Univ. Grenoble Alpes (Ensimag & UFR IM2AG)
M1 MoSIG
Academic year 2023-2024

Principles of Operating Systems — Final Exam December 2023

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 We have seen in the lectures that the code of an operating system (OS) provides mechanisms and policies.

(a) Cite 3 different examples of policies in different parts of an operating system. Please note the following guidelines:

- No need to give a detailed description. Simply explain the aspect to which the policy relates (for instance: heap memory management) and the question addressed by the policy (for instance: the placement policy within a memory allocator determines which available memory block should be selected to fulfill a new allocation request). No need to provide concrete examples of policies (for instance: first fit).
- Each example must correspond to a different aspect.

(b) Most operating systems are designed and implemented in such a way that the code corresponding to mechanisms is separated from the code of policies. Briefly describe the reasons that motivate this approach.

1.2 In this exercise, we consider a machine with 4 CPU cores. There are 5 concurrent processes (named P_1, ..., P_5). Each process encapsulates 3 concurrent threads.
We successively consider 4 different models (similar to the ones studied in the lectures) for the threads:

- **CU :** Cooperative, User-level threads
- **CK :** Cooperative, Kernel-level threads

- **PU :** Preemptive, User-level threads
- **PK :** Preemptive, Kernel-level threads

Note that, for each setup, we assume that all the threads of all the processes are based on the same model.
Process P_5 is launched using the `time` command, which reports the 3 following time measurements after the termination of the process:

- *real*: physical time elapsed between the creation and the termination of the process.
- *usr*: total active CPU time spent in user mode by all the threads of the process.
- *sys*: total active CPU time spent in supervisor mode on behalf all the threads of the process.

(a) Among the 4 threading models, which one(s) support the parallel execution of threads from different processes? (For example, having a thread from P_1 that executes in parallel with a thread from P_2.) No need to justify your answer.

(b) Among the 4 threading models, which one(s) support the parallel execution of several threads from the same processes? No need to justify your answer.

(c) Copy the following table to your answer sheet, and, for each threading model, fill in each slot with "Yes" or "No". You are not required to justify your answers.

	CU threads	CK threads	PU threads	PK threads
It is possible to obtain (for P_5): usr + sys $\leq$ real				
It is possible to obtain (for P_5): usr + sys $>$ real				

1.3 Bob has written a simple file server application for Linux that works as follows:

- The application awaits file requests sent over the network by client processes running on other machines.
- Whenever the application receives a new request, it parses it to extract the name of the file to retrieve, opens a file descriptor corresponding to this file, reads the contents of the file and then writes them into the "network socket"[1] of the corresponding client.

[1] A *network socket* is a special kind of file descriptor, somewhat similar to a Unix pipe but enabling bidirectional communication with a remote client process.

Alice, a friend of Bob, advises him to take advantage of the following Linux system call:
`ssize_t sendfile(int out_fd, int in_fd, off_t offset, size_t count);`
This system call copies `count` bytes (skipping the first `offset` bytes if necessary) from file descriptor `in_fd` to file descriptor `out_fd`. Explain why using this system call may indeed improve the performance of Bob's server application.

Problem 2 (4 points)

2.1 In this exercise, we consider a Linux-like operating system running on a machine with an Intel x86 32-bit CPU. The OS supports virtual memory based on paging and disk swapping.

(a) In this question, we consider the following situation: a process triggers a page fault because an instruction (of the application code) needs to access a data structure (allocated inside a page of the heap) that has been swapped out to the disk. Describe the main steps that the OS kernel must perform in order to "fix" this issue and resume the execution of the process. Note:

- In your answer, describe the different data structures that must be queried and/or modified by the kernel (and the purpose of each of these structures).
- For simplification, you can assume that the kernel has a pool of free page frames when the page fault occurs (i.e., handling this page fault does not require the eviction of another page).
- This question is focused on the mechanisms, not the policies. You are not required to describe the details (algorithms and data structures) used to choose the new page frame where the missing heap page will be brought.

(b) In this question, we now consider the following situation: the operating system has chosen to evict a page (i.e., to swap it out to disk). Assuming that the choice of the "victim" page has already been made, describe the remaining steps that the OS kernel must perform to complete the eviction.

(c) Among the steps described in the previous answer (b), describe which one(s) must be modified and how, if we assume that several coexisting virtual pages (within the same process or in different processes) can be mapped (during overlapping time intervals) to the same physical page.

