

Principles of Operating Systems — Midterm exam #1

October 13, 2025 — Duration: 90 minutes

Important instructions: All documents are forbidden, except 1 (dual-side) handwritten A4 paper sheet. All electronic devices are forbidden (calculators, computers, mobile phones, etc.). The number of points indicated for each exercise is only provided to give you an idea of its weight. We reserve the right to change the exact number of points.

Note: For each problem, unless explicitly mentioned otherwise, all the questions are independent.

For each exercise with a multiple choice question, simply answer (on your separate answer sheet) by indicating the question number followed by one or several letters corresponding to the correct statement(s) — **you are not required to justify your answers**. Note that, **for each question, there is always at least one correct statement, and possibly several correct statements**. A question without any answer is considered empty (no points). A single mistake in an answer (i.e., exactly one missing correct statement or one incorrect statement) voids half of the points allocated for the question. A number of mistakes greater than one voids all the points allocated for the question (there are no negative points).

Problem 1 (5 points)

Question 1.1 Among the following statements, which one(s) is (are) true?

- (a) If two operating systems OS_1 and OS_2 provide the same ABI (*Application Binary Interface*), this implies that the executable files compiled for OS_1 will also work on OS_2 (assuming that all the libraries required by the applications are available on both systems).
- (b) In the case of Linux, the device drivers (for disks, network interfaces, and other peripherals) typically run with kernel privileges. A bug in one of these drivers can jeopardize the safety or security of the whole operating system.
- (c) In the case of Linux, performing an access to read or write into the execution stack from user mode (i.e., from the code of the application or a library) always requires invoking a system call.
- (d) We consider a Linux process running on an Intel x86 32-bit processor supporting virtual memory via paging, using a page size of 4096 bytes (as studied in the lectures). If virtual memory address `0x12345ABC` is valid, then the last (i.e., least significant) three hexadecimal digits of the corresponding physical address are necessarily `0xABC`.

Question 1.2 Among the following statements, which one(s) is (are) true?

- (a) Increasing the duration of the “time quantum” (or “time slice”) used by a scheduler allows decreasing the overhead of the operating system (a greater fraction of CPU time is spent in the application code).
- (b) For a general-purpose scheduler, the duration of a time quantum is usually in the range of 10 nanoseconds.
- (c) Performing context switches between processes too frequently on a CPU can reduce the efficiency of the CPU hardware caches.
- (d) Processes are always protected from each other. For this reason, an OS such as Linux never shares a physical page of memory between several processes.

Question 1.3 Among the following statements, which one(s) is (are) true?

- (a) In a process with multiple threads, it is possible to dynamically allocate a memory block (via `malloc`) from a thread T_1 , to pass a pointer to this block to another thread T_2 , and to later release this block (via `free`) from T_2 .
- (b) In a multithreaded process, each thread requires a distinct stack.
- (c) In an OS such as Linux, the `malloc(...)` and `free(...)` primitives are implemented as library functions (as opposed to system calls). This design choice is motivated by performance reasons.
- (d) On a typical desktop or server machine today, reading some data from the disk is much faster than executing a pair of CPU mode switches (from user to kernel mode and vice versa).

Question 1.4 Let us consider the program shown in Figure 1. Among the following statements, which one(s) is (are) true?

- (a) The exact text string displayed by this program may vary between successive executions (even on the same machine and the same OS version).
- (b) An execution of this program may display the following text string: 01120
- (c) An execution of this program may display the following text string: 00121
- (d) An execution of this program may display the following text string: 01233

```
1  /* Notes:
2   * - #include directives are omitted for simplification
3   * - fflush(stdout) forces the output of printf to be displayed immediately
4   * - fork returns 0 for the child, and the child's pid for the parent
5   */
6  int x = 0;
7  int main(int argc, char **argv) {
8      printf("%d", x); fflush(stdout);
9      if (fork() == 0) {
10         x++;
11     }
12     if (fork() == 0) {
13         x++;
14     }
15     printf("%d", x); fflush(stdout);
16     return 0;
17 }
```

Figure 1: Code listing for `prog1.c`

Question 1.5 Among the following statements, which one(s) is (are) true?

