Univ. Grenoble Alpes (Ensimag & UFR IM2AG)
M1 MoSIG
Academic year 2024-2025

Principles of Operating Systems — Final Exam
December 2024

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 (2 points)

1.1 Today, a traditional hardware/software computer system (for example Linux running on a machine with Intel x86 CPUs) is typically based on the following features:

Feature A: two execution modes on the CPU: *user mode* and *supervisor mode* (and applications are forbidden to access the memory address range that belongs to the kernel);

Feature B: support for virtual memory (i.e., there is a distinct virtual address space for each process);

Feature C: support for hardware interrupts (for example: timer interrupts, disk interrupts).

(a) Consider a design in which *feature B* is not available. Give **3** drawbacks compared to a traditional system? (For each drawback, explain briefly)

(b) Consider a design in which *feature C* is not available. Give **2** drawbacks compared to a traditional system? (For each drawback, explain briefly)

1.2 In this question, we consider a machine with a single CPU. We consider a scenario in which the CPU is time-shared between two threads T_A and T_B. Each thread performs a distinct/independent computation. We compare two setups: in the first one, T_A and T_B belong to the same process, while in the second one T_A and T_B belong to distinct processes. There are no other differences between the two setups (in particular, the threads perform exactly the same work in both cases).
We observe that the total execution time of T_A and T_B is shorter in the first setup.

Give 2 reasons that may explain this phenomenon and justify.

Problem 2 (1.5 points)

In this exercise, we consider a simple machine with a MMU that implements virtual memory based on segmentation. The main specifications of this machine are the following:

- The MMU hardware has 2 pairs of (base, bounds/limit) registers (i.e., a process can at most have 2 segments).
- Virtual addresses (including the explicit segment ID) are stored on 14 bits and physical addresses are stored on 16 bits.
- For simplification, we ignore here the management of segment (read/write, user/supervisor) permissions.

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

Segment	**Base**	**Bound**
0	0x2410	0x250
1	0x1230	0x400

2.1 For this process, what is the virtual address that allows accessing the data stored at physical address `0x1500`?

- Provide your answer in hexadecimal notations and explain
- If you think this process cannot access this physical address, explain why

2.2 Another process P_2 tries to access virtual address `0x1400`. Do you have enough information to conclude whether process P_2 can access this virtual address? (explain in a few words)

Problem 3 (4.5 points)

Note: In this problem, all the exercises can be solved independently.

3.1 In this exercise, we consider an operating system similar to Linux designed for 64-bit x86 processors. On the current version of these processors, the virtual addresses and the physical addresses are both stored on 64 bits, but some of these bits are actually ignored:

- For virtual addresses, only the low 48 bits (i.e., bits 0 to 47) are meaningful.
- For physical addresses, only the low 52 bits (i.e., bits 0 to 51) are meaningful.

The default page size is 4 kB. For this page size, the structure is described as follows:

- It is composed of 4 different types of structures: PML4 (Page Map Level 4), PDPT (Page Directory Pointer Table), PD (Page Directory), PT (Page Table).
- CR3 corresponds to the privileged CPU register that points to the root of the currently active paging structure. This root is a PML4.
- Each instance of each structure is stored in a 4 kB page, whose first address is a multiple of 4096.

(a) In a valid entry of a structure, does the page number of the next level correspond to a virtual page number or a physical page number? For example, does a valid PDPT entry store the *virtual page number* or the *physical page number* of the page that stores a PD? Briefly justify your answer.

(b) We consider the following scenario: the processor is executing the (user-level) code of an application. The next instruction to be executed consists in loading a 64-bit value from memory into a CPU register.

- The instruction has already been fetched and decoded by the processor.
- Reading a 64-bit value from main memory takes 100 ns.
- The costs of a TLB lookup/insertion are negligible. Similarly, the costs of a (L1/L2/L3) CPU cache lookup/insertion are negligible.
- The target virtual address is valid and authorized for the application.
- The target page is currently stored in main memory (no need to access the swap space).

In the worst case, how long will it take for the machine to complete the execution of this instruction ? Briefly justify your answer.

(c) Considering the same assumptions as before, how long will it take for the machine to complete the execution of this instruction in the best case? Briefly justify your answer.

3.2

Note: *In this exercise, we consider the same paging structure as in Exercise 3.1, with the same sizes as in Exercise 3.1 for virtual and physical addresses.*

We study the implementation of page replacement policies, that is, the policy used to select the pages that should be removed from physical memory to make space for new pages, when running out of physical memory space.

