Synchronization

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**.

1. **Blocking send**. The sending process is blocked until the message is received by the receiving process or by the mailbox.
2. **Non-blocking send**. The sending process sends the message and resumes operation.
3. **Blocking receive**. The receiver blocks until a message is available.
4. **Non-blocking receive**. The receiver retrieves either a valid message or a null.

Buffering

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, and the sender can continue execution without waiting. 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.

Communication in Client–Server Systems

1. Sockets

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.

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 telnet, FTP, and HTTP) listen to well-known ports (a telnet server listens to port 23; 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;*** we can use them 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 3.20. The packets traveling between the hosts are delivered to the appropriate process based on the destination port number.

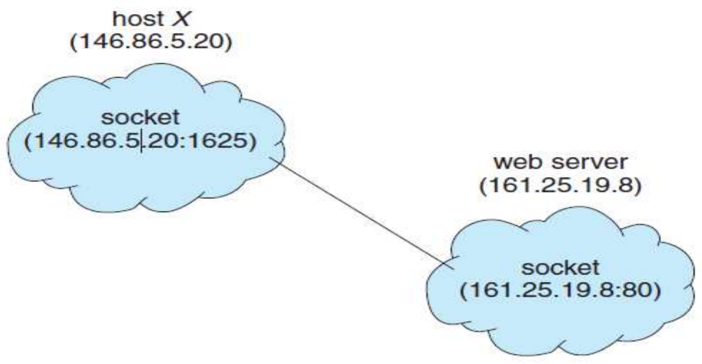

Figure 3.20 Communication using sockets.

2. Pipes

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?
we explore two common types of pipes used on both UNIX and Windows systems: **ordinary pipes and named pipes**.

1. Ordinary Pipes

Ordinary pipes allow two processes to communicate in standard producer– consumer fashion: the producer writes to one end of the pipe (the write-end) and the consumer reads from the other end (the read-end). 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.

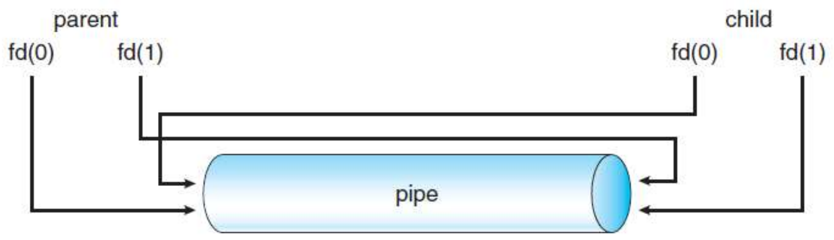

Figure 3.24 File descriptors for an ordinary pipe.

2. Named Pipes

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

Thread

A traditional process has a single thread of control. If a process has multiple threads of control, it can perform more than one task at a time. Figure 4.1 illustrates the difference between a traditional single-threaded process and a multithreaded process.

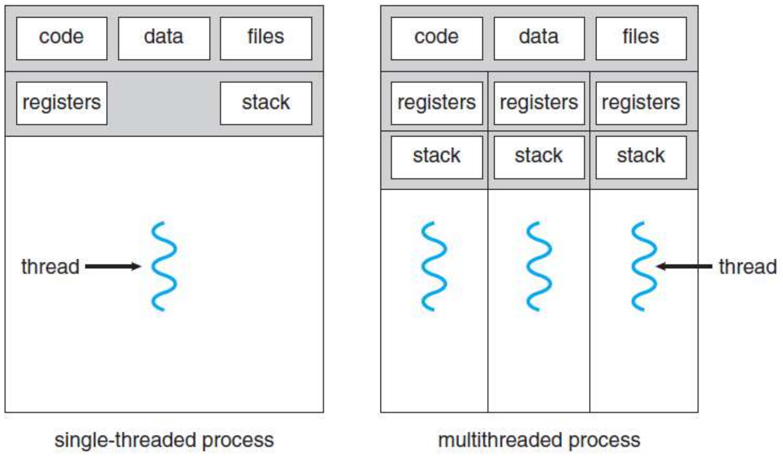

Figure 4.1 Single-threaded and multithreaded processes.

Motivation

Most software applications that run on modern computers are multithreaded. An application typically is implemented as a separate process with several threads of control. A web browser might have one thread display images or text while another thread retrieves data from the network, for example. A word processor may have a thread for displaying graphics, another thread for responding to keystrokes from the user, and a third thread for performing spelling and grammar checking in the background.

Benefits

The benefits of multithreaded programming can be broken down into four major categories:

1. Responsiveness. Multithreading an interactive application may allow a program to continue running even if part of it is blocked.

2. Resource sharing. Processes can only share resources through techniques such as shared memory and message passing.

3. Economy. Allocating memory and resources for process creation is costly. Because threads share the resources of the process to which they belong, it is more economical to create and context-switch threads.

4. Scalability. The benefits of multithreading can be even greater in a multiprocessor architecture, where threads may be running in parallel on different processing cores. A single-threaded process can run on only one processor, regardless how many are available.

Multicore Programming

Earlier in the history of computer design, in response to the need for more computing performance, single-CPU systems evolved into multi-CPU systems.
A more recent, similar trend in system design is to place multiple computing cores on a single chip. Each core appears as a separate processor to the operating system. Whether the cores appear within CPU chips, we call these systems multicore or multiprocessor systems.
Multithreaded programming provides a mechanism for more efficient use of these multiple computing cores and improved concurrency. Consider an application with four threads. On a system with a single computing core, concurrency merely means that the execution of the threads will be interleaved over time (Figure 4.3), because the processing core is capable of executing only one

thread at a time. On a system with multiple cores, however, concurrency means that the threads can run in parallel, because the system can assign a separate thread to each core (Figure 4.4).

