TDDB68 2015
Lesson 2
Introduction to Pintos
Assignments (1) and 2

Erik Hansson

erik.hansson@liu.se

Outline

• Administration
• Introduction to Pintos
• Important concepts
  – Call stack
  – Interrupts
• General description of labs
  – Lab 1
    • User programs
    • System calls
  – Lab 2
    • Threads
    • Timer
    • Interrupts
Administration

• Sign up in Webreg (link on course page)
  - Contact me if you missed to sign up last week.
  - There are 4 groups (A, B, C & D)
  - Most of the labs are scheduled in parallel
  - Contact me (erik.hansson@liu.se) if you have problems

• You will work in pairs
  - Both students will be tested at the demonstration
  - Try to find a partner if you are alone (today during the break...)

• The solution you present must be yours!
A new kind of complexity

• These labs have a reputation of being difficult.
  – Often the first experience with a relatively large piece of software.
    • New kind of complexity.
    • Manage your changes (version control).
  – Read a lot!
  – You will spend more time trying to understand than writing code
    • manual
    • source

Your programs so far?  Pintos.  Most real systems.
What is Pintos?

• Pintos is an educational OS
  – Developed at Stanford
• Works on x86 hardware
• Pintos is clean and well documented C
• Your task: add functionality to Pintos
• The name Pintos is a play on words (mexican food)

Pinto beans
General Description of Labs

• Lab 00: “Introduction to C Programming”
  - Mostly pointers

• Lab 0: “Introduction and installation”
  - Introductory lab
  - Environment
    • debugging

• Lab 1: “System calls”
  - First real lab
  - System calls
  - Single user program
  - Console
General Description of Labs

• Lab 2: “Threads, interrupts and synchronization”
  - Threads
  - Interrupts
  - Timer

• Lab 3: “Execution, termination and synchronization of user programs”
  - Handling program arguments
  - Execution of several user programs
  - Termination of ill-behaving user programs
General Description of Labs

• Lab 4: “File system”
  – Synchronization with multiple readers and writers
Remember: Lab Rules

• Every member of the group participate equally and every member should be able to answer questions related to labs

• Before doing the lab answer *preparatory questions* on the lab page
  - Recall: \texttt{time(reading) > time(coding)}

• No copying of source code from any sources; no cheating, please.
Plan your time to get bonus

- Pass assignments on time to get 4 bonus points on the exam (i.e., 20% towards the threshold for passing). Only for new students.

- Soft deadlines:
  - Lab 00 – Should be done...
  - Lab 1 – Should be (almost) done now...
  - Lab 2 – February 12.
  - Lab 3 – March 3.
  - Lab 4 – March 11.

- FINAL DEADLINE (bonus): March 17. (in webreg)
  - Try to finish before the last lab session.
  - Correction after this session is up to your assistants goodwill.
General Algorithm to Complete the Labs

1. Answer most of the preparatory questions.
2. Understand what you are supposed to do (read instructions carefully)
3. Discuss all the issues with your lab partner to make sure that he/she also understands (that’s your future help!)
4. Understand the relevant source code.
5. Work hard, you must work on non-scheduled time.
6. Follow the soft deadlines.
Important concepts: The call stack and local variables

• When a function is called, a stack frame is pushed on the call stack.
• At return the frame is popped.
• The stack frame contains
  – parameters,
  – return address,
  – local variables.
Call stack: example overflow

• void foo() {
   int a[7000000]; /* huge local array, overflow*/
   /* ... */
}

int main(int argc, char **argv) {
    foo();
}

• We have seen two problems:
  – Dangling pointers
  – Large data doesn't fit on the stack (including recursion)
• The solution to both problems: allocate blocks of memory on the heap
  – malloc() and free()
Interrupts, recapitulation

• The CPU normally operates in a fetch-execute cycle
  - Fetch the next instruction
  - Decode instruction
  - Execute this instruction
  - Update program counter
  
  Check for interrupt!!

