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

---- GROUP ----

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

Ahmed Ibrahim Hafez
Ahmed ElSayed Mahmoud
Mohamed ElSayed Mohamed

---- PRELIMINARIES ----

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

- None.

>> 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.

- strtok_r(), strtok() manual pages: http://linux.die.net/man/3/strtok_r

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 did not define anything new or special for this part.
But we did encapsulate the implementation of argument passing into these 2 functions:

1) static char** extract_command_args(char * file_name, int *argc);
      Splits file_name on space " ", allocates the returned char**, and fills argc with the number of tokens.
2) static void setup_arguments(void **esp, const char * file_name);
      This function is called in 'load' function to fill the stack will the extracted arguments.

- Nothing was used except for simple arrays and pointers.

---- 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?

- Arguments are obtained from the command line input and are arranged as follows: <program name> <arg1> <arg2> … <arg n> separated by spaces.

In load():
    1) Call 'setup_stack()', Set the stack pointer (esp) to PHYS_BASE.
    2) Call 'setup_arguments()', Extract command arguments and fill the stack with the extracted arguments.

In setup_arguments():
    3) Call ‘extract_command_args()’, Return array containing separated arguments after parsing the command and split over spaces and update ‘argc’.
    4) Repeat the following for all extracted arguments (in reverse order):
        a- Update esp by subtracting the size of the current argument.
        b- Store the updated esp in ‘argv_add’ for pushing later.
    5) Update esp and push a NULL pointer indicating the end of argv[].
    6) Update esp and push ‘word_align’ bytes if the argument is less than 4 bytes.
    7) Update esp and push elements in ‘argv_add’ in reverse order.
    8) Update esp and push the address of the top of stack (argv).
    9) Update esp and push ‘argc’ (number of arguments).
   10) Update esp and push a NULL pointer to indicate a fake “return address”.

- We avoid overflowing the stack page by checking the size of the input ‘file_name’. Using a simple calculation we can estimate how much memory is needed for storing these arguments and addresses. If it would overflow the stack page size, we exit.

---- RATIONALE ----

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

- strtok() is not thread-safe, because it uses static variables to remember the tokenizing position for future calls.
So, if 2 threads called strtok() at the same time, this could lead to one thread accessing a string owned by a different thread.

- strtok_r() is thread-safe, because it remembers the tokenizing position via a third argument (savedptr) which is independent from the caller. Thus, each thread can remember its position independently.

>> 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.

- The first advantage of Unix approach is that it is much safer and simpler to use shell-based parsing operations. This way shell could help check any unsafe command line before they arrive at kernel directly, and thus reduce the complexity of kernel operations.

- The second advantage is with the widely used of shells, the shells become have power tools to achieve the requirements, so it is easy to it to handle these separation. But in the kernel, it will need much code to handle this.

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.

- struct process_file {
  struct file *file;
        Carries the opened file by the thread
  int fd;
        File descriptor value for the thread
  struct list_elem elem;
        To be put in thread files list
};
   Data of the file that is opened by this thread

Properties of thread when it became a child
- int load;
  Thread load state (loaded, not loaded, load failed).
- bool wait;
  Parent thread is waiting for me.
- int child_status;
  Thread status to be used in exit() method.
- struct list_elem child_elem;
  To be put in the thread children list.
- struct semaphore wait_sema;
  To make only one thread waiting for this thread.
- struct semaphore load_sema;
  To stop the parent thread till the child finish loading.

```
struct killed_thread
{
    struct list_elem elem;
       To be put in the killed children list of the thread
    int status;
    tid_t tid;
    bool wait;
};
```
    The data of the child thread if it dies before the parent.

- typedef int pid_t;

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

- File descriptors are unique just within a single process. Each process tracks a list of its file descriptors (list of struct process_file that carries unique fd, stored in struct thread), as well as its next available fd number int the thread. Our fd struct is what associates the file descriptor numbers with the corresponding file.

---- ALGORITHMS ----

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

- Read:
  First, check if the buffer and buffer + size are both valid pointers, if not, exit(-1). Acquire the file system lock. After the current thread becomes the lock holder, check if fd is in the two special cases: STDOUT_FILENO and STDIN_FILENO. If it is STDOUT_FILENO, then it is standard output, so release the lock and return -1. If fd is STDIN_FILENO, then retrieve keys from standard input. After that, release the lock and return 0. Otherwise, find the open file according to fd number from the open_files list. Then use file_read in filesys to read the file, get status. Release the lock and return the status.

- Write:
  Similar with read system call, first we need to make sure the given buffer pointer is valid. Acquire the fs_lock. When the given fd is STDIN_FILENO, then release the lock and return -1. When fd STDOUT_FILENO, then use putbuf to print the content of buffer to the console. Other than these two cases, find the open file through fd number. Use file_write to write buffer to the file and get the status. Release the lock and return the status.

