Written Exam

DM510: Operating Systems (Spring 2025)

June 13, 2025

This exam consists of 7 pages in total, including this one, with 4 topics and several questions in each topic. There are 60 points in total. The number of points is given for each question. Along with your answers, also explain how you arrived there, in a clear, but concise way.

Topic 1: Concurrency (12 points)

Recall that Amdahl's law states that

$$\text{speedup} \leq \frac{1}{\mathrm{S} + \frac{1-\mathrm{S}}{\text{cores}}}$$

where S be the ratio of sequential operations to all operations.

The program `Fcast` computes weather forecasts for a Danish news channel. This computation currently takes 9 minutes on a single CPU core. Management is unsatisfied with this and suggests to decrease the time to 3 minutes by using 3 cores instead of one. You estimate that 20% of the operations are sequential.

1.a) ***[3 points]*** Explain intuitively why the suggestion by management will not suffice for their goal. Then using Amdahl's law calculate the minimum number of cores necessary to reduce the time to 3 minutes. What is the maximum speedup that could be achieved by adding more cores?

Solution: The ideal speedup of 3× can only be achieved if the program is fully parallelizable. Since there are sequential operations, which do not benefit from parallelization, the speedup will be worse.

A speedup of at least 3 is necessary.

$$3 \leq \frac{1}{0.2 + \frac{0.8}{\text{cores}}}$$

$$3(0.2 + \frac{0.8}{\text{cores}}) \leq 1$$

$$\frac{0.8}{\text{cores}} \leq 1/3 - 0.2$$

$$\frac{0.8}{1/3 - 0.2} \leq \text{cores}$$

$$\text{cores} \geq 6$$

Hence, at least 6 cores are necessary

For infinite number of cores, the speedup would approach $1/0.2 = 5$.

The following code implements a semaphore using the atomic instructions `compare_and_swap()` and `increment()`:

```
1  void wait(int *sem) {
2    while (true) {
3      int old = *sem;
4      if (old > 0) {
5        if (compare_and_swap(sem, old, old - 1))
6          break;
7      }
8    }
9  }
10
11 void signal(int *sem) {
12   increment(sem)
13 }
```

1.b) ***[2 points]*** For simplicity, your teammate suggests to replace lines 5-6 of the semaphore implementation by `decrement(sem)`, which atomically subtracts 1 from `*sem`. Would the solution still be correct?

Solution: No, for example: value of `*sem` is 1. Then thread T_1 executes `wait()` until line 4 (the if condition is evaluated as true). T_1 is preempted and T_2 executes `wait()` completely. Afterwards, T_1 continues. This results in a negative value for `*sem`, which must not happen.

1.c) ***[2 points]*** Recall that mutex is equivalent to a semaphore that takes only values 0 and 1. Your teammate notes that in an implementation of a mutex the `release()` operation (corresponding to `signal()` here) does not need to be made atomic and suggests to replace line 12 by `*sem = *sem + 1`. Is the suggestion correct for general semaphores?

Solution: No: suppose thread T_1 is preempted after computing `*sem + 1` in a register `reg1`. Then thread T_2 increments `*sem`, but when T_1 writes the contents of `reg1` to `*sem` the increment of T_2 is lost.

1.d) ***[2 points]*** To reduce risk of race conditions, why don't compilers automatically translate all increments of variables (for example, `i++`) into the atomic increment instructions?

Solution: Atomic instructions are slower and in most cases this would be unnecessary. Optional details: the atomic instruction require locking the memory bus, which may cost CPU cycles on this and other cores.

1.e) ***[3 points]*** As in Project 1, critical sections are sometimes implemented by temporarily disabling interrupts. Explain this approach and two drawbacks of it.

