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


---- GROUP 22 ----

(18CS30047) Somnath Jena <somnathjena2011@gmail.com>

(18CS10069) Siba Smarak Panigrahi <sibasmarak.p@gmail.com>

---- PRELIMINARIES ----

1. We had to change the **shutdown.c**, according to Timeout in tests when running pintos.
   This led to the prevention of the TIMEOUT after a certain (generally 61) seconds) issue on qemu simulator.
2. There was an unusual issue while reading the arguments from the stack in the syscall handler due to incorrect pushing of arguments into the stack by the inline assembly macro (such as syscall3()). Reading the arguments in the syscall handler, thus leads to having repeated or lost arguments. But we observe, the arguments are correctly passed in the wrapper function, thus to obtain the correct argument, we had to go deeper into the stack (as arguments were correctly passed to the wrapper function). For **read()** and **write()** syscals (3 arguments) **we had to add 20 to the normal accesses** to the arguments in stack. For system calls with two arguments, such as **create()** we **had to add 16 to the normal accesses** to the arguments in stack.

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.

We have not added or changed any definition or variable for the implementation of argument passing.

---- 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 passing is mainly done inside load(), setup_stack() and setup_args() function in process.c.

Inside load():

When **start_process()** calls **load()**, it passes the complete command line (filename+arguments) to it. In load, **strtok_r()** is used to first extract the main filename which is the first token in the command line, using single space as a delimiter and **save_ptr** as the tokenizer. Then load() reads the program headers from the file if it exists. After this, load calls **setup_stack()** passing the address of stack pointer (esp), filename and the save_ptr as

the parameters.

Inside setup_stack():

The stack is set up for the user program by mapping a zeroed page on top of user virtual memory. After mapping from user virtual address to kernel virtual address, if it is successful, setup_stack calls **setup_args()** passing the stack pointer, filename and save_ptr as parameters. Else frees the allocated page.

Inside setup_args():

An array **argv** is defined for storing the extracted arguments and passing onto the stack. The tokenization runs in a loop using the **save_ptr** tokenizer. The initial value of **num_tokens** and **total_size** is 0. The following procedure ensures that the elements of argv are in right order.

In each iteration of the loop,

->the token is extracted

->the total_size variable (for storing the total number of bytes in arguments list) is incremented by the size of the token

->the stack pointer is decremented by the size of the token(plus one for the null character)

->the token is copied to the stack at the current location of stack pointer

->the num_tokens value is incremented

->token is then copied into argv

In this order the first argument resides at a lower memory address than the last argument. Outside the loop, the filename is passed onto the stack. The **filename** is kept in **argv[0]** and **null** is kept in **argv[num_tokens+1]**. The null pointer sentinel ensures that argv[argc] is a null pointer.

Then a padding of 0s is added to the stack to round the stack pointer to a multiple of 4 to ensure word-aligned addresses for better performance.

Then the char* pointers to all the tokens(including filename and null pointer) are copied to the stack using argv.

Then the pointer to argv (char**) itself is passed in the stack, followed by argc(represented by num_tokens + 1 here) and a fake return address.

To ensure we do not overflow the stack page, we verify that the length of input command line is no longer than **128 bytes**. If it is longer, then we report an error.

---- RATIONALE ----

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

The **_r** version of **strtok** is reentrant: we can call them from multiple threads simultaneously, or in nested loops, etc. Since it is a reentrant version it usually takes an extra argument **save_ptr**, this argument is used to store state between calls instead of using a global variable.

The non-reentrant version **strtok** often uses **global state**, so if we call them from multiple threads, we are probably invoking undefined behavior. Our program could crash, or worse. This is why strtok_r takes 3 arguments while strtok takes 2 arguments.

Thus, Pintos implements strtok_r() to enable it to be called from multiple threads with multiple states.

>> 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.

The advantages of the Unix approach are:

1. **Reduce Kernel Overhead:** If the separation of command into executable name and arguments is done in a shell, it reduces the burden on the kernel. Kernel is an important resource and we won't want the kernel to spend a large amount of resources every time a thread tries to execute a file. Simply speaking, there are a lot of kernel overheads, and we would not want to use kernel resources for tasks which can be done in shell.
2. **Security:** Another advantage is security. If the kernel handles separating commands into an executable name and arguments and if an error occurs during such an operation, it may lead the kernel to crash and may even corrupt the memory or disk. But if a separate shell handles this and some error occurs, that error would be local to the shell process only.
3. **Flexibility:** Another advantage is flexibility. If we want to extend the command line handling or use different methods for separating commands into executable name and arguments for different commands, we can easily do that by creating separate shell programs, rather than modifying the kernel every 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.

