MURDOCH UNIVERSITY PERTH, WESTERN AUSTRALIA

School of Information Technology, Murdoch University

ICT374 ASSIGNMENT ONE

CHECK LIST

Surname: Royans

Given Names: Benjamin Paul

Student Number: 30493093

Tutor's Name: Alvaro Monsalve Ballester

Assignment Due Date: 22/09/2023

Date Submitted: 17/09/2023

Your assignment should meet the following requirements. Please confirm this (by ticking boxes) before submitting your assignment.

- ✓ I have read and understood the Documentation Requirements
- ✓ **This assignment submission is compliant to the Documentation Requirements.**
- ✓ I have included all relevant Linux source code, executables and test files in the tar archive. The file names are chosen according to the assignment specification.
- ✓ I have kept another copy of this assignment and associated programs and files in a safe place.

The unit coordinator may choose to use your submission as sample solutions to be viewed by other students, but only with your permission. Please indicate whether you give permission for this to be done.

- ✓ Yes, I am willing to have my submission without change be made public as a sample solution.

List of Files

The following files are included in this assignment submission:

Filename	Relative Directory	Purpose
memory.c	<root>\Question 2\	The C source code file for Question 2.
memory	<root>\Question 2\	The Linux executable for Question 2.
MemoryMap.ods	<root>\Question 2\	The spreadsheet containing the memory map data for Question 2.
output.txt	<root>\Question 2\	The program's (memory) output for Question 2.
main.c	<root>\Question 3\	The C source code file for Question 3.
q3	<root>\Question 3\	The Linux executable for Question 3.
Documentation.docx	<root>	This document. Contains all the question's answers, source code, and testing evidence.

Question 1: Process Switching

Assume three processes, namely Process A, Process B and Process C, are currently resident in the main memory. Process A is in the running state while process B and C are currently in ready and blocked states respectively. Answer the following three questions:

1. What does the CPU hardware and the operating system kernel do following a clock interrupt?
2. What does the operating system kernel do when Process A needs to read a word from the disk?
3. Assuming that Process C was waiting for a read disk operation, what would happen when the read disk operation is complete, and how would the CPU hardware and the operating system kernel respond to it?

In answering the above questions, you must describe the steps taken by both the hardware and the kernel. You should provide as much details as you can to demonstrate your understanding of how the CPU hardware and the operating system kernel work together to achieve preemptive multi-tasking.

Answer 1: Following a Clock Interrupt

1. CPU Hardware: When a clock interrupt occurs, the CPU hardware performs the following actions:
   a. Saves the current state of the running process (Process A) by saving its registers, program counter, and other relevant information into its Process Control Block (PCB).
   b. Switches the CPU execution mode to kernel mode, which allows it to execute privileged instructions.
   c. Jumps to a predefined interrupt handler routine in the operating system kernel.
2. Operating System Kernel: Upon receiving the clock interrupt, the kernel performs several tasks:
   a. Determines whether it's time to perform a context switch:
      i. If Process A has run for its time slice, it chooses the next process to run based on a scheduling algorithm and updates the Process Control Block (PCB).
      ii. If Process A still has time remaining in its time slice, the kernel updates its timer and returns control to Process A.
   b. If a context switch is required, the kernel updates the CPU's memory management unit to load the memory space of the selected process.
   c. Restores the saved state of the chosen process (Process B or C) from its Process Control Block (PCB).
   d. Sets the CPU's program counter to the saved value for the selected process.
   e. Resets the CPU execution mode to user mode, allowing the selected process to run.

Answer 2: Process A Reading a Work from Disk:

1. Process A: When Process A initiates a disk read request. it enters a blocked state and yields the CPU.
2. CPU Hardware: When the CPU recognises that Process A is blocked it generates an interrupt and transfers control to the Operating System (OS) kernel.
3. Operating System Kernel (After receiving the disk read request interrupt):
    a. Suspends Process A and places it in the blocked state, typically in a queue or a list of processes waiting for I/O operations.
    b. Initiates the disk read operation, which may involve scheduling the operation, configuring the hardware controller, and issuing the appropriate commands to the disk subsystem.
    c. The kernel then allows another process, such as Process B or C, to run while Process A is waiting for the disk operation to complete.

Answer 3: CPU and Kernal Response to Completed Read Disk Operation:

1. The disk hardware performs the requested read operation and signals that it is complete to the Operating System (OS).
2. Operating System Kernel (After receiving the completion signal):
    a. Updates the status of Process C from a blocked state to a ready state, indicating that it is now ready to run.
        i. If Process C has the highest priority or is the next process to run according to the scheduling algorithm, the kernel may choose to schedule it immediately.
        ii. If Process C is not immediately scheduled, the kernel may continue running the currently running process or select another process for execution based on its scheduling policy.
    b. The Operating System (OS) kernel handles the context switch if necessary and updates the CPU hardware to run the selected process.

