National Centre for Computer Animation

# LINUX SYSTEMS PROGRAMMING

JONATHAN MACEY

September 27, 2005

BOURNEMOUTH MEDIA SCHOOL

# Contents

# List of Figures

# List of Tables

Note this lab book is based on the excellent book "Advanced Programming in the Unix Environment" by W. Richard Stevens. Addison Wesley 1993. It is strongly recommended that you read this book in conjunction with this lab book.

# Part I

# Unix Programming

# Chapter 1

# Command Line Arguments

When a program is executed from a command line[1] it may be passed different parameters as arguments to the program. Many examples of this may be seen in the Unix operating system with standard commands. For example **ls** may be used on it's own or with different flags such as **ls -al** where the **-al** is a command line argument.

## 1.1 The main() function

The **main** function is the entry point in any C program and may be declared in a number of ways as shown below

Function Declaration: 1: The different forms of main

```
void main(void)
int main(void)
int main(int argc, char *argv[])
```

The first statement declares main as not returning a value and not receiving any values from the command line.

The second returns an integer value when the program has exited.

An finally main has two parameters, int argc and char *argv[]. These are the only parameters that main in allowed and are used in the following ways

int argc is the argument count and passes the number of arguments passed at the command line.

char * argv is a pointer to a list of command line arguments which may be accessed as if they were a series of strings.

[1]i.e. from an xterm or a dos session

### 1.1.1 A simple Example

The example below shows how to use argc and argv and print out the results.

Program 1: Simple Argument example

```
#include <stdio.h>
int main (int argc, char * argv[])
{
int count;
for(count =0; count < argc; count++)
        printf("argument no %d = %s\n",count,argv[count]);
        return 0;
}
```

Running this program using the following command line

```
#argexample -a -b -c -d e

argument no 0 = argexample
argument no 1 = -a
argument no 2 = -b
argument no 3 = -c
argument no 4 = -d
argument no 5 = e
```

You will notice that the first argument (argv[0]) prints out **argexample**. This is because the whole of the command line typed is passed into the program.

When the program is executed the command line is parsed and each space separated command is placed into each argv array element. Finally the size of the argv array is placed into argc.

## 1.2 Parsing Command line arguments

The following program parses the command line to find one of three flags. These are **-mode1**, **-mode2** and **-help**. The mode flags both accept a further parameter but help does not.

Finally an incorrect parameter causes the program to exit.

Program 2: A more complex argument example

```
#include <stdio.h>
int main (int argc, char * argv[])
{
int argcount=1; /* argv 1 is the first parameter */
int mode;

while(argcount < argc )
     {
     if(strcmp(argv[argcount],"-mode1") == 0)
       {
        argcount++;
        printf("mode 1 parameter = %s\n",argv[argcount ++]);
        mode =1;
        }
     else if (strcmp(argv[argcount],"-mode2") == 0)
        {
        argcount++;
        printf("mode 2 parameter = %s\n",argv[argcount ++]);
        mode =2;
        }
     else if(strcmp(argv[argcount],"-help") == 0)
        {
        argcount++;
        printf("Help mode\n");
        }
     else
        {
        printf("unknown command %s\n",argv[argcount]);
        exit(1);
        }
     }
printf("end of program in mode %d\n",mode);
}
```

## 1.3 Using getopt

An easier way to parse command line options is using the standard C library function getopt as shown below

```
#include <unistd.h>

getopt(int argc, char * const argv[],const char *optstring);
extern char *optarg;
extern int optind, opterr, optopt;
```

For info type *man 3 getopt*

### 1.3.1 A simple getopt program

Program 3: Using getopt

```
#include <stdio.h>   // for printf
#include <unistd.h>  // for various unix defines
#include <stdlib.h>  //for getopt
#include <string.h>  // for strcpy
#include <stdbool.h> //for bool and true false
// define the command line argument parameters : indicates 2nd arg
#define ARGUMENTS "vdhf:"
int main(int argc, char *argv[])
{
```

```
// define some inital global variables for the program
bool Verbose = false;
bool Debug = false;
bool Help = false;
// pointer to hold the string from the command line
char *Argument;
// the character returned from the getopt function
char c;
// now loop and parse the command line options
while( (c=getopt(argc,argv,ARGUMENTS)) !=EOF)
        {
        switch(c) // which option has been chosen
            {
            case ’v’ : // -v
               printf("Setting Verbose Mode\n");
               Verbose = true;
            break;
            case ’d’ : // -d
               printf("Setting Debug Mode\n");
               Debug = true;
             break;
             case ’h’ : //-h
               printf("Help Mode\n");
               Help=true;
              break;
              case ’f’ : // -f [ARG]
             printf("passing an argument The argument is %s\n",optarg);
                Argument = (char *)malloc(sizeof(optarg));
                strcpy(Argument,optarg);
              break;
              case ’?’ : // unknown option report this and exit
                 // where optopt is the current option
                  printf("Unknown argument %c\n",optopt);
                  printf("Valid arguments are -v -d -h -f [name]\n");
                  printf("Will now exit\n");
                 exit(EXIT_FAILURE);
             }
       }
printf("Argument parsed current modes are as follows\n");
printf("Debug = %d\n",Debug);
printf("Verbose = %d\n",Verbose);
printf("Help = %d\n",Help);
printf("Argument passed is  = %s\n",Argument);
// as we used malloc we need to free any memory we allocated
free(Argument);
return EXIT_SUCCESS;
}
```

# Chapter 2

# Environment Variables

## 2.1 The Environment List

As well as command line arguments every C program is also passed an environment list. Like the argument list. the environment list is an array of character pointers, with each pointer containing the address of a null terminated C string. The address of the array of pointers is contained in the global variable environ defined as

```
    extern char **environ;
```

In figure 2.1the environment consists of 5 strings.

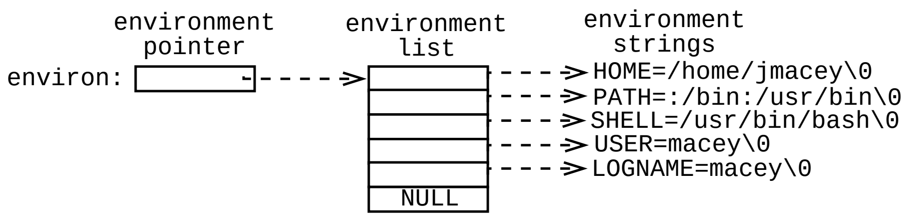

Figure 2.1: Environment List

In figure 2.1 environ is know as the environment pointer. The array of pointers is known as the environment list, and the strings the environment strings.

By convention the environment consists of *name = value* where name is in upper case. However this is just a convention and is not enforced by the operating system.

An example of the environment list is shown in the program below

Program 4: Environment List program

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

    /* some versions of C don’t include the definition of environ in the
```

```
   standard include paths */
extern char **environ;

int main(void)
{
/* create a pointer to point to the current entry in the environment list */
char *current_environ_ptr;

/* now copy this pointer to point to the first entry in the list */
current_environ_ptr=*environ;

/* now we loop through all of the environment table entries until we
   get to the last entry signified by a NULL */

   do
      {
      /* we now print out the environment string */
      printf("%s\n",current_environ_ptr);
      /* now point current_environment_ptr to the next entry in the list */

      current_environ_ptr=*environ++;
      /* and check to see if it is the last */

      }while(current_environ_ptr != NULL);

return 0;
}
```

### 2.1.1 Setting Environment Variables

The Unix kernel never looks at the environment strings - their interpretation is up to application programs. For example at login time the environment variables HOME,USER,PATH ,etc are set for the shell (which is generally a c-program) to use.

Other programs require their own environment variables to be set or for the program to set environment variables. This is done by the following C functions

Function Declaration: 2: Environment Functions

```
#include <stdlib.h>

int putenv(const char *str);
int setenv(const char *name, const char *value, int rewrite);

Both these function return 0 if OK nonzero on error.

void unsetenv(const char *name);
```

*putenv* takes a string of the form *name=value* and places it in the environment list. If the name already exists it's old definition is first removed.

*setenv* sets *name* to *value*. If name already exists in the environment then (a) if *rewrite* is nonzero, the existing definition for name is first removed. (b) if *rewrite* is 0, an existing definition for name is not removed (and name is not set to the new value, and no error occurs).

*unsetenv* removes any definition of name. It is not an error if such a definition does not exist.

**How environment setting works**

Environment strings are typically stored at the top of a processe's memory space (above the stack). Deleting a string is simple as all the function has to do is remove the pointer from the list and shuffle the rest up by one.

However adding a string or modifying and existing string is more difficult, the space at the top of the stack cannot be expanded because it is at the top of the address space of the process. Since it is at the top it can't be expanded upwards and it can't be expanded downward because all the stack frames below can't be moved therefore the following operations must be carried out depending upon the state of the environment.

1. If modifying an existing name :
   (a) If the size of the new *value* is less then or equal to the size of the existing *value*, it may be directly replaced.
   (b) If the new *value* is larger than the old one, the programmer must *malloc* enough room for the new string and then replace the pointer in the environment list with the new pointer from the *malloc* operation.
2. If adding a new *name* first the programmer must *malloc* enough room for *name=value* and copy the string to this area.
   (a) If it's the first time the new *name* has been added a new environment list must be created using *malloc* to create enough room for the new name to be added. Then the names from the old list are copied to the new list, adding the new string and finally the NULL pointer to the end. Finally the global variable *environ* is set to point to the new list. However it must be noted that as most of the pointers in the list we allocated before the program was run they are still located at the top of the stack.
   (b) If it isn't the first time the new string has been added to the environment list then room has already been *malloc*ed for it so *realloc* may be used to allocate more (or less) room for the new string.

However the *setenv* and *putenv* functions make this transparent to the programmer.

### 2.1.2 Getting Environment Variables

There is one function used to fetch values from the environment as detailed below

Function Declaration: 3: getenv function

```
#include <stdlib.h>
char *getenv(const char *name);
```

**returns pointer to value associated with name NULL if not found.**

Although the program earlier manipulates the *environ* variable directly this should not be used in a normal program; instead the getenv function should be used.

# Chapter 3

# The Standard I/O Library

The Standard I/O library is specified by the ANSI C standard because it has been implemented on many operating systems. This library handles details such as buffer allocation and performing I/O in an optimized way as far a reading and writing block sizes are concerned.

This library was written in 1975 by Dennis Ritchie and was a major revision of the Portable I/O library written by Mike Lesk. However little of this library has changed in the last 24 years.

## 3.1 Streams and File Objects

When a file is opened a file descriptor is returned and this file descriptor is used for each subsequent I/O operation, when any operation is carried out on the file descriptor its is known as a stream.

When a stream is opened the standard I/O function fopen returns a pointer to a FILE object. This object is a structure which contains information required by the standard I/O library to perform I/O operations, however this information is effectively transparent to the programmer and is not generally used.[1]

### 3.1.1 Opening a stream

The following three functions are used to open a standard I/O stream

Function Declaration: 4: file open functions

```
#include <stdio.h>
FILE *fopen(const char *pathname, const char * type)
FILE *freopen(const char *pathname, const char *type, FILE *fp)
FILE *fdopen(int filedes, const char *type)
```

**All three of these functions return file pointer if OK or a NULL on error.**

The differences in these three functions are as follows

[1] However this structure may be examined from looking in the stdio.h header file

1. fopen opens a specified file.
2. freopen opens a specified file on a specified stream, closing the stream first if it is already open.
3. fdopen takes an existing file descriptor and associates a standard I/O stream with the descriptor.

ANSI C specifies 15 different values for the type argument as shown in table 3.1

| type | Description |
|---|---|
| r rb | open for reading |
| w wb | truncate to 0 length or create for writing |
| a ab | append; open for writing at the end of file or create for writing |
| r+ r+b rb+ | open for reading and writing |
| w+ w+b wb+ | truncate to 0 length or create for reading and writing |
| a+ a+b ab+ | open or create for reading and writing at end of file |

Table 3.1: Read Write modes for fopen

Using **b** as part of the type allows the standard I/O system to differentiate between a text file and a binary file. Since the Unix kernel doesn't differentiate between these types of file the **b** has no effect, however on other operating system (Windows) it does.

### 3.1.2 Reading and writing a Stream

Once a stream has been opened there are three types of unformatted I/O which may be performed these are as follow

1. Character at a time I/O.
2. Line at a time I/O
3. Direct I/O

### 3.1.3 Input functions

There are three functions which allow the programmer to read one character at a time

Function Declaration: 5: File Input Functions

```
#include <stdio.h>
int getc(FILE *fp);
int fgetc(FILE *fp);
int getchar(void)
```

**All three functions return the next character if OK or EOF on end of file or error.**

Each of these functions return the next character as an unsigned char converted to an int. The reason for specifying unsigned is so that the high order bit, if set, doesn't cause the return value to be negative. This means that if the value is converted to an int all possible ASCII characters may be returned as well as the negative value -1 which generally indicates EOF (end of file).

### 3.1.4 Output Functions

There are three output functions which corresponds to each of the input functions as follow

Function Declaration: 6: File output functions

```
#include <stdio.h>
int putc(int c, FILE *fp);
int fputc(int c, FILE *fp);
int putchar(int c);
```

**All three functions return c if OK or EOF on error.**

### 3.1.5 Line at a time I/O

Line at a time input is provided by the following functions

Function Declaration: 7: Line at a time input functions

```
#include <stdio.h>
char *fgets(char *buf,int n, FILE *fp);
char *gets(char *buf);
```

**Both return buf if OK NULL on EOF or error.**

Both of these functions specify the address of buffer to read the line into. gets reads from stdin while fgets freads from the specified stream.

With fgets the size of the buffer (characters to read) are specified by n. This function reads up to and including the next newline, but no more than n-1 characters into buffer. The buffer is then terminated by a NULL byte (**0**).

Line at a time output is provided by fputs and puts as follows

Function Declaration: 8: Line at a time output functions

```
#include <stdio.h>
int fputs(const char *str,FILE *fp);
int puts(const char *str);
```

**Both functions return nonnegative values if OK or EOF on error.**

### 3.1.6 Binary I/O

The previous functions operated with either one character at a time or a line at a time, however it is sometimes desirable to read or write complete structures to or from a file. This is done using the following binary I/O functions

Function Declaration: 9: Binary IO functions

```
#include <stdio.h>
size_t fread(void *ptr, size_t size, size_t nobj, FILE *fp);
size_t fwrite(const void *ptr, size_t size, size_t nobj, FILE *fp);
```

**Both these functions return the number of object read or written.**

The return type of both the above functions is the variable **size_t** any variable ending in a _t is known as a 'type' variable an is used as part of the platform Independence of Unix based C programs.

For example in most cases **size_t** will be an integer, however on some systems it may be a different type. By defining a function to return a user defined type (such as **size_t**) and then defining **size_t** in a header file as the correct type for the system; code will remain portable.

This is even more important with some structures which hold O/S dependant data such as date / time information or file system information as this may be stored differently on each O/S but all Posix based Unix systems have a common interface to this by use of the **_t** variable types.

## 3.2 Examples

The following program opens the file /etc/passwd and prints the contents to the console

Program 5: simple file read program

```
#include <stdio.h>

int main(void)
{
FILE *input_file;
int ip;
printf("attempting to open passwd file...");
if ((input_file = fopen("/etc/passwd", "rt"))  == NULL)
   {
   printf("File not found or unable to read\n");
   exit(1);
   }
printf("File Open OK\n\n");

while( (ip=fgetc(input_file))!=EOF)
                printf("%c",ip);

printf("\nEnd of File Closing ...");
fclose(input_file);
printf("File Closed\n");
}
```

The following program reads input from the console using getchar and writes it to the file writetest.txt

Program 6: simple file write program

```
#include <stdio.h>

int main(void)
{
FILE *output_file;
unsigned char ip;
printf("attempting to open new file...");
if ((output_file = fopen("writetest.txt", "wt"))  == NULL)
   {
   printf("File not found or unable to write\n");
   exit(1);
   }
printf("File Open OK\n\n");
printf("Every character typed will be entered into a file\n");
printf("Press Enter to write line to File \nPress the ESC key to exit\n");
do
{
  ip=getchar();
  fprintf(output_file,"%c",ip);
}while(ip !=27);
printf("\nEnd of File Closing ...");
fclose(output_file);
printf("File Closed\n");
}
```

### 3.2.1 Positioning a stream

There are several functions which allow the programmer to move the current position of the file stream. These are as follows

1. ftell and fseek. These functions assume that the file's position can be stored as a long integer.
2. fgetpos and fsetpos. These are ANSI C functions and may be used on different non-Unix file systems.

These positioning functions are define as below

Function Declaration: 10: File positioning functions

```
#include <stdio.h>
long ftell(FILE *fp);

    returns the current file position if OK -1L in error
```

```
int fseek(FILE *fp,long offset, int whence);

   returns 0 if OK nonzero on error

void rewind(FILE *fp);
```

---

The ftell function is passed the file pointer of the current stream and returns the current file position as a long integer.

The fseek function sets the file position indicator for the stream pointed to by stream. The new position, measured in bytes, is obtained by adding offset bytes to the position specified by whence. If whence is set to

SEEK_SET, SEEK_CUR, or SEEK_END, the offset is relative to the start of the file, the current position indicator, or end-of-file, respectively.

Finally rewind is used to reset the file position of *fp to the beginning of the file.

## 3.3 Formatted I/O

### 3.3.1 Formated outout

Formated output is implemented using the three printf functions. as printf and sprintf have already been covered in appendix A only fprint will be explained here.

---

Function Declaration: 11: fprintf

---

```
#include <stdio.h>
int fprintf(FILE *fp, const char *fmt, ...);

This function returns the the number of characters output
if OK or a negative value if output error.
```

---

The format function for fprintf is identical to that of printf for example to print the following variable to a file use

---

Program 7: A simple fprintf program snipet

---

```
#include <stdio.h>
int a=10;
char c='x';
char d[12]="hello world";
FILE *fp;
..... now open file etc
fprintf(fp,"this is the text %d %c %s \n",a,c,d);
```

---

### 3.3.2 Formated Input

Formated input is handled by the three scanf functions. As scanf has already been covered only the fscanf variant will be discussed.

Function Declaration: 12: fscanf

```
#include <stdio.h>
int fscanf(FILE *fp, const char *fmt, ...);

This function returns the number of input items assigned,
or EOF if input error or end of file before any conversion.
```

# Chapter 4

# Executing Processes as a Stream

Unix allows the programmer to communicate between processes in a number of ways. One of the simplest is the use of the *popen* and *pclose* functions which allow the execution of a process using streams similar to the opening of a file.

As *popen* returns a file handle the input or output to the process may then be accessed by using the usual file manipulation functions (fprintf, fread etc).

*popen* and *pclose* take the following format

Function Declaration: 13: popen and pclose

```
#include <stdio.h>
FILE *popen(const char *cmdstring, const char *type);

returns : file pointer if OK NULL on error

int pclose(FILE *fp);

returns termination status of cmdstring, or -1 on error
```

The function *popen* does a *fork* and *exec* to execute the *cmdstring*, and returns a standard I/O file pointer. If *type* is “r”, the file pointer is connected to the standard output of *cmdstring* as shown in figure 4.1

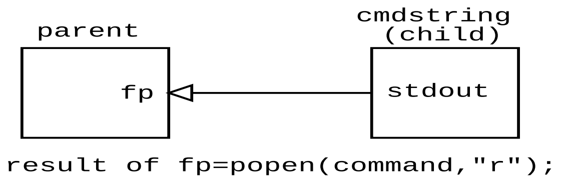

Figure 4.1: fopen in read mode

If *type* is “w”, the file pointer is connected to the standard input of *cmdstring* as shown in figure 4.2

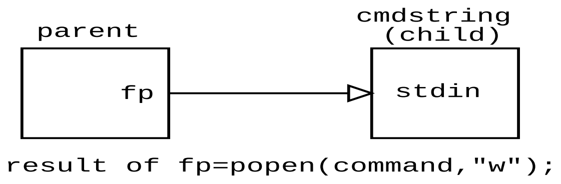

Figure 4.2: fopen in write mode

## 4.1 Reading from a process with popen

The following code illustrates the use of *popen* to read from a process

Program 8: A simple popen program

```
#include <stdio.h>

int main(void)
{
FILE *process; //file pointer for process
char buffer[1024]; //a buffer to read the data from popen
int i=0;

// use popen to create a stream to the ls program so we can read from it
process = popen("ls","r");
//check to see if the process opened correctly
if (!process)
        {
    printf("problem running command\n");
        return 0;
        }
// now read from the process fp into buffer
   while(fgets(buffer, 1024, process))
                printf("%d : %s",i++,buffer);

// finally we must close the process
 pclose(process);
 return 1;
}
```

In the above program the standard output of the *ls* program (usually the console) is redirected to the file pointer *process* this is then read using the *fread* function into the buffer.

This buffer may then be printed out using the *printf* function.

### 4.1.1 Writing to a process using popen

The following program demonstrates the writing to a process created using popen.

Program 9: writing to a stream opened by popen

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
int main(void)
{
// create a data string to be sorted
char data[]="the quick brown fox jumps over the lazy dog";
FILE *process;
int i;
int length;

// we need to know how long the string is so we can loop
length=strlen(data);
printf("data is %s\n",data);

// use popen to create a stream to the sort program so we can write to it
process = popen("sort","w");
//check to see if the process opened correctly
if (!process)
        {
    printf("problem running command\n");
        return 0;
        }
// now feed each of the characters of the data array into the sort process
// each letter must be separated by a new line
 for(i=0; i<length; i++)
        fprintf(process,"%c\n",data[i]);
 printf("done \n");
// finally we must close the process
 pclose(process);

return 1;
}
```

This program opens the *sort* program and reassigns the standard input to the file pointer *process.* This file pointer is then written to using the *fprintf* function passing each element of the array *data* separated by a new line character (\n). When this input has finished and the process is closed the output from sort will be written to the console.

