# Question 1: Process Switching

Pre-emptive multitasking in a modern operating system involves the cooperation of both the CPU hardware and the operating system kernel to efficiently manage multiple processes. Let's address each of your questions step by step:

1. Following a Clock Interrupt:

* CPU Hardware: When a clock interrupt occurs, the CPU hardware performs the following actions:
  + 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).
  + Switches the CPU execution mode to kernel mode, which allows it to execute privileged instructions.
  + Jumps to a predefined interrupt handler routine in the operating system kernel.
* Operating System Kernel: Upon receiving the clock interrupt, the kernel performs several tasks:
  + Determines whether it's time to perform a context switch, i.e., whether Process A has run for its time slice (time quantum). If so, it chooses the next process to run based on a scheduling algorithm and updates the process control block accordingly.
  + If Process A still has time remaining in its time slice, the kernel simply updates its timer and returns control to it.
  + If a context switch is required, the kernel updates the CPU's memory management unit to load the memory space of the selected process.
  + Restores the saved state of the chosen process (Process B or C) from its PCB.
  + Sets the CPU's program counter to the saved value for the chosen process.
  + Resets the CPU execution mode to user mode, allowing the selected process to run.

1. When Process A Needs to Read a Word from the Disk:

* Process A: When Process A initiates a disk read request, it enters a blocked state and yields the CPU.
* CPU Hardware: The CPU hardware, upon recognizing that Process A is blocked and cannot continue executing, generates an interrupt known as a "trap" or "system call" to transfer control to the operating system kernel.
* Operating System Kernel: Upon receiving the disk read request interrupt, the kernel performs the following actions:
  + Suspends Process A and places it in the blocked state, typically in a queue or a list of processes waiting for I/O operations.
  + Initiates the disk read operation, which may involve scheduling the operation, configuring the hardware controller, and issuing the appropriate commands to the disk subsystem.
  + 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.

1. When the Read Disk Operation for Process C Completes:

* Disk Hardware: The disk hardware performs the requested read operation and signals its completion to the operating system.
* Operating System Kernel: Upon receiving the completion signal, the kernel performs the following actions:
  + Updates the status of Process C from a blocked state to a ready state, indicating that it is now ready to run.
  + 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.
  + If Process C is not immediately scheduled, the kernel may continue running the currently running process (e.g., Process B) or select another process for execution based on its scheduling policy.
  + The kernel handles the context switch if necessary and updates the CPU hardware to run the selected process.

In summary, pre-emptive multitasking involves a seamless cooperation between the CPU hardware and the operating system kernel. The hardware generates interrupts (e.g., clock interrupts, I/O interrupts) to trigger kernel intervention, and the kernel manages the scheduling of processes, context switches, and I/O operations to ensure efficient and fair execution of multiple processes. This coordination allows for the illusion of concurrent execution of 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** | **BiB** |
| 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**

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?

**Part D**

what is the order in which the formal parameters of a function are layout in the stack?