UNIVERSITY OF COPENHAGEN
Department of Computer Science
February 13, 2015

G Assignment 2 for "Operating Systems and Multiprogramming", 2015

Deadline: February 23, 2015 at 23:59

Handing in: via Absalon

Assignments in this Course

Your solutions to the G assignments are to be handed in by uploading a single archive file (in `zip` or `tar.gz` format) that contains the developed code (in C) and a brief report giving a high-level description of the solution (in `pdf` or `txt` format). The uploaded files should be named using the surnames of each group member and the assignment number as in:

`Katajainen-Mikkelsen-G2.zip` or `Katajainen-Mikkelsen-G2.tar.gz`.

Provide also a `Makefile` which will enable the teaching assistants to compile the code simply by invoking `make` in the extracted directory.
More information on the assignments is available at the beginning of G1 and in the course plan in Absalon.

G2: Problem Formulation

Topic

This assignment 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 provide a library of helper functions for process control. The two tasks are connected and only separated for documentation purposes. You should work on both tasks simultaneously.
Please consult Section 6 of the Buenos Roadmap for background knowledge.

What to hand in

The uploaded archive file should contain:

1. A source tree for Buenos which includes your work for both tasks—and possibly programs you used for testing.
2. A short report where you document your code, discuss different possibilities to solve the tasks, and explain the design decisions made (why you preferred one particular way out of several choices).

We supply a Buenos source tree with stubs for implementing user processes. Additionally, both `syscall_read` and `syscall_write` works (this might be useful for Task 2), but note that this tree does **not** contain a solution to G1's Buenos task, as we have obfuscated that code. You may choose to use your G1 Buenos part in place of our obfuscated version (the functionality is the same).
Assessment of the quality of your work will be based on the report. Therefore, it is important to *include there* the most *important parts of the C code* that you wrote or modified. Pay particular attention to documenting changes to existing Buenos code. You should also comment the C code itself (and 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` option in its `Makefile`, your code should also compile without warnings.

Task 1. Types and functions for userland 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 assignment text).

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 description of process control blocks in our the textbook [Silberschatz et al. 2014, Section 3.1.3] 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 blocks.

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 Management Functions

```
/* Load and run the executable as a new process in a new thread
   Argument: executable file name; Returns: process ID of the new process */
process_id_t process_spawn(char const* 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, return its return value,
   and mark 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, i.e. 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 below 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 to finish the stub functions in `proc/process.c`. You will most likely find it necessary to write additional utility functions, e.g. for finding available process IDs.

Task 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). Also, you do not need to protect processes from reading each other's data.

1. `int syscall_exec(char const* filename);`
   Create a new (child) user process which loads the file identified by `filename` and executes the code in it. The return value is the process ID of the new process.

2. `void syscall_exit(int retval);`
   Terminate the current process with the exit code `retval`. That is, this function halts the process and 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`.