Operating Systems Notes

June 8, 2025

Contents

1 Process concept

Introduction

A *process* is a program in execution. A process will need certain resources—such as CPU time, memory, files, and I/O devices—to accomplish its task. These resources are typically allocated to the process while it is executing.

A process is the unit of work in most systems. Systems consist of a collection of processes: operating-system processes execute system code, and user processes execute user code. All these processes may execute concurrently.

Modern operating systems support processes having multiple *threads* of control. On systems with multiple hardware processing cores, these threads can run in parallel.

One of the most important aspects of an operating system is how it schedules threads onto available processing cores. Several choices for designing CPU schedulers are available to programmers.

Early computers allowed only one program to be executed at a time. This program had complete control of the system and had access to all the system's resources. In contrast, contemporary computer systems allow multiple programs to be loaded into memory and executed concurrently. This evolution required firmer control and more compartmentalization of the various programs; and these needs resulted in the notion of a **process**, which is a program in execution. A process is the unit of work in a modern computing system.

Although the main concern of the operating system is the execution of user programs, it also needs to take care of various system tasks that are best done in user space, rather than within the kernel. A system therefore consists of a collection of processes, some executing user code, others executing operating system code. Potentially, all these processes can execute concurrently, with the CPU (or CPUs) multiplexed among them.

Chapter objectives

- Identify the separate components of a process and illustrate how they are represented and scheduled in an operating system.
- Describe how processes are created and terminated in an operating system, including developing programs using the appropriate system calls that perform these operations.
- Describe and contrast interprocess communication using shared memory and message passing.
- Design programs that use pipes and POSIX shared memory to perform interprocess communication.
- Describe client-server communication using sockets and remote procedure calls.
- Design kernel modules that interact with the Linux operating system.

The process

Informally, as mentioned earlier, a process is a program in execution. The status of the current activity of a process is represented by the value of the **program counter** and the contents of the processor's registers. The memory layout of a process is typically divided into multiple sections. These sections include:

- **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)

Notice that the sizes of the text and data sections are fixed, as their sizes do not change during program run time. However, the stack and heap sections can shrink and grow dynamically during program execution. Each time a function is called, an **activation record** containing function parameters, local variables, and the return address is pushed onto the stack; when control is returned from the function, the activation record is popped from the stack. Similarly, the heap will grow as memory is dynamically allocated, and will shrink when memory is returned to the system. Although the stack and heap sections grow *toward* one another, the operating system must ensure they do not *overlap* one another.

A program by itself is not a process. A program is a *passive* entity, such as a file containing a list of instructions stored on disk (often called an **executable file**). In contrast, a process is an *active* entity, with a program counter specifying the next instruction to execute and a set of associated resources. A program becomes a process when an executable file is loaded into memory. Two common techniques for loading executable files are double-clicking an icon representing the executable file and entering the name of the executable file on the command line (as in `prog.exe` or `a.out`).

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. It is also common to have a process that spawns many processes as it runs.

A process can itself be an execution environment for other code. The Java programming environment provides a good example. In most circumstances, an executable Java program is executed within the Java virtual machine (JVM). The JVM executes as a process that interprets the loaded Java code and takes actions (via native machine instructions) on behalf of that code. For example, to run the compiled Java program `Program.class`, we would enter `java Program`. The command `java` runs the JVM as an ordinary process, which in turns executes the Java program `Program` in the virtual machine.

Process state

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

These names are arbitrary, and they vary across operating systems. The states that they represent are found on all systems, however. Certain operating systems also more finely delineate process states. It is important to realize that only one process can be *running* on any processor core at any instant. Many processes may be *ready* and *waiting*, however.

Process control block

Each process is represented in the operating system by a **process control block** (*PCB*)—also called a **task control block**. It contains many pieces of information associated with a specific process, including these:

- **Process state.** The state may be new, ready, running, waiting, halted, and so on.
- **Program counter.** The counter indicates the address of the next instruction to be executed for this process.
- **CPU registers.** The registers vary in number and type, depending on the computer architecture. They include accumulators, index registers, stack pointers, and general-purpose registers, plus any condition-code information. 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.

In brief, the PCB simply serves as the repository for all the data needed to start, or restart, a process, along with some accounting data.

Threads

The process model discussed so far has implied that a process is a program that performs a single **thread** of execution. For example, when a process is running a word-processor program, a single thread of instructions is being executed. This single thread of control allows the process to perform only one task at a time. Thus, the user cannot simultaneously type in characters and run the spell checker. 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. This feature is especially beneficial on multicore systems, where multiple threads can run in parallel. A multithreaded word processor could, for example, assign one thread to manage user input while another thread runs the spell checker. On systems that support threads, the PCB is expanded to include information for each thread. Other changes throughout the system are also needed to support threads.

Section glossary

Term	Definition
process	A program loaded into memory and executing.
job	A set of commands or processes executed by a batch system.
user programs	User-level programs, as opposed to system programs.
task	A process, a thread activity, or, generally, a unit of computation on a computer.
program counter	A CPU register indicating the main memory location of the next instruction to load and execute.
text section	The executable code of a program or process.
data section	The data part of a program or process; it contains global variables.
heap section	The section of process memory that is dynamically allocated during process run time; it stores temporary variables.
stack section	The section of process memory that contains the stack; it contains activation records and other temporary data.
activation record	A record created when a function or subroutine is called; added to the stack by the call and removed when the call returns. Contains function parameters, local variables, and the return address.
executable file	A file containing a program that is ready to be loaded into memory and executed.
state	The condition of a process, including its current activity as well as its associated memory and disk contents.
process control block	A per-process kernel data structure containing many pieces of information associated with the process.
task control block	A per-process kernel data structure containing many pieces of information associated with the process.
thread	A process control structure that is an execution location. A process with a single thread executes only one task at a time, while a multithreaded process can execute a task per thread.

2 Process scheduling

The objective of multiprogramming is to have some process running at all times so as to maximize CPU utilization. The objective of time sharing is to switch a CPU core 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 a core. Each CPU core can run one process at a time. For a system with a single CPU core, there will never be more than one process running at a time, whereas a multicore system can run multiple processes at one time. If there are more processes than cores, excess processes will have to wait until a core is free and can be rescheduled. The number of processes currently in memory is known as the **degree of multiprogramming**.

Balancing the objectives of multiprogramming and time sharing also requires taking the general behavior of a process into account. 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.

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.

The system also includes other queues. When a process is allocated a CPU core, it executes for a while and eventually terminates, is interrupted, or waits for the occurrence of a particular event, such as the completion of an I/O request. Suppose the process makes an I/O request to a device such as a disk. Since devices run significantly slower than processors, the process will have to wait for the I/O to become available. Processes that are waiting for a certain event to occur — such as completion of I/O — are placed in a **wait queue**.

A common representation of process scheduling is a **queueing diagram**. Two types of queues are present: the ready queue and a set of wait queues. The circles represent the resources that serve the queues, and the arrows indicate the flow of processes in the system.

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

In the first two cases, the process eventually switches from the waiting state to the ready state and is then 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.

CPU scheduling

A process migrates among the ready queue and various wait queues throughout its lifetime. 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. The CPU scheduler must select a new process for the CPU frequently. An I/O-bound process may execute for only a few milliseconds before waiting for an I/O request. Although a CPU-bound process will require a CPU core for longer durations, the scheduler is unlikely to grant the core to a process for an extended period. Instead, it is likely designed to forcibly remove the CPU from a process and schedule another process to run. Therefore, the CPU scheduler executes at least once every 100 milliseconds, although typically much more frequently.

Some operating systems have an intermediate form of scheduling, known as **swapping**, whose key 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. Later, the process can be reintroduced into memory, and its execution can be continued where it left off. This scheme is known as *swapping* because a process can be "swapped out" from memory to disk, where its current status is saved, and later "swapped in" from disk back to memory, where its status is restored. Swapping is typically only necessary when memory has been overcommitted and must be freed up.

Context switch

Interrupts cause the operating system to change a CPU core from its current task and to run a kernel routine. Such operations happen frequently on general-purpose systems. When an interrupt occurs, the system needs to save the current **context** of the process running on the CPU core so that it can restore that context when its processing is done, essentially suspending the process and then resuming it. The context is represented in the PCB of the process. It includes the value of the CPU registers, the process state, and memory-management information. Generically, we perform a **state save** of the current state of the CPU core, be it in kernel or user mode, and then a **state restore** to resume operations.

