(Latest Revision: Feb 14, 2022)
[2022/02/14: typo correction]
[2021/01/10: updated wording in section 3.7.2]
[2021/01/07: correction to wording in 3.4]
[2021/01/06: correction to wording in 3.3.2.1]
[2020/02/10: put in figure 3.8]
[2019/06/02: format changes]
[2019/05/20: inserted figures]
[2019/03/21: format changes]
[2019/02/10: current 2019 updates]

Chapter Three -- Processes -- Lecture Notes

- **3.0 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.

- **3.1 Process Concept** In a nutshell, a *process* is *a program in execution*. *Process* is actually a short-hand for the term *sequential process*. So we should think of a process as the (sequential) activity that happens when a program is executed, plus all the resources and information required to maintain that execution sequence.

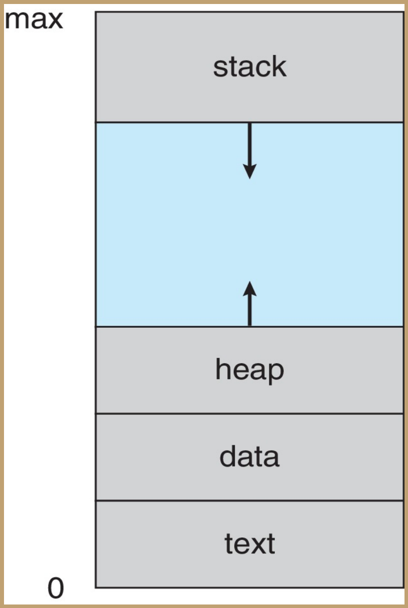


Figure 3.1: Layout of a process in memory

- **3.1.1 The Process**
  - The information required to maintain a process is called the *process context*. Process context is very extensive in a modern operating system, including, but not limited to:
    - program counter
    - CPU register contents
    - the parts of the memory layout of the process, such as the text, data, stack, and heap.
  - *Program* and *process* are very different ideas. A program is a passive text - typically an executable file in secondary storage. This program text is like a recipe.
  - When a program is loaded into memory and executed, it becomes a process. So a process is like the activity that happens when a cook follows a recipe to make something.
  - It is possible for two or more different processes to execute the same program at the same time. For example, five people who are logged in to the same computer might all be using the emacs text editor at the same time. In a situation like that, there would be five processes executing the same program concurrently. There would be only one copy of the program (text) in the computer's main memory. However each process would have its own data, program counter, CPU register contents, and so forth.

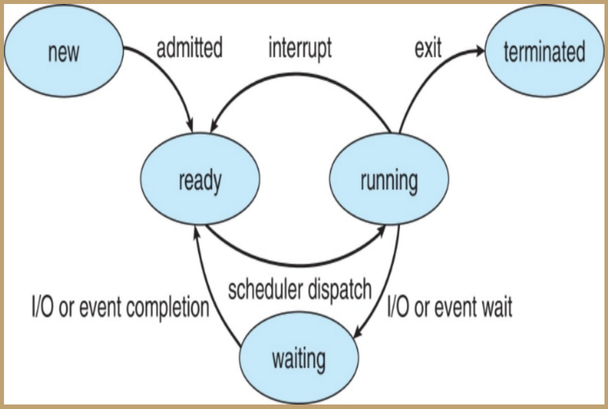


Figure 3.2: Diagram of process state

- **3.1.2 Process State** (See fig 3.2.) Sometimes people use the term *state* to broadly classify the current activity of a process. Once a program has been loaded into main memory, it is considered a process. The possible *states* of a process include *New* (being created), *Running* (executing in a CPU), *Waiting* (waiting for an event, like I/O or a signal), *Ready* (waiting to be assigned to a CPU), and *Terminated* (finished with execution). The *context* of a process includes *all* the information needed to maintain and keep track of the process. Therefore the context of a process includes its *state* and much, much more. Sometimes people (including the authors of our textbook) will say *state* when they actually mean *context*. So we have to be very careful when we see the word *state* used in reference to a process. The meaning depends upon the context (no pun intended) in which the word is used. ☺

  It is very important to understand that only one process can run at a time on a CPU (a single processor core).

- **3.1.3 Process Control Block** (See fig 3.3.)

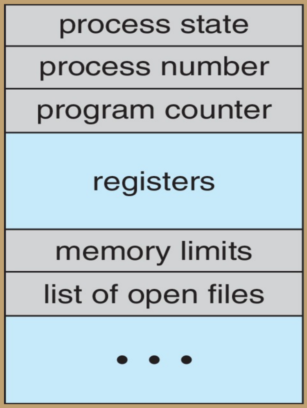


Figure 3.3: Process control block (PCB)

The OS keeps track of each process by representing it with a data structure. We usually call that data structure a "process control block" -- a PCB. The OS stores the entire context of the process in the PCB. The idea of a PCB is an abstraction. It is an abstract data type and different operating systems implement PCBs in different ways.

