PIPE:

We use the term **pipe** when we connect a data flow from one process to another. Generally we attach, or pipe, the output of one process to the input of another. A pipe is a method used to pass information from one program process to another.

Unlike other types of inter process communication, a pipe only offers one-way communication by passing a parameter or output from one process to another. The information that is passed through the pipe is held by the system until it can be read by the receiving process. Pipes are primarily used in programming on UNIX systems.

Pipes are the oldest form of UNIX System IPC and are provided by all UNIX systems. Pipes have two limitations.

1. Historically, they have been half duplex (i.e., data flows in only one direction). Some systems now provide full-duplex pipes, but for maximum portability, we should never assume that this is the case.

2. Pipes can be used only between processes that have a common ancestor. Normally, a pipe is created by a process, that process calls fork, and the pipe is used between the parent and the child.

Despite these limitations, half-duplex pipes are still the most commonly used form of IPC. Every time you type a sequence of commands in a pipeline for the shell to execute, the shell creates a separate process for each command and links the standard output of one to the standard input of the next using a pipe.

PROCESS PIPES:

Perhaps the simplest way of passing data between two programs is with the **popen** and **pclose** functions. These have the prototypes:

```
#include <stdio.h>
FILE *popen(const char *command, const char *open_mode);
int pclose(FILE *stream_to_close);
```

popen:

The popen function allows a program to invoke another program as a new process and either pass data to or receive data from it.

The command string is the name of the program to run, together with any parameters. open_mode must be either "r" or "w".

If the open_mode is "r", output from the invoked program is made available to the invoking program and can be read from the file stream FILE * returned by popen, using the usual stdio library functions for reading (for example, fread).

However, if open_mode is w, the program can send data to the invoked command with calls to fwrite. The invoked program can then read the data on its standard input. Normally, the program being invoked won't be aware that it's reading data from another process; it simply reads its standard input stream and acts on it.

Each call to popen must specify either "r" or "w"; no other option is supported in a standard implementation of popen. This means that we can't invoke another program and both read and write to it. On failure, popen returns a null pointer. If you want bi−directional communication using pipes, then the normal solution is to use two pipes, one for data flow in each direction.

Pclose:

When the process started with popen has finished, we can close the file stream associated with it using pclose. The pclose call will only return once the process started with popen finishes. If it's still running when pclose is called, the pclose call will wait for the process to finish.

The pclose call normally returns the exit code of the process whose file stream it is closing. If the invoking process has executed a wait statement before calling pclose, the exit status will be lost and pclose will return −1, with errno set to ECHILD.

THE PIPE CALL:

We've seen the high−level popen function, but we'll now move on to look at the lower−level pipe function.

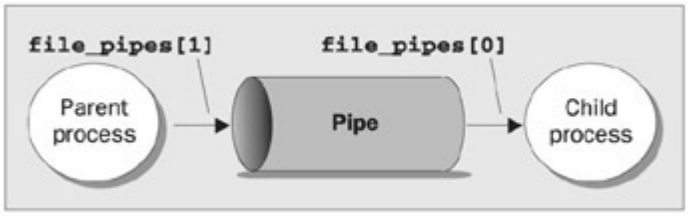
This provides a means of passing data between two programs, without the overhead of invoking a shell to interpret the requested command. It also gives us more control over the reading and writing of data.

The pipe function has the prototype:

```
#include <unistd.h>
int pipe(int file_descriptor[2]);
```

pipe is passed (a pointer to) an array of two integer file descriptors. It fills the array with two new file descriptors and returns a zero. On failure, it returns −1 and sets errno to indicate the reason for failure. Errors defined in the Linux man pages are:

EMFILE	Too many file descriptors are in use by the process.
ENFILE	The system file table is full.
EFAULT	The file descriptor is not valid.



The two file descriptors returned are connected in a special way. Any data written to file_descriptor[1] can be read back from file_descriptor[0]. The data is processed in a **first in, first out** basis, usually abbreviated to **FIFO**. This means that if you write the bytes 1, 2, 3 to file_descriptor[1], reading from file_descriptor[0] will produce 1, 2, 3. This is different from a stack, which operates on a **last in, first out** basis, usually abbreviated to **LIFO**.

Important It's important to realize that these are file descriptors, not file streams, so we must use the lower−level read and write calls to access the data, rather than fread and fwrite.

Here's a program, **pipe1.c** that uses pipe to create a pipe.

Type in the following code; Note the file_pipes pointer, which is passed to the pipe function as a parameter.


```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
int main()
{
int data_processed;
int file_pipes[2];
const char some_data[] = "123";
char buffer[BUFSIZ + 1];
memset(buffer, '\0', sizeof(buffer));
if (pipe(file_pipes) == 0) {
data_processed = write(file_pipes[1], some_data, strlen(some_data));
printf("Wrote %d bytes\n", data_processed);
data_processed = read(file_pipes[0], buffer, BUFSIZ);
printf("Read %d bytes: %s\n", data_processed, buffer);
exit(EXIT_SUCCESS);   }
exit(EXIT_FAILURE);
}
```

When we run this program, the output is:

$ pipe1

Wrote 3 bytes

Read 3 bytes: 123

The real advantage of pipes comes when you wish to pass data between two processes. As we saw in the last chapter, when a program creates a new process using the fork call, file descriptors that were previously open remain open. By creating a pipe in the original process and then forking to create a new process, we can pass data from one process to the other down the pipe.

PIPES ACROSS A FORK:

This is **pipe2.c**. It starts rather like the first example, up until we make the call to fork.

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
int main()
{
int data_processed;
int file_pipes[2];
const char some_data[] = "123";
char buffer[BUFSIZ + 1];
pid_t fork_result;
memset(buffer, '\0', sizeof(buffer));
if (pipe(file_pipes) == 0) {
fork_result = fork();
if (fork_result == −1) {
fprintf(stderr, "Fork failure");
exit(EXIT_FAILURE);
}
```

1. We've made sure the fork worked, so if fork_result equals zero, we're in the child process:

```
if (fork_result == 0) {
data_processed = read(file_pipes[0], buffer, BUFSIZ);
printf("Read %d bytes: %s\n", data_processed, buffer);
exit(EXIT_SUCCESS);
```