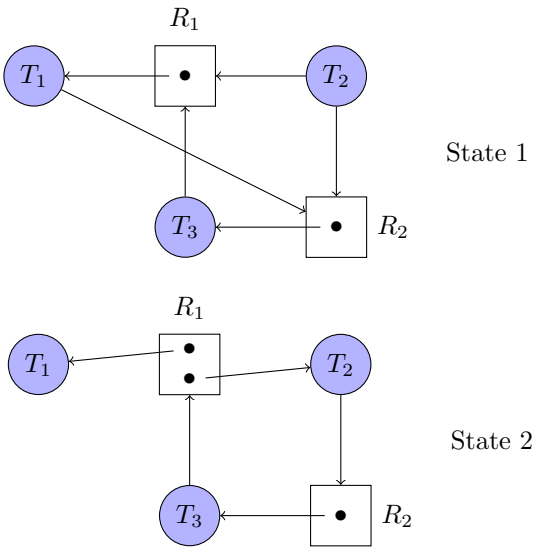
Solution: Critical sections are code sections with the property that only one process may be in a critical section at any given time . On single-core systems we can disable interrupts when entering a critical section and enable them again when leaving the critical section. This guarantees that the current process cannot not be preempted until it leaves the critical section again. Since no other process can become active after disabling interrupts, no other process can enter a critical section simultaneously.

On multi-core systems several processes may run in parallel and remain active even if one disables interrupts. Both may then enter critical sections simultaneously, so disabling interrupts is not sufficient.

Disabling interrupts for a longer period of time can also cause problem with responsiveness or system clocks that rely on regular interrupts.

Topic 2: Deadlocks (16 points)

The following figure shows states of systems where threads request or hold resources. Threads are depicted as cycles and resources are depicted as rectangles, with dots representing the instances of this resource. An arrow from a thread to a resources describes that the thread requests this resource, an arrow from an resource instance to a thread describes that the instance is currently held by the thread.



2.a) ***[3 points]*** Which of the states above are deadlocked and why (or why not)?

Solution: State 1 is deadlocked because of the cycle $T_1 - R_2 - T_3 - R_1 - T_1$.
State 2 is not deadlocked because T_1 can continue and ultimately release its instance of R_1. Then T_3 can aquire R_1 and continue, ultimately releasing R_2. Then T_2 can aquire R_2 and also continue.

Recall, the system call `wait()` suspends the calling process until a child process that was previously created with `fork()` terminates. In the deadlock formalism of resources and threads, we may think of each process P_i having a personal resource R_i with one instance for each child process. A child process of P_i holds one instance of R_i and releases the instance when it terminates. When P_i calls `wait()` this corresponds to waiting for R_i.

2.b) ***[3 points]*** Unless another shared resource is involved, `fork()` and `wait()` alone cannot lead to a deadlock. Explain this by showing that one of the necessary conditions for deadlocks cannot be satisfied.

Solution: The circular wait condition cannot be satisfied. In the formalism above a process can only wait for its own resource and hold its parent's resource. Since parent-child relation is a tree structure there cannot be a cycle.

2.c) *[3 points]* A single mutex without other shared resources involved cannot result in a deadlock. Combined with `fork()` and `wait()`, however, already a single mutex suffices to create a deadlock. Give an example in pseudocode and explain it.

To avoid ambiguities, your example should not call `fork()` while holding a mutex.

Solution:

```
pid = fork();
if (pid == 0) {
  /* child */
  lock(&mutex);
  release(&mutex);
}
else {
  /* parent */
  lock(&mutex);
  wait();
  release(&mutex);
}
```

If the parent first aquires the mutex, the child waits for it infinitely while the parent waits for the child to terminate.

Consider the following snapshot of a system:

	Allocation			Max		
	A	B	C	A	B	C
T_1	2	1	1	2	3	2
T_2	0	0	1	3	0	1
T_3	0	1	0	0	1	1
T_4	1	3	2	2	3	3

$$\text{Available} = (0, 1, 1)$$

2.d) *[4 points]* Determine if the system is in a safe state using the subroutine from Banker's algorithm. If the state is safe, illustrate the order in which the threads may complete. Otherwise, illustrate why the state is unsafe.

Solution: Initialization:

	Need		
	A	B	C
T_1	0	2	1
T_2	3	0	0
T_3	0	0	1
T_4	1	0	1

$\text{Work} = (0, 1, 1)$

T_3 can complete:

	Need		
	A	B	C
T_1	0	2	1
T_2	3	0	0
T_4	1	0	1

Work $= (0, 2, 1)$

T_1 can complete:

	Need		
	A	B	C
T_2	3	0	0
T_4	1	0	1

Work $= (2, 3, 2)$

T_4 can complete:

	Need		
	A	B	C
T_2	3	0	0

Work $= (3, 6, 4)$

Finally, T_2 can complete. This means the state is safe.

