Processes

References:

1. Abraham Silberschatz, Greg Gagne, and Peter Baer Galvin, "Operating System Concepts, Ninth Edition ", Chapter 3

3.1 Process Concept

- A process is an instance of a program in execution.
- Batch systems work in terms of "jobs". Many modern process concepts are still expressed in terms of jobs, (e.g. job scheduling), and the two terms are often used interchangeably.

3.1.1 The Process

- Process memory is divided into four sections as shown in Figure 3.1 below:
  - The text section comprises the compiled program code, read in from non-volatile storage when the program is launched.
  - The data section stores global and static variables, allocated and initialized prior to executing main.
  - The heap is used for dynamic memory allocation, and is managed via calls to new, delete, malloc, free, etc.
  - The stack is used for local variables. Space on the stack is reserved for local variables when they are declared (at function entrance or elsewhere, depending on the language), and the space is freed up when the variables go out of scope. Note that the stack is also used for function return values, and the exact mechanisms of stack management may be language specific.
  - Note that the stack and the heap start at opposite ends of the process's free space and grow towards each other. If they should ever meet, then either a stack overflow error will occur, or else a call to new or malloc will fail due to insufficient memory available.
- When processes are swapped out of memory and later restored, additional information must also be stored and restored. Key among them are the program counter and the value of all program registers.

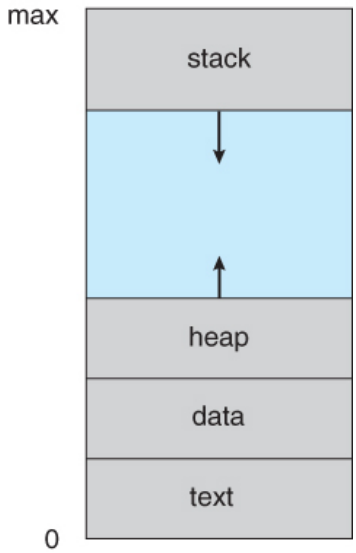


Figure 3.1 - A process in memory

3.1.2 Process State

- Processes may be in one of 5 states, as shown in Figure 3.2 below.
  - **New** - The process is in the stage of being created.
  - **Ready** - The process has all the resources available that it needs to run, but the CPU is not currently working on this process's instructions.
  - **Running** - The CPU is working on this process's instructions.
  - **Waiting** - The process cannot run at the moment, because it is waiting for some resource to become available or for some event to occur. For example the process may be waiting for keyboard input, disk access request, inter-process messages, a timer to go off, or a child process to finish.
  - **Terminated -** The process has completed.
- The load average reported by the "w" command indicate the average number of processes in the "Ready" state over the last 1, 5, and 15 minutes, i.e. processes who have everything they need to run but cannot because the CPU is busy doing something else.
- Some systems may have other states besides the ones listed here.

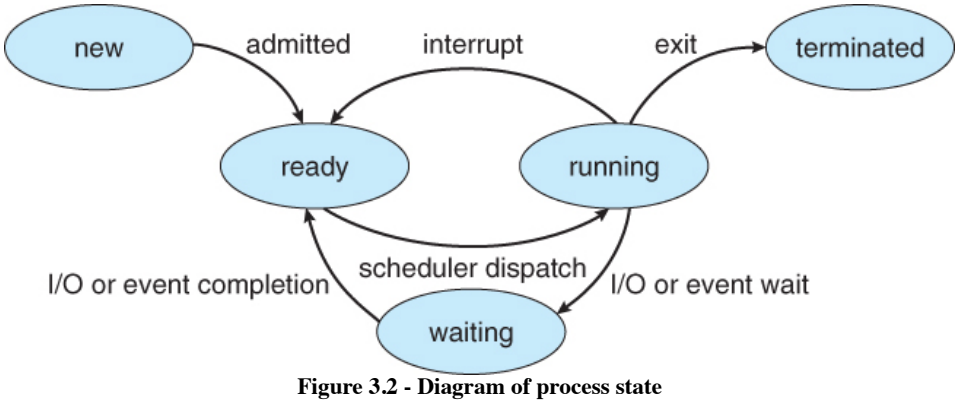


Figure 3.2 - Diagram of process state

3.1.3 Process Control Block

For each process there is a Process Control Block, PCB, which stores the following (types of) process-specific information, as illustrated in Figure 3.1. (Specific details may vary from system to system.)

- **Process State** - Running, waiting, etc., as discussed above.
- **Process ID**, and parent process ID.
- **CPU registers and Program Counter** - These need to be saved and restored when swapping processes in and out of the CPU.
- **CPU-Scheduling information** - Such as priority information and pointers to scheduling queues.
- **Memory-Management information** - E.g. page tables or segment tables.
- **Accounting information** - user and kernel CPU time consumed, account numbers, limits, etc.
- **I/O Status information** - Devices allocated, open file tables, etc.

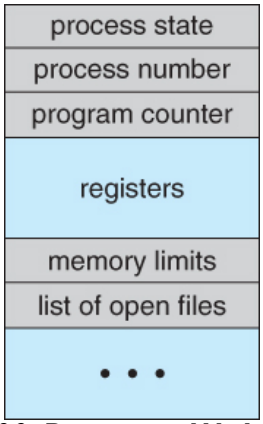


Figure 3.3 - Process control block (PCB)

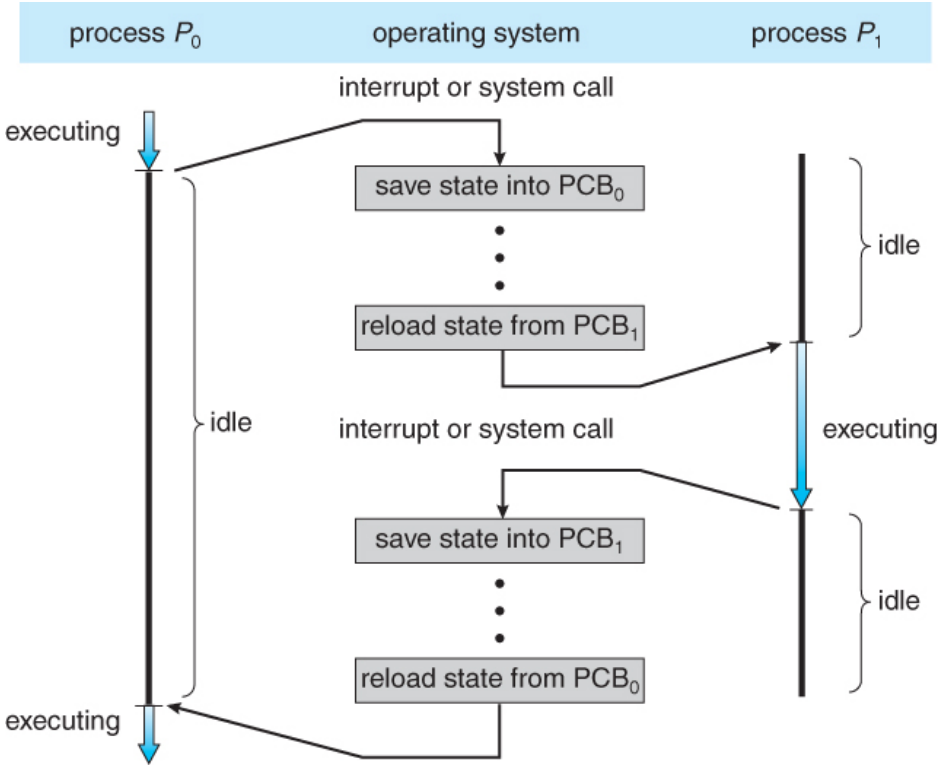


Figure 3.4 - Diagram showing CPU switch from process to process

PROCESS REPRESENTATION IN LINUX

The process control block in the Linux operating system is represented by the C structure `task_struct`. This structure contains all the necessary information for representing a process, including the state of the process, scheduling and memory-management information, list of open files, and pointers to the process's parent and any of its children. (A process's *parent* is the process that created it; its *children* are any processes that it creates.) Some of these fields include:

```
pid_t pid; /* process identifier */
long state; /* state of the process */
unsigned int time_slice /* scheduling information */
struct task_struct *parent; /* this process's parent */
struct list_head children; /* this process's children */
struct files_struct *files; /* list of open files */
struct mm_struct *mm; /* address space of this process */
```

For example, the state of a process is represented by the field `long state` in this structure. Within the Linux kernel, all active processes are represented using a doubly linked list of `task_struct`, and the kernel maintains a pointer —`current`—to the process currently executing on the system. This is shown in Figure 3.5.