(a) We have seen in class that policies used in practice try to approximate the *Least-Recently-Used* policy. Explain what are the main reasons that make *Least-Recently-Used* a good replacement policy in this context. (Reasons could be related to the way the applications, the operating system, or the hardware are working).

(b) The solutions studied in class use a single bit per page and the *clock* algorithm to implement a *pseudo-Least-Recently-Used* policy for page replacement. A group of students decides to implement its own policy, with the main goal of identifying more accurately the *Least-Recently-Used* pages.

To test their idea, they get access to an experimental processor with some advanced features. Specifically, in this processor, the MMU can be configured to save a timestamp each time a page is accessed.

The proposed solution works as follows:

- To better identify the least recently used page, a timestamp is saved in each Page Table Entry, and updated each time the corresponding page is touched, thanks to the new feature provided by the experimental processor.
- To select a page for eviction, the algorithm first selects K *present* pages randomly (Discussing how to implement this random selection is out of scope of this question). Then, among those K pages, the page to evict is chosen using the following criteria:
    1. Pages that are not marked as *dirty* are considered first
    2. The *non-dirty* page with the oldest timestamp is evicted.
    3. If no *non-dirty* page exists among the K selected pages, the *dirty* page with the oldest timestamp is evicted.

After implementing and testing their new solution, they make the following observations:

(A) It is possible to store a timestamp over 12 bits without allocating any additional data structure compared to the original solution.

(B) The proposed solution provides significant improvements compared to an alternative design that would work in the exact same way but without taking into account whether pages are dirty.

(C) In case where the physical memory of the system is only partially used (that is, there is no need for page eviction), applications run slower with the proposed solution compared to the algorithm studied in class.

For each of these observations independently, explain the reasons behind the observation.

3.3 In this exercise, we consider a MMU (memory management unit) design inspired from the one used in ARM 32-bit processors (ARMv7A architecture), with some simplifications.

The principle of this MMU has some similarities with the one of Intel processors, but there are also some differences, especially regarding the size of the elements that compose a paging structure. The main characteristics are summarized as follows:

- Virtual addresses are stored on 32 bits.
- Physical addresses are stored on 32 bits.
- The hierarchical paging structure has two levels: L1 and L2.
- An L1 structure has a size of 16 kB (2^{14} bytes) and each entry has a size of 4 bytes. The starting address of an L1 structure must be aligned on 16 kB (i.e., must be a multiple of 2^{14}).
- Each L2 structure has a size of 1 kB (2^{10} bytes) and each entry has a size of 4 bytes. The starting (physical) address of an L2 structure must be aligned on 1 kB (i.e., must be a multiple of 2^{10}).

(a) Given a virtual address `0x12345678`, describe how one should proceed to go through the paging structure and identify the corresponding physical address.

(b) List 2 advantages and 2 drawbacks of this design compared to the 2-level paging structure of Intel x86 32-bit processors (also a 2-level paging structure but where both the L1 and L2 structures have a size of 4kB)?

For each point, justify briefly.

Problem 4 (2 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 CPU design is quite different from the one of Intel processors regarding the management of memory operations. Indeed, every TLB miss is managed by a software handler (provided by the operating system). Similarly, every cache miss from the L1 cache is also managed by a software handler. *Compared to the Intel design, our design requires more kernel code but is more efficient because the OS can choose the most efficient cache replacement policy for each cache and each process.*

(b) Our default memory allocator uses a linked list of free blocks with a "next-fit" policy and an immediate block coalescing strategy. We have noticed that some of our main applications suffer from severe external fragmentation regarding the management of their memory heap. In addition, some of these applications also suffer from memory leaks. For the moment, we do not have the time to investigate these issues in details, at the level of each application. *Nonetheless, we believe that we have found a generic and fairly effective way to reduce memory waste. We will replace the above allocator with another one, which will always round up the requested block size to next power of two.*

(c) Our platform hosts multiple applications running in parallel. However, one of these applications has a specific role and runs on a single, dedicated CPU core (that is, no other application run on this core). The application is concurrent and based on a single process with multiple threads. It frequently creates and destroys threads. On the other hand, it uses very little amount of memory, and does not interact much with I/O devices. We will use a user-level (also known as "N:1") implementation for the threads of this application (instead of a kernel-level implementation). *This choice will improve the performance of the application and should not have any drawback.*

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 file systems and I/O devices (2 points)

1.1 The data block associated with a directory contains an `inumber` for each file included in this directory. Explain how one can access the content of a file using its `inumber`.