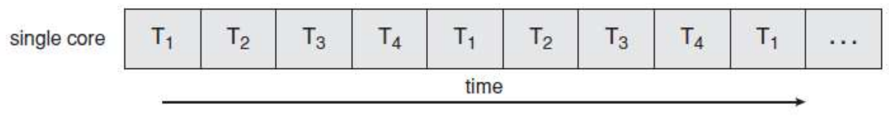

Figure 4.3 Concurrent execution on a single-core system.

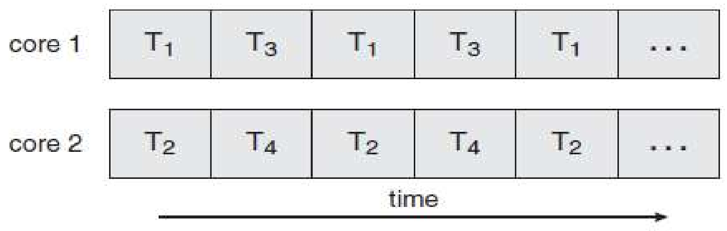

Figure 4.4 Parallel execution on a multicore system.

Multithreading Models

Our discussion so far has treated threads in a generic sense. However, support for threads may be provided either at the user level, for user threads, or by the kernel, for kernel threads. User threads are supported above the kernel and are managed without kernel support, whereas kernel threads are supported and managed directly by the operating system. operating systems—including Windows, Linux, Mac OS X, and Solaris— support kernel threads.

a relationship must exist between user threads and kernel threads. In this section, we look at three common ways of establishing such a relationship: the many-to-one model, the one-to-one model, and the many-to many model.

1. Many-to-One Model

The many-to-one model (Figure 4.5) maps many user-level threads to one kernel thread.

However, the entire process will block if a thread makes a blocking. Also, because only one thread can access the kernel at a time, multiple threads are unable to run in parallel on multicore systems.

Very few systems continue to use the model because of its inability to take advantage of multiple processing cores.

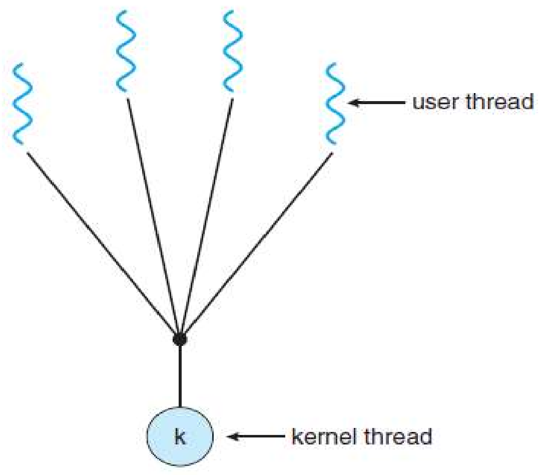

Figure 4.5 Many-to-one model.

2. One-to-One Model

The one-to-one model (Figure 4.6) maps each user thread to a kernel thread. It provides more concurrency than the many-to-one model by allowing another thread to run when a thread makes a blocking system call. It also allows multiple threads to run in parallel on multiprocessors. The only drawback to this model is that creating a user thread requires creating the corresponding kernel thread.

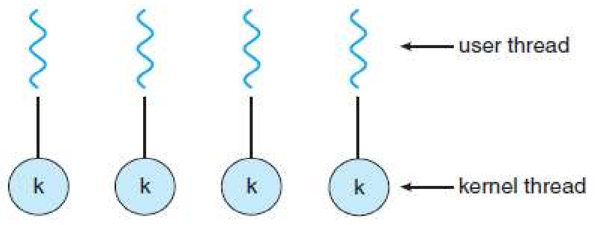

Figure 4.6 One-to-one model.

3. Many-to-Many Model

The many-to-many model (Figure 4.7) multiplexes many user-level threads to a smaller or equal number of kernel threads.
The many-to-many model suffers from neither of these shortcomings:
developers can create as many user threads as necessary, and the corresponding kernel threads can run in parallel on a multiprocessor. Also, when a thread performs a blocking, the kernel can schedule another thread for execution.

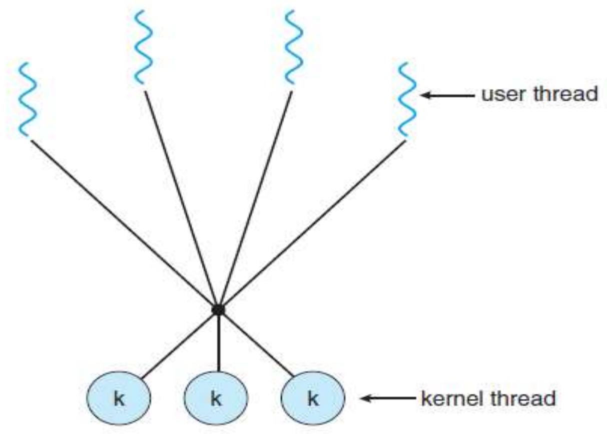

Figure 4.7 Many-to-many model.