PROJECT 2: USER PROGRAMS
DESIGN DOCUMENT

---- GROUP ----

Tammy Chang tammycha@buffalo.edu
Tyler Craven tjcraven@buffalo.edu
LangXian Xu langxian@buffalo.edu

---- PRELIMINARIES ----

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

>> Describe briefly which parts of the assignment were implemented by
>> each member of your team. If some team members contributed significantly
>> more or less than others (e.g. 2x), indicate that here.

Tyler Craven
Tammy Chang
Sunny Xu

>> 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://linux.die.net/man/3/strtok_r

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.

```
#define DELIMITER " "
```
We define this global variable “DELIMITER” for consistency when using strtok_r.

---- 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 implemented argument passing by tokenizing the given file. In process_execute() we take just the file name and make a copy of the original file to use to retrieve the rest of the arguments later. When load() is called it parses the rest of the arguments from file_name and then filling an array with just the arguments. It then takes the arguments and places them in the correct order onto the stack.

We arrange the elements of argv[] to be in the right order by breaking the command using strtok_r and then place the words at the top of the stack (order does not yet matter because they will referenced through pointers). We then round the stack pointer down to a multiple of 4 before the first push because word-aligned accesses are faster than unaligned accesses. After, we push the address of each string plus a null pointer sentinel onto the stack from right-to-left order (these elements are of argv). Then we push the address of argv[0] and argc in that order, followed by pushing a fake “return address.” (Section 3.5.1 of Pintos Guide)

(The reason we are pushing in reverse is because argc after it pushes all the arguments to argv[] is at the last argument in the array)

In the case the stack page does overflow, we always check the number of arguments being parsed in start_process() when filling the array. We compare the size of the args compared to the stack size, and if it overflows, we automatically call exit().

---- RATIONALE ----

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

The major difference between strtok_r() and strtok() is the introduction of the save_ptr variable by strtok_r(). Strtok() function uses a static buffer while parsing, so it is therefore not thread safe (https://linux.die.net/man/3/strtok_r). In other words, the introduction of the save_ptr variable prevents corruption for if another thread were to call strtok_r() this save_ptr would not be changed.

>> 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. One advantage of the shell separating the command name and arguments instead of the kernel, is that it is “safer” for the shell to read the input than for the kernel to. If for instance, something happens to go wrong reading the input, the shell is subject to changes to accommodate for virtually any situation including invalid inputs since it is not part of the kernel.

In addition, the shell does not interrupt any other processes that may be taking place using the kernel.

2. Another advantage of using the shell to read the input is that the kernel handles the commands in a slower manner. When an error occurs while handling a command in the kernel, one or more interrupt handlers are called. This could be problematic since other components could be called and slow down multiple processes which is a slow way to deal with errors. Instead, if we were to use the shell, we pass in the commands as a standard input and get a standard output which is faster than invoking the kernel.

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 child_thread{

tid_t child_id;                  // child id
bool exit_stauts;                // child thread exit status
struct list_elem child_elem;     // child in parent’s child list
struct lock;                     // lock for managing access to struct

}

struct parent_thread{

tid_t parent_id;                 // parent id
Int load_child;                  // checks if child_thread loads correctly
struct lock wait_child;          // lock that used to wait for child
struct list child_list;          // list that keep tracks of all child_threads
struct file *run_file;           // shows the executable file of current thread
}

struct file_descriptor{

int fd_id;                       // id for every unique file returns to user process
tid_t parent_thread;             // paren thread that opens the file
struct list_elem file_elem;      //  file elem from file list that holds all unique opened files

