Exam Questions
Operating Systems

Fall 2025

Vincent Van Schependom

Short questions

Question 1: Name a communication method that can be used between threads of the same process but cannot be used for communication between parent and child process. Explain briefly your answer. **(4 points)**

ANSWER

Different threads inside a process share the (physical) address space of that process. They can thus share variables in memory and communicate via those variables. However, since each process has its own physical address space, threads of *different* processes do *not* share memory by default. This also applies when we call `fork()`: the child process is an exact copy of the parent, but with a different physical address space, even though the virtual address space is the same.

Question 2: What is the main advantage of preemptive scheduling compared to nonpreemptive scheduling? Explain briefly your answer. **(4 points)**

ANSWER

Preemption allows the operating system to interrupt a process at any point, blocking it and allowing another process to run. This means that the operating system can prevent a process from running for too long, which can prevent a process from monopolizing the CPU.

Question 3: What will the code in Figure 1 print? Explain briefly your answer. You can assume that all system call invocations are successful. **(4 points)**

ANSWER

Only the child process wil run the `if`-statement, since `fork()` returns 0 in the child process and a positive value in the parent process. As argued before, child and parent have different physical address spaces, so the child will increment its *own* copy of `counter` and exit, after which it `exit()`s. The parent process waits for any child process to exit (`waitpid(-1,...)`), after which it will print 1, the initial value of `counter`, since the child modified a different copy of `counter`.

Question 4: What will the code in Figure 2 most likely print? Explain briefly your answer. You can assume that all system call invocations are successful. **(4 points)**

ANSWER

The code will most likely print a value less than 1 000 000. Each of the 1000 threads will try to increment the shared (process-wide) `A` variable 1000 times. However, `A++` is *not* an atomic operation: it first loads the value of `A` into a register, increments it, and then stores it back into `A`. There is thus a race condition between the threads: a thread can get blocked after loading the value of `A` into a register. While it is blocked, other threads may read, increment, and update `A`. When the blocked thread resumes, it increments its old value of `A` – which was stored in the thread-local register – and

```c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

int counter = 1;

void child(void) {
    counter++;
    exit(0);
}

int main(void)
{
  if (fork() == 0) {
    child();
  }
  waitpid(-1, NULL, 0);

  printf("%d\n", counter);
  return 0;
}
```

Figure 1

```c
#include <stdio.h>
#include <pthread.h>

int A = 0;
pthread_t thread_id[1000];

void* count(void *input) {
  int i;
  for (i=0;i<1000;i++)
    A++;
  pthread_exit(NULL);
}

int main(void) {
  int i;
  for (i=0;i<1000;i++)
    pthread_create(&thread_id[i], NULL, count, NULL);
  for (i=0;i<1000;i++)
    pthread_join(thread_id[i], NULL);
  printf("%d\n", A);
  return 0;
}
```

Figure 2

writes it back to `A`, effectively overwriting the progress made by the other threads. This results in "lost updates", causing the final value of `A` to be lower than the expected 1 000 000.

```c
#include <pthread.h>
#include <unistd.h>
#include <stdio.h>

int accounts[2];
pthread_mutex_t lock[2];
pthread_t tid1, tid2;

struct Transact {
  int from;
  int to;
  int amount;
};

struct Transact t1 = {.from = 0, .to = 1, .amount = 10};
struct Transact t2 = {.from = 1, .to = 0, .amount = 20};

void *transfer(void *in) {
  struct Transact trs = *(struct Transact*)in;

  pthread_mutex_lock(&lock[trs.from]);
  sleep(1);
  pthread_mutex_lock(&lock[trs.to]);

  accounts[trs.from]-=trs.amount;
  accounts[trs.to]+=trs.amount;

  pthread_mutex_unlock(&lock[trs.to]);
  pthread_mutex_unlock(&lock[trs.from]);
  pthread_exit(0);
}

int main(void) {
  pthread_mutex_init(&lock[0], NULL);
  pthread_mutex_init(&lock[1], NULL);
  accounts[0] = 100;
  accounts[1] = 100;

  pthread_create(&tid1, NULL, transfer, (void*)&t1);
  pthread_create(&tid2, NULL, transfer, (void*)&t2);
  pthread_join(tid1, NULL);
  pthread_join(tid2, NULL);
  printf("%d\n", accounts[0]);
}
```

Figure 3

Question 5: What will the code in Figure 3 most likely print? Explain briefly your answer. You can assume that all system call invocations are successful. **(4 points)**

ANSWER

The code will in fact not print anything, since there is a *deadlock*. Thread 1 acquires the lock on Account 0 and at the same time Thread 2 acquires the lock on Account 1. Thread 1 then tries to acquire the lock on Account 1, but it is blocked since it is already held by Thread 2. Thread 2 then tries to acquire the lock on Account 0, but it is blocked since it is already held by Thread 1. Both threads are now blocked and will never be unblocked, resulting in a deadlock.

