Phase 2 Write Up

1. In the Pintos struct PCB, we added an array of file pointers mapped to file descriptor ints and an array of a process’s children PCB’s. The first array allows us to set new file descriptors, edit a file through its file descriptors, and unlink the file descriptor with its file pointer. The second array allows us to access children processes and get any information fielded in struct PCB. The PCB described in Chapter 3 contains many more fields dealing with control and information. For example, the text PCB includes the parent process\_id, which would allow for a child process to access a parent’s information. The state information is currently very limited in Pintos. Although there are several process states, the PCB can only describe “dead” or “not dead” states. There are also no fields describing a process’s privileges in the Pintos PCB. As is, all processes have unlimited privileges.
2. Every time sys\_exit is called on a process, the exit code of its parent is set to the exit code of the process before it exits. In our implementation, we account for this behavior in syscall.c sys\_exit. Before thread\_exit() is called, we access the the current thread’s PCB and change the exit code to the input exit code (called status in the prompt and prototype definition). When thread\_exit() runs, the parent will correctly grab the right exit code from its child process because we changed it before killing the process.
3. It’s important that inputs be checked because system calls have a lot of power and can quickly mess up various functions of the OS if user input is faulty or the code is not correct. sys\_exec checks that the input user path is a valid, mapped user address with verify\_user. sys\_exit changes the exit\_code to PID\_ERROR, which is the process id - 1, if the input exit\_code is anything less than 0. sys\_wait checks to make sure that the input child process id is greater than 0 because process id’s cannot be less than zero by the nature of our next\_open\_child function and it cannot be zero because that is the first process. The worst case scenario for sys\_wait without input checks is it could wait on a nonexistent child process to finish forever.
4. Big-O analysis (note that n represents the number of processes):
   1. **sys\_exec**: the functions copy\_in\_string and copy\_in\_argv do not depend on the number of processes, so these will be O(1). Within process\_execute, the only non-constant function call is the call to next\_open\_child. In the worst case, this function will iterate through every possible child process, resulting in O(n) runtime. Therefore, the entire sys\_exec call will run in O(1) + O(n) = **O(n)**.
   2. **sys\_exit**: this system call calls thread\_exit, which in turn calls process\_exit. process\_exit iterates through each of the process’ children, which takes O(n) time. thread\_exit then calls syscall\_exit, but this system call runs in O(1) relative to the number of processes. The entire sys\_exit call will also run in **O(n)**.
   3. **sys\_wait**: this system call calls process\_wait. process\_wait iterates over each child process to look for a particular child, so this will take O(n) time. This is the only non-constant operation, so overall, the system call will also run in **O(n)**.