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

---- GROUP 18 ----

Member 1:
Tanmay Bansal 18CS10055
bansaltanmay9@gmail.com
Member 2:
Dhrumil Kavathia 18CS10015
dhrumilkavathia22@gmail.com

Goal:

1) To make our PintOS capable of running user processes by implementing argument passing.
2) We also implemented 4 syscalls namely
   - write()
   - exit()
   - exec()
   - wait()
   - create()
   - remove()
   - open()
   - filesize()
   - read()
   - seek()
   - close()

   in order to perform some basic processes by our PintOS.

ARGUMENT PASSING
==================

--- DATA STRUCTURES ---

a) **Data Structures added or some variable define:**

   We didn't change anything in the existing structures nor did we create any data structure for the argument passing part.

b) **Functions added/implemented in process.c**

   1) **push_to_stack()**
      This function is added to facilitate the checking of proper loading of stack in the memory and then pushing user passed arguments into the stack according to the mentioned format.

2) **load()**
We first extract the "file name" - the first token of the argument list and open this file. We are also calling push_to_stack() from this function to load the arguments in the stack.
3) **setup_stack()**
We are setting the esp pointer to PHYS_BASE - 12 to facilitate the proper argument passing.

--- ALGORITHMS ---

1) We are calling the setup_stack() from load() to check whether the stack has been successfully loaded in the memory.
2) If the stack is loaded in the memory then we are calling push_to_stack()
3) In push_to_stack() we parse the command line and do the following things :
i) First we get the general information about the arguments passed such as number of arguments passed and store it in the argc, total length of the arguments etc. and at the same time we also insert this user passed arguments into this stack.
ii) Now as we always extract arguments from the stack in the sizes of 4 (keeping the word size aligned throughout the code) so we check that the resulting length of the arguments passed is a multiple of 4 and if it is not 4 then we pass null pointers to the stack making it the multiple of 4.
iii) Now we push the null pointer marking the end of the user arguments.
iv) Now we push char ** pointer in this stack, this corresponds to the pushing of argv into the stack.
v) Now we push the number of arguments passed in the stack corresponding to insertion of argc into the stack.
vi) In the end we push the return address to the stack.

A2:
We avoid the overflowing of the stack by keeping the count of the number of arguments passed to the stack and if it overflows the stack page size we exit.

--- RATIONALE ---

A3:
strtok_r() lets the caller provide the save_ptr (pointer to the next token). In strtok() the pointer for the next token is stored internally in the function. Hence, if two or more threads are using strtok() it is possible that one thread may use some token held by another thread.
Hence strtok_r() is more thread safe.

A4:
(i) Kernel is meant for general management. Hence it is better to decrease the workload on the kernel.
(ii) This allows the shell to "sanity check" the arguments before passing it on to the kernel.

SYSTEM CALLS
==================

--- DATA STRUCTURE ---

a) **In struct thread in thread.h**

1) **struct semaphore child_lock;**
   It prevents the parent process from running when its running for its child.
2) **int waiting_on_thread;**
   It is used by thread to denote that whether it's waiting on any of its child and if it is waiting then this variable stores the child's thread id on which the current thread is waiting.
3) **struct thread *parent;**
   It denotes the parent of the thread.
4) **struct list child_proc;**
   List of all the child processes of a process.
5) **struct file *self;**
   Stores the pointer to the file the process is.(working on)
6) **struct list file_list;**
   list of all the files opened/required for that process.
7) **int exit_error;**
   stores the exit error of the process(if any).
8) **int fd_count;**
   Stores the file descriptor value for the next file to be used by the process.
9) **bool success;**
   Denotes whether the file corresponding to the thread has been successfully loaded or not.

b) **In thread.h**

1) **We defined a new struct in thread.h struct child;**
   - **int tid;**
     Thread id used to pick the correct child
   - **struct list_elem elem;**
     Used to get the child * from the *child_proc* list
   - **int exit_error;**
     Used to denote the exit error of the thread associated with this child(i.e thread_id == tid)
   - **bool used;**
     Used to denote that whether we have operated on the thread associated with this child before or not

   It stores the information of all the children of the current thread (used in case of waiting and closing the opened files).

c) **In thread.c**

1) **struct lock filesys_lock;**
   It is used to lock the access of the filesystem to one process at a time.

d) **In syscall.c**

1) **struct proc_file;**
   - **struct file *ptr;**

Pointer to the required file.

- **int fd;**
  fd number of the required file.
- **struct list_elem elem;**
  Used to insert elem into the file_list of each process, so that we can search and retrieve it with list_entry().

B2 :
The file descriptors are unique for each opened file per process. Each process maintains a file descriptor counter which is incremented each time a new file is opened. Thus the file descriptor is only unique within the process.

