Design document (Project 2: User Programs)

Team 3

- ChaYoung You chacha@kaist.ac.kr
- HyunJin Jung jhjnav7@kaist.ac.kr

Use 1 token.

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.

There is no change.

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?

First copy the command line string to new page. Then set `argv` to another new page, make each element points to the result of `strtok_r()`. After the `setup_stack()`, copy each string in `argv` backward. Then make `esp` word-aligned, store each `argv[]` pointer backward and `argv`, `argc`, and return address. We allocates new page, so it can't be overflowed.

Rationale

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

`strtok_r()` is thread-safe, but maybe `strtok()` isn't because of `save_ptr`.

> 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.

When a problem occurs during separating commands, it affects the kernel. But if the shell parses commands, the problem only affects the shell, not the kernel. Also we can keep the kernel small and fast.

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.

- `threads/thread.h`
  - `struct thread`
    - `int max_fd` : The largest file descriptor.
    - `struct list fd_list` : List of file descriptors.
    - `int exit_status` : Exit status.
    - `struct thread *parent` : Parent thread.
    - `struct list child_list` : List of child threads.
    - `struct file *executable` : Executable file.
    - `enum load_status child_status` : The load status of child thread.
    - `struct load load_lock` : The lock for child_status.
    - `struct condition load_cond` : The condition variable for child_status.
  - `struct thread_fd`
    - `int fd` : File descriptor.
    - `struct file *file` : The file.
    - `struct list_elem elem` : List element.
  - `struct thread_child`
    - `tid_t tid` : Thread identifier.
    - `bool exited` : Indicate whether it's exited.
    - `int status` : Exit status code.
    - `struct semaphore sema` : Semaphore for wait.
    - `struct list_elem elem` : List element.
  - `enum load_status` : Load status which is one of `LOADING`, `LOADED`, `FAILED`.

- `userprog/syscall.c`
  - `static struct lock filesys_lock` : Lock for file system.

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

Each process has its list of file descriptors. Except the fd 0 and 1, every file descriptor is unique just within a single process.

Algorithms

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

It's handled by `syscall_handler()` in `userprog/syscall.c` , by `sys_read()` and `sys_write()` . Each file operation should be synchronized, so `struct lock filesys_lock` is used for it. For reading, fd 0 gets keyboard input from `input_getc()` and for other, it reads through `file_read()` . For writing, fd 1 writes to console using `putbuf()` and for other, it writes through `file_write()` .

> 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?

The least possible number of inspections is 8, because `inode_read_at()` reads maximum `DISK_SECTOR_SIZE` bytes, which is 512. The greatest number of inspections is 4096. For 2 bytes of data, inspections can be once or twice. If we increase the chunk size, then may be can reduce the least number.

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

It just calls `process_wait()` and it finds the child thread with `child_tid` . If it isn't terminated yet, calls `sema_down()` to the semaphore of `thread_child` element. Note that `process_exit()` calls `sema_up()` to the semaphore, so waiting thread can get the exit status properly.

> 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.

At first, we ensure the every argument of system calls are in the user memory address. If one of them are in kernel memory or points to kernel memory, then it exits immediately. Also, when a user-specified address is a NULL pointer, points to kernel memory, or points to unmapped memory, it causes page fault. Note that `page_fault()` also calls `sys_exit()`, which closes every allocated resources. So we can ensure the safety about the user-specified address and allocated resources.

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”?

`struct thread` has `enum load_status child_status`, `struct lock load_lock` and `struct condition load_cond`. When we create a process, we set `child_status` of current thread to `LOADING` and `sys_exec()` waits until `child_status` is not `LOADING` using condition variable `load_cond`. Then if the `child_status` is `FAILED`, set by `process_execute()` or `start_process()` due to an error, `exec` return -1. Otherwise, it returns tid returned from `process_execute()`.

> B8: Consider parent process P with child process C. How do you ensure proper synchronization and avoid race conditions when P calls wait© 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?

Each thread has `struct list child_list` consists of `struct thread_child`. The `struct thread_child` has `bool exited`, `int status`. When a child thread is created, parent pushes `struct thread_child` to its `child_list` with `exited = false`. Also `process_wait()` calls `sema_down()` to `sema` of `struct thread_child`, wakes up when `process_exit()` calls `sema_up()`. `process_wait()`

checks `bool exited` before calling `sema_down()`, to know whether the child process already exited.

Rationale

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

We only check if the pointer points under `PHYS_BASE` and then handle page faults in `page_fault()`. This method is normally faster then verifying validity of a pointer.

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

For now we just holds every file descriptors ordered by fd. When we find some file, we look into every element and check its fd. This can be slow some situation.

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

It isn't changed.