Chalmers University of Technology and Gothenburg University

Operating Systems
EDA093, DIT401

Exam 2019-10-26

Date, Time, Place: Saturday 2019/10/26, 08.30-12.30, "Maskin"-salar

Course Responsible:
Vincenzo Gulisano (031 772 61 47),
Marina Papatriantafilou (031 772 54 13)

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.
- If you make **any assumptions** in answering any item, do not forget to clearly state what you assume.
- The exam is organized in groups of questions. The credit for each group of questions is mentioned in the beginning of the respective group. Unless otherwise stated, all questions in a group have equal weight.
- Answer questions in English, if possible. If you have large difficulty with that and you think that your grade can be affected, feel free to write in Swedish.

1. (12 p)

   (a) (4 p) Assume you rely on priority scheduling with multiple queues in an interactive system. All processes start with the highest priority. If they use up all the quanta allocated to them, they are moved down one class and their quanta are doubled. What would be the benefit of such a scheme?

   [**HINT:** I/O-bound processes have high priority (most likely they will finish sooner) while CPU-bound have larger quanta]

   (b) (4 p). Explain why having one OS running independently in each core of a multi-core CPU can lead to data inconsistencies. Do **not** make examples.

   [**HINT:** Because of buffers and caches that would not be consistent among OSes]

   (c) (4 p). Prove that the Shortest Job First (non-preemptive) is optimal to minimize turnaround time.

   [**HINT:** Check the proof sketch found at the beginning of page 158.]

2. (12 p)

   (a) (4 p) Describe at least 2 benefits of running several user-level threads mapped to the same kernel thread (many to one).

   [**HINT:** Faster context-switches and user-defined scheduling, for instance.]

   (b) (4p) Write a C program that can be used to parallelize the search of an element inside a non-sorted array. Do **not** use `phtread_create` nor `phtread_join`.

   [**HINT:** Some process that uses fork and then parent and child search each half of the array, for instance.]

   (c) (4p) Suppose you have a process composed of 3 threads. Make an example about a possible execution of such threads over time that shows the threads run both concurrently and in parallel.

   [**HINT:** The minimal example is to have 2 cores and the 3 threads running concurrently (in a round-robin fashion) on the 2 cores. Many other examples can work, of course.]

3. (12 p)

   (a) (4 p) What are the trade-offs (pros vs cons) associated with the PageSize parameter?

   [**HINT:** A large pagesize increases the amount of wasted space due to internal fragmentation (half a page is wasted by process on average) and may lead to more page faults (fewer frames entail lower freedom with page replacement), however overhead is reduced (smaller page table, less TLB misses as more data are addressable with the same number of pages, etc).]

   (b) (4 p) Consider the following reference string 0 1 2 3 0 1 4 0 1 2 3 4 of memory page accesses, and a physical memory of 3 frames that is initially empty. How many page faults will occur using (i) Fifo, (ii) Optimal, (iii) Least Recently Used (LRU) page replacement algorithm? In which of the previously cited algorithms (i, ii and iii), the number of page faults can increase if more frames are available (*Bélády's Anomaly*)?

   [**HINT:** (i) 9 page faults (ii) 7 page faults (iii) 10 page faults.

FIFO (in the example, 10 page faults with 4 frames) whereas with LRU/Optimal, it cannot increase (in the example, 6 PF for Optimal and 8 PF for LRU).]

(c) (4 p) Let us consider a CPU with 4 cores, 32K of L1 cache, 256K of L2 cache, 4M of L3 cache, and a Translation Lookaside Buffer (TLB) with 1536 entries. Physical memory is 1GB, Virtual memory is 4GB, and PageSize is 4K. Explain briefly your answer to the following questions:

i. How many entries has the page table of each process?

ii. How many different pages can be accessed before a TLB miss occurs?

iii. In our example, is it possible that the code and data corresponding to the working set of one process fits in the L2 cache but the system still experience TLB thrashing?

[**HINT:** (i) 2^{32} virtual memory / 2^{12} page size = 2^{20} = 1 M entries. Amir: While 1M is the maximum number of entries, some students have pointed out that the answer to this part of the question is process dependent.
(ii) 1536 different pages, one for each TLB entry.
(iii) yes, as code/data are cached by cache lines (typically 64 bytes) and not full pages so if the code/data of the working set are spread over many different pages (here more than 1536), TLB thrashing will be experienced.]

4. (12 p)

(a) (4 p) Is "defragmentation" a good idea on a Solid-State-Disk (SSD)? Explain why.

[**HINT:** Defragmentation of a file system aims to put physically closer pieces of files that have been spread over by the file system. That way, sequential read is improved on mechanical drives (traditional hard disks) as reading sequentially the data is faster when it is contiguous on the disk (hence fewer slow arm movements). However, SSDs enjoy, similarly to RAM, random access to any blocks so defragmentation becomes worthless from a performance perspective. Even worse, since SSD have limited number of writes per cell, defragmentation makes your drive wears faster and is only useful in very limited cases (eg in windows, for the system not to reach the maximum fragmentation limit).]

(b) (4 p) Let us consider a file system with 10000 files and where the distribution of file sizes on the file system is as follows:

Size in bytes	**Proportion**
0	10%
1 - 512	20%
513 - 4096	20%
4097 - 16KiB	40%
16KiB - 1GiB	10%

What would be the maximum amount of wasted space on disk in internal fragmentation when using a I/O Block size of 1024 bytes?

[**HINT:** At maximum, 1023 bytes are wasted per file (almost the entire last block except 1 byte). Only empty files do not follow that rule as nothing is stored on disk (albeit eg i-nodes), so we end up with 90% x 10000 x 1023 = 9000x1023 = (without calculator, 9x23 = 10x23-23 = 230-23 = 207 so) 9207000 bytes ≈ 9.2 MB ≈ 8.82 MiB.]

