Project 2: User Programs

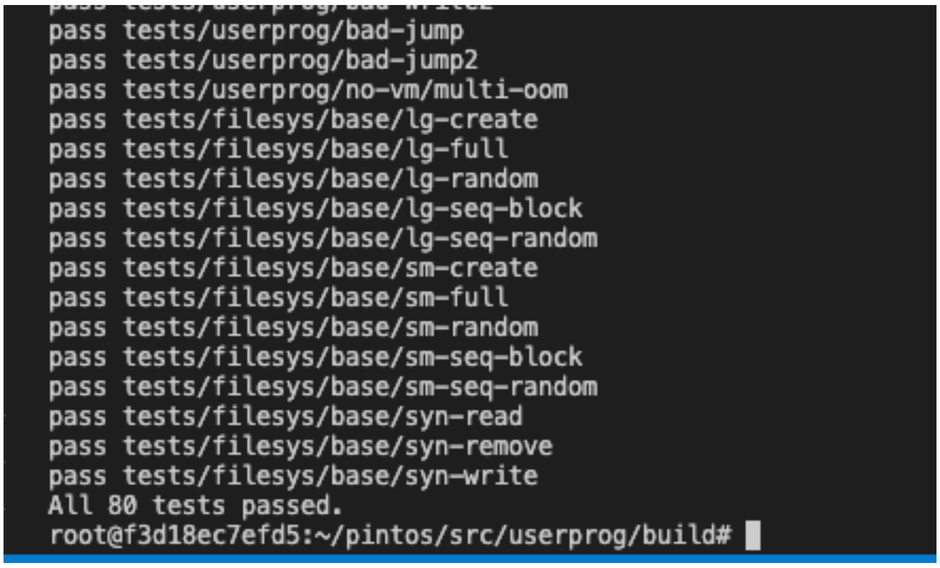
Preliminaries

> Fill in your name and email address.

Jianan Ji 2000012995@stu.pku.edu.cn

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

1. I got all 80 tests passed on my device.



```
pass tests/userprog/bad-write2
pass tests/userprog/bad-jump
pass tests/userprog/bad-jump2
pass tests/userprog/no-vm/multi-oom
pass tests/filesys/base/lg-create
pass tests/filesys/base/lg-full
pass tests/filesys/base/lg-random
pass tests/filesys/base/lg-seq-block
pass tests/filesys/base/lg-seq-random
pass tests/filesys/base/sm-create
pass tests/filesys/base/sm-full
pass tests/filesys/base/sm-random
pass tests/filesys/base/sm-seq-block
pass tests/filesys/base/sm-seq-random
pass tests/filesys/base/syn-read
pass tests/filesys/base/syn-remove
pass tests/filesys/base/syn-write
All 80 tests passed.
root@f3d18ec7efd5:~/pintos/src/userprog/build#
```

2.

> 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.

Nothing in this part.

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?

In `process_execute()` function, I use `strtok_r()` to get the name of the file, which is passed then with the command to `start_process()`.

The argument parsing process mainly happen in `start_process()` function. If loading filename fails, it will call `thread_exit()`. Otherwise, it will start to parse arguments.

Firstly, it will push argument values from the top of the user vm range. Then it will round esp down to multiples of 4. Thirdly the pointers of those values will be pushed in reversely, which are stored in an array when the values are being pushed in. Lastly, argv location, argc and fake return address will be pushed one by one.

After that, it will start the user process by simulating a return from an interrupt.

RATIONALE

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

strtok_r() is a thread-safe version of strtok().

strtok() is not thread-safe because it uses a static buffer internally to store the state from the last call. In a multi-threaded program, this may lead to data races and unpredictable results.

In contrast, strtok_r() is a reentrant version of strtok() that can be safely used in a multi-threaded environment. It avoids the use of the static buffer by receiving an additional parameter (save_ptr), allowing each thread to have its own independent state. This way, when multiple threads use strtok_r() simultaneously, data races will not occur.

