UNIVERSITY OF COPENHAGEN
Department of Computer Science
Version 1.0, 23 March 2015

Final Exam: "Operating Systems and Concurrent Programming", 2015

Exam Period: Monday, 23 March 2015 at 9:00 – Friday, 27 March 2015 at 14:00

Handing in: via Absalon

Contents

In this document we lay down the rules and the questions for the take-home exam of the course "Operating Systems and Concurrent Programming". There are four practical tasks and three theoretical exercises. Together with this document we release an archive that contains:

(1) two concurrent-queue implementations, and

(2) a specific version of `BUENOS` that supports TTY interaction, user processes, user semaphores, TLB handling, and file I/O, as described in [`BUENOS` Roadmap], and as implemented in the G assignments. It also contains an implementation of `syscall_fork`, which you will need in one of the practical assignments.

Your solutions to the practical tasks must be based on these.

Grading

The practical tasks are weighted in total 60%, and theoretical exercises in all 40% of your grade. Your answers will be assessed using the 7-step grading scale with an external examiner. The deadline for our grading is 17 April 2015 and the results will be announced soon after that. As to grading, any news will be announced on the course homepage.

Emergency line

If, for some reason or another, you have to or want to contact the course team—during the exam or after the exam, the course coordinator is your point of contact: Jyrki Katajainen; e-mail: `<jyrki@di.ku.dk>`; telephone: (+45) 35335680.

Official languages

In your answers, you can use Danish, English, Swedish, and Norwegian.

Rules

The exam is ***individual***.

The intention is, like in any other written exam, that you complete the answers *100% individually*. This is an open-book exam and you are welcome to make use of the course textbooks and other reading material from the course. Any other sources must be cited appropriately.

Any errors or ambiguities in the problem formulations, or the handed-out source code, are considered to be part of the exam. Just state clearly the assumptions and/or corrections you had to make in order to solve the issue(s).

To be more specific, you are not permitted

- to discuss or share any part of this exam, including but not limited to solutions, with any other student. Both helping and receiving help are expressly forbidden.
- to copy an answer from the Internet without giving the source.
- to post any questions or answers related to the exam on our Discussion Forum in Absalon, or any other fora, before the exam is over.

This examination format is based on trust. Do not disappoint us (and betray yourself)! *Breaches of the above-mentioned rules will be handled in accordance with our disciplinary procedures* and render the exam `void`.

Files to be handed in

Please submit your solution electronically via Absalon in two files:

1. a *report* (in `pdf` format) with your name and KU user ID on the front page;
2. an *archive* (in either `zip` or `tar.gz` format) that contains a directory `benchmark/`, containing your solution to P1, and a directory `buenos/`, containing a working `BUENOS` source tree, as your solution to tasks P2, P3, and P4.

Your report should consist of two parts: a theoretical part and a practical part, *separated by a page break*. The assessment will be based on your *report*. Your source code will be used to *verify* what you state in your report. If you do any testing (as you should), explain how we can reproduce your results. NB: You must include *all* modified and added code in the appendices of your report for documentation purposes. Remember to comment your programs so they are easy to understand. In particular, it is important to document the changes you made to the handed-out version of `BUENOS`.

The practical part of your report should document which observations you made, and which assumptions your solution depends upon (either those which you chose yourself or those which were made by the designers of `BUENOS`). Document and justify the design decisions you have made, and reflect on your solution to each task.

High-resolution scanned versions of handwritten solutions are acceptable for the theoretical exercises; please leave reasonable margins for printing and marking.

Practical Part

P1 Benchmarking: Thread-safe queues (10%)

In this task, you should evaluate the effectiveness of different thread-safe implementations of the same data structure, and adequately measure and compare their performance. Your test programs should (clearly) be multi-threaded.

In G3, you implemented a thread-safe queue data structure. Alongside this text, we provide two competitors to your solution:

1. `cg_queue.[hc]` that uses a single lock (***coarse-grained locking***) and
2. `fg_queue.[hc]` that uses two locks (***fine-grained locking***).

Measure and compare the performance of these implementations for suitable workloads. Use Pthreads for the implementation of your benchmarks. You can use the timing facilities provided by the `<time.h>` library, in particular the `clock()` function. You should keep in mind that `clock()` issues a system call to get the system time. A system call may be too expensive at ***particular*** points in your benchmarking code, skewing your results.

In your report, justify your benchmarking strategy and tell how we can reproduce your results. Keep in mind that benchmarking results may vary depending on the operating system, hardware, and system load. Can you draw any interesting conclusions based on your experiments?

P2 System calls: A source of entropy (5%)

In this task, you should extend an operating-system kernel with a new service and provide a simple test of your implementation.