### 4.1.2 But why use popen?

One of the most useful applications for *popen* is the re-direction of process I/O to a GUI based system.

For example an application such as an IDE for code development may wish to build an application from a Makefile, it make no sense for the developers of the IDE to re-write make, so with the use of *popen* make can be run and the output from make re-directed to the IDE

### 4.1.3 A GUI based popen example

The following example code uses the fltk gui toolkit to generate a small gui application to re-direct the output of any Unix command run to a 'browser' component.

Program 10: popen used in a gui

```
#include <stdio.h>
#include <FL/Fl.H>
#include <FL/Fl_Browser.H>
#include <FL/Fl_Input.H>
#include <FL/Fl_Window.H>
// declare gui components
Fl_Window *mainWindow;
Fl_Input  *cmd;
Fl_Browser *output;

// define functions
static void cb_Execute(Fl_Input*, void*);

// main program loop

int main(int argc, char **argv)
{
Fl_Window* w; // the main form widget
  {
   // create the main form widget
   Fl_Window* o = mainWindow = new Fl_Window(500, 400, "popen demo type in a command");
    {
        w=o;
        // now add a text input box
    Fl_Input* o = cmd = new Fl_Input(80, 15, 350, 25, "Type Command Press [enter] to execute");
        o->align(FL_ALIGN_TOP);
        // activate the callback cb_Execute when the enter key is pressed
        o->when(FL_WHEN_ENTER_KEY);
    o->callback((Fl_Callback*)cb_Execute);
    }
    // create the browser widget to add the text to
        {
    Fl_Browser * o= output = new Fl_Browser(5, 40, 490, 355);
        // set font and size
        o->textfont(FL_COURIER);
        o->textsize(12);
        // fg colour Yellow bg colour Blue
        o->color(136);
        o->textcolor(95);
    }
    o->end();
  }
  w->show(argc, argv);
  return Fl::run();
}

// callback execute every time return is pressed
static void cb_Execute(Fl_Input*, void*)
{
char buffer[1024]; // buffer for popen
char command[30]; //buffer for command
output->clear(); // first clear the list box
strcpy(command,cmd->value()); // now copy the command from the ip

FILE* f = popen(command,"r"); // now open a process to the command
if (!f) //check to see if it worked
        output->add("problem running command");
else // if it does loop until the process has finished
        {

        while(fgets(buffer, 1024, f))
       output->add(buffer);

    pclose(f); // now close the process
        }
}
```

As this program is a gui based program it needs lots of different libraries and header files. To add these to the standard compilation line in the console can become cumbersome so we use the make utility to build the application.

The following Makefile is used for the program above and is executed by typing make in the console.

Note that the # symbol is used for comments

```
# include directories
INCLUDE_DIR = -I/usr/include -I/usr/local/include
# g++ compiler flags
FLAGS = -g -Wall
#library directories
LIB_DIR = -L /usr/X11R6/lib

# object files used to link the program
OBJECTS = popengui.o

# X windows libs required
XLIBS = -lfltk -lX11 -lXext -lXmu -lXt -lXi -lSM -lICE

# flags for compilation
COMPILE = $(INCLUDE_DIR) $(FLAGS)
#flags for link
LINK = $(OBJECTS) $(INCLUDE_DIR) $(FLAGS) $(LIB_DIR) $(XLIBS)

# popen gui is made from all the objects
popengui : $(OBJECTS)
        g++ -o popengui  $(LINK)

#popengui.o is made from popen.c
popengui.o :  popengui.c
        g++ -c popengui.c $(COMPILE)

clean :
        rm -f *.o
        rm popengui
run :
        make
        popengui&
```

The above Makefile has 3 targets. typing **make** will build the program

typing **make clean** will run the clean target which will remove all **.o** files and the **poopengui** program.

Finally **make run** will call make and the run **popengui&** to run the program.

# Chapter 5

# Showing system Processes

When a process is created in the Unix operating system it is given a unique integer value know as the **pid**. This is done by the kernel and is the only way a process may be safely identified. To determine the **pid** of a process in the console the **ps** command is used as follows

```
[jmacey@localhost jmacey]$ ps
  PID TTY          TIME CMD
  576 ttyp1    00:00:00 bash
  577 ttyp1    00:00:00 ps
[jmacey@localhost jmacey]$
```

The list that follows the **ps** command shows the pid (process id) and other information (this is dependant upon the version of **ps** used use **man ps** to find out the flags for the solaris version).

Typing **ps** in the console will only show the current processes created in the opened console, however there are more processes than this owned by different users on the system. To see these the additional flags **-ef** must be used which will show all processes and all the information about the processes.

For example

```
[jmacey@localhost jmacey]$ ps -ef
UID        PID  PPID  C STIME TTY          TIME CMD
root         1     0  0 14:50 ?        00:00:04 init [3]
root         2     1  0 14:50 ?        00:00:00 [kflushd]
root         3     1  0 14:50 ?        00:00:00 [kpiod]
root         4     1  0 14:50 ?        00:00:00 [kswapd]
root         5     1  0 14:50 ?        00:00:00 [mdrecoveryd]
bin        216     1  0 14:51 ?        00:00:00 portmap
root       263     1  0 14:51 ?        00:00:00 syslogd -m 0
root       274     1  0 14:51 ?        00:00:00 klogd
daemon     288     1  0 14:51 ?        00:00:00 /usr/sbin/atd
root       301     1  0 14:51 ?        00:00:00 /sbin/cardmgr
root       315     1  0 14:51 ?        00:00:00 inetd
root       329     1  0 14:51 ?        00:00:00 lpd
```

## 5.1 Sorting processes

It is sometimes necessary to filter the list of processes to show only the processes belonging to a particular user. To do this the output of **ps -ef** may be fed into the **grep** utility as follows

```
ps -ef |grep $USER
```

Where jmacey is the user name of the process required.

## 5.2 Killing processes

To stop a process the **kill** command is used passing it the **pid** of the process to kill. This command may also be passed a signal to determine how the process is to be terminated. The most common signals used are as follows

```
-9 SIGKILL
-15 SIGTERM
```

Usually the -9 signal is used to kill a process in the following manner

```
kill -9 [pid]
```

# Chapter 6

# Creating processes in a program (fork)

### 6.0.1 The fork/exec process model

The Unix process management model is split into two distinct operations :

- The creation of a process
- The running of a new program

The creation of a new process is done using the fork() system call and a new program is run using the exec(l,lp,le,v,vp) family of system calls.

These are two separate functions which may be uses independently, a call to fork() will create a completely separate sub-process which will be exactly the same as the parent. Conversely calling one of the exec family of functions will terminate the currently running program and starts executing a new one in the context of the existing process.

The main reason this model is used is the simplicity of operation, when creating a new sub-process the current environment from the parent is used so the programmer need not setup a new environment to run the program. After the fork call the program may then use system calls to modify the environment to suit the child process and then use the exec functions to run the new process required.

Function Declaration: 14: fork

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

pid_t fork(void);

returns: 0 in child, process ID of child in parent, -1 on error.
```

The new process created by fork is called the child process. This function is called once but returns twice. The only difference in the returns is that the return value in the child is 0 while the return value in the parent is the process ID of the new child. The reason the child's process ID is returned to the parent is because a process can have more than one child, so there is no function that allows a process to obtain the process ID's of its children.

The reason that fork returns 0 to the child is because a process can have only have a single parent, so the child can always call *getppid* to obtain the process id of the parent.

Both the child and parent continue executing with the instruction that follows the call to fork. The child is a copy of the parent. For example the child gets a copy of the parent's data space, heap and stack. This means that when a child is initially created it is an exact copy of the parent. This is shown in figure 6.1.

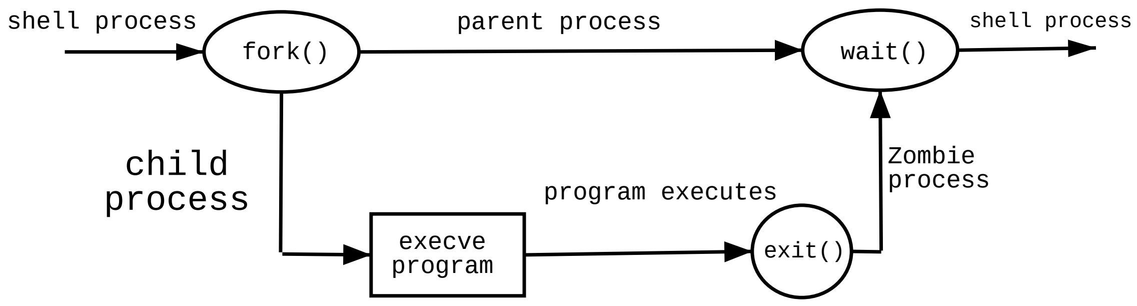

Figure 6.1: How the fork command works

It can be seen from the diagram that if the parent is killed before the child calls an exit() to exit normally a zombie process is created. This process is inherited by the init process which is part of the Unix system boot. These process must then be destroyed by the root user who owns the init function.

## 6.1 A simple fork example

The following program is split into two parts, each of which consists of a simple loop which prints a character to the console. The child part of the program prints the character C to the console whilst the parent prints a P to the console.

To demonstrate the fact that the child is an exact copy of the parent, a global variable called *endvalue* is declared in the parent which is used to set the end value of the loop, the child process will inherit this variable and use it in the loop.

The parent part of the program has a call to the *wait()* function which causes the parent to wait for the child to exit before the parent resumes. Therefor the expected output of the the program will be 600 C followed by 600 P.

Program 11: a simple fork example

```
#include <stdio.h>

int main(void)
{
// process id
int pid,i,endvalue;
// use fork to create a new process
endvalue=1000;
printf("calling fork()\n");
pid=fork();
// check to see if fork worked
if(pid <0)
 {
  printf("Fork failed\n");
  exit(0);
 }
else if (pid ==0)
 {
  // in child process
  for(i=0; i<endvalue; i++)
        {
        printf("C");
        fflush(stdout);
        }
 }

else
 {
 // parent process
  wait(NULL);
for(i=0; i<endvalue; i++)
        {
        printf("P");
        fflush(stdout);
        }

  printf("Child Complete");
  exit(0);
 }

}
```

## 6.2 The exec family of functions

The main use for the fork function is to allow for the execution of another process within a program, fork creates a child process which is an exact clone of the parent process if we wish to execute a new function after this has been done we use the exec family of functions.

These functions replace the current process (which after fork is a clone of the parent) with the new process called by exec. This will replace all of the text, data , stack segments and heap of the child process with that of the new process called.

There are six different exec functions which are described in the exec man pages

Function Declaration: 15: Exec family of functions

```
int execl( const char *path, const char *arg, ...);
int execlp( const char *file, const char *arg, ...);
int  execle( const char *path, const char *arg , ..., char * const envp[]);
int execv( const char *path, char *const argv[]);
int execvp( const char *file, char *const argv[]);
```

The first difference in these functions is that the first four take a pathname argument while the last two take a filename argument. When a filename argument is specified

- if filename contains a slash, it is taken as a pathname,
- otherwise the executable file is searched for in the directories specified by the PATH environment variable

### 6.2.1 A simple fork and exec example

The following program demonstrates how to use fork and exec to run a process and uses a small program called child to demonstrate how the parent and the child are running at the same time. The code for child.c is as follows

Program 12: child program to be run by parent program

```
#include <stdio.h>
int main(void)
{
while(1)
  printf("C");
}
```

Know the parent program which executes the child program

Program 13: program to run child program

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

int main(void)
{
pid_t pid;
int status;
if((pid =fork()) < 0)
        {
        // probably out of processes
        status =-1;
        }
else if (pid == 0)
        {
        // in child so we execute process
        // use the execl function to to run a shell an execute the child program
        execl("/bin/sh","sh","-c","child",(char *)0);
        }
while(1)
    printf("P");
printf("end of program");
}
```

When this program is executed the child program is spawned as a child process, and the main (parent) process continues now the console will print out both C for child and P for parent.

The execl function is used in the above example by calling the /bin/sh shell command and running sh -c which tells the shell (in this case sh the C shell) to start in command mode where the next argument is the command to be run which in the above example is the child program.

### 6.2.2 fork exec example 2

The following example executes a child process and waits for it to die before continuing the parent process. This is split into two sections, first is child2.c

Program 14: Child program to be run by parent program

```
#include <stdio.h>

int main(void)
{
int i=0;
for(i=0; i<100; i++)
  printf("c");
}
```

This program loops 100 times and prints out the character c to the console

The next program executes the above program (child2) and waits for it to finish before it continues

Program 15: Parent program to run child program

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

int main(void)
{
int c=0;
pid_t pid;
int status;
if((pid =fork()) < 0)
        {
        // probably out of processes
        status =-1;
        }
else if (pid == 0)
        {
        // in child so we execute process
         execl("/bin/sh","sh","-c","child2",(char *)0);

        }
else
```

```
              {
              // now wait for child to die
              while(waitpid(pid, &status,0) < 0)
                      {
                  printf("in wait pid \n");
                       if(errno !=EINTR)
                          {
                          status=-1;
                          break;
                              }

                      }
              }
      for(c=0; c<100; c++)
              printf("p");
      printf("\nend of program");
      }
```

---

If the above program is modified to run the original child program the child will run at the same time as the parent process but when the parent die so will the child.

# Chapter 7

# Processes and Threads

A thread, sometimes called a lightweight process, is a unit of cpu utilisation (i.e. a piece of code running on a cpu). Each thread has the following characteristics

- a threadID
- a program counter
- a register set
- a stack

However unlike a process each thread created by the parent shares the following resources

- code section
- data section
- operating system resources (i.e. opened files, signals etc)

As threads share the same memory and data caution must be used when writing multithreaded code as one thread may overwrite another threads data however threads do have advantages over the creation of new processes as it takes much less overhead to create and destroy a new thread as opposed to a process, also communication between threads is a lot quicker than inter process communication.

## 7.1 Benefits of Threads vs Processes

If implemented correctly threads have some advantages of (multi) processes, They take:

- Less time to create a new thread than a process, because the newly created thread uses the current process address space.
- Less time to terminate a thread than a process.
- Less time to switch between two threads within the same process, partly because the newly created thread uses the current process address space.
- Less communication overheads – communicating between the threads of one process is simple because the threads share everything: address space, in particular. So, data produced by one thread is immediately available to all the other threads.

### 7.1.1 Multithreading vs. Single threading

- Improve application responsiveness – Any program in which many activities are not dependent upon each other can be redesigned so that each activity is defined as a thread. For example, the user of a multithreaded GUI does not have to wait for one activity to complete before starting another.
- Use multiprocessors more efficiently – Typically, applications that express concurrency requirements with threads need not take into account the number of available processors. The performance of the application improves transparently with additional processors. Numerical algorithms and applications with a high degree of parallelism, such as matrix multiplications, can run much faster when implemented with threads on a multiprocessor.
- Improve program structure – Many programs are more efficiently structured as multiple independent or semi-independent units of execution instead of as a single, monolithic thread. Multithreaded programs can be more adaptive to variations in user demands than single threaded programs.
- Use fewer system resources – Programs that use two or more processes that access common data through shared memory are applying more than one thread of control. However, each process has a full address space and operating systems state. The cost of creating and maintaining this large amount of state information makes each process much more expensive than a thread in both time and space. In addition, the inherent separation between processes can require a major effort by the programmer to communicate between the threads in different processes, or to synchronize their actions.

## 7.2 Threads libraries

The interface to multithreading support is through a subroutine library, **libpthread** for POSIX threads, and **libthread** for Solaris threads. They both contain code for:

- creating and destroying threads
- passing messages and data between threads
- scheduling thread execution
- saving and restoring thread contexts

### 7.2.1 A simple pthread example

To create a default thread using the pthread library the **pthread_create** function is used as prototyped below

Function Declaration: 16: pthread_create

```
#include <pthread.h>

int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine, void *),void *arg);

returns 0 OK else error number.
```

The first argument to the functions is the id of the thread created and is returned to the user so a particular thread may be identified.

The **attr** parameter is used to set the attributes of the thread, If this value is set to NULL then the defaults are used as follows

- It is unbounded
- It is nondetached
- It has a a default stack and stack size
- It inherits the parent's priority

Next the routine to be created as a thread is passed to the function as well as the arguments. This is illustrated more clearly in the following example

### 7.2.2 Simple pthread program

Program 16: Simple pthread program

```
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>
// we must make the compiler aware that this program
// is to use threads so the thread safe libraries must be used
// to do this we define the _REENTERANT flag
#define _REENTRANT


//next we create a simple function for out thread
void *ThreadRoutine(int number)
{
while(1) // loop forever
{
printf("thread %d running\n",number);
sleep(number); // sleep for a time passed as a parameter
}

}

int main(void)
{
int t;

pthread_t tid[5]; // an array to keep track of the threads
// now loop through and create 4 threads passing t as the
// parameter
for (t=1; t<5; t++)
        pthread_create(&tid[t],NULL,(void *)ThreadRoutine,(int *)t);

// now the parent loops and sleeps for 10
while(1)
        {
        printf("parent running\n");
        sleep(10);
        }
exit(1);
}
```

To compile this program we need to include the **pthread** library as shown in the following command

```
gcc -Wall -g thread1.c -othread1 -lpthread
```

We can now see when running this program that the 1st thread sleeps less than the 4th thread so that the 1st thread appears to be running quicker.

You can also create a default attribute object with **pthread_attr_init()** function, and then use this attribute object to create a default thread.

### 7.2.3 Wait for Thread Termination

The **pthread_join()** function blocks the calling thread until the specified thread terminates. The specified thread must be in the current process and must not be detached. When status is not NULL, it points to a location that is set to the exit status of the terminated thread when **pthread_join()** returns successfully.

Multiple threads cannot wait for the same thread to terminate. If they try to, one thread returns successfully and the others fail with an error of ESRCH. After **pthread_join()** returns, any stack storage associated with the thread can be reclaimed by the application.

The **pthread_join()** routine takes two arguments, giving you some flexibility in its use. When you want the caller to wait until a specific thread terminates, supply that thread's ID as the first argument. If you are interested in the exit code of the defunct thread, supply the address of an area to receive it. Remember that pthread_join() works only for target threads that are nondetached. When there is no reason to synchronize with the termination of a particular thread, then that thread should be detached.

Think of a detached thread as being the thread you use in most instances and reserve nondetached threads for only those situations that require them.

Program 17: pthread_join example program

```
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>
// we must make the compiler aware that this program
// is to use threads so the thread safe libraries must be used
// to do this we define the _REENTERANT flag
#define _REENTRANT


//next we create a simple function for out thread
void *ThreadRoutine(int number)
{
printf("thread %d running\n",number);
sleep(number); // sleep for a time passed as a parameter
}

int main(void)
{
int t;

pthread_t tid; // an array to keep track of the threads
// now loop through and create 4 threads passing t as the
// parameter
pthread_create(&tid,NULL,(void *)ThreadRoutine,(int *)5);
// now the calling process waits for the thread to finish
pthread_join(tid,NULL);
printf("parent running\n");
exit(1);
}
```

## 7.3 Detaching threads

The function **pthread_detach()** is an alternative to **pthread_join()** to reclaim storage for a thread that is created with a detachstate attribute set to PTHREAD_CREATE_JOINABLE

This guarantees that the memory resources consumed by the thread will be freed immediately when the thread terminates. However, this prevents other threads from synchronizing on the termination of the thread using **pthread_join**.

It is prototyped as follows

Function Declaration: 17: pthread_detach

```
#include <pthread.h>
int pthread_detach(thread_t tid);

returns 0 OK else error
```

The **pthread_detach()** function is used to indicate to the implementation that storage for the thread **tid** can be reclaimed when the thread terminates. If **tid** has not terminated, **pthread_detach()** does not cause it to terminate. The effect of multiple **pthread_detach()** calls on the same target thread is unspecified.

**pthread_detach()** returns a zero when it completes successfully. Any other returned value indicates that an error occurred. When any of the following conditions are detected, **pthread_detach()** fails and returns the an error value.

A simple example of calling this function to detach a thread is shown below, where two threads are created. The first is detached so that it will run on it's own. This thread is then joined which should fail as it has been detached.

The second thread is joined and thus the parent waits for the second thread to finish until execution resumes.

Program 18: example of pthread_detach

```
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>
// we must make the compiler aware that this program
// is to use threads so the thread safe libraries must be used
// to do this we define the _REENTERANT flag
#define _REENTRANT


//next we create a simple function for out thread
```

```
void *ThreadRoutine(int number)
{
int i;
for (i=0; i<10; i++) //loop to give the thread something to do
{
printf("thread %d running %d\n",number,i);
sleep(number); // sleep for a time passed as a parameter
}
}

int main(void)
{
pthread_t tid1,tid2; // create 2 thread id's

//now create two threads
pthread_create(&tid1,NULL,(void *)ThreadRoutine,(int *)1);
pthread_create(&tid2,NULL,(void *)ThreadRoutine,(int *)2);

pthread_detach(tid1); //we will now detach thread 1

if(pthread_join(tid1,NULL)>0) // now try to join it
    printf("unable to join thread 1\n");

if(pthread_join(tid2,NULL)>0) // and now join thread 2
    printf("unable to join thread 2\n");

printf("parent finished\n");

exit(1);
}
```

## 7.4 Creating a Key for Thread-Specific Data

Single-threaded C programs have two basic classes of data: local data and global data. For multithreaded C programs a third class is added: thread-specific data (TSD). This is very much like global data, except that it is private to a thread.

Thread-specific data is maintained on a per-thread basis. TSD is the only way to define and refer to data that is private to a thread. Each thread-specific data item is associated with a key that is global to all threads in the process. Using the key, a thread can access a pointer (void *) that is maintained per-thread.

The function **pthread_keycreate()** is used to allocate a key that is used to identify thread-specific data in a process. The key is global to all threads in the process, and all threads initially have the value NULL associated with the key when it is created.

**pthread_keycreate()** is called once for each key before the key is used.There is no implicit synchronization. Once a key has been created, each thread can bind a value to the key. The values are specific to the thread and are maintained for each thread independently. The per-thread binding is deallocated when a thread terminates if the key was created with a destructor function. pthread_keycreate() is prototyped by:

Function Declaration: 18: pthread_key_create

