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

---- GROUP ---

Name	email
Xiaojing Hu	xhu7@buffalo.edu
Jingyi Zhao	zhao.elton@gmail.com
Zishan Liang	zsliang@buffalo.edu

Reference:

- ***Pintos userprog*** https://wenku.baidu.com/view/8569d160581b6bd97f19eae8.html
- ***Waqee, pintos project2***: https://github.com/Waqee/Pintos-Project-2
- **Into the OS project 2: Preliminary design and general strategies** https://wenku.baidu.com/view/b6ec9a60783e0912a2162acc.html?from=search
- ***Argument Parsing* - general strategies:** https://wenku.baidu.com/view/8296f9629b6648d7c1c74649.html?from=search

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?
>> How do you implemented argument parsing?

Ans:
First, we need to get the filename and the arguments. After we
parsed the command line string, we will get a string combined
with the filename and the arguments. Then we pass it to function
thead_create() as the threadname.Then in function start_process()
we pass it to load(), and in load() we pass it to setup_stack().
In setup_stack() we create a stack for the user program. Then in
load() we get the stack pointer saved in if_.esp. Then all we need
to do is to parse out the arguments and push them into the stack.
We implement a function argument_passing() to do this. We pass
the stack pointer and the string we mentioned above to
argument_passing(). Then in this function we make a copy of the
string and parse out every argument using function strt_okr().
And we parse out the arguments and save the number of them into
a variable argc. The we declare a variable argv to save the address
of the arguments. Then we do word align. After that we push a
sentinel 0 into the stack. After that we push the addresses of the
arguments into the stack. Then the addresses of argv[0] and argc
are pushed. Finally we push the return address 0 into stack.
The in load(), after we set up the stack by calling setup_stack(),
we push the arguments into stack by calling argument_passing().

>> How do you arrange for the elements of argv[] to be in the right order?

Ans:
When we push the addresses into the stack, we push them in reverse order.

>> How do you avoid overflowing the stack page?

Ans:
Not implemented.

---- RATIONALE ----

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

Ans:
There is an argument *save_ptr that points to the remaining string,
which we can make of further use. With *save_ptr, it is safer for
thread transfer because we save the remaining string in *save_str
to protect it being modified by other threads.

>> 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.

Ans:
The shell is a user-level process. By using the shell, we spend less time in the kernel. Also, the shell can do many additional checkings to avoid kernel fail. So it is safer to implement command line parsing in the shell.

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.

Ans:

```
/* In thread.h */
struct thread {
    ...
    #ifdef USERPROG

    uint32_t *pagedir;                  /* Page directory. */
    struct file *self;
    struct list process_files;        /* List of files opened by the process */
    struct thread *parent;
    struct list child_process;
    bool load_success;               /* Flag of loading */
    int fd_count;                       /* Counting file descriptor */
     tid_t waiting_for_t;
    struct semaphore load_process_sema;    /* Semaphore of loading */
    struct semaphore wait_process_sema;    /* Semaphore of waiting */

    struct p_info {                      /* Process Information used to track the value */
        tid_t tid;
        int return_record;
        bool is_over;                  /* Signal to indicate the process's status */
        bool is_parent_over;        /* Decide whether quit directly or return value to parent */
        struct list_elem elem;
    };
    struct file_info {
        struct file* target_file;
        int handle;
        struct list_elem elem;
    };
    #endif
    ...
};
```

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

Ans:
In short, file descriptors are integers, known and used by user program.
Kernel needs to know what number is associated with what file.
Specifically, for example, user program α, opens file "input.txt" as file No.1
and reads it and writes to file "output.txt", which it has opened as No.2.
Now for every access to any of these, you don't want to call them by
their filename, open and close them. so α opens "input.txt" as No.1, and
every time it accesses it as No1. So it only needs to say to kernel
(call a syscall): read from No.1, seek in No.1, write in No.1, ... close No.1.
Kernel should tell which file is No.1 referring to, and then use that file* to
call the read, seek, etc.
The file descriptors has a one-to-one mapping to each open file through
Syscall. So it is unique within the entire OS. Besides, each file descriptor
Owned by a thread.

---- ALGORITHMS ----

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

Ans:
Every time we call the system call, we will check whether the buffer pointers
are valid at first through the function “confirm_user_addr()”. If not, quit and
return the error signal “-1”.
If valid, we still need to confirm the the address of the input argument. Like
file descriptor, handler and the address that the buffer pointed to.
We will acquire and release the file system lock through the “try_open_file()”
function and the current process holds it.
And then we check the case of file descriptor (STDOUT_FLIENO and
STDIN_FILENO) to determinate whether it is a standard output or input and

return signal after I/O execution. If file descriptor (*(p+1)) is not equal to STDOUT_FLIENO or STDIN_FILENO, we search the handle in “process_files” list through function “look_up()” and then execute the read/write for that file and return value to EAX register.

