Question1 :

File descriptors is consisted of a vnode, which is an abstract representation of the file. An offset, which is the current position of the file, has modes (O_RDONLY, O_WRONLY, O_RDWR) which represent read-only, write-only, and read-write. Lastly, it is consisted of a lock, which protects the file from being accessed by multiple threads at once.

The kernel data structure that is used in our implementation is a file descriptor manager. Its purpose is to manage the file descriptors in kernel mode. This data structure contains an array of file descriptors that it is managing currently. Our file manager has a function to deal with the different default standard files (STDIN, STDOUT, STDERR). The manager has functions to add/remove/return file descriptors and other various functions to modify its contents.

OPEN is used to open files. vfs_open is called in the beginning to check if the file is openable. If the vfs_open call was not successful, we return the appropriate error. Otherwise, a file descriptor is created with the file name, file mode, the vnode, and a lock. Then it is added to the file manager.

CLOSE is used to close file. When this is called, we access its file descriptor in the file manager, and remove it. If this is not successful, appropriate errors are returned.

READ is used to read a file. A file descriptor is fetched from the file manager of the current thread. We make sure that the file descriptor's file mode is set to an appropriate flag to be read. Otherwise, an error will be returned. We create a uio structure to pass data from the kernel space to the user space containing information such as: number of bytes to be read, file description, etc. Since all of the files are managed by the kernel, the file to be read must be passed onto the user space in order for the file to be read. A lock is acquired during the read, such that the file is only being accessed by a single thread at a time. After the file is finished being read, the lock is released so that other threads can access the file. During the read call, the uio is passed onto VOP_READ so that uio_resid is updated to match the number of bytes read. Finally we return the (number of bytes to be read – uio_resid). This value is the actual bytes that have been read.

WRITE is very similiar to read. The difference between WRITE and READ is that we must make sure the file descriptor is not "read-only" and the uio settings are correctly applied. We still utilize a lock to enforce mutual exclusion and use VOP_WRITE instead of VOP_READ. We update the offset of the file descriptor accordingly and return the number of bytes written.

Question 2:

Our PIDs are not fully implemented. However, the plan is to create a PID manager similiar to the file manager. Everytime a thread is created, the process ID is being tracked by our PID manager. The PID manager is initialized in the bootstrap, and will continiusly be adding unique PIDs when threads are created. PIDs have two members. The first one is an array which contains every PID currently being used, and the data queue contains the next availalbe PID. If the queue is empty, then there is no PID available. It has functions similiar to file manager (ex. adding processes to the manager with a unique ID, removing them, destroying itself, etc).

When the bootstrap function is called, the maximum number of PIDs used is 100. We add the PID 0-99 to our queue, and each time a process is created, it will look at the queue and check if there is a PID available.

When a thread is finished its task, it will call "thread_exit" which will attempt to destroy itself. In doing so, it will call "pidremove" which will add its PID to the end of the queue. It will remove its PID from the PID manager's array.

We did not complete fork, but the plan is to create a new child thread with the same address space of the parent. The child thread also receives the parent's trap frame so memory on the heap needs to be allocated. Next we thread fork to create the child thread. Then we decrement the sem count for child thread's semaphore. We then return the child's PID.

GetPID returns the current thread's PID member.

We did not complete system exit, but the plan is to get the current thread's PID and will get the process from the PID manager. It will then acquire a lock and change the status of that process to represent it has exited with its corresponding exit code. It will then wake up all the threads sleeping on the exit lock. Then it can gracefully exit.

Question 3:

We did not implement the waiting done by the waitpid. But we did have a plan set out. We would first try to get the process that contains the given pid. We would then acquire the process' lock and wait for the process to exit by using a conditional variable for the particular process. Once waiting is finished, the appropriate exit code is set and the lock is released. Finally the process id is returned.

We would have used locks and conditional variables to protect the thread from unwanted behaviors. The locks are to make sure no multiple processes exit at the same time, and the CVs are for allowing the process waiting to "wait" for a "signal" from the process that is finishing up.

We use a lock so that only one thread can exit the process. We also would have used a conditional variable so that the thread can wait for a signal from the process to exit.

We would have implemented logic so that a thread cannot wait on another thread that

is waiting.

Question 4:

We did not implement execv, but the plan is to first determine the number of arguments, save our own copy of the arguments and ensure the arguments follow the proper arguments format. We will have to ensure that the PID of the process has to be perserved along with the address space. We will then load the ELF and then close the program now that we are done with it. We will define the user stack and put the arguments that we saved onto the stack. Then we shall warp to user mode.

For runprogram, it is very similiar to execv. To move all the arguments to the user stack is the difficult part. We did this by first calculating the size of the stack frame for the main function. Since the stack pointer should always start at an address that is 8-byte aligned, we must make sure it is evenly divisible by 8. After we will copy the arguments onto the stack, and then loop through the arguments and use Copyout() to copy the block of memory from the kernel space to the user space. But first we must make sure that we copyout() the first argument of args since we want the name of the test as the first item. Then we warp to user mode. Since we know argc, it is easy to loop through all the arguments in argv when doing the Copyout() calls. That is how we utilize ther Argc and Argv paramters.