Assignment 2 Design Doc

Team Longcat: Abby Lyons, Timothy Tamm

Introduction

In this assignment we need to implement 3 major components of our operating system:

1. Identify processes and handle their creation, execution and finally destruction. For this part we need to:
    a. Develop a good structure for keeping track of all the necessary information a process needs
    b. Implement fork and execv to create new processes and make them do something useful
    c. Implement waitpid and exit to make it possible for processes to exit and to make it possible to keep track of children processes and wait on them
    d. Make sure that processes get cleaned up
2. Implement a good system for file and device I/O. This part involves:
    a. Implementing system calls for opening and closing files and devices and reading, writing and seeking in them
    b. Keeping track of the current working directory of a process and being able to change it
    c. Keeping track of open files and devices in a common kernel filetable, which is referenced by per process file descriptor tables
3. Schedule processes in a way that minimizes users' wait time.

Overview

The first major portion of the assignment is identifying and handling processes. There's already a struct in proc.h, but it doesn't contain all of the things that we'd like it to contain. Each process absolutely must contain a pid, which the kernel can use to identify it from other processes. We also want it to contain the parent's pid, which will ensure that it can be cleaned up if the parent returns without calling waitpid. In addition, we want to add a file descriptor table (see below), exit code (for waitpid), and a wait channel (also for waitpid). We plan on using an array to keep track of live processes.

Here's an outline of the relevant functions that we must write:

- Fork: create a new process, copy all values from the current process into it, and find an available pid. Make a copy of the old process's address space and set the new process's address space to the copy. Then, create a new thread, make appropriate updates to the trapframe, and wake up the new process.
- Execv: open the file as a vnode, create a new address space and deactivate the old one, allocate space for a kernel buffer, and define the user stack. Next, for each argument from the user, copy it into the kernel buffer, checking that we haven't exceeded the

maximum number of arguments at each step. Then, add padding at the top of the user stack and copy all of the pointers to the arguments onto the stack. Finally, load the executable, close the file, and swap back into user mode.

- Exit: first, check the coffin for any previously orphaned processes, and destroy them. Next, close all file descriptors and change the state to indicate that it's a zombie. Wake one on the process's wait channel if the process's parent is still alive. If not, put the process into the coffin, and call thread_exit.
- Waitpid: check the flags and the target process's parent value; if the caller isn't the parent, return an error. Next, check the child's state; if it's a zombie, clean up the zombie and return 0. If it's alive and the waitpid call should not wait, return -1 immediately. Otherwise, sleep on the child process's wait channel, and return 0 after being woken up.
- Kill_curthread: turn the thread into a zombie, and then call thread_switch which will take care of the rest.
- Getpid: return the current process's pid.

The next major portion is the system for file and device I/O. As far as we can tell, there is no way for a process to have files to read/write/etc., so we must design a system of keeping track of open files. We have decided that each process should have a table of its file descriptors, and the kernel should hold a table of file handles; each handle should contain a file's vnode, offset, reference count, flags, state, and two locks, one for reference count and one for file usage. When the user opens a file, the return value is an int, which is just an index into this table. The content at that index in the table will be an index into the kernel's file table.

We chose this method because we wanted to have a specific kind of separation between processes that have the same file open. For example, if we want two processes to read from the same file, the offset for one process should not affect the offset for the other; this means that this file must have two separate entries in the file table, one for each process. However, we also want to be able to use dup2 so one file descriptor can be updated to refer to the same entry in the kernel file table as another.

Here's an outline of the relevant functions that we must write:

- Open: first find an open location in the process's fd table, and find and update an open location in the file table. Next, acquire a vnode for the new file, and initialize a new file handle for it. Update the file table to include the new file handle, and update the process's fd table to hold the index in the file table. Finally, return the file descriptor.
- Read: get the file handle, check that it's readable, and make a new uio. Next, call vop_read on the file handle and uio. Update the file handle's offset, and finally return the vop_read return value.
- Write: get the file handle, check that it's writable, and make a new uio. Next, call vop_write on the file handle and uio. Update the file handle's offset, and finally return the vop_write return value.

- Lseek: get the file handle, check that it’s seekable, and update the offset based on the provided number and the whence argument. Return the new offset.
- Close: get the file handle. If the reference count is greater than 1, decrease the reference count and update the process’s fd table. Otherwise, destroy the file handle before updating the process’s fd table.
- Dup2: check that the oldfd and newfd values are in range, and that oldfd is an open file. Next, if newfd is an open file, close it. Then, set the newfd value in the process’s fd table to equal the oldfd value. Return newfd.
- Chdir: make a copy of the pathname, and call vfs_chdir.
- __getcwd: make a new uio, and use vfs_getcwd, returning whatever it returns.

