Department of Computer Science
University of Copenhagen

Take-Home Exam in
Operating Systems and Concurrent Programming

Deadline: **8 April 2016 16:00**

Version: 1; 4 April 2016

Preamble

This is the exam set for the *individual*, take-home exam in the course Operating Systems and Concurrent Programming, B3-2016. This document consists of 11 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 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 Dining with Monitors (10%)

We present an attempt at a monitor-based solution to the "Dining Philosophers" problem. Our pseudo-code notation closely resembles that of [Hoare (1974)].

A philosopher i can dine more than once, and every time she does, she is expected to use the following API:

```
dining philosophers.pickup(i);
comment // Eat, eat, and eat until I die!
dining philosophers.putdown(i);
```

a) In-how-far this is a solution to the "Dining Philosophers" problem?

b) Describe how it is still possible for some philosophers to starve to death. Give an example.

c) Modify the pseudo-code to fix the starvation problem, ensuring fair service. Our LaTeX source code is provided alongside this document. Argue that your solution solves the problem.

```
dining philosophers: monitor
  begin state:(thinking,hungry,eating);
        phil: array 0..N − 1 of state;
        self: array 0..N − 1 of condition;
        LEFT: array 0..N − 1 of int;
        RIGHT: array 0..N − 1 of int;
        procedure pickup(i : int);
          begin phil[i] := hungry;
                test(i);
                if (phil[i] ≠ eating) then self[i].wait;
          end pickup;
        procedure putdown(i : int);
          begin phil[i] := thinking;
                test(LEFT[i]);
                test(RIGHT[i]);
          end putdown;
        procedure test(i : int);
          begin if ((phil[LEFT[i]] ≠ eating) & (phil[RIGHT[i]] ≠ eating) &
                    (phil[i] = hungry)) then
                  begin phil[i] := eating;
                        self[i].signal;
                  end
          end test;
        for i := 0 step 1 until N − 1 do state[i] := thinking;
        for i := 0 step 1 until N − 1 do LEFT[i] := (i + 1) mod N;
        for i := 0 step 1 until N − 1 do RIGHT[i] := (i − 1) mod N;
  end dining philosophers
```

T2 Pair-Up (15%)

Consider a synchronization mechanism called "pair-up": when a thread calls pair-up, it waits for another thread to call pair-up, whereafter both continue.

a) Write the pseudo-code for a pair-up procedure using semaphores. You are allowed to assume that there are only two threads that call pair-up.

b) Does your implementation work, if more than two threads call pair-up? Argue why, or why not, e.g., using examples.

T3 UNIX-based filesystems (25%)

On the course home page on Absalon, you will find Chapter 12 of the [Dinosaur Book][2]. There you will find a description of the combined index block scheme used in UNIX-based filesystems (see Section 12.4.3 and Figure 12.9).

You should demonstrate understanding of the UNIX way of representing files using inodes and, specifically, the combined scheme. In the following, you may ignore the entries of the inode that are above, and including the "size block count". Do not list these fields when asked to show the contents of an inode.

Assume that there are 15 pointers in an inode, the last three being to a single indirect block, a double indirect block, and a triple indirect block, respectively. Assume that the block size is 4096 bytes and that a pointer is 4 bytes.

You are to allocate blocks for some files as given below.

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.

When asked to show a pointer, merely show its block number. When asked to show an inode, show its contents and make a drawing similar to Figure 12.9. The figure should include the actual block numbers.

a) Allocate a file of size 5,000 bytes. Show the contents of its inode.

b) Allocate a file of size 49,156 bytes. Show the contents of its inode.

c) Allocate a file of size N. You are to choose N, so that the last 4 bytes of the file (and no other bytes of the file) are accessed via the double indirect block.

   i) Show the contents of the inode and the first and last pointer of the block that the single indirect pointer points to.

   ii) Show the contents of the double indirect block.

   iii) How many disk blocks does this file use in total (excluding the space for the inode).

d) Calculate the maximum size of a file that does NOT require a triple indirect block. Justify your answer.

[2] `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 Synchronous Message Passing (25%)

KUDOS is a modern operating systems kernel with support for a high amount of concurrency and parallel execution across multiple processors. However, there is presently no way for processes to communicate, apart from the file system, which is an inefficient and brittle mechanism.

