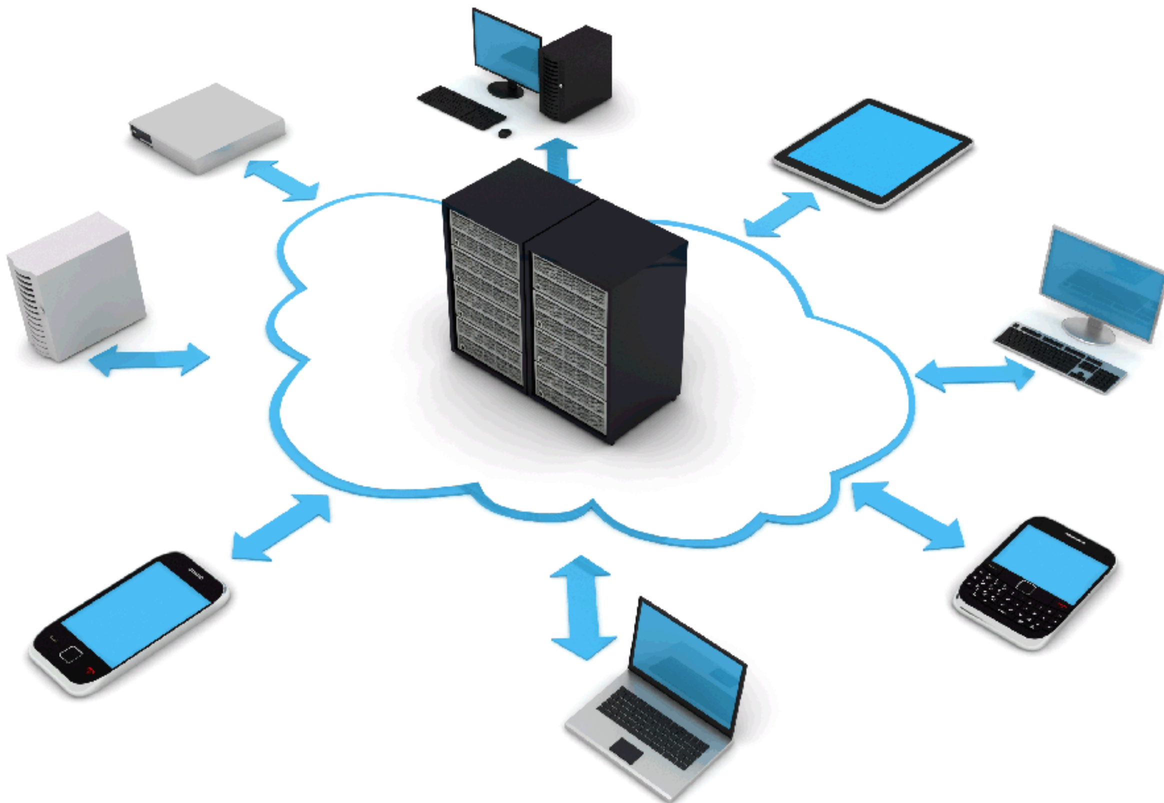
Chapter 5

Concurrent Computing

What is in This Chapter ?

This chapter will introduce you to the basics of concurrent computing. We first discuss some types of **concurrent systems** and a few issues/concerns that we must be aware of when having more than one task being performed at the same time. We then discuss **process management** at the unix shell level and then at the programming level, with functions like **fork**(), **exec**(), **wait**() and **system**() calls. The next section discusses **inter-process communication (IPC)** and the use of **signals** to inform other processes when tasks are complete. The use of **TCP sockets** and **Datagram sockets** are then discussed as they pertain to client/server models. Finally, **threads** are discussed, along with the need to use **semaphores** & **mutexes** to facilitate proper resource-sharing.



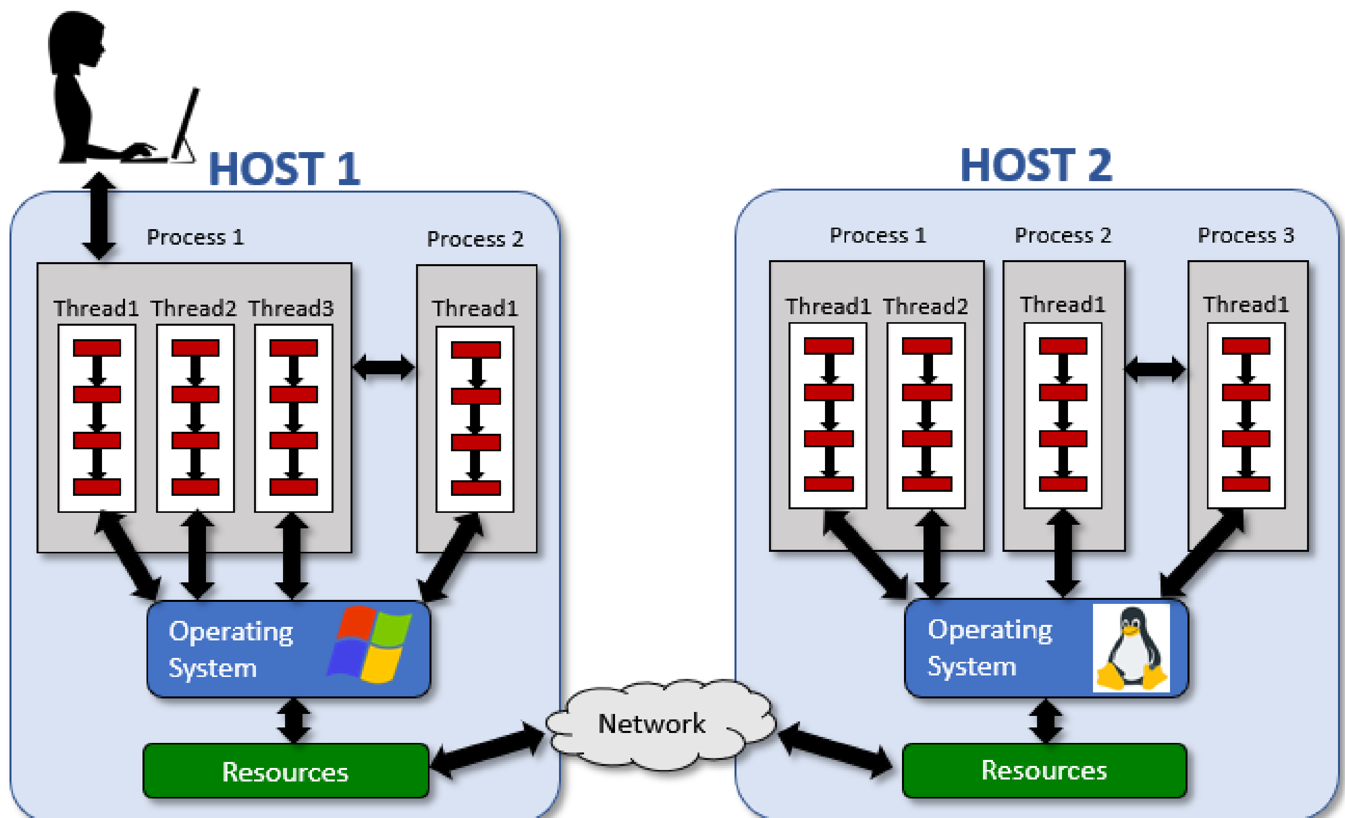
5.1 Concurrent Systems

When we start out to learn how to program, we find it easiest to focus on one task at a time. That is, we imagine our program as being run by a single computer that simply follows the instructions that we give it, based on our source code. It is challenging enough to learn how to program well with a single program.

However, the real world is not so simple. In reality, many things are happening all around us at the same time. In English, the word ***concurrent*** means "occurring or operating at the same time". In computer science, the term concurrency implies that multiple programs (or processes) are working together at the same time … hopefully to accomplish some task faster. Here is a definition extracted from wikipedia:

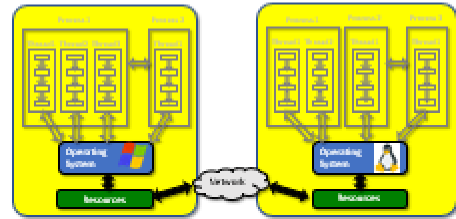
> ***Concurrent computing*** *is a form of computing in which several computations are executed during overlapping time periods (i.e., concurrently) instead of sequentially (i.e., one completing before the next starts).*

A large system makes use of concurrent computing when it is **(a)** multithreaded, **(b)** has multiple processes or **(c)** is distributed. Here is a diagram showing all three. A host computer may run multiple processes (i.e., programs) each working together to perform some task in the system. A single process may have multiple threads running at the same time … all working together. Finally, processes running on different machines on a network may be interacting together, forming a distributed system. Usually, the user interacts with just one process.



We will examine each of these three forms of concurrency.

Distributed Systems



A distributed system is typically a large program that executes over multiple physical host machines. Usually, these machines are in different locations, cities or even countries. The interaction is over a network. This network may be:

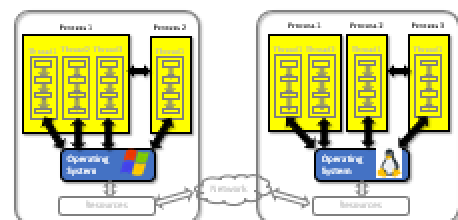
- **Intranet** – a network internal to an organization
- **Internet** – a public network, external to all organizations

One interesting aspect about distributed computing is that each host machine has different resources. That is, they may have different CPUs, different processing capabilities, different file systems, etc..

It sounds a bit complicated (and slower?) to have different types of computers interacting over a network. Why would anyone want to do distributed computing ? Here are some reasons:

- **Speed**: A single host may have insufficient processing power to complete a task in a reasonable time. Having other hosts join in on the work … it will hasten task completion, as long as the amount of network communication is kept low.
- **Necessity**: Often clients need to connect to servers which are in different physical locations. Completing the task-at-hand may require connection to various servers to obtain database information, to record transactions, etc..
- **Convenience**: Users may need to connect to a host that is not in the same location.

Multi-Process Systems



A multi-process system is a system where multiple processes (i.e., executables) are running at the same time and communicating with one another to accomplish a task. The executables need not be unique. There may even be multiple copies of the same program running.

Each executable has its own independent control flow and virtual memory. That is, it operates on its own, although it may rely on data and instructions from other processes in order to complete its individual task. The operating system contains mechanisms that allow Inter-Process Communication (IPC) to allow processes to communicate, usually to have access to shared data.

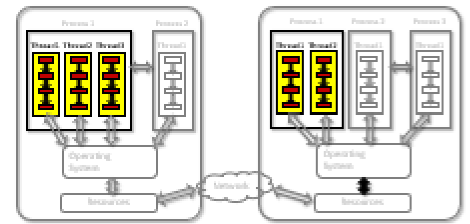
As with distributed systems, it may seem like we are complicating things by having multiple processes communicate through the operating system. Why would anyone want to implement a multi-process system?

Here are a couple of reasons:

- **Simplifying**: Often there are many different tasks to perform which may be independent from one another. It can be easier to schedule a different process for each task.

- **Resource Management**: Certain tasks can be "assigned to" a particular resource (e.g., client to communicate with user, server to handle requests, process to regulate access to database), reducing the need for multiple processes to access the same resources. As a result, the system can reduce bottlenecks and operate more efficiently.

Multi-Threaded Systems



A multi-threaded system is a single process with multiple control flows. That is … multiple tasks are performed by the same CPU but they take turns by sharing the CPU's processing time. The threads share the same virtual memory, address space and resources as they operate in the same process. There is a need at times to synchronize different threads in order to avoid race conditions and deadlocks.

The idea of a multi-threaded system is similar to a multi-process system in that they are often used when different tasks are to be performed. In the multi-threaded system, however, the tasks are usually dependent on each other. The main advantage of using multiple threads is:

- **Simplicity**: It is conceptually simpler to think of two tasks being done separately at the same time, even though they are sharing the CPU … taking turns to get their task done.

Some situations where multi-threading is often beneficial is:

- **Handling user input**. One thread blocks and waits for incoming requests, while another thread processes requests that have already come in.

- **Quick refresh**. Sometimes it is nice to have a thread responsible for refreshing the user interface (e.g., graphics/animation) while the program continues processing.

There are a few (potentially serious) issues that may arise when doing concurrency. As a result, it can be more difficult to write software for concurrent systems. It can also be difficult to debug concurrent systems.

Here are some of these issues:

1. **Shared Resources**. Multiple process (or threads) will at times need the same resource. There needs to be some coordination rules so that this sharing takes place decently and respectfully. Typically, shared resources are files and variables.

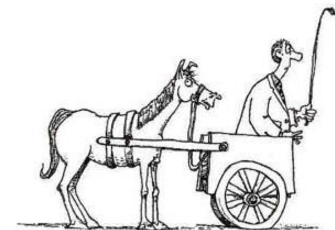
    - When accessing a file, it can be "locked" for use by one process/thread so that others cannot access it while it is in use. Of course, a process/thread that "hogs" a file resource can be slowing down the system if not careful.

    - When accessing a shared variable, a **semaphore** or **mutex** can be used:

        A mutex (mutual exclusion object) is a program object that is created so that multiple program threads can take turns sharing the same resource, such as access to a file. Only the thread that locked or acquired the mutex can unlock it.

        A semaphore is a variable used to control access to a common resource by multiple processes. It is a generalization of a mutex. A thread waiting on a semaphore can be signaled by a different thread so that it can have access.

2. **Deadlocks**. This is a condition that can occur which is similar to the notion of a traffic jam. It is a condition in which multiple threads/processes are blocked … all waiting for a condition that will never occur. It is always due to improper handling of semaphores or mutexes. Careful system design will reduce the likelihood of deadlocks occurring, although sometimes deadlocks occur due to unforeseen situations inherent to the problem at hand.



3. **Race Conditions**. This is a timing problem in which the correctness of a program depends on one thread reaching a point in control flow before another thread. That is, some things have gotten out of order. You can imagine the scenario, for example, of trying to process data before it has been completely entered. Sometimes we have to handle such potential problems because the order that things are processed in is never guaranteed.

5.2 Process Management

Recall that a process is a running executable (e.g., a running program). Processes are managed by the operating system.

Process Management *involves allocating resources to processes, enabling processes to share and exchange information, protecting the resources of each process from other processes and enabling synchronization among processes.*

The operating system is primarily involved with managing the processes, but as software system developers, we need to understand a little about how it is done so that we can make use of multiple processes when we write our programs. In particular, we need to know how to start (i.e., spawn) a process, how to stop and pause it, and how to modify the behaviors of a process using signals.

