+-------------------------+
| CSE 421/521 |
| PROJECT 1: USER PROGRAMS |
| DESIGN DOCUMENT |
+-------------------------+

---- GROUP ----

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

Yizhan Wu <yizhanwu@buffalo.edu>
Zhenyu Yang <zhenyuya@buffalo.edu>
Wenting Wu <wentingw@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.

FirstName LastName: contribution
FirstName LastName: contribution
FirstName LastName: contribution

>> 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.

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.

No struct is created or modified in this phase.

---- 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?

How do you implement argument parsing?

The original process_execute() reads the file name from command as file name with arguments. So, we need to split the file name with arguments first. Thus, we use strtok_r() function to split the command line string.

When we create a new thread, the real filename will be used as the thread name. And the arguments will be passed to function start_process(), load() and setup_stack().

The crucial part of this task lies in setup_stack() function. When initializing the page, the arguments and commands are pushed into the stack.

How do you arrange for the elements of argv[] to be in the right order?

We check the strings in the order of last to first, in which the last string will be the command and the first one will be the last argument. For example, in a string 'run echo x', the first element would be 'x', and 'run' would be the last argument.

How do you avoid overflowing the stack page?

In our design, overflowing will be detected as page fault exception. When overflowing occurs, exit(-1) will execute.

---- RATIONALE ----

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

Strtok_r() would return save_ptr. Save_ptr stores the address of the argument, which allows to be used in future steps. In here, the argument is filename.

>> 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.

Advantages: 1. The Unix way of separation saves kernel cost. 2. The Unix way of separation allow the system to check the filename and arguments are valid or not.

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.

In threads/thread.h, added:

```
    struct thread{
      int child_status; // status of children
      struct list children_list; // list of child threads
      struct file *exec_file; //executable file
    }
    struct childrenStatus{
      tid_t child_tid;// identifier of child thread
      bool is_exited: // existed or not existed
    }
```

>> 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 file will be allocated to a unique descriptor, as a number.

---- ALGORITHMS ----

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

Step 1: the buffer pointer range will be checked;
Step 2: see if the buffer pointer is doing reading operation or writing operation;
step 3: the descriptor of the thread will hold the lock when the file was being written or read. When the reading or writing operating stopped, the lock will be released.

>> 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 inspection is 1, the greatest number is 2. The operation is the same for a system call that only copies 2 bytes of data.

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

The corresponding child thread will be found under the parent thread, and then we execute process_wait() function. When the child thread is terminated, all resources that it hold will be released.

>> 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.

Before accessing memory, we will check the buffer pointer to make sure all arguments of syscall are in the user memory, not in the kernel memory. All pointers with Null value or in the kernel memory will point to kernel to cause page fault. The reason for that is once the page fault occurs, sys_exit() will be called to 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"?

The child_status in thread struct will be updated when the status of child thread changes. When the child thread is created, the its status would be set to LOADING. If the thread is successfully executed, the process_execute() would return the thread id of the thread. If the value of status is FAILED, then process_execute() would return -1.

>> 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?

The expected situation is parent process P executes wait() c before c exits. However, if wait() c is executed before c exits, then p would search the global exit list. If P is terminated without waiting, then c would appear to be losing control, but the operating system would check their conditions.

---- RATIONALE ----

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

>> kernel in the way that you did?

Because validate arguments and status are easy to control and very straightforward.

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

Pros: our design is able to control the thread and the corresponding attributes from a global view. For example, all the children status could be found in the status list.
Cons: The cost might be a little bit high.

>> B11: The default tid_t to pid_t mapping is the identity mapping.
>> If you changed it, what advantages are there to your approach?

There is no need to change.

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?