Department of Computer Science
University of Copenhagen

First Take-Home Re-Exam in
Operating Systems and Concurrent Programming

Deadline: **26 June 2016 23:59**

Version: 1; June 20 2016

Preamble

This is the exam set for the first *individual*, take-home re-exam in the course Operating Systems and Concurrent Programming, B3-2016. The second re-exam will be held in end-August 2016.

This document consists of 8 pages; make sure you have them all. Read the rest of this preamble carefully. The exam set consists of 2 practical tasks and 3 theoretical questions. Your submission will be graded as a whole, on the 7-point grading scale, with an external examiner.

Exam Policy

This is a take-home, open-book exam. You are allowed to use all the material made available during the course, without further citation. Any other sources must be cited appropriately. If you proceed according to some pre-existing solution, you must properly cite it and argue why you think it is correct.

The exam is **100% individual**. You are not allowed to discuss the exam set with anyone else, until the exam is over. It is expressly forbidden:

- To discuss, or share any part of this exam, including, but not limited to, partial solutions, with anyone else.
- To help, or receive help from others.
- To seek inspiration from other sources, including the Internet, without proper citation.
- To post any questions or answers related to the exam on *any fora*, before the exam is over. Piazza is disabled for the duration of the exam.

Breaches of this policy will be handled in accordance with our disciplinary procedures. Possible consequences range from your work being considered `void`, to expulsion from the university[1].

[1] `http://uddannelseskvalitet.ku.dk/docs/Ordensregler-010914.pdf`.

Errors and Ambiguities

In the event of errors or ambiguities in the exam text, you are expected to state your assumptions as to the intended meaning in your report. Some ambiguities may be intentional.

You may request for clarifications by sending an email to our internal mailing list, <osm16@dikumail.dk>, but do not expect an immediate reply. Important clarifications and/or corrections will be posted on the course bulletin board on Absalon. If there is no time to resolve a case, you should proceed according your chosen, documented interpretation.

What to Submit

To pass the exam, you must submit *both* your source code, *and* a report. Your report should *both* document your solution for the practical tasks, *and* answer the theoretical questions.

Your report forms the basis for our grading. Your source code will primarily only serve to confirm what you state in your report. You should:

- Include *all* the source code you touched in the appendices of your report.
- Give an overview of what you've changed in the handed out `KUDOS`.
- Test your code, and explain how we can reproduce your test results.
- Comment your source code so it is easy to understand.

In addition, we state the following formatting and reporting requirements:

- The report (excluding appendices) should be around 5-10 pages. Document your solutions, reflections, and assumptions, if any. Document and justify your design decisions. The report must be a printable, PDF document. State your KU ID on the front page.
- Separate the practical and theoretical parts by a page break. High-resolution scanned versions (no photos, please) of handwritten solutions are acceptable for the theoretical part.
- Please leave reasonable margins for printing and marking, in either case.
- Package your source code in a `.tar.gz` archive, archiving *just* a `src` directory (no copies of the report, please). As with the G-assignments, we hand out a `Makefile` to make this easier for you.

Make sure to follow the formatting requirements (PDF, `.tar.gz`, `kudos`, etc.). If you don't, your work might be considered **`void`**.

Submission

Please submit via Digital Exam (`https://eksamen.ku.dk/`).

Submission via Digital Exam is *not possible at the time of writing*, but should become available during the week. Details about how to submit will be published accordingly. If that fails, you *can* submit via Absalon. If all else fails, submit via e-mail to the course coordinator directly (see below).

Please check your submission after you submit.

Emergency Line

In case of emergencies, the course coordinator is your primary point of contact during the exam: Eric Jul <ericjul@ericjul.dk>. For very urgent matters, send an SMS to Eric at +45 40251650.

Please respect that our TAs might have exams of their own during the week.

Theoretical Part

T1 Critical Intersection (15%)

Consider modelling an n-way road intersection, where $n \geq 2$:

1. For simplicity, assume that vehicles are not allowed to turn, but only drive straight through the intersection.
2. To keep things fair, at most 5 vehicles are allowed to drive through the intersection, before the light changes and vehicles in another direction (if any) can proceed.

Your task is to write the pseudocode for the procedures `init()`, `intersection()`, and `vehicle(i)` using *only* semaphores for synchronization. `init()` and `intersection()` take no arguments, while `vehicle(i)` takes a unique road identifier as argument. You can use `i` as an offset into an array of n roads.

You are guaranteed that `init()` will be called before any calls to `intersection()` or `vehicle(i)`. `intersection()` will serve as the body of one thread, and `vehicle(i)` as the body of some $m \geq 0$ threads.

To ensure a common, machine-readable pseudocode syntax, we've set up a Blockly[2] page with semaphores:

`http://onlineta.github.io/CompSys/osm16-1st-reexam/T1.html`

You can use this interface to design pseudocode and use the "export" feature to export to XML. Submit your XML in a machine-readable format, i.e. submit a file `T1.xml` alongside your KUDOS code and report. It is also recommended to use this export/import feature to save your work along the way — our web interface does not attempt to save your work.

If you do not like using Blockly (let us know!) you can use KUDOS, Python[3], or a semaphore implementation of your own choosing. That is, you should submit either a Blockly XML file or a working demo implementation in a real programming language.

T2 Scheduling (10%)

Given a set of jobs with arrival and execution times in seconds as given below:

1. arrives at time 0, execution time is 100 s
2. arrives at time 1, execution time is 10 s
3. arrives at time 2, execution time is 20 s
4. arrives at time 3, execution time is 1 s
5. arrives at time 10, execution time is 1 s
6. arrives at time 20, execution time is 100 s

[2] See also `https://developers.google.com/blockly/`. (This is not relevant to the exam.)
[3] `https://docs.python.org/2/library/threading.html#semaphore-objects`.

Calcuate **turnaround time** and the **response time** for each job. Furthermore calculate the average **turnaround time** and the average **response time** for each of the following scheduling algorithms when scheduling the above jobs:

1. FIFO
2. SJF
3. STCF
4. RR with a time slice of 1 s

Assume that context switching and other scheduling overhead is so small that it can be ignored.

T3 FAT-based file systems (25%)

On the course home page on Absalon, you will find Chapter 12 of the [Dinosaur Book][4]. There you will find a description of the file allocation table (FAT) scheme (see Section 12.4.2 and Figure 12.7) used originally in MS-DOS, and today popular on small external devices such as USB-sticks.

You should demonstrate understanding of FAT-based file systems.

You are to allocate blocks for some files in a FAT-formatted volume.

Assume that the block size is 4096 bytes and that a pointer is 4 bytes. When asked to show the contents of the FAT, draw a figure as in Figure 12.7, having a floating directory entry for each file allocated. Always show all the files currently maintained in the FAT.

You are to allocate the blocks sequentially starting with a block number that is identical to the last three digits of your KU ID. For example, if your KU ID is abc123, the first block allocated is the one numbered 123.

You are to allocate the files in the order given below, i.e., a new file allocation should continue the sequence of block-allocations from the last.

a) Allocate a file F_1 of size 5,000 bytes. Show the FAT table.

b) Allocate a file F_2 of size 20,000 bytes. Show the FAT table.

c) Extend F_1 with an additional 5,000 bytes. Show the FAT table.

d) Shorten F_2 to 10,000 bytes. Show the FAT table.

e) Allocate a file F_3 of size 20,000 bytes. Show the FAT table.

f) Assuming one block for the FAT, how much space (in bytes) can we manage using the FAT in total?

g) Using FAT can result in a significant number of disk seeks, unless the FAT is cached. What are some problems with caching the FAT? Contrast this with caching the directory control block.

[4] `https://absalon.itslearning.com/File/fs_folderfile.aspx?FolderFileID=3402542`.

Practical Part

The practical part consists of two programming tasks extending the handed out version of KUDOS: A reference solution to this year's G1 and G2, coupled with implementations of threads, mutexes, and condition variables. You will not be needing this latter functionality in the first task.

The tasks are intentionally *independent*. It is *okay* to submit a `.tar.gz` archive with two subdirectories: `src/p1/kudos/` and `src/p2/kudos/`. If you get both to work, we prefer `src/kudos/`. If you are having a hard time getting either to work, we recommend to implement dummy system calls, and to explain your attempt in your report. Please implement the system calls either way.

P1 Monitors in KUDOS (25%)

Monitors are classically a programming language primitive. Such primitives lend themselves to implementation in C using a level of indirection.

