Univ. Grenoble Alpes (Ensimag & UFR IM2AG)
M1 MoSIG
Academic year 2025-2026

Principles of Operating Systems — Final Exam
December 2025

Duration : 3 hours
All documents are forbidden except one (dual-side) handwritten A4 paper sheet.
All electronic devices are forbidden.
The number of points per exercise is only provided for indicative purposes.
The grade will take the quality of the presentation into account.

This exam is made of two parts. Use distinct answer sheets for each part.

Part I

Problem 1 (3 points)

1.1 In this question, we consider a Linux application with the following behavior:

- The application uses 2 execution flows (hereafter named T_1 and T_2) implemented using standard Linux threads.
- As an initial input, T_1 is provided with a list L of file names.
- T_1 executes a loop, which can be summarized as follows. For each file F in L, T_1 obtains its size, dynamically allocates a buffer B of the required size, reads F's data from disk into B, and then forwards the data to T_2 using a producer-consumer scheme. We assume that each file has a small size and can therefore be read entirely with a single disk I/O request.
- T_2 executes a loop, which can be summarized as follows. For each file, T_2 receives (in one batch) the corresponding data provided by T_1, performs an analysis of the data (this analysis requires a single sequential reading pass over all the data) and prints a new line (on the standard output) about the results of the analysis.

Your friend Alicia implements 3 different versions V_i of this application:

- V_1: In this version, T_1 and T_2 are running in distinct processes and the producer-consumer scheme is implemented via a Unix pipe.
- V_2: In this version, T_1 and T_2 are running in the same process and the producer-consumer scheme is implemented using the same approach as the one studied in the lectures and labs, i.e., using a mutex lock and two condition variables, manipulated with pthread functions.

- V_3: In this version, T_1 and T_2 are running in distinct processes and the producer-consumer scheme is implemented with a shared memory segment (created via `mmap()`), and (like in V_2) using a mutex lock and two condition variables, manipulated with pthread functions.[1]

We assume that all of these implementations are correct and implemented as efficiently as possible.

1.1.a) Simply indicate the expected performance hierarchy for the 3 implementations, using the following notation:

$$1)\ V_x \qquad 2)\ V_y \qquad 3)\ V_z$$

This notation must be interpreted as follows: Version V_x has a shorter (total) execution time than V_y, which itself has a shorter execution time than V_z.

1.1.b) Briefly but precisely mention all the reasons that justify your previous answer. In particular, if some parts of your answers rely on specific assumptions regarding how the file data is transferred between T_1 and T_2, you should discuss these assumptions explicitly.

1.2 In the lectures, we have seen that threads can be implemented in different ways by an operating system. In particular, we have discussed two main dimensions:

- Threads can be based either on a *preemptive* (P) or a *cooperative* (C) model.
- Threads can be managed either at the *user level* (U, also known as a *N:1 approach*) or at the *kernel level* (K, also known as a *1:1 approach*). (In this question, we do not consider the hybrid *N:M approach*.)

Overall, this leads to 4 possible models: PU, PK, CU, CK.

We have seen that the default threading implementation used in Linux is based on the PK model. Among the 3 other models, indicate which one is generally the least popular choice for current operating systems and briefly explain why.

1.3 Your friend Bob has written his own threading implementation, which has the following characteristics:

- it is based on the PU threading model (please see the definition given in the previous exercise);
- it is implemented as a library (named `MyThreads`), which is compatible with the Linux operating system;
- the library only implements the following functions, whose purpose is similar to the ones studied in class (for brevity, we omit here the full function signatures including arguments and return values): `mythread_create()`, `mythread_exit()`, `mythread_join()`, and `mythread_yield()`.

[1] Note that pthread mutex locks and condition variables can be used between threads running in the same process or in different processes. The pthread implementation does not introduce any intrinsic difference (in terms of correctness or performance) between these two kinds of setups.

Bob now writes an application to test his library on his Linux machine. The application uses two threads (T_0 and T_1, both created via `mythread_create()`) and implements a producer-consumer buffer managed with 3 semaphores (similarly to the producer-consumer algorithm studied in the lectures and labs). T_0 acts as a producer and T_1 acts as a consumer. Note that, since Bob's library does not provide its own implementation of the semaphores, the application code uses the functions provided by the standard pthread semaphore implementation of Linux (`sem_init()`, `sem_wait()`, `sem_post()`, as shown in the Appendix of this exam document).

Just before running it for the first time, Bob realizes that this application will not work correctly.

1.3.a) Briefly describe what will be the observed (incorrect) behavior of the application.

1.3.b) Briefly (but precisely) explain the cause of this incorrect behavior.

