PintOS System Calls

Introduction

System calls are an interface to kernel services that the kernel provides to user-space programs. Normally, system calls should be used only in user-space code, while kernel code should use other ways to access the same services. You have already implemented systems calls in the Ubuntu environment.

On most CPU architectures, system calls are implemented by placing the arguments to the system call function on the stack or in predefined registers and then raising a software interrupt, that transfers control to the kernel code. The kernel code then interprets the arguments to the system call function and provides the requested service.

In this assignment, you will understand the interface between user-space and kernel-space programs. In addition, you will also be able to extend a kernel interface with new services.

Assignment

We will do this in two parts --- in Part A we will add simple calls and in the next the calls to enable running of user programs.

Part A

In this part, you should implement the following system calls:

- `bool create (const char *file, unsigned initial_size)` - create a file with a given name and size;
- `bool remove (const char *file)` - delete a file with a given name;
- `int open (const char *file)` - returns a file handle to a file identified by its name;
- `void close (int fd)` - closes a file identified by a file handle;
- `int read (int fd, void *buffer, unsigned length)` - reads a requested amount of bytes from a file identified by a handle;
- `int write (int fd, const void *buffer, unsigned length)` - writes a requested amount of bytes from a file identified by a handle;
- `void seek (int fd, unsigned position)` - moves the cursor to a given position inside a file;
- `int filesize (int fd)` - returns the file size in bytes;
- `void halt(void)` - shuts down the operating system.

Task: Read sections 3.1, 3.3.4 and 3.5 from the PintOS documentation.

First, we compile a user-space program that exercises these system calls. The source of the program is in `src/examples/test-syscalls.c`. We build that program by running:

```
$ cd src/examples

$ make
```

which produces an executable, `src/examples/test-syscalls`. To run that executable, do

```
$ cd src/userprog
$ make
$ pintos -k -v -T 60 --qemu --filesys-size=4 -p \
     ../examples/test-syscalls \
     -a t1 -- -q -f ls run t1
```

The output of the emulator also contains the output of program t1, starting after a line containing the string Executing 't1':.

Task: Identify the output of the program (if any) and take note of it.

The user-space implementation of system calls is provided in src/lib/user/syscall.c. Study this implementation as well as the implementation of the syscall macros in the same file.

Task: Provide a written description of the syscall macros. What are they doing? What are their arguments? How are they placed on the stack?

Before we start implementing the handling of system calls, let's locate the syscall_handler function and study its contents. For now, it should print an error message and then exit. However, that error string is not to be found in the output of the emulator. That is because the kernel does not wait for the process running the program to exit.

Task: Locate the function where the kernel waits for a process to terminate[1] and provide an implementation that never exits[2]. Now, after you are done implementing the waiting function, run the emulator again and look for the output of the t1 program. Make sure that the error message from syscall_handler is printed at least once.

Now, we continue with the implementation of the system call handlers for the required system calls.

Task: Provide the kernel-space implementation of system calls by implementing the function static void syscall_handler(struct intr_frame *f), defined in src/userprog/syscall.c. We recommend that you implement each system call in a separate function and just use the syscall_handler function to dispatch to the proper implementation. You should start by implementing the write system call, in particular for the case when fd == STDOUT_FILENO. This enables the use of printf is user-space code and thus facilitates debugging. Some system calls receive pointers as arguments, so the system call handler must ensure that those pointers point into the memory that is accessible for the user. This is achieved by calling the validate_user_addr_range function.

Observe that the argument to the syscall_handler is a pointer to a struct intr_frame. This structure contains information about the CPU state at the moment when the interrupt occurred. For implementing system calls, you will need to read the arguments starting from the address in f->esp, while a return value, if it exists, has to be written to f->eax.

In the figure below, the memory layout for the arguments to the create system call is given. These arguments are made accessible to the system call handler at the address f->esp. The first word (4 bytes) represents the system call number (no), the second word, a pointer to a null-terminated array of characters (fn), while the third word is an integer representing the size of the file.