A monitor is a collection of functions, where at most one thread may be within a given function. Threads queue up to enter the monitor. To enable communication between the threads using the monitor, monitors also come with condition variables. This means that there may be several ways in which a thread enters a monitor: by calling a monitor function, or by waiting on a monitor condition variable.

Your task is to implement a userland monitor type `mon_t`, and the function:

```
int mon_init(mon_t *mon);
int mon_call(mon_t *mon, void (*fun)(void*), void* arg);
```

A thread may call `mon_call` with an arbitrary function `fun` and argument `arg`, to queue up for access to `mon`. Furthermore, declare a userland monitor condition variable type `mon_cond_t`, and the following userland functions:

```
int mon_cond_init(mon_t *mon, mon_cond_t *cond);
void mon_cond_wait(mon_cond_t *cond);
void mon_cond_signal(mon_cond_t *cond);
```

`mon_cond_wait` and `mon_cond_signal` should fail if the thread is not currently inside a monitor function.

Illustrate that your monitor implementation (with condition variables) works as intended by writing a multithreaded KUDOS userland program.

Discuss how the threads (using a mixture of `mon_call`, `mon_cond_wait`, and `mon_cond_signal`) get scheduled to enter the monitor. It is *not* part of your task to design a monitor with fair scheduling: use the synchronization primitives already present in KUDOS, and discuss their shortcomings, if any.

P2 Bunny Allocation (25%)

The present KUDOS userland library uses a simple heap memory allocation scheme: It implements `malloc(size_t size)` and `free(void *)` by maintaining a list of free blocks, sorted by increasing memory address. The free blocks are occupied on a first-fit basis. To avoid internal fragmentation, the first-fit block is split into a best-fit block and a remaining (free) block, if possible.

1. Explain when the current implementation coalesces the free blocks.
2. Give an example of how this scheme suffers from external fragmentation.

This scheme implies that, in the worst case, you have to iterate through the entire free list to find a first-fit (or best-fit) free block. A more *generational* approach is to have a range of free lists, one for each particular size. To keep the number of free lists low, let's choose an infamous and somewhat fast-growing sequence—the Fibonacci sequence starting with byte-sizes $fib(0) = 4$ and $fib(1) = 8$, followed by $fib(2) = 12$, $fib(3) = 20$, $fib(4) = 32$, etc. We call this scheme "bunny allocation", and the sizes 4, 8, 12, etc. "bunny numbers".

Bunny allocation proceeds as follows:

1. To allocate n bytes, find the first m such that $fib(m) \geq n$. For instance, for $n = 30$, we have $m = 4$ because $fib(3) = 20$ and $fib(4) = 32$.
2. Check if the free list for m is empty. If not, use a free block from this list.
3. If it is empty, check the free list for $m + 1$:
   (a) If there is a free block in this list, split this block into two blocks, of size $fib(m)$ and $fib(m - 1)$, respectively. Allocate the block of size $fib(m)$, and add the other block to the free list for $m - 1$.
   (b) If the free list for $m + 1$ is also empty, proceed to $m + 2$, $m + 3$, etc. in a similar fashion, recursively splitting the blocks once an available block is found.

Initially, all free lists are empty except for m, where $fib(m)$ is the largest bunny number that can fit in the heap. The current implementation uses a fixed-size, statically allocated heap of 256 bytes. It is beyond the scope of this task to make the heap grow dynamically, but you should explain how you would grow your heap when you run out of sufficiently large free blocks.

Modify the procedures `malloc(size_t size)` and `free(void *)` in `userland/lib.c` to use the bunny allocation algorithm described above. Your report should (among other things) answer the following questions:

1. What are some advantages/disadvantages of bunny allocation?
2. What is the average-case run-time complexity of your `malloc` and `free`?
3. What changes do you need to make to your data structures to increase the size of the heap, if the size of the heap could increase on-demand?

References

[Hoare (1974)] C. A. R. Hoare. *Monitors: an operating system structuring concept*. Commun. ACM 17(10), pp. 549–557. ACM, 1974.

[Dinosaur Book] Abraham Silberschatz, Peter B. Galvin, and Greg Gagne. *Operating System Concepts*, 9th ed. Wiley, 2014.