Contents

1 Useful Linux Command Info

1.1 Looking around directories

There are a number of commands that are useful in navigating the file system. These may have specific options that are not remembered. They are the following.

1.1.1 ls

To list the items in a directory, use the ls command. This command takes on the following form:

```
ls [options] {directory-path}
```

Essentially, you can print the contents of a given directory (*directory-path* if specificed, else the working directory) to the shell. The output can be modified with certain options, which are as follows:

- *-l* can be used to print out the long form of the listings, which will include the permissions, number of references, owner, group, size and last modification time.
- *-a* can be used to print out all items in the directory (that the user has read permission for), including the hidden files that begin with a full stop, like the current directory '.', and its parent, '..'
- *-i* prepends the inode number to the beginning of the list. This is the numerical address of the inode that corresponds to the item.
- *-h* makes the size in a long listing human readable, by using K, M, G, etc for powers of 2^{10}
- *-r* reverses the order of the listings.

1.1.2 cat

To print a file's contents to standard output (the shell), use cat. The command takes the following form:

```
cat [options] {file}
```

This prints out the contents of *file* (if specified, else stdin), but may change form depending on optional specifiers, which could be:

- *-n* will indent every line and prepend it with a line number. Alternatively, *-b* does the same, but skips numbering blank lines.
- *-E* marks the end of every line with a $ character.
- *-T* replaces all tab characters with '^I'
- *-s* suppresses duplicate empty lines (ie if there are multiple empty lines consecutively, it only prints one). Does this prior to any numbering from *-n*.

1.1.3 less

Useful when you are to be looking at a lot of information and don't want it to mess with the terminal output. Operates like man pages, where you move with the mouse, and exit back to a clean terminal with 'q' afterwards. For example, say there are a lot of files in the directory 'things', then you can see all the details in a more concentrated and cleaner form by using:

```
ls -al things | less
```

1.1.4 Wildcards

Where you want to leave some unknowns, there are two wildcards to use: '*', which will represent any number of characters as needed, or; '?' which will represent one character. For example:

```
# Open sample
$ ls
aardvark app az beachball test test. test.c test.h test.o test.py

# Usage of the * wildcard
$ ls a*
aardvark app az

# Usage of the ? wildcard
$ ls test.?
test.c test.h test.o
```

1.1.5 grep

In order to filter file contents, use the 'grep' command. The command takes the following form:

```
grep [options] {patterns} [file(s)]
```

The general idea is that grep will return all lines from the file(s) specified (or stdin if no file is specified) that contain the given patterns. If multiple patterns are desired, they shall be newline delimited, or specified using one of the optional arguments. The options are as follows:

- *-e* can be used to specify a pattern to look for. Useful where multiple patterns are desired, each can be specified using a *-e* argument.
- *-f* can be used to specify a file containing a set of patterns. For multiple patterns, these are newline delimited.
- *-i* can be used to specify that the pattern is case insensitive, so that in instances where the only difference is case, they shall still be recognised.
- *-v* is used to invert the selection criteria. That is, only select lines that do not contain the specified patterns.
- *-w* specifies that the string must exist as its own word (that is, it is either at the end of a line, or is surrounded by non-alphanumeric characters).
- *-x* specifies that the match must be the whole line.
- *-r* is used when you want to search the contents of every file in a directory.
- *-c* is used to return the number of lines that include the pattern in each file.

These can be demostrated as follows:

```
# Use grep with multiple arguments with no file (ie, with stdin)
$ grep -e frog -e dog
> frogfish fly
frogfish fly
> fishfly do g
> ^d

# Check for lines not including 'ape'
$ grep -v ape
> grape
> frog
frog
> paper
> ^d

# Create a file with patterns
$ cat > patterns
```

```
> animals
> frog
> turtle
> ^d

# Create a file to read from
$ cat > input
> This is a file.
> Read through me to find matching patterns.
> This one likes different animals.
> Like frog and turtle.
> ^d

# Compare the patterns from the file to the contents of the input file
$ grep -f patterns input
This one likes different animals.
Like frog and turtle.
```

