CS130 Project 2: User Programs Design Document

Group 13

Tianyao YAN
yanty@shanghaitech.edu.cn

Wenfei YU
yuwf@shanghaitech.edu.cn

I. Introduction

This document answers the questions in the template for Project 2.

II. Argument Passing

A. Data Structures

- 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 specific struct, struct member, global or static variable is needed for argument passing.

B. Algorithms

- 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?

  1) Split the command.
  2) Push the words onto the stack. Save them in argv[].
  3) Do word alignment.
  4) Push the address of each string stored in argv[].
  5) Push the null-pointer sentinel.
  6) Push the address of argv.
  7) Push argc.
  8) Push return address(0).

C. Rationale

- Why does Pintos implement strtok_r() but not strtok()?

  strtok_r() is safer than strtok() as the remaining string can be saved in save_ptr and can be revisited.

- 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.

  1) Let the kernel do less work. Therefore the program can be executed faster.
  2) Make the execution safer as bad tokens may destroy the kernel and the system.

III. System Calls

A. Data Structures

- 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 thread.h:

```
1  struct thread
2  {
3      ...
4      int exit_code;                  /*
           Exit code. */
5      struct thread *parent;          /*
           Parent of the thread. */
6      struct list children;           /*
           Children of the thread. */
7      bool is_load_success;           /* If
            load child is successful. */
8      struct file *f;                 /*
           The file loaded for this thread
           . */
9      struct semaphore load_child;    /*
           Control of loading child. */
10     struct list files;              /*
           List of the files held by this
           thread. */
11     int fd_count;                   /*
           Count of the files held by this
            thread. */
12     ...
13 };
14
15 struct thread_child
16 {
17     tid_t tid;                      /*
           Child TID. */
18     bool is_dead;                   /* If
            the child is dead. */
19     bool is_parent_waiting;         /* If
            a thread is waiting the child
           to finish executing. */
20     int exit_code;                  /*
           Exit code. */
```

```
    struct semaphore wait_child; /* Control of context switch between parent and child. */
    struct list_elem elem;       /* List element. */
};

struct thread_file
{
    int fd;                 /* File descriptor. */
    struct file *fp;        /* File pointer.*/
    struct list_elem elem; /* List element. */
};
```

In syscall.h:

```
struct lock lock_file; /* File lock. */
```

- 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 used as an item index in a catalog created by a unique process to record the files it opened.

B. Algorithms

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

  Read:

  1) We check whether the args are valid and then whether buffer to buffer + size is valid. We call exit(-1) if any of them is invalid.
  2) If fd == STDIN_FILENO, we use input_getc(), then return size;
  3) If fd == STDOUT_FILENO, we just return -1;
  4) For the other statuses of fd, we first search the file in the current thread. If we can find it, we will acquire lock_file and then do file_read(), release lock_file, lastly return the number of bytes it read. If we cannot find that file according to fd, we will return -1.

  Write:

  1) Similar as read, we first check whether the args are valid and then whether buffer to buffer + size is valid. We call exit(-1) if any of them is invalid.
  2) If fd == STDIN_FILENO, we return -1;
  3) If fd == STDOUT_FILENO, we use putbuf(buffer, size) and then return -1;
  4) If fd is in other statuses, we search for it in the current thread. If we find it, we will acquire lock_file and then do file_write(), release the lock_file and return the number of bytes it wrote. If we cannot find it, we just return -1.

- 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?

  At least 1 inspection is needed, and the greatest number may be 4096. To improve this, if we can ensure that in a page everything is continuous and starts from the head of this page, we can only check the tail, so only need 1 inspection. If we only copy 2 bytes, at least 1 inspection if these 2 bytes are ensured to be together and valid, and at most 2 inspections.

- Briefly describe your implementation of the ”wait” system call and how it interacts with process termination.

  1) First we check if the arg is valid. Otherwise exit(-1).
  2) Then we call process_wait() with the arg;
  3) In process_wait(), we first check if the tid (the arg) is valid. Otherwise return -1;
  4) Then we search the child according to the tid in the list of children in the current thread.
  5) If we find the child, we first check whether the child is already waited for by its parent. If so, we will return -1;
  6) Then we check if the child is alive, if so, we will set is_parent_waiting to be true, then do sema_down();
  7) Then we will record the exit_code of this child, remove it from the list, then free the space allocated to it.
  8) At last we return the exit_code we recorded.

- 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.

  1) First we use a function is_invalid_address() to check if the addresses are valid.
  2) If it returns false, we will call exit(-1) which will make the exit_code to -1 and exit the thread.
  3) Then all the resources will be freed in process_exit().

  For example, in syscall “open”, we first check if the address of the file it wants to open is valid. If not,

we call exit(-1) to exit the thread and raise the error, then the process_exit() will do sema_up for every children, free the space of them, then close all the files of the thread and also free the space.

C. Synchronization

- 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”?
  1) We first initialize tid by the return value of thread_create() in process_execute().
  2) In start_process(), we will do sema_up() and check if the load is success. If it failed, we will set exit_code of the current thread to -1. Then exit the thread.
  3) Then we check if tid == TID_ERROR in process_execute(). If so, load failed and we will return -1.
- 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?

  To avoid race conditions, we call sema_down() in process_wait() and sema_up() will be done in process_exit().

  We freed all the allocated resources in process_exit(). If C has already exited, its resource has been already freed. If C is still alive, after P exits, sema_up() will be called on the semaphore for C, so C will eventually exit and the resource will be freed. So no matter whether process C exits before process P, the resources are freed. If P terminated first, its resources including wait_child will be freed when P exits. If C is alive, it cannot find wait_child, so it will exit and the resources will be freed. If C is dead, it will exit normally and its resources will be freed as well.

D. Rationale

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

  We validate the address before accessing it to make sure that bad pointer will not destroy the system.
- What advantages or disadvantages can you see to your design for file descriptors?

  It is convenient for us to find an opened file in a unique thread or process, but in fact we cannot know which thread owns this.
- The default tid_t to pid_t mapping is the identity mapping. If you changed it, what advantages are there to your approach?

  We didn't change it since PintOS only requires one process together with one threads.

IV. 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?

  Testcase Multi-oom is so difficult that we are struggling against falling asleep before it passes. Unluckily, we fail and we are heartbroken.
- Did you find that working on a particular part of the assignment gave you greater insight into some aspect of OS design?

  Maybe.
- 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?

  Students need to be told which functions need to be modified or added.
- Do you have any suggestions for the TAs to more effectively assist students, either for future quarters or the remaining projects?

  For the physical and mental health of students, TA had better provide more tips.
- Any other comments?

  No.

References

- https://blog.csdn.net/Altair_alpha/article/details/126819252
- https://blog.csdn.net/Altair_alpha/article/details/127177624