We can fix this by ensuring that the threads always acquire the locks in the same global order, e.g. by checking `trs.from < trs.to`. By forcing all threads to acquire locks in a specific global order (e.g., always lock the lower ID first, then the higher ID), we mathematically guarantee that a cycle cannot exist: If Thread A locks Account 1, Thread B cannot lock Account 2 while waiting for Account 1 to become unlocked, because the rules would have required Thread B to lock Account 1 before trying to lock Account 2.

Question 6: 4 You are developing an application that is composed of multiple collaborative processes. You wish to implement the following functionality: if a resource is currently unavailable, the process

should go to sleep until it receives a wakeup signal from another the process. Which method you would use to avoid a race condition? Explain briefly your answer.

ANSWER

A semaphore is the perfect fit for this kind of resource management problem. Specifically, a counting semaphore can track resource availability: `sem_wait` automatically puts the process to sleep if the count is zero, and `sem_post` wakes it up if the resource becomes available (count goes from 0 to 1).

Crucially, because these are separate processes (not threads), the semaphore must be allocated in shared memory (e.g. using `mmap`) so all processes access the same synchronization variable.

Question 7: 4 What is the key advantage of monolithic operating systems compared to microkernel operating systems? Explain briefly your answer.

ANSWER

In monolythic systems, the whole operating system is run in kernel mode, which is faster and more efficient.

In microkernel systems, only the core fundamental services run in kernel mode, while other services run in user mode. They have to use system calls (IPC) to communicate with the kernel, which adds overhead and can slow down performance.

Question 8: 4 Which POSIX system call should you use to send the `SIGUSR1` signal to a child process? Explain briefly your answer.

ANSWER

The `kill()` system call can be used to send a signal to a process. It takes two arguments: the process ID of the process to send the signal to, and the signal to send. In this case, we want to send the `SIGUSR1` signal to a child process, so we can use `kill(child_pid, SIGUSR1)`.

