+--+
| CIS 520 |
| PROJECT 2: USER PROGRAMS |
| DESIGN DOCUMENT |
+--+

---- GROUP ----

>> Fill in the names of your group members.

Cody Murrell <cmurrell@ksu.edy>
Jazz Loffredo <jloffredo@ksu.edu>
Nathan Sellers <sellersn8463@ksu.edu>

---- PRELIMINARIES ----

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

Could not pass “multi-oom” test. All other tests passed consistently.

>> 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.

- https://static1.squarespace.com/static/5b18aa0955b02c1de94e4412/t/5b85fad2f950b7b16b7a2ed6/1535507195196/Pintos+Guide
    - Pretty much just used for setting up the stack, had good example code
- https://github.com/Waqee/Pintos-Project-2
  https://github.com/ChristianJHughes/pintos-project2
  https://github.com/ucrAlliance/Pintos-Project-2
    - Not much code taken from these, mostly just general structure/what to modify
- https://github.com/MohamedSamirShabaan/Pintos-Project-2
    - Heavily inspired child management and how to exec/wait
- https://bitbucket.org/eardic/pintos-project-2/src/master/
    - Helped us to figure out what was going wrong with “bad-read”, “bad-write”, and “bad-jump” tests

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.

We did not add any global/static variables or modify any of the existing structs to accomplish argument passing. All argument passing code was written in process.c and did not require modifying other files. All modifications to structs and additional variables were only added to accomplish various syscalls.

---- 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?

Argument passing is all accomplished in process.c. We begin in process_execute(), where we grab the executable file name (ignoring the args) and pass it to thread_create() as the “name” of the new thread. The thread_create() function spools up a new thread that begins in start_process(). In start_process(), we initialize the frame and jump to load(), which will attempt to open the executable file. In load(), we attempt to open the executable file by passing the filename to the file system. If the file exists, and the load was successful, then we must set up the stack for the new thread in setup_stack(). The setup_stack() function was the method we primarily modified for argument passing.

In setup_stack(), the stack pointer (esp) begins at the top of the frame (PHYS_BASE). We then tokenize the string and count the number of arguments. This allows us to initialize an array that will point to the pushed string args (argv pointers). We then tokenize the string once more and push each string onto the stack, moving downward each time (order doesn’t matter here since they will be referenced by pointers). We save the esp pointer to each argument in our initialized argv pointer array. After we have pushed all the command line arguments, we then word align the stack pointer since memory access is more efficient when aligned. Following the word align, we push four bytes of 0s to represent a null sentinel. Then, we push the pointers to the arguments we first pushed in right to left order. We then push a pointer to the first argument pointer. Then, we write the number of command line arguments (argc). Finally, we add a fake return address (four bytes of 0s).

---- RATIONALE ----

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

The strtok_r() function is a “reentrant” version of strtok(). The reentrant versions of functions take an extra argument to store the state between calls (often called save_ptr). This allows for the reentrant versions of functions to be called from multiple threads simultaneously, used in nested loops, run in parallel, etc. On the other hand, strtok() saves a global state and cannot accomplish many of the parallel uses of strtok_r(). For this reason, Pintos implements strtok_r() because of its numerous benefits when programming in parallel (a common occurrence in operating systems).

(https://stackoverflow.com/questions/22210546/whats-the-difference-between-strtok-and-strtok-r-in-c)

>> 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.

1. Forcing the kernel to parse the commands into executable names and arguments introduces security vulnerabilities. The kernel must carry the added security overhead of verifying existence of files, protecting against buffer overflow of command line input, ensuring the user has permission to access the files they are trying to execute, etc. This could all be handled by the shell before being passed to the kernel, thus, we reduce both the amount of edge cases for the kernel to handle and likelihood for security exploits (simplifies code, easier to track down bugs, new executable extends privileges of shell).

2. If the parsing of command line input were to cause a PANIC and crash the system, then the entire kernel would crash. If we isolate the parsing and validation to the shell, then only the shell would crash on a panic - which is much easier to relaunch than the entire kernel.

3. Since there can only be one kernel, there is not much room to customize it for each user. However, if we implement command line parsing at the shell level, then each user can effectively have their own customizable shells. In terms of design, the shell would also act more as a true interface rather than a pipeline to the kernel (with all the added benefits of code abstraction and interfacing).

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.

