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

---- GROUP ----

>> Fill in the names and email addresses of your group members.
Rahul Aditya 18CS30032
Abhishek Srivastava 18CS10068

---- PRELIMINARIES ----

Most of the things like argument passing and system call handler were implemented in the previous assignment itself. In this assignment we simply completed the remaining system calls.

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 file_struct in threads/thread.h
(Newly created)
(Stores all the details pertaining to files opened by a process)

```
int fid; /* File descriptor number */

struct list_elem thread_file_elem; /* This is to store the file in a list */
struct file *f; /* A pointer to the file in the filesystem */
```

struct thread in threads/thread.h
(Modified)

```
bool process_exited; /* A flag to notify that process has exited*/
bool process_waited; /* A flag to notify that process has been waited already by parent*/


struct semaphore sema_process_load; /* synchronize process loading */
struct semaphore sema_process_wait; /* synchronize process waiting*/
struct semaphore sema_process_exit; /* synchronize process exit */
```

```
struct thread *parent; /* to store the thread that created the process */
struct file *exec; /* to store the file that is being executed by process */
struct list files;. /* to store list of opened files */
int ret_status;. /* to store status returned by process on exit */
```

>> 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 process opens a new file, the OS generates a new file descriptor number. A pointer to the file is stored in the process data (struct thread -> struct list files), along with this file descriptor number(fd).

The file descriptor number is unique with a process, not globally..

---- ALGORITHMS ----

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

First of all, we check if the memory that the process is trying to access is in user space or kernel space using the ‘is_user_vaddr()’ function. If the memory is outside the user space, exit with status -1.

Next the source of reading/writing is determined, using the file descriptor number. If it is 0 then STDIN, if 1 then STDOUT, else some other file.

If the operation is to be done on a file, we obtain the file pointer using the find_file_by_fd function in userprog/syscall.c. Then the read/write syscalls are used to perform the required actions, which in turn use the file_read() and file_write() functions present in filesys/file.h

>> 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?

First we need to verify that the address pointed by both the page beginning and end belong to the user space and are valid. In the worst case, the full page data can span over two pages where two checks would be necessary.

A better solution would be to implement the validation inside page_fault but for this every reference needs to be mapped. This is faster because detection is done by MMU hardware.

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

The validity of the parameters is checked first. We also check if the process that the process is waiting for is in fact its child. This prevents waiting for a child of some other process.

We have initialised a process_wait semaphore to 0. Any process waiting for one of its children basically waits for this semaphore.

Upon exiting the child increases the value of this semaphore to the number of processes that are waiting for this semaphore. Thus, each of them can now proceed, having waited for the child to complete.

>> 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.

When a process tries to access a bad location there is a page fault and the page fault handler calls the exit function with a status -1. This exit function frees all the locks and closes all the files that the exiting process had opened.

For example, consider the page fault handler code :

```
if ((is_kernel_vaddr(fault_addr) && user) || not_present)
        exit_syscall(-1);
```

And the code of the exit function:

```
if (lock_held_by_current_thread(&file_lock))
        lock_release(&file_lock);

while (list_empty(&current_thread -> files) == false) {
        Struct list_elem* element = list_begin(&current_thread -> files);
        close_sycall(list_entry (element, struct file_struct, thread_file_elem)->fid);
}
```

It can be seen that all resources are appropriately closed after use.

---- 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"?

We use a semaphore when we try to load an ELF file and push the arguments in the stack.
We wait on the semaphore when the process thread is created.
The semaphore is only signalled when the loading of the function is complete.

>> 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?

When P calls wait(C) before C exits, P gets blocked on C -> process_wait(semaphore) on which every process waiting for C gets blocked.
When C exits before P calls wait(C), C gets blocked on C -> process_exit(semaphore) until P or any process calls wait(C).

---- RATIONALE ----

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

We used the is_user_vaddr() function to access the user memory from the kernel. We wanted to implement the page fault mechanism.

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

We used a struct file to handle opened files. The file descriptor for a file was a part of this struct. This allowed the system to access the file, and file descriptor simultaneously. Using the find_file_by_fd() function we could get the file pointer from the file descriptor. This made things very simple.

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

No modifications were made in this regard.