> 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. By handling command separation in user-space, Unix-like systems limit the potential damage that can be caused by bugs or security vulnerabilities in the shell.
2. It can leave more resources in the kernel for more critical tasks.

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.

```
/*Child thread*/
struct child_thread
{
    tid_t tid;                      /*Child thread id*/
    int exit_status;                /*Child thread exit status*/
    bool is_alive;                  /*Child thread is alive or not*/
    bool is_waited;                 /*Child thread is waited or not*/
    struct semaphore sema;          /*Child thread semaphore*/
    struct thread *self_t;                 /*Child thread parent*/
    struct thread *parent;              /*Child thread parent*/
    struct list_elem elem;        /*Child thread list element*/
};
/*files of a thread*/
struct thread_file{
      int fd;                         /*file descriptor*/
      struct file *file;              /*file pointer*/
      struct list_elem elem;          /*file list element*/

};
struct thread {

  ...

  /* Added by me in Lab2*/
  struct list child_list; /**< List of child threads. */
  struct child_thread* self_as_child; /**< Self as child thread. */
  struct thread* parent; /**< Parent thread. */
  struct semaphore sc_exec_sema; /**< Semaphore for exec. */
  bool child_exec_success; /**< Child exec success. */
  struct list files_list; /**< List of files. */
  int next_fd; /**< Next file descriptor. */
  struct file* executable_file; /**< Executable file. */

  ...

};
```

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

When a process opens a file or creates a new one, the operating system assigns a file descriptor to that file. A file descriptor is simply a non-negative integer that serves as a handle or reference to the open file. The process can then use this file descriptor to perform various operations on the file, such as reading or writing data, seeking to a specific position, or closing the file.

Just within a single process.

ALGORITHMS

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

Both syscall_read and syscall_write functions have a similar structure, and they each take a pointer to an `intr_frame` structure as an argument. The functions then perform some pointer checking to ensure that the provided arguments are valid and do not cause any memory access violations.

In syscall_read():

1. The file descriptor (fd), buffer pointer (buf), and size are extracted from the stack.
2. If fd is 0 (standard input), the function reads from the input and stores the characters in the buffer (buf) one by one, updating the return value (eax) to the number of characters read.
3. If fd is not 0, the function locates the corresponding file structure associated with the file descriptor using the `get_thread_file` function.
4. If the file structure is found, the function acquires a lock on the file system, performs a read operation using `file_read`, updates the return value (eax) with the number of bytes read, and releases the lock.

In syscall_write():

1. The file descriptor (fd), buffer pointer (buf), and size are extracted from the stack.
2. If fd is 1 (standard output), the function writes the buffer (buf) contents to the output using `putbuf` and updates the return value (eax) to the number of characters written.
3. If fd is not 1, the function locates the corresponding file structure associated with the file descriptor using the `get_thread_file` function.
4. If the file structure is found, the function acquires a lock on the file system, performs a write operation using `file_write`, updates the return value (eax) with the number of bytes written, and releases the lock.

