1

Pintos Project 2
User Programs

COS 450 - Fall 2018

2

Project 1 Submissions
Easy Things to Fix

Project submission

Code style

ASSERT and malloc()

Design document questions

3

Code Style
1.2.2.2

Match the Pintos code style

Indent 2 spaces -- braces by themselves

"`/* */`" not "`//`"

Remove don't comment out unused code

`function (arg1, arg2)`
not `function(arg1,arg2)`

Note the Space

4

ASSERT and malloc

Don't `ASSERT()` on things that fail

Check the return from `malloc()`

5

DESIGNDOC

It's an ASCII text file

Hard wrapped at 79 characters

Use proper spelling and grammar

Answer the questions

Consider alternative designs

`pintos/src/____/DESIGNDOC`

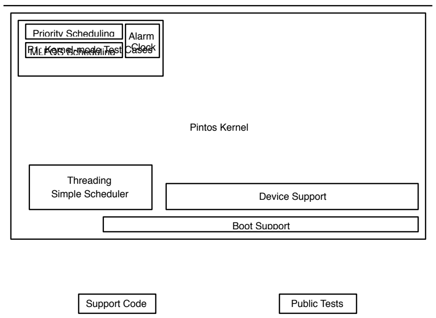
6

Project 1 Review

All code was ***kernel*** code...

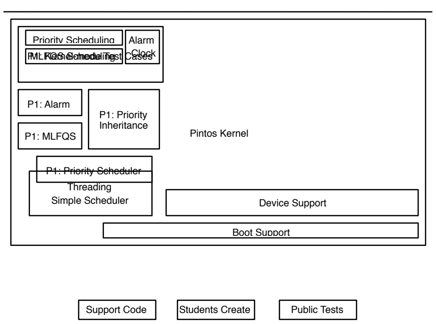
- Alarm Clock
- Priority Scheduling
- Advanced Scheduler

7



Before Project 1

8



After Project 1

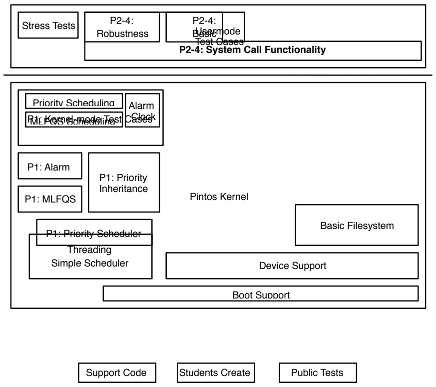
9

Project 2

Enable ***user*** processes in Pintos...

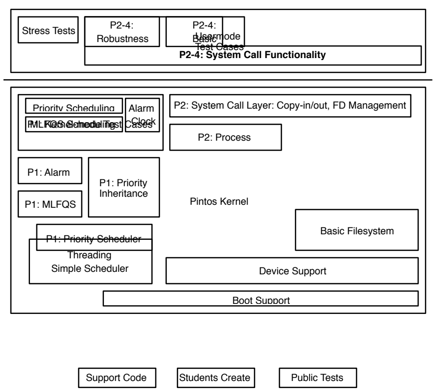
- Set up stack and pass arguments
- Implement process waiting
- Implement system calls

10



Before Project 2

11



After Project 2

12

Getting Started

No code from Project 1 is needed

- You can start from a fresh copy
- Knowledge of threads is important

Read “Testing” section again

- 70+ tests, know how to run them

Where You Work

13

Most Work in "userprog"

- make, make check, make grade here
- All files already exist.

Some external files are useful

- from threads and lib

Files to Modify

14

process.c

- Load and execute of processes
- Stack setup code
- Process waiting code

syscall.c

exception.c

Files to Modify

15

process.c

syscall.c

- All system calls go here
- Stub code exits immediately

exception.c

16

Files to Modify

process.c

syscall.c

exception.c

- Contains exception handling code
- May/may not modify, depends on solution

17

Important Files

pagedir.c

- pagedir_get_page () - validate refs

threads/vaddr.h

- is_user_vaddr() - validate pointers

lib/string.c

- strtok_r()

18

Running a Program

The shell parses a command “cp pintos .”

