Gueho Choi
University of Southern California
CSCI 350: Operating Systems
Pintos Project 2 – User Programs
October 29, 2014

Design Document

I. Which method you used for validating user memory access?

- Whenever stack pointer is accessed through system calls, I performed validation on the stack. Also, whenever pointer is passed, it is validated. That is, I adopted the first method described by pintos document.
    a. How did you implement the above method?
    - Looking at the vaddr.h and pagedir.c as suggested in pintos document, I used 'is_user_vaddr' from vaddr.h and 'pagedir_get_page' from pagedir.c. I noted that 'lookup_page' could be used for the purpose, but I wanted it to have extra check by pagedir_get_page.
      Note that I validate every parameters and pointers coming out of user stack, providing the system bullet proof protection.
      This is a bit more verification than what is required, I suppose. However, it is worthwhile that when I protect everything from user stack, then the OS will become really safe from bad pointers.

II. Where and how did you parse arguments and put into stack?

- I parsed argument in start_process method from process.c. I put it into stack using a stack frame. The stack frame will be pointing at the pointer of first element. It will increment itself to push the addresses of arguments along with arguments pushing above the word_align. After pushing the arguments, I push pointer to pointer of first argument at (stack frame -4), argc at (stack frame – 8) and fake-stack pointer of 0 at (stack frame – 12).

III. Process_wait/wait syscall

- Wait system call calls process_wait. All implementations are inside process_wait. Thread carries list of 'struct child'. And the struct child will contain data such as tid, exist status, synchronizations, and some flags to identify behaviors. By going through the list, the parent checks if it waited for the child before, and checks if the child is direct descendent of the parent process. Such validations are done by flags, owned by struct child.
  To make the better synchronization, I used lock and conditional variables instead of semaphores. Therefore, conditional variable wait function is the core function that waits for the child to be finished. Instead of using semaphore, which is stateful, I used stateless conditional variable to make sure parent process can wait for right child.

IV. Exec syscall

- Exec basically instantiates and initializes the 'child' struct and the spawned child is put in the parent's children list. Also, to make sure that the exec returns when child is completely finished with its spawning, I used semaphore to wait for the child to finish. Semaphore was more appropriate to use because the child may potentially finish before parent actually use synchronization. That is, if lock is used, lock may be released before lock is acquired by parent. Semaphore is stateful, therefore, it is more appropriate to let the parent proceed if child

finished earlier than parent.
Most of executing functions are implemented in process_start function rather than exec itself. In proc_start, I opened the executable file before it loads the executable, and let the thread to hold onto it in 'executed_file,' and then used file_deny_write to prevent write on the executable file. Also, when any error happens, I made sure that semaphore is released before the child exist in order to make sure that waiting parent is awakened.

V. Describe data structures you keep for a process

- Struc thread:
    - Struct list children: hold the list of child process meta data
    - Struct thread *parent_thread: enable the child to be able to climb up to parent in order to release synchronization variables
    - Exit_status: this will be set whenever the process exits, also, it enables exception to printout status code
    - Struct file *executed_file: keep track of file executed
    - Struct file *fdtable[128]: table of fd, size as suggested in pintos documentation
- Struct child_proc
    - Exit_status: this will accessed set during normal exit
    - isFinished: lets the conditional variable to identify if it's time to wake or not
    - waitLock, waitCV: synchronization variables used to process_wait
    - canWait: whether or not the parent process can wait for this child, false if parent once waited for this child

VI. How did you handle file descriptors?

- Each threads hold the table of file descriptors, called fdtable. Fdtable has size of 128. Whenever file descriptors are accessed through system call, the fd parameter in the stack is used as index to the fdtable. The fdtable then bring out the actual struct file from the file descriptor. For example, when close is called, fd is extracted from the stack. Then fd is validated so that user does not close stdin, stdout, or cause some weird exception. Now, fd is ready to be used as index to the fdtable which outputs struct file. The struct file will eventually be used for parameter to the file_close call. After close, the fdtable[fd] is set to NULL so that it can accept more file descriptors. This is kind of 'hash table'. This is better than some linked list because it is extremely faster than linked list.