- (a) Caching is an optimization technique that works usually well because of the *locality* principle. This principle means that, in practice, most applications frequently create and destroy local variables.
- (b) Current processors are typically equipped with several hardware caches (often named L1, L2, and L3 caches). In such caches, the “replacement policy” algorithm is fully implemented in hardware for performance reasons.

- (c) In a language that supports garbage collection (for example, Java), the garbage collector is automatically triggered upon the termination of every function call (in order to deallocate the local variables).
- (d) For an application written in C or C++, using a library supporting garbage collection can help reducing external fragmentation in the memory heap by compacting (i.e., moving) allocated blocks.

Problem 2 (3.5 points)

For all the questions of this exercise, we consider a traditional operating system like Linux running on a machine **with N CPUs**.

Question 2.1 In this question only, we assume that each process has a single thread.

For each state below (*Ready*, *Running*, *Blocked*¹), indicate (on your answer sheet, not on this question sheet) the maximum number of processes that can simultaneously be in that state.

Select only one answer for each line. You are not required to justify your answers.

Process state	Answer			
Running	1	N-1	N	More than N
Ready	1	N-1	N	More than N
Blocked	1	N-1	N	More than N

Question 2.2 For each of the state transitions listed below, describe very briefly (i.e., in one or two sentences, less than 5 lines) an event/situation that would cause a thread to undergo this transition.

- (a) Transition from *Ready* to *Running*
- (b) Transition from *Running* to *Ready*
- (c) Transition from *Running* to *Blocked*
- (d) Transition from *Blocked* to *Ready*

Question 2.3 We consider a single process P_i with X threads, running on the above-mentioned OS and machine.

(a) At a given point in time during the lifespan of P_i , is it possible to have all its threads simultaneously in state *Running*? Simply answer by “Yes” or “No”.

(b) If you answered “No” to the previous question, briefly justify your answer. If you answered “Yes”, briefly discuss the assumption(s) (for example, about the application and/or the OS) under which this is possible.

Problem 3 (3 points)

Question 3.1 In this question, we consider a memory allocator with similar characteristics to the allocator design studied in Lab 2, except the following difference: each requested allocation size is rounded up to the closet (superior) power of two. We want to compare this new design with the one studied in lab 2. Answer the following questions.

¹Note that the *Blocked* state is also known as the *Waiting* state in some documents.

- (a) Briefly describe one potential drawback of this new design compared to the one from the lab.
- (b) Briefly describe one potential advantage of this new design compared to the one from the lab.

Question 3.2 One of your friends is concerned with (external and internal) heap fragmentation for his/her application (written in C). After some thinking, he/she makes the following claim: *“I know that the operating system that I use implements virtual memory via paging with a single page size (e.g., all pages have a size of 4096 bytes). Therefore, this will almost eliminate any fragmentation issue within my application’s memory heap (i.e., `malloc` requests will not fail while there is sufficient aggregate free space in the heap to fulfill them). Indeed, the use of paging with a single page size will completely eliminate external fragmentation within the heap and internal fragmentation will be limited to at most one page.”*

Do you agree with this claim? Justify your answer.

Problem 4 (4.5 points)

In this exercise, we consider a machine with a single CPU that implements virtual memory via segmentation, using an explicit segment identifier. The CPU architecture has the following characteristics:

- There are 4 segments per address space.
- All segments are configured for growth from the base towards higher addresses (like in the lectures).
- Each virtual address (including the segment ID) is stored on 12 bits.
- Each physical address is stored on 11 bits.
- For each segment of the currently running process, the MMU is configured via 3 registers:
 - *Base address*
 - *Size* (in bytes)
 - *U/K permissions*: a boolean indicating if the segment is accessible from user mode (0) or only from kernel mode (1). For simplification, we do not consider the configuration of read-write permissions in this exercise (all the segments have read-write permissions).

We also assume that the operating system always sets up 4 segments for each process: three segments for the application and one for the kernel (kernel code and data are in the same segment). We also assume that the kernel (code and data) is mapped in the same address range for all the processes. In contrast, the size of a user segment is potentially different from one process to another. Besides, the operating system does not support disk swapping (the segments always remain in main memory until the termination of a process).

