Project 2: User Programs

author: 冯宇凡 fengyf2@shanghaitech.edu.cn

杨 斌 yangbin@shanghaitech.edu.cn

date: 2021/11/13

Consulted Sources:

- 斯坦福大学Pintos Project1、2 指南+总结

  https://zhuanlan.zhihu.com/p/104497182

- pintos-pro2-project2-UserProgram - 西安电子科技大学

  https://wenku.baidu.com/view/4e869bed4531b90d6c85ec3a87c24028905f85c8.html

- CSCI 350: Pintos Guide

  https://static1.squarespace.com/static/5b18aa0955b02c1de94e4412/t/5b85fad2f950b7b16b7a2ed6/1535507195196/Pintos+Guide

- Other guy's implementation

  https://github.com/Wang-GY/pintos-project2

  https://github.com/ChristianJHughes/pintos-project2

- Other guy's design documents

  https://github.com/SpaceCowboy100/cs153/blob/0df246ab7148f353e6c3531caa293e907bac667b/pintos/src/userprog/DESIGNDOC

  https://github.com/isaaclong1/UCR_CS153_PintOS/blob/34b394f8899108397445854679c4d3994e779283/proj2_design_doc.txt

  https://github.com/rictic/tiny-os/blob/f097a57575752e03c140b7a781d3dca4495d7882/userprog/DESIGNDOC

ARGUMENT PASSING

DATA STRUCTURES

> 1. 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.

None.

ALGORITHMS

> 2. 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?

1. For a name that needs to be parse, we use `strtok_r()` to get arguments recursively.
2. In `start_process()`, after `load()` was called, we implement argument parsing and save all arguments on the stack.

   We use `int argc` to count the number of arguments, a local variable `char *argv[25]` to store the arguments. The `argv[]` stores all arguments in normal sequence. Then we put the local `argv[]` on the stack reversely from `argv[len(argv)-1]` to `argv[0]`, while the `esp` of stack decrease the length of `argv[i]` every time.
3. By limiting the maximum length of `char *argv[]` to 25, we can avoid overflowing the stack page.

RATIONALE

> 3. Why does Pintos implement `strtok_r()` but not `strtok()`?

From the official guide: *`strtok()` - Uses global data, so it is unsafe in threaded programs such as kernels.*

The `strtok()` function uses a static buffer while parsing. If one thread is using `strtok()`, it will change the original string. When another thread is also trying to call `strtok()` again, the behavior might change. Therefore, it's not thread safe. However, `strtok_r()` is safe.

> 4. 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. In the sense of maintainability, if we want to change the way of argument parsing in the future, we don't have to modify kernel code as we implement this in shell instead of kernel.
2. Implement argument parsing in shell is more safer than directly interact with kernel. The kernel might crash due to some abnormal arguments, however, a shell crashing maybe will not cause kernel crashing.

SYSTEM CALLS

DATA STRUCTURES

> 1. 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 in `threads/thread.h`:

```
/* Store the exit status and tid for each child thread */
struct child_thread
{
   struct thread *t;                  /* Point to the child thread */
   int tid;                           /* The tid of this child thread */
   int exit_status;                   /* The exit status of this child thread*/
   struct list_elem child_elem;       /* List element for children thread list. */
};
```

New struct members in `threads/thread.h:struct thread`:

```
#ifdef USERPROG
    /* Owned by userprog/process.c. */
    uint32_t *pagedir;                  /* Page directory. */
    struct list child_list;             /* List of children threads. */
    struct semaphore waiting_process;   /* Semaphore of should be waiting for child process. */
    struct semaphore load_sema;         /* Semaphore of loading an execute file */
    struct list files;                  /* List of files opened by this thread. */
    struct file *file;                  /* The executable file of this thread. */
    struct lock child_list_lock;        /* Lock of child_list. */
    struct thread *parent;              /* The parent thread of this thread. */
    int load_success;                   /* The flag of load success or not */
    int child_status;                   /* The status of current waiting child thread. */
    int fd;                             /* The available file descriptor for next file. */
    int exit_status;                    /* The exit status of this thread. */
    int has_exit;                       /* The flag of print exit msg */
#endif
```

New static variable in `userprog/process.c`:

```
/* The target tid when calling `find_thread_by_tid ()`. */
static tid_t target_tid;

/* The matched thread when calling `find_thread_by_tid ()`. */
static struct thread * find_it;
```

New `typedef` in `userprog/syscall.h`:

```
/* Process id number pid for convenience. */
typedef int pid_t;
```

New `struct` in `userprog/syscall.h`:

```
/* A file possessed by a thread (user program). */
struct thread_file
{
    struct file *f;                  /* The file. */
    struct list_elem f_listelem;     /* List element for file list of thread. */
    int fd;                          /* The file descriptor. */
};
```

New define in `userprog/syscall.c`:

```
/* Virtual memory address limit. */
#define VADD_LIMIT 0x08048000
```

New static variable in `userprog/syscall.c`:

```
/* Lock of the file system. */
static struct lock filesys_lock;
```

> 2. Describe how file descriptors are associated with open files.
>
> Are file descriptors unique within the entire OS or just within a single process?