1.2 Ending processes

The obvious option is ctrl-C (or ^C), which will send a SIGINT to the process. Alternatively, use ctrl-Z (or ^Z) to 'suspend' the program. This then means that the program can be resumed using 'fg'. For example:

```
# Run a program
$ ./program

# If the program is taking time, and you need to check something
> ^Z
[1]+  Stopped                 ./program

# Check that the process is still running
$ ps
PID TTY          TIME CMD
  9 pts/0    00:00:00 bash
269 pts/0    00:00:00 program
270 pts/0    00:00:00 ps

# Resume the program
$ fg ./program
./program

# Program doesn't terminate, do it yourself
> ^C
```

1.3 File permissions

Files have permissions at each the user, group, and other levels. They can be seen in the long list view:

```
# The start of each line is the permissions
$ ls -l
-rw-r--r-- 1 owner group   162 Apr  6 13:10 Makefile
drwxr-xr-x 4 owner group  4096 Apr  6 22:32 misc-programs
-rwxr-xr-x 1 owner group 16480 Apr  6 14:40 prog
-rw-r--r-- 1 owner group    93 Jun  1 20:28 test
```

These permissions can be edited using the 'chmod' command. This will need to specify the permission group to affect

- 'u' - owner
- 'g' - group
- 'o' - other
- 'a' - all

What to do with the permission

- '-' - remove
- '=' - set
- '+' - add

The permission to change

- 'r' - read [4]
- 'w' - write [2]
- 'x' - execute [1]

The impact can be demonstrated with an example (continued from the above directory):

```
# Add write perms to the group Makefile belongs to
$ chmod g+w Makefile

# Remove execute perms from everyone but the owner of misc-programs
$ chmod go-x misc-programs

# Make it so that the owner can only read prog, and no-one else gets any perms for it
$ chmod u=r,go= test

# Note the changes
$ ls -l
-rw-rw-r-- 1 owner group   162 Apr  6 13:10 Makefile
drwxr--r-- 4 owner group  4096 Apr  6 22:32 misc-programs
-rwxr-xr-x 1 owner group 16480 Apr  6 14:40 prog
-r-------- 1 owner group    93 Jun  1 20:28 test
```

These can also be specified when creating a directory by using the *-m* option, followed by the same formatting as *chmod* takes, for example:

```
# Make a new directory, with set permissions
$ mkdir -m u=rwx,go=r thing

# Note the permissions on the new file
$ ls -l
-rw-rw-r-- 1 owner group   162 Apr  6 13:10 Makefile
drwxr--r-- 4 owner group  4096 Apr  6 22:32 misc-programs
-rwxr-xr-x 1 owner group 16480 Apr  6 14:40 prog
-r-------- 1 owner group    93 Jun  1 20:28 test
drwxr--r-- 1 owner group  4096 Jun  2 08:02 thing
```

1.4 File & Directory Linking

Like shortcuts in windows, you can links around the file system. There are two main types, hard links and symbolic links. Both are controlled by the 'ln' command, which takes the form:

```
ln [options] {target} [destination]
```

This will by default create a hard link to target with the name and path specified by destination (if included, else will be in the working directory with the basename of the target). This can be further specified with some given arguments:

- *-s* can be used to make a symbolic link. Particularly useful for directory linking.
- *-v* provides a command output that outputs the link added.

2 Subversion

Subversion is a version control system. Good to know the commands that it uses. In general, it takes on the form:

```
svn {subcommand} [options] [args]
```

The useful subcommands to know are as follows.

2.1 add

This adds a file to the repository, to be tracked in future commits. It specifically takes the form:

```
svn add [files]
```

The file(s) provided are to be a space delimited list.

2.2 checkout

This checks out a working copy of the repository. It takes the form of:

```
svn checkout {url[@revision]} [destination]
```

The URL is for the host location of the repository, and can optionally specify the specific revision to check out (defaults to the most recent if not specified). The destination for the working repo, if not provided, defaults to the basename of the repo in the current working directory.

2.3 commit

This commits any updates in the working copy to the repository. Only looks at files that have been added. Takes the form:

```
svn commit [arguments]
```

The arguments should provide a commit message to contextualise the update made. This can be done one of two ways:

- *-m* can be used to specify the message on the commandline. Best surrounded by quotations to allow for spaces.
- *-F* can be used to specify a file from which the commit message can be read.