Important: For this exercise, all the addresses are provided in *decimal notation*. **The answers submitted for this exercise must use the same notation.**

Question 4.1 Process P_1 is currently running and the operating system has configured the MMU as shown in the table below:

Segment	Base	Size	U/K
0	96	32	0
1	896	128	0
2	384	384	0
3	1536	512	1

- (a) If the user-level code of P_1 performs a memory access to virtual address 25, will it be successful? And if so, what will be the corresponding physical address? In any case, briefly justify your answer.
- (b) For each segment of P_1 , give the range of valid virtual addresses and the corresponding range of physical addresses. You are not required to justify your answer.

Question 4.2 Let us now assume that the operating system must launch a new process P_2 , while P_1 is still running. The size requirements for the user segments of P_2 are as follows:

- Segment 0: 512 bytes
- Segment 1: 224 bytes
- Segment 2: 112 bytes

(a) Explain in which physical address range each segment of P_2 must be mapped (without impacting P_1). Briefly justify your answer.

(b) Provide a table (like the one above) to summarize how the MMU registers must be configured for running P_2 . You are not required to justify your answer.

Problem 5 (4 points)

We consider a machine with a single Intel x86 32-bit CPU, configured to use a 2-level paging structure and a (single) page size of 4096 bytes — as studied in the lectures. The format of the main data structures is provided in Figure 2.

Additional reminders:

- Each virtual address is stored on 32 bits.
- Each physical address is stored on 32 bits.
- The size of a page directory (PD) instance is 4096 bytes.
- The size of a page table (PT) instance is 4096 bytes.
- The size of a page directory entry (PDE) is 4 bytes.
- The size of a page table entry (PTE) is 4 bytes.

Moreover, we assume that the operating system does not use disk swapping (i.e., each valid virtual page is always stored in main memory).

Question 5.1 For each of the following cases, give the complete size (in number of pages) required by the paging structure of the process. Briefly justify your answers.

Note that, for simplicity, these examples are not necessarily realistic regarding the number of valid virtual memory regions and/or their sizes.

(a) Process P_1 has a single valid virtual memory region, whose size is 4096 bytes.

(b) Process P_2 has a single valid virtual memory region, which corresponds to the upper half of the virtual address space.

(c) Process P_3 has 5 valid virtual memory regions:

- range [0x00001000 ; 0x003FEFFF]: application code
- range [0x003FF000 ; 0x003FFFFFFF]: global variables
- range [0x00C00000 ; 0x00FFFFFFF]: heap
- range [0x7FFFC000 ; 0x7FFFCFFF]: stack
- range [0x80000000 ; 0xFFFFFFFF]: kernel code and data

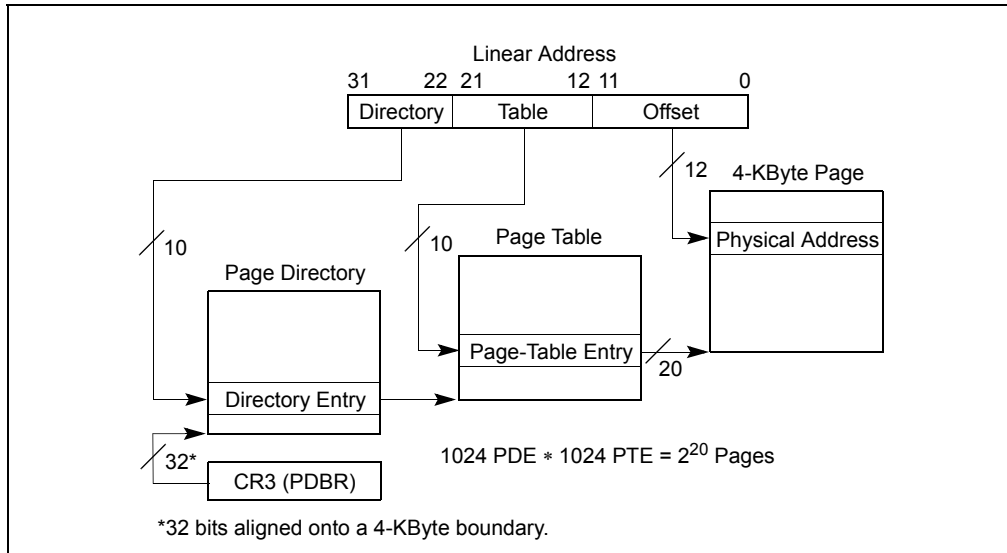


Figure 2: x86 address translation with 4-kilobyte pages (source: Intel documentation)