1 Week 1

1.1 Operating System/Kernel Responsibilities

- Hardware Abstraction: Allows desktop applications to use the hardware of the computer system using an application programming interface (API) regardless of the make of hardware. For example, a Seagate hard drive and Western Digital hard drive are both seen by the application as a storage device with the same function.
- Scheduling: Allowing time for an application to run on a processor
- Interprocess Communication (IPC): Allow communication between two applications which occurs through the kernel. Applications cannot directly communicate with each other
- Memory Management: The kernel will allocate memory for each application to avoid conflict between two applications trying to use the same memory address
- File System: A storage device is defined as a particular type of file system (e.g. NTFS, FAT32) in which the kernel will control how files are read and written
- Initialisation: Commencing the operation of an application
- Security: Kernel encrypts and decrypts data for security
- Daemons: Applications that run all the time (called processes in Windows environment)

1.2 Unix File System Heirachy (FHS)

Unlike the Windows File system, the Unix file system stores the files according to the type of file instead of what application the file belongs to. For example, Microsoft Office can usually be found at C:/Program Files/Office which will contain the executable file for the program as well as manuals for the program. In the Unix filesystem the executable file would be found in the /opt directory and the readme would be found in the /usr/share/man directory.
The following directories can be found in the Unix file system

- /bin: Essential binaries - files read by the CPU
- /boot: Boot file used to initialise the syste.
- /dev: Device files used to allow applications to access to hardware components
- /etc: Files that can be edited by user to configure the system
- /home: Home folders for user to store files

- /lib32 & /lib64: Library binaries
- /media: Removable media such as CD-ROM and floppy drives can be found here:
- /mnt: Directory for mounted file systems
- /opt: Directory for 3rd party software
- /proc: Contains information about the kernel processes
- /root: Root user's home folder
- /sbin: Contains root user binary files
- /sys: Contains information about the hardware, seen as files by the kernel
- /temp: Storage for temporary files which are cleared each time the system boots
- /usr: Contains all installed packages
- /var: Contains system files such as logs, caches, mail directories, print spool etc.

1.3 Basic Commands

- man: shows the manual for a specific command
- ls: list all files in a directory
- cd: change directory
- rm: remove file
- mkdir: make directory
- rmdir: remove directory
- top: show running process
- file: indentify file type
- su: super user (become root user)
- killall -9: kills all applications
- bash_history: displays all commands typed into the terminal
- vi: opens VIM (text editor)
- emerge: Command line interface to the package system
- grep: Searches the named input files for lines containing a match to the given pattern

1.4 Paths

Files locations within the file system can be expressed absolutely or relatively

```
/etc/portage/package.use - absolute path
./package.use - relative path
```

1.5 Text Editing

The text editor used to write C files is the Vi Improved (VIM for short). To launch the application the command is 'vi'.

1.6 Package Management

In Linux systems a

2 Week 2

2.1 GCC Compiler

To turn C programs written in VIM into executable applications in Linux the GNU project C compiler is used. To launch this application the command 'gcc' is used. This application does the preprocessing, compiliation, assembly and linking.

2.2 Hello World Application

To practice using the gcc compiler a simple "hello world" script was written

```
#include <stdio.h>
int main(int argc, char **argv){
        printf("Hello World\n");
        return 0;
}
```

To compile this programing using the gcc compiler, the follow command was written into the terminal

```
gcc -Wall helloworld.c -o helloworld
```

Where 'gcc' is the application, '-Wall' means to display all warnings, helloworld.c is the file name of the script, '-o' is the output file specifier in which helloworld will be the name of the executable file.

2.3 Application Interaction with Kernel

As stated in Week 1, applications do not have direct access to hardware modules or other applications. For example, the 'printf' command does not operate by allowing the application direct access to the output, rather it is triggered by a set of commands

3 Week 3

4 Week 4

5 Week 5 - Linked Lists

An application was developed to illustrate the use of linked lists. A linked list is a dynamic data structure whose length is defined (unlike an array which has a defined length). The memory blocks in the linked list are in random memory locations and are connected using pointers which contain the address for the next memory block.
The following libraries were included for the commands

```
<stdio.h> - For printf, scanf and fprintf
<stdlib.h> - For free, malloc
<string.h> - For strdup
<getopt.h> - For long_options, required_arguement, opt_arg
```

At the beginning of the program file the 'typedef' function was used to declare the structure 'word_object' which is how linked list nodes are defined

```
typedef struct s_word_object word_object
```

