Project 2: User Programs

Preliminaries

> Fill in your name and email address.

Jin Yang 2000012935@stu.pku.edu.cn

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

> I feel ashamed that I didn't find my bugs in process_wait() until I start writing the document. Well, it's fixed now, perhaps. Anyway, the document is based on the latest version instead of the handed one.

> 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.

> The ICS shell-lab.
> The xv6 syscall code.
> Websites about i386 architecture.
> Websites about Linux wait command.

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.

> In process.c

```
typedef union
{
    void *vp;
    char *cp1;
    char **cp2;
    char ***cp3;
    int *ip;
    unsigned u;
    void (**ret_addr)(void);
} esp_t;
```

> This esp_t type avoids type casts, since the stack pointer could be anything.

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?

> How to implement argument parsing?
>
> > Firstly, I separate the first token and the rest in start_process(). The first token is the thread's name and the rest is pass to function arg_pass(), where I separate the argument strings.
> > In arg_pass(), I separate the rest tokens and put them in the array argv[]. Then I add alignment, use memcpy() to push the pointers to argv[]. Finally push argv, argc and fake return address.

> Way of arranging for the elements of argv[] to be in the right order.
>
> > I decrease the esp first, and the use memcpy() to copy the pointers to the right position.

> How to avoid overflowing the stack page?
>
> > I decide not to check the esp until it fails. if overflow happens, it will use an invalid address, causing a page fault exception, which terminates the thread and exit -1. It make sense that to terminate a process with too much arguments.

RATIONALE

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

> The difference of the two functions is that the save_ptr is provided by caller in strtok_r(), while strtok() use a static save_ptr. In pintos, we separate the commands into file name and arguments. We need to put the address of arguments somewhere so we can use later. Also, strtok() is not safe in muti-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. Unix approach Shortens the time inside the kernel and makes the kernel smaller.
> 2. Unix approach is safer since it can checks the arguments outside the kernel, avoiding kernel panic while handling arguments.
> 3. Unix approach is more extendable, since it is much more convenient to write or change codes in a shell.

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

```
struct thread
{
...
#ifdef USERPROG
/* Owned by userprog/process.c. */
uint32_t *pagedir;          /**< Page directory. */
struct process *process; /**< Pointer to process. */
#endif
...
}
```

> Add a pointer to a process.

> In process.h

```
struct process
{
    pid_t pid;             /**< Process identifier. */
    struct thread *thread; /**< Pointer to thread. */
    bool load_success;     /**< Whether successfully load file. */
    int exit_status;       /**< Process exit code. */

    struct list child;      /**< List of child process. */
    bool child_exited;      /**< If the parent needs to free the child. */
    bool parent_exited;     /**< If the parent process exited. */
    struct process *parent; /**< Parent process. */

    struct semaphore sema_exec; /**< Block parent while executing. */
    struct semaphore sema_wait; /**< Block parent while waiting. */

    struct list_elem allelem; /**< List element for all processes list. */
    struct list_elem elem;    /**< List element. */

    int fd;            /**< Max fd of the process. */
    struct list files; /**< Files the process opened. */
    struct file *file; /**< File that the process execute. */
};
```

> A user process to handle process functions

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

> An opened file to hanlde file descriptor and file pointer.

```
typedef int pid_t;
typedef int tid_t;
```

> Pid type and tid type. Tid_t is defined in thread.h, but without typedef here there is an error. I can't figure out why.

> In process.c

```
static struct list all_list;
```

> List of all processes.

```
static struct lock file_lock;
```

> Lock for synchronization in file operation.

> In syscall.c

```
static void (*syscalls[MAX_SYSCALL_NUM])(struct intr_frame *)
```

> A syscall function array, **MAX_SYSCALL_NUM** is 13 in userprog. The idea is from xv6 code.

> 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 use a struct called open_file to store file descriptor(fd) and file pointer(fp). In a process, **fileno()** is used to search for an fd by given fp, while **fdopen()** is used to search for an open_file struct by given fd.
> The file descriptors are unique within single process, since process in pintos is a thread in fact. Threads don't share file descriptors. And it's easier to free memory and recycle resources in each process if file descriptors are unique just with a single process.