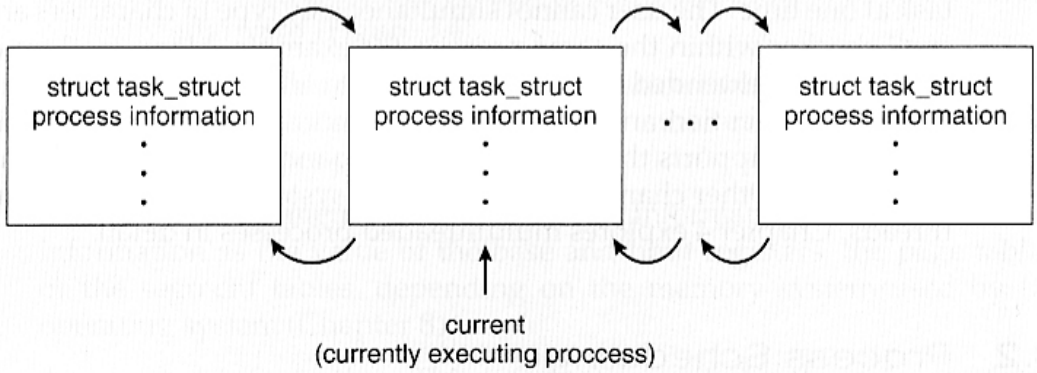


Figure 3.5 Active processes in Linux.

As an illustration of how the kernel might manipulate one of the fields in the `task_struct` for a specified process, let's assume the system would like to change the state of the process currently running to the value `new_state`. If `current` is a pointer to the process currently executing, its state is changed with the following:

```
current->state = new_state;
```

Unnumbered side bar

Digging Deeper: The Linux task_struct definition in sched.h (See also the top of that file.)

3.1.4 Threads

- Modern systems allow a single process to have multiple threads of execution, which execute concurrently. Threads are covered extensively in the next chapter.

3.2 Process Scheduling

- The two main objectives of the process scheduling system are to keep the CPU busy at all times and to deliver "acceptable" response times for all programs, particularly for interactive ones.
- The process scheduler must meet these objectives by implementing suitable policies for swapping processes in and out of the CPU.
- (Note that these objectives can be conflicting. In particular, every time the system steps in to swap processes it takes up time on the CPU to do so, which is thereby "lost" from doing any useful productive work.)

3.2.1 Scheduling Queues

- All processes are stored in the **job queue.**
- Processes in the Ready state are placed in the **ready queue.**
- Processes waiting for a device to become available or to deliver data are placed in **device queues**. There is generally a separate device queue for each device.
- Other queues may also be created and used as needed.

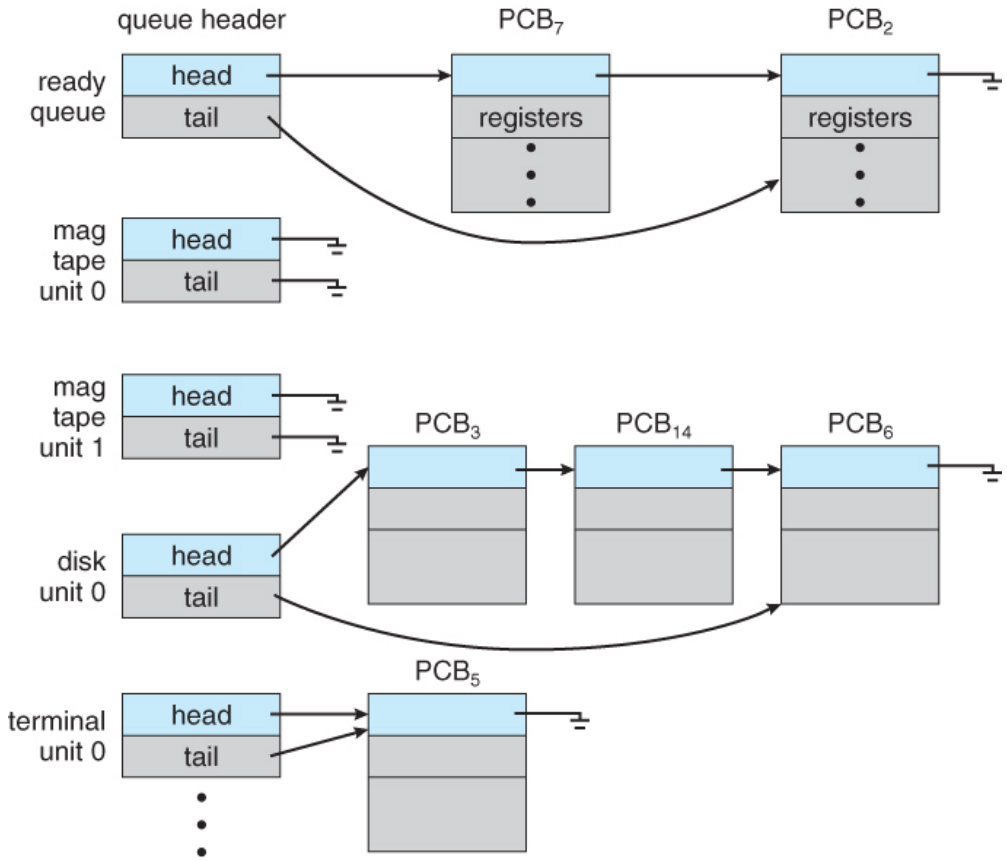


Figure 3.5 - The ready queue and various I/O device queues

3.2.2 Schedulers

- A **long-term scheduler** is typical of a batch system or a very heavily loaded system. It runs infrequently, (such as when one process ends selecting one more to be loaded in from disk in its place), and can afford to take the time to implement intelligent and advanced scheduling algorithms.
- The **short-term scheduler**, or CPU Scheduler, runs very frequently, on the order of 100 milliseconds, and must very quickly swap one process out of the CPU and swap in another one.
- Some systems also employ a **medium-term scheduler**. When system loads get high, this scheduler will swap one or more processes out of the ready queue system for a few seconds, in order to allow smaller faster jobs to finish up quickly and clear the system. See the differences in Figures 3.7 and 3.8 below.
- An efficient scheduling system will select a good **process mix** of **CPU-bound** processes and **I/O bound** processes.

