Assignment 1

ICT374

Ben Royans 30493093

2023

Table of Contents

[Question 1: Process Switching 1](#_Toc145846022)

[1. Following a Clock Interrupt: 1](#_Toc145846023)

[2. Process A Reading a Work from Disk: 1](#_Toc145846024)

[3. CPU and Kernal Response to Completed Read Disk Operation: 1](#_Toc145846025)

[Question 2: The Memory Layout of a Process 2](#_Toc145846026)

[memory.c Code 2](#_Toc145846027)

[Memory Map Table 4](#_Toc145846028)

[Questions 5](#_Toc145846029)

[Part A 5](#_Toc145846030)

[Part B 5](#_Toc145846031)

[Part C 5](#_Toc145846032)

[Part D 6](#_Toc145846033)

[Question 3 7](#_Toc145846034)

[*main.c* Source Code 7](#_Toc145846035)

[Testing 8](#_Toc145846036)

# Question 1: Process Switching

## 1. Following a Clock Interrupt:

1. CPU Hardware: When a clock interrupt occurs, the CPU hardware performs the following actions:
   1. 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).
   2. Switches the CPU execution mode to kernel mode, which allows it to execute privileged instructions.
   3. 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:
   1. Determines whether it's time to perform a context switch:
      1. 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).
      2. If Process A still has time remaining in its time slice, the kernel updates its timer and returns control to Process A.
   2. If a context switch is required, the kernel updates the CPU's memory management unit to load the memory space of the selected process.
   3. Restores the saved state of the chosen process (Process B or C) from its Process Control Block (PCB).
   4. Sets the CPU's program counter to the saved value for the selected process.
   5. Resets the CPU execution mode to user mode, allowing the selected process to run.

## 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):
   1. Suspends Process A and places it in the blocked state, typically in a queue or a list of processes waiting for I/O operations.
   2. Initiates the disk read operation, which may involve scheduling the operation, configuring the hardware controller, and issuing the appropriate commands to the disk subsystem.
   3. 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.

## 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):
   1. Updates the status of Process C from a blocked state to a ready state, indicating that it is now ready to run.
      1. 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.
      2. 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.
   2. 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.

Top of Form

# Question 2: The Memory Layout of a Process

## *memory.c* Code

/\* 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;

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:

// 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", &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);

}

## Memory Map Table

*See also file Memory\_Map.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 |

## Questions

### 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 |

### 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 |

### 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.

### 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

## *main.c* Source Code

/\* 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;

}

## Testing