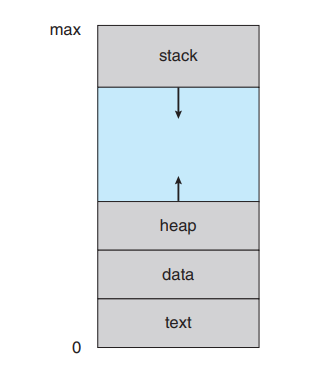
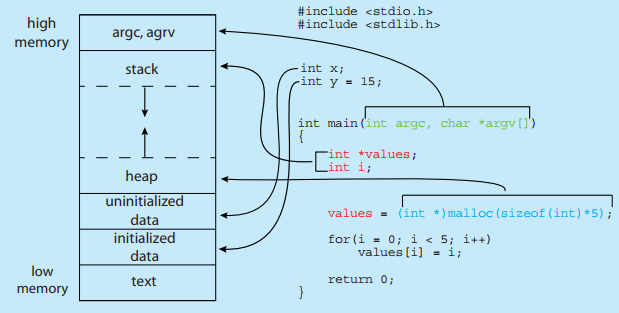
Chapter 3: Processes

3.1 Process Concept

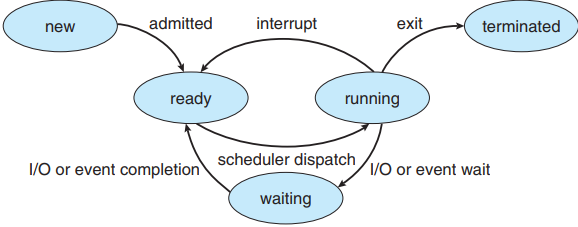
3.1.1 The Process

* The status of a process is represented by the value of program counter.
* The memory layout of a process:
* Text section: the executable code
* Data section: global variables
* Heap section: memory that is dynamically allocated during program run time
* Stack section: temporary data storage when invoking functions (such as function parameters, return addresses, and local variables.)

3.1.2 Process State

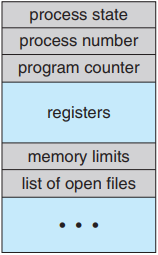
* New: The process is being created
* Running: Instructions are being executed.
* Waiting: The process is waiting for some event to occur (such as an I/O

completion or reception of a signal)

* Ready: The process is waiting to be assigned to a processor
* Terminated: The process has finished execution.

3.1.3 Process Control Block

Each process is represented in the operating system by a process control block (PCB)—also called a task control block. A PCB contains:

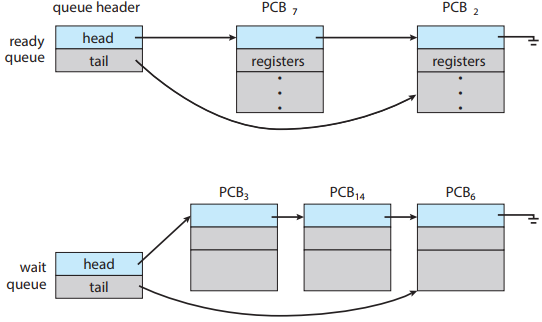
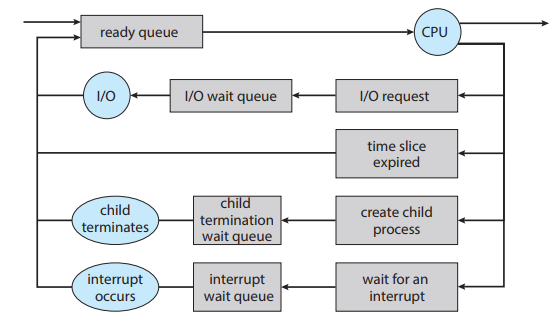
* Process state. The state may be new, ready, running, waiting, halted, and so on.
* Program counter. The counter indicates the address of the next instruction to be executed for this process.
* CPU registers. The registers vary in number and type, depending on the computer architecture. They include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

3.1.4 Threads

* A process is a program that performs a single thread of execution
* Multiple threads of execution help to perform more than one task at a time.