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**. 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 time is pure overhead, because the system does no useful work while switching. Switching speed varies from machine to machine, depending on the memory speed, the number of registers that must be copied, and the existence of special instructions (such as a single instruction to load or store all registers). A typical speed is a several microseconds.

Context-switch times are highly dependent on hardware support. For instance, some processors provide multiple sets of registers. A context switch here simply requires changing the pointer to the current register set. Of course, if there are more active processes than there are register sets, the system resorts to copying register data to and from memory, as before. Also, the more complex the operating system, the greater the amount of work that must be done during a context switch. Advanced memory-management techniques may require that extra data be switched with each context. For instance, the address space of the current process must be preserved as the space of the next task is prepared for use. How the address space is preserved, and what amount of work is needed to preserve it, depend on the memory-management method of the operating system.

Section glossary

Term	Definition
process scheduler	A scheduler that selects an available process (possibly from a set of several processes) for execution on a CPU.
degree of multiprogramming	The number of processes in memory.
parent	In a tree data structure, a node that has one or more nodes connected below it.
children	In a tree data structure, nodes connected below another node.
siblings	In a tree data structure, child nodes of the same parent.
I/O-bound process	A process that spends more of its time doing I/O than doing computations
CPU-bound process	A process that spends more time executing on CPU than it does performing I/O.
ready queue	The set of processes ready and waiting to execute.
wait queue	In process scheduling, a queue holding processes waiting for an event to occur before they need to be put on CPU.
dispatched	Selected by the process scheduler to be executed next.
CPU scheduler	Kernel routine that selects a thread from the threads that are ready to execute and allocates a core to that thread.
swapping	Moving a process between main memory and a backing store. A process may be swapped out to free main memory temporarily and then swapped back in to continue execution.
context	When describing a process, the state of its execution, including the contents of registers, its program counter, and its memory context, including its stack and heap.
state save	Copying a process's context to save its state in order to pause its execution in preparation for putting another process on the CPU.
state restore	Copying a process's context from its saved location to the CPU registers in preparation for continuing the process's execution.
context switch	The switching of the CPU from one process or thread to another; requires performing a state save of the current process or thread and a state restore of the other.
foreground	Describes a process or thread that is interactive (has input directed to it), such as a window currently selected as active or a terminal window currently selected to receive input.
background	Describes a process or thread that is not currently interactive (has no interactive input directed to it), such as one not currently being used by a user. In the Grand Central Dispatch Apple OS scheduler, the scheduling class representing tasks that are not time sensitive and are not visible to the user.
split-screen	Running multiple foreground processes (e.g., on an iPad) but splitting the screen among the processes.
service	A software entity running on one or more machines and providing a particular type of function to calling clients. In Android, an application component with no user interface; it runs in the background while executing long-running operations or performing work for remote processes.

3 Operations on processes

The processes in most systems can execute concurrently, and they may be created and deleted dynamically. Thus, these systems must provide a mechanism for process creation and termination. In this section, we explore the mechanisms involved in creating processes and illustrate process creation on UNIX and Windows systems.

Process creation

During the course of execution, a process may create several new processes. As mentioned earlier, the creating process is called a parent process, and the new processes are called the children of that process. Each of these new processes may in turn create other processes, forming a **tree** of processes.

Most operating systems (including UNIX, Linux, and Windows) identify processes according to a unique **process identifier** (or **pid**), which is typically an integer number. The pid provides a unique value for each process in the system, and it can be used as an index to access various attributes of a process within the kernel.

On UNIX and Linux systems, we can obtain a listing of processes by using the `ps` command. For example, the command `ps -el` will list complete information for all processes currently active in the system. A process tree can be constructed by recursively tracing parent processes all the way to the `systemd` process. (In addition, Linux systems provide the `pstree` command, which displays a tree of all processes in the system.)

In general, when a process creates a child process, that child process will need certain resources (CPU time, memory, files, I/O devices) to accomplish its task. A child process may be able to obtain its resources directly from the operating system, or it may be constrained to a subset of the resources of the parent process. The parent may have to partition its resources among its children, or it may be able to share some resources (such as memory or files) among several of its children. Restricting a child process to a subset of the parent's resources prevents any process from overloading the system by creating too many child processes.

In addition to supplying various physical and logical resources, the parent process may pass along initialization data (input) to the child process. For example, consider a process whose function is to display the contents of a file—say, `hw1.c`—on the screen of a terminal. When the process is created, it will get, as an input from its parent process, the name of the file `hw1.c`. Using that file name, it will open the file and write the contents out. It may also get the name of the output device. Alternatively, some operating systems pass resources to child processes. On such a system, the new process may get two open files, `hw1.c` and the terminal device, and may simply transfer the datum between the two.

When a process creates a new process, two possibilities for execution exist:

1. The parent continues to execute concurrently with its children.
2. The parent waits until some or all of its children have terminated.

There are also two address-space possibilities for the new process:

1. The child process is a duplicate of the parent process (it has the same program and data as the parent).
2. The child process has a new program loaded into it.

To illustrate these differences, let's first consider the UNIX operating system. In UNIX, as we've seen, each process is identified by its process identifier, which is a unique integer. A new process is created by the `fork()` system call. The new process consists of a copy of the address space of the original process. This mechanism allows the parent process to communicate easily with its child process. Both processes (the parent and the child) continue execution at the instruction after the `fork()`, with one difference: the return code for the `fork()` is zero for the new (child) process, whereas the (nonzero) process identifier of the child is returned to the parent.

After a `fork()` system call, one of the two processes typically uses the `exec()` system call to replace the process's memory space with a new program. The `exec()` system call loads a binary file into memory (replacing the memory image of the program containing the `exec()` system call) and starts its execution. In this manner, the two processes are able to communicate and then go their separate ways. The parent can then create more children; or, if it has nothing else to do while the child runs, it can issue a `wait()` system call to move itself off the ready queue until the termination of the child. Because the call to `exec()` overlays the process's address space with a new program, `exec()` does not return control unless an error occurs.

The C program illustrates the UNIX system calls previously described. We now have two different processes running copies of the same program. The only difference is that the value of the variable `pid` for the child process is zero, while that for the parent is an integer value greater than zero (in fact, it is the actual pid of the child process). The child process inherits privileges and scheduling attributes from the parent, as well certain resources, such as open files. The child process then overlays its address space with the UNIX command `/bin/ls` (used to get a directory listing) using the `execlp()` system call (`execlp()` is one of many different versions of the `exec()` system call). The parent waits for the child process to complete with the `wait()` system call. When the child process completes (by either implicitly or explicitly invoking `exit()`), the parent process resumes from the call to `wait()`, where it completes using the `exit()` system call.

Of course, there is nothing to prevent the child from **not** invoking `exec()` and instead continuing to execute as a copy of the parent process. In this scenario, the parent and child are concurrent processes running the same code instructions. Because the child is a copy of the parent, each process has its own copy of any data.

As an alternative example, we next consider process creation in Windows. Processes are created in the Windows API using the `CreateProcess()` function, which is similar to `fork()` in that a parent creates a new child process. However, whereas `fork()` has the child process inheriting the address space of its parent, `CreateProcess()` requires loading a specified program into the address space of the child process at process creation. Furthermore, whereas `fork()` is passed no parameters, `CreateProcess()` expects no fewer than ten parameters.

The two parameters passed to the `CreateProcess()` function are instances of the `STARTUPINFO` and `PROCESS_INFORMATION` structures. `STARTUPINFO` specifies many properties of the new process, such as window size and appearance and handles to standard input and output files. The `PROCESS_INFORMATION` structure contains a handle and the identifiers to the newly created process and its thread. We invoke the `ZeroMemory()` function to allocate memory for each of these structures before proceeding with `CreateProcess()`.

The first two parameters passed to `CreateProcess()` are the application name and command-line parameters. If the application name is `NULL` (as it is in this case), the command-line parameter specifies the application to load. In this instance, we are loading the Microsoft Windows `mspaint.exe` application. Beyond these two initial parameters, we use the default parameters for

inheriting process and thread handles as well as specifying that there will be no creation flags. We also use the parent's existing environment block and starting directory. Last, we provide two pointers to the `STARTUPINFO` and `PROCESS_INFORMATION` structures created at the beginning of the program. In the parent process waits for the child to complete by invoking the `wait()` system call. The equivalent of this in Windows is `WaitForSingleObject()`, which is passed a handle of the child process—`pi.hProcess`—and waits for this process to complete. Once the child process exits, control returns from the `WaitForSingleObject()` function in the parent process.

