Project User Programs Report

Group 48

Name	Autograder Login	Email
Ben Plate	student219	bplate@berkeley.edu
Jeffrey Liang	student50	jeffreyliang@berkeley.edu
Teddy Cho	student139	teddycho@berkeley.edu
Theophilus Pedapolu	student270	theopedapolu@berkeley.edu

Changes

Argument Passing

Our implementation of argument passing is largely the same as described in our design doc, though we push things onto the user stack in a slightly different order. Specifically, we first push the strings referenced in `argv`, then padding for stack alignment, then the address to the strings with a NULL final element (the `argv` array itself), then a pointer to the first element of `argv`, and then finally `argc`.

Process Control Syscalls

The main change from our design doc is that we don't maintain a list of child PCB's in `struct process`. The original intention of this list was to track which child processes were still running. Instead of doing this, we added the field `bool exited` to the shared data `struct exit_status` that is set to true when the child process exits. This change primarily affected the implementation of the `wait` syscall in the `process_wait` function; specifically, instead of traversing the list of child PCB's, we simply check the `exited` field of that child's `exit_status` structure.

We also added a `bool waited` field to `struct exit_status` that is set to true if that child has already been waited on. This is necessary to ensure that the parent process can wait on a child process at most once. If `waited` is set to true when a parent attempts to wait on the corresponding child, the `wait` syscall returns -1.

In addition, we have to keep the executable file backing each process open with write denied to ensure that the executable cannot be modified while the program is running. To do this, we call `file_deny_write` on the file in `load`, store a pointer to the file in the PCB's `exec_file` field, and close that file in `process_exit`.

After these changes, the `exit_status` and `process` looked like this.

```
struct exit_status {
  pid_t pid;                  /* Process id */
  int status;                 /* Exit status */
  bool exited;                /* True if exited, false otherwi
se */
  bool waited;                /* True if waited, false otherwi
se */
  int ref_cnt;                /* Initialize to 2 */
  struct lock ref_cnt_lock;   /* Lock for ref_cnt */
  struct semaphore exit_wait; /* Down'd by parent's process_wa
it, up'd by process_exit */
  struct list_elem elem;      /* List element for PCB's child_
exit_statuses */
};
struct process {
  /* Existing fields omitted */
  struct file* exec_file;          /* The file executed by thi
s process */
  struct exit_status* exit_status; /* Pointer to this proces
s's exit status */
  struct list child_exit_statuses; /* List of children's exit
statuses */
  /* Fields added in file operations syscalls omitted */
};
```

Finally, in our design doc, we incorrectly said that the `exit_wait` semaphore used by the `wait` syscall should be initialized to -1. This semaphore should actually be

initialized to 0, and we did so in our implementation.

File Operation Syscalls

Although there were no design-level changes made since the implementation of file operation syscalls, there was some reorganization of code relating to `user_file` and `user_file_list` in adding helper functions that make handling these structures easier.

The new helper functions are defined in a new file `userfile.c`, which can be found in `userprog`, and have the following signatures:

```
struct user_file {
  int fd;
  struct file* file;
  struct list_elem elem;
};

typedef struct list user_file_list;

int user_file_open(user_file_list* list, const char* file, int fd);
void user_file_close(user_file_list* list, int fd);
struct user_file* user_file_get(user_file_list* list, int fd);
void user_file_list_destroy(user_file_list* list);
```

The `user_file` struct is the same, but we've created `user_file_list` type that distinguishes the `user_files` list found in `process` from other lists. Each function takes in a pointer to a `user_file_list`, which corresponds to the list that will be modified, since this is on a per-process basis.

`user_file_open` runs the `filesys_open` function on `file` and creates a new `user_file` struct with that file and the `fd` supplied. Returns `fd` if the operation was successful and `-1` otherwise.
`user_file_close` removes the file corresponding to `fd` in the list and frees the respective resources.

`user_file_get` returns the `user_file` struct corresponding to the entry found in the list with file descriptor `fd`, or `NULL` if it doesn't exist.
`user_file_list_destroy` is called when the process is exiting and the pcb is being freed; it closes all currently-opened files, frees all the respective `user_file` structs, and destroys the list. There is no corresponding function for initializing the list as this functionality is achieved with `list_init` in `list.c`.

These functions do not do anything different from what was described in the design doc, but rather organize all the procedures that may be repeated into functions as well as abstract away the details of the `user_file_list`, which makes safe implementation easier.

Floating Point Operations

First, there were some basic changes we made in our data structures and functions. We changed the size of our fpu_state variable in the `intr_frame` and `switch_threads_frame` structs to be `char fpu_state[108]` instead of `char fpu_state[128]` since the entire FPU state can be stored in 108 bytes. Moreover, we used `fninit` instead of `finit` to initialize the FPU in `start.s` because we don't want the FPU to check for exceptions using garbage data that was already there.

