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.

File descriptors are implemented as a struct consisting of the attributes: filename, offset, vnode (the abstraction of the actual file substance), and flags. The offset attribute is used to represent the amount of data read/write into/from a buffer during I/O operations. The flag attribute is used to represent how the file is to be accessed (rights restriction flag) after it is opened.

A file descriptor table(*fd_table*) is used to track the number of files opened within a process. The data structure used for the file descriptor table is an array of pointers to the file descriptor struct. *fd_table* is declared inside the proc struct and is shared among all threads in the process.

A pictorial representation of the *fd_table*
Note: row 0,1,2 are saved for standard IO (these rows are not shown below). However, they will only be allocated when they are needed (lazy init)

fd (or, row #)	**filename**	**offset**	**flags**	**vn**

3	FILE1	0	e.g. O_RDONLY	vn1
				{
				vn1_opencount =1,
				vn1_refcount =1
				}

4	FILE1	0	...	&vn1
				{
				vn1_opencount=1,
				vn1_refcount=2
				}

5	FILE2	0	...	vn2
				{
				vn2_opencount=1
				vn2_refcount =1
				}

6	FILE1	0	...	&vn1
				{
				vn1_opencount=1,
				vn1_refcount=3
				}

Instructions (to match table above):
f1 = open(“FILE1”, offset, flags..) //fd=3

f2 = open(“FILE1”, offset, flags..) //fd=4
f3 = open(“FILE2”, offset, flags..) //fd=5
f4 = open(“FILE1”, offset, flags..) //fd=6

Open & Close

Open: An entry is added to *fd_table* when *open* is called. We don’t want to initialize duplicate vnodes for a file that is opened multiple times. Instead, all entries in *fd_table* that refer to the same vnode should just point to the same one. So technically, the vnode of a file is opened only once, and subsequence openings of the same file will just increment the reference count of this vnode (e.g. fd4 and fd6 refer to fd3’s vnode in the figure above).

The program always perform a scan from fd=3...k to find a vacant spot to insert a new file entry, where k is a vacant row (== NULL) in this case.

Close: An entry is cleared or set to NULL once *close* is called. The close operation decrements the reference count of the vnode. The vnode will only get destroyed/cleaned up when reference count is 1: because if reference count > 1 then this mean the vnode is still being used by someone else.

Read & Write

Both read and write are similar in their implementation. We make use of the low-level vnode read/write (VOP_*) operations available to us to perform the actual data transfer.The concept is that when we are reading from a file or writing to a file we keep track of the progress through a uio struct. The uio will provide statistics that will help with calculating the amount of data transferred, the new offset, and the residual data not transferred during the corresponding I/O operation (read or write).

For buffer validation (i.e. check if the buffer pointer is valid), we copy the buffer from its user space address to the kernel space using *copyin* and make use of the *copycheck* function to validate the buffer (i.e. a buffer that is copied successfully into the kernel is a valid pointer; else it would fail). We then just work with the kernel copy of the buffer in read/write, and copy the end-state of the buffer back to the original user space address.

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 determines 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 are implemented by adding an extra p_pid field to the process’ struct, which stores the PID of the process. All the PIDs are stored in a process table (procArray: a global array of pointers to process). The indices of the process table are the PIDs assigned to processes.

The process table is initialized when the kernel process is created. At this time, indices 0 and 1 are reserved for the system, and kernel process takes on the PID of 2. The remaining indices of the process table is set to NULL. When a new process is created, a NULL entry in the process table is found and set to point to the new process. When a process is destroyed, its entry in the process table is reset to NULL.

When sys__exit() is called on a process, the process will set its exit code and get directly destroyed if it has no parent (else it will be destroyed by the parent after the parent collects the child's exit code). However, before it's fully destroyed by sys__exit(), it will set its position in the process table to NULL so that the PID can be re-used by another process.

To implement fork(), we created a new process that contains heap allocated copies of the parent process's trap frame and address space, a new PID, and a copy of the parent's file table (which points to the same vnodes as the parent's, with each vnode's refcount doubled). The parent then calls thread_fork to setup a new thread to attach to the new process. The first function to be run by the new thread is called childPrep(); it modifies the trapframe to show return value of 0, increments the epc by 4 bytes, activates the kernel allocated address space, and returns to user mode. To ensure synchronization, interrupts were turned off for operations involving copying parents' information (trapframe, address space, etc) to the new process. However, interrupts only works for single processors. We tried to use a lock instead to ensure multiprocessor synchronization, but that implementation was negatively affecting our other system calls and so we continued to use interrupts.

Getpid() was implemented to always return the PID of the current running process.

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?

The waiting required by waitpid is done by using a binary semaphore, declared within each processes. when a process A enters waitpid() it asks if the process B it is waiting on has finished running. If not, A will be blocked at a P(sem) = 0 statement. On the other hand, when process B finishes running and enters sys__exit(), it sets its exit code and encounters a V(sem) statement which will allow process A to enter the semaphore and obtain B's exit code. As well, process B will be cleaned up by its parent (A) in waitpid() instead of getting cleaned up inside the sys__exit() function. Here is a figure representing the example:

sys__exit **(process B)**	waitpid **(process A)**
B->exitcode = exitcode V(sem) //1	if B is still running P(sem) //0 get B's exitcode destroy B

We have enforced the following assumption/restrictions:

- a parent is always interested in waiting for its children
- an orphan is not adopted by the kernel process (so it is destroyed directly by sys__exit())
- a process can only wait on its child, and is responsible for destroying its child after the child exits

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.

Since polymorphism is not possible, a runprogram2() method has been created in kern/syscall/runprogram.c to handle programs that takes in 1 argument or more. argc is passed straight from the menu.c kernel thread as it is a long. space to hold argument strings as well as the pointer to them are allocated after the address space is initialized in runprogram2(), a loop runs through the argument strings, and the space is kmalloc'd for it on the userspace stack. After each argument string is stored, an array of char pointers is initialized one at a time to point to the start of the argument string. The char * array is kmalloc'd on the userspace stack after aligning it by 4, and is aligned by 8 after copying to allow for storing of 32-bit data types. A pointer to the start of the char * array is passed to enter_new_process as argv and argc is passed as an int as well.

The arguments passed in by the user program is first copied one by one to the kernel's heap using kmalloc, after doing this, we can call runprogram2() to start execution of the new process, we need to copy it to kernel heap as runprogram2() assumes that its parameters are stored in kernel memory. After copying it to kernel heap, we call runprogram2() to create a new address space for the process and it is then copied from the kernel heap to the userspace stack.

The implementation of execv is based off of runprogram2(). In fact, after copying the program

name and the arguments to the kernel, sys_execv actually calls runprogram2() to start execution of the program. In order to prepare the program before running runprogram2(), we create a new address space for the program to live in as it is a new process, as well as copying the program name and arguments passed in to the kernel. Since runprogram2() is called by the kernel only, we have to make sure that everything is copied to the kernel heap first before calling runprogram2().

Note that the implementation of execv is not complete - some badcall tests are failing because we didn't have enough time to fix them all. We couldn't figure out how to detect some of the errors that execv may encounter.

Exception Handling

Exception handling is implemented by calling sys_exit() with an exit code of -1 in kill_curthread(). sys_exit() is placed right before the panic("I don't know what to…") statement, so that if there is a problem with sys_exit() (which means we will hit the panic statement) then we really should call the panic. However, currently the exit code is always -1, we could elaborate on this and use more descriptive exit codes to represent the exception encountered.