There are two ways that we can manage processes:

- Using **shell commands** – manually as a user of a system
- Using **system calls** – automatically through other programs/processes

How are processes managed ? The operating system maintains certain information about each process that has been created. Each process has the following:

- Process Identifier (**PID**) – unique to each process
- Parent Process Identifier (**PPID**) – the process that spawned it
- Address Space and Virtual Memory – code segment, data segment, stack, heap
- Control Flow(s) – its own order that commands are evaluated in

Let's look first at how to manage a process. The simplest way is from a shell. We can start a process in the foreground or in the background. You have already done this many times. Each time you run your code, for example, you are starting a process. Most of the time, we run it in the foreground. However, you can use the **&** sign to run a process in the background. Recall that the following runs **gedit** in the foreground (i.e., we cannot use the shell until **gedit** completes):

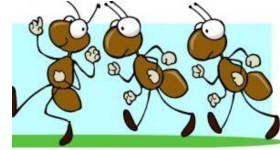
```
student@COMPBase:~$ gedit helloWorld.c
student@COMPBase:~$
```

And the **&** allows us to run **gedit** in the background (i.e., we can continue to use the shell):

```
student@COMPBase:~$ gedit helloWorld.c&
student@COMPBase:~$
```

Consider the following program that runs "forever":

Code from **shellProcess.c**



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

int main() {
  int i = 1;

  while (1) {
    printf("The ants go marching %d by %d, hurrah, hurrah.\n", i, i);
    ++i;
    sleep(1);
  }
}
```

The code displays a message and counter repeatedly, with a 1 second pause (caused by the sleep(1) command which is defined in the **unistd.h** header) in between the messages.

We can run this program in the background in our shell window by using the **&** symbol:

```
student@COMPBase:~$ gcc -o shellProcess shellProcess.c
student@COMPBase:~$ ./shellProcess &
[2] 2513
student@COMPBase:~$ The ants go marching 1 by 1, hurrah, hurrah.
The ants go marching 1 by 1, hurrah, hurrah.
The ants go marching 2 by 2, hurrah, hurrah.
The ants go marching 3 by 3, hurrah, hurrah.
The ants go marching 4 by 4, hurrah, hurrah.
The ants go marching 5 by 5, hurrah, hurrah.
```

One thing to notice is that when we run the program, we immediately get the **PID** which is **2513** this time it runs. This number will allow us to stop the process at a later time.

You will also notice that the process continually displays information to the system shell window that we are using. Because of this, it is a little hard to be able to continue to use the shell window for other commands as it keeps printing stuff out and scrolling.

At any time, we can use the **ps** command to get a list of running processes. Assume that we did a **ps** while the **shellProcess** command was still running. Here is what we might see:

```
student@COMPBase:~$ ps
  PID TTY          TIME CMD
 2366 pts/17   00:00:00 bash
 2495 pts/17   00:00:00 gedit
 2513 pts/17   00:00:00 shellProcess
 2527 pts/17   00:00:00 ps
student@COMPBase:~$
```

This list above shows the current running processes from this terminal window. Notice that the bash shell is running, which allows us to enter commands. Also, the **gedit** editor is opened and running (it happens to have the **shellProcess.c** file opened). Notice as well that the **shellProcess** program is running. Also, the **ps** command that we ran to get this list … it itself is a running process. If we want more detail on the running process, we can use **ps -l** as follows:

```
student@COMPBase:~$ ps -l
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1002  2366  2359  0  80   0 -  2034 wait   pts/17   00:00:00 bash
0 S  1002  2495  2366  0  80   0 - 30417 poll_s pts/17   00:00:00 gedit
0 S  1002  2513  2366  0  80   0 -   549 hrtime pts/17   00:00:00 shellProcess
0 R  1002  2543  2366  0  80   0 -  2174 -      pts/17   00:00:00 ps
student@COMPBase:~$
```

You can see some extra information here such as the size (**SZ**) of the process running (in bytes) as well as the **PPID** that spawned the process and the user ID (**UID**).

The command **ps aux** command gives different information and lists more processes. Here is what you may see (although I removed much of the output to reduce space):

```
student@COMPBase:~$ ps aux
USER         PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root           1  0.1  0.2  24064  4836 ?        Ss   14:31   0:01 /sbin/init spla
root           2  0.0  0.0      0     0 ?        S    14:31   0:00 [kthreadd]
root           3  0.0  0.0      0     0 ?        S    14:31   0:00 [ksoftirqd/0]
...
syslog       453  0.0  0.1  30724  3004 ?        Ssl  14:32   0:00 /usr/sbin/rsysl
root         462  0.0  0.1   4136  3056 ?        Ss   14:32   0:00 /lib/systemd/sy
avahi        474  0.0  0.1   5916  3116 ?        Ss   14:32   0:00 avahi-daemon: r
message+     482  0.0  0.2   6856  4488 ?        Ss   14:32   0:00 /usr/bin/dbus-d
avahi        486  0.0  0.0   5916   288 ?        S    14:32   0:00 avahi-daemon: c
lp           498  0.0  0.2  11228  5196 ?        S    14:32   0:00 /usr/lib/cups/n
...
student     1315  0.0  0.1   6368  4076 ?        Ss   14:33   0:00 /lib/systemd/sy
student     2411  0.0  0.2   8124  4476 pts/18   Ss+  14:34   0:00 bash
student     2495  0.0  2.0 121668 41700 pts/17   Sl   14:36   0:00 gedit shellProc
student     2500  0.0  0.2  12736  5120 ?        S    14:36   0:00 /usr/lib/i386-l
student     2513  0.0  0.0   2196   560 pts/17   S    14:37   0:00 ./shellProcess
root        2549  0.0  0.0      0     0 ?        S    14:47   0:00 [kworker/0:0]
student     2562  0.0  0.1   8972  3224 pts/17   R+   14:52   0:00 ps aux
student@COMPBase:~$
```

There are many parameters to the **ps** command, but they will not be discussed here. To STOP a process, you can use the **kill** command. You just need to know the **PID**:

```
student@COMPBase:~$ kill 2513
student@COMPBase:~$ ps
  PID TTY          TIME CMD
 2366 pts/17   00:00:00 bash
 2495 pts/17   00:00:00 gedit
 2586 pts/17   00:00:00 ps
[2]+  Terminated              ./shellProcess
student@COMPBase:~$
```

After you kill a process, you will get a notification in the terminal window when you enter the next shell command. Above, you can see that once you use **ps** again, the process has been eliminated from the list of running processes. The **kill -stop** command is also used to temporarily *stop*/*pause*/*suspend* a process.

TERMINATED

```
student@COMPBase:~$ kill -stop 2513
student@COMPBase:~$ ps -l
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1002  2366  2359  0  80   0 -  2034 wait   pts/17   00:00:00 bash
0 S  1002  2495  2366  0  80   0 - 30417 poll_s pts/17   00:00:00 gedit
0 T  1002  2513  2366  0  80   0 -   549 signal pts/17   00:00:00 shellProcess
0 R  1002  2543  2366  0  80   0 -  2174 -      pts/17   00:00:00 ps
student@COMPBase:~$
```

To *continue* the process again, we use **kill -cont** with the PID:

```
student@COMPBase:~$ kill -cont 2513
student@COMPBase:~$ ps -l
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1002  2366  2359  0  80   0 -  2034 wait   pts/17   00:00:00 bash
0 S  1002  2495  2366  0  80   0 - 30417 poll_s pts/17   00:00:00 gedit
0 S  1002  2513  2366  0  80   0 -   549 hrtime pts/17   00:00:00 shellProcess
0 R  1002  2543  2366  0  80   0 -  2174 -      pts/17   00:00:00 ps
student@COMPBase:~$
```

You can also use other shell commands to manage processes. For example, the **jobs** command displays a list of all running jobs. While a *process* is any running program with its own address space, a *job* is any program you started that is not a daemon (i.e., not a background service-handling process)**.**

```
student@COMPBase:~$ jobs
[1]   Running                 gedit shellProcess.c &
[2]+  Stopped                 ./shellProcess
[3]-  Done                    ./wait
student@COMPBase:~$
```

Notice that the **jobs** command allows you to see what is *running*, what is currently *stopped* (or paused) and also what processes have just completed (i.e. *done*).

At any time, you can suspend the current running foreground process by pressing **CTRL-Z**. You may also kill the current running process by pressing **CTRL-C**.

You can use the **`fg`** command to resume the last suspended job, or you can use **`fg i`** to resume the job with id **`i`**. So, for example, in the above example, we could resume the **shellProcess** program by typing **`fg 2`** into the shell. It will run in the foreground. We could resume it to run it in the background if we use **`bg 2`** instead.

At this point, you should understand how to manage processes manually in the command shell window in Linux.

But in addition to managing processes from the command line, we can also do so within our C programs. There are 4 system calls that are related to process management:

- **fork** – spawns a clone of the current process
- **exec** – replaces executing code of current process with another program
- **wait** – pauses execution of a parent until a child process terminates
- **system** – runs a specified command as a shell command

We will now examine each of these one at a time…

FORK

Consider first the **fork()** function in C. It creates a new process with the current process being the parent of the new process.

Consider this example:

Code from **fork.c**

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

int main() {
  int childPID;

  printf("Forking...\n");

  childPID = fork();

  if (childPID == 0) {
    printf("fork() returned 0 ... so this is the spawned/child process\n");
    for (int i=1; i<=24; i++) {
      printf("The ants go marching *%2d* by *%2d*, hurrah, hurrah.\n", i, i);
      usleep(500000);
    }
  }
  else {
    printf("fork() returned %d ... so this is the parent process\n", childPID);
    for (int i=1; i<=24; i++) {
      printf("The ants go marching  %2d  by  %2d , hurrah, hurrah.\n", i, i);
      usleep(1000000);
    }
  }
}
```

At this point, both processes continue simultaneously running two copies of the remaining code.

Notice that the **fork()** function returns a PID. It is interesting that the original (i.e., parent) and the spawned (i.e., child) processes both continue with the same code. Hence, there are two copies of the same code running. But after the **fork()** call, the code branches based on the return value of **fork()**. For the child (i.e., spawned) process, the return value is **0**. For the parent, the returned value is the new process' PID (unless there was an error, then **-1** is returned). The **IF** statement checks this return value and allows one chunk of code to be executed by the child and the other by the parent.

Here is an example of the output you would see. **The usleep()** function sleeps for the specified number of microseconds.

```
Forking...
fork() returned 3002 ... so this is the parent process
The ants go marching   1  by   1 , hurrah, hurrah.
fork() returned 0 ... so this is the spawned process (i.e., child)
The ants go marching * 1* by * 1*, hurrah, hurrah.
The ants go marching * 2* by * 2*, hurrah, hurrah.
The ants go marching   2  by   2 , hurrah, hurrah.
The ants go marching * 3* by * 3*, hurrah, hurrah.
The ants go marching * 4* by * 4*, hurrah, hurrah.
The ants go marching   3  by   3 , hurrah, hurrah.
The ants go marching * 5* by * 5*, hurrah, hurrah.
The ants go marching * 6* by * 6*, hurrah, hurrah.
The ants go marching   4  by   4 , hurrah, hurrah.
The ants go marching * 7* by * 7*, hurrah, hurrah.
The ants go marching * 8* by * 8*, hurrah, hurrah.
The ants go marching   5  by   5 , hurrah, hurrah.
The ants go marching * 9* by * 9*, hurrah, hurrah.
The ants go marching *10* by *10*, hurrah, hurrah.
The ants go marching   6  by   6 , hurrah, hurrah.
The ants go marching *11* by *11*, hurrah, hurrah.
The ants go marching *12* by *12*, hurrah, hurrah.
The ants go marching   7  by   7 , hurrah, hurrah.
The ants go marching *13* by *13*, hurrah, hurrah.
The ants go marching *14* by *14*, hurrah, hurrah.
The ants go marching   8  by   8 , hurrah, hurrah.
The ants go marching *15* by *15*, hurrah, hurrah.
The ants go marching *16* by *16*, hurrah, hurrah.
The ants go marching   9  by   9 , hurrah, hurrah.
The ants go marching *17* by *17*, hurrah, hurrah.
The ants go marching *18* by *18*, hurrah, hurrah.
The ants go marching  10  by  10 , hurrah, hurrah.
The ants go marching *19* by *19*, hurrah, hurrah.
The ants go marching *20* by *20*, hurrah, hurrah.
The ants go marching  11  by  11 , hurrah, hurrah.
The ants go marching *21* by *21*, hurrah, hurrah.
The ants go marching *22* by *22*, hurrah, hurrah.
The ants go marching  12  by  12 , hurrah, hurrah.
The ants go marching *23* by *23*, hurrah, hurrah.
The ants go marching *24* by *24*, hurrah, hurrah.
The ants go marching  13  by  13 , hurrah, hurrah.
The ants go marching  14  by  14 , hurrah, hurrah.
The ants go marching  15  by  15 , hurrah, hurrah.
The ants go marching  16  by  16 , hurrah, hurrah.
The ants go marching  17  by  17 , hurrah, hurrah.
The ants go marching  18  by  18 , hurrah, hurrah.
The ants go marching  19  by  19 , hurrah, hurrah.
The ants go marching  20  by  20 , hurrah, hurrah.
The ants go marching  21  by  21 , hurrah, hurrah.
The ants go marching  22  by  22 , hurrah, hurrah.
The ants go marching  23  by  23 , hurrah, hurrah.
The ants go marching  24  by  24 , hurrah, hurrah.
```

Notice how the output between the two processes is interlaced.

Your code can fork many times. But remember … each time that the code forks, your child code may fork as well (depending on how you structure your code). This could cause forking indefinitely. There is a limit to how many forks the operating system will allow. It maintains a process table … which has a finite capacity. It may be best not to test that limit 😉.

A ***fork bomb*** is a process that continually replicates itself and depletes available system resources. A rabbit virus uses this strategy as a denial-of-service attack to slow down and potentially crash a system.

The following code takes an integer as a command-line-argument and then does a double-fork that many times. If the number is high enough, it can slow down and crash the system.

