```
+--------------------------------------------------+
| CS 521                                           |
| PROJECT 2: USER PROGRAMS                         |
| DESIGN DOCUMENT                                  |
+--------------------------------------------------+
```

---- GROUP ----

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


```
yuxiang liu <yliu268@buffalo.edu>
minghua wang <minghuaw@buffalo.edu>
shaoming xu <shaoming@buffalo.edu>
```


---- PRELIMINARIES ----

>> If you have any preliminary comments on your submission, notes for the
>> TAs, or extra credit, please give them here.

>> Describe briefly which parts of the assignment were implemented by
>> each member of your team. If some team members contributed significantly
>> more or less than others (e.g. 2x), indicate that here.

FirstName LastName: contribution
FirstName LastName: contribution
FirstName LastName: contribution

>> 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.

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?

In process_execute(const char *file_name) function, the *file_name contians the arguments from user, so we use the *file_name and the start_process() function to create a new thread.

The start_process(void *file_name) is the thread function. And the *file_name is exactly the same as that in process_execute() which contains the arguments. We use strtok_r to get argument.

Let's take command: " /bin/ls -l foo bar" as example. First, we parse the real_file_name(/bin/ls) and point the real_file_name to it. Then we try to load the the ELF executable from FILE_NAME into the current thread. if not success, we free the page which save the command from user. Otherwise, we start to parse the rest argument and to push them into stack.

First, we allocate a page with all zero to char **arg_p which is to save all parsed arguments. In the example, " /bin/ls -l foo bar", the *arg_p will save the /bin/ls first, and make argc to be 1, then we iterate through the rest arguments, we parse them, increase argc by one, until the last argument. Now **arg_p has all the parsed commands.

Second, we use argc to iterate the **arg_p in reverse order to get and push commands into the if_.esp(Saved stack pointer). After that we use a while loop to make the word-align to round the stack pointer down to a multiple of 4 for performance purpose. In the example " /bin/ls -l foo bar", we push "bar\0", "foo\0", "-l\0", "/bin/ls\0" in order into stack then make a word-align 0 on the top of the stack.

Third, we start push the address of each string plus a null pointer sentinel, on the stack, in right-to-left order. In our program the address of argv[] can be got by *(arg_p + 4 * i), i >= 0 && i<= argc - 1. So First we make argv[argc] to be zero and push on the stack then we iterate through arg_p and push all the addresses in right-to-left order. after that, we push the argc on stack.

Last, we push the fake "return address" in the stack.

---- RATIONALE ----

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

The user command is been saved in char *file_name for the process_execute() function. The *file_name include the command line and arguments. By using the strtok_r(), we can seperate the command line and the arguments. In our case, the *real_file_name save the command line get from strok_r(), than arguments will still be saved in *save_ptr that can be used later.

>> 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 Unix approach allows us to check the command before submitting it to kernel.For example, we can check whether the command length are over limit, the existence of executable file, and so on. To do the pre-check, we can avoid the kernel fail so that to

reduce the waste on creating the new thread, the new page and other new resources for the wrong commands.
Shell does the separation allows us to do preprocess on the command lines which allow the programmer to do the shell programming. For example, the programmer can write the command line like “command_1; command_2”, by using the Unix approach, we can separate this command line to make it legal.

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 process.c

```
/* Lock for allocate pid */
static struct lock pid_lock;
/* Lock for filesys*/
static struct lock filesys_lock;
```

In thread.h

```
#ifdef USERPROG
   /* Owned by userprog/process.c. */
   uint32_t *pagedir;                 /* Page directory. */
   bool is_userprog;                  /* Make diff between kernel and userprog */
   char *process_name;                /* Name of process, print when exit */
   pid_t pid;                         /* Process identifier */
   struct hash* tidmap;               /* Save all child's tid */
   struct hash* pidmap;               /* Save exec success child's pid */
   int exit_status;                   /* Exit status */
   int fd_base;                       /* Allocate different file id */
   struct hash* fdmap;                /* Map from fd to file pointer */
   struct file* file;                 /* Deni writing */

#endif
```

In tidmap.c

```
struct tidmap_entry  /* to contruct the hash_elem */
{
  tid_t tid;
  struct hash_elem helem;
};
```