1.2 One of the challenges associated with the design of file systems is the fact that a file system typically stores many small files and also some big files. Describe one design principle used in some file systems that takes into account this constraint and explain.

1.3 We have seen during the lectures that a system crash (for instance, due to a power outage) during an operation on the file system can leave the file system in an inconsistent state. This is due to the fact that one operation might require several writes. For instance, if one considers the operation of appending user data as one data block to an existing file, it requires writing at least the data bitmap, the inode of the file, and the data block.

For each of the following crash scenarios, tell: (i) if the file system is in an inconsistent state and (ii) if user data is lost (briefly justify your answer):

(a) The data block and the inode have been written but not the bitmap.

(b) The data block and the bitmap have been written but not the inode.

(c) The inode and the bitmap have been written but not the data block.

1.4 One of the solutions to be able to recover the file system in a consistent state after a failure is to use journaling. Before applying a modification operation, this operation is written into a journal that is also stored on the Hard-Disk Drive (HDD).

A technique that can be used to improve the performance of journaling techniques is *delayed checkpointing*. Checkpointing is the name given to the step that correspond to applying a modification to the file system, after it has committed in the journal. With *delayed checkpointing*, the operations are not applied immediately. Instead, we wait until a few operations have been committed to apply all of them.

Discuss the advantages and the drawbacks of this approach for the case of a file system stored on a HDD.

2 Some synchronization problems (3 points)

2.1 Figure 1 presents a solution to the critical section problem for N threads. The presented code is for thread i.

```
1     int busy = 0;
2     int not_turn = 0;
3
4     enterCS() {
5       not_turn = i
6       while (test_and_set(&busy) == 1 && not_turn == i ){;}
7     }
8
9     exitCS() {
10      busy = 0;
11    }
```

Figure 1: Critical section for N threads – code of thread i

(a) Define the three properties associated with the implementation of a critical section.

(b) Does the algorithm presented in Figure 1 ensure the 3 properties of a critical section? Answer YES or NO, and:

- If your answer is YES, discuss the advantages of this solution compared to Peterson's algorithm.
- If your answer is NO, explain the problem and provide an execution scenario.

2.2 I have found on the Web a new thread synchronization library. Contrary to the POSIX interface, in this library the `cond_wait()` function takes a single parameter, which is a pointer to a condition variables.

To allow more efficient executions, this library will allow calling `cond_wait()` without grabbing a mutex first. Apart from this point, condition variables in this library work in the exact same way as in the `pthread` library.

Taking the example of a consume function in a typical producer-consumer algorithm, it will be possible to write it as described in Figure 2.

In your opinion, with this new library, will the consumer function presented in Figure 2 be correct? Answer YES or NO and:

- If your answer is YES, explain why this new implementation can be more efficient than an implementation based on the `pthread` library.
- If your answer is NO, explain the problem and provide an execution scenario.

```
1    int consume(void){
2      int elem;
3
4      while(count == 0){
5        cond_wait(&cond_consumer);
6      }
7
8      mutex_lock(&mutex);
9
10     elem = buf[index];
11     index = (index + 1) % BUFFER_SIZE;
12     count--;
13
14     cond_signal(&cond_producer);
15     mutex_unlock(&mutex);
16
17     return elem;
18   }
```

Figure 2: A consumer function implementation

2.3 The reader-writer problem is a synchronization problem where some reader threads might read shared data and some writer threads might modify this shared data. If a writer wants to modify data, it must first get exclusive access to the data. On the other hand, multiple readers should be able to read at the same time.

Figure 3 describes a solution to the reader-writer problem. Before starting reading the data, a reader has to call the `startRead()` function. It must then call `endRead()` when it finishes. Similarly, a writer has to call `startWrite()`/`endWrite()` when it starts/finishes writing respectively.