Code from **forkTooMuch.c**

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

int main(int argc, char *argv[]) {
  int count;

  if (argc < 2)
    count = 1;
  else
    count = atoi(argv[1]);

  printf("Parent:  %d\n", getpid());

  for (int i=0; i<count; i++) {
    fork();
    fork();
  }

  printf("Child %d with parent %d \n", getpid(), getppid());
}
```

getpid() returns the process id of the current running process.

getppid() returns the process id of the parent process… which may be ***reparented*** if the parent has already completed.

EXEC

Now let's look at the **exec** "family" of functions in C. It allows different code to be run with the same process id. Basically, the code goes off and runs another program instead of continuing with this one. So, after a call to one of these **exec** functions, the program **does not** continue to the line of code after the **exec** call. The new program will have the same PID as the process that called the **exec** function.

There are different functions that we can use: **execl()**, **execlp()**, **execle()**, **execv()**, **execvp()**. All of them are similar in that they call another program; but they differ in terms of parameters and environment settings. Recall that when running a program, we can supply command-line arguments. These **exec** functions allow you to specify the program that you want to run as well as the command-line-arguments (as strings) required for it to run.

These functions take the command-line-arguments as an array:

```
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
```

These take the command-line-arguments as a list of parameters:

```
int execl(const char *path, const char *arg, ...);
int execlp(const char *file, const char *arg, ...);
```

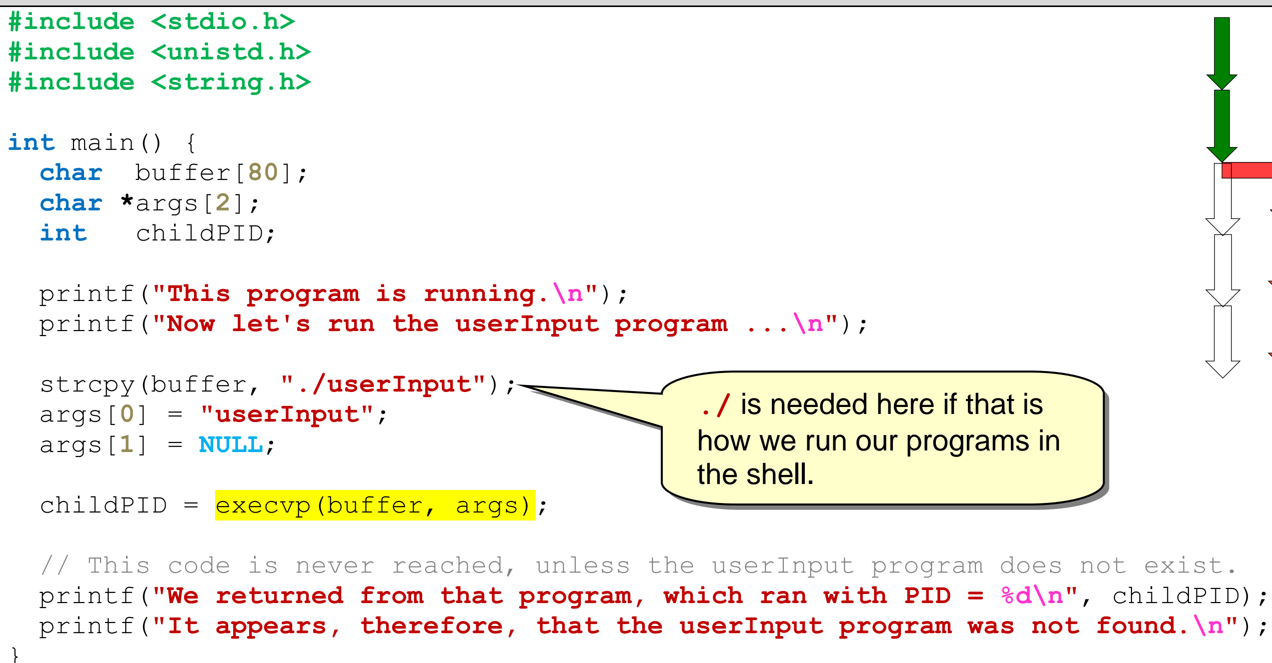
execv and **execl** both take a path to the program to run, while **execvp** and **execlp** take just the filename.

By convention, the first argument should be the name of the file being executed and the list of args should always be terminated with a **NULL** pointer.
The **execvpe** and **execle** both allow an additional array of environment pointers, but we will not discuss these in this course.

It is possible that a call to **exec** may fail. In that case, the original program simply continues.

Consider this example which calls our **userInput** program from chapter 1, which simply asks for the user's name and prints it out:

Code from **execTest.c**



```
#include <stdio.h>
#include <unistd.h>
#include <string.h>

int main() {
  char  buffer[80];
  char *args[2];
  int   childPID;

  printf("This program is running.\n");
  printf("Now let's run the userInput program ...\n");

  strcpy(buffer, "./userInput");
  args[0] = "userInput";
  args[1] = NULL;

  childPID = execvp(buffer, args);

  // This code is never reached, unless the userInput program does not exist.
  printf("We returned from that program, which ran with PID = %d\n", childPID);
  printf("It appears, therefore, that the userInput program was not found.\n");
}
```


Here is what happens when we run:

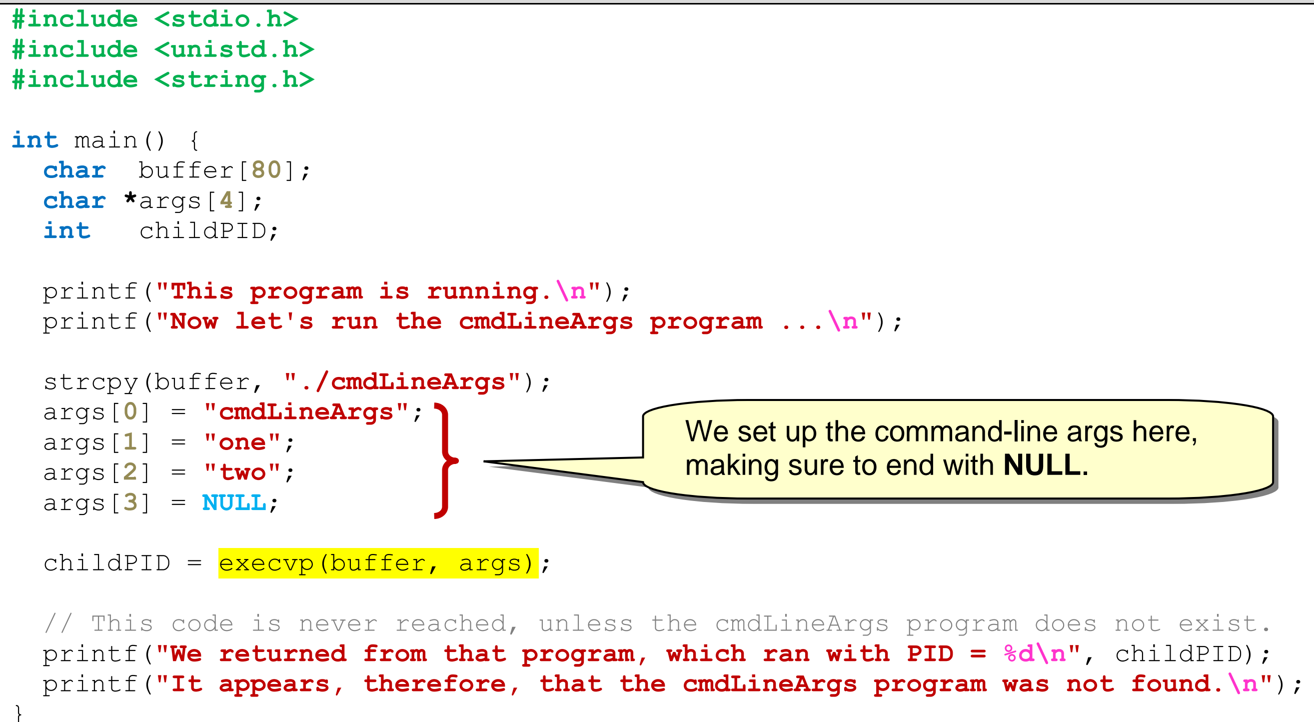
```
This program is running.
Now let's run the userInput program ...
What is your name ?
Mark
Hello, Mark
```

Of course, if the **userInput** program cannot be found, we would get this output:

```
This program is running.
Now let's run the userInput program ...
We have returned from that program, which ran with PID = -1
It appears, therefore, that the userInput program was not found.
```

Here is a variation that allows us to pass command line arguments into a program through a call to **exevvp()**. It makes use of our **cmdLineArgs** program that we wrote in chapter 3:

Code from **execTest2.c**



```
#include <stdio.h>
#include <unistd.h>
#include <string.h>

int main() {
  char  buffer[80];
  char *args[4];
  int   childPID;

  printf("This program is running.\n");
  printf("Now let's run the cmdLineArgs program ...\n");

  strcpy(buffer, "./cmdLineArgs");
  args[0] = "cmdLineArgs";
  args[1] = "one";
  args[2] = "two";
  args[3] = NULL;

  childPID = execvp(buffer, args);

  // This code is never reached, unless the cmdLineArgs program does not exist.
  printf("We returned from that program, which ran with PID = %d\n", childPID);
  printf("It appears, therefore, that the cmdLineArgs program was not found.\n");
}
```

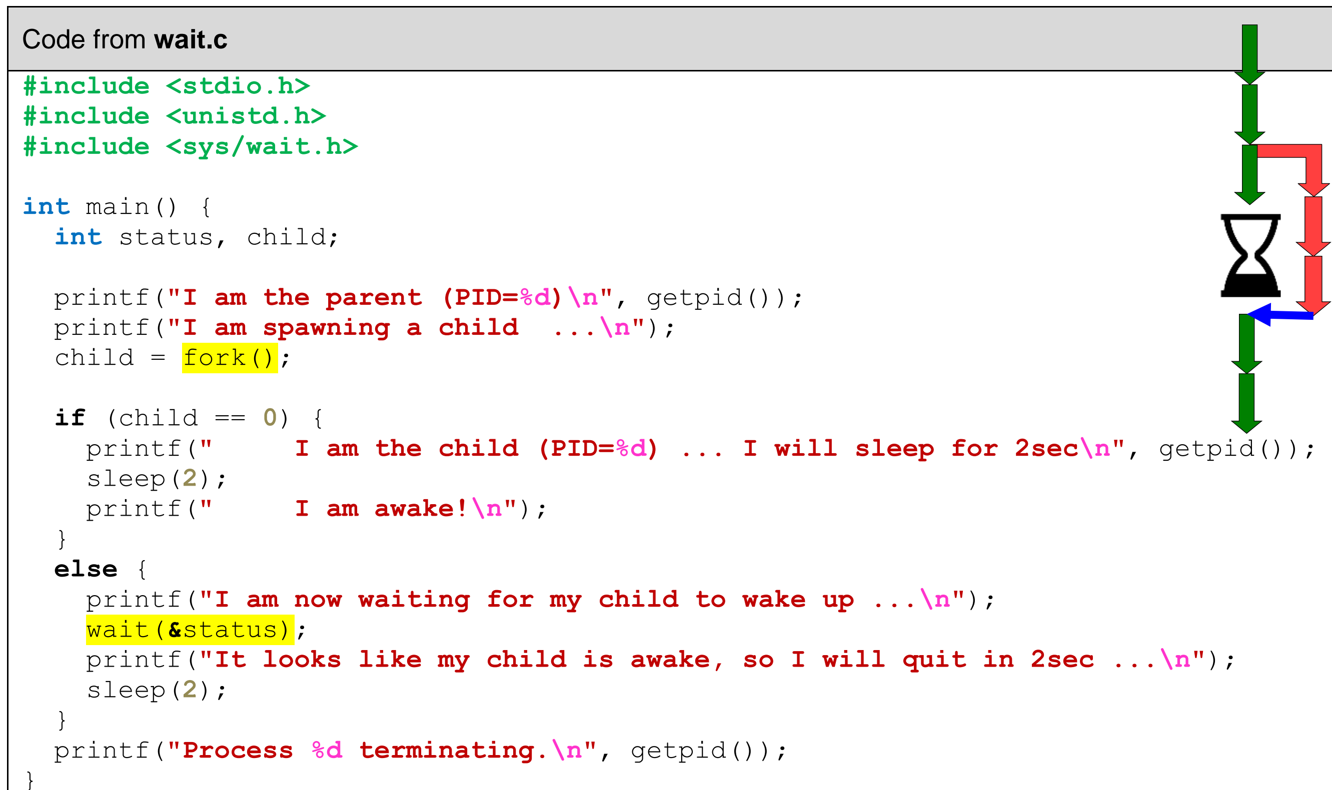

Here is the expected output:

```
This program is running.
Now let's run the cmdLineArgs program ...
There are 3 arguments
Argument 0 is cmdLineArgs
Argument 1 is one
Argument 2 is two
```

WAIT

The **wait()** function in C allows us to put a delay in a parent program so that it waits until one of its child processes has completed. It returns the PID of the child that completes, if successful, otherwise it returns **-1**. In addition, there is a **waitpid()** command that allows the parent process to delay until a specific child process has completed.

Consider this program which shows a basic use of the **wait()** function:



Code from **wait.c**

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

int main() {
  int status, child;

  printf("I am the parent (PID=%d)\n", getpid());
  printf("I am spawning a child  ...\n");
  child = fork();

  if (child == 0) {
    printf("     I am the child (PID=%d) ... I will sleep for 2sec\n", getpid());
    sleep(2);
    printf("     I am awake!\n");
  }
  else {
    printf("I am now waiting for my child to wake up ...\n");
    wait(&status);
    printf("It looks like my child is awake, so I will quit in 2sec ...\n");
    sleep(2);
  }
  printf("Process %d terminating.\n", getpid());
}
```

Here is the output:

Parent's output in blue.

Child's output in red.

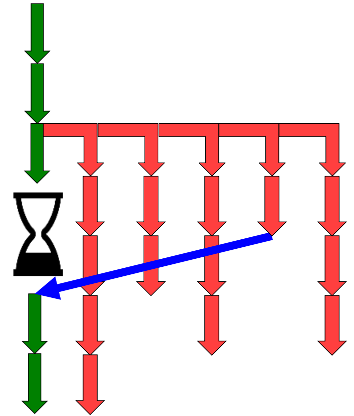
```
I am the parent (PID=24439)
I am spawning a child  ...
I am now waiting for my child to wake up ...
     I am the child (PID=24440) ... I will sleep for 2sec
     I am awake!
Process 24440 terminating.
It looks like my child is awake, so I will quit in 2sec ...
Process 24439 terminating.
```

The above example had only one child. The **wait()** command allows the process to wait for ANY child to complete. The PID of the child that completes will be returned from the **wait()** command.

Here is an example that spawns 5 children, each one sleeping for a random number of seconds, then waking up and quitting. The parent spawns all 5 children and then waits for each one to complete. Note that the order in which the children complete will be different from the order that they are spawned in, due to the random sleep time.

Notice that the children each quit by using the **exit(0)** function. The parameter to the **exit()** function is arbitrary, but zero usually indicates that all went well and negative numbers or positive numbers usually indicate error codes.