Some things in a PCB: the state (new, running, waiting, ready, terminated); program counter; the contents of CPU registers; scheduling information; memory-management information; accounting information; and I/O status information.

- **3.1.4 Threads** Chapter 4 is a whole chapter about threads, but here is some preliminary information.

  When two separate processes share as much context as they possibly can (or nearly as much) - we may call them separate *threads* of the same process. For example, suppose two processes share the same code (the same program), the same data and variables, the same open files. One may refer to them as two separate threads of execution within one single process. An example would be the case where one thread checks the spelling of a document while another thread of the same process accepts characters from a user typing at the keyboard.

- **3.2 Process Scheduling** Chapter 5 is a whole chapter about scheduling, but here is some preliminary information.

  Terms: *the degree of multiprogramming* is the number of processes currently in memory; *an I/O-bound* process makes frequent requests for I/O; a *CPU-bound* process 'wants' to run in the CPU most of the time, and makes few I/O requests.

  The OS *schedules* processes (selects the next one to run), while trying to balance the needs of I/O- and CPU-bound processes.

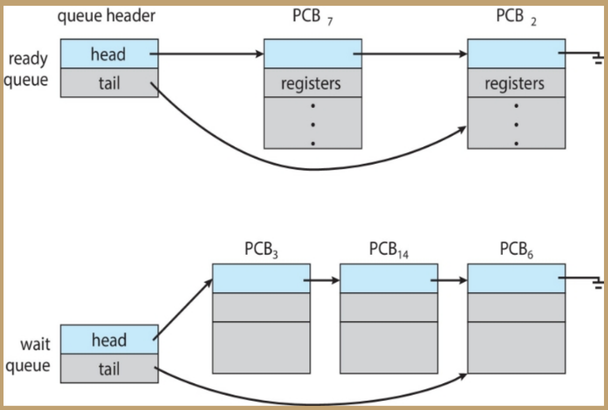


Figure 3.4: The ready queue and wait queues

- **3.2.1 Scheduling Queues** (See figure 3.4.) The OS maintains a *ready queue* containing the PCBs of processes that are ready to execute, and waiting to be *dispatched to* (selected for execution on) a CPU.

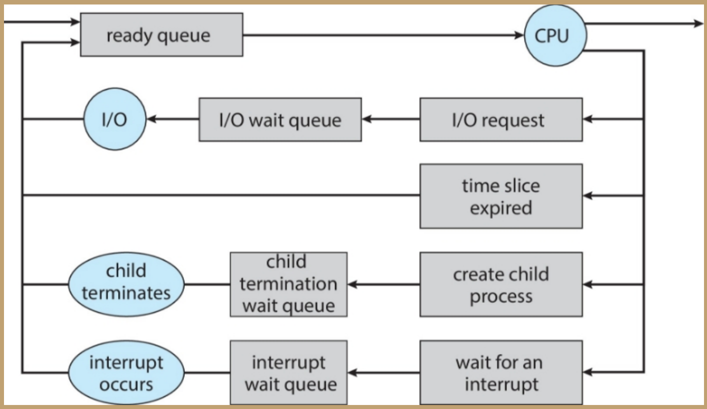


Figure 3.5: Queueing-diagram representation of process scheduling

(See figure 3.5.) The system also maintains other queues, *wait queues*, where processes wait for events, such as I/O completion, an interrupt, or the termination of a child process.

- **3.2.2 CPU Scheduling** The job of the (short term) CPU scheduler is to find and dispatch the next process to execute in a CPU. A process usually only executes for a brief *burst* in a CPU before it makes a system call or is interrupted for some other reason. The average burst may be only a few milliseconds. Therefore, to get good CPU utilization, the CPU scheduler has to finish its work in much less time than a few milliseconds. For example, if there are 10 successive CPU bursts of 3 milliseconds that follow each other as closely as possible, and if the average time taken by the scheduler to put each process in the CPU for its burst is 1 millisecond, then 25% of the elapsed time is spent on the work of the scheduler, and only 75% of the time on the actual work of the processes. It would go something like this: 1 ms for the scheduler, then 3 ms for a burst, then another 1 ms for the scheduler, then another 3 ms for a burst, and so on.

