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

February 17, 2012

G-Exercise 2 for "Operating System and Multiprogramming", 2012

Hand in your solution by uploading files in Absalon before Feb.28

Exercise Structure for the Course

Please see the introduction on exercise 1 for information about the exercise structure.
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 document the code by comments. 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

This exercise will accomodate you with the Buenos system, how to run it using the simulator YAMS, and how to compile and work with its source code. Buenos is an educational operating system developed at the University of Helsinki, which runs on a MIPS32 platform simulated by YAMS. Links and manuals for both can be found on Absalon (see *Undervisningsmateriale*). In order to modify and compile source code for Buenos, a cross-compiler to MIPS is usually required. There is a manual which describes the setup, and also a virtual machine with a complete development environment for Buenos. Please ask your teaching assistant for help in case of technical setup problems.
In the exercise, you will implement some essential system calls for user processes and basic terminal I/O. For user processes, you need to also define data types and implement a library of helper functions. Work on both tasks at a time, as they refer to each other.

What to hand in

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

1. A source tree for Buenos which includes your work on task 1 and task 2 (as documented in the report) – and possibly programs you have used for testing.

2. A short report in pdf or txt format.
   Your report should discuss possibilities to solve the tasks, and explain the design decisions you took for your solution (why you preferred one particular way out of several choices). It is important that you also comment the C code itself (refer to the report for longer explanations). Pay particular attention to documenting changes to existing Buenos code.

The Buenos code you hand in should compile without errors with the original setup. As Buenos uses `-Werror` in its Makefile, this implies that your code is also warning-free.

Tasks

1. **Types and Functions for User Processes in Buenos**

   The basic Buenos system supports Kernel threads as defined in file `kernel/thread.h` with types `thread_table_t`, but there is no notion of *user processes*. In this task, we implement basic user process abstractions and a small library for the kernel to manage them. Main changes for your implementation should go into files `proc/process.h` and `proc/process.c`.

   (a) Define a data structure to represent a user process (process abstraction) in Buenos. The data structure should, as a minimum, contain the name of the respective executable. Most likely you will need a lot more data in order to solve the next task (for instance, for management of open files and child processes). Look at `thread_table_t` and the process control block description in the book [Silberschatz,p.104] for inspiration.

   (b) The kernel should be able to control all processes, so define a process table or similar structure to hold all processes. It is *strongly recommended* to use a fixed upper bound for the number of user processes that can be managed by the kernel simultaneously.

   (c) Implement a library of helper functions for process management which contains the functions described below. Function `process_start` is already defined, but might require some changes.

   Process Mgmt. Functions

   ```
   /* Run process in new thread , returns PID of new process */
   process_id_t process_spawn( const char *executable );

   /* Run process in this thread , only returns if there is an error */
   int process_run( const char *executable ) ;

   process_id_t process_get_current_process( void ) ;

   /* Stop the current process and the kernel thread in which it runs */
   void process_finish( int retval );

   /* Wait for the given process to terminate , returning its return value,
    * and marking the process table entry as free */
   uint32_t process_join( process_id_t pid ) ;

   /* Initialize process table. Should be called before any other process-related calls */
   void process_init ( void ) ;
   ```

   You should make sure that access to the process abstraction is *atomic*, i.e. while one kernel threads is reading the structure, no other thread should be able to modify it.

   It will be helpful to have a look into file `init/main.c` and see how the first program (`initprog`) is started using `init_startup_thread`. Modify this code to use your new library functions.

2. **System Calls in Buenos**

   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.

The effect of executing `syscall` is to generate an exception (usually called a *trap*). Execution proceeds 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 on the right).

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;
}
```

When `syscall_handle` returns, the return value of the system call is expected in register `v0` and control returns to the user program. 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, no system call besides `halt` is implemented). In order to do anything useful with Buenos, basic console I/O and user process control need to be implemented as new system calls.

Implement the following system calls with the behaviour as outlined below (and also described in Section 6.4 of the Buenos roadmap). The system call interface in `tests/lib.c` already provides wrappers for these calls. Use the system call numbers defined in `proc/syscall.h`.

proc/syscall.h

```
#define SYSCALL_EXEC 0x101
#define SYSCALL_EXIT 0x102
#define SYSCALL_JOIN 0x103
...
#define SYSCALL_READ 0x204
#define SYSCALL_WRITE 0x205
```

It is not necessary to make the system call code "*bullet-proof*" (as it is called in the Buenos roadmap). You also don't need to protect processes from reading each other's data.

(a) `int syscall_exec(const char *filename);`
Create a new (child) user process which loads the file identified by `filename` and executes it.

This should be easy after working on the first task.

(b) `void syscall_exit(int retval);`
Terminate the current process with exit code `retval`. Never returns. `retval` must be positive, as a negative value indicates a system call error in `syscall_join` (see next).

Important: When implementing this system call, use the code shown on the right before calling `thread_finish`, where `thr` is the kernel thread executing the process that exits. This cleans up the virtual memory used by the running user process, a topic handled later in the course.

inside syscall exit

```
vm_destroy_pagetable(thr->pagetable);
thr->pagetable = NULL;
```

(c) `int syscall_join(int pid);`
Wait until the child process identified by `pid` is finished (i.e. calls `process_exit`), and return its exit code. Return a negative value on error.

Important: Note that it should be possible to use this system call with the `pid` of a process that already exited (such processes are sometimes called *zombies*). A related problem is what to do with child processes of a process that calls `syscall_exit`.

(d) `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 when an error occurred.

Simplification: Your implementation should only be allowed to read from file handle `FILEHANDLE_STDIN`, (which has number 0, see `proc/syscall.h`). You will use the *generic character device* driver in `drivers/gcd.h`.

(e) `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 be allowed to write to file handle `FILEHANDLE_STDOUT`, (which has number 1, see `proc/syscall.h`). You will use the *generic character device* driver in `drivers/gcd.h`.