```
+-------------------------+
|          OS 211         |
|  TASK 2: USER PROGRAMS  |
|     DESIGN DOCUMENT     |
+-------------------------+
```

---- GROUP ----

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

Shiyuan Feng <shiyuan.feng18@imperial.ac.uk>
Hongyuan Yan <hongyuan.yan18@imperial.ac.uk>
Manshu Wang <manshu.wang18@imperial.ac.uk>
Zhige Yu <zhige.yu18@imperial.ac.uk>

---- PRELIMINARIES ----

>> If you have any preliminary comments on your submission, or notes for the
>> markers, 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: (1 mark)
>> 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 roughly 25 words.

We did not add any new struct or struct member, global or static variable, typedef or enum in this task.

---- ALGORITHMS ----

>> A2: (2 marks)
>> How does your argument parsing code avoid overflowing the user's stack page?

We use strlcpy() instead of strcpy() to copy strings and strtok_r() to get the filename before the first space instead of strtok().

strlcpy() takes in a size as a parameter and will not copy more than that many bytes of data. This prevents buffer overflow.

Also, instead of first tokenizing the whole command line and storing all tokens in a 2-D array before argument passing, we decide to push the token to the stack immediately after each tokenization. Our implementation takes less memory since it would only require the memory of the buffer.

Furthermore, before each PUSH, we detect whether esp exceeds the page's limit. If so, we free all allocated memories and return false directly.

>> What are the efficiency considerations of your approach?

In argument passing. We did not pre-calculate the position of esp after argument passing, but to check if each PUSH is valid before pushing values to the stack. The former method requires to tokenize the string twice, the first time is to calculate the amount of space tokens take, and the second time is to push to the stack (we can store the tokens from the first tokenization and use them when pushing, but as we have discussed earlier, it is takes much more memory), which means we need to make another copy of the original command line, which is not efficient in terms of the memory usage.

---- RATIONALE ----

>> A3: (3 marks)
>> Why does Pintos implement strtok_r() but not strtok()?
strtok() is not thread safe. strtok() uses global variable, so when two threads are calling strtok() at the same time, this can lead to race condition.
strtok_r() is re-entrant and uses local variables only to avoid race conditions.
strlcpy() takes in a size as a parameter and will not copy more than that many bytes of data. This prevents buffer overflow.

>> A4: (2 marks)
>> In Pintos, the kernel separates commands into an executable name and arguments.
>> In Unix-like systems, the shell does this separation.
>> Identify two advantages of the Unix approach.
1. In Unix-like systems, if the argument is invalid, it would only cause a process to exit instead of a kernel panic.
2. Unix-like approach reduces the workload of the kernel because the shell handles this task for it.

SYSTEM CALLS

---- DATA STRUCTURES ----

>> B1: (6 marks)
>> 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 roughly 25 words.

```
struct file_fd_entry {
    struct file *file;              /* A pointer to file struct */
    int fd;                         /* File descriptor of the file */
    struct list_elem local_elem;    /* Form a list of files */
};

struct lock file_lock;              /* Lock to synchronise file usages*/


struct exit_status {
    tid_t tid;                      /* tid of the thread this struct
                                       refers to */
    struct semaphore has_exited;    /* Semaphore, parent thread calls wait
                                       on this thread and waits for this
                                       thread to exit */
    int status;                     /* the current status of thread */
    struct thread *thread;          /* corresponding child thread */
    struct list_elem elem;          /* list_elem for child_list */
};
```

```
struct thread {
#ifdef USERPROG
    /* Owned by userprog/process.c. */
    struct thread *parent;              /* Parent thread of this thread */
    struct thread *last_created_child;  /* Temporary pointer for keeping
                                           track of last created process'
                                           thread
                                         * Used only in process_exec, allow
                                           child process to set the parent
                                           thread's
                                         * last_created_child, so that the
                                           parent can add child to
                                           exit_status */
      struct list child_list;          /* Keep track of childrens'exit_status
                                                                entries */
    struct list local_file_fd_list;     /* List for storing fd and files */

    struct semaphore added_entry_for_child;       /* Semaphore, parent tells
                                                     child it has setup the
                                                     entry */
    struct semaphore child_reported_load_status;  /* Semaphore, child tells
                                                     parent it has reported
                                                    load_status to parent */

    bool child_load_successful;         /* Child updates this member of its
                                           parent after load */
    struct file *exec_file;             /* file that the current process is
                                           executing on */
    uint32_t *pagedir;                  /* Page directory. */

#endif
}
```

---- ALGORITHMS ----

>> B2: (2 marks)
>> Describe how your code ensures safe memory access of user provided data from within the kernel.

