Chalmers University of Technology and Gothenburg University

Operating Systems
EDA093, DIT 401

Exam 2025-10-28

Date, Time: Tuesday 2025/10/28, 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

- Do not forget to write your personal number, if you are a GU or CTH student and at which program ("linje").
- Start answering each assignment on a new page; number the pages and use only one side of each sheet of paper.
- 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)

   Two friends, A and B, are discussing Lamport's Bakey Algorithm (see below).

Algorithm 1 Lamport's Bakery Algorithm for process $i \in \{0, \ldots, N-1\}$

```
1: Shared: choosing[0..N − 1] : bool ← false
2: Shared: number[0..N − 1] : nonneg int ← 0
3: function LOCK(i)                                          ▷ enter protocol for process i
4:     choosing[i] ← true
5:     number[i] ← 1 + max{number[0], . . . , number[N − 1]}
6:     choosing[i] ← false
7:     for j ← 0 to N − 1 do
8:         if j ≠ i then
9:             while choosing[j] do                          ▷ wait until j picked a ticket
10:            end while
11:            while number[j] ≠ 0 and (number[j], j) < (number[i], i) do   ▷ lexicographic order
12:            end while
13:        end if
14:    end for
15: end function
16: function UNLOCK(i)                                       ▷ exit protocol for process i
17:    number[i] ← 0
18: end function
```

   A claims the check on number[j] being different from 0 in line 11 is not strictly necessary. B disagrees. Who is right and why? If you think B is right, which guarantee would be compromised and why?

   [**Answer:** B is right. If the check is not there, a process wanting to enter its critical section might wait - possibly forever - for a process that is not trying to access its critical section. This breaks the progress guarantee.]

2. (10 p)

   A round–robin scheduler with two priority classes schedules I/O-bound processes first (high priority) and CPU-bound processes second (low priority).

   There are 3 I/O-bound processes. Each one, when scheduled, performs CPU work for only 0.2 ms and then issues an I/O request.

   There are 4 CPU-bound processes that always use the full quantum.

   The context-switch overhead is $t_{cs} = 0.01$ ms per switch (0.01 in total, for saving the context of the pervious process and load that of the following one).

   Determine the minimum quantum q (in milliseconds) such that the total CPU time lost to context-switch overhead is less than 1% of total CPU time.

   [**Answer:** Let $N_{\text{IO}} = 3$ be the number of I/O-bound processes and $N_{\text{CPU}} = 4$ be the number of CPU-bound processes.

   Per scheduling cycle. Each cycle runs all high-priority I/O processes first, then the low-priority CPU processes.

   - Each I/O process runs for only 0.2 ms.
   - Each CPU process runs for the full quantum q.

   Total *useful CPU time* per cycle:

$$T_{\text{useful}} = N_{\text{IO}} \cdot 0.2 + N_{\text{CPU}} \cdot q = 3 \cdot 0.2 + 4q = 0.6 + 4q \quad \text{ms.}$$

   Total number of context switches per cycle = total number of process slices $= N_{\text{IO}} + N_{\text{CPU}} = 7$ (we switch after each slice).

Context-switch overhead per cycle:

$$T_{\text{over}} = 7\, t_{cs} = 7 \cdot 0.01 = 0.07 \text{ ms}.$$

Fraction of time lost:

$$f = \frac{T_{\text{over}}}{T_{\text{useful}} + T_{\text{over}}}.$$

We want $f < 0.01$:

$$\frac{0.07}{0.6 + 4q + 0.07} < 0.01.$$

Solve for q:

$$0.07 < 0.01(0.6{+}4q{+}0.07) \quad \Rightarrow \quad 0.07 < 0.0067{+}0.04q \quad \Rightarrow \quad 0.0633 < 0.04q \quad \Rightarrow \quad q > 1.58\,\text{ms}.$$

Result. To keep overhead below 1%, choose a quantum:

$$\boxed{q \;\geq\; 1.6\,\text{ms (approximately)}}.$$

]

3. (10 p)

Describe in detail how a *buffer overflow* attack works. Your answer must:

(a) show a clear understanding of the relevant **process memory layout** (which areas of memory are involved and why);

(b) discuss the **implications** if part of the attack is executed in **kernel mode** (is kernel-mode execution necessary? what additional consequences arise if kernel code is subverted?);

(c) describe in detail a compiler-based **software hardening** measure that mitigates buffer-overflow style attacks.

