Chalmers University of Technology and Gothenburg University

Operating Systems
EDA093, DIT 401

Exam 2026-01-05

Date, Time: Monday 2026/01/05, 08:30-12:30

Course Responsible:
Vincenzo Gulisano (031 772 61 47)

Auxiliary material: You may have with you
- An English-Swedish, Swedish-English dictionary.
- No other books, notes, calculators, etc.

Grade-scale ("Betygsgränser"):
CTH: 3:a 30-39 p, 4:a 40-49 p, 5:a 50-60 p
GU: Godkänd 30-49p, Väl godkänd 50-60 p

Exam review ("Granskningstid"):
Will be announced after the exam.

Instructions

- Write in a **clear manner** and **motivate** (explain, justify) your answers. If it is not clear what is written, your answer will be considered wrong. If it is not explained/justified, even a correct answer will get **significantly** lower (possibly zero) marking.
- Please make an effort to have nice calligraphy!
- If you make **any assumptions** in answering any question, do not forget to clearly state what you assume.
- Answer questions in English, if possible. If you have a large difficulty with that and you think that your grade can be affected, feel free to write in Swedish.

Good luck !!!!

1. (6 p)

   Consider an operating system that supports both processes and threads within a process.

   (a) Explain why creating a *thread* is typically faster than creating a *process*.
   (b) Explain why using separate *processes* instead of threads can provide better isolation.
   (c) Give one very detailed example where threads are preferable, and one very detailed example where processes are preferable, and briefly justify your choices (do not write code/pesudocode).

   [**Answer:**

   (a) Creating a *thread* is typically faster than creating a *process* because a thread is created within an existing address space. The operating system does not need to duplicate memory mappings, page tables, file descriptors, or other process-level resources. Instead, it allocates only a thread control block and a stack. Creating a process, on the other hand, requires setting up a full new execution environment, often involving copying or duplicating large portions of the parent's state.
   (b) Using separate *processes* provides better isolation because each process has its own virtual address space. A fault or memory corruption in one process (e.g., buffer overflow, invalid pointer use) cannot directly corrupt another process. The OS enforces protection boundaries at the process level, whereas threads share memory by default and therefore can interfere with each other—intentionally or accidentally.
   (c) **Example where threads are preferable:** A high-performance web server managing thousands of concurrent connections. Each connection handler performs small computations and shares large read-only data structures. Creating a new thread for each connection minimizes overhead and allows fast sharing of in-memory state without expensive inter-process communication.

       Example where processes are preferable: A browser running multiple tabs, where each tab executes potentially untrusted JavaScript from arbitrary websites. By assigning each tab to its own process, a memory corruption or malicious script in one tab cannot access or modify another tab's memory.

   ]

2. (6 p)

   Consider a demand-paged system using the LRU (Least Recently Used) page replacement algorithm.

   (a) Choose a number of page frames $F > 3$ (you may pick any value, but state it clearly!).
   (b) Construct a reference string such that:
       - The number of *distinct* pages referenced is strictly greater than F, and
       - The number of page faults under LRU is the *minimum possible* for your chosen number of distinct pages.

(c) Explain briefly why, for your reference string, LRU indeed incurs the minimum possible number of page faults.

[**Answer: Example solution (one of many possible):**

Step 1 (choose F): Let us choose $F = 4$ frames.

Step 2 (construct a reference string): Let the distinct pages be $\{1, 2, 3, 4, 5\}$, i.e. 5 distinct pages, which is more than $F = 4$.

A suitable reference string is:

$$1, 2, 3, 4, 5, 2, 3, 4, 5, 2, 3, 4, 5$$

Step 3 (argue minimal faults): We start with empty frames. The first time we access each of the pages $1, 2, 3, 4$ we incur 4 page faults and fill the 4 frames:

$$[1], [1, 2], [1, 2, 3], [1, 2, 3, 4].$$

When we access page 5 for the first time, we get one more fault and LRU evicts the least recently used page among $\{1, 2, 3, 4\}$; at that point, page 1 is the least recently used, so the frames become:

$$[2, 3, 4, 5].$$

From then on, the reference string uses only pages $2, 3, 4, 5$ repeatedly. All of these are already in memory, and under LRU they stay in memory, so there are *no further page faults*.

In total, there are 5 page faults, exactly one for each distinct page in the reference string.

This is the minimum possible, because every distinct page must incur a page fault the first time it is referenced (the frames are initially empty), and we have arranged the reference string so that after the first occurrence of each distinct page, there are no further misses. Therefore, LRU incurs the minimum possible number of faults for this set of distinct pages.]

3. (6 p) List at least *two* scheduling goals that are important for **batch systems** and **two** scheduling goals that are important for **interactive systems**. Explain what each goal means and why it makes sense for one type of system but not as much for the other.