1. **In pintos/src/userprog/syscall.h**

   MACROS:

   #define ERROR -1

   #define USER_VADDR_BOTTOM ((void *) 0x08048000)

   Modifications/Additions:

   ERROR: to exit with error value -1 on invalid pointer reference inside syccall.c

   USER_VADDR_BOTTOM: bottommost virtual memory address of a user process in its stack

2. **In pintos/src/userprog/syscall.c**

   MACROS:

   #define MAX_ARGS 3

   #define STD_INPUT 0

   #define STD_OUTPUT 1

Modifications/Additions:

MAX_ARGS: maximum number of arguments required for a syscall, e.g. write()

STD_INPUT: standard input = 0

STD_OUTPUT: standard output = 1

3. **In pintos/src/threads/thread.h**

STRUCTURES:

```
struct waiting_thread {
        tid_t parent_tid;
        tid_t child_tid;
        struct semaphore wait_lock;
        struct list_elem waitelem;
        int exit_status;
         int exitted;
};
```

Modifications/Additions:

struct waiting_thread: It is a structure whose pointer to such a variable is used as shared memory between parent thread and child thread. Its members and descriptions are as follows:

parent_tid: thread id of parent thread which shares this memory

child_tid: thread id of child thread which shares this memory

wait_lock: it is a binary semaphore which is initialized to 0 during child thread's creation. It is used to send the exit status of a child thread to its parent thread. Only after the child thread signals this semaphore in its exit call that the parent thread can return from its wait if it is waiting for this child.

waitelem: list_elem type variable used as the list member of waiting_threads list inside parent thread. Parent thread maintains a list of all waiting threads.

exit_status: the status returned by a child thread after it exits

exitted: flag to indicate that the child thread has exited and then only exit_status is valid.

```
struct file_info {
        struct file* file;
        int fd;
        struct list_elem fileelem;
};
```

Modifications/Additions:

struct file* file: It is a pointer to an open file

int fd: file descriptor of the above file

struct list_elem fileeelm: list_elem type variable used as the list member of file_list inside the thread that opened this file with fd as file descriptor

```
struct thread
 {
        /* Owned by thread.c. */
        tid_t tid;                  /* Thread identifier. */
        enum thread_status status;          /* Thread state. */
        char name[16];              /* Name (for debugging purposes). */
        uint8_t *stack;             /* Saved stack pointer. */
        int priority;               /* Priority. */
        int nice;                   /* Nice value. */
        int recent_cpu;             /* recent cpu time. */
        struct list_elem allelem;           /* List element for all threads list. */

        int64_t wake_tick;          /* tick when a sleeping thread will wake up */
        struct list_elem sleepelem;         /* list element for all sleeping threads list */

        /* Shared between thread.c and synch.c. */
        struct list_elem elem;              /* List element. */

        /* Owned by userprog/process.c. */
        uint32_t *pagedir;          /* Page directory. */
        struct thread* parent;              /* parent thread */
        bool success;               /* for exec */
        struct semaphore load;              /* load semaphore */

        struct waiting_thread* child_shm;   /* waiting_thread shared memory of current thread as a
child */
        struct list waiting_threads;        /* list of waiting_threads of current thread as a parent */

        struct list file_list;          /* list of files */
        int fd;                     /* file descriptor to assign to new open files */

        struct file* exec_file;         /* executable file of this thread */
        /* Owned by thread.c. */
        unsigned magic;             /* Detects stack overflow. */
 };
```

Modifications/Additions:

struct thread* parent: pointer to parent thread of this thread

bool success: success value returned to parent thread on load

struct semaphore load: binary semaphore used in exec. It is signalled by child thread only after its load

is complete (maybe successful or an unsuccessful load) and only then can the parent thread return its status.
struct waiting_thread* child_shm: shared memory between current thread and its parent
struct list waiting_threads: list of shared memories between current thread and all its children
struct list file_list: list of all files opened by this thread
int fd: file descriptor counter. It is initialized to 2 (0 and 1 are reserved for stdin and stdout). On opening each new file it assigns the current value of fd to that file and increments fd by 1.
struct file* exec_file: Stores the pointer to the executable file of this thread.

VARIABLES:
struct lock main_lock: main_lock is used to update the shared memory and user program related members.
struct lock filesys_lock: filesys_lock is used in mutual exclusion in accessing, updating or executing files

>> B2: Describe how file descriptors are associated with open files. Are file descriptors unique within the entire OS or just within a single process?
The member **int fd** in struct thread provides the file descriptor to the opened files. In **thread_init()** (for "main" thread) and **thread_create()** (for other threads) the fd member of thread is assigned to be 2 (leaving 0 and 1 for STDIN and STDOUT respectively). When a new file is opened within a thread, it is assigned the current value of fd, and fd is then incremented by 1. Thus when the thread opens the first file, the file descriptor value is 2, and the fd value is increased to 3. This implies the next file opened by thread is assigned the file descriptor value to be 3, fd value increases to 4 and so on.
File descriptors are **unique just within a single process** as explained above. So the same file can have different file descriptors when opened by different processes. The **file_list** stores all opened files for a thread in its **list_elem struct file_info**.

