Significant updates since initial Design Doc submission:

- The lock and cv in the `proc` struct have been replaced with a spinlock and wchan because locks KASSERTed from interrupts
- The two locks in vfile have been combined into a single spinlock that protects refcount and offset, and most importantly a uio field was abandoned in favor of a vf_offset and vf_flags field.
- We scrapped the idea of combining independent file descriptors for unseekable/read-only files because it would be needlessly complicated (and add unnecessary overhead to each vfile creation)

Scheduling:

We tried several different types of schedulers and tried to fine tune the constants involved, but none produced meaningfully different results in the test case of `schedpong`, unless the pongers were favored at the severe detriment of the thinkers, which we did not deem particularly fair.

We settled on a multilevel (3 levels) feedback queue, with two priority flags (low priority for neither, medium for either, and high for both):

- io_priority (turned on by most I/O system calls that imply future calls, like open(), read(), write(), or lseek(), but not ones like close())
- sleep_priority (turned on by wchan_sleep()).

Each time a thread is scheduled, another field for switches_left decrements from a defined DEPRIORITIZE_THRESHOLD, here 16 (could be tuned for performance in the future if desired), and when it reaches 0, its priorities are reset to false and it's placed in the bottom queue.

The three queues are:

- `c_hp_runqueue`, the high priority FIFO queue
- `c_mp_queue`, the medium priority FIFO queue
- `c_runqueue`, the low priority FIFO queue; elements from the other queues are deprioritized and return here over time

Jobs from these queues are chosen in a 1:2:5 ratio based on a modulus of curcpu's hardclock counter. This (along with FIFO policy) ensures that while higher priority threads are given more CPU time, low priority ones are not starved.

Computer Science 161: Operating Systems
Assignment 2 Design Document

David Li and Garrett Tanzer[1]
February 10, 2017

1. Introduction

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

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()`, `close()` and other I/O system calls will act at a per-process level but remain synchronous with others through the underlying file system. The main challenge in I/O is assuring safe transfer of data to and from user space while in kernel mode, as well as ensuring robust error checking.

Our implementation of processes will require some additions to the `proc` struct and the implementation of a global array that stores pointers to 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.

2. Overview

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, `p_fds`, with arbitrary length `MAX_FDS`, whose values correspond the indices in a single, global array (as defined in array.h) of `struct vfiles`, which will be called `vfiles`. The following is the definition of `vfile`:

```
struct vfile {
      char *vf_name;                /* unique identifier for comparison */
      struct lock vf_flock;         /* protects vf_uio access */
      struct lock vf_rlock;         /* protects refcount access */
      struct uio *vf_uio;           /* uio with metadata for I/O */
      struct vnode *vf_vnode;       /* abstract file representation */
      int refcount;                 /* number of fds pointing here */
};
```

When a file is opened, either its existing `vfile` is located (if the file is not seekable) or a new one is opened, then placed in the global array. The pointer to the `vfile` is placed as the value in the process-level array at an arbitrary free index, and that arbitrary index (the file descriptor) is returned for future use. -1 represents an unused slot in the per-process array, while NULL does

[1] Our review partners were Carlos Mendizábal and Davey Hughes. The main changes we made in response to their feedback were significant updates to our `vfile` struct (the inclusion of `uio` particularly), broader use of array.h, and much better documentation of required error codes.

for the global array. `read()`, `write()`, etc. will now translate this file descriptor into the underlying virtual file, preventing redundant file access and helping to ensure synchronization for shared files. There can be multiple instances of a single seekable file in the global array to facilitate specific behavior when forking that will be explained later. Two locks are used to protect each `vfile` so that `read()`, `write()`, and `lseek()` do not unnecessarily block `open()`, `close()`, and `dup2()`.

We considered using a combination of a linked list of removed elements and a monotonically increasing index so we wouldn't need to iterate over all file descriptors (it's explained more thoroughly in our peer review commit), but we decided that such a convoluted scheme will probably not be necessary (and would not work for unseekable files, where we will iterate over the array to find existing entries).

