# Written Exam

DM510: Operating Systems (Spring 2025)

August 6th, 2025

This exam consists of 6 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: CPU Scheduling (16 points)

Consider the following CPU bursts by processes P1-P5.

| Process | Burst Time | Priority |
|---|---|---|
| P1 | 10 | 5 |
| P2 | 2 | 3 |
| P3 | 7 | 4 |
| P4 | 5 | 1 |
| P5 | 12 | 2 |

**1.a)** ***[4 points]*** For each of the following algorithms produce a schedule for the given CPU bursts and compute the average response time (the time the CPU burst has finished):

1. Priority scheduling
2. Shortest-Job-First
3. Round-Robin with time quantum $q = 5$

**Solution:** Priority scheduling:

| Process | Burst Time | Completion Time |
|---|---|---|
| P4 | 5 | 5 |
| P5 | 12 | 17 |
| P2 | 2 | 19 |
| P3 | 7 | 26 |
| P1 | 10 | 36 |

Average response time: $(5 + 17 + 19 + 26 + 36)/5 = 20.6$.

SJF:

| Process | Burst Time | Completion Time |
|---|---|---|
| P2 | 2 | 2 |
| P4 | 5 | 7 |
| P3 | 7 | 14 |
| P1 | 10 | 24 |
| P5 | 12 | 36 |

Average response time: $(2 + 7 + 14 + 24 + 36)/5 = 16.6$.

RR:

| Process | Start | End | Completion Time |
|---|---|---|---|
| P1 | 0 | 5 | x |
| P2 | 5 | 7 | 7 |
| P3 | 7 | 12 | x |
| P4 | 12 | 17 | 17 |
| P5 | 17 | 22 | x |
| P1 | 22 | 27 | 27 |
| P3 | 27 | 29 | 29 |
| P5 | 29 | 34 | x |
| P5 | 34 | 36 | 36 |

Average response time: $(7 + 17 + 27 + 29 + 36)/5 = 23.2$.

**1.b)** ***[2 points]*** Apart from response time, name two other criteria for evaluating scheduling algorithms.

**Solution:** The following have been named in the course:

- CPU utilization: Executing as many process instructions as possible
- Throughput: Complete as many processes/tasks per time unit as possible
- Turnaround time: Minimize time to complete a particular process
- Waiting time: Minimize time a process waits in ready queue
- Fairness: Make sure that every process/task gets a fair share of CPU time and no task "starves", i.e., never completes
- Efficiency of algorithm: Scheduling algorithm itself should not create significant latency/overhead

**1.c)** ***[2 points]*** Give an advantage of Round-Robin over Shortest-Job-First.

**Solution:** None of the processes can starve in Round-Robin, i.e., we are always making progress on every process.

**1.d)** ***[3 points]*** Describe how the hardware clock is used to implement a *preemptive* scheduling algorithm. Which of the scheduling algorithms above are preemptive?

**Solution:** The scheduler needs to be able to regain control over the CPU, even if a process keeps running. For that the kernel sets a timer that causes an interrupt after a specified duration. In the interrupt handler, it will then run the scheduler, which decides whether to preempt the current process.

Only Round-Robin is preemptive.

**1.e)** ***[2 points]*** You are using a text editor and experience significant delays on user inputs, for example, when pressing a key. When running the command "top" you see that the CPU utilization is at 100% due to a background process. Explain how the delay could be related to CPU scheduling and give a suggestion on how to improve the user experience.

**Solution:** The text editor process might have a lower priority than the other process. Increasing its priority could help. It might also be that the scheduler uses a variant of Round-Robin and the time quantum is set too high.

**1.f)** ***[3 points]*** Describe in 2-3 sentences what a context switch is. Explain the role of context switches in the overall system performance.

**Solution:** A context switch is when the CPU changes from executing one process to another. It needs to save the current state of the old process to memory, including all register values, and restore the previous state of the new process from memory. Context switches can take substantial time and if a system has many context switches, it cannot utilize the CPU for other tasks, potentially slowing down the system.

## Topic 2: Synchronization (14 points)

The following code is supposed to calculate the number of non-zero entries in an array:

```
1 int array[1024];
2 int count = 0;
3 initialize(array);
4
5 #pragma omp parallel for
6 for (int i=0; i < 1024; ++i) {
7   count += !!array[i];
8 }
```

Here, note that `!!a` is 0 if `a` is 0 and 1 if `a` is anything else. Due to the openMP parallelization, the iterations of the loop may be executed in different threads.

**2.a)** ***[2 points]*** Elaborate the race condition that can occur in the code above.

**Solution:** Several threads may concurrently execute line 7. They will load `count` into a register, perform operations on it and then write the result back to `count`. It might happen that one thread $T_1$ writes back the (increased) old value losing the update that another thread has made after $T_1$ read `count`.

**2.b)** ***[3 points]*** Describe how the race condition can be avoided using the atomic operation `int compare_and_swap(int* obj, int old, int new)`. Provide the corresponding code.