Code from **multiChildWait.c**



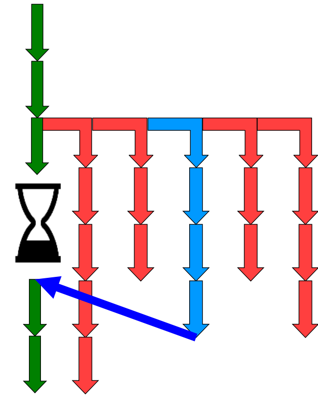
```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>

int main() {
  int status, child, parent, children[5], sleepTimes[5];

  printf("I am the parent (PID=%d)\n", parent = getpid());

  // Choose 5 random sleep times
  for (int i=0; i<5; i++) {
    sleepTimes[i] = rand()%5 + 5;
  }

  printf("I am spawning 5 children  ...\n");
  for (int i=0; i<5; i++) {
    if (getpid() == parent)
      children[i] = fork();
    if (children[i] == 0) {
      printf("     I am a child (PID=%d) ... I will sleep for %dsec\n",
             getpid(), sleepTimes[i]);
      sleep(sleepTimes[i]);
      printf("     I am awake!  Process %d terminating.\n", getpid());
      exit(0);
    }
  }

  printf("I am now waiting for all of my children to wake up ...\n");
  for (int i=0; i<5; i++) {
    child = wait(&status);
    printf("It looks like one of my children (PID=%d) has awoken.\n", child);
  }
  printf("All children are done. Process %d terminating.\n", getpid());
}
```

Here is some output:

```
I am the parent (PID=3099)
I am spawning 5 children  ...
I am now waiting for all of my children to wake up ...
     I am a child (PID=3104) ... I will sleep for 8sec
     I am a child (PID=3103) ... I will sleep for 5sec
     I am a child (PID=3102) ... I will sleep for 7sec
     I am a child (PID=3101) ... I will sleep for 6sec
     I am a child (PID=3100) ... I will sleep for 8sec
     I am awake!  Process 3103 terminating.
It looks like one of my children (PID=3103) has awoken.
     I am awake!  Process 3101 terminating.
It looks like one of my children (PID=3101) has awoken.
     I am awake!  Process 3102 terminating.
It looks like one of my children (PID=3102) has awoken.
     I am awake!  Process 3104 terminating.
It looks like one of my children (PID=3104) has awoken.
     I am awake!  Process 3100 terminating.
It looks like one of my children (PID=3100) has awoken.
All children are done, so I will quit now. Process 3099 terminating.
```

The **waitpid()** function can be used to wait for a particular child to complete. It returns the child PID if successful, otherwise **-1** if an error occurred. As an example, we could determine the child that would likely take the longest to complete the work and then wait just for that child. The sleep times are hardcoded to make it clearer:

Code from **waitpid.c**



```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>

int main() {
  int status, child, parent, children[5];
  int sleepTimes[5] = {1, 5, 8, 2, 4};

  printf("I am the parent (PID=%d)\n", parent = getpid());
  printf("I am spawning 5 children  ...\n");
  for (int i=0; i<5; i++) {
    if (getpid() == parent)
      children[i] = fork();
    // Note that for the parent process, children[i] is set to
    // the pid of the newly-created child process.   However,
    // for the child process program, all the values of children[i]
    // will be 0.
    if (children[i] == 0) {
      printf("     I am a child (PID=%d) ... I will sleep for %dsec\n",
             getpid(), sleepTimes[i]);
      sleep(sleepTimes[i]);
      printf("     I am awake!  Process %d terminating.\n", getpid());
      exit(0);
    }
  }

  printf("I am now waiting for child 3 to wake up ...\n");
  child = waitpid(children[2], &status, 0);
  printf("It looks like my slowest child (PID=%d) has awoken.\n", child);
  printf("All children are done. Process %d terminating.\n", getpid());
}
```

Here is the output:

```
I am the parent (PID=3303)
I am spawning 5 children  ...
I am now waiting for child 3 to wake up ...
    I am a child (PID=3308) ... I will sleep for 4sec
    I am a child (PID=3307) ... I will sleep for 2sec
    I am a child (PID=3306) ... I will sleep for 8sec
    I am a child (PID=3305) ... I will sleep for 5sec
    I am a child (PID=3304) ... I will sleep for 1sec
    I am awake!  Process 3304 terminating.
    I am awake!  Process 3307 terminating.
    I am awake!  Process 3308 terminating.
    I am awake!  Process 3305 terminating.
    I am awake!  Process 3306 terminating.
It looks like my slowest child (PID=3306) has awoken.
All children are done.  Process 3303 terminating.
```

Notice that the **waitpid()** function takes a third parameter … these are options. We will not discuss the various **status** results from the function, nor these options. Please see the **man** pages if you are interested in more details.

SYSTEM

The **system()** function in C allows us to run the specified command (or program) as a shell command. When called, the process blocks until the system call is done and then control returns to the program. The return value from this function call is the value that is returned from the system call command, or **-1** if an error has occurred.

Here is a simple program that calls a couple of shell commands as well as running another program from within it:

Code from **systemCall.c**

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

int main() {
  // Show a list of files
  system("clear");
  system("ls");
  printf("\n");

  // Find out who the user is
  system("who");
  printf("\n");

  // Run the userInput program
  system("./userInput");
}
```

Here is the output, which of course depends on the directory contents:

```
cmdLineArgs  execTest.c   forkTooMuch.c     shellProcess.c  wait
execTest     fork         multiChildWait    systemCall      wait.c
execTest2    fork.c       multiChildWait.c  systemCall.c    waitpid
execTest2.c  forkTooMuch  shellProcess      userInput       waitpid.c

student  tty7        2018-06-05 10:45 (:0)

What is your name ?
Mark
Hello, Mark
```

5.3 Inter-Process Communication

Now that you have a good understanding of how to create multiple processes, you probably realize that this is most useful when the processes have the ability to communicate with one another as they are running. This relates to reality since people often work as a team, each doing their own task, yet coordinating through careful communication.

In computer science, this communication is done through …

> ***Inter-Process Communication*** *(IPC) is the sending and receiving of information between processes.*

Communication between processes can occur on the same host machine or between processes running on different hosts across a network.

There are two main approaches to IPC. The first (and most basic) is that of using signals:

> *A* ***signal*** *is a value (integer) sent from one process to another.*

A signal is used as a rudimentary form of communication to do simple things like informing processes of an error or telling a process to terminate. It is a very limited kind of communication that can only be used between processes running on the same host machine.

In C, there are a fixed set of existing signal values defined in the **<signal.h>** header file, but only two are user-defined:

```
#define      SIGHUP       1       /* Hangup (POSIX).  */
#define      SIGINT       2       /* Interrupt (ANSI).  */
#define      SIGQUIT      3       /* Quit (POSIX).  */
#define      SIGILL       4       /* Illegal instruction (ANSI).  */
#define      SIGTRAP      5       /* Trace trap (POSIX).  */
#define      SIGABRT      6       /* Abort (ANSI).  */
#define      SIGIOT       6       /* IOT trap (4.2 BSD).  */
#define      SIGBUS       7       /* BUS error (4.2 BSD).  */
#define      SIGFPE       8       /* Floating-point exception (ANSI).  */
#define      SIGKILL      9       /* Kill, unblockable (POSIX).  */
#define      SIGUSR1      10      /* User-defined signal 1 (POSIX).  */
#define      SIGSEGV      11      /* Segmentation violation (ANSI).  */
#define      SIGUSR2      12      /* User-defined signal 2 (POSIX).  */
#define      SIGPIPE      13      /* Broken pipe (POSIX).  */
#define      SIGALRM      14      /* Alarm clock (POSIX).  */
#define      SIGTERM      15      /* Termination (ANSI).  */
#define      SIGSTKFLT    16      /* Stack fault.  */
#define      SIGCLD       SIGCHLD      /* Same as SIGCHLD (System V).  */
#define      SIGCHLD      17      /* Child status has changed (POSIX).  */
#define      SIGCONT      18      /* Continue (POSIX).  */
#define      SIGSTOP      19      /* Stop, unblockable (POSIX).  */
#define      SIGTSTP      20      /* Keyboard stop (POSIX).  */
```

```
#define     SIGTTIN     21     /* Background read from tty (POSIX).  */
#define     SIGTTOU     22     /* Background write to tty (POSIX).  */
#define     SIGURG      23     /* Urgent condition on socket (4.2 BSD).  */
#define     SIGXCPU     24     /* CPU limit exceeded (4.2 BSD).  */
#define     SIGXFSZ     25     /* File size limit exceeded (4.2 BSD).  */
#define     SIGVTALRM   26     /* Virtual alarm clock (4.2 BSD).  */
#define     SIGPROF     27     /* Profiling alarm clock (4.2 BSD).  */
#define     SIGWINCH    28     /* Window size change (4.3 BSD, Sun).  */
#define     SIGPOLL     SIGIO  /* Pollable event occurred (System V).  */
#define     SIGIO       29     /* I/O now possible (4.2 BSD).  */
#define     SIGPWR      30     /* Power failure restart (System V).  */
#define     SIGSYS      31     /* Bad system call.  */
#define     SIGUNUSED   31
```

There are two steps to using signals: (1) install a signal handler, (2) send a signal.

Installing a signal handler is really just a matter of indicating which function will be called when the signal is received. It is similar to setting up an event handler in JAVA.

Every signal should have its own signal handler. There is a default signal handler for every signal … which, by default, will usually terminate the program.

To install our own signal handler, we use the **signal()** function which takes the signal number/code (i.e., `SIGUSR1` or `SIGUSR2`) as its first parameter and the signal-handler function name as its second parameter. The signal handler function must take a single **int** parameter and have a **void** return type. Optionally, instead of supplying a signal handler function, we can use the constant `SIG_IGN` to tell the OS to ignore the signal and do nothing … or we can use `SIG_DFL` to tell the OS to use the default signal handler.

Here is an example of a program that will wait for some incoming signal from another process. It does not do anything interesting, but it shows the mechanics of setting up inter-process communications between processes.

Code from **handler.c**

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>

void handleSig1(int);
void handleSig2(int);

int main() {
  signal(SIGUSR1, handleSig1);
  signal(SIGUSR2, handleSig2);

  printf("\n  HANDLER: Running (PID=%d)\n", getpid());

  // Go into an infinite loop
  while (1)
    sleep(1);

  printf("This line of code is never reached.\n");
}
```

```
void handleSig1(int i) {
  printf("  HANDLER: Signal 1 has been received.  Continuing...\n");
}

void handleSig2(int i) {
  printf("  HANDLER: Signal 2 has been received.  Quitting...\n");
  exit(SIGUSR2);
}
```

Notice that when the program receives signal **1**, it prints a message and the program continues. When it receives signal **2**, however, it stops running.

We will run this program in the background and then set up another program that allows us to send signals to it:

```
student@COMPBase:~$ gcc -o handler handler.c
student@COMPBase:~$ ./handler &
[8] 4018
student@COMPBase:~$
HANDLER: Running (PID=4018)

student@COMPBase:~$
```

To send a signal to a process, we need to know the PID and the signal number. Then we make use of the **kill()** function which takes the PID as its first parameter and the signal number as its second parameter. (I know, it doesn't make sense to use kill() to send a signal instead of something like send(), but often signals are sent to kill a process). The function will return **-1** if there was a problem (e.g., process does not exist) and **0** otherwise. Now let us write the sending program:

Code from **sender.c**

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

int main() {
  int pid, choice, result;

  printf("SENDER: Enter PID that you want to signal: ");
  scanf("%d", &pid);

  while (1) {
    printf("SENDER: Enter signal number (1 or 2), use 0 to quit: ");
    scanf("%d", &choice);
    switch(choice) {
      case 0: exit(0);
      case 1: result = kill(pid, SIGUSR1); break;
      case 2: result = kill(pid, SIGUSR2);
    }
    if (result == -1)
      printf("SENDER: *** Error sending signal to Process %d ***\n", pid);
  }
}
```

The code allows us to first enter the PID of the process that we want to communicate with. Then it goes into an infinite loop allowing us to send repeated signals. The only two signals that we will send are **SIGUSR1** and **SIGUSR2** which are selected based on the value that the user enters. If the **kill()** function returns **-1**, then we know there was a problem (e.g., the process may no longer be running).

Assuming that the **handler** program is already running in the background, on the next page it shows what we may see as output from this program. The values entered by the user in the **sender** program are highlighted as yellow and the output from the **handler** program is shown in orange so that it is easier to see what is happening.

```
student@COMPBase:~$ gcc -o sender sender.c
student@COMPBase:~$ ./sender &
SENDER: Enter PID that you want to signal: 4018
SENDER: Enter signal number (1 or 2), use 0 to quit: 1
  HANDLER: Signal 1 has been received.  Continuing...
SENDER: Enter signal number (1 or 2), use 0 to quit: 1
  HANDLER: Signal 1 has been received.  Continuing...
SENDER: Enter signal number (1 or 2), use 0 to quit: 2
  HANDLER: Signal 2 has been received.  Quitting...
SENDER: Enter signal number (1 or 2), use 0 to quit: 1
SENDER: *** Error sending signal to Process 4018 ***
SENDER: Enter signal number (1 or 2), use 0 to quit: 2
SENDER: *** Error sending signal to Process 4018 ***
SENDER: Enter signal number (1 or 2), use 0 to quit: 0
[8]+  Exit 12                 ./handler
student@COMPBase:~$
```

Here is an example that shows how we can send a "kill" command (**SIGKILL**) to spawned child processes to have them stop right away. Notice the use of the **system("ps")**. This will allow us to print out the running processes on the terminal that we are using so that we can see that the processes are started and stopped:

Code from **stopChildren.c**

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>

int main() {
  int parent, childProcess[5];

  printf("I am the parent (PID=%d)\n", parent = getpid());
  printf("I am spawning 3 children  ...\n");
  for (int i=0; i<3; i++) {
    if (getpid() == parent)
      childProcess[i] = fork();
    if (childProcess[i] == 0) {
      for (int j=30; j>0; j--) {
        printf("     Child (PID=%d) sleeping for %d more sec\n", getpid(), j);
        sleep(1);
      }
```

```
      exit(0);
    }
  }
  system("ps");

  printf("I am now waiting for 3 seconds then will stop all the children ...\n");
  sleep(3);
  for (int i=0; i<3; i++)
    kill(childProcess[i], SIGKILL);

  system("ps");
  printf("I stopped all child processes ... terminating now.\n");
}
```

Here is the output that can be expected:

```
I am the parent (PID=3318)
I am spawning 3 children  ...
     Child (PID=3321) sleeping for 30 more sec
     Child (PID=3320) sleeping for 30 more sec
     Child (PID=3319) sleeping for 30 more sec
  PID TTY          TIME CMD
 2691 pts/0    00:00:00 bash
 3318 pts/0    00:00:00 stopChildren
 3319 pts/0    00:00:00 stopChildren
 3320 pts/0    00:00:00 stopChildren
 3321 pts/0    00:00:00 stopChildren
 3322 pts/0    00:00:00 sh
 3323 pts/0    00:00:00 ps
I am now waiting for 3 seconds then will stop all the children ...
     Child (PID=3321) sleeping for 29 more sec
     Child (PID=3320) sleeping for 29 more sec
     Child (PID=3319) sleeping for 29 more sec
     Child (PID=3320) sleeping for 28 more sec
     Child (PID=3321) sleeping for 28 more sec
     Child (PID=3319) sleeping for 28 more sec
     Child (PID=3321) sleeping for 27 more sec
     Child (PID=3320) sleeping for 27 more sec
     Child (PID=3319) sleeping for 27 more sec
  PID TTY          TIME CMD
 2691 pts/0    00:00:00 bash
 3318 pts/0    00:00:00 stopChildren
 3319 pts/0    00:00:00 stopChildren <defunct>
 3320 pts/0    00:00:00 stopChildren <defunct>
 3321 pts/0    00:00:00 stopChildren <defunct>
 3332 pts/0    00:00:00 sh
 3333 pts/0    00:00:00 ps
I stopped all child processes ... terminating now.
```

Parent is running

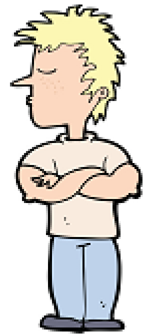
Children are running

Children are no longer running

Let us try dealing with the **SIGINT** signal. This is the signal that occurs when the system tries to interrupt the process. One way that we can generate the signal is to press the **CTRL-C** keys. By default, this quits the program. But we can disable this … by ignoring that signal (not a good idea usually). Here is a program that does this. We'll first ignore the **CTRL-C** for 5 seconds … then we'll handle it ourselves for 5 seconds by simply printing a message out, then finally we'll spend the last 5 seconds with the restored default, which will allow us to quit the program.