For the scheduler, we implemented a multilevel feedback queue. See the “Schedule” section for implementation details!

Global constants

PID_MAX = 4096 (or maybe 32767 as currently defined in kern/include/kern/limits.h)
PID_MIN = 2
PID_INVALID = 0
OPEN_MAX = 128
FD_FREE = 255
FT_MAX = 256
__CPY_BUF_MAX = 5
PRIORITY_MAX = 20
YIELD_BOOST = 10

Identifying processes

The first process (**init**) is going to be created using a special **bootstrap_process** function, which sets the process up, bootstraps the process tables and file tables.

A new process will have its parent’s working directory as its working directory when it is created using fork. However on the init process, this should probably be bootfs_vnode (defined in kern/vfs/vfslookup.c).

A new process will always have 3 open file descriptors: stdin, stdout, and stderr. Initially all three of these will be connected to the ‘con:’ device. I think this can be accomplished using dev_lookup function (kern/vfs/device.c) with dir = bootfs_vnode; pathname = ‘con:’.

To keep track of file descriptors, the process should have a table of offsets into the kernel's file table. We will define three macros for this, OPEN_MAX (the number of fds a given process can have at once. This is already defined), FD_FREE (a reserved value in the kernel's file table that the fd table elements are initialized to, signifying that the table location is open), and FH_MAX (the size of the kernel's file table). See the File Descriptors section for more information.

Here is our **proc** struct and the structs related to it:

```
struct proc {
    char *p_name;                   /* Name of this process */
    unsigned p_numthreads;          /* Number of threads in this process */
    pid_t p_parent;                 /* Parent's pid (to make zombie killing easier) */

    /* VM */
    struct addrspace *p_addrspace;  /* virtual address space */

    /* VFS */
    struct vnode *p_cwd;             /* current working directory */

    uint8_t p_fds[OPEN_MAX];        /* table of indices into the kernel file table */
    pid_t p_pid;                    /* pid of this process */
    int p_exit_code;                /* code the process exited with */
    proc_state_t p_state;           /* current state of the process */

    /* SYNCH STUFF */
    struct spinlock p_lock;         /* Lock for this structure */
    struct cv *p_cv;                /* cv used for waitpid */
    struct lock p_waitlock;         /* lock used for waitpid */
};
```

```
enum proc_state_t {
    P_ALIVE,             /* state of an alive process */
    P_ZOMBIE             /* state of a process that has exited, but   hasn't been reaped */
};
```

Keeping track of processes

To keep track of the running processes, the kernel should have:

1. An array of size **PROC_MAX** of pointers to processes. With a small **PROC_MAX**, this will not take up too much memory, and the array will offer faster lookup times than a linked list.
2. An integer that keeps track of the most recently assigned pid.
3. A variable **pid_t pt_coffin** that keeps track of any orphaned processes before they can be reaped.
4. A lock to protect the above data structures.

PIDs will be allocated numerically, starting from **PID_MIN** and ending at **PID_MAX,** ignoring pids that become available from processes exiting. Only pids for which the (pid mod **PROC_MAX)** entry in the proc table is empty are valid. After the **PID_MAX** pid has been allocated, the pid allocation wraps around to the start, allocating the first non-used pid, starting at the least recently allocated pid.

Pid 0 is reserved and should not be allocated to any process.

Here's the struct for our proc table:

```
struct proc_table {
    struct proc *pt_procs[PROC_MAX];    /* list of processes */
    pid_t pt_most_recent_pid;           /* the most recently allocated pid */
    pid_t pt_coffin;                    /* coffin for orphaned zombie processes */
    struct lock *pt_lock;               /* lock to protect the table */
    struct copy_buffer *pt_cb;          /* copy buffers used for copying arguments in execv
*/
};
```

Here's the declaration for our kernel proc table:

```
struct proc_table *k_proctable;
```

