

Lab 1: Fun with system calls

Handed out Monday July 1, 2024

Due Friday July 19, 2024

Introduction

In this assignment, you will design a new system call for xv6, and use it in some fork synchronization problems. The goals of this assignment are:

- Understand the system call interface
- Understand how user programs send parameters to the kernel, and receive values back
- See how the event handling in the Operating System works
- Understand the process structure and modify it

The deliverables for the assignment are the code that includes the required changes (the new system call, and modifications to `exit`), as well as the user test program(s). You will be graded at least partially by interview and expected to demonstrate that you understand what you did and why you did it that way for the full project (including the part done by other partner).

Assignment

Part 1 (25 points): Add a new system call `getsiblings()`. This system call takes no arguments, and prints the id of any sibling processes of the calling process. Sibling processes are those that share the same parent.

Part 2 (20 points): Extend the current xv6 process implementation to maintain an exit status. To get this done, add a field to the process structure (see `proc.h`) in order to save an exit status of the terminated process. We will need this for implementing `wait`. Next, you have to change all system calls affected by this change (e.g., `exit`, `wait` etc.).

Part 3 (20 points): Change the `exit` system call signature to `void exit(int status)`. The `exit` system call must act as previously defined (i.e., terminate the current process) but it must also store the exit status of the terminated process in the corresponding structure. In order to make the changes in `exit` system call you must update the following files: `user.h`, `defs.h`, `sysproc.c`, `proc.c` and all the user space programs that uses `exit` system call. Note, from now on, all user space programs must supply an exit status when terminated.

Hassle: one hassle that this change (and the one in part b below) introduces is that all existing places that used `exit()`, including ones that are in test programs have now to be changed to use the new prototype. You can either do that yourself (e.g., use `grep` to find all locations of this call and change them), or create a new `exit` call to match the new prototype.

Goals of this part of the assignment: Get familiar with system call arguments and how arguments are passed given the presence of two stacks (user mode and kernel mode). Understand the backward compatibility hassles that come from modifying the system call prototype. Carry out a gentle modification to an existing system call and to the Process Control Block (PCB), which will be needed by the next part of the Lab.

Part 4 (20 points): Update the `wait` system call signature to `int wait(int *status)`. The `wait` system call must prevent the current process from execution until any of its child processes is terminated (if any exists) and return the terminated child exit status through the `status` argument. The system call must return the process id of the child that was terminated or -1 if no child exists (or unexpected error occurred). Note that the `wait` system call can receive `NULL` as an argument. In this case the child's exit status must be discarded.

Goal of this part of the assignment: Continue to get familiar with system call arguments, in this case with how to return a value.

Part 5 (15 points + 10 points bonus) Add a `waitpid` system call: `int waitpid(int pid, int *status, int options)`. This system call must act like `wait` system call with the following additional properties: The system call must wait for a process (which must be one of your child processes) with a `pid` that equals to one provided by the `pid` argument. The return value must be the process id of the process that was terminated or -1 if this process does not exist, if it is not a child of the process that called the `waitpid`, or if an unexpected error occurred. We are required only to implement a blocking `waitpid` where the kernel prevents the current process from execution until a process with the given `pid` terminates. In other words, you do not need to worry about the `options` field for now.

Write an example program to illustrate that your `waitpid` works. You have to modify the makefile to add your example program so that it can be executed from inside the shell once `xv6` boots.

Help, hints, questions to explore, etc...

We will be extending this section based on how I see progress (for example, to help you understand the code base, or provide ideas to make you more effective in working with the code base).

- **Investing in learning gdb will really make your life easier this quarter!** There are only a few commands that you need to learn to get started.
 - `break` places a break point which causes the program to stop when it reaches that point. There are many ways to set a break point; I like to use `break` function to cause a stop at the function that I want to look at.
 - `continue` causes `gdb` to continue execution until the next break point. If you want to execute only one line of the code use `next` or `step`. The difference between the two is if the next line contains a function call, `next` will execute the full call before stopping while `step` steps into the function and stops at the first line there.
 - `print` allows you to examine a variable. You can use some of the C semantics here, for example to dereference a pointer and see the memory location its point to (e.g., `print *p`), to print a specific format such as string or hex (e.g., `print %x p`). `display` is like `print` but it repeats the print every time `gdb` stops (e.g., after a break point or after a `step` or `next`)
 - `list` shows the code around the area you are.

You can find more `gdb` help in the `gdb` resources linked on the class page under resources.

- Another really useful tool is `grep`. You can `grep` for a keyword (or if you are adventurous, a regular expression) in a file or group of files to find if/where it exists. So, let's say I am interested in a specific function or variable and want to see where it is used. In the `xv6` directory, you can type: `grep wait *.c` to search for the string in all files ending with `.c`

There are a couple of other useful options for `grep`. The `-i` option (e.g., `grep -i wait *.c` will make `grep` ignore case. The `-v` option excludes a pattern. So, let's say you want to search for `wait`, but not `waitpid`. One way you can do it is `grep wait *.c | grep -v waitpid`

- **Background/deeper understanding:** When any event in `xv6` occurs, the hardware switches mode to the operating system, and jumps to the code in `trapasm.S` (starting at label `alltraps`). This code builds the trap frame (first segment), and then sets up the memory segments for the kernel (second segment--unimportant for now).

The trap frame is a data structure built on the stack that is used to pass some important arguments to the trap handler including pointers to the user stack to enable getting arguments to system calls. Note that the user stack and the kernel stack for a process are separate. We discussed why in class. So, when we push

arguments for a system call on the user stack, we need to give a pointer to the kernel to be able to access them.

You can see the trapframe structure in `x86.h` -- its ok to abstract it away since we will not be working with it in detail.

After filling parts of the trapframe, it calls `trap()` with the trap frame as argument, which takes us to `trap.c`

Here you find a big switch statement based on the `trapno` that caused the trap (this is stored in the trap frame). Each case represents an event and implements its trap handler. Only a few events are currently supported such as system calls and the timer interrupt which are necessary for the barebones `xv6` to run.

If you look under the system call case (the first case), you see some sanity checks (OS code is typically paranoid to avoid kernel panics), you see that we eventually call `syscall()` which is the top level handler for system calls. Lets follow `syscall` which is located at the bottom of `syscall.c`

We get the system call number from the `eax` register in the user code (saved on the trapframe) and use it to index into the system call table and pick up the appropriate handler to call. The return value from the system call handler is stored in the register `eax` in the trapframe (which is used by convention to store return values in Linux/x86) to provide the return value back to the user.

The handlers are all of the same function type (they are all called `sys_xyz` where `xyz` is the system call they handle) which enables us to use this trap handler table and call any one of them as appropriate.

Lets look at the implementation of one of these, lets say `sys_kill()` which passes a signal to a process, often to kill it.

`sys_kill()` and several other handlers are implemented in `sysproc.c` because they have to do with processes. Other handlers are implemented in the file system code, or memory code as appropriate to their operation. Use `grep` to find them if you can't figure out where they are.

You will note that `sys_kill` eventually calls `kill` where the real implementation of the system call is. This enables us to pass different parameters to each system call to get around the fact that the `sys_` handler are all the same type.

Note also that `sys_kill` had to get those parameters from the user stack. To help with this low level read, there are a number of accessor functions (`argint`, `argstr`, etc...) defined to get access to the arguments based by the user by using information in the trapframe.