We implemented a function check_vaddr(), if the given vaddr is not valid, the system does the process to exit(FAIL).

To check if the vaddr is valid, there are three main things to consider.

1. The vaddr pointer is not NULL.
2. The vaddr pointer points to a user virtual address.
3. The kernel address corresponds to the virtual address can be found.

Each time before we pop the next value from the given pointer, we check if the esp is valid. Also, if the result of dereferencing the given pointer gives another pointer, say p'. The system also checks the validity of p'.

>> B3: (3 marks)
>> Suppose that we choose to verify user provided pointers by validating them before use (i.e. using the
>> first method described in the spec).

>> What is the least and the greatest possible number of inspections of the page table (e.g. calls to
>> pagedir_get_page()) that would need to be made in the following cases?
>> a) A system call that passes the kernel a pointer to 10 bytes of user data.

At least 1 page inspection and at most 2 page inspections if the data is spread across 2 pages. This is inspected by calling check_vaddr().

>> b) A system call that passes the kernel a pointer to a full page
>> (4,096 bytes) of user data.

At least 1 and at most 2 page inspections, when the pointer points to the start of a page, we only need 1 inspection. Otherwise we need 2 inspections. This is inspected by calling check_vaddr().

>> c) A system call that passes the kernel a pointer to 4 full pages
>> (16,384 bytes) of user data.

At least 4 and at most 5 page inspections, when the pointer points to the start of a page, 4 inspections are enough. Otherwise it takes 5 inspections. This is inspected by calling check_vaddr().

>> B4: (2 marks)
>> When an error is detected during a system call handler, how do you ensure
>> that all temporarily allocated resources (locks, buffers, etc.) are freed?

In our previous task 1 implementation, each thread has a list containing all the locks it is holding. When a process is killed during a system call handler, it calls exit(-1), which would eventually call thread_exit(), which calls process_exit(). In process_exit(), we close all files that the current thread is opening (and free corresponding fd_file entry) and free its children's exit_status entries. In thread_exit(), the exiting thread releases all locks it is holding.

>> B5: (8 marks)
>> Describe your implementation of the "wait" system call and how it interacts with process termination for
>> both the parent and child.

When a thread (the parent, namely p) executes a new program (the child, namely c), it also creates a <struct exit_status> entry for c. This entry contains c's tid, a semaphore for synchronization purpose, c's thread and an integer for c to update its exit_status and for p to retrieve c's exit_status.

When c exits, it finds the correct exit_status entry owned by p, updates its status and sema_up(&entry->has_exited) to tell p that I have updated my status.

When a wait system call is triggered, p calls process_wait() directly and returns what process_wait() returns.

In process_wait(), p first checks if child_tid passed in is a child of the current thread by calling function get_exit_status_by_tid() (this function simply search for the exit_status entry with the correct tid, and returns NULL if no such entry can be found):

>> If the function call returns NULL, it means either the given tid is not a child of the current process, or the current process has already waited for the tid (In this case, the entry will be removed from the child_list and be freed in the first wait). In either of these cases, return -1 directly.

>> If an entry has been found, the thread waits for c to finish updating its exit status by sema_down(&entry->has_exited) (if c has already exited, the has_exited semaphore must have been sema_uped by c). Then it retrieves c's exit status, removes the entry from its child_list, frees the entry, and finally returns c's exit status.

---- SYNCHRONIZATION ----

>> B6: (2 marks)
>> 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?

We added a semaphore 'child_reported_load_status' in our thread struct to track whether the current thread's child has reported its load status.

The exec() system call calls process_execute(). In process_execute(), we initialize this semaphore to 0 and sema_down it before return.

