UNIVERSITY OF COPENHAGEN
Department of Computer Science
February 6, 2015

G Assignment 1 for "Operating Systems and Multiprogramming", 2015

Deadline: Febuary 16, 2015 at 23:59

Exercise Structure for the Course

The course Operating Systems and Multiprogramming ("Styresystemer og Multiprogrammering") includes five G assignments ("Godkendelsesopgaver") which are *published on Fridays* (at 12:00) and are due *ten days later on Monday by midnight (at 23:59)*. G assignments should be solved in groups of two to three students. Each of the G assignments will be graded with up to 3 points. To be admitted to the exam, you need to score points (> 0) on at least four G assignments, and a total of at least 8 points. There is *no resubmission* of G assignments in this course.
Solutions to G assignments are to be handed in by uploading an archive file (`zip` or `tar.gz`) to Absalon. When you upload your solution to Absalon, please use the last names of each group member in the file name and the assignment number as the file name, like so:

`lastname1_lastname2_G1.zip` or `lastname1_lastname2_G1.tar.gz`.

Use plain text or PDF when handing in your short report, and explain your code with comments. When you write C code, you must also write a `Makefile` which enables the teaching assistants to compile your code by a simple `make` invocation on the command line.

About this Exercise

Topics

The first task is an exercise on **C programming with pointers**. You should implement a priority queue in an object-oriented style. The second task will introduce you to the **operating system Buenos**. You should implement **essential system calls for terminal input and output**.
Buenos is an educational operating system developed at the University of Helsinki, which runs on a MIPS32 platform simulated by the program YAMS. Guides and links can be found on Absalon. Take a look at the *Buenos roadmap* to get an overview of the system. In order to modify and compile source code for Buenos, a cross-compiler to MIPS is required. There is a manual which describes the setup, and also a virtual machine with a complete development environment for Buenos. Your teaching assistant can help in case of technical setup problems.

What to hand in

Please hand in your solution by uploading a single archive file (`zip` or `tar.gz` format) with:

1. A directory containing your code for Task 1 (with a `Makefile`).
2. A source tree for Buenos which includes your work on Task 2 (as documented in the report)—and programs you have used for testing.
3. A **short** report in plain text (`.txt`) or PDF (`.pdf`), which discusses possible solutions and explains design decisions you took (why you preferred one particular solution out of several choices). Pay particular attention to documenting changes to existing Buenos code.

As a minimum, the C code you hand in should compile without errors. Preferably, you should also eliminate all warnings issued when compiling with flags `-Wall -pedantic -std=c99`.

Task 1. A priority queue

You work for Jyrki Katajainen and Company, where they need an efficient priority-queue implementation.
The current solution is based on a linked list of elements sorted by priority. With a frown on his face your boss mutters something about $O(n^2)$, and that you need a better solution. He also reveals that he had a great idea[1] when he woke up this morning: There exists this tree-like data structure, called a *heap*, which has the property that the root always contains the maximal element. Surely, it must be possible to use this fact when implementing a priority queue.
Of course, your implementation should be *generic* (be able to store any type of elements) and *dynamic* (be able to store an arbitrary number of elements and use linear space), but these are just practicalities.
Then the boss went skiing, leaving you to work out the details.
Implement the maximum priority queue using a heap with the following operations (with functionality suggested by their names):

- heap_initialize for initializing a new, empty heap; running time: $O(1)$
- heap_clear for releasing the resources used by a priority queue; running time: $O(1)$
- heap_size for getting the number of elements stored; running time: $O(1)$
- heap_top for accessing an element with the highest priority; running time: $O(1)$
- heap_insert for inserting an element; running time: $O(\log n)$
- heap_pop for removing an element with the highest priority; running time: $O(\log n)$

```
#include <stddef.h> // size_t

typedef struct {
  void* thing;
  int priority;
} node;

typedef struct {
  node* root;
  size_t size;
  /* You may want to allocate more
     space than strictly needed. */
  size_t alloc_size;
} heap;

void heap_initialize(heap*);
void heap_clear(heap*);
size_t heap_size(heap*);
void* heap_top(heap*);
void heap_insert(heap*, void*, int);
void* heap_pop(heap*);
```

Along with your implementation, provide a `Makefile` and a small test program which uses all implemented functions.

