OPERATING SYSTEMS PINTOS PHASE 2

December 17, 2016

Name : Ahmed Eid Abdelmoniem
ID : 10
Email : ahmed.eid.csed18@gmail.com

Name : Shehab Kamal Mohmaed
ID : 33
Email : shehabelsemany@gmail.com

Name : Ahmed Helmi El-wakil
ID : 06
Email : a.helmielwakil@gmail.com

1 ARGUMENT PASSING

1.1 DATA STRUCTURES

1.1.1 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.

In function process_start()

- **char *token, *save_ptr :** Used in the string splitting.
- **char *token_, *esp_ :** Used to copy token and esp for adding the the arguments to the stack without manipulating esp and token.
- **int i :** For the loop.
- **char *addrs[100] :** Used to save the addresses of the arguments in the stack.
- **int num_of_args :** Save number of arguments.
- **char *full_arg :** To save the full cmd line to split it without effecting the real line.
- **int full_length :** Saving the full length of all the arguments for the alignment.

1.2 ALGORITHMS

1.2.1 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 full String is passed to process_execute() which creates a new thread that works on process_start() passing the full argument to it. in process_start() after successfully calling the load function we start to tokenize the full argument on the spaces, reduce the esp pointer by the length of the tokenized string plus one, copy the sting to the top of the stack, save the starting address of the string and loop on the string until the end of all the arguments. Then adding the alignment bytes, adding the null, Then adding the addresses of the arguments from the last address (the addres of the last argument) to the first address (the address of the file name) then adding the address of the file name address, then adding the false return address (as structured in the document).
HOW I ARRANGE THE ELEMENTS. Exactly we add the arguments to the stack from the beginning to the end normally with saving the address of each argument in an array, then by looping on that array in a reversed order and adding the addresses in the reversed order like in a stack (the address of the first argument last, the address of the last argument first).
HOW I AVOID OVERFLOWING THE STACK PAGE. By checking if the length of the full argument is larger than the page size terminate it.

1.3 RATIONALE

1.3.1 A3

Why does Pintos implement strtok_r() but not strtok()?

FROM THE strtok() MAN PAGE : "The strtok_r() function is a reentrant version strtok()" and "The strtok() function uses a static buffer while parsing, so it's not thread safe."
Mainly the difference between strtok() and strtok_r() that strtok() uses static character pointer for parsing the string in different calls while strtok_r() makes the caller provide the pointers. So the strtok_r() can distinguish between different callers through the pointer sent to it while strtok() can't distinguish between the calls from different threads as it uses only one static pointer .
While pintos is a multithreaded system so there is a possibility that different threads could call strtok() concurrently which means that both threads will be editing in the static pointer which will eventually give a corrupted output if not crash.

1.3.2 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.

By making the shell handle this separation it :

- Shortens the time inside kernel.
- it's safer that makes the shell check if the executable is there or not before sending it to the kernel.
- it's also safer where if there any problem in the command that may cause a fail, the shell is the one going to fail which better than having a kernel fail.
- reduces the workload on the kernel.

2 SYSTEM CALLS

2.1 DATA STRUCTURES

2.1.1 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.

- **Modifications in struct Thread :**
  - **struct semaphore_object *semaphore_object :** semaphore object shared between the parent and the child.
  - **struct semaphore thread_ready**Semaphore used to be sure that the semaphore object was intialized.
  - **struct list child_list :** List of child processes created by this thread.
  - **struct list opened_files :** List of opened files which created by this thread.
  - **int terminated_status :** Describe statuss of exit thread.
- **New struct semaphore_object :**
  - **struct semaphore sema :** Semaphore used for communication between a thread and its child so that they are synchronized.
  - **int has_error:** to check if the child loaded successfully or not.
- **New struct thread_child:**
  - **pid_t tid :**thread identifier for the child.
  - **struct semaphore_object semaphore_object** Semaphore object used to communicate between the thread and its child so that they are synchronized.
  - **struct list_elem elem:** To be able to iterate throw the list.
- **New struct file_directory_entry:**
  - **struct file *file:** pointer to the opened file.
  - **fd_t fd;** file descriptor.
  - **struct list_elem elem:** To be able to iterate throw the list.
- **New struct file lock:** Lock to be sure that only one process could access the file in same time.
- **New static SYS_WRAPPER system_calls_array :** Array of system calls.

2.1.2 B2

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

Each opened file has a unique file descriptor from other file and Every time file is opened it takes a new unique file descriptor. Each thread has a list of opened files. Open files list consists of Structs of file_directory_entry which consist of pointer to the opened file and its file descriptor. When we need to get file with FD we search throw the list of the current thread and check if its found or not. When a new file is opened we allocate a new struct for it and pushes it into current thread list of opened files.

In our design we make a file descriptor unique within the entire OS because We don't want to put too much information on the thread struct. So we see that it is more suitable to keep unique file descriptor for entire os and keep a list of opened file with each thread to describe files which opened by this thread.

2.1.3 B3

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

- Read :

  First, check if the buffer and buffer + size are both valid pointers, if not, exit(-1). Acquire the lock (file system lock). After the current thread becomes the lock holder, check if fd is in the two special cases: STDOUT_FILENO and STDIN_FILENO. If it is STDOUT_FILENO, then it is standard output, so release the lock and return -1. If fd is STDIN_FILENO, then retrieve keys from standard input. After that, release the lock and return 0. Otherwise, find the open file according to fd number from the open_files list. Then use file_read in filesys to read the file, get status. Release the lock and return the status.

