Project 2: User Programs

Preliminaries

> Fill in your name and email address.

Tong WU 2200013212@stu.pku.edu.cn

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

In my local test, my implementation passed all test points.

> 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.

- userprog/process.{h, c}

```
typedef int tid_t; // Defines the type of thread ID

/* The structure for saving parsed arguments. */
struct parsed_cmd
  {
    char file_name[MAX_ARGUMENT_LENGTH+1];
    char argv[MAX_ARG_NUM][MAX_ARGUMENT_LENGTH+1]; // argv[0] is the
file name
    int argc;
    struct semaphore load_finished; // Semaphore to wait for the
process to finish loading
    bool load_success; // Whether the process loaded successfully
  };

/* The struct of a process, which maintains its all information in
need. */
struct process {
    struct thread *thread; // Thread of the process
    tid_t pid; // Process ID, the same as the thread ID
    struct list_elem child_elem; // List element for the child process
    struct list_elem elem; // List element for the list containing all
processes
    struct list child_list; // List of child processes
    struct process *parent; // Parent process
    struct semaphore death;   // Semaphore for waiting for the process
to finish
    int exit_status; // Exit status of the process

    /* Members below are for maintaining file descriptors. */
    struct list open_files; // list<open_file> for the process
    int fd_count; // Number of open files in the process
};

struct list all_process; // List of all processes
```

ALGORITHMS

> A2: Briefly describe how you implemented argument parsing. How do you arrange for the elements of argv[] to be in the right order?

I implement argument parsing mainly in the function `parse_cmd()` in `process.c`. Specifically, I define the `struct parsed_cmd` to carry filename and parsed arguments, which is also passed to child process in `thread_create()`. In `parse_cmd()`, I iteratively use `strtok_r()` to seperate filename and arguments from the input and load them into the return `parsed_cmd` structure. Also, `argc`

records #arguments and `char[] argv` carries all arguments, so that arguments are guaranteed to be in the right order.

> How do you avoid overflowing the stack page?

Later, `start_process()` loads the executable file from and pushed all required arguments into the new process's stack. We note here that only one `struct parsed_cmd` is passed from the parent process to its child, which takes no more than one page of memory, thus overflowing of the stack page is avoided.

Also, we carefully follow the instruction in the PintOS documentation to ensure the correctness of memory layout in the stack. We also implement synchronization with its parent process, as well as check for loading.

RATIONALE

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

`strtok_r()` is the reentrant version of `strtok()`. In other words, `strtok()` keeps the string remained in a static variable, which may bring risks in multithread programming scenario. On the contrary, `strtok_r()` saves the string remained into `saved_ptr`, which is a pointer provided pointer, thus ensuring the safety when there are multi threads.

> 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. The shell's functionality should be concise and specific. The process of parsing and checking can be easily realized by user programs, such as the shell, which reduces the burden of kernel design.
2. The kernel memory space is limited compared to user space, especially considering kernel's complexity. Parsing arguments in kernel may cause memory fragments and unnecessary consumption of kernel space.

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.

```
struct open_file {
    int fd; // File descriptor
    struct file *file; // File pointer
    struct list_elem elem;
};

struct lock global_file_lock; // The global lock for accessing files

// A macro to check the condition in expr., exit(-1) if it is not satisfied.
#define CHECK(expr) \
  if (!(expr)) {      \
    thread_exit();  \
  }

/* Type definition of syscall handler. */
typedef void syscall_handler_t (struct intr_frame *f);

/* Slots for syscall handlers. */
static syscall_handler_t *syscall_handlers[MAX_SYSCALL_NUM];

/* #args for syscall handlers. */
static int syscall_argc[MAX_SYSCALL_NUM];
```

> 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 associated a open file with a unique descriptor when the file is opened. In my implementation, I define `struct open_file` to record the file descriptor and the pointer to the file, which is linked into a list in each process. For every process, the `open_file` can be found given its file descriptor with the utility function `get_open_file()`.

File descriptors are unique just within a single process, but for different processes there may exist file descriptors of the same value, which represent different open files.