The structure 's_word_object' is a block containing a 'char' type variable and a pointer *next contains the address of the next node. The structure is defined in a recursive manner

```
struct s_word_object{
        char *word;
        word_object *next;
};
```

A static variable called 'list_head' is defined which will point to the first item on the list.

```
static word_object *list_head;
```

To add words to the list a function called 'add_to_list' is created which will input the data from stored at the address 'word'

```
static void add_to_list(char *word)
```

To store the memory location of the last word another node is created called 'last_object'

```
word_object *last_object
```

An 'if' statement is then used to check whether the address of the first item on the list is 'NULL'. If this is the case then a dummy address is created for the node then the address of list_head is stored into last_object. In doing so when the memory allocation is 'freed' then both of the nodes have the addresses unallocated.

```
if (list_head == NULL){
          last_object = malloc(sizeof(word_object));
          list_head = last_object;
}
```

If there is an address in list_head (an input from the scanf function) then this becomes the last_object and the address of the next node is generated for the next iteraton of the function. This next address will be stored in the list_head and the process will be repeated.

```
else {
          last_object = list_head;
          while(last_object->next){
                    last_object = last_object->next;
          }
          last_object->next = malloc(sizeof(word_object));
          last_object = last_object->next;
     }
```

Once the next object have been

```
 last_object->word = strdup(word);
 last_object->next = NULL;
```

A function is created caled "print and free" which is an iterative process which will print the input words and free the allocated memory. The current_object pointer is used to

```
void print_and_free(void){
          word_object *current_object;
          word_object *old_object;
          current_object = list_head;

          while(1){
                    printf("%s\n", current_object->word);
                    free(current_object->word);
                    old_object = current_object;
          if(current_object->next){
                              current_object = current_object->next;
                              free(old_object);
                    }
                    else{
                              free(old_object);
                              break;
                    }
          }
}
```

The main function takes two inputs into the application by using the 'getopt_long' function. The main function has the capabilities to parse more than one arguement. However this application will only parse the function 'c' which provides the 'count' option. This 'count' option allows the user to specify how many strings will be saved and printed.

```
int main(int argc, char **argv) {
        char input_word[256];
        int c;
        int option_index = 0;
        int count = -1;
        static struct option long_options[] = {
                {"count",  required_argument, 0, 'c'},
                {0,        0,                 0,  0 }
        };
```

The getopt_long function will search through the options until it is correctly found. If an option is found then getopt_long returns -1. A blocking statement using the if command waits for the return of '-1' before preceeding. As the arguement parsed through the main function is 'c', the switch statement goes to case 'c' where the string passed through the main function is converted into an integer using the 'atoi' function and stored into the variable 'count'.

```
while(1){
                c = getopt_long(argc, argv, "c:", long_options, &option_index
                if(c==-1){break;}

                switch(c){
                case 'c':
                        count = atoi(optarg);
                        break;
                }
        }
```

The text is printed using the 'fprintf' into the 'stderr' output which infoms the user as to how many strings will be printed

```
fprintf(stderr, "Accepting %i input strings\n",count);
```

Using in the 'scanf' function the input word is passed into the add_to_list function. This is repeated until the value of end of file is reached. When this occurs it breaks out of the while loop.

```
while(scanf("%256s", input_word) != EOF){
                add_to_list(input_word);
                if (!--count) break;
        }
```

Once the end of file is reached, the print_and_free function is executed which will print input words and free the allocated memory.

```
print_and_free();
```

```
        return 0;
}
```

The when the application is run with 3 strings accepted and the inputs as “first” “second” “third” the terminal prints

```
init_user@gentoo-usb ~/link $ ./hello_world -c 3
Accepting 3 input strings
first
second
third
first
second
third
```

6 Threading

6.1 Introduction to Threadings

The purpose of threading is to use the CPUs multi-threading capabilties so that the mutliple cores of the processor can be executing tasks simultaneously which will speed up the execution of the application. By splitting the a program into threads, each thread acts like its own individual program except they work in the same shared memory space. This makes communication between threads easy and each a halt in one thread will not impact on the other threads performing tasks.
In this exercise the linked list application developed in Week 5 is edited to use threading.