```
}
2. Otherwise, we must be the parent process:
else {
data_processed = write(file_pipes[1], some_data,
strlen(some_data));
printf("Wrote %d bytes\n", data_processed);
}
}
exit(EXIT_SUCCESS);
}
```

When we run this program, the output is, as before:

$ **pipe2**

Wrote 3 bytes

Read 3 bytes: 123

PARENT AND CHILD PROCESSES:

The next logical step in our investigation of the pipe call is to allow the child process to be a different program from its parent, rather than just a different process. We do this using the exec call. In our previous example, this wasn't a problem, since the child had access to its copy of the file_pipes data.

After an exec call, this will no longer be the case, as the old process has been replaced by the new child process. We can get round this by passing the file descriptor (which is, after all, just a number) as a parameter to the execed program.

To show how this works, we need two programs. The first is the 'data producer'. It creates the pipe and then invokes the child, the 'data consumer'.

PIPES AND EXEC:

For the first program, we adapt pipe2.c to **pipe3.c**. The lines that we've changed are shown shaded:

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
int main()
```

```
{
int data_processed;
int file_pipes[2];
const char some_data[] = "123";
char buffer[BUFSIZ + 1];
pid_t fork_result;
memset(buffer, '\0', sizeof(buffer));
if (pipe(file_pipes) == 0) {
fork_result = fork();
if (fork_result == (pid_t)−1) {
fprintf(stderr, "Fork failure");
exit(EXIT_FAILURE);
}
if (fork_result == 0) {
sprintf(buffer, "%d", file_pipes[0]);
(void)execl("pipe4", "pipe4", buffer, (char *)0);
exit(EXIT_FAILURE);
}
else {
data_processed = write(file_pipes[1], some_data,
strlen(some_data));
printf("%d − wrote %d bytes\n", getpid(), data_processed);
}
}
exit(EXIT_SUCCESS);
}
```

2. The 'consumer' program, pipe4.c that reads the data is much simpler:

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
int main(int argc, char *argv[])
{
int data_processed;
char buffer[BUFSIZ + 1];
int file_descriptor;
memset(buffer, '\0', sizeof(buffer));
sscanf(argv[1], "%d", &file_descriptor);
data_processed = read(file_descriptor, buffer, BUFSIZ);
```

```
printf("%d – read %d bytes: %s\n", getpid(), data_processed, buffer);
exit(EXIT_SUCCESS);
}
```

Remembering that pipe3 invokes the pipe4 program for us, when we run pipe3, we get the following output:

$ **pipe3**

980 – wrote 3 bytes

981 – read 3 bytes: 123

The pipe3 program starts like the previous example, using the pipe call to create a pipe and then using the fork call to create a new process. It then uses sprintf to store the 'read' file descriptor number of the pipe in a buffer that will form an argument of pipe4.

A call to execl is used to invoke the pipe4 program. The arguments to execl are:

The program to invoke.

- argv[0], which takes the program name.
- argv[1], which contains the file descriptor number we want the program to read from.
- (char *)0, which terminates the parameters.

The pipe4 program extracts the file descriptor number from the argument string and then reads from that file descriptor to obtain the data.

NAMED PIPES: FIFOs

So far, we have only been able to pass data between programs that are related, i.e. programs that have been started from a common ancestor process. Often, this isn't very convenient, as we would like unrelated processes to be able to exchange data.

We do this with **FIFOs**, often referred to as **named pipes**. A named pipe is a special type of file (remember everything in UNIX is a file!) that exists as a name in the file system, but behaves like the unnamed pipes that we've met already.

We can create named pipes from the command line and from within a program. Historically, the command line program for creating them was mknod:

$ **mknod filename p**

However, the mknod command is not available on all UNIX systems. The preferred command line method is to use:

$ **mkfifo filename**

Some older versions of UNIX only had the mknod *command.* From inside a program, we can use two different calls. These are:

#include <sys/types.h>
#include <sys/stat.h>
int mkfifo(const char *filename, mode_t mode);
int mknod(const char *filename, mode_t mode | S_IFIFO, (dev_t) 0);

Like the mknod command, you can use the mknod function for making many special types of file. Using a dev_t value of 0 and ORing the file access mode with S_IFIFO is the only portable use of this function which creates a named pipe. (**Note:** dev_t → ID of device containing file, S_IFIFO → FIFO Special File Type)

EXAMPLE CREATING A NAMED PIPE:

For fifo1.c, just type in the following code:

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
int main()
{
int res = mkfifo("/tmp/my_fifo", 0777);
if (res == 0) printf("FIFO created\n");
exit(EXIT_SUCCESS);
}
```

$ **./fifo1**
FIFO created

We can look for the pipe with:

$ **ls –lF /tmp/my_fifo**
prwxr−xr−x 1 rick users 0 Dec 10 14:55 /tmp/my_fifo|

Notice that the first character is a p, indicating a pipe. The | symbol at the end is added by the ls command's −F option and also indicates a pipe.

HOW IT WORKS:

The program uses the mkfifo function to create a special file. Although you ask for a mode of 0777, this is altered by the user mask (umask) setting (in this case 022), just as in normal file creation, so the resulting file has mode 755. If your umask is set differently, for example to 0002, you will see different permissions on the created file.

You can remove the FIFO just like a conventional file by using the rm command, or from within a program by using the unlink system call.

Accessing a FIFO:

One very useful feature of named pipes is that, because they appear in the file system, you can use them in commands where you would normally use a filename. Before you do more programming using the FIFO file you created, let's investigate the behavior of the FIFO file using normal file commands.

Accessing a FIFO File:

1. First, try reading the (empty) FIFO: $ **cat < /tmp/my_fifo**

2. Now try writing to the FIFO. You will have to use a different terminal because the first command will now be hanging, waiting for some data to appear in the FIFO.

$ **echo "Hello World" > /tmp/my_fifo**

You will see the output appear from the cat command. If you don't send any data down the FIFO, the cat command will hang until you interrupt it, conventionally with Ctrl+C.

3. You can do both at once by putting the first command in the background:

$ **cat < /tmp/my_fifo &**
[1] 1316
$ **echo "Hello World" > /tmp/my_fifo**
Hello World
[1]+ Done cat </tmp/my_fifo
$

HOW IT WORKS:

Because there was no data in the FIFO, the cat and echo programs both block, waiting for some data to arrive and some other process to read the data, respectively. Looking at the third stage, the cat process is initially blocked in the background. When echo makes some data available, the cat command reads the data and prints it to the standard output.

Notice that the cat program then exits without waiting for more data. It doesn't block because the pipe will have been closed when the second command putting data in the FIFO completed, so calls to read in the cat program will return 0 bytes, indicating the end of file.

Now that you've seen how the FIFO behaves when you access it using command-line programs, let's look in more detail at the program interface, which allows you more control over how reads and writes behave when you're accessing a FIFO.

