

Systems Software Open Book Exam

BSc in Computer Science

**C18322011**

School of Computer Science

TU Dublin – City Campus

**14th January 2022**

**Exam Duration: 2.30pm to 4.30pm**

**(Note: you have the exam duration plus 30 minutes upload time)**

**Please mark the questions you have attempted with an X**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Question** | **Q1** | **Q2** | **Q3** | **Q4** |
| **Attempted** | **X** | **x** | **x** |  |

**Question 1**

# ***1(a) Answer:***

*[Max Word Count: 400 words]*

A makefile is used to auto compile a number of different c files with a single command, allowing for the easy execution and compilation of different c files.

The structure of a makefie consists of 3 distinct sections, the target, dependencies and rules. The target is usually an executable or an object file name, the dependencies are source code or other things which are needed to make a target, and finally the rules are the commands used to make the target. It can also contain definitions of variables and inclusion of other makefiles.

The c files would ordinarily be compiled individually using e.g. ‘gcc -o program program.o’, however when there are many files being used and compiled at the same time for a single larger program this becomes wholly inefficient and inadequate, this is even worse when the programs may need to be edited and and changed meaning theu need to be all deleted separately and compiled again. Make is therefore the perfectly solution for this as not only does it compile the programs it also can debug the program by outputting where the problems occur.

Text

Description automatically generated

The order of rules is not significant, except for determining the *default goal*: the target for make to consider, if you do not otherwise specify one. The default goal is the target of the first rule in the first makefile. If the first rule has multiple targets, only the first target is taken as the default.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

# ***1(b) Answer:***

*[Max Word Count: 700 words]*

Systemd offers many improvements on systems including faster boot times, moving away from more complicated init scripts for easier to manage config files for daemons. It achieves this by allowing multiple services to run concurrently which wasn’t previously possible and features auto restart after crashes, and very importantly it has auditd which cant be stopped which improves security a lot. Service file reside in many places on the system and some can be configured/made by the user to start on boot. The location od service files we dealt with is in the /etc/systemd/system.

Units in system refer to any resources the systems knows how to operate and mange.

[Unit]

Description=sample unit file

[Service]

User=root

WorkingDirectory=/

KillMode=process

ExecStart=/usr/bin/myDaemon

Restart=on-failure

[Install]

WantedBy=multi-user.target

Alias=webDaemon.service

When dealing with system you will usually need sudo level permissions. These are a few commands which can be used to manage these files.

To read = cat myDaemon.service

To edit = sudo gedit myDaemon.service

to start = sudo systemctl start myDaemon.service

To stop = sudo systemctl stop myDaemon.service

To restart = sudo systemctl restart myDaemon.service

Very important is enable and disable startup on boot

Startup on boot = sudo systemctl enable myDaemon.service

Disable on boot = sudo systemctl start myDaemon.service

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

# ***1(c) Answer:***

*[Max Word Count: 1000 words]*

In unix there is a simple idea of using fork() and exec() for the execution model. Fork is for the creation brand new proceses and exec which replaces the current process with a brand new program.with these 2 the entire unix execution model can be constructed.

The exec() call replaces the entire current contents of the process with a new program. It loads the program into the current process space and runs it from the entry point. So fork() and exec() are often used together to get a new program running as a child of a current process.

The execv(), execvp(), and execvpe() functions provide an array of pointers to

null-terminated strings that represent the argument list available to the new program.

The first argument, by convention, should point to the filename associated with the file

being executed. The array of pointers must be terminated by a NULL pointer.

The difference between execl and execv is the argument passing. execl requires a list of arguments while execv require a vector of arguments.  
A list of arguments is useful if you know all the arguments at compile time. In your case the arguments will be entered by the user and you have to construct a vector of arguments at run time, so you should use one of the execv functions.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**Question 2**

# ***2(a) Answer:***

*[Max Word Count: 400 words]*

A file descriptor is a number or id for an open file in the process and is needed to perform File IO operations. File descriptor offer a primitive low level interface. A FD can connect to a file, device (terminal), or a pipe or socket to communicate with another process.

Example: The way this works is to first find the existing file on disk, create file table entry, set first unused file descriptor to point to file table entry, return file descriptor used, -1 upon failure

#include<stdio.h>

#include<fcntl.h>

#include<errno.h>

extern int errno;

int main()

{

    // if file does not have in directory

    // then file foo.txt is created.

    int fd = open("foo.txt", O\_RDONLY | O\_CREAT);

    printf("fd = %d/n", fd);

    if (fd ==-1)

    {

        // print which type of error have in a code

        printf("Error Number % d\n", errno);

        // print program detail "Success or failure"

        perror("Program");

    }

    return 0;

}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

# ***2(b) Answer:***

*[Max Word Count: 700 words]*

Canonical mode is when Terminal input is processed in lines that are terminated by newline, EOF or EOL characters, and input is not read until a complete line has been entered by the user. Input editing can be provided for some characters to be dealt with in a special way, and limitations can be set on the length of the line, using \_POSIX\_MAX\_CANON and MAX\_CANON However Noncanonical mode is kind of the opposite. In Non canonical mode, terminal input characters are not grouped into lines, and do not provide features such as input editing unlike canonical mode.

