```
+--------------------------+
|          CS 5600             |
| PROJECT 2: USER PROGRAMS     |
|       DESIGN DOCUMENT        |
+--------------------------+
```

---- GROUP ----

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

Douglas Johnston <johnston.5douglas@gmail.com> <douglasj>
Heng Xu <xu.he@husky.neu.edu> <hengx>
Wanting Jiang <jiang.wa@husky.neu.edu> <wanting>

---- 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.
Online sources for syscall_handler:
https://github.com/ryantimwilson/Pintos-Project-2/blob/master/src/userprog/syscall.c
Online sources for argument passing:
https://github.com/ryantimwilson/Pintos-Project-2/blob/master/src/userprog/process.c

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 new or changed ’struct’ or `struct' member for argument passing.

---- 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?

First pass the while file name from process_exec() to start_process(), then pass it to
load(), int method load(), call setup_stack() to finish the argument parsing.
1. set an array argv[] to record the argument address in the stack;
2. get all the arguments, put them into the stack, then record their address in the argv[];
3. word align;
4. push argv[] into the stack, start from the last element in the argv[], so the address is put
   as a reverse order in the stack. That make sure we can get the argument in order;
5. push the address of argv to the stack, so process can get the argument's address later;
6. push the size of argv into the stack;
7. push the return address.

Every time, before push something into the stack, we will call is_kernel_addr() to avoid
overflowing the stack page.

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 to record the child threads status*/

```
struct child_process{
     int pid;
     int status;     /* Process exit status. */
```

```
    bool waited;    /* True: Process is waited by parent 
process;
                       False: Process is not waited by parent 
process. */
    bool exit;      /* True: Process has been exit;
                       False: Process is still running. */
    int loaded;     /* Status of loading executalble file.
                       0: not loaded; -1: load failed; 1: load 
success. */
    struct semaphore sema_exec;  /* Semaphore used in 
sys_exec.*/
    struct semaphore sema_wait;  /* Semaphore used in sys_wait. 
*/
    struct list_elem elem;
};


/* Struct to record opened file. */
struct open_file {
    struct file *file;
    int fd;       /* file descriptor. */
    struct list_elem elem;
};


/* Added the following fields to struct thread. */
struct list opened_files; /*list of open_files that the thread 
is referencing*/
struct thread *parent;             /* Parent thread. */
struct list children;              /* List of child_process. */
struct child_process *self_child;  /* Self Status. */
struct file *exec_file;           /* Executable file this 
thread opened. */
```

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

We added a new structure, struct open-file. This contains a file descriptor, a pointer to a file, and a list-enum to combine them into a list.
Each process has its own set of file descriptors, so a file with fd 1 for process A might be associated with fd 4 in process B. So, they are only unique within a single process.

---- ALGORITHMS ----

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

1. Reading:
   1) if it is STOUT_FILENO, call exit(-1);
   2) if it is read from buffer, which type is STDIN_FILENO, call input_getc(), to get the input and return the size.
   3) if it is read from from a file. First get the open_file from the opened_files list, then call file_read() to read content from the file.

2. Writing:
   1) if it is STOUT_FILENO, call exit(-1);
   2) if it is read from buffer, which type is STDIN_FILENO, call putbuf(), to get the input and set the exit status if this process to size.
   3) if it is read from from a file, call file_write() to write to a file.

>> 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, the least number is 1. If the first inspection returns page table successfully,
it can contain on page of data, so we don’t need to inspect any

more.
The greatest number might be 4096 if the page is not contiguous. In this case, we need to check
4096 time to make sure the access is valid.

For 2 bytes data, the least number is 1. If we get a kernel address which has more than
2 bytes space, then that’s done.
the greatest number is 2. If every time the kernel address we got is only 1 bytes.

We have no idea about the improvement.

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

The implementation of “wait” is in process_wait().
We added a new struct child_process to record the status of a child process. And every process
has a filed called children, which record all its child processes.
So every time we created a child process, we will create a “struct child_process” for this
child process, then add is to parent’s list children.
When “wait” is called, we get the child process from the children list.
After getting the child process, first check the “waited” filed, if it is true, that means this
process has been waited before, call exit(-1). Otherwise, set the “waited” to true, then sema_down
the semaphore “sema_wait”, with this semaphore parent process can communicate with child process.
When the child process exits, it will sema_up this semaphor, then parent process can continue.
If the parent process terminate before the child process exit, it will release all the list, then child
process will know parent process exits.

---- 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 used a semaphore “sema_exec” to ensure “exec” won’t be returned before the new executable
has completed.
In exec(), after calling process_execute(), we will check all the “loaded” field of the child process.
If the “loaded” is 0, which means the executable has not been loaded yet, then sema_down “sema_exec”.
After the child process loaded its executable, it will sema_up(sema_exec), then parent process can continue.
If the “loaded” is -1, which means the child process load the file failed, so process should exit(-1).
After checking all the child process, exec() can return.

>> 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?

As mentioned in B5, we used a semaphore “sema_wait” in process_wait().
If P called wait(C) before C exits, it will sema_down(sema_wait) then waiting
for sema_wait to be sema_up. And sema_up will only be called in exit(), so if
process C doesn’t exist, P will wait C forever, because it cannot sema_down(sema_wait) won’t
be success.
If p called wait(C) after C exits, then sema_up(sema_wait) has been called,
so sema_down(sema_wait) will be success.

Semaphore is defined in the struct child_process. If a parent process terminates,

it will free all its lists, including the children list which records all the child_process.
That ensures all records are freed in each case.

P terminates without waiting should be same as after waiting, because no matter C is waited
or not, when P exists, it will free all the resources.

---- RATIONALE ----

>> B9: Why did you choose to implement access to user memory from the
>> kernel in the way that you did?
We verified the validity of a user-provided poster every time before dereference it because it
is the first idea we have in mind. But it seems we miss some place so it still cause the “page fault”
exception in some tests, so we also modifying the page_fault() code to make sure we handled all the
“page fault” exception.

>> B10: What advantages or disadvantages can you see to your design
>> for file descriptors?
Advantages:
1. Easy to implement.
2. We can find the file easy from the children list.

Disadvantages:
Different processes can open a same file, so we stored the same file information in different,
that wastes the kernel space.

>> 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 the tid_t to pid_t 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?

>> 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?