Operating Systems - Monsoon 2022
Mid-Semester Examinations Grading Rubric

Arani Bhattacharya, Sambuddho Chakravarty

Date: Oct 19, 2022

Student's Details

Name of the Student:

Roll Number:

Stream:

Question Structure and Instructions

This paper consists of three sections:

1. The first section consists of 8 short conceptual questions each carrying 3 points.
2. The second section consists of 2 longer questions each carrying 5 points.
3. The third section consists of 2 long questions each carrying 8 points.

You are required to answer the questions in the question paper itself in complete sentences, but as briefly as possible. We have given an additional 1 page at the end if extra space is needed, but please note that no extra sheets will be given. You need to write your roll number on each page of the question paper.

1 Short Conceptual Questions

There are 8 questions in this section each worth 2 points. You may write the answers to these questions succintly.

1. The system call `write()` can be used to write text to a file. Can it be executed using the CALL instruction, or is an INT or SYSCALL instruction absolutely necessary to use it? Justify the reason.

   Answer: No, SYSCALL/INT is essential as the write() function is a system call (*i.e.* implemented inside the OS). The only way to get there is via a software interrupt. It is in a different address space all together. The write() system call requires operations that can only be executed via kernel's access to functions and drivers.

   Grading Scheme:

   (a) No – 1 mark

   (b) CALL used to invoke ordinary functions, not system calls – 2 marks

   OR

CALL can be used indirectly to invoke the write() system call by using a wrapper function with the same name.

Grading Scheme:

(a) Yes – 1 mark

(b) CALL used to invoke wrapper function – 2 marks

2. Is it possible for Completely Fair Scheduler (CFS) to have so many process switches that more time is spent on process switch than on actual running of the process? Justify in one line the reason.

   Answer: No, this is not possible. This is because at some point, when there are too many processes in a system, and the time slice per process is never reduced below a threshold (it is floored to a particular value below which it is never reduced). This is becomes less than the time it takes to perform a context switch. At this stage, the scheduling action is not fair anymore.

Grading Scheme:

(a) No – 1 mark

(b) Time slice limited/threshold – 2 mark

3. Describe two advantages of the UEFI firmware over traditional BIOSes.

Answer:

(a) UEFI firwares switch to long mode (64-bit) by themselves and doesn't require the OS developer to do so.

(b) Traditional BIOSes could support only 4 primary partitions. This number is over 128.

(c) In case of traditional BIOSes, each primary partition could be of 2TB size. This number for UEFI is about 9ZB.

Grading Scheme: Any two of the above reasons – 1.5 points per reason

4. Do modern OSes generally tend to disable interrupts while handing system calls and interrupts? If yes then explain why and what are its advantages. Else, if you think they aren't then also explain why and the corresponding advantages.

Answer: No they usually do not require to disable interrupts. Only in some cases the interrupts are disabled, example when nested interrupts could lead to inconsistent states within the kernel (*e.g.* during process creation). Long system calls are preemptible. The process control block has information of which system call was being executed when an interrupt (and eventual context switch) occurs. This allows resuming system calls. The OS also needs to ensure that most internal system call implementations be thread safe, to avoid any kind of race condition. This model improves system interactivitiy, especially required for non-realtime OSes *e.g* those used for desktops and mobiles.

Grading Scheme:

(a) No – 1.5 mark

(b) System calls are preemptible – 1.5 marks

5. Can JMP (Jump) substitute the use of CALL (procedure/function call) and INT (software interrupt) instructions? If yes, justify how and if not, then explain why not.

Answer: Yes, JMP can substitute CALL instruction if the address of the next instruction is pushed before to the stack. Directly pushing the EIP/RIP register won't work, as x86 doesn't allow that. However, if the offset to the next instruction is present in the form of a label to another address then it can be. However, JMP cannot substitute INT instruction, since INT also changes privilege level, along with saving the flags on the stack, and sends control to the kernel code. Further, return from the interrupt handler usually involves the IRET instruction that pops the flag register values from the stack and restores it appropriately.

Grading Scheme:

(a) Yes for JMP/CALL – 1 mark

(b) Using PUSH/Cannot use EIP,RIP register directly – 0.5 marks

(c) No for JMP/INT – 1 mark

(d) Changes privilege level/sends control to kernel – 0.5 marks

6. Is copy-on-write used for both `pthread_create()` and `fork()`? Explain when it is needed or not needed and why.

Answer: Yes it is used for both as both use the `clone()` system call. However, for the case of threads created with pthread_create() function, the two threads involve sharing the same code and stack. A new thread definitely requires a new stack and code section to run on. Rest of the memory regions are shared with the parent thus there is no need for a copy anyways. Thus, while technically both involve clone() that uses copy-on-write functionality, for threads it has no significance.

Grading Scheme:

(a) Yes for fork() – 1.5 marks

(b) No/No significance for pthread_create() – 1.5 marks

(c) If both fork() and pthread_create() eventually CoW is used – 3 marks

7. Which of the system calls among `fork()` and `vfork()` has higher overhead and why?

   Answer: Historically speaking, while `fork()` did a complete clone, vfork() didn't involve copying any of the code sections (or even setting up appropriate page table entries) until the child thread write to write to any of the sections. As of now, `fork()` also employs COW feature, just that it still however sets up appropriate page table entries for the child process.