[**Answer:** A buffer overflow occurs when a program writes more data into a fixed-size memory region than it was allocated, allowing the excess bytes to overwrite adjacent memory. On typical processes this matters because local buffers live in the stack frame next to control data (saved registers, frame pointer, return address); overwriting the return address lets an attacker redirect execution when the function returns. Depending on protections, the attacker may inject and run code on the stack or, if execution is prevented there, use return-oriented techniques that reuse existing code snippets. Kernel-mode execution is not required for many attacks—user-mode compromise can already let an attacker abuse that process's privileges—but if an overflow can be escalated into the kernel (via a vulnerable syscall path or driver) the impact is far greater because kernel compromise breaks system isolation and enables persistent, powerful attacks. Defenses combine safer coding with compiler/OS hardening: stack canaries to detect overwrites. A random value (the canary) is placed between a function's local buffers and its control data on the stack. Before the function returns the compiler-generated code checks the canary; if it has changed, the program aborts. This detects many buffer overwrites that would otherwise corrupt the return address.]

4. (3 points)

   Below are three Access Control Lists (ACLs). Each ACL lists subjects (users or groups) with their rights: R = read, W = write, X = execute.

   Simplify each ACL keeping only the effective rights.

   ACL 1: `A:RWX, B:W, B:RWX, C:R, G1:RX, G2:RW, G1:R, O:R`

   ACL 2: `ROOT:RW, BACKUP:R, BACKUP:RWX, G1:RWX, G2:RWX, O:RWX`

   ACL 3: `D:RW, B:R, B:RW, C:RW, E:R, G1:R, G2:RW, O:-`

   Assume B and C are members of G2.

   [**Answer:**

   Simplified ACL 1: `A:RWX, B:W, C:R, G1:RW, G2:RW, O:R`

   Simplified ACL 2: `ROOT:RW, BACKUP:R, G1:RWX, G2:RWX, O:RWX`

   Simplified ACL 3: `D:RW, B:R, E:R, G1:R, G2:RW` Explanation: C is in G2 with the same rights, so their explicit entries can be merged into the group entry.

   ]

5. (8 points)

   The examiner believes the following sentences are true. State clearly plausible assumptions the examiner is making to claim each sentence is true.

   (a) A computer with one CPU that has one core can run a program concurrently.

   (b) A computer with one CPU that has one core can run a process concurrently.

   (c) A computer with one CPU that has multiple cores can run a program concurrently.

   (d) A computer with one CPU that has multiple cores can run a process concurrently.

   (e) Two threads from one process can have the program counter pointing to the same instruction and can be executing the related instruction at the same exact time.

   (f) Two threads from two processes can have the program counter pointing to the same instruction.

   (g) Two threads from one process can have the program counter pointing to the same physical memory address and can be executing the related instruction at the same exact time.

   (h) Two threads from two processes can have the program counter pointing to the same physical memory address.

   [**Answer:**

   (a) **True under the assumption** that a program consists of multiple independent processes, which can be scheduled *concurrently* (i.e., with overlapping progress in time) even on a single-core CPU by time-sharing. It is also true if we interpret the statement as referring to this program and another distinct program running concurrently.

(b) **True under the assumption** that a process is multithreaded, so its threads can be executed *concurrently* (time-sliced) on a single-core CPU. It is also true if we interpret the statement as referring to this process and another independent process running concurrently.

(c) **Same reasoning as (1):** multiple cores do not change the conceptual definition of concurrency (overlapping progress).

(d) **Same reasoning as (2):** on a multicore architecture, multiple threads of one process (or multiple processes) can execute concurrently.

(e) **True under the assumption** of a multicore architecture with kernel-level threads: both threads belong to the same process, share the same code segment, and each has its own program counter that can point to the same instruction, executed simultaneously by two cores.

(f) **True under the assumption** that the two processes are copies of the same program (e.g., created via `fork()`), so their virtual address spaces contain the same code, and their program counters can point to the same instruction (though at different virtual addresses or mapped to the same physical code).