6.2 Library for Threading

The header file used for threading is <pthread.h> which includes the following functions used in the program

- PTHREAD_MUTEX_INITIALIZER
- PTHREAD_COND_INITALIZER

The purpose of these functions are to create mutexes for the application. A mutex is like a lock which protects memory from being modified by other threads which may cause crashes to the application or system. The mutexes created for these application are

```
pthread_mutex_t list_lock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t  list_data_ready = PTHREAD_COND_INITIALIZER;
pthread_cond_t  list_data_flush = PTHREAD_COND_INITIALIZER;
```

For this program it is to imperative that the list is locked at times to ensure that multiple threads do not perform functions simultaneously on the list (e.g. one thread reading and another deleting a node).

6.3 Editing the 'add_to_list' Function Call

In the original linked list application there was no list locking mechanism. As list locking depends on the blocking functions use (e.g. malloc, strdup) the code needs to be rearranged. This needs to be done because blocking functions can remove the thread from the scheduler so should never been called with lock. Firstly a new variable with the 'word_object' type is created called tmp_object which will be used to allocate memory for the next object.

```
word_object *last_object, *tmp_object;
```

Next the blocking functions are taken outside of the if/else statement and temporary variables are created.

```
char *tmp_string = strdup(word);
tmp_object = malloc(sizeof(word_object));
```

Then the lock is put on to prevent changes to the memory locations of the list while this function is performing its operations

```
pthread_mutex_lock(&list_lock);
```

Firstly consider the scenario where there is no next object (i.e list_head has no value). The list head points to the last object and the list is unlocked.

```
if (list_head == NULL){
        last_object = tmp_object;
        list_head = last_object;
        pthread_mutex_unlock(&list_lock);
}
```

If there is a next object then the pointer to that object is created and awaits the next value of the pointer in the list head.

```
else {
        last_object = list_head;
        while(last_object->next){
                last_obe ject = last_object->next;
        }
        last_object->next = tmp_object;
        last_object = last_object->next;
}
last_object->word = tmp_string;
last_object->next = NULL;
```

After the list operations have been completed the list is unlocked and a signal is sent out stating there is data ready for printing.

```
pthread_mutex_unlock(&list_lock);
pthread_cond_signal(&list_data_ready);
```

6.4 Main Function

In order to explain the other functions used in the application, the edits made to the main program need to be explained. The print and free function from the linked list application has been replaced with another function called 'print_func'. This function is called using the pthread_create command in which a new thread is started, calling the function 'print_func' with a pointer to the thread stored in the stack (&print_thread). The NULL arguments passed through the command specify that the thread is created with default attributes.

```
pthread_create(&print_thread, NULL, print_func, NULL);
```

The 'print_func' function's purpose is to gather the first object from the linked list by calling the function 'list_get_first', printing this string and then freeing the memory space once the string has been printed. The function awaits the signal from the 'add_to_list' function to indicate there is data using the 'pthread_cond_wait' command. This command are used to block on the condition of the mutex and atomically releases the mutex. After this the mutex is locked and is owned by the calling thread. The 'return arg' command has no purpose except to stop the warning from the compiler.

```
static void *print_func(void *arg){
        word_object *current_object;
    fprintf(stderr, "Print Thread Starting\n");
    while(1){
                        pthread_mutex_lock(&list_lock);
                 while(list_head == NULL){
                                pthread_cond_wait(&list_data_ready, &list_loc
                        }
            current_object = list_get_first();
            pthread_mutex_unlock(&list_lock);
            printf("Print Thread: %s\n",current_object->word);
                        free(current_object->word);
                        free(current_object);
            pthread_cond_signal(&list_data_flush);
    }
        return arg;
}
```

It also consists of a 'flushing' mechanism where any additional strings that have been inputted into the application are freed. No output is generated from this and it is executed at the end of the program

```
static void list_flush(void){
                pthread_mutex_lock(&list_lock);
                while (list_head != NULL){
                        pthread_cond_signal(&list_data_ready);
pthread_cond_wait(&list_data_flush, &list_lock);
                }
```

```
        pthread_mutex_unlock(&list_lock);
}
```

Without the flushing mechanism, the program exists prematurely without printing the input strings

```
init_user@gentoo-usb ~/thread $ ./hello_world -c 3
Accepting 3 input strings
Print Thread Starting
one two three four
```

With the flushing mechanism the terminal output becomes

```
init_user@gentoo-usb ~/thread $ ./hello_world -c 3
Accepting 3 input strings
Print Thread Starting
one two three four
Print Thread: one
Print Thread: two
Print Thread: three
```

The code for the user to input the strings into the application has not been altered except the 'list_flush' function is executed at the end of the program.