Jack Dent
Graham Lustiber

CS161, Assignment 2 Design Document

Contents

1) Introduction

The goal of this assignment is to implement system calls that interact with the file system, system calls that interact with processes, and algorithm that schedules processes to our LobsterOS kernel.

In our kernel, we only implement single threaded processes and disable interrupts for all system calls. We support multiple processes, which can be run concurrently.

<u>2) Code walkthrough</u>

<u>Part A</u>

1) What are the ELF magic numbers?

ELFMAG0 is 0x7f; ELFMAG1 is 'E' (69); ELFMAG2 is 'L' (76); ELFMAG3 is 'F' (70), which tells the kernel that the current file is in ELF format, so that it can continue to look for the rest of the metadata.

2) What is the difference between UIO_USERISPACE and UIO_USERSPACE? When should one use UIO_SYSSPACE instead?

UIO_USERISPACE denotes executable process code, while UIO_USERSPACE denotes process data.

UIO_SYSSPACE should be used for the kernel, since it doesn't need a buffer to pass data within itself.

3) Why can the struct uio that is used to read in a segment be allocated on the stack in load_segment() (i.e., where does the memory read actually go)?

The struct contains an iovec struct, which contains a buffer for the memory read (supplied by the user) that shouldn’t be declared on the stack, as well as the virtual address from the thread's perspective. Thus, the memory read goes into a buffer in the user's address space.

4) In runprogram(), why is it important to call vfs_close() before going to usermode?

Since we then enter the new process, this is our last chance to close the ELF file, otherwise, there would be a dangling ref count on the file.

5) What function forces the processor to switch into usermode? Is this function machine dependent?

mips_usermode, which *is* machine dependent

6) In what file are copyin and copyout defined? memmove? Why can't copyin and copyout be implemented as simply as memmove?

copyin and copyout are defined in kern/vm/copyinout.c, and memmove is defined in common/libc/string/memmove.c. copyin and copyout check to make sure the provided address falls within the user's address space, a security check memmove does not make.

7) What (briefly) is the purpose of userptr_t?

userptr_t is used to differentiate at the C-code level between kernel pointers and pointers the user passes in. This prevents dumb kernel-programmer mistakes.

Part B

1) What is the numerical value of the exception code for a MIPS system call?

EX_SYS is 8

2) How many bytes is an instruction in MIPS? (Answer this by reading syscall() carefully, not by looking somewhere else.)

4 bytes

3) Why do you "probably want to change" the implementation of kill_curthread()?

The kernel will panic, thus killing all other threads/processes running as well, which probably should only happen in the most extreme cases.

4) What would be required to implement a system call that took more than 4 arguments?

We'd need to store how many more there are a register, and then get them starting at sp+16 from the thread's stack.

Part C

1) What is the purpose of the SYSCALL macro?

SYSCALL gives the kernel a single function to handle system calls, done via putting the call number into v0 and then jumping to the handling code.

2) What is the MIPS instruction that actually triggers a system call? (Answer this by reading the source in this directory, not looking somewhere else.)

Line 84 of userland/lib/libc/arch/mips/syscalls-mips.S

3) After reading syscalls-mips.S and syscall.c, you should be prepared to answer the following question: OS/161 supports 64-bit values; lseek() takes and returns a 64-bit offset value. Thus, lseek() takes a 32-bit file handle (arg0), a 64-bit offset (arg1), a 32-bit whence (arg3), and needs to return a 64-bit offset value. In void syscall(struct trapframe *tf) where will you find each of the three arguments (in which registers) and how will you return the 64-bit offset?

Since arguments in registers must align to 64 bits if a 64 bit argument is present,
arg0 will be in a0, a1 will be empty, arg1 will be spread between a2 and a3, and arg3 will
be at sp+16. The return value will then be spread between v0 and v1.

3) Data structures

File descriptor

```
struct file {
        struct vnode *f_vnode;
        struct lock *f_lock;
        int f_flags;
        int f_offset;
        int f_refcount;
}
```

Each process owns a table of pointers to fd structs, where the index of the pointer in the table is the 'file descriptor'/'handle'. We do not keep all the fd structs in a contiguous array of kernel memory: instead, whenever we need to create a new fd struct, we simply ask 'kmalloc' for a chunk of free memory.

