

OSM

G assignment 2

Tobias Hallundbæk Petersen (xtv657)

Ola Rønning (vdl761)

Nikolaj Høyer (ctl533)

February 24, 2014

Contents

| | | |
|----------|---|----------|
| 1 | Types and Functions for Userland Processes in Buenos | 2 |
| 1.1 | Define a data structure to represent a user process | 2 |
| 1.2 | Implement a library of helper functions | 2 |
| 1.2.1 | <code>process_spawn</code> | 3 |
| 1.2.2 | <code>process_finish</code> | 4 |
| 1.2.3 | <code>process_join</code> | 5 |
| 1.2.4 | <code>process_init</code> | 5 |
| 2 | System Calls for User-Process Control in Buenos | 6 |

1 Types and Functions for Userland Processes in Buenos

1.1 Define a data structure to represent a user process

We start by defining the data structure needed for representing user processes. This is done in `syscall.h`. First we define a process state as consisting of a number of different states, then we define our process control block as using this process state, along with a process id (an integer), a parent id, a name, a return value and an integer representing the number of child processes.

```
#define CONFIG_MAX_NAME 128
typedef enum {
    PROC_FREE,
    PROC_RUNNING,
    PROC_READY,
    PROC_SLEEPING,
    PROC_ZOMBIE,
    PROC_NONREADY,
    PROC_DYING,
    PROC_NOTFREE
} process_state_t;

typedef struct {
    process_state_t state;
    process_id_t id;
    process_id_t parentid;
    char name[CONFIG_MAX_NAME];
    int retval;
    int children;
} process_control_block_t;
```

1.2 Implement a library of helper functions

We start by modifying the existing `process_start` function, making it take a process id as input, instead of an executable. We make sure that the thread entry `my_entry` gets the process id of the input, and also we modify the `file = ...` line to update the process table with the process name, as well as opening the file. This also requires a spinlock, to prevent corruption of the process table.

```
void process_start(process_id_t pid)
{
    ...
    my_entry->process_id = pid;
```

```

...
spinlock_acquire(&process_lock);
file = vfs_open((char *)process_table[pid].name);
spinlock_release(&process_lock);
...

```

We also need a helper function, which creates a new process id. Such a function is defined below. We look through the process table and checks for a free process. This is then returned. Also, a spinlock is required when accessing the process table. If this in some way or another fails, -1 is returned.

```

process_id_t process_new_id() {
    int pid;
    process_control_block_t process;
    spinlock_acquire(&process_lock);
    for (pid = 0; pid < CONFIG_MAX_PROCESSES; pid++) {
        process = process_table[pid];
        if (process.state == PROC_FREE){
            process.parentid = -1;
            process.children = 0;
            process.state = PROC_NOTFREE;
            spinlock_release(&process_lock);
            return pid;
        }
    }
    spinlock_release(&process_lock);
    return -1;
}

```

1.2.1 process_spawn

When spawning a process, we give an executable as input and return its process id. Thus, we need the helper function described above - we start by acquiring an available process id, then update our process table. Again, we apply a spinlock before making changes to the table. The parent process for the new process must be the current running process, hence `parentid` is set with the value of `process_get_current_process()`. We also use `stringcopy` to place the name of the executable in the table. The other table values should be clear from the context.

We then make a check - if `pid` is 0, then it is our init process running. If this is the case, we simply set the process as running and return. Otherwise, its some sub-process and we create a new thread for the child process and run it. Also, we increment the parents child-counter.

```

process_id_t process_spawn(const char *executable) {
    process_id_t pid;
    TID_t child_tid;
    pid = process_new_id();

    spinlock_acquire(&process_lock);
    process_id_t parent_process = process_get_current_process();
    process_table[pid].state = PROC_READY;
    process_table[pid].parentid = parent_process;
    process_table[pid].id = pid;
    process_table[pid].children = 0;
    strcpy(process_table[pid].name, executable, CONFIG_MAX_NAME);
    if (pid == 0) {
        process_table[pid].state = PROC_RUNNING;
        spinlock_release(&process_lock);
        return pid;
    }
    child_tid = thread_create((void (*)(uint32_t))process_start, (uint32_t) pid) ←
        ;
    if (child_tid < 0){
        process_table[pid].state = PROC_FREE;
        return -1;
    }
    process_table[parent_process].children += 1;
    process_table[pid].state = PROC_RUNNING;
    spinlock_release(&process_lock);
    thread_run(child_tid);
    return pid;
}

```

1.2.2 process_finish

```

/* Stop the process and the thread it runs in. Sets the return value as well ←
 */
void process_finish(int retval) {
    thread_table_t *thr;
    interrupt_status_t intr_status;

    intr_status = _interrupt_disable();
    spinlock_acquire(&process_lock);
    process_id_t current_process = process_get_current_process();
    process_table[current_process].state = PROC_DYING;
    process_table[current_process].retval = retval;
    sleepq_wake_all(&(process_table[current_process]));
    spinlock_release(&process_lock);
    _interrupt_set_state(intr_status);

    thr = thread_get_current_thread_entry();
    vm_destroy_pagetable(thr->pagetable);
    thr->pagetable = NULL;
    thread_finish();
}

```

```
}
```

1.2.3 process_join

```
int process_join(process_id_t pid) {
    interrupt_status_t intr_status;
    int retval;

    intr_status = _interrupt_disable();
    spinlock_acquire(&process_lock);

    if (process_get_current_process() != process_table[pid].parentid) return -1;
    while (process_table[pid].state != PROC_DYING) {
        sleepq_add(&(process_table[pid]));
        spinlock_release(&process_lock);
        thread_switch();
        spinlock_acquire(&process_lock);
    }

    retval = process_table[pid].retval;
    spinlock_release(&process_lock);
    _interrupt_set_state(intr_status);

    return retval;
}
```

1.2.4 process_init

Process init simply runs through the process table, setting its state to **FREE** and initialising all other values its values to defaults as well. It also resets the spinlock, to make sure it has its default state.

```
void process_init() {
    int i;
    spinlock_reset(&process_lock);
    for (i=0; i<CONFIG_MAX_PROCESSES; i++) {
        process_table[i].state      = PROC_FREE;
        process_table[i].id         = -1;
        process_table[i].parentid   = -1;
        process_table[i].name[0]    = '\0';
        process_table[i].retval     = -1;
        process_table[i].children   = 0;
    }
}
```

2 System Calls for User-Process Control in Buenos

We now modify `syscall.c` to use the functions implemented in `process.c`. We simply add functions we can call when the appropriate syscalls are invoked (see below).

```
int syscall_exec(char const* filename){
    int pid = process_spawn(filename);
    return pid;
}
void syscall_exit(int retval){
    process_finish(retval);
}
int syscall_join(int pid){
    return process_join(pid);
}
```

We now add cases to the syscalls in `syscall.c`, that loads and writes to registers, calling the functions defined above.

```
switch(user_context->cpu_regs[MIPS_REGISTER_A0])
...
case SYSCALL_EXEC:
    V0 = syscall_exec((char *) A1);
    break;
case SYSCALL_EXIT:
    syscall_exit(A1);
    break;
case SYSCALL_JOIN:
    V0 = syscall_join(A1);
    break;
...
```