Most programs use canonical input mode, because this gives the user a way to edit input line by line. The usual reason to use noncanonical mode is when the program accepts single-character commands or provides its own editing facilities. The choice of canonical or noncanonical input is controlled by the ICANON flag.

Example would be that after entering in a command int eh terminal and hitting enter that command cannot be edited unless it is started over from the beginning and run again,

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

# ***2(c) Answer:***

*[Max Word Count: 1000 words]*

Signals are a software interrupt, and programs and processes must deal with any signals or software interrupts it gets. There are a variety of signals to be sent, and what kind of signal received will determine how the program will proceed with the process A program has 3 options when a signal is received. The process can tell the kernel to simply just ignore the signal, catch the signal or just go with the default signal, but it is to note that specific signals such as SIGKILL which kills the process cannot be ignored or caught. This is due to any developer needing to kill or stop a process immediately.

The header file that contains signal functionality is in signal.h.

Text

Description automatically generated

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**Question 3**

# ***3(a) Answer:***

*[Max Word Count: 400 words]*

Named pipes are given a name and exist as a file in a system represented by an inode, unnamed pipes have no names and are identified by their two file descriptors.

Named pipes can be used even among unrelated processes, whereas unnamed pipes cannot be used between unrelated processes.

Named pipes are bidirectional which means first in first out can be read and written into, unnamed pipes are only one-way therefore they need separate pipes for reading and writing.

Named pipes exist independently in the file system from the process after they are created and can be used by other processes. Unnamed pipes are gone as when it is closed or one of the related processes terminates.

Finaly named pipes can be used across networks for communication of systems, unnamed are only local therefore they cannot be used across networks.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

# ***3(b) Answer:***

*[Max Word Count: 700 words]*

Ipc will be made through using pipes with a file descriptor so that two processes can communicate to each other, one for the processes to read in from the pipe in the kernel memory and the other to write into.

2 functions will call the first then the second if statements. Pid will be storing the process ID while pipefd array stores the file descriptors. The first function creates the pipe and stores the associated file descriptors in the pipefd array as intended. Then the second will fork a child process. If the pid is 0, we have a child process, and calling the exec1 swaps the child process to a different task.

int pid;

int pipefd[2];

//creating pipe1

if(pipe(pipefd) == -1) {

    perror("Error Init Pipe");

    exit(1);

}

//fork to create (ps aux)

if ((pid = fork()) == -1) {

    perror("Error Init Fork 1");

    exit(1);

}else if (pid == 0 ) {

    exec1();

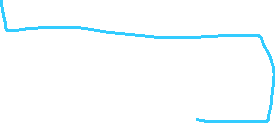
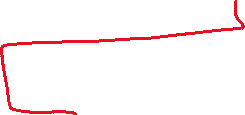
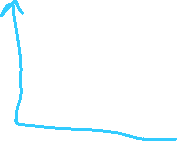
}

Processes 1 processes 2

parent child process



kernel



pipe



\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

# ***3(c) Answer:***

*[Max Word Count: 1000 words]*

It takes 5 steps to make a daemon. First is to create an orphan process, because it will default back to init as its parent procees.

2 is to elevate it into session leader so that it can loose its controlling terminal. Session leaders aer typically the process that created the session through the use of setsid() system call, in which it’s pid will be the sid as well. Every session is associated with a terminal, and get I/O from the terminal. A session is linked to a terminal called the TTY. A process can disconnect from its controlling terminal when it calls setsid to become the leader of a new session

// Implementation for Singleton Pattern if desired (Only one instance running)

   printf("Service Running \n");

   // Create a child process

   int pid = fork();

   if (pid > 0) {

      // if PID > 0 :: this is the parent

      // this process performs printf and finishes

      printf("Parent process \n");

      sleep(10);  // uncomment to wait 10 seconds before process ends

      exit(EXIT\_SUCCESS);

   } else if (pid == 0) {

      // Step 1: Create the orphan process

      printf("Child process\n");

      // Step 2: Elevate the orphan process to session leader, to loose controlling TTY

      // This command runs the process in a new session

      if (setsid() < 0) { exit(EXIT\_FAILURE); }

      // We could fork here again , just to guarantee that the process is not a session leader

      int pid = fork();

      if (pid > 0) {

         exit(EXIT\_SUCCESS);

3 is umask so that the daemon can read and write to files

// Step 3: call umask() to set the file mode creation mask to 0

         // This will allow the daemon to read and write

         // files with the permissions/access required

         umask(0);

4 is changing the dir so no conflicts occur when trying to reach different directories

 // Step 4: Change the current working dir to root.

         // This will eliminate any issues of running on a mounted drive,

         // that potentially could be removed etc..

         if (chdir("/") < 0 ) { exit(EXIT\_FAILURE); }

5 closing the file descriptors as they are not needed anymore

// Step 5: Close all open file descriptors

         /\* Close all open file descriptors \*/

         int x;

         for (x = sysconf(\_SC\_OPEN\_MAX); x>=0; x--){

             close (x);

         }

**Question 4**

# ***4(a) Answer:***

*[Max Word Count: 800 words]*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

# ***4(b) Answer:***

*[Max Word Count: 800 words]*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

# ***4(c) Answer:***

*[Max Word Count: 700 words]*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*