}

struct list files;               // list of all unique opened files
```

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

When a file is opened, the operating system creates a file descriptor to keep track of the information stored for that particular opened file. Every file that is opened, a corresponding file descriptor is assigned to that file. Each process has to keep track of all the file descriptors. So the file descriptors are unique within a single process only. When a file is closed, the file descriptor is freed and can be used for another file.

---- ALGORITHMS ----

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

In our implementation, in both reading and writing, all the arguments(file descriptor, pointer to a buffer, size of the buffer) will be checked to see if they are valid inputs or not. If anyone of these arguments is invalid, the thread will be exited immediately and the process will be terminated. Then we proceed to check if the thread is reading from input and writing from output, otherwise the process gets terminated.

After all the conditions have been checked, the process will read and write files using methods from filesys and lib directories.

>> 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?

For a full page of data to get copied, the least number of possible inspections is 1. With one call to pagedir_get_page() it can see that memory has enough space for 4096 bytes of data.

The greatest number of possible inspections is 4096. This is because of the possibility that each ELF segment is one byte in size. So there would be one byte for one inspection, amounting to 4096 inspections.

For a system call that only copies 2 bytes of data, the least number of inspections possible would be 1. With one call to pagedir_get_page() it will see the kernel virtual address has room for 2 bytes of data, hence only one inspection is needed.

The greatest number of inspections would be 2. This accounts for the possibility that there is only ELF segments that are one byte in size. If this occurs, there would be two inspections needed, one byte for each inspection.

We do not believe there is much room for improvement.

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

In our implementation, the wait system call first checks if there’s any child thread available. If there is, it then checks parent thread’s child list to see if the specified child pid is matched. In case of child thread finishes running, it would simply return exit status and the child thread will be removed from child list. Otherwise, child thread’s semaphore will be downed until the thread is done, and semaphore will be upped as soon as the wait is over.

If a child process terminates, it checks its parent thread, return exit status, and signals parent thread so it could see the updated status and return. If a parent process terminates, it attempts to remove all the child threads from the child list, and then the child struct will be freed and update its statu to exit.

>> 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.

In our implementation, we would avoid bad user memory access by taking all the arguments from each system call and making sure every argument is valid from the stack. If there’s any exception/error, the process will terminates. No resources, such as locks, buffers,etc, would be temporarily allocated before all the arguments are being checked and verified. If there’s a bad memory access after arguments were checked, temporarily allocated resources would be freed inside process_exit and before process terminates.

For instance, if a process tries to read a pointer that is not in the stack or is null, the process would simply terminates and thread would exit. If there are any temporarily allocated resources, all the resources are free/return before the process_exit.

---- 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"?

When the start process calls the executable in the current implementation it will load exec(). If exec() loads successfully then the current and the new threads function properly. However, if exec() fails to load the thread it fails and this will cause any new thread calling to start_process() to fail as well.

To avoid the situation, we can load exec() before hand, and return a -1 to start_process() for the failed loading. We could make a new method which could be used to preload exec() and call this method preexec(). In preexec(), if exec() fails loading, then preexec() would return a -1, which would be then be given to the start process function. So this would avoid any issues that would be encountered by the other threads or new threads calling start_process(). If exec() loads perfectly in preexec() then preexec() can return exec() and exec() can be called by start_process().

>> 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?

In our implementation, we ensure proper synchronization using the wait semaphore. Since P always owns each of its child thread's memory and child thread is responsible to set its status in parent's thread struct, whenever a wait() returns from in parent, parent frees the memory and release C from its child list or release every C if parent terminates.

When P calls wait(C) before C exits, P will acquire the semaphore and wait until C exits. Then parent will retrieves the child's exit status.

When P calls wait(C) after C exits, P acquires the semaphore and checks C already exited so P can just free them from its child list.

When P terminates without waiting before C exits, P frees every C in its child list and the semaphore will be released as well. P sets C's parent pointer to null, and since every C has finds out that its parent has been already terminated, it will continue and ignore updating its status in P.

When P terminates without waiting after C exits, as mentioned above, P frees every C in its child list and sets C's parent pointer to null, so C won't update its status to a dereference dead memory.

All resources are freed upon process_exit.
---- RATIONALE ----

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

We decided to follow the pintos manual suggestion and accessed user memory from the kernel by verifying the validity of a pointer and then dereferencing it. If we encountered an invalid user pointer after, we made sure to free the page of memory. This was the easiest way to avoid memory issues and possible page faults.

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

Some advantages to our design for file descriptors is that the kernel can see the amount of open files so it can account for the problems that may arise. Also since each thread has an array containing the file descriptors, there is virtually no limit on the number of file descriptors. Since the file descriptors are all stored in the same way, they are also used in the same way, which is easier to read.

A disadvantage for our design of file descriptors is that having an array for each thread consumes a lot of space, which may be prone to crash the kernel depending on the size of the array.

>> 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 change the identity mapping.

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?

I think that there was enough time to finish the project. It was challenging but still doable.

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

Yes, this project in conjunction with the first pintos project provided a well rounded overview of OS design and implementation. Specifically, creating system calls so that the computer program can request a specific resource from the kernel allowed us to better understand operating systems as a whole.

>> 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?

No we did not find any guidance misleading, I think the stack being inverted was confusing at first, but was explained in lecture.

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

I think that the TA's have provided ample assistance to the students. With recitations and supplementary office hours, in addition with the professors it has been more than enough.

>> Any other comments?

None.