ALGORITHMS

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

I adopt the first (simpler) implementation of handling user data in the PintOS document, which means check whether the buffer, pointer of string pointer provided by the user is legal (totally in user memory space, non-NULL pointers, existence in the

thread's page directory) in advance of the syscall. Specifically, I define following check functions and macros to support conveniently checking arugments buffer and exiting once there is an error.

```
// A macro to check the condition in expr., exit(-1) if it is not satisfied.
#define CHECK(expr) \
  if (!(expr)) {      \
    thread_exit();  \
  }

/* Check if ptr is valid */
static bool
valid_ptr(const void *ptr) {
  return ptr != NULL && is_user_vaddr(ptr) &&
pagedir_get_page(thread_current()->pagedir, ptr) != NULL;
}

/* Check if buffer: [ptr, ptr+size) is valid */
static bool
valid_buffer(const void *ptr, size_t size) {
  return valid_ptr(ptr) && valid_ptr(ptr + size - 1); //TODO: check every
page
}

/* Check if str is valid */
static bool
valid_str(const char *str) {
  for (char *p = (char*)str;;) {
    if (!valid_ptr(p)) {
      return false;
    }
    // We should first carefully ensure p is still valid, before
dereferencing it.
    if (*p == '\0') {
      break;
    }
    p++;
  }
  return true;
}
```

> 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?

For question 1, my implementatiom requires at least one inspection (for the whole buffer) and at most 4096 inspections (if there is a string pointer taking 4095 bytes of

space).

For question 2, my implementatiom requires at least one inspection (for the whole buffer) and at most 2 inspections.

There is room for improvement in the maximum of inspections. If we implement the second way mentioned in the document, where cheking is carried out in the page fault handler, it can take no inspection

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

First, the caller(parent process) checks whether the pid belongs to one of its children. Second, it waits for the child process by calling `sema_down(child->death)`, which means it will be waken up only when its child terminates. Third, once the caller waits up, it gets its child's exit status and clears the child process's memory remained.

> 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.

As I mentioned above, I define handy check functions and macros to support conveniently checking arugments buffer and exiting once there is an error.

Also, I register the size of arguments and the corresponding handler for each syscall. When such a syscall arrives, I first check the argument buffer according to its size, then dispatch to its handler, where the string pointers other arguments (e.g. the file descriptors) are validated. For example, for `syscall open`, I first register that it has 1 argument in `syscall_init()`:

```
register_syscall_handler (SYS_OPEN, syscall_open_handler, 1);
```

Then in `syscall_handler()`, the argument buffer is checked before the syscall is dispatched to `syscall_open_handler()`

```
/* The general syscall handler that dispatches the syscall according to the
intr. no. */
static void
syscall_handler (struct intr_frame *f)
{
  // printf ("system call!\n");
  CHECK (valid_buffer(f->esp, sizeof(int)));
  int intr_no = *(int*)f->esp;
  CHECK (intr_no >= 0 && intr_no < MAX_SYSCALL_NUM &&
syscall_handlers[intr_no] != NULL);
  CHECK (valid_buffer(f->esp, sizeof(int) * (syscall_argc[intr_no] + 1)));
  syscall_handlers[intr_no] (f); // Dispatch to the appropriate syscall
handler
}
```

Lastly, in `syscall_open_handler()`, remaining checks are conducted:

```
CHECK(valid_str(*(const void**)(f->esp + 4)));
```

Once an error is detected in the checks above, the `CHECK` macro will force the current thread to exit, which calls `process_exit()` to release its page, open file table, the space allocated to the thread pointer, and the space allocated to the process pointer remains to be swept by its parent 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"?

I add a semaphore called `load_finished` and a flag called `load_success` in the parsed_cmd struct, which is an argument passed to the created process. When the child process tries to load the executable file, the parents wait for it owing to the semaphore. Once the parent process wakes up when the loading is finished, it checks the `load_success` flag. If true, it adds the child to its child list and temporary buffers. Otherwise, it first releases buffers and then reports the error.

> 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?

In my implemenetation, P first searchs the pointer to C, which is called `child` in its child list. If `child` does not exits, which means an illegal input or C has died, P returns with error. Only when P ensures C is still running, it waits for C by called `sema_down(child->process)`. In any scenario above, C will release all its resources when it exits, apart from the process pointer, which should be obtained by P to get its exit status and swept then.

RATIONALE

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

I check user memory in advance because it's simple and easy to implement. However, handling them in a page fault handler may be a little more efficient, since user memory is legal in common scenarios.

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

Adj:

- Simple. Fd is directly correlated with the pointer to the file in a struct variable.
- It is easy to going through all open files for each process.
- Memory-saving. A new struct gets its space only when a new file is opened.

Dis:

- Finding a file with its corresponding fd is a bit time-consuming, since we have to go through a process's all open files. If the design is like that pointers to opening files are saved in an array indexed by the fd, the process may be more efficient.

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

In fact, identity mapping is adopted in my implementation owing to its simplicity. But I guess if we orgnize the process id in some way that the parents can find their child explicitly via the pid (and vice versa), it may be beneficial to the whole OS's efficiency, since we do not need to record the parent and child for every running process.