Process termination

A process terminates when it finishes executing its final statement and asks the operating system to delete it by using the `exit()` system call. At that point, the process may return a status value (typically an integer) to its waiting parent process (via the `wait()` system call). All the resources of the process—including physical and virtual memory, open files, and I/O buffers—are deallocated and reclaimed by the operating system.

Termination can occur in other circumstances as well. A process can cause the termination of another process via an appropriate system call (for example, `TerminateProcess()` in Windows). Usually, such a system call can be invoked only by the parent of the process that is to be terminated. Otherwise, a user—or a misbehaving application—could arbitrarily kill another user's processes. Note that a parent needs to know the identities of its children if it is to terminate them. Thus, when one process creates a new process, the identity of the newly created process is passed to the parent.

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.

Some systems do not allow a child to exist if its parent has terminated. In such systems, if a process terminates (either normally or abnormally), then all its children must also be terminated. This phenomenon, referred to as **cascading termination**, is normally initiated by the operating system.

To illustrate process execution and termination, consider that, in Linux and UNIX systems, we can terminate a process by using the `exit()` system call, providing an exit status as a parameter:

```
/* exit with status 1 */
exit(1);
```

In fact, under normal termination, `exit()` will be called either directly (as shown above) or indirectly, as the C run-time library (which is added to UNIX executable files) will include a call to `exit()` by default.

A parent process may wait for the termination of a child process by using the `wait()` system call. The `wait()` system call is passed a parameter that allows the parent to obtain the exit status of the child. This system call also returns the process identifier of the terminated child so that the parent can tell which of its children has terminated:

```
pid_t pid;
int status;

pid = wait(&status);
```

When a process terminates, its resources are deallocated by the operating system. However, its entry in the process table must remain there until the parent calls `wait()`, because the process table contains the process's exit status. A process that has terminated, but whose parent has not yet called `wait()`, is known as a **zombie** process. All processes transition to this state when they terminate, but generally they exist as zombies only briefly. Once the parent calls `wait()`, the process identifier of the zombie process and its entry in the process table are released.

Now consider what would happen if a parent did not invoke `wait()` and instead terminated, thereby leaving its child processes as **orphans**. Traditional UNIX systems addressed this scenario by assigning the `init` process as the new parent to orphan processes. The `init` process periodically invokes `wait()`, thereby allowing the exit status of any orphaned process to be collected and releasing the orphan's process identifier and process-table entry.

Although most Linux systems have replaced `init` with `systemd`, the latter process can still serve the same role, although Linux also allows processes other than `systemd` to inherit orphan processes and manage their termination.

Android process hierarchy

Because of resource constraints such as limited memory, mobile operating systems may have to terminate existing processes to reclaim limited system resources. Rather than terminating an arbitrary process, Android has identified an *importance hierarchy* of processes, and when the system must terminate a process to make resources available for a new, or more important, process, it terminates processes in order of increasing importance. From most to least important, the hierarchy of process classifications is as follows:

- **Foreground process**—The current process visible on the screen, representing the application the user is currently interacting with
- **Visible process**—A process that is not directly visible on the foreground but that is performing an activity that the foreground process is referring to (that is, a process performing an activity whose status is displayed on the foreground process)
- **Service process**—A process that is similar to a background process but is performing an activity that is apparent to the user (such as streaming music)
- **Background process**—A process that may be performing an activity but is not apparent to the user
- **Empty process**—A process that holds no active components associated with any application

If system resources must be reclaimed, Android will first terminate empty processes, followed by background processes, and so forth. Processes are assigned an importance ranking, and Android attempts to assign a process as high a ranking as possible. For example, if a process is providing a service and is also visible, it will be assigned the more-important visible classification.

Furthermore, Android development practices suggest following the guidelines of the process life cycle. When these guidelines are followed, the state of a process will be saved prior to termination and resumed at its saved state if the user navigates back to the application.

Section glossary

Term	Definition
tree	A data structure that can be used to represent data hierarchically; data values in a tree structure are linked through parent-child relationships.
process identifier (pid)	A unique value for each process in the system that can be used as an index to access various attributes of a process within the kernel.
cascading termination	A technique in which, when a process is ended, all of its children are ended as well.
zombie	A process that has terminated but whose parent has not yet called wait() to collect its state and accounting information.
orphan	The child of a parent process that terminates in a system that does not require a terminating parent to cause its children to be terminated.

4 Interprocess communication

Processes executing concurrently in the operating system may be either independent processes or cooperating processes. A process is **independent** if it does not share data with any other processes executing in the system. A process is **cooperating** if it can affect or be affected by the other processes executing in the system. Clearly, any process that shares data with other processes is a cooperating process.

There are several reasons for providing an environment that allows process cooperation:

- **Information sharing.** Since several applications may be interested in the same piece of information (for instance, copying and pasting), we must provide an environment to allow concurrent access to such information.
- **Computation speedup.** If we want a particular task to run faster, we must break it into subtasks, each of which will be executing in parallel with the others. Notice that such a speedup can be achieved only if the computer has multiple processing cores.
- **Modularity.** We may want to construct the system in a modular fashion, dividing the system functions into separate processes or threads, as we discussed in the previous chapter Operating-System Structures.

Cooperating processes require an **interprocess communication** (*IPC*) mechanism that will allow them to exchange data—that is, send data to and receive data from each other. There are two fundamental models of interprocess communication: **shared memory** and **message passing**. In the shared-memory model, a region of memory that is shared by the cooperating processes is established. Processes can then exchange information by reading and writing data to the shared region. In the message-passing model, communication takes place by means of messages exchanged between the cooperating processes.

Both of the models just mentioned are common in operating systems, and many systems implement both. Message passing is useful for exchanging smaller amounts of data, because no conflicts need be avoided. Message passing is also easier to implement in a distributed system than shared memory. (Although there are systems that provide distributed shared memory, we do not consider them in this text.) Shared memory can be faster than message passing, since message-passing systems are typically implemented using system calls and thus require the more time-consuming task of kernel intervention. In shared-memory systems, system calls are required only to establish shared-memory regions. Once shared memory is established, all accesses are treated as routine memory accesses, and no assistance from the kernel is required.

Chapter objectives

- Describe and contrast interprocess communication using shared memory and message passing.
- Design programs that use pipes and POSIX shared memory to perform interprocess communication.
- Describe client-server communication using sockets and remote procedure calls.

Section glossary

Term	Definition
interprocess communication (IPC)	Communication between processes.
shared memory	In interprocess communication, a section of memory shared by multiple processes and used for message passing.
message passing	In interprocess communication, a method of sharing data in which messages are sent and received by processes. Packets of information in predefined formats are moved between processes or between computers.
browser	A process that accepts input in the form of a URL (Uniform Resource Locator), or web address, and displays its contents on a screen.
renderer	A process that contains logic for rendering contents (such as web pages) onto a display.
plug-in	An add-on functionality that expands the primary functionality of a process (e.g., a web browser plug-in that displays a type of content different from what the browser can natively handle).
sandbox	A contained environment (e.g., a virtual machine).

5 IPC in shared-memory systems

Interprocess communication using shared memory requires communicating processes to establish a region of shared memory. Typically, a shared-memory region resides in the address space of the process creating the shared-memory segment. Other processes that wish to communicate using this shared-memory segment must attach it to their address space. Recall that, normally, the operating system tries to prevent one process from accessing another process's memory. Shared memory requires that two or more processes agree to remove this restriction. They can then exchange information by reading and writing data in the shared areas. The form of the data and the location are determined by these processes and are not under the operating system's control. The processes are also responsible for ensuring that they are not writing to the same location simultaneously.

To illustrate the concept of cooperating processes, let's consider the producer-consumer problem, which is a common paradigm for cooperating processes. A **producer** process produces information that is consumed by a **consumer** process. For example, a compiler may produce assembly code that is consumed by an assembler. The assembler, in turn, may produce object modules that are consumed by the loader. The producer-consumer problem also provides a useful metaphor for the client-server paradigm. We generally think of a server as a producer and a client as a consumer. For example, a web server produces (that is, provides) web content such as HTML files and images, which are consumed (that is, read) by the client web browser requesting the resource.

One solution to the producer-consumer problem uses shared memory. To allow producer and consumer processes to run concurrently, we must have available a buffer of items that can be filled by the producer and emptied by the consumer. This buffer will reside in a region of memory that is shared by the producer and consumer processes. A producer can produce one item while the consumer is consuming another item. The producer and consumer must be synchronized, so that the consumer does not try to consume an item that has not yet been produced.

Two types of buffers can be used. 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. 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.

