---- TEAM ----

**>> Team name.**

57

**>> Fill in the names, email addresses and contributions of your team members.**

Kyumin Park <pkm9403@kaist.ac.kr> (50)

Keon Lee <rjsdlwwkd@kaist.ac.kr> (50)

contribution1 + contribution2 = 100

**>> Specify how many tokens your team will use.**

0

---- PRELIMINARIES ----

**>> If you have any preliminary comments on your submission, notes for the TAs, or extra credit, please give them here.**

**>> Please cite any offline or online sources you consulted while preparing your submission, other than the Pintos documentation, course text, lecture notes, and course staff.**

ARGUMENT PASSING

===========

---- DATA STRUCTURES ----

**>> A1: Copy here the declaration of each new or changed `struct' or `struct' member, global or static variable, `typedef', or enumeration. Identify the purpose of each in 25 words or less.**

No data structure added for argument passing

---- ALGORITHMS ----

**>> A2: Briefly describe how you implemented argument parsing. How do you arrange for the elements of argv[] to be in the right order? How do you avoid overflowing the stack page?**

We simply stacked arguments up one-by-one, according to 80x86 convention. First, we stacked each tokenized argument up as it tokened. Simultaneously we stored address of each stacked argument in newly allocated memory by page allocator.

After stacking up all the arguments, we made a word alignment by calculating grown stack’s reminder by 4, then subtract esp using remainder, to become address multiple of 4. We didn’t care of the value of offset memory area, since program will never access that area.

As a third step we added address of each argv[i], which saved in allocated page. Here we stacked up in descending order of argv[i], following 80x86 convention. Finally we added argv, argc to stack, then returned return address.

---- RATIONALE ----

**>> A3: Why does Pintos implement strtok\_r() but not strtok()?**

Strtok\_r uses three parameters – character to tokenize, delimiter, and save address for further tokenize. In contrast, strtok() declares save pointer as a global variable. Using global save pointer could result in conflict such that several process call strtok() concurrently. Since global save pointer would be dangerous in such case, Pintos implement strtok\_r(), which declares save pointer in local area, instead of strtok().

**>> A4: In Pintos, the kernel separates commands into a executable name and arguments. In Unix-like systems, the shell does this separation. Identify at least two advantages of the Unix approach.**

One possible advantage of Unix-like system is efficiency. Kernel has lots of work, such as handling interrupt and context changing. If parsing command is held by kernel, kernel gets more work, which means it needs more running time. With Unix-like system, however, kernel can save more time without hesitation of parsing command, so that work can be done more efficiently.

Another possible advantage is stability. If critical error occurred in parsing period, parsing program may need to be restarted. With allocating parsing procedure to shell, system can just restart shell program if such critical problem occurred. However, if kernel got such problem and restarts, lots of other running programs will be affected, may resulting in another severe problem in system.

SYSTEM CALLS

===================

---- DATA STRUCTURES ----

**>> B1: Copy here the declaration of each new or changed `struct' or `struct' member, global or static variable, `typedef', or enumeration. Identify the purpose of each in 25 words or less.**

struct process: store file descriptor list, parent, and exit status.

struct fd\_data: map struct file and fd

In struct thread:

struct process\* process: Direct process structure owned by current thread

typedef int pid\_t: identify process

static list proc\_list: list of all processes

static lock filesys\_lock: lock for mutual exclusive file system approach

static lock proclist\_lock: lock for mutual exclusive approach to process list

**>> B2: Describe how file descriptors are associated with open files. Are file descriptors unique within the entire OS or just within a single process?**

Struct fd\_data is allocated to each file open so that user can easily reach file structure with file descriptor number. File descriptor is just within a single process, since file usage of each process is independent

---- ALGORITHMS ----

**>> B3: Describe your code for reading and writing user data from the kernel.**

In both system calls, check fd number first and determine next procedure. If fd is STDIN\_FILENO (== 0), only read system call works, reading data from keyboard input approached by input\_getc(). In case of STDOUT\_FILENO (== 1), likewise, only write call works, writing data in buffer into console using pufbuf() function. In neither situation (fd > 1), process get file structure from fd and work on the file, depending on system call.