Unlike a pipe created with the pipe *call, a FIFO exists as a named file, not as an open file descriptor, and it must be opened before it can be read from or written to. You open and close a FIFO using the same* open *and* close *functions that you saw used earlier for files, with some additional functionality. The* open *call is passed the path name of the FIFO, rather than that of a regular file.*

OPENING A FIFO WITH OPEN:

The main restriction on opening FIFOs is that a program may not open a FIFO for reading and writing with the mode O_RDWR. If a program violates this restriction, the result is undefined. This is quite a sensible restriction because, normally, you use a FIFO only for passing data in a single direction, so there is no need for an O_RDWR mode. A process would read its own output back from a pipe if it were opened read/write.

If you do want to pass data in both directions between programs, it's much better to use either a pair of FIFOs or pipes, one for each direction, or (unusually) explicitly change the direction of the data flow by closing and reopening the FIFO.

The other difference between opening a FIFO and a regular file is the use of the open_flag (the second parameter to open) with the option O_NONBLOCK. Using this open mode not only changes how the open call is processed, but also changes how read and write requests are processed on the returned file descriptor.

There are four legal combinations of O_RDONLY, O_WRONLY, and the O_NONBLOCK flag. We'll consider each in turn.

open(const char *path, O_RDONLY);

In this case, the open call will block; it will not return until a process opens the same FIFO for writing. This is like the first cat example.

open(const char *path, O_RDONLY | O_NONBLOCK);

The open call will now succeed and return immediately, even if the FIFO has not been opened for writing by any process.

open(const char *path, O_WRONLY);

In this case, the open call will block until a process opens the same FIFO for reading.

open(const char *path, O_WRONLY | O_NONBLOCK);

This will always return immediately, but if no process has the FIFO open for reading, open will return an error, –1, and the FIFO won't be opened. If a process does have the FIFO open for reading, the file descriptor returned can be used for writing to the FIFO.

READING AND WRITING FIFOS

Using the O_NONBLOCK mode affects how read and write calls behave on FIFOs.

A read on an empty blocking FIFO (that is, one not opened with O_NONBLOCK) will wait until some data can be read. Conversely, a read on a nonblocking FIFO with no data will return 0 bytes.

A write on a full blocking FIFO will wait until the data can be written. A write on a FIFO that can't accept all of the bytes being written will either:

- ✓ Fail, if the request is for PIPE_BUF bytes or less and the data can't be written.
- ✓ Write part of the data, if the request is for more than PIPE_BUF bytes, returning the number of bytes actually written, which could be 0.

The size of a FIFO is an important consideration. There is a system-imposed limit on how much data can be "in" a FIFO at any one time. This is the #define PIPE_BUF, usually found in limits.h. On Linux and many other UNIX-like systems, this is commonly 4,096 bytes, but it could be as low as 512 bytes on some systems. The system guarantees that writes of PIPE_BUF or fewer bytes on a FIFO that has been opened O_WRONLY (that is, blocking) will either write all or none of the bytes.

SEMAPHORES:

When you write programs that use threads operating in multiuser systems, multiprocessing systems, or a combination of the two, you may often discover that you have *critical sections* of code, where you need to ensure that a single process (or a single thread of execution) has exclusive access to a resource.

Semaphores have a complex programming interface. Fortunately, you can easily provide a much simplified interface that is sufficient for most semaphore-programming problems. When we access a database — the data could be corrupted if multiple programs tried to update the database at exactly the same time.

There's no trouble with two different programs asking different users to enter data for the database; the only potential problem is in the parts of the code that update the database. These sections of code, which actually perform data updates and need to execute exclusively, are called *critical sections*.

Frequently they are just a few lines of code from much larger programs. To prevent problems caused by more than one program simultaneously accessing a shared resource, you need a way of generating and using a token that grants access to only one thread of execution in a critical section at a time.

There are some thread-specific ways you could use either mutex or semaphores to control access to critical sections in a threaded program.

It's surprisingly difficult to write general-purpose code that ensures that one program has exclusive access to a particular resource, although there's a solution known as Dekker's Algorithm.

Unfortunately, this algorithm relies on a "busy wait," or "spin lock," where a process runs continuously, waiting for a memory location to be changed. In a multitasking environment such as Linux, this is an undesirable waste of CPU resources. The situation is much easier if hardware support, generally in the form of specific CPU instructions, is available to support exclusive access.

An example of hardware support would be an instruction to access and increment a register in an atomic way, such that no other instruction (not even an interrupt) could occur between the read/increment/write operations.

One possible solution that you've already seen is to create files using the O_EXCL flag with the open function, which provides atomic file creation. This allows a single process to succeed in obtaining a token: the newly created file. This method is fine for simple problems, but rather messy and very inefficient for more complex examples.

An important step forward in this area of concurrent programming occurred when Edsger Dijkstra, a Dutch computer scientist, introduced the concept of the semaphore which is a special variable that takes only whole positive numbers and upon which programs can only act atomically.

A more formal definition of a semaphore is a special variable on which only two operations are allowed; these operations are officially termed *wait* and *signal*. Because "wait" and "signal" already have special meanings in Linux programming, we'll use the original notation:

- P(semaphore variable) for wait
- V(semaphore variable) for signal

These letters come from the Dutch words for wait (*passeren*: to pass, as in a checkpoint before the critical section) and signal (*vrijgeven*: to give or release, as in giving up control of the critical section). You may also come across the terms "up" and "down" used in relation to semaphores, taken from the use of signaling flags.

Semaphore Definition

The simplest semaphore is a variable that can take only the values 0 and 1, a *binary semaphore.* This is the most common form. Semaphores that can take many positive values are called *general semaphores*.

The definitions of P and V are surprisingly simple. Suppose you have a semaphore variable sv. The two operations are then defined as follows:

P(sv)	If sv is greater than zero, decrement sv. If sv is zero, suspend execution of this process.
V(sv)	If some other process has been suspended waiting for sv, make it resume execution. If no process is suspended waiting for sv, increment sv.

A THEORETICAL EXAMPLE:

Suppose you have two processes *proc1* and *proc2*, both of which need exclusive access to a database at some point in their execution.

You define a single binary semaphore, sv, which starts with the value 1 and can be accessed by both processes.

Both processes then need to perform the same processing to access the critical section of code; indeed, the two processes could simply be different invocations of the same program.

The two processes share the sv semaphore variable. Once one process has executed P(sv), it has obtained the semaphore and can enter the critical section.

The second process is prevented from entering the critical section because when it attempts to execute P(sv), it's made to wait until the first process has left the critical section and executed V(sv) to release the semaphore.

