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

---- GROUP 11 ----

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

Sagnik Roy 18CS10063 <sagnikr38@gmail.com>
Debajyoti Kar 18CS10011 <debajyoti.apeejay@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.

https://static1.squarespace.com/static/5b18aa0955b02c1de94e4412/t/5b85fad2f950b7b16b7a2ed6/1535507195196/Pintos+Guide
https://jeason.gitbooks.io/pintos-reference-guide-sysu/content/userprog-systemcall.html

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.

- No new struct member was defined for the sole purpose 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?

To implement argument passing and storing the arguments on to the stack, the following auxiliary functions were used:

1. split_and _insert(): splits the name into individual tokens and pushed them into the stack.

2. add_word_align(): Self explanatory
3. add_null_char(): Self explanatory
4. add_argument_address(): Add the argument addresses (where on the stack they are) to the stack
5. add_return_address(): Self explanatory.

During the initial insertion we insert the arguments into the stack in the order in which they are encountered.
For example, if the string is “echo sagnik roy cse iit kgp” we will decrement the stack pointer by 5, insert “echo” using memcpy, decrement it again by 7, insert “sagnik” and so on for each of the tokens. This is the opposite of what is given in the example involving “/bin/ls -l foo bar”. However it does not matter as ultimately if we push the argument addresses in the right order, it will suffice. So after all the arguments are inserted we remember the address of the last inserted argument. While inserting the argument addresses, we start from this point and while we decrement the stack pointer, we increment the location where the arguments are stored. Thus we push arguments into the stack.

We did not implement any explicit checks for the stack pointer. So we keep on pushing the arguments into the stack. If and when the stack is overflowed it causes a page fault and is handled appropriately by the page_fault handler in exception.c which basically sends a kill signal and calls exit(-1).

---- RATIONALE ----

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

Pintos implements strtok_r() and not strtok(). strtok_r() is basically a reentrant version of strtok(). strtok() uses a global variable inside the C runtime library to keep track of the string position. However, since strtok_r() is a reentrant version, it takes in an extra argument which is used to store the state between calls instead of a global variable. Hence using strtok() leaves us prone to race condition. Suppose two threads are calling strtok(), there is a possible data race condition where one thread would use the last token held by another thread. This would be incorrect answers and hence incorrect argument passing.

>> 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.

1. In the unix approach, it makes the shell allocate memory for argument parsing, instead of the kernel.So, it effectively reduces workload on the kernel. If a user process runs out of memory, it can still be handled, but if the kernel runs out of memory, the entire system might crash.

2. It allows the shell to perform basic sanity checks before passing to the kernel, like if the command is valid or not, if the command is within limit or not etc.

```
                   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.
In thread.h:
#define KILLED 0                    /* thread killed */
#define EXITED 1                    /* thread exited */
#define ALIVE 2               /* thread alive */
#define UNDEFINED_EXIT -999         /* initial exit status*/

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. */
      struct list_elem allelem;           /* List element for all threads list. */

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

      struct file *exec_file;                    /*exec file held by this thread*/
      struct list fd_list;                /*list of file descriptors of this thread*/
      int fd_size;                  /*Size of fd_list*/
      struct semaphore wait_load;   /*Semaphore to load executable*/
      struct list child_list;       /*Child List*/
      struct thread * parent;       /*Parent Pointer*/


#ifdef USERPROG
      /* Owned by userprog/process.c. */
      uint32_t *pagedir;                  /* Page directory. */
#endif

      /* Owned by thread.c. */
      unsigned magic;                     /* Detects stack overflow. */
};

struct child_element
{
      struct list_elem child_elem;
      struct thread * child_thread;  /*Pointer to corresponding thread*/
    int child_pid;                  /*Child pid*/
    bool load_status;          /*Executable load status true if success false if not*/
    int cur_status;                 /*Current Status*/
    int exit_status;                /*Exit Status*/
};
```

>> 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 using the open() system call, it gets added to the current process' list of open files (fd_list) and is assigned a unique file descriptor. Thus each process keeps track of its file descriptors and the number of such file descriptors is stored in the variable fd_size. When the file is closed using the close() system call, the corresponding file descriptor is freed and fd_size is decremented.

So file descriptors are unique within a single process only.

---- ALGORITHMS ----

>> B3: Describe your code for reading and writing user data from the
>> kernel.
In both cases we first check validity of pointers using check_valid_pointer() function that ensures that the pointers point below PHYS_BASE and is within the page directory of the current process.
For reading:
We first check the value of fd. If it is 0, input needs to be read from the keyboard and we use the input_getc() function defined in devices/input.c for the purpose. Otherwise reading is to be performed from a file, so we call the function get_fd() which iterates on the fd_list of the current process and gets the required file. After this we acquire the file_lock to ensure this is the only process currently using this file and then read the file contents using the file_read() function defined in filesys/file.c. This function returns the number of bytes read from the file. Finally we release the file_lock and return the number of bytes returned by the file_read function.
For writing:
Again we first check the value of fd. If it is 1 (STDOUT_FILENO), writing needs to be done to the console and we use the putbuf() function defined in lib/kernel/console.c for this purpose after acquiring the stdout_lock (this was implemented in part i). Otherwise writing needs to be performed to a file, so we get the required file using get_fd() as before and acquire the file_lock to prevent other processes from accessing this file. Then we use the file_write() function defined in filesys/file.c to write the contents present in buffer to this file. This function returns the number of bytes written, which we finally return after releasing the file_lock.

>> 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?
The least number of inspections is 1 if the entire 4,096 bytes is present in a single page. The greatest number can be 4,096 if it is not contiguous and 2 if it is contiguous. If it is not contiguous, we might have to check every address to ensure a valid access. But when it is contiguous, in the worst case, the data might span across 2 pages.

For 2 bytes of data, the least number is 1. This is when the 2 bytes are present in a single page. The greatest number is 2 when 1 byte lies at the end of one page and the other byte lies at the start of another page.

An improvement would be to check if the address is less than PHYS_BASE and not a NULL, then we dereference it, letting it page fault if it is an invalid address. If it is invalid, then page fault will occur which can be handled.

>> 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.

Before using an argument pointer,we are using the function void check_valid_ptr() in order to detect any bad pointer value. All these checks are done before a syscall starts it's functionality. Upon detecting the error we call exit(-1), which calls to thread_exit(), which in turn calls process_exit(). Process exit handles deallocation of resources properly.
---- RATIONALE ----

>> B6: Why did you choose to implement access to user memory from the
>> kernel in the way that you did?
Our approach was to validate the user memory before using it. This approach is easier to implement, and in case of bad pointer value helps to deallocate resources directly by call to exit(-1).
>> B7: What advantages or disadvantages can you see to your design
>> for file descriptors?
Advantages:

1. thread-struct's space is minimised.
2. Kernel has access to all the open files, which gives much more flexibility to manipulate the open files.

Disadvantages:

1. It consumes a lot of kernel space, user process may open a large number of files and kernel might crash.
2. Accessing a file descriptor requires linear search over the entire fd_list of the thread. This takes O(n) time in the worst case, which might be quite slow.