Here are some API functions for interacting with the proc table:

```
/*
 * Convenience function to initialize a new proc_table.
 * This should be called by the kernel only.
 */
struct proc_table *
pt_init(void)
{
    struct proc_table *pt;
    pt = kmalloc(sizeof(*pt));
    if (pt == NULL)  return NULL;

    // initialize all the entries to be empty except for
    // the first one which should be the kernel process
    for (int i = 0; i < PROC_MAX; i++) {
        pt->pt_procs[i] = NULL;
    }

    // set the first proc to be the kernel process
    pt->pt_procs[PID_MIN - 1] = curproc;

    pt->pt_most_recent_pid = PID_MIN - 1;

    pt->pt_coffin = PID_INVALID;

    pt->pt_lock = lock_create("K_PT_lock");
    if (pt->pt_lock == NULL)  goto cleanup2;

    pt->pt_cb = cb_create();
    if (pt->pt_cb == NULL)  goto cleanup1;

    return pt;

cleanup1:
    lock_destroy(pt->pt_lock);
cleanup2:
    kfree(pt);
    return NULL;
}

/*
 *  returns an unused pid. You must hold the kernel process table's
 *  lock before running this to avoid race conditions
 */
pid_t
pt_get_open_pid(void)
{
    KASSERT(lock_do_i_hold(k_proctable->pt_lock));

    int i = k_proctable->pt_most_recent_pid + 1;
    for (int j = 0; j < PROC_MAX; j++) {

        // Wrap around if necessary
        if (i >= PID_MAX) {
            i = PID_MIN;
        }
        if (i == k_proctable->pt_most_recent_pid) {
            // at this point we have gone through all pids and
            // have not found a valid one
            return PID_INVALID;
        }

        if (k_proctable->pt_procs[i % PROC_MAX] == NULL) {
            k_proctable->pt_most_recent_pid = i;
            return (pid_t) i;
        }
        i++;
    }

    /* we have checked PROC_MAX pids and have not found
     * a valid pid. That means that there are already
     * PROC_MAX processes running */
    return PID_INVALID;

}

/*
 *  Returns the process associated with the given pid
 */
struct proc *
pt_get_proc(pid_t pid)
{
    struct proc *pr;
    lock_acquire(k_proctable->pt_lock);
    pr = k_proctable->pt_procs[pid % PROC_MAX];
    lock_release(k_proctable->pt_lock);
    return pr;
}
```

Copy Buffers

The process table contains a copy_buffer, which is a struct that holds **__CPY_BUF_MAX** char arrays of size **ARG_MAX**. Access to these buffers is managed by a semaphore and there's a **proc *** array for keeping track of which process holds which buffer. When a process execs, it will try to acquire one of these buffers and will use it to copy arguments. This strategy helps us ensure that processes will still have access to a big buffer, even after memory has become segmented, because these buffers are created during the boot process.

File descriptors (FD)

A file descriptor is just an int, at least from the user side. It represents an index into a process's file descriptor table, which in turn holds indices into the kernel's file table.

We're interested in keeping track of multiple things:

- The vnode that is associated with the open file.
- The current offset in the file.
- A count of processes that reference a file handle
- The flags with which the file was opened.
- The state of the file handle. This is used to make sure that a process doesn't try to use a filehandle while it is being destroyed.
- Additionally we have to store 2 locks:
    - One to protect the reference count of the file handle
    - One to protect the vnode and offset.

File descriptors should be unique within a process and given out in numeric order; that is, when a new fd is created, its value is going to be the smallest unused table index. At that location in the file descriptor table, the value stored there depends on whether we want it to "point" to a new file or an existing one.

All actions on **file_handles** should make sure that its state is active before using it.

And here's the struct for a **file_handle**, which is the data type for the kernel's file table:

```
struct file_handle {
    struct vnode *fh_file;      /* vfs node associated with the fd */
    off_t fh_off;               /* offset in the current file */
    uint32_t fh_refcount;       /* how many FD's reference this entry */
    struct lock *fh_ref_lock;   /* lock for updating refcount */
    struct lock *fh_use_lock;   /* lock for using the file */
    int fh_open_flags;          /* flags with which the file was opened */
    fh_state_t fh_state         /* state of the file_handle. */
};
```

```
enum fh_state_t {
    FH_ACTIVE,              /* state of an active file_handle */
    FH_INACTIVE             /* state of a file_handle that is being destroyed*/
};
```

Here’s a helper function for initializing a new **file_handle**:

```
/*
 * Convenience function to initialize a new file_handle.
 */
struct file_handle *
fh_init(struct vnode *file, int flags)
{
    struct file_handle *fh;
    fh = kmalloc(sizeof(*fh));
    if (fh == NULL)  return NULL;

    fh->fh_off = 0;
    fh->fh_refcount = 1;
    fh->fh_ref_lock = lock_create(curproc->p_name);
    if (fh->fh_ref_lock == NULL)  return NULL;
    fh->fh_use_lock = lock_create(curproc->p_name);
    if (fh->fh_use_lock == NULL) {
        lock_destroy(fh->fh_ref_lock);
        return NULL;
    }
    fh->fh_open_flags = flags;
    fh->fh_file = file;

    return fh;

}
```

