CS 140 PROJECT 2: USER PROGRAMS DOCUMENT

GROUP

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

- Chuyi Zhao zhaochy1@shanghaitech.edu.cn
- Bowen Xu xubw@shanghaitech.edu.cn

PRELIMINARIES

If you have any preliminary comments on your submission, notes for the TAs, or extra credit, 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.

These documents help me understand the project requirements:

- *Pintos Guide* by Stephen Tsung-Han Sher
- A doc from github
- A doc from baidu

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.

No `struct` changed nor new global or static variable introduced in the implementation of argument parsing and passing.

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?

- arg parsing:
  First use `fn_copy` allocating a page for the file_name (actually the cmd line). Then use `strtok_r` to get the file executable `exe_name` and parse it to the `thread_create()`. This way we don't change the context of fn_copy and can prase it to `thread_create()` for later use.
- put argv[] in order:
  First use `strtok_r()` to split the parsed strings into exe_name and args, and store them into `char *args[256]` in order and record `argc`, the number of arguments. But we want to push them into stack top-down in reverse order. So I reversely
- avoid overflowing:
  When in `process_execute`, parsing the exe_name to the process, we check the length of the

arguments by `strlcpy` it to a page.Also we can use the limitation. if it is too long, `process_execute()` shall call `exit(-1)`。

RATIONALE

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

`strtok_r()` is more safe for threads than `strtok()` because the return pointer is malloc dynamically.

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.

1. Making argument substitution possible
2. Making pipelining possible

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 `struct thread`:

```
#ifdef USERPROG
    /* Owned by userprog/process.c. */
    uint32_t *pagedir;                  /* Page directory. Given originally */
    int exit_status;                    /* for exit syscall */
    struct list file_descriptor_list;   /* list of `struct file_descriptor`s,
described below */
    int fileNum_plus2;                  /* literally, help to set the fd int for
opened files */
    bool halted;                        /* whether halt is called */
    struct list child_thread_list;      // child process list
    struct list_elem child_thread_elem; // list_elem for child process list
#endif
```

in `syscall.c`:

```
struct file_descriptor{
  struct list_elem elem;  // for list operations in thread->file_descriptor_list

  int fd;                 // nonnegative int fd (0, 1 reserved, therefore
beginning from 2)
  struct file *file;      // the opened file
};
```

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

1. As I designed, each file descriptor is a struct file_descriptor, with a member `int fd` and a `struct file* file`. Therefore I write a function (below)

   ```
   struct file_descriptor* find_file_descriptor_by_fd(int fd);
   ```

   to find each open file with an integer fd.

2. File descriptors are unique just within a single process, because one file could be opened by several processes and their file descriptors are independent.

```
struct file_descriptor*
find_file_descriptor_by_fd(int fd){
  struct list l = thread_current()->file_descriptor_list;
  struct list_elem *e;

  for (e = list_begin (&l); e != list_end (&l); e = list_next (e)){
      struct file_descriptor *f = list_entry (e, struct file_descriptor, elem);
      if(f->fd == fd && check_pointer(f->file)){
        return f;
      }
  }

  return NULL;
}
```

ALGORITHMS

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

First check the valid pointer. Then use the fd to decide what the open file is. The 0, 1 are already reserved for input and output. And I write a function to find the corresponding file descriptor (thus the open file) by the fd number bu traversing the thread `struct list file_descriptor_list`. Then use functions from the `file.c`, `filesys.c` to finish reading and writing.

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?

As long as the size of the data is less or equal than one full page and larger than 1 byte, it may fit into 1 single page or 2 neighbouring pages. The least number of inspections of the page table should be 1 and the greatest number should be 2, for both full page of data or only 2 bytes of data. The address of the data is not guaranteed to be aligned to the page, there may be no room for improvement.

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

The `wait` system call directly calls function `process_wait`. In function `process_wait`, the thread handle accroding to the tid is fetched from the child process list. Then the father process will try to down the `being_waited_by_father_sema` semaphore of the child process and thus blocked until the semaphore being uped when the child process exits. The child thread will be removed from the child process list (, or more specifically, the ok-to-wait child process list) before father thread trying to down the semaphore, so that each child process can only be waited once. If a child thread is not in the list (,or if the list is empty), it must be waited beforehand and then not be waited again.

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.

The 2 functions `check_pointer` and `check_pointers` are implemented to check whether a bad pointer value is provided by the user program. In the `syscall_handler` and each system call handling function, `check_pointer` is called to check the pointers. If the pointer is checked to be bad, the temporary resources within the syscall function (files, buffers, locks, etc.) will be freed by the corresponding exception handling branch while the resources obtained by the child process outside the syscall function (other files, user pages, stacks, etc.) will be freed by calling the `exit` system call.

For example, in the very begining of the `syscall_handler` function:

```
int *sp = (int *)f->esp;
if(!check_pointer(sp)){
  exit(-1);
}
```

The stack pointer, also the pointer of the syscall number is checked by the `check_pointer` function. If the stack pointer provided is not in the user memory space, `exit` will be called immediately.

In system calls like `SYS_CREATE` , `SYS_READ` , `SYS_WRITE` , `SYS_SEEK` , multiple consequent pointers have to be checked. The `check_pointers` function is implemented for caarying out the check more conveniently.

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 tried to use a semaphore to prevent the `process_exec` function from returning the tid (, and also adding the child process into the child process list) before the loading of the excutable is completed. The success/failure status will be stored into a variable owned by the father process once the `load` function returns. Then the father process will be waked from blocking and check whether the loading of executable is successed or not. Unfortunately, this sometime causes the `exec` function being blocked forever and has to be disabled to not disrupt other features.

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?

The parent process waits for the child process by trying to down a semaphore which will be uped by the child process on its exit. Since it doesn't matter whether the semaphore is uped after or before the down, the `exit` can happen before `wait`. The freeing of the resources is carryed out in the `exit` function of the child process, which is not affected by the status of the parent process. Since the semaphore is stored in the `struct thread` of the child process, the early termination of parent process won't affect the `exit` of its child processes.

RATIONALE

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

As the Pintos Document says, there are 2 ways to choose for accessing
the user memory from the kernel:

> There are at least two reasonable ways to do this correctly. The first method is to verify the validity of a user-provided pointer, then dereference it. If you choose this route, you'll want to look at the functions in `userprog/pagedir.c` and in `threads/vaddr.h`. This is the simplest way to handle user memory access.
>
> The second method is to check only that a user pointer points below `PHYS_BASE`, then dereference it. An invalid user pointer will cause a "page fault" that you can handle by modifying the code for `page_fault()` in `userprog/exception.c`. This technique is normally faster because it takes advantage of the processor's MMU, so it tends to be used in real kernels (including Linux).

I choose the latter one because it is more simple, I only need to write a function to validate access and call `exit(-1)` immediatly if the user pointer is invalid.

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

- Advantages:
  It is simple.
  I add only one list and several struct members to manage the correspondence between the fd number and the files.
- Disadvantages:
  It could be slow.
  For my design, to get a file through a fd integer, you need to to call the function `find_file_descriptor_by_fd()`, which is of O(n) complexity. This could have been released.

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 did not change that.

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?

Not hard but too busy.

Did you find that working on a particular part of the assignment gave you greater insight into some aspect of OS design?

The implementation of syscall involves modifying context infomation of processes including stack pointers. Debuging the code with context switch helps understanding pintos a lot.

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?

The exit of user program and the termination of the kernel's `wait` function on the user program is carried out seperatedly. No testcase will be pass until a minimal `wait` function is implemented.

Do you have any suggestions for the TAs to more effectively assist students, either for future quarters or the remaining projects?

emmmm. Try persuade professor Ying prepose the lecture corresponded to the project or postpone the lecture-corresponding project so that they can meet? Lectures can teaches us the knowledge, only before we learn them during the project.

Any other comments?

Is it possible to provide us with a Markdown version template of the designdoc?
Thx