Pintos Overview

November 20, 2018

Welcome to Pintos World!

- What is Pintos?
  - An instructional operating system
    - Developed by Stanford University
  - A real OS for the 80x86 architecture
    - Run on a regular IBM-compatible PC or an x86 emulator.
  - Written in C language
    - You will need only code in C.
- What is our task?
  - Understand the underlying design of Pintos OS and add more functions to extend it and make it complete.
- These projects are hard. Hang on!
  - "CS 140 is the **hardest coding class** you'll take at Stanford. It might be the hardest CS class you take at Stanford."

Environments

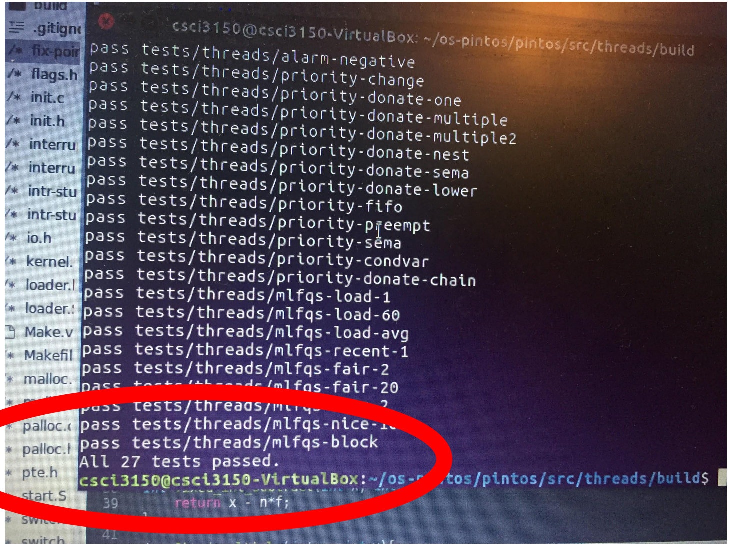
- QEMU
    - Emulator provides the full emulation of 80x86 physical cpu and its peripheral devices.
    - We will run Pintos on this emulator.
    - Qemu makes it easy to **develop and debug** Pintos projects(refer to Pintos Guide).
- Linux + QEMU
    - Compile the os kernel in linux environment and then run it on QEMU emulator.

Working with Pintos

- Each of the four projects has its own main directory:
  - Project 1: src/threads
  - Project 2: src/userprog
  - Project 3: src/vm
  - Project 4: src/filesys
- Each project consists of two parts:
  - Programming
  - Design Document
- For each project, type ***'make'*** in the project's main directory to compile your project
  - E.g. Type ***'make'*** in src/threads
  - This will create a new directory 'build/' (linux kernel is inside it)

Working with Pintos...

- Type ‘***make check***’ to run all the tests in the ‘*build/*’ directory
    - You can also run a single test directly.
- Please refer to Pintos Guide to know how to debug on Pintos.



How does Pintos work?

- Booting
    - Entry point(Boot loader) of Pintos is ***threads/start.S***(this is an assembly file)
    - *start.S* will initialize operating system resources, and call main() in **threads/init.c**
- ***main()*** will parse command line arguments, setup kernel memory, initialize the interrupt system, and call *thread_start()* in ‘*threads/thread.c*’ for the main thread
    - The main thread is the ancestor of all later derived thread.

How does testing work?

- After you finish one part of Pintos, you want to test if it works properly.
  - E.g. pintos -q run **my-test**
- Principle under testing
  - The main thread will run the test file you provide in the command parameter.
  - The pintos kernel command line is stored in the boot loader.
    - The **pintos** command actually modifies a copy of the boot loader on disk each time it runs the kernel, inserting the command line the user supplies into the loader.
    - Then at boot time, the kernel(main thread) reads those arguments out of the load loader.
    - Not elegant, but simple and effective.

Run “hello world” on your Pintos OS

- Simple task
  - Write a test that prints “Hello World!”
- Hints
  - The files that will be modified under “src/tests/threads/”
    - test.h
    - test.c
    - add new file“hello-world.c”
    - Make.tests
  - Use ‘msg()’ function to print.
  - Clean and recompile the kernel under “src/threads/” after above steps
  - Run ‘pintos -q run hello-world’