The required Pseudocode is identical for both processes:

```
semaphore sv = 1;
loop forever {
P(sv);
```

```
critical code section;
V(sv);
noncritical code section;
}
```

SEMAPHORE FACILITIES:

All the semaphore functions operate on arrays of general semaphores rather than a single binary semaphore. At first sight, this just seems to make things more complicated, but in complex cases where a process needs to lock multiple resources, the ability to operate on an array of semaphores is a big advantage. *The semaphore function definitions are:*

```
#include <sys/sem.h>
int semctl(int sem_id, int sem_num, int command, ...);
int semget(key_t key, int num_sems, int sem_flags);
int semop(int sem_id, struct sembuf *sem_ops, size_t num_sem_ops);
```

The header file sys/sem.h usually relies on two other header files, sys/types.h and sys/ipc.h. Normally they are automatically included by sys/sem.h and you do not need to explicitly add a #include for them.

semget:

The semget function creates a new semaphore or obtains the semaphore key of an existing semaphore:

int semget(key_t key, int num_sems, int sem_flags);

The first parameter, key, is an integral value used to allow unrelated processes to access the same semaphore.

All semaphores are accessed indirectly by the program supplying a key, for which the system then generates a semaphore identifier. The semaphore key is used only with semget. All other semaphore functions use the semaphore identifier returned from semget.

There is a special semaphore key value, IPC_PRIVATE, that is intended to create a semaphore that only the creating process could access, but this rarely has any useful purpose. You should provide a unique, non-zero integer value for key when you want to create a new semaphore.

The num_sems parameter is the number of semaphores required. This is almost always 1. The sem_flags parameter is a set of flags, very much like the flags to the open function. In addition, these can be bitwise ORed with the value IPC_CREAT to create a new semaphore.

It's not an error to have the IPC_CREAT flag set and give the key of an existing semaphore. The IPC_CREAT flag is silently ignored if it is not required.

You can use IPC_CREAT and IPC_EXCL together to ensure that you obtain a new, unique semaphore. It will return an error if the semaphore already exists.

The semget function returns a positive (nonzero) value on success; this is the semaphore identifier used in the other semaphore functions. On error, it returns –1.

semop:

The function semop is used for changing the value of the semaphore:

int semop(int sem_id, struct sembuf *sem_ops, size_t num_sem_ops);

The first parameter, sem_id, is the semaphore identifier, as returned from semget.

The first parameter, sem_id, is the semaphore identifier, as returned from semget. The second parameter, sem_ops, is a pointer to an array of structures, each of which will have at least the following members:


```
struct sembuf {
short sem_num;
short sem_op;
short sem_flg;
}
```

The first member, sem_num, is the semaphore number, usually 0 unless you're working with an array of semaphores. The sem_op member is the value by which the semaphore should be changed. In general, only two values are used, –1, which is your P operation to wait for a semaphore to become available, and +1, which is your V operation to signal that a semaphore is now available.

The final member, sem_flg, is usually set to SEM_UNDO. This causes the operating system to track the changes made to the semaphore by the current process and, if the process terminates without releasing the semaphore, allows the operating system to automatically release the semaphore if it was held by this process.

Semctl:

The semctl function allows direct control of semaphore information:

int semctl(int sem_id, int sem_num, int command, ...);

The first parameter, sem_id, is a semaphore identifier, obtained from semget. The sem_num parameter is the semaphore number. You use this when you're working with arrays of semaphores. Usually, this is 0, the first and only semaphore. The command parameter is the action to take.

There are many different possible values of command allowed for semctl. Only the two that we describe here are commonly used.

The two common values of command are:

- ✓ **SETVAL**: Used for initializing a semaphore to a known value. The value required is passed as the val member of the union semun. This is required to set the semaphore up before it's used for the first time.
- ✓ **IPC_RMID**: Used for deleting a semaphore identifier when it's no longer required.

The semctl function returns different values depending on the command parameter. For SETVAL and IPC_RMID it returns 0 for success and –1 on error.

MESSAGE QUEUE:

Message queues provide a reasonably easy and efficient way of passing data between two unrelated processes. They have the advantage over named pipes that the message queue exists independently of both the sending and receiving processes.

Message queues provide a way of sending a block of data from one process to another. Additionally, each block of data is considered to have a type, and a receiving process may receive blocks of data having different type values independently. The good news is that you can almost totally avoid the synchronization and blocking problems of named pipes by sending messages. Even better, you can "look ahead" for messages that are urgent in some way. The bad news is that, just like pipes, there's a maximum size limit imposed on each block of data and also a limit on the maximum total size of all blocks on all queues throughout the system.

The message queue function definitions are:

```
#include <sys/msg.h>
int msgctl(int msqid, int cmd, struct msqid_ds *buf);
int msgget(key_t key, int msgflg);
int msgrcv(int msqid, void *msg_ptr, size_t msg_sz, long int msgtype, int msgflg);
int msgsnd(int msqid, const void *msg_ptr, size_t msg_sz, int msgflg);
```

MSGGET:

You create and access a message queue using the msgget function:

int msgget(key_t key, int msgflg);

The program must provide a key value that, as with other IPC facilities, names a particular message queue. The special value IPC_PRIVATE creates a private queue, which in theory is accessible only by the current process. As with semaphores and messages, on some Linux systems the message queue may not actually be private.

The second parameter, msgflg, consists of nine permission flags. A special bit defined by IPC_CREAT must be bitwise ORed with the permissions to create a new message queue. It's not an error to set the IPC_CREAT flag and give the key of an existing message queue. The IPC_CREAT flag is silently ignored if the message queue already exists. The msgget function returns a positive number, the queue identifier, on success or –1 on failure.

MSGSND:

The msgsnd function allows you to add a message to a message queue:

int msgsnd(int msqid, const void *msg_ptr, size_t msg_sz, int msgflg);

The structure of the message is constrained in two ways. First, it must be smaller than the system limit, and second, it must start with a long int, which will be used as a message type in the receive function.

When you're using messages, it's best to define your message structure something like this:

```
struct my_message {
long int message_type;
/* The data you wish to transfer */
}
```

Because the message_type is used in message reception, you can't simply ignore it. You must declare your data structure to include it, and it's also wise to initialize it so that it contains a known value.

The first parameter, msqid, is the message queue identifier returned from a msgget function.

The second parameter, msg_ptr, is a pointer to the message to be sent, which must start with a long int type as described previously.

The third parameter, msg_sz, is the size of the message pointed to by msg_ptr. This size must not include the long int message type.

The fourth parameter, msgflg, controls what happens if either the current message queue is full or the system wide limit on queued messages has been reached. If msgflg has the IPC_NOWAIT flag set, the function will return immediately without sending the message and the return value will be –1. If the msgflg has the IPC_NOWAIT flag clear, the sending process will be suspended, waiting for space to become available in the queue.

On success, the function returns 0, on failure –1. If the call is successful, a copy of the message data has been taken and placed on the message queue.

MSGRCV:

The msgrcv function retrieves messages from a message queue:

int msgrcv(int msqid, void *msg_ptr, size_t msg_sz, long int msgtype, int msgflg);

The first parameter, msqid, is the message queue identifier returned from a msgget function.

