# Project 2: User Programs

## Preliminaries

> Fill in your name and email address.

Sun Shaocong 2000012977@stu.pku.edu.cn

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

None.

> 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.

About overall architecture of system calls:

- https://github.com/mit-pdos/xv6-riscv/blob/riscv/kernel/syscall.c

About files struct of a thread:

- https://blog.csdn.net/mediatec/article/details/86735691

## 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.

Definations:

```
#define ALIGNMENT 4                         /* Alignment. */
#define DECREASE_PTR(ptr,bytes)             /* Decrease PTR by BYTES. */
#define PUT_4BYTES(ptr,value)               /* Put a 4 bytes VALUE at PTR. */
#define ALIGN(ptr)        /* Round PTR down to a multiple of ALIGNMENT. */
#define OVERFLOW(cur_esp,init_esp)          /* Stack overflow? */
#define TEST_AND_PUSH(ptr,value,init_esp)
/* Push a 4 bytes VALUE on stack PTR,but check before push. Return whether push
succeeds. */
```

### 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?

- If a command line has B bytes, the total length of all arguments(include '\0') is at most B+1. So, after a return from load, I save initial if_.esp as save_esp and decrease it by B+1 to leave enough space for the argument strings. Use save_esp to push argument strings while pushing the address of each string through if_.esp. Make alignment and store a NULL pointer at if_.esp. Each time we get a token out of command line, decrease save_esp by it's length(include '\0') and copy the token string to save_esp, after that push the value of save_esp(string address) at if_.esp. When all tokens are processed, argument strings are copied to user stack and their addresses are stored in a left-to-right order. Argv is current if_.esp and argc can be counted before. To make the elements of argv[] in a right order, just **reverse the argv array**. The rest is relatively simple. Push argv, argc, "return address" at if_.esp and finally done argument passing.

- Another thing is about how to avoid overflowing the stack page. As you may notice, I define TEST_AND_PUSH rather than PUSH. This inline routine will test stack overflow before truly putting the value to stack. I ensure that whenever to put a value at address A, a lower address B shall be tested some time earlier. So all writing is safe.

**RATIONALE**

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

- Maybe because strtok_r() is more flexible in use. The former one returns save_ptr to the user. The latter one only saves save_ptr in a fixed location. After one call to strtok() on string S, we have to finish all calls on string S because we don't hold any pointer to the rest part of string S. If a call to strtok() on string T break in, we will lose the opportunity to operate on the rest of S.

> 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.

- It will decreases kernel processing time.
- The shell can make some vaildity check before passing command to kernel, reduce the possibility of kernel errors.

## 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.

New struct member in struct thread:

```
/* Used in filesys syscall. */
struct files_struct files;              /**< Files struct. */
/* Used in syscall exec, wait and exit. */
struct thread *parent;                  /**< Current thread's parent. */
/* Used in syscall exec to synchronize during loading. */
bool child_load_success;                /**< Child loading result. */
struct semaphore child_load_sema;       /**< Semaphore for load. */
/* Used in syscall exit and wait. */
bool is_parent_died;                    /**< Its parent died or not. */
```

```
struct list child_list;                 /**< A list of its child. */
struct child *child_msg;                /**< Child struct of this thread
                                         owned by its parent. */
/* Used for deny write to executables. */
struct file *executable;                /**< Executable file. */
```

New definations in thread.h:

```
/** Thread's open file definition. */
typedef int fd_t;                       /**< File descriptor. */
#define FD_ERROR ((fd_t) -1)            /**< Error value for fd_t. */
typedef int fd_set_t;                   /**< Bitmap of opening fds. */
#define INIT_FD_SET 0x3                 /**< Initial fd set. */
#define MAX_FILE 8*sizeof(fd_set_t)     /**< Maximum files a thread can open.   */
#define USER_FD_MIN (fd_t)2             /**< Lowest fd user can open. */
#define USER_FD_MAX (fd_t)(MAX_FILE-1)  /**< Highest fd user can open. */
#define STDIN_FD 0                      /**< STDIN file descriptor. */
#define STDOUT_FD 1                     /**< STDOUT file descriptor. */
/** Three basic operation on a fd set. */
#define IS_FREE_FD(fd_set,fd)           /* Is FD allocatable in FD_SET. */
#define ALLOCATE_FD(fd_set,fd)          /* Add FD to FD_SET. */
#define REMOVE_FD(fd_set,fd)            /* Remove FD from FD_SET. */
```