  There is an intermediate form of scheduling called *medium term* scheduling, or *swapping*. There's more information about swapping in Chapter 9. The basic idea is that when resources become scarce, the OS may select some of the processes in the primary memory and "remove" them from primary memory (*swap them out*), thus freeing up their resources for the use of the remaining processes. Eventually the OS *swaps them back in* and allows them to execute again.

- **3.2.3 Context Switch** When a process P is interrupted, the hardware automatically saves the value of P's program counter in a location where the operating system can find it and copy it. Depending on details of the computer hardware architecture, the hardware may also automatically save other items that are parts of P's context. When the operating system is in the CPU, after the interrupt, the OS will save any additional parts of P's context that need saving, and see that all the saved context is placed in P's PCB. The operating system will use the saved context later, when it *restores P*, that is, when it resumes the execution of P from the point where P was interrupted.

  Be sure you understand how the system works: **The hardware saves some context automatically** and then **the operating system does the rest of the saving of the context.**

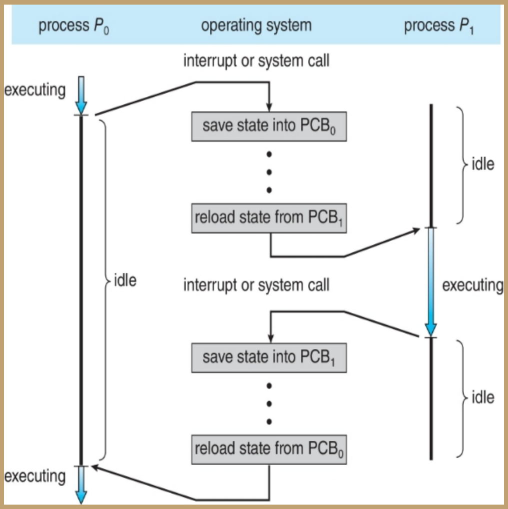


Figure 3.6: Diagram showing context switch from process to process

  The OS has to perform a *context switch* to change the process executing in the CPU. The OS makes sure that the context of the current process is saved in its PCB, and then it copies the context of the next process into the CPU (loads the next context).

  The speed of a context switch depends on features of the hardware and also features of the operating system. Typically it requires only several microseconds. (Remember milliseconds (ms) are thousandths of a second, and microseconds (μs) are millionths of a second.)

- **3.3 Operations on Processes**
  - **3.3.1 Process Creation** (See figures 3.8 and 3.10.) Most operating systems build a few processes "by hand" at boot time. After that, existing *parent processes*

create new *child processes* by making a system call. Child processes can use the system call to make their own children. Thus there are often extensive "family trees" of processes in modern computing systems.