```
#include <pthread.h>
int pthread_key_create(pthread_key_t *key, void (*destructor) (void *));

returns 0 OK else error;
```

When pthread_keycreate() returns successfully, the allocated key is stored in the location pointed to by key. The caller must ensure that the storage and access to this key are properly synchronized. An optional destructor function, destructor, can be used to free stale storage. When a key has a non-NULL destructor function and the thread has a non-NULL value associated with that key, the destructor function is called with the current associated value when the thread exits. The order in which the destructor functions are called is unspecified.

**pthread_keycreate()** returns zero after completing successfully. Any other returned value indicates that an error occurred. When any of the following conditions occur, pthread_keycreate() fails and returns an error value.

### 7.4.1 Delete the Thread-Specific Data Key

The function pthread_keydelete() is used to destroy an existing thread-specific data key. Any memory associated with the key can be freed because the key has been invalidated and will return an error if ever referenced. (There is no comparable function in Solaris threads.)

Function Declaration: 19: pthread_key_delete

```
#include <pthread.h>
int pthread_key_delete(pthread_key_t key);

returns 0 ok else error
```

Once a key has been deleted, any reference to it with the **pthread_setspecific()** or **pthread_getspecific()** call results in the EINVAL error.

It is the responsibility of the programmer to free any thread-specific resources before calling the delete function. This function does not invoke any of the destructors.

**pthread_keydelete()** returns zero after completing successfully. Any other returned value indicates that an error occurred. When the following condition occurs, **pthread_keycreate()** fails and returns the corresponding value.

### 7.4.2 Set the Thread-Specific Data Key

The function **pthread_setspecific()** is used to set the thread-specific binding to the specified thread-specific data key. It is prototyped by :

Function Declaration: 20: pthread_setspecific

```
#include <pthread.h>
int pthread_setspecific(pthread_key_t key, const void *value);

returns 0 OK else error
```

**pthread_setspecific()** returns zero after completing successfully. Any other returned value indicates that an error occurred. When any of the following conditions occur, **pthread_setspecific()** fails and returns an error value.

**Note: pthread_setspecific()** does not free its storage. If a new binding is set, the existing binding must be freed; otherwise, a memory leak can occur.

### 7.4.3 Get the Thread-Specific Data Key

Use **pthread_getspecific()** to get the calling thread's binding for key, and store it in the location pointed to by value. This function is prototyped by:

Function Declaration: 21: pthread_getspecific

```
#include <pthread.h>
int pthread_getspecific(pthread_key_t key);

returns 0 OK else error
```

## 7.5 An example program

The following program demonstrates the use of thread specific data by creating 10 threads each of which reference a global variable called buffer_key. This variable is allocated memory in the thread using malloc and each thread puts a different character string to the thread.

Program 19: using thread specific data

```
#include <unistd.h>
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
// we must make the compiler aware that this program
// is to use threads so the thread safe libraries must be used
// to do this we define the _REENTERANT flag
#define _REENTRANT

/*function prototypes for the re-use of buffer_key */

static void buffer_destroy(void * buf);
void buffer_alloc(void);
char * get_buffer(void);
static void buffer_destroy(void * buf);
static void buffer_key_alloc();

/* Key for the thread-specific buffer */
static pthread_key_t buffer_key;

/* Allocate the thread-specific buffer */
```

```
void buffer_alloc(void)
{
 printf("calling malloc ..");
 //when this is called the buffer key is malloced with 50 bytes
 pthread_setspecific(buffer_key,(char *) malloc(50));
}

/* Return the thread-specific buffer */
char * get_buffer(void)
{
// this function returns a pointer to the TSD buffer
return (char *) pthread_getspecific(buffer_key);
}

/* Allocate the key */
static void buffer_key_alloc()
{
//This function allocates the TSD key for the buffer as
// well as the destructor function
pthread_key_create(&buffer_key, buffer_destroy);
}

/* Free the thread-specific buffer */
static void buffer_destroy(void * buf)
{
 printf("free buffer\n");
 free(buf);
}
//next we create a simple function for out thread

void *ThreadRoutine(char character)
{
int i;
// this is the local TSD reference we use in the thread routine
char *threadBuffer;
// now init the TSD buffer
buffer_key_alloc();
buffer_alloc();

//now we get a pointer to that buffer so we can use it
threadBuffer=get_buffer();
printf("filling buffer with %c\n",character);

for (i=0; i<50; i++) //now stuff the character passed into the buffer
  threadBuffer[i]=character;
printf("thread sleeping\n");
//now sleep
sleep(2);
printf("thread %c string = %s\n",character,threadBuffer);
//finally signal that the thread is to exit which
//will call the destructor function
pthread_exit(NULL);
}


int main(void)
{
pthread_t tid1[10]; // create an array thread id’s
int i;
//now create 10 threads
for(i=0; i<10; i++)
        pthread_create(&tid1[i],NULL,(void *)ThreadRoutine,(char *)i+57);
printf("joining final thread\n");
//now we join the last thread so the parent waits till it has finished
pthread_join(tid1[9],NULL);

printf("parent finished\n");
exit(1);
}
```

## 7.6 Getting Thread Identifiers

The function **pthread_self()** can be called to return the ID of the calling thread. It is prototyped by:

Function Declaration: 22: pthread_self

```
#include <pthreead.h>
pthread_t pthread_self(void);

returns thread id
```

It is use is very straightforward:

Program 20: pthread_self usage

```
#include <pthread.h>
pthread_t tid;
tid = pthread_self();
```

### 7.6.1 Comparing Thread IDs

The function **pthread_equal()** can be called to compare the thread identification numbers of two threads. It is prototyped by:

Function Declaration: 23: pthred_equal

```
#include <pthread.h>
int pthread_equal(pthread_t tid1, pthread_t tid2);
```

It is use is straightforward to use

Program 21: pthread_equal usage

```
#include <pthread.h>
pthread_t tid1, tid2;
int ret;
ret = pthread_equal(tid1, tid2);
```

As with other comparison functions, **pthread_equal()** returns a non-zero value when tid1 and tid2 are equal; otherwise, zero is returned. When either tid1 or tid2 is an invalid thread identification number, the result is unpredictable.

## 7.7 Initializing Threads

Use **pthread_once()** to call an initialization routine the first time **pthread_once()** is called Subsequent calls to have no effect. The prototype of this function is:

Function Declaration: 24: pthread_once

```
#include <pthread.h>
int pthread_once(pthread_once_t *once_control,void (*init_routine)(void));

always returns 0
```

## 7.8 Yield Thread Execution

The function **sched_yield()** causes the current thread to yield its execution in favor of another thread with the same or greater priority. It is prototyped by:

Function Declaration: 25: sched_yield

```
#include <sched.h>
int sched_yield(void);

returns 0 OK; -1 error
```

**sched_yield()** returns zero after completing successfully. Otherwise -1 is returned and errno is set to indicate the error condition.

### 7.8.1 Set the Thread Priority

Use **pthread_setschedparam()** to modify the priority of an existing thread. This function has no effect on scheduling policy. It is prototyped as follows:

Function Declaration: 26: pthread_setschedparam

```
#include <pthread.h>
int pthread_setschedparam(pthread_t tid, int policy, const struct sched_param *param);

returns 0 OK; else error
```

and is used as follows:

Program 22: pthread_setschedparam example

```
#include <pthread.h>
pthread_t tid;
int ret;
struct sched_param param;
int priority;
/* sched_priority will be the priority of the thread */
sched_param.sched_priority = priority;
/* only supported policy, others will result in ENOTSUP */
policy = SCHED_OTHER;
/* scheduling parameters of target thread */
ret = pthread_setschedparam(tid, policy, &param);
```

**pthread_setschedparam()** returns zero after completing successfully. Any other returned value indicates that an error occurred. When either of the following conditions occurs, the **pthread_setschedparam()** function fails and returns an error value.

### 7.8.2 Get the Thread Priority

To get the priority of a theread the following function is used

Function Declaration: 27: pthread_getschedparam

```
#include <pthread.h>
int pthread_getschedparam(pthread_t tid, int policy, struct schedparam *param);

returns 0 OK else error;
```

An example call of this function is:

Program 23: pthread_getschedparam example

```
#include <pthread.h>
pthread_t tid;
sched_param param;
int priority;
int policy;
int ret;
/* scheduling parameters of target thread */
ret = pthread_getschedparam (tid, &policy, &param);
/* sched_priority contains the priority of the thread */
priority = param.sched_priority;
```

**pthread_getschedparam()** returns zero after completing successfully. Any other returned value indicates that an error occurred. When the following condition occurs, the function fails and returns the error value set.

### 7.8.3 Send a Signal to a Thread

Signal may be sent to threads is a similar fashion to those for process as follows:

Program 24: Sending a signal to a thread

```
#include <pthread.h>
#include <signal.h>
int sig;
pthread_t tid;
int ret;
ret = pthread_kill(tid, sig);
```

**pthread_kill()** sends the signal **sig** to the thread specified by **tid**. tid must be a thread within the same process as the calling thread. The sig argument must be a valid signal of the same type defined for signal() in signal.h

When sig is zero, error checking is performed but no signal is actually sent. This can be used to check the validity of tid.

This function returns zero after completing successfully. Any other returned value indicates that an error occurred. When either of the following conditions occurs, **pthread_kill()** fails and returns an error value.

### 7.8.4 The Signal Mask of the Calling Thread

The function **pthread_sigmask()** may be used to change or examine the signal mask of the calling thread. It is prototyped as follows:

Function Declaration: 28: pthread_sigmask

```
#include <pthread.h>
#inclue <signal.h>
int pthread_sigmask(int how, const sigset_t *new, sigset_t *old);

returns 0 OK else error
```

**how** determines how the signal set is changed. It can have one of the following values :

- SIG_SETMASK Replace the current signal mask with new, where new indicates the new signal mask.
- SIG_BLOCK Add new to the current signal mask, where new indicates the set of signals to block.
- SIG_UNBLOCK Delete new from the current signal mask, where new indicates the set of signals to unblock.

When the value of new is NULL, the value of how is not significant and the signal mask of the thread is unchanged. So, to inquire about currently blocked signals, assign a NULL value to the new argument. The old variable points to the space where the previous signal mask is stored, unless it is NULL.

**pthread_sigmask()** returns a zero when it completes successfully. Any other returned value indicates that an error occurred. When the following condition occurs, pthread_sigmask() fails and returns an errro value.

Example uses of this function include:

Program 25: pthread_sigmask example

```
#include <pthread.h>
#include <signal.h>
int ret;
sigset_t old, new;
ret = pthread_sigmask(SIG_SETMASK, &new, &old); /* set new mask */
ret = pthread_sigmask(SIG_BLOCK, &new, &old); /* blocking mask */
ret = pthread_sigmask(SIG_UNBLOCK, &new, &old); /* unblocking */
```

### 7.8.5 Terminate a Thread

A thread can terminate its execution in the following ways:

- By returning from its first (outermost) procedure, the threads start routine; see **pthread_create()**
- By calling **pthread_exit()**, supplying an exit status
- By termination with POSIX cancel functions; see **pthread_cancel()**

The **pthread_exit()** is used terminate a thread in a similar fashion the exit() for a process :

Program 26: pthread_exit example

```
#include <pthread.h>
int status;
pthread_exit(&status); /* exit with status */
```

The pthread_exit() function terminates the calling thread. All thread-specific data bindings are released. If the calling thread is not detached, then the thread's ID and the exit status specified by status are retained until the thread is waited for (blocked). Otherwise, status is ignored and the thread's ID can be reclaimed immediately.

The **pthread_cancel()** function is used to cancel a thread

Function Declaration: 29: pthread_cancel

```
#include <pthread.h>
int pthread_cancel(pthread_t thread);

returns 0 OK else error
```

and is used as follows :

Program 27: pthread_cancel example

```
#include <pthread.h>
pthread_t thread;
int ret;
ret = pthread_cancel(thread);
```

How the cancellation request is treated depends on the state of the target thread. Two functions,

**pthread_setcancelstate()** and **pthread_setcanceltype()** (see man pages for further information on these functions), determine that state.

**pthread_cancel()** returns zero after completing successfully. Any other returned value indicates that an error occurred. When the following condition occurs, the function fails and returns an error value.

# Part II

# Inter-process Communication

Chapter 8

# Introduction to Inter-process Communication (IPC)

Unix inter-process communication is a big area which has evolved with the different strains of Unix into different mechanisms to communicate between processes. The table 8.1 shows the different IPC methods available to the different versions of Unix.

| IPC Type | POSIX1 | XPG3 | V7 | SVR2 | SVR3.2 | SVR4 | 4.3BSD | 4.3+BSD | Linux |
|---|---|---|---|---|---|---|---|---|---|
| **Pipes** | * | * | * | * | * | * | * | * | * |
| **FIFO's** | * | * |  | * | * | * | * | * | * |
| **stream pipes** |  |  |  |  | * | * | * | * | * |
| **Named Stream pipes** |  |  |  |  | * | * | * | * | * |
| **message queues** |  | * |  | * | * | * |  |  | * |
| **semaphores** |  | * |  | * | * | * |  |  | * |
| **shared memory** |  | * |  | * | * | * |  |  | * |
| **sockets** |  |  |  |  |  | * | * | * | * |
| **streams** |  |  |  |  | * | * |  |  | * |

Table 8.1: Different IPC methods

As table 8.1shows the only method of IPC available to all versions of Unix are half duplex pipes. The first seven entries in the table are usually used for IPC between processes on the same host while the last two are used for communicating between different hosts.

## 8.1 Pipes

Pipes are the oldest form of Unix IPC and are available to all versions of Unix but they have two limitations

1. They are half-duplex :- data flows in one direction.
2. The can be used only between processes that have a common ancestor. Normally a pipe is created by a process, the process calls fork, and the pipe is used between the parent and the child.

A pipe is created by using the pipe function as follows

Function Declaration: 30: pipe

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

pipe returns 0 if OK -1 on error
```

Two file descriptors are returned through the filedes argument. filedes[0] is open for reading and filedes[1] is open for writing. The output of filedes[1] is the input for filedes[0].

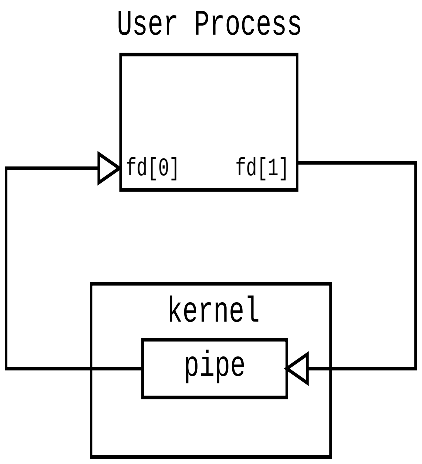

Figure 8.1: A simple pipe example

As the figure 8.1shows a pipe has been created between the same process which is not really useful as a single process can easily talk to itself. A more useful method of using a pipe is when it is used between two processes after a fork has been called to communicate between a parent and a child as shown in figure 8.2.

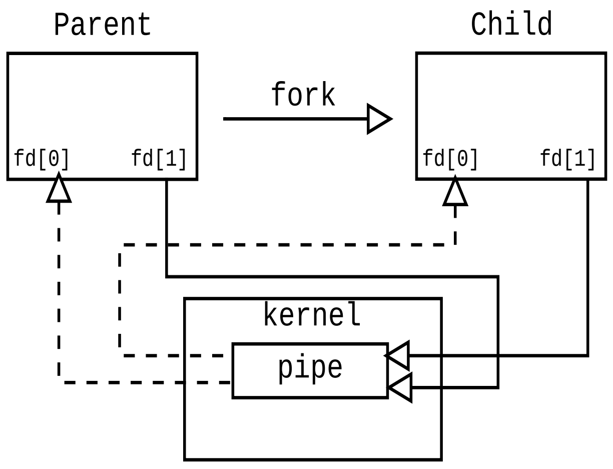

Figure 8.2: Complex pipe example

What happens after the fork depends on which direction of data flow is required. If data is to be sent from the parent to the child the parent closes the read end of the pipe (fd[0]) and the child closes the write end (fd[1]). For communication from child to parent the opposite is done.

The following program demonstrates the creation of a pipe to send data from parent to a child.

Program 28: Simple pipe program

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

int main(void)
{
int n,fd[2];
pid_t pid;
char line[50];

if ( pipe(fd) <0)
        {
        printf("pipe error\n");
        exit(1);
        }
if( (pid =fork()) <0)
    {
        printf("fork error\n");
        exit(1);
        }
else if(pid >0)
        {
        //parent
        close(fd[0]);
        printf("Parent writing to child\n");
        write(fd[1],"Hello child process\n",20);
        }
else
        {
        // child
        close(fd[1]);
        n=read(fd[0],line,21);
        printf("Child n=%d String = %s\n",n,line);
        write(STDOUT_FILENO,line,n);
        }
exit(0);
}
```

This program first creates a pipe and grabs the file descriptor for each end of the pipe in the array fd. next a child is created using the fork command and then the parent sends some data to the child using the write command. The child then uses the read command to read the data sent from the parent.

### 8.1.1 popen

This method of talking between processes is nicely wrapped up in the popen and pclose functions. These two functions do exactly the same as above only the file descriptors and the fork are all part of the function call;

## 8.2 FIFO's

FIFO's are sometimes called named pipes, as shown previously pipes can only be used to communicate between related processes with a common ancestor (i.e. parent -> fork -> child) however FIFO's allow unrelated processes to communicate with each other.

This is implemented by creating a FIFO as a type of file, using the mkfifo function as shown below

Function Declaration: 31: mkfifo

```
#include <sys/types.h>
#include <sys/stat.h>

int mkfifo(const char *pathname, mode_t mode);

returns 0 if OK, -1 on error
```

The mode argument for the mkfifo function is the same as that for the open function and the access permissions follow the same structure as that for any Unix file.

To communicate with the FIFO the open command is used to open the FIFO and the standard I/O functions close, read, write, unlink etc are used to communicate with it.

### 8.2.1 FIFO Uses

There are two ways to use a FIFO these are

1. Used by shell commands to pass data from one shell pipeline to another without creating intermediate temporary files.
2. Used in a client server application to pass data between a client and a server.

### 8.2.2 Using a FIFO in a shell

FIFO's can be used to duplicate output streams in a series of shell commands. The following example demonstrates how this may be done (however it is a theoretical example and will not work in the console).

This example is shown in figure 8.3where a file needs to be processed by two programs

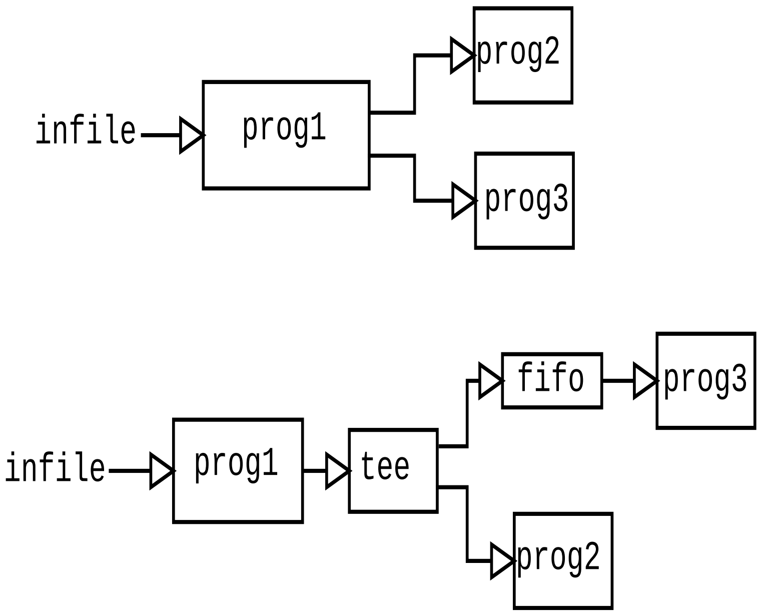

Figure 8.3: Using a FIFO in a shell

The top part of figure 8.3 shows how the file infile is to be processed and the bottom part shows how this could be implemented using shell commands

First a FIFO is created using the command mkfifo, after that prog3 run as a background process with its input directed from the fifo.

After this the tee utility is used to split the output of prog1 into two (this can be thought of as a plumbing tee piece which has one input and two outputs).

Finally prog1 is started with it's output fed into tee which then feeds to the fifo and prog2 so the file may be proceed by both programs.

This is done in the console as follows

```
# mkfifo fifo1
# prog3 < fifo1&
# prog1 < infile |tee fifo1 | prog2
```

As mentioned previously this is only an example and will not work however the mkfifo fifo1 will create the fifo, it is also interesting to see what the default file permissions are as shown below with the output of ls -al fifo1

```
prw-r--r--   1 jmacey   users      0 Jan 11 13:27 fifo1
```

### 8.2.3 Client Server FIFO operations

The following programs show a simple client / server system using a FIFO (named pipe) the first program FifoServer.c is shown below

Program 29: Simple fifo Server program

```
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <stdio.h>
// First lets define the name for the FIFO
#define FIFONAME "myfifo"
//and next RW RW RW permisons to be used by the chmod function
#define FIFOPERM S_IWUSR | S_IRUSR | S_IWGRP | S_IRGRP | S_IWOTH | S_IROTH

int main(void)
{
int fd;
int exitflag=1,count;
char buffer[50];
printf("Starting FiFo Server\n");
//first we create a FIFO with no permissions
if(mkfifo(FIFONAME,0)!=0)
        {
        printf("error creating FIFO\n");
        exit(1);
        }

// next we change the file permissons
if(chmod (FIFONAME,FIFOPERM) == -1)
        {
        printf("error in chmod \n");
        exit(1);
        }

// finally we open the FIFO as read only
if ( (fd = open(FIFONAME,O_RDONLY)) <0)
        {
        printf("error opening FIFO\n");
        exit(1);
        }

printf("FIFO open Waiting\n");
// Now we wait for some Input
do
{
//the read command constantly polls the fifo for data but
//will only respon if count is > 0
count=read(fd,buffer,50);

if(count >0)
        {
        // now we print out the responce
        printf("From Client : %s\n",buffer);
        if(buffer[0] == '!')
                exitflag=0;
        }
}while(exitflag!=0);

//finally we close the fifo

close(fd);
// and finally remove the fifo from the system
remove(FIFONAME);
return 0;
}
```

### 8.2.4 Testing FIFO server

As the FIFO is effectively a file on the system it is possible to test the server without the client program. This is done in the following way

1. Compile the Server using *gcc -Wall -g FifoServer.c -o FifoServer*
2. Now open two consoles
3. In the first console run the server by typing *FifoServer*
4. In the second console type *echo "hello fifo" > myfifo*
5. The server window will respond with *From Client : hello fifo* this will be followed by some garbage characters which is due to the server program reading a buffer of 50.
6. Now try cat FifoServer.c >myfifo which will send the C source file to the server
7. Finally echo ! > myfifo which will terminate the server and remove the fifo

### 8.2.5 Client Program

The client program below reads the input from stdin and sends it to the server via a FIFO when the return key is pressed. A string starting with an ! will cause the client and the server to quit

Program 30: fifo server program

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
// define the name of the fifo to use
#define FIFONAME "myfifo"

int main(void)
{
int fd;
char buffer[50];

printf("Opening FiFo to Server\n");
//open the fifo to Write
if ( (fd = open(FIFONAME,O_WRONLY)) <0)
        {
        printf("error opening FIFO is server running?\n");
        exit(1);
        }

printf("FIFO open Waiting\n");
do
{
printf("Enter a string Ret to send ! Enter to quit\n");
//now get some ip into the buffer
fgets(buffer,50,stdin) ;
// NULL terminate the string
buffer[strlen(buffer)-1]='\0';
//and write it to the fifo
write(fd,buffer,sizeof(buffer));


}while(buffer[0] !='!');
//finally close the fifo
close(fd);
return 0;
}
```

---

### 8.2.6 Testing the Client / Server system

1. Compile the Client using *gcc -Wall -g FifoClient.c -o FifoClient*
2. Now open three consoles
3. In the first console run the server by typing *FifoServer*
4. In the second console run the *FifoClient*
5. Now type in the client window and the data will be sent to the server
6. run another client in the third server window and type in the messages will also be sent to the server.