In this task, you will be implementing a simple message passing system that allows processes to send messages to each other. The general idea is that there is a number of *mailboxes* available on the system, that processes can opt in to and use to communicate.

a) System calls

There is a finite number of slots for mailboxes in the system. A process can allocate a mailbox via an appropriate system call, in which it also provides an integer as a unique identifier for the mailbox. There may not already exist a mailbox with this identifier. This is done with the following system call:

```
int syscall_msg_mailbox(int mid)
```

The parameter must be non-negative. Returns zero on success. After this call, a mailbox with the mailbox identifier `mid` will be associated with the calling process. The system call may fail for the following reasons:

- There is no available slot for a new mailbox (similar to running out of room in the process table).
- Another mailbox with the desired identifier already exists.

After succesfully allocating a mailbox, the process can then receive messages with another system call:

```
int syscall_msg_receive(int mid, void *dest)
```

A call `msg_receive(mid, dest)` blocks until a message is sent to the mailbox identified by `mid` (see below), at which point the message is written to `dest` and the system call returns zero. The system call may fail for the following reasons:

- The mailbox identified by `mid` was not allocated by the calling process.
- There is no mailbox with the identifier `mid`.

A message is sent to a mailbox with the following system call:

```
int syscall_msg_send(int mid, size_t size, void *src)
```

The call `msg_send(mid, size, src)` sends a message consisting of `n` bytes read from `src` to the mailbox identified by `mid`. The call blocks until the message has been delivered (i.e. until the corresponding call to `msg_receive()` in the receiver returns). If several concurrent processes all try to send to the same mailbox, messages are delivered in FIFO order. The system call may fail for the following reasons:

- There is no mailbox with the identifier `mid`.
- The `size` argument exceeds the maximum message size (see below).

The following constants are used for fundamental limits and error reporting in the message passing subsystem:

```
/* The maximum size of a message */
#define MSG_MAX_SIZE  (1024)

/* Error codes */
#define MSG_INVALID_MAILBOX         (1)
#define MSG_NOT_YOUR_MAILBOX        (2)
#define MSG_TOO_BIG                 (3)
#define MSG_NO_MORE_MAILBOX_SLOTS   (4)
#define MSG_MAILBOX_ALREADY_EXISTS  (5)
```

They should be returned by the system calls as appropriate.

Note that there is no way to *close* a mailbox. This means that mailbox slots are "leaked" whenever a process with owned mailboxes exits. You are not expected to handle this problem.

b) Implementation hints

The mailbox table should be a table of static size, much like the thread table, process table, and semaphore table. However, in this case, the mailbox identifier is in general *not* equal to its location in the mailbox table. You are recommended to define a utility function:

```
/* Return index of the mailbox with the given ID, or -1 if there is no
   such mailbox. */
int mailbox_by_id(int mid);
```

The mailbox table is a shared resource, and should be appropriately protected from race conditions.

When a process attempts to send a message to a mailbox, there are two situations to consider:

1. Either the owning process of the mailbox is already waiting for a message...
2. ...or it isn't.

Ensure correct behaviour in both of these cases. If several processes are concurrently attempting to send messages to the same mailbox, ensure that they do not trample over each other once the owner of the mailbox calls `syscall_msg_receive()`.

The actual delivery of the message requires copying data from the sender into the memory space of the receiver. This is done with the following function, which is found in `vm/memory.h`:

```
/* Write 'buflen' bytes from 'source' to the memory space
   indicated by the given page table, starting at address
   'target'. */
void vm_memwrite(pagetable_t *pagetable, unsigned int buflen,
                 virtaddr_t target, const void *source);
```

For example, if `tid` is a known thread ID and `buf` is some local buffer, we can write `1337` bytes to address `1024` of the adress space of `tid` as follows:

```
vm_memwrite(thread_get_thread_entry(tid)->pagetable,
            1337, 1024, buf);
```

c) Testing

A test suite constituting of the program `msgtest`, `tmanager`, and `tclient` has been provided as part of the exam handout. This test suite uses the message passing API to perform a kind of race-free multiplexing of the terminal, where processes can print entire lines atomically.

Is the provided suite an acceptable and covering test of the message passing facility? If yes, provide an analysis explaining how. If no, explain why not, and provide test programs that exercise the remainder.

