Project 2: User Programs

Preliminaries

> Fill in your name and email address.

Hao Wang tony.wanghao@stu.pku.edu.cn

> If you have any preliminary comments on your submission, notes for the TAs, please give them here.

I got 100% score on my laptop.

> 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.

JHU PINTOS

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.

I declared no new struct or struct member.

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?

I used the `esp` pointer in the interrupt frame. I put all the strings at top first, then comes the blanks to make all the stuff 4-aligned. Then, I put `argv[0..argc]` (`argv[argc]` = `NULL`), then `argv` and finally `argc`. All the strings are ended with a `\0`. The elements doesn't necessarily need to be in the right order. Rather, the pointers should be. I put from top to bottom `argv[argc]` through `argv[0]`.

Overflowing the stack page is avoided by limiting the number of arguments allowed.

RATIONALE

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

The `strtok` in C standard library has a saved string, which is both space-consuming and dangerous since many threads might call `strtok`, which could mess up its shared status. `strtok_r` is not only faster, but also safer in an asynchronous scenario.

> 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.

- If anything goes wrong during the separation, Unix could recover by shutting down the shell. However, our kernel could only crash and fail the whole system.
- Letting shell do the work in User Mode avoids any possible vulnerabilities should there be flaws in the implementation of Unix Kernel.
- The Unix approach is more efficient when parsing a huge argument since shell could be scheduled, but kernel might not.

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.

In `process.h`:

```
struct process {
    int exit_status;                    /**< Exit status. */
    bool load_success;                  /**< If load successful. */
    bool running;                       /**< If still running. */
    bool free_self;                     /**< If need to free itself. */
    pid_t pid;                          /**< Process Id. */

    /** Sema to block the process waiting for this one. */
    struct semaphore wait_for_process;
    struct semaphore loading;           /**< Sema indicate loading. */
    struct list_elem all_elem;          /**< Elem for all_list */
    struct list_elem elem;              /**< Elem for child in father. */

    struct list childs;                 /**< Child process. */
    /** The corresponding executable file */
    struct file *executable;

    struct list opened_files;           /**< Opened file list. */
    int next_fd;                        /**< Next fd assigned. */
};
```

`process` is the core struct in which we save all the data exclusive to one process and that need to be stored even after the process has exited.

`exit_status` specifies the exit number of this process. `load_success` and `loading` work together to ensure `exec` would be block until this thread finished loading and get the true loading status (success/failure). `running` shows to the father process if this process is still running. If set to `true`, father process would set `free_self` when it terminates to let child reap itself whenever it terminates. `pid` is the process ID of the current process. `wait_for_process` is the semaphore father process blocks on when it tries to `wait` its child process. `all_elem` and `elem` are respectively for `all_list` and `childs` in father process. `childs` is the list containing all the childs of this process. `executable` is a file handle to the

executable of this process, used to block writing when the process is still running. `opened_files` is this process' exclusive fd table. `next_fd` indicates the next assigned fd.

In `thread.h/struct thread`:

```
struct process *process;            /**< Corresponding Process. */
```

`process` here is a pointer to the process this thread represents.

In `syscall.h`:

```
struct opened_file {
    struct file *file;    /**< File handle. */
    int fd;               /**< fd. */
    struct list_elem elem;/**< Elem for opened_file_list. */
};
struct lock file_lock;
```

`opened_file` is the struct representing an opened file entry in a process' fd table. `fd` is its assigned file descriptor, `file` is the `struct file` corresponding to this entry, and `elem` is just for inserting this entry to `opened_files` in a process.

`file_lock` is a lock to avoid racing when functions from `filesys.h` or `file.h` are called.

In `syscall.c`:

```
/* The number of parameters required for each syscall. */
int syscall_param_num[13] = {0, 1, 1, 1, 2, 1, 1, 1, 3, 3, 2, 1, 1};
```

`syscall_param_num` is a global array, defined only to ease the pain when coding the part trying to check the input validity for syscalls.

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

The file descriptors are one-to-one assigned to each opened file entry. However, each opened file could have several opened file entries. File descriptors are unique within a single process, whereas 0 and 1 are respectively STDIN and STDOUT.