Synchronisation: we need a lock around the refcount in case a parent and child process happen to access an fd simultaneously. We don't use a spinlock, since there are some instances in which we'll want to hold the lock for a fairly long time. See the discussion of synchronisation under the read system call, for instance.

Process

```
struct proc {
        pid_t pid;
        struct threadarray *p_threads;          // only one thread in OS/161
        pid_t pid *p_parent;                    // parent's pid
        struct procarray p_children;            // array of children
        struct addrspace *p_as;                 // address space
        struct fdtable *p_fds;                  // fd table
        vnode *p_cwd;                   //current working directory
        int p_exit_status;              // contains exit status once exited
        struct semaphore* wait_sem      // for parent's to wait on child
}
```

Tracking pids:

We allocate a global dynamic process array PROC_ARRAY of size INITIAL_PROC_COUNT=512, where the pid of each process is given by its index in the array.

Assigning pids:

When creating a new process, we acquire a lock on PROC_ARRAY and then scan from left to right, looking for a NULL entry. Once we have found an empty slot *i*, we can insert our proc struct into PROC_ARRAY[i], and set proc->pid to *i*.

Recycling pids:

We have an integer array called EXIT_PROC_ARRAY that keeps track of the exit statuses of exited processes. When waitpid is called on it or a child exits after its parent, that pid becomes available for reuse so we replace PROC_ARRAY[pid] with NULL.

The benefit of this system is that is allows for easy look-up and deletion of pids. We only keep an array of pointers to process structs, so they can be allocated when needed to avoid consuming INITIAL_PROC_COUNT * sizeof(proc) bytes of memory when only a few process may be active. A stack might allow for O(1) retrieval and returning, but for debugging purposes, always assigning the lowest available pid to a new process makes things easier. This is something that could be optimized when everything is fully working.

File descriptor tables

For the most part, we reuse the same data structure and protocol that we devised for assigning pids: we assign a dynamic file descriptor table to each table of size INITIAL_FD_COUNT=256, where each file descriptor indexes the memory address of its fd struct.

Queue

For now, a simply queue structure implemented as a linked list. This will likely change once scheduling is designed.

4) Bootstrapping/Initialization

In thread_bootstrap(), setup the process struct, thread struct, and fd table for the kernel. Initialize spinlock for proc_table.

5) File system calls

open("path", flags, [optional] mode)

Each process struct contains a table of open file descriptors. Opening a file consists of

1. Searching for the vnode associated with the file. If not found, go to make_file
2. Checking the user has the permissions to open that file specified by the flags argument
3. Creating a `struct file` with the right metadata and inserting it into the proc table

make_file: if the file doesn't exist, create one and create a vnode for it.

Once the vnode has been located, find an unused fd for it in the fd table for the process. Allocate an fd struct in kernel memory and set the fd_array[fd] pointer to be it. Fill in f, flags, lk, and set offset and refcount to 0 and 1 respectively. Return the fd number to the user.

Note that if we open the same file twice we'll have two separate file descriptors referring to the same vnode, each of which will have an offset that is independent of the other.

Synchronisation: since parent and child processes have access to the same fd struct, we need a lock to enforce mutual exclusion around the fd struct's refcount.

close(fd)

First check that fd_array[fd] points to a valid file descriptor. Acquire the lock on the vnode, decrement the vnode refcount, and release the vnode lock. If fd_array[fd]->fd_refcount == 1, then free the file descriptor memory; otherwise, decrement the refcount, and be aware that other processes could still have access to the open file. Finally, set fd_array[fd] = NULL.

Synchronisation: see the commentary in the open() system call.

read(fd, buf, buflen)

First check that fd_array[fd] points to a valid file descriptor struct. Then, acquire the fd struct lock, and check that struct has the read flag set in its flags. Make a struct iovec and struct uio, supplying the user's buffer, buflen and the fd struct's offset to the uio_kinit function. Call uiomove to read the data into a uio struct. If this fails, return the error code. Update the file offset, and return the number of bytes read (len - uio.uio_resid). We make sure to release the fd struct lock before returning any value.

Synchronisation: There are two synchronisation problems here. Firstly, there's the issue of a parent a child process which both have access to the same fd struct manipulating the offset/current position in the file, so we need to acquire a lock on the fd struct itself. You can have multiple fds that refer to the same file reading from separate offsets, but a single file descriptor can only have the read or write syscalls called on it by different processes sequentially, as opposed to concurrently.

