國立臺灣大學 National Taiwan University

EE 5173 Operating System
lab 01: system call

Chen-Yin, Lee (李臻茵)

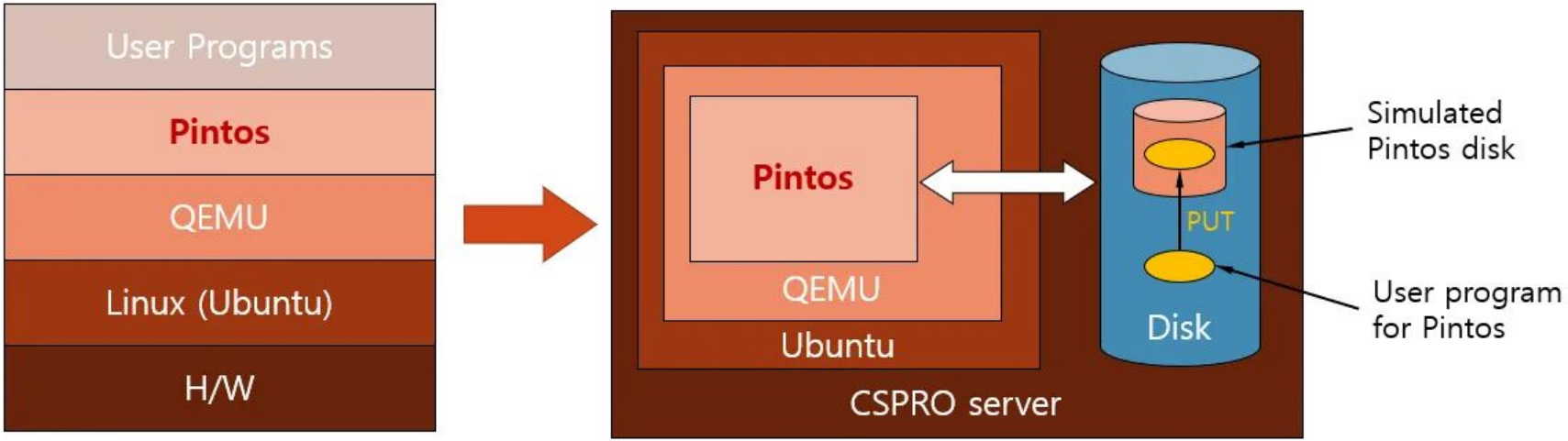
r13921090@ntu.edu.tw

2025/03/12

Goal

- Understand how system calls are interacted with OS
- Understand how the user programs work in Linux environment
- Understand the difference between user mode & kernel mode

Background


User Programs
Pintos
QEMU
Linux (Ubuntu)
H/W
Pintos
QEMU
Ubuntu
CSPRO server
PUT
Disk
Simulated
Pintos disk
User program
for Pintos

Part I: Argument Passing

- Trace how argument passing work in OS

  When we start execution, kernel need to load ELF from disk into memory by using load() function, then initialize the program's stack. Finally, we jump to the program start address, allowing it to execute on its own.

- Implement push_argument() in userprog/process.c

  The kernel must put the arguments for the initial function on the stack before it allows the user program to begin executing. The arguments are passed in the same way as the normal calling convention (80x86 Calling Convention).

Requirement in tech report

- **push_argument() implementation details**
  - Explain how you implement this function.
  - A diagram or structured description of stack layout after argument passing.
- **Explain how this work in the beginning of execution, including:**
  - Overview of function flow with key parameter setting.
  - Code snippets (if needed) to support your explanation.

Part II: System Calls

- Print a process termination message whenever a user process terminates. The message should include the full cmdline name as passed to process_execute() and the exit status. The message should be formatted as follows:

```
printf("%s: exit(%d)\n", process_name, exit_status);
```

- Implement 12 system calls in [userprog/syscall.c]:
  - Process Management: exit(), exec(), wait()
  - File I/O Management :

    create(), remove(), open(), filesize(), read(), **write()**, seek(), tell(), close()

System Calls Details

- **Process Control System Calls**

void exit(int status);

Terminates the cur process and returns status to the kernel.

pid_t exec(const char *cmdline);

Runs a program with cmdline and returns its pid.