2.4 diff

Prints the changes between revisions. Takes the form:

```
svn diff [arguments]
```

Without arguments, this compares the local changes to the most recent commit. Otherwise, it can take on the following arguments:

- *-r* is used to specify a revision to compare to. This can be one revision number to compare local changes to, or two colon-delimited revision numbers to compare between.
- *-c* is used to see the changes that the specified revision changed, *-c x* is equivalent to *-r x:x-1*.

2.5 revert

Restores the working copy to the most recently committed state. Takes the form:

```
svn revert
```

2.6 status

Shows the status of each file in the repository. Takes the form:

```
svn status
```

Files can be of the following states:

- 'A' for added, was not in the last commit.
- 'D' for deleted, will not be in the next commit.
- 'M' for modified, has been changed since the last commit, was present at both times.
- '?' for not under version control. The file is in the working copy, but has not been added to the version control.

2.7 update

Updates the working copy to be in line with the repository. Discards any changes that haven't been commited. Takes the form:

```
svn update [-r REVISION]
```

A revision number can optionally be provided, to change the working directory to whichever revision is specified. Else, it will be the most recent commit.

2.8 log

Show the updates made to the specified file. Takes the form:

```
svn log [options] {path[@rev]}
```

Provides the log history of the file or directory found at *path*. A revision can be appended to the path similar to checkout to just show the log message for that revision, and options can be specified, including:

- *-r* is an alternative to the *@rev* option, that works like the *-r* argument in diff.
- *-l* can be used to specify the maximum number of log entries to document.

3 Networking

With networking questions, there are a few main things that should be known. This is a summary of some of the factual information that may be needed and embedded into questions.

- On unix systems, ports below 1024 are restricted, and are a 16 bit integer.
- IP address standard is big-endian (linux x86 uses little-endian).
- Port zero, the ephemeral port, assigns you an open port.
- IPv4 addresses are 32 bits long, typically expressed in bytes in base 10, delimited by a decimal place.
- Subnets have two reserved addresses, the "network address", where all host bits are zero, and the "broadcast address" where all host bits are one.
- Valid subnet masks must be a block of ones followed by a block of zeros, no intermingling.
- Some addresses are link local, and aren't used on the public internet. These are:
  - 10.0.0.0 / 8
  - 172.16.0.0 / 12
  - 192.168.0.0 / 16
  - 169.254.0.0 / 16
  - 127.0.0.0 / 8
  - 224.0.0.0 / 3
- There are two main IP representations of the network:
  - Netmask sets all of the bits that are conserved among accessible network addresses to one, and the variable bits to zero. This is such that a bitwise and with any of the members of the subnet will yield the same address.
  - CIDR keeps all constant bits, and sets the variable bits to zero. It also notes how many bits are constant.

4 Memory & Pages

Virtual memory operates on the principal of virtual addresses that map to physical addresses. These addresses are broken down into either page and offset (virtual) or frame and offset (physical). The size of a page tells you how many bytes are allocated to the offset, with the remaining representing the page number. The conversion from virtual address to physical address is:

$$Page_{\#} = \left\lfloor \frac{VirtualAddress}{PageSize} \right\rfloor$$
$$Physical = \left(\frac{VirtualAddress}{PageSize} - Page_{\#} + Frame_{\#} \right) \times PageSize$$

where $Frame_{\#}$ is determined by passing $Page_{\#}$ into the page table.

Given that a virtual address is x bits long, the page size is y bytes, a page table entry is z bytes, and there are p levels in the page table, then the following values are given by:

- **Maximum process memory** is found by considering that each address points towards one byte of memory. Therefore there are as many bytes of memory as there are addresses. Thus, the maximum memory a process can use is 2^x bytes.
- **Bits per page table layer** is a crucial quantity to use within calculations. Begin by finding the length of the page number, given by $x - \lceil \log_2(y) \rceil$. All but the first level page table are stored in standard pages, and will have a length of $\log_2(\frac{y}{z})$. The first page table layer will have the remaining bits in the page number, given by $(x - \lceil \log_2(y) \rceil) - (p-1) \times \log_2(\frac{y}{z})$.
- **Number of page tables needed** is highly useful in finding the memory that a page table takes up. Consider that there are a bytes of process memory to get stored. The number of pages to store the memory will be given by $\lceil \frac{a}{y} \rceil$. Each of these must be pointed to by an entry in a lowest level page table, and there are a total of $\frac{y}{z}$ entries. The number of lowest level page tables needed is therefore the quotient of these: $\left\lceil \lceil \frac{a}{y} \rceil / \frac{y}{z} \right\rceil$. For any intermediate page table level, similar logic applies, where there are the same number of page table entries per table, and the number of pages it points to is given by the lower level. There will always be one level one page table, which will have a size of $2^c \times z$, where c is the number of bits in the level one page table (per above).
- **Memory to store page table** is found once the number of page tables at each level is known. This is simply the product of the number of page tables by their size. Be cautious where the level one page table is of a different size to lower levels.

5 File Systems

File systems revolve around inodes storing file metadata, along with pointers to the storage that it uses. These pointers may be direct (directly points to the storage block) or indirect (similar to multi-level page tables, is a block containing pointers). A number of calculations may be used during the exam, which will for ease use the standard variables b to represent the size of a block in bytes, p to represent the size of a block pointer in bytes, and n_i to represent the number of i indirect pointers (where $i = 0$ implies a direct pointer).

- **Maximum file size** is found by simply considering the total possible space taken up by blocks that the pointers refer to. This will be one block for every direct pointer, $\frac{b}{p}$ blocks for every single indirect pointer, etc. This gives a formula of $\sum_i \left[n_i \times \left(\frac{b}{p} \right)^i \right]$ blocks, which in turn means the file size is $b \times \sum_i \left[n_i \times \left(\frac{b}{p} \right)^i \right]$ bytes.
- **Maximum size without a k indirect pointer** is similar to the maximum file size, but you ignore any indirect pointer of at least degree k. Mathematically, that means that you assign all $n_i = 0$ for $i \geq k$.
- **What block number requires use of a k indirect pointer?** This question is simply a matter of considering the first index of a block referred to by the given level of indirection, it will be the value of $\sum_{i=0}^{k-1}(n_i)$.
- **How many blocks accessed to read block #j?** This will require you to consider which level of indirection that block $\#j$ lives in. At a level k indirection, this will mean you need to access k pointer blocks, plus the block $\#j$. Therefore, you will need to access $k + 1$ blocks.
- **How many blocks accessed to read bytes x to y?** Begin by working out what blocks each listed byte lies in, which will be $\lfloor \frac{x}{b} \rfloor$ and $\lfloor \frac{y}{b} \rfloor$ respectively. Then, this will be very similar to the previous question, except you will need to consider every level of indirection in the block range.
- **What is the maximum file size if no disk blocks are used?** This is asking what file size is available purely within the pointers. This is simply given by $p \times \sum_i (n_i)$ bytes.

6 Makefile

Makefiles may come up, here's some information to help in writing / analysing them. A lot of the features are essentially bash scripts, but there are specific features too.

6.1 Standard Format

Here's a basic layout of a simple Makefile with some of the common features:

```
# Makefile
# Sets the compiler
CC=gcc
# Sets the compiler flags
CFLAGS= -Wall -std=gnu99 -pedantic -g
# Sets the linker flag, used whenever an executable is created
LDFLAGS=
# Tells make that the names are phony / not representative
.PHONY: all
# When nothing is specified, chooses the default
.DEFAULT_GOAL := all

# Sample compilation
all: program
program: program.c
    $(CC) $(CFLAGS) $(LDFLAGS) $< -o $@

# Cleans up any leftover files, in this case removing object files
clean:
    rm -f *.o
```

6.2 Layout

There is a basic layout that is to be observed at all points, and can be noted in the sample above:

```
target: prerequisite
    recipe
    recipe
```

The target is the name of the file to create, or an action name. The prerequisites (or dependencies) are files that must be found or targets that must be carried out prior to the target being built. The target will only be rebuilt if it doesn't already exist, or if it is older than its dependencies. The recipes are bash commands to carry out, and are to be tab indented.