Here’s a helper function for closing a **file_handle**:

```
/*
 * Convenience function for closing a file_handle
 */

struct file_handle *
fh_close(struct file_handle *fh)
{
    // Make sure pointer is valid
    KASSERT(fh != NULL);

    lock_acquire(fh->fh_ref_lock);
    if (fh->fh_refcount > 1) {
        fh->fh_refcount--;
        lock_release(fh->fh_ref_lock);
        return fh;
    }
    lock_release(fh->fh_ref_lock);

    lock_destroy(fh->fh_ref_lock);
    lock_destroy(fh->fh_use_lock);

    vfs_close(fh->fh_file);

    kfree(fh);
    return NULL;
}
```

Notice that **fh_close** doesn't necessarily destroy the **file_handle**. It does so only if the closing process is the last one that is referencing it.

Here's the struct for our system file table:

```
struct file_table {
    struct file_handle *ft_fhs[FT_MAX];
    struct lock *ft_lock;
};
```

Here are the functions that implement an API for dealing with the system file table:

```
/*
 * Convenience function to initialize a new file_table
 */
struct file_table *
ft_init(void)
{
    struct file_table *ft;
    ft = kmalloc(sizeof(struct file_table));
    struct file_handle *con_fh[3];

    if (ft == NULL)  return NULL;

    // initialize stdin, stdout, stderr:
    char *con_path = kstrdup("con:");
    struct vnode *con_dev[3];
    int result = vfs_open(con_path, O_RDONLY, 0, &con_dev[0]);
    if (result != 0)  goto cleanup7;
    con_fh[0] = fh_init(con_dev[0], O_RDONLY);
    if (con_fh[0] == NULL) goto cleanup6;
    // stdin
    ft->ft_fhs[0] = con_fh[0];
    for (int i = 1; i < 3; i++) {
        con_path = kstrdup("con:");
        result = vfs_open(con_path, O_WRONLY, 0, &con_dev[i]);
        if (result != 0) {
            if (i == 1)  goto cleanup5;
            goto cleanup3;
        }
        con_fh[i] = fh_init(con_dev[i], O_WRONLY);
        if (con_fh[i] == NULL) {
            if (i == 1)  goto cleanup4;
            goto cleanup2;
        }
        // stdout, stderr
        ft->ft_fhs[i] = con_fh[i];
    }

    kfree(con_path);

    // initialize all the entries to be empty except for
    // stdin, stderr, stdout
    for (int i = 3; i < FT_MAX; i++) {
        ft->ft_fhs[i] = NULL;
    }
    ft->ft_lock = lock_create("K_FT_lock");
    if (ft->ft_lock == NULL)  goto cleanup1;

    return ft;

cleanup1:
    fh_close(con_fh[2]);
cleanup2:
    kfree(con_path);
    vfs_close(con_dev[2]);
cleanup3:
    fh_close(con_fh[1]);
cleanup4:
    vfs_close(con_dev[1]);
cleanup5:
    fh_close(con_fh[0]);
cleanup6:
    vfs_close(con_dev[0]);
cleanup7:
    kfree(ft);
    return NULL;
}
```

The following function is used to retrieve **file_handle**'s by file descriptors:

```
/*
 * gets the file_handle related to the given fd and a process
 */
struct file_handle *
ft_get(int fd, struct proc *proc)
{
    struct file_handle *fh;

    uint8_t ftid = proc->p_fds[fd];
    KASSERT(ftid < FT_MAX);

    lock_acquire(k_filetable->ft_lock);
    fh = k_filetable->ft_fhs[ftid];
    lock_release(k_filetable->ft_lock);

    return fh;
}
```

The following functions are used to increment or decrement the **fh_refcount** of a **file_handle**:

```
/*
 * Increments the refcount of a given file handle
 */
void
fh_incref(struct file_handle *fh) {

    lock_acquire(fh->fh_ref_lock);
    fh->fh_refcount++;
    lock_release(fh->fh_ref_lock);
}

/*
 * Increments the refcount of a given file handle
 */
void
fh_decref(struct file_handle *fh) {

    lock_acquire(fh->fh_ref_lock);
    fh->fh_refcount--;
    lock_release(fh->fh_ref_lock);
}
```

The global **file_handle** table will be stored in:

```
struct file_table *k_filetable;
```

