Computer Science 161: Operating Systems
ASST2: Processes, System Calls

CS161 Course Staff
cs161@eecs.harvard.edu

February 18, 2013

1 Administrivia

- *Tuesday in-class*: Peer design review. Please bring in a printed copy of your rough draft.
- *Thursday 9:OOPM*: Design documents due.
- *Friday, March 7th, 5:OOPM*: ASST2 due. Do not leave this until the last minute; it is more work than ASST1.

2 Process Basics

Process states:

- `S_RUN`
- `S_READY`
- `S_SLEEP`
- `S_ZOMB`

What do you think a zombie is? Where are these constants defined in OS/161?

Where does the switching happen? Look in *thread_switch* and *switchframe_switch*. Here we have the machine-independent and the machine-dependent switch functions.

How to use processes:

```
if (fork() == 0) {
/* We're in the child */
} else {
/* We're the parent */
}
```

Let's now get into the details of assignment 2.

3 File Descriptors and System Calls

You are tasked with implementing system calls that allow access to the file system. It should be noted that these file system calls are not implementation dependent – they should work with any file system, due to the higher-level VFS layer. The file system related calls you need to write are: `open()`, `read()`, `write()`, `lseek()`, `close()`, `dup2()`, `chdir()`, and `getcwd()`.

The first thing that needs to be designed is the per-process file table. The file descriptor that you hand back to an application is just the index into the process's file table. You need to decide what goes into this file table. A familiarity with the VFS layer is a must before designing this. There are a number of things to note here:

- How are open files represented in user space? File descriptors.
- There are three standard file descriptors, `STDIN`, `STDOUT`, and `STDERR`. To what device should these be initially attached?
- File descriptors are indexes into the file handles. How do you want to assign new file descriptors, and should there be a limit to the number of file descriptors? How do you assign file descriptors?
- Each file descriptor has an associated file offset. What happens with the offset if you open the same file twice?
- How should you convert between file descriptors and more meaningful information? How are open files represented in the kernel? `vnode`. Keep track of this correspondence using the file table.
- The file table helps associate open file descriptors with the corresponding `vnodes`, keeps track of the current file offset.
- What happens to a file table when you fork a process, or when the process exits? Each process has one file table, thus it is copied the a process forks. What happens to the seek position when you:
  - `dup2()`?
  - `fork()`?

  - open the same file twice?

- When a process exits, its a good idea to close all its open files (be careful here. What if the process has forked?).

The system calls you need to implement are:

- `open(const char *filename, int oflag, mode_t mode)` - should be simple thanks to the `vfs` layer, plus some bookkeeping.
- `dup2(int oldfd, int newfd)` - if `newfd` is already opened, close it. Upon successful completion, both file descriptors refer to the same file table object and share all properties of the object.
- `close(int fd)` - might be a little bit more interesting. Why? Refcounting (see dup2).
- `lseek(int fd, off_t pos, int whence)` - can we always perform an lseek? What if you lseek() beyond the end of the file? Is that legal? Use `VOP_TRYSEEK`. We've also added 64-bit values this year for `offset` and the value returned by `lseek()`. Be sure to think about the code-reading question and make sure you're confident in your answer, before coding `lseek()`.
- `read(int fd, void *buf, size_t nbytes)` - you will want to use `struct uio`s, so make sure you understand them. You may want to check out `VOP_READ`.
- `write(int fd, const void *buf, size_t nbytes)` - involves I/O to userland. Again, `struct uio`s will be helpful, as will `VOP_WRITE`.
- `getcwd(char *buf, size_t size)` - this should be very straightforward.
- `chdir(const char *pathname)` - again, a lot of the work has been done for you already.

4 Process Management System Calls

The process management system calls are probably the harder part of this assignment, so we'll just delve right in.

- What is a process and how does it relate to an os161 thread?
- `fork()` -
  - As you've learned, this system call creates an exact copy of the process that calls it. What does that entail?
  - How would you copy the file table? What happens to open descriptors?

  - You need to duplicate the address space. Look through the dumbvm code for a function that might be quite useful.
  - Is there anything else you will need to replicate/copy? Current working directory.
  - The new process should get a PID. Here `fork()` needs to abide by the semantics of your PID dispensing mechanism.
  - The trickiest part is making the child return 0 and behave exactly like the parent. It will be useful to understand the machine dependent system call mechanism for return values and errors in `kern/arch/mips/mips/syscall.c`. You will probably want to do something similar in `md_forkentry()`. This is very subtle – think about this and be sure to address it in your design document.
  - Trapframe handling – this can be tricky so be sure to discuss it in your design. When a process makes a system call, where how does it know where to return? - It saves a return address on the trapframe. Therefore, the trapframe also needs to be copied. (Otherwise, child process would not know where to return.)
  - Return 0 to the child, process id of the child to the parent. How to return a different value to the child process? Look at how other system calls return and fiddle with the trapframe appropriately. This is machine-dependent code. Place it into an appropriate directory. Look at `md_forkentry`. The skeleton is conveniently provided.

- `getpid()` - Very easy to implement and it doesn't even fail.
- `waitpid(pid_t wpid, int *status, int options)` and `_exit()`
  - These system calls are very closely tied to your PID management system. It's a synchronization problem.
  - How are you going to assign PIDs? You shouldn't just run out. PID recycling?
  - Be very specific when you define the semantics of your `waitpid()` and `_exit()` interactions. The man pages give the minimum requirements. Note that when a process `_exits()` its PID may not be given to a new process right away since the parent might be interested in finding the exit code. What if the parent has already exited itself?
  - What does *interested* mean? Unix defines it strictly in terms of parent/child. The parent can get the child's exit status.
  - You may wish to implement `WNOHANG` for `waitpid()`, which allows `waitpid()` to be non-blocking.
  - What PID related data structures are you going to keep in the parent and in the child? Do you need to synchronize the access to those structures and if yes then how?