Pre-emptive multitasking involves coordination between the CPU hardware and the operating system kernel. The hardware generates interrupts (such as clock interrupts or I/O interrupts) to trigger kernel intervention, and the kernel manages the scheduling of processes, context switches, and I/O operations to provide efficient and managed execution of multiple processes. This allows the CPU to appear to be concurrently executing multiple processes on a single CPU core.

Question 2: The Memory Layout of a Process

Question 1

Complete the following C program, memory23.txt to display the addresses of the following entities:

a. all functions
b. global constant gc and string literal "Computer Science"
c. all initialized global variables;
d. all un-initialized global variables;
e. all dynamically allocated memories;
f. all formal parameters in any function;
g. all local variables in any function;
h. the array containing the addresses of all command line arguments and the array containing the addresses of all environment variables;
i. start and end of its command line arguments;
j. start and end of its environment (variables and values).

Answer 1: *memory.c* Source Code Listing

Note: The following code was written and debugged in Vim launched from Ubuntu's command line. It was compiled using the GCC compiler in Ubuntu's command line and executed in Ubuntu's command line also. It has been opened in VS Code to copy the source code with syntax highlighting for readability.

```
/* name:          memory.c
 * aims:          to see how the compiler allocates memory to each region of
 *                the process (user-visible part), including text region (program instructions),
 *                data region, heap, stack, command line arguments, and process environment region
 *
 * author:        HX
 * updated:       2023.08.31
 */

#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <string.h>
#include <unistd.h>
#include <sys/resource.h>

extern char **environ;

int gx = 10;                  // initialized global
int gy;                        // uninitialized global
char gname1[] = "Hi, there!";
char *gname2 = "Computer Science";
const int gc = 100;
int gz;

void printAddress(char *description, void *addr)
{
    unsigned long a = (unsigned long)addr;

    unsigned long b = a & 0x3ff;
    unsigned long kib = a >> 10;
    kib = kib & 0x3ff;
    unsigned long mib = a >> 20;
    mib = mib & 0x3ff;
    unsigned long gib = a >> 30;
    gib = gib & 0x3ff;
    unsigned long tib = a >> 40;
    tib = tib & 0x3ff;
```

```c
    printf("%70s: %16p (%luTiB, %luGiB, %luMiB, %luKiB, %luB)\n", description, addr, tib, gib, mib,
kib, b);

    return;
}

int f1(int x1, int x2, float x3, double x4, char x5, int x6)
{
    int f1_l1;
    float f1_l2;
    char f1_l3;
    char f1_l3b;
    double f1_l4;
    int f1_l5;
    int f1_l6;

    printf("\n==== formal parameters in function f1 ====\n");
    // TO DO:
    // print the addresses of all formal parameters of function f1
    printAddress("x1", &x1);
    printAddress("x2", &x2);
    printAddress("x3", &x3);
    printAddress("x4", &x4);
    printAddress("x5", &x5);
    printAddress("x6", &x6);

    printf("\n==== local variables in function f1 ====\n");
    // TO DO:
    // print the addresses of all local variables of function f1
    printAddress("f1_l1", &f1_l1);
    printAddress("f1_l2", &f1_l2);
    printAddress("f1_l3", &f1_l3);
    printAddress("f1_l3b", &f1_l3b);
    printAddress("f1_l4", &f1_l4);
    printAddress("f1_l5", &f1_l5);
    printAddress("f1_l6", &f1_l6);

    return 0;
}

void f2()
{
    #define BUFSIZE 1024*1024
    char buf[BUFSIZE];
    char *p;
    p = malloc(BUFSIZE);
    if (p == NULL)
    {
        perror("malloc memory");
        exit(1);
    }

    printf("\n==== local variables in function f2 ====\n");
    // TO DO:
    // print the addresses of local variables buf and p of function f2
    printAddress("buf", &buf);
    printAddress("p", &p);

    printf("\n==== heap ====\n");
    // TO DO:
    // print the addresses of heap allocated memory pointed to by p in function f2
    printAddress("Heap memory (p)", p);

    printf("\n==== call function f1 in function f2  ====\n");
    f1(10, 20, 10.2, 20.3, 'a', 100);

    return;
}

int main(int argc, char *argv[], char *env[])
{
    printf("==== program text ====\n");
    printAddress("start address of function printAddress", printAddress);
    // TO DO:
```

```c
    // print the addresses of function f1, f2, and main
    printAddress("main", main);
    printAddress("f1", f1);
    printAddress("f2", f2);

    printf("\n==== constants and initialized globals ====\n");
    // TO DO:
    // print the addresses of constant gc and string literal "Computer Science"
    // print the addresses of initialized global variables gx, gname1, gname2
    printAddress("gc", (void *)&gc);
    printAddress("gname2", &gname2);
    printAddress("gx", &gx);
    printAddress("gname1", gname1);

    printf("\n==== uninitialized globals ====\n");
    // print the addresses of uninitialized global variables gy, gz
    printAddress("gy", &gy);
    printAddress("gz", &gz);

    printf("\n==== formal parameters in function main ====\n");
    // TO DO:
    // print the addresses of formal parameters argv, argv, and env
    printAddress("argc", &argc);
    printAddress("argv", argv);
    printAddress("env", env);

    printf("\n==== heap ====\n");
    char *p1 = malloc(200);
    char *p2 = malloc(10000);

    printf("\n==== local variables in main ====\n");
    // TO DO:
    // print the addresses of local variables p1, p2
    printAddress("p1", p1);
    printAddress("p2", p2);

    printf("\n==== heap ====\n");
    // TO DO:
    // print the addresses of heap-allocated memory pointed to by p1 and p2
    printAddress("Heap memory (p1)", p1);
    printAddress("Heap memory (p2)", p2);

    printf("\n==== call function f2 from the main function ====\n");
    f2();

    printf("\n==== arrays of pointers to cmd line arguments and env variables ====\n");
    // TO DO:
    // print the addresses of arrays of pointers pointing to cmd line arguments and env variables
    printAddress("argv", argv);
    printAddress("env", env);

    printf("\n==== command line arguments ====\n");
    // TO DO:
    // print start and end addresses of cmd line arguments
    printAddress("argv[0]", argv[0]);
    printAddress("argv[argc-1]", argv[argc - 1]);

    printf("\n==== environment ====\n");
    // TO DO:
    // print start and end addresses of environment variables
    printAddress("env[0]", env[0]);
    printAddress("env[1]", env[1]);

    exit(0);
}
```

