CS342: Operating Systems Lab
Department of Computer Science and Engineering,
Indian Institute of Technology, Guwahati
North Guwahati, Assam 781 039 Exercise UP-02

OS Lessons: Interrupts, System calls, Exit Status, Return Value
Rating: Moderate
Last updated: 20 February 2017

Please do not start on this exercise until you have successfully completed Exercise UP-01. Attempting to progress without a good understanding of the solutions needed in exercise UP01 is likely to be expensive on your time and efforts.

Task for the Exercise:

System calls are user program's requests to the kernel to perform some service. User programs are prevented from performing these activities directly because their actions may affect the other processes (user programs and even the kernel) running on the system. The kernel provides the coordinating authority to oversee these activities.

A consequence of this restrain is that the kernel must work in a robust isolation from the user program that has requested a service. Thus, kernel and user stacks are separate entities.

At the same time, as a kernel provides the requested services to the user programs it needs to receive the system-call requests and their arguments. Further, the kernel needs a way to communicate the results and service status information back to the user programs (or processes). So we need communication channels between the kernel and the user virtual address spaces. To understand the issues at hand, these sections in PintDoc are definitely your essential readings before you start work on this exercise:

- Section 3.1.5 *Accessing User Memory*,
- Section 3.4.2 *System Calls FAQ*,
- Section 3.5.2 *System Call Details*,
- Section 3.3.2 *Process Termination Messages*,
- Appendix A.4 *Interrupt Handling*,
- Appendix A.4.1 *Interrupt Infrastructure*,
- Appendix A4.2 *Internal Interrupt Handling*

It is important that you take advantage of the program pattern that PintOS code (See function `run_actions()` in file `threads/init.c`) uses to call functions. The pattern maintains an array of pointers to functions. The system call number is used as index in the array to jump to the appropriate function implementing the system call. This is not just efficient, it is less vulnerable to errors and eases the task of adding new functions later. The pattern also keeps the implementation of each system call in a separate function. Function `syscall_handler()` in file `userprog/syscall.c` referred to in Section 3.3.4 *System Calls* becomes a convenient arrival and departure point for all system calls.

Your first and very easy task is to create the skeleton functions for each of the system call mentioned in User Program project and provide a way to call the right function for each system call number for which a request arrives at function `syscall_handler()` in file `userprog/syscall.c`. Each of these functions can initially be set to mirror the older behavior of function `syscall_handler()`.

Goal for the current exercise is to implement and have (at least) the following system calls (see section 3.3.4 S*ystem Calls*) running: `halt()`, `exit()`, and `write()` on `stdout` (fd = 1). This will give you at least 10 working test cases from `make check` command.

The write system call requires your implementation to return a value back to the user program. The task needs an arrangement that would need a bit of your attention.

System call `halt()` is easy to implement once you work out a small correction you need to make in the specification given on page 29.

System call `exit()` requires whole of the advice given in Section 3.2 *Suggested Order of Implementation* and a bit more.

Our advice is to implement function `process_wait()` in file `process.c` to work as follows:

> While the thread associated with `child_tid` is not in state `THREAD_DYING` call `thread_yield()` to avoid spin wait. However, you need a small but subtle change in the provided kernel code to have this part working. This change relates to an allocated resource and when it is safe to release the resource. For the present, we may decide to keep it instead of deallocating the resource. This decision leads to what is described as resource leak as the resource is not being returned to the pool of available resources.
>
> In the last exercise of project *User Program* (UP04) we will revisit the issue of the resource deallocation more earnestly. Unallocated (or leaked) resources degrade the Kernel's ability to host multiple threads/processes over time. Every allocated resource need to be deallocated if the kernel must sustain its vitality indefinitely.

Many a times, PintDoc makes an important suggestion without stressing its importance. The second paragraph after description of system call `close` on page 32 is a good advice. It says that the addresses provided to the system calls by the user program must be carefully checked. PintDoc recommendation is to do these sanitization checks now.

A sanitation check has two parts. First, determine that the address is a valid user virtual address. Second part is to ensure that the address does translate into a real physical address. If either condition is violated, the address is a likely source of trouble for the kernel integrity and stability.

