Project 2: User Programs

Preliminaries

> Fill in your name and email address.

Yixin Yang (杨艺欣) yangyixin@stu.pku.edu.cn

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

All tests for lab 2 passed (run for 5 times).

My lab 1 submission passed all tests for lab 1 as well. I forgot to mention this in my lab 1 doc orz.

I built lab 2 on top of a clean code base with only alarm clock implemented.

> 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.

- DeepSeek: design-level advice
- 知乎：inline关键字相关知识
- strtok

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.

```
/* src/threads/thread.h */
struct thread
  {
    /* ... */
#ifdef USERPROG
    /* Owned by userprog/process.c. */
    uint32_t *pagedir;                  /**< Page directory. */
    struct proc_info *proc_info ;       /**< Process info. */
#endif
    /* ... */
  };

/* src/userprog/process.h */
struct proc_info
  {
    char **argv;                    /**< Command line arguments. */
    /* ... */
    int ref_count;                  /**< Reference count for process. */
  };


/* src/userprog/process.c */
/* Free the proc_info structure. This is a helper function for
   free_proc_info_refcnt and should not be called directly. */
inline static void free_proc_info (struct proc_info *proc_info);

/* Decrement the reference count of proc_info and free it if it reaches 0.
   This function is thread-safe. */
void free_proc_info_refcnt (struct proc_info *proc_info);

/* Initializer for proc_info. */
static void init_proc_info (struct proc_info *proc_info, char **argv);

static bool setup_stack (void **esp, char **argv); /* Added parameter char **argv. */
```

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?

Every thread has a pointer to a `proc_info` structure, where its command line arguments (and many other information) are stored. The `proc_info` structure is allocated by `malloc` and is managed by a reference count.

In `process_execute`, the `const char *` command line is parsed into `char **argv`. The details are as follows:

1. The command line argument is copied into a new page which is allocated by `palloc_get_page`. We also allocate a new page for the `argv` vector.
2. The copy is then parsed using `strtok_r` into a vector of strings.
3. Then we point each element of `argv` to the corresponding argument in the copy.
4. The `argv` vector is stored in the `proc_info` structure.

`proc_info` is eventually passed to `setup_stack`, where arguments in `argv` are pushed onto the stack. The order is the same as in the PintOS document:

1. `argv[0:argc][...]` are pushed in reverse order.
2. Some padding bytes to word-align the stack pointer.
3. The `argv` vector is pushed onto the stack all at once by `memcpy`.
4. `argc`, `argv` and the fake return address.

In `process_execute`, we limit the length and number of arguments to avoid overflowing the stack page.

- The whole command line string should not exceed `PGSIZE / 2` bytes.
- The `argv` vector takes up `(argc + 1) * sizeof (char *)` bytes. Padding bytes, `argc`, `argv` pointer and the fake return address each take up (at most) `sizeof (char *)` bytes. So we limit `argc` to `PGSIZE / (2 * sizeof (char *)) - 8`.

RATIONALE

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

`strtok` uses static storage to keep track of the current string position between calls, thus is not thread-safe.

`strtok_r` contains an additional parameter `saveptr` instead of using static storage, this avoids the problem.

> 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. It is easier to implement advanced features like command substitution and piping in the shell than in the kernel. Implementing these features in the kernel would add a lot of complexity to the kernel (e.g., auto completion needs to interact with filesystem), making it harder to maintain and vulnerable to bugs.
2. Different shells can implement different features and syntax, allowing users to choose the shell that best fits their needs. This also allows for more flexibility in the design of the shell, as it can be tailored to specific use cases or user preferences.
3. Keep parsing logic in the shell enables shell scripting, which allows automation.

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.

```c
/* src/filesys/filesys.h */
struct lock filesys_lock;    /**< Lock for file system. */

/* src/userprog/process.h */
#define MAX_FD 128 /**< Maximum number of file descriptors. */

typedef int pid_t; /**< Process ID type. */
#define PID_ERROR ((pid_t) -1)          /**< Error value for pid_t. */

struct proc_info
  {
    char **argv;                    /**< Command line arguments. */
    pid_t pid;                      /**< Thread ID. */
    int exit_status;                /**< Exit status. */
    struct file *executable;        /**< Executable file. */
    bool waited;                    /**< True if process has been waited on. */
    bool loaded;                    /**< True if process has been loaded. */
    struct thread *parent;          /**< Parent thread. */
    struct list child_list;         /**< List of child processes. */
    struct list_elem child_elem;    /**< List element for child list. */
    struct file *fd_table[MAX_FD]; /**< File descriptor table. */
    struct semaphore wait_sema;     /**< Semaphore for syscall wait */
    struct lock lock;               /**< Lock for proc_info. */
    int ref_count;                  /**< Reference count for process. */
  };

/* pid-tid mapping */
static inline pid_t tid_to_pid (tid_t tid);
static inline tid_t pid_to_tid (pid_t pid);

/* src/userprog/syscall.c */
/* Checks if addr is a valid user address. Helper function for
   is_valid_nbyte_ptr and is_valid_string. */
static bool is_valid_addr (const void *addr);

/* Checks if address [ptr, ptr + n - 1] is valid. */
static bool is_valid_nbyte_ptr (const void *ptr, size_t n);

/* Checks if a string is valid. */
static bool is_valid_string (const char *str);

/* Gets file * from fd. Also checks if fd and file * are valid. */
static struct file *get_file (int fd);

/* System call inplementations. */
static RET_TYPE syscall_NAME (PARAMS);

/* Argument count for each system call. */
static int syscall_argc[] = {
  [SYS_HALT] = 0,
  /* ... */
  [SYS_CLOSE] = 1
};
```

> 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 has its own `fd_table`, which maps its file descriptors to an open file `file *` (file descriptor 0 and 1 are reserved for stdin and stdout).