# Chapter 9

# Semaphores

Semaphores are a method for allowing processes to lock the entry to data or other processes and may be considered similar to using traffic lights to stop the flow of a process.

Any process creating a semaphore and accessing a resource is know to be in a ’critical section’. Once one process is within the critical section no other process is allowed to enter this section so the semaphore is used to signal that the resource is in use and may not be accessed until the process leaves the ’critical section’ and removes the semaphore lock.

In a computer the semaphore appears as a simple integer. A process (or thread) waits for permission to proceed by waiting for the integer to become 0. The process if it proceeds sends a signal which increments the integer to 1. When it is finished, the process changes the semaphores value by subtracting one from it.

Semaphores let processes query or alter status information and they are often used to monitor and control the availability of system resources such as shared memory segments.

To obtain a shared resource a process needs to do the following:

1. Test the semaphore that controls the resource.
2. If the value of the semaphore is positive the process can use the resource. The process decrements the semaphore value by 1 indicating that it has used one unit of of the resource.
3. If the value of the semaphore is 0, processes goes to sleep until the semaphore value is greater than 0. When the process wakes up it returns to step 1.

When a process is done with a shared resource that is controlled by a semaphore, the semaphore value is incremented by 1. If any other processes are asleep, waiting for the semaphore they are awakened.

To implement semaphores correctly, the test of a semaphores value and the decrementing of this value must be an atomic operation. For this reason semaphores are usually implemented in the kernel.

When using semaphores the first function to call is semget which obtains a semaphore id. this is prototyped as shown below

Function Declaration: 32: semaphore

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semget(key_t key, int nsems, int flag);
```

**returns semaphore ID if OK, -1 on error**

Where *key* is the private key to reference the semaphore, *nsems* is the number of semaphores available to the semaphore set. This value is set to the number of semaphores required if we are creating a semaphore set, or 0 (don't care state) if we are attaching to an existing semaphore set.

The *flag* element of the structure define the access permissions to the semaphore as well as the method of creation.

The function *semop* atomically performs an array of operations on a semaphore set.

Function Declaration: 33: semop

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semop(int semid,struct sembuf semoparray[],size_t nops);

returns 0 if OK, -1 on error
```

*semoparray* is a pointer to an array of semaphore operations

Function Declaration: 34: sembuf

```
struct sembuf
  {
  ushort sem_num; //member # in set (0 .1 nsems-1)
  short  sem_op;  //operation negative 0 or positive
  short  sem_flg; // IPC_NOWAIT, SEM_UNDO
};
nops specifies the number of operations (elements) in the array.
```

The operation on each member of the set is specified by the corresponding value; this value can be negative, 0 or positive.

The operation on a semaphore is as follows

1. The easiest case is when *sem_op* is positive. This corresponds to the returning of resources by the process. The value of *sem_op* is added to the semaphores value. If the undo flag is specified, *sem_op* is also subtracted from the semaphores adjustment value for the process.

2. If *sem_op* is negative this means we want to obtain resources that the semaphore controls.
   If the semaphore value is greater than or equal to the absolute value of *sem_op* (the resources are available) the absolute value of *sem_op* is subtracted from the semaphore and this guarantees the resulting value for the semaphore is greater or equal to 0. If the undo flag is specified the absolute value of *sem_op* is also added to the semaphores adjustment value for this process.
   If the semaphores value is less than the absolute value of *sem_op* (resource not available)

   (a) if IPC_NOWAIT is specified, return is ,made with an error of EAGAIN;
   (b) if IPC_NOWAIT is not specified, the semncnt value of the semaphore is incremented (as the process is going to sleep) and the calling process is suspended until one of the following occurs
      i. The semaphores value becomes greater than or equal to the absolute vale of *sem_op* (i.e. the resources have been released) and the *sem_op* value is decremented.
      ii. The semaphore is removed from the system and an ERMID is created
      iii. A signal is caught by the process and the EINTR error is created.

3. If *sem_op* is 0 this means that we wait until the semaphores value is 0 If the semaphores value is currently 0 the function returns immediately; If the semaphores value is non zero :
   (a) if IPC_NOWAIT is specified return with error EAGAIN;
   (b) if IPC_NOWAIT is not specified, the *semzcnt* value for the semaphore is incremented (as we are going to sleep) and the calling process is suspended until on of the following occurs
      i. The semaphores value becomes 0. (done waiting)
      ii. The semaphore is removed from the system and an ERMID is created
      iii. A signal is caught by the process and the EINTR error is created.

The atomicity of the semop is because it either does all the operations in the array or it does none of them.

## 9.1 A semaphore example

Program 31: Semaphore example program

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

int main(void)
{
int i,j;
int pid; //p[id of process
int semid; //return value from sem get
key_t key =1234; // key to be passed to semget
int semflg = IPC_CREAT |0666; //creation flags for semaphore set
int nsems =1; // number of semaphores
int nsops; //number of operations to do
struct sembuf sops[2]; //array of operations to perform

printf("init semaphore\n");
//setup a semaphore
if ((semid = semget(key,nsems, semflg)) == -1)
    {
     perror("Semget failed ");
     exit(1);
    }
// now get a child process
if((pid = fork()) < 0)
    {
     perror("fork failed ");
     exit(1);
    }
// child code
if(pid == 0)
    {
     i=0;
     printf("in child\n");
```

```
          while(i <3)
          {
           nsops=2;
           sops[0].sem_num=0; // use 1 track
           sops[0].sem_op =0; //wait for flag to become 0
           sops[0].sem_flg = SEM_UNDO; //take off ASYNC
           sops[1].sem_num=0;
           sops[1].sem_op =1; // take control of track
           sops[1].sem_flg = SEM_UNDO | IPC_NOWAIT; //take off sem
           // now report the semaphore state
           if((j= semop(semid,sops,nsops))==-1)
             {
              perror("Child  : semop failed");
             }
           else
             {
              printf("Child  : semop returned %d\n",j);
              printf("Child  : process taking control of track %d\n",semid);
              sleep(5); //do nothing i.e. simulate some process
              nsops =1; //
              // now give up the semaphore by setting op to -1
              sops[0].sem_num=0;
              sops[0].sem_op = -1; //by setting op to -1
              sops[0].sem_flg = SEM_UNDO |IPC_NOWAIT;
              // and finally report/set the semaphore state
              if((j=semop(semid,sops,nsops)) == -1)
                 {
                  perror("Child  : semop failed");
                 }
              else
                 {
                  printf("Child  : process giving up\n");
                  sleep(5);
                 }
              }
           ++i;

           }
     }
     else //parent this has the same operation as the child
        {
         i=0;
         printf("in parent\n");
         while(i <3)
           {
            nsops=2;
            sops[0].sem_num=0;
            sops[0].sem_op =0;
            sops[0].sem_flg = SEM_UNDO;
            sops[1].sem_num=0;
            sops[1].sem_op =1;
            sops[1].sem_flg = SEM_UNDO |IPC_NOWAIT;

            if((j= semop(semid,sops,nsops))==-1)
              {
               perror("Parent : semop failed");
              }
            else
              {
               printf("Parent : semop returned %d\n",j);
               printf("Parent : process taking control of track %d\n",semid);
               sleep(5);
               nsops =1;
               sops[0].sem_num=0;
               sops[0].sem_op = -1;
               sops[0].sem_flg = SEM_UNDO |IPC_NOWAIT;
               if((j=semop(semid,sops,nsops)) == -1)
                  {
                   perror("Parent : semop failed");
                  }
               else
                  {
                   printf("Parent : process giving up\n");
                   sleep(5);
                  }
              }
           ++i;
          }

       }
```

```
printf("Done\n");
return 0;
}
```

This program creates a child process and uses a single track semaphore to allow only the parent or the child be running at one time. The output of this program is shown below

```
init semaphore
in parent
Parent : semop returned 0
Parent : process taking control of track 0
in child
Parent : process giving up
Child  : semop returned 0
Child  : process taking control of track 0
Child  : process giving up
Parent : semop returned 0
Parent : process taking control of track 0
Parent : process giving up
Child  : semop returned 0
Child  : process taking control of track 0
Child  : process giving up
Parent : semop returned 0
Parent : process taking control of track 0
Parent : process giving up
Child  : semop returned 0
Child  : process taking control of track 0
Child  : process giving up
Done
Done
```

# Chapter 10

# Shared Memory

Using pipes two processes can communicate with each other by passing messages from one process to the other via the kernel. The use of shared memory allows two processes to communicate by using a given region of memory and reading or writing messagesto the shared memory area. This is the fastest form of IPC as no data needs to be copied between client and server. An outline of these two methods are shown in figure 10.1.

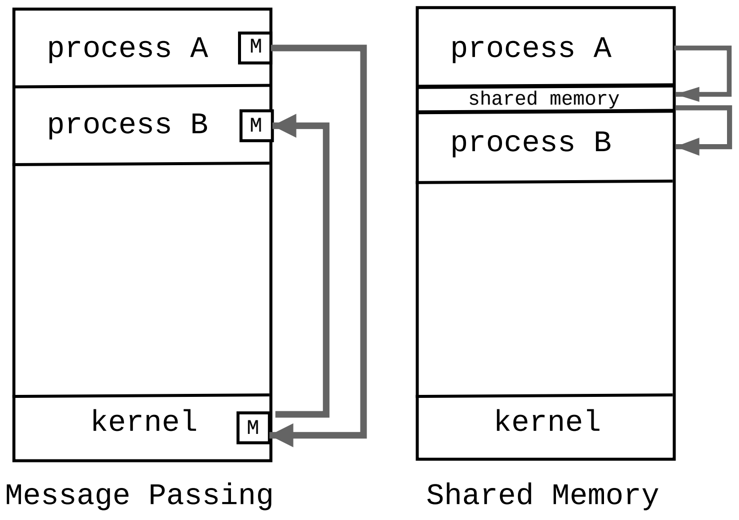

Figure 10.1: Pipes vs Shared Memory for IPC

The main difficulty of using shared memory is the synchronisation of processes so they can share the same region of memory without corrupting the system integrity. For example if a server is placing data into a section of shared memory the client should not try to access it until the server has finished. To stop this semaphores or record locking are used.

A process creates a shared memory segment using shmget(). The original owner of the shared memory segment can assign ownership to another user with the shmctl() function, another process which has the correct permissions can also perform operations on the shared memory segment by using the shmctl() function.

The first function called when using shared memory is usually the shmget function as detailed below

Function Declaration: 35: shmget

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int shmget(key_t key, int size, int flag)

returns shared memory ID if OK, -1 on error
```

The *key* argument is used to uniquely identify the memory area and is the same as the *key* argument has in the semaphore.

The *size* argument is the size in bytes of the requested shared memory and the *shmflg* argument specifies the initial access permissions an control creation permissions.

When the call succeeds it returns the shared memory segment ID. This call is also used to get the ID of the existing shared segment from another process.

## 10.1 Shared memory Control

To alter the permissions and other characteristics of an allocated shared memory segment the *shmctl* function is used

Function Declaration: 36: shmctl

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int shmctl(int shmid, int cmd,struct shmid_ds *buf)

returns 0 if OK, -1 on error
```

The process must have the effective *shmid* of owner, creator or superuser to perform this command. The *cmd* argument is set as one of the commands in table 10.1.

| **Command** | **Action** |
|---|---|
| SHM_LOCK | Lock the specified shared memory segment in memory |
| SHM_UNLOCK | Unlock the shared memory segment |
| IPC_STAT | return status information of memory into buf |
| IPC_SET | Set the effective user/group id and permissions |
| IPC_RMID | remove shared memory segment |

Table 10.1: shared memory control definitions

### 10.1.1 Attaching and detaching a shared memory segment

Once a shared memory segment has been created a process attaches to it's address space by calling *shmat* as prototyped below

Function Declaration: 37: shmat

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

void *shmat(int shmid,void *addr,int flag)

returns pointer to shared memory seg if OK, -1 on error
```

The address in the calling process at which the segment is attached depends on the *addr* argument and whether the SHM_RND bit is specified in *flag*.

1. if *addr* is 0 the segment is attached at the first available address selected by the kernel. This is the recommended technique.
2. If *addr* is non zero and SHM_RND is not specified, the segment is attached at the address given by *addr*.
3. If *addr* is non zero and SHM_RND is specified, the segment is attached at the address given by (*addr* - (*addr* modulus SHMLBA)). The SHM_RND command stands for "round". SHMLBA stands for "low boundary address multiple" and is always a power of 2. What the arithmetic does is round the address down to the next multiple of SHMLBA

If the SHM_RDONLY bit is set in the flag argument the memory is attached as read only. Otherwise the memory segment is attached as read and write.

When the shared memory is no longer required the *shmdt* function is called passing the address of the shared memory segment attached. This function does not remove the shared memory segment from the system and the identifier created from the call to *shmget* is still in existence. To remove it the *shmctl* function is called passing the command IPC_RMID.

Function Declaration: 38: shmdt

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int shmdt(void *addr);

returns 0 if OK, -1 on error
```

### 10.1.2 A shared memory Client - Server system

The following programs show a shared memory system allocating 27 bytes of shared memory in the server and placing some text in it.

The client attaches to the memory segment and reads the data, finally the client changes the first entry in the shared memory segment into a * and the server exits.

Program 32: Shared memory Server program

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

// first define the size of the memory segment
#define SHMSIZE 27

int main(void)
{
char c;
int shmid;
key_t key;
char *shm, *s;
// now define a unique key for the area
key = 5678;
//now create a shared memory segment with RW permissions
if((shmid = shmget(key,SHMSIZE,IPC_CREAT |0666)) <0)
        {
        perror("shmget");
        exit(1);
        }
// now attach to it
if((shm = shmat(shmid,NULL,0)) == (char *) -1)
        {
         perror("shmat");
         exit(1);
        }

//now we assign the pointer s to point to the base of
//the shared memory segment
s=shm;
// now step through the alphabet and put this value in
// the shared memory location
for(c='a'; c<='z'; c++)
        *s++=c;
*s=(char )NULL;
// now we loop checking the first character in the memory
// segment to see if it is changed into a *
while(*shm !='*')
        {
        printf(".");
        fflush(stdout);
        sleep(1);
        }
// detach and remove the shm ID so it can be reused
// this is a bit like using a free after a malloc
if(shmdt(shm)<0)
        {
        perror("detach failed");
        exit(1);
        }

if(shmctl(shmid,IPC_RMID,0) <0)
        {
        perror("removal of shared memory failed");
        exit(1);
        }
printf("Server Exiting\n");
exit(0);
}
```

### 10.1.3 Client Program

Program 33: Shared Memory client

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

// define the size of the shared mem region
#define SHMSIZE 27

int main(void)
{
int shmid;
key_t key;
char *shm, *s;
// and specify a key
key = 5678;
//now get the segment
if((shmid = shmget(key,SHMSIZE,0666)) <0)
        {
        perror("shmget");
        exit(1);
        }
// and now attach to it
if((shm = shmat(shmid,NULL,0)) == (char *) -1)
        {
         perror("shmat");
         exit(1);
        }
//finally step through the segment printing out the data
for(s= shm; *s !=(char )NULL; s++)
        putchar(*s);
putchar('\n');
//now change the first element in the segment
*shm='*';
//finally detach the segment
if(shmdt(shm)<0)
        {
        perror("detach failed");
        exit(1);
        }
printf("client Exiting \n");
exit(0);
}
```

## 10.2 The ipcs utility

The ipcs utility is a console based utility which prints out the information on IPC facilities for the user of the system.

To test the ipcs function run the MemServer program in a console window. In another console type ipcs which will return the following

```
------ Shared Memory Segments --------
key       shmid     owner     perms     bytes     nattch    status
0x7b01217c 0         jmacey    600       1024      3         dest
0x0000162e 513       jmacey    666       27        1

------ Semaphore Arrays --------
key       semid     owner     perms     nsems     status

------ Message Queues --------
key       msqid     owner     perms     used-bytes  messages
```

This shows two shared memory segments belonging to jmacey. The second segment shows the segment allocated to the MemServer program (27 bytes).

If a program does not clear a shared memory segment properly the ipcrm utility may be used to remove a redundant ipc.

# Chapter 11

# Introduction to Unix Socket Programming

Like most Unix resources sockets are implemented through the file abstraction. When a socket is created a file descriptor is returned. Once this file descriptor has been properly initialised processes may read and write to as with a normal file.

When the socket is no longer required it should be closed so that the resources associated with it may be freed.

## 11.1 Creating a socket

New sockets are created using the socket() system call, which returns a file descriptor for the uninitialised socket. When created the socket must be tied to a protocol but it is not connected to anything.

The function prototype for **socket** is as follows

Function Declaration: 39: socket

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

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

**`-1 is returned if an error occurs;  otherwise  the  return value is a descriptor referencing the socket`**

The three parameter specify the protocol to use as shown in table 11.1

| **Parameter** | **Use** |
|---|---|
| int domain | Specifies the protocol family |
| int type | Either SOCK_STREAM or SOCK_DGRAM |
| int protocol | Specifies which protocol to use. |

Table 11.1: Protocols available to the Socket Command

The protocol families usedfor parameter one of the socket call are shown in table11.2 .

| Address | Protocol | Protocol Description |
|---|---|---|
| AF_UNIX | PF_UNIX | Unix Domain |
| AF_INET | PF_INET | TCP/IP (Version 4) |
| AF_AX25 | PF_AX25 | AX.25, used by amateur radio |
| AF_IPX | PF_IPX | Novell IPX |
| AF_APPLETALK | PF_APPLETALK | ApleTalk DDS |
| AF_NETROM | PF_NETROM | NetROM, used by amateur radio |

Table 11.2: Socket Protocol Definitions

### 11.1.1 Establishing Connections

After a stream socket is created it needs to be connected to it before it is of any use. Connecting to a socket is inherently asymmetrical as each side does it differently.

The server side of the socket gets the socket ready to be connected to and then waits for something to connect to it. This usually requires the server to start and then continuously wait for a client to connect to it.

The Client side of the system will create a socket and tell the system which address to connect to. The client then attempts to establish a connection, if the connection is successful bi-directional communication is allowed.

### 11.1.2 Binding an address to a socket

Both server and client processes need to tell the system which address to use for the socket. Attaching an address to the local side of the socket is called binding the socket and is done through the **bind()** system call.

Function Declaration: 40: bind