[**Answer: Batch systems.**

- **Throughput.** Throughput measures how many jobs are completed per unit of time. It is essential in batch systems because users typically submit long-running, non-interactive jobs and care about how many total jobs the system can finish. In interactive systems, this metric is less relevant because users care about the responsiveness of *their* task, not aggregate system productivity.
- **Turnaround time.** Turnaround time is the time from job submission to job completion. In batch environments this is a key objective because jobs may run for minutes or hours, and minimizing their total completion time maximizes efficiency and user satisfaction. In interactive systems, completion time is less important since users continuously interact with the system and expect *immediate feedback* rather than fast completion.

Interactive systems.

- **Response time.** Response time is the delay between a user action and the system's reaction. It is critical for interactive workloads (e.g., terminals, GUIs, web servers) where even small delays degrade usability. For batch systems, response time is largely irrelevant because jobs run unattended.
- **Proportionality (or fairness).** Proportionality means the system should respond in a way that roughly matches user expectations: smaller or less demanding tasks should generally feel faster. This is vital for interactive users who quickly perceive unfair slowdowns. In batch systems, fairness is less important because jobs are not user-visible in real time and may legitimately run according to priorities or job size.

]

4. (6 p) For scheduling on a multi-core CPU, would it be a good idea to assign all OS processes to one core and all user processes to the remaining cores? Explain why or why not.

   [**Answer:** Generally, no. Reserving one core exclusively for OS processes and using all other cores only for user processes is not an effective scheduling strategy.

   First, OS activity is usually short and event-driven (interrupt handling, kernel threads, I/O completion), so dedicating an entire core to it often leads to underutilization: the OS core would be idle most of the time while other cores may be overloaded with user work. This reduces overall throughput and increases waiting time.

   Second, isolating OS work on a single core can create a bottleneck: if many I/O events or kernel services occur in a short interval, they all contend for the same core, increasing latency and delaying user-level execution that depends on OS services. Modern kernels benefit from distributing OS activity across cores (e.g., parallel interrupt handling, per-CPU data structures).

   Finally, user processes often need to interact frequently with the kernel (system calls, page faults, scheduling events). Forcing these interactions to migrate to a dedicated core introduces unnecessary cross-core communication, cache misses, and added scheduling overhead.

   In short, dedicating one core to the OS wastes CPU capacity and increases latency. Modern OS schedulers instead allow both OS and user activities to run on all cores, balancing load while respecting affinity, fairness, and cache locality.]

5. (6 p) Explain the difference between a *semaphore* and a *condition variable* for synchronization. Describe when each should be used, and provide one example scenario where a semaphore is appropriate and one where a condition variable is preferable (do not write code).

   [**Answer:** A *semaphore* is a synchronization primitive that maintains an integer counter. It can be used both for *mutual exclusion* (binary semaphores) and for *signaling* between threads. The `wait()` and `signal()` operations affect a numeric value and do not require holding a lock. A thread calling `wait()` may block immediately (if the counter is 0) or proceed (if the counter is positive).

   A *condition variable*, on the other hand, has no counter and cannot be used on its own. It must always be used together with a lock. A thread calling `wait(cond, lock)` blocks until another thread executes `signal(cond)` or `broadcast(cond)`, and blocking always releases the associated lock atomically. Condition variables are used to wait for *state changes*, not to count events.

Semaphores are appropriate when the programmer needs to keep track of a numeric resource, such as the number of free slots in a bounded buffer. Condition variables are preferable when waiting depends on a logical predicate involving shared state, such as waiting for a queue to become non-empty or for a shared variable to reach a specific value. Once signaled, the thread rechecks the predicate under the lock.]

6. (6 p) Consider the Dining Philosophers problem. Each philosopher needs to acquire two chopsticks (left and right) before eating. If every philosopher picks up the left chopstick first and then the right, the system may deadlock.

   (a) Explain why this deadlock occurs (which conditions hold true).

   (b) Show how deadlock can be prevented by *breaking the circular wait* condition.

   (c) Provide clear pseudocode (no actual C code required) for a solution in which the circular wait is broken by imposing an order on resource acquisition.

[**Answer: 1. Why deadlock occurs.** The needed four conditions hold at the same time: mutual exclusion, hold and wait, no preemption, and circular wait.

2. Preventing deadlock by breaking circular wait. By breaking the circular wait, you can show that when at least one philosopher who manages to get a chopstick will not compete with another philosopher trying to get the other chopstick. This is show in Slides 21 and 22 of the respective lesson.

3. Pseudocode solution.

Pseudocode for philosophers 0 **to** $N-2$**:**

```
loop forever:
  think()
  lock(fork[$i$])   // left fork first
  lock(fork[($i+1$) mod $N$])   // then right fork
  eat()
  unlock(fork[($i+1$) mod $N$])
  unlock(fork[$i$])
```

Pseudocode for philosopher $N-1$ **(reversed order):**

```
loop forever:
  think()
  lock(fork[($i+1$) mod $N$])   // right fork first
  lock(fork[$i$])   // then left fork
  eat()
  unlock(fork[($i+1$) mod $N$])
  unlock(fork[$i$])
```

]

7. (6 points) Would it be possible, in principle, to design an operating system that runs *entirely* from secondary storage (e.g., disk), without using any main memory, caches, or CPU registers to store instructions or data during execution? Justify your answer in detail.