Answer 1: Program Output

```
stupidflanders@stupidflanders:~/GitHub/ICT374_Ass1/Question 2$ ./memory
==== program text ====
                                start address of function printAddress:   0x56056dabb1e9 (86TiB, 21GiB, 730MiB, 748KiB, 489B)
                                                                  main:   0x56056dabb552 (86TiB, 21GiB, 730MiB, 749KiB, 338B)
                                                                    f1:   0x56056dabb2a0 (86TiB, 21GiB, 730MiB, 748KiB, 672B)
                                                                    f2:   0x56056dabb42a (86TiB, 21GiB, 730MiB, 749KiB, 42B)

==== constants and initialized globals ====
                                                                    gc:   0x56056dabc01c (86TiB, 21GiB, 730MiB, 752KiB, 28B)
                                                                gname2:   0x56056dabe028 (86TiB, 21GiB, 730MiB, 760KiB, 40B)
                                                                    gx:   0x56056dabe010 (86TiB, 21GiB, 730MiB, 760KiB, 16B)
                                                                gname1:   0x56056dabe018 (86TiB, 21GiB, 730MiB, 760KiB, 24B)

==== uninitialized globals ====
                                                                    gy:   0x56056dabe034 (86TiB, 21GiB, 730MiB, 760KiB, 52B)
                                                                    gz:   0x56056dabe038 (86TiB, 21GiB, 730MiB, 760KiB, 56B)

==== formal parameters in function main ====
                                                                  argc:   0x7ffcb53f4acc (127TiB, 1010GiB, 851MiB, 978KiB, 716B)
                                                                  argv:   0x7ffcb53f4bf8 (127TiB, 1010GiB, 851MiB, 978KiB, 1016B)
                                                                   env:   0x7ffcb53f4c08 (127TiB, 1010GiB, 851MiB, 979KiB, 8B)

==== heap ====

==== local variables in main ====
                                                                    p1:   0x56056e54a6b0 (86TiB, 21GiB, 741MiB, 297KiB, 688B)
                                                                    p2:   0x56056e54a780 (86TiB, 21GiB, 741MiB, 297KiB, 896B)

==== heap ====
                                                      Heap memory (p1):   0x56056e54a6b0 (86TiB, 21GiB, 741MiB, 297KiB, 688B)
                                                      Heap memory (p2):   0x56056e54a780 (86TiB, 21GiB, 741MiB, 297KiB, 896B)

==== call function f2 from the main function ====

==== local variables in function f2 ====
                                                                   buf:   0x7ffcb52f4a90 (127TiB, 1010GiB, 850MiB, 978KiB, 656B)
                                                                     p:   0x7ffcb52f4a88 (127TiB, 1010GiB, 850MiB, 978KiB, 648B)

==== heap ====
                                                       Heap memory (p):   0x7f3331cff010 (127TiB, 204GiB, 796MiB, 1020KiB, 16B)

==== call function f1 in function f2  ====

==== formal parameters in function f1 ====
                                                                    x1:   0x7ffcb52f4a3c (127TiB, 1010GiB, 850MiB, 978KiB, 572B)
                                                                    x2:   0x7ffcb52f4a38 (127TiB, 1010GiB, 850MiB, 978KiB, 568B)
                                                                    x3:   0x7ffcb52f4a34 (127TiB, 1010GiB, 850MiB, 978KiB, 564B)
                                                                    x4:   0x7ffcb52f4a28 (127TiB, 1010GiB, 850MiB, 978KiB, 552B)
                                                                    x5:   0x7ffcb52f4a30 (127TiB, 1010GiB, 850MiB, 978KiB, 560B)
                                                                    x6:   0x7ffcb52f4a24 (127TiB, 1010GiB, 850MiB, 978KiB, 548B)

==== local variables in function f1 ====
                                                                 f1_l1:   0x7ffcb52f4a50 (127TiB, 1010GiB, 850MiB, 978KiB, 592B)
                                                                 f1_l2:   0x7ffcb52f4a54 (127TiB, 1010GiB, 850MiB, 978KiB, 596B)
                                                                 f1_l3:   0x7ffcb52f4a4e (127TiB, 1010GiB, 850MiB, 978KiB, 590B)
                                                                f1_l3b:   0x7ffcb52f4a4f (127TiB, 1010GiB, 850MiB, 978KiB, 591B)
                                                                 f1_l4:   0x7ffcb52f4a60 (127TiB, 1010GiB, 850MiB, 978KiB, 608B)
                                                                 f1_l5:   0x7ffcb52f4a58 (127TiB, 1010GiB, 850MiB, 978KiB, 600B)
                                                                 f1_l6:   0x7ffcb52f4a5c (127TiB, 1010GiB, 850MiB, 978KiB, 604B)

==== arrays of pointers to cmd line arguments and env variables ====
                                                                  argv:   0x7ffcb53f4bf8 (127TiB, 1010GiB, 851MiB, 978KiB, 1016B)
                                                                   env:   0x7ffcb53f4c08 (127TiB, 1010GiB, 851MiB, 979KiB, 8B)

==== command line arguments ====
                                                               argv[0]:   0x7ffcb53f63b9 (127TiB, 1010GiB, 851MiB, 984KiB, 953B)
                                                          argv[argc-1]:   0x7ffcb53f63b9 (127TiB, 1010GiB, 851MiB, 984KiB, 953B)

==== environment ====
                                                                env[0]:   0x7ffcb53f63c2 (127TiB, 1010GiB, 851MiB, 984KiB, 962B)
                                                                env[1]:   0x7ffcb53f63d2 (127TiB, 1010GiB, 851MiB, 984KiB, 978B)
```