Note: You are not expected to explain how Bob's design/implementation should be modified to avoid the problem. Simply explain what is the problem (symptom) and why it occurs.

Problem 2 (4.5 points)

Question 2.1 In this question, we consider a machine with the following characteristics:

- The machine has a single CPU.
- The machine implements virtual memory via paging, with a single page size of 4096 bytes.
- The MMU (memory management unit) of this machine includes a "tagged" TLB (i.e., each TLB entry has a field storing the corresponding "Address Space ID").
- The CPU instruction set of this machine features a specific instruction : `TLB_INVALIDATE start_VPN nb_Pages`. This instruction invalidates the TLB entries corresponding to the address range defined by (`start_VPN, nb_Pages`). If the corresponding entries are not present in the TLB, the execution of the instruction produces no effect.
- The machine runs the Linux operating system.

Briefly describe two distinct situations in which the Linux kernel must necessarily use the `TLB_INVALIDATE` instruction. Hint: For example, you could consider the side effects of two distinct (and well-known) Linux system calls.

Expected answer length: Approximately 5–10 lines.

Question 2.2 In this question, we study a new hardware mechanism introduced in some recent 64-bit processors (the description below is adjusted/simplified for pedagogical purposes). This mechanism is based on the following principles:

- In essence, the mechanism extends traditional virtual memory translation (based on paging) with additional protection checks. It relies on the notions of *secret* and *key*, which are implemented as integer values (of the same size). The mechanism allows choosing a *secret*

and associating this secret to a (small) virtual address range inside a process address space. When performing a (read or write) memory access to an address associated with a secret value, the instruction must provide a *key* that matches with the secret.

- The secret values associated with the virtual address space of the currently running process are stored in a special memory region managed by the kernel. For simplicity, we omit the hardware/kernel details regarding the storage and retrieval of the secrets for the different processes.

- The CPU provides an instruction `REGISTER_MEM_SECRET`. It allows registering a 4-bit *secret value*, associated with a 16-byte aligned virtual memory range of the current process. For example, the following instruction allows registering the secret value `0xC` for the virtual address range [`0x00057800` ; `0x0005780F`] of the currently running process:
`REGISTER_MEM_SECRET 0xC 0x00057800`

- The CPU also provides another instruction `CLEAR_MEM_SECRET`. It allows removing the secret previously set for a given region. For example, the following instruction allows removing the secret value that was previously set for the virtual address range [`0x00ABC000` ; `0x00ABC00F`] of the currently running process:
`CLEAR_MEM_SECRET 0x00ABC000`

- The execution of the two above-mentioned instructions is allowed in user mode, as long as the target virtual address range is accessible from user mode.

- Regarding the storage of the key (that must be provided when performing a memory access), the mechanism leverages the fact that, within the 64-bit format of the virtual addresses, some of the top bits are ignored by the (paging-based) address translation mechanism. Among these unused bits, 4 bits are re-purposed to store the key.

- When the CPU executes an instruction referencing a (virtual) memory address V, before performing the address translation, the CPU hardware checks if a secret value has been registered for the 16-byte aligned virtual address range to which `V` belongs. If a secret value is registered, the CPU checks if this value matches with the key contained in `V`. In case of correct match (or absence of registered secret), the address translation proceeds normally. In case of mismatch, the CPU triggers a trap to kernel mode.

In the lectures and labs, we have seen that application-level bugs causing incorrect accesses to data and/or metadata stored in the heap can introduce various kinds of problems.

2.2.a) The above-mentioned hardware facility can be useful to improve/facilitate the detection of various types of bugs related to incorrect heap memory accesses. Briefly (but precisely) describe one such type of bug.

2.2.b) Describe how a typical heap memory allocator design (such as the ones studied in the lectures) could be modified to leverage this hardware feature, in order to detect the chosen type of bug (i.e., the type of bug described in the answer given to the previous question).

Expected answer length: Approximately 10 lines.

Question 2.3 In this exercise, we study the virtual memory features available in some recent versions of ARM 64-bit processors (some details have been omitted/modified for simplification).

The virtual addresses are stored on 64 bits. However, within a virtual address, the top 16 bits (bits 63–48) are subject to specific rules. These bits must be either all set to 0 or all set to 1. If they are set to 0, the virtual address belongs to the *"user"* part of the address space. If they are set to 1, the virtual address belongs to the *"kernel"* part of the address space.

The physical addresses are stored on 48 bits.