   Grading Scheme:

   (a) fork() has higher overhead – 1.5 marks

   (b) fork() uses copy-on-write/sets up memory, vfork() does not copy page tables – 1.5 marks

8. Consider two processes P1 and P2. Process P1 has not created any new thread, whereas P2 has created a new thread by calling `pthread_create`. How many `task_struct` data structures are present in total, and why?

   Answer: There are a total of 3 task_struct's, one because of P1 and two because of each thread of P2.

   Grading Scheme:

   (a) 3 task_struct's in total – 3 points.

   (b) 1 task_struct for P1, 2 task_structs for P2 – 1.5 marks

Roll Number:

2 Long Questions

1. Suppose you create a structure in assembly which contains an array of 5 chars followed by a 64-bit integer. Will this structure be usable by simply defining the variables, or are additional commands necessary? What if we reverse the sequence of variable definitions?

   Answer: Yes, after defining an array of 5 chars, you need to explicitly add an "align" statement to align the memory. Only after proper alignment, can the 64-bit integer be defined. This is not needed if we reverse the definitions, since by default the integer is of 64 bits.

   Grading Scheme:

   (a) First part, no – 1.5 marks

   (b) First part, requires "align" after char – 1.5 marks

   (c) Second part, yes – 2 marks

2. Suppose you want to use an assembly instruction "MOV rax, #5" from within a C program. Is it possible syntactically to embed this instruction within the C program? If possible, then is it the right way to do it or is an alternative approach better?

   Answer: Yes, it is possible to use it synctatically, but it is not recommended as the register rax may be used by some other instruction of the compiler. It is much better to use extended assembly syntax with the required modifier to ensure that the value in rax is not destroyed.

   Grading Scheme:

   (a) Yes – 1 mark

   (b) Not recommended – 1 mark

   (c) Register rax might be used – 1 mark

   (d) Recommended way is to use extended assembly/modifier – 2 marks

3 Very Long Questions

1. (a) Consider the FIFO scheduling policy. Is this policy preemptive in nature? If so, justify. If not so, how would the policy change if it is made preemptive?

   Answer: By default FIFO is not preemptive in nature. The processes/jobs are chosen to run as and when they arrive. If existing processes are running they are not preempted. One way to make it preemptive is to add a clock timer which would invoke the scheduler that would in turn select a different process to run. This would make the policy similar to round-robin.

   Grading Scheme:

   i. No – 0.5 marks

   ii. Processes can run as long as necessary / is not preemtpive by default – 0.5 marks

   iii. Add clock timer / made to round-robin – 1 mark

   (b) Is the Shortest Job Remaining First a good policy for interactive processes? If so, why then is it not used in practice? If not so, then justify why it is not.

   Answer: Yes, it is a good policy for interactive processes. However, it can lead to starvation for CPU-bound processes, as their jobs are typically longer in length. Such jobs may not get access to the processor at all.

   Grading Scheme:

   i. Yes – 1 mark

   ii. Possible starvation for CPU-bound processes – 1 mark

   (c) Consider the scheduler of the Linux kernel where two processes in `SCHED_NORMAL` class have different nice values. Can a process with higher nice value suffer from starvation and why?

   Answer: No, processes with high nice values cannot suffer from starvation because the CFS scheduler ensures that every process gets at least some of the processor time.

   Grading Scheme:

   i. No – 1 mark

   ii. CFS scheduler ensures every process gets some time – 1 mark

   (d) Are there corner cases when the linux CFS scheduler ceases to be "fair"? If so, explain how this could happen.

   Answer: Yes, CFS stops being fair when the time slice cannot be made any smaller due to a large number of processes being present. In those cases, processes may have to wait for much longer than usual to get access to the processor.

   Grading Scheme:

   i. Yes – 1 mark

   ii. When number of processes present is too high – 1 mark

2. Write a pseudo-code / snippet for a program that would involve starting multiple threads each reading and printing the contents of individual files and printing it on screen. Your code need not mention the exact flags needed, but need to utilize the sequence of functions that are needed to construct a correct program.

   Answer: The program requires two functions. One function should create threads and join them. The right sequence of functions to use is `pthread_create()` and `pthread_join()` (should be invoked at least 2 or more times). `pthread_create()` The other function should use `open() / fopen()`, `read() / fscanf() / fgets()`, `write() / fprintf() / fputs()` and `close() / fclose()` functions.

   Grading Scheme:

   (a) Multiple threads created using `pthread_create()` , but no `pthread_join()` and the read/print operations from two files – 2 marks.

   (b) Multiple threads created using `pthread_create()` , and `pthread_join()`, but no read/print operations from two files – 4 marks.

   (c) Multiple threads created using `pthread_create()` , along with `pthread_join()` and the read/print operations from two files – 8 marks.