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

---- OSAURON ----

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

Avijeet Mishra <avijeetm@buffalo.edu>
Khushbu Mittal< <khushbum@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.

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

void thread_block_processing(); //helper function to disable the interrupt to block the thread and then enable interrupt

---- 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?

//Start of Change//
/*Get the file name with parameters in function start_process*/
Get the file name with parameters in function load
//End of Change//
and store it in args along with number of elements of args in argc. Then loop on args in reverse order to push data below PHYS_BASE. Then push the word alignment, null reference, address of elements in args, pointer of the first address of elements in args, argc and finally set the esp to the last address.

---- RATIONALE ----

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

strtok() uses a static pointer to store the position in the string. The static pointer is subjected to potential race conditions and is not thread-safe while strtok_r() takes a third argument to determine the place within the string to search tokens. The space to store the states is offered by the caller, and thus works in a multi-threaded environment in Pintos.

>> 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.

1. Since most of the tasks are getting executed in shell that means kernel has to do less work which would decrease the probability of having a bug or security vulnerability in host machine so the Unix approach is safer.
2. The Unix approach is flexible as different shells can choose to interpret and parse arguments in different ways, which could be useful depending on how the user wishes to communicate with programs, or, more importantly, how programs wish to communicate with one another.

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.
//Begin of Changes//

syscall.c:

```
static struct list file_list;
static struct lock file_lock;
static struct file *find_file_by_fd (int fd);
static struct fd_elem *find_fd_elem_by_fd (int fd);
static struct fd_elem *find_fd_elem_by_fd_in_process (int fd);
```

syscall.h:

```
/* It stores the all the description related to file */
struct fd_elem
{
   int fd;
   struct file *file;
   struct list_elem elem;
   struct list_elem thread_elem;
};

struct fd_buffer {
 int ref_count;
 char fd_buf[244];
```

```
 int first;
 int last;
 struct lock buf_lock;
};
//End of Changes//
```

thread.h:

```
struct thread{ //following are the newly added fields to the struct
for                        //running system calls
 #ifdef USERPROG
   /* Owned by userprog/process.c. */
   uint32_t *pagedir;                   /* Page directory. */
   struct thread *par_t;                 /*parent thread. */
   struct list fd_list;                  /* file descriptor*/
   struct intr_frame  i_f;         /*parent temporary state*/
   struct condition wait_child;    /*waiting for child while it is alive*/
   struct lock wait_child_lock;    /*lock parent process for wait_child*/
   struct list files;              /*child's list*/
   struct file *running_code_file;
   struct semaphore sema;
   char running_code_filename[20];
   uint32_t next_fd;
   int return_stat;
   bool ready;
   #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?

File descriptors are unique within a process only. Every process maintains a list of the file descriptors (struct list fd_list) owned by it (struct thread). It also keeps a track of the next available file descriptor number.

---- ALGORITHMS ----

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

We will first check if the file descriptor is valid. Then we will check if the addresses provided are valid. For every page in the range of memory that can be reached from user supplied address we first verify a user virtual address is below PHYS_BASE, and then we verify that the particular page is mapped. If there is any other situation that will cause unexpected termination of the process, the page fault handler will handle the situation. If all goes well, we can access memory.

>> 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?

If pagedir_get_page() is not used for pointer validation, then the least number of inspections of the page table could be 1 when the entire data would be stored on a single page and the greatest number of inspections of the page table could be 4096 when the data is distributed in byte-sized segments.

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

Whenever wait system call is called, the current thread will search for the child thread in its child list to find if there is one process that matches the given pid. If a match is found, child’s metadata is retrieved. Then we will examine if the parent process has already waited for the child. If the parent has already waited once for the child, then it immediately returns -1. If the process has not been waited for this child before, then we set the “already waited” information to true, preventing further waits in the future.

Next, before actually waiting for the child, we first check whether the child is still alive in the process metadata, as it is perfectly normal for a wait system call to be executed when the child is already dead. This is important since an already dead child would not have the chance to wake up its parent process, which would lead to the parent to wait forever. Note that even though the child may already be dead, child_info is not freed until both of the parent and child is dead, so we are still able to get the dead child's exit-status at this time. If the child is not already dead, then we will put the parent to wait.

>> 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 are validating all the pointers to make sure that the pointer is less than PHYS_BASE, and therefore not a kernel pointer by calling function pagedir_get_page(), and checking that the returned value is not null. We also ensure that this pointer reference to a valid address by using is_user_vaddr function. All the arguments in which pointers are passed, we have added a field containing its length so that it can be ensured that pointer is also valid. So this helps in avoiding obscuring the primary function of code.

For Example, in read function call, If validation for is_user_vaddr function fails then we can kill the process and free all the files opened by that process and rest of the things would be taken care automatically by syscall_exit().

---- 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"?

Whenever exec is being called we are utilizing start_process function as it allows the current thread to maintain its information without starting a child process. We have created an interrupt frame that keeps the information of the executable. If the test call successfully loads, then we will continue with the start_process. Else, we will 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 a condition variable (waiting_for_child) and lock (waiting_child_lock) specifically for this purpose. After the creation of C, P adds a struct child_thread_info containing information about C to its list. This information includes a boolean indicating whether the child is alive or dead
(i)When P calls wait(C) before C exits: P will wait until C enters process_exit, where C will update its status in P's list. Then P will remove the child_thread_info struct corresponding to C from its list and returns C's exit status.
(ii)When P calls wait(C) after C exits: P will find from its list of child_thread_info that C is dead, then it will remove C’s info from the list and will return C’s exit status. From then on, everytime wait(C) is called P won’t need to check the list but can return -1 immediately.
(iii) if P is terminated without waiting, before C exits: C's parent pointer is nulled out and P's list of child_thread_info structs is freed, along with all of its other malloc'd data.
(iv) if P is terminated without waiting, after C exits: P’s data will be freed

A special case would be when C gets terminated because of an exception. In that case, we need to make sure that C’s info in P’s list is updated correctly.

---- RATIONALE ----

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

Since it is easier and more straightforward logically to get the test out of the way to begin with.

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

Regardless of whether our file descriptors are created by pipe or open, the same structure can store the necessary information, and be used in essentially the same way. 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).

Disadvantages:

There could be duplicate file descriptor structs, for stdin and stdout as each thread contains structs for these 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?

It has not been changed because of the simplicity of this approach.

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?