2.e) ***[3 points]*** Suppose that in the state above T_1 requests $(0, 0, 1)$. Determine with Banker's algorithm whether this request can be granted.

Solution: After granting the resource, the state would change to

	Allocation			Max		
	A	B	C	A	B	C
T_1	2	1	2	2	3	2
T_2	0	0	1	3	0	1
T_3	0	1	0	0	1	1
T_4	1	3	2	2	3	3

Available $= (0, 1, 0)$

We test if this is a safe state as before. Initialization:

	Need		
	A	B	C
T_1	0	2	0
T_2	3	0	0
T_3	0	0	1
T_4	1	0	1

Work $= (0, 1, 0)$

Work is not large enough to guarantee that any of the threads can complete. Thus the state is unsafe. Therefore, the request cannot be granted.

Topic 3: Main Memory (18 points)

We consider a CPU architecture with 16-bit addresses and a naive page table implementation. The first 3 bits of the address specify the page or frame number. The current page table of a process contains the following entries (in binary):

page	frame
000	010
001	110
010	-
011	111
100	100
101	-
110	-
111	101

3.a) ***[2 points]*** How large is a page or frame in the architecture above?

Solution: A page/frame has size $2^{16-3} = 8196$.

3.b) ***[2 points]*** Based on the page table above, translate the following logical addresses into physical addresses:

0110111010110000
0011010000010110
1111111110000000

Solution:

1110111010110000
1101010000010110
1011111110000000

In the following example, process A wants to create a process B and pass a large chunk of raw data as a parameter. Since it would not be feasible to pass it as a command line argument, process A instead wants to write the data into memory, which process B then should read.

The following shows an implementation using the `fork()` system call.

```
char* data;
...
pid = fork();
if (pid == 0)
{ /* process B */
  consume(data);
  ...
}
else
{ /* process A */
  free(data);
  ...
}
```

3.c) *[4 points]* Describe the following terms in 2-3 sentences each: page fault, least-recently-used replacement (LRU), inverted page table, translation look-aside buffer

Solution: page fault: if a page is accessed, which is not in memory, a page fault occurs. This page is then usually loaded from the backing store, which causes a delay.

LRU: If a new page has to be brought to memory (e.g. because of a page fault), but there is not enough space, then another page has to be removed. LRU removes the page that has not been used for the longest time.

inverted page table: A page table implementation which maps physical frames to logical pages (instead of the other way around). This means the size of the table is exactly the number of physical frames, which may be much lower than the number of pages. However, mapping pages to frames is no longer a simple lookup.

translation look-aside buffer: A cache implemented in hardware that stores some page-frame pairs. It reduces the number of page table lookups.

3.d) *[2 points]* Describe what the system call `fork()` does using the example above.

Solution: fork() creates two copies of the current process, including two exact copies of the address space. The only difference between the two processes is the return value of fork(). Thus, process B can read the contents of data as intended.

3.e) *[2 points]* Discuss race conditions: does the behavior of the program depend on whether `consume(data)` executes before `free(data)`?

Solution: The order of execution does not matter, since all changes that process A makes (including freeing data) only affect its own copy of the address space, but not the one that process B is working with.

A pointer can be cast into an integer (describing the raw logical address). Below is an attempt at an alternative solution for the previous problem. In this solution we write two entirely separate programs for process A and B and process B is not a child process of A. Instead, process A is started from the terminal and prints out the address of `data`:

```
1 char* data;
2 ...
3 printf("%lld\n", (long long int)data);
4 ...
```

The output of this may for example be 106934719226528. Then concurrently to process A, we start process B from another terminal and pass the previous number (106934719226528) as a command line argument. Process B then executes the following code:

```c
char* data;
int main(int argc, char** argv) {
  /* atoll converts string to long long int */
  data = (char*) atoll(argv[1]);
  consume(data);
  ...
}
```

3.f) ***[4 points]*** The approach above does not work as intended. Explain this using logical and physical addresses and page tables.

Solution: The approach does not work as intended. Processes A and B have seperate page tables. Here, we are passing a logical address from process A to B, but this logical address is mapped to different physical addresses in both processes. In fact, there is not any logical address for process B that would map to the physical address that process A wants to share. (Optional: The code above would most likely result in a segmentation fault.)

3.g) ***[2 points]*** Describe what shared memory is and (on a high level) how this could be used to repair the example above.