2.2 In this exercise, we consider the paging design of an imaginary processor (partially inspired from the RISC-V Sv39 architecture). Its main characteristics are summarized as follows:

- Virtual addresses occupy 39 bits (they are actually stored on 64 bits but the top bits are ignored).
- Physical addresses occupy 56 bits.
- The paging structure is hierarchical (like in the examples studied in the lectures). It can have up to 3 levels (named L1, L2, L3) and supports multiple page sizes.
- The smallest page size is 4 kB (4096 bytes).

- The paging structure of a process has always exactly one L1 instance.
- A valid L1 entry (L1E) points either to an L2 instance or directly to a physical page.
- A valid L2 entry (L2E) points either to an L3 instance or directly to a physical page.
- A valid L3 entry (L3E) points to a physical page.
- All the L1, L2, and L3 instances have the same size.
- All entries (L1E, L2E, L3E) have the same size (8 bytes each).

(a) What are the different page sizes supported by this design? Briefly justify your answer.

(b) What is the smallest possible size (in bytes) for the paging structure of a process, assuming that all its virtual addresses are configured as valid? Briefly justify your answer.

(c) What is the largest possible size (in bytes) for the paging structure of a process, assuming that all its virtual addresses are configured as valid? Briefly justify your answer.

2.3 In this exercise, we consider a machine with a single CPU core, whose MMU supports paging-based virtual memory. The CPU has the following characteristics:

- Virtual addresses are stored on 32 bits.
- Physical address are stored on 32 bits.
- The MMU supports a single page size of 64 kB (2^{16} bytes).
- The TLB has a capacity of only 4 entries.
- The TLB is fully associative: any translation can be stored in any entry.
- Initially (upon the start of the process), all the entries are empty/invalid.

We assume that the execution of the process triggers the following sequence of (virtual) memory accesses. For each step in the sequence, indicate if it results in a TLB hit or in a TLB miss. Briefly justify your answer (for example, by showing the step-by-step evolution of the TLB contents).

(a) `0x80001000`

(b) `0x5AC00B00`

(c) `0x5AE00200`

(d) `0x5AFFF000`

(e) `0x5AC00B00`

(f) `0x5AF32000`

(g) `0x80001020`

(h) `0x5AC00B20`

(i) `0x80001440`

(j) `0x5AE00900`

2.4 Some recent processors feature an optimization named "TLB coalescing", which allows improving the TLB efficiency even for systems that only use a single, small page size (e.g., 4096 bytes). In essence, the principle is the following:

- When handling a TLB miss regarding a given VPN (virtual page number), the MMU hardware analyses the adjacent entries in the paging structure to check if it is possible to "coalesce" (in other words, to summarize) them in the same TLB entry.
- The format of a TLB entry is modified as follows and the (hardware) algorithm for a TLB lookup is modified accordingly:
  - Old format: *(VPN, Present, PPN, Permissions)*
  - New format: *(Lowest VPN of the coalesced range, number of pages, Present, Lowest PPN, Permissions)*

During the execution of the TLB miss (hardware) algorithm, what are the precise conditions that must be checked regarding the adjacent entries in the paging structure to determine if they can be "coalesced" in the same TLB entry?

Problem 3 (3 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) We have designed a CPU that provides new instructions to save/restore the execution context of a thread (i.e., to save the state of the CPU registers into a data structure stored in memory, or conversely to copy back the fields of an in-memory data structure into the CPU registers) in a very fast way (10 nanoseconds instead of 500 nanoseconds). *Now that this important source of overheads has been removed, we think that we can afford to perform context switches much more frequently. In particular, we should drastically reduce the duration of the time quantum (also known as "time slice") used by the kernel scheduler to configure the periodic timer interrupts. This will enable the OS to make much more frequent, efficient and fair scheduling decisions.*

(b) Many of our applications are written in C and suffer from typical memory management bugs ("double frees", "use after free", "memory leaks", etc.). Besides, our CPU architecture uses paging with 64-bit virtual addresses but, like in the case of Intel 64-bit CPUs, the 16 highest bits of a virtual address are ignored by the hardware MMU (Memory Management Unit). We can modify the code of our memory allocation library (implementing the `malloc` and `free` functions) in order to store information in the "available" bits of an address returned by `malloc` and later (when calling `free`) compare these information with the internal metadata of the memory allocator. *We believe that this simple idea, which does not require any modification of the hardware or the application code, will help us to detect much more easily most of the above-mentioned bugs.*

(c) Our machine is dedicated to running a single application, which is written in Java. The application uses a large volume of dynamically-allocated data structures, which exceeds the capacity of the physical memory. The Java virtual machine uses a tracing and compacting garbage collector (GC). We have observed that a significant part of the disk swapping activity is caused by the GC. Given that we have a significant manpower and budget as well as a good expertise in C programming, *we think that we can significantly improve the performance of this application by rewriting it in C (instead of Java) and we believe that this approach will not introduce any drawback (except possibly some risks of memory leaks).*

