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

---- **GROUP 9** ----

Aditya Singh <aditya390402@gmail.com>
Itish Agarwal <itishagarwal2000@gmail.com>

---- **PRELIMINARIES** ----

>> References
https://web.stanford.edu/class/cs140/projects/pintos/pintos_3.html
https://www.youtube.com/watch?v=0E79vNZp1KI&t=1239s
http://bits.usc.edu/cs350/assignments/project2.pdf
https://inst.eecs.berkeley.edu/~cs162/sp16/static/projects/project2.pdf

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.

For argument passing, no extra 'struct' or global variable was declared. We took care of argument passing in process.c, where we defined a new function get_stack_args(char *file_name, void **esp, char **tok_ptr), which is used to add arguments to the stack and add padding and alignment if needed and ultimately update the stack pointer(esp).

---- **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?

The raw pintos comes with no feature to accommodate command line arguments passed when running the executable. Any process (whether kernel or user's) directly or indirectly calls process_execute. Inside process_execute, start_process is called which in turn calls function load, which pushes the arguments on the stack.
As the load() function is used to load files we also load values in the stack. This is done by calling a function 'setup_stack()' which is used to add values to the stack. If the stack is successfully set we then call the function 'get_stack_args()' where we add arguments to the stack.
We then take the variable 'total_length' which will store the total length of the arguments. We take a loop and subtract the required space for arguments. We also maintain a variable to keep a count of the number of arguments(ie, 'argc'). We then save the current position of the stack in a variable.
After this, null character is added and esp is updated by subtracting the sizeof(char*).
In this way addresses of the arguments are added. After adding addresses we add a pointer to the first address and update the esp. At last we update the stack pointer and add a fake return address. After adding the arguments we maintained the stack as a multiple of 4 (called word alignment).

We avoid overflowing the stack by keeping a check on the total size of the arguments being passed. If at any point the size of arguments exceeds the size of stack, we exit.

---- **RATIONALE** ----

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

In strtok_r(), the placeholder is provided by the caller, unlike strtok(). Since PintOS separates commands into executable names and arguments via the kernel, we need to store the address of the arguments to an accessible place so that we can make sure that the arguments don't get mixed up when multiple threads call strtok_r(). Each thread has a pointer (tok_ptr), independent from the caller, which is a determiner for its position.

>> A4: In Pintos, the kernel separates commands into an executable name and arguments. In Unix-like systems, the shell does this separation. Identify at least two advantages of the Unix approach.

The advantages of the Unix approach are as follows:

1. Input validation is much safer if done by the shell instead of the kernel. A problem could be caused if the kernel failed to parse data (for example, a large amount of text is passed as input) whereas in shell, the worst case scenario is that it simply crashes.

2. Separating the commands from the arguments can allow for some input pre-processing and is generally a cleaner approach since commands and arguments are separate from each other. Plus, there's no reason for a kernel to parse data, which can be easily done by a user program.

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.

We added some variables and structures and defined them in /thread/thread.c:
Variables in thread.c:
1. struct file* exec_file (executable file of this thread)

2. struct list fd_list (list of file descriptors of this thread)
3. int fd_size (present size of fd_list)
4. struct list child_list (list of children of this thread)
5. struct thread *parent (parent of this thread)
6. struct semaphore sema_exec (to wait for this child to get loaded)
7. struct semaphore sema_wait (to wait for child to finish)

struct for member of fd_list :

```
struct fd_elem {
     int fd;                      //File Descriptor ID
     struct file *myfile;         //Actual File
     struct list_elem element;     //List element to add the fd_elem
//in fd_list
};
```

struct for member of child_list

```
struct child_element {
    // List element for the child
    struct list_elem child_elem;
    // Pointer to the child thread
    struct thread * child_thread;
    // Check if wait function is called for the first time
    bool first_time_wait;
    bool loaded;
    int child_pid;
    int cur_status;
    int exit_status;
};
```

>> 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 single process. Each process has its own set of file descriptors, ie, a table of file entries. In other words, each process tracks a list of its file descriptors (list of struct fd, stored in struct thread), as well as its next available fd number, which is given as soon as a new file is opened. When close is

called on a file, the corresponding file descriptor is released and is available for the next file that is being opened.
Specifically, the open() system call on a file returns a file descriptor for the named file that is the lowest file descriptor currently not open for that process (or in pintos, thread).
Furthermore, for each process,file descriptors numbered 0, 1 and 2 are reserved for standard input, standard output and standard error. Hence the first opened file will have a file descriptor number 3 and onwards.