```
#include <sys/types.h>
#include <sys/socket.h>

int bind(int sock, struct sockaddr *my_addr, int addrlen);

returns 0 OK -1 on error
```

The first parameter is the socket being bound , the socket handle returned from the call to **socket()**, and the other parameters specify the address to use for the local endpoint.

### 11.1.3 Waiting for connections

After creating a socket the server binds the socket to the address they are listening to. After the socket is bound to an address the server tells the system it is willing to allow other processes to establish connections to the socket (at the specified address) by calling listen. Once a socket is bound to an address the operating system is able to handle processes attempts to connect to that address.

The connection is not immediately established. The listen function must first accept the connection attempt through the accept system call. New connections attempts that have been made to addresses that have been listened to are called pending connections until the connections have been accepted.

The listen and accept functions are prototyped as follows

Function Declaration: 41: listen accept

```
#include <sys/socket.h>
#include <sys/types.h>

int listen(int sock, int backlog);

returns 0 OK else -1;

int accept(int sock, struct sockaddr * addr, int * addrlen);

returns -1 on error else socket descriptor
```

Both of these functions expects the socket file descriptors as the first parameter. Listens other parameter, backlog, specifies how many connection may be pending on the socket before further connection attempts are refused. Historically this value is set to 5 however a larger value must sometimes be used.

The accept function changes a pending connection to an established connection. The established connection is given a new file descriptor which accept returns. The new descriptor inherits its attributes from the socket that was listened to.

The **addr** and **addrlen** parameters point to data that the operating system fills in with the address of the remote (client) end of the connection. Initially, **addrlen** should point to an integer containing the size of the buffer **addr** points to. accept returns a file descriptor if the connection is accepted or less than 0 if the connection has failed.

The whole client / server connection is shown in the figure 11.1.

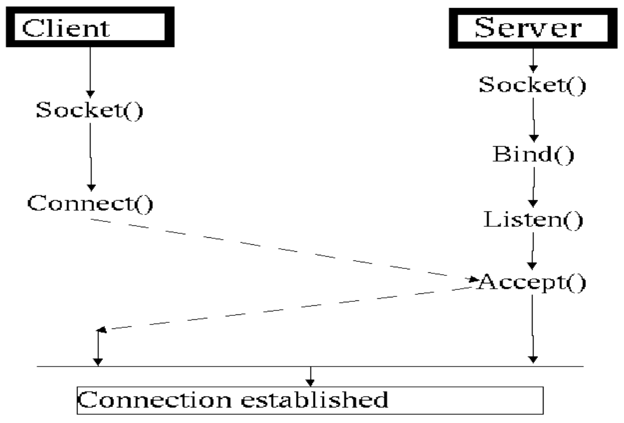

Figure 11.1: Client - Server connection using a socket

### 11.1.4 Connecting to a server

Like servers clients may bind the local address to the socket immediately after creating it. Usually the client doesn't do this and skips the bind stage.

The client then connects to a server using the connect function shown below

Function Declaration: 42: connect

```
#include <sys/socket.h>
#include <sys/types.h>

int connect(int sock, struct sockaddr * servaddr, int addrlen);

returns 0 OK else -1;
```

The process passes to connect the socket that is being connected, followed by the address to which the socket should be connected.

## 11.2 Networking with TCP/IP

The primary use for sockets is to allow applications running on different machines to talk to one another. The TCP/IP protocol family is the protocol used on the Internet and Unix allows for the use of TCP/IP to act as both server and client.

### 11.2.1 Byte Ordering

Different kinds of computers use different conventions for the ordering of bytes within a word. Some computers put the most significant byte within a word first (this is called "big-endian" order), and others put it last ("little-endian" order).

So that machines with different byte order conventions can communicate, the Internet protocols specify a canonical byte order convention for data transmitted over the network. This is known as the network byte order.

When establishing an Internet socket connection, the program must make sure that the data in the **sin_port** and **sin_addr** members of the **sockaddr_in** structure are represented in the network byte order. If the program is encoding integer data in the messages sent through the socket, the program should convert this to network byte order too. If the program doesn't do this, the program may fail when running on or talking to other kinds of machines.

If the program uses **getservbyname** and **gethostbyname** or **inet_addr** to get the port number and host address, the values are already in the network byte order, and they can be copied directly into the **sockaddr_in** structure.

Otherwise, they have to be converted explicitly. To do this **htons** and **ntohs** are used to convert values for the **sin_port** member; **htonl** and **ntohl** to convert values for the **sin_addr** member. (Remember, **struct in_addr** is equivalent to **unsigned long int**.) These functions are declared in **netinet/in.h** and are shown below

Function Declaration: 43: host to network conversion functions

```
#include <netinet/in.h>

unsigned long int htonl(unsigned long int hostlong);
unsigned short int htons(unsigned short int hostshort);
unsigned long int ntohl(unsigned long int netlong);
unsigned short int ntohs(unsigned short int netshort);
```

The **htonl()** function converts the long integer hostlong from host byte order to network byte order.

The **htons()** function converts the short integer hostshort from host byte order to network byte order.

The **ntohl()** function converts the long integer netlong from network byte order to host byte order.

The **ntohs()** function converts the short integer netshort from network byte order to host byte order.

On the i80x86 (PC based machines) the host byte order is Least Significant Byte first, whereas the network byte order, as used on the Internet, is Most Significant Byte first.

## 11.3 A TCP / IP Client Server system

The following design example shows a simple TCP / IP Server and Client. The Server is used to manipulate a message sent to it by the client depending upon a mode string sent in the message from the Client.

### 11.3.1 Client program

The client program accepts 4 argument and they must be in the correct order. The arguments are as follows :-

```
Client <hostname> <port num> <-mode[1,2]> <”message”>
```

host : this is the hostname of the server machine. This may either be the a text based hostname (such as madon12) or a dot seperated IP address (100.200.11.82)

portnum : the portnum parameter tells the client which port the server is listening on.

-mode[1 or 2] mode 1 tells the server to return the ASCII number values of the message sent. mode 2 tells the server to reverse the message and return it to the client

message this is the message sent to the server to be manipulated. If a space is to be used in the message the message must be placed in quotes e.g. (“message with space”)

Program 34: SocketClient

```
/*****************************************************************
   SocketClient.c
   Usage SocketClient hostname port_num -mode[a,b]
   -mode1 "Message" message is returned as ASCII Numbers
   -mode2 "Message" is reversed

   Simple Unix Socket based client which sends a  message to
   Server  in the form -mode1 < "message" > or -mode2 < "message" >
   the server returns ASCII num of message to Client in mode1
   and returns message reversed.

   Written by J.P. Macey
   Version 1.0
   14/2/1999
*******************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <netinet/in.h>
#include <netdb.h>
#include <arpa/inet.h>
#include <sys/socket.h>
/******************************************************************
   Function Prototypes
******************************************************************/

void usage(void);
void ErrorExit(char *message);

int main(int argc, char * argv[])
{
/* local variables */
char hostname[20];
int portnum;

int sockethandle; /* handle of the socket */
int mode; /* mode handle to be sent to the Server */
int ammount; /* ammount of data rx from the Server */
char buf[1024]; /* buffer to send data to server */
struct sockaddr_in hostaddress;
struct in_addr inaddr;
struct hostent *host;
char ClientData[1024];

/* check to see if correct ammount of arguments */
 if(argc<5)
    usage();
/* now copy command line parameter so we can use them */
else
    {
/* now copy hostname and portnum */
        strcpy(hostname,argv[1]);
        portnum = atoi(argv[2]);
/* now decide which mode were in */
        if (strcmp(argv[3],"-mode1")==0)
       mode = 1;
    else if(strcmp(argv[3],"-mode2")==0)
       mode =2;
    else ErrorExit("Client: invalid command line");

    printf("Client: Hostname = %s port = %d\n",hostname,portnum);
    printf("Client: in mode %d\n",mode);
/* finally create data to send to Server */
    sprintf(ClientData,"mode%d %s",mode,argv[4]);
    printf("Client: Sending %s to Server\n",ClientData);
        }
/* now see if we can resolve the hostname
   first we see if it is a dot seperate ip addr
    i.e. 194.34.34.124 */
 if(inet_aton(hostname,&inaddr))
     host = gethostbyaddr( (char *)&inaddr, sizeof(inaddr),AF_INET);
 else
     {
/* we must have a text based host name so we must check the
   DNS to see if we can resolve it */
     host = gethostbyname(hostname);
     printf("Client: Doing gethostby name %s\n ",host->h_name);
     }
/* it looks as if the host is not valid so we exit */
     if(!host)
        ErrorExit("Client: Host name lookup failed");
/* create a simple socket */
/* now we create a socke to connect to the server
   using TCP/IP and STREAM connection */
if((sockethandle=socket(AF_INET,SOCK_STREAM,0)) <0)
        ErrorExit("Client: An Error has occured\n");
/* now load the port number into the ip structure and save it */
  hostaddress.sin_family = AF_INET;
  hostaddress.sin_port = htons(portnum);
/* finally we copy the first ip addr of the host into the hostaddr structure */

  memcpy(&hostaddress.sin_addr,host->h_addr_list[0],sizeof(hostaddress.sin_addr));

/* now see if we can connect to the socket and thus the server */
  if(connect(sockethandle, (struct sockaddr *)&hostaddress,
               sizeof(hostaddress)))
     ErrorExit("Client: Error connecting to socket\n ");

/* It seems the socket is active and the server is running
   So we can send the data to the Server :-)
   this is done using the write function passing it the
   sockethandle and the data */
  ClientData[strlen(ClientData)]='\0';
  write(sockethandle,ClientData,strlen(ClientData));

/* now we read the reply from the server */
  ammount=read(sockethandle,buf,sizeof(buf));

  printf("Client: received %d bytes of data\n",ammount);
  printf("Client: data is %s\n",buf);

/* and finally close the socket handle so it may be used again */
  close(sockethandle);
  return 0;
}

/*****************************************************************
   void usage(void) print program usage and exit
   No return value
   No parameters
*****************************************************************/
void usage(void)
{
printf("Usage SocketClient hostname port_num -mode[a,b] \n");
printf("-mode1 \"Message\" message is returned as ASCII Numbers\n");
printf("-mode2 \"Message\" is reversed \n");
exit(1);
}

/*****************************************************************
   void ErrorExit(char *message) print error message and exit
   No return value
   char *message : message text to be printed before exiting
*****************************************************************/
void ErrorExit(char *message)
{
printf("Client: %s\n",message);
printf("Client: Exiting with value 1\n");
exit(1);
}
```

### 11.3.2 Server program

The server program opens a socket and listens for the client to connect to it with a message. When the message arrives it is parsed by the Server and then acted upon depending upon the mode string.

The server is used in the following way

```
Server <portnum> where portnum is the port to which the server will listen.
```

Program 35: SocketServer

```
/******************************************************************
   Server.c Usage Server portNUM
   Simple Unix Socket based server which recives messages from
   Client in the form -mode1 < "message" > or -mode2 < "message" >
   mode1 returns ASCII num of message to Client
   mode2 returns message reversed to Client

   Written by J.P. Macey
   Version 1.0
   14/2/1999
******************************************************************/
#include <sys/socket.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <netinet/in.h>

/******************************************************************
   Function Prototypes
*****************************************************************/
void usage(void);
void ErrorExit(char *message);
void GetClientMessage(int ConnectionHandle);
void parseClientMessage(int ConnectionHandle,char * message);
/****************************************************************/

int main(int argc, char * argv[])
{
/* local variables */
int portnum; /* port number used by socket */
int sockethandle,connectionhandle; /* handles used by sockets */
/* ip addr structures used by server */
struct sockaddr_in hostaddress;
int addrLength;

/* check command line args to see if correct */
 if (argc<2)
        usage();
 else
        {
/* convert command line portnum to integer */
        portnum = atoi(argv[1]);
        printf("Server: Using port = %d\n",portnum);
        }

/* create a simple socket */
/* using TCP/IP and STREAM connection */

 if((sockethandle=socket(AF_INET,SOCK_STREAM,0)) <0)
        ErrorExit("Server: Unable to Create Socket");

/* now load the port number into the ip structure and save it */

  hostaddress.sin_family = AF_INET;
  hostaddress.sin_port = htons(portnum);
  memset(&hostaddress.sin_addr,0,sizeof(hostaddress.sin_addr));

/* Bind the socket to port address */
  if(bind(sockethandle,(struct sockaddr *) &hostaddress,sizeof(hostaddress)))
        ErrorExit("Server: Failed to bind Port address ");
  else printf("Server: IP - Addr / Port Bound \n");

/* now tell the server to listen to the ip */
 if(listen(sockethandle,5))
        ErrorExit("Server: Error listening to socket");
 else printf("Server: Listening on port %d\n",portnum);

/* now tell the server to loop and wait for the client to
   connect to it. If the client contacts the server the GetClientMessage
   code is executed */

while(( connectionhandle = accept(sockethandle,(struct sockaddr * ) &hostaddress,&addrLength )) >=0)
```

```
        {
        GetClientMessage(connectionhandle);
        /* it is important to close connection handle once it has been used */
        close(connectionhandle);
        }
/* an error occured so we must close the handle and exit nicely
   this error is most likely that the socket has been closed
   by external influences */
  if(connectionhandle <0)
        ErrorExit("Server: Lost connection handle");
  close(sockethandle);
  return 0;

}
/*****************************************************************
   Function Prototypes
*****************************************************************/


/*****************************************************************
   void ErrorExit(char *message) print error message and exit
   No return value
   char *message : message text to be printed before exiting
******************************************************************/
void ErrorExit(char *message)
{
printf("Server: %s\n",message);
printf("Server: Exiting with value 1\n");
exit(1);
}

/*****************************************************************
   void usage(void) print program usage and exit
   No return value
   No parameters
******************************************************************/

void usage(void)
{
printf("Usage Server port_num \n");
exit(0);
}


/*****************************************************************
   void GetClientMessage(int ConnectionHandle) get message from client
   No return value
   int ConnectionHandle pointer to socket handle
******************************************************************/

void  GetClientMessage(int ConnectionHandle)
{

char buf[1024]; /* buffer to receive data */
int ammount; /* counter to size of data */

 printf("Server: contacted by client\n");
 printf("Server: Reading data into Buffer\n");
/* use the read function to read data from socket into
   buffer buf size of the data returned in ammount */
 ammount=read(ConnectionHandle,buf,sizeof(buf));

 printf("Server: received %d bytes of data\n",ammount);
 printf("Server: data is %s\n",buf);
/* now we have the data from the client we have to do
   something with it */
 parseClientMessage(ConnectionHandle,buf);
}

/*****************************************************************
   void parseClientMessage(int ConnectionHandle,char * message)
   take the buffer read from the  client and do something
   No return value
   int ConnectionHandle pointer to socket handle
   char * message : message received from the client
******************************************************************/

void parseClientMessage(int ConnectionHandle,char * message)
{
int i,length,j;
```

```
char tempbuff[6];
char returnbuff[1024];
char conv[4];

/*first parse first 5 character to find which mode */
for (i=0; i<5; i++)
                tempbuff[i]=message[i];
/* as we’re using strings we need to null terminate ;-) */

tempbuff[i]=’\0’;
length = strlen(message);
printf("Server: mode message = %s length =%d\n",tempbuff,length);

/* now find which mode so we can construct a return message */
if(strcmp(tempbuff,"mode1") == 0)
        {
/* now we take the individual characters from the message and
   write them as ASCII numbers to an array called returnbuff */
        sprintf(conv,"%d",(int)message[6]);
        strcpy(returnbuff,conv);
        for(i=7; i<length; i++)
                {
                sprintf(conv,"%d",(int)message[i]);
                strcat(returnbuff," ");
                strcat(returnbuff,conv);
                }
        printf("Server: returning %s\n",returnbuff);
        }
/* This mode we need to reverse the string */
else if(strcmp(tempbuff,"mode2") == 0)
        {
     for (i=length-1, j=0; j<length-6; i--, j++)
                returnbuff[j]=message[i];
/* and don’t forget to NULL terminate :-< */
     returnbuff[j]=’\0’;
        }
/* this must mean an error has occured in transmission
   as the client checks for this when parsing the command line
   the error must have occured in tx. Which is unlikely but it is
   best to check just in case :-) */
else ErrorExit("Server: incorrect message passed");

/* finally we use the write function to send the data back to
   the client
   Note the use of strlen(returnbuff)+1 rather than using sizeof
   this means that only the length of the string is sent back
   and not the full buffer which is 1024 long. This saves
   transmission bandwidth and makes the program more efficient */
write(ConnectionHandle,returnbuff,strlen(returnbuff)+1);
}
```

### 11.3.3 Message structure

The message passed to the Server from the client is constructed as follows

modeX message data

where X will be either 1 for returning ASCII or 2 for reversing the string. The message start from the 6th byte and ends when a NULL (\0) is found.

The message passed back from the Server is a simple char array with no encoding.

### 11.3.4 Testing the system

```
To compile the client type
gcc -Wall -g SocketClient.c -o SocketClient
To compile the server
gcc -Wall -g SocketServer.c -o SocketServer
The Server may be started in the following way
SocketServer 2001&
After this the Client may be used as follows
SocketClient localhost 2001 -mode1 “this is a test”
```

### 11.3.5 Testing system across a network

To test the system across a network we need to open two shells. In the first shell do the following

rlogin madonXX -l u97xxxxxx

where madonXX is one of the madon machines from 1 -16 and u97xxxxx is your user number.

Change to the directory where the server executable is and run it with a portnumber. It is important to use a unique port number so if the server failes to open a port it is likely that that number is in use to try a different one.

Now on the second console run the client program passing the name of the host the server is on and the port that was used. The Client server system should now work properly across the network.

# Part III

# Appendices

# Appendix A

# Basic C Programs

C is a compiled language. This means that it has to go through a number of stages before a program is generated. Figure A.1 show the stages of generating an Executable program from C source files.

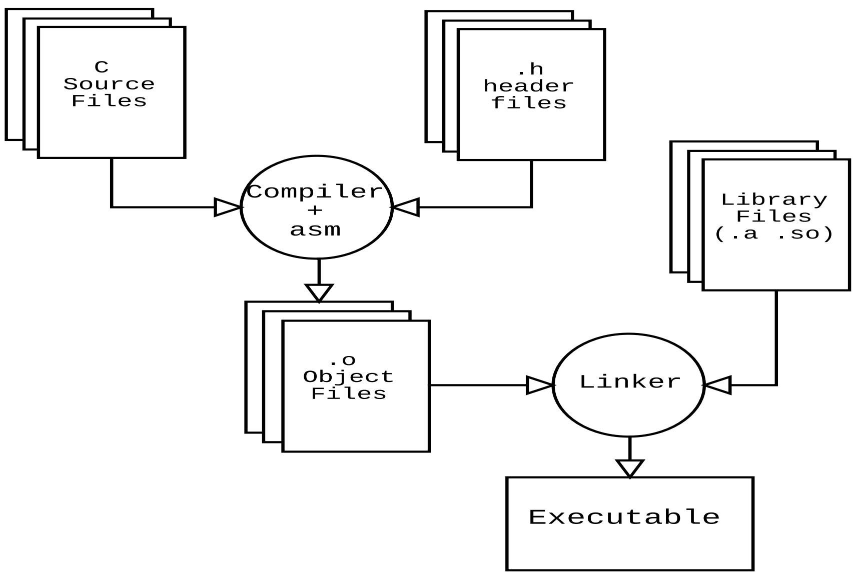

Figure A.1: The C compilation process

The C compiler takes C source files which may include other resource files know as header files (identified by a .h extension). If the syntax of the C program is correct a machine level object file (.o) is generated[1].

This object file is then linked by the linker program to produce an executable program.

If more than one C source file is used for the program an object file will be generated for each C source file. The linker will then combine all of the object files into an executable program.

## A.1 The Anatomy of a C program

To keep the layout of a C source file consistent and easier to maintain we will break it into three sections as follows

[1]Just because a source file compiles doesn't necessarily mean it will link or even run if it does link.

- Declaration Block
- Main Code Block
- Function Declaration Block

### A.1.1 The Declaration Block / Function Declaration Block

The first part of the C source file is the Declaration Block. This section is used to Declare any resources, variables or functions that the C program will use.

#### The #include directive

The #include directive is a precompiler instruction which is used to include other files into the main source file. The syntax is as follows #include <stdio.h> which includes the function prototypes for the standard input output routines[2]. This statement is similar to the java Import statement.

#### Function Prototypes

If the C source file uses functions which are defined by the user, the prototypes for the functions need to be declared. This is to allow the compiler to identify any functions the user has defined. The following example show a user defined function called add_numbers

Function Declaration: 44: user defined functions

```
/* function prototype */
int add_numbers(int num1, int num2);
.
......Main code block
.
.
/* function declaration */
int add_numbers(int num1, int num2)
{
return num1+num2;
}
```

#### Variable Declarations

If global variables are to be used[3] they are also declared in the first section of the C source file.

[2]There are many different header files in the C language all prototyping different functions. As more C functions are used more Header files will be included.

[3]The uses of global variables is discouraged as they are very prone to errors, however it is inevitable that they are sometimes used!

### A.1.2 The Main code block

The main code block is where the main structure of the program is placed. This contains all of the function calls and control structures of the program. The entry point into a program is a special function called main. There can only be one instance of main in a C program, and it is from this point that all other elements of the program are called.

### A.1.3 First C Program

The following code is the classic hello world program used for generations to show the basic of any programming language.

Program 36: hello world

```
#include <stdio.h>
int main(void)
{
printf("Not Hello World again\n");
return 1;
}
```

To compile this C source code save it into a file Hello.c

## A.2 Compiling C programs

On Unix systems there are many different C compilers, however the standard C compiler on most systems is gcc. gcc has a number of different options to allow the user to customise the behavior of the compiler. For the most part the following options will be used :-

```
-o <Name>  Produces an output program <name>
-Wall      Print all Warnings
-g         Add debug information
```

The format of gcc is the following gcc <source file>.c -Wall -o <executable name> so for the first example we would use the following

