Western Norway University of Applied Sciences

Department of Computer science, Electrical engineering and Mathematical sciences

Examination in:	DAT103 – Computers and Operating Systems
Day of examination:	10 January 2022
Time of examination:	9:00 – 13:00
Permitted aids:	Both written and printed aids. No communication with other people. (I.e., exam problems should be solved without help from others; accepting help or providing help to others is considered as cheating.)
Language:	English

This problem set consists of 10 pages, excluding this cover page.
You can answer in either English or Norwegian.

!!! VERY IMPORTANT !!! Read this before you continue.

- During the exam, you can contact us by Zoom, between 9:30 – 13:00:
  Meeting ID: 658 1074 1912
  Passcode: h21-dat103
  You will have to wait at a virtual waiting room until we let you into the meeting room to ask questions.
- Problems 3, 4.2 and 6.1 are **specific** to your candidate number (kandidatnummer) of this DAT103 exam.
- These problems will be corrected according to your candidate number.
- **Half of the points** will be deducted if you do not answer these problems according to your candidate number.
- In Problems 3, 4.2 and 6.1, we use
  Even to indicate questions and values for **even candidate numbers**
  Odd to indicate questions and values for **odd candidate numbers**
- Should you have any doubt, you should contact us.

Good Luck!
Dag and Violet

Problem 1 – Multiple Choice Questions (28%)

For each of the multiple choice questions below, select at most one choice.

1.1 Which one of the following is **correct**?

(a) Program counter contains the instruction the processor is currently executing
(b) Program counter contains the address of the instruction the processor is currently executing
(c) Program counter is a register
(d) All of the above
(e) None of the above

Solution: Program counter is a register □

1.2 Which one of the following is **incorrect** about pipelining?

(a) Pipelining allows executing multiple instructions at the same time on one core
(b) The stages of an instruction cycle in the pipeline require different processing times
(c) Sometimes the prefetched instruction is not the instruction that is executed next
(d) All of the above
(e) None of the above

Solution: Pipelining allows *executing* multiple instructions at the same time on one core □

1.3 A page fault occurs when ...

(a) A requested page is in the backing store
(b) A requested page number is found in translation look-aside buffer (TLB)
(c) A requested page is in memory
(d) All of the above
(e) None of the above

Solution: None of the above □

1.4 Consider a processor that needs 5 ns to access the cache memory and 200 ns to access the main memory. Assume the cache hit ratio is 95%, what is the average memory access time of the processor?

(a) 5 ns
(b) 10 ns
(c) 15 ns
(d) 20 ns
(e) 100 ns
(f) None of the above

Solution: $0.95 \times 5 + 0.05 \times (5 + 200) = 15$ ns □

1.5 Consider the instruction `mov edx [hello]`. How many operands does this instruction have?

(a) 3
(b) 2
(c) 1
(d) 0
(e) None of the above

Solution: 2 □

1.6 Consider an instruction with two operands O_1, O_2. If O_1 uses *PC-relative* addressing mode, O_2 uses *indirect* addressing mode. How many memory accesses are required in total to fetch O_1 and O_2?

(a) 4
(b) 3
(c) 2
(d) 1
(e) 0
(f) None of the above

Solution: 1+2=3 □

1.7 Let ℓ be a lock, and `x` a shared `int` variable initialised to `0`. Consider two processes P_1 and P_2 that concurrently run the following pieces of code:

```
1 // Run by P1 //
2 x--;
```

```
3 // Run by P2 //
4 acquire(ℓ);
5 x--;
6 release(ℓ);
```

Which one of the following is **incorrect**?

