Styresystemer og Multiprogrammering

G-opgave #2

Jenny-Margrethe Vej
(`rwj935@alumni.ku.dk`)
Klaes Bo Rasmussen
(`twb822@alumni.ku.dk`)

Datalogisk Institut, Københavns Universitet
Blok 3 - 2013

1 Types and Functions for User Processes in Buenos

1. We defined the required datastructure `process_control_block_t` within process.h. For now it only contains what we belive to be required to complete the given task. It is as minimalistic as possible.

We can save a state, given by the `enum process_state_t`. There are only three different states, PROCESS_FREE, PROCESS_RUNNING or PROCESS_ZOMBIE, keeping the naming convention with "fileName_name". FREE is for processes that are ready to take on a new thread. RUNNING states that the process is currently in use by another thread. ZOMBIE is for processes that are done running but have yet to be joined so they become FREE again.

It keeps a return value, which is used in `process_join`, `process_finish` and it also acts as the exit code for `syscall_exit`.

It contains a small buffer which is used for saving the name of the executeable to be run, currently programs can have a name of maximum 32 chars.

Last, and most importantly, it keeps track of which process it is with a process identifier. This process id is used any time a process is called upon for any functionality.

2. We have implemented all the given functions in the second task inside the file `process.c`. I will explain in the order they are inside the file.

`process_init()`

We initialize the processes by first resetting the spinlock on the spinlock variable we have defined as `process_table_slock`. Then we run through all of the processes and set their state to `PROCESS_FREE`.

`_process_id_t process_spawn(const char *executable)`

We have followed buenos roadmap for "Figure 5.1: Code executed by a thread wishing to go to sleep." So, first we disable interrupts, then acquire the spinlock for the process table.

We look for any free processes in a for loop, as it is now, if none are free, we simply throw a kernel panic. If we find a free process, we use its id, pid, and copy the name of the program we want to run to the `process_table[pid].executable` variable using the function `stringcopy` included from `lib/libc.h`. We also set the state of the process to `PROCESS_RUNNING` so it won't be used again by other threads.

Then we create a new thread which runs `process_start` on the process. In turn, we have changed the `process_start` function to accept a `pid` rather than a file to be executed. And in `main.c` we now call `process_spawn` and `process_join` rather than `process_start`. The new thread is started with `thread_run`. Then we need to release the spinlock and set the interrupt status to what it was before we disabled it. Lastly we return the `pid` of the process we spawned a thread on.

`void process_finish(int retval)`

Create a variable which can contain a `thread_table_t`, and one for holding the interrupt status. Also get and save the pid for the current process.

Disable interrupts and acquire the spinlock. Then set the retval value in the process table for the given pid, also set the state of the process to `PROCESS_ZOMBIE`. Then destroy the current pagetable as instructed. We are not sure if this could be done after the spinlock instead, but you requested that we moved the spinlock to just before the `thread_finish` call, so now it's like this.

Wake up the process sleeping on the address of `process_table[pid]`. Release the spinlock and set the interrupt status back to what it was before the spinlock. Then let the thread finish with `thread_finish()`.

`int process_join(process_id_t pid)`

We initialize the usual interrupt status valuable, and this time also an integer retval. Disable interrupts and acquire the spinlock. Then we wait while the process given by `pid` is not a zombie. During this we add the process to the sleep queue, release the spinlock and switch thread before we acquire the spinlock again. All as prescribed by buenos roadmap figure 5.2.

When we are done waiting in the while loop, we set the state of the given process to `PROCESS_FREE`, and the variable `retval` to the return value from the process. Release the kraken! Or rather spinlock, and set the interrupt status back to what it was before. Then return with `retval`.

All these functions were so short that we didn't find it necessary to write extra helper functions outside. So to keep it simple, we wrote all the extra code in the predefined functions.

2 System Calls for User Process Control in Buenos

Making the system calls were all very straight forward as most of the functionality for them was implemented in the first task, but we will just descripe them here shortly.

1. `syscall_exec` runs `process_spawn` with the given filename as an argument, and then saves the returned `pid` to register `V0`.

2. `syscall_exit` runs `process_finish` with a given return value. It saves nothing, but the return value should be positive, as negative values would indicate an error in `syscall_join`.

3. `syscall_join` runs `process_join` with the given `pid` and saves the return value from this, to `V0`. A negative value indicates that there was some error along the way.

3 Running the tests

We mostly worked with the exec test handed out, and of course had to write the hello world test to the disk aswell. Now both run seemingly flawlessly. exec is startet with initprog when running buenos, and this starts hw without any problems. Then we get to write my name as many times as we want, before pressing enter instead of writing a name. The machine halts and shuts down by software request. Looking at the calc.c file we are guessing that this isn't for this assignment, and just hoping everything else is in order. We have confirmed that new processes now spawn on different `pids`, and that the test programs do what they should, hoping that is sufficient to claim that everything works as intended.