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

int main(void){
  fork();
  fork();
  printf("a\n");
  exit(0);
}
```

Figure 4

Question 9: 4 How many times will the letter 'a' be printed if we execute the code in Figure 4? Explain briefly your answer. You can assume that all system call invocations are successful.

ANSWER

The letter ‘a’ will be printed 4 times.

The parent process will create a new child on the first `fork()`. Both processes (parent and child) will execute all code below the first `fork()`. This means that both parent and child will create a new child on the second `fork()`. In total, we now have 4 processes (the original parent, the original child and the new two children of each of them). Each of these processes will execute the `printf("a")` statement, resulting in 4 prints of the letter ‘a’.

Question 10: What is the primary role of the Memory Management Unit (MMU)? Explain briefly your answer. **(4 points)**

ANSWER

The primary role of the Memory Management Unit (MMU) is to manage virtual memory.

It is responsible for translating virtual addresses to physical addresses. More specifically, it maps *virtual pages* to *physical frames* and enforces logic along the way. It does this using a *page table* (often organized as a *multi-level* hierarchy to save memory).

During this process, the MMU enforces logic via *control bits* in the *page table entry*:

- **Present Bit:** A page fault occurs if the page is not in RAM.
- **Protection Bits:** Read/Write/Execute permissions.
- **Referenced Bit:** Used by page replacement algorithms.
- **Changed Bit:** If page was modified, it requires writing back to disk before eviction.

Question 11: Which is the key disadvantage of the NFU (Not Frequently Used) page replacement algorithm? Explain briefly your answer. **(4 points)**

ANSWER

NFU doesn’t forget.

If a page is used a lot in the beginning of the process, but afterwards never again, the use frequency will be high and it will thus never be evicted. This implies that there is less space for pages that are used later on a (less) frequent basis.

Question 12: Assuming the current state of the memory is as shown in Figure 5, name a virtual address that will generate a page fault if accessed? Explain briefly your answer. **(4 points)**

ANSWER

The virtual page from byte address 24 KiB to 30 KiB will generate a page fault if accessed, because it is not mapped to any physical frame. Hence, a byte address in this range, for example byte address $24\,\text{KiB} + 1\,\text{B} = 24\,577\,\text{B}$, will generate a page fault.

Question 13: The DMA chip can transfer data from and to the memory without using the CPU. Name a scenario that it is not efficient to use DMA for memory transfer? Explain briefly your answer. **(4 points)**

ANSWER

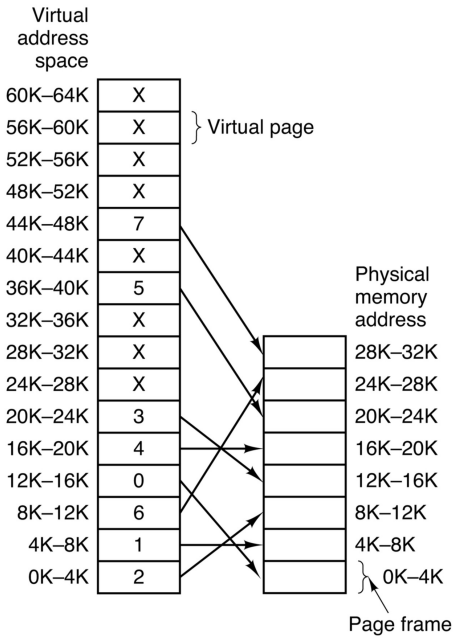

Figure 5

The DMA is slower than the CPU. If the CPU is idle, it is thus more efficient to let the CPU handle the memory transfer itself at a higher speed.

Furthermore, in (non-battery-powered) embedded systems, getting rid of DMA makes sense, as it saves money. (It is, however, more energy-efficient in battery-powered embedded systems.)

Question 14: Which is the purpose of a watchdog timer? Explain briefly your answer. **(4 points)**

ANSWER

A watchdog timer is a timer that resets the system if it expires.

It gets reset by the OS periodically, which means that if it expires, the OS is frozen. In this case, the watchdog timer will reset the system, such that the OS runs again.

Question 15: Which is the key advantage of developing device drivers using interrupt-driven I/O? Explain briefly your answer. **(4 points)**

ANSWER

The key advantage is that it eliminates *busy waiting* (or polling).

In interrupt-driven I/O, the CPU does not need to continuously check the device's status register. Instead, it can put the calling process to sleep and switch to executing other processes while the I/O device performs its task. The device notifies the CPU via an interrupt only when it is finished, thereby significantly improving CPU utilization and system efficiency compared to *programmed I/O*.

Question 16: In computer systems with multiple processors, name a challenge that characterises distributed systems as opposed to multicore processors? Explain briefly your answer. **(4 points)**

ANSWER

A major challenge is the lack of shared memory and the reliance on network communication.

In multicore systems (multiprocessors), processors share main memory and can communicate very quickly (nanoseconds). In contrast, distributed systems consist of independent computers that must communicate via message passing over a network, which introduces significantly higher latency (milliseconds) and unreliability issues, such as lost or out-of-order packets.

Long Questions

Question 1 (12 points)

i. Summarise the advantages and disadvantages of processes and threads.

ii. You are developing a web browser. The web browser needs to support multiple parallel tabs, so that the user can browse multiple websites in parallel. Would you implement the browser using multiple processes (one process per tab) or multiple threads (one thread per tab)? Motivate your answer and discuss if the disadvantages of your solution (processes or threads) are relevant in this use case. If relevant, also discuss how you would overcome them.

Question 2 (12 points)

i. Explain in your own words when using a spinlock (i.e., a busy-waiting mutex) in a multiprocessor system can improve the performance.

ii. You are using a multiprocessor system. The operating system implements hybrid mutexes. These mutexes operate as follows. If a thread requests to acquire a locked mutex, the thread first continuously polls the mutex (spins) for a period of time, T. After the time T passes, the thread yields (context switch) and retries when it gets rescheduled. The parameter T is configurable, taking values in μs in the range $[0, 65535]$. Setting $T = 0$ disables spinning.

After long-term statistical analysis, you know that the time a thread needs to wait for a locked mutex to be released follows the histogram provided in Figure 6. Moreover, a context switch takes 1000 μs.

Calculate the optimum value for the configuration parameter T, which minimises the overhead (i.e. the sum of time the CPU wastes spinning and switching). For simplicity, you can consider that when the thread gets rescheduled after the first context switch, it finds the mutex unlocked.

Question 3 (12 points)

After the presentation of the experiments of all other groups, propose a new experiment for the OS Challenge that your group has not tested before. Motivate the experiment by explaining why you believe it will improve the performance and describe how you would test if your hypothesis is true.