Shell calls fork() and execve(“cp”, argv, env)

cp uses file system API to copy files

cp (might) print to the console

cp exits and returns an exit code to shell

19

Pintos Chain of Execution

threads/init.c

main() → run_actions(argv)

run_actions(argv) → run_task(argv)

run_task(argv) →
process_wait(process_execute(task))

20

Pintos Chain of Execution

userprog/process.c

process_execute() creates thread that runs start_process(filename, ...)

start_process() → load(filename)

load() does all the remaining work

set up stack, data, code, etc.

21

Project Requirements

Passing arguments to process

Safe memory access

Process waiting

System calls (implement them)

Process termination messages

Deny write to open executable files

This is just a list of tasks not the order of implementation

Passing Arguments

22

Parse command line string:

```
cp -r pintos .
```

into individual tokens onto user stack.

Spec says in process_execute()

You can do this in many ways...

strtok_r() a really good choice

Setting up the Stack

23

Push the arguments, word align

Push a NULL sentinel (0)

Push pointers to arguments (in reverse)

Push a pointer to the first pointer

Push the argument count

Push a fake return address (0)

24

Address	Name	Data	Type
0xbffffffc	argv[3][...]	"bar\0"	char[4]
0xbffffff8	argv[2][...]	"foo\0"	char[4]
0xbffffff5	argv[1][...]	"-l\0"	char[3]
0xbfffffed	argv[0][...]	"/bin/ls\0"	char[8]
0xbfffffec	word-align	0	uint8_t
0xbfffffe8	argv[4]	0	char *
0xbfffffe4	argv[3]	0xbffffffc	char *
0xbfffffe0	argv[2]	0xbffffff8	char *
0xbfffffdc	argv[1]	0xbffffff5	char *
0xbfffffd8	argv[0]	0xbfffffed	char *
0xbfffffd4	argv	0xbfffffd8	char **
0xbfffffd0	argc	4	int
0xbfffffcc	return address	0	void (*) ()

hex_dump()

```
bfffffc0                                      00 00 00 00 |            ....|
bfffffd0  04 00 00 00 d8 ff ff bf-ed ff ff bf f5 ff ff bf |................|
bfffffe0  f8 ff ff bf fc ff ff bf-00 00 00 00 00 2f 62 69 |............./bi|
bffffff0  6e 2f 6c 73 00 2d 6c 00-66 6f 6f 00 62 61 72 00 |n/ls.-l.foo.bar.|
```

this all should happen in start_process() after the interrupt frame (and the stack pointer) is initialized

Accessing User Memory

25

When user processes make system calls the kernel needs to deal with the pointers...

- NULL pointers
- Pointers to unmapped memory
- Pointers to kernel memory (invalid)

Once identified, just kill the process...

Accessing User Memory

26

Verify before dereference

- is in user space -- is_user_vaddr()
- is mapped -- pagedir_get_page()
- at start and end of buffers and strings

Modify fault handler

Accessing User Memory

27

Verify before dereference

Modify fault handler

- only check if in user space
- invalid access triggers page fault
- modify page fault handler
- much better performance (§ 3.1.5)

Typical Implementation

28

Check address and use page fault handler

Don’t pass user addresses into kernel

copy_from_user(void *dst, void *src)

copy_to_user(void *dst, void *src)

Copy to/from User

29

```
/* Reads a byte at user virtual address UADDR.
   UADDR must be below PHYS_BASE.
   Returns the byte value if successful, -1 if a segfault
   occurred. */
static int
get_user (const uint8_t *uaddr)
{
  int result;
  asm ("movl $1f, %0; movzbl %1, %0; 1:"
       : "=&a" (result) : "m" (*uaddr));
  return result;
}

/* Writes BYTE to user address UDST.
   UDST must be below PHYS_BASE.
   Returns true if successful, false if a segfault occurred. */
static bool
put_user (uint8_t *udst, uint8_t byte)
{
  int error_code;
  asm ("movl $1f, %0; movb %b2, %1; 1:"
       : "=&a" (error_code), "=m" (*udst) : "q" (byte));
  return error_code != -1;
}
```

