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

---- GROUP ----

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

Deepika Ghodki <dghodki@buffalo.edu>
Palek Naithani <paleknai@buffalo.edu>
Kartikeyan Lakshminarayanan Vignesh <klakshmi@buffalo.edu>

---- 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://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.

```
char **arglist;
unsigned int *addr_list;
```

arglist is used to store the tokenised arguments temporarily before we push them to the stack.
addr_list is used to store the address of the tokenised arguments temporarily before we push them to the stack.
---- 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?
In load():

- We first parsed through the arguments to get the size (no of tokens (arguments)) and the executable that we then sent to filesys_open.
- We parse through the tokens and store them in arglist[].

In load() after the stack is setup (setup_stack is called):

- We parse through arglist[] and push each char onto the stack and in addr_list[] we add the address of every argument stored.We also calculate the total_size.
- We use total_size to calculate the padding to be added and push the stack and then push the sentinel onto the stack.
- We parse the addr_list[] we had created and push it onto the stack (in reverse order).Then push the address to the first address and finish by pushing a fake return address and freeing arglist and addr_list.

We avoid overflowing the stackpage by only pushing the arguments needed and freeing the lists used once we have pushed them.

---- RATIONALE ----

>> A3: Why does Pintos implement strtok_r() but not strtok()?
The strtok() function uses a static buffer while parsing, so it's not thread safe. It is unsafe to use static data in threaded programs such as kernels therefore we use strtok_r(), which is thread safe as it uses an extra argument - a pointer to char * in order to maintain context between successive calls that parse the same string.

>> 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.
Separating the commands into an executable name and arguments has a few advantages -

1. The division of such commands results in lesser code running in the Kernel providing more reliability and extending the OS is easier.
2. Porting the OS from one hardware design to another is easier.

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 child
{
 tid_t tid; //thread tid
 int exit_status; // status returned on exiting
 bool file_exists; //set to true if load was successful
 bool is_waiting; //set to true if parent is waiting on this thread
 struct list_elem child_elem; //elem for list
 bool has_exited; //set to true if child exits before parent calls wait
};
```

This struct keeps track of the child threads spawned by the exec sys call. tid is the thread identifier, file_exists captures the error while loading file (if true exec returns -1), is_waiting is set to check if the parent thread is waiting on this thread and has_exited is used to indicate that this thread has already exited and the parent must not wait for it (if wait is called).

```
struct fd_file_map
{
 struct file* file;
 int fd;
 struct list_elem fd_elem;
};
```

This struct holds the file to file descriptor mapping for any file that is opened in a thread.