• Sometimes things happen where we must temporarily stop this cycle
  - e.g. user input (mouse, keyboard), or a clock tick (Lab 2)

• The method used to stop the cycle is an interrupt
  - Save state (PC, ...)
  - Jump to interrupt handler (interrupt service routine (ISR)) in the kernel
  - Restore processor state
  - Jump back and continue fetch-execute cycle
Interrupts in Pintos

• Internal interrupts
  – caused by software
    • invalid memory access (seg. fault)
    • division by 0
    • system calls (Lab 1)

• External interrupts
  – comes from outside the CPU
    • keyboard
    • serial port
    • system timer, every clock tick (Lab 2)

Ext. interrupts can be delayed
intr_disable();
->synchronization!
Interrupt context in Pintos

- The handler of an external interrupt is not allowed to block:
  - Sleep, sema_down, or lock_acquire

- Consider the following scenario:
  - Thread $x$ is running in the kernel
  - Thread $x$ enters a critical region: `lock_acquire(L)`
  - An external interrupt occurs; it will be handled in the kernel running as $x$
  - Interrupt handler does `lock_acquire(L)`
  - DEADLOCK: $L$ will never be released.
Lab 1: Pintos kernel overview

Pintos kernel

- Synchronization primitives
- File system
- Drivers
- Hardware
- Interrupt handlers: syscall, timer, kbd, etc
- System call
- Scheduler
- Threads
- Page table

User programs

LAB1
Lab 1
Systems calls at a high level

• System calls:
  - communication between user program and the kernel
  - functions, called from the user program and performed by the kernel
  - computers use *interrupts* to accomplish that switch from user code to system code

![Diagram showing the relationship between user, kernel, syscall handler, drivers, and scheduler]
Lab 1
What to implement?

• You will need to implement these system calls:
  - create - creates a file.
  - open - opens a file.
  - close - closes a file.
  - read - reads from a file or the console (the keyboard).
  - write - writes to a file or the console (the monitor).
  - halt - halts the processor.
  - exit - Terminates a program and deallocates resources allocated to it.

  • (will be extended in Lab 3, don't worry about saving exit status yet.)
Lab 1:

• `bool create (const char *file, unsigned size)`

• Example: `create("file.txt", 1000);`

How to get them?

Answer: they are passed on the stack: `f->esp`

Hint:
… note that, in order to get a string, you will need get a pointer from esp. This pointer will point to the first element of the string …

How to return a value?

Answer: `f->eax`
The syscall handler (1)

- Currently the syscall handler kills the thread
- You must:
  - Get the syscall number
    - Remember `f->esp`
      - pointer
  - “Get” the correct number of arguments
    - Each system call are defined using different...
  - Hint: Pointers and arrays are very related in C
    `ptr + i == &(ptr[i])`

```c
static void syscall_handler (struct intr_frame *f)
{
    printf ("system call!\n");
    thread_exit ();
}
```
The syscall handler (2) & thread

static void syscall_handler (struct intr_frame *f)
{
    printf("system call!\n");
    thread_exit();

    //Get syscall number from stack!

    //HALT:
    //shutdown the machine

    //WRITE:
    //Get parameters...
    //call the right functions..

    //....
}

thread_current()
Lab 1: File descriptors

- **int open (const char *file)**
  Open returns a file descriptor (fd) that is a handle to the open file

- Your task is to map the fd's to open files

- 0 and 1 must be always reserved for the console!

- Each process shall have an **independent set** of file descriptors
  - You can place it in `struct thread` in thread.h
  - Use suitable data structure...
  - Pintos has a bitmap data structure..

- User program shall be able to **have up to 128** files open at the same time.

- File descriptors are not inherited by child processes.

- Pintos does not manage file descriptors yet, **you need** to implement this
Test program lab1test.c

• A small test program is provided, it will:
  - Create files
  - Open files
  - Test console
  - Try to use fake fd's

• Gotchas
  - Remove all files before running it again:
    - pintos --qemu -- rm test0 rm test1 rm test2

• If you have problems when running it. Try to isolate the problem
  - Create your own (simpler) tests in (example directory)
    - Don't forget to add them in the Makefile

• Your Lab 1 implementation will be checked more thoroughly once Lab 3 is finished
  - (gmake check won't work now)
Lab 2: Pintos kernel overview

Pintos kernel

- Synchronization primitives
- Interrupt handlers: syscall, timer, kbd, etc
- Timer
- File system
- Drivers
- Hardware
- System call
- User programs
- Scheduler
- Threads
- Page table

LAB2
Lab 2 overview

• First contact with
  – Multiple threads
  – External interrupts
  – Scheduling

• Your task: implement the function
timer_sleep(int64_t ticks)

• The current solution is not acceptable:

```c
int64_t start = timer_ticks ();
while (timer_elapsed (start) < ticks) {
    thread_yield (); // Wasting CPU-cycles!
}
```
Lab2:

Thread states

- Yielding puts a thread back in the ready queue
- Ready threads are scheduled by the kernel
- Sleeping threads should be blocked

```c
int64_t start = timer_ticks();
while (timer_elapsed (start) < ticks) {
    thread_yield (); //Wasting CPU-cycles!
}
```
Example with Threads