In your opinion, does the code provided in Figure 3 solve the reader-writer problem for **multiple readers and multiple writers**.? Answer YES or NO and:

- If your answer is YES, discuss whether the solution would still work if the 2 calls to `cond_broadcast()` are replaced with `cond_signal()`.
- If your answer is NO, explain the problem and provide an execution scenario.

```
1   /* global variables */
2   int readerCount = 0:
3   int writer = 0;
4   mutex_t mutex = MUTEX_INITIALIZER;
5   cond_t cond = COND_INITIALIZER;
6
7   void startRead(void) {
8     mutex_lock(&mutex);
9     while(writer == 1) { cond_wait(&cond, &mutex); }
10    readerCount = readerCount + 1;
11    mutex_unlock(&mutex);
12  }
13
14  void endRead(void) {
15    mutex_lock(&mutex);
16    cond_broadcast(&cond);
17    readerCount = readerCount - 1;
18    mutex_unlock(&mutex);
19  }
20
21  void startWrite(void) {
22    mutex_lock(&mutex);
23    while((readerCount > 0) || (writer == 1)) { cond_wait(&cond, &mutex);}
24    writer = 1;
25    mutex_unlock(&mutex);
26  }
27
28  void endWrite(void) {
29    mutex_lock(&mutex);
30    cond_broadcast(&cond);
31    writer = 0;
32    mutex_unlock(&mutex);
33  }
```

Figure 3: Reader-Writer synchronization

3 Multi-thread programming (5 points)

We would like to implement a library that provides advanced synchronization mechanisms between threads.

The general instructions below apply to all questions:

- You are allowed to introduce any global variable you want, as well as 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.
- Regarding the synchronization primitives that you can use, carefully read the **specific instructions** associated with each question.

3.1 The first synchronization mechanism we consider is a special case of the producer-consumer problem, where consumers only want to consume the *last* value produced. Such a mechanism, called `monitoring_channel`, can be used to monitor changes in the configuration of a system.

In this exercise:

- We assume that the `monitoring_channel` stores Integers
- As a consequence of the specification of the problem, a single Integer is used to store the most recent value of the `monitoring_channel`: there is no need to maintain an array of values since only the last inserted value is useful.

More specifically, two functions are used to access a `monitoring_channel`, that are specified as follows:

- `void monitoring_channel_put(int val)`: Called by a producer thread to insert a new value in the `monitoring_channel`. The function never blocks the calling thread. If the previous value was not consumed, it is over-written.
- `int monitoring_channel_get(void)`: Called by a consumer thread to get the most recent value from the `monitoring_channel`. A value can be consumed by only one consumer. The function blocks the calling thread if there is no new value to consume.

Specific instructions:

- Implement the functions `monitoring_channel_put()` and `monitoring_channel_get()` using semaphores.
- The solution should work with multiple producers and multiple consumers.

3.2 We want to implement a basic synchronization mechanism between threads called `waitable_event`.

A `waitable_event` is a mechanism to which is associated an integer variable:

- It allows threads to wait until the event variable is set to a specific positive value.
- The event can be set to a positive value only if it was previously reset.

The following functions are associated with a `waitable_event`:

- `void waitable_event_set(int val)`: Set the value to `val` only if the `waitable_event` was previously reset. Otherwise, does nothing. Calling threads are never blocked
- `void waitable_event_reset(void)`: Reset the value of the `waitable_event`. This function always succeeds and never blocks the calling thread.
- `void waitable_event_wait(int val)`: Blocks the calling thread until the value of the `waitable_event` is `val`.

Note that if the function `waitable_event_reset()` is called shortly after a call to the function `waitable_event_set()` and some waiting threads have not been able to exit the function `waitable_event_wait()`, they might be blocked again.

Specific instructions:

- Implement the solution using only atomic operations as synchronization mechanism.
- Your solution should work with any number of threads.
- Your solution can induce busy waiting.
- To design your solution, you can take advantage of the fact that threads can only wait for positive values.

3.3 We want to implement one more synchronization mechanism between threads called `controlled_barrier`.

Contrary to the traditional barrier mechanism, the criteria to unblock threads from a `controlled_barrier` is not that `N` threads have reached the barrier. Instead, threads blocked at the barrier are unblocked if another thread calls `controlled_barrier_unblock()`.

The `controlled_barrier` mechanism provides the two following functions:

- `void controlled_barrier_wait(void)`: Blocks the calling thread until the **next** call to `controlled_barrier_unblock()`.
- `void controlled_barrier_unblock(void)`: Unblocks all the threads that have called `controlled_barrier_wait()` before the call to `controlled_barrier_unblock()`.

Specific instructions:

- Implement the solution using mutexes and condition variables.
- The solution should work for any number of threads
- The specification of the problem implies that a `controlled_barrier` can be used multiple times

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.