(a) The value of `x` is -1 when P_1 and P_2 finish
(b) The value of `x` is -2 when P_1 and P_2 finish
(c) There is a race condition
(d) All of the above
(e) None of the above

Solution: None of the above □

1.8 Which one of the following is a function of a dispatcher?

(a) To context-switch between processes
(b) To give control of the CPU to the process selected by the short-term scheduler
(c) To jump to the proper location in the user program to restart that program
(d) All of the above
(e) None of the above

Solution: All of the above □

1.9 Which one of the following is **incorrect**?

(a) Paging allows memory allocated to a process to be noncontiguous
(b) Paging solves the problem of internal fragmentation
(c) Paging solves the problem of external fragmentation
(d) Paging avoids the problem of thrashing
(e) All of the above
(f) None of the above

Solution: Paging solves the problem of internal fragmentation □

1.10 Which one of the following scheduling algorithms has starvation problem?

(a) Shortest job first
(b) Multilevel Queue Scheduling
(c) Shortest-remaining-time-first
(d) All of the above
(e) None of the above

Solution: All of the above □

1.11 Which one of the following is **correct**?

(a) PCBs are not actually needed by the operating system, but is only maintained in order to provide the user about statistics for the processes

(b) PCBs are the data structure needed for the operating system to implement multi-programming, context shift, and even process forks
(c) A PCB does not contain a program counter
(d) A PCB contains the computer's host name
(e) All of the above
(f) None of the above

Solution: PCBs are the data structure needed for the operating system to implement multi-programming, context shift, and even process forks □

1.12 Which one of the following is **incorrect**?

(a) The `fork()` system call is used to create a new process on a UNIX computer
(b) A new child process created using the `fork()` system call is initially identical to the parent process
(c) A process can use the return value from the `fork()` system call to decide if it is the parent or child process
(d) The `exec()` system call can be used to replace the program content of a child process
(e) All of the above
(f) None of the above

Solution: None of the above □

1.13 Which one of the following is **correct**?

(a) Creating a new process hardly requires any resources at all, thus there is no need to limit the usage of this
(b) A process and a thread are just different names for exactly the same thing
(c) A thread is also called a light process
(d) It is only possible to create threads on a multi-CPU/core system
(e) All of the above
(f) None of the above

Solution: A thread is also called a light process □

1.14 Which one of the following is **incorrect**?

(a) A system call is used by a user program to request the operating system to perform a task on its behalf
(b) A system call may perform a task that a user program would not be allowed to perform itself
(c) Usually system calls are used through high-level system libraries
(d) A system call is also called a software interrupt
(e) None of the above

Solution: None of the above □

Problem 2 – Miscellaneous (13%)

2.1 Let ℓ_1 and ℓ_2 be locks, and `x` a shared `int` variable initialised to `1`. Consider three processes P_1, P_2 and P_3 that concurrently run the following pieces of code:

```
1  // Run by P1 //
2  acquire(ℓ1);
3  x++;
4  release(ℓ1);
```

```
5  // Run by P2 //
6  acquire(ℓ2);
7  x--;
8  release(ℓ2);
```

```
9  // Run by P3 //
10 acquire(ℓ1);
11 x++;
12 release(ℓ1);
```

List all possible values that `x` could have when all three processes have finished.

Solution: 0, 1, 2, 3 □

2.2 When an application has a need to perform multiple tasks at the same time (in parallel), this can be achieved by either starting additional processes or threads. What are the differences between processes and threads? Give examples of "typical" use cases for both.

Solution: In general, starting a new process is significantly more resource demanding than starting a thread. A new thread only requires a new thread control block, while memory and oher resources are shared. A new process, on the other hand, requires its own copy of the memory block and all child threads. Typically, a thread is created when a program has a need to perform several tasks concurently, in such way that memory and other resources may be shared. An example of this may be a media player that needs to download media from a server and decode it at the same time. A process on the other hand may be more suitable when a program has a need to launch a child process that will operate more "independently" in separate memory and having the executable code (text) replaced. □

2.3 Hardware components like CPU, memory, I/O controllers, etc., can communicate either via a shared bus or via a point-to-point–connection. What are the differences between these two types of connections, and what are the relative advantages/disadvantages of each? Point-to-pont–connections are becoming increasingly popular. Why is this?

