Operating Systems Engineering
Assignment 1: Q2 & Q4

R00159222 - Zachary Dair
zachary.dair@mycit.ie

This document provides answers to the theory questions, **Q2(parts b, c, d)** and **Q4 (parts a and b)**

Question 2:

Part B:

A user program can be considered as a custom program written in C to accomplish a certain task. This type of program can be seen in the hello.c and head.c programs previously implemented for the Xv6 OS, additionally cat, ls, etc are all further examples of user programs. A list of user programs is typically defined in the Makefile, in our case it's the uprogs list in the Makefile.

A user program links to library functions (printf, etc) in the makefile where the required library functions are assigned to the ULIB, these are linked and subsequently disassembled and then dumped into the symbol table.

The user programs access the operating system, using system calls. These can be seen as requests to the OS kernel. The kernel performs the service and returns a response interpretable by the process. System calls are typically used when a process needs creating or executing and for communicating between kernel services.

Part C:

The Xv6 shell code can be found in Sh.c, the shell is the interface used when interacting with the OS. The Shell program is essentially a user program which therefore executes in both the user space, and the kernel through the usage of system calls. From looking at the Sh.c program we see the usage of both Fork and Exec system calls. Fork is used to create child processes, allocate resources for new processes based on the parent process. In the

case of Sh.c Fork is used to create an empty child process which can then be utilised by a user program. However before the new user program can run it must be loaded into memory. Exec is the system call that handles loading a new program into the blank process, Exec takes a filename and an argument values list as parameters. The corresponding file be formatted as to include ELF instructions specifying the structure of the program. Finally the loaded program will be executed until exit is called, which will cause the return to the base sh code.

Part D:

ls.c or the ls program, is a user program utilised to list the contents of either the current directory or the directory given as a parameter. In the source code found in ls.c we can see that ls expects an argument and this is the path of the directory to examine. If the ls command is ran with no additional parameters it will call the ls function with "." as the parameter. This dot file contains the configuration of the directory, the implementation of ls in xv6 handles some errors at this stage, in the case that the dot file cannot be opened, the user is notified. If the file is opened, the inode table is examined and each of the dirent structs are read in a loop and using the inode table and inode number (assigned on creation of the file) the program retrieves the struct contents and outputs it for the user to see. The struct contents outputted is the file name, the type, the inode number and the size. Finally once the contents of the dot file has been read and the contents outputted to the user the dot file is closed, and the program exits returning to the shell program.

Question 4:

Part A:

In Xv6 the kernel handles interrupts, such interrupts are dispatched when a specific device requires attention from the operating system, the interrupt code can be found in trap.c of the Xv6 source code. Typically the kernel will be the only one with permissions to execute the interrupt hence why it handles them instead of the programs. The OS must set up the hardware to generate these interrupts, this is done by mapping 32 hardware interrupts (32-63) and utilising 64 as a system interrupt. Each of these are mapped into the interrupt descriptor table, this is accomplished by the Tvinit function call found in main.c of the source code. Each of these handlers then sets up a trap frame and then will call the

function trap found in Trap.c which requires a trap number to be passed into the struct which will then identify if it belongs to an interrupt or system call or a hardware device looking for attention (See switch cases in trap.c line 39-95). If it belongs to an interrupt we assume a process has acted unexpectedly, if this is the case then the killed attribute of the proc struct is set accordingly. In the case that the kernel was running not a user program it will output the information and assume it was a kernel bug.

In the case of system calls, the system call number from the trap frame and the status of the process is identified, if the process is killed, the program exits, otherwise the syscall function is used (trap.c line 43), this function returns the value of the system call handler, typically it will return negative numbers representing errors and positive numbers for success. System calls can contain arguments, a series of functions are used to extract these values as they can't be accessed through the function parameters. These functions are argint, argptr, argstr and argfd. The usage of argint can be seen in the trace program where we identify if we need to enable or disable tracing for a process (1= enable, 0= disable) seen in sysproc.c. These functions utilise the user space esp to locate the arguments which can then be cast to their required data type, once the arguments are retrieved they can be used in the implementations of the system call functions, as outlined in the trace example.

Part B:

The code to handle the keyboard driver in Xv6 can be found in Uart.c. In this case we use a UART device based on the Intel 8250 serial port to process the serial input from the keyboard. The UART registers must be set up accordingly, this is accomplished using the outb function with parameters COM1+2, 0 which refers to the COM1 address (0x3f8) plus the offset and 0 as a boolean to disable FIFO. Following this interrupts are enabled on COM1, which links to the trap.c code where interrupts are processed, specifically found in the switch case (line 70) where we identify where the interrupts come from (internal, external(COM1), system call). Keyboard interrupts from UART then trigger the uartintr() function found in uart.c which calls the consoleinter() function in console.c. Inside this function the keystrokes from the COM1 UART "keyboard" are placed into a buffer. This buffer stores data in a cyclic manner, meaning once full it begins filling at the start again. The operating system can then use the contents of this buffer when a process is ready to read. This buffer prevents keystrokes entered from being lost if the process was not ready

to read from the keyboard. In uartputc() found in uart.c we see the character being sent out using the outb function and the COM1+0 address. This handles sending the character to the UART, this is done by looping until the UART is free to transmit data, this status is indicated by the line status register (COM1+5). Similarly uartgetc() handles retrieving the character and we check the status of the line status register to ensure that the buffer is ready to be read.