Fork

To implement fork, we plan on doing the following:

- Acquire the process table’s lock.
- Go through the process table to find the first free pid. If no pid is available error with **ENPROC**.
- Create a new process (checking for NULL return value from **proc_create**).
- Make the **k_proctable** entry point to the new process

- Release the process table's lock.
- Set the values of the new process as follows:
    - Set **p_parent** to the pid of the current process.
    - Set **p_cwd** to be the same as the current process' **p_cwd**. Then call **VOP_INCREF** on it to increase the refcount of the vnode.
    - Copy **p_fds** from the parent process
    - For each **p_fd**, call **ft_get** to get the associated **file_handle** and if it's valid call **fh_incref** to increment its refcount.
    - Set **p_pid** to the new pid we previously found.
    - There's no need to initialize **p_exit_code**.
    - Set **p_state** to **P_ALIVE**.
    - Initialization of **p_lock** and **p_wchan** should be handled by **proc_create**.
- Next, create a copy of the old process' address space using **as_copy** and set the new process' address space to it, checking the return value to make sure we didn't run out of memory. If we did, error out with **ENOMEM**.
- Make a copy of the current trapframe in the heap
- Then, create a new thread using **thread_fork** that will execute the child process, passing in the name of the new process as name, **enter_forked_process** as the entrypoint, the new process as **proc** and the current trapframe as **data1**.
- In **enter_forked_process**:
    - Copy the trapframe from the heap into the new thread's strack
    - update the trapframe as follows:
        - Set **tf->tf_a3** to 0 to indicate no error.
        - Set **tf->tf_v0** to 0 to indicate that it's the child process.
        - Increment **tf->tf_epc** by 4 to avoid fork bombs!
    - At this point do some **KASSERT**s to make sure spl is low and we aren't holding any spinlocks
    - Finally, wake up the new process and give it to the user by calling **mips_usermode**.
- Back in the main thread, set **retval** to the **pid** of the child.

Execv

Execv executes a file and requires a file path and an argument list, so to implement this we plan on doing the following:

- Acquire a copy buffer.
- Open the file as a vnode using **vfs_open**. We don't have to add this to our file descriptor table because we're going to close it soon. If **vfs_open** returns a non-zero value, error with that.

- Copy the user supplied **args** into the kernel heap and keep track of them in a linked list. Count how many there are..
- We then create a new address space using **as_create**.
- We deactivate the old address space by using **as_deativate**, then set the address space of the proc to the new address space using **proc_setas** and **as_activate**, while keeping track of the old address space.
- Use **load_elf** to load the executable that we previously opened using **vfs_open**.
- We define the user stack using **as_define_stack**.
- Close the file using **vfs_close**.
- Initialize an array of **argc** pointers in the kernel for keeping track of the arg positions in the new address space.
- Initialize a counter variable that will count how big the total argument vector is and initialize it to 0.
- Swap back to the old address space using **as_deactivate**, **proc_setas** and **as_activate**.
- Now for every argument that the user supplied, starting from the last one (**args[argc-1]**) we do the following:
    - Use **copyinstr** to copy the the argument into the kernel buffer.
    - Increment the counter variable with the number of bytes copied. If the counter variable becomes greater than **ARG_MAX**, error out with **E2BIG**.
    - Calculate the amount of padding necessary.
    - Swap back to the new address space using **proc_setas** and **as_activate**.
    - Use **copyoutstr** to copy the arguments into the new address space at the top of the current user stack. Adding in NULLs at the top of the stack if necessary to ensure 4 byte alignment.
    - Save the pointer to the just copied argument in the arg position array at the appropriate index.
    - Move the stack pointer down to just below the just copied argument.
- At this point all the arguments should be copied into the new address space.
- Release the copy buffer
- Write 4 bytes of NULL characters as padding at the top of the user stack and move the stack pointer down 4 bytes.
- Copy all of the pointers to the arguments that we previously stored to the top of the user stack, starting from the last one and moving the stack pointer down appropriately.
- Destroy the old address space
- Call **enter_new_process** to swap back into user mode.

Throughout this function, we should check to make sure we don’t run out of memory, and check for errors on each of the other function calls.

waitpid/exit

_exit

When a process is done, it should call **_exit**.
In **_exit**, we first check the global orphaned process coffin **k_proctable->pt_coffin**. If there is a process in it, we call **proc_destroy** on it.