Solution: A shared bus will typically connect several pieces of hardware, for example CPU, RAM, disk controller, graphics controller, network controller, sound controller, etc. A point-to-point–connection, on the other hand, only connects exactly two two pieces of hardware. For example RAM and CPU or CPU and hard disk controller. A shared bus needs more advanced methods to prevent several pieces of hardware to utilise the bus at the same time. The overhead of preventing collisions will typically make it difficult to utilse the full bandwidth of the bus. Also, a shared bus is harder to design robustly for higher data rates as the increased number of connections will distort the electrical signals. For these reasons, point-to-point–connections are becoming increasingly popular. □

2.4 What are the differences between a CPU and a core? Are there any potential implications with respect to the performance if two cores are physically located on the same chip, or if they are located on two different chips? Why and why not?

Solution: A CPU usually refers to a physical die, that may contain several cores internally. For some applications it does not matter if the cores are on different physical CPU die or on different ones. An example of this is if the processes running on them do not need to excange data. However, if two processes need to share data, they can do this via the on-chip cache memory of the cores are on the same physical die, but if they run on different CPU dies, they will need to excanhe the data via a significantly slower system bus. □

2.5 Early computer systems did not have the ability to store the program (processing instructions) as "data". Rather, they had to be "hard-coded" using wires. Why is it important to be able to store the program as data? What are the implications if this is not possible?

Solution: To be able to store a program as data is one of the main foundations for modern computer systems. Reprogramming a computer using cables and wires is very time consuming, and is not practically possible with the size and complexity of modern programs. One may imagininging installing an app on a mobile phone requiering the user to reconnect millions of tiny wires using a microscope...! □

Problem 3 – Paging (10%)

Consider 1 KB is 1024 bytes, and assume the page size is

Even ***candidate numbers:*** 4 KB
Odd ***candidate numbers:*** 8 KB

3.1 Write down, in decimal, the *page number* and the *offset* for the address `32543`.

Solution:

	page number	offset
Even	7	3871
Odd	3	7967

□

3.2 Consider a computer system that uses 64-bit logical addresses and has physical memory that is divided into F frames.

(a) What is the logical address space?
Solution: 2^{64} bytes □

(b) What is the size of each frame in the physical memory?
Solution: *Even*: 4 KB; *Odd*: 8 KB □

(c) If the system uses an an *inverted* page table, how many entries does the table have?
Solution: F entries □

Problem 4 – CPU Scheduling (16%)

4.1 In general, a process in a system is in one of these five states: `new`, `ready`, `running`, `waiting`, `terminated`. A *state transition* of a process refers to a process that is moving from one state to another, e.g., a state transition, `ready` → `running` refers to a process switching from `ready` state to `running` state.

Write down the state transition(s) where scheduling decision will happen if the operating system uses *preemptive* scheduling algorithm?

Solution: new → ready (optional)
running → ready
running → terminated
running → waiting
waiting → ready □

4.2 Consider the following table of burst time for processes P_1, P_2, P_3 and P_4 arriving at *time* = 0:

Process	Burst Time
P_1	4
P_2	7
P_3	6
P_4	2

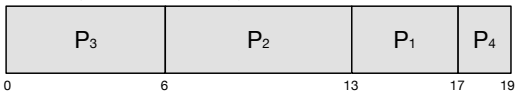
Assume the arrival order is

Even ***candidate numbers:*** P_3, P_2, P_1, P_4
Odd ***candidate numbers:*** P_4, P_1, P_3, P_2

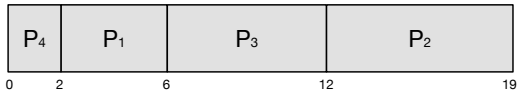
If *First Come First Served* scheduling algorithm is used, what is the average turnaround time? Use a Gantt chart to justify your answer.

