Process Control Syscalls

TABLE OF CONTENTS

Pintos currently only supports one syscall, `exit`, which terminates the calling process. You will add support for the following new syscalls: `practice`, `halt`, `exec`, `wait`.

Implementation details

To implement syscalls, you first need a way to safely read and write memory that's in a user process's virtual address space. The syscall arguments are located on the user process's stack, right above the user process's stack pointer. You are not allowed to have the kernel crash while trying to dereference an invalid or null pointer. For example, if the stack pointer is invalid when a user program makes a syscall, the kernel ought not crash when trying to read syscall arguments from the stack. Additionally, some syscall arguments are pointers to buffers inside the user process's address space. Those buffer pointers could be invalid as well.

Try to avoid writing large amounts of repetitive code for implementing system calls. Each system call argument, whether an integer or a pointer, takes up 4 bytes on the stack. You should be able to take advantage of this to avoid writing much near-identical code for retrieving each system call's arguments from the stack.

You will need to gracefully handle cases where a syscall cannot be completed due to invalid memory access including null pointers, invalid pointers (e.g. pointing to unmapped memory), and illegal pointers (e.g. pointing to kernel memory). Beware: a 4-byte memory region (e.g. a 32-bit integer) may consist of 2 bytes of valid memory and 2 bytes of invalid memory, if the memory lies on a page boundary. You should handle these cases by terminating the user process. **We recommend testing this part of your code before implementing any other system call functionality**. See Accessing User Memory and Syscalls for more information.

Syscall signatures

practice

```
int practice (int i)
```

A "fake" syscall designed to get you familiar with the syscall interface This syscall increments the passed in integer argument by 1 and returns it to the user.

halt

```
void halt (void)
```

Terminates Pintos by calling the `shutdown_power_off` function in `devices/shutdown.h`. This should be seldom used, because you lose some information about possible deadlock situations, etc.

exit

```
void exit (int status)
```

Terminates the current user program, returning status to the kernel. If the process's parent waits for it (see below), this is the status that will be returned. Conventionally, a status of 0 indicates success and nonzero values indicate errors. Every user program that finishes in normally calls `exit` – even a program that returns from `main` calls `exit` indirectly (see Program Startup). In order to make the test suite pass, you need to print out the exit status of each user program when it exits. The format should be `%s: exit(%d)` **followed by a newline**, where the process name and exit code respectively subsitute `%s` and `%d`. You may need to modify the existing implementation of `exit` to support other syscalls.

exec

```
pid_t exec (const char *cmd_line)
```

Runs the executable whose name is given in `cmd_line`, passing any given arguments, and returns the new process's program id (`pid`). If the program cannot load or run *for any reason*, return -1. Thus, the parent process cannot return from a call to `exec` until it knows whether the child process successfully loaded its executable. You must use appropriate synchronization to ensure this.

Keep in mind `exec` is different from Unix `exec`. It can be thought of as a Unix `fork` + Unix `exec`.

wait

```
int wait (pid_t pid)
```

Waits for a child process `pid` and retrieves the child's exit status. If `pid` is still alive, waits until it terminates. Then, returns the status that `pid` passed to exit. If `pid` did not call `exit` but was terminated by the kernel (e.g. killed due to an exception), `wait` must return `-1`. It is perfectly legal for a parent process to wait for child processes that have already terminated by the time the parent calls wait, but the kernel must still allow the parent to retrieve its child's exit status, or learn that the child was terminated by the kernel.

`wait` must fail and return `-1` immediately if any of the following conditions are true:

- `pid` does not refer to a direct child of the calling process. `pid` is a direct child of the calling process if and only if the calling process received `pid` as a return value from a successful call to `exec`. Note that children are not inherited: if A spawns child B and B spawns child process C, then A cannot wait for C, even if B is dead. A call to `wait(C)` by process A must fail. Similarly, orphaned processes are not assigned to a new parent if their parent process exits before they do.

- The process that calls `wait` has already called `wait` on `pid`. That is, a process may wait for any given child at most once.

Processes may spawn any number of children, wait for them in any order, and may even exit without having waited for some or all of their children. Your design should consider all the ways in which waits can occur. All of a process's resources, including its `struct thread`, must be freed whether its parent ever waits for it or not, and regardless of whether the child exits before or after its parent.

You must ensure that Pintos does not terminate until the initial process exits. The supplied Pintos code tries to do this by calling the `process_wait` function in `userprog/process.c` from the `main` function in `threads/init.c`. We suggest that you implement the `process_wait` function in `userprog/process.c` according to the docstring and then implement `wait` in terms of `process_wait`.

Implementing wait requires considerably more work than the other syscalls.