Acquire the spinlock of the exiting process. This is necessary to avoid a deadlock situation where:

1. a parent process which calls waitpid, checks the state of a process, sees that it is still alive
2. The parent gets descheduled and the child calls exit.
3. The child goes through **_exit**, calling **cv_signal** to wake up any waiting parents.
4. The child finishes and the parent is scheduled back on. The parent adds itself to the child's wait channel and is never woken up.

Once the lock has been acquired we do the following:

- Copy the passed in exitcode to the process' struct. Using the MKWAIT_EXIT macro if the process called exit and the MKWAIT_SIG macro if the we got here through kill_curthread
- Call close on all of the process' open file descriptors.
- Change the **p_state** to **P_ZOMBIE**.
- Clear any children the process has by calling **waitpid** with **NOHANG** on all of its children.
- Release **p_lock**.
- Check if the process' parent is still active. To do this call **pt_get_proc** passing in the current process's **p_parent**. If **pt_get_proc** returns NULL, then assume that the parent is dead. Otherwise check the parent process' **p_status** to determine if it's active or dead.
    - If the parent is active call acquire the **p_waitlock** and call **cv_signal** on the process' **p_cv**.
    - If the parent is dead, set the value of the **k_proctable->pt_coffin** to this process' **p_pid**
- If the parent was alive, release **p_waitlock**.
- Call **thread_exit**

During **thread_exit** the thread should be shut down and made reapable. The process is now ready to be cleaned

Waitpid

When waitpid gets called we first check if the supplied options are valid by comparing them to all the available options (currently only valid option is **WNOHANG**). If the option is invalid we error with **EINVAL**. We then check if the supplied PID is valid by making sure that it is greater than **PID_INVALID** and smaller than **PID_MAX**. If it is not we error with **ESRCH**. If the pid is valid, we try to get the child process' struct by calling **pt_get_proc**. If it returns **NULL**, error with **ESRCH**. we check if the current process is its parent by comparing **curproc->p_pid** to the target process' **p_parent** value. If the target is not a child we error with **ECHILD**. At this point we acquire the target process' **p_lock** and do the following:

- Check the child process' **p_state**.
- If it's **P_ALIVE** and **WNOHANG** was provided for the **waitpid** system call, then we release the process' spinlock and return -1.
- If it's **P_ALIVE** and **WNOHANG** was not provided, then we acquire the **p_waitlock** lock and call **cv_wait** on the child process's **p_cv**. Until we're woken up. At this point the child should have terminated
- After this we do the following:
- we move the exit code stored in **p_exit_code** to the provided memory location in userspace using **copyout** (after checking for **EFAULT**), remove the child from the parent's list of children, release the process' lock and clean the process using **proc_destroy** and return 0.

In **proc_destroy** we have to:

- Free the memory used to hold **p_name**.
- Deactivate and destroy **p_addrspace**.
- Destroy **p_cv**.
- Decrease the refcount of **p_cwd**.

Kill_curthread

Since each process can only have one thread, we can just call **kern__exit**., which is a helper function that does most of the work for **sys_exit**, but allows you to pass in a signal code.

Getpid

For **getpid** we simply return **curproc->p_pid**.

File I/O

Uio_uinit

In order to support the easy transfer of data between userspace and a file, we define a new function **uio_uinit**, which is similar to **uio_kinit**, but instead of preparing a **uio** that can be used to transfer data between the kernel and a file, it prepares a uio that can be used to transfer data between userspace and a file. The code for it is:

```
/*
 * Convenience function to initialize an iovec and uio for userspace I/O.
 */

void uio_uinit(struct iovec *iov, struct uio *u, struct addrspace *uspace,
        userptr_t ubuf, size_t len, off_t pos, enum uio_rw rw)
{
    iov->iov_ubase = ubuf;
    iov->iov_len = len;
    u->uio_iov = iov;
    u->uio_iovcnt = 1;
    u->uio_offset = pos;
    u->uio_resid = len;
    u->uio_segflg = UIO_USERSPACE;
    u->uio_rw = rw;
    u->uio_space = uspace;
}
```

Open

- Find the first free file descriptor in the process's **p_fds**, erroring with **EMFILE** if one can not be found.
- Acquire the **k_filetable->ft_lock**.
- Find the first empty slot in **k_filetable->ft_fhs**. If not error with **ENFILE**.
- Call **vfs_open** to get a new **vnode** that points to the newly acquired file. If the call returns an error value, return **EINVAL** or **EFAULT**.
- Create a new **file_handle** using **fh_init** to create a new file handle, passing in the new **vnode** and the provided flag. If **fh_init** returns NULL error with **ENOMEM**.
- Make the previously found entry in **k_filetable->ft_fhs** point to the new **file_handle**.
- Release the **k_filetable->ft_lock**.
- Set the previously found entry in the process' **p_fds** table to point to the **k_filetable->ft_fhs** index, where we stored the new **file_handle**.
- Update the return value in syscall.c to the new file descriptor.
- Return 0.

