Q1: File Descriptors

There is a file table for every thread (process), and it stores an array of files descriptors. When a thread create, one new file table is created. When a thread exits, the file table is destroyed. And when during fork, the file table is duplicated.

file table - I used array structure to store an array of file descriptors. For file descriptors, I have vnode to record the file, off_set to check the offset for read/write, and two locks (for atomic read/write).

open - I used vfs_open to open the file, and added the new file descriptor to the file table. Similarly, close used vfs_close and removes the file descriptor (set the element to NULL) from the file table.

write - I configured uio with related file descriptor and use VOP_WRITE to write, then update the file descriptor. Similarly, read has the same process except using VOP_READ.

Q2: Process Identifiers

PID was saved in a global variable "pt" which is a type of proc_table structure. The proc_table structure keeps an array of process information structures. PIDs get generated every time sys_fork function is called. Inside the sys_fork function, a child thread gets created with thread_create function. For every child thread, a new process ID gets assigned to that thread using proc_table_add which is supposed to add the new process into the process table as well as return the PID that was just added. When a sys_exit gets invoked, a process gets flagged as "inactive". So when another process gets added in the process table, the table iterates itself to see which PID is inactive and replaces that PID with the new process information structure.

fork - a new thread is created by using thread_create(), then new PID is created after calling thread_create().memcopy is used to copy the stack of the current thread to the new thread, and file_table_duplicate()clones the file table to the new thread. Now a clone of the current thread has been made. Then a new process gets created where the address space from the current thread' process is copied over. The new thread gets added to the new process. Also, a new trapframe is made and memcpy is used to duplicate the current thread's trapframe into the new trapframe for the enter_fork_process. Finally, new thread is switched to user mode and gets activated. PID should be the return value.

getpid - returns the current process's PID. The current PID can be found in the thread structure because "pid" field has been added. Current PID gets updated in sys_fork function when a new process gets assigned for a new child thread.

_exit - changes the current process's state by accessing the corresponding PID's information structure in the process table and sets it to an inactive mode (since it exited) and updates exit status field. thread_exit is called at the end.

Q3: Waiting for Processes

Waitpid takes in 3 parameters: pid, status, options and retval. Error checks are done in the beginning of the function: making sure option argument is 0, pid argument is an existing process, status is a valid pointer, and pid references a process that has exited. If an error occurs, the error code gets saved in retval which is a type of integer pointer. Also, the function returns -1. If no error occurred, then we use the lock, "plock", and cv lock, "pcv", to update the status with pid's exitcode which was assigned in sys_exit function. We need to make sure the process we are trying to access is not currently active or in use. If so, we acquire plock and use cv_wait to wait

until the process exits (becomes inactive). We will know when a process becomes inactive when sys_exit uses cv_signal after updating the process's exit code and flags itself as inactive. Once we know the process exited, we update the status parameter using process's exit code and release the lock.

Q4: Argument Passing

For runprogram, argc and argv are placed on the kernel stack at beginning, but we need use them in user level. Thus, they have to be copied out to the user level and they must be properly aligned. Firstly, we should make an array of address, and the size of this array should be argc +1. runprogram can get the length of every argv and store a properly aligned address to the address array based on the length. Then we need to copy out the arguments by using the value in the address array. Overall, we can run the enter_new_process.

We can use a table (see the below table) to explain when there are 2 arguments "foo" and "1". We can know the end pointer address and there are 4 bytes aligned, so we can find the each address of the pointer for every argument. Then we can copy out.

For execv, the argc and argv need to be duplicated from the user level to the kernel stack, and then we can use rungropram to copy out to user level.

end	\0
	\0
	\0
address	\0
	\0
	\0
	\0
address	\1
	\0
	o
	o
address	f
address	argv[3]

address	arg[2]
begin	arg[1]

1	1
	\0
2	F
	O
	o
	\0