Add a new system call to `BUENOS` which enables a userland process to use the operating system as a source of entropy. *Hint:* `BUENOS` already provides a `_get_rand(uint32_t range)` function as part of `lib/libc.h`, which returns a pseudo-random number between 0 and `range` − 1.

`uint32_t syscall_rand(uint32_t range)`

Effects: Advances a pseudo-random number seed in the kernel (perhaps).

Returns: A pseudo-random number between 0 and `range` − 1.

P3 Process communication: Unidirectional pipes (30%)

In this task, you should extend an operating-system kernel with a new feature, as well as make changes to existing parts of the system in order to expose this new feature. You must also adequately test your implementation.

A ***pipe*** is a unidirectional data channel that can be used for interprocess communication. The pipes in this task are inspired by the pipes of the POSIX standard; for more information, see the manual page for `pipe(2)`.

A pipe is only useful if multiple processes can use the same pipe. To that end, it must be possible for a parent process to pass on a pipe to its child. For this task, we supply a `fork` syscall in the `BUENOS` handout, which "splits" a process in two, and runs each part as its own process, duplicating all memory of the main process (thereby also duplicating the pipes). It too is inspired by the POSIX standard.

`int syscall_fork()`

Effects: Create a new process that is a duplicate of the current process. The new child process continues from the same program counter as the parent process, but in a new memory space.

Returns: In the child process, it returns 0. In the parent process, it returns the process id of the new child process. If an error occured, a child process is not created, and the parent returns a negative integer.

a) Implement system call `syscall_pipe` (5%)

Implement unidirectional pipes in `BUENOS`:

`int syscall_pipe(int fds[2])`

Effects: The array `fds` is used to return two file descriptors referring to the ends of the pipe. The file descriptor in `fds[0]` refers to the read end of the pipe and `fds[1]` refers to the write end.

Returns: 0 upon success, otherwise -1.

b) Modify existing system calls (10%)

To orchestrate the kernel to operate with pipes, the following changes to existing system calls are necessary:

- If a read (`syscall_read`) is performed on an empty pipe, the call should block until data is available.
- If a write (`syscall_write`) is performed on a full pipe, the call should block until data is removed from the pipe.
- Both `syscall_read` and `syscall_write` should fail if called on a closed pipe.

- You should extend the existing syscall `syscall_close` to work on file descriptors referring to unidirectional pipes as well. When there are no longer file descriptors referring to a pipe, the resources used by the pipe should be reclaimed by the kernel. (Hint: Allocate space for a fixed number of pipes, and for each mark whether or not it is in use.)
- If a process has open pipes and calls `syscall_fork` or `syscall_exec`, then the child process should inherit the file descriptors referring to the ends of each pipe. For instance, if a process has redirected file descriptor 1 (standard out) to a user-created pipe, and then spawns a child process, then the child process should also have file descriptor 1 point to that pipe.

c) Implement additional system support (10%)

Furthermore, implement the following system calls:

`int syscall_dup(int oldfd, int newfd)`

Effects: Copies the file descriptor `oldfd` to `newfd` and closes `oldfd`. If `newfd` is open upon calling `syscall_dup`, it should be closed before `oldfd` is copied. This functionality must not be limited to pipes.

The call fails if `oldfd` is not a valid file descriptor.

Returns: 0 upon success, otherwise -1.

`int syscall_select(int nfds, int fds[])`

Effects: This syscall should block until one or more file descriptors in the `fds` array are ready to be read from/written to. `nfds` refers to the number of file descriptors in `fds`. It is an error to call `syscall_select` with a file descriptor which does not refer to a unidirectional pipe.

Returns: Upon success the file descriptor which became ready (if more than one file descriptor is ready, an arbitrary one is chosen), otherwise -1.

d) Test the implemented functions (5%)

The extended kernel should be able to handle the following example. Program `pipe1` (Listing 1) should be blocked on `syscall_read` until program `pipe2` (Listing 2) writes to the pipe. Then program `pipe1` should be able to read `"Hello world!\n"` from the pipe and print it to the terminal.

You must also design a consumer program that forks multiple times, with a distinct pipe for each new process, and uses `syscall_select` to retrieve data from all subprocesses until they stop producing data.

Listing 1: `tests/pipe1.c`

```
int main() {
  int ifds[2];
  int ofds[2];
  int efds[2];
  char buf[256];
  (void) syscall_pipe(ifds);
  (void) syscall_pipe(ofds);
  (void) syscall_pipe(efds);
  pid_t pid = syscall_fork();
  if (pid == 0) {
    syscall_dup(ifds[0], 0);
    syscall_dup(ofds[1], 1);
    syscall_dup(efds[1], 2);
    syscall_exec("[disk]pipe2");
  }
  else {
    int n = syscall_read(ofds[0], buf, sizeof(buf));
    (void) syscall_join(pid);
    syscall_write(stdout, buf, n);
  }
  return 0;
}
```

Listing 2: `tests/pipe2.c`

```
int main() {
  char hello[] = "Hello␣world!\n";
  syscall_write(stdout, hello, sizeof(hello) - 1);
  return 0;
}
```

P4 Process management: Lethal injection (15%)

In this task, you should extend an operating-system kernel with the ability to forcefully terminate running processes. Forceful process termination should halt process execution as soon as possible, and relinquish any system resources held by the process, while leaving the operating system in a consistent state.