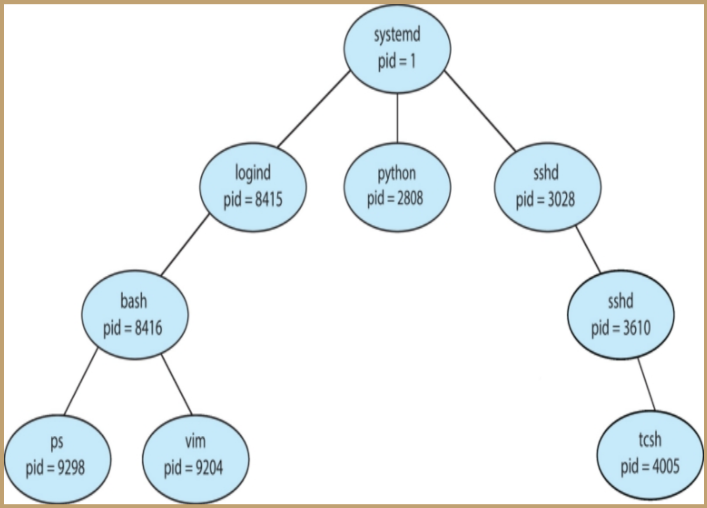


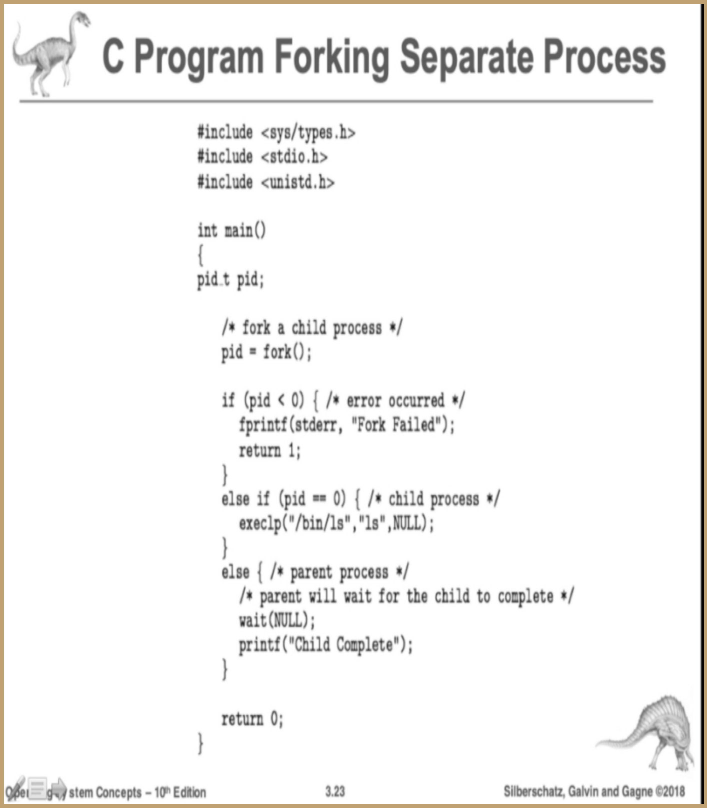
Figure 3.7: A tree of processes on a typical Linux system

`ps` is a command that works on unix command-line interfaces to display information about the current processes. Try

```
ps -el | more
```

to get a display on some "flavors" of unix through which you can advance page by page. (space bar goes forward, b takes you back, q quits) There's a `pstree` command that works on "flavors" of Linux.

The system call that creates a child process for a parent usually provides a way for the parent to pass information to the child process. Often a parent creates a child to perform a specific task, and it's important that the parent pass information to the child about what it wants done.



C Program Forking Separate Process

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

int main()
{
pid_t pid;

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

   if (pid < 0) { /* error occurred */
     fprintf(stderr, "Fork Failed");
     return 1;
   }
   else if (pid == 0) { /* child process */
     execlp("/bin/ls","ls",NULL);
   }
   else { /* parent process */
     /* parent will wait for the child to complete */
     wait(NULL);
     printf("Child Complete");
   }