Solution:

Even: (6+13+17+19)/4 = 13.75



Odd: (2+6+12+19)/4 = 9.75



□

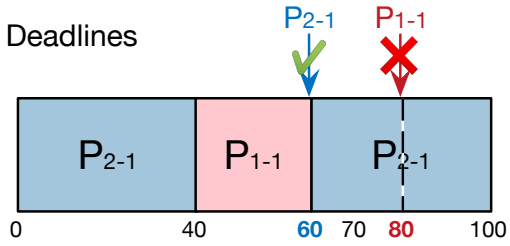
4.3 Consider the following table of processing time and period for two *periodic* processes P_1 and P_2:

Process	Processing time	Period
P_1	30	80
P_2	40	60

The completion deadlines of each process are the beginning of its next period. For example, the completion deadlines of P_1 are at time $80, 160, 240, \ldots$, while the completion deadlines of P_2 are at time $60, 120, 180, 240, \ldots$. Assume scheduling is carried out only at the arrival or completion of processes.

(a) Assume P_2 has higher priority than P_1. Use *Rate-monotonic Scheduling* to schedule the two processes. At *time* $= 280$, can the two processes meet all their deadlines?

Solution: No, P_1 misses its deadline at *time* $= 80$; but P_2 meets its deadlines, at least until *time* $= 100$. Note that P_1 is preempted at *time* $= 60$ because P_2 occurs and has higher priority.



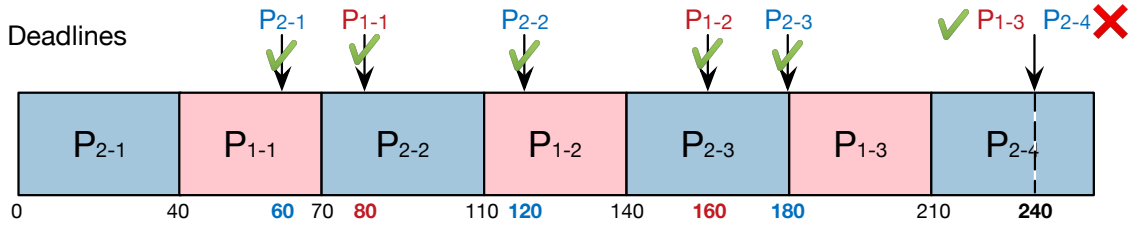
□

(b) Use *Earliest Deadline First* to schedule the two processes. At *time* $= 280$, can the two processes meet all their deadlines?

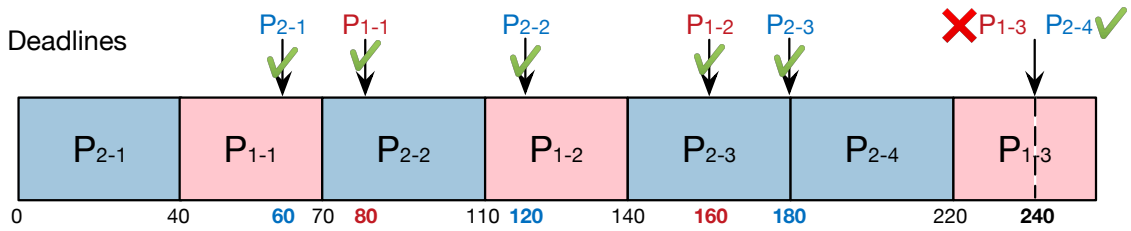
Solution: At *time* $= 180$, either P_1 or P_2 can be scheduled because they have the same deadline, which is at *time* $= 240$. Therefore, there are two possible solutions:

(Either of the solutions is accepted as a correct answer.)

If P_1 is scheduled at *time* $= 180$, P_2 will miss its deadline at *time* $= 240$, as shown in the following gantt chart:



If P_2 is scheduled at *time* $= 180$, P_1 will miss its deadline at *time* $= 240$, as shown in the following gantt chart:



□

Draw a *Gantt chart* of the execution for each of the scheduling algorithms above to justify your answer. The time line of the Gantt charts should go until *time* $= 280$, or until the time when a process does not meet its deadline, i.e., if a process does not meet its deadline at *time* $= 150$, then the time line of the Gantt chart should be until 150.

