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

----- GROUP 34 -----
Gajula Sai Chaitanya 18CS30018 <chaitanya6575@gmail.com>
Gunda Maneesh Kumar 18CS10020 <manishkumargunda@gmail.com>

---- PRELIMINARIES ----
Resources closely followed:
1) https://web.stanford.edu/class/cs140/projects/pintos/pintos_3.html#SEC49
2) https://stackoverflow.com/questions/52638134/pintos-syscalls-project-2

---- 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 haven't used any new structs/global varibles/typedefs nor updated any data structures for the argument passing part.

---- 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?

=> Divide the sentence by spaces then push element by element from right to left to the stack(using strtok_r()).
=> Then the multiple spaces are made equivalent to single space, alongside calculating the offsets for each argument.
=> If loading of ELF executable is done succesfully, then setup the stack using the push_args(...) function making use of offsets evaluated in above step;
=> To avoid overflow, performed a check on the total size of the args being passed.

---- RATIONALE ----

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

=> strtok_r() is reentrant version of strtok() and can be called from multiple threads simultaneously, or in nested loops. strtok_r is more threadsafe. Reentrant versions usually take an extra argument, this argument is used to store state between calls instead of using a global variable.
=> On the other hand, strtok() uses global state, so if it is called from multiple threads, program could crash because of the undefined behaviour.

> 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.

=> It's better to seperate arguments before parsing the args to kernel. It shouldn't be the kernel's job to parse that, there's no reason it couldn't be Allowing less of work to be done in kernel is more secure and preferred whenever possible.
=> Also, validation of the input could be done by the shell more safely than by kernel.

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 src/userprog/syscall.c

```
   (i) struct file_descriptor
       {
           int fd;
           struct file *file; struct
           list_elem elem; struct
           list_elem thread_elem;
       };
```

purpose: A struct for file descriptor containg fd,corresponding file and two other attributes to search for the file through the list of files used by the process.

```
   (ii)   static struct lock fl_lock;
```

purpose: A static struct lock variable used for synchronization to avoid illegal access to system calls simultaneously while using the file system.

```
   (iii)  static struct list fl_list;
```

purpose: A static struct list variable to maintain the list of files to be used by the system calls.

=> Few changes are made to struct thread in src/threads/thread.h, to be used across argument passing and system calls implementation.

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

  + struct file *self;                   /* the image file on filesystem */
  + struct thread *parent;               /* parent process of current thread */
  + struct list children;                /* list for child processes */
  + struct list_elem children_elem;      /* children list element structure */
  + struct semaphore wait;               /* semaphore to be used in process_wait*/
  + struct semaphore sema_begin;         /* semaphore for process_execute*/
  + struct list files;                   /* Files used by current thread */
  + int exit_status;                     /* return status of the thread */
  + bool exited;                         /* whether the thread is exited or not */
#endif
```

```
>  B2: Describe how file descriptors are associated with open files.
>  Are file descriptors unique within the entire OS or just within a
>  single process?
```

=> In our implementation, File descriptors are mapped to the files opened, using a struct file_descriptor defined in userprog/syscall.c. It contains fd number, a file pointer file, elem and thread_elem as its members. We maintain a list of 'files' per thread. We add a file_descriptor element each time open call is made and remove if close call is made. Also, we put a static function alloc_id which provides the fd number at every open() call by incrementing the static variable fd. The file descriptors are unqiue within entire OS as we used a static function to allocate the fds while opening a file.

```
---- ALGORITHMS ----

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

=> For reading, First fl_lock is acquired and if the reading is to be done from STDIN, we use input_getc to read the characters. And if it is STDOUT do nothing, just release the lock acquired. Else, we get the file pointer required using the search_file function. If file is not NULL, then use file_read to complete reading, followed by releasing the lock acquired.
=> For writing, similarly as for reading we first acquire the fl_lock and if if STDIN, do nothing just release the lock. Otherwise if we need to write to STDOUT, we used putbuf function to write to the buffer. Else, we search for file pointer through the fl_list using search_file function and write to file pointer using the file_write() function and release the 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 done would be 1, If in the first inspection using pagedir_get_page gets a page head back and all the data is contained on a single page.
=> The greatest number of inspections would be 4096 when the allocation is not contiguous and each byte is in a different page.
=> For 2 bytes of data, the least inspections will be 1 and highest inspections would be 2. There is a room for improvement if allocation is done such that all data is in a single page.

```
>  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.
```

=> We validate the stack pointer to avoid a bad memory access using validate_address() function. And further we use is_user_addr() to validate.
=> For example in read system call, first the syscall handler valiadtes the stack pointer esp, then the three pointers(arguments) are validated and also the whether the buffer space is valid memory access or not. We also check whether the access till the end of buffer+size is bad pointer or not. If it is a bad pointer then release the lock acqquired, close all files opened and exit.
=> Also, in expception.c we reduced esp to PHYS_BASE by 12 to avoid any erroneous kernel accessses.

---- RATIONALE ----

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

=> We first used validate_address() function to validate the stack pointer to check for user validity. Further on, while implementing system calls we used the is_user_addr() function to check the validity of each user memory access and proceed with the execution only if user memory access is valid. This implementation is simple way of access to user memory

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

=> An easy way to provide fds to a file while opening.Also kernel is aware of all opened files which provides more flexibility to manipulate them.
=> A possible disadvantage could be that, user program may open a lot of files which may possibly consumes a lot of kernel space as file descriptors are same within entire OS.