CS330 Project 2: USER PROGRAMS DESIGN DOCUMENT

---- TEAM ----

>> Team name.

#OS_A+

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

Dauren Baitursyn <biddy.as.diddy@kaist.ac.kr> (50)

Yekaterina Abileva <k.abileva@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.

Wikipedia

http://stackoverflow.com/questions/1706551/parse-string-into-argv-argc

https://www.csee.umbc.edu/~chang/cs313.s02/stack.shtml

http://www.java-samples.com/showtutorial.php?tutorialid=570

CS330 Project 2: USER PROGRAMS DESIGN DOCUMENT

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.

process.c

- struct args { */*Structure for storing the parsed command line and some information about the loading status of the process */*
    - size_t argc - *number of argv (file_name + # of arguments)*
    - char** argv - *list of the arguments together with the filename*
    - size_t length - *length of the command line*
    - bool load_success - *indicates whether the process was loaded successfully*
    - struct semaphore block - *used while starting the process for blocking interrupts*
- static struct args* parse_cmd(const char *fn_copy) - *parsing the command line with strtok_r and assigning the args structure members*
- static struct args* init_args (struct args* args) - *initializing args structure }*

---- 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?

The argument passing starts from process_execute. First, we allocate a page and copy the contents of string into that page to avoid racing in accessing and modifying original string (or command line that was passed to execute the process). We have separate structure for holding the processed arguments. Second, we parse the command line using white space delimiter by the help of PintOS provided function strtok_r. Then we pass parsed arguments to start_process as arguments. In start_process, it retrieves the first argument as a filename, checks if it exists, sets up its stack frame, and executes with rest arguments. Setting up the stack frame goes in double pointer address. The arguments structure passed to setup_stack function has length aligned of bytes to be written to pointer, thus we can write the arguments one by one along with writing their addresses to stack. Then, finally we push argument pointer, the number of arguments, and fake return address which is 0.

---- RATIONALE ----

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

strtok_r() saves the address of the parsed argument, which is needed for setuping the stack, since in the stack structure we are given the arguments should be stored together with the addresses. As oppose to it, strtok() just parses the string without storing the save_ptr.

>> 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.

1) We can avoid kernel error by parsing the command line before sending it to the kernel (if more arguments than allowed, or the executable function does not exist, etc.)
2) In addition, it's more suitable task for the shell than for the kernel, because in this case user programmes can control arguments without calling the kernel, whereas the kernel doesn't have any need to do it.

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.

syscall.c

- static struct fd { */*for storing the file together with its descriptor and list_elem elem to use it in the list */*
    - struct file* file - *pointer to the file*
    - struct list files - *List which stores all the file which are executed when the thread was running*
    - int descriptor - *file descriptor value of the file. Used for finding the file in the list, changes if the file is opened so that when it is opened multiple times another descriptor is returned*
    - struct list_elem elem - *for storing the fd in a list. }*

thread.h

- struct thread { /* Only newly added struct members are here: */
    - struct list children - *List of children processes (threads to to be exact).*
    - struct thread *parent - *Parent of user program*
    - struct file* file - *Pointer to the file which was executed with the thread and is opened now (for denying write) }*
- struct child-meta {
    - tid_t tid - *ID of the thread*
    - int status - *Status in case of set by child thread in case of termination.*
    - bool wait - *Boolean to check if the thread is being waited by some other thread, probably parent*
    - struct thread *child - *Pointer to child process*
    - struct semaphore finished - *Semaphore in case parent needs to wait for child to terminate*
    - struct list_elem elem - *List element that is linked to parent's children 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?

File descriptors are associated with process that operates it with, and are unique to that process only. In other words, each process has its own set of file descriptors it is dealing with.

---- ALGORITHMS ----

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

We check the given arguments for validity - before actually handling the filesys call, we check if they map to the virtual address current process. We do that by following procedures. Given esp of interrupted process's interrupt frame, we check if it is pointing user virtual address and if it is from the process's page directory.

After making sure that esp is pointing to valid address, we obtain the syscall number, and process the arguments accordingly to each syscall. If we are dealing with base types that fit into stack address, we just retrieve it. If the argument is pointer, then we check it just like we checked the esp pointer.

>> 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?

The least possible number of inspections is 1: if during the first inspection gets the page head. The largest number is 4096. In the case of copying 2 bytes the largest number is 2 and the least is also 1.

It can be possibly improved by putting several segments into one page (is the sum of their sizes is still suitable for one page), instead of storing every segment on its own page.

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

First, we check the argument, thread ID, if it is -1 (in case it waited process terminated by kernel). Then, check if given thread ID is child of current thread (we do that by checking the list of children of current thread, and if any has same thread ID), and if it is true, proceed further. Otherwise return error.

Then set the meta information variable wait of children in the list (we store special struct for holding the meta information of child) to true. If the wait, before setting it, was true (which, basically means that some other process called wait on that process), return error. After all this, wait for child process to terminate (wait is done through semaphore).

Upon termination, child process sets the meta information status of itself in parent's children's list to its exit status, and ups the semaphore, so that parent process can proceed with execution. It retrieves the exit status of child, and returns it.

>> 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.

We use ptr_to_int() and get_user() functions to check the validity of the pointers. Consequently, every system calls checks the pointers with these functions before starting allocating resources and executing the calls.

For example, if there is a call to the unmapped region, get_user returns -1 which is signaled to the other functions and the resources are freed.

---- 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 have bool load_success variable in the args structure, which changes its value to false if there was an error in loading the process. Consequently, if it’s false process_execute() returns TID_ERROR which is then also returned by sys_exec() call.

>> 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?

When parent process executes new file (creates new child process), it keeps the meta information of its child information in struct child_meta.

The key structure here is the semaphore in that structure, it is initialized with value 0, which should be incremented by child process, when terminating. So when parent waits for child, it just tries to down the semaphore (if child process didn’t terminate yet, then the semaphore value is still 0, or if it has terminated, the semaphore is 1, in which case parent can down the semaphore).

Case I -> Parent exited before Child:

Parent, when terminated, for each of its child’s meta information, if child pointer is not NULL (it is set to NULL, if child terminated first), sets child threads parent pointer to NULL.

Case II -> Child exited before Parent:

Child, when terminated, checks if its parent pointer is not NULL (if not NULL, then it still exists), and if not NULL, then changes the parents meta information for the child process so that it ups the semaphore, and sets child pointer to NULL in meta information.

We allocate the children meta information when children are created, deallocate them when the parent process terminates (we do not deallocate then when child terminates, because parent might call wait in future, and we need child’s exit status that stored in child meta struct).

---- RATIONALE ----

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

We used get_user() and checking whether the pointer is below the PHYS_BASE for checking the memory as it is described in the reference. We did it this way, because for us it was easier to verify the pointer first then to handle the page fault.

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

Advantages:

- No interleavings between process when allocating fd for the open files.
- Abstraction - one process doesn't know much about other processes open files.

Disadvantages:

- If different processes open same file, for each process fd struct allocated, thus it could be seen as a waste of space (instead, we could just create one struct for open file, and have list 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?

Didn't change 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 two problems in it, too easy or too hard? Did it take too long or too little time?

Some system calls implementation was relatively easy (sys_seek, sys_tell), and they all were pretty similar to each other. The most difficult for us was to setup the stack and understand how the syscalls get arguments from the stack.

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

Yes, we understood the structure of the stack, how the argument passing works, etc.

>> 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?

Carefully read the test cases and look at all the functions which are used there, because it’s a great hint for designing the program.

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

>> Any other comments?