**Solution:**

```
1 int array[1024];
2 int count = 0;
3 initialize(array);
4
5 #pragma omp parallel for
6 for (int i=0; i < 1024; ++i) {
7   while (true) {
8     int old = count;
9     if (compare_and_swap(&count, old, old + !!array[i])) {
10      break;
11    }
12  }
13 }
```

The thread reads count, then replaces it with the updated value only if it has not changed. Otherwise, tries again.
Another way to avoid the race condition is to use mutexes:

```
1 int array[1024];
2 int count = 0;
3 mutex_t m;
4 initialize(array);
5
6 #pragma omp parallel for
7 for (int i=0; i < 1024; ++i) {
8   acquire(&m)
```

```
9     count += !!array[i];
10    release(&m)
11  }
```

**2.c)** *[2 points]* The solution using mutexes would not be faster than a naive sequential implementation. Explain why.

**Solution:** There is no parallelism, because the mutex forces threads to be running line 9 exclusively. With the overhead of the mutex operations, this might even slow down the implementation.

**2.d)** *[3 points]* Still using mutexes, modify the code so that it would run faster and argue why it would.

*Hint:* introduce a count variable for each parallel thread.

**Solution:**

```
int array[1024];
int count = 0;
initialize(array);

#pragma omp parallel for
for (int j=0; j < 8; ++j) {
  int local = 0;
  for (int i=1024/8*j; i < 1024/8*(j+1); ++i) {
    local += !!array[i];
  }
  acquire(&m)
  count += local;
  release(&m)
}
```

Here, each thread counts locally the non-zeroes of a part of the array. We synchronize only at the point where we add the local count to the global count. Thus, the sequential part of the program is becomes smaller and the speedup is higher, as seen e.g. from Amdahl's law.

**2.e)** *[4 points]* Two alternative implementations of an mutex use (1) busy waiting or (2) moving the thread/process to the waiting queue of the CPU scheduler. Explain both approaches in 2-3 sentences and name one advantage of each.

**Solution:** (1) Busy waiting means that in a loop the process repeatedly reads the value of a mutex until it is available. This means the CPU wastes cycles for waiting. It is efficient if the probability that a mutex is held by another process is very low or it is only held for a short time. This is because busy waiting does not require switching to kernel mode, which can be a significant overhead.

(2) The process may also request that it is suspended until the mutex becomes available. Once the mutex is available, the CPU scheduler will move the process to the ready queue so that it can continue. This means the process does not waste CPU cycles while it is waiting. However, this requires the kernel to execute and the overhead of switching to kernel mode.

## Topic 3: Main Memory (14 points)

We consider a CPU architecture with 32-bit addresses and a naive page table implementation. The first 4 bits of the address specify the page or frame number. The current page table of a process contains the following entries (in hexadecimal; dashes "-" indicate that the invalid bit is set):

| page | frame |
|---|---|
| 0 | - |
| 1 | - |
| 2 | - |
| 3 | 0 |
| 4 | 1 |
| 5 | - |
| 6 | - |
| 7 | - |

| page | frame |
|---|---|
| 8 | - |
| 9 | - |
| A | - |
| B | 2 |
| C | - |
| D | - |
| E | - |
| F | - |

**3.a)** ***[2 points]*** Based on the page table above, translate the following logical addresses (given in hexadecimal) into physical addresses:

3A44
B40B
BE3F

**Solution:**

0A44
240B
2E3F

**3.b)** ***[2 points]*** Describe one disadvantage of having too large page sizes. Describe one disadvantage of having too small page sizes.

**Solution:** Large page sizes can cause internal fragmentation: because we can only allocate multiples of the page size, we might need to allocate more memory than a process requested.

Small page sizes mean that there are many pages, which requires a larger page table.

**3.c)** ***[3 points]*** Explain in detail what happens when an instruction references the logical address 7AA0 with the page table above.

**Solution:** The page entry for 7 has the invalid bit set. This means a page fault occurs. The kernel needs to check if the page is available in the backing store and in this case fetch it. It might also need swapping out another page to create a free frame for this page. It then restarts the instruction.

It might also be that the kernel determines that access to the memory address is not allowed, which results in abnormally terminating the process.

**3.d)** ***[4 points]*** Assume that process $P$'s page table has the state above and there are 3 frames allocated to $P$ (frames $0, 1, 2$). The last three pages that were referenced by $P$ are (in this order) B, 4, 3. Consider the following page reference string, describing exactly which pages $P$ references next and in which order:

$$1, 2, 4, 3, 2, 1, 2, 3, 6, 3, 2, 1$$

How many page faults would occur for the following replacement algorithms?