Problem 5 – Memory Access Time (7%)

5.1 Consider a paging hardware with a translation look-aside buffer (TLB). Assume that the page table and *all the pages* a process requests are in the physical memory. Suppose the percentage of times that the page number of interest is found in the TLB[1] is 85%, each TLB lookup takes 1 ns, and each memory access takes 200 ns. What is the effective access time (EAT)?

Solution: $0.85 \times (1 + 200) + 0.15 \times (1 + 200 + 200) = 231$ ns □

5.2 Consider a process having the page table and **some** of its requested pages in the physical memory. Assume that the process tries to access a page and there is a *page fault*. How many page transfers, i.e., to move pages in to (or out of) the backing store, are required to fetch this page if

(a) there are free frames in the main memory?

Solution: one □

(b) there is no free frame in the main memory, but all frames are clean?

Solution: one □

Problem 6 – Cache (13%)

6.1 Consider B_0, B_2, B_4, B_6 and B_8 are mapped to C_0, and B_1, B_3, B_5, B_7 and B_9 are mapped to C_1. Assume that the following are in the cache:

Even ***candidate numbers*****:** B_3 and B_8

Odd ***candidate numbers*****:** B_2 and B_9

[1] It is also called TLB hit ratio

where B_i refers to the i-th block in the main memory and C_j refers to the j-th cache line. What are the number of cache misses if B_5, B_2, B_8, B_9, B_8 are accessed in the shown order? Justify your answer by identifying which accesses lead to cache misses.

Solution: *Even*: 4; *Odd*: 3

	B_5	B_2	B_8	B_9	B_8
Even	m	m	m	m	h
Odd	m	h	m	m	h

□

6.2 A single-threaded application needs to process a significant amount of data. The data consists of values that should be processed in a number of processing operations (steps), in a similar way, and independently of the other data values. This means that the processing can be performed in two ways (approaches):

- Same *operation* first: first perform the same (single) operation on all values, then move on to the next (single) operation.
- Same *value* first: first perform all operations on the same (single) value, then move on to the next (single) value.

(Instructions that can work on vectors of values are not available.) To decide which approach is more efficient, both approaches have been implemented and tested. The tests reveal that the later method is faster. Based on your knowledge on CPU caches, explain how this can happen.

Solution: Using the first method requires the CPU to constantly switch between different data to process. This means thet there will constantly be cache misses, and data has to be fteched from the (slow) system bus. The later method, on the other hand, will allow the the CPU to process the same data for as long as possible, which will lead to fewer cache misses. Note however that this assumes that the size of the executable code for processing the data is (much) smaller thatn the size of the data to be processed. □

6.3 During the testing mentioned above, it is also noticed a significant speed-up on a CPU that has twice more L1 cache than another CPU that is otherwise identical. What are possible reasons for this? Is there anything you can try to also improve the performance of the application on the CPU with less L1 cache (without modifying the hardware)?

Solution: This difference in processing speed may be becasue in one case, all values and executable code for one set of data will fit into the L1 cache of the CPU, while in the other case it will not, constantly causing cache misses, and data having to be retrieved from the (slow) system bus. It may be possible to optimise the processing code, for example using assembly programming, to make both values and processing code fit into the size of the L1 cache. Perhaps this will not be entiredly possible, but it may at least be possible to reduce the number of cache misses. □

6.4 Between the CPU and the main system memory (RAM), there are often three layers (L1, L2, L3) of cache memory. This memory is decreasingly fast, and increasingly large. Explain the reason for this, and how these different layers interact. Why is it so "complicated" — why not just use a single layer of cache?