1. We newly create a struct in `userprog/syscall.h`, which contains the file itself and its' descriptor. For every file opened by a user program, we add the file packed in this struct into thread's file list. So that every file has its file descriptor.
2. The file descriptors is unique within a single process. The field `fd` of `thread` struct indicates the next available file descriptor for this thread's file. Every time when we get a new file, we assign this value to the new file and add this `fd` by 1, so that the file descriptors are unique within this process.

> 3. Describe your code for reading and writing user data from the kernel.

When reading and writing user data from the kernel, `userprog/syscall.c:syscall_handler()` will be called, and the value of `f->esp` will be equal to `SYS_READ` or `SYS_WRITE`.

- Get all the arguments of write: `int fd`, `const void *buffer`, `unsigned size`.
- Check if the buffer pointer is valid. Then translate it into physical address.
- Call `int read (int fd, const void *buffer, unsigned size)` (or write function in the same format)
- In the function:
  - Get the lock of file system
  - If `fd == 0` for read, call `input_getc()` and reads from keyboard. If `fd == 1` for write, call `putbuf()` and writes to console.
  - Else, find if there is a file belongs to current thread and has the same `fd`. If so, call `file_read()` or `file_write()` and return the actual size it read or wrote. Else return `-1` (read) or `0` (write).

> 4. 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. **Least** - 1: when we can allocate a whole empty single page for the data;

   Greatest - 2: when we have to allocate the page among two sequencely pages.

2. For 2 bytes

   Least - 1

Greatest - 2

3. There is not much room for improvement in these numbers.

> 5. Briefly describe your implementation of the "wait" system call and how it interacts with process termination.

We use a semaphore named `waiting_process`.

1. If current thread's child list is empty, return -1 as the false child status.
2. Get the lock of current thread's child list, traverse through the list and find a child whose tid matches. If not found, release the lock and return -1.
3. Remove the found thread from the child list, then sema down the child's `waiting_process` which suggests that there is a thread waiting for it.
4. When child process is exiting, it set the exit status for its parent and sema up the `waiting_process` of itself and tell parent to finish waiting.
5. Then parent can return the exit code of child.

> 6. 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 implemented a function `void check_valid (const void *add)`. This function is used for checking if a pointer `add` is valid in virtual memory. We check if it is `NULL`, or if it is not a user virtual address, or we can't get page or it is not in the boundary. If so, call `exit(-1)` to terminate the process. In `exit()` function, all resourses of this process will be free.

SYNCHRONIZATION

> 7. 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"?

We create a semaphore `load_sema` and a int variable `load_success` inside `struct thread`.

The creation of new child process happens in `process_execute ()`. The `thread_create ()` function will return before the child process finish loading. So we might return a normal tid in the end of `process_execute ()` even if child process loading failed.

Therefore, we sema down the parent's `load_sema` before finishing `process_execute ()`. This will block our parent process. In `start_process ()` of the child process and after `load ()` finishes, we set the `load_success` of parent and sema up the parent's `load_sema`. After that, parent can continue finishing `process_execute ()` and return the correct tid.

> 8. 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?

1. As mentioned before in question B5, we implemented a semaphore to ensure that P will be blocked until C exits. When C exits, it will find the corresponding `child_thread` struct that stores in parent's child list, save its return value and all information that parent needs to know. So that P will get C's information after C exits.
2. Resources of C will be free after C finish processing. And the corresponding `child_thread` struct stored in P's list will be free after we make use of the informations.
3. If P terminates without waiting C, then C will become an orphaned thread. However, this will not happen in our implementation. When a process exits, we will look up into its child list and wait until all child terminate.

RATIONALE

> 9. Why did you choose to implement access to user memory from the kernel in the way that you did?

We choose to implement in the first way: *verify the validity of a user-provided pointer, then dereference it.* That is because it is easy to implement as we only need to call functions in `userprog/pagedir.c` and in `threads/vaddr.h`. Those function are already implemented, so it will be much safer.

> 10. What advantages or disadvantages can you see to your design for file descriptors?

Advantage: As mentioned before in question B2, our file descriptors are unique within a process. Actually, we newly set a file descriptor each time when we open a file, no matter if it has been opened before. This can ensure us open a same file in different or one process at the same time.

Disadvantage: It is slow to find the matching descriptor along all files. Also, the value of file descriptors cannot be reused after file closing.

> 11. The default `tid_t` to `pid_t` mapping is the identity mapping.
>
> If you changed it, what advantages are there to your approach?

We still use the default mapping since there will be only one thread in a process.

If we don't map `tid_t` to `pid_t` identically, we may store few `tid` in a process, which is one pid matches multiple tid. In this way, we can implement multiple thread per process.

Contribution

Modification to `process.c` was mostly done by pair programming.

`syscall.c` was implemented seperately then merge together.

We debug seperately on different testcase.

Yang Bin is responsible for most formating and comments. Feng Yufan is responsible for the design document.