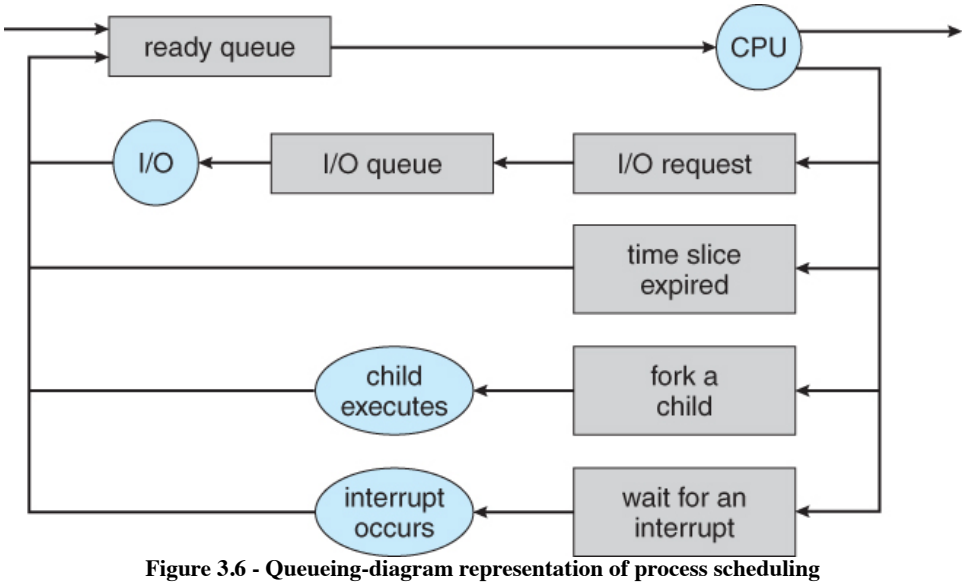


Figure 3.6 - Queueing-diagram representation of process scheduling

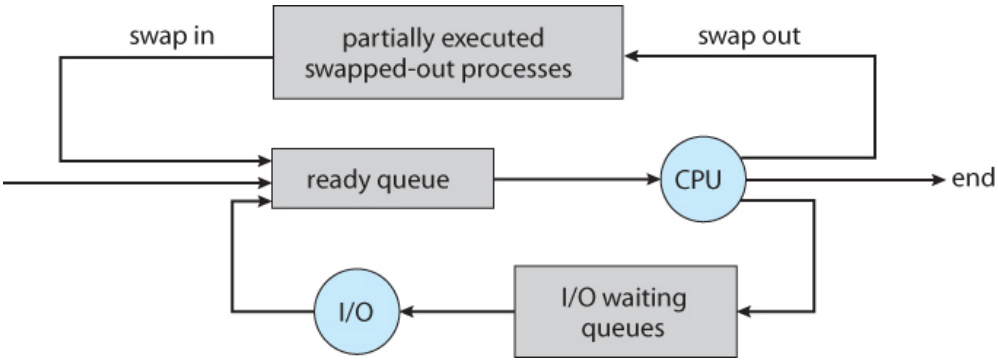


Figure 3.7 - Addition of a medium-term scheduling to the queueing diagram

3.2.3 Context Switch

- Whenever an interrupt arrives, the CPU must do a **state-save** of the currently running process, then switch into kernel mode to handle the interrupt, and then do a **state-restore** of the interrupted process.
- Similarly, a **context switch** occurs when the time slice for one process has expired and a new process is to be loaded from the ready queue. This will be instigated by a timer interrupt, which will then cause the current process's state to be saved and the new process's state to be restored.
- Saving and restoring states involves saving and restoring all of the registers and program counter(s), as well as the process control blocks described above.
- Context switching happens VERY VERY frequently, and the overhead of doing the switching is just lost CPU time, so context switches (state saves & restores) need to be as fast as possible. Some hardware has special provisions for speeding this up, such as a single machine instruction for saving or restoring all registers at once.
- Some Sun hardware actually has multiple sets of registers, so the context switching can be speeded up by merely switching which set of registers are currently in use. Obviously there is a limit as to how many processes can be switched between in this manner, making it attractive to implement the medium-term scheduler to swap some processes out as shown in Figure 3.8 above.

MULTITASKING IN MOBILE SYSTEMS

Because of the constraints imposed on mobile devices, early versions of iOS did not provide user-application multitasking; only one application runs in the foreground and all other user applications are suspended. Operating-system tasks were multitasked because they were written by Apple and well behaved. However, beginning with iOS 4, Apple now provides a limited form of multitasking for user applications, thus allowing a single foreground application to run concurrently with multiple background applications. (On a mobile device, the **foreground** application is the application currently open and appearing on the display. The **background** application remains in memory, but does not occupy the display screen.) The iOS 4 programming API provides support for multitasking, thus allowing a process to run in the background without being suspended. However, it is limited and only available for a limited number of application types, including applications

- running a single, finite-length task (such as completing a download of content from a network);
- receiving notifications of an event occurring (such as a new email message);
- with long-running background tasks (such as an audio player.)

Apple probably limits multitasking due to battery life and memory use concerns. The CPU certainly has the features to support multitasking, but Apple chooses to not take advantage of some of them in order to better manage resource use.

Android does not place such constraints on the types of applications that can run in the background. If an application requires processing while in the background, the application must use a **service**, a separate application component that runs on behalf of the background process. Consider a streaming audio application: if the application moves to the background, the service continues to send audio files to the audio device driver on behalf of the background application. In fact, the service will continue to run even if the background application is suspended. Services do not have a user interface and have a small memory footprint, thus providing an efficient technique for multitasking in a mobile environment.

3.3 Operations on Processes

3.3.1 Process Creation

- Processes may create other processes through appropriate system calls, such as **fork** or **spawn**. The process which does the creating is termed the **parent** of the other process, which is termed its **child**.
- Each process is given an integer identifier, termed its **process identifier**, or PID. The parent PID (PPID) is also stored for each process.
- On typical UNIX systems the process scheduler is termed **sched**, and is given PID 0. The first thing it does at system startup time is to launch **init**, which gives that process PID 1. Init then launches all system daemons and user logins, and becomes the ultimate parent of all other processes. Figure 3.9 shows a typical process tree for a Linux system, and other systems will have similar though not identical trees:

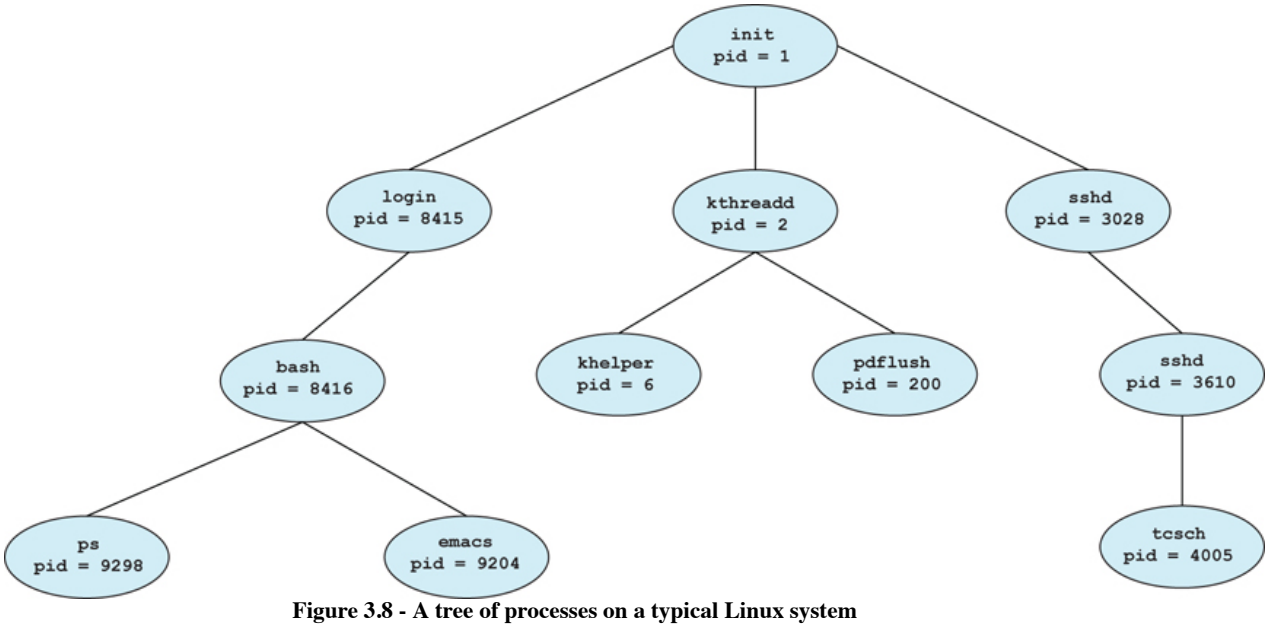


Figure 3.8 - A tree of processes on a typical Linux system

- Depending on system implementation, a child process may receive some amount of shared resources with its parent. Child processes may or may not be limited to a subset of the resources originally allocated to the parent, preventing runaway children from consuming all of a certain system resource.
- There are two options for the parent process after creating the child:
    1. Wait for the child process to terminate before proceeding. The parent makes a wait( ) system call, for either a specific child or for any child, which causes the parent process to block until the wait( ) returns. UNIX shells normally wait for their children to complete before issuing a new prompt.
    2. Run concurrently with the child, continuing to process without waiting. This is the operation seen when a UNIX shell runs a process as a background task. It is also possible for the parent to run for a while, and then wait for the child later, which might occur in a sort of a parallel processing operation. (E.g. the parent may fork off a number of children without waiting for any of them, then do a little work of its own, and then wait for the children.)
- Two possibilities for the address space of the child relative to the parent:
    1. The child may be an exact duplicate of the parent, sharing the same program and data segments in memory. Each will have their own PCB, including program counter, registers, and PID. This is the behavior of the **fork** system call in UNIX.
    2. The child process may have a new program loaded into its address space, with all new code and data segments. This is the behavior of the **spawn** system calls in Windows. UNIX systems implement this as a second step, using the **exec** system call.
- Figures 3.10 and 3.11 below shows the fork and exec process on a UNIX system. Note that the **fork** system call returns the PID of the processes child to each process - It returns a zero to the child process and a non-zero child PID to the parent, so the return value indicates which process is which. Process IDs can be looked up any time for the current process or its direct parent using the getpid( ) and getppid( ) system calls respectively.

```c
#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");
      exit(-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");
      exit(0);
    }
}
```

Figure 3.10 C program forking a separate process.