Let's look more closely at how the bounded buffer illustrates interprocess communication using shared memory. The following variables reside in a region of memory shared by the producer and consumer processes:

```
#define BUFFER_SIZE 10

typedef struct {
   . . .
} item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

The shared `buffer` is implemented as a circular array with two logical pointers: `in` and `out`. The variable `in` points to the next free position in the buffer; `out` points to the first full position in the buffer. The buffer is empty when `in == out`; the buffer is full when

```
((in + 1) % BUFFER_SIZE) == out
```

The producer process has a local variable `next_produced` in which the new item to be produced is stored. The consumer process has a local variable `next_consumed` in which the item to be consumed is stored.

```
item next_produced;

while (true) {
     /* produce an item in next_produced */

     while (((in + 1) % BUFFER_SIZE) == out)
       ; /* do nothing */

     buffer[in] = next_produced;
     in = (in + 1) % BUFFER_SIZE;
}
```

```
item next_consumed;

while (true) {
     while (in == out)
       ; /* do nothing */

     next_consumed = buffer[out];
     out = (out + 1) % BUFFER_SIZE;

     /* consume the item in next_consumed */
}
```

This scheme allows at most `BUFFER_SIZE - 1` items in the buffer at the same time. We leave it as an exercise for you to provide a solution in which `BUFFER_SIZE` items can be in the buffer at the same time. In Section POSIX shared memory, we illustrate the POSIX API for shared memory.

One issue this illustration does not address concerns the situation in which both the producer process and the consumer process attempt to access the shared buffer concurrently. In chapter Synchronization Tools and chapter Synchronization Examples, we discuss how synchronization among cooperating processes can be implemented effectively in a shared-memory environment.

Section glossary

Term	Definition
producer	A process role in which the process produces information that is consumed by a consumer process.
consumer	A process role in which the process consumes information produced by a producer process.
unbounded buffer	A buffer with no practical limit on its memory size.
bounded buffer	A buffer with a fixed size.

6 IPC in message-passing systems

In Section 3.5, we showed how cooperating processes can communicate in a shared-memory environment. The scheme requires that these processes share a region of memory and that the code for accessing and manipulating the shared memory be written explicitly by the application programmer. Another way to achieve the same effect is for the operating system to provide the means for cooperating processes to communicate with each other via a message-passing facility.

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. For example, an Internet `chat` program could be designed so that chat participants communicate with one another by exchanging messages.

A message-passing facility provides at least two operations:

```
send(message)
```

and

```
receive(message)
```

Messages sent by a process can be either fixed or variable in size. If only fixed-sized messages can be sent, the system-level implementation is straightforward. This restriction, however, makes the task of programming more difficult. Conversely, variable-sized messages require a more complex system-level implementation, but the programming task becomes simpler. This is a common kind of tradeoff seen throughout operating-system design.

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. This link can be implemented in a variety of ways. We are concerned here not with the link's physical implementation (such as shared memory, hardware bus, or network, which are covered in chapter Networks and Distributed Systems) but rather with its logical implementation. 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

We look at issues related to each of these features next.

¡h3¿Naming¡/h3¿ Processes that want to communicate must have a way to refer to each other. They can use either direct or indirect communication.

Under **direct communication**, each process that wants to communicate must explicitly name the recipient or sender of the communication. In this scheme, 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.

This scheme exhibits **symmetry** in addressing; that is, both the sender process and the receiver process must name the other to communicate. A variant of this scheme employs **asymmetry** in addressing. Here, only the sender names the recipient; the recipient is not required to name the sender. In this scheme, the `send()` and `receive()` primitives are defined as follows:

- `send(P, message)`—Send a `message` to process `P`.
- `receive(id, message)`—Receive a `message` from any process. The variable `id` is set to the name of the process with which communication has taken place.

The disadvantage in both of these schemes (symmetric and asymmetric) is the limited modularity of the resulting process definitions. Changing the identifier of a process may necessitate examining all other process definitions. All references to the old identifier must be found, so that they can be modified to the new identifier. In general, any such **hard-coding** techniques, where identifiers must be explicitly stated, are less desirable than techniques involving indirection, as described next.

With **indirect communication**, the messages are sent to and received from **mailboxes**, or **ports**. A mailbox can be viewed abstractly as an object into which messages can be placed by processes and from which messages can be removed. Each mailbox has a unique identification. For example, POSIX message queues use an integer value to identify a mailbox. A process can communicate with another process via a number of different mailboxes, but two processes can communicate only if they have a shared mailbox. 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.

Now suppose that processes P_1, P_2, and P_3 all share mailbox A. Process P_1 sends a message to A, while both P_2 and P_3 execute a `receive()` from A. Which process will receive the message sent by P_1? The answer depends on which of the following methods we choose:

- Allow a link to be associated with two processes at most.

- Allow at most one process at a time to execute a `receive()` operation.
- Allow the system to select arbitrarily which process will receive the message (that is, either P_2 or P_3, but not both, will receive the message). The system may define an algorithm for selecting which process will receive the message (for example, **round robin**, where processes take turns receiving messages). The system may identify the receiver to the sender.

A mailbox may be owned either by a process or by the operating system. If the mailbox is owned by a process (that is, the mailbox is part of the address space of the process), then we distinguish between the owner (which can only receive messages through this mailbox) and the user (which can only send messages to the mailbox). Since each mailbox has a unique owner, there can be no confusion about which process should receive a message sent to this mailbox. When a process that owns a mailbox terminates, the mailbox disappears. Any process that subsequently sends a message to this mailbox must be notified that the mailbox no longer exists.

In contrast, a mailbox that is owned by the operating system has an existence of its own. It is independent and is not attached to any particular process. The operating system then must provide a mechanism that allows a process to do the following:

- Create a new mailbox.
- Send and receive messages through the mailbox.
- Delete a mailbox.

The process that creates a new mailbox is that mailbox's owner by default. Initially, the owner is the only process that can receive messages through this mailbox. However, the ownership and receiving privilege may be passed to other processes through appropriate system calls. Of course, this provision could result in multiple receivers for each mailbox.

¡h3¿Synchronization¡/h3¿ Communication between processes takes place through calls to `send()` and `receive()` primitives. There are different design options for implementing each primitive. Message passing may be either **blocking** or **nonblocking**—also known as **synchronous** and **asynchronous**. (Throughout this text, you will encounter the concepts of synchronous and asynchronous behavior in relation to various operating-system algorithms.)

- **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.

Different combinations of `send()` and `receive()` are possible. When both `send()` and `receive()` are blocking, we have a **rendezvous** between the sender and the receiver. The solution to the producer-consumer problem becomes trivial when we use blocking `send()` and `receive()` statements. The producer merely invokes the blocking `send()` call and waits until the message is delivered to either the receiver or the mailbox. Likewise, when the consumer invokes `receive()`, it blocks until a message is available. This is illustrated in Figures 3.6.1 and 3.6.2.

```
message next_produced;

while (true) {
     /* produce an item in next_produced */

     send(next_produced);
}
```

Figure 1: The producer process using message passing.

```
message next_consumed;

while (true) {
     receive(next_consumed);