Answer 1: Self-Diagnosis and Evaluation

The *memory23* C Programming has been completed and tested to be working. All the stated variables used the *printAddress* function to print the required information. Initially, there was a warning stating that passing the address of the variable *gc* would be discarding the 'const' qualifier from the pointer target type. This was rectified by casting the *gc* variable address to *void**. This program compiles using GCC with no errors or warnings.

This source code is a modified version (as requested) made by Ben Royans to the original source code as part of the assignment, written by Hong Xie.

Question 2: Construct a Memory Map

Based on the information generated from the above program, produce a memory map table showing the layout of items listed in subquestion 1 above.

The memory map table must show the start addresses of each entity. It should also show the start and end addresses of the environment and the command line arguments.

The memory map table must contain at least the following columns:

a. The start address of an entity
b. The name of the entity, such as p in f2
c. The nature of the item, such as *function*, *local variable*, or *uninitialsed global variable, etc*
d. The memory region, eg, environment, command line arguments, code (or text), constants and global initialised data, global uninitialed data, stack, and heap, etc.

In addition, you must use seven different background colours to highlight the following seven memory regions as indicated below:

- [Green] initialised global variables (including constants and literals)
- [Red] uninitialised global variables
- [Blue] stack (containing the local variables and formal parameters of functions)
- [Magenta] heap (containing the dynamically allocated memories)
- [Yellow] text (program code, ie, functions)
- [Cyan] process environment
- [White] command line arguments

Please also note that in your memory table, *the memory addresses must be strictly sequential, from the lowest address to the highest address* to reflect how different entities of a process are layout in the virtual memory space. Your memory table will **not** be accepted if the addresses are not lined up sequentially in the table. If you find that entities from the same memory region are split in more than one continuous area of memory, it is a sure indication that there is something wrong with your memory map and you should find out what went wrong and fix it.

Answer 2: Memory Map Table

See also file MemoryMap.ods.

Entity Name	Item Nature	Memory Region	Start Address	TiB	GiB	MiB	KiB	B
printAddress	Function	Code/Text	0x559c97bbe1e9	85	626	379	760	489
f1	Function	Code/Text	0x559c97bbe2a0	85	626	379	760	672
f2	Function	Code/Text	0x559c97bbe42a	85	626	379	761	42
main	Function	Code/Text	0x559c97bbe552	85	626	379	761	338
gc	Global Variable	Initialised Globals	0x559c97bbf01c	85	626	379	764	28
gx	Global Variable	Initialised Globals	0x559c97bc1010	85	626	379	772	16
gname1	Global Variable	Initialised Globals	0x559c97bc1018	85	626	379	772	24
gname2	Global Variable	Initialised Globals	0x559c97bc1028	85	626	379	772	40
gy	Global Variable	Uninitialised Globals	0x559c97bc1034	85	626	379	772	52
gz	Global Variable	Uninitialised Globals	0x559c97bc1038	85	626	379	772	56
p1	Local Variable	Heap	0x559c98e926b0	85	626	398	585	688
Heap memory (p1)	Local Variable	Heap	0x559c98e926b0	85	626	398	585	688
p2	Local Variable	Heap	0x559c98e92780	85	626	398	585	896
Heap memory (p2)	Local Variable	Heap	0x559c98e92780	85	626	398	585	896
Heap memory (p)	Local Variable	Heap	0x7fd146b09010	127	837	107	36	16
x6	Formal Parameter	Stack	0x7ffe11d9dd54	127	1016	285	631	340
x4	Formal Parameter	Stack	0x7ffe11d9dd58	127	1016	285	631	344
x5	Formal Parameter	Stack	0x7ffe11d9dd60	127	1016	285	631	352
x3	Formal Parameter	Stack	0x7ffe11d9dd64	127	1016	285	631	356
x2	Formal Parameter	Stack	0x7ffe11d9dd68	127	1016	285	631	360
x1	Formal Parameter	Stack	0x7ffe11d9dd6c	127	1016	285	631	364
f1_l3	Local Variable	Stack	0x7ffe11d9dd7e	127	1016	285	631	382
f1_l3b	Local Variable	Stack	0x7ffe11d9dd7f	127	1016	285	631	383
f1_l1	Local Variable	Stack	0x7ffe11d9dd80	127	1016	285	631	384
f1_l2	Local Variable	Stack	0x7ffe11d9dd84	127	1016	285	631	388
f1_l5	Local Variable	Stack	0x7ffe11d9dd88	127	1016	285	631	392
f1_l6	Local Variable	Stack	0x7ffe11d9dd8c	127	1016	285	631	396
f1_l4	Local Variable	Stack	0x7ffe11d9dd90	127	1016	285	631	400
p	Local Variable	Stack	0x7ffe11d9ddb8	127	1016	285	631	440
buf	Local Variable	Stack	0x7ffe11d9ddc0	127	1016	285	631	448
argc	Formal Parameter	Stack	0x7ffe11e9ddfc	127	1016	286	631	508
argv	Formal Parameter	Stack	0x7ffe11e9df28	127	1016	286	631	808
env	Formal Parameter	Stack	0x7ffe11e9df38	127	1016	286	631	824
argv[0]	Formal Parameter	Command Line Arguments	0x7ffe11e9f3c5	127	1016	286	636	965
argv[argc-1]	Formal Parameter	Command Line Arguments	0x7ffe11e9f3c5	127	1016	286	636	965
env[0]	Formal Parameter	Environment	0x7ffe11e9f3ca	127	1016	286	636	970
env[1]	Formal Parameter	Environment	0x7ffe11e9f3da	127	1016	286	636	986

Question 3: Memory Map Analysis

Based on the experiment and analysis you have carried out in 1 and 2 above, answer the following questions:

a. what is the approximate size of the virtual address space of that process? Please express it in terms of tebibytes (TiB) and gibibytes (GiB).
b. what is the approximate size of each of the seven memory regions? Please express it in terms of tebibytes (TiB), gibibytes (GiB), mebibytes (MiB) and Kibibytes (KiB).
c. how does the compiler and operating system on your machine layouts the following process entities in the virtual address space: command line arguments, environment, literals, initialised global variables, uninitialised global variables, functions, formal parameters and local variables of a function, and dynamically allocated memories?
d. what is the order in which the formal parameters of a function are layout in the stack?

Answer 3: Part A

The approximate total size of this process is 4142 TiB and ~209 GiB.