>> 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 a full page of data:
The least number is 1. If the first pagedir_get_page get a page head back, which can be tell from the address, we don't actually need to inspect any more, it can contain one page of data.

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, if we get a kernel virtual address that is not a page head, we surely want to check the start pointer and the end pointer of the full page data, see if it's mapped.

For 2 bytes of data:
The least number will be 1. Like above, if we get back a kernel virtual address that has more than 2 bytes space to the end of page, we know it's in this page, another inspection is not necessary.

The greatest number will also be 2. If it's not contiguous or if it's contiguous but we get back a kernel virtual address that only 1 byte far from the end of page, we have to inspect where the other byte is located.

Improvements:
We don't see much room to improve.

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

- We ensure that the given tid corresponds to a child process by iterating through the list of child processes stored by the current thread. And iterating the killed_threads list as well. If it is not found, that means that a thread with that id doesn't exist, or wait had already been called and thus that child list_elem had already been removed, and it returns -1. If we found this child we check if it is in child_list or killed_list if it's in child_list we will call sema_down to make sure that the parent stop running until the child exit and call sema_up. And if it's in killed_list we do not call sema_down but only return the child state as we do if it was in 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.

- First, avoiding bad user memory access is done by checking before validating, by checking we mean using the function check_valid_ptr, It checks whetehr it's NULL or whether it's a valid user address and whether it's been mapped in the process's page directory. Taking "write" system call as an example, the esp pointer and the first argument pointer will be checked first, if anything is invalid, terminate the process. Then the buffer beginning pointer and the buffer ending pointer (buffer + size) will be checked before entering the "write" method.

Second when error still happens, we handle it in kill() in exception.c, as if it was a user page fault in user code so instead of just exiting the thread we call the exit(-1) to close the thread with setting its state to error.

---- 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"?

- when calling exec to load a new executable we call process_execute which try to create a new thread by calling thread_create() and we use a semaphore with n = 0 by calling sema_down we make sure that parent cannot continue running and it will wait for the new executable to be loaded and when the new thread created and try to be loaded we set the loading success state in our thread struct to be passed back and call sema_up to stop the new thread and get the parent back to running. checking the loading state and return -1 if the loading failed.

>> 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?

- The mechanism of the wait() function is to create a semaphore with n = 0 and calling sema_down() when you try to wait for a child process to make sure that the parent process will stop running until the child calls exit and when it exits we call sema_up() to get the parent back to running again and free all the resources of the child process like structs it contain or lists of children or files.

- When the parent process p calls wait(c) before c exits there is no matter and it will run as we described above the semaphore ensures prprer synchronization and avoid race condition and if p calls wait(c) after c exits there is also no matters as when c exit we free all its resources but we keep a pointer of it in the parent process in list called killed_threads so we can access the process c and get its status back even if after it killed

- When the parent process p terminates without waiting it does not matter if it terminate after c exit or before it does, as we will free all the parent process resources when it exits and when child c exit we will also free its resources, so it will not cause any problems.

---- RATIONALE ----

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

- To can validate it before using it so you can exit the thread when you try to access a wrong address.

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

- Advantages:
    1- Thread-struct’s space is minimized
    2- Because each thread has a list of its file descriptors, there is no limit on the number of open file descriptors (until we run out of memory).
    3- you can open the same file from multiple threads or open it from the same thread multiple times with new descriptor at every time

    4- Kernel is aware of all the open files, which gains more flexibility to
       manipulate the opened files.
    5- it is easy to free all resources files of the thread whit it exits.
- Disadvantages:
    1- Accessing a file descriptor is O(n), where n is the number of file
       descriptors for the current thread (have to iterate through the entire fd
       list). Could be O(1) if they were stored in an array.
    2- Consumes kernel space, user program may open lots of files to crash the
       kernel.

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

- We didn’t change it. We think it’s reasonable and implementable.

SURVEY QUESTIONS

Answering these questions is optional, but it will help us improve the
course in future quarters. Feel free to tell us anything you
want--these questions are just to spur your thoughts. You may also
choose to respond anonymously in the course evaluations at the end of
the quarter.

>> In your opinion, was this assignment, or any one of the three problems
>> in it, too easy or too hard? Did it take too long or too little time?

>> Did you find that working on a particular part of the assignment gave
>> you greater insight into some aspect of OS design?

>> Is there some particular fact or hint we should give students in
>> future quarters to help them solve the problems? Conversely, did you
>> find any of our guidance to be misleading?

>> Do you have any suggestions for the TAs to more effectively assist
>> students, either for future quarters or the remaining projects?

>> Any other comments?