ALGORITHMS

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

I have taken the first approach in this. I implemented three auxiliary functions for this: `check_ptr_validity`, `check_mem_validity`, and `check_str_validity`. The first one checks whether a given pointer is valid by verifying that it's not `NULL`, it is within user address (`is_user_addr`),

and it has been mapped (`pagedir_get_page`). `check_mem_validity` is a continuous version, where I check both the start, the end, and every page in between of a given memory area using `check_ptr_validity`. `check_str_validity` checks if a given string is all valid by verifying that each consecutive byte, up until `\0`, are all valid.

After checking all this, we could be sure the pointer is fine and use it to read/write from the kernel. Another thing is that I added a line to set the `exit_status` of a process to -1 if a `page_fault` happens.

> 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 former would result in three checks, one at start and two at the end. The latter would call twice `pagedir_get_page`.

I think the former one could be optimized to only be called twice, once at the start and once at the end. However, the latter case is different. We must call it twice just to ensure the two pointers are both valid since they could be in different pages.

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

`wait` just calls `process_wait`, which downs a semaphore on its child process. This semaphore is upped when that process exits. In this case, `process_wait` also reaped that process by calling `free_process`. If the process corresponding to the `pid` given is not a child of the current process, `process_wait` just calls `exit(-1)` and terminate the process.

> 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.

As is mentioned before, I implemented functions to check the validity of a given memory. Whenever a syscall with pointers as arguments are called, these pointers are first checked for validity. If a pointer touches unallowed region, `exit (-1)` is called, which terminates the whole process, calling `free_process`. This function is also called when a `page_fault` happens. `free_process` basically cleans up all the resources occupied by a process, including all the locks and buffers.

We could give some examples:

- If `read` is called with an invalid memory for buffer `p ~ p+size`, it would trigger the `check_mem_validity` at the very beginning of the function, which then throw an `exit (-1)` and cause the process to be shut down. In this case, the `file_lock` is never touched as well as any buffer.

- If `filesize` is called such that the arguments are in a forbidden area, the `syscall_handler` will terminate the whole process when it was checking the validity of the corresponding memory region.
- If `NULL` is dereferenced, `page_fault` will readily terminate the process.

Basically, the idea is to check all the pointers before dereferencing them in syscall and whenever there is an error, terminate the process by `free_process` before allocating any resources.

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"?

I defined a semaphore `load_success` in `struct process` that is initialized with 0. `exec` tries to down this semaphore, which would only be upped once the process has finished loading. For passing the load status, I defined a boolean variable `load_success` to indicate whether the loading of the new process succeeded or not.

> 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?

- P calls wait before C exits
  - synch: P gets blocked on C's `waiting_for_thread`, which would only be upped in C's `process_exit`
  - reap: P would reap C once P is unblocked in `process_wait`, which is done by calling `free_process`
- P calls wait after C exits
  - synch: `waiting_for_thread` is a semaphore, which has already been upped by C to 1. Thus, P would not get blocked and continue to execute the rest of the function.
  - reap: Same as before.
- P terminates before C exits
  - First, P remove C from its children list. P would then set C's `free_self` to true, which would later let C know that it needs to free itself in `process_exit`. C would call `free_process` on itself after it frees its children.
- P terminates after C exits
  - C is not freed at first. When P terminates and calls `process_exit`, it would free all the resources occupied by its children, during which C gets freed.

I don't think there are any special cases. The thread_exit could not be interrupted, so there wouldn't be any other racings except for the preceding four.

RATIONALE

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

First of all, this way is easier to design and more importantly, less prone to tricky bugs. Additionally, the code for this way is simpler and allows for better maintainability.

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

- Advantages:
  - Less space occupied
  - A simple design allows for better maintainability
- Disadvantages:
  - Slow when searching the file with a given fd
  - FD grows forever, does not support reusing a fd

> B11: The default tid_t to pid_t mapping is the identity mapping. If you changed it, what advantages are there to your approach?

I didn't change it.