CS130 Project 2 Design Document

Group16

- Cunhan You: youch@shanghaitech.edu.cn
- Junda Shen: shenjd@shanghaitech.edu.cn

Reference:

> Pintos Guide by Stephen Tsung-Han Sher

Task 1: Argument Passing

Data Structure

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.

```
/* Store the address of each argv stored at the top of the stack. */
char *argvs[128];
/* Store the number argv pushed. */
int argc;
```

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?

- The process of argument parsing
    1. First, break the command into words: $program\_name\ argv_1\ argv_2\ ...\ argv_n$ using `strtok_r()`. Place the words at the top of the stack. Order doesn't matter, then store them to `argvs`.
    2. Then, do word alignment, make sure the stack pointer used next is the multiple of 4.
    3. Then, push the address of each string(stored in `argvs`) plus a null pointer sentinel, on the stack, in right-to-left order--push `argvs` from `argvs[argc]`(should be a `'\0'`) to `argvs[0]`. These are the elements of `argv`. The null pointer sentinel ensures that `argv[argc]` is a null pointer, as required by the C standard. The order ensures that `argv[0]` is at the lowest virtual address.

4. Then, push `argv` (the address of `argv[0]`, should be current stack pointer + 4) and `argc`.
5. Finally, push a fake "return address" 0.

- Way to arrange the elements of `argv[]` in the right order

  We push the arguments on the stack first, and record addresses. Then scan the list backwards and push these addresses on the stack from above to bottom.

- Way to avoid overflowing

  We don't know how much arguments that user passes to us. So we can not pre-allocate enough space for arguments. So we push to the argument stack until it fails.

Rationale

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

`strtok()` stores the rest string into a static buffer. `strtok_r()` stores the rest string into the given pointer `save_ptr`. It is not safe and not convenient for pintos to use a static buffer for this function. Saving it into `save_ptr` allows us to reuse later for split arguments.

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. Let kernel do less operations and save kernel's time for doing arguments parsing in kernel.
2. Make it safer for kernel. If argument parsing is done in shell, it is done in user mode. Since kernel has more access rights, if argument parsing is done in kernel, the kernel may be under attack and is risky, because some people with ulterior motives can write some specific aggressive arguments to attack the system.
3. Checking if the arguments and executable file exist first can avoid some kernel panic, improve the robustness of the system.
4. Make it more convenient to inform the users when the command typed have errors.

Task 2: 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 `thread.h`:

```
# Inside struct thread
tid_t parent_tid;                    /* tid for parent */
struct list child_thread_list;        /* Status list for children */

struct file *running_file;           /* The file this thread loaded from */
struct list fd_list;                 /* List of file descriptors. */
int file_num;                        /* Number of files opened. */

struct semaphore load_sema;          /* Semaphore for loading executable */
int load_state;                      /* State if loading success */
```

```
/* Element of child_thread_list inside struct thread */
struct child_status
{
  tid_t child_tid;                           /* Child tid */
  int child_exit_code;                       /* Child exit status */
  bool child_waited;                         /* If the child has been waited */
  struct semaphore child_wait_sema;          /* Parent should wait this sema */
  struct list_elem child_elem;               /* Used in child_list */
};
```

```
# Enumeration for load_state inside struct thread
enum load_status
{
  LOAD_INIT,
  LOAD_SUCCESS,
  LOAD_FAIL
};
```

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 open file is corresponding to a unique file descriptor like a person corresponding to his ID number. The open files are stored in a list in the thread opening it, which is unique within a single process, thus file descriptors are unique within a single process and not the entire OS.

Algorithms

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

- For reading

  Syscall: `read(int fd)`
  If `fd` is `STDOUT_FILENO (1)`, return -1 because it does nothing about reading.
  If `fd` is `STDIN_FILENO (0)`, we call `input_getc()` to read std input from console to buffer and return the size.
  Else, we search the file by `fd` in `fd_list` in the current thread. If we find it we aquire the lock and call `file_read()` from filesys to ensure the operation of changing files is atomic.

- For writing:

Syscall: `write(int fd)`
If `fd` is `STDIN_FILENO (0)`, return -1 because it does nothing about writing.
If `fd` is `STDOUT_FILENO (1)`, we call `putbuf()` to read std input from console to buffer and return the size.
Else, we search the file by `fd` in `fd_list` in the current thread. If we find it we aquire the lock and call `file_write()` from filesys to ensure the operation of changing files is atomic.

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?

1. For 4096 bytes of data:
   - If all data belong to a single page that is mapped, then only one inspection is needed. **Thus the minimum number is 1.**
   - If data is split into multiple pages that are mapped, then more than one inspections are needed. However, since data are contiguous, no more than 2 inspections are needed. **Thus the maximum number is 2.**
2. For 2 bytes of data:
   - Similar to the above situation, **the minimum number is 1 and the maximum number is 2**, although it is more likely to only inspect for once.
3. No, they've reached the limit.

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

When a child thread is created, a `struct child_status` will also be created and initialized, and note that the semaphore `child_wait_sema` will be initialized to 0.