Figure 3.9 Creating a separate process using the UNIX fork( ) system call.

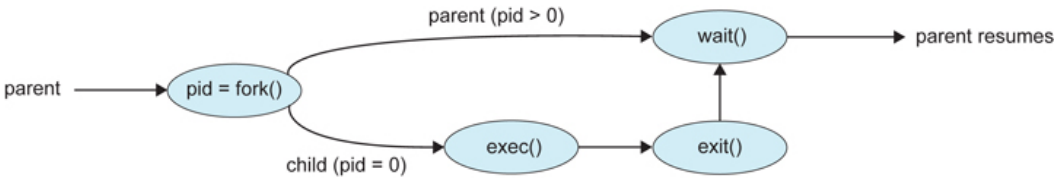


Figure 3.10 - Process creation using the fork( ) system call

- Related man pages:
  - fork(2)
  - exec(3)
  - wait(2)

- Figure 3.12 shows the more complicated process for Windows, which must provide all of the parameter information for the new process as part of the forking process.

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

int main(VOID)
{
STARTUPINFO si;
PROCESS_INFORMATION pi;

    // allocate memory
    ZeroMemory(&si, sizeof(si));
    si.cb = sizeof(si);
    ZeroMemory(&pi, sizeof(pi));

    // create child process
    if (!CreateProcess(NULL, // use command line
     "C:\\WINDOWS\\system32\\mspaint.exe", // command line
     NULL, // don't inherit process handle
     NULL, // don't inherit thread handle
     FALSE, // disable handle inheritance
     0, // no creation flags
     NULL, // use parent's environment block
     NULL, // use parent's existing directory
     &si,
     &pi))
    {
      fprintf(stderr, "Create Process Failed");
      return -1;
    }
    // parent will wait for the child to complete
    WaitForSingleObject(pi.hProcess, INFINITE);
    printf("Child Complete");

    // close handles
    CloseHandle(pi.hProcess);
    CloseHandle(pi.hThread);
}
```

Figure 3.12 Creating a separate process using the Win32 API.

Figure 3.11

3.3.2 Process Termination

- Processes may request their own termination by making the **exit( )** system call, typically returning an int. This int is passed along to the parent if it is doing a **wait( )**, and is typically zero on successful completion and some non-zero code in the event of problems.
  - child code:

```
int exitCode;
exit( exitCode );  // return exitCode; has the same effect when executed from main( )
```

  - parent code:

```
pid_t pid;
int status
pid = wait( &status );
// pid indicates which child exited. exitCode in low-order bits of status
// macros can test the high-order bits of status for why it stopped
```

- Processes may also be terminated by the system for a variety of reasons, including:
  - The inability of the system to deliver necessary system resources.
  - In response to a KILL command, or other un handled process interrupt.

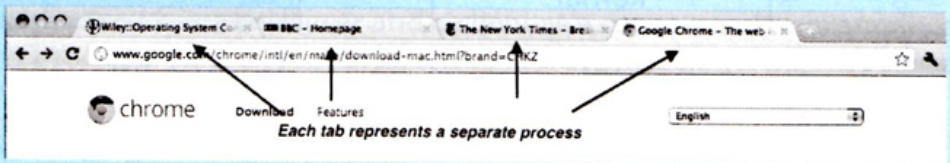
    - A parent may kill its children if the task assigned to them is no longer needed.
    - If the parent exits, the system may or may not allow the child to continue without a parent. (On UNIX systems, orphaned processes are generally inherited by init, which then proceeds to kill them. The UNIX ***nohup*** command allows a child to continue executing after its parent has exited.)
- When a process terminates, all of its system resources are freed up, open files flushed and closed, etc. The process termination status and execution times are returned to the parent if the parent is waiting for the child to terminate, or eventually returned to init if the process becomes an orphan. (Processes which are trying to terminate but which cannot because their parent is not waiting for them are termed **zombies**. These are eventually inherited by init as orphans and killed off. Note that modern UNIX shells do not produce as many orphans and zombies as older systems used to.)

3.4 Interprocess Communication

- **Independent Processes** operating concurrently on a systems are those that can neither affect other processes or be affected by other processes.
- **Cooperating Processes** are those that can affect or be affected by other processes. There are several reasons why cooperating processes are allowed:
    - Information Sharing - There may be several processes which need access to the same file for example. (e.g. pipelines.)
    - Computation speedup - Often a solution to a problem can be solved faster if the problem can be broken down into sub-tasks to be solved simultaneously (particularly when multiple processors are involved.)
    - Modularity - The most efficient architecture may be to break a system down into cooperating modules. (E.g. databases with a client-server architecture.)
    - Convenience - Even a single user may be multi-tasking, such as editing, compiling, printing, and running the same code in different windows.

MULTIPROCESS ARCHITECTURE—CHROME BROWSER

Many websites contain active content such as JavaScript, Flash, and HTML5 to provide a rich and dynamic web-browsing experience. Unfortunately, these web applications may also contain software bugs, which can result in sluggish response times and can even cause the web browser to crash. This isn't a big problem in a web browser that displays content from only one website. But most contemporary web browsers provide tabbed browsing, which allows a single instance of a web browser application to open several websites at the same time, with each site in a separate tab. To switch between the different sites , a user need only click on the appropriate tab. This arrangement is illustrated below:



A problem with this approach is that if a web application in any tab crashes, the entire process—including all other tabs displaying additional websites —crashes as well.

Google's Chrome web browser was designed to address this issue by using a multiprocess architecture. Chrome identifies three different types of processes: browser, renderers, and plug-ins.

- The **browser** process is responsible for managing the user interface as well as disk and network I/O. A new browser process is created when Chrome is started. Only one browser process is created.
- **Renderer** processes contain logic for rendering web pages. Thus, they contain the logic for handling HTML, Javascript, images, and so forth. As a general rule, a new renderer process is created for each website opened in a new tab, and so several renderer processes may be active at the same time.
- A **plug-in** process is created for each type of plug-in (such as Flash or QuickTime) in use. Plug-in processes contain the code for the plug-in as well as additional code that enables the plug-in to communicate with associated renderer processes and the browser process.

The advantage of the multiprocess approach is that websites run in isolation from one another. If one website crashes, only its renderer process is affected; all other processes remain unharmed. Furthermore, renderer processes run in a **sandbox**, which means that access to disk and network I/O is restricted, minimizing the effects of any security exploits.

- Cooperating processes require some type of inter-process communication, which is most commonly one of two types: Shared Memory systems or Message Passing systems. Figure 3.13 illustrates the difference between the two systems:

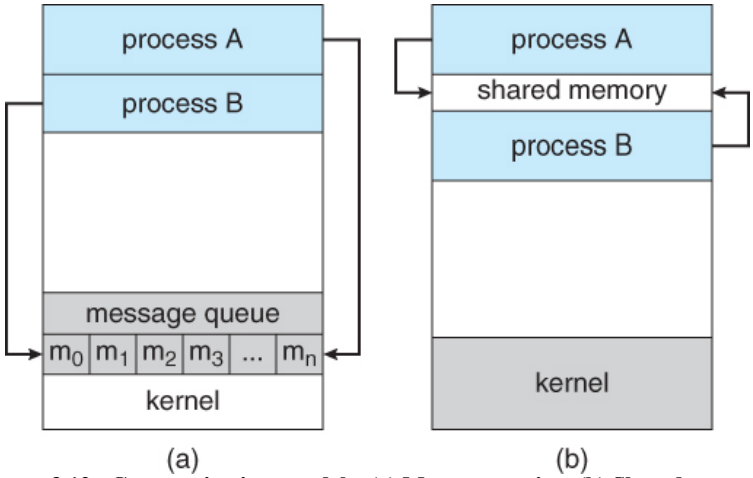


Figure 3.12 - Communications models: (a) Message passing. (b) Shared memory.

- Shared Memory is faster once it is set up, because no system calls are required and access occurs at normal memory speeds. However it is more complicated to set up, and doesn't work as well across multiple computers. Shared memory is generally preferable when large amounts of information must be shared quickly on the same computer.
- Message Passing requires system calls for every message transfer, and is therefore slower, but it is simpler to set up and works well across multiple computers. Message passing is generally preferable when the amount and/or frequency of data transfers is small, or when multiple computers are involved.

3.4.1 Shared-Memory Systems

- In general the memory to be shared in a shared-memory system is initially within the address space of a particular process, which needs to make system calls in order to make that memory publicly available to one or more other processes.
- Other processes which wish to use the shared memory must then make their own system calls to attach the shared memory area onto their address space.
- Generally a few messages must be passed back and forth between the cooperating processes first in order to set up and coordinate the shared memory access.

Producer-Consumer Example Using Shared Memory

- This is a classic example, in which one process is producing data and another process is consuming the data. (In this example in the order in which it is produced, although that could vary.)
- The data is passed via an intermediary buffer, which may be either unbounded or bounded. With a bounded buffer the producer may have to wait until there is space available in the buffer, but with an unbounded buffer the producer will never need to wait. The consumer may need to wait in either case until there is data available.
- This example uses shared memory and a circular queue. Note in the code below that only the producer changes "in", and only the consumer changes "out", and that they can never be accessing the same array location at the same time.
- First the following data is set up in the shared memory area:

```
#define BUFFER_SIZE 10

typedef struct {
     . . .
} item;

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

- Then the producer process. Note that the buffer is full when "in" is one less than "out" in a circular sense:

```
// Code from Figure 3.13

item nextProduced;

while( true ) {

    /* Produce an item and store it in nextProduced */
    nextProduced = makeNewItem( . . . );

    /* Wait for space to become available */
    while( ( ( in + 1 ) % BUFFER_SIZE ) == out )
          ; /* Do nothing */

    /* And then store the item and repeat the loop. */
    buffer[ in ] = nextProduced;
    in = ( in + 1 ) % BUFFER_SIZE;
```

```
}
```

- Then the consumer process. Note that the buffer is empty when "in" is equal to "out":

```
// Code from Figure 3.14

item nextConsumed;

while( true ) {

     /* Wait for an item to become available */
     while( in == out )
           ; /* Do nothing */

     /* Get the next available item */
     nextConsumed = buffer[ out ];
     out = ( out + 1 ) % BUFFER_SIZE;

     /* Consume the item in nextConsumed
          ( Do something with it ) */

}
```

3.4.2 Message-Passing Systems

- Message passing systems must support at a minimum system calls for "send message" and "receive message".
- A communication link must be established between the cooperating processes before messages can be sent.
- There are three key issues to be resolved in message passing systems as further explored in the next three subsections:
  - Direct or indirect communication (naming)
  - Synchronous or asynchronous communication
  - Automatic or explicit buffering.

3.4.2.1 Naming

- With **direct communication** the sender must know the name of the receiver to which it wishes to send a message.
  - There is a one-to-one link between every sender-receiver pair.
  - For **symmetric** communication, the receiver must also know the specific name of the sender from which it wishes to receive messages.
    For **asymmetric** communications, this is not necessary.
- **Indirect communication** uses shared mailboxes, or ports.
  - Multiple processes can share the same mailbox or boxes.
  - Only one process can read any given message in a mailbox. Initially the process that creates the mailbox is the owner, and is the only one allowed to read mail in the mailbox, although this privilege may be transferred.
    - (Of course the process that reads the message can immediately turn around and place an identical message back in the box for someone else to read, but that may put it at the back end of a queue of messages.)
  - The OS must provide system calls to create and delete mailboxes, and to send and receive messages to/from mailboxes.

3.4.2.2 Synchronization

- Either the sending or receiving of messages (or neither or both) may be either **blocking** or **non-blocking**.

3.4.2.3 Buffering

- Messages are passed via queues, which may have one of three capacity configurations:
  1. **Zero capacity** - Messages cannot be stored in the queue, so senders must block until receivers accept the messages.
  2. **Bounded capacity**- There is a certain pre-determined finite capacity in the queue. Senders must block if the queue is full, until space becomes available in the queue, but may be either blocking or non-blocking otherwise.
  3. **Unbounded capacity** - The queue has a theoretical infinite capacity, so senders are never forced to block.

```
message next_produced;

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

     send(next_produced);
}
```

Figure 3.15 The producer process using message passing.

```
message next_consumed;

while (true) {
     receive(next_consumed);

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

Figure 3.16 The consumer process using message passing.

3.5 Examples of IPC Systems

3.5.1 An Example: POSIX Shared Memory (Eighth Edition Version)

1. The first step in using shared memory is for one of the processes involved to allocate some shared memory, using shmget:

```
int segment_id = shmget( IPC_PRIVATE, size, S_IRUSR | S_IWUSR );
```

   - The first parameter specifies the key (identifier) of the segment. IPC_PRIVATE creates a new shared memory segment.
   - The second parameter indicates how big the shared memory segment is to be, in bytes.
   - The third parameter is a set of bitwise ORed flags. In this case the segment is being created for reading and writing.
   - The return value of shmget is an integer identifier

2. Any process which wishes to use the shared memory must *attach* the shared memory to their address space, using shmat:

```
char * shared_memory = ( char * ) shmat( segment_id, NULL, 0 );
```

   - The first parameter specifies the key (identifier) of the segment that the process wishes to attach to its address space
   - The second parameter indicates where the process wishes to have the segment attached. NULL indicates that the system should decide.
   - The third parameter is a flag for read-only operation. Zero indicates read-write; One indicates readonly.
   - The return value of shmat is a void *, which the process can use (type cast) as appropriate. In this example it is being used as a character pointer.

3. Then processes may access the memory using the pointer returned by shmat, for example using sprintf:

```
sprintf( shared_memory, "Writing to shared memory\n" );
```

4. When a process no longer needs a piece of shared memory, it can be detached using shmdt:

```
shmdt( shared_memory );
```

5. And finally the process that originally allocated the shared memory can remove it from the system suing shmctl.

```
shmctl( segment_id, IPC_RMID );
```

6. Figure 3.16 from the eighth edition illustrates a complete program implementing shared memory on a POSIX system:

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

int main()
{
/* the identifier for the shared memory segment */
int segment_id;=
/* a pointer to the shared memory segment */
char* shared_memory;
/* the size (in bytes) of the shared memory segment */
const int size = 4096;

    /* allocate a shared memory segment */
    segment_id = shmget(IPC_PRIVATE, size, S_IRUSR | S_IWUSR);

    /* attach the shared memory segment */
    shared_memory = (char *) shmat(segment_id, NULL, 0);

    /* write a message to the shared memory segment */
    sprintf(shared_memory, "Hi there!");

    /* now print out the string from shared memory */
    printf("*%s\n", shared_memory);

    /* now detach the shared memory segment */
    shmdt(shared_memory);

    /* now remove the shared memory segment */
    shmctl(segment_id, IPC_RMID, NULL);

    return 0;
}
```

Figure 3.16 C program illustrating POSIX shared-memory API.

3.5.1 An Example: POSIX Shared Memory (Ninth Edition Version)

1. The ninth edition shows an alternate approach to shared memory in POSIX systems. Under this approach, the first step in using shared memory is to create a shared-memory object using shm_open( ),in a fashion similar to other file opening commands. The name provided will be the name of the memory-mapped file.

   ```
   shm_fd = shm_open( name,O_CREAT | O_RDRW,0666 );
   ```

2. The next step is to set the size of the file using ftruncate:

   ```
   ftruncate( shm_fd, 4096 );
   ```

3. Finally the mmap system call maps the file to a memory address in the user program space.and makes it shared. In this example the process that created the shared memory will be writing to it:

   ```
   ptr = mmap( 0, SIZE,PROT_WRITE, MAP_SHARED, shm_fd, 0 );
   ```

4. The "borrower" of the shared memory, (not the one who created it), calls `shm_open( )` and `mmap( )` with different arguments, skips the `ftruncate( )` step and unlinks (removes) the file name when it is done with it. Note that the "borrower" must use the same file name as the "lender" who created it. (This information could have been passed using messages.)

   ```
   shm_unlink( name );
   ```

5. Note that writing to and reading from the shared memory is done with pointers and memory addresses (sprintf) in both the 9th and 8th edition versions, even though the 9th edition is illustrating memory mapping of a file.

6. Figures 3.17 and 3.18 from the ninth edition illustrate a complete program implementing shared memory on a POSIX system:

```
#include <stdio.h>
#include <stlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.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 shm_fd;
/* pointer to shared memory obect */
void *ptr;

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

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

    /* memory map the shared memory object */
    ptr = mmap(0, SIZE, PROT_WRITE, MAP_SHARED, shm_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 3.17 Producer process illustrating POSIX shared-memory API.