In processinfo.c

/* List for tell exec program that callee with tid = ** has pid = ** */

```
static struct hash hash_process_infos;
static struct lock process_info_lock;
struct process_info
{
  tid_t tid;                          /* Identify process */

  pid_t pid;                          /* Data for syscall_exec() */
  bool finish_pid;
  struct condition pid_cond;

  int exit_status;                                        /* Data for syscall_exit() */
  bool finish_exit;
  struct condition exit_cond;

  struct hash_elem helem;             /* Use for hash map */
};
```

In processinfo.c

```
/* List for tell exec program that callee with tid = ** has pid = ** */
static struct hash hash_process_infos;
static struct lock process_info_lock;
struct process_info{
  tid_t tid;                          /* Identify process */

  pid_t pid;                          /* Data for syscall_exec() */
  bool finish_pid;
  struct condition pid_cond;

  int exit_status;                                        /* Data for syscall_exit() */
  bool finish_exit;
  struct condition exit_cond;

  struct hash_elem helem;             /* Use for hash map */
};
```

In pidmap.c

```
struct pidmap_entry
{
        pid_t pid;
        tid_t tid;
        struct hash_elem helem;
};
```

In fdmap.c

```
struct fdmap_entry              //HashMap entry to make fd as the key the file as the value.
{
```

```
    int fd;
    struct file* file;
    struct hash_elem helem;
};
```

>> 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 syscall_open(struct intr_frame *f), if the file_name is valid, we will call struct file* file = filesys_open (file_name). Then we will set the thread's fd_base++ as the file descriptor fd. The initial value of fd_balse is 2 .Finally, take fd as the key, file as the value and save them in thread's fdmap hashmap. Here, we can see the file descriptors are just within a single process. Because the thread's fdmap hashmap, each time we get the file descriptor, we can use it to get file pointer so that to associate the file descriptor with open files.

---- ALGORITHMS ----

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

In our code, the function syscall_read (struct intr_frame *f) deals with reading and the function syscall_write (struct intr_frame *f) deals with writing.

Since the both them needs int fd, void *buffer, unsigned size as parameters, we use same code to extract the parameters from intr_frame *f.

By using the rule that each system call argument takes up 4 bytes on the stack, we extract the pointers from intr_frame and check if the pointers are valid or not one by one. We use the first method mentioned in 3.1.5 Accessing User Memory by implementing two function: check_user_vaddr (void *vaddr, size_t size) and check_user_vaddr_single (void *vaddr) in syscall.c. The key idea of the method is to detect the wrong pointers -- the null pointer, the pointer to unmapped virtual memory, or the pointer to kernel virtual address space.

If the pointers are valid, we start implemeting reading and writing.

There are two cases in reading. First, reading from keyboard which fd == 0. The other is reading from file.

For reading from keyboard, we iterate through the size to check if buffer pointer is valid, get input from input_getc(), save it in buffer and increase the count by one. Finally save the count in f->eax. The buffer pointer pointed by (void **)f->esp+8. So we use the intr_frame to save the data which can be used later.

For reading from file, First we get file from a hashmap which can use the fd to get the struct file* file. if the file == NULL, return -1. Otherwise, we set a lock on the file, read data from file, release the lock, and save the result in f->eax.

Same as reading, writing also has two cases. The first is writing to console which indicated by hd == 1, the other is writing to file.

For writing to console, we just calls putbuf(buffer, size), the save size in f->eax.

For writing to file, we get struct file* file by calling fdmap_get (thread_current()->fdmap, fd);. Then for file == NULL case, we save -1 in f->eax. Otherwise, we set a lock, use int result = file_write (file, buffer, size), release the lock, and finally set f->eax as result. Here the f is the pointer for struct intr_frame.

>> 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 is 1.

The greatest number is 4096.

For the system call that only copies 2 bytes of data, the lease number 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.

Same as question B3, first we extract data from struct intr_frame *f by increasing f->esp by 4 each time and check if all pointers valid by calling check_user_vaddr() function. If the first step goes well, then we use the pid as the key to find the tid in thread’s pidmap Hashmap and check if the tid is valid or not by checking if it is not equal to TID_ERROR. If the tid is valid, we call process_wait(tid).