In order to keep track of multiple runnable processes, we're adding a global array (as defined in array.h) of `struct proc`s called `procs`. We will modify the `struct proc` to facilitate this organization scheme; 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 */
      struct proc *p_parent;         /* pointer to parent process */
      pid_t pid;                     /* index for global procs, process id */
      cv *p_cv;                      /* parent waits on child's cv */
      int p_fds[MAX_FDS];            /* array of file descriptors */
      int exit_code;                 /* only used after _exit() */
};
```

Each `proc`, whose index in the global array is represented by its `pid`, will be protected by its own lock, 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 find children's `p_cv`s for `waitpid()` and to help deal with zombie/orphan processes in `_exit()`. The `p_fds` array was explained above. Process creation and destruction will have to be adjusted to properly initialize and clean up these new fields.

3. Topics

File Descriptors

Much of the nuance in file descriptor behavior comes from interaction with `fork()`. Files that were already opened prior to forking share the same offset position even after they diverge, meaning that they must share their `vf_uio`, and so share their entry in the global file descriptor table.

By default, 0, 1, and 2 in each process's `p_fds` refer to the global 0, 1, and 2, which are permanently stdin, stdout, and stderr. These can be redirected like any other file descriptor, simply represented by changing the integer value stored at a certain index (`oldfd`).

Here is an example of file descriptor behavior with our specific data structures. The contents of `vfiles` are simplified for clarity. Some file descriptors are "out of order" in the per-process table to demonstrate that assignments to slots are not guaranteed to be meaningful.

	0	1	2	3	4	5
`vfiles`	std in	std out	std err	foo.txt offset 78	/dev/null	foo.txt offset 12

	0	1	2	3	4	5	
`p_fds`	0	1	2	4	3	-1	**1**

	0	1	2	3	4	5	
`p_fds`	0	1	2	-1	3	-1	**2**

	0	1	2	3	4	5	
`p_fds`	0	4	2	1	5	4	**3**

One possible way to obtain this structure is:

- Process 1 opens foo.txt, then forks Process 2. Now actions on fd = 4 from both Process 1 and 2 share an offset.
- Process 1 opens /dev/null.
- Process 3 is running a separate program and opens /dev/null and foo.txt. foo.txt has its own file description with a unique offset, while /dev/null points to the same description because it is not seekable and thus does not need distinct offsets.
- Process 3 redirects file descriptor 3 to 1 (stdin), and then 1 to 5.

For reference, the `vfile` associated with an int `fd` may be accessed with (we will define a macro VFILES_GET(fd)): `vfilearray_get(vfiles, curproc->p_fds[fd]).`

Processes

Empty slots in the array will be initialized/reset to NULL, and a single lock will protect write access to all NULL slots until a lock has been created specifically for it (like in the file descriptor table). There is no "special" process like `init`, so processes can have any pid >= 0. See the sections below on `fork()`, `execv()`, `waitpid()`, and `_exit()` for more on processes, including our strategy for reaping zombies and avoiding orphans.

A process with pid `pid` is accessed by `procarray_get(procs, pid)`. (We will implement a macro PROCS_GET(pid) for this functionality.) It is crucial here (as in `vfiles`) that we use array.h's get and set methods—NOT add and remove—because those shift around the members of the array, which ruins the value of pids and fds as indices.

System Calls

In order to implement a system call in OS/161, changes need to be made in several locations:

- Add a function header to ~/cs161/os161/kern/include/syscall.h
- Check if syscall number is defined in ~/cs161/os161/kern/include/kern/syscall.h, and if not, add it or uncomment its existing entry
- Add a case in the switch statement 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 will have two additional files in /syscall : one for I/O-related system calls and another for process-related ones)
- Add the user-facing header to ~/cs161/os161/userland/include/unistd.h
- Update conf.kern accordingly

All the system calls below return -1 and set `errno` accordingly upon failure unless otherwise noted. Error codes and their descriptions are from the OS/161 Reference Manual.

```
int open(const char* pathname, int flags);
```

Open the file described by `pathname` with flags `flags`, and return the file descriptor. These flags must include exactly one of the following:

`O_RDONLY`	Open for reading only.
`O_WRONLY`	Open for writing only.

`O_RDWR`	Open for reading and writing.

These flags may also include any number of the following (though O_EXCL is only meaningful when paired with O_CREAT).

`O_CREAT`	Create the file if it doesn't exist.
`O_EXCL`	Fail if the file already exists.
`O_TRUNC`	Truncate the file to length 0 upon open.
`O_APPEND`	Open the file in append mode.

`open()` serves mainly as a wrapper for `vfs_open()`, where the additional functionality comes from navigating the structure of file descriptors.

First we iterate through the global array of file descriptors and find the first slot with value `NULL`; this `NULL`-checking for the entire array will be protected by a single lock (not the fine-grained locks for each entry, because those don't exist yet for a `NULL` entry). Once this is found (unless the file is not seekable, in which case the entire array needs to be checked for matches), a `vfile` is created and its lock is acquired instead of the one for `NULL`-checking. If a match is found for an unseekable file, we increment its refcount (with lock protection), add its index to `p_fds`, and return.

Now `vfs_open()` is used to actually open the file and the `vfiles` entry is finalized. `uio_uinit()`, as described below in the helper functions section, will be used to create `vf_uio`. The two locks in `vfile` should be initialized and `refcount` set to 1 initially.

Finally, in order to create the local file descriptor, we iterate through `p_fds` and find the first available slot with value -1. We do not need to worry about synchronization here because multithreading is not implemented, so there is only ever one thread inside a process's fd table at a time. Return the file descriptor.

The following errors must be handled:

`ENODEV`	The device prefix of filename did not exist.
`ENOTDIR`	A non-final component of filename was not a directory.
`ENOENT`	A non-final component of filename did not exist.
`ENOENT`	The named file does not exist, and O_CREAT was not specified.
`EEXIST`	The named file exists, and O_EXCL was specified.
`EISDIR`	The named object is a directory, and it was to be opened for writing.

EMFILE	The process's file table was full, or a process-specific limit on open files was reached.
ENFILE	The system file table is full, if such a thing exists, or a system-wide limit on open files was reached.
ENXIO	The named object is a block device with no filesystem mounted on it.
ENOSPC	The file was to be created, and the filesystem involved is full.
EINVAL	flags contained invalid values.
EIO	A hard I/O error occurred.
EFAULT	filename was an invalid pointer.

`ssize_t read(int fd, void *buf, size_t buflen);`

Read up to `buflen` bytes from the file represented by `fd` at the current offset, then store the result in the space indicated by `buf`. Advance the seek location by the number of bytes read. Return the number of bytes successfully read, or 0 if end-of-file.

`read()` serves mostly as a wrapper for `VOP_READ()`. Some prep work will need to be done to make `vf_uio` reference the appropriate locations and address space (namely setting `uio_iov->iov_base` to `buf`, `uio_rw` to `UIO_READ`, and `uio_space` to `curproc`'s address space). Reading is protected by `vf_flock`, except unseekable files, which can be read simultaneously.

The following errors must be handled:

EBADF	fd is not a valid file descriptor, or was not opened for reading.
EFAULT	Part or all of the address space pointed to by buf is invalid.
EIO	A hardware I/O error occurred reading the data.

`ssize_t write(int fd, const void *buf, size_t buflen);`

Write up to `buflen` bytes from the location indicated by `buf` to the file represented by `fd` at the current offset. Advance the seek location by the number of bytes written. Return the number of bytes successfully written, or 0 if end-of-file in a fixed-size file.

`write()` serves mostly as a wrapper for `VOP_WRITE()`. Some prep work will need to be done to make `vf_uio` reference the appropriate locations and address space (namely setting `uio_iov->iov_base` to `buf`, `uio_rw` to `UIO_WRITE`, and `uio_space` to `curproc`'s address space). Writing is protected by `vf_flock`.

The following errors must be handled:

`EBADF`	fd is not a valid file descriptor, or was not opened for writing.
`EFAULT`	Part or all of the address space pointed to by buf is invalid.
`ENOSPC`	There is no free space remaining on the filesystem containing the file.
`EIO`	A hardware I/O error occurred writing the data.

