```
                    +--------------------------+
          | CS 140                             |
                    | PROJECT 2: USER PROGRAMS   |
                    | DESIGN DOCUMENT            |
                    +--------------------------+
```

---- GROUP ----

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

Zhaozheng Shen <shenzhzh@shanghaitech.edu.cn>
Wenhui Qiao <qiaowh@shanghaitech.edu.cn>

---- 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.

CSCI 350: Pintos Guide (Written by: Stephen Tsung-Han Sher)

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.

None.

---- 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?

1. Briefly describe how you implemented argument parsing, how do you arrange for the elements of argv[] to be in the right order?

Firstly, in process_execute(), the input sentence is composed of program name and arguments, that is, the first word is the program

name, the second word is the first argument, and so on. So I need make a copy of the input, pass the first word into thread_create's thread name, and pass the copy sentence into thread_create's last argument as the input of function: start_process(). When calling thread_create(), it calls start_process(). In start_process(), we first initialize interrupt frame and load executable, which calls load() function. if we load successfully, we need to set up stack, put the arguments into the stack, and then, the new process successfully start, otherwise we exit the new thread. And what we need to do in argument parsing is to set up stack, put the arguments into the stack after load success.

Since esp is initialized with PHYS_BASE, which is the top of the stack. So I need to increase esp and put the argument into the stack. Since I need to put everything in reverse order, so I first segment the input argument sentence with blank space, and store them by a array following the order from the way of going, the left to right. Next, I put these arguments into the stack in reverse order, from the right to left, and each space I allocate length(argument)+1 space, to ensure every argument ends with '\0'. Then, do the word alignment operation for best performance. Then, push the address of argv[i] in reverse order, right to left, and push argc. Finally, push a fake "return address".

2. How do you avoid overflowing the stack page?

We didn't check esp overflowing until it fails. When it fails, it turns into page fault exception in exception.c, and before it print the error message, we change the process_terminate_message into -1 and call thread_exit(), it is the same as calling syscall_exit().

---- RATIONALE ----

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

It is safer since strtok() has a global variable, and the global variable is unsafe when we have many threads, and the global variable is can be uncertain.

>> 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. Faster. We don't have to change into kernel mode to do the separation operation.
2. Safer. The shell does this separation so kernel will not be

affected by the separation.

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.

Add a new struct:

```
        struct struct_child
        {
                tid_t tid;                                          /
* child thread's tid */
                int process_terminate_message;              /*
child thread's exit code*/
                bool bewaited;                                  /*
check whether be waited */
                struct list_elem child_thread_elem;         /* The
element as a child. */
                struct semaphore wait_child_process;/* Semaphore to
wait child process. */
        };
```

Global variable:

```
        #define PROCESS_FILE_MAX 128    /* the limit number of
files owned by a process.*/
```

Added to struct thread:

```
        /* Members for implementing system call. */
        int process_terminate_message;              /* The
terminate messge of thread. */

        bool check_load_success;              /* Indicate whether
the load is success. */
        struct semaphore child_lock;                          /*
Semaphore on the lock. */
        struct thread* parent;                          /* Pointer
to the parent thread. */

        struct file *process_files[PROCESS_FILE_MAX];  /* The file
that the procss has. */
        int fd;                                             /* fd
value of the process. */
        struct file *this_file;                         /* Pointer
to the file itself. */
```

```
        struct list child_list;                                 /*
The list of children. */
        struct struct_child *children;                           /*
element as a child. */
```

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

1. Describe how file descriptors are associated with open files.
Then original value of every process' fd is 2, since fd = 0 is for STDIN_FILENO, and fd = 1 is for STDOUT_FILENO. When one process open a file, the process's fd increase one based on its original value. When one process close a file, the process's fd will not decrease.

2. Are file descriptors unique within the entire OS or just within a single process?
In out implementation, it is not unique within the entire OS, but it is unique within a single process. Since when one process open a file, only the process's fd increase one based on its original value. But for those processes who have the same fd value, their corresponding files may be different because, they all own fd by themselves.

---- ALGORITHMS ----

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

Reading:
If fd == 0, it means out put to console, then call input_getc() to read from the keyboard.
If fd == 1, then return 0 immediately since STDIN_FILENO should not write.
If fd is invalid, i.e. negative or beyond the limit, then return 0 immediately.
If the file is null, then return 0 immediately
If everything is ok, then call the function file_read(). Use a lock to deal with the synchronization. Return the value returned by file_read(), which is the size bytes read.

Writing:
If fd == 1, it means writes to the console, then call putbuf() to write all of buffer.
If fd == 0, then return 0 immediately since STDOUT_FILENO should not read.

