UNIVERSITY OF COPENHAGEN
Department of Computer Science
Jost Berthold Jyrki Katajainen
berthold@di.ku.dk jyrki@di.ku.dk

February 7, 2014

G-Assignment 1 for "Operating System and Multiprogramming", 2014

Deadline: Feb. 17 at 23:59

Exercise Structure for the Course

As in previous years, the course "Operating Systems and Multiprogramming" (Styresysteme og Multiprogrammering) includes a number of G-assignments (Godkendelsesopgave) that are *published on Fridays* and have to be *handed in by the next Monday (at midnight, 23:59)*. G-assignments should be solved in groups of two students. Each of the five G-assignment will be graded with up to 3 points. To be admitted to the final 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-assigments in this years course.
Solutions to G-assignments 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 plain text format or pdf for portability when handing in text, 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.
Please also read the document *Krav til G-opgaver* (in Danish) provided on Absalon. It details some 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 doubly-linked list in an object-oriented style, which holds two pointers (sort-of) in the same location.
The second task 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 *Code and additional material*). 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 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`.

Tasks

1. **A space-efficient doubly-linked list**

   You work for Jyrki Katajainen and Company, where they need a space-efficient implementation of a doubly-linked list. The standard solutions provided by others are not satisfactory for the embedded-system application you are developing using the C language. Your boss had a great idea[1]: One can represent the two pointers in one by storing (in node x) the "pointer" x.ptr = x.next ⊕ x.prev, where ⊕ is the bitwise XOR operation (^ in C). He then left to go fishing, leaving you to work out the details.

   Implement the list with the following operations (with functionality suggested by their name):

   (a) insert and extract in $O(1)$ time, with a parameter indicating whether to insert at (or extract from) the head or the tail.

   (b) A search procedure which works in linear time, $O(n)$.

   (c) A reverse function which works in $O(1)$ time (yes, reversing the whole list).

   Together with your implementation, provide a Makefile and a small test program which uses all implemented methods.

```
typedef int bool;
typedef void item; /* we store pointers */

typedef struct node_ {
    item          *thing;
    struct node_ *ptr;
} node;
typedef struct dlist_ {
    node *head, *tail;
} dlist;

void insert(dlist *this, item* thing,
            bool atTail);
item* extract(dlist *this, bool atTail);
void reverse (dlist *this);

item* search(dlist *this,
             bool (*matches(item*)));
```

   Hint: Your implementation needs pointers to the head and the tail of the list. Use the fact that XOR (⊕) is associative, and that x ⊕ x = 0.

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 can be found in your algorithms textbook, see Cormen et al. 2009, Exercise 10.2-8.

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

```
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 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`).

(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 sections 2.6-2.7 of the Buenos roadmap for instructions and explanations).