The second parameter, msg_ptr, is a pointer to the message to be received, which must start with a long int type as described previously in the msgsnd function.

The third parameter, msg_sz, is the size of the message pointed to by msg_ptr, not including the long int message type.

The fourth parameter, msgtype, is a long int, which allows a simple form of reception priority to be implemented. If msgtype has the value 0, the first available message in the queue is retrieved. If it's greater than zero, the first message with the same message type is retrieved. If it's less than zero, the first message that has a type the same as or less than the absolute value of msgtype is retrieved.

If you simply want to retrieve messages in the order in which they were sent, set msgtype to 0. If you want to retrieve only messages with a specific message type, set msgtype equal to that value. If you want to receive messages with a type of n or smaller, set msgtype to -n

The fifth parameter, msgflg, controls what happens when no message of the appropriate type is waiting to be received. If the IPC_NOWAIT flag in msgflg is set, the call will return immediately with a return value of –1. If the IPC_NOWAIT flag of msgflg is clear, the process will be suspended, waiting for an appropriate type of message to arrive.

On success, msgrcv returns the number of bytes placed in the receive buffer, the message is copied into the user-allocated buffer pointed to by msg_ptr, and the data is deleted from the message queue. It returns –1 on error.

MSGCTL:

The final message queue function is msgctl, which is very similar to that of the control function for shared memory:

int msgctl(int msqid, int command, struct msqid_ds *buf);

The msqid_ds structure has at least the following members:

```
struct msqid_ds {
uid_t msg_perm.uid;
uid_t msg_perm.gid
mode_t msg_perm.mode;
}
```

The first parameter, msqid, is the identifier returned from msgget. The second parameter, command, is the action to take. It can take three values, described in the following table:

Command	Description
IPC_STAT	Retrieves the msqid_ds structure for a queue, and stores it in the address of the buf argument.
IPC_SET	Sets the value of the ipc_perm member of the msqid_ds structure for a queue. Takes the values from the buf argument.
IPC_RMID	Removes the queue from the kernel.

0 is returned on success, –1 on failure. If a message queue is deleted while a process is waiting in a msgsnd or msgrcv function, the send or receive function will fail.

SHARED MEMORY:

Shared memory is one of the three IPC facilities. It allows two unrelated processes to access the same logical memory. Shared memory is a very efficient way of transferring data between two running processes.

Shared memory is a special range of addresses that is created by IPC for one process and appears in the address space of that process. Other processes can then “attach” the same shared memory segment into their own address space.

All processes can access the memory locations just as if the memory had been allocated by malloc. If one process writes to the shared memory, the changes immediately become visible to any other process that has access to the same shared memory.

Shared memory provides an efficient way of sharing and passing data between multiple processes. By itself, shared memory doesn’t provide any synchronization facilities.

Because it provides no synchronization facilities, you usually need to use some other mechanism to synchronize access to the shared memory. Typically, you might use shared memory to provide efficient access to large areas of memory and pass small messages to synchronize access to that memory.

There are no automatic facilities to prevent a second process from starting to read the shared memory before the first process has finished writing to it. It's the responsibility of the programmer to synchronize access.

The functions for shared memory resemble those for semaphores:

```
#include <sys/shm.h>
void *shmat(int shm_id, const void *shm_addr, int shmflg);
int shmctl(int shm_id, int cmd, struct shmid_ds *buf);
int shmdt(const void *shm_addr);
int shmget(key_t key, size_t size, int shmflg);
```

As with semaphores, they include files sys/types.h and sys/ipc.h are normally automatically included by shm.h.

SHMGET:

You create shared memory using the shmget function:

```
int shmget(key_t key, size_t size, int shmflg);
```

As with semaphores, the program provides key, which effectively names the shared memory segment, and the shmget function returns a shared memory identifier that is used in subsequent shared memory functions. There's a special key value, IPC_PRIVATE that creates shared memory private to the process. You wouldn't normally use this value, and you may find the private shared memory is not actually private on some Linux systems.

The second parameter, size, specifies the amount of memory required in bytes.

The third parameter, shmflg, consists of nine permission flags that are used in the same way as the mode flags for creating files. A special bit defined by IPC_CREAT must be bitwise ORed with the permissions to create a new shared memory segment. It's not an error to have the IPC_CREAT flag set and pass the key of an existing shared memory segment. The IPC_CREAT flag is silently ignored if it is not required.

The permission flags are very useful with shared memory because they allow a process to create shared memory that can be written by processes owned by the creator of the shared memory, but only read by processes that other users have created.

You can use this to provide efficient read-only access to data by placing it in shared memory without the risk of its being changed by other users. If the shared memory is successfully created, shmget returns a nonnegative integer, the shared memory identifier. On failure, it returns –1.

SHMAT:

When you first create a shared memory segment, it's not accessible by any process. To enable access to the shared memory, you must attach it to the address space of a process. You do this with the shmat function:

void *shmat(int shm_id, const void *shm_addr, int shmflg);

The first parameter, shm_id, is the shared memory identifier returned from shmget.

The second parameter, shm_addr, is the address at which the shared memory is to be attached to the current process. This should almost always be a null pointer, which allows the system to choose the address at which the memory appears.

The third parameter, shmflg, is a set of bitwise flags. The two possible values are SHM_RND, which, in conjunction with shm_addr, controls the address at which the shared memory is attached, and SHM_RDONLY, which makes the attached memory read-only. It's very rare to need to control the address at which shared memory is attached; you should normally allow the system to choose an address for you, because doing otherwise will make the application highly hardware-dependent.

If the shmat call is successful, it returns a pointer to the first byte of shared memory. On failure –1 is returned.

The shared memory will have read or write access depending on the owner (the creator of the shared memory), the permissions, and the owner of the current process. Permissions on shared memory are similar to the permissions on files.

An exception to this rule arises if shmflg & SHM_RDONLY is true. Then the shared memory won't be writable, even if permissions would have allowed write access.

SHMDT:

The shmdt function detaches the shared memory from the current process. It takes a pointer to the address returned by shmat. On success, it returns 0, on error –1. Note that detaching the shared memory doesn't delete it; it just makes that memory unavailable to the current process.

SHMCTL:

The control functions for shared memory are (thankfully) somewhat simpler than the more complex ones for semaphores:

int shmctl(int shm_id, int command, struct shmid_ds *buf);

The shmid_ds structure has at least the following members:

```
struct shmid_ds {
uid_t shm_perm.uid;
uid_t shm_perm.gid;
mode_t shm_perm.mode;
}
```

The first parameter, shm_id, is the identifier returned from shmget.