Secondly, we also have the more fundamental issue of the readers/writers problem on the file itself. Fortunately, the VFS/vnode layer provides us with abstractions and handles the synchronisation for us (N.B. emufs, semfs and sfs both implement locking primitives around the reads and writes).

write(fd, buf, nbytes)

First check that fd_array[fd] points to a valid file descriptor struct. Then, acquire the lock on the fd struct, and check that struct has the write flag set in its flags. Make a struct iovec and struct uio, supplying the user's buffer, buflen and the fd struct's offset to the uio_kinit function. Call uiomove to write from the uio struct. If this fails, return the error code. Update the file offset, and return the number of bytes written (len - uio.uio_resid). We make sure to release the fd struct lock before returning any value.

Synchronisation: see the commentary in the read() system call.

lseek(fd, pos, whence)

If we lseek on fd, the offset should also update for all forked child processes. We implement lseek as follows: first check that fd_array[fd] points to a valid file descriptor struct; otherwise, we return EBADF. Then, we acquire a lock on the fd struct and check that VOP_ISSEEKABLE returns true; otherwise, we return ESPIPE. We then switch on the value of whence: if is one of SEEK_SET, SEEK_CUR, or SEEK_END, we calculate the new file offset and assign it to a local variable new_pos; otherwise we return EINVAL. Finally, we check that new_pos is within the bounds of the vnode: if it is, we update the fd struct's offset; otherwise, we return EINVAL. We make sure to release the fd struct lock before returning any value.

Synchronisation: see the commentary in the first half of the read() system call (we need to prevent a parent and a child process that share a fd struct from lseek'ing at the same time).

dup2(oldfd, newfd)

Lock the the file descriptor table. Assert that fd_array[oldfd] points to a valid file descriptor. If fd_array[newfd] points to a valid file descriptor, call close(newfd). Set newfd to point to the same file descriptor as oldfd in the process table, and increment the ref count by one. The seek position of oldfd remains unchanged.

Synchronisation: the lock of the file descriptor table and file descriptor itself ensure no other process can interfere.

chdir()

First, check the pathname exists and is a valid directory (throwing the relevant ENODEV, ENOTDIR, ENOENT error if this is not the case). Then, copy the user's string into a buffer in kernel memory (making sure the string is null terminated and doesn't overrun the buffer) using copyinstr, acquire a lock on the proc struct, free the memory for the old cwd string and update the proc's cwd pointer to the start of the buffer. Finally, release the lock.

Synchronisation: since our processes are single-threaded, chdir and getcwd will only ever be called sequentially, and we therefore don't need a lock.

getcwd(buf, buflen)

Acquire a lock on the proc struct, copy up to buflen bytes from the proc's cwd field into buf using copyoutstr, without null terminating the buffer, and finally release the lock.

Synchronisation: see the commentary in the chdir system call

6) Process management

fork()

First make a copy of the parent's trapframe for the child process. Find an available pid. Copy over all the fields from the parent's proc struct. Make a copy of the parent's trapframe and address space and set them as the child's trapframe and address space. Copy over the new pid. Increment the refcounts for open files in the parent. Save the child's pid in the list of the parent's children, and set it in the return register. Call threadfork() on a function launch_child, which will call mips_usermode.

Synchronisation: The only synchronization issue we need to worry about is getting a new pid for the child. This will be managed by a lock on PROC_ARRAY.

waitpid(pid)

First check to make sure that pid is a child process of the caller. Check that the option is 0 and that status is a valid pointer. Then call P() on the child's semaphore. If the child exited, it will immediately return. Otherwise, it will sleep until the child exits. Remove the child from the list of children, and save the exit status in status. Mark the pid as free to be reaped.

Synchronisation: by only allowing a parent to wait on a child, we prevent the potential deadlock of 2 processes waiting on each other. The other issue is a child exiting just before a parent sleeps assuming the child is still running. This is solved by the semaphore and the underlying lock associated with it. That is, the V() signal will not be lost whether it comes before or after the parent calls P(). Finally, we keep a lock on PROC_ARRAY

Keeping the exit status in the proc struct until it's reaped has the drawback of consuming pid's even after the process has terminated. However, the benefit of a simple lookup system is more valuable. Otherwise, some external data structure would need to be kept with orphan children and exit statuses in the kernel. If a pid were reassigned, we would then also need a system of keeping track of whether the current process with that pid is the child, or whether it's the exited one, a needlessly difficult system

getpid()

Every process stores their pid in the process struct. Two different processes can make this syscall at the same time (we don't need to disable interrupts).