   return 0;
}
```

Operating System Concepts – 10th Edition 3.23 Silberschatz, Galvin and Gagne ©2018

Figure 3.8: Creating a separate process using the UNIX fork() system call

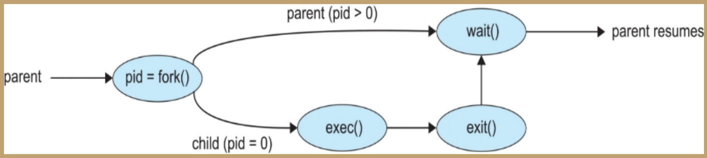


Figure 3.9: Process creation using the fork() system call

(See figure 3.9.) Most operating systems give parent processes the option of either suspending themselves until their child has terminated (example: unix wait() system call), or continuing to execute concurrently with the child.

There are other standard options.

- It is common for operating systems to provide a way to make the child a *clone* of the parent. In other words the child process can execute the same program as the parent process, using the same data (input). Example: variations of the unix fork() system call
- On the other hand, operating systems allow the parent to create a child that runs a different program on different data. Example: variations of the unix exec() system call

        Example: The Windows CreateProcess() system call is like a combination of fork() and exec().
        - An OS often allows parent processes to share some of their context with their children (like program, data, open files, and so on). Naturally, the OS will have to provide to the child any necessary items of context the child does not share with its parent.

    - **3.3.2 Process Termination** Usually, a process that is ready to stop executing makes a system call to the OS, asking to be terminated. Unix has the exit() system call. A parent that waits for its child to exit can collect a status code that the OS records when the child terminates.

      The effect of termination is that the OS deallocates and reclaims all the resources of the process, including memory allocations and open files.

      One process can sometimes make a system call that causes another process to terminate. Usually this happens when a process terminates a child or other descendent, for example, when the work the descendent is doing is no longer needed.

        - **3.3.2.1 Android Process Hierarchy** Android mobile devices are often relatively "resource poor." They lack large memories and powerful CPUs. Sometimes Android has to terminate processes in order to reclaim their resources for other processes. Android classifies processes according to an *importance hierarchy*. When it has to select a process for termination, it chooses the least important current process. Additionally Android saves the context of this *victim* on secondary memory, and if the user navigates back to the application, the system will restart it, making it go forward from where it was stopped.

- **3.4 Interprocess Communication** By definition, *cooperating* concurrent processes can affect each other, and *independent* concurrent processes don't share data.

  Programmers want to write programs that implement cooperating processes. Why?
    - It may be a way for processes to share resources or information.
    - On a multiprocessor, multiple processes can get more work done on an application in less time.
    - Whether or not it is more efficient, it may be that one gets a better program design by using cooperating processes.

  For the reasons given above, operating systems should facilitate process cooperation by providing one or more *interprocess communication* (IPC) facilities.

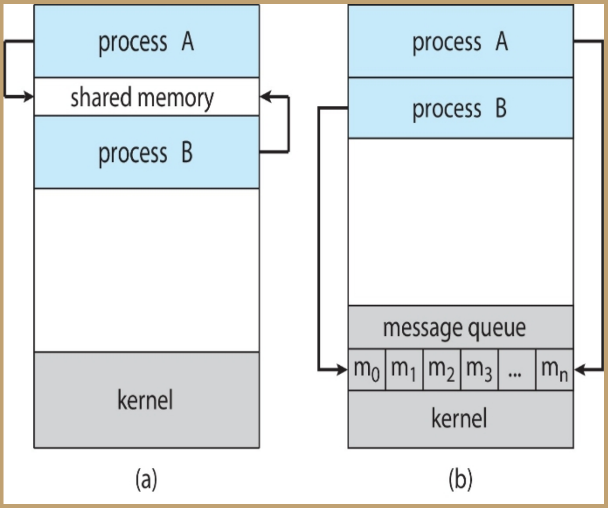


Figure 3.11: Communication models (a) shared memory (b) message passing

The two basic models of IPC are *shared memory* and *message passing*. (See figure 3.11)

In the shared-memory model, somehow a region of memory is established that two different processes are able to use. Then the two processes communicate just by writing on and reading from areas in the shared memory.

In the message-passing model, messages are exchanged by using system calls, and thus the OS serves as an intermediary, a pick-up and delivery service, for the communicating processes.

Message passing is usually relatively easy to implement, and easy for the communicating processes to use without problems, partly because going through the OS keeps the sender and receiver in synch with each other. It can be inefficient when messages are large, because of the need to somehow copy the message from sender memory to recipient memory, and also because sending and receiving messages requires the overhead of a system call.

Depending on the details of how memory is organized, the shared memory model of communication can be very fast because, after the region of shared memory is established, usually with system calls, there is no further need for the processes to use system calls to communicate. However communication is complicated by the fact that the processes have to properly synchronize their actions. For example, if one process is writing and the other is reading, how does the reader keep from getting ahead of the writer? If the reader reads too fast, it could get ahead of the writer and start reading "garbage" from memory where nothing has been written yet.

- **3.5 IPC In Shared-Memory Systems**

- The sample producer-consumer code provides us with an example of a protocol that allows two processes to use a shared-memory method of communication and to keep in synch with each other. When two processes cooperate, it's common for them to have a producer/consumer relationship. One process creates a stream of items for the other process to use in some way. Errors can occur if the communicating processes don't stay in synch.

  In the following example, after placing a new item, call it X, in a slot in the shared buffer, the producer makes sure to wait, if necessary, for the consumer to read item X before the producer places another item in that slot (which would overwrite X). Likewise, the consumer waits, if necessary, for the producer to place a new item in a buffer slot before the consumer reads from the slot.

  (The example code, which is written in a pseudo-code similar to the C programming language, is perhaps deceptively simple-looking. Algorithms that do these kinds of synchronization jobs can be very difficult to get right. All of Chapter 6 is about synchronization tools.)

- Producer-Consumer Problem Solution

```
-----------------------------------------
Assumptions: Loads and Stores Are Atomic.
(We'll learn about that in Chapter 6. )
-----------------------------------------
/* The variables defined in this section are placed in a block of
      memory that is shared by the producer and consumer
processes */

       /* This declares BUFFER_SIZE and sets it equal to 10.
          BUFFER_SIZE is used as a constant in the program.
          It represents the size of the buffer shared by the two
          processes.  */
#define BUFFER_SIZE 10

         /* The buffer will be an array of individual pieces of
data,
            called 'items.' Here we define the data type of an
item,
            and name the data type 'itemType.' (In the C
programming
            language, a struct is like a class with data members
            only. A struct has no methods.) */
typedef struct
(
       /* If this was real code, here we would define whatever
fields
          of data we want our data items to have.  Because this
is just
          an example, we don't put anything specific here. */
) itemType ;

          /* This declares an array named 'buffer' with slots
that
```

```
             are objects of type itemType.  The number of slots
in
             the array is BUFFER_SIZE. */
itemType buffer [BUFFER_SIZE] ;

          /* The next two statements declare a couple of
variables.
             The first represents the location in the buffer into
which
             the producer will place its next item.
             The second variable represents the location from
which the
             consumer will take its next item. Both variables are
initialized
             to the value 0.  */
int in = 0 ;
int out = 0 ;