Implement a `BUENOS` system call which enables a userland process to kill the userland process identified by the given process ID (for instance, itself).

`int syscall_kill(pid_t pid, int retval)`

Effects: Forcefully terminate the userland process with the given `pid`. The effect is the same as forcing the identified process to call `syscall_exit(retval)`. Every process is allowed to kill any other process, including itself.

Returns: 0 upon success, a negative value on error. It is acceptable for the system call to return before the killed process is actually terminated, although it must be guaranteed to terminate in the (near) future.

Throughout this course, we have only considered BUENOS as running on a single processor core. We stick to this choice in this task. This means that at any point in time, at most one kernel thread is in a running state. Recall that BUENOS implements a one-to-one multi-threading model, so there is a one-to-one correspondence between a process and a kernel thread. If the running process executes a system call, it is safe to modify kernel data structures, including the global thread table, for (merely) as long as thread switching is disabled.

There are some pitfalls to avoid. If the about to be killed thread was scheduled out while in kernel mode, it means that it was in the middle of a system call. Naïvely finishing a thread in the middle of a system call may leave the operating system in an inconsistent state. If instead, the thread was in user mode, it is safe to modify the context stored in its thread control block to force the thread to commit suicide as soon as it is scheduled in by the scheduler. See also § 4.3.3, page 25 and § 6.4.1, page 42 of [BUENOS Roadmap]. For instance, you could set the userland program counter to a `syscall` machine instruction.

Hint: Luckily, every BUENOS test program has a `syscall` machine instruction, forcing it to issue an exit system call, if the main function ever returns. This `syscall` instruction is always at address `0x00001014`, something you can learn by disassembling a BUENOS test program:

```
prompt> mips-elf-objdump --disassemble tests/halt
...
00001000 <ro_segment>:
...
    1014: 0000000c  syscall
...
```

Design, implement, and test system call `syscall_kill` (15%)

Your task is to clarify the remaining design considerations in your report, and to implement the kernel functionality and system call. Demonstrate that your solution works as intended by adequate test programs, and describe them in your report.

Theoretical Part

T1 Concurrency: Parts must come together (15%)

To build a bicycle, many parts must come together. At the chaos factory, we assemble elegant bicycles that just have a steel frame and two wheels. (Who needs gears anyway?)
The chaos factory has two thread machines: one which creates and runs a thread for each steel frame produced, and another which creates and runs a thread for each wheel produced. The threads are then let loose on the factory floor to manifest sheer beauty out of utter chaos.
Your task is to synchronize the correct assembly of bicycles at the chaos factory. That is, you have to ensure that if the thread machines produce an adequate number of steel frames and wheels, the frames and wheels eventually assemble to form magnificent bicycles.

a) Pseudo-code (10%)

Write the pseudo-code for the functions `frame` and `wheel` that take no arguments, to be run by a steel-frame thread and by a wheel thread, respectively. You can employ semaphores and other synchronization primitives that we have covered in the course. You should use a pseudo-code notation similar to that found in [Dinosaur Book, Chapter 6].

b) Correctness (5%)

Argue that your solution is deadlock-free, and only allows usable bicycles (having one steel frame and two wheels) to be assembled.

T2 Online algorithms: Binary buddy system (15%)

Consider the design of a dynamic storage allocation system for servicing memory requests from a contiguous memory segment `A` of size 2^m (bytes), for a positive integer m. Let the system limit the *minimum size* of an allocated block to 2^k (bytes), for some $0 \leq k \leq m$. For instance, we might have $m = 32$ and $k = 9$, giving a contiguous memory segment of size 4 GB, and a minimum block size of 512 bytes.
In a ***binary buddy system*** [Knowlton 1965], the contiguous memory segment is initially thought of as one big free block of size 2^m. As memory allocation requests (`malloc`) are serviced, a free block is recursively ***split*** into adjacent equi-sized blocks (buddies). As memory deallocation requests (`free`) are serviced, adjacent equi-sized free blocks (buddies) are recursively ***coalesced*** into bigger free blocks. In particular, if there are no memory leaks in a program, we should always arrive at one big free block of size 2^m, once the program is finished.
Each block is therefore of size 2^i, for some $k \leq i \leq m$ — we say that a block is of ***log-size*** i. To quickly service `malloc` and `free` requests, a doubly-linked list of free blocks is maintained for each log-size i, where $k \leq i < m$. Initially, all free lists are empty. The free lists are populated both during `malloc` and `free` (as free blocks come and go during both operations).
A `malloc` request of x (bytes) is serviced by first rounding up to the nearest block size, i.e. $i = \max\left(2^k, \lceil \log_2 (x+1) \rceil\right)$. We then check the free list for log-size i, and if there is a free block, it gets allocated. If the free list is empty, we check the free list for log-size $i+1$. If a free block of log-size $i+1$ is available, we split the block into two blocks of log-size i, and allocate one of them. If the free list of log-size $i+1$ is empty, we check the free list for log-size $i+2$, and so on until $m-1$, at which point `malloc` ***breaks down*** (fails to allocate and returns an error).
The benefit of the ***binary*** buddy system is that we can compute the buddy of a block in $O(1)$ time, knowing only the address and size of the block, using the "xor" operation. (How?)

