```
                +--------------------------+
                |           EE461S         |
                | PROJECT 2: USER PROGRAMS |
                |         DESIGN DOCUMENT  |
                +--------------------------+
```

---- GROUP ----

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

Amy Reed, amy_hindman@yahoo.com
Carmina Francia carmina_francia32@yahoo.com

---- PRELIMINARIES ----

Professor noted the multi-oom test was extra credit, this should be working in our assignment.

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.

process.c:

The following two structures were created in class to keep the child process command and arguments.

```
typedef struct{
  char *name;
  int len;
}args_t;

typedef  struct  {
  int argc;
  args_t args[100];
} child_t;
```

---- 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?

Argument passing was implemented in class and we used the professors code. Two structs were created to hold the command and null terminated strings. The child struct which contains the command and arguments are passed into the create_thread function. Here the command and arguments are retreived and placed onto the stack starting with the last argument placed at the bottom, thus the top of the stack will contain the command, followed by the necessary arguments.

---- RATIONALE ----

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

strtok_r() is rentrant therefore execution could be interrupted and then restarted and it should keep correct behavior. It can be called by multiple threads and perform correctly.

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.

thread.h:
The following structure is part of a list containing all data for managing all child
processes containing items like id, necessary synchronization elements, and pointer to actual thread struct.
is created, includes the ID

```
struct child_process {
   struct list_elem elem;
   int pid;
   int exit_code;
   bool exit;
   bool wait;
   struct thread *process;
   char *file_name;
   struct semaphore sema;
};
```

List element for file descriptor table. The actual file descriptor table is maintained in the thread struct.

```
struct opened_file {
   struct list_elem elem;
   int fd;
   struct file* file;
};
```

thread.c
Global file system lock

```
struct lock fs_lock;
```

>> B2: Describe how file descriptors are associated with open files.
>> Are file descriptors unique within the entire OS or just within a
>> single process?
A file descriptor in our program is unique only to a process. It has a relationship to the actual file pointer returned when a file is opened and used by the file system calls.

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

Whenever user data is accessed we very that the address is valid first. We implement the second mechanism discussed in the homework documentation where we first verify that a user provided pointer is below PHYS_BASE, we then call a function to verify that the pointer is valid by dereferencing it. With the dereference if a fault occurs the address is not valid and the call will exit and return -1. We used the code provided by the documentention to read or write a byte to execute the dereference. When a failure occurs, the page code fault was modified and the call exits and indicates the failure.

>> 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 inspection(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.

We don’t see much room for improvement.

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

We implemented sys_wait in term of process_wait.

We define a new struct child_process to hold the child’s exit status. And a list representing all children the parent owns (child_list) is added into parent’s thread struct. We also introduce a parent_id inside child’s struct, to ensure child can find parent and set it’s status if parent still exists.

A child_process is created and added to list whenever a child is created. The parent busy waits until the child exited. To improve this code, a lock or semaphore can be used instead of calling thread_yield() or busy waiting.

If child calls sys_exit to exit, a boolean signal that indicate sys_exit is called and the child’s exit status (exit_code) will be set into the corresponding
child_list struct in parent’s children list.

If child is terminated by kernel, the boolean signal mentioned above is remain as false, which will be seen by parent, and understood child is terminated by kernel.

If parent terminates early, the list and all the structs in it will be free, then the child will find out the parent already exited and give up setting the status, continue to execute.

>> 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.

Avoiding bad user memory access is done by checking before validating, by checking we mean using the function check_ptr_get we wrote to check whether it’s NULL, 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 void pointer buffer parameter was checked first, if it’s invalid, immediately call sys_exit to terminate.

Assuming that the user address has already been verified to be below PHYS_BASE and the a page fault in the kernel merely sets eax to 0xffffffff and copies its former value into eip, but an error still happens, we handle it in page_fault exception. The user page fault calls sys_exit, returning status as -1, and terminate the process.

---- 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"?

We use a semaphore for synchronization between the parent and child. The parent lets the child process proceed via a semaphore once it has obtained the thread ID, and then the parent uses a semaphore to wait for all of the thread/process setup occurs and can proceed right before the assembly call to actually kick off the new process.

>> 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 use a child_process struct to hold each child process’s status, and a list of child_process (child_list) inside parent’s struct to represent all the children that
the process has. And use a monitor to prevent race condition.

Child is responsible to set it’s status in parent’s thread struct. When parent exits, the list inside it will be free.

So, in the cases above:
* P calls wait(C) before C exits
P will monitor child->exit. If so, P busy waits by calling the function thread_yield() until it exits by checking the child’s exit status. Then parent retrieves the child’s exit status.

* P calls wait(C) after C exits
P will monitor child->exit and found out C already exits and check it’s exit status (exit_code) directly.

* P terminates without waiting before C exits
The list inside P will be free. When C tries
to set it’s status and find out parent has exited, it will ignore it and continue to execute.

* P terminates after C exits
The same thing happen to P, which is free all the resources P has.

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

We chose to use the second method described in documentation to verify the address location was below PHYS_BASE and then dereference the pointer. We chose this method
as the documentation seemed to indicate this was the most appropriate way to handle

this check, in addition we did not have as much familiarity with virtual memory and the methods that would be required in method 1.

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

We created the struct opened_file that has a list element for file descriptor table. The actual file descriptor table is maintained in the thread struct.

Advantage:
-Kernel is aware of all opened file, which gains more flexibility to manipulate the opened file

Disadvantage:
-Consumes kernel space, user program may open lots of files that the kernel may not be able to handle.

>> B11: The default tid_t to pid_t mapping is the identity mapping.
>> If you changed it, what advantages are there to your approach?
No changes were made.

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?

Had we been required to complete the setup stack/child arguments portion of the assignment without help. This would have been very challenging so having the assistance with this portion was very reasonable. The assignment in the way it was set up and given was very reasonable for the time we had, and was consistent with other homeworks.

>> Did you find that working on a particular part of the assignment gave
>> you greater insight into some aspect of OS design?
Working with the filesystem calls, file descriptor table, and examining the filesystem implementation gave a great deal of insight into OS design. The exec/wait implementation also gave good insight into the OS functions we made use of earlier in the course.

>> 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?
I would continue to do the in class portions for setting up the stack, showing how to execute a test, and run the debugger with gcc.

>> Do you have any suggestions for the TAs to more effectively assist
>> students, either for future quarters or the remaining projects?
>> Any other comments?
The level of guidance form the professor and TA were appropriate to make
the assignment educational but still challenging.

Again - for future classes, the portion we reviewed in class were extremely helpful
to allowing us to complete this assignment. Critical pieces for success were
showing how to run the test,
some examples with the debugger, and actually having the professor walk us through
setting
up the arguments - i.e. setup stack implementation.