# Project Group 3: Design Document

**Question 1 (file descriptors):** How are file descriptors implemented? What kernel data structures were created to manage file descriptors? Briefly describe the implementation of **`open`**, **`close`**, **`read`**, and **`write`**.

We use fd_init() within fdtable.c to implement a file descriptor. File descriptors are stored in an array owned by each thread that holds the following data: vnode, dups, offset, and a mutex. Kernel structures `fcntl` (for flag definitions) and `iovec` (for flow of data to/from buffer) are created to manage file descriptors.
For **`read`** and **`write`** we use a mutex, acquire the lock to read or write to a vnode. The uio struct is set up with an iovec which contains the user buffer and size. uio is also initialized with offset, address space, and read/write options. VOP_WRITE/VOP_WRITE is called then we release the lock. The number of bytes written are given to the caller through *retval.
For **`open`** we first copy the filename into kernel space using copyinstr. Then, we find an available file descriptor and set up the initial values of the file descriptor, including a vnode (an abstract structure for an on-disk file) and a mutex, within the kernel's file descriptor table. We call vfs_open, which initializes the vnode with the opened file. Finally, the file descriptor is given to the caller through *retval.
For **`close`** we acquire a lock to the specified file descriptor. If there is more than 1 thread using this file, we simply decrement dups. Otherwise, we close its vnode using vfs_close, release and destroy locks created, and free memory created for the specified file descriptor.

**Question 2 (process identifiers):** Briefly explain how you implemented PIDs. How does your kernel generate a PID for each new process? How does your kernel determine that a PID is no longer needed by the process to which it was assigned (and is therefore available for re-use)? Briefly describe the implementation of **`fork`**, **`getpid`**, and **`_exit`**.

PIDs were implemented with the help of a process table. The table contains a list of processes indexed by their PID % NPROCS_MAX. Their PID is generated via a helper function that takes into account the nextpid (a process table attribute). nextpid is just always incremented every time a new pinfo (process info) is assigned to the process table. The kernel determines that a PID is no longer needed by the process to which it was assigned when it sees that the pinfo no longer points to a process.

Fork takes the trapframe passed to syscall as a parameter, and the retval which is the child pid. Fork disables interrupts and maintains a synchronization primitive with the child process to ensure that it will complete before the child runs. The process is copy the trapframe, copy the address space, copy the process, change a few things about the child's process, like it's pid, ppid, and where it's address space is located. After that we fork the calling thread, passing the trapframe and the address space along with it to a new enter_forked_process function that will run when the parent is done running. The parent is about done after this. The child that runs after is makes a trapframe in the stack which copies the passed trapframe but with a few changes. a3 will be 0 to indicate a successful syscall, v0 will be 0 since that's what fork should return for the child, and the epc must be incremented by 4 to avoid doing the same syscall over and over again. The address space is then copied and activated, then kernel goes back to usermode with that trapframe.

Getpid very simply returns the pid of the current process.

_exit change the exitcode to a code that corresponds to a normal exit, and set exited to true. After that it just has to make sure that whoever is waiting on the process (via a condition variable) will be signaled once _exit is done destroying the process and threads associated.

**Question 3 (waiting for processes):** Briefly explain how your kernel implements the waiting required by **`waitpid`**. Did you use synchronization primitives? If so, which ones and how are they used? What restrictions, if any, have you imposed on which processes a process is permitted to wait for?

Waiting has a lot of parameters that need to be checked. We haven't implemented options because we didn't think it was necessary, so any options would return an EINVAL error. After that, we have to check that the status pointer is valid and doesn't point to kernelspace, which we check with USERSPACETOP from vm.h. We then check if the status pointer is properly aligned (%4), if the pid is within the bounds and valid, if the process it points to hasn't already exited, and if the process we're waiting for is a child process (since only parents can wait on their children). If all of these are fine, then we acquire a the pinfo lock, express interest (modifying a pinfo attribute that keeps track of interested processes), and wait for the condition variable. After reacquiring the lock, we are no longer interested in the process. The exitstatus is the child process' exitcode which we copyout. We release the lock, then check if any other processes are interested in the process. If not, we destroy pinfo, which is all that is left of the process.

**Question 4 (argument passing):** How did you implement `argc` and `argv` for **`execv`** and **`runprogram`**? Where are the arguments placed when they are passed to the new process? Briefly describe the implementation of `execv`.

argc is implemented by counting how many argument strings are in char **, which was passed in as an argument. argc is then passed to enter_new_process.
argv is implemented by a stack pointer that points to the first char* in the user address space. argv is then passed to enter_new_process.
First, we copy in the program name and argument strings from user space to kernel space. Then, we get a vnode by calling vfs_open on the program name’s file. An address space is created, which is set as the current process’ address space and the elf is loaded into it. The argument strings from kernel space are written to the current process’ address space followed by the pointers to the argument strings while updating the stack pointer, so that we know where argv starts. Finally, we return to user mode using enter_new_process.