6.3 Default Operation

By default, the first target will be the one to execute, this can be altered by setting the default goal flag as seen in the standard format above. The default variables "$<" can be used to refer to the first dependency, and "$@" to the target's name. The phony flag tells make that a given target is not a real file, and to just treat it as a an action.

6.4 Implicit Rules

Make has some implicit rules that can be used for standard compilation processes. This can simplify what needs to be written. Make can by default build executables from C, objects from C, and executables from objects. It will automatically apply the CC, CFLAGS, and LDFLAGS when called. Note that the names must align:

```
# This will build the executable named program if program.c is present
program: program.c
# This will not build an executable
file: code.c
# This will compile example.o automatically (with -c) if example.c is present, then link from that
example: example.o
```

7 Useful C Functions

With programming questions, there are a number of useful functions that will be very useful to know the specifics of. This will summarise them.

7.1 Printing

The standard *printf()* function is always useful, and can use the following type inserts:

- "%d" or "%i" to insert an integer.
- "%o" for octal, "%u" for unsigned decimal, "%x" for hexadecimal.
- 'l' can be prepended to the above for a long version, eg "%ld" is a long integer.
- "%f" for a double precision floating point number.
- "%c" for a character.
- "%s" for a string, or null-terminated char*

There may be other cases where the standard printf function doesn't cut it. In these cases, the following could be useful:

- *fprintf()* can be used to print to a FILE*. Takes the file pointer as the initial argument, followed by the standard arguments for printf.
- *dprintf()* can be used to write to a file descriptor (fd). Takes the fd integer as the first argument.
- *sprintf()* does not print in the regular sense, it just formats a string. The first argument it takes is the char* to which the formatted string is to get output.
- *snprintf()* is a safer form of sprintf, in that it also takes a buffer length (of type size_t) as its second argument to avoid buffer overflow.

Make sure to flush the buffers where necessary any time printing to a file stream.

7.2 Working with Files

Working with files is an important aspect of the programming questions in the final. This section will run through the useful functions to do so.

7.2.1 Opening

The first way to open files is the more primitive *open()* function, which takes the form

```
1 int open(const char* pathname, int flags, [mode_t mode]);
```

The output of the function will be an integer file descriptor. The pathname is self-explanatory, and can be either absolute or relative. The flags that it can must include one of the following access modes:

- *O_RDONLY* will open the file in read only form, allowing it to be read from, but not written to.
- *O_WRONLY* will open the file in write only form, allowing it to be written to, but not read from.
- *O_RDWR* will open the file to be both written to and read from.

In addition to these, one of the following can be added in with the logical or operation, '|'.

- *O_APPEND* opens the file in append mode, where every write action adds to the end of the file.
- *O_CLOEXEC* is a protective practice to make the file descriptor close upon an execution call, to ensure that the new program doesn't have access to any files it shouldn't.

- *O_CREAT* creates the file if it can't be found. When using this flag, open must supply the optional mode argument, which can be one of the following:
  - To set owner permissions, use the *S_IRUSR*, *S_IWUSR*, *S_IXUSR* for read, write, or execute respectively. Multiple can be logically or'd together with '|', or all three can be used with *S_IRWXU*.
  - To set the group permissions, similarly use *S_IRGRP*, *S_IWGRP*, *S_IXGRP*, or for all three use *S_IRWXG*.
  - To set the other permissions, similarly use *S_IROTH*, *S_IWOTH*, *S_IXOTH*, or for all three use *S_IRWXO*.
- *O_TRUNC* will truncate the contents of a file if it already exists. That is, if the file already exists and your are trying to open it to write to, it will delete the contents.

This method of opening files is a lot more primitive, and comes with less of the features that make the process simpler. For safer usage of files, operating with a FILE* is preferable in most cases.

This can be done from the file descriptor obtained with *open()* by use of the *fdopen()* function:

```
1 FILE* fdopen(int fd, const char* mode);
```

or straight from the path, skipping the call to *open()* entirely with *fopen()*:

```
1 FILE* fopen(const char* path, const char* mode);
```

The path or file descriptor are the same as relating to *open()*, but the mode is instead a more human readable form, taking the form of one of the following strings:

- "*r*" will open the file for reading with the pointer at the beginning of the file.
- "*r+*" will open the file for reading and writing, with the pointer at the beginning of the file.
- "*w*" will open the file for writing, either creating the file or truncating it.
- "*w+*" will open the file for reading and writing, creating or truncating it as needed.
- "*a*" will open the file for writing, with the pointer at the end of the file (for appending).
- "*a+*" will open the file for reading and writing, with the pointer placed for appending.