- How can you make a parent wait for a child? What happens if a child tries to wait for its parent?
- How can you deadlock? (You shouldn't, of course.) - Two processes waiting for each other.
- `_exit()` releases all resources used by the process. What are these? Do we always free *all* resources? What about the exit code? What happens if a child exits before its parent or before any other process that has "expressed interest" in the exit status?
- Don't forget `kill_curthread()`

- `exec(const char *path, const *char argv[])` –
  - The idea is that we want to load a new executable in the address space of a process.
  - `execv()` is quite similar to `runprogram()` in `kern/userprog/runprogram.c`.
  - Load the executable. Let's see what `runprogram` does.
    * It opens the file.
    * Creates an address space into which the image would be loaded and activates it. (Don't worry too much about what this does until assignment 3, but remember to do it).
    * Loads the executable into that address space.
    * `load_elf` returns `entrypoint`. What is this?
    * Define user stack.
    * Return to user mode.

    You can just follow all these steps but before 6, you need to `copyout` arguments to the user stack. Anything else? What about the old address space?
  - Coping with the argument vector is the hard part.
  - Where do we get the arguments from the old program? They are user-level pointer that are passed as arguments to the system call. How can you get hold of them? `copyin` for pointers, `copyinstr` for strings. You need to `copyin` both the pointers and the strings. Where in the process's address space should we put the argument vector? On the stack. Where do we get the arguments from the old program? They are a user-level pointer that are passed as arguments to the system call.
  - What is the stack? It's just a region of memory in the address space. The stack is used for temporary data during function calls. Why do we need it? To make efficient use of memory. Now getting the arguments into place is basically the opposite of getting them from user space. You will again need `copyin/copyout`. Place things wherever you want, but above the stackpointer. The stackpointer is where the process will start scratching on the stack.

  - Note that the argument vector comes from user space. The functions in `kern/lib/copyinout.c` will be very useful in copying everything to/from the kernel address space. Why is this always important? Consider this example. User Linus has a secret file. He's been using it, so it's buffered in memory. Malicious user Mal wants to get to it, and he has a pretty good guess where it's buffered. Malicious Mal can pass a kernel address to open with `O_CREAT`. Whoops, the filename appears with the contents of the file. This is **bad**, which is why we can't trust any thing coming from userland. Thus, `copyin/copyout`.
  - Each of the elements of the argument vector (`argv[i]`) is a string residing in userland and needs to be copied with care.
  - What happens to `argv[i]` in the new address space?
  - Make sure you null-terminate `argv` (i.e. `argv[argc] = NULL`).
  - Make sure that all of your pointers are word-aligned.
  - Don't forget to set up `stdin`, `stdout` and `stderr` (also in `runprogram`).
- Remember to handle all the corner cases. Can you think of some good ones for these system calls?

Since handling arguments in `execv` is tricky, lets do an example. We need to place `ls foo` on the user stack. This is what it should look like.

800	
799	∅
798	o
797	o
796	f
795	[padding]
794	∅
793	s
792	l
791	∅
790	∅
789	∅
788	∅ [null-terminate]
787	argv[1]
786	argv[1]
785	argv[1]
784	argv[1] = 796
783	argv[0]
782	argv[0]
781	argv[0]
780	argv[0] = 792 = stackptr

Why is there nothing in location 795? Because pointers have to be word-aligned (word = 4 bytes). Why? CPUs are designed this way. So make sure all your pointers are aligned on a 4-byte boundary. Otherwise your user program will crash, and you won't know why.

Why are 4 locations occupied by `argv[0]` and `argv[1]`? Because pointers are 4 bytes long.

Hopefully this has made things a little clearer.

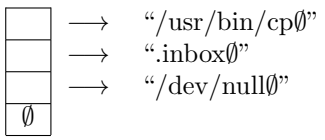

Figure 1: An illustration of an example `char **argv`, where ∅ represents the traditional C `NULL` value.

5 Scheduler

You must implement a scheduler to complement the one we have provided for you. You may choose whatever scheduling algorithm you'd like. Choices include a round robin with priority, multi-level feedback queue scheduler, fair-share, a lottery scheduler, a priority scheduler, and a random scheduler. Keep the original round-robin scheduler, it can come in handy for testing etc.

You should provide a mechanism for switching between the schedulers. You can have them chosen at compile time with some `#define` directives. You can use the config system to set these `#defines`. Switching schedulers at runtime is neat but impractical and usually not worth the trouble.

Key Metrics:

CPU Utilization % of time that the CPU is running threads

CPU Throughput # jobs per second

Turnaround time {End Time} - {Start Time}

Response time total time jobs spend on ready queue

Waiting time total time jobs spend on wait/sleep queue

The main goals:

Efficiency Efficieny of resource utilization - Adjust CPU throughput

Latency Minimize response time.

Fairness Distribute resources equitably

Scheduling algorithms: **FCFS**, **RR**, **STCFSJF**, **Fair Share**, **Priority**, **Lottery**, **MLFQ**, Shortest Remaining Time First **SRTF**