>>thread.h

>> struct thread (modified)

{

#ifdef USERPROG

uint32_t *pagedir; - Page directory

struct semaphore load_sema; - Parent waiting for child to load

struct semaphore exec_sema; - Parent waits for child to finish exec

struct list children; - List of children the thread has

struct thread * parent; - Pointer to the threads current parent

struct list open_files; - A list of struct open file

struct file *executable_file; - Pointer to executable file that started thread

int fd_counter; - Seed generator for files

#endif

}; - Added to thread struct to keep track of child and parent processes along with files it currently has open/access to

>> struct thread_open_file (new)

{

struct list_elem elem; - list elem used to add in open_files list

int fd; - File descriptor

struct file *file; File pointer of current file

}; - A struct to contain information about the current open file of a thread

>> struct thread_child (new)

{

struct list_elem child_elem; - list elem used to add in child_list

struct thread * child_thread; - pointer to the real thread child

tid_t tid ; - tid of this child (identity map with pid)

bool has_been_waited_on; - to check if wait() already called before

bool load_success; - to check if load succeeded or not

int exit_status; - the status the child thread exit with

}; - Contains all necessary information for managing child threads.

>>syscall.c

>> static struct lock file_lock; - Lock for syscalls dealing with critical sections of files.

>> static struct lock sys_lock; - Lock for syscalls dealing with read/write system

>> 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 open files via a seed (counter) that we assign to each file upon creation. We start our seeding out at 2 (because 0 and 1 are reserved for STDIN and STDOUT) and increment the seed property on the thread after we open a new file. This guarantees each new file opened will get a unique seed (within a process, up to max size of integer).

The descriptors 0 and 1 are unique within the entire OS and cannot be assigned to any file. The file descriptors we generate are unique within a single process. Thus, two threads may have the same file open, and each thread may have a different file_descriptor that references that same file.

---- ALGORITHMS ----

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

For reading a file, we make sure that we are not reading from STDOUT. If it is, we return 0. If the file is STDIN, we acquire a lock, then read the file line by line, storing the contents into an unsigned character buffer, then finally release the lock and return the size. If it is neither of these file types, we acquire a lock, find the correct open file based off of the file descriptor (we exit the function if the file is not found), gather the number of bytes in the file using the ‘file_read’ function, then finally release the lock and return the number of bytes in the file.

For writing to a file, we make sure that we are not writing to STDIN. If the file is in STDIN, we exit the function. If the file is in STDOUT, we call ‘putbuf’, which takes a buffer and the size of the file, and finally return the size. Otherwise, we acquire a lock, find the desired open file based on our file descriptor parameter (if the file isn’t found, we exit), we write to the given file using our buffer parameter, as well as our size parameter, and keep track of how many bytes have been written. Finally, we release the lock and return the number of bytes written.

>> 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 number of possible inspections if all the space is being copied into one full page is one inspection (considering the page is large enough in size to take all of this data). However, if the each byte is stored on a separate page, the maximum number of inspections would be 4,096. For a system call that only copies 2 bytes of data, the minimum number of inspections would be 1, and max being two (if those two bytes got split into two pages).

There is room for improvement in these numbers by reducing the amount of internal fragmentation and keeping a high degree of locality when copying user memory into the kernel. This would drastically decrease the possible number of inspections needed when running user programs.

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

Our system call for wait simply calls process_wait() in process.h since we handle process code inside of that file. It interacts with process termination by getting the current child based off the passed in child_tid (utilizing a helper method) and then makes sure we have not waited on this child before (aka it has become a zombie), if so this returns -1 and terminates. If it makes it past this, we set its waited on property to true and then checks to see if the child process is still alive. If this process is still alive we wait, otherwise just return the exit status of the thread.

>> 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.

Managing bad passed in pointer values from user specified addresses can pose several problems ranging from design, error checking, and security issues. In order to best handle the

acquisition of user data (such as executable names, arguments, etc.) in a way that was not prone to errors or crashes, we decided to immediately check to make sure the executable passed in was indeed from the user's address (safe code option, instead of handling in page_fault()). Any subsequent arguments were checked on a case by case basis depending on which system call was to be utilized.

To avoid obscuring this, we utilized a helper method that verified a certain address was indeed a valid address for a user to be accessing (within bounds and mapped). This helped us re-use code and keep design much simpler. When an error did arise, we always called exit with an error status of -1. Inside of exit, we ensured we properly disposed of all traces of an error causing thread by removing it from the child list of its' respective parent. We did not have to worry about releasing locks in this case because it never got to a critical section that is inside of system call functions.

