```
           +-------------------------+
           |          CS 130         |
           | PROJECT 2: USER PROGRAMS |
           |     DESIGN DOCUMENT      |
           +-------------------------+
```

---- GROUP ----

```
>> Fill in the names and email addresses of your group members.
Group 08
Yi Qian             <qianyi1@shanghaitech.edu.cn>
Guancheng Li        <ligch@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:
https://github.com/DennielZhang/SUSTech_CS302_OS_Pintos_User-Programs
(Refer to some data structures added to thread.h)

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 variables for this part.

---- 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?

Use strtok_r() to parse the raw filename and save the beginning position
for every argument.
Then travel the array of poiters in reverse order and push each argument
to stack.

To avoid overflowing the stack page, we set a maximum number of arguments
allowed to pass in.
And we check at the end of set up stack if the magic has been changed.

---- RATIONALE ----

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

strtok_r() is safe in multi-thread environment but strtok() is not.

>> 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.

The kernel code could be less than Pintos approach, so the kernel is smaller.
It's safer as if some accidents happen during parsing process, it would not affect the kernel.
And it is more convenient to change code about parsing since it is not kernel code.

```
              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":

   struct thread
   {
      ...
   #ifdef USERPROG
      ...

      /* Created for userprog. */
      struct thread* parent;
      /* Its parent process. */

      int exit_status;
      /* Thread's exit status. */

      int waited_child;
      /* Thread current thread is waiting for. */

      bool child_load_status;
      /* Whether child process is loaded successfully. */

      struct semaphore load_sema;
      /* Whether thread is waiting for another thread loading. */

      struct list child_processes;
      /* List of child processes. */

      /* Created for file system. */
      struct list opened_files;
      /* List of opened files. */

      struct file* executable_file;
      /* Current executable file. */

      int fd_num;
      /* File descriptor number. */
   #endif
      ...
```

```
    };

    struct child_process
    {
       tid_t tid;
       /* Thread identifier. */

       struct semaphore wait_sema;
       /* Semaphore used by process_wait(). */

       bool waited;
       /* Whether this child process has been waited. */

       int child_exit_status;
       /* Child process's exit status. */

       struct list_elem elem;
       /* List element. */
    };

    struct file_with_fd
    {
       struct file* file;
       /* File struct. */

       int fd;
       /* File descriptor. */

       struct list_elem elem;
       /* List element. */
    };

    In "userprog/syscall.c":
    struct lock lock_of_filesys;
    /* A lock to ensure sync of file system. */
```

```
>> 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 created a new struct called file_with_fd. It save an opened file and its file descriptor. We consider it as a whole part. When a file is opened by a process, a new file descriptor will be released to the file. Even the file is reopened by thesame process, it will also have a new file descriptor.

Within a single process.

---- ALGORITHMS ----

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

```
    Reading:
    Before calling read(int fd, void *buffer, unsigned size), we first check
    whether each argument is valid, especially buffer. If not, exit(-1)
    immediately.
    After calling read, we firstly acquire the lock of file system, then check
    the arg fd. If fd is 0, it means that it is stdin. So we use "input_getc()"
    to read into buffer and return value is the size it read.
    If fd isn't 0, we should check whether there is a file_with_fd struct
    in current process's opened file list. If we can find the struct by fd in
    current process, we use file_read() to read the file, return value is same
    as file_read()'s return value.
    Finally, release the lock of file system.

    Writing:
    Before calling writing(int fd, void *buffer, unsigned size), we first check
    whether each argument is valid, especially buffer. If not, exit(-1)
immediately.
    After calling write, we firstly acquire the lock of file system, then check
    the arg fd.
    If fd is 1, it means that it is stdout. So we use "putbuf(buffer, size)" to
read
    into buffer and return value is the size it write.
    If fd isn't 1, we should check whether there is a file_with_fd struct in
current
    process's opened file list. If we can find the struct by fd in current
process, we
    use file_write() to read the file, return value is same as file_write()'s
return value.
    Finally, release the lock of file system.

>> 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?

    /* Haven't figured out */

    Full page:
    Least: 1.
    Most: 4096.

    2 bytes:
    Least: 1.
    Most: 2.

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

    In wait() function, we call process_wait(pid) directly and return its
    return value.

    In process_wait() function, we first check whether the arg pid is valid.
```

if pid is less than 1 (often -1). It is invalid, because tid is allocated bigger than 1, we should return -1 immediately.