Solution: In general, becausue of the way memory is implemented, a larger memory will also be a slower memory. Further, a fast memory is more complicated to implement than a slow memory, hence more expensive. Thus, a price/value trade-off will favour a cache memory with several layers of decrasingly fast and increasingly large memory. □

Problem 7 – Process synchronisation (13%)

7.1 A media player application has two threads. One (the downloader) thread will download the media data from a remote server, and place it into a local buffer with 10 slots. The other (the decoder) thread will read the data from this buffer and decode it. The array `buffer` is placed in shared memory to exchange data, and a normal variable `bufferStatus`, also in shared memory, is used to indicate the status of this buffer. It is initialised to 0.

```
1  // Code used by the downloader //
2  writePos=0;
3  while(true){
4    while(bufferStatus<10){
5      DownloadToBuffer(buffer[writePos]);
6      writePos++;
7      writePos=writePos%10;
8      bufferStatus++;
9    }
10 }
```

Listing 1

```
1  // Code used by the decoder //
2  readPos=0;
3  while(true){
4    while(bufferStatus>0){
5      DecodeFromBuffer(buffer[readPos]);
6      readPos++;
7      readPos=readPos%10;
8      bufferStatus--;
9    }
10 }
```

Listing 2

Most of the time, this media player is working fine. However, in rare cases it crashes with an error message indicating that either of the threads are trying to access the wrong slot of the buffer array. Based on your knowledge on process synchronisation, explain how this can happen.

Solution: There is a small possibility that there will be a context shift from one of the threads to the other just as it is trying to update the `bufferStatus` variable. To be updated, the value of the buffer has to be copied from memory/cache to a CPU register, then incremented or decremented, and finally copied back to cache/memory. If there is a context shift before this seqence of operations has been completed, the other thread may update the now invalid value of the bufferStatus variable, and when the first thread regains control, it will again update the bufferStsuts variable with a value that does not take into consideration the update performed by the second thread. □

7.2 Modify the code in Listings 3 and 4 above to use semaphores to synchronise the threads. The solution should satisfy the requirements for mutual exclusion, progress and bounded waiting.

```
1  wait(int *semaphore){
2    while(*semaphore <= 0) {} ; // busy waiting //
3    *semaphore--;
4  }
5
6  signal(int *semaphore){
7    *semaphore++;
8  }
```

Solution:

```
1  // Shared data structures //
2  semaphore full=0;
3  semaphore mutex=1;
4  semaphore empty=10;
```

```
1  // Code used by the downloader //
2  writePos=0;
3  while(true){
4    wait(empty);
5    wait(mutex);
6    DownloadToBuffer(buffer[writePos]);
7    signal(mutex);
8    signal(full);
9    writePos++;
10   writePos=writePos%10;
11   bufferStatus++;
12 }
```

Listing 3

```
1  // Code used by the decoder //
2  readPos=0;
3  while(true){
4    wait(full);
5    wait(mutex);
6    DecodeFromBuffer(buffer[readPos]);
7    signal(mutex);
8    signal(empty);
9    readPos++;
10   readPos=readPos%10;
11   bufferStatus--;
12 }
```

Listing 4

Note that more efficient implementation is possible. □

7.3 The original solution in Listings 3 and 4 utilises busy waiting. What are the disadvantages with this? Why is it sometimes still the preferable solution? Explain a technique to avoid busy waiting.

Solution: Busy waitng essentially means that CPU time is spent doing "nothing". However, techniques trying to utilise this "wasted" time also come at a price and a overhead. If the waiting time is rather short, this overhead may not be justifiable, and just waiting may be more "profitable". Trying to utilise the waiting time will in general require implementing a special waiting queue where threads may put themselves when they reach a point where they have to wait for some semaphore. The process may also register som conditions that will allow the operating system or some system library to resume them when this condition has been met. Often, such functionality is implemented using monitors. □

7.4 Consider the following approach used by two processes to synchronise access to shared data:

```
1  // Code used by P1 //
2  while(p!=1) {} ; // busy waiting //
3  // critical section //
4  p=2;
5  // non-critical section //
```

```
6  // Code used by P2 //
7  while(p!=2){} ; // Busy waiting //
8  // critical section //
9  p=1;
10 // non-critical section //
```

(a) Does this approach guarantee progress (yes/no)?

(b) Does this approach guarantee mutual exclusion (yes/no)?

Solution:

Progress: no;

mutual exclusion: yes □