Student: Vũ Việt Hoàng

ID: 15041

Chapter 3: Processes

3.1 Process Concept

3.1.1

Is divided into multiple sections:

* 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)

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 towardone another, the operating system must ensure they do not overlap one another

A program is a  
***passive*** entity, such as a file containing a list of instructions stored on disk  
(often called an **executable fil**). 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.

3.1.2 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.

3.1.3 Process Control Block

Each process is represented in the operating system by a process controlblock (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.  
  (Chapter 5 describes process scheduling.)
* **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 (Chapter 9).
* **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.

3.1.4 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.

**3.2 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.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.

3.2.1 Scheduling Queues

As processes enter the system, they are put into a **ready queue**, where they are  
ready and waiting to execute on a CPU’s core This queue is generally stored as  
a linked list; a ready-queue header contains pointers to the first PCB in the list,  
and each PCB includes a pointer field that points to the next PCB in the ready  
queue.  
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.

3.2.2 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.  
Swapping is discussed in Chapter 9.

3.2.3 Context SwitchInterrupts 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.

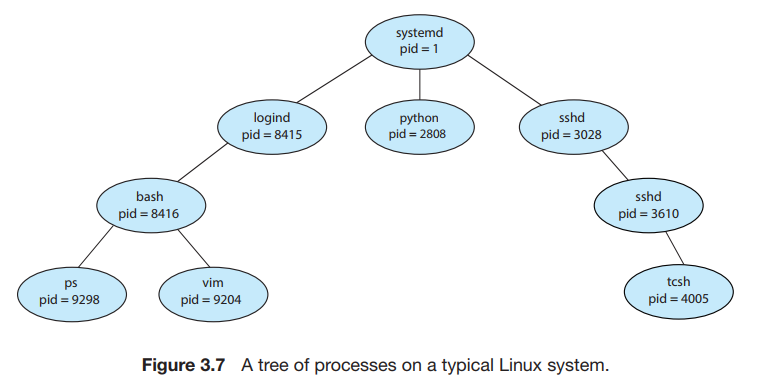
**3.3 Operations on Processes**

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.

3.3.1 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 identifie** (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 similar to the one shown in Figure 3.7 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 (destroying 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.

3.3.2 Process Termination

A process terminates when it finishes executing its final statement and asks  
the operating system to delete it by using the exit() system call. 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);

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. (Recall from Section 3.3.1 that init serves as the  
root of the process hierarchy in UNIX systems.) 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.

3.3.2.1 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.

**3.4 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 Chapter 2

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.

**3.5 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.

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.

**3.6 IPC in Message-Passing Systems**

* Message passing provides a mechanism to allow processes to communicate and to synchronize their actions without sharing the same address space. It is particularly useful in a distributed environment, where the communicating processes may reside on different computers connected by a network
* If processes P and Q want to communicate, they must send messages to and receive messages from each other: a communication link must exist between them. Here are several methods for logically implementing a link and the send()/receive() operations:
* Direct or indirect communication
* Synchronous or asynchronous communication
* Automatic or explicit buffering

3.6.1 Naming

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

3.6.2 Synchronization

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

3.6.3 Buffering

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

**3.7 Examples of IPC Systems**

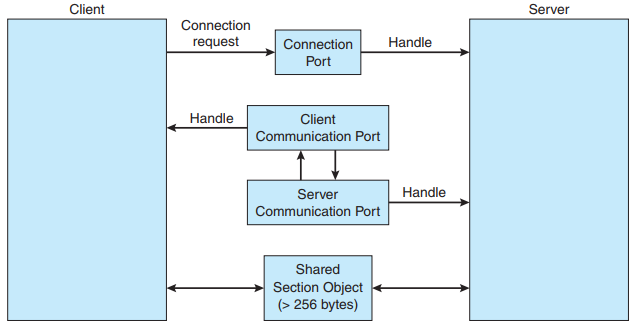
3.7.1 POSIX Shared Memory

* POSIX shared memory is organized using memory-mapped files, which associate the region of shared memory with a file. A process must first create a shared-memory object using the shm open() system call.
* Once the object is established, the ftruncate() function is used to configure the size of the object in bytes.
* Finally, the mmap() function establishes a memory-mapped file containing the shared-memory object. It also returns a pointer to the memory-mapped file that is used for accessing the shared-memory object.

3.7.2 Mach Message Passing

* The Mach kernel supports the creation and destruction of multiple tasks, which are similar to processes but have multiple threads of control and fewer associated resources.
* Most communication in Mach—including all intertask communication—is carried out by messages. Messages are sent to, and received from, mailboxes, which are called ports in Mach.
* Associated with each port is a collection of port rights that identify the capabilities necessary for a task to interact with the port.
* When a task is created, two special ports—the Task Self port and the Notify port—are also created. The kernel receives rights to the Task Self port, which allows a task to send messages to the kernel. The kernel can send notification of event occurrences to a task’s Notify port.

3.7.3 Windows

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

3.7.4 Pipes

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

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

2. If two-way communication is allowed, is it half duplex (data can travel only one way at a time) or full duplex (data can travel in both directions at the same time)?

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

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

3.7.4.1 Ordinary pipes (Unnamed, Anonymous)

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

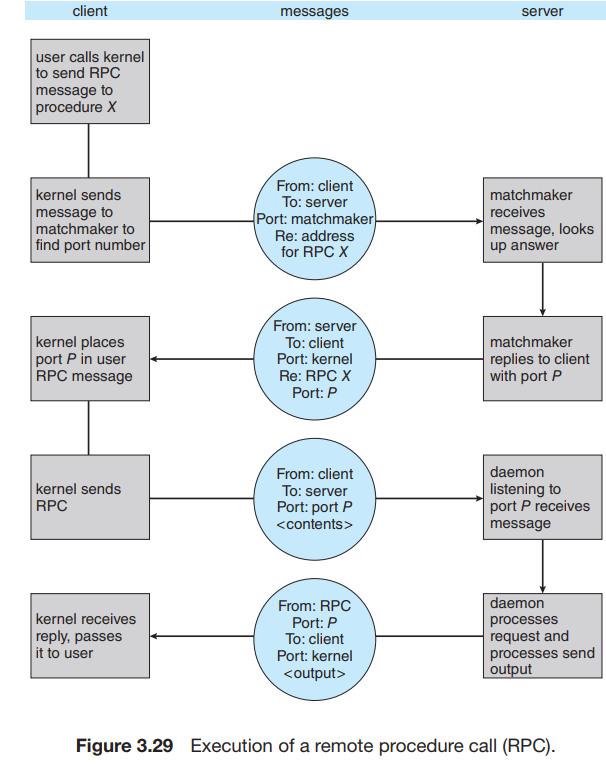
3.7.4.2 Named pipes

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

**3.8 Communication in Client-Server Systems**

3.8.1 Sockets

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

3.8.2 Remote Procedure Calls