New structs in thread.h:

```
/** A thread's opening files struct. */
struct files_struct
{
   fd_set_t       fd_set;               /**< Open fd set. */
   struct file    *fd_table[MAX_FILE];  /**< Open fd table. */
};
/** A child struct of a thread. */
struct child
{
   struct thread *t;                    /**< Child thread struct. */
   tid_t tid;                           /**< Child tid. */
   bool is_terminated;                  /**< Child exit or not. */
   int saved_exit_status;               /**< Saved exit status. */
   struct semaphore terminated_sema;    /**< Semaphore for wait. */
   struct list_elem elem;               /**< List elem. */
};
```

New definations and variables in syscall.c:

```
#define SYSNUM 13
#define MAXARGS 4

static bool in_syscall_context;         /**< Are we processing a syscall? */

static struct lock filesys_lock;        /**< A lock for filesys operations. */
static struct lock exit_lock;           /**< A lock for exit call. */

static int (*syscalls[])(uint32_t *argv);   /** Handler list. */
static int _argc[];                     /** Argc list. */
```

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

- An opening file descriptor is recorded in fd_set as a bit "1" for a thread. Specifically, if file descriptor FD is opened, The FD bit of fd_set in thread's files struct will be set to 1 and fd_table[FD] will be pointed to file struct of the open file.
- File descriptors are unique just within a single process.

Following functions are provided for manipulate files struct of a single thread in thread.h:

```
fd_t thread_open_file(struct file*);
int thread_close_file(fd_t fd);
void thread_close_all(void);
struct file *thread_get_file(fd_t fd);
```

#### ALGORITHMS

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

- Reading and writing are implemented based on get_user() and put_user() provided in the Pintos documentation. To make sure these two functions work correctly, page_fault() is also modified according to the Pintos documentation. When page fault detected during system call processing, page fault handler returns control back to syscall handler instead of killing the process. Function syscall_context() in syscall.h returns whether we are processing a syscall.
- There are two kinds of reading(or checking exactly). The function check_user_string() assumes that the user string is terminated with '\0'. It calls is_user_addr() and get_user() on user string byte after byte. If it doesn't detect an error before it encounters '\0', user string is considered legal. Another function check_user_string_bound() is similar, except that check_user_string_bound() finishes checking when it meets length limitation without considering any byte value in user string.
- Only sys_read() requires writing to user address. So it makes user address check while writing to user buffer byte after byte.

> 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?

- In my implementation, the kernel will do this byte after byte. So in the first case, the answer is always 4096. In the second case, it's always 2.
- There is certainly room for improvement. If we know how many bytes to check, we can make this check one time per page. So in the first case, the answer is 1(least) and 2(greatest). The second case is the same.

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

- Consider a situation that a parent thread P creates a child thread C through exec() and wait for C through wait(). When P calls exec(), it will call thread_create() to create thread C. In thread_create(), a child struct is initialized and inserted into P's child_list. In child struct, a semaphore named terminated_sema is initialized to value 0, which will play a key role in the wait process. In a call to wait() from P, P tries to down the terminated_sema related to C. However, the semaphore would only be upped when C calls exit (willingly or unwillingly), where is also the point to save C's exit status in its child struct. So P won't return from sema_down() until C exits. This is exactly what we expect.

> 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.

