```
+--------------------------------------------+
|                  CS39002                   |
| PROJECT 2: USER PROGRAMS  |
|         DESIGN DOCUMENT           |
+--------------------------------------------+
```

---- GROUP ----

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

Aayush Prasad 18CS30002 aayuprasad@gmail.com
Rajdeep Das 18CS30034 rajdeepdasren@gmail.com

---- 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.

In thread.h file, line number 97 onwards

In struct thread, the following data members were added:

bool success; /* A flag to reflect parent process that child process has been loaded successfully */
int exit_error; // Exit code of the thread/process It will be used for printing the exit message
struct list child_proc; // List of child processes in the form of struct child defined in process.h
struct thread* parent; // To keep the thread of the parent process
struct file *self; //to store the executable file for the process
struct list files; // to store the files required for running the process
int fd_count; // count of file descriptors associated with the thread
struct semaphore child_lock; // semaphore to ensure synchronization between child processes
int waiting_on; // stores thread id of the thread the process is waiting on

A new structure was added with the following definition

```
struct child {
        int tid; /* The thread's tid.*/
        struct list_elem elem; /* List element for a parents children list.*/
        int exit_error; /* The thread's exit status. */
        bool used;
};
```
Then child structure is added to store the information of child process example, the thread id,
In syscall.c, a structure was added with the following definition:
```
struct proc_file {
        structure file* ptr;
        int fd;
        struct list_elem elem;
}
```
The proc_file was created to store the information related to the file like file pointer, file descriptor.

---- 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?

First, we're breaking the command into words. Then, placing the words at the top of the stack. Order doesn't matter, because they will be referenced through pointers. Then, pushing the address of each string plus a null pointer sentinel, on the stack, in right-to-left order. These are the elements of argv. The null pointer sentinel ensures that argv[argc] is a null pointer, as required by the C standard. The order ensures that argv[0] is at the lowest virtual address. Word-aligned accesses are faster than unaligned accesses, so for best performance round the stack pointer down to a multiple of 4 before the first push. Then, pushing argv (the address of argv[0]) and argc, in that order. Finally, pushing a fake "return address": although the entry function will never return, its stack frame must have the same structure as any other. We avoid overflowing the stack page by performing a check on the total size of the args being passed. If it would overflow the stack page size, we exit.

---- RATIONALE ----

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

The difference between strtok_r() and strtok() is that the save_ptr (placeholder) in strtok_r() is provided by the caller. In Pintos, the kernel separates commands into executable name and arguments. We need to make sure that if there are more than one thread calling strtok_r(), each thread should have a pointer (save_ptr) which is independent from the caller. So, we need to put the address of the arguments somewhere we can reach later.

``>> A4: In Pintos, the kernel separates commands into an executable name
>> and arguments. In Unix-like systems, the shell does this
>> separation. Identify at least two advantages of the Unix approach.

1- It seems cleaner to separate the executable name from the arguments before passing it off to the kernel, since they represent different things. It shouldn't be the kernel's job to parse that, as it can also be done by a user program easily.

2- Perhaps some validation of the input could be done by the shell more safely than by the kernel. If someone entered a very large amount of text, perhaps it would cause the kernel a problem if the kernel tried to parse it, whereas if the shell takes care of it, the worst case is the shell crashes.

3- it can separate the commands, it can do advanced pre-processing, acting more like an interpreter not only an interface. Like passing more than 1 set of command line at a time

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.

>> B2: Describe how file descriptors are associated with open files.
>> Are file descriptors unique within the entire OS or just within a
>> single process?
When a file is opened by calling the open system call it gets added to the current process's list of open files and assigned a unique file descriptor. So, file descriptors are unique within a single process only. Also, when a file gets closed its file descriptor is freed and it can be reused by the system later for another file.

---- ALGORITHMS ----

>> B3: Describe your code for reading and writing user data from the
>> kernel.
In read, if we are supposed to read from STDIN, we call input_getc Otherwise, we get relevant file pointer using passed fd from the file_list of current thread. Filesys lock is acquired, filesys function file_read is called and then lock is released.

In write, if we are supposed to write to STDOUT, we call putbuf. Otherwise, we get relevant file pointer using passed fd from the file_list of the current thread. Filesys lock is acquired, filesys function file_write is called and then lock is released.

>> 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?

There are a couple of methods for copying data from user space into the kernel. The first is if it is stored as an ELF executable (possibly as another file type, but ELF is the only one implemented at the moment). There is an inspection of the page table required every time a new page is allocated. The least number of possible inspections with 4,096 bytes is one inspection in the case where a segment size for ELF in memory is greater than or equal to the segment size in files. The page size also needs to be large enough (In this case it is). The greatest number can be 4,096 number of inspections, if each ELF segment is only one byte in size i.e. 1 inspection for each byte.

For 2 bytes the least number of inspections is also 1. The greatest number of inspections can only be 2 (no of bytes).

There can be room for improvement on these numbers because segments do not have to be loaded into their own pages, many segments could fit inside a page if the page size is much greater than it. For example, if the page size was 2 bytes and the segment size (on file) was 1, then instead of loading each segment into its own page and zeroing the other byte, they could share the same page. This would result in half the number of inspections of the page table.

>> B5: 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.

Each system call first tries to get its arguments from the stack and check that they are valid. After this happens the system calls continues execution if everything is fine or fails/returns an error value if not. No resources such as locks or buffers are allocated before all the system call arguments are verified.

When either a page fault occurs from bad memory access, or malloc fails, the process exits via kill_process and thread_exit. Inside process_exit there are semaphores which either allow for freeing of some structs or flagging it for later freeing if it is being used by another process.

For example, when a process tries to access an unmapped piece of memory, get_user is called, which will attempt to access the memory and cause a page fault. The page fault handler will simply return to get_user, signaling the fault. Process_exit is then called and any pointers to structs or lists of structs will be attempted to be freed or flagged for deletion. The thread will set its status to dying in the threas_exit call, and then the thread struct itself will be freed by the kernel in schedule().

---- RATIONALE ----

>> B6: Why did you choose to implement access to user memory from the
>> kernel in the way that you did?

We implemented the safe access to user memory the way we did because it is faster to signal a bad pointer via a page fault than checking if the pointer is NULL because it utilizes the MMU. Of course it is not faster if the pointer is actually bad because the page fault interrupt will slow down the computer a lot more. But it leads to better performance in general because this check does not need to be performed all the time, and if the pointer does cause a page fault then the thread will have to exit anyway so it does not matter if it is slower in this case.

>> B7: What advantages or disadvantages can you see to your design
>> for file descriptors?

We see simplicity of the code and the fact that a process can have as many files open as it wants (well not really but it is not limited because of file descriptor design) as an advantage.

There is a disadvantage of quite costly lookups for internal translation of file descriptors to file structs. The whole list of open files needs to be searched through to find a struct with a given file descriptor.

We thought about this and one other possible solution came to our minds. If we imposed a limit on the maximum number of open files by a process (which would be a disadvantage compared to the current state) we could easily use arrays and do file descriptor-based lookups in constant time by using array indices. However, setting this limit too low would be restricting and setting the limit too high would require allocating memory for the array which would be wasted if there were only a few open files. That is why we preferred our solution to this one.