```
+-------------------------+
|      CSE 421/521        |
|  PROJECT 2: USER PROGRAMS  |
|      DESIGN DOCUMENT    |
+-------------------------+
```

---- GROUP: Pint-US ----

Allen Daniel Yesa <allendan@buffalo.edu>
Bharadwaj Parthasarathy <bparthas@buffalo.edu>
Nimisha Philip Rodrigues <nr57@buffalo.edu>

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.

```
struct argument
  {
 char *arg; //points to starting address of argument
 size_t length; //lenth of the argument
 void *esp; //address of the argument in stack
 struct list_elem elem; //list for listing the argument
 };
```

It is new structure. It is used to store the list of arguments so that we can push it into 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?

Argument passing was implemented by extending process_execute() by using strtok_r().The elements in argv[] will be pushed into a stack from a argument list so that it maintains an order. We can avoid overflowing the stack page by limiting the arguments to those that will fit in a single page (#define MAX_ARG 128)

---- RATIONALE ----

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

strtok_r()allows interrupts in between the execution, service the interrupt service routine whereas strtok() does not allow interrupts. Only one thread executes at a time in pintos and hence we use strtok_r. strtok_r() is a reentrant version of strtok().

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 two advantages are:
1. If the number of arguments are greater it will crash the shell instead of the kernel in the case of unix.
2. It will reduce the kernel time as the amount of work done will be less.

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

The above structure is used to store the file details of each file used with its file descriptor

```
struct list file_table;
```

This list is used by each thread to maintain the list of files that are opened and used under it. It is declared in thread.h

```
int fd_value;
```

fd_value is used to allocate file descriptor for each file opened

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

When a file is opened by a process it open returns a new file descriptor. It is added to the file descriptor list of the process and removed when the file is closed. Each process has an independent set of file descriptors, so they are unique within a process.

---- ALGORITHMS ----

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

Kernel will access memory through pointers provided by a user program. The kernel must be very careful about doing so, because the user can pass a

null pointer, a pointer to unmapped virtual memory, or a pointer to kernel virtual address space (above PHYS_BASE). All of these types of invalid pointers must be rejected without harm to the kernel or other running processes. An invalid user pointer will cause a “page fault” that you can handle by modifying the code for page_fault() in ‘userprog/exception.c’.

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?

The least number of inspections one inspection The greatest number can be 4096 inspections. For 2 bytes the least number of inspections is also 1. The greatest number of inspections can only be 2 (no of bytes). We are yet to decide with the improvement.

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

Waits for a child process pid and retrieves the child’s exit status.
If pid is still alive, waits until it terminates. Then, returns the status that pid passed to exit. If pid did not call exit(), but was terminated by the kernel (e.g. killed due to an exception), wait(pid) will return -1.

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 check the pointer value if it is valid. If the pointer has a null value or pointer is pointing to pointer to kernel virtual address space (above PHYS_BASE) it must be rejected. Invalid pointers must be rejected

without harm to the kernel or other running processes, by terminating the offending process and freeing its resources and because of this validation the function is not obscure.

---- 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 have a variable success for every thread. This variable is used to pass the status back to the called thread.

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?

Every parent process/ thread maintains a list of Children. When a wait is called on the parent it waits for the child process to complete and only then the parent process exits.

---- RATIONALE ----

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

We used the first method i.e. to verify
the validity of a user-provided pointer, then dereference it. We have checked the memory access and handled it using is_valid_user(). This is much simpler and easy to implement.

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

File descriptors are maintained for each file in the thread context. So when any of the file operations are taking place involving file description, the current thread's open files are iterated using the file descriptors and the corresponding file is fetched.

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 did not change the mapping.

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?

Ans. Wait system call is difficult to implement.

Did you find that working on a particular part of the assignment gave

you greater insight into some aspect of OS design?

Ans. Yes, argument passing.

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?
Ans. More hint can be given for implementing wait system call.

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

Any other comments?