```
off_t lseek(int fd, off_t pos, int whence);
```

Move the current seek position of the file represented by `fd` based on `pos` (which is notably signed) and `whence`. If `whence` is:

`SEEK_SET`	the new position is `pos`
`SEEK_CUR`	the new position is the current position plus `pos`
`SEEK_END`	the new position is the end-of-file plus `pos`

Every time a file is opened, that instance has a separate seek position, which means that file descriptors opened before a `fork()` share their seek position (for that file) in the parent and child process. Changing offset is accomplished by modifying `vf_uio`.

The following errors must be handled:

`EBADF`	`fd` is not a valid file handle.
`ESPIPE`	`fd` refers to an object which does not support seeking.
`EINVAL`	`whence` is invalid.
`EINVAL`	The resulting seek position would be negative.

```
int close(int fd);
```

Close the file descriptor `fd`, and the file represented by `fd` if it is the only reference to it. Return 0 on success.

In OS/161, this means change the value of `curproc->p_fds[fd]` to -1, decrease the `refcount` of the associated `vfile`, then remove the entry from `vfiles` if `refcount` is now 0 (properly cleaning up the locks and other fields of `vfile`). `flock` is used to protect `close()` (until it needs to be cleaned up, at which point it switches to the NULL-checking lock mentioned in `open()` documentation).

The following errors must be handled:

`EBADF`	fd is not a valid file handle.
`EIO`	A hard I/O error occurred.

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

Clone the file descriptor `oldfd` onto `newfd`. Close the file previously represented by `newfd`, if any. Return `newfd` upon success.

In OS/161, we accomplish this by first checking whether `curproc->p_fds[newfd]` has a value other than -1, and if so calling `close()` on it. Next we set the value at index `newfd` to the existing value at index `oldfd`, then increment the `refcount` of the associated `vfile`. `dup2()` acquires `vf_rlock` when modifying the `vfile`, but its changes to `p_fds` do not need synchronization because we are not implementing multithreading support.

The following errors must be handled:

`EBADF`	`oldfd` is not a valid file handle, or `newfd` is a value that cannot be a valid file handle.
`EMFILE`	The process's file table was full, or a process-specific limit on open files was reached.
`ENFILE`	The system's file table was full, if such a thing is possible, or a global limit on open files was reached.

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

      // check invalid params

      if(curproc->p_fds[newfd] != -1)
            close(newfd);

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

      lock_acquire(VFILES_GET(fd).vf_rlock);
      VFILES_GET(fd).refcount++;
      lock_release(VFILES_GET(fd).vf_rlock);

```

```
        // error handling
        return newfd;
}
```

`int chdir(const char *pathname);`

Set the working directory to the given path name. Return 0 upon success.

In OS/161, serves as a wrapper to call `vfs_chdir()` and handle errors.

The following errors must be handled:

ENODEV	The device prefix of `pathname` did not exist.
ENOTDIR	A non-final component of `pathname` was not a directory.
ENOTDIR	`pathname` did not refer to a directory.
ENOENT	`pathname` did not exist.
EIO	A hard I/O error occurred.
EFAULT	`pathname` was an invalid pointer.

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

Store the name of the current working directory in the space pointed to by `buf`, an area of length `buflen`—do not zero terminate. Return the size of the data copied in bytes.

In OS/161, serves as a wrapper to call `vfs_getcwd()` and handle errors. We'll use `uio_uinit()` to create the `uio` used as `vfs_getcwd()`'s parameter.