Task 2. Buenos system calls for basic I/O

System calls are the mechanism by which user programs can call kernel functions and get OS service. Section 6.3 and 6.4 in the Buenos Roadmap describe the mechanism.
Each system call in Buenos has a unique number (defined in `proc/syscall.h`), and it can use up to three arguments in registers. A Buenos user process makes a system call by using the MIPS instruction `syscall`, with register `a0` containing the number of the desired system call, and `a1`, `a2` and `a3` containing arguments to the kernel function.

[1] In fact, the great idea is from your algorithm textbook (see [Cormen et al. 2009, Chapter 6, Section 5]).

The `syscall` instruction generates an exception, called a *trap*, and execution continues in kernel mode. With interrupts disabled, a jump is performed: program execution continues at a fixed address with code to save the current (user) context, and then (with interrupts enabled again) proceeds by jumping to a function `syscall_handle` in file `proc/syscall.c` which executes the system call itself (shown here). When `syscall_handle` returns, the return value of the system call is stored in register `v0`, and control returns to the user program.

proc/syscall.c

```
void syscall_handle(context_t* user_context) {
  /*
  reg a0 is syscall number, a1, a2, a3 its arguments
  userland code expects return value in register
  v0 after returning from the kernel. User context
  has been saved before entering and will be
  restored after returning.
  */
  switch (user_context->cpu_regs[MIPS_REGISTER_A0]) {
  case SYSCALL_HALT:
      halt_kernel();
      break;
  default:
      KERNEL_PANIC("Unhandled system call\n");
  }
  /* move program counter to next instruction */
  user_context -> pc +=4;
}
```

The `_syscall` function (defined in MIPS assembler in file `tests/_syscall.S`) acts as a wrapper around the MIPS machine instruction `syscall` and can be called from C. It is used inside `tests/lib.c` to define a "system call library".
As can be seen in the code above, only one system call `halt` is implemented. In order to do anything useful with Buenos, new system calls for basic console I/O support are essential.

1. **Implement system calls `read` and `write`** with the behaviour outlined below (also described in Section 6.4 of the Buenos roadmap). The system call interface in `tests/lib.c` already provides wrappers for these calls, and their system call numbers are defined in `proc/syscall.h`. It is not necessary to make the system call code "*bullet-proof*" (as it is called in the Buenos roadmap).

   (a) `int syscall_read(int fhandle, void* buffer, int length);`
   Read at most `length` bytes from the file identified by `fhandle` (at the current file position) into `buffer`, advancing the file position. Returns the number of bytes actually read (before reaching the end of the file), or a negative value on error.
   Simplification: Your implementation should only read from `FILEHANDLE_STDIN`, (number 0 in `proc/syscall.h`), using the *generic character device* driver.

   (b) `int syscall_write(int fhandle, void const* buffer, int length);`
   Write at most `length` bytes from `buffer` to the open file identified by `fhandle`, starting at the current position and advancing the position. Returns the number of bytes actually written, or a negative value on error.
   Simplification: Your implementation should only write to `FILEHANDLE_STDOUT`, (number 1 in `proc/syscall.h`), using the *generic character device* driver.

   Look at the code inside `init_startup_fallback` (file `init/main.c`) to see how to acquire and use the generic character device (also see `drivers/gcd.h`).

2. **Test the implemented system calls** by a small C program `readwrite.c` in directory `tests/` (see `halt.c` there for an example). Copy the compiled program to the Buenos disk `fyams.harddisk` using `tfstool` and start it using boot argument `initprog=[disk]readwrite` (see Sections 2.6–2.7 of the Buenos roadmap for instructions and explanations).