Another change we made was reordering our assembly code from the design doc in `intr_stubs.S` and `switch.S`, i.e. the interrupt handling and thread switching code. In particular, we chose to decrement the stack pointer before saving the FPU state on `intr_frame` and `swtich_threads_frame` and increment it after restoring the FPU state. The new code is as below:

Saving

```
addl $-108, %esp
fsave 0(%esp)
```

Restoring

```
frstor 0(%esp)
addl $108, %esp
```

Since `fsave` and `frstor` interact with the 108 bytes of memory above `%esp` and don't change `%esp`, we need to make space on the stack before saving (hence decrementing `%esp`) and, after restoring, we need to move `%esp` to the next parameter in the `intr_frame` and `switch_threads_frame` structs (hence

incrementing). The location of these code segments in `intr_stubs.S` and `switch.S` remain the same.

Additionally, we removed the `fwait` instruction in `intr_stubs.S` and `switch.S` as well as in the asm volatile code used to initialize the FPU during thread and process creation. It was not necessary because `frstor` does not raise any floating-point exceptions. The new assembly code (written via `asm volatile`) in the `thread_create` function of `thread.c` is as follows:

```
fsave 0(temp)
fsave 0(sf->fpu_state)
frstor 0(temp)
```

In our design doc, we forgot to initialize the FPU during process creation, so we also added this exact code in the `start_process` function of `process.c`, but replacing `sf→fpu-state` with `if_.fpu_state` to store the initialized FPU on the `intr_frame` of the new process instead.

Reflection

Each of our group members contributed to the following parts of this project.

- We collectively wrote the Argument Passing section of the design doc during one of our group meetings.
- Jeffrey wrote the Process Control Syscalls section of the design doc and implemented the changes outlined in that section. Specifically, this involved adding the discussed data structures to `userprog/process.h` and `userprog/process.c` and modifying the functions `process_execute`, `start_process`, `process_wait`, `process_activate`, and `userprog_init` in `userprog/process.c`.
- Ben wrote the File Operations Syscalls section of the design doc and implemented the changes outlined in that section. This also involved contributing to the process structure in `userprog/process.c` for the relevant file attributes, implementing the relevant file operation syscall handlers in `userprog/syscall.c`, as well as creating the data structures and functions found in the new files `userprog/userfile.c` and `userprog/userfile.h`. He also

implemented the code for Argument Passing and wrote up the Concept Check section of the design doc.
- Theo wrote the Floating Point Operations section of the design doc and implemented the code outlined there. In particular, he modified the structs in `threads/interrupt.h` and `threads/switch.h`, wrote assembly code in `threads/intr_stubs.S` and `threads/switch.S`, modified the `thread_create` function in `threads/thread.c`, the `start_process` function in `userprog/process.c`, and assembly code in `threads/start.S` and implemented the `compute_e` syscall in `userprog/syscall.c`
- Teddy helped brainstorm for the overall design of the project and wrote the tests outlined in the testing portion of this document. In particular, he wrote the files `test/userprog/my-test-1.c` and `test/userprog/my-test-2.c`, wrote the Perl scripts in `test/userprog/my-test-1.ck` and `test/userprog/my-test-2.ck` to execute the tests, and made the necessary changes in `tests/userprog/Make.tests` to run make check. He wrote the testing portion of this document, including troubleshooting for potential bugs in our kernel implementation.

For the most part, our working environment was smooth and relatively unproblematic. We held a lot of regular meetings at the beginning to help each other get accustomed to the PintOS codebase and Project 1 specifications. During the design process, we collectively wrote the argument passing section and talked in big picture ideas about the remaining sections. Then, we each worked independently on different sections of the design doc and held a meeting afterward to explain our designs to everyone, allowing everyone to get on the same page and point out possible design errors before we submitted the doc. During the coding/implementation portion, we each worked on our design sections on different branches, making occasional commits and merges with the main branch. In hindsight, we could have done this differently, perhaps pair programming or even writing code as a group in certain sections, because working mostly independently led to a lack of communication. Changes in the codebase conflicted with each other, and sometimes it took longer than expected to understand and integrate these changes together. Moreover, since parts of the project like Floating Point Operations and certain syscalls were dependent on other parts, we could not test some sections until these parts were finished, which reduced our productivity considerably. We could have improved our workflow by programming together more often during the implementation phase.

Testing

my-test-1.c

Description

Invokes a system call with an invalid system call number. In the event where the user makes a system call with a number that does not have a corresponding handler, the process should be terminated with -1 exit code.

Overview

In line 8, the number 32 (0x20) is pushed to the stack; since system call numbers are only defined from 0 (SYS_HALT) to 31 (SYS_INUMBER), the kernel is not able to handle this system call and should therefore kill the process. Line 8 passes the `int $0x30` instruction, which triggers an interrupt. The system call handler should recognize that 32 is not a valid system call number and exit the program with a status of -1; if that does not happen, the program resumes and line 9 is run, failing the test since the process should have exited earlier.