Item	TiB & GiB	TiB only	GiB only
Process	4,142 TiB + ~209 GiB	~4,142.2 TiB	~4,241,617 GiB

Answer 3: Part B

Memory Region	Approximate Size	TiB only	GiB only
Code/Text	342 TiB + ~457.48 GiB	~342.44 TiB	~350,665.48 GiB
Initialised Globals	342 TiB + ~457.48 GiB	~342.44 TiB	~350,665.48 GiB
Uninitialised Globals	171 TiB + ~228.74 GiB	~171.22 TiB	~175,332.74 GiB
Heap	470 TiB + ~270.66 GiB	~470.26 TiB	~481,550.66 GiB
Stack	2302 TiB + ~885.02 GiB	~2303.86 TiB	~2,359,157.02 GiB
Command Line Arguments	255 TiB + ~1008.56 GiB	~255.98 TiB	~262,128.56 GiB
Environment	255 TiB + ~1008.56 GiB	~255.98 TiB	~262,128.56 GiB

Answer 3: Part C

Command Line Arguments

Command line arguments are typically stored on the stack. They are pushed to the stack upon the launch of the program (as arrays of character pointers).

Environment Variables

Environment variables are also typically stored on the stack. They are pushed to the stack after the command line arguments have been pushed to the stack. They are also usually stored as arrays of character pointers.

Literals

String literals and other constant variables are stored in a read-only section of memory for optimisation as they do not require writing to once stored.

Initialised Global Variables

Initialised global variables are stored in the *data* section of memory. They are allocated memory space at compile time and given their specified values.

Uninitialised Global Variables

Uninitialised global variables are stored in the BSS (Block Started by Symbol) section of memory. The allocation of memory also occurs at compile time with the values being initialised to 0.

Functions

Functions are stored in the *text/code* section of memory, which is marked as read-only.

Formal Parameters of a Function

Formal parameters of a function are stored on the stack section of memory and are local to the called function.

Local Parameters of a Function

Local parameters of a function are stored on the stack once the function is called. A stack frame is created once the function has been called, in which all local variables/parameters of the function are stored. Once the function has returned all the variables/parameters within the stack frame are de-allocated.

Dynamically Allocated Memories

Dynamically allocated memory is stored on the heap.

Answer 3: Part D

The formal parameters of a function are laid out in the stack in reverse order. This is due to the *cdecl* calling convention used in the C programming language.

Question 3

Write a C program that takes a list of command line arguments, each of which is the full path of a command (such as /bin/ls, /bin/ps, /bin/date, /bin/who, /bin/uname etc). Assume the number of such commands is *N*, your program would then create *N* direct child processes (ie, the parent of these child processes is the same original process), each of which executing one of the *N* commands. You should make sure that these *N* commands are executed concurrently, not sequentially one after the other. The parent process should be waiting for each child process to terminate. When a child process terminates, the parent process should print one line on the standard output stating that the relevant command has completed successfully or not successfully (such as "Command /bin/who has completed successfully", or "Command /bin/who has not completed successfully"). Once all of its child processes have terminated, the parent process should print "All done, bye-bye!" before it itself terminates.

Note: do not use function system, or any other function that will invoke a shell program in this question.

Answer: *main.c* Source Code Listing

Note: The following code was written and debugged in Vim launched from Ubuntu's command line. It was compiled using the GCC compiler in Ubuntu's command line and executed in Ubuntu's command line also. It has been opened in VS Code to copy the source code with syntax highlighting for readability.

```c
/* File Name:       main.c
 * Author:          Ben Royans
 * Date Modified:   04/09/2023
 * Assignment:      1
 * Question:        3
 */

#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>

int main(int argc, char* argv[])
{
    if (argc < 2)
    {
        fprintf(stderr, "Must have at least 1 command line argument!\n");
        exit(EXIT_FAILURE);
    }

    int n = argc - 1;
    pid_t childrenPID[n];

    // Fork and execute command line arguments
    for (int i = 0; i < n; i++)
    {
        // Fork the process
        pid_t curPID = fork();

        if (curPID == -1)
        {
            fprintf(stderr, "Failed to fork process");
            exit(EXIT_FAILURE);
        }
        else if (curPID == 0)
        {
            // This is child process, execute command line argument(s)
            int executionCode = execl(argv[i + 1], argv[i + 1], NULL);

            if (executionCode == -1)
            {
                fprintf(stderr, "Failure to execute command: %s\n", argv[i + 1]);
                exit(EXIT_FAILURE);
            }
        }
        else
        {
            // This is parent process, store child's PID
            childrenPID[i] = curPID;
        }
    }

    int successfulExecutions = 0;
    int unsuccessfulExecutions = 0;

    for (int i = 0; i < n; i++)
    {
        int status;
        waitpid(childrenPID[i], &status, 0);

        if (WIFEXITED(status))
        {
            switch(WEXITSTATUS(status))
            {
                case 0:
                    printf("[%s] has executed successfully\n", argv[i + 1]);
                    successfulExecutions++;
                    break;

                default:
                    printf("[%s] has not executed successfully\n", argv[i + 1]);
                    unsuccessfulExecutions++;
                    break;
            }
        }
    }

    printf("All done, bye-bye!\n");

    return (unsuccessfulExecutions == 0) ? 0 : 1;
}
```