Basic of Pintos: Threads

1. Every thread has the ***struct thread*** to store some basic information
   a. This struct is also be used as the stack.
2. The Pintos is a single processor system.
   a. Alhough support multi-thread(multi-programming), everytime only one thread is running.
   b. Preemptible kernel
3. The whole system is motivated by the **timer interrupt**.
   a. Round-robin: every thread has the fixed time slice(e.g. 4 time ticks) to execute on CPU.
   b. At every timer tick, the running thread is interrupted by the timer interrupt handler.
   c. Time interrupt handler increases the system time, checks if the time slice of running thread expires, and if so, schedule next thread to execute.

```
      4 kB +---------------------------------+
           |          kernel stack           |
           |               |                 |
           |               |                 |
           |               V                 |
           |         grows downward          |
           |                                 |
           |                                 |
           |                                 |
           |                                 |
           |                                 |
           |                                 |
           |                                 |
           |                                 |
   thread) +---------------------------------+
           |              magic              |
           |                :                |
           |                :                |
           |              status             |
           |               tid               |
      0 kB +---------------------------------+
```

```
struct thread
{
    /* Owned by thread.c. */
    tid_t tid;
    enum thread_status status;
    char name[16];
    uint8_t *stack;
    int priority;
    struct list_elem allelem;
        */

    /* Shared between thread.c and synch.
    struct list_elem elem;
#ifdef USERPROG
    /* Owned by userprog/process.c. */
    uint32_t *pagedir;
#endif

    /* Owned by thread.c. */
    unsigned magic;
};
```

Programming on Pintos: Utilities

- Position: ‘**src/lib/kernel/**’
- List
  - Initialize(), insert(), delete()
  - Looping through Lists
    - list_begin(), list_end(), list_next(); Like STL
- Hash table
  - Initialize(), hash_insert(), hash_delete()
- Bitmap
  - bitmap_set(), bitmap_set(), bitmap_reset()..

First part of Project 1: Alarm Clock

- Reimplement ***void timer_sleep(int64_t ticks)***
  - Defined in ‘devices/timer.c’
  - Requirement: avoid busy waiting.
- Original implementation

```c
/* Sleeps for approximately TICKS timer ticks.  Interrupts must
   be turned on. */
void
timer_sleep (int64_t ticks)
{
  int64_t start = timer_ticks ();

  ASSERT (intr_get_level () == INTR_ON);
  while (timer_elapsed (start) < ticks)
    thread_yield ();
}
```

Alarm Clock: test cases

- Five test cases to test alarm clock.
- Two test cases of them
  - **Alarm-negative**: Test timer_sleep(-100), only requirement is that it does not crash.
  - **Alarm-simultaneous**: create **N** threads, each of which sleeps a different, fixed duration, M times. Records the wake-up order and verifies that it is valid.

alarm-negative.c

```c
/* Tests timer_sleep(-100).  Only requirement is that it not crash. */

#include <stdio.h>
#include "tests/threads/tests.h"
#include "threads/malloc.h"
#include "threads/synch.h"
#include "threads/thread.h"
#include "devices/timer.h"

void
test_alarm_negative (void)
{
  timer_sleep (-100);
  pass ();
}

```

alarm-simultaneous.c   thread.h   timer.c   proc

```c
/* Creates N threads, each of which sleeps a different, fixed
   duration, M times.  Records the wake-up order and verifies
   that it is valid. */

#include <stdio.h>
#include "tests/threads/tests.h"
#include "threads/init.h"
#include "threads/malloc.h"
#include "threads/synch.h"
#include "threads/thread.h"
#include "devices/timer.h"

static void test_sleep (int thread_cnt, int iterations);

void
test_alarm_simultaneous (void)
{
  test_sleep (3, 5);
}

/* Information about the test. */
struct sleep_test
  {
    int64_t start;              /* Current time at start of test. */
    int iterations;             /* Number of iterations per thread. */
    int *output_pos;            /* Current position in output buffer. */
  };

static void sleeper (void *);

/* Runs THREAD_CNT threads thread sleep ITERATIONS times each. */
static void
test_sleep (int thread_cnt, int iterations)
{
  struct sleep_test test;
  int *output;
  int i;

  /* This test does not work with the MLFQS. */
  ASSERT (!thread_mlfqs);

```

Alarm Clock: overview of my solution

- Not wake up during sleeping period
  - Create a new linked list to store sleeping thread.
    - Every time a thread calls timer_sleep(), put it into the **sleeping_thread list,** and set its state as **blocked.**
  - Since the scheduler only schedules the threads on the ready queue, the thread will not wake up during sleeping period.
- Wake up(put it on the ready queue) at **x** timer_ticks
  - **Timer interrupt handler**
    - At every timer interrupt, the handler traverse the sleep_thread list to check if the thread has slept enough, if so, put it on the ready queue.