When using the *fdopen()* function, the mode specified must be compatible with the fd passed in.

7.2.2 Interacting

Once a file is open, you'll need to be able to interact with it. How this is done will depend on whether you are working with an fd or a FILE*. When using an fd, consider the *read()* and *write()* functions. Read takes the form of:

```
1 ssize_t read(int fd, void* buf, size_t count);
```

The first argument is the fd to read from, the second argument is the pointer to the start of the buffer to post the data to, and the final argument is the maximum number of bytes to read. On success, it returns the number of bytes read, else it returns -1 and sets errno. Write takes the form of:

```
1 ssize_t write(int fd, const void* buf, size_t count);
```

The first argument is the fd to write to, the second is the buffer to write from, and the third is the maximum number of bytes to write. Success returns the number of byte written, failure returns -1 and sets errno. Alternatively, consider the *dprintf()* function discussed in the printing section.

Working with a FILE* simplifies the process, writing is done using the *fprintf()* function. Reading can be either done character by character with *fgetc()*, or by using a char* buffer with *fgets()*. Fgetc takes the form of:

```
int fgetc(FILE* stream);
```

Simply, pass in the file pointer, and it will return the next character. The return value is an int as it could return EOF, which is not a character. If this is not the case, then it can just be cast. Fgets takes the form of:

```
char* fgets(char* s, int size, FILE* stream);
```

The first two arguments describe the location and size of the buffer to use, and the last argument is the stream to read from. It reads until either a newline '
n' (which it will include in the string), or EOF.

7.2.3 Routing

Knowing how to route file descriptors can massively simplify some problems, so useful to know. Duplicating file descriptors can be useful sometimes, for this the *dup()* and *dup2()* functions can be used. Dup takes the form:

```
int dup(int oldFd);
```

Basically, it duplicates the old fd to the lowest open fd, resulting in two equivalent file descriptors. This is useful when it comes to network sockets, to split the read and write processes. Dup2 takes the form:

```
int dup2(int oldFd, int newFd);
```

This duplicates the old fd to the new fd specified. If the new fd already is in use, it will get closed.

Alternatively, being able to send information between different parts of a program is useful, and can be done with *pipe()*. This takes the form:

```
int pipe(int pipeFd[2]);
```

This takes in the array to place the pipe fds into, and fills it with file descriptors for the read and write ends of the pipe, in that order. On error, returns -1 and sets errno, else returns 0.

7.2.4 Closing

Once files or pipes are finished with, then they should be closed. How this is done will depend on whether it is an fd or FILE* used. For an fd, use:

```
int close(int fd);
```

This closes the specified file descriptor, returning 0 on success, or -1 (and errno) on failure. For a FILE*, use:

```
int fclose(FILE* stream);
```

This closes stream, and returns 0 on success, or EOF on failure (and sets errno).

7.3 Forking

Especially useful when executing another program, forking splits the program into two separate processes, that run concurrently. The basis of this is:

```
pid_t fork();
```

This will fork the program, and the return value depends on which fork you look at. The parent's fork will have the function return the process ID (pid) of the child. The child's fork will see a return value of zero. If it fails, no child is created and a return value of 1 is found. This will also duplicate any memory, and buffers, so it is highly recommended to call *fflush()* on any open file streams prior to forking.