     /* consume the item in next_consumed */
}
```

Figure 2: The consumer process using message passing.

¡h3¿Buffering¡/h3¿ Whether communication is direct or indirect, messages exchanged by communicating processes reside in a temporary queue. Basically, such queues can be implemented in three ways:

- **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.

The zero-capacity case is sometimes referred to as a message system with no buffering. The other cases are referred to as systems with automatic buffering.

Section glossary

Term	Definition
direct communication	In interprocess communication, a communication mode in which each process that wants to communicate must explicitly name the recipient or sender of the communication.
blocking	In interprocess communication, a mode of communication in which the sending process is blocked until the message is received by the receiving process or by a mailbox and the receiver blocks until a message is available. In I/O, a request that does not return until the I/O completes.
nonblocking	A type of I/O request that allows the initiating thread to continue while the I/O operation executes. In interprocess communication, a communication mode in which the sending process sends the message and resumes operation and the receiver process retrieves either a valid message or a null if no message is available. In I/O, a request that returns whatever data is currently available, even if it is less than requested.
synchronous	In interprocess communication, a mode of communication in which the sending process is blocked until the message is received by the receiving process or by a mailbox and the receiver blocks until a message is available. In I/O, a request that does not return until the I/O completes.
asynchronous	In I/O, a request that executes while the caller continues execution.
rendezvous	In interprocess communication, when blocking mode is used, the meeting point at which a send is picked up by a receive.
communication link	A link between processes that allows them to send messages to and receive messages from each other.
symmetry	In direct communication, a scheme in which both the sender process and the receiver process must name the other to communicate.
asymmetry	In direct communication, a scheme in which only the sender names the recipient; the recipient is not required to name the sender.
hard-coding	Techniques where identifiers must be explicitly stated.
indirect communication	A communication mode in which messages are sent to and received from mailboxes, or ports.
mailboxes	In indirect communication, objects into which messages can be placed by processes and from which messages can be removed.
ports	In indirect communication, objects into which messages can be placed by processes and from which messages can be removed.
round robin	An algorithm for selecting which process will receive a message (e.g., processes take turns receiving messages).

7 Examples of IPC systems

In this section, we explore four different IPC systems. We first cover the POSIX API for shared memory and then discuss message passing in the Mach operating system. Next, we present Windows IPC, which interestingly uses shared memory as a mechanism for providing certain types of message passing. We conclude with pipes, one of the earliest IPC mechanisms on UNIX systems.

¡h3¿POSIX shared memory¡/h3¿ 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.

The programs shown in Figure 3 and Figure 4 use the producer-consumer model in implementing shared memory. The producer establishes a shared-memory object and writes to shared memory, and the consumer reads from shared memory.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>

#include <sys/mman.h>

int main()
{
/* the size (in bytes) of shared memory object */
const int SIZE = 4096;
/* name of the shared memory object */
const char *name = "OS";
/* strings written to shared memory */
const char *message_0 = "Hello";
const char *message_1 = "World!";

/* shared memory file descriptor */
int fd;
/* pointer to shared memory obect */
char *ptr;

   /* create the shared memory object */
   fd = shm_open(name,O_CREAT | O_RDWR,0666);

   /* configure the size of the shared memory object */
   ftruncate(fd, SIZE);

   /* memory map the shared memory object */
   ptr = (char *)
    mmap(0, SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);

   /* write to the shared memory object */
   sprintf(ptr,"%s",message_0);
   ptr += strlen(message_0);
   sprintf(ptr,"%s",message_1);
   ptr += strlen(message_1);

   return 0;
}
```

Figure 3: Producer process illustrating POSIX shared-memory API.

The producer, shown in Figure 3, creates a shared-memory object named `OS` and writes the infamous string `"Hello World!"` to shared memory. The program memory-maps a shared-memory object of the specified size and allows writing to the object. The flag `MAP_SHARED` specifies that changes to the shared-memory object will be visible to all processes sharing the object. Notice that we write to the shared-memory object by calling the `sprintf()` function and writing the formatted string to the pointer `ptr`. After each write, we must increment the pointer by the number of bytes written.

The consumer process, shown in Figure 4, reads and outputs the contents of the shared memory. The consumer also invokes the `shm_unlink()` function, which removes the shared-memory segment after the consumer has accessed it. We provide further exercises using the POSIX shared-memory API in the programming exercises at the end of this chapter. Additionally, we provide more detailed coverage of memory mapping in Section 13.5.

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>

#include <sys/mman.h>

int main()
{
/* the size (in bytes) of shared memory object */
const int SIZE = 4096;
/* name of the shared memory object */
const char *name = "OS";
/* shared memory file descriptor */
int fd;
/* pointer to shared memory obect */
char *ptr;

   /* open the shared memory object */
   fd = shm_open(name, O_RDONLY, 0666);

   /* memory map the shared memory object */
   ptr = (char *)
    mmap(0, SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);

   /* read from the shared memory object */
   printf("%s",(char *)ptr);

   /* remove the shared memory object */
   shm_unlink(name);

   return 0;
}
```

Figure 4: Consumer process illustrating POSIX shared-memory API.

¡h3¿Mach message passing¡/h3¿ As an example of message passing, we next consider the Mach operating system. Mach was especially designed for distributed systems, but was shown to be suitable for desktop and mobile systems as well, as evidenced by its inclusion in the MacOS and iOS operating systems, as discussed in the previous chapter Operating-System Structures.

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 inter-task communication—is carried out by **messages**. Messages are sent to, and received from, mailboxes, which are called **ports** in Mach. Ports are finite in size and unidirectional; for two-way communication, a message is sent to one port, and a response is sent to a separate **reply** port. Each port may have multiple senders, but only one receiver. Mach uses **kernel abstractions** to represent resources such as tasks, threads, memory, and processors, while message passing provides an object-oriented approach for interacting with these system resources and services. Message passing may occur between any two ports on the same host or on separate hosts on a distributed system.

Associated with each port is a collection of **port rights** that identify the capabilities necessary for a task to interact with the port. For example, for a task to receive a message from a port, it must have the capability `MACH_PORT_RIGHT_RECEIVE` for that port. The task that creates a port is that port's owner, and the owner is the only task that is allowed to receive messages from that port. A port's owner may also manipulate the capabilities for a port. This is most commonly done in establishing a reply port. For example, assume that task $T1$ owns port $P1$, and it sends a message to port $P2$, which is owned by task $T2$. If $T1$ expects to receive a reply from $T2$, it must grant $T2$ the right `MACH_PORT_RIGHT_SEND` for port $P1$. Ownership of port rights is at the task level, which means that all threads belonging to the same task share the same port rights. Thus, two threads belonging to the same task can easily communicate by exchanging messages through the per-thread port associated with each thread.

When a task is created, two special ports—the **Task Self** port and the **Notify** port—are also created. The kernel has receive 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 (to which, of course, the task has receive rights).

The `mach_port_allocate()` function call creates a new port and allocates space for its queue of messages. It also identifies the rights for the port. Each port right represents a **name** for that port, and a port can only be accessed via a right. Port names are simple integer values and behave much like UNIX file descriptors. The following example illustrates creating a port using this API:

```
mach_port_t port; // the name of the port right

mach_port_allocate(
        mach_task_self(), // a task referring to itself
        MACH_PORT_RIGHT_RECEIVE, // the right for this port
        &port); // the name of the port right
```

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**. Once a port has been registered with the bootstrap server, other tasks can look up the port in this registry and obtain rights for sending messages to the port.

The queue associated with each port is finite in size and is initially empty. As messages are sent to the port, the messages are copied into the queue. All messages are delivered reliably and have the same priority. Mach guarantees that multiple messages from the same sender are queued in first-in, first-out (FIFO) order but does not guarantee an absolute ordering. For instance, messages from two senders may be queued in any order.

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. Out-of-line data pointers are especially useful when a message must pass large chunks of data. A simple message would require copying and packaging the data in the message; out-of-line data transmission requires only a pointer that refers to the memory location where the data are stored.

The function `mach_msg()` is the standard API for both sending and receiving messages. The value of one of the function's parameters—either `MACH_SEND_MSG` or `MACH_RCV_MSG`—indicates if it is a send or receive operation. We now illustrate how it is used when a client task sends a simple message to a server task. Assume there are two ports—`client` and `server`—associated with the client and server tasks, respectively. The code in Figure 5 shows the client task constructing a header and sending a message to the server, as well as the server task receiving the message sent from the client.

```
#include<mach/mach.h>

struct message {
  mach_msg_header_t header;
  int data;
};

mach_port_t client;
mach_port_t server;

       /* Client Code */

struct message message;

// construct the header
message.header.msgh_size = sizeof(message);
message.header.msgh_remote_port = server;
message.header.msgh_local_port = client;

// send the message
mach_msg(&message.header, // message header
  MACH_SEND_MSG, // sending a message
  sizeof(message), // size of message sent
  0, // maximum size of received message - unnecessary
  MACH_PORT_NULL, // name of receive port - unnecessary
  MACH_MSG_TIMEOUT_NONE, // no time outs
  MACH_PORT_NULL // no notify port
);

      /* Server Code */

struct message message;

