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

---- GROUP ----

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

何敖南 10132510231 <10132510231@ecnu.cn>
李梦芸 10132510105 <10132510105@ecnu.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.

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.

We only do design separate functions for the task,
These are in process.c:

```
/* parse cmd_line, separate program file name and following arguments
   and push to stack exactly as illustrated in pintos document 3.5.1. */
static bool argument_passing (const char *cmd_line, void **esp);

/* push 4 bytes of data on top of stack at *p_stack, adding safety check
   to ensure the push does not overflow stack page */
static bool push_4byte (char** p_stack, void* val, void** esp);

/* separates the program file name from command line  */
static void get_prog_file_name (const char* cmd_line, char* prog_file_name);
```

---- 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?

We scan in reverse direction to find each argument within the input command line, and for each argument encountered, we push it on top of user stack beginning from PHYS_BASE - 1; after successfully pushing all arguments and performing word-alignment, we then scan from PHYS_BASE - 1 to find the beginning address of each argument, and push it on stack. This way, when we want to get arguments back, we can just using pop operation to ensure the right order of arguments.

We add a utility function push_4byte to push pointers or integers on top of user stack page, and make boundary check to make sure the push does not overflow the stack page. For other push operations used to store argument string, we also do similar boundary checks.

---- RATIONALE ----

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

strtok() typically uses a internal buffer to store the states. The static pointer is subjected to potential race conditions and is not thread-safe.
strtok_r() takes a third arguments to determine the place within the string to go on searching tokens, and thus works in multi-thread systems like Pintos.

>> 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. when we want to extend the function or change the function of the separate command, with Unix approach, it will more convenient and with less code to change.
2. the Unix approach is safer, because the commands is separated in the shell, not in the kernel, so it will not affect the kernel
3. the Unix approach is faster, because it don't have to switch to the kernel mode to separate the command.

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 thread
{     /* Owned by thread.c. */
```

```
    tid_t tid;                       /* Thread identifier. */      enum thread_status status;
    /* Thread state. */
    char name[16];                   /* Name (for debugging  purposes). */
    uint8_t *stack;                  /* Saved stack pointer. */
    int priority;                    /* Priority. */
    struct list_elem allelem;        /* List element for all threads  list. */        /*
                                                     Shared between thread.c and
                                                     synch.c. */
    struct list_elem elem;           /* List element. */    #ifdef USERPROG      /*
                                                     Owned by userprog/process.c. */
                                                     uint32_t *pagedir;                /* Page
                                                     directory. */      //gzc_start
    struct list files;
    int ret_status;
    struct semaphore wait;
    struct semaphore t_sema;
    struct thread *parent;
    struct file *self;                      //gzc_end  #endif        /* Owned by thread.c. */
    unsigned magic;                  /* Detects stack overflow. */    };
struct fd_elem
{
    int fd;
    struct file *file;
    struct list_elem elem;
    struct list_elem thread_elem;
};
```

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

The descriptors are fid, which means the file id, and the *file, which is the pointer of file. It associated by fd_elem, and the user can only use the fid, so it's safer.
The *file is unique, it's generated by the virtual memory. The fid is also unique, because I define it as static

---- ALGORITHMS ----

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

1.static int sys_read (int fd, void *buffer, unsigned size)
① we should lock the file
② if fd == STDIN_FILENO, use input_getc() to read.
③ if fd == STDOUT_FILENO,or buffer is invalid,or buffer+size if invalid then release the lock and call sys_exit(-1).
④ use the function find_file_by_fd() to get the *f of the file.
the fuction is

```
struct fd_elem *f_elem;
struct list_elem *iter;
for(iter =
list_begin(&file_list);iter!=list_end(&file_list);iter =
list_next(iter))
{
        f_elem = list_entry(iter,struct fd_elem,elem);
        if(f_elem->fd == fd)
        return f_elem->file;
}
```

it just traverse all the list, and use the struct fd_elem to match the fd and *f.

⑤ if the f is not NULL,then call file_read(f,buffer,size).

⑥release the lock.

2.static int sys_write (int fd, const void *buffer, unsigned length)

this function is almost the same as read,except it's use for write.

①lock the file

②if fd == STDIN_FILENO or invalid buffer or invalid buffer+length then release and call sys_exit(-1)

③if fd == STDOUT_FILENO use putbuf()

④else find the file by fd

⑤if *file!=NULL call file_write.

⑥release lock and return the exit status

>> 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?

In both cases have to at least one call and at most two call. Because the content can be in one page, or spread to two page.
It can be improved to 1 page.

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

int syscall.c, the sys_wait is call process_wait().I just talk about the process_wait in process.c

① first we have to get the thread *t of the thread.I add a function in thread.c

t = get_thread_by_tid(child_tid);

this function just traverse the all_list,if the tid is equal ,then return the pointer.

② if the status is THREAD_DYING,or t->parent have already waited

the child_tid return -1.
③ if t->status is not the default status,return the status(the status is -1,I have try many times).
④t->parent = thread_current();this is to mean that the parent have waited the child.
⑤sema_down(&t->t_sema).wait until the child thread have died.
⑥when the child thread have sema_up(&t->t_sema),then printf the termination messages.
so you will see I use the semaphore to interact with process termination.

>> 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.

1. At any time we retrieve a pointer , we will check if it's null.
2. We will use is_user_vaddr() to see if it's the correct argument.
3. At any condition I end a function, we will first release the lock and free the resources.
4.when we pass a argument , we will first check the argument is correct,if not ,just exit.
example:
in the sys_write()
before we call this function, we will first check all the argument to see whether it's correct. If not ,exit.
then call the function.
if fd == STDIN_FILENO, we will first release the lock. Then return.
before I start to write, we will check

```
if(!is_user_vaddr(buffer)||!is_user_vaddr(buffer+length))
{
lock_release(&file_lock);
sys_exit(-1);
}
```

To see whether the buffer is valid the the buffer is enough to write.
If invalid, we also have to release the lock first.
At the end of the function ,we have to release the lock, then return.

---- 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"?

This problem I have mention in argument passing.
1. The status is passed back by a variable 'ret_status'.
2. In process_execute() , we will sema_down(&t->wait) to wait for loading in start_process().And then it have execute the load(),it will call sema_up (&t->wait).then in process_execute() will call process_wait(t->tid),if load failed.

>> 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 We will first find the thread *t of the child_tid, if the t == NULL, return -1,without waiting. If t->status == THREAD_DYING, return -1.
② When c is exit, it will first close all the files it have opened, and close the *self(the file it is executed).those code are in process_exit().so they can freed.
③ Then p terminates without waiting, it just have changed the child thread's t_sema, has no effect to child thread.
special cases:
1.p have already waited c.
in this case it will return -1.

---- RATIONALE ----

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

It's more efficient.because we choose the second method, only use the is_user_vaddr() to check. When you use the function pagedir_get_page(), it will waste lots of time.
Another advantage of this implementation is to prevent fault from actually happening. In each system call, we check the virtual address provided by the user, if the address is not valid, the process would be terminated rather than allowing it to execute for a while. This may reduce the risk of potential resource waste.

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

The advantages include multiple processes being able to read from different places in a single file at the same time, little information

about our implementation is leaked to user programs, UNIX file closing semantics (it wouldn't be hard to get replacement of stdin and stdout working), and good speed characteristics (no walking lists e.g.).
On the other hand, our solution limits a process to a static maximum number of open files.

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

Advantages:
1.a process can create many threads, because a pid_t can have many tid_t.
2.user can only use the pid_t, so the threads can only accessed through pid_t.

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?
No
>> Any other comments?