Sometimes there may be a need to wait until one fork process has finished running. This should be done from the parent process, as it will have the child's pid. How precise this needs to be will dictate the function to use. If it just needs to wait for any child process to finish, then use:

```
pid_t wait(int* wstatus);
```

If there is a specific process to wait for, then use:

```
pid_t waitpid(pid_t pid, int* wstatus, int options);
```

In both cases, a pointer is passed in to which the exit status is written, and both return the pid of the process it affects. The second form also passes in the pid to target, and has the option to pass options in. Note that wait is just a special case, of waitpid:

```
wait(wstatus) = waitpid(-1, wstatus, 0);
```

The useful option that might be used is *WNOHANG*, which tells the function to immediately return (and not be left hanging) should the child not have exited. The exit status (the value itself, not the pointer passed into the function) needs to be interpreted using other functions, which include:

- *WIFEXITED(wstatus)* which returns true of the program exited normally (by calling exit, or returning from main).
- *WEXITSTATUS(wstatus)* returns the exit status if it exited. Should be checking with WIFEXITED before using this.
- *WIFSIGNALED(wstatus)* returns true if the program was killed by a signal.
- *WTERMSIG(wstatus)* returns the termination signal, if it was signalled.

7.4 Executing Programs

You may want to push the load off to another existing program, in which case the execution family of functions is very useful. There are four main denominations that may be useful:

```
    int execl(const char* pathname, const char* arg, ..., NULL);
    int execlp(const char* pathname, const char* arg, ..., NULL);
    int execv(const char* pathname, char* const argv[]);
    int execvp(const char* pathname, char* const argv[]);
```

All four of these programs will execute the program found at pathname. The first two variants with the 'l' take the argument list as arguments to the function itself. The latter two with the 'v' take a vector of arguments, a null terminated list of the arguments to pass in (including the program name at index 0). The variants ending in a 'p' will also look to the path variable when considering the pathname. If successful, there will be no return value and any following code will not be executed, but if the program fails, it will return -1 and set errno.

7.5 Threading

Threading is a means by which processing can be optimised. Rather than performing operations sequentially, it means that things can be done in parallel. Unlike forking the process, which duplicates and separates all of the memory, threading will share a lot of memory (not registers or the stack, but the heap, etc is shared). Threading can also be made to occur over multiple physical cores to accelerate processes. When creating a thread, it will need a starting function, of the form:

```
    // Must be of a void pointer type, and take in a void pointer.
    void* thread(void* stuff) {
        // Good to cast the void pointer to whatever type you are using
        type* useful = (type*)stuff;

        /* Do stuff here */

        return (void*)result;
    }
```

To create a thread, call:

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

The first argument to this is a pointer to where the thread id is to be stored (useful for the calling thread); the second is the a set of attributes, can be NULL for default operation; the third is the name of the function to create it with (for example, from the earlier example this would just be *thread*); the final argument is the argument to pass into the thread function. When a thread exits, it shouldn't call exit, but instead:

```
void pthread_exit(void* retval);
```

To avoid zombie threads, they need to be reaped similar to waiting on forks using:

```
int pthread_join(pthread_t thread, void** retval);
```

This takes in the thread id, and a pointer to where the return value should be placed. Successful joins return 0, otherwise an error code. An alternative to having to reap zombies is to detach the thread so that it cleans up on exit (you just won't be able to get the return value), using:

```
int pthread_detach(pthread_t thread);
```

7.6 Networking

Sometimes you need to talk to another computer, so networking is useful. Thankfully, it's linux, so everything is a file and it's very similar to working with a file normally. It's all based around sockets, which are created with:

```
int socket(int domain, int type, int protocol);
```

The domain argument specifies the type of communication being done, which will be using IPv4, with *AF_INET*. The second arg is the type of socket connection, which will either be a reliable, two-way connection with *SOCK_STREAM*, or a one-way datagram with *SOCK_DGRAM*. Use the default protocol with 0. On success, this returns the fd for the created socket, otherwise -1 and sets errno.

7.6.1 One-Way (UDP)

This should not be needed, just included for completeness. The first steps of creating a datagram application are similar to the server below, create a socket and bind it. As it's one-way, no connection needs to be established, at this stage it's purely a case of sending to and receiving from the source:

```
    ssize_t recvfrom(int sockfd, void* buf, size_t len, int flags, struct sockaddr* src_addr, socklen_t* addrlen);
    ssize_t sendto(int sockfd, void* buf, size_t len, int flags, struct sockaddr* dest_addr, socklen_t* addrlen);
```

7.6.2 Two-Way Client (TCP)

A client program is the simplest. Once it has a socket, then it needs to connect to the server:

```
int connect(int sockfd, const struct sockaddr* addr, socklen_t addrlen);
```