Output

```
Copying tests/userprog/my-test-1 to scratch partition...
qemu-system-i386 -device isa-debug-exit -hda /tmp/MY6WQYpNcO.d
sk -m 4 -net none -nographic -monitor null
PiLo hda1
Loading..............
Kernel command line: -q -f extract run my-test-1
Pintos booting with 3,968 kB RAM...
367 pages available in kernel pool.
367 pages available in user pool.
Calibrating timer...  135,168,000 loops/s.
hda: 5,040 sectors (2 MB), model "QM00001", serial "QEMU HARDD
ISK"
hda1: 226 sectors (113 kB), Pintos OS kernel (20)
hda2: 4,096 sectors (2 MB), Pintos file system (21)
```

```
hda3: 124 sectors (62 kB), Pintos scratch (22)
filesys: using hda2
scratch: using hda3
Formatting file system...done.
Boot complete.
Extracting ustar archive from scratch device into file syste
m...
Putting 'my-test-1' into the file system...
Erasing ustar archive...
Executing 'my-test-1':
(my-test-1) begin
my-test-1: exit(-1)
Execution of 'my-test-1' complete.
Timer: 63 ticks
Thread: 3 idle ticks, 27 kernel ticks, 33 user ticks
hda2 (filesys): 67 reads, 252 writes
hda3 (scratch): 123 reads, 2 writes
Console: 872 characters output
Keyboard: 0 keys pressed
Exception: 0 page faults
Powering off...
```

Result

```
PASS
```

Potential Bugs

1. When the system call handler `syscall_handler` gets control, the system call number is in the 32-bit word at the caller's stack pointer, accessible via the esp member of the `struct intr_frame` passed to it. If the kernel didn't properly save the system call number at the caller's stack pointer, then `f->esp[0]` would access incorrect memory, and indexing into `syscall_table` at that location would most likely produce a segfault.

2. In `syscall_handler`, we call `are_valid_args(args, 1)` before accessing the syscall number in `args[0]` to check that the syscall number is in user memory. If the kernel didn't do this, then we cannot confirm that the syscall argument exists in valid user memory, and indexing into `syscall_table` at that location would most likely produce a segfault.

my-test-2.c

Description

Attempts to write past end-of-file. The program should write as many bytes as possible up to end-of-file and return the actual number written.

Overview

In line 16, the write system call is invoked with the `size` argument set to `sizeof sample + 100`, which is 101 bytes past end-of-file. However, the program should successfully write up to end-of-file and return the actual number of bytes written, which should be `sizeof sample - 1`. Line 17 checks that the return value `byte_cnt` is correctly set to `sizeof sample - 1` and otherwise fails the test.

Output

```
Copying tests/userprog/my-test-2 to scratch partition...
qemu-system-i386 -device isa-debug-exit -hda /tmp/tUQmEGOWWH.dsk -m 4 -net none -nographic -monitor null
PiLo hda1
Loading..............
Kernel command line: -q -f extract run my-test-2
Pintos booting with 3,968 kB RAM...
367 pages available in kernel pool.
367 pages available in user pool.
Calibrating timer...  183,500,800 loops/s.
hda: 5,040 sectors (2 MB), model "QM00001", serial "QEMU HARDDISK"
hda1: 226 sectors (113 kB), Pintos OS kernel (20)
hda2: 4,096 sectors (2 MB), Pintos file system (21)
hda3: 125 sectors (62 kB), Pintos scratch (22)
```

```
filesys: using hda2
scratch: using hda3
Formatting file system...done.
Boot complete.
Extracting ustar archive from scratch device into file syste
m...
Putting 'my-test-2' into the file system...
Erasing ustar archive...
Executing 'my-test-2':
(my-test-2) begin
(my-test-2) create "test.txt"
(my-test-2) open "test.txt"
(my-test-2) end
my-test-2: exit(0)
Execution of 'my-test-2' complete.
Timer: 65 ticks
Thread: 6 idle ticks, 29 kernel ticks, 30 user ticks
hda2 (filesys): 94 reads, 260 writes
hda3 (scratch): 124 reads, 2 writes
Console: 945 characters output
Keyboard: 0 keys pressed
Exception: 0 page faults
Powering off...
```

Result

```
PASS
```

Potential Bugs

1. When the system call handler `syscall_handler` gets control, the system call number is in the 32-bit word at the caller's stack pointer, the first argument is in the 32-bit word at the next higher address, and so on. If the kernel didn't properly save the system call arguments at the correct locations above the caller's stack

pointer, then `syscall_handler` would most likely exit with -1 after checking that the arguments weren't in user memory.
2. In `syscall_handler`, we call `are_valid_args(args + 1, num_args)` to check that the arguments are in user memory. If the kernel didn't do this, then the program would copy the wrong arguments to kernel stack, and `syscall_write_handler` would proceed with the wrong arguments, which would most likely produce a segfault.

Testing Experience

Our experience writing tests for Pintos was pretty straightforward, and we found the detailed instructions for the Pintos testing framework under "Adding new tests to Pintos" very useful, although they were a bit buried under the Pintos Documentation. We also found writing the Perl scripts in the `.ck` files for the tests pretty unintuitive, especially since we never learned Perl in class, and we think that removing the need for writing those scripts would be a big improvement to the Pintos testing system. Finally, we learned that next time, we should write more tests in general as we work through the project; we wrote all our tests at the very end this time, and in hindsight, coming up with them and writing them during development might have helped us identify and fix bugs a lot earlier.