Return -1 on failure.

int wait(pid_t pid);

Waits for child pid to terminate and returns its exit status.

Returns -1 if pid is invalid or not a direct child.

System Calls Details

- **File System System Calls**

bool create(const char *file, unsigned initial_size);

Creates a new file with the given name and size.

bool remove(const char *file);

Deletes the specified file.

int open(const char *file);

Opens file and returns a file descriptor (fd).

Returns -1 if the file cannot be opened.

int filesize(int fd);

Returns the size of the file associated with fd.

System Calls Details

- **File System System Calls**

int read(int fd, void *buffer, unsigned size);

Reads size bytes from fd into buffer.

int write(int fd, const void *buffer, unsigned size);

Writes size bytes from buffer to fd.

void seek(int fd, unsigned position);

Moves the file read/write position to the given position.

unsigned tell(int fd);

Returns the current file read/write position.

void close(int fd);

Closes the file descriptor fd.

Requirement in tech report

- **Modifications in threads/thread.h**

  Describe any new fields added to struct thread and their purpose.

- **System Call Handling in syscall.c**

  Explain how you implement system calls, including argument retrieval and error handling.

- **Test Results**
  - Provide a snapshot of test results and your **make check** score.
  - If any tests fail, briefly explain the possible reasons.

```
PASS tests/filesys/base/sm-full
PASS tests/filesys/base/sm-random
PASS tests/filesys/base/sm-seq-block
PASS tests/filesys/base/sm-seq-random
FAIL tests/filesys/base/syn-read
PASS tests/filesys/base/syn-remove
FAIL tests/filesys/base/syn-write
10 of 10 tests failed.
make: *** [../../tests/Make.tests:27: check] Error 1
```

Hints

- This part is mainly modified code under userprog/. However, you may want to check/modify threads/threads.ch, filesys/file.ch for more information.
- You may want to implement write() first for debugging purpose; and always verify the validity of memory addresses before accessing them, especially when handling user arguments.
- How to test/debug a single file during your implementation?

  > pintos --qemu --gdb --filesys-size=2 -p tests/userprog/[test file name] -a [test file name] -- -q -f run '[test file] [argv]'

```
pintos -v -k -T 60 --qemu  --filesys-size=2 -p tests/userprog/sc-bad-arg -a sc-bad-arg -- -q  -f run sc-bad-arg < /dev/null 2> tests/userprog/sc-bad-arg.errors > tes
ts/userprog/sc-bad-arg.output
perl -I../.. ../../tests/userprog/sc-bad-arg.ck tests/userprog/sc-bad-arg tests/userprog/sc-bad-arg.result
PASS tests/userprog/sc-bad-arg
pintos -v -k -T 60 --qemu  --filesys-size=2 -p tests/userprog/sc-boundary -a sc-boundary -- -q  -f run sc-boundary < /dev/null 2> tests/userprog/sc-boundary.errors >
 tests/userprog/sc-boundary.output
perl -I../.. ../../tests/userprog/sc-boundary.ck tests/userprog/sc-boundary tests/userprog/sc-boundary.result
PASS tests/userprog/sc-boundary
pintos -v -k -T 60 --qemu  --filesys-size=2 -p tests/userprog/sc-boundary-2 -a sc-boundary-2 -- -q  -f run sc-boundary-2 < /dev/null 2> tests/userprog/sc-boundary-2.
errors > tests/userprog/sc-boundary-2.output
```

Hints for testing you code!

- For Part 1, remember to check how this process works. You may need to modify other parts of the code, not just implement the code in push_argument().
- After finishing argument passing & related work and system call exit() and write(), you may be able to pass around 20 Pintos tests, especially all tests under args-*.