Synchronisation: We don't need a lock, because pids are immutable for the duration of a process's lifetime, and we're running single-threaded processes, so no other thread can call exit().

execv(const char *program, char **args)

Check that the pointers passed in argv are all in the user's address space. Check that the user has supplied <= ARG_MAX arguments. If so, copy them over to a buffer. Check that the program_name pointer is in the address stack, and that it refers to an actual program via vfs_open() that the user has permission to execute. Load the executable via load_elf(). Establish a stack in the address via as_define_stack(), and copy in the arguments to the stack via copyinstr() and copyin(), ensuring that they are word-aligned. Close the ELF, and call enter_new_process. Panic if it returns for some reason.

Synchronisation: Since we are not interacting with multiple threads, getting new pids, etc., we can rely on the synchronization invariants associated with opening files, kmalloc(), etc.

_exit(exitcode)

First iterate through children to set the parents as kernel, as they are now zombie children. Set the exit status to be the passed in exitcode. Cleanup the thread via thread_exit(). If this process is an orphan itself, mark it as reaped in the pid array. Otherwise, there could be a parent waiting, so call V() on wait_sem. Free up the allocated stack space, kernel stack, process struct, and file descriptor structs

Synchronisation: We have a lock on PROC_ARRAY when the current process is being marked an orphan. Moreover, the invariants of the semaphores ensure that the V() call will not be lost, regardless of when the parent calls P().

kill_curthread(vaddr_t epc, unsigned code, vaddr_t vaddr)

Call exit() with WSIGNALED

Synchronisation: Maintained by exit()

7) Scheduler

Our scheduler is similar to the O(n) scheduler discussed in class. Each thread is assigned a counter t_priority, initialized to 100, the same as HZ is defined. Whenever a thread is descheduled, t_priorityis decremented if the thread has status S_READY, i.e., it used its full time slice. Otherwise, if it has status S_SLEEP, t_priority is incremented. Whenever schedule is called, it moves the thread with the highest t_priority to the head of the runqueue. Every 100 calls to schedule, all threads on the runqueue get their t_priority reset to 100.

Usage: To use the new scheduler, #define USING_SCHEDULER 1 in thread.h. Otherwise, #define it as 0 to use the default round robin scheduler.

Tunable parameters are also found in thread.h. The main parameters of interest are COUNTER_BOUND, S_READY_DEC, and S_READY_INC. The first is how many calls to schedule() it takes before all threads are reset to PRIORITY_INIT. S_READY_DEC and S_READY_INC control how much a thread’s priority is changed based on its computational or IO behavior.

Extensive analysis would likely test across all reasonable ranges of these values to find some best setting. If one were doing a lot of interactive work, this might just look like a human choosing the one that feels the most responsive in normal usage. For computational systems, one would probably run a series of typical jobs and measure the total time needed to complete. In reality, most people do a combination of these, so a mix of each testing environment would be needed.

Though this scheduler is O(n), as discussed in class it is fast enough for most purposes. We chose it because it is a simple algorithm that requires basically no space overhead and doesn't use "magic numbers." It gives priority to IO programs by starting them early in a scheduling cycle in comparison to computational programs. We avoided a randomized scheduler because though asymptotically there may be no starvation, it is still very possible that an interactive program may get laggy IO because it gets unlucky a number of times in a row. We also avoided the O(log n) and O(1) schedulers discussed in class because they have a higher memory overhead cost and more complicated implementation. On os161, when programs are often simple, few programs are run in parallel, and memory is very limited, these fancier schedulers seemed unnecessary.

One feature to be implemented in the future is a nice() system call that lets processes assign a static priority level themselves, as exists on Linux.

8) Timeline

Plan of action/dividing up work

- Pair program on execv, fork, and open
- Jack: read, write, lseek, dup2, chdir, getcwd
- Graham: waitpid, exit, close, getpid, kill_curthread