Computer Science 161: Operating Systems
Assignment 2 Design Document Draft

David Li and Garrett Tanzer
February 7, 2017

1 Introduction

Assignment 2 can be broken up into two major components: processes and I/O.

Our implementation of processes will require some additions to the `proc` struct and an expansion of the global variable `kproc` to an array of all runnable processes. The related `fork()` and `exec()` system calls will hook into this structure for process creation, while `waitpid()` and `_exit()` deal with process termination; a main challenge in both is the safe transfer of data to and from kernel space.

For I/O, we will build on top of the existing virtual file system abstraction with two levels of file descriptor tables. `open()`, `read()`, `write()`, and other I/O system calls will act at a per-process level but remain synchronous with others through the underlying file system.

Related to this, we still need to properly figure out exactly how `buf.c` factors into our implementation of I/O system calls and the file descriptor abstraction.

2 Overview

In order to keep track of multiple runnable processes, we're changing the `kproc` global variable to an array of `procs` called `procs` with arbitrary length `MAX_PROCS`, with process 0 always set to the init process. We will modify the `struct` proc to facilitate this structure; our implementation is as follows (our changes bolded):

```
struct proc {
      char *p_name;                   /* Name of this process */
      struct lock p_lock;             /* Lock for this structure */
      unsigned p_numthreads;          /* Number of threads in this process */
      struct addrspace *p_addrspace;/* virtual address space */
      struct vnode *p_cwd;            /* current working directory */
      struct procarray *p_children; /* defined using array.h */
      proc *p_parent;                 /* pointer to parent process */
```

```
    pid_t pid;                    /* index for global procs, process id */
    cv *p_cv;                     /* cv for parent to wait on */
    struct fd_off *p_fds;         /* array of file descriptors */
                                  // includes int fd, off_t off
};
```

Each `proc`, whose index in the global array is represented by its `pid`, will be protected by its own spinlock, as shown above. A process may have no children, but it must have one (and only one) parent. The `p_children` list will be used both to transfer ownership of orphaned processes to the init process (which will periodically reap them) in the case of a premature exit and to get the `p_cv` belonging to each child to wait on with `waitpid()`. The `p_fds` array will be explained below. Process creation and destruction will have to be adjusted to properly initialize and clean up these new fields.

To isolate each process's file system access, we're using a two-tier system with file descriptors. Each process has its own array of file descriptor integers + offsets, `p_fds`, whose values are the indices in a single, global array of `vnodes`, `vfiles`, with arbitrary length `MAX_FDS`. When a file is opened, either its existing `vnode` is located or a new one is opened, then placed in the global array. This index is placed as the value in the process-level array at an arbitrary index, and that arbitrary index (the file descriptor) is returned for future use. `read()`, `write()`, etc. will now translate this file descriptor into the underlying virtual file, preventing redundant file access and helping to ensure synchronization (`vnodes` have spinlocks inside them). We still need to figure out exactly how uio, buf, VOP, vfs, and fs functions interact for the actual read/write operations.

3 Topics

Processes

This might be premature optimization, but to find free process slots we plan to use a combination of a running maximum index and a linked list of terminated processes. As pids are used, the running max increases, but at the same time, as those pids are freed by processes fully exiting, they are added to a linked list that gets precedence over the running max. This way, instead of having to iterate through the entire list of processes each time we want to start a new one (and acquire/release a spinlock every single time), we would use a single spinlock to check both the running max and the linked list, pick the earliest entry, then initialize a process with that pid (of course acquiring its spinlock in the process to prevent race conditions).

Empty slots in the array will be initialized/reset to NULL, and the single spinlock will protect write access to all NULL slots until a spinlock has been created specifically for it. We might periodically iterate through the array of processes to prevent the linked list from becoming unnecessarily large. See the sections below on `fork()`, `execv()`, `waitpid()`, and `_exit()` for more on processes.

File Descriptors

We will use an analogous process to the aforementioned one to find free spots in the file descriptor arrays as in the global process array.