--- AIGORITHMS ---

a) **In process.c**
   1) **process_execute()**
      We first tokenized our file_name according to spaces and then created the thread according to the first token of the string passed to the PintOS.
   2) **start_process()**
      When the process starts we check whether it is loading properly or not and if it's not loading properly then we release the semaphore and call the thread_exit() to terminate this not loaded thread. And if the thread is loaded properly then we just release the semaphore.
   3) **process_wait()**
      We iterate over the child list of the thread on which process_wait() was called. If a child process with the required process id is found, then we update the waiting_on_thread attribute in the thread to the process id of that child process, and then wait till the semaphore child lock is released by the child process.
   4) **load()**
      We tokenized our argument string and opened the file named as the first token of the string and then called push_to_stack() to insert the arguments into the stack. We also made sure that no parallel accesses are made to the filesystem by using a global lock called filesystem_lock. We added locks at all points the filesystem was being accessed and made sure no concurrent access by two different processes was possible.
   5) **process_exit()**
      We lock the filesystem, close all the files opened/used by the process and unlock the filesystem again.

b) **In syscall.c**
   1) **syscall_handler()**
      We first take the syscall number and the pointer to the stack and then run a switch case matching each syscall number to its respective function.
   2) **SYS_EXEC :**
      We first check that the pointers are valid or not and if the pointers are valid then call the *use_exec()* method.
      use_exec()
      Tokenize the string and open the file after creating a lock for accessing the files then if file exists then call the *process_execute* by the *file_name*.

3) **SYS_EXIT :**
   We first check for the valid pointer and then call the *exit()* function
   exit()
   Mark all the children of the current thread as used and set their *exit_error* same as that of the current thread which is an argument of this function. Then if the current thread was waiting on any of its child then lift the lock and then call *thread_exit()*, an inbuilt function in thread.c
4) **SYS_WAIT :**
   We check for valid pointer and then call for *process_wait()* function in process.c
5) **SYS_CREATE :**
   We check for the correct pointers then we lock the file system such that only this thread is accessing it and and then we create the file using filesys_create(), a function defined in *filesys.c*
6) **SYS_OPEN :**
   We check for the correct pointer then we lock the file system such that only this thread is accessing it and and then we open the file using *filesys_open()*, a function defined in *filesys.c* which return the file pointer pointing to that file, then if the file exists we create a variable of *proc_file* type and we fill the information into it then we insert this structure in the file list of the current thread and increase *fd_count* of the thread by 1.
7) **SYS_REMOVE :**
   We check for the correct pointer then we lock the file system such that only this thread is accessing it and and then we remove the file using filesys_remove(), a function defined in *filesys.c*
8) **SYS_FILESIZE :**
   We check for the correct pointer then we lock the file system such that only this thread is accessing it and then we get the size of the file using file_length(), a function defined in *file.c* by passing the file pointer to the function.
9) **SYS_READ :**
   We check for the correct pointers. After that we find out whether reading is to be done from console or some file. In the first case we read the contents written on the console by using *input_getc()* function defined in *devices/input.c* and in the latter case we first get the *proc_file* structure from the list and then we lock the file system such that only this thread is accessing it and call *file_read()* function defined in *file.c*
10) **SYS_WRITE :**
   We check for valid pointers then we call inbuilt function putbuf()(in console.c) and then points the stack to the next suitable pointer.

c) **In thread.c**
   1) **lock_filesystem()**
      We lock filesys_lock, required to access the filesystem.
   2) **unlock_filesystem()**
      We unlock filesys_lock, required to access the filesystem.
   3) **init_thread()**
      We initialize struct thread *parent as the current thread that called to create that thread, waiting_on_thread=0 because when a thread is created, it doesn't have a child. We also initialize the lock.

B3:
Firstly we check the validity of the stack pointer and if it's valid then we dereference the stack to know which system call is to be run. Then we retrieve the arguments from the stack while checking for their validity simultaneously. Finally syscall functions can be called with the arguments, where the user data can be returned by the use of eax register.

B4:
As of now for each byte we have to call *pagedir_get_page()* once, as we don’t know how many next bytes pointed by the user lie on the same page. Now by the use of virtual memory we can modify this such that for 2 or 4096 bytes we only require 2 calls at most, as each number of bytes can be split into 2 pages at most. Implementation of virtual memory is an advanced project.

B5:
To best avoid obscuring the primary function of code along with error-handling, we first use the *prt_is_valid()* function to validate that all the necessary addresses and address values corresponding to them point to user space of memory. After this validation is done, we proceed onto the functional parts of the system calls. Because we had already checked about the required pointers being valid, we can go ahead without worrying. In the case some address is found to be invalid, we terminate the process trying to access it prematurely by using the *exit()* syscall. We have implemented the exit syscall in such a way that necessary files are closed, locks are released and taken care of.

--- RATIONALE ---

B6:
For every argument/value the user program wants to access, we have made a function *ptr_is_valid()* to check whether that points to a valid space in user space. IF it does belong to the user space, *exit(-1)* is that process is terminated. This is slightly easier to implement compared to the page fault memory handling.

B7:
Advantages:
In our design, file descriptor values are unique for files used/required for each process. But they may be the same for files required for other processes. The file descriptor value for each file required can be obtained from the process which requires the file. This makes implementation simpler.
Disadvantages:
Because we have implemented storing files as a list, accessing some file is O(n). This can however be improved by storing the files in some different data structure.