(g) **Same as (5):** in a multicore system, two threads of the same process can execute the same instruction at the same physical address concurrently, since they share the same code section mapped to that address.

(h) **True under the assumption** that the two processes are using *copy-on-write* (COW) memory after a `fork()`, so their code pages are still shared and mapped to the same physical memory addresses.

]

6. (5 points)

A disk has size D bytes and is divided into fixed-size blocks of B bytes. Two free-space management schemes are considered:

- **Bitmap:** one bit per block indicates whether the block is free (1) or allocated (0).
- **Linked list of free blocks:** each free block is recorded by storing a pointer (address) of size A bytes to the next free block (singly linked).

Let F be the number of free blocks at some moment. For which values of F does the linked-list representation use strictly less space than the bitmap?

[**Answer:**

The bitmap uses

$$\text{Bitmap bytes} \;=\; \frac{D/B}{8} \;=\; \frac{D}{8B}\,.$$

The singly linked list uses one pointer per free block:

$$\text{Linked-list bytes} \;=\; F \cdot A\,.$$

The linked list is smaller iff

$$F \cdot A \;<\; \frac{D}{8B}$$

Thus, for

$$\boxed{F \;<\; \frac{D}{8AB}}$$

]

7. (6 p) The Linux `man sched_setaffinity(2)` page includes the following statement:

> "A process's CPU affinity mask is restricted by the cpuset in which it resides. If the mask specified in a `sched_setaffinity()` call contains CPUs that are not in that cpuset, those bits are silently ignored (i.e., masked out)."

Explain what this means in practice. Your answer should mention both the concept of CPU affinity and what "*the cpuset in which it resides*" means.

[**Answer:** This means that the CPUs a process or thread can run on are determined by the **intersection** of two sets: the CPUs allowed by the process's own **affinity mask**, and the CPUs allowed by the **cpuset** of the cgroup to which the process belongs.

The cpuset acts as a **hard restriction** imposed by the system or container runtime. If a process tries to pin itself to CPUs outside its cpuset (using `sched_setaffinity()`), those CPUs are ignored and the process can run only on CPUs common to both sets. If there is no overlap, the process becomes unrunnable until one of the masks is changed.]

8. (6 p) Consider a system using **lottery scheduling**. There are n processes, labeled $P_1, P_2, \ldots, P_n$. The scheduler assigns to each process a number of lottery tickets, determining its probability of being chosen for execution.

Distribute the tickets so that each process P_i has **double the chance** of being scheduled compared to process P_{i-1}. Express both the number of tickets per process and the total number of tickets in the system as functions of n (just a reminder: $\sum_{k=0}^{n-1} ar^k = a\,\frac{r^n-1}{r-1}$).

[**Answer:** Let the number of tickets of P_1 be t. Then:

$$T_i = t \times 2^{i-1}$$

The total number of tickets is the geometric series:

$$T_{\text{total}} = t(2^n - 1)$$

Therefore, the probability that process P_i is selected is:

$$P(P_i) = \frac{2^{i-1}}{2^n - 1}$$

For example, if $t = 1$ and $n = 4$, the distribution is:

$$P_1 : 1,\ P_2 : 2,\ P_3 : 4,\ P_4 : 8$$

and the probabilities are 1/15, 2/15, 4/15, and 8/15, respectively.]

9. (6 p) Explain the difference between **Memory-Mapped I/O (MMIO)** and **Direct Memory Access (DMA)**. Describe how each interacts with the CPU and the system memory.

[**Answer:** In **Memory-Mapped I/O**, device control registers are mapped to specific memory addresses. The CPU communicates directly with I/O devices using regular load/store instructions to these addresses. Every data transfer involves the CPU, which explicitly moves data between device registers and memory.

In **Direct Memory Access (DMA)**, a dedicated DMA controller performs data transfers between devices and memory without continuous CPU intervention. The CPU initiates the transfer by programming the DMA controller, which then handles the entire operation and interrupts the CPU only when the transfer is complete.

In short: MMIO involves the CPU in every data movement, while DMA allows data transfer to occur independently of the CPU, reducing CPU overhead and improving performance.]