All 80 test cases

argument passing	basic system call	file system		Advanced
tests/userprog/args-none	tests/userprog/halt	tests/userprog/create-normal	tests/userprog/close-normal	tests/userprog/no-vm/multi-oom
tests/userprog/args-single	tests/userprog/exit	tests/userprog/create-empty	tests/userprog/close-twice	tests/filesys/base/lg-create
tests/userprog/args-multiple	tests/userprog/exec-once	tests/userprog/create-null	tests/userprog/close-stdin	tests/filesys/base/lg-full
tests/userprog/args-many	tests/userprog/exec-arg	tests/userprog/create-bad-ptr	tests/userprog/close-stdout	tests/filesys/base/lg-random
tests/userprog/args-dbl-space	tests/userprog/exec-bound	tests/userprog/create-long	tests/userprog/close-bad-fd	tests/filesys/base/lg-seq-block
	tests/userprog/exec-bound-2	tests/userprog/create-exists	tests/userprog/read-normal	tests/filesys/base/lg-seq-random
system call boundary	tests/userprog/exec-bound-3	tests/userprog/create-bound	tests/userprog/read-bad-ptr	tests/filesys/base/sm-create
tests/userprog/sc-bad-sp	tests/userprog/exec-multiple	tests/userprog/open-normal	tests/userprog/read-boundary	tests/filesys/base/sm-full
tests/userprog/sc-bad-arg	tests/userprog/exec-missing	tests/userprog/open-missing	tests/userprog/read-zero	tests/filesys/base/sm-random
tests/userprog/sc-boundary	tests/userprog/exec-bad-ptr	tests/userprog/open-boundary	tests/userprog/read-stdout	tests/filesys/base/sm-seq-block
tests/userprog/sc-boundary-2	tests/userprog/wait-simple	tests/userprog/open-empty	tests/userprog/read-bad-fd	tests/filesys/base/sm-seq-random
tests/userprog/sc-boundary-3	tests/userprog/wait-twice	tests/userprog/open-null	tests/userprog/write-normal	tests/filesys/base/syn-read
	tests/userprog/wait-killed	tests/userprog/open-bad-ptr	tests/userprog/write-bad-ptr	tests/filesys/base/syn-remove
memory access	tests/userprog/wait-bad-pid	tests/userprog/open-twice	tests/userprog/write-boundary	tests/filesys/base/syn-write
tests/userprog/bad-read			tests/userprog/write-zero	tests/userprog/multi-recurse
tests/userprog/bad-write			tests/userprog/write-stdin	tests/userprog/multi-child-fd
tests/userprog/bad-read2			tests/userprog/write-bad-fd	tests/userprog/rox-simple
tests/userprog/bad-write2				tests/userprog/rox-child
tests/userprog/bad-jump				tests/userprog/rox-multichild
tests/userprog/bad-jump2				

- For more details, please check source file under tests/

Grading Policy

- Part I (Trace argument passing) - 30%
- Part II (Implement system calls) - 70%
  - Test Score – 20%
  - Tech Report – 50%

Deadline 2025/4/3 23:59

- Filename (-3%)
- Late submission (-10%), over 1 weeks (-20%) over 2 weeks (Immediate 0 points)
- Plagiarism or All/Most LLM (Immediate 0 points)
- Bonus (Other implementations: Max 10)

Report Format

- Content format: 12pt front,16pt row height, and align to the left.
- Caption format: 18pt and **Bold font**.
- Figure: center with single line row height.
- Upload with the file structure format :

G[team number]_1.zip (e.g. G01_1.pdf)
├── G[team number]_1.pdf
├── pintos
│ ├ devices
│ ├ …
│

Be sure to unzip and double check your .zip before uploading!!!

Reminder

- The test score is considered passed only if the TA job passes.
- 0 will given to cheaters. Do not copy & paste!
  - TA will check your repository
- Feel free to ask TA questions, but TA won't help you debug your code.
- Have fun ☺

Related stuff for this lab

- GitHub repository:

  git clone git@github.com:Xueyi-Chen-David/pintos.git

- Explanation video from TA [link]

File system in Linux

Integer	file stream
0	standard input (STDIN)
1	standard output (STDOUT)
2	standard error (STDERR)