In either case, it is recommended that you write smaller test programs to help you with the implementation.

P2 Read-Write locks(25%)

The handed out KUDOS supports user-level threads: You can have up to 16 threads per process. The handed out KUDOS is not based on a reference solution of G4: TLB exceptions are not handled. Having more than 16 threads per process (the number of entries in the YAMS TLB) would result in *undefined behaviour*. Solving this (again) is not part of the exam.

To spin off a thread, we provide a `syscall_thread`, which takes a pointer to the function to execute in the new thread, and an argument to pass to that function. The thread is run "immediately", i.e., there is no separate "thread run" system call. Every thread must call `syscall_exit` upon exit. The last `syscall_exit` kills off the process and sets the process exit code. The process exit code is hereby unpredictable. Solving this is not part of the exam.

To make the resulting level of concurrency manageable, we've also added mutexes and condition variables.

Mutexes (mutual exclusion locks) are controlled using the following library functions, having the (hopefully) obvious semantics. `mutex_t` (like the following functions) is defined if you `#include "lib.h"` in your userland program:

```
void syscall_mutex_init(mutex_t *mutex);
void syscall_mutex_lock(mutex_t *mutex);
void syscall_mutex_unlock(mutex_t *mutex);
```

Condition variables are controlled in a similar fashion, using the following library functions:

```
void syscall_cond_wait(cond_t *cond, mutex_t *mutex);
void syscall_cond_signal(cond_t *cond);
void syscall_cond_broadcast(cond_t *cond);
```

The `mutex` is assumed to be *held* when you call `syscall_cond_wait`, and *will be held* when it returns. Your thread will be put on the sleep queue until some other thread "signals" or "broadcasts" the "condition". `syscall_cond_signal` will awake one waiting thread, whereas `syscall_cond_broadcast` will awake all waiting threads. There is *no guarantee* that any given condition holds when your thread is awoken, but you do hold the `mutex`, and *something* has happened.

If in doubt, see `kudos/proc/mutex.c` and `kudos/proc/cond.c`, as well as the handed out userland programs `initthr` and, in particular, `threads`. (`initthr` wraps the `threads` program to make sure to halt the system after use.)

Your task is to use available kernel and/or userland primitives to implement support for *read-write locks* in KUDOS userland. A read-write lock supports multiple concurrent readers, but *exclusive* writers. You should implement them in `rwlocks.[hc]`, which already declares the following API:

`rwlock_t rwlock_create()` — Creates a new *read-write lock*.

Returns: A handle to a new *read-write lock*.

`int rwlock_rlock_acquire(rwlock_t rwlock)` — Obtain the *read-lock*.

Returns: Zero on success and non-zero on error.

`int rwlock_wlock_acquire(rwlock_t rwlock)` — Obtain the *write-lock*.

Returns: Zero on success and non-zero on error.

`int rwlock_rlock_release(rwlock_t rwlock)` — Release the *read-lock*.

Returns: Zero on success and non-zero on error.

`int rwlock_wlock_release(rwlock_t rwlock)` — Release the *write-lock*.

Returns: Zero on success and non-zero on error.

`int rwlock_destroy(rwlock_t rwlock)` — Destroy the *read-write lock*.

Returns: Zero on success and non-zero on error.

Read-write locks are like spinlocks with the following exceptions:

1. Multiple threads can hold a *read-lock* simultaneously.
2. All attempts to obtain a *write-lock* should block, while one or more threads holds a read-lock.
3. Only one thread can hold a *write-lock* at any given time.

4. All attempts to obtain a *read-lock* or a *write-lock* should block, while a *write-lock* is held by a thread.

One use of *read-write locks* is to synchronize data structure access (possibly backed by a file) different threads, where multiple threads are allowed to read the data structure but only one thread is allowed to write to the data structure, and only when nobody is reading from it.

Consider the situation where multiple threads rapidly are obtaining a *read-lock*, and a single thread is blocked in the attempt to obtain a *write-lock*. Depending on your implementation strategy, this is a likely source for *starvation*. Discuss in your report which implementation strategies one can use to accommodate this problem. You are allowed to extend the library call `rwlock_create()` to take an additional argument that chooses among a range of possible starvation-avoidance strategies.

Write a test program (`rwl`) to illustrate that your implementation works.

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.