File descriptors are unique within a single process.

ALGORITHMS

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

User data accesses are checked by `is_valid_nbyte_ptr` and `is_valid_string`. These functions call a helper function `is_valid_addr`, which checks if an address is a user virtual address and is mapped to a page.

> 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 a contiguous address range `[bg, ed)`, if both `bg` and `ed - 1` are valid user addresses, the whole range is valid. Therefore, my `is_valid_nbyte_ptr` function only checks the first and last address of the range.

The least and greatest number of inspections are 2 for both cases.

There is room for improvement. If `bg` and `ed - 1` are in the same page, there is no need to call `pagedir_get_page` twice. The optimal least number of inspections for both cases is 1. For simplicity, I did not implement this optimization.

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

When a parent process P calls `wait (child_pid)`:

1. P checks its `proc_info->child_list` to search for the child process C with `pid = child_pid`.
2. If C does not exist or is already waited on, `wait` returns -1.
3. Set C's `proc_info->waited` to true. `sema_down` on C's `wait_sema` to wait for C to exit.
4. When C exits, it calls `sema_up` on its `wait_sema`, which wakes up P.
5. P removes C from its `proc_info->child_list` and calls `free_proc_info_refcnt`.
6. P returns C's exit status.

`free_proc_info_refcnt` will also be called when process exits. This ensures that C's `proc_info` is freed properly.

> 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 encapsulated the error handling logic in functions `is_valid_*`. I also implemented a helper function `get_file` to prevent repeated code in file-related system calls. This makes my system call implementations simple and clear.

When an error is detected, I typically release all acquired locks (and free other resources if needed) and call `syscall_exit`, where stuff like `proc_info` and file descriptor table will be handled properly.

An example (unrelated logic omitted):

```
static int
syscall_open (const char *file)
{
  /* Helper function to prevent repeated code. */
  if (!is_valid_string (file))
    syscall_exit (-1);    /* Call syscall_exit to handle error. */

  /* ... */

  lock_acquire (&proc_info->lock);
  lock_acquire (&filesys_lock);

  struct file *file_ptr = filesys_open (file);
  if (file_ptr == NULL)
  {
    /* Release locks and return -1. */
    lock_release (&filesys_lock);
    lock_release (&proc_info->lock);
    return -1;
  }

  /* Find an empty fd. */
  int fd = -1;
  for (int i = STDOUT_FILENO + 1; i < MAX_FD; i++)
      if (proc_info->fd_table[i] == NULL)
      {
        fd = i;
        break;
      }

  /* Finding fd failed. */
  if (fd == -1)
  {
    /* Close opened file and release locks. */
    file_close (file_ptr);
    lock_release (&filesys_lock);
    lock_release (&proc_info->lock);
    return -1;
  }

  /* ... */
}
```

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"?

In `process_execute` , after `thread_create` succeeds, the parent process waits for the child process to finish loading by calling `sema_down (&child_proc_info->wait_sema)` . The child process will call `sema_up (&proc_info->wait_sema)` when it finishes loading.

The success/failure status is stored in the child process's `proc_info->loaded` field, which will be checked by the parent process after `sema_down` returns. If `proc_info->loaded` is `false` , `process_execute` will return -1.

> 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?

Synchronization:

- As decribed in B5, parent process calls `sema_down` on its child's `wait_sema` when it waits. When `sema_down` returns, the child process must have exited.
- There is no synch problems if parent process terminates without waiting.
- `proc_info` struct has a `lock` to achieve mutual exclusion between parent and child processes. Every time it is accessed, it is protected by the lock. So there is no race condition in all cases.

Resource freeing:

- File descriptors are freed in `process_exit`. They are process specific, so they do not need a reference count.
- `proc_info` is shared between parent and child processes. It has a reference count to ensure that it is freed only when both processes are done with it. In particular, it is freed by `free_proc_info_refcnt` calls in the following cases:
    - When a process exits, it calls `free_proc_info_refcnt` to **both its own and all its children's proc_info**.
    - When a parent process waits for a child process, it **removes the child process from its `child_list`** and calls `free_proc_info_refcnt` on the child's `proc_info`.
    - `free_proc_info_refcnt` will also be called when an error occurs in `process_execute` or `start_process`.

Note that if a parent have waited a child process, **it will not free the child's `proc_info` again when it exits**, because it has removed the child from its `child_list` when it called `wait`.

Therefore, resources are freed properly in all cases.

RATIONALE

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

The PintOS documentation suggested two ways to implement user memory access. I chose the simpler one, following the software design principle of KISS (Keep It Simple, Stupid). I believe over-designing in early stages leads to unnecessary complexity and bugs.

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

Disadvantages:

1. The total number of file descriptors is limited to `MAX_FD`, which is 128. This may not be enough for some applications.
2. The file descriptor table is a `file *` array stored in `proc_info`, which takes up `MAX_FD * sizeof (file *)` bytes. This is a waste of space if the process does not use many file descriptors.
3. I did not utilize `file_reopen ()` to speed up opening opened files, because my current implementation is fast enough to pass all the tests.

Advantages:

1. Implement file descriptor table as a `file *` array makes it easy and natural to get `file *` from a 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 did not change the default mapping because identity mapping is already the best choice for now.

However, I provided inline functions `pid_to_tid` and `tid_to_pid` to convert between `pid_t` and `tid_t` . If in the future I need a better mapping, I can switch to that mapping by only changing these functions, without modifying any rest of the code.