(c) (4 p)

i. We execute the following commands in a shell

```
user:machine$ touch bob
user:machine$ chmod 000 bob
user:machine$ ls -al
total 8
drwxrwxr-x 2 user groupA 4096 Oct 18 17:23 .
drwxrwxr-x 3 user groupA 4096 Oct 18 17:22 ..
---------- 1 user groupA    0 Oct 18 17:23 bob
user:machine$ rm bob
```

What is the result of executing the last command?

ii. How would you reverse `sudo chmod 000 \`which chmod\``? And what about `sudo chmod -R 000 /`?

[**HINT:** (i) The file is deleted. File permissions that have been checked are those of the parent directory where user has indeed a right to write. (ii) The shell program is unavailable and we need to use a direct OS system call (eg in C) to change its rights back to normal 755. In the second situation, all files are unreadable/writable rendering the full system unusable, extra-OS means are necessary, as mounting the damage partition from another OS/live USB-stick and fixing the rights back.]

5. (12 p)

(a) (4 p) (i) Describe the safety, progress and fairness requirements from a solution to the critical section problem.
(ii) Explain how it is possible to solve the critical problem for n threads, using binary semaphores. Describe the solution using pseudocode.
(iii) Argue about the properties of the solution that you describe with respect to the requirements.

[**HINT:** (i) Safety or mutual exclusion (S/ME): only one thread executes in its critical section (CS) at a time. Progress (P): if one or more threads need to enter their CS and no other thread is executing in its CS, one of the competing threads will be able to do so in bounded time. Fairness (F): each tread that needs to enter its CS can do so after finite waiting (a stronger fairness requirement can be FCFS).
(ii) A simple solution uses a binary semaphore `entry`, initialized to 1; to enter and exit the critical section each thread invokes `wait(enter)` and `signal(entry)` respectively.

```
shared var binary semaphore access, initialized to 1;
thread-i {
repeat
wait(access) ;
// ...  critical section ...
signal(access) ;
// other operations
forever
}
```

(iii) S/ME is guaranteed since after a thread t has returned from the invocation of the `wait` (ie t enters the CS, until `signal` is invoked (ie t exits the CS), any thread invoking the `wait` must wait. P is guaranteed since when one or more threads invoke `wait`, one invocation will return, hence that thread will enter the CS. F depends on the fairness of the semaphore implementation; if the latter is done through a blocking queue, then the F property is FCFS; if it is done through busy-wait/spinning, F depends

on the system scheduler and the hardware accessing the memory, ie under unlucky scenarios some thread(s) might suffer unbounded waiting.]

(b) (8 p) Consider the readers-writers critical section (CS) problem: a shared resource, say a file, is accessible by several concurrent threads for reading or writing. Because the readers do not change the file, any number of them may be granted simultaneous read access. On the other hand, only one writer is allowed to write on the file at a time and the write should not be concurrent with any read access. Consider the following proposal for solving the problem:

```
shared var general semaphore access initialized to m;

reader-thread {
    repeat
        wait(access) ;
        // ... read-CS ...
        signal(access) ;
        // other operations
    forever
}

writer-thread {
    for k = 1 ... m do wait(access) end-for;
    // ... write-CS ...
   for k = 1 ... m do signal (access) end-for;
   // ... other operations ...
}
```

(i) Describe the safety, progress and fairness requirements from a solution to the problem.
(ii) Argue about the proposed solution with respect to the requirements, under the assumption that there are m reader threads and 1 writer thread that may need access to the file.
(iii) Argue about the proposed solution with respect to the requirements, under the assumption that there are m reader threads and $n > 1$ writer threads that may need access to the file.
[**HINT:** (i) Safety: (S1) write and read access is mutually exclusive; (S2) write accesses are mutually exclusive. Progress (P) if one or more threads need to enter their CS and no other thread is executing in its CS, one of the competing threads will be able to do so in bounded time. Fairness (F): each tread that needs to enter its CS can do so after finite waiting (a stronger fairness requirement can be FCFS, ie no by-passing).
(ii) Viewing each "unit" in the `access` semaphore as a *token* (i.e. a mutually exclusive, reusable resource), we observe that the proposed method requires each reader to obtain one such token to proceed and read and the writer to obtain all of them to proceed and write.
S1: following the observation, if the writer is writing, it must have gotten all the tokens, hence no reader can enter before the writer exits; similarly, if one or more reader threads are reading, the writer cannot start writing since it will have to wait for the tokens occupied by those readers, ie at least until they finish reading. S2: trivial, since there is only one writer thread.
P: If only readers compete, there are enough tokens for all of them to enter their read-CS. If the writer needs to access its write-CS and there

is no competing read-thread, there are enough tokens for it to proceed. If the writer is competing with readers, the writer can enter its write-CS if it succeeds to obtain all the tokens first. If the writer has fewer than m tokens, it will need to wait until readers release them, while readers can concurrently enter their read-CSs in the meanwhile.
F: depends on the fairness of the semaphore implementation; if the latter is done through a blocking queue, then the F property is FCFS; if it is done through busy-wait/spinning, F depends on the system scheduler and the hardware accessing the memory, ie under unlucky scenarios some thread(s) might suffer unbounded waiting.
(iii) There is possibility for deadlock; consider two (or more) writers having obtained one or more tokens each. Each one will hold them and try to get the rest (ie the "hold-and-wait", and "no-preemption" conditions hold). Since there are m tokens in total and each writer needs all m of them to proceed to the write-CS, there is implied circular waiting, hence the deadlock, i.e. P is violated.]