You must create handy functions in file `syscall.c` to test the validity of the addresses passed as arguments in the system calls. The arguments to be tested for valid address and nature of the test depends on the system call. It suffices at this stage to only exercise this test for the system calls you have implemented in this exercise.

You may continue to implement other system calls related to files if you wish. Or you may wait for the next exercise. We recommend that you leave system calls `exec()`, `wait()` for the last exercise. This also may apply to the task described in Section 3.3.5 *Denying Writes to Executables*.

Our experiences with `make check`

Once again, we have tried to revert our implementation to the stage at the completion of this exercise after we completed the project. Therefore, the tests that are passed and not passed by your implementation of the exercise may not exactly match the results printed below.

```
[vmm@progsrv build]$ make check
pass tests/userprog/args-none
pass tests/userprog/args-single
pass tests/userprog/args-multiple
pass tests/userprog/args-many
pass tests/userprog/args-dbl-space
pass tests/userprog/sc-bad-sp
pass tests/userprog/sc-bad-arg
pass tests/userprog/sc-boundary
pass tests/userprog/sc-boundary-2
pass tests/userprog/halt
pass tests/userprog/exit
FAIL tests/userprog/create-normal
pass tests/userprog/create-empty
FAIL tests/userprog/create-null
FAIL tests/userprog/create-bad-ptr
pass tests/userprog/create-long
FAIL tests/userprog/create-exists
FAIL tests/userprog/create-bound
FAIL tests/userprog/open-normal
FAIL tests/userprog/open-missing
FAIL tests/userprog/open-boundary
FAIL tests/userprog/open-empty
pass tests/userprog/open-null
FAIL tests/userprog/open-bad-ptr
FAIL tests/userprog/open-twice
FAIL tests/userprog/close-normal
FAIL tests/userprog/close-twice
pass tests/userprog/close-stdin
pass tests/userprog/close-stdout
pass tests/userprog/close-bad-fd
FAIL tests/userprog/read-normal
FAIL tests/userprog/read-bad-ptr
FAIL tests/userprog/read-boundary
FAIL tests/userprog/read-zero
pass tests/userprog/read-stdout
pass tests/userprog/read-bad-fd
FAIL tests/userprog/write-normal
FAIL tests/userprog/write-bad-ptr
FAIL tests/userprog/write-boundary
FAIL tests/userprog/write-zero
pass tests/userprog/write-stdin
pass tests/userprog/write-bad-fd
FAIL tests/userprog/exec-once
FAIL tests/userprog/exec-arg
FAIL tests/userprog/exec-multiple
FAIL tests/userprog/exec-missing
pass tests/userprog/exec-bad-ptr
FAIL tests/userprog/wait-simple
```

```
FAIL tests/userprog/wait-twice
FAIL tests/userprog/wait-killed
pass tests/userprog/wait-bad-pid
FAIL tests/userprog/multi-recurse
FAIL tests/userprog/multi-child-fd
FAIL tests/userprog/rox-simple
FAIL tests/userprog/rox-child
FAIL tests/userprog/rox-multichild
pass tests/userprog/bad-read
pass tests/userprog/bad-write
pass tests/userprog/bad-read2
pass tests/userprog/bad-write2
pass tests/userprog/bad-jump
pass tests/userprog/bad-jump2
FAIL tests/userprog/no-vm/multi-oom
FAIL tests/filesys/base/lg-create
FAIL tests/filesys/base/lg-full
FAIL tests/filesys/base/lg-random
FAIL tests/filesys/base/lg-seq-block
FAIL tests/filesys/base/lg-seq-random
FAIL tests/filesys/base/sm-create
FAIL tests/filesys/base/sm-full
FAIL tests/filesys/base/sm-random
FAIL tests/filesys/base/sm-seq-block
FAIL tests/filesys/base/sm-seq-random
FAIL tests/filesys/base/syn-read
FAIL tests/filesys/base/syn-remove
FAIL tests/filesys/base/syn-write
47 of 76 tests failed.
make: *** [check] Error 1
[vmm@progsrv build]$
```

Contributing Authors:

Vishv Malhotra, Gautam Barua, Rashmi Dutta Baruah