By default, 0, 1, and 2 in processes' descriptor tables will refer to 0, 1, and 2 in the global file array, which are permanently set to stdin, stdout, and stderr.

When a process exits, we have to iterate over its `p_fds` and decrease reference counts in the vnodes where appropriate (i.e. we have to close all open file descriptors).

Because the global file array will have fine-grained locking, multiple files can be accessed simultaneously, but each file can only be modified by one process at a time.

The `off` field of `fd_off` enables `lseek()` to keep a different offset in each process.

Scheduling

TBD

3.1 Functions

Follow these steps to add a system call:
- Add a function header to ~/cs161/os161/kern/include/syscall.h
- Check that the syscall is defined in ~/cs161/os161/kern/include/kern/syscall.h, and if not, add it or uncomment its existing entry
- Add a case in ~/cs161/os161/kern/arch/mips/syscall/syscall.c
- Add to an appropriate file in the syscall directory, or make a new one if none of the existing ones fit (we'll probably make one document for processes and one document for I/O)
- Add the user-facing header to ~/cs161/os161/userland/include/unistd.h
- Add that file to around line 381 of conf.kern if you made a new one.

Error handling:
- success: 0 in $a3, return in $v0
- failure: nonzero in $a3, errno in $v0

`open()`

```
int open(const char* pathname, int flags, int mode) {
      getLock(availableVnodes->lock);
      vnode* workingNode;
      if(FirstVNode != NULL){
            getLock(firstVNode->Wrapper->lock);
workingNode = FirstVNode;
            FirstVNode = firstVNode->next;
```

```
      }else{
            Graceful quit if highest element is more than allowable
            getLock(fileTable[highestElem]->lock);
            workingNode = fileTable[highestElem];
            Increment highestElem;
      }
      relinquish(availableVnodes->lock);
      workingNode->taken //May be redundant.
      getlock(curproc->descriptorTable->lock);
      Int n = iterate through curproc->discriptorTable->array to find lowest
NULL entry.
      N =
      Int r = vfs_open(pathname, flags, mode, &(workingNode->meat));
      Handle return value;
      relinquish (workingNode->lock);
      Return 0;
}
```

read()

```
ssize_t read(int fd, void *buf, size_t count) {
      getlock(curproc->descriptorTable->lock);
      struct* working = curproc->descriptorTable->array[fd];
      getlock(working->lock);
      relinquish(curproc->descriptorTable->lock);
      if(working == NULL) Handle EBADF
      Struct uio newUIO;//not absolutely sure if following is right config.
      newUIO.uio_seg = UIO_USERSPACE;
      newUIO.uio_iov->iov_ubase = buf; newUIO->uio_iovcnt = 1;
      newUIO.uio_rw = UIO_READ;
      newUIO.uio_offset = working->filePosition; newUIO->uio_resid = count;
      newUIO.uio_space = getAddressSpaceOfCaller();
//Above calls can also be implemented using uio_kinit;
      Int r = VOP_READ(working->VNode, newUIO);
      if(r==error) handle EIO;
      r = uiomove(kernelBuffer, count, newUIO); //is uiomove really needed?
      if(r==error) handle EFAULT
      Advance file position inside working.
      relinquish(working->lock);

      Return 0;
}
```

Issue: Does read hang currently? If so, how to make it not hang? Implement multiple threads? Wait channels?

write()

```
ssize_t write(int fd, const void *buf, size_t count) {
```

```
	getlock(curproc->descriptorTable->lock);
	struct* working = curproc->descriptorTable->array[fd];
	getlock(working->lock);
	relinquish(curproc->descriptorTable->lock);
	if(working == NULL) Handle EBADF
	Struct uio newUIO;
	newUIO.uio_iov->iov_ubase = buf;
	newUIO.uio_iovcnt = 1;
	newUIO.uio_offset = working->fileOffset;
	newUIO.uio_resid = count;
	newUIO.uio_segflg = UIOUIO_USERSPACE;
	newUIO.uio_rw = UIO_WRITE;
	//call to uio_move needed??
	Int r = VOP_WRITE(buf, newUIO);
	Handle error cases.
	Advance file pointer within working.
	relinquish(working->lock);
}
```

Should the UIO be global? Is it worth it to generate a new uio every time?