(d) When a new process is created, many of its pages (such as the heap and stack pages) are expected to have all their bytes set to zero. Our operating system uses the following strategy to efficiently deal with this requirement: the kernel always keeps a "zero page" (i.e., a physical page full of zeros) that it maps in "copy-on-write" mode to all the virtual pages of a newly created (heap or stack) region. We have noticed that many applications suffer from long startup times, which seem to be caused by a large number of page faults. Besides, our new CPU hardware design provides an instruction that allows resetting all the bytes of a page very quickly. *We think that we can solve the problem of long startup times by changing our strategy: instead of using copy-on-write with the "zero page", we will systematically allocate and map a new set of physical pages whenever a new (heap or stack) region is created, and we will reset the contents of these pages with the new CPU instruction. Furthermore, we believe that this new approach will not introduce 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 Some synchronization problems (3 points)

1.1 Figure 1 presents a solution to the critical section problem for two threads 0 and 1. The presented code is for thread i.

```
int wants[2]= {0, 0};

enterCS() {
  wants[i] = 1;
  while (wants[1-i] == 1){;}
}

exitCS() {
  wants[i] = 0;
}
```

Figure 1: Critical section for two threads 0 and 1 – code of thread i

Does the algorithm presented in Figure 1 ensure the two main properties of a critical section? Answer YES or NO, and provide an explanation for your answer.

1.2 Figure 2 presents a solution to the critical section problem using `futex`.

```
int flag=0;

lock() {
  while (test_and_set(flag)){
    futex(&flag, FUTEX_WAIT, 1);
  }
}

unlock() {
  flag=0;
  futex(&flag, FUTEX_WAKE, 1);
}
```

Figure 2: Critical section using a futex

Does the algorithm presented in Figure 2 ensure the two main properties of a critical section? Answer YES or NO and:

- If your answer is YES, explain what is the main drawback of this solution.

- If your answer is NO, explain which property is not ensured and illustrate it through an execution scenario.

1.3 Figure 3 presents a solution to the barrier problem. We recall that a call to `barrier()` blocks the calling thread until all other threads have also called `barrier()`.

In this exercise, the total number of threads is N. Also, the solution is not meant to implement a reusable barrier: it assumes that each thread will call `barrier()` only once.

```
int nb = 0;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

barrier() {
  pthread_mutex_lock(&mutex);
  nb++;
  while (nb < N){
    pthread_cond_wait(&cond, &mutex);
  }
  pthread_cond_signal(&cond);
  pthread_mutex_unlock(&mutex);
}
```

Figure 3: A barrier

Assuming that each thread will call `barrier()` only once, is the algorithm presented in Figure 3 correct? Answer YES or NO and explain (using an execution scenario if need be).

1.4 Figure 4 presents a solution to the reader-writer problem using semaphores and atomic operations.

Answer the following questions (justify your answer):

(a) Assuming that there are only writer threads, does this solution prevent multiple writers from accessing the protected data at the same time?

(b) Assuming that there are only reader threads, does this solution allow multiple readers to access the protected data at the same time?

(c) Assuming that there are both reader and writer threads, does this solution prevent readers and writers from accessing the protected data at the same time?

```
/* global variables */
int readerCount = 0:
int writer = 0;
sem_t S1;
sem_t S2;

void init(void) {
  sem_init(&S1, 0, 1);
  sem_init(&S2, 0, 0);
}

void startRead(void) {
  int c = fetch_and_add(&readerCount, 1);
  if(c == 0){
    sem_wait(&S1);
  }
  else{
    sem_wait(&S2);
  }
  sem_post(&S2)
}

void endRead(void) {
  int c = fetch_and_add(&readerCount, -1);
  if(c == 1){
    sem_post(&S1);
  }
}

void startWrite(void) {
  sem_wait(&S1);
}

void endWrite(void) {
  sem_post(&S1);
}
```

Figure 4: Reader-writer synchronization

2 A multi-threaded application (4 points)

A datacenter is an infrastructure that includes a large number of servers. It allows clients to reserve servers to execute their applications. A resource manager needs to be used to assign servers to different clients. In this exercise, we are going to look at the implementation of this resource manager.

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.

2.1 To run an application in the datacenter, a client needs to reserve some servers first. To this end, each client connects to the resource manager. Each connected client is represented by an independent thread (called *client thread* hereafter) in the resource manager.

