FIFOs.
* Two common forms of client–server communication are sockets and remote procedure calls (RPCs). Sockets allow two processes on different machines to communicate over a network. RPCs abstract the concept of function (procedure) calls in such a way that a function can be invoked on another process that may reside on a separate computer.
* The Android operating system uses RPCs as a form of interprocess communication using its binder framework.