CSE 421/521 PROJECT 2

---- GROUP ----

>> Fill in the names and email addresses of your group members.

SHUBHAM PANDEY <spandey4@buffalo.edu>

SHASHANK RANGHUNATH <sraghuna@buffalo.edu>

ANKESH N BHOI <ankeshni@buffalo.edu>

---- 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.

Sources referred -

<https://en.wikipedia.org/wiki/File_descriptor>

<https://www.bottomupcs.com/file_descriptors.xhtml>

<http://ytliu.info/notes/linux/file_ops_in_kernel.html>

<https://stackoverflow.com/questions/29945171/difference-between-page-table-and-page-directory>

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.

* **char\* argv[]** inside struct thread for storing list of arguments obtained from command line and copy them onto the stack

---- 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 have a reference to command (filename + arguments) in process\_execute() method. We split the command into filename and arguments using strtok\_r() method and store it in the calling thread’s char\* argv[] struct member.
* At the time of pushing these arguments to the stack in setup\_stack() method, we loop over the argv[] array in reverse order and add to stack. Eg: echo x on command line will store arg[0] as ‘echo’ and arg[1] as ‘x’. These will be copied to the stack in reverse order though. This logic works as the stack grows downwards, so we want the first argument on a lower address.
* Before copying the arguments to stack, we decrement the \*esp pointer so that, we have enough space for the arguments to fit on the stack and not touch the kernel memory.
* Each time we calculate the length of argument and decrement the stack pointer by the nearest multiple of 4. Eg: If argument length is 13 bytes, we decrement stack pointer by 16 bytes and pad the remaining 3 bytes with 0.

---- RATIONALE ----

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

* strtok\_r() has an extra argument called \*save\_ptr which provides a reference to the next character after first delimiter. The extra pointer is absent in strtok().The reference to next pointer is helpful in separating individual tokens in a loop.

>> A4: In Pintos, the kernel separates commands into an executable name

>> and arguments. In Unix-like systems, the shell does this

>> separation. Identify at least two advantages of the Unix approach.

* Kernel stays busy in Pintos approach by first tokenizing the command then adding it to the stack to execute, Unix approach avoids this.
* Filenames are limited to 14 characters in Pintos which will also limit the number of arguments, effectively supporting fewer configurations than in Unix.

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 file\_desc** {

**int file\_num;**

//Unique file descriptor number

**struct file \*file;**

//Reference to the file, file descriptor corresponds to

**struct list\_elem file\_desc\_elem;**

// List element for file descriptor list stored in struct thread

}

struct thread{

**tid\_t parentid**;

//tid of parent process

**struct list child\_id\_list**;

//List of child threads identifiers created

**struct semaphore wait\_sema**;

//Semaphore to block parent if child exiting

**int status\_val**;

//Thread exit status value

**bool wait\_done\_already**;

//Avoiding parent wait if this boolean is set

**struct list file\_desc\_list**;

//List of file descriptors with the current thread

**int file\_desc\_count**;

//Count of files associated with the thread, used to allocate new

//file\_desc\_num in case new file\_desc is added to the thread.

**struct semaphore exec\_sema**;

//Semaphore to block parent if child is still loading

**int load\_failed**;

//Decides success of child load

**struct file \*read\_only\_file;**

//Pointer to the file which should have deny\_write

}

We needed to store a list of child tids for a parent process. But, tid\_t alone couldn’t be assigned list\_elem crucial for maintaining this list. So, we introduced a new struct.

**struct child\_identifier**{

**struct list\_elem child\_elem**;

//List elem of child tid for the child\_ids list in struct thread

**tid\_t tid**;

//Child process’s id

**int status\_val;**

//Thread exit status value\*/

**bool exit\_done;**

//Boolean to check if child has exited already

**struct semaphore try\_sema;**

//Avoid calling wait while deny write is true. Used for sync read

}

>> B2: Describe how file descriptors are associated with open files.

>> Are file descriptors unique within the entire OS or just within a

>> single process?