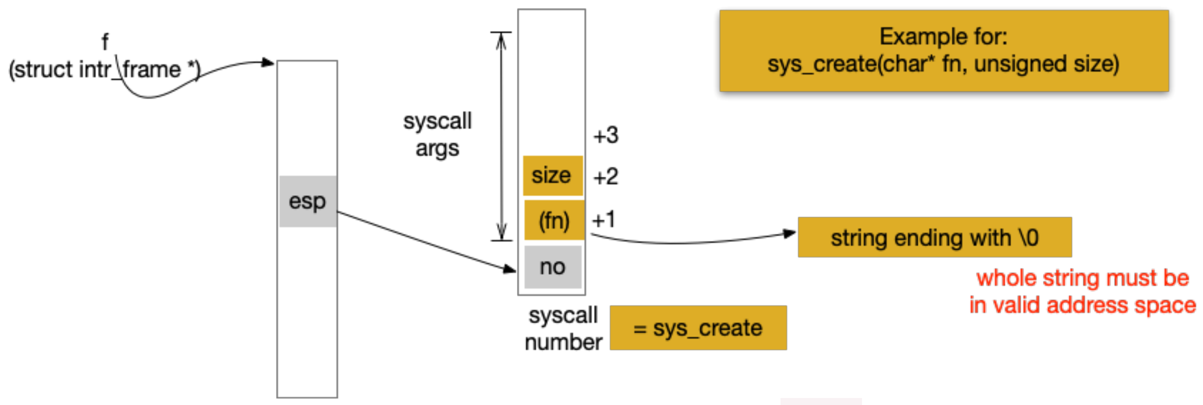


Fig: Memory layout of the arguments for the create system call

To avoid the use of pointer arithmetic to access these values, we recommend that you overlay a structure at the address f->esp. The members of this structure depend on the system call that is handled. For the create system call this can be achieved as in the following code snippet:

```
struct create_args {
    int id;
    const char *file;
    unsigned initial_size;
};

struct create_args *args = (struct create_args *) f->esp;
```

Task: You may have observed that there is no test code for filesize, remove, seek and halt. Extend the test in src/examples/test-syscalls.c to demonstrate the these system calls work as expected.

Hand-in

- Answers to the questions listed above.
- The modified PintOS source code that is passing the tests implemented in src/examples/test-syscalls.c.

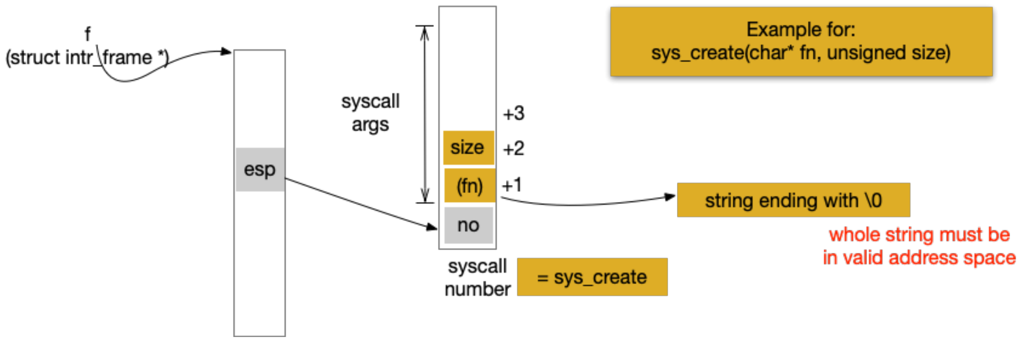
Hints:

- This assignment does not require writing any assembly code.
- You do not have to use any containers (e.g. lists, hash sets) to solve this exercise; simple structs and arrays are enough.
- The estimated time needed to solve this exercise is 16h.
- The approximate size of the code changes is:

```
src/examples/test-syscalls.c |  33 ++++++++++++++++--
```

```
src/threads/thread.c          | 69 +++++++++++++++++++------------------
src/threads/thread.h          | 13 ++++++++
src/userprog/Make.vars        |  4 +-
src/userprog/process.c        | 69 +++++++++++++++++++------------------
src/userprog/syscall.c        | 207 +++++++++++++++++++++++++++++++++++++++++++++++
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++----
```

---

1. In src/userprog/process.c.
2. Hint: a simple infinite loop will do.
3. 

Part B

Introduction

One of the main functionalities provided by operating system is the management of user processes. To achieve this in PintOS, you will implement three system calls exec(), wait() and exit().