The second parameter, command, is the action to take. It can take three values, shown in the following table.

Command	Description
IPC_STAT	Sets the data in the shmid_ds structure to reflect the values associated with the shared memory.
IPC_SET	Sets the values associated with the shared memory to those provided in the shmid_ds data structure, if the process has permission to do so.
IPC_RMID	Deletes the shared memory segment.

The third parameter, buf, is a pointer to the structure containing the modes and permissions for the shared memory. On success, it returns 0, on failure, –1.

IPC STATUS COMMANDS:

Most Linux systems provide a set of commands that allow command-line access to IPC information, and to tidy up stray IPC facilities. These are the **ipcs** and **ipcrm** commands, which are very useful when you're developing programs.

One of the irritations of the IPC facilities is that a poorly written program, or a program that fails for some reason, can leave its IPC resources (such as data in a message queue) loitering on the system long after the program completes. This can cause a new invocation of the program to fail, because the program expects to start with a clean system, but actually finds some leftover resource.

The status (ipcs) and remove (ipcrm) commands provide a way of checking and tidying up IPC facilities.

DISPLAYING SEMAPHORE STATUS:

To examine the state of semaphores on the system, use the ipcs -s command. If any semaphores are present, the output will have this form:

$ **./ipcs -s**

------ Semaphore Arrays --------

key semid owner perms nsems

0x4d00df1a 768 rick 666 1

You can use the ipcrm command to remove any semaphores accidentally left by programs. To delete the preceding semaphore, the command (on Linux) is

$ **./ipcrm -s 768**

Some much older Linux systems used to use a slightly different syntax:

$ **./ipcrm sem 768**

DISPLAYING SHARED MEMORY STATUS:

Like semaphores, many systems provide command-line programs for accessing the details of shared memory. These are ipcs -m and ipcrm -m <id> (or ipcrm shm <id>).

Here's some sample output from ipcs -m:

$ **ipcs -m**

------ Shared Memory Segments --------

key shmid owner perms bytes nattch status

0x00000000 384 rick 666 4096 2 dest

This shows a single shared memory segment of 4 KB attached by two processes.

The ipcrm -m <id> command allows shared memory to be removed. This is sometimes useful when a program has failed to tidy up shared memory.

DISPLAYING MESSAGE QUEUE STATUS:

For message queues the commands are ipcs -q and ipcrm -q <id> (or ipcrm msg <id>).

Here's some sample output from ipcs -q:

$ **ipcs -q**

------ Message Queues --------

Key	msqid	owner	perms	used-bytes	messages
0x000004d2	3384	rick	666	2048	2

This shows two messages, with a total size of 2,048 bytes in a message queue. The ipcrm -q <id> command allows a message queue to be removed.

SOCKET:

A *socket* is a communication mechanism that allows client/server systems to be developed either locally, on a single machine, or across networks. Linux functions such as printing, connecting to databases, and serving web pages as well as network utilities such as rlogin for remote login and ftp for file transfer usually use sockets to communicate.

Sockets are created and used differently from pipes because they make a clear distinction between client and server. The socket mechanism can implement multiple clients attached to a single server.

SOCKET CONNECTIONS:

You can think of socket connections as telephone calls into a busy building. A call comes into an organization and is answered by a receptionist who puts the call through to the correct department (the server process) and from there to the right person (the server socket).

Each incoming call (client) is routed to an appropriate end point and the intervening operators are free to deal with further calls. Before you look at the way socket connections are established in Linux systems, you need to understand how they operate 1for socket applications that maintain a connection.

First, a server application creates a socket, which like a file descriptor is a resource assigned to the server process and that process alone. The server creates it using the system call socket, and it can't be shared with other processes.

Next, the server process gives the socket a name. Local sockets are given a filename in the Linux file system, often to be found in /tmp or /usr/tmp. For network sockets, the filename will be a service identifier (port number/access point) relevant to the particular network to which the clients can connect. This identifier allows Linux to route incoming connections specifying a particular port number to the correct server process.

For example, a web server typically creates a socket on port 80, an identifier reserved for the purpose. Web browsers know to use port 80 for their HTTP connections to web sites the user wants to read. A socket is named using the system call bind. The server process then waits for a client to connect to the named socket. The system call, listen, creates a queue for incoming connections. The server can accept them using the system call accept.

When the server calls accept, a new socket is created that is distinct from the named socket. This new socket is used solely for communication with this particular client. The named socket remains for further connections from other clients. If the server is written appropriately, it can take advantage of multiple connections. A web server will do this so that it can serve pages to many clients at once.

For a simple server, further clients wait on the listen queue until the server is ready again. The client side of a socket-based system is more straightforward. The client creates an unnamed socket by calling socket. It then calls connect to establish a connection with the server by using the server's named socket as an address.

Once established, sockets can be used like low-level file descriptors, providing two-way data communications.

SOCKET ATTRIBUTES:

Sockets are characterized by three attributes: *domain*, *type*, and *protocol*. They also have an address used as their name. The formats of the addresses vary depending on the domain, also known as the *protocol family*. Each protocol family can use one or more address families to define the address format.

SOCKET DOMAINS:

Domains specify the network medium that the socket communication will use. The most common socket domain is ***AF_INET***, which refers to Internet networking that's used on many Linux local area networks and, of course, the Internet itself.

The underlying protocol, Internet Protocol (IP), which only has one address family, imposes a particular way of specifying computers on a network. This is called the IP address. Although names almost always refer to networked machines on the Internet, these are translated into lower-level IP addresses. An example IP address is 192.168.1.99.

All IP addresses are represented by four numbers, each less than 256, a so-called *dotted quad*. When a client connects across a network via sockets, it needs the IP address of the server computer. There may be several services available at the server computer.

A client can address a particular service on a networked machine by using an IP port. A port is identified within the system by assigning a unique 16-bit integer and externally by the combination of IP address and port number.

The sockets are communication end points that must be bound to ports before communication is possible. Servers wait for connections on particular ports. Well-known services have allocated port numbers that are used by all Linux and UNIX machines. These are usually, but not always, numbers less than 1024.

Examples are the printer spooler (515), rlogin (513), ftp (21), and httpd (80). The last of these is the standard port for web servers. Usually, port numbers less than 1024 are reserved for system services and may only be served by processes with superuser privileges.

The UNIX file system domain, AF_UNIX, can be used by sockets based on a single computer that perhaps isn't networked.

SOCKET TYPES:

A socket domain may have a number of different ways of communicating, each of which might have different characteristics. This isn't an issue with AF_UNIX domain sockets, which provide a reliable two way communication path. In networked domains, however, you need to be aware of the characteristics of the underlying network and how different communication mechanisms are affected by them.

Internet protocols provide two communication mechanisms with distinct levels of service: *streams* and *datagrams*.