One example of this was the boundary issue checking that a first byte of memory of passed in arguments was okay but the subsequent passed in bytes were not. We handled this by checking to see if the very end of the arguments address was valid, ensuring us that the last argument did not go out of it's bounds to cause an issue.

---- 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"?

In our thread_child struct, we have two semaphores - a load_sema and an exec_sema (both initialized to sema values of 0). When the "exec" syscall is initiated, the parent calls process_execute() and attempts to create the new thread to host the executable. The new thread will be created before returning from process_execute(), so we add the new thread as a child of the currently running parent thread. There are two cases after returning from thread_create(): the parent retains control and finishes process_execute() or the child preempts the parent and begins with process_start ().

In the first case, we return to the exec function and have not finished loading the executable. So we "down" the load_sema and the parent thread is immediately put to sleep (since load_sema's original value was 0). This allows the child thread to be scheduled and continue with execution. The child thread will attempt to load the given executable. It will then mark, in the parent's children list, if it was successful and "up" the load_sema that the parent thread is sleeping on. The parent thread can then guarantee that the executable has at least attempted to load. It checks whether the load was successful, and returns -1 if it was not.
In the second case, the child thread is loaded before the parent is stuck in the load_sema. It will mark if it was successful or not, then "up" the sema. When the parent thread regains control and

“downs” the sema in the “exec()” syscall, it will just continue and act on a failed load if it occurred.

>> 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?

P calls wait(C) before C exits: In this case, we use the exec_sema on the child to put the parent thread P to sleep until C is finished executing. Once C exits, the parent thread is woken up by “upping” the exec_sema on the child.

P calls wait(C) after C exits: In this case, the child C has upped the exec_sema and the parent P will not be put to sleep. The exit_status of C is stored as a property of the child in the parent P’s children list.

How to ensure all resources are freed in each case: In both cases, the child thread must always call the exit() syscall, then thread_exit(), then process_exit(). Right before the child is finished exiting (in process_exit()), we always free all of its resources, no matter the case. So even if P calls wait(C) after C exits, its resources will still be removed.

P terminates without waiting, before C exits: In this case, C would become an orphan (parentless). We would like to prevent children from becoming orphans, so right before P finishes terminating, we also terminate/free all of its children.

P terminates without waiting, after C exits: In this case, P would never know to free C’s resources. However, in exit() syscall, we handle this by removing the child_elem from the list it resides in (which would be the parent’s children list). So even if P never waits on C, it will still eventually be removed from its children list.

Special cases: If a parent thread P tries to wait on the same child C twice, we prevent this and just return -1 (no zombies). If P attempts to wait on multiple children at the same time, then it will wait on the one who acquires the semaphore first. If P does not wait on C, and C terminates, then C will remain a zombie until P exits and frees children.

---- RATIONALE ----

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

We had two options for accessing and validating user memory access from the kernel. The quicker way (used in real operating systems, but more difficult to implement) was checking if the thread was below PHYS_BASE and then dereferencing it. If an error occurred, it would be handled in page_fault(). This method is much more efficient because it utilizes the MMU. We, however, decided to play it safe and check that each address/byte was valid before dereferencing it. Although this was more inefficient in terms of performance (since we always have to check that a byte is valid), we could handle all errors directly in syscall and did not have to directly get/put user memory using assembly instructions. We were also able to verify that entire addresses were valid before dereferencing them - ultimately being easier to maintain.

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

Our file descriptor design has several advantages that led us to implementing it in the way that we did. One of the main one's was the light-weightness of just using essentially an integer primary key to keep track of open files for a given thread. Another advantage is we can hold the max possible size of an integer of files per thread since we are using an integer as a seed. One last advantage of this was that we could ensure no thread would ever assign a fd with a value of 0 or 1 (since this is reserved for STD_IN and STD_OUT) which would cause many issues.

Possible disadvantages include if a thread somehow got enough open files to max out an integer, we would run into some overflow issues/fd being written over which would no longer make them unique. Another disadvantage is we never recycle fd if files are being closed out which theoretically a long running thread could run into issues with this if it opened 2 million+ files during its lifetime.

>> 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 did not see the need to change the mapping between tid_t and pid_t because only one thread was generated per process. However, if we were required to implement multiple threads per process, then it would make sense to have some sort of mapping mechanism that is able to map multiple tids to a single pid so that each process can manage multiple threads.

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?