`lseek()`

```
off_t lseek(int fd, off_t offset, int whence) {
	getlock(curproc->descriptorTable->lock);
	struct*working = curproc->descriptorTable->array[fd];
	getlock(working->lock);
	relinquish(curproc->descriptorTable->lock);
	if(working == NULL) Handle EBADF;
	if( ! VOP_ISSEEKABLE(working->vnode))Handle ESPIPE
	Error check whence value
	if(whence == SEEK_SET) working->position = offset;
	if(whence == SEEK_CUR) working->position += offset;
	if(whence == SEEK_END) {
		VOP_STAT(working->vnode, statbuf );
		working->position = statbuf->size + offset;
	}
	Working -> position can’t be negative
	relinquish(working->lock);
	Return 0;
}
```

`close()`

```
int close(int fd) {
	getlock(curproc->descriptorTable->lock);
	struct*working = curproc->descriptorTable->array[fd];
	getlock(working->lock);
	getLock(availableVnodes->lock);
```

```
        vfs_close(working->vnode);
        A = firstVNode
        firstVNode = working;
        firstVNode->Next = A;
        relinquish(availableVNodes->lock;
        free(guts of working);
        relinquish(working->lock);
        Delete junk from curproc->descriptorTable->array[fd];

        relinquish(curproc->descriptorTable->lock);
        Return 0;
}
```

`dup2()`

Duplicates an entry in the current process's file descriptor table—doesn't affect the underlying global structure.

```
int dup2(int oldfd, int newfd) {

        // check invalid params

        curproc->p_fds[oldfd] = curproc->p_fds[newfd];

        // error handling/return
}
```

`chdir()`

Serves as a wrapper to call `vfs_chdir()` and handle errors.

```
int chdir(const char *path) {

        // check invalid param

        vfs_chdir(path);

        // error handling/return
}
```

`__getcwd()`

Serves as a wrapper to call `vfs_getcwd()` and handle errors.

```
int __getcwd(char *buf, size_t buflen) {

        // check invalid params

```

```
      vfs_getcwd(...);

      // error handling/return
}
```

`getpid()`

Serves as a simple wrapper to surface the `pid` field as something meaningful for the user.

```
pid_t getpid(void) {
      // spinlock isn't needed because pid is never modified

      return curproc->pid;
}
```

`fork()`

Make a copy of the process (as a trapframe) (deep copy of the address space and file descriptors), assign a new pid and set it as the return value—then context switch to the trapframe (from enter_forked_process in syscall.c using thread_fork and the new process)

```
pid_t fork(void);
```

`execv()`

Use modified proc_create_runprogram to make blank new process, give it an empty stack and address space, and run the program from its path name with vfs

- iterate over args
- need to think more about specific implementation

```
int execv(const char *program, char **args);
```

`waitpid()`

Sleep until a specified child thread has no threads left, then grab its exit code and destroy it

```
pid_t waitpid(pid_t pid, int *status, int options) {
      // error checking
      lock_acquire(procs[pid].p_lock);
      while(procs[pid].p_numthreads > 0) {
            cv_wait(procs[pid]->p_cv, &procs[pid].p_lock);
      }
      save exit code (will store in ->pid maybe?)
```

```
      proc_destroy(...)
      lock_release(procs[pid].p_lock);
      Clean up process carcass
}
```

`_exit()`

transfer ownership of children to init
iterate over and close file descriptors
decrease numthreads (to 0)
free other proc fields

```
void _exit(int exitcode) {

}
```

`kill_curthread()`

Remove process from run queue
proc_destroy() it

3.2 Plan of Action

First we'll finish the rest of the design doc, then we'll see about the coding schedule.