Code from **ignoreInterrupt.c**

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>

void ignoreMessage(int);
void sleep5();

int main() {
  printf("  Process %d running\n", getpid());
  printf("  Ignoring the interrupt signal...\n");
  signal(SIGINT, SIG_IGN);

  sleep5();

  printf("\n  Really ignoring the interrupt signal...\n");
  signal(SIGINT, ignoreMessage);

  sleep5();

  printf("\n  Restoring the default handler...\n");
  signal(SIGINT, SIG_DFL);

  sleep5();

  printf("\n  All done!\n");
}

void ignoreMessage(int x) {
  printf("  Stop bugging me.\n");
}

void sleep5() {
  for (int i=1; i<=5; ++i) {
    sleep(1);
    printf("  Sleeping %d\n",i);
  }
}
```



Here is the output, showing **^C** when **CTRL-C** was pressed:

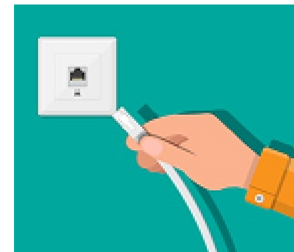
```
  Process 4340 running
  Ignoring the interrupt signal...
  Sleeping 1
  Sleeping 2
^C  Sleeping 3
  Sleeping 4
^C  Sleeping 5

  Really ignoring the interrupt signal...
  Sleeping 1
  Sleeping 2
^C  Stop bugging me.
  Sleeping 3
  Sleeping 4
```

```
^C  Stop bugging me.
  Sleeping 5

  Restoring the default handler...
  Sleeping 1
^C
```

As you can see, simple communication between two processes is not difficult. However, with the signaling approach, there are obvious limitations in that we can only ***signal*** another process … we cannot really exchange data.

Of course, we can "fake" data exchange by, for example, having one process write data to a file and then signal the other process to read the file when it is done. But this is cumbersome and also limited in regard to how many processes can be involved in this type of communication. A better way to do this is by using ***sockets***:

*A **socket** is an endpoint for sending or receiving data between processes.*



You can think of two hosts communicating with one another through a physical cable (or through wifi these days). The socket is like the connector that we plug the cable into. Each host has its own socket and all communication to other hosts takes place through this socket connection.

Since each computer/host on a network has a unique **IP address**, we will need to use this address in order to communicate with that host through the socket. It uniquely identifies a computer at the network layer. Also, since multiple processes may run on the same host machine, they too must be uniquely identifiable through a **port number** which will be unique to all applications running on that host. The port uniquely identifies a process (e.g., app) at the transport layer. Only a specific range of values can be used … from **1025** to **65536** … (i.e., **0** through **1024** are reserved). Perhaps you can think of putting a note in a friend's locker at school. The ***IP address*** corresponds to the address of the particular school … while the ***port*** would correspond to the locker number in that school.

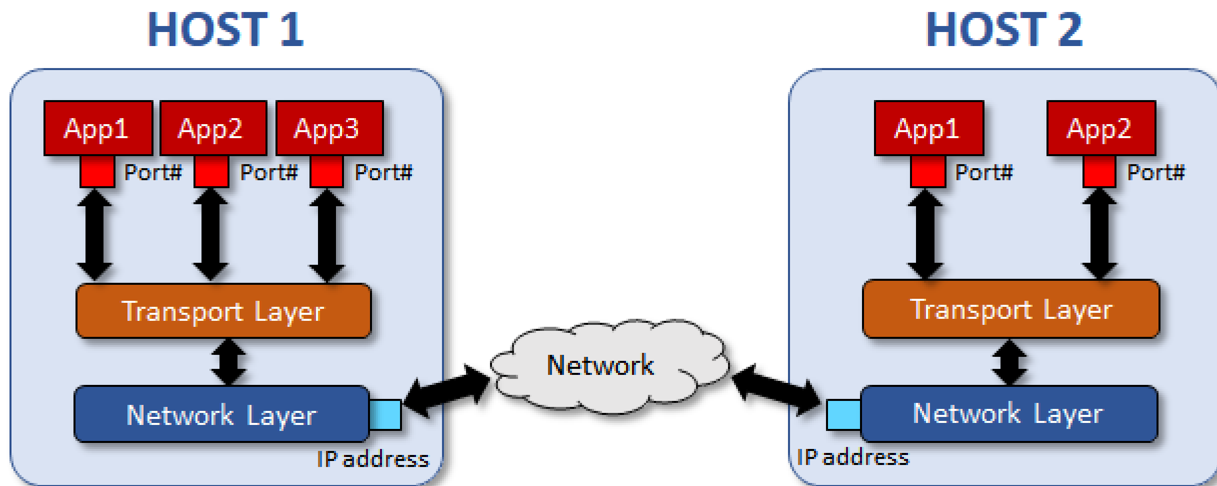
Communication between the processes occurs through a couple of layers. The **Network Layer** provides the means of sending packets of data from a source host to a destination host over one or more networks. It is basically like the mailman delivering letters from one building in one city to another building in another city.

The **Transport Layer** is a conceptual layer that indicates how exactly the data is to be transferred from the source to the destination. There are two main strategies for doing this:

(1) **Transmission Control Protocol** (TCP), and

(2) **User Datagram Protocol** (UDP).

The following diagram shows how things are organized:



There are 3 types of sockets:

1. Stream sockets

- These are connection-based sockets.
    - Connection must first be established between the sender and receiver before any data exchange can take place (e.g., like making a phone call).
    - Connection must be closed (i.e., must hang up the phone) when communication is finished (i.e., no "call-waiting" option).
- Best used for reliable packet delivery … so that the packet is correct and in a reliable order.
- Works with the TCP (Transmission Control Protocol) method of data exchange.

2. Datagram sockets

- These are connectionless sockets.
    - Don't need to first establish a connection between sender and receiver, data is just sent out when ready (e.g., like mailing a package via Canada Post).
- Best used for faster packet sending (i.e., but not necessarily faster receiving). No need to establish a connection beforehand.
- Works with the UDP (User Datagram Protocol) method of data exchange.
- Disadvantage is that the packets can be corrupted, received out of order, lost altogether or delivered multiple times.

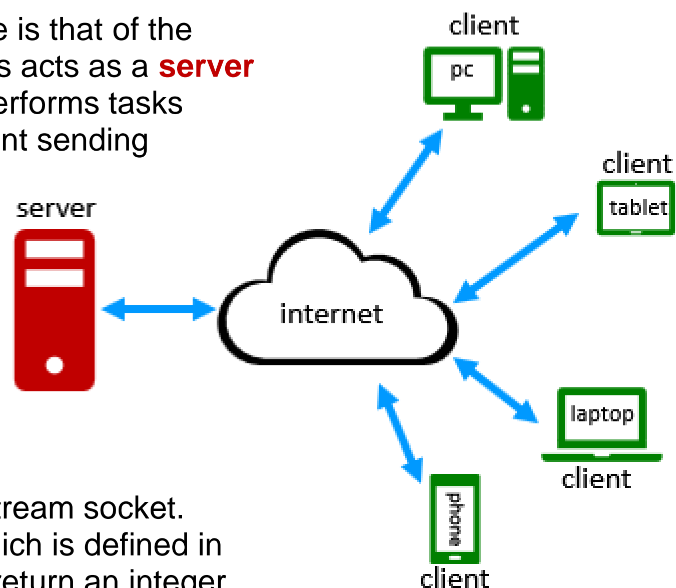
3. **Raw sockets**
   - Bypasses the Transport protocol all together.

The basic idea being socket communications is as follows:

1. Each endpoint (i.e., sender and receiver) opens a socket … and a connection is established if using stream sockets.
2. Packets are sent and received.
3. Each endpoint closes their socket.

Client/Server Model - TCP

In IPC, one commonly used type of architecture is that of the client/server model. In this model, one process acts as a **server** that receives requests from **clients** and then performs tasks accordingly. There may be more than one client sending requests to the server at any time.



Let us look now at an example that uses stream sockets to perform connection-based communications between two processes. We will run two processes on the same machine and have data passed back and forth between them.

Starting with the **server**, we need to create a stream socket. This can be done with the **socket()** function which is defined in the **<sys/socket.h>** header. The function will return an integer representing the socket descriptor (i.e., ID), or **-1** if the socket cannot be opened for any reason. The function takes three parameters with this template:

socket(<*domain*>, <*type*>, <*protocol*>)

There are many options for these parameters, but just a couple will be mentioned here.

The <*domain*> is the address domain family that we want to use:
- AF_INET = communication over a network
- AF_LOCAL = communication on the local host

The <*type*> is the type of socket that we want to use:
- SOCK_STREAM = connection-based
- SOCK_DGRAM = connection-less

The <*protocol*> is the protocol that we want to use:
- IPPROTO_TCP = Transmission Control Protocol
- IPPROTO_UDP = User Datagram Protocol

The opening of the socket can fail if:

- The implementation does not support the specified address family.
- No more file descriptors are available for this process/system.
- The protocol is not supported by the address family/implementation.
- The socket type is not supported by the protocol.
- The process does not have appropriate privileges.
- Insufficient resources were available in the system to perform the operation.
- Insufficient memory was available to fulfill the request.

Once the socket has been opened, we then need to assign an IP address to the socket from which we will accept messages and we also need to assign a port number to the socket. We do this by using the **bind()** function which has this format:

bind(<*serverSocket*>, <*address*>, <*address_Length*>)

The <*serverSocket*> parameter is the socket descriptor (i.e., ID) that was returned from the **socket()** function call. The <*address*>, however, is a bit more complicated. It is a **struct** sockaddr data structure and the <*address_Length*> is the length of the **struct** sockaddr structure supplied as the 2nd parameter. The function will return **-1** if an error occurred, otherwise **0** is returned.

What does the **struct** sockaddr look like ? Well, this is a protocol-independent structure. At the general level, it is defined like this:

```
struct sockaddr {
  unsigned short   sa_family;   // address family
  char             sa_data[14]; // protocol address
};
```

The **sa_data** field is quite general and allows **14** bytes to be adjustable for various types of protocols. We generally set things up for **IPv4** (i.e., version 4 of the internet protocol) by using **struct** sockaddr_in instead, which is defined as follows:

```
struct sockaddr_in {
  short            sin_family;  // e.g. AF_INET, AF_LOCAL
  unsigned short   sin_port;    // port number
  struct in_addr   sin_addr;    // see below
  char             sin_zero[8]; // unused
};
```

where **struct** in_addr is defined as follows:

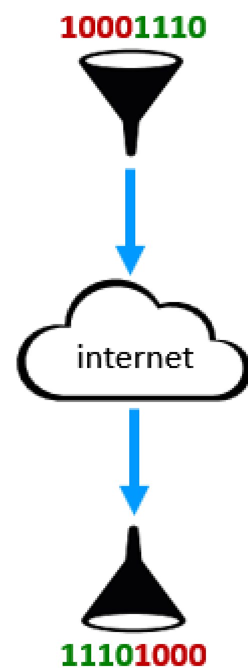
```
struct in_addr {
  unsigned int   s_addr;      // set to internet address
};
```

You may have noticed that if we add up the bytes required for **sin_port**, **sin_addr** and **sin_zero** … they add to the **14** bytes defined in **sa_data** from the **sockaddr** struct, since an **unsigned int** is only **4** bytes on the virtual machine that we are using. So the **sin_zero** field

of the **sockaddr** struct is just a placeholder to use up the remaining required **14** bytes (that we do not need) in order to the **sizeof**(**struct** sockaddr_in) to be the same as **sizeof**(**struct** sockaddr). This will allow us to typecast (**struct** sockaddr_in*) to (**struct** sockaddr*) later.

Now … what should we set sin_family, sin_port and sin_addr to ? We can set the sin_family to AF_INET, or whatever we used to set up the socket. The sin_port number can be arbitrary (e.g., 6000). The sin_addr can be set to any internet address. If we just want the server to receive requests from the local lost machine … we would set this to the specific IP address **inet_addr("127.0.0.1")**. However, for servers, we generally want to accept incoming requests from any network interface. In that case, we can set the sin_addr to INADDR_ANY … which will allow the server to accept all UDP packets and TCP connection requests made for its port, regardless of the network interface on which the requests arrived.

There is one concern though in setting up the struct. The IP address and port number are to be sent over the internet as **bytes** but interpreted as **ints** and **longs**.



Recall that some machines use little-endian format and some use big-endian format. So, sending out a **short** or a **long** from one machine that uses one format … might be misinterpreted if read in from a machine that uses a different format. To deal with this, there are some handy conversion functions for converting to a common ordering. As it turns out, network protocols assume big-endian format. The host format can be either format. Here are the functions that we can use to convert from the host format to the network format and vice versa:

- **htons()** – convert **short** from **host** format to **network** format.
- **htonl()** – convert **long** from **host** format to **network** format.
- **ntohs()** – convert **short** from **network** format to **host** format.
- **ntohl()** – convert **long** from **network** format to **host** format.

Therefore, this is how we would set the addres information for a server:

```
#define    SERVER_PORT 6000

struct sockaddr_in    address;

memset(&address, 0, sizeof(address)); // zeros the struct
address.sin_family = AF_INET;
address.sin_addr.s_addr = htonl(INADDR_ANY);
address.sin_port = htons((unsigned short) SERVER_PORT);
```

Once this has been set up, we can call **bind()** with the **address** variable:

```
bind(serverSocket, (struct sockaddr *)address, sizeof(address));
```

Notice the typecast of the address. This is necessary since the function wants something of type **sockaddr**, not **sockaddr_in**. After calling this function, we will also need to check to make sure that the **bind()** function did not return **-1** before we continue.

Once the socket is opened and bound, we are ready to start listening for incoming requests. The **listen()** function is used to set the socket up for listening, which has this format:

> **listen**(<*serverSocket*>, <*backlog*>)

Again, the socket descriptor is used. The <*backlog*> is a value that indicates the number of pending connections that may be queued (i.e., the number of clients allowed to wait in line before being turned away). This can be set to something small, such as **5** or **10**. For the **listen()** function, a return value of **0** indicates that all went well, otherwise **-1** is returned.

Finally, we need to use the **accept()** function to "wait for" and "accept" an incoming client request. It has the following format:

> **accept**(<*serverSocket*>, <*clientAddress* ***** >, <*clientAddressLength* ***** >)

Once again, the socket descriptor is used. The <*clientAddress* ***** > is a **struct** sockaddr * just as we had used for the server address. This, however, is a pointer to a variable that will contain the client address once the message arrives.

The <*clientAddressLength* * > should point to an integer that represents the exact size of the *clientAddress* struct. It is a pointer, because before returning, the function will change this integer to represent the size required to represent the address of the connecting socket. Once again, a return value of **-1** from **accept()** is used to indicate that an error has occurred. When all went well, however, the **accept()** function returns a socket descriptor (i.e., clientSocket) that corresponds to the client that just connected to the server.

At this point, we have established a one-on-one connection between the server and the client. We can now read in the information that was sent from the client by using the **recv()** function which has this format:

> **recv**(<*clientSocket*>, <*buffer*>, <*bufferLength*>, <*flags*>)

Notice that we now use the <*clientSocket*> as the first parameter … this is NOT the server socket. It is the socket descriptor that is returned from the call to **accept()**. The <*buffer*> is a pointer to some memory that can take the incoming request. We can set it up as a **char** *. Finally, the <*bufferLength*>is the number of bytes that the buffer can hold. It should not exceed the amount of memory reserved for the buffer itself. We will not discuss the <*flags*> here … but will set them to **0**. The **recv()** function will return the number of incoming bytes that were received.

We can even send information back to the client using the **send()** function with this format:

> **send**(<*clientSocket*>, <*buffer*>, <*bufferLength*>, <*flags*>)

The idea is the same. We simply set up the buffer that we want to send and then send it.

Normally, with a server, we have a kind of recv/send sequence in a loop of some sort, so that communication between the client and server can go back and forth for a while. We will also likely want the server to serve many clients, so another loop is normally used to keep accepting new clients.

Here, on the right, is the pseudocode for setting up the server →