```
#include <stdio.h>
#include <stlib.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.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 shm_fd;
/* pointer to shared memory obect */
void *ptr;

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

    /* memory map the shared memory object */
    ptr = mmap(0, SIZE, PROT_READ, MAP_SHARED, shm_fd, 0);

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

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

    return 0;
}
```

Figure 3.18 Consumer process illustrating POSIX shared-memory API.

3.5.2 An Example: Mach

- Recall that the Mach kernel is a micro kernel, which performs few services besides delivering messages between other tasks (both system tasks and user tasks.)
- Most communication in Mach, including all system calls and inter-process communication is done via messages sent to mailboxes, also known as ports.
- Whenever a task (process) is created, it automatically gets two special mailboxes: a Kernel mailbox, and a Notify mailbox.
  - The kernel communicates with the task using the Kernel mailbox.
  - The kernel sends notification of events to the Notify mailbox.
- Three system calls are used for message transfer:
  - msg_send( ) sends a message to a mailbox
  - msg_receive( ) receives a message.
  - msg_rpc( ) sends a message and waits for exactly one message in response from the sender.
- Port_allocate( ) creates a new mailbox and the associated queue for holding messages (size 8 by default.)
- Only one task at a time can own or receive messages from any given mailbox, but these are transferable.
- Messages from the same sender to the same receiver are guaranteed to arrive in FIFO order, but no guarantees are made regarding messages from multiple senders.
- Messages consist of a fixed-length header followed by variable length data.
  - The header contains the mailbox number (address) of the receiver and the sender.
  - The data section consists of a list of typed data items, each containing a type, size, and value.
- If the receiver's mailbox is full, the sender has four choices:
  1. Wait indefinitely until there is room in the mailbox.
  2. Wait at most N milliseconds.
  3. Do not wait at all.
  4. Temporarily cache the message with the kernel, for delivery when the mailbox becomes available.
     - Only one such message can be pending at any given time from any given sender to any given receiver.

    - Normally only used by certain system tasks, such as the print spooler, which must notify the "client" of the completion of their job, but cannot wait around for the mailbox to become available.
- Receive calls must specify the mailbox or mailbox set from which they wish to receive messages.
- Port_status( ) reports the number of messages waiting in any given mailbox.
- If there are no messages available in a mailbox (set), the receiver can either block for N milliseconds, or not block at all.
- In order to avoid delays caused by copying messages (multiple times), Mach re-maps the memory space for the message from the sender's address space to the receiver's address space (using virtual memory techniques to be covered later), and does not actually move the message anywhere at all. (When the sending and receiving task are both on the same computer.)

3.5.3 An Example: Windows XP

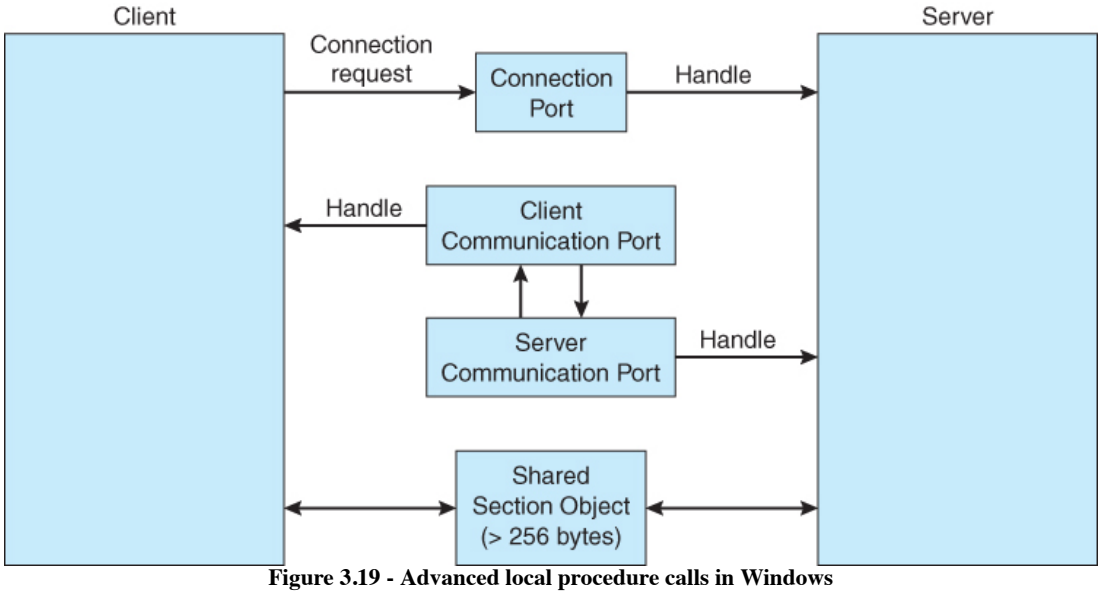


Figure 3.19 - Advanced local procedure calls in Windows

3.6 Communication in Client-Server Systems

3.6.1 Sockets

- A **socket** is an endpoint for communication.
- Two processes communicating over a network often use a pair of connected sockets as a communication channel. Software that is designed for client-server operation may also use sockets for communication between two processes running on the same computer - For example the UI for a database program may communicate with the back-end database manager using sockets. (If the program were developed this way from the beginning, it makes it very easy to port it from a single-computer system to a networked application.)
- A socket is identified by an IP address concatenated with a port number, e.g. 200.100.50.5:80.

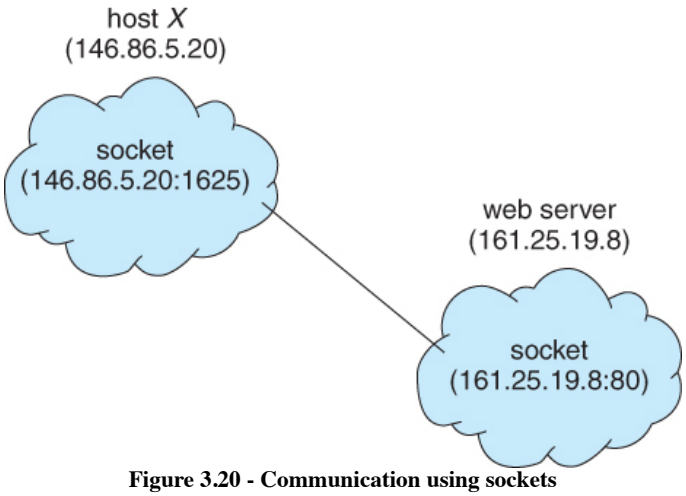


Figure 3.20 - Communication using sockets

- Port numbers below 1024 are considered to be *well-known*, and are generally reserved for common Internet services. For example, telnet servers listen to port 23, ftp servers to port 21, and web servers to port 80.
- General purpose user sockets are assigned unused ports over 1024 by the operating system in response to system calls such as socket( ) or soctkepair( ).
- Communication channels via sockets may be of one of two major forms:
    - **Connection-oriented (TCP, Transmission Control Protocol)** connections emulate a telephone connection. All packets sent down the connection are guaranteed to arrive in good condition at the other end, and to be delivered to the receiving process in the order in which they were sent. The TCP layer of the network protocol takes steps to verify all packets sent,

re-send packets if necessary, and arrange the received packets in the proper order before delivering them to the receiving process. There is a certain amount of overhead involved in this procedure, and if one packet is missing or delayed, then any packets which follow will have to wait until the errant packet is delivered before they can continue their journey.
    - **Connectionless (UDP, User Datagram Protocol)** emulate individual telegrams. There is no guarantee that any particular packet will get through undamaged (or at all), and no guarantee that the packets will get delivered in any particular order. There may even be duplicate packets delivered, depending on how the intermediary connections are configured. UDP transmissions are much faster than TCP, but applications must implement their own error checking and recovery procedures.
- Sockets are considered a low-level communications channel, and processes may often choose to use something at a higher level, such as those covered in the next two sections.
- Figure 3.19 and 3.20 illustrate a client-server system for determining the current date using sockets in Java.