/* To communicate, each process executes its own piece of code.
   The producer process repeatedly makes an item and copies it
   into the next available slot in the buffer.  The consumer
process
   repeatedly reads an item from the next available slot.  When
either
   process gets to the end of the array, it starts over at the
beginning
   of the array, continuing with the same actions.
*/

/* As a simplification for purposes of this example, we use
   "while (true)" to make infinite loops.  That way
   the producer and consumer "do their dance eternally."
   In a real application, we'd customize the code to stop
   the processes when required. */

--------------------------------
(producer code)

while (true)
{
          /* The producer creates the next item with.
             the local function 'makeItem' and stores the
             item in a local variable 'nextProduced.' This
             function and variable are not in the
             shared memory. */
  nextProduced = makeItem() ;

         /* This may look mysterious, but the loop condition
            below just means that there's only one empty slot
            left in the buffer. So the meaning of the while-loop
            below is that if there a too few empty slots, the
            producer will be delayed until there are more empty
slots.
            In other words, if it looks like the producer is
close to
            catching up with the consumer, then there's a danger
that
```

```
            the producer could start overwriting data in the
buffer
            - data that the consumer has not yet read.
            So the producer waits until the consumer has read
            more items from the buffer. */
  while (  ( (in+1) % BUFFER_SIZE ) == out )
         /* do nothing */   ;

     /* Copy the item into the next available slot in the buffer.
        The buffer is in shared memory. */
  buffer[in] = nextProduced ;

     /* Advance the variable that tracks the next slot to use.
The
        use of the % operator assures that the value of the
        variable will be set back to 0 after reaching the end of
the
        buffer. */
  in = (in+1) % BUFFER_SIZE ;
}

--------------------------------
(consumer code)

while (true)
{
         /* delay while the buffer is 'empty,'
            meaning that there are no new items
            to read.  In this situation, the consumer
            is waiting, if necessary, for the producer
            to catch up. */
  while ( in == out )
         /* do nothing */ ;

        /* Copy the next available item into a local variable. */
  nextConsumed = buffer[out] ;

       /* Advance the variable that tracks the next slot to use.
*/
  out = (out+1) % BUFFER_SIZE ;

           /* Call a local function that 'consumes' the item,
              making whatever use of the item the consumer
process
              needs. */
  consume(nextConsumed) ;