```
Open the socket
Bind the socket
Listen on the socket
while (true) {
  Accept a socket request
  while (client has not "hung up" yet) {
    Receive the buffer from the client
    Process the request
    Send a response to the client
  }
  Close client socket
}
Close server socket
```

Here is the code for the server in its entirety:

Code from **server.c**

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/socket.h>
#include <netinet/in.h>

#define  SERVER_PORT 6000

int main() {
  int                 serverSocket, clientSocket;
  struct sockaddr_in  serverAddress, clientAddr;
  int                 status, addrSize, bytesRcv;
  char                buffer[30];
  char               *response = "OK";

  // Create the server socket
  serverSocket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
  if (serverSocket < 0) {
    printf("*** SERVER ERROR: Could not open socket.\n");
    exit(-1);
  }

  // Setup the server address
  memset(&serverAddress, 0, sizeof(serverAddress)); // zeros the struct
  serverAddress.sin_family = AF_INET;
  serverAddress.sin_addr.s_addr = htonl(INADDR_ANY);
  serverAddress.sin_port = htons((unsigned short) SERVER_PORT);

  // Bind the server socket
  status = bind(serverSocket, (struct sockaddr *)&serverAddress,
                                                  sizeof(serverAddress));
  if (status < 0) {
    printf("*** SERVER ERROR: Could not bind socket.\n");
    exit(-1);
  }

  //… more on next page
```

```
  // Set up the line-up to handle up to 5 clients in line
  status = listen(serverSocket, 5);
  if (status < 0) {
    printf("*** SERVER ERROR: Could not listen on socket.\n");
    exit(-1);
  }

  // Wait for clients now
  while (1) {
    addrSize = sizeof(clientAddr);
    clientSocket = accept(serverSocket,(struct sockaddr *)&clientAddr,&addrSize);
    if (clientSocket < 0) {
      printf("*** SERVER ERROR: Could accept incoming client connection.\n");
      exit(-1);
    }
    printf("SERVER: Received client connection.\n");

    // Go into infinite loop to talk to client
    while (1) {
      // Get the message from the client
      bytesRcv = recv(clientSocket, buffer, sizeof(buffer), 0);
      buffer[bytesRcv] = 0; // put a 0 at the end so we can display the string
      printf("SERVER: Received client request: %s\n", buffer);

      // Respond with an "OK" message
      printf("SERVER: Sending \"%s\" to client\n", response);
      send(clientSocket, response, strlen(response), 0);
      if ((strcmp(buffer,"done") == 0) || (strcmp(buffer,"stop") == 0))
        break;
    }
    printf("SERVER: Closing client connection.\n");
    close(clientSocket); // Close this client's socket

    // If the client said to stop, then I'll stop myself
    if (strcmp(buffer,"stop") == 0)
      break;
  }

  // Don't forget to close the sockets!
  close(serverSocket);
  printf("SERVER: Shutting down.\n");
}
```

Now, what about the client ? The client is structured very similarly. The socket is created the same way. Instead of using **bind()** though, we use **connect()** … which has the same parameters.

For the s_addr of the **struct** sockaddr_in, however, we will set it to **inet_addr("127.0.0.1")**, which is the local machine.

Here is the completed client code:

Code from **client.c**

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#define  SERVER_IP   "127.0.0.1"
#define  SERVER_PORT 6000

int main() {
  int                 clientSocket;
  struct sockaddr_in  serverAddress;
  int                 status, bytesRcv;
  char                inStr[80];  // stores user input from keyboard
  char                buffer[80]; // stores user input from keyboard

  // Create the client socket
  clientSocket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);

  if (clientSocket < 0) {
    printf("*** CLIENT ERROR: Could not open socket.\n");
    exit(-1);
  }

  // Setup address
  memset(&serverAddress, 0, sizeof(serverAddress));
  serverAddress.sin_family = AF_INET;
  serverAddress.sin_addr.s_addr = inet_addr(SERVER_IP);
  serverAddress.sin_port = htons((unsigned short) SERVER_PORT);

  // Connect to server
  status = connect(clientSocket, (struct sockaddr *) &serverAddress,
                                                       sizeof(serverAddress));
  if (status < 0) {
    printf("*** CLIENT ERROR: Could not connect.\n");
    exit(-1);
  }

  // Go into loop to commuincate with server now
  while (1) {
    // Get a command from the user
    printf("CLIENT: Enter command to send to server ... ");
    scanf("%s", inStr);

    // Send command string to server
    strcpy(buffer, inStr);
    printf("CLIENT: Sending \"%s\" to server.\n", buffer);
    send(clientSocket, buffer, strlen(buffer), 0);

    // Get response from server, should be "OK"
    bytesRcv = recv(clientSocket, buffer, 80, 0);
    buffer[bytesRcv] = 0; // put a 0 at the end so we can display the string
    printf("CLIENT: Got back response \"%s\" from server.\n", buffer);
```

```
    if ((strcmp(inStr,"done") == 0) || (strcmp(inStr,"stop") == 0))
      break;
  }

  close(clientSocket);  // Don't forget to close the socket !
  printf("CLIENT: Shutting down.\n");
}
```

As a minor detail, **scanf()** will not allow blanks to be entered. If you want that to be allowed, use this instead of the **scanf()** line:

```
fgets(inStr, sizeof(inStr), stdin);
inStr[strlen(inStr)-1] = 0;
```

Now once we have these compiled, we can run the server in the background:

```
student@COMPBase:~$ ./server &
[5] 4242
student@COMPBase:~$
```

Once the server has been started and stopped a few times in our virtual environment, it is sometimes not possible to run it right away. You may have to wait a bit before running it. Once it is running, we can run the client. Here is an example of some output that you may see. The client code is highlighted in one color, the server in another, and the user-entered command in a third color:

```
student@COMPBase:~$ ./client
SERVER: Received client connection.
CLIENT: Enter command to send to server ... Hello
CLIENT: Sending "Hello" to server.
SERVER: Received client request: Hello
SERVER: Sending "OK" to client
CLIENT: Got back response "OK" from server.
CLIENT: Enter command to send to server ... Fun
CLIENT: Sending "Fun" to server.
SERVER: Received client request: Fun
SERVER: Sending "OK" to client
CLIENT: Got back response "OK" from server.
CLIENT: Enter command to send to server ... Bored
CLIENT: Sending "Bored" to server.
SERVER: Received client request: Bored
SERVER: Sending "OK" to client
CLIENT: Got back response "OK" from server.
CLIENT: Enter command to send to server ... done
CLIENT: Sending "done" to server.
SERVER: Received client request: done
SERVER: Sending "OK" to client
SERVER: Closing client connection.
CLIENT: Got back response "OK" from server.
CLIENT: Shutting down.
student@COMPBase:~$
```

At this point, the client has stopped and the server is still running. We can run the client again and it will work with the server. Here is an example where we tell the server to **stop**:

```
student@COMPBase:~$ ./client
SERVER: Received client connection.
CLIENT: Enter command to send to server ... ItsMeAgain
CLIENT: Sending "ItsMeAgain" to server.
SERVER: Received client request: ItsMeAgain
SERVER: Sending "OK" to client
CLIENT: Got back response "OK" from server.
CLIENT: Enter command to send to server ... stop
CLIENT: Sending "stop" to server.
SERVER: Received client request: stop
SERVER: Sending "OK" to client
SERVER: Closing client connection.
SERVER: Shutting down.
CLIENT: Got back response "OK" from server.
CLIENT: Shutting down.
[5]+  Done                    ./server
student@COMPBase:~$
```

At this point, the server has also shut down.

There is more to learn about client/server communications and socket connections. Feel free to look up more information on your own. For example, we can add some code to the server that will display the IP address of the client as follows:

```
char *s = inet_ntoa(clientAddr.sin_addr);
printf("IP address: %s\n", s);
```

This will display the client's IP address, which in our example is 127.0.0.1.

Client Server Model - UDP

Let us now consider the UDP model for client/server communications.

The UDP server's socket is created in the same way as the TCP server, except that we use **IPPROTO_UDP** in place of **IPPROTO_TCP**:

```
serverSocket = socket(AF_INET, SOCK_STREAM, IPPROTO_UDP);
```

The server socket is then bound to its own IP address and port number in the same way by using the **bind()** function. There is no need to use the **listen()** function, since we are not setting up a one-to-one communication with anyone. We will simply be accepting whatever packets come in, regardless of who they are from.

Similar to TCP, the server should go into an infinite loop to accept incoming requests.

When using a UDP server, incoming information from a client socket will make use of what is known as a ***file descriptor***:

> *A **file descriptor** is an integer ID (i.e., handle) used to access a file or other input/output resource, such as a pipe or network socket.*

In order to receive an incoming packet, we need to use the **select()** function, which will allow us to be notified when an incoming packet is available, or if a "time out" has occurred if things are taking too long. It allows us to accept packets from more than one socket (i.e., multiple clients). For this reason, we cannot simply just call a ***read*** command for a particular socket, otherwise our code would lock-up waiting on only one socket channel.

The **select()** function has this format:

select(<*numDescriptors*>, <*readFDS*>, <*writeFDS*>, <*exceptFDS*>, <*timeout*>)

Here, <*numDescriptors*> is the number of file descriptors (i.e., potential clients) that we'd like to check for. The usual value is FD_SETSIZE … which is the maximum number possible.

The <*readFDS*> and <*writeFDS*> are the sets of file descriptors (i.e., sockets) that are ready for reading and writing, respectively. The <*exceptFDS*> are the file descriptors checked for exceptional conditions … we will set this to **NULL** in our examples. These are structures of type **fd_set**.

For the <*readFDS*> and <*writeFDS*>, we use the following macros to clear and set them for the socket. Here, we see that the given **socket** is added to the set readfds … meaning we would like to be able to read from this **socket**. →

```
int       socket;
fd_set    readfds;

FD_ZERO(&readfds);
FD_SET(socket, &readfds);
```

Regarding the <*timeout*>, this is a **struct** timeval type. If set to **NULL**, the **select()** function will block and wait indefinitely until a client packet comes in. It is the easiest option to use. Otherwise, we can set the <*timeout*> to **{0,0}** if we don't want to wait at all. We will not discus the timeout any further in this course.

The **select()** function will return **0** if a timeout occurred, **-1** if an error occurred … or a positive value otherwise. To read in the client request packet, we use the **recvfrom()** function which has this format:

recvfrom(<*socket*>, <*buffer*>, <*bufLen*>, <*flags*>, <*clientAddr* * >, <*clientAddrLength* * >)

The <*socket*> is the value returned from the **socket()** function. As with the TCP example, the <*buffer*> and <*bufLen*> work the same way. We will not discuss the <*flags*> here … but will set them to **0**. The <*clientAddr* * > is the address to a **struct** sockaddr as with the TCP example and the <*clientAddrLength* * > is the address of an **int** that holds the **sizeof**(<*clientAddr*>). The **recvfrom()** function returns the number of bytes received from the socket. We can do what we want with the buffer data at this point.

To send something back to the client, we use the **sendto()** function which has this format:

sendto(<*socket*>, <*buffer*>, <*bufLen*>, <*flags*>, <*clientAddr* * >, <*clientAddrLength*>)

The idea is the same … but the **clientAddrLength** is not a pointer now. We simply set up the buffer that we want to send and send it. Here is the pseudocode for setting up the server:

```
Open the socket
Bind the socket
while (true) {
  Select a socket request
  Receive the buffer from the client
  Process the request
  Send a response to the client
}
Close server socket
```

Here is the code for the server in its entirety:

Code from **udpServer.c**

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/socket.h>
#include <netinet/in.h>

#define  SERVER_PORT 6000

int main() {
  int                   serverSocket;
  struct sockaddr_in    serverAddr, clientAddr;
  int                   status, addrSize, bytesReceived;
  fd_set                readfds, writefds;
  char                  buffer[30];
  char                 *response = "OK";

  // Create the server socket
  serverSocket = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
  if (serverSocket < 0) {
    printf("*** SERVER ERROR: Could not open socket.\n");
    exit(-1);
  }

  // Setup the server address
  memset(&serverAddr, 0, sizeof(serverAddr)); // zeros the struct
  serverAddr.sin_family = AF_INET;
  serverAddr.sin_addr.s_addr = htonl(INADDR_ANY);
  serverAddr.sin_port = htons((unsigned short) SERVER_PORT);

  // Bind the server socket
  status = bind(serverSocket,(struct sockaddr *)&serverAddr, sizeof(serverAddr));
```

```
  if (status < 0) {
    printf("*** SERVER ERROR: Could not bind socket.\n");
    exit(-1);
  }

  // Wait for clients now
  while (1) {
    FD_ZERO(&readfds);
    FD_SET(serverSocket, &readfds);
    FD_ZERO(&writefds);
    FD_SET(serverSocket, &writefds);
    status = select(FD_SETSIZE, &readfds, &writefds, NULL, NULL);
    if (status == 0) {  // Timeout occurred, no client ready
    }
    else if (status < 0) {
      printf("*** SERVER ERROR: Could not select socket.\n");
      exit(-1);
    }
    else {
      addrSize = sizeof(clientAddr);
      bytesReceived = recvfrom(serverSocket, buffer, sizeof(buffer),
                               0, (struct sockaddr *) &clientAddr, &addrSize);
      if (bytesReceived > 0) {
        buffer[bytesReceived] = '\0';
        printf("SERVER: Received client request: %s\n", buffer);
      }

      // Respond with an "OK" message
      printf("SERVER: Sending \"%s\" to client\n", response);
      sendto(serverSocket, response, strlen(response), 0,
             (struct sockaddr *) &clientAddr, addrSize);

      // If the client said to stop, then I'll stop myself
      if (strcmp(buffer, "stop") == 0)
        break;
    }
  }
}
```

When **select**() exits, each of the file descriptor sets is modified to indicate which file descriptors actually changed status. So, when using **select**() within a loop, the sets must be reinitialized before each call to **select**().