// receive the message
mach_msg(&message.header, // message header
  MACH_RCV_MSG, // receiving a message
  0,  // size of message sent
  sizeof(message), // maximum size of received message
  server, // name of receive port
  MACH_MSG_TIMEOUT_NONE, // no time outs
  MACH_PORT_NULL // no notify port
);
```

Figure 5: Example program illustrating message passing in Mach.

The `mach_msg()` function call is invoked by user programs for performing message passing. `mach_msg()` then invokes the function `mach_msg_trap()`, which is a system call to the Mach kernel. Within the kernel, `mach_msg_trap()` next calls the function `mach_msg_overwrite_trap()`, which then handles the actual passing of the message.

The send and receive operations themselves are flexible. For instance, when a message is sent to a port, its queue may be full. If the queue is not full, the message is copied to the queue, and the sending task continues. If the port's queue is full, the sender has several options (specified via parameters to `mach_msg()`):

1. Wait indefinitely until there is room in the queue.
2. Wait at most n milliseconds.
3. Do not wait at all but rather return immediately.
4. Temporarily cache a message. Here, a message is given to the operating system to keep, even though the queue to which that message is being sent is full. When the message can be put in the queue, a notification message is sent back to the sender. Only one message to a full queue can be pending at any time for a given sending thread.

The final option is meant for server tasks. After finishing a request, a server task may need to send a one-time reply to the task that requested the service, but it must also continue with other service requests, even if the reply port for a client is full.

The major problem with message systems has generally been poor performance caused by copying of messages from the sender's port to the receiver's port. The Mach message system attempts to avoid copy operations by using virtual-memory-management techniques (Chapter Virtual Memory). Essentially, Mach maps the address space containing the sender's message into the receiver's address space. Therefore, the message itself is never actually copied, as both the sender and receiver access the same memory. This message-management technique provides a large performance boost but works only for intrasystem messages.

¡h3¿Windows¡/h3¿ The Windows operating system is an example of modern design that employs modularity to increase functionality and decrease the time needed to implement new features. Windows provides support for multiple operating environments, or **subsystems**. Application programs communicate with these subsystems via a message-passing mechanism. Thus, application programs can be considered clients of a subsystem server.

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. (Remote procedure calls are covered in detail in Section Remote procedure calls.) 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**.

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.

When an ALPC channel is created, one of three message-passing techniques is chosen:

1. 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.
2. Larger messages must be passed through a **section object**, which is a region of shared memory associated with the channel.
3. 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.

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. This method is more complicated than the first method listed above, but it avoids data copying. The structure of advanced local procedure calls in Windows is shown in Figure 6.

Figure 6: Advanced local procedure calls in Windows.

It is important to note that the ALPC facility in Windows is not part of the Windows API and hence is not visible to the application programmer. Rather, applications using the Windows API invoke standard remote procedure calls. When the RPC is being invoked on a process on the same system, the RPC is handled indirectly through an ALPC procedure call. Additionally, many kernel services use ALPC to communicate with client processes.

¡h3¿Pipes¡/h3¿ A **pipe** acts as a conduit allowing two processes to communicate. Pipes were one of the first IPC mechanisms in early UNIX systems. They typically provide one of the simpler ways for processes to communicate with one another, although they also have some limitations. 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?

In the following sections, we explore two common types of pipes used on both UNIX and Windows systems: ordinary pipes and named pipes.

¡h4¿Ordinary pipes¡/h4¿ 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. We next illustrate constructing ordinary pipes on both UNIX and Windows systems. In both program examples, one process writes the message `Greetings` to the pipe, while the other process reads this message from the pipe.

On UNIX systems, ordinary pipes are constructed using the function

```
pipe(int fd[])
```

This function creates a pipe that is accessed through the `int fd[]` file descriptors: `fd[0]` is the read end of the pipe, and `fd[1]` is the write end. UNIX treats a pipe as a special type of file. Thus, pipes can be accessed using ordinary `read()` and `write()` system calls.

An ordinary pipe cannot be accessed from outside the process that created it. Typically, a parent process creates a pipe and uses it to communicate with a child process that it creates via `fork()`. Recall from Section Process creation that a child process inherits open files from its parent. Since a pipe is a special type of file, the child inherits the pipe from its parent process. Figure 7 illustrates the relationship of the file descriptors in the `fd` array to the parent and child processes. As this illustrates, any writes by the parent to its write end of the pipe—`fd[1]`—can be read by the child from its read end—`fd[0]`—of the pipe.

Figure 7: File descriptors for an ordinary pipe.

In the UNIX program shown in Figure 8, the parent process creates a pipe and then sends a `fork()` call creating the child process. What occurs after the `fork()` call depends on how the data are to flow through the pipe. In this instance, the parent writes to the pipe, and the child reads from it. It is important to notice that both the parent process and the child process initially close their unused ends of the pipe. Although the program shown in Figure 8 does not require this action, it is an important step to ensure that a process reading from the pipe can detect end-of-file (`read()` returns 0) when the writer has closed its end of the pipe.

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. In addition, reading and writing to the pipe can be accomplished with the ordinary `ReadFile()` and `WriteFile()` functions. The Windows API for creating pipes

```
#include <sys/types.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>

#define BUFFER_SIZE 25
#define READ_END 0
#define WRITE_END 1

int main(void)
{
  char write_msg[BUFFER_SIZE] = "Greetings";
  char read_msg[BUFFER_SIZE];
  int fd[2];
  pid_t pid;

   /* create the pipe */
   if (pipe(fd) == -1) {
     fprintf(stderr,"Pipe failed");
     return 1;
   }

   /* fork a child process */
   pid = fork();

   if (pid < 0) { /* error occurred */
     fprintf(stderr, "Fork Failed");
     return 1;
   }

   if (pid > 0) { /* parent process */
     /* close the unused end of the pipe */
     close(fd[READ_END]);

     /* write to the pipe */
     write(fd[WRITE_END], write_msg, strlen(write_msg)+1);

     /* close the write end of the pipe */
     close(fd[WRITE_END]);
   }
   else { /* child process */
     /* close the unused end of the pipe */
     close(fd[WRITE_END]);

     /* read from the pipe */
     read(fd[READ_END], read_msg, BUFFER_SIZE);
     printf("read %s",read_msg);

     /* close the read end of the pipe */
     close(fd[READ_END]);
   }
   return 0;
}
```

Figure 8: Ordinary pipe in UNIX.

is the `CreatePipe()` function, which is passed four parameters. The parameters provide separate handles for (1) reading and (2) writing to the pipe, as well as (3) an instance of the `STARTUPINFO` structure, which is used to specify that the child process is to inherit the handles of the pipe. Furthermore, (4) the size of the pipe (in bytes) may be specified.

Figure 9 illustrates a parent process creating an anonymous pipe for communicating with its child. Unlike UNIX systems, in which a child process automatically inherits a pipe created by its parent, Windows requires the programmer to specify which attributes the child process will inherit. This is accomplished by first initializing the `SECURITY_ATTRIBUTES` structure to allow handles to be inherited and then redirecting the child process's handles for standard input or standard output to the read or write handle of the pipe. Since the child will be reading from the pipe, the parent must redirect the child's standard input to the read handle of the pipe. Furthermore, as the pipes are half duplex, it is necessary to prohibit the child from inheriting the write end of the pipe. The program to create the child process is similar to the program in Figure 3.3.4, except that the fifth parameter is set to `TRUE`, indicating that the child process is to inherit designated handles from its parent. Before writing to the pipe, the parent first closes its unused read end of the pipe. The child process that reads from the pipe is shown in Figure 10. Before reading from the pipe, this program obtains the read handle to the pipe by invoking `GetStdHandle()`.

Note that ordinary pipes require a parent-child relationship between the communicating processes on both UNIX and Windows systems. This means that these pipes can be used only for communication between processes on the same machine.

¡h4¿Named pipes¡/h4¿ Ordinary pipes provide a simple mechanism for allowing a pair of processes to communicate. However, ordinary pipes exist only while the processes are communicating with one another. On both UNIX and Windows systems, once the processes have finished communicating and have terminated, the ordinary pipe ceases to exist.

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. In fact, in a typical scenario, a named pipe has several writers. Additionally, named pipes continue to exist after communicating processes have finished. Both UNIX and Windows systems support named pipes, although the details of implementation differ greatly. Next, we explore named pipes in each of these systems.

Named pipes are referred to as FIFOs in UNIX systems. Once created, they appear as typical files in the file system. A FIFO is

```
#include <stdio.h>
#include <stdlib.h>
#include <windows.h>

#define BUFFER_SIZE 25

int main(VOID)
{
  HANDLE ReadHandle, WriteHandle;
  STARTUPINFO si;
  PROCESS_INFORMATION pi;
  char message[BUFFER_SIZE] = "Greetings";
  DWORD written;

/* set up security attributes allowing pipes to be inherited */
SECURITY_ATTRIBUTES sa = {sizeof(SECURITY_ATTRIBUTES),{NULL},{TRUE};
/* allocate memory */
ZeroMemory(&pi, sizeof(pi));

/* create the pipe */
if (!CreatePipe(&ReadHandle, &WriteHandle, &sa, 0)) {
   fprintf(stderr, "Create Pipe Failed");
   return 1;
}

/* establish the START_INFO structure for the child process */
GetStartupInfo(&si);
si.hStdOutput = GetStdHandle(STD_OUTPUT_HANDLE);

/* redirect standard input to the read end of the pipe */
si.hStdInput = ReadHandle;
si.dwFlags = STARTF_USESTDHANDLES;

/* don't allow the child to inherit the write end of pipe */
SetHandleInformation(WriteHandle, HANDLE_FLAG_INHERIT, 0);

/* create the child process */
CreateProcess({NULL}, "child.exe", {NULL, NULL},
 TRUE, /* inherit handles */
 0, NULL, NULL, &si, &pi);

/* close the unused end of the pipe */
CloseHandle(ReadHandle);

/* the parent writes to the pipe */
if (!WriteFile(WriteHandle, message,BUFFER_SIZE,&written,NULL))
   fprintf(stderr, "Error writing to pipe.");

/* close the write end of the pipe */
CloseHandle(WriteHandle);

/* wait for the child to exit */
WaitForSingleObject(pi.hProcess, INFINITE);
CloseHandle(pi.hProcess);
CloseHandle(pi.hThread);
return 0;
}
```

Figure 9: Windows anonymous pipe—parent process.

created with the `mkfifo()` system call and manipulated with the ordinary `open()`, `read()`, `write()`, and `close()` system calls. It will continue to exist until it is explicitly deleted from the file system. Although FIFOs allow bidirectional communication, only half-duplex transmission is permitted. If data must travel in both directions, two FIFOs are typically used. Additionally, the communicating processes must reside on the same machine. If intermachine communication is required, sockets (Section Sockets) must be used.

Named pipes on Windows systems provide a richer communication mechanism than their UNIX counterparts. Full-duplex communication is allowed, and the communicating processes may reside on either the same or different machines. Additionally, only byte-oriented data may be transmitted across a UNIX FIFO, whereas Windows systems allow either byte- or message-oriented data. Named pipes are created with the `CreateNamedPipe()` function, and a client can connect to a named pipe using `ConnectNamedPipe()`. Communication over the named pipe can be accomplished using the `ReadFile()` and `WriteFile()` functions.

```
#include <stdio.h>
#include <windows.h>

#define BUFFER SIZE 25

int main(VOID)
{
HANDLE Readhandle;
CHAR buffer[BUFFER_SIZE];
DWORD read;

   /* get the read handle of the pipe */
   ReadHandle = GetStdHandle(STD_INPUT_HANDLE);

   /* the child reads from the pipe */
   if (ReadFile(ReadHandle, buffer, BUFFER_SIZE, &read, NULL))
     printf("child read %s",buffer);
   else
     fprintf(stderr, "Error reading from pipe");

return 0;
}
```

Figure 10: Windows anonymous pipes—child process.

Section glossary

Term	Definition
message	In networking, a communication, contained in one or more packets, that includes source and destination information to allow correct delivery. In message-passing communications, a packet of information with metadata about its sender and receiver.
port	A communication address; a system may have one IP address for network connections but many ports, each for a separate communication. In computer I/O, a connection point for devices to attach to computers. In software development, to move code from its current platform to another platform (e.g., between operating systems or hardware systems). In the Mach OS, a mailbox for communication.
bootstrap port	In Mach message passing, a predefined port that allows a task to register a port it has created.
bootstrap server	In Mach message passing, a system-wide service for registering ports.
advanced local procedure call (ALPC)	In Windows OS, a method used for communication between two processes on the same machine.
connection port	In Windows OS, a communications port used to maintain connection between two processes, published by a server process.
communication port	In Windows OS, a port used to send messages between two processes.
section object	The Windows data structure that is used to implement shared memory.
pipe	A logical conduit allowing two processes to communicate.
write end	In ordinary pipes, the end of the pipe to which the producer writes.
read end	In ordinary pipes, the end of the pipe from which the consumer reads.
anonymous pipes	Ordinary pipes on Windows systems.

8 Communication in client-server systems

In Section 3.4, we described how processes can communicate using shared memory and message passing. These techniques can be used for communication in client-server systems (sub-section Client-server computing in 1.10) as well. In this section, we explore two other strategies for communication in client-server systems: sockets and remote procedure calls (RPCs). As we shall see in our coverage of RPCs, not only are they useful for client-server computing, but Android also uses remote procedures as a form of IPC between processes running on the same system.

¡h3¿Sockets¡/h3¿ 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. 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.

When a client process initiates a request for a connection, it is assigned a port by its host computer. This port has some arbitrary number greater than 1024. For example, if a client on host `X` with IP address 146.86.5.20 wishes to establish a connection with a web server (which is listening on port 80) at address 161.25.19.8, host `X` may be assigned port 1625. The connection will consist of a pair of sockets: (146.86.5.20:1625) on host `X` and (161.25.19.8:80) on the web server. This situation is illustrated in Figure 11. The packets traveling between the hosts are delivered to the appropriate process based on the destination port number.

Figure 11: Communication using sockets.

All connections must be unique. Therefore, if another process also on host `X` wished to establish another connection with the same web server, it would be assigned a port number greater than 1024 and not equal to 1625. This ensures that all connections consist of a unique pair of sockets.

Although most program examples in this text use C, we will illustrate sockets using Java, as it provides a much easier interface to sockets and has a rich library for networking utilities. Those interested in socket programming in C or C++ should consult the bibliographical notes at the end of the chapter.

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.

Our example describes a date server that uses connection-oriented TCP sockets. The operation allows clients to request the current date and time from the server. The server listens to port 6013, although the port could have any arbitrary, unused number greater than 1024. When a connection is received, the server returns the date and time to the client.

The date server is shown in Figure 12. The server creates a `ServerSocket` that specifies that it will listen to port 6013. The server then begins listening to the port with the `accept()` method. The server blocks on the `accept()` method waiting for a client to request a connection. When a connection request is received, `accept()` returns a socket that the server can use to communicate with the client.

```
import java.net.*;
import java.io.*;

public class DateServer
{
  public static void main(String[] args) {
    try {
      ServerSocket sock = new ServerSocket(6013);

      /* now listen for connections */
      while (true) {
        Socket client = sock.accept();

        PrintWriter pout = new
         PrintWriter(client.getOutputStream(), true);

        /* write the Date to the socket */
        pout.println(new java.util.Date().toString());

        /* close the socket and resume */
        /* listening for connections */
        client.close();
      }
  }
  catch (IOException ioe) {
    System.err.println(ioe);
  }
 }
}
```

Figure 12: Date server.

The details of how the server communicates with the socket are as follows. The server first establishes a `PrintWriter` object that it will use to communicate with the client. A `PrintWriter` object allows the server to write to the socket using the routine `print()` and `println()` methods for output. The server process sends the date to the client, calling the method `println()`. Once it has written the date to the socket, the server closes the socket to the client and resumes listening for more requests.

A client communicates with the server by creating a socket and connecting to the port on which the server is listening. We implement such a client in the Java program shown in Figure 13. The client creates a `Socket` and requests a connection with the server at IP address 127.0.0.1 on port 6013. Once the connection is made, the client can read from the socket using normal

stream I/O statements. After it has received the date from the server, the client closes the socket and exits. The IP address 127.0.0.1 is a special IP address known as the **loopback**. When a computer refers to IP address 127.0.0.1, it is referring to itself. This mechanism allows a client and server on the same host to communicate using the TCP/IP protocol. The IP address 127.0.0.1 could be replaced with the IP address of another host running the date server. In addition to an IP address, an actual host name, such as `www.westminstercollege.edu`, can be used as well.