ALGORITHMS

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

> Reading:
> Firstly I use **check_valid()** to check whether the arguments are valid. If not valid, it triggers a page fault and returns -1. Then I check the fd. If fd is 0, the STDIN_FILENO, I use **input_getc()** to read from keyboard. For other cases, I use **fdopen()** to find the open_file, which searchs for the open_file by given fd in a process' file list. If it doesn't exist, return -1. If the open_file is found, I acquire the lock **file_lock** first, then call **file_read()** and finally release the lock. Fd 1 is a special case and exits with -1 at once.

Writing:
Similar with reading, check the arguments first. If fd is 1, the STDOUT_FILENO, use **putbuf()** to print to the console. Other cases are almost the same as reading, and the only difference is using **file_write()** to write to the found file. Fd 0 is also a special case and exits with -1.

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?

Both have the greatest 2 if the data is not in one page and the least 1 if the data is in one page. I cannot find a way to improve it.

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

The wait syscall calls process_wait(), so I just talk about process_wait().
In process_wait(), I check if the current thread is initial thread. Then I search for the process in the current thread's child list(for initial thread, the all_list). If it is not found, return -1. If the current thread has a child with the given pid, sema_down() **sema_wait** to wait until its child exits. Then I get the exit_status before free the memory of the child process. Finally return the exit code.
If the child exits normally in sys_exit(), its exit code will be set in sys_exit().
If the child is terminated by kernel, its exit code will not change and the default value is -1.
If the parent waits twice or more, since the resource is recycled, the child will not be found. So the process_wait() returns -1.
If the parent is terminated before the child, process_wait() just frees the memory and then exits.

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.

> First, avoiding bad user memory access is done by **check_valid()**, which checks whether the given address is a user address, checks whether it is mapped and uses **get_user()** to check whether its 4 bits can be accessed. Taking **write(int fd, const void * buf, unsigned count)** as an example, it checks its 3 arguments and it checks *buf to find whether the string is valid. If one is invalid when checking, a page fault occurs and terminates the process with exit code -1.

> Second, when an error happened, I handle it in page_fault handler. Taking bad-jump2 as an example, it attempts to access the invalid address **0xc000000**, triggering a page fault. In page_fault handler, it is found that the address is invalid and it is a bad reference caused by syscall. So the thread is terminated with exit code -1. And I don't have to free the resources since the whole page will be freed after the thread exits.

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"?

> To ensure that, I use a **sema_exec**. After successfully creating a new thread in process_execute(), I **sema_down()** the **sema_exec** to run the child process. After executing the code in start_process(), whether successfully or not, **sema_up()** the **sema_exec** to go back to the parent process.
> To pass back the status, I use a **bool load_success**. It is set to true if load successfully, and false if not.

> 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?

To ensure proper synchronization, I use a **sema_wait**. In process_wait(), I **sema_down()** the **sema_wait** and parent waits until child finishes. And the child resources are freed in process_wait(), so if child fails to load in process_execute(), I **sema_up()** the **sema_wait** to free child's memory. It is also used in process_exit() for the same reason.
So in the cases above:

1. P calls wait(C) before C exits
   This is the most common situation. P just waits until C finishes and wait() returns the exit code of C.
2. P calls wait(C) after C exits
   In process_wait(), it tries to find C and fails. So it returns -1 at once.
3. P terminates without waiting before C exits.
   In my latest version, I set the **parent_exited** for each child when parent exited. If P terminates without waiting before C exits, C will free the resource in **process_exit()**.
4. P terminates after C exits
   Also, I set the **child_exited** when a process exits if its parent hasn't exited yet. When P terminates, P will free C.

RATIONALE

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

I use the second approach. As is said in the pintos web, it is faster and more extendable. However, I can't figure out the way of using **put_user()** by now.

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

I use an **open_file** struct. Each process has a list of open_files and has its own max_fd.

> Advantages:

1. Thread-struct’s space is minimized
2. Each process has its own file descriptors, which makes it easier to extend the function e.g. file mode.

Disadvantages:

1. Consumes kernel space, user program may open lots of files to crash the kernel.
2. Consumes too much time to find fd or fp. Hash table may be a better choice rather than list.

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. It’s reasonable and implementable.