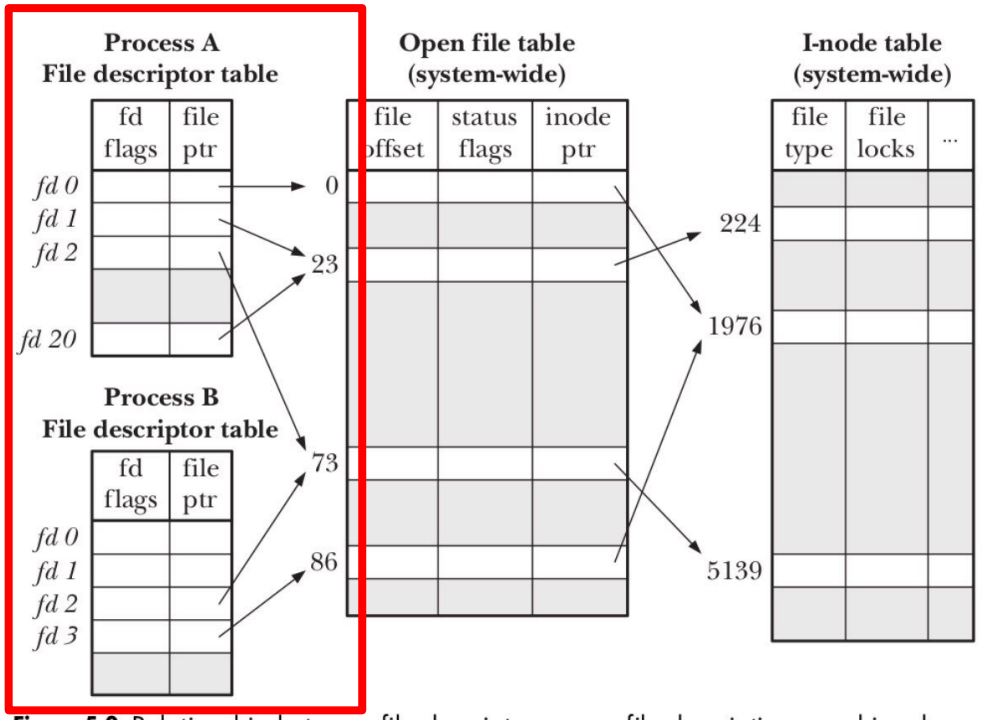


Figure 5-2: Relationship between file descriptors, open file descriptions, and i-nodes

A Simple Guide for Debugging Pintos

EE 5173 Operating System

TA: Chen-Yin, Lee. Advisor: Farn, Wang.

2025/03/12

Useful debugging tools for Pintos

1. printf()

   call it as the normal C language from anywhere in your projects.

2. ASSERT()

   for testing the value of expression

3. GDB

GDB for Pintos Debugging

step1. build pintos in your project (e.g. ~/pintos/userprog)

```
$ make
$ pintos --gdb -- run <test program>
```

step2. open another terminal on same machine for debugging

```
$ docker exec -it pintos bash
$ cd ~/pintos/threads/build
$ pintos-gdb kernel.o
(gdb) debugpintos
...
```

Reminder

If the Pintos GDB macros do not activate successfully, please check your GDB macro path settings in utils/pintos-gdb
However, for the convenience of your debugging, try not to modify the chosen debug tool.

Some useful GDB Commands

- Quit the debugging session

  (gef) monitor quit

- Add breakpoint

  (gef) break <filename>:<linenumber> -- set a breakpoint

- Program execution control

  (gef) c (continue) -- Continues normal execution

  (gef) s (step) -- execute one line of code

  (gef) n (next) -- execute the entire function

  (gef) info local -- print all local variables

  (gef) bt (backtrace) -- show the calling order of each func & the input form params of each

Tips:
You can use break main to make GDB stop when Pintos starts running.

Some useful GDB Commands

- Memory Information

  (gef) p *[structure] -- Print structure information

```
(remote) gef➤  p *thread_current()
$10 = {
  tid = 0x1,
  status = THREAD_RUNNING,
  name = "main", '\000' <repeats 11 times>,
  stack = 0xc000f000 "",
  priority = 0x1f,
  ori_priority = 0x1f,
  allelem = {
    prev = 0xc003823c <all_list>,
    next = 0xc0038244 <all_list+8>
  },
  elem = {
    prev = 0xc003824c <ready_list>,
    next = 0xc0038254 <ready_list+8>
  },
  lock_list = {
    head = {
      prev = 0x0,
      next = 0xc000e03c
    },
    tail = {
      prev = 0xc000e034,
      next = 0x0
    }
  },
  magic = 0xcd6abf4b
}
```

Reference

- Pintos Projects: Debugging Tools
- GDB manual