As we have seen in the lectures, an Intel x86 CPU has a single register (named `CR3`) pointing to the root of the current paging structure. In contrast, the ARM CPU architecture that we study here has two distinct *Translation Table Base Registers (TTBR)*:

- `TTBR0` points to the root of the paging structure that is used for the translation of addresses that belong to the *"user"* part of the address space.
- `TTBR1` points to the root of the paging structure that is used for the translation of addresses that belong to the *"kernel"* part of the address space.

Note that bits 63–48 in a virtual address are only used to select the involved TTBR. They have no other impact on the address translation process.

Describe/explain one reason that motivates this design (i.e., the choice to have two distinct registers). (Expected answer length: about 5–10 lines.)

Question 2.4 In this exercise, we still consider the ARM 64-bit processors, already (partially) described in the previous exercise (i.e., Question 2.3 — make sure to read it first.)

The general principles of the paging structure are fairly similar to the ones of Intel x86 64-bit processors (studied in the lectures). In particular, a paging structure can have up to 4 levels (we will name these levels T0, T1, T2, T3) and supports multiple page sizes. However, our processor provides more configuration options and some of the data structures are different. We will only focus on the main configuration parameter, which is called the *granule size* and is configured via a specific privileged CPU register.

The value chosen for the granule defines the size of the smallest possible page size that can be configured (for example, a granule size of 4 kB implies that the smallest page size is 4 kB). Furthermore, it also has an impact on the other page sizes that can be used, and the size of the different structures (T0, T1, T2, T3).

Regarding the data structures (T0, T1, T2, T3), note that:

- For all these data structures, the size of an entry is always the same: 64 bits (8 bytes).
- Unlike the design of Intel processors, these data structures do not necessarily all have the same size (and this size may be smaller or greater than 4 kB).

- For a given choice of granule, T1, T2, and T3 structures always have the same size, which is equal to the granule size. However, note that, depending on the setup chosen for a given (valid) virtual address range, T2 and T3 structures may or may not exist.
- The size of T0 can vary significantly depending on the chosen granule size. For some setups, it can be significantly smaller than the granule size.

There are 3 possible values for the granule size: 4 kB, 16 kB and 64 kB.

As an illustration, let us consider the case of a 4kB granule size. In such a setup, in order to define a valid virtual address range, there are always at least 2 levels of structures (T0, T1), and up to 4 levels (T0, T1, T2, T3), and there are 3 page sizes that can be used: 4 kB, 2 MB and 1 GB.

2.4.a) In the case of a 4 kB granule, what is the number of entries in a T1/T2/T3 structure? You can provide your answer using the following format: 2^x entries. Briefly justify your answer.

2.4.b) In the case of a 64 kB granule, in order to define a valid virtual address range, there are always at least 2 levels of structures (T0, T1), and up to 3 levels (T0, T1, T2).

- What is the number of entries in a T1/T2 structure? Briefly justify your answer.
- What is the number of entries in a T0 structure? Briefly justify your answer.
- What are the different page sizes that can be used? Briefly justify your answer.

Problem 3 (2.5 points)

A team of hardware and software engineers is working on the design of a new computer platform. Given that this is a completely new product, there are no constraints of backwards compatibility with existing applications. Besides, the team has full control over the hardware design, the operating system design, and the design of the main applications.
Below is a list of technical statements made by some of the team members. For each of the statements, you are asked (i) to say if the highlighted part of the text (*in italic font*) is technically correct or incorrect and (ii) to briefly justify your answer. Expected answer length: about 5-10 lines for each statement.

Notes:

- In this exercise, the part of the text that is not in italic font must be assumed as being correct. Only the highlighted part is subject to discussion/debate.
- Each statement (a, b, ...) must be considered independently from the others (each statement is about a distinct product).
- If a statement contains several claims (for example: *"technique X is faster and has no drawback"*), each claim must be assessed. A single incorrect claim renders the whole statement incorrect.

Statements:

(a) Our platform does not need to run several applications concurrently. Besides, each one of our applications completely fits in main memory. The CPU that we use has a flexible MMU: it can be configured to use either paging (with details quite similar to the paging design of Intel processors) or segmentation (supporting 4 segments). Currently, our operating system configures the MMU to work in paging mode. We have noticed that the performance of our applications is negatively impacted by a large number of TLB misses. *We believe that modifying the operating system to use segmentation instead of paging will allow us to mitigate this performance problem without introducing any major drawback.*

(b) Our applications are written in Java, and the Java virtual machine that we use relies on a tracing GC (garbage collector). More precisely, our GC has the following characteristics: it is a "moving" (also known as "compacting") and "concurrent" GC.
As a consequence, we can provide the following guarantees to application developers:

(1) applications will never suffer from "use-after-free" bugs;

(2) an application will never run out of heap memory due to internal fragmentation;

(3) application performance will never be strongly impacted by the GC overhead.