3.2 Process Scheduling

3.2.1 Scheduling Queues

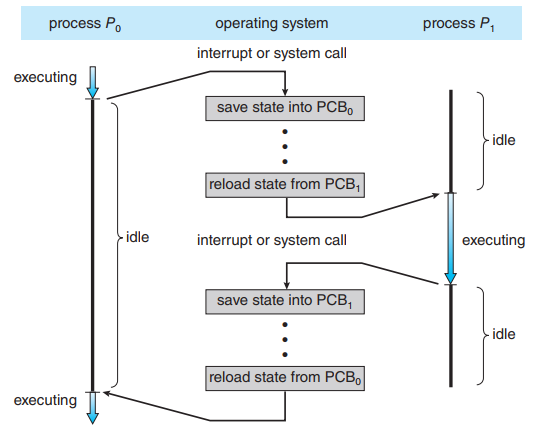
* As processes enter the system, they are put into a ready queue, where they are ready and waiting to execute on a CPU’s core This queue is generally stored as a linked list; a ready-queue header contains pointers to the first PCB in the list, and each PCB includes a pointer field that points to the next PCB in the ready queue.
* Processes that are waiting for a certain event to occur — such as completion of I/O — are placed in a wait queue
* Once the process is allocated a CPU core 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 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

3.2.2 CPU Scheduling

* The role of the CPU scheduler is to select from among the processes that are in the ready queue and allocate a CPU core to one of them
* Swapping’s idea is that sometimes it can be advantageous to remove a process from memory (and from active contention for the CPU) and thus reduce the degree of multiprogramming.

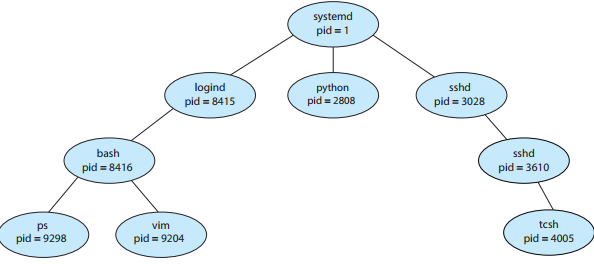
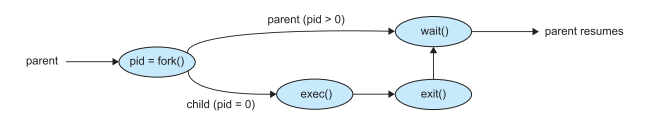
3.2.3 Context Switch

* Switching the CPU core 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:



3.3 Operations on Processes

3.3.1 Process Creation

* Most operating systems identify processes according to a unique process identifie (or pid), which is typically an integer number.
* The systemd process (which always has a pid of 1) serves as the root parent process for all user processes, and is the first user process created when the system boots. It has 2 child processes: logind and sshd.
* The logind process is responsible for managing clients that directly log onto the system.
* The sshd process is responsible for managing clients that connect to the system by using ssh.
* A new process is created by the fork() system call.
* The parent waits for the child process to complete with the wait() system call. When the child process completes (by either implicitly or explicitly invoking exit()), the parent process resumes from the call to wait(), where it completes using the exit() system call.

3.3.2 Process Termination

* A process terminates when it finishes executing its final statement and asks the operating system to delete it by using the exit() system call. All the resources of the process —including physical and virtual memory, open files, and I/O buffers—are deallocated and reclaimed by the operating system
* A process can cause the termination of another process via an appropriate system call.
* A parent may terminate the execution of one of its children for a variety of reasons, such as these:
  + The child has exceeded its usage of some of the resources that it has been allocated. (To determine whether this has occurred, the parent must have a mechanism to inspect the state of its children.)
  + The task assigned to the child is no longer required.
  + The parent is exiting, and the operating system does not allow a child to continue if its parent terminates.