Stream Sockets:

Stream sockets (in some ways similar to standard input/output streams) provide a connection that is a sequenced and reliable two-way byte stream. Thus, data sent is guaranteed not to be lost, duplicated, or reordered without an indication that an error has occurred. Large messages are fragmented, transmitted, and reassembled. This is similar to a file stream, which also accepts large amounts of data and splits it up for writing to the low-level disk in smaller blocks. Stream sockets have predictable behavior.

Stream sockets, specified by the type SOCK_STREAM, are implemented in the AF_INET domain by TCP/IP connections. They are also the usual type in the AF_UNIX domain.

Datagram Sockets:

In contrast, a datagram socket, specified by the type SOCK_DGRAM, doesn't establish and maintain a connection.

There is also a limit on the size of a datagram that can be sent. It's transmitted as a single network message that may get lost, duplicated, or arrive out of sequence—ahead of datagrams sent after it.

Datagram sockets are implemented in the AF_INET domain by UDP/IP connections and provide an unsequenced, unreliable service. (UDP stands for User Datagram Protocol.) However, they are relatively inexpensive in terms of resources, because network connections need not be maintained. They're fast because there is no associated connection setup time.

Datagrams are useful for "single-shot" inquiries to information services, for providing regular status information, or for performing low-priority logging. They have the advantage that the death of a server doesn't unduly inconvenience a client and would not require a client restart. Because datagram-based servers usually retain no connection information, they can be stopped and restarted without disturbing their clients.

SOCKET PROTOCOLS:

Where the underlying transport mechanism allows for more than one protocol to provide the requested socket type, you can select a specific protocol for a socket. The socket system call creates a socket and returns a descriptor that can be used for accessing the socket.

The socket created is one end point of a communication channel. The socket system call returns a descriptor that is in many ways similar to a low-level file descriptor. When the socket has been connected to another end-point socket, you can use the read and write system calls with the descriptor to send and receive data on the socket. The close system call is used to end a socket connection.

SOCKET ADDRESSES:

Each socket requires its own address format. For an **AF_UNIX** socket, the address is described by a structure, sockaddr_un, defined in the *sys/un.h* include file.

```
struct sockaddr_un {
sa_family_t sun_family; /* AF_UNIX */
char sun_path[]; /* pathname */
};
```

So that addresses of different types may be passed to the socket-handling system calls, each address format is described by a similar structure that begins with a field (in this case, sun_family) that specifies the address type. In the AF_UNIX, the address is specified by a filename in the sun_path field of the structure.

In the **AF_INET**, the address is specified using a structure called sockaddr_in, defined in netinet/in.h, which contains at least these members:

```
struct sockaddr_in {
short int sin_family; /* AF_INET */
unsigned short int sin_port; /* Port number */
struct in_addr sin_addr; /* Internet address */
};
```

Socket:

To perform network I/O, the first thing a process must do is, call the **socket** function, specifying the type of communication protocol desired and protocol family, etc.

```
#include <sys/types.h>
#include <sys/socket.h>
int socket (int family, int type, int protocol);
```

This call returns a socket descriptor that you can use in later system calls or -1 on error, **family** specifies the protocol family and is one of the constants shown below:

Family	Description
AF_INET	IPv4 protocols
AF_INET6	IPv6 protocols
AF_LOCAL	Unix domain protocols
AF_ROUTE	Routing Sockets

type specifies the kind of socket you want. It can take one of the following values:

Type	Description
SOCK_STREAM	Stream socket
SOCK_DGRAM	Datagram socket
SOCK_SEQPACKET	Sequenced packet socket
SOCK_RAW	Raw socket

protocol − The argument should be set to the specific protocol type given below, or 0 to select the system's default for the given combination of family and type:

Protocol	Description
IPPROTO_TCP	TCP transport protocol
IPPROTO_UDP	UDP transport protocol
IPPROTO_SCTP	SCTP transport protocol

Naming a Socket:

To make a socket (as created by a call to socket) available for use by other processes, a server program needs to give the socket a name. Thus, AF_UNIX sockets are associated with a file system pathname, AF_INET sockets are associated with an IP port number.

#include <sys/socket.h>
int bind(int socket, const struct sockaddr *address, size_t address_len);

The **bind** system call assigns the address specified in the parameter, address, to the unnamed socket associated with the file descriptor socket. The length of the address structure is passed as address_len.

The length and format of the address depend on the address family. A particular address structure pointer will need to be cast to the generic address type (struct sockaddr *) in the call to bind.

On successful completion, bind returns 0. If it fails, it returns -1 and sets errno to one of the following values:

ERRNO VALUE	DESCRIPTION
EBADF	The file descriptor is invalid.
ENOTSOCK	The file descriptor doesn't refer to a socket.
EINVAL	The file descriptor refers to an already-named socket.
EADDRNOTAVAIL	The address is unavailable.
EADDRINUSE	The address has a socket bound to it already.

There are some more values for AF_UNIX sockets:

ERRNO VALUE	DESCRIPTION
EACCESS	Can't create the file system name due to permissions.
ENOTDIR, ENAMETOOLONG	Indicates a poor choice of filename.

Creating a Socket Queue:

To accept incoming connections on a socket, a server program must create a queue to store pending requests. It does this using the **listen** system call.

#include <sys/socket.h>
int listen(int socket, int backlog);

A Linux system may limit the maximum number of pending connections that may be held in a queue. Subject to this maximum, listen sets the queue length to backlog. Incoming connections up to this queue length are held pending on the socket; further connections will be refused and the client's connection will fail.

This mechanism is provided by listen to allow incoming connections to be held pending while a server program is busy dealing with a previous client. A value of 5 for backlog is very common.

The listen function will return 0 on success or -1 on error. Errors include EBADF, EINVAL, and ENOTSOCK, as for the bind system call.

Accepting Connections:

Once a server program has created and named a socket, it can wait for connections to be made to the socket by using the accept system call.

#include <sys/socket.h>
int accept(int socket, struct sockaddr *address, size_t *address_len);

The **accept** system call returns when a client program attempts to connect to the socket specified by the parameter socket. The client is the first pending connection from that socket's queue. The accept function creates a new socket to communicate with the client and returns its descriptor. The new socket will have the same type as the server listen socket.

The socket must have previously been named by a call to bind and had a connection queue allocated by listen. The address of the calling client will be placed in the sockaddr structure pointed to by address.

A null pointer may be used here if the client address isn't of interest. The address_len parameter specifies the length of the client structure. If the client address is longer than this value, it will be truncated.

Before calling accept, address_len must be set to the expected address length. On return, address_len will be set to the actual length of the calling client's address structure.