- The general idea of error handling is that errors caused by bad pointers will terminate the process, and other types of errors will return an error code -1 to the calling process. Errors caused by bad pointers include failure to get syscall number, failure to get syscall arguments and bad addresses provided by user process. For example, a "write" syscall will be considered to cause a bad pointer error if it fails at the points mentioned above. However, the kernel will only return error code -1 if the failure is caused by a bad file descriptor.
- Checking syscall number and arguments are completed uniformly in syscall_handler() through get_arg_n() and get_args(). Then syscall_handler() calls specific handler according to syscall number. All handlers other than sys_read() and sys_write() are quite direct. These handlers just get the arguments from argv[], call for checks on user addresses or file descriptors, then call the corresponding interfaces or handle the error according to the check result. Read and write are a bit more complex, but they follow the same error handling idea. It should be easy to understand following the comments in the function.
- Now let's talk about how to make sure all resources are released. Locks are only used on routines with a single entry and a single exit. So all locks acquired at the entry will be released at the exit. I only use a

buffer in sys_read() and it will be freed on every way out of the function.

#### 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"?

- The "exec" mainly calls process_execute(). All interesting things are in this function. Before a calling to thread_create(), parent thread P will initialize a semaphore called child_load_sema to value 0 in its thread struct. Then P calls thread_create() to create child thread C and may be blocked at a down operation on child_load_sema. Then C gets to run. When C finishes its loading, it will save loading result at child_load_success in P's thread struct and call sema_up on P's child_load_sema to notify it parent. Finally, P gets control again and it can get the load success/failure status from its child_load_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?

- First, let's talk about freeing resources. If P calls wait() on C, things we talked about in B5 ensure wait() will return at right time. And the child struct will be freed after P finishes waiting, which ensures the second call wait() on child C will return -1. If P doesn't call wait() on C, any remaining child struct will be freed when P exits through free_child_list(). So all resources are freed when P exits.

- Then I will talk about synchronizations. The child struct of C is shared between P and C. This struct is created and freed by P, but C can modify it. Only P has the right to access its child list. C can modify the child struct through a pointer called child_msg in its thread struct. But C has to make sure its parents is alive when it wants to make some modifications (otherwise may visit some freed memory). C gets this information through is_parent_died in C's thread struct. P will set C's is_parent_died when P calls free_child_list(). Another problem comes here. P must make sure C is alive when P tries to set C's is_parent_died. So P will check the child struct (owned by P) to know whether C is alive. Well, it's indeed a little complicated...

- In short, both P and C should know whether the other one is alive when they try to modify the other's data. P will check child status and decide whether to set C's is_parent_alive in free_child_list(). C will check is_parent_died and decide whether to modify the child struct. The key to avoiding race is to check the conditions and make corresponding modifications without interruption. So I use a global lock named exit_lock to make a part of _exit() function mutually exclusive. P's check-and-modify and C's check-and-modify both happen in this part.

- Now think about some cases mentioned above. If P waits for C, C exits before P without any race. However, now it's the case P doesn't wait for C. Our exit_lock ensures us to take _exit() as somehow an atom routine. If C exits before P, C is safe to modify its child struct because P hasn't free it. P won't try to modify C's is_parent_died because P knows C has exited. On the other side, P exits before C. P is safe to set C's is_parent_died. C won't try to modify its child struct because P has set C's is_parent_died.

- One special case is that loading failed when P call exec() to create C. But thread_create() has create C's child struct in P's child_list. In this case, C should notify P that C's thread is terminated in start_process().

#### RATIONALE

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

- I choose the second way mentioned in the Pintos documentation. This technique is normally faster because it takes advantage of the processor's MMU. And it can perform read and write operations while checking the address.

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

- File descriptors are unique just within a single process. It's more convenient for a single process to manage its open file in an array. Since I use a int as a fd_set, the maximum open files is 32. Although it seems enough in pintos, maximum 32 open files may not satisfy some big processes.

> 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.