---- ALGORITHMS ----
>> B3: Describe your code for reading and writing user data from the kernel.
When reading arguments for a system call from a user program, we have to ensure that all pointers passed to the syscall are in valid user memory. Similarly while writing user data, we have to ensure that the data is in valid user memory. To ensure this, we have implemented the functions **validate_ptr()**, **validate_buf()**, **validate_str()** and **getpage_ptr().**
The descriptions are as follows:
validate_ptr(): It takes an address vaddr and validates if the address lies in user virtual memory by comparing it against the values of **USER_VADDR_BOTTOM**(bottommost user virtual memory address) and **PHYS_BASE**(via the **is_user_vaddr()** function).
getpage_ptr(): It takes a virtual address vaddr, checks its corresponding page in the physical memory and returns the kernel virtual address corresponding to it. This is to ensure that the user virtual address provided is actually mapped to a physical address.

validate_str(): It takes a string and validates that the address of each character has been mapped to a kernel virtual address.
validate_buf(): It takes a buffer and validates each pointer in the buf is a valid virtual memory address.

>> 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?
For 4096 bytes of data, the least number of inspections of page table would be 1. This is when the complete 4096 bytes is present in a single page. The greatest number might be 4096 if it's not contiguous, in that case we have to check every address to ensure a valid access. When it's contiguous, the greatest number would be 2 when the data spans across 2 pages.

Also, for 2 bytes of data, the least number of inspections of page table would be 1. This is when the 2 bytes are present in a single page. The greatest number of inspections of page tables would be 2. This is when 1 byte lies at the end of one page and another byte lies at the start of another page.

We could improve these numbers by allocating blocks such that they do not cross boundaries unless necessary. For example, if we would normally allocate a block of 4096 across page boundaries, we could instead allocate a whole new page for it. This is when we assume page allocations to be continuous for the data to be copied.

>> 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.
It is true that the syscall can fail at any point during its execution, due to bad pointer value. We immediately terminate the process with the aid of **exit()** call passing -1 as the status. This leads to **thread_exit()** which further leads to **process_exit()**. The **process_exit()** frees the current process's resources. We close all the files held by the process. The current process's page directory is destroyed and switched back to the kernel only page directory. Assign NULL to the current page directory before switching. Activate the base page directory, then the active page directory is freed and cleared. This is done with the aid of repetitive **palloc_free_page()** call in **page_dir_destroy()**.
An error might occur in any of the validation functions. Say, a user program calls a write system call and passes a very long string that cannot fit in its virtual memory. Then the **validate_buf()** inside **write()** will fail and will call exit(-1). When exit syscall is called, the status is set to -1 and the **wait_lock** semaphore is released by this thread so that now the parent thread can acquire this semaphore to return the status value from **wait()** if it was waiting.

The resources for the thread that is called write are freed using repetitive **palloc_free_page()** call in **page_dir_destroy()** as described above.

---- RATIONALE ----

>> B6: Why did you choose to implement access to user memory from the kernel in the way that you did?
We implemented access to user memory from the kernel with a **two-step verification process**.
First we verify a given address is within the **PHYS_BASE** and **USER_VADDR_BOTTOM** (using **is_user_vaddr()** call). If the condition is false, we call **exit()** with status as **ERROR (i.e. value -1)**. This check provides the information regarding the existence of the given address to be in the user's address space.
We now move to obtaining all the relevant pages associated with the validated addresses, using **pagedir_get_page().** It looks up for the physical address corresponding to the given virtual address in the user process's page directory. An error (say page fault occurs), leads to conclude that the requested page does not exist. We can thus return a NULL pointer to denote the pointer is invalid and exit with **ERROR** status.
We have implemented this particular way because of its ease of understanding and smooth control flow from one function to another. This enables an easier mechanism to debug and can easily add further functionalities into the code.

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

1. Irrespective of procedure of file opening, the structure of file descriptor assignment remains the same across all the processes.
2. Our implementation eliminates the need to handle race conditions, since file descriptor is unique for each process
3. Our implementation also ensures quick addition of a new opened file to the file list of the current process. (Done using list_push_back())

Disadvantages:

1. The major disadvantage is the access time, which is O(n), where n is the number of file descriptors for the current thread. This is simply because, we have to iterate through the entire fd list. In a simple array implementation i.e. one file descriptor for a file system-wise, then it could have been O(1).