   [**Answer:** Yes, in principle such a system could be designed, because nothing in the abstract definition of an operating system requires the existence of fast volatile memory. One could imagine a CPU whose architecture allows every instruction fetch and every read/write operation to be performed directly on persistent storage.

   However, such a design would be extremely inefficient. Every instruction execution, data access, context switch, and system call would require disk I/O, which is orders of magnitude slower than memory access. Since disk access latency is measured in microseconds or milliseconds rather than nanoseconds, even a simple context switch would become prohibitively expensive, making the system effectively unusable for interactive workloads or multiprogramming. The OS would spend nearly all its time waiting for disk operations to complete.

   Thus, while not theoretically impossible, an OS that executes entirely from disk would be so slow that it is impractical for any general-purpose computing. This is why real systems rely on fast volatile memory (registers, caches, RAM) to keep the performance of instruction execution and context switching within reasonable bounds.]

8. (6 p) A file system stores files on disk using fixed-size blocks. Assume the disk is used to store the following files:

   - 10 files of size 1 KB each,
   - 5 files of size 3 KB each,
   - 2 files of size 5 KB each.

   Assume that reading from disk has the following cost model:

   - Each block read has a fixed overhead of 1 ms (seek/rotation, etc.).
   - Additionally, it takes 0.5 ms per KB of data contained in the block (transfer time).

   Consider three possible block sizes: 1 KB, 2 KB, and 4 KB.

   (a) For each block size, compute:
      i. the total number of blocks used to store all the files,
      ii. the total amount of disk space allocated,
      iii. the amount of wasted space.

   (b) For each block size, compute the total time to read *all* files.

   (c) Based on your numbers, which block size wastes the least disk space?

   Please provide your answer as a table, columns for the various thing to compute, rows for the various block sizes.

   [**Answer:**

Block size	Total blocks	Allocated space	Wasted space	Total time
1	35	35	0	52.5
2	26	52	17	52
4	19	76	41	57

]

9. (6 p) Explain how the Unified Extensible Firmware Interface (UEFI) works during the boot process. In your answer, describe at least:

   - the role of the EFI System Partition (ESP),
   - how UEFI loads an operating system,
   - how UEFI differs from legacy BIOS in terms of architecture and extensibility.

   [**Answer:** UEFI is a modern firmware interface that replaces legacy BIOS. When the system powers on, UEFI firmware initializes hardware and then loads boot applications from the *EFI System Partition* (ESP), which is a FAT-formatted disk partition containing EFI executables (*.efi files). Each operating system places its own boot loader (e.g., `grubx64.efi`, `shimx64.efi`, or `bootmgfw.efi`) inside the ESP.

   The UEFI firmware maintains a set of boot entries in non-volatile memory that specify which EFI application to load. The firmware reads these entries, locates the corresponding *.efi file on the ESP, and directly executes it. The OS loader then takes over, initializes the kernel, loads necessary drivers, and transitions the machine into the OS runtime environment.

   Unlike BIOS, which works through interrupt-driven real-mode code and loads only the first sector (MBR) of a disk, UEFI operates in 32- or 64-bit mode, supports larger disks (GPT), and provides a modular, extensible architecture. Additional drivers and applications can be stored on the ESP and loaded dynamically. UEFI also supports secure boot, network booting via built-in drivers, and does not rely on the fixed memory locations or real-mode constraints of BIOS. This makes UEFI more flexible, more secure, and suitable for modern hardware.]

10. (6 p) Explain the main differences between virtualization using *virtual machines (VMs)* and using *containers*. Your answer should describe how they isolate applications, how they use hardware resources, and the implications for performance and portability.

    [**Answer:** Virtual machines provide virtualization by emulating a full hardware platform. Each VM contains its own guest operating system, kernel, system libraries, and applications. This provides strong isolation, since faults or compromises inside one VM do not affect the host or other VMs. However, running a full OS per VM increases resource usage and results in slower startup times and larger memory footprints.

    Containers, instead, rely on OS-level virtualization: they share the host's kernel and isolate processes using namespaces and cgroups. Each container includes only the application and its user-level dependencies, not a separate kernel. This results in very fast startup, low memory overhead, and high density of deployable units. Isolation is weaker than VMs because all containers rely on the same kernel; a kernel-level vulnerability may break containment.

    In terms of performance, containers generally run closer to native speed due to the lack of hardware emulation and reduced overhead. VMs provide stronger security and portability across different operating-system kernels, while containers provide lighter-weight and more efficient execution on a shared OS kernel.]