```
gcc Hello.c -Wall -o Hello
```

## A.3 C Input and Output functions

C has a number of input and output functions all defined in the **stdio.h**[4] header file however for general use two of the formatted io functions will be used. These are **printf** and **scanf**.

[4]open the file /usr/include/stdio.h to see all of the functions prototyped in stdio.h

### A.3.1 Standard IO streams[5]

Under normal circumstances every Unix program has three streams opened for it when it starts up, one for input, one for output, and one for printing diagnostic or error messages. These are typically attached to the user's terminal but might instead refer to files or other devices, depending on what the parent process chose to set up.

The input stream is referred to as "standard input"; the output stream is referred to as "standard output"; and the error stream is referred to as "standard error". These terms are abbreviated to form the symbols used to refer to these files, namely **stdin, stdout,** and **stderr**.

The stream **stderr** is unbuffered. The stream **stdout** is line-buffered when it points to a terminal. Partial lines will not appear until **fflush** or **exit** is called, or a newline is printed. This can produce unexpected results, especially with debugging output.

### A.3.2 Formatted Output

In it's simplest form **printf** may be used to print simple text strings. An example of this was shown in the previous section with the code **printf("Not Hello world again\n");** This prints to the standard output (**stdout**) which is directed to the terminal that the process is executed in.

### A.3.3 Simple Format Codes

The **printf** function uses format codes to modify the behavior of the text contained within the quotes. The simplest of these are the escape sequences which are preceded by a \ and are described in table A.1.

| Escape Sequence | Function |
|---|---|
| \n | Prints a Newline and Flushes stdout |
| \r | Forces the line back to the beginning |
| \t | Prints a Tab space |
| \\ | Prints a single \ |
| \" | Prints a single " |
| %% | Prints a single % |

Table A.1: printf escape sequences

An Example of using these escape sequences is shown in the example code below

Program 37: examples of printf

```
#include <stdio.h>
int main(void)
{
printf("Simple printf Example\n");
printf("\tThis\tis\ttab\tspaced\n");
printf("\n\nNow Two New Lines before and after\n\n");
printf("Now a \nNewline\nin the middle\n");
printf("Now to print a \\ or a %% or  \" \n");
printf("\nEach\rtime\rthe\rline\ris\rprinted\ra\r\return\ris\rcalled\n");
return 1;
}
```

[5]From the Redhat Linux UNIX Programmer's Manual V2.0

### A.3.4 Conversion Specification Characters

Conversion specification characters are used to convert and print different data types. The **printf** function does this in the following way

```
int printf(const char *format, ...);
```

The integer return parameter of printf returns how many characters have been printed. The format string is contained in double quotes ("") and can contain the following

- Plain Text
- Escape Sequences beginning with a /
- Conversion Specification Characters identified by %

After the quotes a list of the conversion characters are separated by a comma in the following format

```
printf("This is a number %d\n",2);
```

The %d specifies that the first argument after the end quote is a number and should be converted and printed.Table A.2 lists of the some of the format strings.

| Format String | Meaning |
|---|---|
| %d | Integer Decimal |
| %o | Octal Decimal |
| %x | Hexadecimal |
| %f | Floating Point (Decimal Notation) |
| %e | Floating Point (1.E notation) |
| %c | First Character or argument is printed |
| %s | Argument is taken to be a string |

Table A.2: printf format strings

As well as these argument width specifiers may also be specified, for example to print a decimal in the format 001 002 ...100 the format string **%003d** would be used indicating that the string should be padded with leading zeros and that the string will not exceed 3 digits.

### A.3.5 Using printf with variables

**printf** may also be used to print the value of a variable or to construct a string of variables. The following example prints out the value of three variable types

Program 38: more printf

```
#include <stdio.h>
int main(void)
{
int a=10;
float b=35.32;
char c[6]={’h’,’e’,’l’,’l’,’o’,’\0’};
printf("int a = %d float b = %f string c= %s\n",a,b,c);
return 1;
}
```

In this example three variables are declared, a an integer, b a floating point and finally c an array of characters (a sting)[6]. The **printf** function is then used to print out the values of the variables.

It is important that the variables are placed in the correct sequence according to the conversion characters else the results will be unpredictable.

It is also important that the correct number of arguments are present else garbage will be printed, however if too many arguments are presented any without a conversion specifier will be ignored.

## A.3.6 Formatted Input using scanf

The **scanf** function is used to collect user input from **stdin**. It used a similar method of format specifiers to printf, however the input is terminated by the return key being pressed.

To input a decimal value the following code is used

Program 39: using scanf

```
#include <stdio.h>

void main(void)
{
int a;
printf("Enter a number :");
scanf("%d”,&a);
printf("The Number Entered was a= %d”,a);
return 1;
}
```

It is important to notice the **&a** in the parameter to scanf. This indicates that the data input is passed to the memory location pointed to by the address of a.

## A.3.7 Creating Compound Strings using sprintf

A compound string is a string constructed from other variables and text. This is useful for generating a longer text string from other data types.

[6]C does not have an explicit string data type so strings are declared as arrays of characters.

The function to create compound strings is **sprintf**[7]. **sprintf** is very similar to **printf** however the output is placed into a string rather than to the screen. The format for this function is as follows

Function Declaration: 45: sprintf

```
#include <stdio.h>

sprintf(<compound string>,"Conversion string",...);
```

The first parameter is either a character array (char String[50]) or a char pointer (char *String). The second parameter is used to specify any text to be used and the conversion specification characters in the same format as **printf** , finally the variables to be converted are placed in sequence and the same restrictions as **printf** apply.

The code below shows how this is implemented

Program 40: creating a compound string using sprintf

```
#include <stdio.h>

int main(void)
{
char inputa[10],inputb[10],compound[50];

printf("Enter a String :\n");
scanf("%s",&inputa);
printf("\nEnter another String :\n");
scanf("%s",&inputb);

sprintf(compound,"compound string = %s %s\n",inputa,inputb);
printf("%s\n",compound);
return 1;
}
```

In this example three character arrays[8] are initialized inputa and inputb are assigned to be 10 characters and compound 50 characters to hold the result. Then scanf is used to input the values into inputa and inputb. sprintf is then used to create a compound string from the two inputted values and then the string is output using printf.

It must be noted that this example is in no way perfect. The two input character arrays can only hold a string 10 characters long. If this length is exceeded the program will have an undetermined output an may also be liable to core dump (crash). There are a number of ways to overcome this problem which will be discussed later.

[7]string print formatted

[8]C has no String data type, however as a string is just a character array a string may be constructed by char string[10] which creates a 10 character string.

# Appendix B

# Basic C syntax

The following appendix gives a list of basic 'C' syntax which will be used throughout the assignments and exercises in this unit.

## B.1 Comparison and Logical Operators

For selection and iteration the following operators are used

| **C Notation** | **Meaning** |
|---|---|
| < | less than |
| > | greater than |
| <= | less than or equal to |
| >= | greater than or equal to |
| == | equal to |
| != | not equal to |
| && | logical and |
| \|\| | logical or |
| ! | not |

Table B.1: Comparison and logical operators

From the table B.1 we can say that < > <= >= are all relational operators and == != are equality operators.

In C the expression x < b is of integer type and its value is either 0 (false) or 1 (true). Also any other negative number is also interpreted as false. Therefore the following statement

**if (n!=0) ....**

may be written as **if(n) .....**

### B.1.1 Bitwise Logical Operators

Table B.2 shows operators for bit manipulation and they can be applied to the following data types **int, short, long, unsigned, char**

| **C Notation** | **Meaning** |
|---|---|
| **&** | bitwise AND |
| \| | bitwise OR |
| ^ | bitwise EXCLUSIVE OR |
| << | left Shift |
| >> | right Shift |
| ~ | one's compliment |

Table B.2: Bitwise Logical Operators

For example if a=23 b=26 the value of c=a&b is 18 as shown below

```
    a= 23 = 00010111
    b= 26 = 00011010 &
    c= 18 = 00010010
```

For each of the bit places in the above example the logical & is taken and the result placed in c.

## B.1.2 Assignment operators

In C a single equal sign (=) is the assignment operator as shown in the example x=a+b.

There are, however, some short hand versions of the assignment operators which at first may seem confusing but allow for very compact code. These are as follows

| Assignment | Short hand |
|---|---|
| x=x+a | x+=a |
| y=y-b | y-=b |
| u=u+1 | ++u |
| v=v-1 | –v |

The value of ++u is the incremented value of u, so first u is incremented and then it's value is used. The opposite order is also possible. If u++ is used the old value of u is used first and u is then incremented afterwards. so after the execution of

```
u=5; v=5; x=++u; y=v++; we have u=v=x=6 and y = 5
```

## B.1.3 Conditional Expressions

The special character pair **? :** are used to build special conditional expressions in the format

expression 1 ? expression 2 : expression 3

For example

```
a<b ? b-a : a-b
```

so b-a is executed if a is less than b

or a-b if a in not less than b

In most cases the the **if** statement is used for conditional execution of statement and is used in the following way

```
if (expression) {statement}
```

The if statement may also be expanded to the following form

```
if (expression) {statement1}
else {statement2}
```

or even further to

```
if (expression) {statement1}
else if (expression) {statement2}
else {statement 3}
```

### B.1.4 Iterations

C has the following constructions for loops

1. The while-statement
2. The for-statement
3. the do - while -statement

A while statement has the following form

```
while (expression) {statement}
```

and has the advantage that the expression is first evaluated and then the statement(s) executed only if the condition is true.

Conversely the do - while statement shown below will always execute the statement(s) at least once and then repeat the statements dependant upon the condition

```
do
  {
   statement(S)
  }while(condition);
```

Finally the for loop which has the following syntax

for(start value; test; increment)

For example the following code will execute a simple loop to print out the value of x

int x;

```
for (x=0; x<10; x++)
     printf("x= %d \n",x);
```

### B.1.5 break and switch

The statement break; terminates the (innermost) loop that contains the statement. It may be used in any of the iteration statements for example

```
while(1)
{
ch = getchar();
if(ch == 'q')
    break;
else printf("%c",ch);
}
```

In the above example the variable ch is used to read input from the standard input (key board) ch is then checked to see if it is the character 'q'. If is is the loop is broken out of using the break statement else the loop continues and prints out the character pressed.

The case statement is a very elegant way of doing multiple tests on an ordinal value (such as char int etc) and has the following format

```
ch=getchar();
switch(ch)
     {
      case 'a' :
         statement1
      break;
      case 'b' :
         statement2
      break;
      default :
         do this by default
      break;
      }
```

In the above example if the key a is pressed the case statement will execute statement 1. If b is pressed statement2 is executed. If any other value is assigned to ch the default value will be executed.

## B.2 C Data Types and Structures

C has a number of data types and has the ability to allow the programmer to define custom data types built up from the basic C data types.

### B.2.1 #define

#define is a pre-processor function which the C compiler uses to allow code to be more readable. For example if the value LENGTH is to be used the following code would be generated

```
#define LENGTH 100
char data[LENGTH];
```

When the C compiler is used it will first replace any instance of the define LENGTH with the value 100. If the definition is used throughout the code it makes it easier to change the value LENGTH as only the definition need be changed.

It is also standard practice to use upper case when defining constants so they may be differentiated from variables.

#define may also be used to define macros and code segments in the following example a macro Max is defined and then used in the main code

Program 41: using #define

```
#include <stdio.h>
#define Max(x,y) x > y ? x : y
int main(void)
{
int i,j;
float a,b;
printf("Enter two ints and two real numbers\n");
scanf("%d %d %f %f",&i,&j,&a,&b);
printf("\nMaximum values %d %f",Max(i,j),Max(a,b));
return 1;
}
```

When this code is compiled the reference to Max is replaced with the code x > y ? x : y. The ? operator returns either x or y depending upon the operator x > y if x is > y x is returned else y is returned.

### B.2.2 char data type

A *char* is the smallest data type available in the C language. It is equal to one byte and is generally used to represent characters.

The range of a *char* is -127 -> 0 -> 128. Therefore if a value greater than 128 is assigned to a *char* it will be converted to a negative number where 129 == -127 and 255 == -1

To force a *char* to be positive the ***unsigned*** prefix is used in the following way ***unsigned char*** and will force the char to be in the range 0 - 255.

***unsigned*** may be used with any data type to force it to be positive.

### B.2.3 Strings

As mentioned previously there is no string data type, however a string may be created by using an array of chars as follows. *char* String[50];. This creates an array index from 0 to 49 to hold individual characters.

Although this is simple to create and use this method is very inefficient as the memory allocated is 50 bytes.

### B.2.4 Numeric data types

The simplest numeric data type is the ***int*** this is used to store whole numbers and by default an *int* is signed.

Integer literals may also be used to allow the assignment of Decimal (default) Hexadecimal, characters and octal values. These may be written in the following way

| Literal | Format |
|---|---|
| decimal | 0 2 63 85 |
| octal | 00 02 077 0123 |
| Hexadecimal | 0x0 0x2 0x3f 0x53 |

Characters may be assigned to an integer using the single quotes ” as follows

int a=’c’;

### B.2.5 Floating point data types

The floating point data type is used to represent floating point numbers. Like integers floating point types come in three sizes, *float* (single-precision), *double* (double precision) and *long double* (extended precision).

### B.2.6 Void

The *void* is syntactically a fundamental type. It can, however, be used only as part of a more complicated type. It is used either to specify that a function does not return a value or as the base type for pointers to objects of an unknown type.

### B.2.7 Size

The size of C data types are usually measured in multiples of the size of a *char*, however the size of a *char* is dependant upon the architecture of the machine and OS being used.

The program below is used to display the size of various data types using the *sizeof* function.

Program 42: size of C data types

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

int main(void)
{
char a;
int b;
float c;
double d;
long e;
void *f;
short l;
long int g;
short int h;

char *i;
int *j;
float *k;

printf("a char is %d bytes\n",sizeof(a));
printf("an int is %d bytes\n",sizeof(b));
printf("a float is %d bytes\n",sizeof(c));
printf("a double is %d bytes\n",sizeof(d));
printf("a long is %d bytes\n",sizeof(e));
printf("a void* is %d bytes\n",sizeof(f));
printf("a short is %d bytes\n",sizeof(l));
printf("Extended data types \n");
printf("a long int is %d bytes\n",sizeof(g));
printf("a short int is %d bytes\n",sizeof(h));
printf("a char* is  %d bytes\n",sizeof(i));
printf("an int* is  %d bytes\n",sizeof(j));
printf("a float* is  %d bytes\n",sizeof(k));
return 1;
}
```

# Appendix C

# Pointers

Pointers get a lot of bad press, and to most people they are the most difficult (both conceptually and practically) elements of the C/C++ language for the novice programmer to grasp.

### C.0.8 What are pointers?

Pointers are variable just like any other variables. They may be defined using any data types and are differentiated from other data types by the use of the asterisk * being placed before the variable name as shown in the examples below

float *ptrRealNumber;

char *ptrCharacterString;

int *ptrFunction();

int *ptrArray;

### C.0.9 So what are Pointer Used for ?

Pointers are used to point(!) into a program, this means that they can directly access areas of memory in use by the program by storing the address of another variable. This is similar to the pass by reference method of passing data to a function.

Due to this flexibility pointers allow the programmer to create and manipulate data structures which may grow or shrink known as dynamic data structures.

### C.0.10 Pointer mechanics

Pointers differ from normal variables in the following ways

- Normally a variable directly contains a specific value
- A pointer contains the (memory) address of a variable that contains a specific value
- A variable name *directly* references a value and a pointer *indirectly* references a value.
- Referencing a value through a pointer is called *indirection.*

This is shown in figure C.1.

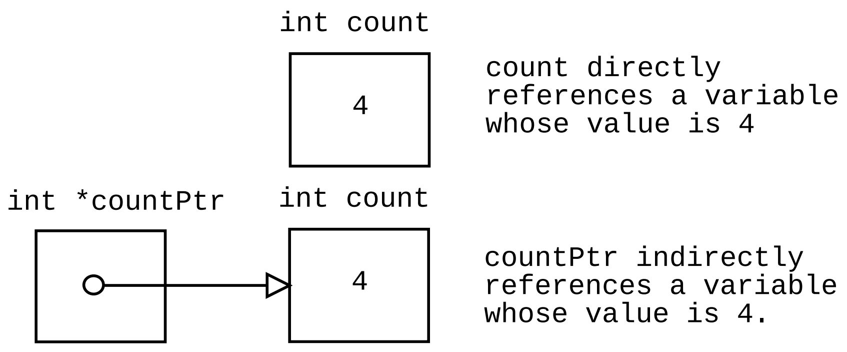

Figure C.1: Simple pointers

In the above example *int count* is an integer variable which has had the value 4 assigned to it by using *count=4;* i.e. *count* is referenced directly.

In the second section a pointer *int *countPtr* has been declared and is indirectly referenced to *count* i.e. *countPtr* points to the memory address of *count* by using the unary & prefix as follows

*countPtr = &count;*

In words the above code means *countPtr* is pointing to the area of memory (the address) of *count*.

## A simple pointer example

The following program demonstrates a simple pointer usage

Program 43: A simple pointer program

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

int main(void)
{
int i;
int *ptrI;
i=5;
printf("I given the value 5 directly %d\n",i);
ptrI=&i; // ptrI now points to i
*ptrI=10;
printf("Now using the pointer ptrI now equals %d\n",i);
return 1;
}
```

First the integer i is declared, after this an integer pointer(*ptrI)* is declared using the * to indicate it is a pointer. After this the variable *i* is directly given the value 5 by using the assignment operator (=) and just to prove that this has happened the value is printed out.

Next the pointer *ptrI* is assigned to point to the memory address where the variable *i* resides.

The next assignment **ptrI=10;* tells the program to "store 10 in the location pointed at by *ptrI*".

If we assume that the program's memory starts at address 0x100 and the variable *i* is at 0x102 and *ptrI* at 0x106 as shown by the memory map in figure C.2.

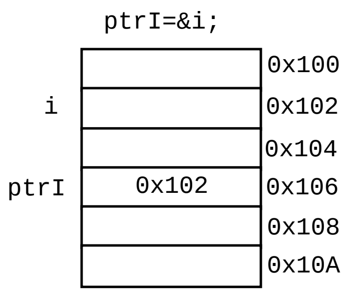

Figure C.2: Memory map for pointer example

In the memory map in figure C.2 ptrIholds the value of &i (0x102) when the assignment **ptrI=10;* is made the value is placed into the address pointed to by *ptrI* as shown in figure C.3.

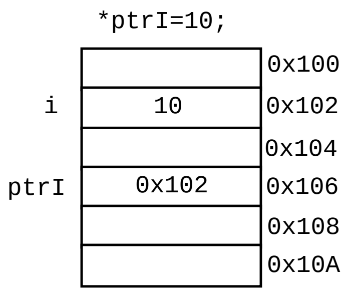

Figure C.3: Modified Memory map for pointer example

### C.0.11 void pointers

To avoid compilation errors a pointer must be of the same type as the variable it is pointing to. For example *intPtr = &charC;* would give a compilation error. However to overcome this problem the *void* data type may be used and allows a pointer to point to any data type, however we must explicitly tell the compiler what data type the *void* pointer should be as shown in the example program below

Program 44: void pointer example

```
#include <unistd.h>
#include <stdio.h>
int main(void)
{

int i;
char c;
float f;
void  *ptrMorph;
ptrMorph = &i;
*((int *)ptrMorph) =10;

ptrMorph =&c;
*((char *)ptrMorph)='c';

ptrMorph=&f;
*((float *)ptrMorph)= 25.45f;


printf("i=%d c=%c f=%f\n",i,c,f);
return 1;
}
```

The main difference in this program from the previous example is the use of the *void *ptrMorph* and the use of typecasting on the variable assignment. This is explained in figure C.4.

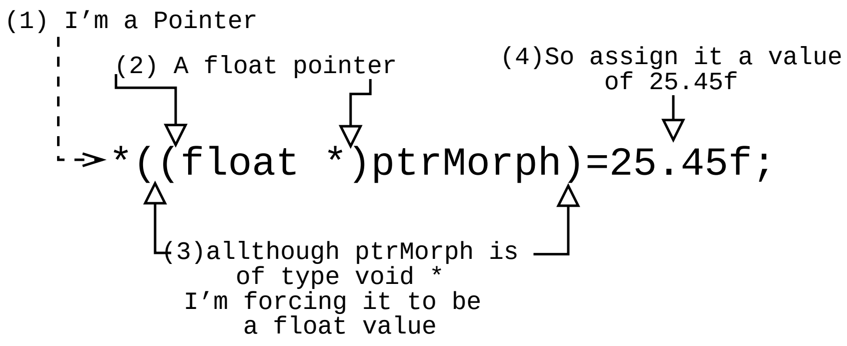
(1) I'm a Pointer
(2) A float pointer
(4)So assign it a value of 25.45f
*((float *)ptrMorph)=25.45f;
(3)allthough ptrMorph is of type void * I'm forcing it to be a float value

Figure C.4: void pointers explained

### C.0.12 In Conclusion

Every expression has a type as well as a value, the type of the expression *&i* is a pointer and tells the compiler that 'we want the address off i'. To make the use of pointers easy we must match the type of the pointers exactly such as

int i;

int *ptrI;

ptrI=&i;

However to allow for more flexibility we can use the *void* pointer type to point to any data type **but** we must type cast the *void* pointer to the correct type.

# Appendix D

# Unix Line Editors

*vi* provides basic text editing capabilities. Three aspects of *vi* make it appealing. First, *vi* is supplied with all UNIX systems. Second, *vi* uses a small amount of memory, which allows efficient operation when the network is busy. Third, because *vi* uses standard alphanumeric keys for commands, you can use it on virtually any terminal or workstation in existence without having to worry about unusual keyboard mappings. As a point of interest, *vi* is actually a special mode of another UNIX text editor called *ex*. Normally you do not need to use *ex* except in *vi* mode.

## D.1 Starting VI

To start vi use the following command