Always remember to release **k_filetable->ft_lock** before erroring if it is held!

Read

- Get the **file_handle** using **ft_get**. If it returns NULL, error with **EBADF**.
- Check the **file_handle**'s open flags **fh_open_flags**. If the file was not opened for reading, error with **EBADF**.
- Acquire the **file_handle**'s **fh_use_lock**.
- Make a new **uio** using **uio_uinit** passing in a blank **uio** and **iovec**, a pointer to the user address space (**curproc->p_addrspace**), a pointer to the user supplied buffer **buf**, the length of the buffer **buflen**, the current offset of the file **fh_off** as **pos** and **UIO_READ** as **uio_rw**.
- Call **vop_read** on **fh_file** and the new **uio**, and check that it succeeded. If it failed return the error it failed with.
- Update **fh_off** by adding the value that **vop_read** returned.
- Release the **file_handle**'s **fh_use_lock**.
- Update the return value in syscall.c to the number of bytes read, stored in the uio.
- Return 0.

Always remember to release the **fh_use_lock** before erroring if it is held!

Write

- Get the **file_handle** using **ft_get**. If it returns NULL, error with **EBADF**.
- Check the **file_handle**'s open flags **fh_open_flags**. If the file was not opened for writing, error with **EBADF**.
- Acquire the **file_handle**'s **fh_use_lock**.
- Make a new **uio** using **uio_unit** passing in a blank **uio** and **iovec**, a pointer to the user address space (**curproc->p_addrspace**), a pointer to the user supplied buffer **buf**, the length of the buffer **bufflen**, the current offset of the file **fh_off** as **pos** and **UIO_WRITE** as **uio_rw**.
- Call **vop_write** on **fh_file** and the new **uio**, and check that it succeeded. If it failed return the error it failed with.
- Update **fh_off** by adding the value that **vop_write** returned.
- Release the **file_handle**'s **fh_use_lock**.
- Update the return value in syscall.c to the number of bytes written, stored in the uio.
- Return 0.

Always remember to release the **fh_use_lock** before erroring if it is held!

Lseek

- Get the **file_handle** using **ft_get**. If it returns NULL, error with **EBADF**.

- Acquire the **file_handle**’s **fh_use_lock**.
- Use **vop_isseekable** to check if the **fh_file** is seekable. If it is not error with **ESPIPE**.
- Do the following based on the **whence** argument:
    - **SEEK_SET**: Check if **pos** is negative. If it is, error with **EINVAL**. Otherwise set **fh_off** to **pos**.
    - **SEEK_CUR**: Check if **pos** + **fh_off** is negative. If it is error with **EINVAL**. Otherwise add **pos** to **fh_off**
    - **SEEK_END:** Create a new **struct stat st_buff** and call **vop_stat** to get the stats on the file. Then check if **st_buf.st_size** + **pos** is negative. If it is, error with **EINVAL**. If it is not, then set **fh_off** to **pos** + **st_buf.st_size**.
    - If the **whence** argument is none of above, error with **EINVAL**.
- Put the new **fh_off** in the return value in syscall.c.
- Release **fh_use_lock**.
- Return 0.

Always remember to release the **fh_use_lock** before erroring if it is held!

Close

- Acquire the **k_filetable->ft_lock**.
- Index into the process’s file descriptor table **p_fds**. If the index isn’t valid error with **EBADF**. If it is valid, follow it into **k_filetable->ft_fhs** and call **fh_close** on the associated **file_handle**.
- Set the entry in **k_filetable->ft_fhs** to whatever **fh_close** returns.
- If that return value was NULL, change **p_fds[fd]** to **FD_FREE**.
- Release the system **k_filetable->ft_lock**.
- Return **0**.

Always remember to release **k_filetable->ft_lock** before erroring if it is held!

Dup2

- Use **ft_get** to attempt to get the file handle of **oldfd**. If it returns NULL, error with **EBADF**.
- Check if **oldfd** is the same as **newfd**, and if they are, update the syscall.c return value to **newfd** and return 0.
- Use **ft_get** to attempt to get the file handle of **newfd**. If it doesn’t return NULL, call **close** on **newfd**.
- Acquire the **fh_ref_lock** for the old **file_handle**, increment **fh_refcount**, and release the lock.
- Set the value of **p_fds[newfd]** to be the same value as **p_fds[oldfd]**.
- Update the syscall.c return value to **newfd** and return 0.