}
--------------------------------
```

- **3.6 IPC In Message-Passing Systems**

The message-passing model has its advantages. An important example is in computer networks, because two processes on different computers often have no means to share memory.

At the minimum, operating systems must provide an operation to send a message, and one to receive a message. If messages are allowed to vary greatly in size, it is convenient for the programmers who write calls to these operations. However, from the point of view of the designer of the communication facility, it's easier to implement operations that assume all messages are the same size.

- **3.6.1 Naming** Message-based communication may be direct or indirect, meaning that processes may address messages directly to each other, or address them instead to data structures known as *mailboxes* or *ports*. Mailboxes have some similarities to files or devices, so they can have owners, and permission settings.

  The details of how the sending and receiving functions work (the *semantics*) vary from system to system, and one system may offer several versions of message-passing, with different semantics for different versions.

- **3.6.2 Synchronization** Both the send and the receive operation can be either *blocking* or *non-blocking*. With a blocking send or receive, it works like I/O. The process has to wait until the message is sent or received. With the non-blocking send, the process returns immediately after calling the send operation, and is free to continue with other work. With the non-blocking receive, the process returns immediately from the call with either a valid message, or a null message, in the case that no message was available at the time of the call.

  The non-blocking versions give the processes more options, but it's more complicated to write the programs that use non-blocking send() and receive() operations, basically because processes have to figure out what to do during times when messages are in transit. (There are also blocking and non-blocking forms of I/O, and non-blocking I/O has the same kinds of challenges as non-blocking send() & receive().)

- **3.6.3 Buffering** Designers and implementors of messaging systems have to decide on how much *buffering* will be supported. Will there be *zero capacity* to queue (temporarily store) messages awaiting delivery? Will there be *bounded capacity*? *Unbounded capacity*?

- **3.7 Examples of IPC Systems**

  - **3.7.1 POSIX Shared Memory**
    - Use of shm_open() system call, ftruncate() function, and mmap() function to create a shared memory object, and establish a memory mapped file containing the shared memory object.
    - Other processes besides the creator, processes that know the name of the shared memory object, can open it and map it to a file with shm_open() and mmap().

  - This is how multiple processes can share memory for the purpose of communicating, under the POSIX API.
  - There is also a shm_unlink() function for removing a shared-memory object.

- **3.7.2 Mach Message Passing**
  - Mach supports message passing.
  - Mach employs message mailboxes called ports.
  - System calls are implemented by messaging.
  - Remote Procedure Call is implemented with messaging.
  - Processes use the mach_msg() to send or receive a message.
  - The creator is the default owner of the port.
  - Rights on ports can be transferred.
  - Messages consist of a fixed-length header and a variable-sized body that can be structured or unstructured.
  - Message headers include destination and return port addresses
  - A disadvantage of message systems in general is double-copying: often messages are copied from sender memory to kernel memory, and then again from kernel memory to recipient memory.
  - In the Mach system, memory-mapping techniques are used to avoid some cases of double-copying. It will work within a system that has shared memory.

- **3.7.3 Windows**

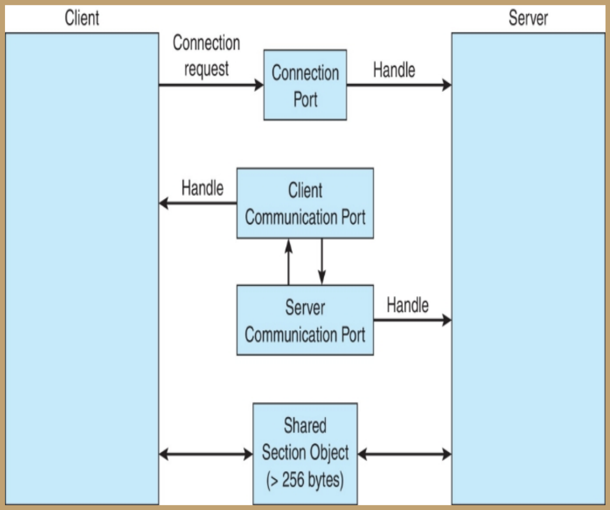

Client
Server
Connection request
Connection Port
Handle
Handle
Client Communication Port
Server Communication Port
Handle
Shared Section Object (> 256 bytes)

Figure 3.19: Advanced local procedure calls in Windows

  - user processes communicate with subsystems (multiple operating environments) via a message-passing facility called "advanced local procedure call" -- ALPC.
  - Windows uses ports like Mach ports.

        - User processes establish communication by functioning as clients of the Windows subsystem server.
        - Windows has three styles of message passing. One involves double copying, one uses shared memory to avoid double copying, and one allows a server to read and write the address space of the client directly.
        - The ALPC facility is below and transparent to the level of the Windows API.

    - **3.7.4 Pipes**
        - **3.7.4.1 Ordinary Pipes**
            - Ordinary pipes are an IPC mechanism used for a long time in unix to allow a pair of processes on the same host to operate as a producer and consumer. The pipe is a data structure. One process can write to it, and the other can read from it.

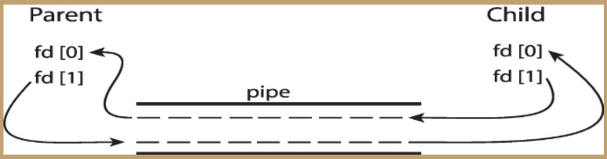


Figure 3.20: File descriptors for an ordinary pipe