Then we check whether we can find child process in parent(current process) process's child processes list. If not, return -1 immediately. If we find child process's info in parent's child processes list by pid but the child process has already been waited before, return -1 immediately.

If process_wait() haven't return yet, it means that pid is valid and we can find child process's info and it isn't waited before.
But child process may be executing when we call wait. So we check whether child
process is executing by checking its exit status. If its exit status is equal to
the initial status, parent process should wait until child process finished executing.
We use semaphore wait_sema in child struct to sync the state, if child process exit,
it sema_up the semaphore.
After parent process sema_down successfully, it can get child process's exit status.
So process_wait() returns the value.

If child process has already finished executing, process_wait() returns the exit
status of child process and no need to wait.

```
>> 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 implemented a function called check_pointer:

```
void check_pointer(void* ptr, size_t offset)
    {
    if (ptr == NULL)
        exit(-1);

    bool flag = false;
    struct thread* cur = thread_current ();
    for (int i = 0; i < offset; i++)
        {
        /*  1. NULL pointer.
```

```
            2. Below kernel memory.
            3. User virtual address starts at 0x08048000.
            4. Mapped pagedir. */
          flag = ((ptr + i) != NULL)
                && is_user_vaddr(ptr + i)
                && ((ptr + i) >= (void*) 0x08048000)
                && (pagedir_get_page(cur->pagedir, ptr + i) != NULL);
          if (!flag)
            exit(-1);
          }
      }
```

   Which can check whether address space between ptr and ptr + offset is valid.
   Before we get the syscall number, we should check esp first. Then for each
   syscall function, before we dereference (esp + some offset) to get the args,
   we should check whether (esp + offset) is valid, if not, exit(-1)
immediately.

   For two special functions 'int read(int fd, void *buffer, unsigned size)' or
   'int write(int fd, const void *buffer, unsigned size)'. We should check
whether
   address space between buffer and buffer + size is valid. It also use the
function
   check_pointer.

   And there also some circumstances will cause page fault with no related to
bad
   pointer. So we let process exit(-1) when page fault.

   Resources will be released in process_exit() function.
   To ensure release the resources, besides original design in process_exit(),
we
   implemented a function called release_resources_in_thread(). The function
free
   all file_with_fd in opened files and current process's executable file
   that were once malloc() because of opening files.And free all
child_process()
   in current thread's child processes list.

---- 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 add struct thread* parent to let every process know its parent
   process when created. And there is a semaphore load_sema in parent
   thread. When parent process calls exec(), it try to sema_down the
   load_sema. When child process has completed loading, it will sema_up
   the load_sema no matter loading successfully or not. This move let
   parent process know that child process finished loading.

```
    In parent process, there is a bool called child_load_status. In
    function start_process() current thread will know whether loaded
    successfully by function load()'s return value and it also know its
    parent process. So it save its load status to parent process's
    child_load_status and then exit(-1). This ensure even though chile process
    exit, parent process can also know child process's load status.

>> 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 illustrated above, if C doesn't exit, C's exit status will be the
    initial state. P sync with C use a semaphore called wait sema.
    If C exits and C hasn't waited before, wait(C) will return C's exit status
    which stored in P's child processes list.
    As long as C called exit(), it will release resources in exit().
    If P terminates before C exits, P will release resources, so C won't get its
info
    in P's child processes list. C won't let parent know its exit status and
    sema_up the semaphore.
    If P terminates after C exits, P may be call exit(-1) by itself. So the
resources
    will also be released.

---- RATIONALE ----

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

    I try a lot in some small cases but will called bugs which are very
    difficult to find. And the structure now I think is safe and robust.

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

    Advantages: Process can control its own opened file easily.
    Disadvantages: May be crush because of open so many files mallocing
    so many spaces.

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

    Make no 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?
```