CS 130 Project 2: User Programs Design Document

Tianyi Zhang
2018533074
zhangty2@shanghaitech.edu.cn

Haoran Dang
2018533259
danghr@shanghaitech.edu.cn

Derun Li
0000000000
lidr@shanghaitech.edu.cn

0 PRELIMINARIES

0.1 Preliminary Comments

No preliminary comment for this project.

0.2 References

-

1 ARGUMENT PASSING

1.1 Data Structures

1.1.1 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. No modified data structure in this part.

1.2 Algorithms

1.2.1 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? In function `process_execute()` of `userprog/process.c`, we create the new thread with the thread name picked by our newly-added function `find_exec_name()`, which picks the name of the executable from the `file_name` so that the name of the thread is correct.

Then in function `setup_stack()` of `userprog/process.c`, we add two pointers to store `argc` and `argv` after calling our newly-added `find_args()` in `load()` to seperate them, which ensures that they are arranged in correct order. Then we push these arguments in correct places according to Section 3.5.1 of Pintos Document.

Number of `args[]` is limited by `MAX_ARGS` and `MAX_CMD_LENGTH`, which ensures that the stack is not overflowed.

1.3 Rationale

1.3.1 Why does Pintos implement `strtok_r()` but not `strtok()`? `strtok()` does not save the remaining part of the process but `strtok_r()` saves it in `*save_ptr`, which keeps the safety of the operation when multiple thread is accessing the same string.

1.3.2 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.

- It is easier to identify the file name instead of fetching from the command so that time can be saved when finding and opening the executable.
- If the name of the executable file contains special characters (like spaces), the kernel can also handle it easily.

2 SYSTEM CALLS

2.1 Data Structures

2.1.1 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 file `threads/threads.h`

```
/* List element of waiting child */
struct waiting_sema
{
  /* List element */
  struct list_elem elem;
  /* TID of children */
  int child_tid;
  /* Semaphore corresponding to thread
     TID */
  struct semaphore sema;
};

struct thread
{
      ...
      /* Exit status of the thread */
      int exit_status;
      /* Next file descriptor */
      int next_fd;
      /* List of opened files */
      struct list opened_files;
      /* Pointer to the executable of
         thecurrent thread */
      struct file *executing_file;
      ...
      /* List used to store its child threads */
      struct list child_threads_list;
      /* List element of parent's
         child_threads_list */
      struct list_elem child_elem;
      /* List of waiting_sema of waiting
         threads */
      struct list waiting;
      /* Status of exited */
      bool is_exited;
      /* Status of being waited */
      bool is_waited;
      /* Status of waiting others */
      bool is_waiting;
      /* Pointer to parent thread */
      struct thread* parent_thread;
```

```
    }

    /* File operation lock */
    struct lock file_lock;
```

- In file `userprog/syscall.c`

```
    /* Interrupt handler wrapper functions */
    static int (*syscall_handler_wrapper[20])
        (struct intr_frame *);
```

2.1.2 Describe how file descriptors are associated with open files. Are file descriptors unique within the entire OS or just within a single process? A fixed integer `next_fd` is saved in each thread. When opening a new file, if it exists, `allocate_fd()` will be called and it will return `next_fd++` whose value will be assigned to the opened file.

The file descriptors are unique within just a single process.

2.2 Algorithms

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

2.2.2 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?*

2.2.3 Briefly describe your implementation of the "`wait`*" system call and how it interacts with process termination.*

2.2.4 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.*

2.3 Synchronization

2.3.1 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`*"?*

2.3.2 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?*

2.4 Rationale

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

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

2.4.3 The default `tid_t` *to* `pid_t` *mapping is the identity mapping. If you changed it, what advantages are there to your approach?*

3 SURVEY QUESTIONS

In your opinion, was this assignment, or any one of the two 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? Not yet.

Any other comments? Not yet.

CONTRIBUTORS

Task		Tianyi Zhang	Haoran Dang	Derun Li
Task 1 Argument Passing	Concept	✓	✓	✓
	Implementation	✓	✓	✓
	Debugging		✓	
	Design Document		✓	
Task 2 System Calls	Concept	✓	✓	✓
	Implementation	✓	✓	✓
	Debugging	✓	✓	✓
	Design Document	✓	✓	✓