Project 2: User Programs

Preliminaries

> Fill in your name and email address.

Chang Shi 2300017795@stu.pku.edu.cn

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

I implement project 2 based on project 1, and I use priority scheduling in my implementation.

> 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.

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.

1. Add a struct `args` to pass arguments and other information (TID of its parent and whether successfully loading) from `process_execute` to `start_process`.

   ```
   struct args{
     char* fn_copy;
     tid_t parent_tid;
     int success;
   };
   ```

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?

I deal with argument passing in function `load`.
First I use `strtok_r()` to separate the tokens apart and count `argc`, before which I set up a new page `argv` in order to record the tokens'addresses. During that process, `cmdline` is modified that each token is ended with `\0`.
Then I deal with `esp` alignment and use `memcpy` to copy the modified `cmdline` to the stack and update the tokens' new addresses (on stack).
Next I use `memcpy` again to copy `argv` to the stack, and sequentially put `argv`, `argc` and fake return address `0` onto the stack.
Due to the operation sequence, the elements of `argv[]` are always in good order.
I check `esp` every time I modify it, to ensure avoiding overflowing the stack page.

RATIONALE

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

`strtok()` is not thread-safe because it uses a static internal pointer to keep track of the tokenization state. If multiple threads call `strtok()` concurrently, this shared state can lead to race conditions and incorrect behavior.
`strtok_r()` is the reentrant version of `strtok()`, where the state is maintained in a caller-provided pointer (`saveptr`) rather than a static variable. This makes it safe for use in multithreaded environments.

> 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. Greater flexibility. The Unix approach allows different shells to implement different parsing rules, without modifying the kernel.
2. Safer. The shell is a user-space program, so when it goes wrong, it wouldn't hurt the kernel.
3. Low kernel complexity. This design makes the kernel as simple as possible. The kernel has less functions that are not so important.

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.

1. Add `exit_status` in `struct thread` in order to record a process's exit status.

2. Add `parent` in `struct thread` in order to record the pid (or tid) of its parent process.

3. Add a vector `all_files[]` in `struct thread` in order to store the mapping from `fd` to `file`.

4. Add a list `dead_children` in `struct thread` in order to record the related status of its children which have exited.

5. Add a semaphore `s` for `process_wait`. A parent process sleep on this semaphore of its child to wait.

6. Add a pointer to a file `exe` in order to record thte process's executable file.

   ```
   struct thread{
     //something
     int exit_status;
     tid_t parent;
     struct file* all_files[MAX_FD];
     struct list dead_children;
     struct semaphore s;
     struct file* exe;
     //something
   }
   ```

7. Add a struct `exec_info` to record exited processes' information and be put in parent processes' `dead_children` list.

```
struct exec_info{
  struct list_elem elem;
  tid_t tid;
  int exit_status;
};
```

8. Add a global constant `MAX_FD`, representing the maximum number of `fd` a process can handle.

```
#define MAX_FD 32
```

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

I create a vector `all_files` in every process to record the mapping from `fd` to the actuall file. Each time a file is to be opened, we allocate a new `fd` and map it to a pointer to the file. Each time a file is to be modified or closed, we find the file according to `fd` and the mapping.

File descriptors are unique just within a single process.

ALGORITHMS

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

First we verify whether the pointers passes from user program are valid. If not, we call `exit` to end the process. We also verify the validness of `fd`, if not we return `0`. Then,
In function `read`: If `fd == 0`, we call `input_getc()` for `size` times. Otherwise, we call `file_read`.
In function `write`: If `fd == 1` we call `putbuf` and return `size`. Otherwise, we call `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?

4096 bytes case: at least 1 time and at most 2 times
2 bytes case: the same as above
Improvement: We may use TLB to make it more efficient, in which design there's usually no need to actually visit memory.

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

It directly calls `process_wait`, and I will describe my implementation of that function.
First, we find the child's information in the current process's `dead_children` list.
If we havn't found, perhaps the child is not our own, or is our own but hasn't exited. Then we call `get_thread`

to get the child process. If we fail, it implies that it's an exited child and is not our own, and we just return `-1`. Otherwise we check if the parent of the child is the current process, if not we just return `-1`. Now we claim that it's our own child and hasn't exited. So the current process sleep on the child's semaphore `s` and update `child_info` from `dead_children` list after awakening.
At this time, we claim that it's our own child and has exited. We get it's `exit_status`, remove `child_info` from the list and release its space, after which the function `process_wait` returns.

> 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 check the validness of a pointer, and if it points to a normal data (occupying 4 bytes), we check `ptr + 3` too. For `buffer`, I believe `file_read` and `file_write` already have mechanisms to avoid error.

All temporarily allocated resources are freed as soon as they are no longer used. In `syscall_handler`, we do not allocate additional resources.

Example: When a pointer passed to the kernel is invalid (no matter it is `esp` or other pointer), it fails `assert_pointer` check, and calls `exit(-1)` to terminate the process.

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"?

First, I set the prority of the new thread to `PRIORITY_DEFAULT + 5`, to ensure that its loading result is returned as soon as possible. Before simulating a return from an interrupt, it sets its prority back to `PRIORITY_DEFAULT`.
There is a flag variable in `argss`, which passes information between the parent thread and the child thread. Initially the flag is set to `0`, after the child thread loads, it is set to `3` if success, `2` otherwise.
After calling `thread_create`, the parent thread disables interruption and checks if the child thread has known its loading status. If not, it blocks itself and wait for its child to unblock it.
After calling `load`, the child thread disables interruption and checks if the parent thread is blocked. If is, it unblocks its parent thread.

> 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?

My implementation of the "wait" system is discusses in B5.

When P calls `wait(C)` before C exits, P sleeps on a semaphore held by C. After P is awakened, C's useful information is already in P's `dead_children` list.
When P calls `wait(C)` after C exits, P directly find information needed in `dead_children` list.

We only discuss the structures for the "wait" system here.
For each process, when it exits, if its parent process has terminated, there wouldn't be any process waiting for it, so there's no need to allocate space for its information.
If its parent process hasn't terminated, it allocates space to store its information, and its parent is supposed to free the resources when it's about to exit.

When P terminates without waiting before C exits, nothing special would happen, and there's no need for C to allocate additional space.
When P terminates without waiting after C exits, it silently free all the resources related in `dead_children` list.

I haven't come up with any special case to hack my algorithm.

RATIONALE

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

I choose to check whether each pointer passed to the kernel is valid or not by calling `pagedir_get_page`.
I think it's simple although somewhat inefficient.

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

Advantages: My design is simple and efficient, and it has realized the demanded functionality.
Disadvantages: The file descriptors in my design couldn't be reused.

> 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 don't have changed it.

Grading

All 80 tests passed in my local environment.