```
    #vi sample
```

The terminal window will clear and displays the contents of the file, *sample*, Since this file doesn't contain any text vi uses the tilde (~) character to indicate lines on the screen beyond the end of the file. vi uses a cursor to indicate where the next command or text insertion will take effect. The cursor looks like a small rectangle the size of one character, the character inside the cursor is known as the current character.

At the bottom of the *vi* window is a line called the *modelin*e. This is used to display the current line, the name of the file and the current status of *vi*.

## D.2 Command Mode and Input Mode

*vi* has two modes, *command mode* and *input mode*, In *command mode vi* executes different commands dependent upon which keys and key combinations are pressed such as search and replace, cut and copy text etc[1]. When *vi* is started it is in *command mode*.

To switch from *command* to *input mode* press the "*i*" key (you don't need to press the *RETURN*). *vi* then lets you insert text at the current cursor position. To switch back to the *command mode* press the *ESC* key. The *ESC* key may also be used to cancel an uncompleted command in the *command mode*.

Some versions of *vi* do not indicate which *mode* the editor is in so pressing the *ESC* key a couple of times will return to the command mode and this is usually indicated by *vi* beeping.

[1]for a list of these commands see Appendix B

## D.3 Inserting Text

While in *input mode* all text typed is inserted into the document, however *vi* recognises some special keystrokes as shown in table D.1.

| **Command** | **Action** |
|---|---|
| CTRL + W | Erase the previous word |
| CTRL + U | Erase the current line |
| BkSp | Delete previous character |
| RETURN | Start a new line |

Table D.1: Commands to remove text

### D.3.1 Document navigation

In most modern versions of *vi* the arrow keys may be used to move within the document. This will work in both modes of *vi* and allows easy navigation, however in older versions of *vi* and when *vi* is used on a terminal some of these key will not work. So to allow movement within a document in command mode the special keys shown in table D.2are used.

| **Key** | **Action** |
|---|---|
| k | Move up one line |
| h | line move one character to the left |
| l | line move one character to the right |
| j | Move down one line |

Table D.2: Navigation keys.

To add to this modern versions of *vi* also recognise the following special keys for navigation in either *insert* or *command mode* as shown in table D.3.

| **Key** | **Action** |
|---|---|
| Home | Return cursor to beginning of the current line |
| End | Place cursor at the last character of the current line |
| PgUp | Move up one page |
| PgDn | Move down one page |

Table D.3: Special Navigation keys

### D.3.2 Document Structure

*vi* has it’s own way of specifying the structure of a document as described below

| | |
|---|---|
| *Sentence* | A sentence is all the characters between normal sentence punctuation marks such as fullstops (.) question marks (?) and exclamation marks (!). A blank line also ends a sentence. |
| *Line* | The text between two RETURN characters forms a line. Hence it is possible to have |
| *Paragraph* | A paragraph is a sequence of lines which are not interrupted by any blank lines. |

Using these definitions the navigation commands shown in table 5 may be used in the command mode.

| Command | Cursor Moves to |
|---|---|
| b | beginning of previous word |
| w | beginning of next word |
| e | end of current / next word |
| 0 (zero) or ^ | beginning of line |
| $ | end of line |
| ( | beginning of current / previous sentence |
| ) | beginning of next sentence |
| { | beginning of current / previous paragraph |
| } | end of current paragraph |
| H | top line on screen |
| M | middle line on screen |
| L | bottom line on screen |

Table D.4: Document navigation

## D.4 Deleting Text

Sometimes text needs to be deleted, in modern versions of *vi* the *BkSp* and *Del* keys may be used in *insert mode* as with a normal editor, so *Bksp* will delete to the left of the current cursor position and *Del* will delete to the right.

There are also other delete commands which may be used in command mode as shown in table D.5.

| Command | Action |
|---|---|
| x | Delete only the current character |
| D | Delete to the end of the line |
| db | Delete from the current character to the beginning of the current word |
| de | Delete from the current character to the end of the current word |
| dd | Delete the current line |
| dw | Delete from the current character to the beginning of the next word |

Table D.5: Delete commands

Notice that the second letter of the command specifies the same chunk of text as the cursor movement commands shown in table 5 and you can use delete with any of these movement specifiers.

## D.5 Making corrections

Instead of deleting a character or word when it is incorrect *vi* has the ability to change or replace words and characters. These commands are shown in table D.6.

| Command | Action |
|---|---|
| cw | Change word. |
| C | Overwrite to the end of the line |
| r | Replace a single character with another one |
| R | Overwrite characters starting from the current cursor position |
| s | Substitute one or more characters for a single character |
| S | Substitute the current line with a new one |

Table D.6: Correction / Replacement commands

The change command *c* works like the delete command and any of the text portion specifiers from table 5 can be used.

## D.6 Undoing

*vi* will allow the user to undo the last change by pressing the *u* key. Modern versions of *vi* allow for multiple undos but this must be used with caution as the last thing entered may be the text itself and this will leave a blank document.

### D.6.1 Joining lines

It is possible to link two or more lines together, usually because deleting text has created a lot of empty space. The *J* command combines the current line with the line below it.

## D.7 Saving work

*vi* provides several ways of saving changes as shown below

```
      :w RETURN
```

This saves the current file (“w” is short for “write”), to exit *vi* use the following command

```
      :q RETURN
```

These commands may also be combined to save work and exit *vi* as shown below

```
      :wq RETURN
```

A shorthand for the above command (write and quit) is *ZZ* (shift zz).

Sometimes a file has been modified and the user wishes to exit without saving the file, to do this use the following command

```
      :q! return
```

Note that this command must be used with caution as exiting *vi* in this method will lose any changes made to the file (and *vi* will not prompt to ask the user if they are sure!)

## D.8 Repeating a command

*vi* allows the user to repeat a command by pre-fixing the command with a number which indicates how many times the command is to be repeated so for example typing *3dw* will delete three words from the current cursor position.

This will also work for insertion of text so the command *10iHello! ESC* will insert the Hello! 10 times from the current cursor position.

If a number is typed in error (and remember that when in command mode *vi* doesn't display any of the characters typed) just press the ESC key which will reset the command.

## D.9 Line numbers

Many *vi* commands use line numbers, *vi* counts the number of return characters and each of these consists of a line (as some lines may be longer than the *vi* display width). *vi* uses line numbers in cut copy and paste commands, and many programs such as compilers give error messages and warnings based on line numbers therefore moving to a line is a useful feature.

Enable *vi* to print line numbers next to each line issue the following command

```
:set number RETURN
```

This command displays the line numbers in the Left margin of the *vi* window, however it may cause long lines to wrap but this will not damage the text in the document.

To jump to any line in the document the *G* command is used. The following example shows some of its uses

```
1G
G
6G
```

The first command moves the cursor to the 1st line of the file. G on its own will move the cursor to the end of the document and finally 6G moves the cursor to the sixth line of the file.

Modern versions of *vi* also allow movement to a line by typing the *:* followed by the line number as shown below

```
:12 RETURN
```

To turn off line numbers the following command is typed

```
:set nonumber
```

To find out more detailed information about where the cursor press *Ctrl + G* which gives the following display

```
"sample" [Modified][New file] line 2 of 18 --11%-- col 1
```

## D.10 Markers

*vi* allows the user to set markers within a document to allow quick navigation to various parts of the document. These markers are only valid for the length of the session and will not be restored when a file is exited and re-edited.

To set a marker the characters “a” - “z” are used to indicate the marks and to set a marker in command mode the *m* key is pressed followed by the mark character. For example to set a marker on a line using a as the marker the following command will be used.

```
        ma
```

To navigate to a set mark the ’ is used followed by the mark character as follows

```
        'a
```

Which will move the cursor to the mark position. Unfortunately *vi* does not indicate where the marks are set in the text, however if a mark does not exist the mode line will inform the user by displaying “*Mark not set*”.

## D.11 Other input modes

Besides inset mode *vi* has other modes which allow for the input of text. These are shown in table D.7 and *vi* will display the current mode on the *modeline.*

| **Command** | **Mode Name** | **Insertion Point** |
|---|---|---|
| a | append | just after the current character |
| A | Append | end of the current line |
| i | insert | just before the current character |
| I | Insert | beginning of the current line |
| o | open | new line below the current line |
| O | Open | new line above the current line |

Table D.7: Input Modes

## D.12 Cut, Copy and Paste

*vi* allows sections of text to be cut, copied or pasted to other parts of the document. First the text is cut or copied to a temporary buffer then it is pasted into a new location.

### D.12.1 Buffers

*vi* uses a buffer to store the temporary text. There are nine numbered buffers in addition to an undo buffer. The undo buffer contains the most recent delete. Usually buffer 1 contains the most recent delete, buffer 2 the next and so on until buffer 9, and deletes after 9 are lost. *vi* also has 26 named buffers (a-z) These buffers are useful for storing blocks of text for later retrieval. These buffers are independent of marker letters.

The contents of the buffer does not change until different text is put into the buffer. Unless the text is changed the buffer remains until the end of the session, as with the marker buffers the text buffers are lost when the current vi session ends.

### D.12.2 By line number

Two simple commands from the *ex* command set let the user cut and copy text by entering the range (in lines) and the destination line. The *m* command moves (cuts and pastes) a range of text, and the *t* command transfers (copies and pastes) it all of the commands have the following format

*:linemdestline* Move (cut) line number, line1, to the line just below line number, destline.

*:line1,line2mdestline* Move (cut) lines between line1 and line2 below line number, destline.

*:line1tdestline* Transfer (copy) line number, line1, to the line just below line number, destline.

*:line1,line2tdestline* Transfer (copy) lines between line1 and line2 below line number, destline.

### D.12.3 Cut and Copy (Delete and Yank)

*vi* calls cut and copy delete and yank respectively. When text is deleted or yanked it is possible to place it into a specified buffer. If no buffer is specified the default buffer is used (buffer 0).

The delete and yank commands take the following form

1. Move the cursor to one end of the desired text.
2. If desired, specify a named buffer by typing a buffer name (a-z), else *vi* uses the automatic buffers 1-9
3. Type a repetition number, if needed (to copy 5 words or 8 lines for example)
4. Type d to delete text or y to yank text
5. Type a cursor movement (see table ) or *dd* to delete lines or *yy* to yank lines.

For example to copy 17 lines from the current position use the following commands

```
     17yy
```

The mode line will then indicate how many lines were yanked.

### D.12.4 Using markers

Text may be deleted or yanked using buffers to do this use the following command sequence

1. Move the cursor to one end of the text to select
2. Type *m* letter to specify a buffer
3. Move the cursor to the other end of the text selection
4. If desired specify a buffer to save the text into using “ letter, if none is specified *vi* uses the automatic buffer.
5. Type *d* or *y* to delete or yank the text
6. Using letter from the marker type *’letter* to delete or yank the text between the mark and the current cursor location.

### D.12.5 Paste

To paste text from the buffer involves three steps as follows

1. Move the cursor to the desired pasting location
2. If retrieving text from a named buffer, specify the buffer by typing *"letter.* Otherwise *vi* uses the automatic buffers
3. Type *p* to paste the buffered text just after the current character or type *P* to paste it just before the current character.

## D.13 Search and Replace

*vi* can search the entire file for a given string of text by using the */* character followed by the desired string which will search forward form the current cursor position or backwards using the *?* key followed by the search string. To execute the search press the RETURN key.

To find the next occurrence of the string press the *n* key to move forwards to the next occurrence or *N* to move backwards. When *vi* reaches the end of the file it will move to the beginning of the file to the next / first occurrence of the string.

### D.13.1 Special characters

*vi* supports special characters which act as wildcards or search exclusions. These special characters are shown in table D.8.

| **Usage** | **Action** | **Example** | **Matches** |
|---|---|---|---|
| [cccc] | match any of the characters cccc | /sa[fn] | any string beginning with *sa* followed either by an *f* or an *n* |
| [^cccc] | match all characters except cccc | /[^a]nd | any string containing *nd* not preceded by an *a.* (i.e. not and) |
| [c1-c2] | match any character between c1 and c2 | /[d-h]er | any string containing *er* preceded by either *d e f g* or *h* |
| \<cccc | match words beginning cccc | /\<eac | any string beginning with *eac* |
| cccc\> | match words ending in cccc | /und\> | any string ending with *und* |
| ^cccc | match lines beginning with ccc | /^In | any line beginning with *In* |
| cccc$ | matches lines ending with cccc | /stop$ | any line ending with *stop* |
| . | match any single character | /i.l | any string with the character i[any char]l in sequence |
| c* | match any single character 0 or more times | /mb*d | any string containing b and d 0 or more times |
| .* | match any characters | /b.*k | any string containing b and k |

Table D.8: Special search characters

Note that cccc stands for any number of characters (including numbers) and most other characters. Special characters are $ . * [ ] ^ \. If these characters are required in the search string the backslash is used before the character to allow it to be used . For example if $14F0 is required in a search the command to search would be /\$14F0. To specify a single backslash use \\.

### D.13.2 Search and Replace

*vi* can also search and replace using one of the built in *ex* commands. The format of the command is as follows

```
:line1,line2s/oldstring/newstring
```

So if every occurrence of the text *if* were to be replaced with *else if* on lines 15 - 32 the following command would be used

```
15,32s/if/else if
```

If only one line number is specified the command only works on that line. If no line is specified then the action will take place on the current line. It must also be noted that only the first occurrence of the search string in the line is modified and any further occurrences are ignored. The search and replace may also be repeated using the *&* command on the current line or *:linenumnber&* will repeat the command for the line number passed, or to repeat across number of lines *line1,line2&*.

### D.13.3 Special flags

It is possible to add a flag to the search and replace command; the flag then tells *vi* to replace every occurrence or to ask for confirmation before each replacement. To add a flag the following form is used

```
:line1,line2s/oldstring/newstring /flag
```

c

If the flag is *c vi* will wait for conformation before each change and the user will have to press y or n followed by a RETURN to accept or reject the change. If the flag used is a *g* a global search and replace is executed this will replace every occurrence of the string in the current line (not just the first as in the previous example) this is known as a global replace.

### D.13.4 A more powerful Search and Replace

The *ex* command *g* can be used with the substitute command *s* to find and replace every occurrence of a string pattern in an entire file. The syntax for the global command is

```
:g/string/commands
```

The global command finds each line in the file that has string in it and then applies the commands to it. This can be applied in the following way

```
:g/oldstring/s//newstring/g
```

The oldstring does not have to be added to the in the search part of the command as it is already present in the global command.

## D.14 Variables

*vi* maintains several variables that controls different aspects of its appearance. Some of these have already been explained such as *:set number :set showmode*.

### D.14.1 Toggle and Numeric variables

The two types of variables are toggle variables and numeric variables. Toggle variables turn an option on or off (like displaying line numbers), while numeric variables take a number as an argument.

To turn on an off a toggle variable the following syntax is used

```
      :set variable
      :set novariable
```

Numeric variables are set with an equals sign and the corresponding value. For example to set the tab stops to be 4 spaces the following is typed

```
      :set tabstop=4
```

Table D.9 shows some of the variable which may be set.

| Variable | Default | Description |
|---|---|---|
| ignorecase | noignorecase | Do not distinguish between capitals and lower case letters in searches |
| number | nonumber | Display line numbers |
| showmode | noshowmode | Displays the input mode, bland for command mode. |
| wrapscan | wrapscan | When a search completes it goes back to the beginning of the file |
| report | report=5 | When more than this number of lines are change vi reports it |
| tabstop | tabstop=8 | Sets tab stops to multiples of this value |
| wrapmargin | wrapmargin=0 | Sets the right margin . |

Table D.9: vi variables

### D.14.2 Useful variables

The variables shown in table D.9 are only a small subsection of the variables used within *vi* to see all of the variables use the following command

```
      :set all
```

## D.15 Mapping keys

A single keystroke may be mapped to a sequence of commands, to do this the function keys are used with the following commands

```
      :map <F2> 1G
```

Note that the <F2> shown above will be printed on the mode line when the F2 key is pressed. The rest of the command tells *vi* to go to the first line of the file when F2 is pressed.

## D.16 Executing Console Commands

*vi* allows the user to execute Unix commands whilst within the editor. To do this the ! is used followed by the command to be executed as the following example shows