---- ALGORITHMS ----

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

The call for read() or write() system call is handled in syscall_hander. For both read and write, there are 3 arguments that need to be taken care of.
For read -> file descriptor number, buffer to read from, length of buffer
For write -> file descriptor number, buffer to write into, length of buffer
To extract these 3 arguments, inside both read() and write() functions in syscall.c, void **args_extract_3()** is called which extracts the required 3 arguments from the stack. We put a check so that all the 3 addresses are valid or not using **check_valid_ptr()** function, then we call read() or write() function accordingly.

- **int read (int fd, void *buffer, unsigned int read_size)** -> returns the number of bytes actually read, or -1 if file could not be read.
  - ➢ if(fd == 1), we read from the terminal character by character using the **input_getc()** function (which is defined in devices/input.c)
  - ➢ if(fd != 1), get the **fd_elem** using the **get_fd()** function and now we have the actual file pointer

    - ➢ Acquire the lock by calling **lock_cs()** which locks the critical section for synchronization problems
    - ➢ Read from the file into the buffer using the **file_read(myfile, buffer, read_size)** function (this function is defined in filesys/file.c)
    - ➢ After reading is done, release the lock and return value returned by **file_read()** function
- int write (int fd, const void *get_buffer, unsigned size) -> writes ‘size’ number of bytes from get_buffer into file pointed to by ‘fd’ and returns the number of bytes actually written.
    - ➢ if(fd == 0), it is standard output file descriptor and we print on the terminal using the putbuf() function (which is defined in lib/kernel/console.c)
    - ➢ if(fd != 0), we get the **fd_elem** using the **get_fd()** function and now we have the actual file pointer
    - ➢ Acquire the lock by calling **lock_cs()** which locks the critical section for synchronization problems
    - ➢ Write into the file using the **file_write(myfile, get_buffer, size)** function (defined in filesys/file.c)
    - ➢ After writing is done, release the lock and return value returned by **file_write()** function

>> 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 a system call causes a full page of data to be copied from user space into the kernel, the least possible number of inspections of the page table is 1 and the greatest possible number is 2. This depends on the fact whether data spans 1 page or 2 pages.

For a system call that copies 2 bytes of data, the least possible number of inspections of the page table is 1 and the greatest possible

number is 2 (both same as above). This depends on how many pages the data spans.

An improvement is to check if the address is less than PHYS_BASE and not NULL the dereference it, otherwise (it is invalid) page fault will occur.

>> B5: 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 first check the interrupt frame (esp) if valid using **check_valid_ptr()** function, if valid, then deference it and call for a particular system call.
Secondly, when extracting arguments from the stack, we check if they point to valid user addresses or not. If any of these checks fail, the further execution of system call is stopped. No resources like buffer, memory and semaphores values are allocated before all system call arguments are verified to be valid. Any page fault that may occur is handled using the **exit(-1)** call.

Next, we ensure that all resources are freed by calling **exit(-1)** that calls **thread_exit()** which in turn calls **process_exit()** which releases all the resources that were occupied by the thread.

Example : Dealing with bad pointers during the read() system call. What may happen is that the buffer may grow out of user memory space and try to enter into kernel memory space. To handle this, we check beforehand that the whole buffer must lie within user memory using

check_valid_ptr(starting_address + buffer_size) and then only proceed with read() system call.

---- RATIONALE ----

>> B6: Why did you choose to implement access to user memory from the kernel in the way that you did?

It is faster to signal a bad pointer via a page fault than checking if the pointer is NULL. This is because of the utilization of the MMU. Hence, we implement access to user memory from the kernel in the way that we did because it leads to better performance since this check is not being performed all the time. Page fault interrupts cause computers to slow down a lot but since the thread exits anyways if the pointer does encounter a page fault, the performance doesn't degrade in this way.

>> B7: What advantages or disadvantages can you see to your design for file descriptors?

Advantages :-

-> All opened files are visible to the kernel, making it easy for it to access and modify the opened files

-> Space of the thread structure is minimized by this file descriptor design

-> The data can be stored in the file descriptor structure irrespective of their method of creation (pipe or open)

Disadvantages :-

-> Accessing file descriptors is of complexity O(n) since we have to iterate through the whole fd_list for the thread

-> Our method consumes kernel space. Kernel crash is possible if user programs open a lot of files