main.c Testing

Test Case 1: Execute Program With No Arguments

Purpose of the Test

To test the handling of the command line argument count being less than 2.

Test Output

```
stupidflanders@stupidflanders:~/GitHub/ICT374_Ass1/Question 3$ ./q3
Must have at least 1 command line argument!
```

Explanation

This test demonstrates that executing this program without any command line arguments provides the user a statement stating that at least 1 command line argument should be included with the execution of the program.

Test Case 2: Execute Program With 1 Argument

Purpose of the Test

To test the execution of the program with a singular command line argument.

Test Output

```
stupidflanders@stupidflanders:~/GitHub/ICT374_Ass1/Question 3$ ./q3 /bin/who
stupidflanders tty2         2023-09-17 11:18 (tty2)
[/bin/who] has executed successfully
All done, bye-bye!
```

Explanation

This test demonstrates that when a single command line argument is provided, it is executed and then completes successfully. Command line argument given in this test case:

```
1.   /bin/who
```

Test Case 3: Execute Program With 2 Arguments

Purpose of the Test

To test the execution of the program with two command line arguments.

Test Output

```
stupidflanders@stupidflanders:~/GitHub/ICT374_Ass1/Question 3$ ./q3 /bin/who /bin/ls
main.c     q3
stupidflanders tty2         2023-09-17 11:18 (tty2)
[/bin/who] has executed successfully
[/bin/ls] has executed successfully
All done, bye-bye!
```

Explanation

This test demonstrates that when two command line arguments are provided, they are executed and then completed successfully. Command line arguments given in this test case:

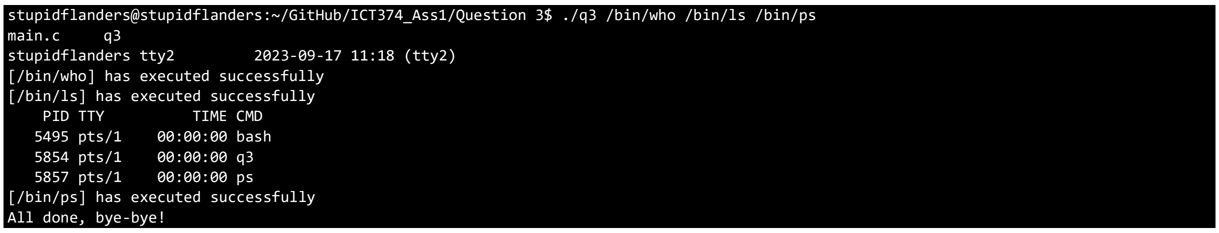
```
1.   /bin/who
2.   /bin/ls
```

Test Case 4: Execute Program With 3 Arguments

Purpose of the Test

To test the execution of the program with three command line arguments.

Test Output



```
stupidflanders@stupidflanders:~/GitHub/ICT374_Ass1/Question 3$ ./q3 /bin/who /bin/ls /bin/ps
main.c     q3
stupidflanders tty2         2023-09-17 11:18 (tty2)
[/bin/who] has executed successfully
[/bin/ls] has executed successfully
    PID TTY          TIME CMD
   5495 pts/1    00:00:00 bash
   5854 pts/1    00:00:00 q3
   5857 pts/1    00:00:00 ps
[/bin/ps] has executed successfully
All done, bye-bye!
```

Explanation

This test demonstrates that when three command line arguments are provided, they are executed and then completed successfully. Command line arguments given in this test case:

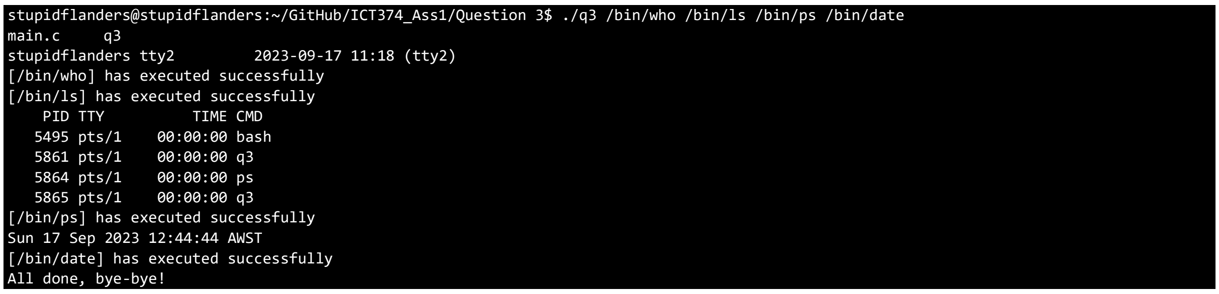
1. `/bin/who`
2. `/bin/ls`
3. `/bin/ps`

Test Case 5: Execute Program With 4 Arguments

Purpose of the Test

To test the execution of the program with four command line arguments.

Test Output



```
stupidflanders@stupidflanders:~/GitHub/ICT374_Ass1/Question 3$ ./q3 /bin/who /bin/ls /bin/ps /bin/date
main.c     q3
stupidflanders tty2         2023-09-17 11:18 (tty2)
[/bin/who] has executed successfully
[/bin/ls] has executed successfully
    PID TTY          TIME CMD
   5495 pts/1    00:00:00 bash
   5861 pts/1    00:00:00 q3
   5864 pts/1    00:00:00 ps
   5865 pts/1    00:00:00 q3
[/bin/ps] has executed successfully
Sun 17 Sep 2023 12:44:44 AWST
[/bin/date] has executed successfully
All done, bye-bye!
```

Explanation

This test demonstrates that when four command line arguments are provided, they are executed and then completed successfully. Command line arguments given in this test case:

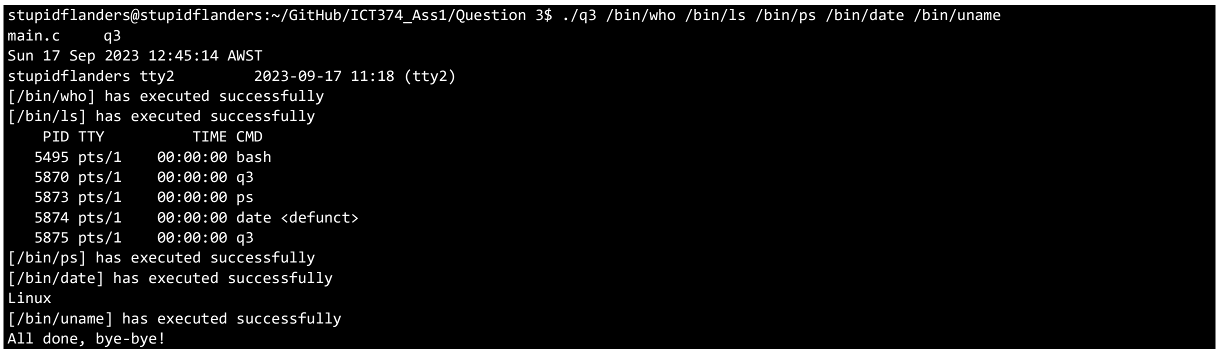
1. `/bin/who`
2. `/bin/ls`
3. `/bin/ps`
4. `/bin/date`

Test Case 6: Execute Program With 5 Arguments

Purpose of the Test

To test the execution of the program with five command line arguments.

Test Output



```
stupidflanders@stupidflanders:~/GitHub/ICT374_Ass1/Question 3$ ./q3 /bin/who /bin/ls /bin/ps /bin/date /bin/uname
main.c     q3
Sun 17 Sep 2023 12:45:14 AWST
stupidflanders tty2         2023-09-17 11:18 (tty2)
[/bin/who] has executed successfully
[/bin/ls] has executed successfully
    PID TTY          TIME CMD
   5495 pts/1    00:00:00 bash
   5870 pts/1    00:00:00 q3
   5873 pts/1    00:00:00 ps
   5874 pts/1    00:00:00 date <defunct>
   5875 pts/1    00:00:00 q3
[/bin/ps] has executed successfully
[/bin/date] has executed successfully
Linux
[/bin/uname] has executed successfully
All done, bye-bye!
```

Explanation

This test demonstrates that when five command line arguments are provided, they are executed and then completed successfully. Command line arguments given in this test case:

1. `/bin/who`
2. `/bin/ls`
3. `/bin/ps`
4. `/bin/date`
5. `/bin/uname`

Self-Diagnosis and Evaluation

All requirements in the C Programming exercise for Question 3 (main.c ⟶ q3) has been completed and tested to be working successfully. The program takes *n* number of command line arguments, forks the parent process and executes a particular command line argument on the child process. The parent process waits for the completion of the child processes and prints a message to the console detailing the success of the execution of the command. This program compiles using GCC with no errors or warnings.