This connects the client's socket at sockfd to the address specified by addr (the size of which is specified by addrlen). At this point, the socket's fd is how communication to the server will be carried out, and it can be treated like a standard file. Here's a lecture sample for a client program:

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

int main(int argc, char** argv) {
    const char* port = argv[1];
    struct addrinfo* ai = NULL;
    struct addrinfo hints;
    memset(&hints, 0, sizeof(struct addrinfo));
    hints.ai_family = AF_INET; // IPv4, can use AF_UNSPEC for generic
    hints.ai_socktype = SOCK_STREAM;
    int err;
    if ((err = getaddrinfo("localhost", port, &hints, &ai))) {
        freeaddrinfo(ai);
        fprintf(stderr, "%s\n", gai_strerror(err));
        return 1; // Couldn't work out the address
    }

    int fd = socket(AF_INET, SOCK_STREAM, 0);
    if (connect(fd, ai->ai_addr, sizeof(struct sockaddr))) {
        perror("Connecting");
        return 2; // Connection error
    }

    // Separate the streams
    int fd2 = dup(fd);
    FILE* read = fdopen(fd, "r");
    FILE* write = fdopen(fd2, "w");

    /* Do stuff in here, treating read and write like normal. */

    return 0;
}
```

7.6.3 Two-Way Server (TCP)

Connecting to the server is a bit more complex, instead of connecting, it must bind an address, listen for connections, then accept connections. To bind the address, use:

```
int bind(int sockfd, const struct sockaddr* addr, socklen_t addrlen);
```

These arguments are the same as those passed into connect for the client. Returns zero on success, -1 and sets errno otherwise. After the address is bound, it must listen for connections using:

```
int listen(int sockfd, int backlog);
```

The first argument is the socket fd to be listening on, the backlog is the maximum length of the queue of pending connections. Returns zero on success, -1 and sets errno on failure. To communicate with any of the clients, it must then accept the connection:

```
int accept(int sockfd, struct sockaddr* addr, socklen_t* addrlen);
```

The first argument is the socket that is being listened on. The second and third arguments get filled in with the details of the client peer. On success, the function returns the file descriptor for communications with the client, otherwise it returns -1 and sets errno. For better understanding, here's a sample program from the lectures:

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

int main(int argc, char** argv) {
    char* port;
    // Gives ephemeral port if none provided
    if (argc > 2) {
        port = argv[1];
    } else {
        port = "0";
    }
    struct addrinfo* ai = NULL;
    struct addrinfo hints;
    memset(&hints, 0, sizeof(struct addrinfo));
    hints.ai_family = AF_INET; // IPv4, can use AF_UNSPEC for generic
    hints.ai_socktype = SOCK_STREAM;
    hints.ai_flags = AI_PASSIVE; // Want to bind on all interfaces
    int err;
    if ((err = getaddrinfo(NULL, port, &hints, &ai))) {
        freeaddrinfo(ai);
        fprintf(stderr, "%s\n", gai_strerror(err));
        return 1; // Couldn't work out the address
    }

    int server = socket(AF_INET, SOCK_STREAM, 0);
    if (bind(fd, ai->ai_addr, sizeof(struct sockaddr))) {
        perror("Binding");
        return 2; // Connection error
    }

    // Print the port used
    struct sockaddr_in ad;
    memset(&ad, 0, sizeof(struct sockaddr_in));
    socklen_t len = sizeof(struct sockaddr_in);
    if (getsockname(server, (struct sockaddr*)&ad, &len)) {
        perror("sockname");
        return 3;
    }
    printf("%u\n", ntohs(ad.sin_port));

    // Opens port for listening
    if (listen(serv, 10)) {
        perror("Listening");
        return 4;
    }

    // Connect to a client
    int fd = accept(server, 0, 0);
    if (fd < 0) {
        perror("Accepting");
        return 5;
    }

    // Separate the streams
    int fd2 = dup(fd);
    FILE* read = fdopen(fd, "r");
    FILE* write = fdopen(fd2, "w");

    /* Do stuff in here, treating read and write like normal. */

    return 0;
}
```