(c) We have built a new (persistent) storage device. The distinguishing characteristic of this device is that its access time is much smaller (i.e. faster) than for a traditional storage device. For example, reading (or writing) a data block (of 4096 bytes) only takes 100 nanoseconds. All the other characteristics of the device are similar to a traditional SSD (solid state disk). We have a system with a standard Linux operating system and an application that frequently needs to access (read and write) data in disk files, and we would like to leverage this opportunity to improve the application's performance as much as possible.
We believe that achieving this goal is actually quite easy. We will simply write a new device driver to provide compatibility with this new device. It is not necessary to modify/extend the rest of the Linux kernel code (Input/Output management operations, process/thread scheduling, etc.).

Part II (remember to use a separate answer sheet for this part)

Note that the signature of the main synchronization functions is provided in appendix.

1 About synchronization (4 points)

1.1 For each of the following statements, answer **True** or **False** and explain in a few words. Answers without a correct explanation will not give points.

(a) Multi-thread programming is useful even on machines with a single-core processor.

(b) The safety property of mutual exclusion cannot be totally guaranteed because of a problem called spurious wake-ups.

(c) The Futex system call allows implementing sleeping locks without using any *read-modify-write* atomic instructions.

1.2 Figure 1 presents an implementation of a `increment()` function that allows incrementing the counter passed as parameter. It uses the `CompareAndSwap()` atomic operation.

```
1   void increment(int *counter){
2     compare_and_swap(counter, *counter, (*counter)+1);
3   }
```

Figure 1: Counter with CAS

When testing the code multiple times, I got different results. Each time I tested using 3 threads calling one time the `increment()` function on the same counter. In each case the initial value of the counter was 0.

For each of the following result, describe a scenario that could lead to this result:

(a) The final value of the counter is 3

(b) The final value of the counter is 1

1.3 The algorithm described in Figure 2 implements a barrier:

- The function `init_barrier()` is called by a single thread and initializes the barrier with the number of threads (N) that should arrive at the barrier before any thread can continue its execution.
- The function `barrier()` blocks the calling threads until N threads have called `barrier()`.

This algorithm either:

(a) Works correctly and the barrier is reusable (By reusable, we mean that the same barrier can be used to block N threads multiple times. Said differently, after N threads have reached the barrier, they are unblocked, and if more threads call `barrier()` they are blocked until N calls have been made again).

(b) Works correctly only if the barrier is used once (That is, the barrier is not reusable).

(c) Never works correctly

Which of the previous statement is correct? (Justify your answer)

```
1   mutex_t mutex = MUTEX_INITIALIZER;
2   cond_t cond = COND_INITIALIZER;
3   int count;
4   int nb_to_wait;
5
6   /* called by a single thread */
7   void init_barrier(int n) {
8       nb_to_wait=n;
9       count = n;
10  }
11
12  void barrier(void) {
13      mutex_lock(&mutex);
14      count--;
15
16      while(count > 0){
17        cond_wait(&cond, &mutex);
18      }
19
20      if(count == 0){
21        cond_broadcast(&cond);
22        count = nb_to_wait;
23      }
24
25      mutex_unlock(&mutex);
26  }
```

Figure 2: A barrier

2 Multi-thread programming (4 points)

Recent processors include mechanisms to control the percentage of memory bandwidth that threads can use on the processor. For instance, this mechanism is called MBA (Memory Bandwidth Allocation) in Intel processors. This exercise is inspired from the existence of such a feature, even if MBA does not work exactly as described here.

For all questions below, the following instructions apply:

- You are allowed to introduce any global variable you want in your solutions, and an `init()` function if need be.
- Describe your code in C. Strict correctness of the syntax in C will not be evaluated but your notations should be clear enough. No need to comment the code.

2.1 In this question, we will design a solution to control the memory bandwidth used by different threads running on a processor. With this solution, threads will be able to reserve a percentage of the total memory bandwidth of the processor. If not enough bandwidth is available to satisfy the request of one thread, it is blocked until enough bandwidth has been released.

More specifically, threads are provided with the following functions:

- `void reserve_mba(unsigned int B)`: Called by a thread to reserve B percent of the total memory bandwidth. The calling thread is blocked if not enough bandwidth is available.
- `void release_mba(unsigned int B)`: Called by a thread to release B percent of the total memory bandwidth. The calling thread is never blocked.

Implement the specified function using **only** locks and condition variables as synchronization mechanisms.

Note 1: In function `reserve_mba()`, you are not asked to implement a First-In-First-Out strategy.