Both handlers perform proper error checking, returning -1 in eax when an error occurs (e.g., invalid file descriptor, null file structure, etc.).

```c
static void syscall_read(struct intr_frame* f) {
  pointer_checker(f->esp + 1, sizeof(int), 0);
  pointer_checker(f->esp + 2, sizeof(int), 0);
  pointer_checker(*((unsigned*)f->esp + 2), *((unsigned*)f->esp + 3), 2);
  pointer_checker(f->esp + 3, sizeof(int), 0);
    int fd = *(int*)(f->esp + sizeof(void*));
    char* buf = *(char**)(f->esp + 2 * sizeof(void*));
    int size = *(int*)(f->esp + 3 * sizeof(void*));

    //printf("fd: %d, buf: %s, size: %d\n", fd, buf, size);

    if (fd == 0) {
        int i;
        for (i = 0; i < size; i++) {
            buf[i] = input_getc();
        }
        f->eax = size;
    } else {
        struct thread_file *tf = get_thread_file(fd);
        if (tf == NULL) {
            f->eax = -1;
            return;
        }
        lock_acquire(&filesys_lock);
        f->eax = file_read(tf->file, buf, size);
        lock_release(&filesys_lock);
    }
}
static void syscall_write(struct intr_frame* f) {
    pointer_checker(f->esp + 1, sizeof(int), 0);
    pointer_checker(f->esp + 2, sizeof(int), 0);
    pointer_checker(*((unsigned*)f->esp + 2), *((unsigned*)f->esp + 3), 2);
    pointer_checker(f->esp + 3, sizeof(int), 0);
    int fd = *(int*)(f->esp + sizeof(void*));
    char* buf = *(char**)(f->esp + 2 * sizeof(void*));
    int size = *(int*)(f->esp + 3 * sizeof(void*));

    //printf("fd: %d, buf: %s, size: %d\n", fd, buf, size);

    if (fd == 1) {
        putbuf(buf, size);
        f->eax = size;
    }
    else {
        struct thread_file *tf = get_thread_file(fd);
        if (tf == NULL) {
            f->eax = -1;
            return;
        }
        lock_acquire(&filesys_lock);
        f->eax = file_write(tf->file, buf, size);
        lock_release(&filesys_lock);
    }
}
```

> 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. Full page (4,096 bytes) of data copied from user space into the kernel:

- Least number of inspections: If the user-space buffer is page-aligned and occupies a single page, it would require only one inspection of the page table, as the entire buffer falls within the same page.
- Greatest number of inspections: If the user-space buffer starts at the last byte of a page and spans across multiple pages (4095 bytes in the next page), it would require two inspections of the page table, one for each page.

1. Only 2 bytes of data copied from user space into the kernel:

- Least number of inspections: If both bytes are within the same page, it would require only one inspection of the page table.
- Greatest number of inspections: If the first byte is at the end of one page and the second byte is at the beginning of the next page, it would require two inspections of the page table, one for each page.

In both cases, the room for improvement would come from optimizing the memory access pattern or memory management implementation. For instance, using a more efficient data structure for the page table or caching page table entries could reduce the number of inspections needed. The actual improvement depends on the specific memory management implementation and the nature of the memory access patterns in the system calls.

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

1. `syscall_wait`: This function is the system call handler for the "wait" system call. It takes a pointer to an `intr_frame` structure as an argument. It performs pointer checking to ensure the provided argument is valid and does not cause any memory access violations. Then, it extracts the child process's thread ID from the stack and calls the `process_wait` function with the extracted tid. Finally, it sets the return value (eax) to the exit status returned by `process_wait`.
2. `process_wait`: This function takes a thread ID (tid) as an argument and checks if the thread with the given tid is a direct child of the calling process. If it's not a direct child or tid is TID_ERROR, the function returns -1. If the child_tid is valid, the function iterates through the child list of the current thread (parent process) to find the corresponding child thread structure.

For the first child thread structure found:

a. If the child_tid matches and the child process has already been waited upon (`is_waited` is true), the function returns -1.

b. If the child_tid matches and the child process is not alive (`is_alive` is false), it means the child has already terminated, so the function removes the child thread from the child_list and returns the exit status of the child.

c. If the child_tid matches, the child process is alive, and it hasn't been waited upon, the function sets the `is_waited` flag to true, and then it uses a semaphore (`sema_down`) to block the parent process until the child process terminates. Once the child process terminates, the function retrieves the exit status, removes the child thread from the child_list, and returns the exit status.

If no matching child_tid is found in the child_list, the function returns -1.

> 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.

I use a function `pointer_checker()` at the begining of every syscall function to check the validity of a user-provided pointer and its memory range before accessing it in a system call. As for the resources-free problem, I will carefully check every resource used and free them whenever an error my occur or the function returns.