When the child has successfully updated its load status to its parent, it sema_up its parent's child_reported_load_status.

Therefore, the parent's exec system call cannot return before its child completes loading.

>> How is the load success/failure status passed back to the thread that calls "exec"?

Each thread has a member <bool child_load_successful>, which is for recording whether the child is successfully loaded or not.

When a child finishes its load, it updates its parent's child_load_successful with its return value from load(). Then it sema_up its parent's child_reported_load_status semaphore to tell parent that this value is now up-to-date.

>> B7: (5 marks)
>> Consider parent process P with child process C.
>> How do you ensure proper synchronization and avoid race conditions when:

Our wait() function calls process_wait() immediately.

>> i) P calls wait(C) before C exits?

Before C exits, the <exit_status> of C can be found in P's child_list. The process_wait() function calls sema_down() on the semaphore ('has_exited') which C holds in its <exit_status> to ensure synchronization.

So P has to wait until C calls update_exit_status() during exit. The call to update_exit_status() sema_up 'has_exited' in the exit_status entry owned by P whose tid == C->tid.

>> ii) P calls wait(C) after C exits?

After C exits, C's corresponding entry remains in P's child_list, the has_exited semaphore in that entry is sema_uped by C and the status is updated by C already when C calls exit. Hence when P sema_down that semaphore, P can immediately continue and retrieve C's status stored in that entry.

>> iii) P terminates, without waiting, before C exits?

We implemented a helper function, free_child_list_entries(), to free all entries in child_list and set children's parent to NULL.

In this case, P sets C's parent to NULL, removes C from P's child_list and frees memory of the <exit_status> struct corresponding to C.
During this process, the interrupt is disabled to avoid race conditions. And interrupt is enabled after finishing all the memories.

Hence when C exits, C will know its parent no longer exists. Hence it will not update its status to its parent. (update_exit_status also disable and re-enable interrupts before and after updating the exit status to avoid race conditions)

>> iv) P terminates, without waiting, after C exits?
In free_child_list_entries(), since P iterates through its child_list, one of its child processes C will finally be found. As C has already exited, we only remove C's entry from P's child_list and free the memory of the entry. The whole iteration is protected by disabling interrupts.

>> Additionally, how do you ensure that all resources are freed regardless of the above case?
In exit(),we release all the thread's memory by calling thread_exit(). In thread_exit() we call process_exit() which has 3 functionalities:
1.Close all the files. All the files that local thread opened are stored in the local_file_fd_list. We iterate over the list to close the files and free the struct memory.
2.Free all the child_list entries of the current thread.
3.Destroy current page.

---- RATIONALE ----

>> B8: (2 marks)
>> Why did you choose to implement safe access of user memory from the kernel in the way that you did?
Our implementation is to verify the validity of a user-provided pointer before dereferencing it.
It's the most straightforward way of doing this. Because if we would try to do the error handling in page-fault handler, we need to figure out a way to return the error code from the page-fault handler. Whereas in our implementation, failed pointer check leads to exit(-1) directly.
Also, it separates the pointer verification process from page-fault handler, hence our design has more clarity.

>> B9: (2 marks)
>> What advantages and disadvantages can you see to your design for file descriptors?
Our implementation is to store each pair of file and its file descriptor using a structure that couples file and an integer id called fd. We store this structure in a list in each thread's structure in its system call to open a file. We also provide a static global variable - fd_allocator – which increments itself by 1 each time when a descriptor is required; therefore, each file open can have a unique fd.
There are several advantages of this design. Firstly, we use a list to store these descriptor structures instead of an array; this requires much smaller space on each thread's page and moreover, we can dynamically allocate descriptors with unlimited demands. Secondly, because each process can check

whether the fd is in its own file_fd_list, it is easy to check if an fd is opened by each process in system calls including read, write, etc.

There are also some disadvantages in our design. Comparing to an array, traversing through a list takes O(n) time. Besides, we need malloc in the system calls for the descriptor structures, which are inefficient memory usages. However, we consider that the space that an array needs on each thread's stack is too large. Furthermore, using an array, either locally or globally, always needs to pre-allocate memory resources for it, which not only demands resources, but also has strict limitations on the number of files to be opened.