Solution: Shared memory allows two processes to share frames, i.e., both processes have logical addresses that map (in their page tables) to the same physical frame. After setting up shared frames between process A and B, process A could write to them and process B could read from them in a similar manner as attempted in the previous example.

Topic 4: Storage and File Systems (14 points)

Consider a disk drive with cylinders numbered from 0 to 1999. The drive is currently serving a request at cylinder 401 and the previous served request was at cylinder 350. The queue of pending requests in FIFO order is: 30, 1910, 500, 37, 1400, 783.

4.a) ***[2 points]*** In which order would the SCAN and FCFS algorithms serve the requests? What distance (in number of cylinders) would the disk arm move in each algorithm?

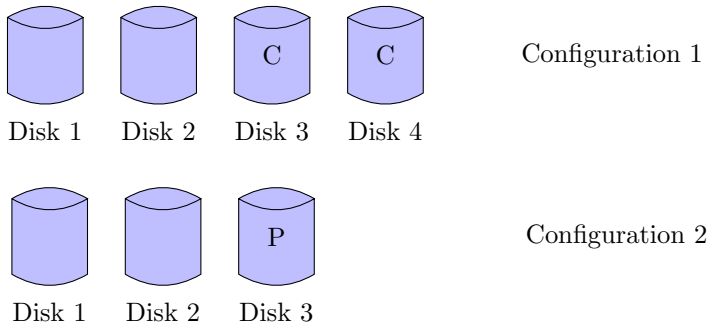
Solution: SCAN: 500, 783, 1400, 1910, 37, 30 (distance: $1910-401+1910-30 = 3389$).

FCFS: 30, 1910, 500, 37, 1400, 783 (distance: $401-30+1910-30+1910-500+500-37+1400-37+1400-783 = 6104$).

4.b) ***[2 points]*** Explain why SCAN and C-SCAN are used with hard disks (HDD), but not with flash drives (NVMs/SSDs).

Solution: SCAN and C-SCAN aim to reduce the movement of the disk arm. An NVM does not have an arm or physical movements that would benefit from subsequent requests being close to each other.

The following picture shows two RAID configurations: Configuration 1 uses RAID 1 (mirrored disks), where Disk 3 mirrors Disk 1 and Disk 4 mirrors Disk 2. Configuration 2 uses RAID 4 (block-interleaved parity), where Disk 3 is the parity disk.



4.c) ***[2 points]*** For each configuration: what is the minimum number of simultaneous disk failures that can result in data loss? Which combinations of disks would lead to data loss if they fail simultaneously?

Solution: In both configurations 2 disk failures can lead to data loss.

In Configuration 1 the following combinations result in data loss: Disk 1 + Disk 3; Disk 2 + Disk 4; (optional: any combination of 3 or more disks).

In Configuration 2 any combination of 2 or more disks results in data loss.

4.d) ***[3 points]*** Consider the basic implementation of a file system via FUSE as in Project 3 (without potential extensions). Name three ways in which this implementation differs from practical file systems.

Solution: (It is intentional that the students have to know/remember the project.) Example answers (there are more):

- The naive file system from Project 3 was implemented as an unstructured array of inodes. Essentially any operation on the file system requires a scan of the entire array. For practical file systems this would be too inefficient.
- In Project 3, each file has the same fixed size. A practical file system would need variable size files, which also requires block allocation and other features.
- FUSE implements a file system in user mode. Usually, file system functionality runs in kernel mode.

4.e) ***[3 points]*** Describe in 2-3 sentences what contiguous allocation is in the context of a file system. Name one advantage and one disadvantage.

Solution: In contiguous allocation we allocate a contiguous range of space/blocks of a storage system to files. The file-control-block/inode then contains the first and the first physical address of the file and the length.

Advantage: easy and efficient to determine logical-to-physical mapping.

Disadvantage: can lead to external fragmentation.

4.f) ***[2 points]*** In 2-3 sentences, describe an alternative to contiguous allocation.

Solution: Several possible answers. Alternatives discussed in class were linked allocation and indexed allocation.

Linked allocation: the blocks allocated to a file form a linked list. The file-control-block/inode contains a pointer to the first block, then each block contains (apart from the data) a pointer to the next block.

Indexed allocation: The file-control-block/inode contains a pointer to a special index block. The index block contains an array of pointers to all blocks allocated to this file.