Now what about the client ? The socket is set up in the same way. The **sendto()** and **recvfrom()** functions are also used, just as with the server. Here is the completed code:

Code from **udpClient.c**

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#define  SERVER_IP   "127.0.0.1"
#define  SERVER_PORT 6000


int main() {
  int                 clientSocket, addrSize, bytesReceived;
```

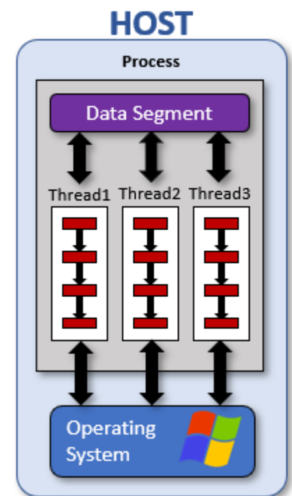
```
  struct sockaddr_in  serverAddr;
  char                inStr[80];     // stores user input from keyboard
  char                buffer[80];    // stores sent and received data

  // Create socket
  clientSocket = socket(PF_INET, SOCK_DGRAM, IPPROTO_UDP);
  if (clientSocket < 0) {
    printf("*** CLIENT ERROR: Could open socket.\n");
    exit(-1);
  }

  // Setup address
  memset(&serverAddr, 0, sizeof(serverAddr));
  serverAddr.sin_family = AF_INET;
  serverAddr.sin_addr.s_addr = inet_addr(SERVER_IP);
  serverAddr.sin_port = htons((unsigned short) SERVER_PORT);

  // Go into loop to commuincate with server now
  while (1) {
    addrSize = sizeof(serverAddr);

    // Get a command from the user
    printf("CLIENT: Enter command to send to server ... ");
    scanf("%s", inStr);

    // Send command string to server
    strcpy(buffer, inStr);
    printf("CLIENT: Sending \"%s\" to server.\n", buffer);
    sendto(clientSocket, buffer, strlen(buffer), 0,
           (struct sockaddr *) &serverAddr, addrSize);

    // Get response from server, should be "OK"
    bytesReceived = recvfrom(clientSocket, buffer, 80, 0,
                             (struct sockaddr *) &serverAddr, &addrSize);
    buffer[bytesReceived] = 0; // put a 0 at the end so we can display the string
    printf("CLIENT: Got back response \"%s\" from server.\n", buffer);

    if ((strcmp(inStr,"done") == 0) || (strcmp(inStr,"stop") == 0))
      break;
  }

  close(clientSocket);  // Don't forget to close the socket !
  printf("CLIENT: Shutting down.\n");
}
```

Assuming that the udpServer has been started, the output is as follows:

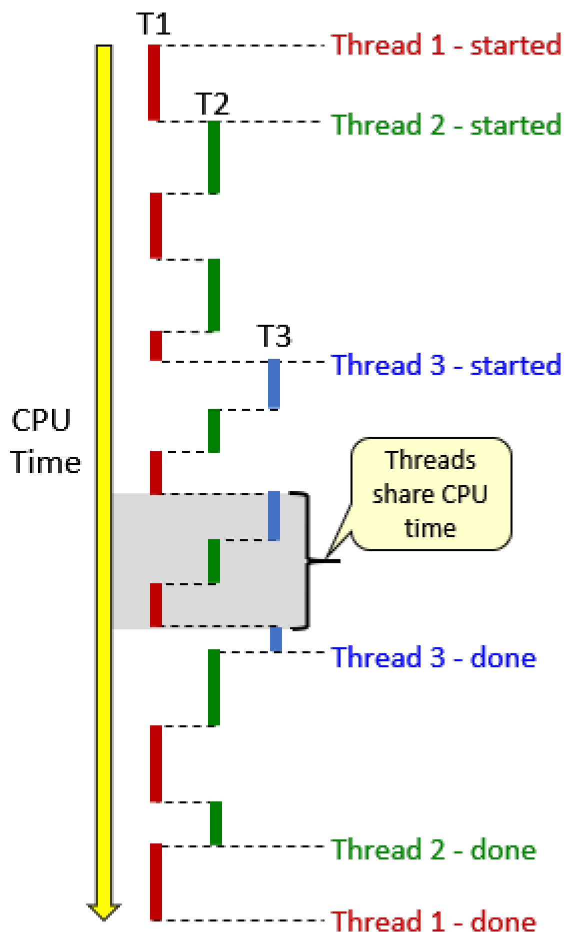
```
student@COMPBase:~$ ./udpClient
CLIENT: Enter command to send to server ... Hello
CLIENT: Sending "Hello" to server.
SERVER: Received client request: Hello
SERVER: Sending "OK" to client
CLIENT: Got back response "OK" from server.
CLIENT: Enter command to send to server ... Fun
CLIENT: Sending "Fun" to server.
SERVER: Received client request: Fun
SERVER: Sending "OK" to client
```

```
CLIENT: Got back response "OK" from server.
CLIENT: Enter command to send to server ... stop
CLIENT: Sending "stop" to server.
SERVER: Received client request: stop
SERVER: Sending "OK" to client
SERVER: Shutting down.
CLIENT: Got back response "OK" from server.
CLIENT: Shutting down.
[3]+  Done                    ./udpServer
student@COMPBase:~$
```

5.4 Threads



We have discussed, in detail, the C-language mechanisms that allow two processes to communicate on the same host or over a network, where the processes are running simultaneously. There are many issues that we have not discussed which pertain to distributed computing, as this course just provides an *introduction* to systems programming. Likely, you can perceive by now that the code for handling timing and resource sharing can get tricky and much more complicated as more and more processes are added to the software framework. A simpler way to manage separate tasks is to use threads:



*A **thread** is a sequence of programmed instructions that can be managed independently by the operating system*

Threads are similar to processes in that they "logically" run separate tasks simultaneously. They are used for smaller tasks, as oposed to larger ones. Multiple threads can be running within a single process. However, only one thread's instructions can actually be executed at a time by the CPU. The threads all share the CPU processing time, often in a round-robin fashion (i.e., everyone gets their turn). Since the threads each run separately on the CPU, this greatly simplifies the likelihood of race conditions and deadlocks occurring, although if we are not careful, we may still end up with poor code that causes these situations to occur.

Similar to the **fork()** command, which spawns new processes, threads can be created from the main program thread or from other created threads.

As far as the logic is concerned, the threads can be considered as all running "in parallel" and are scheduled automatically by the operating system kernel. Switching between threads is faster than switching between processes.

Each thread runs as a separate program. They have a unique ***thread context*** (i.e., resources) that includes:

- Thread ID – a unique ID.
- Function call stack – keeps track of function call ordering, parameters, and variables.
- Program counter – keeps track of program instruction that is currently executing.

One very nice feature of threads is that all threads belonging to the same process share:

- Address space
- Data segment (i.e., global variables and allocated heap memory)
- Code segment (i.e., program instructions)

That means, the value of a global variable at any point in time is the same across all threads and that any thread can access and modify it.

To create a thread, we us the **pthread_create()** function which is defined in the **<pthread.h>** header file. It takes these 4 parameters:

1. A pointer to a **pthread_t** variable, which stores an integer representing the handle (i.e., ID) of the newly-created thread (we pass a pointer so that the variable can be set by the function).

2. Some attributes that can de used by the thread (we will use **NULL** to indicate defaults).

3. A pointer to a *start function* that will be called to start the thread.

4. A single parameter that can be passed to the *start function*.

To stop/terminate a thread, **pthread_exit(void** *status**)** can be called, where **status** will end up being the return value of the thread. Alternatively, one thread can wait for the termination of another thread by using the **pthread_join(pthread_t** thread**, void** **status**)** function which specifies which **thread** to wait for and also allows a value to be returned in the **status** pointer, although we will use **NULL** in our examples.

Consider this simple example that creates 3 threads and allows them to run for 4, 8 and 2 seconds, respectively. The main program keeps running and waits for thread 1 to complete, then for thread 2 to complete and then for thread 3 to complete (which had already completed).

Code from **thread.c**

```
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>

void *printMsg(void *);
int  times[] = {4, 8, 2};  // # of seconds for each thread to run


int main() {
  pthread_t    t1, t2, t3;

  pthread_create(&t1, NULL, printMsg, "1");
  pthread_create(&t2, NULL, printMsg, "2");
  pthread_create(&t3, NULL, printMsg, "3");

  printf("\nThreads all created. \nWaiting for Thread 1 now ...\n");
  pthread_join(t1, NULL);
  printf("\nThread 1 is back. \nWaiting for Thread 2 now ...\n");
  pthread_join(t2, NULL);
  printf("Thread 2 is back. \nWaiting for Thread 3 now ...\n");
  pthread_join(t3, NULL);
  printf("Thread 3 is back. \nTime to quit.\n");
}

// Function called at the start of each thread
void *printMsg(void *str) {
  char   threadNum = ((char *)str)[0] - '0';
  for (int i=0; i<times[threadNum-1]; i++) {
    for (int j=0; j<threadNum; j++) // indent a bit for visual clarity
      printf("  ");
    printf("Thread %d \n", threadNum);
    sleep(1);
  }
}
```

To compile/link this program we have to include the **pthread** library, so we add **-lpthread** to the **gcc** command line as follows:

```
student@COMPBase:~$
gcc -o thread thread.c -lpthread
student@COMPBase:~$
```

The expected output is show here on the right →

```
Threads all created.
Waiting for Thread 1 now ...
      Thread 3
    Thread 2
  Thread 1
      Thread 3
    Thread 2
  Thread 1
    Thread 2
  Thread 1
    Thread 2
  Thread 1
    Thread 2

Thread 1 is back.
Waiting for Thread 2 now ...
    Thread 2
    Thread 2
    Thread 2
Thread 2 is back.
Waiting for Thread 3 now ...
Thread 3 is back.
Time to quit.
```

Make sure that you understand the output.

Notice how thread **3** stopped fairly quickly … just after **2** seconds … and thread **1** just after **4** seconds.

Of course, it can be a problem if two threads attempt to modify the same data at the same time. The results will be unpredictable because we don't know which thread will modify it first as it depends when the CPU decides to give each thread its share of CPU time. Therefore, if one thread is in the middle of updating a variable and another comes along and tries to update the variable as well, the update may not work as desired.

Consider a single integer variable, called **count** with an initial value of **0**. Assume that two threads attempt to update the variable by adding one to it as follows: **count = count + 1**. If both threads run one after the other, then there is no issue since each will increase the count by one and count will have the value of **2**. However, let's break down the simple line of code. In order to increase the count variable, the following must occur:

```
1. Read the count variable.
2. Add 1 to its value.
3. Store the new value back into the count variable.
```

Since there are three stages to this simple operation, it is not ***atomic*** (i.e., smallest level … unable to be split any further). So, there is potential for corruption when multiple threads/processes are modifying the variable. Consider what happens if one thread performs step 1 … reading a value of **0** for the **count** … and then a context switch happens (i.e., the thread pauses and the other thread is given CPU control). What will happen ? The second thread will perform step 1 and read a value of **0** for the **count** as well. Then suppose the second thread completes its steps 2 and 3, thereby setting the **count** variable to **1**. Now suppose control goes back to the first thread, which will continue on to step 2. It has already read the value of the **count** variable (from before the context switch) which had a value of **0**. So it will perform steps 2 and 3 to increase that value to **1** and then store the value of **1** into the **count** variable. So, the result is that **count** has the value of **1** despite the fact that both had increased the value by **1**! Therefore, the value is 1 instead of 2, which is wrong. Of course, sometimes, the first thread will complete all three steps before a context switch. So, it is possible that the **count** will be updated to **2** correctly. But this really is very unpredictable, as there is no certainty as to when the context switch will occur.

Here is a diagram showing two threads, each attempting to increase a shared **count** variable by 1 for 5 iterations. In this instance, the context switch happens nicely (i.e., ideally) after each three-line chunk of code. You can see that the **count** variable is updated properly the whole time such that it reaches the correct count of **10**.

CPU time

Thread 1	count	Thread 2
Read count (= 0)	0	
Add 1 to its value	0	
Store back into count	1	
	1	Read count (= 1)
	1	Add 1 to its value
	2	Store back into count
Read count (= 2)	2	
Add 1 to its value	2	
Store back into count	3	
	3	Read count (= 3)
	3	Add 1 to its value
	4	Store back into count
Read count (= 4)	4	
Add 1 to its value	4	
Store back into count	5	
	5	Read count (= 5)
	5	Add 1 to its value
	6	Store back into count
Read count (= 6)	6	
Add 1 to its value	6	
Store back into count	7	
	7	Read count (= 7)
	7	Add 1 to its value
	8	Store back into count
Read count(= 8)	8	
Add 1 to its value	8	
Store back into count	9	
	9	Read count (= 9)
	9	Add 1 to its value
	10	Store back into count

Now here is the same example with the context switching happening after every 4 lines of code. You will notice that the **count** variable is not properly updated each time so that the **count** is not 10 at completion. This is a more realistic example. However, the context switching does not happen at nice clean intervals like this. It could vary each time. Therefore, it is impossible to predict the final value for **count**.

CPU time

Thread 1	count	Thread 2
Read count (= 0)	0	
Add 1 to its value	0	
Store back into count	1	
Read count (= 1)	1	
	1	Read count (= 1)
	1	Add 1 to its value
	2	Store back into count
	2	Read count (= 2)
Add 1 to its value	2	
Store back into count	2	
Read count (= 2)	2	
Add 1 to its value	2	
	2	Add 1 to its value
	3	Store back into count
	3	Read count (= 3)
	3	Add 1 to its value
Store back into count	3	
Read count (= 3)	3	
Add 1 to its value	3	
Store back into count	4	
	4	Store back into count
	4	Read count (= 4)
	4	Add 1 to its value
	5	Store back into count
Read count(= 5)	5	
Add 1 to its value	5	
Store back into count	6	
	6	Read count (= 6)
	6	Add 1 to its value
	7	Store back into count

Here is some code that verifies this problem:

Code from **badThread.c**

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

void *threadFunc(void *);

int  count = 0;

int main() {
  int           numInc = 100000000; // count to 100 million
  pthread_t     t1, t2;

  pthread_create(&t1, NULL, threadFunc, &numInc);
  pthread_create(&t2, NULL, threadFunc, &numInc);
  pthread_join(t1, NULL);
  pthread_join(t2, NULL);

  if (count != (2 * numInc))
    printf("Error: Count is %d instead of 200,000,000.\n", count);
  else
    printf("Count is %d, which is correct.\n", count);

  return(0);
}

// Function to increase count variable by amount specified by arg
void *threadFunc(void *arg) {
  int inc = *((int *)arg);
  for (int i=0; i<inc; i++)
    count++;
  return(0);
}
```

Notice that when we run it, we get a different result each time … proving that the result is unpredictable:

```
student@COMPBase:~$ gcc -o badThread badThread.c -lpthread
student@COMPBase:~$ ./badThread
Error: Count is 197308945 instead of 200,000,000.
student@COMPBase:~$ ./badThread
Error: Count is 190625336 instead of 200,000,000.
student@COMPBase:~$ ./badThread
Error: Count is 196187270 instead of 200,000,000.
student@COMPBase:~$
```

So, how do we fix the problem ?

A solution is to protect all shared data. We can also make sure that changes are made at the *atomic* level. The two mechanisms that we use to protect shared data are (1) the ***semaphore*** and (2) the ***mutex*** … which were both described earlier in this chapter.