>> **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?

Ans:
Obviously, the least number will be 1 in both case. A system call causes a full page of data to be copied from user space into the kernel. The worst situation is that each bytes is distributed to different frames, then it needs 4,096 number of inspections of the page table. So 4,096 is the greatest possible number.
Similarly, the greatest number of coping 2 bytes of data from the system call is 2.
Improvements: Adopt the first-fit method to choose holes in order to take more advantage of the address space, but it still faces the external fragmentation problem. Put the reentrant code into the sharing page.

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

Ans:
The wait system would be implemented in the following approach:
Each process has a parent process (thread *parent), a list of child process (list child_process), a waiting child process (waiting_for_t), a semaphore (wait_process_sema).
Whenever a parent process calls process_wait(), it search the list of child_process to see whether there is a child process executing.
If not, just return “-1” to the EAX register.
If a child process exit, store its tid to the parent’s “waiting_for_t” and wait for the child process’s over. If the child process is not over yet, set semaphore (wait_process_sema) down.
After that, remove its tid from the current element list and return the child process’ “reutrn_record” value to EAX register.

>> **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.

Ans:
We avoid the bad user memory access by checking if the user pointer is below PHYS_BASE through the “confirm_usse_address()” function. And make sure to release the lock or free the page of memory under a safe condition. The function is_user_vaddr() in threads/vaddr.h is where we implement the verification. And we also check whether is mapped or unmapped though the function pagedir_get_page() in userprog/pagedir.c. By this approach, it will runs normally faster for it takes advantage of processor’s MMU.
Take “write” system call as example, the esp pointer and the arguments will be checked at first and it will be terminated if the pointer is invalid. Then we will check the size of the buffer to make sure the upper and lower bound are still in a valid range.
However, it is hard to handle an invalid pointer which is occur after the user pointer verification. And that’s why we still need to modify the fault handler in the userprog/exception.c. We check whether the fault_addr is valid pointer. If it’s invalid, we let the process return status and then terminate the process.

---- 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"?

Ans:
We first check if the pointer passed by system handler is a valid one.
The exec system call would parse in the file name. We try to create a

currFile that is of struct *file. If this currFile cannot be loaded by filesys_open, we are certain this char *file_name is an invalid input, and we return -1. We have an eax defined in intr_frame handler. The value of eax is updated with -1 or correct tid if the execution loads successfully.

>> **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?

Ans:

Each thread has a parent thread, and a list of **child_process** to store the child process being opened during the main process. Each thread has a **tid waiting_for_t** representing if this thread is waiting for the completion of another thread/process. Each process has a **bool is_over** indicating if this process is finished, and a **bool is_parent_over** indicating if parent process is over. We use 2 semaphore to avoid race conditions**. semaphore load_process_sema** for load, and **semaphore wait_process_sema** for wait.

- P calls wait C before C exits :

    P calls process_wait. If C has not exit yet, it's bool is_over would show this status. P would call a semaDown on its **wait_process_sema**.If C is over, its status would be captured during process_exit. The parent's **wait_process_sema** would be updated using semaUp, allowing P to exit. Everytime after process_wait(child process) is called, we remove/free this instance of C from P's **list child_process** to avoid P being stuck on multiple same C.

- P calls wait C after C exits

    When C exits, it would first check if P is alive or not according to **is_parent_alive**. If P is alive, it would semaUp P's wait_process_sema in our quit(int status) function before process_exit. Then, P calls process_wait, and discover C is over. P just go forward to remove/free C from its **list of child_process.** If P is not alive, C would just free itself.

● P terminates before C exit, no waiting

When P calls quit() and process_exit(), it first check if there is any alive parent of P (PP) that is waiting for P. SemaUp its parent's wait_process_sema if PP is not over. Or directly free P itself if P is over. P free all its resources and C would execute normal unaffected.

● P terminate after C exits, no waiting

When C exits, it would update its return value that is stored in P's list_of_child_process. Then P would normally free all lists, close all files, and output exit message.

---- RATIONALE ----

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

Ans:
We validate memory pointers parsed by user before performing them in SysCall. We use functions available in Pintos: is_user_addr & is_kernel_addr. By definition, a valid user memory pointers should be both present in user_addr and kernel_addr (beyond PHYS_BASE). It is a easy method.

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

Ans:
1) Advantages:
(1) It can minimize the struct space of threads.
(2) Besides the opened files can be easily manipulated.

2) Disadvantages:
(1) It will occupy more kernel space which may lead to the overflow when there is a big number of the opened 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?

Ans:
The identify function is quite intuitive, we think 1 to 1 mapping may takes more space. However, according to the Moore's Law, the technique of hardware allow us to ignore the space consuming problem except some specific occasions. Maybe we will trying to use hashmap to resolve this issue. HashMap allows a list of multiple tid_t to share a same pid_t. It would save some memory, but I don't think it is so necessary.