a) Implementation details (5%)

We consider a statically allocated contiguous memory segment (a statically-allocated array). As such, m and k are both static. In general, we maintain the free lists in the free blocks themselves, except for the $m - k - 1$ pointers to the heads of the lists (one for each i, for all $k \leq i < m$).

A block is either free or in use, as indicated by the 0^{th} byte at the block address. The log-size of the block (in either case) is given by the 1^{st} byte at the block address. If the block is free, the size is followed by a pointer to the next and previous free blocks in the free list (neither need point to a buddy!). If the block is in use, the data starts at the ***second word*** at the block address (which is typically the 4^{th} or 8^{th} byte at the block address, depending on your architecture).

Here is a partial implementation in C-like pseudo-code for guidance:

```
#include <stdlib.h>
#define m ...
#define k ...
#define LOG_SIZES (m - k - 1)

static char MEMORY[...];

union block_descr_t {
  struct {
    unsigned char in_use; // 0 if not in use
    unsigned char log_size;
  };
  // align the struct to fill sizeof(void*)
  size_t ptr_alignment;
};

struct free_list_node {
  union block_descr_t block_descr; // always not in use
  struct free_list_node* next;
  struct free_list_node* previous;
};

struct free_list_node* free_list_heads[LOG_SIZES];

// if is_in_use(ptr) == 1,
//   then ptr + 1 points to data,
//   else ptr points to a free_list_node struct
unsigned char is_in_use(void* ptr) {
  // A block is in use, if the first byte is not 0
  return *((unsigned char*) ptr) != 0;
};
```

Write the following two functions in C-like pseudo-code; both functions should run in $O(1)$ worst-case time.

`struct free_list_node* split(struct free_list_node* node)`

Effects: Assume that `node` is not in use. Split the block into two buddies.

Returns: A pointer to the higher order buddy, `NULL` if the block cannot be split any further.

`struct free_list_node* coalesce(struct free_list_node* node)`

Effects: Assume that `node` is not in use. Coalesce the block with its buddy.

Returns: A pointer to the coalesced block, `NULL` if the buddy is not free.

b) Runtime analysis (5%)

Analyse the running time of `malloc` and `free` in the binary buddy system. When doing that, it might be necessary to specify the operations a bit more precisely.

c) Fragmentation analysis (5%)

Let the contiguous memory segment `A` be called an ***arena***. If we are guaranteed that an allocated block is never larger than $\texttt{T} = 2^t$ (bytes), where $k \leq t$, how big an arena does the buddy system need to have without `malloc` breaking down? This number is a function of k and t, call it $\texttt{A}(k, t)$. Give as tight a lower bound (breaks down) and upper bound (works) for $\texttt{A}(k, t)$ as possible.

T3 Protocols: The fundamentals (10%)

In computing, a ***protocol*** is a set of rules and formal behaviour that control data processing between two or more parties to accomplish a task in cooperation. In this course, we have looked at protocols that deal with threads, processes, caches, file systems, disks, as well as other parts of an operating system and system resources.

Describe formally ***two*** protocols—only explained informally in [Dinosaur Book]—that you found most interesting. Name each protocol, describe it in sufficient detail so that it can be implemented in a straightforward manner, and tell us why you found it interesting. For instance, you could describe the protocol in the context of a small operating system like `BUENOS`.

Your answer should not be longer than one A4 page per protocol.

References

[`BUENOS` Roadmap] Juha Aatrokoski, Timo Lilja, Leena Salmela, Teemu J. Takanen, and Aleksi Virtanen. *Roadmap to the `BUENOS` System*, Version 1.1.2, Aalto University School of Science (2012).

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

[Knowlton 1965] Kenneth C. Knowlton. A fast storage allocator. *Communications of the ACM* **8**(10) (1965), 623–625.