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

---- GROUP 19----

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

Group 19
李鹤洋 lihy3@shanghaitech.edu.cn
韩嘉恒 hanjh1@shanghaitech.edu.cn

Contributions:
Discuss together, and coding part:
Argument parsing and system call halt, exit, exec, wait by Han Jiaheng.
User space memory accessing and system call create, remove, open, filesize, read, write, seek, tell, close by Li Heyang.

---- 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.

No new declaration added in 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?

Argument parsing is done by the following steps. First we extract the name of the process before other arguments because it is used in thread initializing and the other argument parsing is done in load (), and use strtok_r() to parse the rest argument. One advantage to do argument parsing at this time is that we do not need extra memory (i.e. directly from string to stack).

The order is following the 80x86 calling convention, the arguments order does not matter but we need to push pointers to the arguments on stack from last to first.

When calling process_execute (), we will first check the length of the input string, and check whether it can be obtained by a single page. If not, we will return TID_ERROR immediately, without parsing the argument and push them on stack.

---- RATIONALE ----

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

strtok () uses global data and it is not safe if more than one thread called it and timer switches between them. strtok_r () use a local variable instead to avoid this problem.

>> 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. Shell can take this work and the pressure of the kernel will be reduced.
2. Shell provide a interface between user and kernel, so that shell can protect the kernel from being annoyed by user commend or even invalid commend.

```
			 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.
```

```c
struct thread
  {
    /* Original members... */
#ifdef USERPROG
    /* Next file descriptor number to use when open files. */
    int fd;
    /* List of files owned by thread. */
    struct list files;
    /* Pointer to parent process, NULL for main process. */
    struct thread *parent;
    /* List of all children process. */
    struct list children;
    /* Owned by list children. */
    struct list_elem child_elem;
    /* Semaphore used for EXEC syscall synchronization, shared
with parent process. */
    struct semaphore load_sema;
    /* 1 if load success, 0 otherwise. */
    bool load_success;
    /* Semaphore used for WAIT syscall synchronization, shared
with parent process. */
    struct semaphore wait_sema;
    /* Exit state of the thread. */
    int exit_state;
    /* 1 if already called on wait by parent process, 0
otherwise. */
    bool wait;
#endif
    /* Original members... */
  };
```

In ‘syscall.c’ :

```c
static struct lock my_lock;
```

Used for acquire lock and release lock when we do actions about file.

```
struct my_file_struct
{
  int fd;                /* File descriptor. When we want to find file,
                          we use fd to locate. */
  struct file *file;     /* File struct. Save the file here. */
  struct list_elem elem; /* Use for constructing a list. */
};
```

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

In our implementation, file descriptors are unique just within a single process. Each process holds a list of files it owned, and the 'fd' in struct 'thread' is the next available fd for the process. In the code, 'my_file_struct' is the way we associate file descriptors with open files. When opening files, we create a 'my_file_struct', save the next available fd of the process into this struct, save the 'file' as well, and push it into the list that we already constructed of the process.

---- ALGORITHMS ----

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

Read:
We firstly check whether all the pointers are valid, and check the buffer because it' s a pointer to a pointer. If it is not valid, we call exit the process with error code -1. After validating all the pointers, if fd is 0, it reads from the keyboard using input_getc(). Else, we read size bytes from the file open as fd into buffer. First we find file using fd, then acquire the lock, reads file, release the lock, and return the number of bytes actually read, or -1 if the file could not be read.

Write:
We firstly check whether all the pointers are valid, and check the buffer because it’ s a pointer to a pointer. If it is not valid, we call exit the process with error code -1. After validating all the pointers, if fd is 1, we write to console. Else, we write size bytes from buffer to the open file fd. First we find file using fd, then acquire the lock, write to file, release the lock, return the number of bytes actually written, which may be less than size if some bytes could not be written.

>> 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?

4,096 bytes: Least number is 1 and greatest number is 2. If the data buffer cross two pages, we need to check it at least twice to determine whether it belongs to this process. 4096 bytes can fit one page so the least number of inspection is 1.

2 bytes: Least number is 1 and greatest number is 2. If the data buffer cross two pages, we need to check it at least twice to determine whether it belongs to this process. 2 bytes can fit one page so the least number of inspection is 1.

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

Each process has a semaphore with its parent process. When wait () is invoked, the parent process will call sema_down () on that semaphore and when the child process exit () sema_up () will be invoked. The order of wait () and exit () doesn’ t matter because semaphore is used instead of lock and semaphore ensure one process can continue until other is done. So wait () on terminated process just return immediately. If wait () is invoked with a TID not in list children return -1 immediately.

>> 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.

Firstly we check the validity of the pointer to the pointer, if it is legal and can be accessed, then we check the pointer whether it' s valid or not. When across errors, we call a special exit, which saves error code into the process' s 'exit_state' and then call 'thread_exit' . During the exit of the thread, we close all the files opened by using the list that we have already constructed, and free all the resources we allocated, ensuring we exit in a correct way. For example, during the progress of system call 'write' , if error happens when we check the validity of the pointer to the pointer 'buffer' , we immediately call special exit with error code -1, when exiting the process, we do several operations to correctly free resources we have allocated to ensure the system run without memory leak.

---- 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"?

The parent and child process share a load semaphore, and when load is finished, the bool load_success is in the child TCB will be set and sema_up () will be invoked. The parent can get the child process from children list and know whether it load successfully by checking load_success.

>> 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, wait () and exit () are implemented by semaphore. To insure process P can get information of C after C exit, the TCB of C is not freed immediately after it exit. Instead, when a process exit, first it need to check all its child process and free their resource if they have exited and then check whether its own parent has exited, if true, free its own TCB, if false, do nothing. All other resources such as page table and files are frees immediately when a process exit.

In this example, if P exits first, P will free P’ s TCB and other resources when P exit and C will free C’ s TCB and other resources when C exit. If C exits first, C will free C’ s other resources but not TCB when C exit and P will free P’ s and C’ s TCB and P’ s other resources when P exit.

---- RATIONALE ----

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

We verify the validity of a user-provided pointer, then dereference it, because it’ s much easier to get information about page fault. The other way using ‘get_user’ and ‘put_user’ needs support of a more complicated structure of processor, called MMU, and we know little about it, so we choose a simple and efficient way to achieve the validation and access user memory from kernel.

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

Advantages:

1. The struct thread occupy very small space, because we maintain a list in the struct thread, but not directly save all the files and fds into it.
2. The number of open files are not limited as soon as we still have memory.
3. It’ s more flexible because the kernel can be aware of all the opened files.

Disadvantages:

1. Although the thread occupy the least space, the burden is put on the kernel, opening too many files could affect kernel’s utility.
2. We maintain a list, meaning each time accessing the element costs O(n) time complexity, we could use hash or other advanced data structures to decrease the time.
3. If we exit the process, the list is deleted and all the space are freed, so it has little flexibility.

>> 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 did not change it for simplify and it is possible in Pintos because each process can have exactly one thread. Usually modern OS will not do the identity mapping to support multiple thread for one process.

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?