In process_wait(tid), if the current thread’s tidmap Hashmap contains the tid, then we call get_exit_status(child_tid). The get_exit_status(tid_t tid) function will lock the struct process_info search_key, get the process_info* value, and a while loop to check if the process is finish by checking the value->finish_exit. If not, call an monitor’s cond_wait() to make the current thread wait. If the target thread is finished, we release the lock and return the process exit_status. When the target thread is finished, it will call process_exit(void) function and signal the waiting thread. So that the waiting thread can out the while loop.

>> 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.

Take the “write” system call as the example. As the description mentions, it requires four data from user stack, and we need to make sure all the address of these data are valid. In question B3, we have discussed the key idea deal with corrupt pointers and the two functions -- check_user_vaddr (void *vaddr, size_t size) and check_user_vaddr_single (void *vaddr) in syscall.c -- to avoid obscuring the primary function of code in a morass of error-handling.

When the error is detected, we will call thread_exit(). In thread_exit(), we will free that user process’ page, close its file, destroy its page directory, and destroy its all hashmaps. By doing so, we ensure all temporarily allocated 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"?

After get the char *cmd_line from the intr_frame and make sure the cmd_line is valid. we call tid_t child_tid = process_execute (cmd_line) which will call thread_create() and start_process() two function to make the child process. In start_process() function, the new executable starts loading. The preocess_execute can’t not return tid for syscall_exec() until the loading finished which makes sure the ”exec” system call cannot return before the new executable has completed loading.

In start_process (void *file_name_), if the load failure, we will call update_pid(t->tid, -1), otherwise we will call update_pid(t->tid, t->pid). t denotes the current thread. Then in syscall_exec(struct intr_frame *f), we can call get_pid(child_tid) to get the pid we just update. By doing the things above, we make the load status passed back to the thread that calls "exec".

>> 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?

int **wait** (pid_t pid)() pid is the child process pid.

First let’s consider the P calls wait(C).

If P calls wait(C) before C exits, first we use the pid to get C’s tid, then call process_wait(tid) and it will return the exits status of C’s tid and save it in result. In process_wait, we return the get_exit_stauts (child_tid).The get_exit_stauts(child_tid) will check if process of that tid is finished or not by checking its finish_exit value which is boolean type and true means the process is finished. If it is not finished, we use a monitor to wait for the exit_cond in a while loop.

When the C exits, it will call singal_exit_status(cur->tid), which will change the the finish_exit to be true and signal the exit_cond so that the get_exit_stauts(tid_t tid) can leave the while loop and return the exit_status.

If P calls wait(C) after C exits. Now the result of checking process' finish_exit is true which means the child process has exits, the parent process does not need to wait inside the monitor.

Second let's consider the P terminates without waiting.

If it happens before C exits. when the P terminates, it will call process_exit(void) function and which will release all resources including pages, file, pagedir, tidmap, pidmap, and fdmap. It's means the information of C kept by P has gone when P terminates. And when C terminates, it also will release all its resources, so that the system is clean.

If it happen after C exits. Same as before C exits, P will release all resouces. Because the C exits before P exits, C has release all its resources.

The key idea here is the data structure that store the pid and tid are separate with the thread, so we can easily release one's resources without affecting the other thread.

---- RATIONALE ----

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

We choose the first method -- to verify the validity of a user-provided pointer, then dereference it -- because it is the simplest way to handle user memory access. Compared with the second method, implementing first method we can save time, avoid bugs, and make code much more readable. All these are good to make the system reliable and extendable in future. Furthermore, it a infant system, there are so many things we can do to speed the performance up. For example, we implement the hashtable to save many resources which is much faster than LinkedList. We think make more time on these staffs can make more benefits in the initial phase on designing operating system.

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

Advantage:

1. Fast. Hashmap is faster than LinkedList.
2. flexible. Kernel can easily get the file by file descriptor and manipulate the file.

Disadvantage:

1. Hashmap consume more space than linkedlist.
2. If processes open too much files, the kernel space may be exhaust.

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

The main advantage is fast, we use the hashmap mapping which can do the mapping in constant time.

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?