Report

---- TEAM ----

>> Team name

14

>> Fill in the names, email addresses and contributions of your team members.

Sungmin Choi <smc9601@kaist.ac.kr> (contribution1)

Taeyoon Kim <tykimseoul@gmail.com> (contribution2)

50 + 50 = 100

>> Specify how many tokens your team will use.

0

---- PRELIMINARIES ----

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

None.

>> 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.

Pintos 2-2
https://medium.com/@minjw1026/pintos-2-2-cdcfb84bfdf

Pintos - User Program 2
https://blog.naver.com/PostView.nhn?blogId=adobeillustrator&logNo=220857007737&categoryNo=6&parentCategoryNo=0&viewDate=&currentPage=1&postListTopCurrentPage=1&from=postList&userTopListOpen=true&userTopListCount=5&userTopListManageOpen=false&userTopListCurrentPage=1

https://static1.squarespace.com/static/5b18aa0955b02c1de94e4412/t/5b85fad2f950b7b16b7a2ed6/1535507195196/Pintos+Guide

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.

We declared no new or changed `struct` or `struct` member, global or static variable, `typedef` or enumeration for argument 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?

We first copied the command string into a separate string. Then for every occurrence of `' '`, we replaced it with a `'\0'`.

- When we put the arguments into the stack, we first counted the number of arguments (`argc`) to allocate the memory space for the stack.
- Iterate the argument tokens, memcpy'ing into stack pointer (`esp`) and saving the stack pointer to `argv` array.

- Push appropriate amount of word align through modulus calculation.
- Push a byte of `NULL`.
- Memcpy addresses stored in `argv` into `esp`.
- Push address of start of `argv` or (`argv[0]`).
- Push `argc`.
- Push return address.
- Free calloc'ed array `argv`.

---- RATIONALE ----

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

The difference between `strtok()` and `strtok_r()` is that `strtok()` stores the return variable inside the function whereas `strtok_r()` returns the pointer, and in usage, we have to declare a variable `strtok_r()` returns the value to. The benefit of storing this information outside is that if more than one processes try to call `strtok()` on the same string, it might cause a race condition when referring to the same internal variable `strtok()` stores the tokens in, resulting in bugs. By saving it outside, such conditions are avoided.

>> 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.

As the load of separating the executable name from arguments is shifted from kernel to shell,

- kernel code becomes much simpler (and hence better performing and less buggy).
- and there would be more protection for possible exploits.

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.

thread.h

```
struct thread {
    ...

#ifdef USERPROG
    /* Owned by userprog/process.c. */
    uint32_t *pagedir;                  /* Page directory. */
    struct list children;               //list of child processes of this thread
    struct list_elem child_elem;        // list_elem for children list
    int exit_status;                    // store exit status of this thread
    struct semaphore child_sema;        // semaphore for the child
    struct file* files[FILE_MAX_COUNT]; // file array
    bool load_success;                  // store result of load
    struct semaphore exit_sema;         // separate semaphore for exiting
#endif
    ...
};
```

syscall.c

```
...
struct lock file_lock;

void syscall_init(void) {
    lock_init(&file_lock);
    intr_register_int(0x30, 3, INTR_ON, syscall_handler, "syscall");
}
...
```

We declared a global lock named `file_lock` to handle synchronization of reading from/writing to files.

>> B2: 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 describes all the files that are open at the moment by assigining each new file opened a new index inside the file descriptor and processes can

refer to files previously opened via that index. Similarly, when a process closes a file (via referring to the open file with the index), file descriptor helps locating the file easily and close it. As we can see from above declaration of `files[]` in `thread.h > struct thread`, file descriptors are unique within a single process/thread.

---- ALGORITHMS ----

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

Reading data from the kernel

When a process reads from the kernel, we use `input_getc()` to read from it.

Writing data to the kernel

When a process writes to the kernel, we use `putbuf()` to write to it.

>> 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 both cases of assigning a full page or just 2 bytes, the worst case is that we have to check `pagedir_get_page()` twice since the data can span over two pages.

We don't know how to improve these numbers. :(

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

In `wait` system call, the current thread iterates over the list of its children threads and finds the thread it is currently waiting for. It downs the `child_sema` of the target thread to wait for the thread to end which is upped in `process_exit` of the child thread. Once it happens, it continues to save the `exit_status` of the just exited child process and returns it.

One problem here is that after `child_sema` is upped, the child thread might be freed from memory before the parent gets to save its `exit_status`. To prevent

this, each thread also has an `exit_sema`, which is downed in `exit_process` before freeing its memory and upped in `process_wait` after saving the exit status, preventing such race conditions from happening.

>> 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.

We tried to handle issues such as bad pointer, access to kernel, etc. by checking the validity of the pointers as soon as possible, *i. e.*, in syscall_handler. By doing so, we can make sure that the addresses are validated before allocating any resources required to continue with the system call, making it easier to free temporarily allocated resources (since validity is checked before assigning these resources).

Our approach to avoiding obscuring the primary function of code in error-handling was to separate them in different function calls so that error-handling is hidden from view at the system calls.

---- 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"?

We perform this functionality by using a semaphore. We wait in the parent process by downing the `child_sema` of the newly executed process. Once load is done in the new process, this semaphore is upped, regardless of the success

state of `load`. Whether `load` succeeded or failed is passed to the parent through a struct member `bool load_success` as declared above. After the down is released in the parent process, it checks the member `load_success` of the child and correctly returns either -1 or `tid` of the child depending on the state.

>> 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?

As briefly explained in **B5**, we use a set of semaphores to ensure proper exiting of both parent and child processes and avoid race conditions both when P calls wait(C) before and after C exits. In `process_exit`, each process destroys its page if it exists, making sure all resources are freed, regardless of whether P or C exits first.

---- RATIONALE ----

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

We chose to use a single global lock `file_lock`. We knew of two ways we could implement access to user memory.

1. to use a single `file_lock` and allow only one thread at a time to access `filesys`.
2. to use a set of semaphores and locks to allow simultaneous reading to files but only allow a single process to write (and during which reads are also not allowed). (first read and write)

We thought method 1 is easier to implement, so we did so.

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

Our file descriptor is an array of pointers to files. We could use the given library functions associated to linked lists, which could be easier to iterate through and use given library functions. However, such method of implementation would be heavier, having to declare all the elems and push and pop into the list, etc. Given that a process can access up to 128 files, using a simple array of pointers might be faster.

>> 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 didn't change it.

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 two problems in it, too easy or too hard? Did it take too long or too little time?

No comment

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

No comment

>> 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?

Initially, the codes given to us to test a single test case:

```
cd build
pintos-mkdisk filesys.dsk --filesys-size=2
pintos -f -q
pintos -p ./tests/filesys/base/syn-read -a syn-read -- -q
pintos -q run syn-read
```

wasn't working. It worked with `.../examples/echo` but not for anything else. We didn't get much help on what was our problem and how to solve it. But at some point of time it randomly started working, so it wasn't a very big deal.

We noticed that the result from `make check` was different from issuing the above command. We don't understand why, but it was important to know that it does, so we can compare what our result is and how it is different from the expected result.

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

No comment

>> Any other comments?

No comment