UNIVERSITY OF COPENHAGEN
Department of Computer Science
Robert Glück glueck@diku.dk
Jost Berthold berthold@diku.dk

February 8, 2013

G-Exercise 1 for "Operating System and Multiprogramming", 2013

Hand in your solution by uploading files in Absalon before Feb.19

Exercise Structure for the Course

As in previous years, the course "Operating Systems and Multiprogramming" (Styresysteme og Multiprogrammering) includes a number of G-exercises (Godkendelsesopgave, approval exercises) that are *published on Fridays* and have to be *handed in before Monday (at midnight, 23:59)*. G-exercises should be solved in groups of two students. Each G-exercise will be graded with either 0, $\frac{1}{2}$ or 1 point; and you need a total of at least $3\frac{1}{2}$ points to be admitted to the exam. As there will be five exercises in the 2013 course, this requires you to hand in at least four exercises. For the first four exercises, you can resubmit an improved solution one week later (if you handed in a reasonable solution in the first round).
Solutions to G-exercises are in general 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 exercise number as the file name, like so:
`lastname1_lastname2_G1.zip` or `lastname1_lastname2_G1.tar.gz`.
Use simple txt format or pdf for portability when handing in text, and explain your code with comments. When you write C code, it is required to also write a Makefile which enables the teaching assistants to compile your code by a simple "make" invocation.
Please also read the document *Krav til G-opgaver* (in Danish) provided on Absalon. It explains the idea of the exercises and general requirements for code that you hand in.

About this Exercise

Topics

The first task in this exercise is a small **exercise in C programming with pointers**. You will implement a simple queue which holds data in a linked list internally, and add a special summation function which takes a function pointer argument.
The second part will introduce you to the **operating system Buenos**. You will implement **essential system calls for terminal I/O**. Buenos is an educational operating system developed at the University of Helsinki, which runs on a MIPS32 platform simulated by YAMS. Links and manuals can be found on Absalon (see *Læsemateriale*). Take a look at the *Buenos roadmap* document 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, and on the DIKU systems. 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 pdf or txt format, which discusses possible solutions and explains design decisions you took (why you preferred one particular way 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. For best results, you should also eliminate all warnings issued when compiling with flags `-Wall -pedantic -std=c99`.

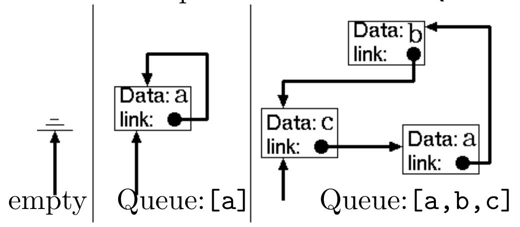
Tasks

1. **A simple queue using a linked list**

   A queue can be a useful data structure for system programming in many areas, for example when implementing scheduling mechanisms. Queues support (at least) the following two operations: enqueue adds an element to the end of the queue, dequeue removes the first element from the queue and returns it. The element which was enqueued first is the first to be removed by dequeue (as it should be in a *"waiting queue"*). In this exercise, you will implement a simple queue by queue nodes connected in a circular linked list (and containing some data). Any data type could be the content of a node; we define the `Data` type to be `void*`. The shown interface file `queue.h` is provided on Absalon. Implement the declared functions in a file `queue.c` and use the queue in a suitable test program.

   Use the following **implementation**:
   - A queue is stored as a pointer to a QNode. If the queue is empty, the pointer is NULL. If the queue is not empty, the stored pointer points to the element that was *enqueued last* (`c` in the picture below).
   - Elements of the queue are stored in a linked list of QNodes.

     The link field of every queue node holds a pointer to the next queue element. The link field of the *last* queue element contains a pointer to the *first* queue element – if the queue contains exactly one element, the link points to the same QNode.



queue.h

```
#ifndef QUEUE_H
#define QUEUE_H

typedef void* Data;

typedef struct QNode_ {
  Data content;
  struct QNode_ *link;
} QNode;

/* return number of elements in queue */
int length(QNode* queue);

/* add element at rear end */
void enqueue(QNode** queue, Data el);

/* remove and return front element */
Data dequeue(QNode** queue);

/* sum values of all data in queue */
int sum(QNode* queue, int (*val)(Data));

#endif /* QUEUE_H */
```

   (a) Implement functions enqueue and dequeue (described above), and a length function.

   (b) Explain why enqueue and dequeue must use an argument of type QNode∗∗.

   (c) Implement the sum function, which returns a sum of values of data in the queue, and uses function pointer ∗val to get the values.

   For example, if the value function maps `a` to 1, `b` to 2, and `c` to 3, and the queue `qu` contains `[a,b,c]` (see picture above), the call sum(qu,value) will return 6. Called with a constant function which always returns 1, the sum function will return the queue's length.

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.

Instruction `syscall` generates an exception (called a *trap*), 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

```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;
}
```

Function `_syscall` (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.

(a) **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).

   i. `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.

   ii. `int syscall_write(int fhandle, const void *buffer, int length);`
   Write `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`).

(b) **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 using *tfstool* and start it using boot argument `initprog=[root]readwrite` (see page 6ff of the Buenos roadmap for instructions and explanations).