Page Fault Handler

30

```
/* Page fault handler.  This is a skeleton that must be filled in
   to implement virtual memory.  Some solutions to project 2 may
   also require modifying this code.

   At entry, the address that faulted is in CR2 (Control Register
   2) and information about the fault, formatted as described in
   the PF_* macros in exception.h, is in F's error_code member.  The
   example code here shows how to parse that information.  You
   can find more information about both of these in the
   description of "Interrupt 14--Page Fault Exception (#PF)" in
   [IA32-v3a] section 5.15 "Exception and Interrupt Reference". */
static void
page_fault (struct intr_frame *f)
{
  bool not_present;  /* True: not-present page, false: writing r/o page. */
  bool write;        /* True: access was write, false: access was read. */
  bool user;         /* True: access by user, false: access by kernel. */
  void *fault_addr;  /* Fault address. */

  /* Obtain faulting address, the virtual address that was
     accessed to cause the fault.  It may point to code or to
     data.  It is not necessarily the address of the instruction
     that caused the fault (that's f->eip).
     See [IA32-v2a] "MOV--Move to/from Control Registers" and
     [IA32-v3a] 5.15 "Interrupt 14--Page Fault Exception
     (#PF)". */
  asm ("movl %%cr2, %0" : "=r" (fault_addr));

  /* Turn interrupts back on (they were only off so that we could
     be assured of reading CR2 before it changed). */
  intr_enable ();

  /* Count page faults. */
  page_fault_cnt++;

  /* Determine cause. */
  not_present = (f->error_code & PF_P) == 0;
  write = (f->error_code & PF_W) != 0;
  user = (f->error_code & PF_U) != 0;

  /* To implement virtual memory, delete the rest of the function
     body, and replace it with code that brings in the page to
     which fault_addr refers. */
  printf ("Page fault at %p: %s error %s page in %s context.\n",
          fault_addr,
          not_present ? "not present" : "rights violation",
          write ? "writing" : "reading",
          user ? "user" : "kernel");
  kill (f);
}
```

System Calls

31

System Calls allow *user processes* to ask the *kernel* to perform operations they don't have permission to do themselves.

...this is done through syscall_handler()

System Calls

32

Read the system call number (f->esp)

Read the arguments above the stack pointer

Pass to the appropriate function (that you write)

Return any results in f->eax

Avoid duplicate code, keep it clean

Array of Function Pointers

33

the functions

```
int sum(int a, int b);
int subtract(int a, int b);
int mul(int a, int b);
int div(int a, int b);
```

the array

```
int (*p[4]) (int x, int y);
```

setup

```
int main(void)
{
  int result;
  int i, j, op;

  p[0] = sum; /* address of sum() */
  p[1] = subtract; /* address of subtract() */
  p[2] = mul; /* address of mul() */
  p[3] = div; /* address of div() */
[...]
```

use

```
// op being the index of one of the four functions
result = (*p[op]) (i, j);
```

34

Filesystem Calls

file.h and filesys.h

Don't need to modify these.

Syscalls use *file descriptors* and the file system uses *struct file*.

Make sure to **synchronize** access to the file system -- only one change at a time.

STDOUT_FILENO and STDIN_FILENO

35

process_wait()

Calling function blocks (using synchronization) waiting for the child process to exit.

Syscall wait() is trivial after this is complete.

Returns exit status of child, or -1

Most work of all system calls.

READ SPECIFICATION CAREFULLY

36

process_wait()

Child may exit **before** parent calls process_wait()

Parent may **never** call process_wait()

Child may exit **after** parent is gone

37

Deny Write to Executables

Don’t allow changes to files that are currently loaded as executables.

Use “file_deny_write()” to prevent writes to an open file.

Use “file_allow_write()” to allow.

Executables should be kept open and unwritable as long as the process is running

38

Order of Implementation

Temporarily set up the stack

Implement safe memory access

Basic system call handler

Implement exit system call

Implement write system call (to console)

Make process_wait() an infinite loop

Everything else...

39

Walkthrough

code not walls :-(

40

Do This!

Match coding style

Package for grading correctly

Answer questions in DESIGNDOC

Think about alternate designs

41

Argument Passing

String parsing

No limit to “command” size

Avoid stack overflow, abort if needed

Synchronize parent’s startup of child

42

System Calls

Keep it clean, abstract, easily extensible

Synchronize access to filesystem
(don’t disable interrupts!)

Map *fd* to *struct file* * in your code

Synchronize *process_wait()* properly

Cover all cases of *process_wait()*

43

User Memory Access

Choose an implementation (there are two)

Don't pass bad data further into kernel

Don't get too creative or complex

Look ahead, what will work in Project 3?

44

End

Pintos Project 2 - User Programs