# 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 file Memory\_Map.ods.*

## Questions

1. The approximate total size of this process is

|  |  |  |  |
| --- | --- | --- | --- |
| **Item** | **TiB & GiB** | **TiB only** | **GiB only** |
| Process | 4142 TiB and ~209 GiB. | ~4142.2 TiB | ~4241617 GiB |

|  |  |  |  |
| --- | --- | --- | --- |
| **Memory Region** | **TiB & GiB** | **TiB only** | **GiB only** |
| Code/Text |  |  |  |
| Initialised Globals |  |  |  |
| Uninitialised Globals |  |  |  |
| Heap |  |  |  |
| Stack |  |  |  |
| Command Line Arguments |  |  |  |
| Environment |  |  |  |

# Question 3: Executing Commands in Child Processes