If there are no connections pending on the socket's queue, accept will block (so that the program won't continue) until a client does make connection. You can change this behavior by using the O_NONBLOCK flag on the socket file descriptor, using the fcntl function.

The accept function returns a new socket file descriptor when there is a client connection pending or -1 on error.

Possible errors are similar to those for bind and listen, with the addition of EWOULDBLOCK, where O_NONBLOCK has been specified and there are no pending connections. The error EINTR will occur if the process is interrupted while blocked in accept.

Requesting Connections:

Client programs connect to servers by establishing a connection between an unnamed socket and the server listen socket. They do this by calling **connect**.

#include <sys/socket.h>
int connect(int socket, const struct sockaddr *address, size_t address_len);

The socket specified by the parameter socket is connected to the server socket specified by the parameter address, which is of length address_len. The socket must be a valid file descriptor obtained by a call to socket.

If it succeeds, connect returns 0, and -1 is returned on error. Possible errors this time include the following:

ERRNO VALUE	DESCRIPTION
EBADF	The file descriptor is invalid.
EALREADY	A connection is already in progress for this socket.
ETIMEDOUT	A connection timeout has occurred.
ECONNREFUSED	The requested connection was refused by the server.

If the connection can't be set up immediately, connect will block for an unspecified timeout period. Once this timeout has expired, the connection will be aborted and connect will fail.

However, if the call to connect is interrupted by a signal that is handled, the connect call will fail (with errno set to EINTR), but the connection attempt won't be aborted—it will be set up asynchronously, and the program will have to check later to see if the connection was successful.

As with accept, the blocking nature of connect can be altered by setting the O_NONBLOCK flag on the file descriptor. In this case, if the connection can't be made immediately, connect will fail with errno set to EINPROGRESS and the connection will be made asynchronously. Though asynchronous connections can be tricky to handle, you can use a call to select on the socket file descriptor to check that the socket is ready for writing.

Closing a Socket:

You can terminate a socket connection at the server and client by calling close, just as you would for low level file descriptors. You should always close the socket at both ends. For the server, you should do this when read returns zero. Note that the close call may block if the socket has untransmitted data, is of a connection-oriented type.

int close(int sockfd);

This call returns 0 on success; otherwise it returns -1 on error, **sockfd** is a socket descriptor returned by the socket function.

SOCKET COMMUNICATIONS:

Now that we have covered the basic system calls associated with sockets, let's take a closer look at the example programs. You'll try to convert them to use a network socket rather than a file system socket.

The file system socket has the disadvantage that, unless the author uses an absolute pathname, it's created in the server program's current directory. To make it more generally useful, you need to create it in a globally accessible directory (such as /tmp) that is agreed between the server and its clients. For network sockets, you need only choose an unused port number.

For the example, select port number 9734. This is an arbitrary choice that avoids the standard services (you can't use port numbers below 1024 because they are reserved for system use).

You'll run your client and server across a local network, but network sockets are not only useful on a local area network; any machine with an Internet connection (even a modem dial-up) can use network sockets to communicate with others.

You can even use a network-based program on a stand-alone UNIX computer because a UNIX computer is usually configured to use a loopback network that contains only itself. For illustration purposes, this example uses this loopback network, which can also be useful for debugging network applications because it eliminates any external network problems.

The loopback network consists of a single computer, conventionally called localhost, with a standard IP address of 127.0.0.1. This is the local machine. You'll find its address listed in the network hosts file, /etc/hosts, with the names and addresses of other hosts on shared networks.

Each network with which a computer communicates has a hardware interface associated with it. A computer may have different network names on each network and certainly will have different IP addresses.

Network Client:

Here's a client program, client2.c, to connect to a network socket via the loopback network.

1. ***Make the necessary includes and set up the variables***:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <stdio.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <stdlib.h>
int main()
{
int sockfd;
int len;
struct sockaddr_in address;
int result;
char ch = 'A';
```

2. ***Create a socket for the client:***

```
sockfd = socket(AF_INET, SOCK_STREAM, 0);
```

3. ***Name the socket, as agreed with the server:***

```
address.sin_family = AF_INET;
address.sin_addr.s_addr = inet_addr("127.0.0.1");
address.sin_port = 9734;
len = sizeof(address);
```

4. **Connect your socket to the server's socket:**

```
result = connect(sockfd, (struct sockaddr *)&address, len);
if(result == -1) {
perror("oops: client1");
exit(1);
}
```

5. ***You can now read and write via sockfd:***

```
write(sockfd, &ch, 1);
read(sockfd, &ch, 1);
printf("char from server = %c\n", ch);
close(sockfd);
exit(0);
}
```

When you run this program, it fails to connect because there isn't a server running on port 9734 on this machine.

$ **./client2**

oops: client2: Connection refused

$

Network Server:

You also need to modify the server program to wait for connections on your chosen port number. Here's a modified server: server2.c.

1. ***Make the necessary includes and set up the variables:***

```
#include <sys/types.h>
#include <sys/socket.h>
#include <stdio.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <stdlib.h>
int main()
{
int server_sockfd, client_sockfd;
int server_len, client_len;
struct sockaddr_in server_address;
struct sockaddr_in client_address;
```


2. ***Create an unnamed socket for the server:***

```
server_sockfd = socket(AF_INET, SOCK_STREAM, 0);
```

3. ***Name the socket:***

```
server_address.sin_family = AF_INET;
server_address.sin_addr.s_addr = inet_addr("127.0.0.1");
server_address.sin_port = 9734;
server_len = sizeof(server_address);
bind(server_sockfd, (struct sockaddr *)&server_address, server_len);
```

4. ***Create a connection queue and wait for clients:***

```
listen(server_sockfd, 5);
while(1) {
char ch;
printf("server waiting\n");
```

5. ***Accept a connection:***

```
client_len = sizeof(client_address);
client_sockfd = accept(server_sockfd,
(struct sockaddr *)&client_address, &client_len);
```

6. ***Read and write to client on client_sockfd:***

```
read(client_sockfd, &ch, 1);
ch++;
write(client_sockfd, &ch, 1);
close(client_sockfd);
}
}
```


When you run the server program, it creates a socket and waits for connections. If you start it in the background so that it runs independently, you can then start clients in the foreground.

```
$ ./server2 &
[1] 1094
$ server waiting
```

As it waits for connections, the server prints a message. Now, when you run the client program, you are successful in connecting to the server. Because the server socket exists, you can connect to it and communicate with the server.

```
$ ./client2
server waiting
char from server = B
$
```

The output from the server and the client get mixed on the terminal, but you can see that the server has received a character from the client, incremented it, and returned it. The server then continues and waits for the next client. If you run several clients together, they will be served in sequence, although the output you see may be more mixed up.