```
   struct list fd_file_map_list; //list of fd-file mapping
   int fd_counter; //next fd value
   struct list children; //list of child processes
   struct semaphore exec_sema; //semaphore to synchronise exec
   struct semaphore wait_sema; //for wait synchronisation
   struct thread *parent;
   struct file * file; //holds reference to executable
```

The above changes have been added to thread struct

>> 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 used to hold reference to an open file. We use integers to store this identifier. For every open file a new fd is generated, which is unique within a process. The same file can be opened multiple times and each time a new fd value is generated. Every thread keeps its own counter for the fd value. File descriptor values may repeat across processes, they are unique only within a process.

---- ALGORITHMS ----

>> B3: Describe your code for reading and writing user data from the
>> kernel.
In the syscall_handler we first read the syscall number from the user stack. Depending on the value the subsequent arguments are read. In case of read and write, we read the file descriptor, buffer and size values and convert them into their respective data types. Before accessing their values we check if the addresses read from stack are pointing to valid for each argument. In case of buffer we check two cases - the address of buffer and the address to which buffer points. Once we have the valid arguments, respective system calls are invoked. In both cases, we first get the file reference from the file descriptor, the mapping of which is stored in the thread. If the file pointer is not obtained, we return -1. For read, if fd = 1, -1 is returned, else the file is read using the filesystem function. In case of write, if fd=1, the buffer value is written to the stdout, else to the file obtained from fd mapping. The bytes read or written are returned accordingly.

>> 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 (4096 bytes), the least number of calls can be 1 when all the bytes are continuous, else we have to inspect each address making the calls upto 4096 (for this case). Similarly for copying 2 bytes, minimum is 1 and maximum number of calls is 2.
If pagedir_get_page() could check for the first byte and continue to check for the other non continuous bytes, the number of calls can always be one.

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

We have used a semaphore to ensure that a thread waits for its child to exit. A list of children is maintained by the parent. This is added as soon as the child is created. When wait is called on a child thread, the parent blocks itself after ensuring the child is valid and has not already exited. To check if a child is valid, we iterate through the list of children and check if input tid matches with any element in the list. If not, we return -1. The parent waits until the child has terminated. When a child terminates, its exit status is updated in the parent's list and the parent is unblocked. The parent then returns the updated exit status.

>> 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.

We handle such cases separately. The syscall_handler takes care of routing a system call to the respective implementation on the basis of the sys call number which is read from stack. Depending on the system call number arguments are read from stack and validated separately. In order to validate them a common function isValidAddress is invoked, which checks if the memory being accessed is safe. If not, then the process exits with a status -1. This error handling is not mixed with the system call logic.

All temporarily allocated resources are freed in the exit system call just before calling thread_exit. As exit is invoked at every invalid memory access, we ensure the memory is freed irrespective of the way the process exits.

For example, if in a write system call, the user gives an invalid buffer, the isValidAddress() function would invoke exit(-1). This will update the exit status in parent's list and free all additional resources allocated.
Another example, if a process exits via page fault, even then we call exit in exception.c, this again invokes the exit system call and all resources are freed.

---- 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"?
To make sure that exec returns only after the creation of a new thread and loading of the new executable, we use a semaphore. Once a new thread is created, this element is added to its child list. The parent then waits until the new process has loaded. If the load was not successful, a file_exits flag is updated, in the parent's list. When the control then returns to exec, and if flag_exits is set to false, instead of returning tid, we return -1.

>> 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?
We have used a semaphore to ensure that a thread waits for its child to exit. A list of children in maintained in P. This is added as soon as the child is created. The child element comprises certain flags in order to handle edge cases. Following conditions have been handled

- If C exits after P waits, the parent is blocked as soon as wait(C) is called. The thread is unblocked (sema_up) only when C exits. So, in the exit syscall of C, the parent is unblocked.
- If C exits before P, the corresponding child element in P is updated with the exit_status and has_exited flag. And when wait is called later by P, then P does not wait and directly returns the exit_status of C that was stored in the list.
- Whenever C exits, all it's resources are freed up in the exit syscall. The list of children maintained by P are not freed until P exits.
- If P terminates without waiting for C, all it's resources will be freed up. P will not know that C is it's child and hence it will return -1.
- If P terminates after C exits, and does not wait, it will hold the information related to the child in it's list till it's running. These will be freed when it exits.
- Special cases include if wait is called on the same child twice. To handle this we set is_waiting to true for the child element in P's child list. Hence if wait is called and is_waiting is already true, P directly returns -1.

---- 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?
In our design every thread keeps a count of the current fd value, which is initialised to 2 to exclude STDIN and STDOUT. Every time a file is opened (open syscall), the counter is incremented and a new value is generated. The counter update is guarded by a lock to ensure fd is incremented by only one open syscall at a time.
We create an instance of struct fd_file_map and add this to the list of fd_file_map of the thread. For all file related operations, we iterate through this list to get the file reference from the fd value or vice versa.

The advantage of this approach is that we are able to generate fd dynamically and file reference from fd whenever needed. A disadvantage is that we now need extra space to store this mapping as a list in the thread instance.

>> 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 have not changed the tid_t to pid_t mapping in our implementation.

```
                           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?