* File descriptors are like a unique ID for each **open** file associated with a thread to keep track of them for read, write, etc. A pointer to the actual file is stored in the file descriptor struct. File descriptors are unique within a single process. We update the file\_desc\_count each time new file is opened by a thread.

---- ALGORITHMS ----

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

>> kernel.

* We get the file desc number, char\* buffer and filesize to read/write from the next consecutive elements in the stack after stack pointer.

We implemented a method called find\_file(), which will return the file pointer and accepts int file descriptor(fd) as the argument. If the file corresponding to the fd is open in the current thread, we return that file else, we return file.

For read we first check that if file descriptor number is 0, then that is character read from the keyboard, else we simply call file\_read() on the file obtained from find\_file().

For write we first check that if file descriptor number is 0, then that represents console write and we call putbuf() function in console.c for the buffer obtained from the stack. If this is not the case, then we simply call file\_write() on the file obtained from find\_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?

* Best case scenario is if we have entire 4kB continuous memory addresses and directly copy the data using memcopy in strings.c. So only 1 inspection. Worst case is that we don’t even have 2 byte addresses continuous. So, all addresses have to be inspected, hence 4096 inspections.
* For 2 bytes of data if continuous we need 1 inspection. If not continuous, then 1 check for each address so 2.

>> B5: Briefly describe your implementation of the "wait" system call

>> and how it interacts with process termination.

* wait(pid\_t pid) waits for the child process to exit.

We maintain a semaphore to be shared between a parent and child process called wait\_sema and initialize it with 0. We call sema\_down() inside parent process’s wait(). So, it goes to THREAD\_BLOCKED state. The child thread then does sema\_up() upon exiting and then terminates itself using thread\_exit() and returns the exit status to the parent. Parent process eventually gets to the THREAD\_RUNNING state and sets a boolean wait\_done\_already as true. This boolean is necessary so that the parent process doesn’t wait more than once for a child process which has already terminated.

Regardless of the order of wait and exit call times, we have sema\_up() in exit. So, if the child has already terminated before wait is called by the parent process, the parent process will no longer block as the semaphore value would have been 1 already.

>> 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.

* The error handling code is put in each system call that deals with user memory. We check if the stack pointers and the adjacent address correspond to user memory first. We then check whether there is a page allocated to the current\_thread from the active\_pd().Page will be freed in case of any exception.

---- 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 maintain a semaphore between parent and child process. We initialize this semaphore with 0 and parent process calls sema\_down() and so blocks. Child process in the meantime finishes up the loading and sets its load value in its struct.

We have implemented a method called find\_thread\_from\_tid(tid) which returns a thread pointer and accepts a tid as argument. Parent thread has child tid in its child\_ids list. Using the above function for the child tid, parent gets the child thread and looks at its load value and simply returns it.

>> 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?

* If P calls wait(C) before C exits, P will go to blocked state due to sema\_down() with value 0 and come back in the ready list only after child calls sema\_up() in its exit() method.
* If P calls wait(C) after C exits, P will call sema\_down() with sema value 1 and hence, won’t wait.
* To ensure resources are freed, we check if page is still allocated or not. Free the page if it is allocated to that process.

---- RATIONALE ----

>> B9: Why did you choose to implement access to user memory from the

>> kernel in the way that you did?

* Due to the simplicity of this approach and the time-constraint for the project, we believed that using methods in pagedir.c and vaddr.h were the obvious choice rather than exception handling.

>> B10: What advantages or disadvantages can you see to your design

>> for file descriptors?

Advantages -

* We can always see what files are opened by a thread using the list of file descriptors stored in its struct. We get a 1:1 mapping using simple integers. Using just a simple for loop on the list of file descriptors, we can find the file we are concerned with.

Disadvantages –

* Maintaining a file descriptor list in struct thread takes a lot of space and can be slow too if there are many files opened within a single thread.

>> B11: The default tid\_t to pid\_t mapping is the identity mapping.

>> If you changed it, what advantages are there to your approach?

* We have not changed it.

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 three 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?