```
import java.net.*;
import java.io.*;

public class DateClient
{
  public static void main(String[] args)  {
    try {
      /* make connection to server socket */
      Socket sock = new Socket("127.0.0.1",6013);

      InputStream in = sock.getInputStream();
      BufferedReader bin = new
        BufferedReader(new InputStreamReader(in));

      /* read the date from the socket */
      String line;
      while ( (line = bin.readLine()) != null)
        System.out.println(line);

      /* close the socket connection*/
      sock.close();
    }
    catch (IOException ioe) {
      System.err.println(ioe);
    }
  }
}
```

Figure 13: Date client.

Communication using sockets—although common and efficient—is considered a low-level form of communication between distributed processes. One reason is that sockets allow only an unstructured stream of bytes to be exchanged between the communicating threads. It is the responsibility of the client or server application to impose a structure on the data. In the next subsection, we look a higher-level method of communication: remote procedure calls (RPCs).

¡h3¿Remote procedure calls¡/h3¿ One of the most common forms of remote service is the RPC paradigm, which was designed as a way to abstract the procedure-call mechanism for use between systems with network connections. It is similar in many respects to the IPC mechanism described in Section 3.4, and it is usually built on top of such a system. Here, however, because we are dealing with an environment in which the processes are executing on separate systems, we must use a message-based communication scheme to provide remote service.

In contrast to IPC messages, the messages exchanged in RPC communication are well structured and are thus no longer just packets of data. Each message is addressed to an RPC daemon listening to a port on the remote system, and each contains an identifier specifying the function to execute and the parameters to pass to that function. The function is then executed as requested, and any output is sent back to the requester in a separate message.

A **port** in this context is simply a number included at the start of a message packet. Whereas a system normally has one network address, it can have many ports within that address to differentiate the many network services it supports. If a remote process needs a service, it addresses a message to the proper port. For instance, if a system wished to allow other systems to be able to list its current users, it would have a daemon supporting such an RPC attached to a port—say, port 3027. Any remote system could obtain the needed information (that is, the list of current users) by sending an RPC message to port 3027 on the server. The data would be received in a reply message.

The semantics of RPCs allows a client to invoke a procedure on a remote host as it would invoke a procedure locally. The RPC system hides the details that allow communication to take place by providing a **stub** on the client side. Typically, a separate stub exists for each separate remote procedure. When the client invokes a remote procedure, the RPC system calls the appropriate stub, passing it the parameters provided to the remote procedure. This stub locates the port on the server and **marshals** the parameters. The stub then transmits a message to the server using message passing. A similar stub on the server side receives this message and invokes the procedure on the server. If necessary, return values are passed back to the client using the same technique. On Windows systems, stub code is compiled from a specification written in the **Microsoft Interface Definition Language** (*MIDL*), which is used for defining the interfaces between client and server programs.

Parameter marshaling addresses the issue concerning differences in data representation on the client and server machines. 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. Neither order is "better" per se; rather, the choice is arbitrary within a computer architecture. 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*). On the client side, parameter marshaling involves converting the machine-dependent data into XDR before they are sent to the server. On the server side, the XDR data are unmarshaled and converted to the machine-dependent representation for the server.

Another important issue involves the semantics of a call. Whereas local procedure calls fail only under extreme circumstances, RPCs can fail, or be duplicated and executed more than once, as a result of common network errors. One way to address this problem is for the operating system to ensure that messages are acted on **exactly once**, rather than **at most once**. Most local procedure calls have the "exactly once" functionality, but it is more difficult to implement.

First, consider "at most once." This semantic can be implemented by attaching a timestamp to each message. The server must keep a history of all the timestamps of messages it has already processed or a history large enough to ensure that repeated messages are detected. Incoming messages that have a timestamp already in the history are ignored. The client can then send a message one or more times and be assured that it only executes once.

For "exactly once," we need to remove the risk that the server will never receive the request. To accomplish this, the server must implement the "at most once" protocol described above but must also acknowledge to the client that the RPC call was received and executed. These ACK messages are common throughout networking. The client must resend each RPC call periodically until it receives the ACK for that call.

Yet another important issue concerns the communication between a server and a client. With standard procedure calls, some form of binding takes place during link, load, or execution time (Chapter Main Memory) so that a procedure call's name is replaced by the memory address of the procedure call. The RPC scheme requires a similar binding of the client and the server port, but how does a client know the port numbers on the server? Neither system has full information about the other, because they do not share memory.

Two approaches are common. First, the binding information may be predetermined, in the form of fixed port addresses. At compile time, an RPC call has a fixed port number associated with it. Once a program is compiled, the server cannot change the port number of the requested service. Second, binding can be done dynamically by a rendezvous mechanism. Typically, an operating system provides a rendezvous (also called a **matchmaker**) daemon on a fixed RPC port. A client then sends a message containing the name of the RPC to the rendezvous daemon requesting the port address of the RPC it needs to execute. The port number is returned, and the RPC calls can be sent to that port until the process terminates (or the server crashes). This method requires the extra overhead of the initial request but is more flexible than the first approach. Figure 14 shows a sample interaction.

Figure 14: Execution of a remote procedure call (RPC).

The RPC scheme is useful in implementing a distributed file system (Chapter Networks and Distributed Systems). Such a system can be implemented as a set of RPC daemons and clients. The messages are addressed to the distributed file system port on a server on which a file operation is to take place. The message contains the disk operation to be performed. The disk operation might be `read()`, `write()`, `rename()`, `delete()`, or `status()`, corresponding to the usual file-related system calls. The return message contains any data resulting from that call, which is executed by the DFS daemon on behalf of the client. For instance, a message might contain a request to transfer a whole file to a client or be limited to a simple block request. In the latter case, several requests may be needed if a whole file is to be transferred.

¡h4¿Android RPC¡/h4¿ Although RPCs are typically associated with client-server computing in a distributed system, they can also be used as a form of IPC between processes running on the same system. The Android operating system has a rich set of IPC mechanisms contained in its **binder** framework, including RPCs that allow one process to request services from another process.

Android defines an **application component** as a basic building block that provides utility to an Android application, and an app may combine multiple application components to provide functionality to an app. One such application component is a **service**, which has no user interface but instead runs in the background while executing long-running operations or performing work for remote processes. Examples of services include playing music in the background and retrieving data over a network connection on behalf of another process, thereby preventing the other process from blocking as the data are being downloaded. When a client app invokes the `bindService()` method of a service, that service is "bound" and available to provide client-server communication using either message passing or RPCs.

A bound service must extend the Android class `Service` and must implement the method `onBind()`, which is invoked when a client calls `bindService()`. In the case of message passing, the `onBind()` method returns a `Messenger` service, which is used for sending messages from the client to the service. The `Messenger` service is only one-way; if the service must send a reply back to the client, the client must also provide a `Messenger` service, which is contained in the `replyTo` field of the `Message` object sent to the service. The service can then send messages back to the client.

To provide RPCs, the `onBind()` method must return an interface representing the methods in the remote object that clients use to interact with the service. This interface is written in regular Java syntax and uses the Android Interface Definition Language—AIDL—to create stub files, which serve as the client interface to remote services.

Here, we briefly outline the process required to provide a generic remote service named `remoteMethod()` using AIDL and the binder service. The interface for the remote service appears as follows:

```
/* RemoteService.aidl */
interface RemoteService
{
  boolean remoteMethod(int x, double y);
{
```

This file is written as `RemoteService.aidl`. The Android development kit will use it to generate a `.java` interface from the `.aidl` file, as well as a stub that serves as the RPC interface for this service. The server must implement the interface generated by the `.aidl` file, and the implementation of this interface will be called when the client invokes `remoteMethod()`.

When a client calls `bindService()`, the `onBind()` method is invoked on the server, and it returns the stub for the `RemoteService` object to the client. The client can then invoke the remote method as follows:

```
RemoteService service;
   . . .
service.remoteMethod(3, 0.14);
```

Internally, the Android binder framework handles parameter marshaling, transferring marshaled parameters between processes, and invoking the necessary implementation of the service, as well as sending any return values back to the client process.

Section glossary

Term	Definition
socket	An endpoint for communication. An interface for network I/O.
connection-oriented socket (TCP)	In Java, a mode of communication.
connectionless socket (UDP)	In Java, a mode of communication.
loopback	Communication in which a connection is established back to the sender.
port	A communication address; a system may have one IP address for network connections but many ports, each for a separate communication. In computer I/O, a connection point for devices to attach to computers. In software development, to move code from its current platform to another platform (e.g., between operating systems or hardware systems). In the Mach OS, a mailbox for communication.
stub	A small, temporary place-holder function replaced by the full function once its expected behavior is known.
Microsoft Interface Definition Language	The Microsoft text-based interface definition language; used, e.g., to write client stub code and descriptors for RPC.
big-endian	A system architecture in which the most significant byte in a sequence of bytes is stored first.
little-endian	A system architecture that stores the least significant byte first in a sequence of bytes.
external data representation	A system used to resolve differences when data are exchanged between big- and little-endian systems.
matchmaker	A function that matches a caller to a service being called (e.g., a remote procedure call attempting to find a server daemon).
binder	In Android RPC, a framework (system component) for developing object-oriented OS services and allowing them to communicate.
application component	In Android, a basic building block that provides utility to an Android app.
service	A software entity running on one or more machines and providing a particular type of function to calling clients. In Android, an application component with no user interface; it runs in the background while executing long-running operations or performing work for remote processes.

9 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.