```
      :!sh
```

This will drop *vi* into the command shell and allow the user to execute any Unix commands required. To return to *vi* exit followed by RETURN must be typed.

# Appendix A

# Unix Commands

Command 1: cd [dir]

Usage :
change directory

Flags :
no flags

Examples :
cd /etc changes to the etc directory
It must be noted that cd is built into the shell

Command 2: chmod [options] mode files

Usage :
change the access mode of one or more files. Only the owner of the file or a super user may change the mode

Flags :
-R recursively descend directory arguments whilst setting modes

File permissions are set on the basis of User Group and world and each section may have a Read Write and eXecute bit set. These are set using an octal number for each of the three groups. To set each bit the following values are used

4 Read
2 Write
1 Execute

A fourth bit may be set which precedes the User Group World flags. These use the following octal values

4 sets the user ID on execution
2 Set the group ID on execution
1 set sticky bit

Examples :
chmod 700 * set all files to have rwx permissions for owner and no permissions for group and world

chmod 755 * set file permissions to rwxr-xr-x for all files

---

Command 3: cp [options] file1 file2
cp [options] files directory

---

Usage :
Copy file1 to file2 or copy one or more files to the same names to a directory

---

Flags :
-i interactive mode (prompts for y/n for each file
-r recursively copy a directory, its files and subdirectories

---

Examples :
cp test.c test.c.old copy the file test.c to a new file test.c.old
cp * ./backup copy all the files in the current directory to a directory called backup

---

Command 4: du [options] [directories]

---

Usage :
prints the disk usage of the directory specified or present directory if not specified.

---

Flags :
-a print usage for all files not just subdirectories
-s printf on the grand total for cache named directory (i.e. silent mode)
-k print disk usage in K bytes not blocks

---

Examples :
du -ks print the total disk usage for the current directory

---

Command 5: find pathname(s) condition(s)

---

Usage :
Used to find files, find has numerous uses dependant upon the conditions set in the command line

---

Flags :
-exec command{} execute a unix command on finding a file
-name find a file with a specific name
-ok same as exec but prompts for y / n

---

Examples :
find ./ -name "*.c" finds all files with a .c extention
find ./ -name "*.o" -ok {} \;

---

Command 6: grep [options] [regexp] [files]

---

Usage :
search one or more files for lines that match the regular expression regexp

---

Flags :
-c print out a count of matched lines
-i ignore case

-l list file names not matched lines
-s suppress error messages

---

Examples :
grep main * find all files which contain the phrase main
grep -i myFunction *.c search for the text myFunction ignoring case in all files in the current directory

---

Command 7: gunzip [options] filename.gz

---

Usage :
unzip a GNU zipped file

---

Flags :
-l list contents but dont unzip file

---

Examples :
gunzip test.gz unzip the file test.gz

---

Command 8: gzip [options] filename.gz

---

Usage :
create a GNU zipped file

---

Flags :
-# 1 -9 compression ration 1 == fast 9 == best compression

---

Examples :
gzip -9 test.tar compress the file test.tar

---

Command 9: head [-n] [files]

---

Usage :
print the first n lines of a file

---

Flags :
-n number of lines to print

---

Examples :
head -n1 /etc/* prints the first line of every file in /etc

---

Command 10: lp / lpr [options] files

---

Usage :
sends files to print spooler

---

Flags :
-P [name] specifies the name of the printer
-#n number of copies to print

---

Examples :
lpr -P DrEvil notes.ps will print the file notes.ps

---

Command 11: ls [options] [names]

Usage :
List information obout files - current directory is used by default

Flags :
-l list in long format
-a show all files
-R list subdirectories recursively
-c list by file creation / modification time
-d show directories

Examples :
ls -al list all files in a directory
ls -lR list all files including subdirectories
ls -ld /bin /etc list the status of directories /bin and /etc
ls *.c list all of the .c files in the current directory

Command 12: mkdir [options] directories

Usage :
make a directory(s)

Flags :
-m mode used to set the access mode for the new directory
-p create parent directories as needed

Examples :
mkdir test creates a directory called test
mkdir -p /test/d1/old creates the whole directory structure
mkdir -m 700 test creates a directory called test with rwx—— permissions (see chmod for more details on permissions

Command 13: more [options] files

Usage :
Displays the named files in the console one screen at a time

Flags :
use the space key to scroll pages
PgUP moves up
PgDn moves down
q exits

Examples :
more /etc/passwd displays the contents of the /etc/passwd file

Command 14: mv [options] sources target

Usage :

mv is used to move or rename files.

---

Flags :
-i interactive mode prompt user y/n
-f force move even if target file exists

---

Examples :
mv file1.c file2.c renames file1.c file2.c
mv * ./backup moves all files to the backup directory
mv mydir myolddir rename a directory for mydir to myolddir

---

Command 15: pwd

---

Usage :
print working directory

---

Flags :
no flags

---

Examples :
pwd will print the current directory within the shell

---

Command 16: rm [options] files

---

Usage :
delete on or more files.

---

Flags :
-f force removal
-i interactive mode prompt y/n
-r recurse subdirectories

---

Examples :
rm *.c remove all c files from the current directory
rm -rf mydir remove contents of mydir as well as the directory itself
rm -rf * remove every thing in the current directory downward

---

Command 17: rmdir [options] directories

---

Usage :
remove directory

---

Flags :
-P recurse subdirectories
-s suppress standard error messages

---

Examples :
rmdir temp removes the temp directory

---

Command 18: tar [options] files

---

Usage :

create a tape archive (or a file in the current directory)

---

Flags :
v verbose mode
f specify file name and do not look for tape drive
c create a tape archive (tar file)
x extract an existing tar file

---

Examples :
tar cfv mydir.tar ./mydir/* create a tar file of the directory mydir called mydir.tar
tar vfx mydir.tar extracts the contents of the tar file mydir.tar

# Appendix B

# vi reference

## Legenda

| | |
|---|---|
| default values | 1 |
| <*> | '*' must not be taken literally |
| [*] | '*' is optional |
| ^X | <ctrl>X |
| <sp> | space |
| <cr> | carriage return |
| <lf> | linefeed |
| <ht> | horizontal tab |
| <esc> | escape |
| <erase> | your erase character |
| <kill> | your kill character |
| <intr> | your interrupt character |
| <a-z> | an element in the range |
| N | number ('*' = allowed, '-' = not appropriate) |
| CHAR | char unequal to <ht>\|<sp> |
| WORD | word followed by <ht>\|<sp>\|<lf> |

## Searching

| **Command** | **Meaning** |
|---|---|
| | |
| :ta <name> | Search in the tags file[s] where <name> is defined (file, line), and go to it. |
| ^] | Use the name under the cursor in a ':ta' command. |
| ^T | Pop the previous tag off the tagstack and return to its position. |
| :[x,y]g/<string>/<cmd> | Search globally [from line x to y] for <string> and execute the 'ex' <cmd> on each occurrence. Multiple <cmd>'s are separated by '\|'. |
| :[x,y]g/<s1>/,/<s2>/<c> | Search globally [from line x to y] for <s1> and execute the 'ex' command <c> on each line between <s1> and the line that matches <s2>. |
| :[x,y]v/<string>/<cmd> | Execute <cmd> on the lines that don't match. |

## Undoing changes

| **Command** | **Meaning** |
|---|---|
| | |
| u | Undo the latest change. |
| U | Undo all changes on a line, while not having moved off it (unfortunately). |
| :q! | Quit vi without writing. |
| :e! | Re-edit a messed-up file. |

## Deleting text

Everything deleted can be stored into a buffer. This is achieved by putting a "'' and a letter <a-z> before the delete command. The deleted text will be in the buffer with the used letter. If <A-Z> is used as buffer name, the conjugate buffer <a-z> will be augmented (*i.e.*, appended) instead of overwritten with the text. The undo buffer always contains the latest change. Buffers <1-9> contain the latest 9 LINE deletions ("'1' is most recent). See also 'remembering text'.

| **N** | **Command** | **Meaning** |
|---|---|---|
| | | |
| * | x | Delete <*> chars under and after the cursor. |
| * | X | <*> chars before the cursor. |
| * | d<move> | From begin to endpoint of <*><move>. |
| * | dd | <*> lines. |
| - | D | The rest of the line. |
| * | <<move> | Shift the lines described by <*><move> one shiftwidth to the left. |
| * | << | Shift <*> lines one shiftwidth to the left. |
| * | . | Repeat latest command <*> times. |
| - | :[x,y]d | Delete lines x through y (default current line and next). |

## Move commands

| N | Command | Meaning | | |
|---|---|---|---|---|
| | | | | |
| * | h | ˆH | <erase> | <*> chars to the left. |
| * | j | <lf> | ˆN | <*> lines downward. |
| * | l | <sp> | <*> chars to the right. | |
| * | k | ˆP | <*> lines upward. | |
| * | $ | To the end of line <*> from the cursor. | | |
| - | ˆ | To the first CHAR of the line. | | |
| * | _ | To the first CHAR <*> - 1 lines lower. | | |
| * | - | To the first CHAR <*> lines higher. | | |
| * | + | <cr> | To the first CHAR <*> lines lower. | |
| - | 0 | To the first char of the line. | | |
| * | \| | To column <*> (<ht>: only to the endpoint). | | |
| * | f<char> | <*> <char>s to the right (find). | | |
| * | t<char> | Till before <*> <char>s to the right. | | |
| * | F<char> | <*> <char>s to the left. | | |
| * | T<char> | Till after <*> <char>s to the left. | | |
| * | ; | Repeat latest ‘f’\|‘t’\|‘F’\|‘T’ <*> times. | | |
| * | , | Idem in opposite direction. | | |
| * | w | <*> words forward. | | |
| * | W | <*> WORDS forward. | | |
| * | b | <*> words backward. | | |
| * | B | <*> WORDS backward. | | |
| * | e | To the end of word <*> forward. | | |
| * | E | To the end of WORD <*> forward. | | |
| * | G | Go to line <*> (default EOF). | | |
| * | H | To line <*> from top of the screen (home). | | |
| * | L | To line <*> from bottom of the screen (last). | | |
| * | M | To the middle line of the screen. | | |
| * | ) | <*> sentences forward. | | |
| * | ( | <*> sentences backward. | | |
| * | } | <*> paragraphs forward. | | |
| * | { | <*> paragraphs backward. | | |
| - | ]] | To the next section (default EOF). | | |
| - | [[ | To the previous section (default begin of file). | | |
| - | ‘<a-z> | To the mark. | | |
| - | ’<a-z> | To the first CHAR of the line with the mark. | | |
| - | “ | To the cursor position before the latest absolute jump (of which are examples ‘/’ and ‘G’). | | |
| - | ” | To the first CHAR of the line on which the cursor was placed before the latest absolute jump. | | |
| - | /<string> | To the next occurrence of <string>. | | |
| - | ?<string> | To the previous occurrence of <string>. | | |
| - | /<string>/+[n] | To n-th (default 1st) line after next occurrence of <string>. | | |
| - | ?<string>?+[n] | Idem, searching in the opposite direction. | | |
| - | /<string>/-[n] | To n-th (default 1st) line before next occurrence of <string>. | | |
| - | ?<string>?-[n] | Idem, searching in the opposite direction. | | |
| - | <find>[;<find>] | Perform successive ‘/’\|‘?’ actions. For example, /foo/;/bar - to next ‘foo’, then to next ‘bar’ ?foo?-;/bar - to line before previous ‘foo’, then to next ‘bar’ | | |
| - | n | Repeat latest ‘/’\|‘?’ (next). | | |
| - | N | Idem in opposite direction. | | |
| - | % | Find the next bracket and go to its match (also with ‘{’\|‘}’ and ‘[’\|‘]’). | | |

## Appending text (end with <esc>)

| N | Command | Meaning |
|---|---|---|
| | | |
| * | a | <*> times after the cursor. |
| * | A | <*> times at the end of line. |
| * | i | <*> times before the cursor (insert). |
| * | I | <*> times before the first CHAR of the line |
| * | o | On a new line below the current (open). The count is only useful on a slow terminal. |
| * | O | On a new line above the current. The count is only useful on a slow terminal. |
| * | ><move> | Shift the lines described by <*><move> one shiftwidth to the right. |
| * | >> | Shift <*> lines one shiftwidth to the right. |
| * | [“<a-zA-Z1-9>]p | Put the contents of the (default undo) buffer <*> times after the cursor. A buffer containing lines is put only once, below the current line. See ‘deleting text’. |
| * | [“<a-zA-Z1-9>]P | Put the contents of the (default undo) buffer <*> times before the cursor. A buffer containing lines is put only once, above the current line. See ‘deleting text’. |
| * | . | Repeat previous command <*> times. If the last command before a ‘.’ command references a numbered buffer, the buffer number is incremented first (and the count is ignored): |
| - | :[x,y]t<l> | Copy lines x through y (default current line) to be after line <l>. See ‘remembering text’. |

## Changing text (end with <esc>)

| N | Command | Meaning |
|---|---|---|
| | | |
| * | r<char> | Replace <*> chars by <char> - no <esc>. |
| * | R | Overwrite the rest of the line, appending change <*> - 1 times. |
| * | s | Substitute <*> chars. |
| * | S | <*> lines. |
| * | c<move> | Change from begin to endpoint of <*><move>. |
| * | cc | <*> lines. |
| * | C | The rest of the line and <*> - 1 next lines. |
| * | =<move> | If the option ‘lisp’ is set, this command will realign the lines described by <*><move> as though they had been typed with the option ‘ai’ set too. |
| - | | Switch lower and upper cases (should be an operator, like ‘c’). |
| * | J | Join <*> lines (default 2). |
| * | . | Repeat latest command <*> times (‘J’ only once). |
| - | & | Repeat latest ‘ex’ substitute command, e.g. ‘:s/wrong/good’. |
| - | :[x,y]j | Join lines x through y (default current line and next). |
| - | :[x,y]j! | Idem, but with no space inbetween. |
| - | :[x,y]m<l> | Move lines x through y (default current line) to be after line <l>. See ‘remembering text’. |
| - | :[x,y]s/<p>/<r>/<f> | Substitute (on lines x through y) the pattern <p> (default the last pattern) with <r>. Useful flags <f> are ‘g’ for ‘global’ (i.e. change every non-overlapping occurrence of <p>) and ‘c’ for ‘confirm’ (type ‘y’ to confirm a particular substitution, else <cr>). Instead of ‘/’ any punctuation CHAR unequal to <lf> can be used as delimiter. |

## substitute replacement patterns

The basic meta-characters for the replacement pattern are ‘&’ and ‘ ’; these are given as ‘\&’ and ‘\ ’ when ‘nomagic’ is set. Each instance of ‘&’ is replaced by the characters which the regular expression matched. The meta-character ‘ ’ stands, in the replacement pattern, for the defining text of the previous replacement pattern. Other meta-sequences possible in the replacement pattern are always introduced by the escaping character ‘\’. The sequence ‘\n’ (with ‘n’ in [1-9]) is replaced by the text matched by the n-th regular subexpression enclosed between ‘\(’ and ‘\)’. The sequences ‘\u’ and ‘\l’ cause the immediately following character in the replacement to be converted to upper- or lower-case respectively if this character is a letter. The sequences ‘\U’ and ‘\L’ turn such conversion on, either until ‘\E’ or ‘\e’ is encountered, or until the end of the replacement pattern. See the ‘magic’ option for additional meta-characters. Some examples of substitutions are shown below.

| | |
|---|---|
| :s/foo/\u& | - turn ‘foo’ into ‘Foo’ |
| :s/foo/\U& | - turn ‘foo’ into ‘FOO’ |
| :s/\(foo\) \(bar\)/\U\1\E \u\2 | - turn ‘foo bar’ into ‘FOO Bar’ |
| :s/foo/\u&/\|s/bar/ | - capitalize foo, then capitalize bar |

## Remembering text (yanking)

With yank commands you can put “‘<a-zA-Z>’ before the command, just as with delete commands (see ‘deleting text’). Otherwise you only copy to the undo buffer. Using the capital letters appends to the buffer. The use of buffers <a-z> is THE way of copying text to another file; see the ‘:e <file>’ command.

| N | Command | Meaning | | |
|---|---|---|---|---|
| | | | | |
| * | y<move> | Yank from begin to endpoint of <*><move>. | | |
| * | yy | <*> lines. | | |
| * | Y | Idem (should be equivalent to ‘y$’ though). | | |
| - | :[x,y]y<a-zA-Z> | Yank lines x through y into named bufer. Using the capital letter will append to the buffer. | | |
| - | m<a-z> | Mark the cursor position with a letter. | | |
| - | :[x]k<a-z> | Mark line x (default current) with a letter. The letter can be used to refer to the line in another ex command: | :/aaa/ka<br>:’a,’a+3d<br>:?bbb?kb<br>:’bm. | - mark next line matching aaa<br>- delete that line and the three following it<br>- mark previous line matching bbb<br>- move that line to be after current line |

## Commands while in append|change mode

| Command | Meaning | |
|---|---|---|
| | | |
| ˆ@ | If typed as the first character of the insertion, it is replaced with the previous text inserted (max. 128 chars), after which the insertion is terminated. | |
| ˆV | Deprive the next char of its special meaning (e.g. <esc>). | |
| ˆD | One shiftwidth to the left, but only if nothing else has been typed on the line. | |
| 0ˆD | Remove all indentation on the current line (there must be no other chars on the line). | |
| ˆˆD | Idem, but it is restored on the next line. | |
| ˆT | One shiftwidth to the right, but only if nothing else has been typed on the line. | |
| ˆH | <erase> | One char back. |
| ˆW | One word back. | |
| <kill> | Back to the begin of the change on the current line. | |
| <intr> | Like <esc> (but you get a beep as well). | |

## Writing, editing other files, and quitting vi

In ‘:’ ‘ex’ commands - if not the first CHAR on the line - ‘%’ denotes the current file, ‘#’ is a synonym for the alternate file (which normally is the previous file). As first CHAR on the line ‘%’ is a shorthand for ‘1,$’. Marks can be used for line numbers too: ’<a-z>. In the ‘:w’|‘:f’|‘:cd’|‘:e’|‘:n’ commands shell meta-characters can be used.

| Command | Meaning |
|---|---|
| | |
| :q | Quit vi, unless the buffer has been changed. |
| :q! | Quit vi without writing. |
| ˆZ | Suspend vi. |
| :w | Write the file. |
| :w <name> | Write to the file <name>. |
| :w >> <name> | Append the buffer to the file <name>. |
| :w! <name> | Overwrite the file <name>. |
| :x,y w <name> | Write lines x through y to the file <name>. |
| :wq | Write the file and quit vi; some versions quit even if the write was unsuccessful! Use ‘ZZ’ instead. |
| ZZ | Write if the buffer has been changed, and quit vi. If you have invoked vi with the ‘-r’ option,<br>you’d better write the file explicitly (‘:w’ or ‘:w!’), or quit the editor explicitly (‘:q!’) if you don’t want to overwrite the file<br>- some versions of vi don’t handle the ‘recover’ option very well. |
| :x [<file>] | Idem [but write to <file>]. |
| :x! [<file>] | ‘:w! [<file>]’ and ‘:q’. |
| :pre | Preserve the file - the buffer is saved as if the system had just crashed; for emergencies,<br>when a ‘:w’ command has failed and you don’t know how to save your work (see ‘vi -r’). |
| :f <name> | Set the current filename to <name>. |
| :cd [<dir>] | Set the working directory to <dir> (default home directory). |
| :cd! [<dir>] | Idem, but don’t save changes. |
| :e [+<cmd>] <file> | Edit another file without quitting vi - the buffers are not changed (except the undo buffer),<br>so text can be copied from one file to another this way. [Execute the ‘ex’ command <cmd> (default ‘$’)<br>when the new file has been read into the buffer.] <cmd> must contain no <sp> or <ht>. See ‘vi startup’. |
| :e! [+<cmd>] <file> | Idem, without writing the current buffer. |
| ˆˆ | Edit the alternate (normally the previous) file. |
| :rew | Rewind the argument list, edit the first file. |
| :rew! | Idem, without writing the current buffer. |
| :n [+<cmd>] [<files>] | Edit next file or specify a new argument list. |
| :n! [+<cmd>] [<files>] | Idem, without writing the current buffer. |
| :args | Give the argument list, with the current file between ‘[’ and ‘]’. |

## Display commands

| **N** | **Command** | **Meaning** |
|---|---|---|
| | | |
| - | ˆG | Give file name, status, current line number and relative position. |
| - | ˆL | Refresh the screen (sometimes ‘ˆP’ or ‘ˆR’). |
| - | ˆR | Sometimes vi replaces a deleted line by a ‘@’, to be deleted by ‘ˆR’ (see option ‘redraw’). |
| * | ˆE | Expose <*> more lines at bottom, cursor stays put (if possible). |
| * | ˆY | Expose <*> more lines at top, cursor stays put (if possible). |
| * | ˆD | Scroll <*> lines downward (default the number of the previous scroll; initialization: half a page). |
| * | ˆU | Scroll <*> lines upward (default the number of the previous scroll; initialization: half a page). |
| * | ˆF | <*> pages forward. |
| * | ˆB | <*> pages backward (in older versions ‘ˆB’ only works without count). |
| - | :[x,y]l | List lines x through y (default current), making invisible characters visible. |
| - | :[x,y]p | Print lines x through y (default current). |
| - | :[x,y]nu | List lines x through y (default current), with line numbers next to each line. |

If in the next commands the field <wi> is present, the windowsize will change to <wi>. The window will always be displayed at the bottom of the screen.

| **N** | **Command** | **Meaning** |
|---|---|---|
| | | |
| * | z[wi]<cr> | Put line <*> at the top of the window (default the current line). |
| * | z[wi]+ | Put line <*> at the top of the window (default the first line of the next page). |
| * | z[wi]- | Put line <*> at the bottom of the window (default the current line). |
| * | z[wi]ˆ | Put line <*> at the bottom of the window (default the last line of the previous page). |
| * | z[wi]. | Put line <*> in the centre of the window (default the current line). |

## Switch and shell commands

| **N** | **Command** | **Meaning** | | |
|---|---|---|---|---|
| | | | | |
| - | Q | ˆ\ | <intr><intr> | Switch from vi to ‘ex’. |
| - | : | An ‘ex’ command can be given. | | |
| - | :vi | Switch from ‘ex’ to vi. | | |
| - | :sh | Execute a subshell, back to vi by ‘ˆD’. | | |
| - | :[x,y]!<cmd> | Execute a shell <cmd> [on lines x through y; these lines will serve as input for <cmd> and will be replaced by its standard output]. | | |
| - | :[x,y]!! [<args>] | Repeat last shell command [and append <args>]. | | |
| - | :[x,y]!<cmd> ! [<args>] | Use the previous command (the second ‘!’ in a new command. | | |
| * | !<move><cmd> | The shell executes <cmd>, with, as standard input, the lines described by <*><move>, next the standard output replaces those lines. | | |
| * | !<move>!<args> | Append <args> to the last <cmd> and execute it, using the lines described by the current <*><move>. | | |
| * | !!<cmd> | Give <*> lines as standard input to the shell <cmd>, next let the standard output replace those lines. | | |
| * | !!! [<args>] | Use the previous <cmd> [and append <args> to it]. | | |
| - | :x,y w !<cmd> | Let lines x to y be standard input for <cmd> (notice the <sp> between the ‘w’ and the ‘!’). | | |
| - | :r!<cmd> | Put the output of <cmd> onto a new line. | | |
| - | :r <name> | Read the file <name> into the buffer. | | |

## The most important options

| **Option** | **Meaning** |
|---|---|
| | |
| ai | autoindent - In append mode after a <cr> the cursor will move directly below the first CHAR on the previous line.<br>However, if the option ‘lisp’ is set, the cursor will align at the first argument to the last open list. |
| aw | autowrite - Write at every shell escape (useful when compiling from within vi). |
| dir=<string> | directory - The directory for vi to make temporary files (default ‘/tmp’). |
| eb | errorbells - Beeps when you goof (not on every terminal). |
| ic | ignorecase - No distinction between upper and lower cases when searching. |
| lisp | Redefine the following commands: ‘(’, ‘)’ - move backward (forward) over S-expressions<br>‘{’, ‘}’ - idem, but don’t stop at atoms<br>‘[[’, ‘]]’ - go to previous (next) line beginning with a ‘(’<br>See option ‘ai’. |
| list | <lf> is shown as ‘$’, <ht> as ‘ˆI’.<br>Instead of <sp> a <ht> can be used, instead of ‘vi’ there can be ‘ex’. |
| nu | number - Numbers before the lines. |
| para=<string> | paragraphs - Every pair of chars in <string> is considered a paragraph delimiter nroff macro (for ‘{’ and ‘}’).<br>A <sp> preceded by a ‘\’ indicates the previous char is a single letter macro.<br>‘:set para=P\ bp’ introduces ‘.P’ and ‘.bp’ as paragraph delimiters.<br>Empty lines and section boundaries are paragraph boundaries too. |
| redraw | The screen remains up to date. |
| remap | If on (default), macros are repeatedly expanded until they are unchanged.<br>Example: if ‘o’ is mapped to ‘A’, and ‘A’ is mapped to ‘I’, then ‘o’ will map to ‘I’ if ‘remap’ is set,<br>else it will map to ‘A’. |
| report=<*> | Vi reports whenever e.g. a delete or yank command affects <*> or more lines. |
| ro | readonly - The file is not to be changed. However, ‘:w!’ will override this option. |
| sect=<string> | sections - Gives the section delimiters (for ‘[[’ and ‘]]’); see option ‘para’.<br>A ‘{’ beginning a line also starts a section (as in C functions). |
| sh=<string> | shell - The program to be used for shell escapes (default ‘$SHELL’ (default ‘/bin/sh’)). |
| sw=<*> | shiftwidth - Gives the shiftwidth (default 8 positions). |
| sm | showmatch - Whenever you append a ‘)’, vi shows its match if it’s on the same page;<br>also with ‘{’ and ‘}’. If there’s no match at all, vi will beep. |
| taglength=<*> | The number of significant characters in tags (0 = unlimited). |
| tags=<string> | The space-separated list of tags files. |
| terse | Short error messages. |
| to | timeout - If this option is set, append mode mappings will be interpreted only if they’re typed fast enough. |
| ts=<*> | tabstop - The length of a <ht>; warning: this is only IN the editor,<br>outside of it <ht>s have their normal length (default 8 positions). |
| wa | writeany - No checks when writing (dangerous). |
| warn | Warn you when you try to quit without writing. |
| wi=<*> | window - The default number of lines vi shows. |
| wm=<*> | wrapmargin - In append mode vi automatically puts a <lf> whenever<br>there is a <sp> or <ht> within <wm> columns from the right margin<br>(0 = don’t put a <lf> in the file, yet put it on the screen). |
| ws | wrapscan - When searching, the end is considered ‘stuck’ to the begin of the file. |

## + changing and viewing options

| Command | Meaning |
|---|---|
| | |
| :set <option> | Turn <option> on. |
| :set no<option> | Turn <option> off. |
| :set <option>=<value> | Set <option> to <value>. |
| :set | Show all non-default options and their values. |
| :set <option>? | Show <option>'s value. |
| :set all | Show all options and their values. |