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

---- GROUP ----

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

group number: 11

Yining She sheyn@shanghaitech.edu.cn

Shaoting Peng pengsht@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.

Online source: https://wenku.baidu.com/view/eed6bbdaa48da0116c175f0e7cd184254b351ba8.html#

Pintos document

```
            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.

Ideally, no new struct is needed to do argument passing. But in order to run the child thread, we have to introduce a semaphore and a thread in `thread.h` (naïvely):

```
struct thread
    {
      ...
      // record the information of a child process's parent thread
      struct thread* parent;
      // a lock for the kernel process
      struct semaphore csema;
      ...
    }
```

---- 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, in `process_execute()`, I use `strtok_r()` to get the real file name for `thread_create()` **without** arguments (arguments have been copied as `fn_copy`), and does the similar thing in `start_process()`. But I pass the full name to `load()`, and cut the name again just for `setup_stack()`.

In `set_up stack()`, In order to store the arguments in reversed order, I use the following code to traverse `file_name`:

```
for (token = strtok_r (file_name, " ", &save_ptr); token != NULL;
     token = strtok_r (NULL, " ", &save_ptr))
```

and get `argv` and `argc`. Then, decrease the `esp` `total_length` and use `memcpy()` to copy each `token` into `esp`, while move `esp` to a **high** place. In this case we can make sure that the elements of `argv[]` are arranged in the right order.

Then, use a `while` loop to check for word alignment, and copy the rest elements into stack according to the instruction in the `3.5.1 Program Startup Details`. To avoid overflowing the stack page, we set a limit to the size of arguments to be fit in less than one page size.

---- RATIONALE ----

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

`strtok_r()` has an argument `save_ptr`, which can be used to interact with the functions **outside** `strtok()`. Also, The `strtok()` function uses a static buffer while parsing, so it's not thread-safe.

> 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. The shell does separation for kernel, which can release the workload kernel is facing in Unix-like systems, thus the kernel can be more efficient.
2. The shell does separation for kernel, which can detect errors earlier to avoid possible kernel PANIC.

```
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 new struct members to `struct thread`:

```
struct thread
    {
      ...
```

```
        // record the total file descriptor number
        int fd_num;
        // a lock for the kernel process
        int exit_code;
        // record a parent thread's child threads
        struct list children;
        // a list of files opened in the current thread
        struct list files;
        // use file_deny_write() to prevent writes to an open file
        struct file * self_file;
        ...
     }
```

Create a new struct to record a child thread's info:

```
struct child_info
   {
      // record thread's id
      tid_t tid;
      // record the exit_value when using
      int exit_value;
      // check if the child process is already dead
      bool is_dead;
      // the child is not dead, parent wait for the process
      bool parent_wait;
      // similar to the original 'csema' above
      struct semaphore wait_child_sema;
      // used to traverse
      struct list_elem elem;
   };
```

Check for the loading status:

```
struct exce_file_info
  {
      // same as 'file_name' in process.
      void * file_name;
      // semaphore used to switch process between parent and child
      struct semaphore load_sema;
      // whether load is success
      bool success;
  };
```

> 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 descriptor is just like an index of a certain file. When a process wants to open or create a file, the kernel should return a file descriptor to the thread indicating the file the process is looking for.

According to the Wikipedia: `实际上，它是一个索引值，指向内核为每一个进程所维护的该进程打开文件的记录表`, so each process should have its own file list's descriptors, and they are unique within a single process.

---- ALGORITHMS ----

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

1. Read:

   First, we check whether the `fd`, `buffer`, `size` is put in the user memory. Also pay attention to the validity of pointers from `buffer` to `buffer + size`. Then, in `read1`, check whether `fd` is `STDIN_FILENO`. If so, use `input_getc()` to get input chars to buffer. If not, acquire the `file_lock` and use `file_read()` acording to the `fd` number from the open files list. Finally release the `file_lock`.

2. Write:

   First, we also check the three arguments like before. Then, in `write1`, check whether `fd` is `STDOUT_FILENO`. If so, use `putbuf()` to print the buffer context to the console. If not, acquire the `file_lock` and use `file_write()` acording to the `fd` number from the open files list. Finally release the `file_lock`.

> 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 inspections is 1, which is obvious: check for the head of that page.

The greatest number is 4096. If there's no order in the page, we need to check each address whether it's valid. However, if we know that it's contiguous, we only need to check twice: one is the head, the other one is the tail. That's enough to ensure it's validity.

As for a system call that only copies 2 bytes of data, the analysis is like above. the least number of inspections is 1 and the greatest number is 2.

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

"wait" syscall returns the value of `process_wait()`.

In `process_wait()`, we first search for `child_info` according to the argument `child_tid` in order to get the Boolean variable `is_dead`. If it's already dead, return `exit_value` stored in `child_info`; else, use `sema_down()` to block the parent process in order to run the user process. One thing to notify is that we need to free the `child_info`, or there will be a memory leak.

> 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.

We detect the error using two functions provided: `is_user_vaddr()` and `pagedir_get_page()`. One is to check whether the pointer is stored in the valid virtual user memory, the other one is to check whether the map from virtual memory to physical memory is valid. If any of them failed, we record the `exit_code` as -1 and use `thread_exit()` to kill the thread. In this function, `process_exit()` is futhermore called, and that's where we free all the allocated resources.

For example, in "write" syscall, if the `buffer` is not legal for one of the above reasons, we record this thread's exit status as -1(which means wrong), then in `process_exit()`, we need to free two things: the current thread's children `child_info` and the opened files `files` the thread is associated to. In this case, we can assure that all the resources are freed.

---- 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 use a boolean variable called `success` in struct `exce_file_info` introduced before to record whether the loading process is successful. In `start_process()`, if loading is failed, we record this value as 0, and in `process_exec()` we directly 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?

When P calls wait(C), P will `sema_down()` the corresponding semaphore and wait until C exits (i.e. `thread_exit()`), where we have a `sema_up()` to that same semaphore, thus there will be no race conditions.

As for ensuring all resources are freed, we did this in `process_wait()`, where we freed all the allocated resources such as the files list. It has nothing to do with the order of process P and process C.

---- RATIONALE ----

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

We use `esp` to get the content of user memory by manipulating this pointer. Because it is the simplest idea in our minds. Also, it is convenient to check its validity.

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

We assign each thread its own `fd_num`, and associate it with the `file_node` struct. The advantage is that for each thread it's very clear about its files list and thus easy to maintain(i.e. delete files when a thread exit).

However, the disadvantage is that we may sometimes ignore the file descriptors and may even confuse about the owner of them.

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

If the mapping between `tid_t` and `pid_t` is not identity, maybe some threads' crash will not cause a process's exit. Also, we can do multi-thread operations with the support of CPU.

```
             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?