The following errors must be handled:

ENOENT	A component of the pathname no longer exists.
EIO	A hard I/O error occurred.
EFAULT	buf points to an invalid address.

`pid_t getpid(void);`

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

`getpid()` never fails. Hooray!

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

      return curproc->pid;
}
```

`pid_t fork(void);`

Duplicate the current process, but give it a different pid. Do a deep copy of address space and a shallow copy of existing file descriptors. Return the child's pid to the parent process and 0 to the child.

In OS/161, first we make a new `proc` struct and fill it with the same data as `curproc`, except with the next available pid in `procs`. We put this `proc` in `procs`, then call `enter_forked_process()` from `syscall.c`.

This function will be based substantially on `thread_fork()` from `thread.c`, but will deep copy the entire address space (using `as_copy()` as described in the helper function section) and relevant contents of the trapframe into a new `switchframe`—with the exception of the return value (register v0), which will be set to 0.

This `switchframe` will be attached to a new thread (again with properties copied from the original thread), and then we'll call `thread_make_runnable()` on it. The original trapframe will have the return value set to the new process's pid that we determined earlier, and the system call will return.

The following errors must be handled:

`EMPROC`	The current user already has too many processes.
`ENPROC`	There are already too many processes on the system.
`ENOMEM`	Sufficient virtual memory for the new process was not available.

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

Replace the current process with a new program. `program` is the name of the program to run, and `argv` is an array of null-terminated strings, itself terminated by a `NULL` pointer. This array has maximum size `ARG_MAX`. The pid and file descriptor table are left unchanged.

The OS/161 implementation will be largely similar to `runprogram()`, but it will need to construct a new stack based on the parameters of `execv()`. This will be accomplished by copying `argv` and the values it points to from the user address space with `copyin()` and `copyinstr()`, then to a new address space with `uio`.

This data will be arranged from the top of the stack down, as follows:

- the values pointed to by `argv`, aligned to 4 bytes and padded with null characters
- four null characters (a `NULL` pointer)
- pointers to the values stored above

To clarify, here is an example memory diagram provided by Peter Kraft in section notes:

800	
799	$\emptyset$
798	o
797	o
796	f
795	[padding]
794	$\emptyset$
793	s
792	l
791	$\emptyset$
790	$\emptyset$
789	$\emptyset$
788	$\emptyset$ [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

The address of final piece of data stored (`argv[0]`) will be set as the stack pointer. When we then call `enter_new_process()`, a new trapframe is constructed with other register state reset.

The following errors must be handled:

`ENODEV`	The device prefix of `program` did not exist.
`ENOTDIR`	A non-final component of `program` was not a directory.
`ENOENT`	`program` did not exist.
`EISDIR`	`program` is a directory.
`ENOEXEC`	`program` is not in a recognizable executable file format or is for the wrong platform
`ENOMEM`	Insufficient virtual memory is available.
`E2BIG`	The total size of the argument strings exceeds ARG_MAX.
`EIO`	A hard I/O error occurred.
`EFAULT`	One of the args is an invalid pointer.

`pid_t waitpid(pid_t pid, int *status, int options);`

Block until the process identified by `pid` exits (or act immediately if it already has), then collect and store its encoded exit code in the region referenced by `status`. If `status` is `NULL`, do not do anything with the encoded exit status. Return `pid` upon success. Currently no `options` are supported. In OS/161, only the parent of a process can wait on it.

Exit status should be encoded with the `_MKWAIT` flags. The macros `WIFEXITED()`, `WIFSIGNALED()`, and `WIFSTOPPED()` (to find out what happened); `WEXITSTATUS()`, `WTERMSIG()`, and `WSTOPSIG()` (to get the exit code or signal number), and `WCOREDUMP()` (to check if a core file was generated) should be supported.

We will implement `waitpid()` by indexing into the global `procs` array with `pid`, then acquring `p_lock` and checking the value of `exit_code`. If it is the default value, which will be set to -1, then the parent will `cv_wait()` on the child's `p_cv` (it will be woken up when the child calls `_exit()`). When the exit code was or has reached some other value, then the child has exited, and we call `thread_destroy()` on it (with modifications to decrement `proc->p_numthreads` so we can destroy the process with `proc_destroy()`, which needs to be updated to clean up additions to the `proc` struct (like the file descriptors)).

The following errors must be handled:

`EINVAL`	The `options` argument requested invalid or unsupported options.
`ECHILD`	The `pid` argument named a process that was not a child of the current process.
`ESRCH`	The `pid` argument named a nonexistent process.
`EFAULT`	The `status` argument was an invalid pointer.

`void _exit(int exitcode);`

Cause the current thread to exit with the specified exit code (encoded with `_MKWAIT` flags).

Our implementation of `_exit()` can be characterized as the "coffin" approach, where an orphaned process buries itself in a global "coffin" variable so that the next process to exit can destroy it.

First, `_exit()` iterates through `curproc`'s `p_children` array, checking if their exit code is not the default -1. If that is the case (it has exited), call `waitpid()` on it. Otherwise, set the process's `p_parent` field to `NULL`. We don't need to protect this step with synchronization primitives because these fields are only modified by one thread of execution on the parent process at a time. We also check the `coffin` variable that I describe momentarily and `thread_destroy()` it if it isn't `NULL` (then mark it as `NULL`).

Next, `_exit()` checks its own `p_parent` field. If it has a value, the thread calls `thread_exit()` like one might expect, because there is still a parent to reap its zombified child. If `p_parent` is `NULL`, that means the exiting thread is an orphan and doesn't have anyone to reap it. So, before it `thread_exit()`s, it puts a pointer to itself in a global `coffin` variable (type `struct thread *`), which the next thread to `_exit()` will use to put it out of its misery.

`_exit()` has no return value because you never get back to the same thread of execution, and it can't fail. Hooray again!

Miscellaneous Functions

`static void kill_curthread(vaddr_t epc, unsigned code, vaddr_t vaddr);`

Must be updated to support `_exit()` so that exit codes can be communicated productively as signals, rather than causing kernel panic.

`void uio_uinit(struct iovec *iov, struct uio *u, void *kbuf, size_t len, off_t pos, enum uio_rw rw, struct addrspace *as);`

Convenience function to initialize an `iovec` and a `uio` for user process data, as opposed to kernel space with `uio_kinit()`.