After completing this lab, you will have an understanding on how an operating system manages user processes.

In this part, you continue the work on implementing system calls, that you have started in the previous part. The system calls you need to implement are:

- pid_t exec (const char * cmd_line) - runs the executable given in cmd_line, providing any existing arguments;
- void exit (int status) - terminates the current process and returns its status to the kernel;
- int wait (pid_t child) - waits for a child process to finish and returns its exit status.

Before proceeding, re-read sections 3.1, 3.3.4, 3.5, A.2 and A.3 from the PintOS documentation.

The main building directory for this assignment is src/userprog/build. To run the tests, run make check in this directory. All of them are expected to fail, since nothing is implemented.

In this task there are two situations when the execution of a thread cannot proceed until an event has happened in another thread. This synchronization between threads is achieved using semaphores. PintOS already provides a semaphore implementation, declared in src/thread/synch.h.

Task Study the interface of the semaphore function and describe the sequence of calls required for a thread to be able to wait for an event that happens in another thread.

Task: The first step is to implement the exec() system call.

This makes use of the process_execute() function, defined in src/userprog/process.c. This function is already used by the operating system entry point to run commands that are given as arguments to the kernel. The major difference between the existing implementation of process_execute() and the requirements for the exec system call is that the process_execute() may return before the executable program is loaded.

The current implementation of process_execute() calls the thread_create() function, which, among other things, creates a new kernel thread and schedules it for execution. When this thread is scheduled, its execution starts with the start_process() function. Observe that process_execute() and start_process() are running on different threads and we want to block the execution of process_execute() until start_process() has attempted to load the executable from disk (i.e. the call to load() has returned).

To achieve this synchronization, we need a semaphore in the process_execute() function and pass a pointer to it to the thread create function. Observe that any pointer that we pass in the aux argument of thread_create() is forwarded as the load_p argument of the start_process() function. So instead of passing only a pointer to the executable name, we can use this argument to pass a pointer to a struct that contains the executable name, a pointer to the semaphore and other information that may be useful.

Task: Implement the wait() and exit() system calls. When implementing the wait() call we have to consider that the call to this function has to block the calling process until the waited-for process has exited.

To implement the required system calls, we need a way to keep track for each thread what its children and parent are. For this purpose, fields can be added as needed to struct thread. In addition, we also need a way to pass the exit code of a thread to its parent. In the current implementation the memory backing the thread struct is deallocated when the thread is dying, in thread_schedule_tail(). A good idea is to keep this structure around until any waits in the parent are dealt with. This can be implemented by adding a list of dying child-threads in struct thread and moving the dying thread to this list instead of deallocating the memory immediately.

Hand-in

- Answers to the questions listed above.
- The modified PintOS source code that is passing all the make check tests, except the following:

```
FAIL tests/userprog/rox-simple
FAIL tests/userprog/rox-child
FAIL tests/userprog/rox-multichild
FAIL tests/userprog/bad-read
FAIL tests/userprog/bad-write
FAIL tests/userprog/bad-read2
FAIL tests/userprog/bad-write2
FAIL tests/userprog/bad-jump
FAIL tests/userprog/bad-jump2
FAIL tests/userprog/no-vm/multi-oom
```

Hints

- This assignment does not require writing any assembly code.
- The data structures needed to solve this exercise are lists, arrays and structs. In particular, you will not need to use hash maps.
- The estimated time needed to solve this exercise is 12h.
- The approximate size of the code changes is:

```
src/threads/thread.c     | 24 ++++++++++++++++++++++++
src/threads/thread.h     | 13 +++++++++++++
src/userprog/Make.vars   |  4 ++--
src/userprog/exception.c |  3 +++
src/userprog/process.c   | 125 +++++++++++++++++++++++++++++++++++++++++++
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
++++++++++++++++++
src/userprog/syscall.c   |  69 ++++++++++++++++++++++++++++++++++++++++++++
+++++++++++++++++++++++--
```

In the demo, you will have to show the code working on your own machine/VM.