            - Unix has a system call pipe() that creates the data structure and gives access to two small integers called file descriptors, one used to get access to the writing end of the pipe, and one to get access to the reading end.
            - A unix child process inherits the open files and pipes of its parent, so parent and child can communicate through an ordinary pipe. However there's no mechanism other than inheritance for allowing processes to share an ordinary pipe.
            - Windows employs "anonymous pipes" that are similar to the unix ordinary pipes.
            - Ordinary pipes are unidirectional.

        - **3.7.4.2 Named Pipes**
            - Unix has something called "named pipes" (aka FIFO's) that are more powerful and feature-rich than ordinary pipes.
            - Windows has named pipes too, but they are implemented differently from the way they are implemented in unix.
            - Unix named pipes appear as objects in the file system - if you go to a directory and do an 'ls', one of the things displayed might be the name of a FIFO.
            - FIFO's allow bidirectional communication, but only in one direction at a time.

        - Two processes have to reside on the same machine in order to communicate using a FIFO.
        - The Windows versions of named pipes are more versatile. They support full duplex communication (it can flow in both directions at the same time). Processes on different machines can communicate using a Windows named pipe. On both Windows and Unix machines, a named pipe can be used to transmit a simple stream of bytes. Windows named pipes can also be used to transmit structured messages.

- **3.8 Communication in Client-Server Systems**
    - **3.8.1 Sockets**

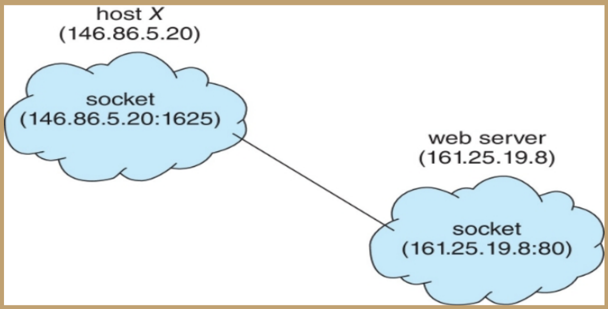


Figure 3.26: Communication using sockets

        - Processes communicating over a network may employ a "socket interface" - data structures called sockets are used as endpoints of communication - sort of like telephone handsets that you can talk into and hear out of.
        - There's no analog of a "ring tone" in networking, so servers have to wait with "their ears to" a socket, until a client request comes in.
        - A socket connection is associated with two numbers - a network address (usually an IP number) and a port number. The network address identifies the host computer and the port number identifies the specific process on the host using the socket.
        - Port numbers below 1024 are reserved for servers offering well-known services like e-mail transmission, file transfer and hypertext transfer.
        - Socket communication is considered low-level in one sense - it supports sending and receiving only unstructured streams of bytes, not structured data types and objects.
    - **3.8.2 Remote Procedure Calls** (See figure 3.29.)
        - RPC supports transmission of structured data.
        - RPC semantics allow a client to invoke a procedure on a remote host, connected only via a network, in the same way the client would call a procedure locally.

- When the client calls a non-local procedure, it results in a call to the RPC system, which calls a special stub procedure on the client side. The RPC system passes the function parameters to the stub.

- The stub contacts a port on a server, marshals parameters, and sends them in a message to the server.

- A stub on the server side receives the message, invokes the procedure on the server side, and returns any needed results back to the client stub.
- The function call made by the client returns normally - just as if it was a call to a local procedure.
- In marshaling and transmitting parameters, care must be taken to allow for the fact that the two different hosts may not represent a data type in the same way.
- Implementations of RPC have to be able to deal with the possibility that network packets may be lost or duplicated, because these are common occurrences on computer networks, The underlying protocol software can manage this by, in essence, placing serial numbers in each message and requiring acknowledgment to the sender after messages are received.
- Port numbers of RPC servers may be *hardwired*, which means they are compiled explicitly into client stub code. Another option is for client stubs to determine port numbers dynamically by querying a matchmaker daemon that listens on a well-known port.
- RPC can be used to implement remote file service - like NFS.

- **3.8.2.1 Android RPC** The Android OS includes RPCs among its IPC mechanisms. A process can use an RPC to request a service from another process. An RPC may be used, even if both processes are running on the same system.

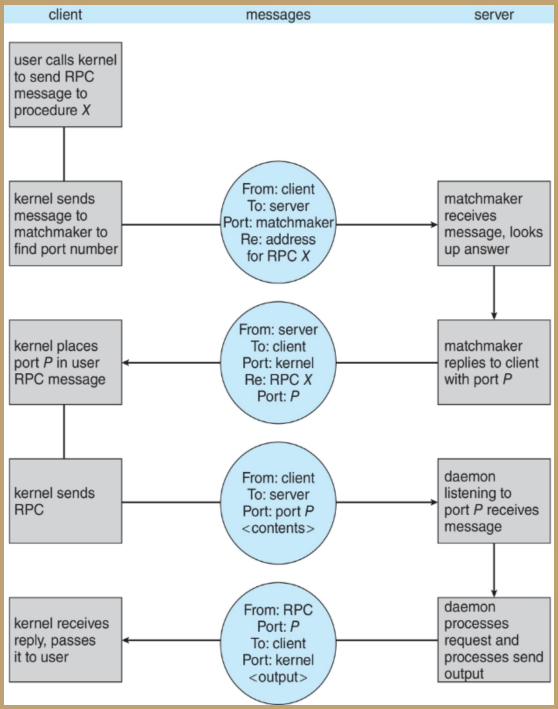

client
messages
server
user calls kernel to send RPC message to procedure X
kernel sends message to matchmaker to find port number
From: client To: server Port: matchmaker Re: address for RPC X
matchmaker receives message, looks up answer
kernel places port P in user RPC message
From: server To: client Port: kernel Re: RPC X Port: P
matchmaker replies to client with port P
kernel sends RPC
From: client To: server Port: port P <contents>
daemon listening to port P receives message
kernel receives reply, passes it to user
From: RPC Port: P To: client Port: kernel <output>
daemon processes request and processes send output

Figure 3.29: Execution of a remote procedure call (RPC)