The "wait" system call will be handled by `syscall_handler()` and will be dispatched to function `wait()` if it is safe to do so. Then `wait()` calls `process_wait()`, which tries to find the child thread by identifying child's tid. If no legal child is found or the found child is already being waited, return -1, otherwise, invoke `sema_down()` on `child_status`'s `child_wait_sema`, and the parent (current thread) will be blocked since the initial value of `child_wait_sema` is 0.

When the child process terminates, `process_exit()` is invoked and it tries to find the parent thread by invoking `thread_get_by_tid()`, and then find the corresponding `child_status` by searching child's tid. Once the `child_status` is found, invoke `sema_up()` on its `child_wait_sema` to wake up the parent thread, and the parent thread will return child's exit status, which is stored in `child_exit_code`.

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 implement two functions to validate pointer address: `validate_addr()` and `validate_string()`. The first function checks if each byte of the given pointer (4-byte) belongs to user virtual address and if that page is mapped to the current thread. The second function checks if all address within the string (before `'\0'`) are valid.

The `syscall_handler()` function validates stack pointers for all arguments by invoking the above two function and will then dispatch the system call to corresponding functions if both are validate. We also modify `exception.c` to further check address validity. It will check if the process pretends to be a user process while providing a kernel address.

All resource will be freed eventually since we free them in `process_exit()`, which is assured to be a function that will be invoked no matter what had happened. In case of error occurrences, `exit(-1)` will be invoked, which will then invoke `process_exit()` to free the resource.

For example, if a pointer that only the last byte exceeds the user boundary, we would check all bytes of that pointer be invoking `validate_addr()` on each address. This will eventually find the problem and calls `exit(-1)`.

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"?

When there is an "exec" system call, the `syscall_handler()` would call `exec()` which further calls `process_execute()` if it is safe to do so. `process_execute()` would invoke `load()` function to actually load the executable file. Note that during syscall handling, a `sema_down()` would be invoked on current thread's `load_sema`, which is initialized to 0. That means, the current (parent) thread would be blocked until the `load_sema` becomes non-negative (by `sema_up()`), which would happed after the `load()` function of the child thread is finished and the load result is returned to the current (parent) thread.

The load result is passed to current (parent) thread's `load_status` variable, which is initialized to `LOAD_INIT` and would be modified to `LOAD_SUCCESS` or `LOAD_FAIL` in `start_process()`. Since the current (parent) thread is blocked during loading the new executable, the rest of the code inside `exec()` can be executed only after the load result is clear. Thus, finally, `exec()` returns the result, -1 if fail or pid if success.

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?

1. P calls `wait(C)` before C exits

   The `child_wait_sema` is initialized to 0. When `wait(C)` is called, that semaphore would block process P due to `sema_down()` inside `process_wait()`. When C exits, `sema_up()` would be invoked on the corresponding `child_wait_sema` inside `process_exit()` and that `child_wait_sema` would unblock process P, then process P is free to continue.

2. P calls `wait(C)` after C exits

   The `child_wait_sema` is initialized to 0. When C exits, `sema_up()` would be invoked on the corresponding `child_wait_sema` inside `process_exit()` and modify it to be 1, when `wait(C)` is called, that semaphore would become 0 due to `sema_down()` inside `process_wait()`, which would not block process P and process P is free to continue.

3. Free resources

   Every thread has its own `child_status` and `child_thread_list` structs and will be freed when that thread is exiting. Since we assure `process_execute()` would be invoked at every end, those two structs would always be freed. As for other resources, they will be freed like `child_status` and `child_thread_list`.

4. P terminates without waiting, before C exits

   P's `child_status` and `child_thread_list` would be freed when P exits, and C can never be waited since it is impossible to find it's corresponding `child_wait_sema`. Therefore, C will eventually exit and its `child_status` and `child_thread_list` will be freed as well. As for other resources, they will be freed like `child_status` and `child_thread_list`.

5. P terminates without waiting, after C exits

   Since C has already exited, C's `child_status` and `child_thread_list` are been freed before P terminates, and P is free to exit and free its resources (`child_status` and `child_thread_list`) since it is not blocked. As for other resources, they will be freed like `child_status` and `child_thread_list`.

Rationale

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

Simply validating the pointer before accessing it is a direct solution for bad pointer checking. Whenever a pointer is about to be dereference, check it. By doing this, we can avoid all bad-pointer problems.

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

It is very convenient to search open files by `fd` in a given thread.

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

If the mapping is changed, it means one process may contain more than one thread, or one thread may contain more than one process. However, since Pintos reference only requires one to one mapping between processes and threads, we did not change the mapping.

Contributions

Cunhan You:

- Argument Passing
- Syscalls corresponding to filesys:
  - `create`
  - `remove`
  - `open`
  - `filesize`
  - `read`
  - `write`
  - `seek`
  - `tell`
  - `close`

Junda Shen:

- Improving the robustness of the system
  - `validate_addr`
  - `validate_string`
  - Handling exception
- Syscalls corresponding to process:
  - `halt`
  - `exit`
  - `exec`
  - `wait`