The semaphore acts as a locking mechanism to prevent other threads from accessing or modifying a resource (e.g., variable) at the same time. While the resource is locked, other threads are waiting. Once unlocked, there is no guarantee as to which thread gets to have access next. In our example, we need to use a semaphore to coordinate the sharing of the **count** variable. What we need to do is to "lock" the usage of the **count** variable by one thread until the read+increase+write operations have all completed so that there is no interference in between.

The semaphore itself is actually a counter as well. We typically set it to some non-zero initial value. A thread can have access to the shared resource as long as the value of the semaphore is greater than zero. A ***mutex*** is a binary semaphore, with a value of 0 or 1. Only one thread can access it at a time. In our example, we will use a simple mutex semaphore which will have a value of **1** (indicating that the resource is unlocked and available) or **0** (indicating that the resource is locked and being used).

A semaphore is defined as a **sem_t** type and we need to include the **<semaphore.h>** header in our code in order to use it. The first function that we need to call is **sem_init()** which allows us to initialize the semaphore:

```
sem_t   semaphore;

sem_init(&semaphore, 0, 1);
```

In the above code, the semaphore is initially given a value of 1 as the third parameter to the function. The second parameter has a value of **0**, indicating that the semaphore will just be used between threads, as opposed to between multiple processes. If the function returns a negative value, then something went wrong (e.g., the value exceeds **SEM_VALUE_MAX**, the limit on the number of semaphores has been reached, process does not have privileges, etc.).

When a thread is ready to use the shared resource (e.g., the **count++** line of code), then it must "surround that code" with code beforehand to wait on the semaphore and code afterwards to release the semaphore.

The `sem_wait(&semaphore)` function is used to wait on the semaphore. That is, when we call it, our code waits there until it is this thread's turn to use the shared resource. The function returns **-1** if the wait fails (e.g., semaphore already locked, deadlock has been detected, a signal interrupted, or the parameter is invalid) … otherwise **0** is returned. This function decrements the value of the semaphore. If the value of the semaphore is zero, it waits until it is non-zero.

The `sem_post(&semaphore)` function is used to release the lock on a semaphore so that others can use the resource. It fails only if the parameter is invalid, in which case **-1** is returned … otherwise **0** is returned. This function increments the semaphore's value.

Here is the updated code that will work properly to increase the **count** via the two threads:

Code from **semaphore.c**

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>

void *threadFunc(void *);

volatile int count = 0;
sem_t   mutex;


int main() {
  int          numInc = 100000000; // count to 100 million
  pthread_t    t1, t2;

  if (sem_init(&mutex, 0, 1) < 0) {
    printf("Error: on semaphore init.\n");
    exit(1);
  }

  pthread_create(&t1, NULL, threadFunc, &numInc);
  pthread_create(&t2, NULL, threadFunc, &numInc);
  pthread_join(t1, NULL);
  pthread_join(t2, NULL);

  if (count != (2 * numInc))
    printf("Error: Count is %d instead of 200,000,000.\n", count);
  else
    printf("Count is %d, which is correct.\n", count);
}


// Function to increase count variable by amount specified by arg
void *threadFunc(void *arg)  {
  int inc = *((int *)arg);

  for (int i=0; i<inc; i++) {
    if (sem_wait(&mutex) < 0) {
      printf("Error: on semaphore wait.\n");
      exit(1);
    }
    count++;
    if (sem_post(&mutex) < 0) {
      printf("Error: on semaphore post.\n");
      exit(1);
    }
  }
  return(0);
}
```

Notice how the **sem_wait()** and **sem_post()** functions wrap around the **count++** statement. This is how we lock use of that shared resource.

Notice also that the keyword **volatile** is used in the declaration of the **count** variable. This is a special keyword in C that indicates that the value of the **count** variable could change unexpectedly. The **volatile** keyword should ALWAYS be used when global variables are accessed by multiple tasks in a multi-threaded application. The reason is that the compiler needs to know that it will be accessed/modified by multiple threads in order to prevent the compiler optimization from introducing unexpected behavior.

What is the result when we run the code ? It runs slower (because there is a lot of locking/waiting going on by the threads. However, the code produces the correct result:

```
student@COMPBase:~$ gcc -o semaphore semaphore.c -lpthread
student@COMPBase:~$ ./semaphore
Count is 200000000, which is correct.
student@COMPBase:~$
```

A Client/Server Example

Just for fun … let us see if we can create a client/server example similar to what we did before … but by using *threads* instead of *processes*. To do this, we will create a server thread and also three client threads. So our **main** function will look as follows:

```
void *runClient(void *num);     // We will write this
void *runServer(void *notUsed); // We will write this

int main() {
  pthread_t     serverThread;
  pthread_t     client[3];

  // Start the server
  pthread_create(&serverThread, NULL, runServer, NULL);

  // Start up 3 client threads
  pthread_create(&client[0], NULL, runClient, "1");
  pthread_create(&client[1], NULL, runClient, "2");
  pthread_create(&client[2], NULL, runClient, "3");

  // Wait for the server to come back
  pthread_join(serverThread, NULL);
}
```

Notice that it will start a **runServer** thread and then each **runClient** thread with it's own number as a parameter passed in. It looks very similar to our previous code. The join function at the end will make sure that the **main** function does not complete until the **runServer** has completed and rejoined the **main** thread.

So then, what do the server and client do ? Well, in our previous example, we had the client send commands based on what the user entered through the keyboard. This time, we will have the clients send 4 fixed command strings and then quit. We will set it up so that the clients connect to the server, exchange data, and then rest for a bit (i.e., sleep for some random amount of time) in order to let other clients get in on the action.

But since we are not using sockets or official connections … how does the server communicate with the clients? Well, because they are all threads running in the same process and share the same address space … we can have them share the same data buffers. We will set up one buffer for sending and another for receiving:

```
#define  BUFFER_SIZE    30

char    requestBuffer[BUFFER_SIZE];   // data from client to server
char    responseBuffer[BUFFER_SIZE];  // data from server to client
```

The clients will fill in the **requestBuffer** and the server will read it. The server will then fill in the **responseBuffer** and the client will read that one. We will likely want to have a way of telling the server that we have finished filling in the **requestBuffer**. A simple way to do this is to make a binary flag. We can use another one for the server to inform the client that the **responseBuffer** is ready. We will add these for that purpose:

```
char  requestReady = 0;  // flag to tell server that request is ready
char  responseReady = 0; // flag to tell client that response is ready
```

Now, we are ready to write the server. It should run forever, or at least until told to STOP. Notice how logical the code below is:

```
void *runServer(void *notUsed) {
  while (1) {
    // Wait for an incoming client request
    while (requestReady == 0); // wait for a request

    // Get/Receive the message from the client into the char buffer
    requestReady = 0; // reset for next time
    printf("SERVER: Received client request: %s\n", requestBuffer);

    // Respond with an "OK" message
    responseBuffer[0] = requestBuffer[0];
    strcpy(responseBuffer+1, "ok\0");
    printf("SERVER: Sending \"%s\" to client\n", responseBuffer);
    responseReady = 1;

    // Quit if someone sent a STOP command
    if (strcmp(&requestBuffer[1], "STOP") == 0)
      break;
  }
  printf("SERVER: Shutting down.\n");
}
```

First of all, the code loops (i.e., server thread runs) until a STOP message has been received. Inside the loop, we first wait for a client request by examining the **requestReady** flag. As soon as it becomes **1**, we know that the client has set up his/her message in the **requestBuffer**. We reset it back to **0** right away (so that we don't forget) for the next request. Then the request is printed and a response is set up. We will assume that the first character of the client request is the single-digit client id (for our example only). We will copy this into our response so that client 1 gets a response of "1ok", client 2 gets "2ok" etc… Once the response is ready to go, we set the **responseReady** flag to tell the client.

So now just the **runClient** remains to be written. We need to make sure that only one client is using the buffers at a time. So what we will do, is set up a simple mutex semaphore that allows only one client to communicate with the server at a time:

```
sem_t   serverBusyIndicator;
```

This, of course, needs to be initialized in our **main** function:

```
sem_init(&serverBusyIndicator, 0, 1);
```

Now we can write the **runClient** function. Let us write it without the semaphore first. We will set the following global variable up to store the commands to be sent:

```
#define  NUM_COMMANDS   4

char *clientCommands[NUM_COMMANDS] = {"Hello ","Funny","Stuff","STOP"};
```

Then we will set the client up to send all 4 commands in a loop:

```
void *runClient(void *num) {
  int  command = 0;

  while (command < NUM_COMMANDS) {
    // Send command string to server
    requestBuffer[0] = ((char *)num)[0];
    strcpy(requestBuffer+1, clientCommands[command++]);
    printf("CLIENT: Sending \"%s\" to server.\n", requestBuffer);
    requestReady = 1;

    // Get response from server, should be "OK"
    while (responseReady == 0); // wait for a response

    printf("CLIENT: Got back response \"%s\" from server.\n\n",
                                                    responseBuffer);
    responseReady = 0;

    // Sleep from 0 to 4 seconds randomly
    sleep((int)(rand()/(double)RAND_MAX*5));
  }
}
```

The code is straight forward, isn't it? Notice that we are using the first character in the incoming function parameter and appending it to the start of the **requestBuffer** so that the server knows which client this is. The buffers and flags are used the same way as with the server. After a send and receive is done, the client waits for a random number of seconds before sending the next command.

Now what do we do with the semaphore? Well, the client should only try to access the requestBuffer when no other clients are using it … when the semaphore is free. So we need to wrap this code up using semaphore wait and post calls as follows:

```
void *runClient(void *num) {
  int  command = 0;

  while (command < NUM_COMMANDS) {
    // Wait for the server
    sem_wait(&serverBusyIndicator);

    // Send command string to server
    requestBuffer[0] = ((char *)num)[0];
    strcpy(requestBuffer+1, clientCommands[command++]);
    printf("CLIENT: Sending \"%s\" to server.\n", requestBuffer);
    requestReady = 1;

    // Get response from server, should be "OK"
    while (responseReady == 0); // wait for a response

    printf("CLIENT: Got back response \"%s\" from server.\n\n",
                                                   responseBuffer);
    responseReady = 0;

    // Tell the server we are done
    sem_post(&serverBusyIndicator);

    // Sleep from 0 to 4 seconds randomly
    sleep((int)(rand()/(double)RAND_MAX*5));
  }
}
```

That is it! We are done. Here is the completed code:

Code from **csThreadExample.c**

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <pthread.h>
#include <time.h>
#include <semaphore.h>

#define NUM_COMMANDS   4
#define BUFFER_SIZE    30

// This will be used to ensure that only one client communicates with
// the server at a time, so that the variables below are used properly
sem_t   serverBusyIndicator;


// These are the variables used to pass data between threads
char    requestBuffer[BUFFER_SIZE];   // data from client to server
char    responseBuffer[BUFFER_SIZE];  // data from server to client
char    requestReady = 0;    // flag to tell server that request is ready
char    responseReady = 0;   // flag to tell client that response is ready
```

```
// These are the commands sent from each client
char   *clientCommands[NUM_COMMANDS] = {"Hello ", "Funny", "Stuff", "STOP"};

// Set up the client so that it sends 4 commands
void *runClient(void *num) {
  int   command = 0;

  // Go into infinite loop to communicate with server now
  while (command < NUM_COMMANDS) {
    // Wait for the server
    sem_wait(&serverBusyIndicator);

    // Send command string to server
    requestBuffer[0] = ((char *)num)[0];
    strcpy(requestBuffer+1, clientCommands[command++]);
    printf("CLIENT: Sending \"%s\" to server.\n", requestBuffer);
    requestReady = 1;

    // Get response from server, should be "OK"
    while (responseReady == 0); // wait for a response

    printf("CLIENT: Got back response \"%s\" from server.\n\n", responseBuffer);
    responseReady = 0;

    // Tell the server we are done
    sem_post(&serverBusyIndicator);

    // Sleep from 0 to 4 seconds randomly
    sleep((int)(rand()/(double)RAND_MAX*5));
  }
}

void *runServer(void *notUsed) {
  // repeat forever
  while (1) {
    // Wait for an incoming client request
    while (requestReady == 0); // wait for a request

    // Get/Receive the message from the client into the char buffer
    requestReady = 0; // reset for next time
    printf("SERVER: Received client request: %s\n", requestBuffer);

    // Respond with an "OK" message
    responseBuffer[0] = requestBuffer[0];
    strcpy(responseBuffer+1, "OK\0");
    printf("SERVER: Sending \"%s\" to client\n", responseBuffer);
    responseReady = 1;

    // Quit if someone sent a STOP command
    if (strcmp(&requestBuffer[1], "STOP") == 0)
      break;
  }

  printf("SERVER: Shutting down.\n");
}
```

```
// This main function starts a server and then three clients.
int main() {
  pthread_t     serverThread;
  pthread_t     client[3];

  srand(time(NULL));

  // Initialize semaphore
  sem_init(&serverBusyIndicator, 0, 1);

  // Start the server
  pthread_create(&serverThread, NULL, runServer, NULL);

  // Start up 3 client threads
  pthread_create(&client[0], NULL, runClient, "1");
  pthread_create(&client[1], NULL, runClient, "2");
  pthread_create(&client[2], NULL, runClient, "3");

  // Wait for the server to come back
  pthread_join(serverThread, NULL);
}
```

Remember to include the **-lpthread** library when compiling.

The output (although it will vary due to the randomness) is as shown here on the right →

Notice that in this particular run, the first client did not get to send all his/her requests because client 2 stopped the server.

```
CLIENT: Sending "3Hello " to server.
SERVER: Received client request: 3Hello
SERVER: Sending "3OK" to client
CLIENT: Got back response "3OK" from server.

CLIENT: Sending "2Hello " to server.
SERVER: Received client request: 2Hello
SERVER: Sending "2OK" to client
CLIENT: Got back response "2OK" from server.

CLIENT: Sending "1Hello " to server.
SERVER: Received client request: 1Hello
SERVER: Sending "1OK" to client
CLIENT: Got back response "1OK" from server.

CLIENT: Sending "2Funny" to server.
SERVER: Received client request: 2Funny
SERVER: Sending "2OK" to client
CLIENT: Got back response "2OK" from server.

CLIENT: Sending "2Stuff" to server.
SERVER: Received client request: 2Stuff
SERVER: Sending "2OK" to client
CLIENT: Got back response "2OK" from server.

CLIENT: Sending "3Funny" to server.
SERVER: Received client request: 3Funny
SERVER: Sending "3OK" to client
CLIENT: Got back response "3OK" from server.

CLIENT: Sending "3Stuff" to server.
SERVER: Received client request: 3Stuff
SERVER: Sending "3OK" to client
CLIENT: Got back response "3OK" from server.

CLIENT: Sending "1Funny" to server.
SERVER: Received client request: 1Funny
SERVER: Sending "1OK" to client
CLIENT: Got back response "1OK" from server.

CLIENT: Sending "2STOP" to server.
SERVER: Received client request: 2STOP
SERVER: Sending "2OK" to client
SERVER: Shutting down.
```