New Files Added:

userprog/filetablefunction.c
userprog/userfile.c
userprog/userfile.h

Question 1

Filedescriptor is a structure that contains a pointer to a vnode, the flags and offset. As it can be seen in filetablefunction.c, we define a createfiletable() function that creates and returns an array. Weee use the kernel's definition and implementation of array throughout our code. The way file descriptors are implemented is straight forward: we maintain an array of FD's that indicates the open / "in use" files. When a file is closed or done, that index in the array is set to NULL (and is available for any other FD that may be "looking for a spot" in the array). The system call "open" adds a FD element to the array (first available index, or if not – then increase size of array and add it as last element – handled by array_setguy / array_add). Other than adding the entry to the array of file descriptions, the vfs_open function is called which actually opens the file. Although some of our code is non-functional, the login was discussed and somewhat implemented in the files: **filetablefunction.c, userfile.c**. **Please note that most of the code is commented out right now as we were having compiling issues (unable to fix due to lack of time). Hence, the open/close implementation is incomplete and is possibly missing some aspect which we are still debugging.** Some minor details are lacking which prevent the program from working; however, the login in theory should work. For read and write, a FD is given to the function getIndex which returns an index (corresponding to the file in the file table) if the FD is found successfully. If not, an error is returned (trying to open a non-existent file). **Again, the implementation for read and write is not complete and does not function properly (to pass test cases)**. However, the logic and the design was discussed and should work in theory. Although code was written for both read and write, it was not tested due to its non-functionality. The write implementation relies mostly on using putch (for now) which is more or less the "dirty implementation" of write. It passes the testbin/palin test using this.

Question 2

We came up with a basic design for fork but due to lack of time, we could not cover the details and make the program run. The implementation is incomplete and commented out (in userfile.c under the userprog directory) to ensure the kernel compiles okay. The code we wrote is not complete and possibly lacks some details which we are still debugging. So our basic design consists of declaring pointer to two threads called parent and child. Also declare an integer variable called copyResult so we can check whether the process has been copied correctly from parent to child. We also declare a stackptr to keep track of stack memory. We set parent to point to the curthread. We then create a child thread using thread_create. If this child create fails, then

we return -1. We then give random values to the child user stack. Copy the address space of parent thread to the child thread. If this fails, we return -1. We set the stackptr to point to the child's address space. This is what our design consists off however is incomplete. For _exit (sys__exit), the focus is to clean up the process (that called _exit) and release thread control. The main call we made in _exit was thread_exit().

No code at all was written for getpid / waitpid but here was the idea we discussed briefly but did not get time to implement / discuss further. This plan was part of fork too but we never got to it. We wanted to create a new struct relating to the process a thread is part of and add it to the struct thread. So each thread structure would contain this new structure (lets call it struct proc) which would store its process related information. This would include the PID (process ID) of the thread, the PID of its parent (if exists) , and other information (which we did not get to decide/discuss) relevant to the process that a thread is part of. This would make the implementation of getpid quite straightforward, as all we would have to do is return current thread's PID (so something like: return curthread->proc.PID).

Question 3

The implementation of waitpid is something we only barely discussed and **did not code at all**. The general idea was that the current process (that calls waitpid) "waits" (stops execution) until a child has finished execution and exited. We were thinking of placing the restriction of the calling process to be only allowed to wait on its child processes. So the pid passed in must be a child of the calling process. We based these design decision and schema based mostly on the UNIX implementation of waitpid which we looked at and analyzed a bit. As mentioned in the man page of waitpid, waitpid is supposed to return immediately if the process pid (argument 1) has exited already and waitpid fails if the pid (arg 1) does not exist. The status of pid is determined through its status code (this was our plan at least). To distinguish between "exited already" vs. "does not exist", we planned the criteria to be: if the process pid has called _exit(), it has a specific status code (set in _exit()) which we can check against.

Question 4

For runprogram, the argc and argv have to be loaded into the user stack before transferring control to the user. To do this, first we copy all the actual strings (argv[0], argv[1], etc) into the user stack, and then we create new pointers in the stack using copyout. Initially, for each argument, we decrement the stackpointer by the string length and then copy the argument string (using copyoutstr). As we do this, we keep decrementing the stackptr (stack pointer) appropriately. Then, once we have all the argument strings stored, our approach was to move stackptr to an address divisible by 4. The approach now is to allocate n chunks of memory, each

4 bytes and n = (number of arguments + 1) . We do “+1” part because the last chunk stores null. So after storing null, we recreate pointers that point to the right data (strings) in memory. At this point, we have the argument array correctly stored in the user stack, and we just need to pass the starting address of where the pointers start (argv[0] pointer first, then argv[1], etc). One last step is to ensure that we move stackptr to an address divisible by 8 to adhere to given constraints and then call md_usermode. We were able to implement runprogram completely and make it run (tested using testbin/add a1 a2). Here is a diagram of what the argument structure looks in the user stack: