PROJECT 2: USER PROGRAMS DESIGN DOCUMENT

Zhang Yichi zhangych6@shanghaitech.edu.cn
Hu Aibo huab@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.

- get_user function
- How does thread/process switching work in 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.

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?

We add a `setup_arguments()` function called by `setup_stack()`.

1. Check the length of arguments. If the length is too long, return false.
2. Copy arguments from kernel space to user stack.

3. Use `strtok_r()` to split arguments and setup `argv[]` and `argc`.
4. Reverse `argv[]` to make it in the right order.
5. Set `esp` to the top of the stack.

To avoid overflowing the stack page, we check the length of arguments and the number of arguments.

RATIONALE

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

The `strtok_r()` function is a reentrant version of `strtok()`. `strtok()` is "stateful", which means that it keeps track of where it is in the string. This makes it impossible to use `strtok()` in a multithreaded program, because the state of the function is global. `strtok_r()` is a thread-safe version of `strtok()`.

> 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.

- The shell can do more complex parsing, such as quoting and escaping.
- Less code in the kernel.
- Less time spent in the kernel.

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.

```
// src/threads/thread.h
struct thread
  {
    ...
    int exit_status;                    /* Exit status of the thread. */
    struct list file_list;              /* List of files opened by the thread. */
    uint32_t next_fd;                   /* Next file descriptor to be assigned. */
    ...
  };
/* Element of file_list in thread */
struct file_node
  {
```

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

// src/userprog/syscall.c
typedef int pid_t; /* Corresponding to the documents */
int syscall_arg_number[30]; /* The number of arguments of every syscall */
void *syscall_func[30]; /* The function pointer of every syscall */

// src/filesys/file.c
static struct lock filesys_lock; /* Lock for file system operations. */
```

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

A file descriptor is a number. For each thread, we use a linked list `file_list` to store the opened files. In `struct file_node`, `fd` is the file descriptor and `file` is a `struct file` pointer.

When accessing a file by a file descriptor, we iterate over the `file_list` and return the `struct file_node *` whose `fd` equals to the file descriptor we given. See `static struct file_node *get_file_node_by_fd (struct list *file_list, int fd);` in `src/userprog/syscall.c`.

File descriptors are unique just within a single process. When `open` syscall is called, the new file descriptor is determined by `thread.next_fd`. `next_fd` is not shared between processes.

ALGORITHMS

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

Because interrupts does not change the page directory, we can access the user memory directly in the interrupt handler. After checking the address is valid, we can read or write the memeory directly.

The method we adopted to check the validity of the user memory address is described in **B6**.

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?

In a naive implementation, we simply check every address before reading. Thus the greatest possible number of inspections is 4096.

However since the page size is 4KB, the data is on at most 2 pages. We can use `lookup_page` and other functions to check the validity of the address. Thus the least possible number of inspections is 2.

For 2 bytes, the maximum number of inspections is 2048 and the minimum number is also 2.

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

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.

After reading the Pintos documentation, we modified `page_fault` exception handler, which is called when a page fault occurs. If the page fault is occurred in kernel mode, instead of panicking, we set `eax` to -1 and set `eip` to `eax` because `eax` stores the address of the next instruction.

We use `get_user` and `put_user` to check the validity of the user memory address. Besides, we write some helper functions, such as `check_int_get`, `check_buffer_get`, `check_buffer_put`, `check_string`.

To keep code clean, checking the validity of syscall id and arguments is done in the `syscall_handler`. Checking the validity of `buffer` or `file` or `cmd_line` is done in the corresponding system call function.

Checking is done at the begining, before any resource allocation.

For example, when `syscall_handler` is invoked

1. Check whether `f->esp` points to a valid user memory using `check_int_get`.
2. If `f->esp` is valid, read the integer which indicates the syscall id Let's assume it is `write`.
3. `write` needs 3 arguments. Chcek the validity of `f->esp + 1`, `f->esp + 2`, `f->esp + 3`.
4. Check `buffer` and `buffer + size - 1` is writable for user.
5. Do stuff...

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"?

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?

RATIONALE

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

We choose the method that check only that a user pointer points below PHYS_BASE, then dereference it. According to Pintos documentation, this method is normally faster. Once we verified the address is valid, we can simply use it without copying it to kernel space, which is more efficient.

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

We use linked list to store file nodes.

Advantages:

- `open` syscall inserts a file node after the list, which is fast.

- The file descriptor counter is not shared between processes, so there is no synchronization cost.

Disadvantages:

- `read/write/...` syscalls need to iterate over the list to get `file` pointer, which is slow.

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

We does not change it because multithreaded processes are not supported in Pintos.

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?

> 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?

> Any other comments?