- One time unit is a tick
- Every 4\textsuperscript{th} tick threads are forced to yield
- Default in Pintos: 100 ticks per second
The critical section problem

• **Critical Section**: A set of instructions, operating on shared data or resources, that should be executed by a single process without interference

• Properties:
  – **Mutual exclusion**: At most one process should be allowed to operate inside at any time
  – **Consistency**: Intermediate states of shared data not visible to other processes

• Pintos has 3 basic synchronization primitives...
Semaphores

• **Semaphore s:**
  - shared integer variable
  - two atomic operations to modify s: `sema_down(s)` and `sema_up(s)`
  - Sometimes called P() and V()

Example: At most five passengers in the car

• Initialize s to 5
• function use_car() {
  `sema_down(s);`
  `enter_car();`
  `leave_car();`
  `sema_up(s);`
}
void sema_down (struct semaphore *sema) {
    enum intr_level old_level;
    ...
    old_level = intr_disable ();
    while (sema->value == 0) {
        list_push_back ( cur_thread );
        thread_block ();
    }
    sema->value--;
    intr_set_level (old_level);
}
Semaphores in Pintos

```c
void sema_up (struct semaphore *sema) {
    enum intr_level old_level;

    old_level = intr_disable ();
    if (!list_empty (&sema->waiters)){
        thread_unblock (list_pop_front(…) );
    }
   .sema->value++;
    intr_set_level (old_level);
}
```

value >0 is a green light
A good way to wake someone up:

```c
// A global pointer
struct semaphore *sleeper;

// SLEEPER thread:
struct semaphore s;
sema_init(&s, 0);
sleeper = &s;
sema_down(&s); // zzz..

// WAKER
sema_up(sleeper);
```

Hint: extend so that you have a list of semaphores and wakeup times!
Locks

• Similar to a binary semaphore
  – Either locked (0) or unlocked (1)

• Only the locker can unlock it
  – The locker has the key

• Advantage: Good to use for mutual exclusion
  – Recall: semaphores can be upped by anyone.

• Essential for security, for example, access to the shared data, arrays, lists, and etc.
Conditions

• **Queuing mechanism** on top of Locks

• **Prevents** deadlock situations by temporarily allowing someone else in the critical region:

```c
lock_acquire(l);
... //Must wait for something to happen
cond_wait(c,l);  //Waiting inside crit. sec.
...
lock_release(l);
```

• **Example:** Fixed size synchronized list.
  - writing when internal buffer is full
  - reading when internal buffer is empty
Pintos list

- Pintos has a clever doubly linked list (see lib/kernel/list.h)
- Example: you need a list of `struct foo` items
- Add a `struct list_elem` to `foo`

```c
struct foo{
    ... //data
    struct list_elem elem;
};
struct list foo_list;
init_list(&foo_list);
```

- `for` (e = list_begin (&foo_list);
  e != list_end (&foo_list);
  e = list_next (e)) {
  struct foo *cur_foo = list_entry(e, struct foo, elem);
  ... //use cur_foo ...
}

foo are not put into the list, the list is attached to foo
Lab 2: What to do...

- Implement the function (devices/timer.c)
  - timer_sleep(int64_t ticks)
  - Some more...

- The current solution is not acceptable:

  ```c
  int64_t start = timer_ticks ();
  while (timer_elapsed (start) < ticks) {
    thread_yield (); // Wasting CPU-cycles!
  }
  ```

- Make threads being able to sleep
  - Keep track of them
  - Wake up times

- Check if it is time to wake (some of) them up

- Disable interrupts
Lab2 --- Testing

• Run the tests
  - alarm-single
  - alarm-multiple
  - alarm-simultaneous
  - alarm-zero
  - alarm-negative

• gmake SIMULATOR=--qemu check
  - Must pass all tests!

• Run individual tests like this:
  pintos --qemu -- run alarm-simultaneous

• Will also pass before you do any modifications
Remember: Debugging

- Use DDD

- `printf`
  - Do not forget to remove them...
  - Otherwise will `gmake check` fail!!!

- `backtrace`
General advice

• Work hard

• You need to spend (much) more than the scheduled time!

• **Plan your** time **together** with your lab partner!
  
  • Week 6: 4 hours...
  • Week 7: 2 hours...
  • Week 9: 6 hours...

  Are scheduled...

• Read the manual (really).

• Read the source code; use emacs tags:
  
  - `gmake TAGS` (in pintos/src)
  - `M-.` go to definition
  - `M-0 M-.` go to next definition
  - `M-*` jump back
Next lesson

• February 11.
• In ~1 week from now
• Lab 3 & Lab 4

exit(0);