Chdir

- Make a copy of **pathname** in kernel space by creating a buffer of size **PATH_MAX**, and using **copyinstr** (kern/vm/copyinout.c) to copy the string into kernel space. If this call does not return 0, error with the value it returned.
- Return **vfs_chdir** (kern/vfs/vfscwd.c), passing in the copied **pathname** as the parameter.

__getcwd

- Make a new **uio** using **uio_unit** passing in a blank **uio** and **iovec**, a pointer to the user address space (**curproc->p_addrspace**), a pointer to the user supplied buffer **buf**, the length of the buffer **bufflen**, 0 as **pos** and **UIO_READ** as **uio_rw**.
- Use **vfs_getcwd** (kern/vfs/vfscwd.c), passing in the **uio** to copy the path string into userspace.
- If **vfs_getcwd** returns 0, update the syscall.c return value to the length of the data read and return **0**. Otherwise return the value **vfs_getcwd** returned.

Scheduling

We first implemented a multilevel feedback queue that allows for process aging (preventing starvation), prioritizes tasks that yield frequently, and prevents priority inversion. This scheduler will be used if **USE_PRIORITY_SCHEDULER** is uncommented in thread.h.

Data Structures

- Limits.h: define new limits **PRIORITY_MAX** set to 20 and **YIELD_BOOST** set to 10.
- Thread.h: #define some variable so the scheduler can easily be turned on and off, and add an enum **threadyield_t** with values **YIELD_FORCED**, **YIELD_VOLUN**, and **NOT_RUN**.
- Thread.h: in the thread struct, add a uint8_t **t_priority** representing regular priority, and a uint8_t **t_ptotal** that represents the total priority (regular + **YIELD_BOOST** if the thread voluntarily yielded). Also add an enum **t_yielded** that indicates whether the thread voluntarily yielded since the last call to **schedule**.
- Synch.h: in the lock struct, add a priority value **lk_priority**.
- Cpu.h: in the cpu struct, add an array of threadlists **c_plists** of size **PRIORITY_MAX + YIELD_BOOST**.

Cpu_create

- Iterate through **c_plists**, initializing all of the threadlists.

Lock_create

- Set **lk_priority** to 0.

Lock_acquire

- Before the lock is acquired, if the current thread's **t_priority** is lower than the lock's **lk_priority**, update **t_priority** to equal **lk_priority**.
- After the lock is acquired, update **lk_priority** to equal **t_priority**.

Lock_release

- Set **lk_priority** to 0.

Thread_create

- Set **t_priority** to 0.
- Set **t_yielded** to **NOT_RUN.**

Thread_switch

- If **t_yielded** is not **YIELD_FORCED**, change it to **YIELD_VOLUN**.

Hardclock

- Change **t_yielded** to equal **YIELD_FORCED**.

Schedule

- Acquire the run queue lock.
- For each thread in the run queue:
    - If **t_yielded** is **NOT_RUN**, increase **t_priority** by 1. Else, set **t_priority** to 0.
    - Set **t_ptotal** to **t_priority**. If **t_yielded** is **YIELD_VOLUN**, increase **t_ptotal** by **YIELD_BOOST**. Set it to **NOT_RUN**.
    - Move the thread to **c_plists[t_ptotal]** using **threadlist_remhead** and **threadlist_addtail**.

- Iterate backward through **c_plists** using **threadlist_remhead** to remove each thread and **threadlist_addtail** to add it to the run queue.
- Release the run queue lock.

We also implemented an nclock scheduler, which can be turned on by uncommenting **USE_NCLOCKED_SCHEDULER** in thread.h. Every time **thread_switch** is called, the new thread's priority is increased; and when **schedule** is called, the runqueue sorted by priority in increasing order.

Plan of action

We were planning on finishing all system calls by Saturday, February 18th. We actually finished them on Sunday, February 19th, which is pretty close to our goal. We spent Monday the 20th designing and implementing the first pass of the scheduler, but it was pretty slow and we let it sit until Thursday night. We were going to spend Wednesday and Thursday doing more testing, but we gave up on that and just worked on the scheduler and fine-tuning.

Credits

We would like to thank Prof. James Mickens for teaching us the concepts, Team Harvard Turkey and Alex Patel for helping us refine our design, and Victor Domene for debugging with us.