For example, in my systemically_write function, pointer_checker function will be called fourth to check safety.

```
static void syscall_write(struct intr_frame* f) {
    pointer_checker(f->esp + 1, sizeof(int), 0);
    pointer_checker(f->esp + 2, sizeof(int), 0);
    pointer_checker(*((unsigned*)f->esp + 2), *((unsigned*)f->esp + 3), 2);
    pointer_checker(f->esp + 3, sizeof(int), 0);
    int fd = *(int*)(f->esp + sizeof(void*));
    char* buf = *(char**)(f->esp + 2 * sizeof(void*));
    int size = *(int*)(f->esp + 3 * sizeof(void*));

    //printf("fd: %d, buf: %s, size: %d\n", fd, buf, size);

    if (fd == 1) {
        putbuf(buf, size);
        f->eax = size;
    }
    else {
        struct thread_file *tf = get_thread_file(fd);
        if (tf == NULL) {
            f->eax = -1;
            return;
        }
        lock_acquire(&filesys_lock);
        f->eax = file_write(tf->file, buf, size);
        lock_release(&filesys_lock);
    }
}
```

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"?

1. In `syscall_exec()`, the function `process_execute()` is called with the command line as an argument.
2. In `process_execute()`, copies of the command line are created in `fn_copy` and `fn_copy_`. This is necessary to avoid race conditions between the calling thread and the `load()` function. The function then extracts the file name from the command line and creates a new thread to execute the file using `thread_create()`.
3. The calling thread waits for the child thread to signal whether the executable has been successfully loaded or not, using a semaphore `sc_exec_sema`. The calling thread blocks at `sema_down(&thread_current()->sc_exec_sema)` until the child thread signals it.
4. In the child thread's `start_process()` function, the `load()` function is called to load the executable. If loading is successful, the child thread sets `child_exec_success` to `true` in its parent thread. If loading fails, it sets `child_exec_success` to `false`.
5. The child thread then signals the calling thread using `sema_up()`, unblocking the calling thread and allowing it to continue execution.
6. Back in `process_execute()`, the calling thread checks the value of `child_exec_success`. If it's `false`, it returns -1, indicating that loading the new executable has failed. If it's `true`, it returns the TID of the newly created 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?

The synchronization and avoidance of race conditions between parent process P and child process C are achieved through the use of semaphores and data structures.

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

- In the `process_wait()` function, the parent process iterates through its `child_list` to find the child process with the matching `tid`.
- If the child is found and is still alive, the parent sets `is_waited` to `true` for that child and then blocks on the child's semaphore `sema` using `sema_down(&(c->sema))`.
- When the child process exits, it signals the parent by calling `sema_up()` on the same semaphore. The parent then resumes execution, collects the child's exit status, and removes the child from the `child_list`. This ensures proper synchronization between the parent and child and avoids race conditions.

1. When P calls `wait(C)` after C exits:

- In the `process_wait()` function, if the child process is found in the `child_list` and is not alive, the parent process directly collects the exit status and removes the child from the `child_list`, freeing the resources associated with the child.

1. When P terminates without waiting, before C exits:

- In the parent's `process_exit()` function, the parent should iterate through its `child_list` and release any resources associated with its child processes. This can include removing the child processes from the `child_list`, deallocating any memory associated with them, and releasing any held semaphores.
- The child processes can be assigned to another process (e.g., the init process) to ensure they are properly reaped when they eventually exit.

1. When P terminates without waiting, after C exits:

- If the parent process terminates without waiting for the child, and the child has already exited, the child process's resources should have been released when it exited. The only remaining resource to handle would be the child's entry in the `child_list`. This can be addressed in the same way as the previous scenario (when P terminates without waiting, before C exits).

RATIONALE

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

I use struct lock filesys_lock to ensure thread safety by acquire it before accessing the memory and release after that. I also use former pointer_checker at the begining of every syscall function to check the validity of a user-provided pointer and its memory range before accessing it in a system call.

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

Advantage: By employing a linked list for storing file descriptors, it becomes simple to add and remove file descriptors.

Disadvantage: Searching for a file descriptor using the file pointer is not straightforward, as it requires traversing the entire linked list to locate the desired file descriptor.

> 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 didn't change it. After weighing potential advantages against the increased complexity that may come with implementing and maintaining a custom mapping, I think it is already convenient enough and can achieve the goals of the projects to change nothing here.