```
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 3.19 Date server.

```
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 3.20 Date client.

Figure 3.21 and Figure 3.22

3.6.2 Remote Procedure Calls, RPC

- The general concept of RPC is to make procedure calls similarly to calling on ordinary local procedures, except the procedure being called lies on a remote machine.
- Implementation involves ***stubs*** on either end of the connection.
  - The local process calls on the stub, much as it would call upon a local procedure.
  - The RPC system packages up (marshals) the parameters to the procedure call, and transmits them to the remote system.
  - On the remote side, the RPC daemon accepts the parameters and calls upon the appropriate remote procedure to perform the requested work.
  - Any results to be returned are then packaged up and sent back by the RPC system to the local system, which then unpackages them and returns the results to the local calling procedure.
- One potential difficulty is the formatting of data on local versus remote systems. (e.g. big-endian versus little-endian.) The resolution of this problem generally involves an agreed-upon intermediary format, such as XDR (external data representation.)
- Another issue is identifying which procedure on the remote system a particular RPC is destined for.
  - Remote procedures are identified by *ports*, though not the same ports as the socket ports described earlier.
  - One solution is for the calling procedure to know the port number they wish to communicate with on the remote system. This is problematic, as the port number would be compiled into the code, and it makes it break down if the remote system changes their port numbers.
  - More commonly a *matchmaker* process is employed, which acts like a telephone directory service. The local process must first contact the matchmaker on the remote system (at a well-known port number), which looks up the desired port number and returns it. The local process can then use that information to contact the desired remote procedure. This operation involves an extra step, but is much more flexible. An example of the matchmaker process is illustrated in Figure 3.21 below.
- One common example of a system based on RPC calls is a networked file system. Messages are passed to read, write, delete, rename, or check status, as might be made for ordinary local disk access requests.

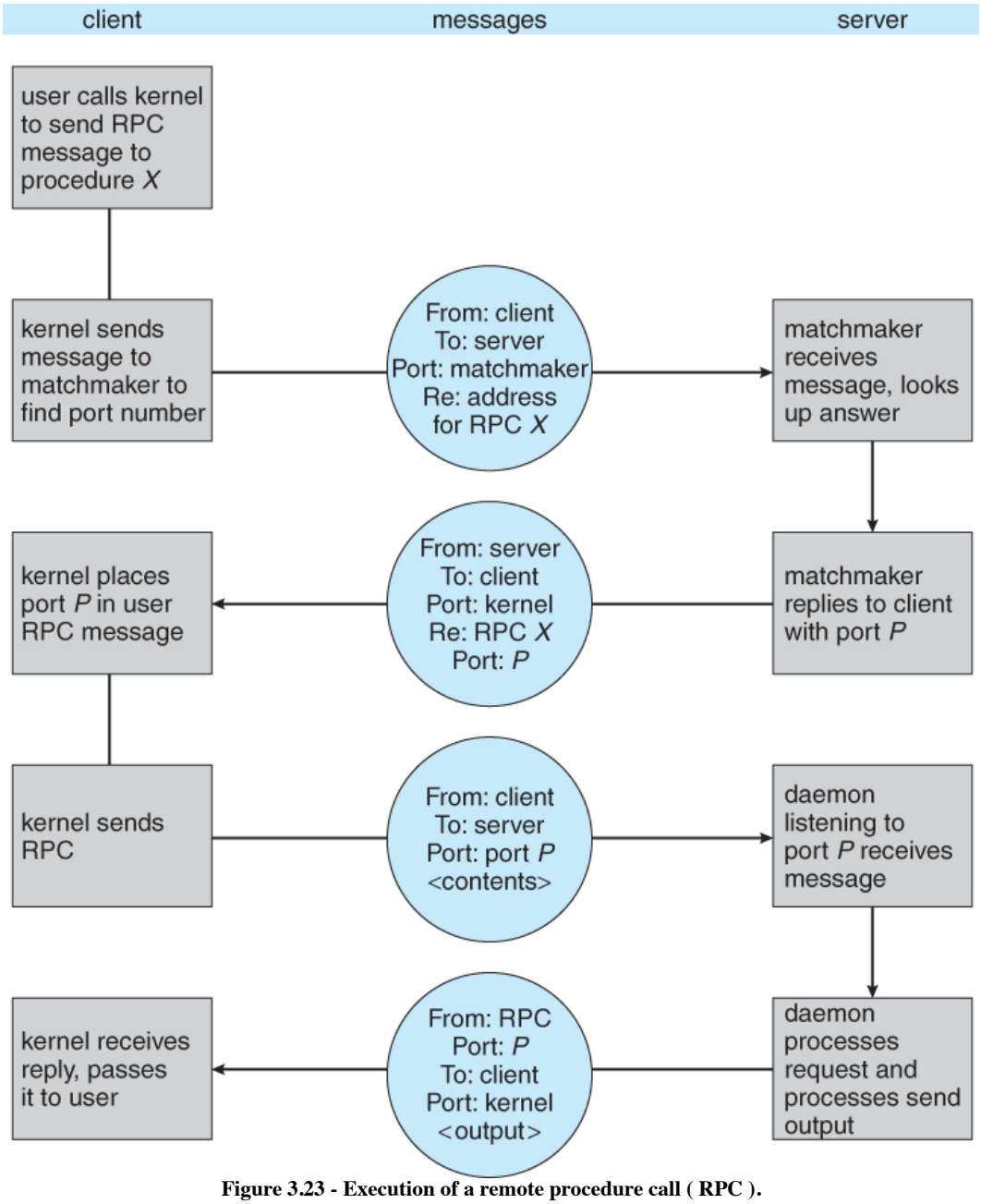


Figure 3.23 - Execution of a remote procedure call (RPC).

3.6.3 Pipes

- ***Pipes*** are one of the earliest and simplest channels of communications between (UNIX) processes.
- There are four key considerations in implementing pipes:
  1. Unidirectional or Bidirectional communication?
  2. Is bidirectional communication half-duplex or full-duplex?
  3. Must a relationship such as parent-child exist between the processes?
  4. Can pipes communicate over a network, or only on the same machine?
- The following sections examine these issues on UNIX and Windows

3.6.3.1 Ordinary Pipes

- Ordinary pipes are uni-directional, with a reading end and a writing end. (If bidirectional communications are needed, then a second pipe is required.)
- In UNIX ordinary pipes are created with the system call "`int pipe( int fd [ ] )`".
  - The return value is 0 on success, -1 if an error occurs.
  - The int array must be allocated before the call, and the values are filled in by the pipe system call:
    - fd[0] is filled in with a file descriptor for the reading end of the pipe
    - fd[1] is filled in with a file descriptor for the writing end of the pipe
  - UNIX pipes are accessible as files, using standard read( ) and write( ) system calls.
  - Ordinary pipes are only accessible within the process that created them.
    - Typically a parent creates the pipe before forking off a child.
    - When the child inherits open files from its parent, including the pipe file(s), a channel of communication is established.

- Each process (parent and child) should first close the ends of the pipe that they are not using. For example, if the parent is writing to the pipe and the child is reading, then the parent should close the reading end of its pipe after the fork and the child should close the writing end.

- Figure 3.22 shows an ordinary pipe in UNIX, and Figure 3.23 shows code in which it is used.

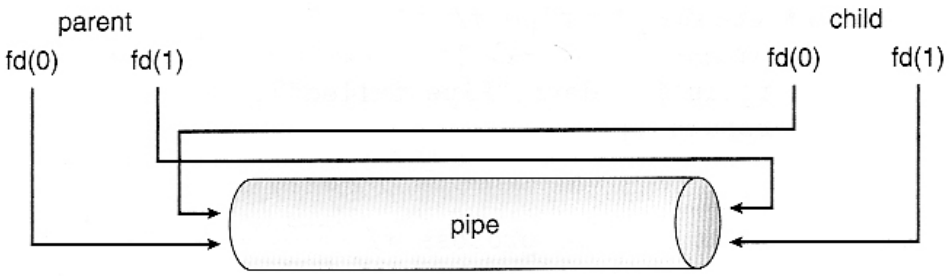


Figure 3.22 File descriptors for an ordinary pipe.

Figure 3.24

```
#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 write end of the pipe */
   close(fd[READ_END]);
}

return 0;
}
```

Figure 3.23 Ordinary pipes in UNIX.

Figure 3.24 Continuation of Figure 3.23 program.

Figure 3.25 and Figure 3.26

- Ordinary pipes in Windows are very similar
  - Windows terms them ***anonymous*** pipes
  - They are still limited to parent-child relationships.
  - They are read from and written to as files.
  - They are created with `CreatePipe( )` function, which takes additional arguments.
  - In Windows it is necessary to specify what resources a child inherits, such as pipes.