Alarm Clock: my implementation

- void timer_sleep
  - If ticks <= 0, directly return
  - Record current_time and waitting_time in the current *thread* struct.
  - Invoke sema_sleep()
    - Put current thead into a linked list and then shcedule other threads.
    - How to **synchronize** the access to the shared linked list?(**Appendix A.3**)
- void timer_interrupt
  - Timer interrupt handler
  - Invoke sema_wake(), check and wake up the threads who has slept enough.

```c
void
timer_sleep(int64_t ticks)
{
  if(ticks <= 0){
    return;
  }

  /*ticks > 0, the thread is put on blocked queue*/
  struct thread *t = thread_current();
  t->start = timer_ticks();
  t->waittime = ticks;
  sema_sleep(&sleeping_thread);
}
```

```c
/* Timer interrupt handler. */
static void
timer_interrupt (struct intr_frame *args UNUSED)
{
  ticks++;
  /*wake up the thread that has slept enough*/
  sema_wake(&sleeping_thread);
  thread_tick ();
}
```

Overview of All Projects

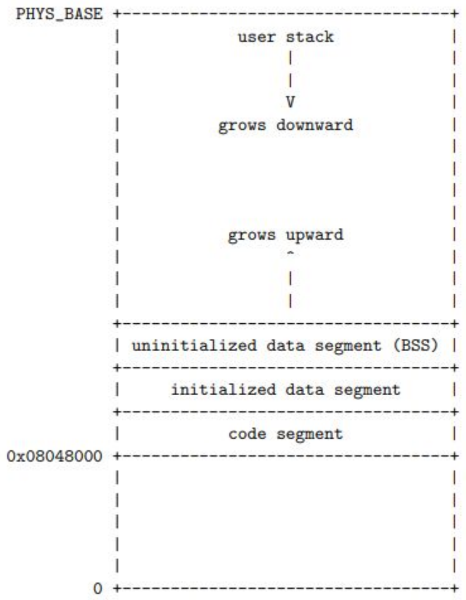
- Project 1: Threads
- Project 2: User Programs
- Project 3: Virtual Memory
- Project 4: File Systems

Project 1: Threads

1. Alarm Clock
    a. Reimplement timer_sleep().
2. Priority Scheduling
    a. In the ready queue, the thread who has the highest priority should execute first.
    b. Problem: priority inversion
        i. Thread H(high priority) must wait for the lock held by thread L(low priority). But thread L may never get the CPU.
        ii. Solution: priority donation H->L, L gives up the donation after lock release
3. Advanced Scheduler
    a. Multi-level feedback queue scheduler(4.4BSD scheduler)
    b. Not include priority donation

Project 2: User Programs

1. Setup Stack
   a. Load a executable file to the memory and set up initial stack for the new process.
2. System call handler
   a. System call: allow a user program to perform a kernel level task.
   b. Systems calls
      i. Process execution
         1. exit()
         2. exec(char *cmd_line)
         3. wait(pid_t pid)
      ii. File operation
         1. create(), remove()
         2. open(), read(), write(), close()
         3. Denying writes to process's executable

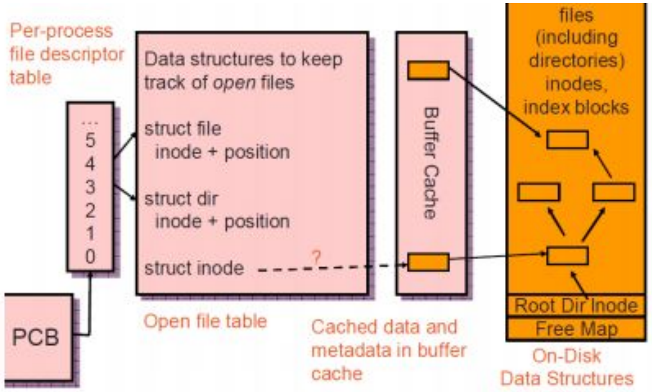


Project 3: Virtual Memory

1. Project 3 and 4 are the hardest parts of Pintos projects
    a. Collaboration with group members.
2. Part 1: Growing the Stack
    a. If the process wants to have more memory-> page fault
    b. Validate the address, allocate new page
    c. Data strucutre: Page table, frame table....
3. Part 2: Swap, Eviction, and Reclamation
    a. Access more memory than available physical memory
        i. Solution: **Swap**
    b. Eviction: Least recently used(LRU) policy, clock policy
    c. Reclamation: read the page in swap back to memory
4. Part 3: Memory mapping
    a. mapid_t mmap(int fd, void* addr)
    b. void munmap(mapid_t mapping)

Project 4: File System

1. Part 1: Buffer Cache
    a. The bridge between memory and disk.
2. Part 2: Indexed and Extensible Files
    a. Last question in your final exam
    b. To support file growth.
3. Part 3: Subdirectory
    a. Implement a hierarchical name space
        i. Basic file system: all files are at root
    b. Upate and add system calls
        i. open(), close(), remove()
        ii. mkdir(), chdir()
        iii. relative path, absolute path.



Suggestions

- Read pintos guide(ours), pintos document(Stanford) and design document before you start coding.
- Useful tools
    - GDB, version control system(Github)

Thank you!

Q&A