1. LRU replacement
2. Optimal replacement

**Solution:**

LRU:

| ref | 1 | 2 | 4 | 3 | 2 | 1 | 2 | 3 | 6 | 3 | 2 | 1 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| pages | 4,3,1 | 3,1,2 | 1,2,4 | 2,4,3 | 4,3,2 | 3,2,1 | 3,1,2 | 1,2,3 | 2,3,6 | 2,6,3 | 3,6,2 | 3,6,1 |
| page fault | x | x | x | x | | x | | | x | | | x |

Number of page faults: 7

Optimal:

| ref | 1 | 2 | 4 | 3 | 2 | 1 | 2 | 3 | 6 | 3 | 2 | 1 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| pages | 1,4,3 | 2,3,4 | 2,3,4 | 2,3,4 | 2,3,4 | 1,2,3 | 1,2,3 | 1,2,3 | 2,3,6 | 2,3,6 | 2,3,6 | 1,2,3 |
| page fault | x | x | | | | x | | | x | | | x |

Number of page faults: 5

**3.e)** ***[3 points]*** A research lab at SDU uses a computer to run 50 scientific simulations. Each of the simulations involves heavy computation on a large data set, requiring around 2GB of memory in each simulation. The lab computer has four CPU cores and 8GB of main memory. You consider three alternative setups:

1. Running the simulations after each other.
2. Running 5-10 simulation in parallel and starting a new one each time another has finished.
3. Running all simulations in parallel.

Which of these alternatives do you expect to finish all simulations in the shortest time? Justify your answer.

**Solution:** Alternative 2 would probably perform the best: 1 might suffer from under-utilization of the CPU. 3 could lead to thrashing: each process is allocated a too low number of frames, causing many page faults and the system spending much of its time transferring data between RAM and backing store.

## Topic 4: Network and Security (16 points)

**4.a)** ***[2 points]*** Video streaming is an application that requires a high network throughput. Explain how direct-memory access (DMA) can improves the performance of such an application compared to more naive device I/O.

**Solution:** In DMA, the device (network card) would write a large amount of data directly into a designated part of main memory. Only when it is completely finished, it will notify the CPU via an interrupt, which can then work on the data in memory. This means the CPU does not need to read byte by byte from the device.
In the following, we consider the example application of accessing a Linux server via a remote shell (for example the program `ssh`): here the server allows clients to send terminal input over the internet and executes it in a terminal/shell.

**4.b)** ***[2 points]*** Discuss whether TCP or UDP would be more suitable for implementing a remote shell and justify why.

**Solution:** Many of the features of TCP are appropriate for the application: the correct order of input/output must be guaranteed, reliability and recovery from packet loss is most likely important. Since TCP implements these features and UDP does not, TCP is the natural choice.

**4.c)** ***[2 points]*** Describe two possible security violations that could occur if the remote shell connection is not encrypted.

**Solution:** The following security violations were discussed in the course:

- Breach of confidentiality: Unauthorized reading of data
- Breach of integrity: Unauthorized modification of data
- Breach of availability: Unauthorized destruction of data
- Theft of service: Unauthorized use of resources
- Denial of service (DOS): Prevention of legitimate use

Each of them could easily occur if an attacker intercepts the connection or inserts packets.

**4.d)** ***[3 points]*** Describe what *symmetric* encryption is and how it could be used to secure a remote shell connection.

**Solution:** In symmetric encryption, both client and server have the same key that they use to encrypt their messages. One could imagine the server storing the users password and this password could then be used to encrypt the messages. Without the password and if implemented properly, an attacker could not read the messages and it could also not forge and insert messages in the connection.

Consider a server that processes requests from a TCP connection line by line. The following code waits until a full line has been read (identified by the line-break `'\n'`) and only then consumes it.

```
while (true) {
  char c;
  char line[1024];
  int pos = 0;
  while (true) {
    read(fd, &c, sizeof(char)); //read one byte from TCP connection
    if (c == '\n') {
      line[pos] = 0; // end of string
      break;
    } else {
      line[pos] = c;
      pos++;
    }
  }
  consume(line);
}
```

**4.e)** ***[3 points]*** Explain why this program is vulnerable to code injection and explain how such an attack would work.

**Solution:** The buffer `line` is 1024 bytes long. An attacker could send a longer message, which would cause the program to overwrite memory after the end of the buffer. The attacker could insert instruction data in the buffer and use the buffer overflow to modify the return address to this injected instruction data. This way it could take over control of the CPU.

**4.f)** ***[2 points]*** Modify the code so that it is safe.

**Solution:**

```
while (true) {
  char c;
  char line[1024];
  int pos = 0;
  while (true) {
    read(fd, &c, sizeof(char));
    if (c == '\n') {
      line[pos] = 0;
      break;
    } else {
      line[pos] = c;
      pos++;
      if (pos >= 1024) {
        // possibly add error handling here
        break;
      }
    }
  }
  consume(line);
}
```

**4.g)** ***[2 points]*** Describe two measures that an operating system can take to mitigate the risks of code injection.

**Solution:** The system could flag pages as executable and non-executable. This makes it harder for the attacker to execute its inserted code.

The "Principle of least priviledge" helps reduce the power of attackers in case they do gain control.