- Write :

  Similar with read system call, first we need to make sure the given buffer pointer is valid. Acquire the _lock. When the given fd is STDIN_FILENO, then release the lock and return -1. When fd is STDOUT_FILENO, then use putbuf to print the content of buffer to the console. Other than these two cases, find the open file through fd number. Use file_write to write buffer to the file and get the status. Release the lock and return the status.

2.1.4 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?

- in case of continuous 4,096 bytes data:
  - the least amount of inspections is 1 : in case of the required page of data coincident over a kernel page exactly (have the same start and end indexes).
  - the greatest amount of inspections is 2 : where the required page of data lies between two pages of data in kernel page.
- in case of discontinuous 4,096 bytes data:
  - the greatest amount of inspections is 4,096 : where each byte of data is found in a different page in kernel space.
- in case of continuous/discontinuous 2 bytes data:
  - the least amount of inspections is 1 : in case of the required data are found in a single page of kernel space.
  - the greatest amount of inspections is 2 : where each byte of data in user space are found in a different page in kernel space.

2.1.5 B5

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

we validate user program arguments to be sure that it's not in kernel space, then we call process_wait() method which do the following procedures .
First : we get the child of the parent thread by searching child list by id if found else return -1, this child has a common semaphore used to communicate between the parent and it's child.
Secondly : now there are two possible cases one of them is that, this child is already terminated so that the semaphore is already up then the parent made it down again and remove this child from it's child list.the other possible case is that the child hasn't terminated yet so the parent will down the semaphore and enters waiting list till the child terminates and up the common semaphore.

2.1.6 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.

- Firstly, We check the interrupt frame ESP if its valid we then check the system call number if there is an error from any one of both checks we exit by changing the status of current thread then call thread exit method.
- We also check the arguments and the pointer if they are null or out of the user space we directly return false which mean we have an error.
- In case of WRITE and READ system calls, we additionally check that the buffer spans in user page by using validate_user_pointer to make that we have a valid pointer.
- We handle any page fault may happen by call Exit(-1) when any of checking methods return false. Page fault may happen because we have an invalid pointer which lay in kernal mode. When we call Exit(-1) we also waking up parents if the parent is waiting for this process.
- We ensure the all resources are freed by calling EXIT(-1) that call THREAD_EXIT which called PROCESS_EXIT where we release all the resources acquired by the thread.
- **EXAMPLE :** If we received a bad pointer, We firstly check it if it is NULL or greater than PHYS_BASE which mean that it is lay in kernal mode not user space mode (we need to generate an error) . The methods for check pointers will directly return false which mean an error happened which validating the pointer . We also check that the buffer spans in user page to be valid.

2.2 SYNCHRONIZATION

2.2.1 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"?

- **in process_execute () method :**
  the parent thread allocate a size for the child and call thread_create() which creates a child which starts it's life by executing start_process() method then the parent will down the common semaphore shared between parent and it's child entering a waiting list so that it waits it's child till it's child finishes it's loading and wakes the parent up.and when the parent wakes up it returns the status of it's child.
- **in process_start () method :**
  the child enters here and tries to load itself and when it finishes loading it make up to the common semaphore shared with it's parent and set it's loading status.
- there is a struct which contains the common semaphore and loading status of the child that's how the parent gets the child loading status.

2.2.2 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?

- **scenario 1 : p calls wait(c) then c exits**
  - P do sema Down to the sharable semaphore and waits the child.(parent will iterate through all of it's child list and get it's child C , then it found the counter of sharable semaphore is 0 so it update it to -1 and waits in waiting list.when it wakes up again by C it removes the child struct form it's list.)
  - C do sema up when it exits .so P wakes up and dislocate the sharable semaphore.(when process C exits, it tries to close all of opened files and remove it's children structs and then it check if it's parent is still alive then it wakes him up in this case set the semaphore counter from -1 to 0)
- **scenario 2 : c exits then p calls wait(c)**
  - when C exits it does sema up and set the counter of sharable semaphore to be equal 1.
  - when P call wait(c) it tries to do sema down to the sharable semaphore and it founds it's counter = 1 so it returns it to zero again without waiting and dislocate the sharable semaphore object.and return the exit status of C.but if P waits again on C for the second time it will find the child equal null so it return -1.
- when a thread enters thread_exit() it closes all the opened files and removes all sharable objects with it's children threads.

- when P terminates before C exits it removes all it's sharable semaphore_objects so that when C exits it finds that it's pointer to sharable object with parent is null so it doesn't try to wake P.
- when P terminates after C exits: firstly when C exits, it does it work as usual and do sema up for parent.
- we have no special cases in this part.

2.3 RATIONALE

2.3.1 B9

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

Actually, This design is used by most real kernels (including Linux) because it is usually faster because it is nearly work like the processor's MMU which is resebonsable to check valid addresses. Maybe that's easier to implement a simple function that takes an address and checks whether it's valid or not, if it's not valid it will exit with status -1.

2.3.2 B10

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

- Advantages :
  - Each process has an independent set of file descriptors which describe the opened files which opened by this process.
  - The Struct describing an opened file is minimum.
- Disadvantages :
  - The size can be very large if files are kept being opened which will cause a fault to occur when the number of opened files is not fit in integer.
  - Searching for the file using its descriptor requires a lot of time as it searches in the list of opened files in current thread untill we find a file with same file descriptor or return null in case we can't find any matched file. This requires O(n) time to get a file where n is number of opened file from the current thread (size of opened file list) .
  - we can solve this disadvantages by using array or hashtable to get pointer on target file in o(1).

2.3.3 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 didn't change it.
- We see that it may be an advantage to keep tid of the parent into the child process.