For the reservation of machines, the functions `reserve_servers()` and `release_servers()` are used. The specification of these two functions is as follows:

- `void reserve_servers(int nb)`: Called by a client thread to reserve `nb` servers. The function blocks the calling thread if the number of free servers is less than `nb`.
- `void release_servers(int nb)`: Called by a client thread to release `nb` servers it had previously reserved.

These two functions only manage the number of servers. **They do not deal with the question of which server is assigned to which user**. We assume that this problem would be handled by other functions, not discussed in this exercise.

Specific instructions:

- Implement the functions `reserve_servers()` and `release_servers()` using mutexes and condition variables.
- We assume that the datacenter includes N servers.
- Each server is reserved exclusively: a server can only be part of one reservation at a time.
- We will assume that the functions are always used correctly. In particular, we assume that a client does not try reserving a negative number of servers. We also assume that a client always releases the number of servers it had previously reserved. Said differently: you do not have to deal with error cases.
- The resource manager does not have to be fair. A request can be served before the request from another client even if it arrived after.

2.2 In this step, the management of the resources evolves compared to Question 2.1. Clients have the opportunity to sign a *premium* agreement with the datacenter. It means that these clients will have a better quality of service compared to normal clients.

Premium clients make their request through a dedicated function called `reserve_premium()`. The function has the same basic specification as the `reserve_servers()` function:

- `void reserve_premium(int nb)`: Called by a *premium* client's thread to reserve `nb` servers. The function blocks the calling thread if the number of free servers is less than `nb`.

However, the introduction of this new kind of clients changes the expected behavior of the resource manager. The new expected behavior is as follows:

- if requests from premium clients are pending, requests from normal clients cannot be served.

Propose an implementation of the functions `reserve_premium()`, `reserve_servers()`, and `release_servers()`, for this new version of the system where there are *premium* and *normal* clients.

Specific instructions: The instructions remain the same as for the previous exercise. In particular:

- You should implement the functions using mutexes and condition variables.
- The resource manager does not have to be fair for clients inside each category. A request from a premium client can be served before the request from another premium client even if it arrived after. The same rule applies between requests from normal clients.

3 About Scheduling (3 points)

3.1 Scheduling processes

(a) When evaluating a scheduling policy for processes, two metrics are often considered: the *throughput* (defined as the number of processes that complete per time unit) and the *turnaround time* (defined as the time for each process to complete).

Explain why *turnaround time* if often considered as a more important metric (you are encouraged to present execution scenarios to demonstrate your point).

(b) We consider a simple *round-robin* scheduling strategy and the case of *latency-sensitive* processes. A latency-sensitive process can be defined as follows: it is a job with short CPU bursts but for which the time between the moment it is ready, and the moment it is scheduled should be short.

Experts consider that *round-robin* is not a good strategy for handling such processes. Explain why.

(c) We consider an Operating System where processes are classified into different categories based on their needs with respect to scheduling (according to information provided by the user). Processes in the `SCHED_IDLE` category are processes that can wait for a long time in the ready list and can be interrupted before the end of their time slice. Processes with very low priority are put in the `SCHED_IDLE` category.

In the context of a multicore processor, we assume that a *round-robin* scheduling policy is applied on each core, and that affinity scheduling is used (a process always stays on the same core by default). The following strategy is proposed to better handle *latency-sensitive* processes.

- Based on the input of the user, *latency-sensitive* processes are put in a `SCHED_LATENCY` category

- When a `SCHED_LATENCY` process becomes ready, instead of keeping it on the same core, we check whether a core is currently executing a `SCHED_IDLE` process, and if yes, we interrupt the `SCHED_IDLE` process to execute the `SCHED_LATENCY` process on that core.

List some advantages and drawbacks of this new strategy (you should list 3 points in total). For each advantage or drawback, explain briefly.

3.2 I/O scheduling

(a) The NOOP scheduler is a very simple I/O scheduling strategy. All I/O requests are put in a FIFO queue and are executed in order. When using a Solid-State Drive (SSD), experiments show that the NOOP scheduler can provide much better performance than more complex schedulers that try to re-order requests (such as the ones studied in class). Explain why (a detailed explanation is expected).

(b) We studied *elevator* I/O scheduling strategies in class (SCAN and CSCAN). When considering Hard-Disk Drives (HDDs), one technique to try improving elevator scheduling is called *anticipation*. The idea is as follows. After serving a request of process `A`, instead of moving immediately to the next request as specified by the elevator scheduling algorithm, the scheduler will pause for a short period of time to check whether process `A` will make another I/O request soon (it *anticipates* a new request from process `A`). Hence, this new I/O request by process `A` will be considered when deciding which request should be served next.

Based on your understanding of HDDs and your understanding of how file systems are designed, explain why such an *anticipation* technique can help improving performance.

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.