Department of Computer Science
University of Copenhagen

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

Deadline: **August 26, 2016 16:00**

Version: 1; August 22, 2016

Preamble

This is the exam set for the secon *individual*, take-home re-exam in the course Operating Systems and Concurrent Programming, B3-2016. The last re-exam will be held between Block 1 and Block 2, study-year 2016/2017.

This document consists of 7 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 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.
- 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.

Submission

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

Theoretical Part

T1 Merge-Semaphores (14%)

We can define a "merge-semaphore" as a regular semaphore, where the P operation has been replaced by two operations, P1 and P2, that both work like the usual P operation with the modification that a process calling P1 will block not only if the semaphore is non-positive but also if the latest process that passed the semaphore also called P1, and, conversely, that a process calling P2 will block not only if the semaphore is non-positive but also if the latest process that passed the semaphore also called P2. Thus processes that pass the semaphore will alternately have called P1 and P2.

Write an implementation of merge-semaphores (i.e. declare `merge_sem_t`, and define `merege_sem_init`, `merege_sem_P1`, and `merege_sem_P2`) using regular semaphores as a basis. Please don't re-implement regular semaphores. You can either provide pseudocode or an implementation in your favourite programming language which already supports semaphores. If you provide pseudocode, follow the notation of Chapter 31 in OSTEP[2].

If you are in doubt about the exact semantics of merge-semaphore, you should merely provide your interpretation of the semantics. For example, in the above, the initial state of a merge-semaphore is not really mentioned.

T2 Simulating MLFQ (18%)

Use your favourite programming language to implement a program `mlfq` to simulate the behaviour of a 5-level feedback queue scheduler for a single-core processor. The levels numbered 1 (top-most), 2, 3, 4, and 5 (bottom-most), and time is measured in ticks. The scheduler must adhere to the following rules:

Rule 1 If Level(A) < Level(B), A runs (B doesn't).

Rule 2 If Level(A) = Level(B), A & B run in RR.
The time-slice on level i is $i \times$ "your exam number" number of ticks.

Rule 3 When a job enters the system, it is placed in queue 1.

Rule 4 Once a job uses up its time-slice in the RR-scheme (see Rule 2), it is moved down one level in the queue, unless the job finished, or it is already on the bottom-most queue.

Rule 5 After a time period of 1000 ticks, move all the jobs in the system to the topmost queue.

These rules are similar, but not identical to the rules in chapter 8 of OSTEP[3].

Tasks are specified via standard input, one line at a time, with the `scanf`-format `"%s %u %u"`, with a task identifier first, the arrival time second, and the execution time third. You can assume that the task identifiers are always fresh (have not occurred previously), and that tasks are ordered by arrival time.

[2] `http://pages.cs.wisc.edu/~remzi/OSTEP/threads-sema.pdf`.
[3] `http://pages.cs.wisc.edu/~remzi/OSTEP/cpu-sched-mlfq.pdf`.

```
a 1 4
b 2 1
c 3 3
```

You should output a sequence of task identifiers and run-times, separated by whitespace. The sequence indicates the order in which the tasks would execute and for how long. For instance, if we assume our exam number is 1, for the above input, we would like the following output:

```
a1 b1 c1 a2 c2 a1
```

T3 Simulating Buddy Allocation (18%)

Use your favourite programming language to implement a program `buddy` which simulates the servicing of memory requests using a buddy memory allocation system[4]. The program is to service requests from a total of 64 × "your exam number" KB of overall system memory (where 1KB is 1024 bytes).

Your program should handle both allocation and deallocation requests from standard input. An allocation request has the `scanf`-format `"%s = malloc %u"`, while a deallocation request has the format `"free %s"`.

You can assume that variable names in allocation requests are fresh (have not occurred previously), and that only previously `malloc`'ed variables are ever `free`'d. Note however, that not all `malloc` requests are guaranteed succeed. `free`'ing a variable that failed to get `malloc`'ed has no effect.

For instance, the following constitutes a valid input sequence:

```
a = malloc 7000
b = malloc 7000
c = malloc 7000
d = malloc 7000
e = malloc 20000
free b
free d
f = malloc 9000
g = malloc 9000
```

The simulator should allocate 8192 bytes for `a`, `b`, `c`, and `d`, wasting 1192 bytes for each. Then, it should allocate 32768 bytes for `e`, wasting 12768 bytes.

Assuming that our exam number is 1, this fills up the memory. As `b` and `d` are freed, two non-coalescing blocks of size 8192 become available. Neither is sufficient for a 9000-byte block. So allocations of `f` and `g` fail.

For the above input, we would like the following output:

```
f = malloc 9000 failed, with 16384 bytes available overall.
g = malloc 9000 failed, with 16384 bytes available overall.
49152/65536 bytes allocated, with 15152 bytes wasted.
```

That is, first report on the failed requests, then report on the resulting memory occupation and waste.

[4] `http://pages.cs.wisc.edu/~remzi/OSTEP/vm-freespace.pdf`.

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.

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 Thread-Safe `malloc` and `free` (25%)

The current KUDOS userland library implements `malloc` and `free` in a thread-unsafe manner: If two threads call either of these functions simultaneously, the behaviour is *undefined*.

Solve this in as fine-grained a fashion as possible: Allow as many threads as possible, to simultaneously call, and *progress* through `malloc` and `free`.

Write tests to show that memory allocation still works as intended. The handed out KUDOS contains many of the test programs handed out throughout the course. In-how-far do these test the functionality of your `malloc` and `free`?

Write tests to exhibit the level of multi-programming that your solution allows, i.e., show in-how-far your synchronization mechanism is *fine-grained*.

Suggested workflow:

1. Start with a *coarse-grained* (i.e., global lock) implementation.
2. Test thoroughly, write more tests.
3. Do some work on P2 before attempting a fine-grained solution.

P2 Buffered I/O (25%)

The current KUDOS userland library implements a range of I/O functions by building upon the functions `getc` and `putc`. These functions perform a system call for each character read/written. System calls are expensive, so real-world C libraries implement buffered I/O, where a system call is only occasionally issued to fill a userland buffer, and library functions like `getc` and `putc` read/write a buffer instead. Implement buffered I/O in the KUDOS userland library:

1. Consider changing the definition of the type `FILE` in `userland/lib.h`. (See below for its intended use.)
2. Declare a function `FILE *fopen(const char *pathname)`, which uses the underlying `syscall_open(const char *pathname)` to open a file descriptor, and allocate a buffer for I/O. The buffer must be of at least 32 bytes.
3. Change `fgetc` and `fputc` to use the buffer instead.
4. Declare a function `int fflush(FILE *stream)`, for flushing the buffer of the given stream to the underlying file.

5. Declare a function `int fclose(FILE *stream)`, for flushing the buffer to the underlying file, and closing the underlying file descriptor.

Write tests to show that I/O functions work as intended.
Write tests that showcase that buffering is actually going on.

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.