```
#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;
```

```
return 0;
}
```

Figure 3.25 Windows anonymous pipes — parent process.

Figure 3.26 Continuation of Figure 3.25 program.

Figure 3.27 and Figure 3.28

```
#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 3.27 Windows anonymous pipes — child process.

Figure 3.29

3.6.3.2 Named Pipes

- Named pipes support bidirectional communication, communication between non parent-child related processes, and persistence after the process which created them exits. Multiple processes can also share a named pipe, typically one reader and multiple writers.
- In UNIX, named pipes are termed fifos, and appear as ordinary files in the file system.
  - (Recognizable by a "p" as the first character of a long listing, e.g. /dev/initctl)
  - Created with `mkfifo( )` and manipulated with `read( ), write( ), open( ), close( )`, etc.
  - UNIX named pipes are bidirectional, but half-duplex, so two pipes are still typically used for bidirectional communications.
  - UNIX named pipes still require that all processes be running on the same machine. Otherwise sockets are used.
- Windows named pipes provide richer communications.
- Full-duplex is supported.
- Processes may reside on the same or different machines
- Created and manipulated using `CreateNamedPipe( ), ConnectNamedPipe( ), ReadFile( ),and WriteFile( ).`

Race Conditions (Not from the book)

Any time there are two or more processes or threads operating concurrently, there is potential for a particularly difficult class of problems known as ***race conditions.*** The identifying characteristic of race conditions is that the performance varies depending on which process or thread executes their instructions before the other one, and this becomes a problem when the program runs correctly in some instances and incorrectly in others. Race conditions are notoriously difficult to debug, because they are unpredictable, unrepeatable, and may not exhibit themselves for years.

Here is an example involving a server and a client communicating via sockets:

1. First the server writes a greeting message to the client via the socket:

```
const int BUFFLENGTH = 100;
char buffer[ BUFFLENGTH ];
sprintf( buffer, "Hello Client %d!", i );
write( clientSockets[ i ], buffer, strlen( buffer ) + 1 );
```

2. The client then reads the greeting into its own buffer. The client does not know for sure how long the message is, so it allocates a buffer bigger than it needs to be. The following will read all available characters in the socket, up to a maximum of BUFFLENGTH characters:

```
const int BUFFLENGTH = 100;
char buffer[ BUFFLENGTH ];
read( mysocket, buffer, BUFFLENGTH );
cout << "Client received: " << buffer << "\n";
```

3. Now the server prepares a packet of work and writes that to the socket:

```
write( clientSockets[ i ], & wPacket, sizeof( wPacket ) );
```

4. And finally the client reads in the work packet and processes it:

```
read( mysocket, & wPacket, sizeof( wPacket ) );
```

The Problem: The problem arises if the server executes step 3 before the client has had a chance to execute step 2, which can easily happen depending on process scheduling. In this case, when the client finally gets around to executing step 2, it will read in not only the original greeting, but also the first part of the work packet. And just to make things harder to figure out, the cout << statement in step 2 will only print out the greeting message, since there is a null byte at the end of the greeting. This actually isn't even a problem at this point, but then later when the client executes step 4, it does not accurately read in the work packet because part of it has already been read into the buffer in step 2.

Solution I: The easiest solution is to have the server write the entire buffer in step 1, rather than just the part filled with the greeting, as:

```
write( clientSockets[ i ], buffer, BUFFLENGTH );
```

Unfortunately this solution has two problems: (1) It wastes bandwidth and time by writing more than is needed, and more importantly, (2) It leaves the code open to future problems if the BUFFLENGTH is not the same in the client and in the server.

Solution II: A better approach for handling variable-length strings is to first write the length of the string, followed by the string itself as a separate write. Under this solution the server code changes to:

```
sprintf( buffer, "Hello Client %d!", i );
int length = strlen( buffer ) + 1;
write( clientSockets[ i ], &length, sizeof( int ) );
write( clientSockets[ i ], buffer, length );
```

and the client code changes to:

```
int length;
if( read( mysocket, &length, sizeof( int ) ) != sizeof( int ) ) {
     perror( "client read error: " );
     exit( -1 );
}

if( length < 1 || length > BUFFLENGTH ) {
     cerr << "Client read invalid length = " << length << endl;
     exit( -1 );
}

if( read( mysocket, buffer, length ) != length ) {
perror( "client read error: " );
exit( -1 );
}

cout << "Client received: " << buffer << "\n";
```

Note that the above solution also checks the return value from the read system call, to verify that the number of characters read is equal to the number expected. (Some of those checks were actually in the original code, but were omitted from the notes for clarity. The real code also uses select( ) before reading, to verify that there are characters present to read and to delay if not.)

Note also that this problem could not be (easily) solved using the synchronization tools covered in chapter 6, because the problem is not really one of two processes accessing the same data at the same time.

3.7 Summary

OLD 3.6.3 Remote Method Invocation, RMI (Optional, Removed from 8th edition)

- RMI is the Java implementation of RPC for contacting processes operating on a different Java Virtual Machine, JVM, which may or may not be running on a different physical machine.
- There are two key differences between RPC and RMI, both based on the object-oriented nature of Java:
  - RPC accesses remote procedures or functions, in a procedural-programming paradigm. RMI accesses methods within remote Objects.
  - The data passed by RPC as function parameters are ordinary data only, i.e. ints, floats, doubles, etc. RMI also supports the passing of Objects.
- RMI is implemented using stubs (on the client side) and skeletons (on the servers side), whose responsibility is to package (marshall) and unpack the parameters and return values being passed back and forth, as illustrated in Figures 3.22 and 3.23:
- See the Java online documentation for more details on RMI, including issues with value vs. reference parameter passing, and the requirements that certain passed data items be *Serializable*.

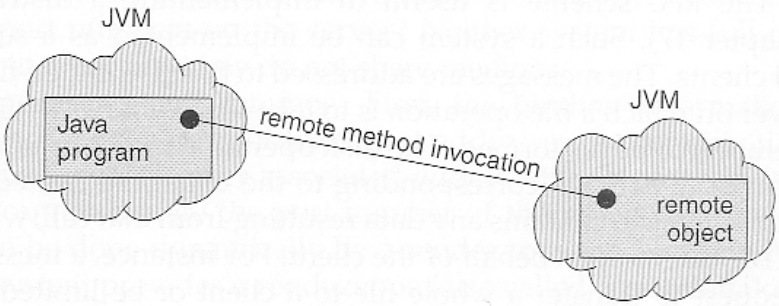


Figure 3.22 Remote method invocation.

Figure omitted in 8th edition

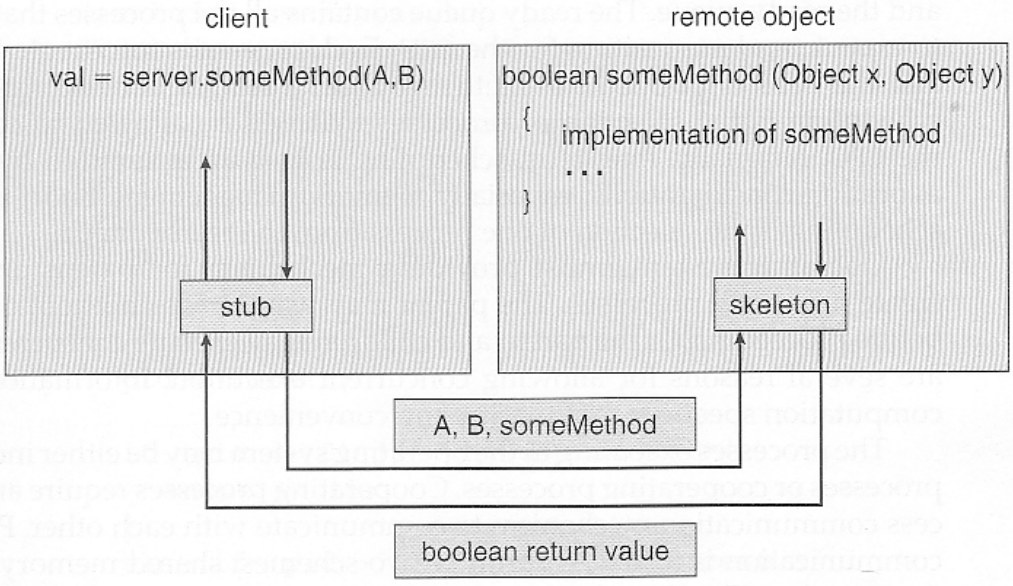


Figure 3.23 Marshalling parameters.

Figure omitted in 8th edition