```
void uio_uinit(struct iovec *iov, struct uio *u, void *kbuf, size_t len,
      off_t pos, struct addrspace *as) {

      iov->iov_kbase = kbuf;
      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_space = as;
}
```

`int as_copy(struct addrspace *src, struct addrspace **ret);`

Iterate through `src` and place a deep copy of the address space at the location indicated by `ret`. This will require using `copyout()` to get data from `src` and then `copyin()` to put it in the new address space (created with `as_create()`).

Scheduling

We have not yet decided on a specific scheduler and intend to wait until we go over case studies in lecture to do so. (Unless we are misinterpreting the assignment's wording, this is somewhat expected.) In general, we anticipate our scheduler to have a number of priority levels based primarily on number of I/O requests, with aging included to prevent starvation of CPU-bound processes. This will require a couple new fields in the `proc` struct, to be determined once our approach is finalized.

4. Plan of Action

We will largely work together on each step, but where they are not dependent on each other (e.g. on the day with `lseek()`, `dup2()`, `chdir()`, `__getcwd()`, we will divide the work).

Sunday	data structures for processes and file descriptors, `getpid()`
Monday	`open()`, `close()`
Tuesday	`read()`, `write()`
Wednesday	catch-up day
Thursday	`lseek()`, `dup2()`, `chdir()`, `__getcwd()`
Friday	`fork()`
Saturday	catch-up day / `fork()` continued / starting `execv()`
Sunday	`execv()`
Monday	`waitpid()`, `_exit()`
Tuesday	scheduler
Wednesday	catch-up day
Thursday	test
Friday	test