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

February 15, 2013

G-Exercise 2 for "Operating System and Multiprogramming", 2013

Hand in your solution by uploading files in Absalon before Feb.25

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 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

This exercise is about implementing support for user processes in Buenos. You are asked to define process-related data types and implement essential system calls as well as a library of helper functions for process control. The tasks of this exercise are connected and only separated for documentation purposes. You should work on all tasks simultaneously.
Please consult Section 6 of the Buenos Roadmap for background knowledge.

What to hand in

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

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).

Your report is the basis for evaluation of the exercise. Therefore, it is important to *include in the report* the most *important parts of the C-code* that you yourself wrote or modified. Pay particular attention to documenting changes to existing Buenos code. You should also comment the C code itself (refer to the report for longer explanations).
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.

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` (provided with this exercise).

1. 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 and the state of the process (dead, zombie, running, etc). Most likely you will need a lot more data in order to solve the next task. Look at `thread_table_t` and the process control block description in the book [Silberschatz,p.104] for inspiration. In `proc/process.h`, the struct type `process_control_block_t` is already defined for you, although it does not contain any of the information you will need to implement the rest of this task. Likewise, in `proc/process.c`, the process table is already defined as an array of `process_control_block_t`s.

2. Implement a library of helper functions for process management in the kernel, which contains the functions described below. A function `process_start` is already defined, but will require some changes to use the new data structure.

Process Mgmt. Functions

```
/* Load and run the executable as a new process in a new thread
 * Argument: executable file name. Returns: PID of new process. */
process_id_t process_spawn( const char *executable );

/* Stop the current process and the kernel thread in which it runs
 * Argument: return value */
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 );
```

Hints:

(a) You should ensure mutual exclusion when accessing the process table, ie. while one kernel thread is reading the structure, no other thread should be able to modify it.

(b) The `process_spawn` function should call the already defined function `process_start`, but you may need to modify the latter to take a process ID rather than a program as argument. If you store the program name in the process control block, `process_start` can use the process ID to look it up in the process table.

(c) When implementing `process_join`, the calling process will need to wait until a given event occurs. This is best implemented using the kernel-provided *sleep queue*; see section 5.2 in the Buenos roadmap.

(d) In `process_finish`, 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 `process_finish`

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

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.

You will need to modify the definition of `process_control_block_t` in `proc/process.h`, and you will need to finish the stub functions in `proc/process.c`. You will most likely find it necessary to write additional utility functions, eg. for finding available process IDs.

2. System Calls for User Process Control in Buenos

The functionality to start user processes implemented in Task 1 is made accessible to users by means of system calls for process control.

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
...
```

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.

1. `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). The return value is the process ID of the new process.

2. `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).

3. `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`.