Note 2: In function `release_mba()`, you are not asked to verify that the calling thread had previously reserved at least B percent of the total memory bandwidth. You can assume that it is always the case.

2.2 Now we would like to create a more complete version of the solution to control access to resources, compared to Question 2.1. More specifically, we would like to add the following features:

- Reserving Last-Level Cache capacity: A mechanism similar to MBA also exists for the Last-Level Cache (LLC). This mechanism is called CAT (Cache Allocation Technology). The new version should allow threads to reserve at the same time a percentage of the memory bandwidth and of the LLC.
- Priority to threads requiring a large bandwidth: It was observed that a solution as developed in Question 2.1 often leads to starvation in practice for threads requiring a large bandwidth. Hence, the new version should give priority to these threads.

Here are the functions to implement:

- `void reserve_resources(unsigned int B, unsigned int C)`: Called by a thread to reserve B percent of the total memory bandwidth and C percent of the LLC capacity.
- `void release_mba(unsigned int B)`: Called by a thread to release B percent of the total memory bandwidth. The calling thread is never blocked.

- `void release_cat(unsigned int C)`: Called by a thread to release C percent of the total LLC capacity. The calling thread is never blocked.

Also, the following behavior with respect to the priority given to threads requesting a large memory bandwidth should be implemented:

- Requesting a large bandwidth means requesting more than 20 percent of the total bandwidth, no matter the percentage of the LLC capacity requested by the thread.
- As long as a thread requesting a large bandwidth has not been served, threads with *normal* requests (that is, threads requesting less than 20 percent of the total bandwidth) cannot be served.

Implement the specified functions and behavior using **any** synchronization mechanism you want.

Note 1: As in the previous question, you are not asked to implement a FIFO policy for threads in the `reserve_resources()` function, but you should take into account the priority constraint defined above. Also, you are not asked to verify that the parameter passed to `release_*()` functions is correct.

Note 2: Your solution should ensure that threads are not needlessly blocked. If a thread A is blocked because there is not enough resources for it to run, it should not block other threads that requested less resources from running if possible (Still, your solution should always comply with the priority constraint).

3 Other topics (2 points)

3.1 The Multi-Level Feedback Queue (MLFQ) algorithm for scheduling processes is based on the following 5 rules.

Rule 1: If priority(A) > priority(B), then A runs (B doesn't)

Rule 2: If priority(A) == priority(B), Round Robin is applied

Rule 3: When a job enters the system, it is placed at the highest priority (the topmost queue)

Rule 4: Once a job uses up its time slice at a given level (regardless of how many times it has given up the CPU), its priority is reduced (i.e., it moves down one queue).

Rule 5: After some time period S, move all the jobs in the system to the topmost queue.

MLFQ has been designed to favor processes for which I/O operations limit the performance. For each of the following questions, a detailed explanation is expected.

(a) Explain how the MLFQ algorithm favors such processes.

(b) Explain why it is important to favor I/O bound processes for such a scheduling algorithm.

3.2 On a HDD, it can be observed that tracks close to the center of a platter are smaller than tracks on the outside. It implies that they include fewer sectors than outermost tracks.

We have seen in class that file systems should take into account the geometry of disks to achieve high performance.

Propose two strategies that could be used in the design of a file system to take into account this characteristic of the geometry of HDDs. For each strategy: i) Describe it in a few words; ii) Justify briefly why it could be beneficial.

Appendix

Please find below a list of the synchronization primitives available:

Mutexes

- `mutex`: variable of type `pthread_mutex_t`
- `pthread_mutex_init(&mutex, ...)`: initialize the mutex
  - The macro `PTHREAD_MUTEX_INITIALIZER` can be used to initialize a mutex allocated statically with the default options
- `pthread_mutex_lock(&mutex)`
- `pthread_mutex_unlock(&mutex)`

Condition Variables

- `cond`: variable of type `pthread_cond_t`
- `pthread_cond_init(&cond, ...)`: initialize the condition
  - The macro `PTHREAD_COND_INITIALIZER` can be used to initialize a condition variable allocated statically with the default options
- `pthread_cond_wait(&cond, &mutex)`
- `pthread_cond_signal(&cond)`
- `pthread_cond_broadcast(&cond)`

Semaphores

- `sem`: variable of type `sem_t`.
- `sem_init(&sem, int pshared, unsigned int value)`: initialize the semaphore to `value`. Set `pshared` to 0 for a semaphore to be shared between threads.
- `sem_wait(&sem)`
- `sem_post(&sem)`

Atomic operations

- `test_and_set(type *ptr)`
- `fetch_and_add(type *ptr, type val)`
- `compare_and_swap(type *ptr, type oldval, type newval)`: returns `true` on success.