```
+---------------------------------------------------+
|                      CS 140                       |
|     PROJECT 2: USER PROGRAMS     |
|           DESIGN DOCUMENT                         |
+---------------------------------------------------+
```

---- GROUP ----

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

Ahmed Gaber <ahmedgaber20@gmail.com>
Monica Salama <monicasalama6@gmail.com>
Mostafa Gabriel <gabriel.mostafa@gmail.com>

---- PRELIMINARIES ----

>> If you have any preliminary comments on your submission, notes for the
>> TAs, or extra credit, please give them here.

- Don't have any.

>> 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.

- Nothing.

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.

- Nothing

---- 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?

- Getting the tokens from the command line in reverse order to be pushed in the stack after being setted-up.

---- RATIONALE ----

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

- To save the position of the next token in save_ptr instead of being in global buffer like strtok which makes it thread safe.

>> 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.

- In Pintos you can’t run multiple commands in the same line which could be done in Unix, Also in shell you don’t have to write the full path to the command as the full path will be stored in PATH system variable unlike in Pintos every time you should write the full path to the executable.

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 syscall.c:
```
struct lock filesys_lock;
```

- in syscall.h:
```
struct file_desc {
  int id;
  struct list_elem elem;
  struct file* file;
};
```

- in thread.h:
```
#ifdef USERPROG
    /* Owned by userprog/process.c. */
```

```
    uint32_t *pagedir;                  /* Page directory. */
    struct list file_descriptors;       /* List of the file_descriptors owned by this thread */
    struct list childs;
    struct child_process* parent_child_list;
    struct file *exe;
#endif
```

- in process.h:

```
  struct child_process {
    pid_t pid;
    struct list_elem elem;
    struct semaphore wait_load;
    struct semaphore waiting;
    int loaded;
    bool exited;
    bool wait;
    int exit_state;
  };
```

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

- each file has it's own file descriptor that includes its information.
  file descriptors is attached with each thread to hold it's opened files.

---- ALGORITHMS ----

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

- We make use of the given get_user() and put_user() functions to deal with
  the stack. At each call, we first check that the program is actually working
  on the user memory space, then perform the system call.

>> 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?

- Every one page can be allocated in a maximum of 2 pages. Therefore, when the

user wants to write, we call pagedir_get_page twice.
If 2 bytes of data needs to be copied, the situation is the same as there may
be one byte in a page, and the other in another page.

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

- We loop over all the processes in the (childs struct) and if a process with the given id is not found or that the wait was called for this child already , a -1 value is returned.
- If the child was found , and has been killed before the parent waits on it , we retrieve its exit status from the parent's child list and return it back.
- Other than that , a semaphore is used to signal the parent when the child exit or is killed(process exit), then this child is removed from the parent’s child list.

>> 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.

- At the beginning, we have to realize that the user program is the thread that is executing the system call to execute a code that is written by the OS.
  We check the "esp" (the stack pointer) before reading any argument, and check that the data pointed by it is also valid and within the user allowed memory.
  If we found an error, we call exit (-1) to tell the kernel to kill the thread.
  We also use a lock in each system call to make sure no concurrency issues occurs.

---- 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"?

-for each process , a pointer to itself in the parent's (if exists) child list is kept.
-for each child process , a semaphore is used to indicate the finish of loading whether it loaded successfully or not , and a variable to indicate its state.
-a semaphore is put to sleep,until the load has completed then it signals to the parent that has called exec and created this child that it has terminated loading.

>> 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?
-case 1: using a semaphore
-case 2: keep the child process's state (using struct child process)in the parent's child process until the parent is killed.
-case 3: any process that is killed or exited calls process exit, and then all its resources is cleared.
-all are the same the waiting of the parent does not affect the child process.

---- RATIONALE ----

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

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

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