Read: kernel reads file data, then store into temporarily allocated space. Then kernel moves data from temporary space to buffer, checking each data is stored appropriately.

Write: kernel first check validity of entire data buffer, then write data directly into file.

**>> B4: Suppose a system call causes a full page (4,096 bytes) of data to be copied from user space into the kernel. What is the least and the greatest possible number of inspections of the page table (e.g. calls to pagedir\_get\_page()) that might result? What about for a system call that only copies 2 bytes of data? Is there room for improvement in these numbers, and how much?**

**>> B5: Briefly describe your implementation of the "wait" system call and how it interacts with process termination.**

Wait system call runs following procedure: verify validity, wait child, obtain child status and free resource. Validity includes double call, child-parent relationship, and child existence. Wait is implemented by semaphore in child process. All interaction include waiting semaphore is done by modifying and reading process structure of child process.

**>> B6: Any access to user program memory at a user-specified address can fail due to a bad pointer value. Such accesses must cause the process to be terminated. System calls are fraught with such accesses, e.g. a "write" system call requires reading the system call number from the user stack, then each of the call's three arguments, then an arbitrary amount of user memory, and any of these can fail at any point. This poses a design and error-handling problem: how do you best avoid obscuring the primary function of code in a morass of error-handling? Furthermore, when an error is detected, how do you ensure that all temporarily allocated resources (locks, buffers, etc.) are freed? In a few paragraphs, describe the strategy or strategies you adopted for managing these issues. Give an example.**

For user memory access, we used get\_user and put\_user function to every single address so that any access to unmapped user memory fall into page fault. In page fault handler, we verified error occurrence source, whether kernel (by system call) or user. All errors caused by user terminated corresponding process by exception handler. If page fault caused by kernel, handler simply returns -1 so that kernel can detect error occurrence and handle appropriately.

Ex) In user: int a = \*NULL; => Terminates process

In kernel: int a = \*NULL; => returns -1, kernel handle result as an error occurrence.

---- SYNCHRONIZATION ----

**>> B7: The "exec" system call returns -1 if loading the new executable fails, so it cannot return before the new executable has completed loading. How does your code ensure this? How is the load success/failure status passed back to the thread that calls "exec"?**

We added semaphore into process structure. After creating child thread, parent create struct process of child, unless thread\_create() failed. In process creation step, parent initializes semaphore in child process to 0 and down it so that it can wait for its child is done loading. After child is done loading, child first waits for its process created using busy waiting method. Once child find its process, it records load success information in its process structure, then up semaphore so that parent wakes up. By this way, parent can wait child’s loading, and child can wait parent’s process structure creation.

**>> B8: Consider parent process P with child process C. How do you ensure proper synchronization and avoid race conditions when P calls wait(C) before C exits? After C exits? How do you ensure that all resources are freed in each case? How about when P terminates without waiting, before C exits? After C exits? Are there any special cases?**

We used same semaphore in B7 to implement wait. When wait() called, parent verifies child such as parent-child relationship, then it downs semaphore so that parent waits child until child exits. When child calls exit(), child store its exit status into process structure and up semaphore. In this way, parent will wait child until child finishes its work, and also parent will not wait child when wait called after child exits, since semaphore value is already been up to 1 once child exits.

---- RATIONALE ----

**>> B9: Why did you choose to implement access to user memory from the kernel in the way that you did?**

**>> B10: What advantages or disadvantages can you see to your design for file descriptors?**

**>> B11: The default tid\_t to pid\_t mapping is the identity mapping. If you changed it, what advantages are there to your approach?**

SURVEY QUESTIONS

================

Answering these questions is optional, but it will help us improve the course in future quarters. Feel free to tell us anything you want--these questions are just to spur your thoughts. You may also choose to respond anonymously in the course evaluations at the end of the quarter.

**>> In your opinion, was this assignment or any one of the two problems in it, too easy or too hard? Did it take too long or too little time?**

**>> Did you find that working on a particular part of the assignment gave you greater insight into some aspect of OS design?**

**>> Is there some particular fact or hint we should give students in future quarters to help them solve the problems? Conversely, did you find any of our guidance to be misleading?**

**>> Do you have any suggestions for the TAs to more effectively assist students, either for future quarters or the remaining projects?**

**>> Any other comments?**