3.4 Interprocess Communication

* There are several reasons for providing an environment that allows process cooperation: Information sharing, Computation speedup, Modularity
* Cooperating processes require an interprocess communication (IPC) mechanism that allow them to exchange data – send and receive data. There are two fundamental models of IPC: shared memory and message passing.

|  |  |
| --- | --- |
| Shared memory:   * Processes can then exchange information by reading   and writing data to the shared region   * Faster than message passing * All accesses are treated as routine memory accesses, and no assistance from the kernel is required. | Message passing:   * Communication takes place by means of messages exchanged between the cooperating processes * Useful for exchanging smaller amounts of data, because no conflicts need be avoided * Easier to implement in a distributed system than shared memory |

3.5 IPC in Shared-Memory Systems

* Normally, the operating system tries to prevent one process from accessing another process’s memory. Shared memory requires that two or more processes agree to remove the restriction. They can then exchange information by reading and writing data in the shared areas. The form of the data and location are determined by these processes and are not under the operating system’s control. The processes are also responsible for ensuring that they are not writing to the same location simultaneously.

3.6 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. It is particularly useful in a distributed environment, where the communicating processes may reside on different computers connected by a network
* If processes P and Q want to communicate, they must send messages to and receive messages from each other: a communication link must exist between them. Here are several methods for logically implementing a link and the send()/receive() operations:
* Direct or indirect communication
* Synchronous or asynchronous communication
* Automatic or explicit buffering

3.6.1 Naming

* Direct communication: the send() and receive() primitives are defined as:
* send(P, message)—Send a message to process P
* receive(Q, message)—Receive a message from process Q
* A communication link in this scheme has the following properties:
  + A link is established automatically between every pair of processes that want to communicate. The processes need to know only each other’s identity to communicate
  + A link is associated with exactly two processes
  + Between each pair of processes, there exists exactly one link
* Indirect communication: the messages are sent to and received from mailboxes, or ports. The send() and receive() primitives are defined as follows:
* send(A, message)—Send a message to mailbox A.
* receive(A, message)—Receive a message from mailbox A
* In this scheme, a communication link has the following properties:
* A link is established between a pair of processes only if both members of the pair have a shared mailbox.
* A link may be associated with more than two processes.
* Between each pair of communicating processes, a number of different links may exist, with each link corresponding to one mailbox.

3.6.2 Synchronization

* Message passing may be either blocking or nonblocking— also known as synchronous and asynchronous.
* Blocking send. The sending process is blocked until the message is received by the receiving process or by the mailbox.
* Nonblocking send. The sending process sends the message and resumes operation.
* Blocking receive. The receiver blocks until a message is available.
* Nonblocking receive. The receiver retrieves either a valid message or a null.

3.6.3 Buffering

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

3.7 Examples of IPC Systems

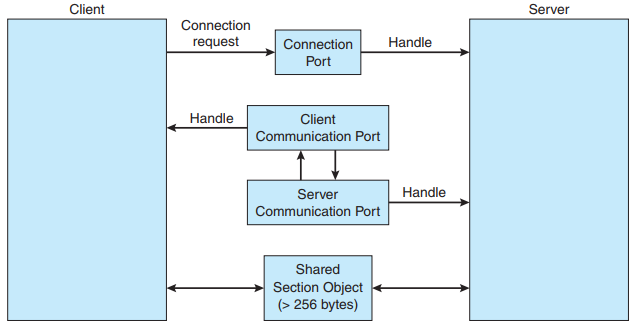
3.7.1 POSIX Shared Memory

* 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.
* Once the object is established, the ftruncate() function is used to configure the size of the object in 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.

3.7.2 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.
* When a task is created, two special ports—the Task Self port and the Notify port—are also created. The kernel receives rights to the Task Self port, which allows a task to send messages to the kernel. The kernel can send notification of event occurrences to a task’s Notify port.

3.7.3 Windows

* The message-passing facility in Windows is called the advanced local procedure call (ALPC) facility.
* Windows uses two types of ports: connection ports and communication ports.
* Server processes publish connection-port objects that are visible to all processes. When a client wants services from a subsystem, it opens a handle to the server’s connection-port object and sends a connection request to that port. The server then creates a channel and returns a handle to the client. The channel consists of a pair of private communication ports: one for client–server messages, the other for server–client messages. Additionally, communication channels support a callback mechanism that allows the client and server to accept requests when they would normally be expecting a reply. The client has to decide when it sets up the channel whether it will need to send a large message. If the client determines that it does want to send large messages, it asks for a section object to be created. Similarly, if the server decides that replies will be large, it creates a section object. So that the section object can be used, a small message is sent that contains a pointer and size information about the section object.

3.7.4 Pipes

* A pipe acts as a conduit allowing two processes to communicate. In implementing a pipe, four issues must be considered:

1. Does the pipe allow bidirectional communication, or is communication unidirectional?

2. 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)?

3. Must a relationship (such as parent–child) exist between the communicating processes?

4. Can the pipes communicate over a network, or must the communicating processes reside on the same machine?

3.7.4.1 Ordinary pipes (Unnamed, Anonymous)

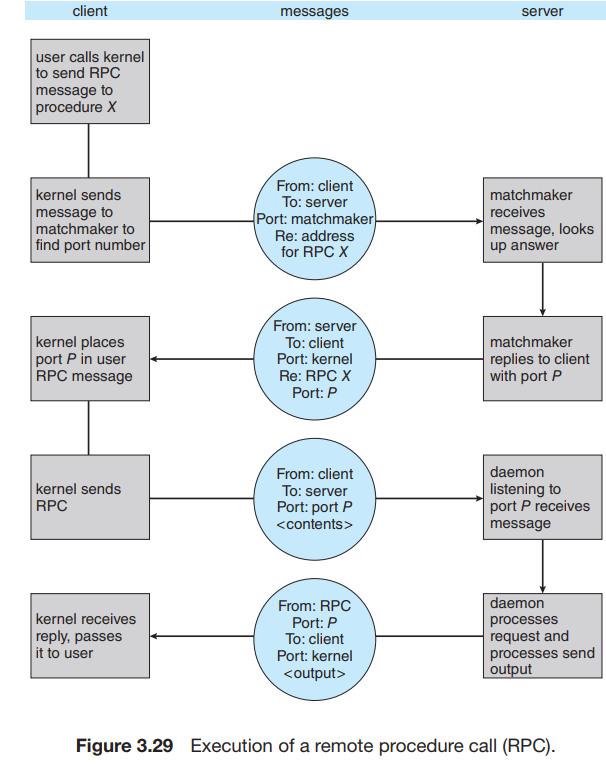
* Ordinary pipes allow two processes to communicate in standard producer– consumer fashion: the producer writes to one end of the pipe (the write end) and the consumer reads from the other end (the read end). As a result, ordinary pipes are unidirectional, allowing only one-way communication. If two-way communication is required, two pipes must be used, with each pipe sending data in a different direction.

3.7.4.2 Named pipes

* Named pipes provide a much more powerful communication tool. Communication can be bidirectional, and no parent–child relationship is required. Once a named pipe is established, several processes can use it for communication.

3.8 Communication in Client-Server Systems

3.8.1 Sockets 3.8.2 Remote Procedure Calls



* A socket is defined as an endpoint for communication. A pair of processes communicating over a network employs a pair of sockets—one for each process. A socket is identified by an IP address concatenated with a port number. In general, sockets use a client–server architecture. The server waits for incoming client requests by listening to a specified port. Once a request is received, the server accepts a connection from the client socket to complete the connection.
* Java provides three different types of sockets. Connection-oriented (TCP) sockets are implemented with the Socket class. Connectionless (UDP) sockets use the DatagramSocket class. Finally, the MulticastSocket class is a subclass of the DatagramSocket class. A multicast socket allows data to be sent to multiple recipients.