If fd is invalid, i.e. negative or beyond the limit, then return 0 immediately.
If the file is null, then return 0 immediately.
If everything is ok, then call the function file_write(). Use a lock to deal with the synchronization. Return the value returned by file_write(), which is the size bytes write.

>> 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?

1. full page of data:
The least number is 1, they are all in the same page.
The greatest number:
  a.If contiguous, the greatest number is 2.
  b.If incontiguous, the worst case is that they are all in the different page, so the   number id 4096.

2. 2 bytes of data:
The least number is 1, they are all in the same page.
The greatest number is 2, they are all in the different page.

3. Is there room for improvement in these numbers, and how much?
They can be improved into one page.

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

After checking it and validating it before using it in system call, we call process_wait() immediately. The input is the tid the current process need to wait, and the return value is the child thread's process_terminate_message.

We first find the process with tid,
If TID is invalid, returns -1 immediately, without waiting.
If it was not a child of the calling process, returns -1 immediately, without waiting.
If process_wait() has already been successfully called for the given TID, returns -1 immediately, without waiting.
Else, everything is ok, use sema_down to let the current process wait, and let the child process with that tid run. Only after the child process finish running and sema_up, will the parent process

recover to run. And the first thing the parent process need to do is remove the child thread from its children list and then, it can continue its work.

>> 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 calling system call function, we need to check, whether the address pointer is null, whether it is user address, whether it is kernel address, whether it is mapped into the page directory. If any of the check operation is not satisfied, the process will call syscall_exit() immediately to end finish running the process and close the file corresponding to the process, and close all file owned by the process.

For example, we need to call write system call. So firstly, check each parameter(fd, *buffer, size) valid or not, we need to check those address pointer valid or not by checking: whether null, whether is_user_vaddr(), whether is_kernel_vaddr(), whether pagedir_get_page() is valid or null. If any of the condition above is not satisfied, the process will call syscall_exit() immediately to end finish running the process and close the file corresponding to the process, and close all file owned by the process. If all the conditions are satisfied, then check whether ending pointer of the second parameter, buffer, is valid allow, since we need all address is valid. If not valid, call syscall_exit() immediately to end finish running the process and close all file related to this process. Finally, all conditions are satisfied, and we can do the write operation.

But during the write operation, we need to check fd value is correct or not(if fd <= 0 or fd-2 > PROCESS_FILE_MAX), then it is totally wrong, and return immediately. Then we can do the write operation with corresponding fd.

---- 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"?

1. How does your code ensure this?
I add some member into struct thread:
        bool check_load_success;
        struct semaphore child_lock;
        struct thread* parent;

After call thread_create(), we first call sema_down() to let the parent process wait, since we need to check whether load success, and decide whether to return -1.
Since thread_create() calls start_process(), and start_process() call load(), so we can check whether load success at the end of start_process(). If load success, we set the thread's parent check_load_success = false, otherwise we set it true. And then in both cases, we sema_up() to let the parent process run.
If check_load_success = false, then return -1 immediately since "exec" system call fails, otherwise, we can add the new process into the father process's children list.

2. How is the load success/failure status passed back to the thread that calls "exec"?
As mentioned above, we add the bool check_load_success to check success/failure, if failure, we set the child's parent check_load_success = false, and sema_up(), let the parent run, and the parent do the judge. Since false, we return -1 immediately. Otherwise is similarly.

>> 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?

1.
P calls wait(C) before C exits: P wait until C exit, and return child's exit code.
P calls wait(C) after C exits: P find C have been exited since P didn't find the C in child list, so it return -1 immediately.

2. How do you ensure that all resources are freed in each case?
When exiting a process, we run process_exit(). In this function, we first close all the files it has opened, and then, close the file itself. In this way, all the file resources are freed.

3.
P terminates without waiting, before C exits: P free all his children, and C find out that his father has exited, it can continue to execute.
P terminates without waiting, after C exits: after C exits, and when P exit, it free all the children.

---- RATIONALE ----

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

We check it and validate it before using it in system call, because it is time saving, we don't need to check and validate all of the memory, only need to validate what we need to use.

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

As you can see, I use an array to store the files for a process, and the array[i-2] item stores the file with its file descriptor value: i. If array[i-2] is null, it means the file at this file descriptor may have been closed or not have been opened. The array has PROCESS_FILE_MAX = 128 entries since one question on the Stanford website asked the limit number of files owned by a process, and the answer is better not but if have to, the limit is 128.

Advantage: it is very very easy and time saving to find the file with a certain file descriptor when doing system call: open, filesize, read, write, seek, tell, close. We don't have to search all files owned by a process.

Disadvantage: since we limit the number of files owned(opened) by a process, our operating system is not davanced enough. It is better not to set an arbitrary limit.

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

No, we didn't change it.

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?