TDDB68 2015
Lesson 1
Introduction to Pintos
Assignments (00), 0, 1
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
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). Full
    - New students have priority.
    - Old students: contact your lab assistant from previous year(s).
  - Erik, Nicolas, Simon and Zeinab
  - 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. Should be done by now!

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

- **Lab 1: “System calls”**
  - First *real* lab
  - System calls
  - Single user program
  - Console, (input and output)
General Description of Labs

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

• Lab 3: “Execution, termination and synchronization of user programs”
  - Exec and wait system call
  - 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
    - Readers writers algorithm
    - Multiple threads
  • Hard to debug...
Lab Rules

- **Every member** of the group participate **equally** and every member should be able to answer questions related to labs
  - When presenting solutions both of you must be present.
  - Assistants can split the group, if needed.

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

- No copying of source code from any sources; no cheating, please.

- If you discuss particular concepts, problems, implementations etc. with other groups, write that in your code...
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 by now...
  - Lab 1 – January 30.
  - 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.
Lab 00

- You will need to debug a small program `debugthis.c` with `ddd` after compiling it with `gcc -g`.
- Create a linked list.
- If you are having much problems with this lab you should focus on learning C before you proceed.
  - I suggest the book known as K&R (Kernighan & Ritchie, The C Programming language).
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

int * foo(int f_arg) {
    int a=f_arg;
    return &a;
}

int main(int argc, char **argv) {
    int * ap;
    ap = foo(17);
    bar();
    *ap += 1;
}

CALL STACK

<table>
<thead>
<tr>
<th>stack frame of main()</th>
</tr>
</thead>
<tbody>
<tr>
<td>argc, argv</td>
</tr>
<tr>
<td>The return address</td>
</tr>
<tr>
<td>int * ap (4 bytes)</td>
</tr>
</tbody>
</table>

Stack growth
Call stack: example

- int * foo(int f_arg) {
  int a=f_arg;
  return &a;
}

int main(int argc, char **argv) {
  int * ap;
  ap = foo(17);
  bar();
  *ap += 1;
}
Call stack: example

```
• int * foo(int f_arg) {
    int a=f_arg;
    return &a;
}

int main(int argc, char **argv) {
    int * ap;
    ap = foo(17);
    bar();
    *ap += 1; /* What will happen here? */
}
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

- 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

- 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 (trap)
  – 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

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: $\text{lock_acquire}(L)$
  - An external interrupt occurs; it will be handled in the kernel running as $x$
  - Interrupt handler does $\text{lock_acquire}(L)$
  - DEADLOCK: $L$ will never be released.
The Pintos boot process

• You boot pintos like this:
  
  \texttt{pintos --qemu \ -- run alarm-single}

  \begin{itemize}
  \item Perl script that runs pintos in a simulator
  \item Arguments to the run script
  \item Arguments passed straight to the kernel
  \end{itemize}

• Assembler magic in \texttt{loader.S} (don't worry about it)

• \texttt{threads/init.c, main()}: initializes the subsystems of Pintos

• \texttt{run_actions()}: runs the tasks given on the kernel command line
Lab 0
Installing and debugging

- Debugging (from the build directory):

  Starting Pintos:
  pintos --qemu --gdb --dport=port_number -- run testname

  Start ddd
  ddd --gdb --debugger pintos-gdb kernel.o&

  Attach ddd to your Pintos, type in the ddd prompt
  target remote localhost:port_number

- Thin clients -> choose port_number that no one else uses
Lab 1: Pintos kernel overview

Pintos kernel
- Synchronization primitives
- Interrupt handlers
- File system
- System call
- User programs
- Timer
- Drivers
- Scheduler
- Threads
- Page table
- Hardware
Lab 1
System calls
- User program (only one at a time)
- System calls
- Console (keyboard, monitor)
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
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. (shut down the machine)
  - **exit** - Terminates a program and deallocates resources allocated to it.

  *(will be extended in Lab 3, don't worry about saving exit status yet.)*
System calls

• Number of system calls

• Pintos
  - Lab1: 7
  - Lab2: 0
  - Lab3: 9
  - Lab4: 13

• Linux
  - Around 320 system calls.

• Windows
  - More than 1000 system calls
Lab 1 Files

You have to have a look at:
- pintos/src/lib/user/syscall [.h|.c] – the wrapper...
- userprog/syscall [.h|.c] – Your implementation of system calls!
- threads/interrupt.[h|c] – important structures
- threads/thread.[h|c] – Thread implementation. You need to add...
- lib/syscall-nr.h – system call constants
- filesys/filesys.[h|c] Pintos file system implementation...
- examples/lab1test.c – the “official” test program for this lab
- filesys/file.[h|c] – useful functions for operations with files. Things which you don’t find in “filesys”, you find here
- userprog/process.c
Lab 1: How Do I Start? (1)

• **STEP 1:**
  
  - Understand what a user program is
  
  - Look into the “examples” directory and a couple of the user programs, especially “halt.c”
  
  - Look into **Makefile** in this directory to understand how the user programs are compiled
  
  - Feel free to add your own if you need!
  
  - Compile the user programs by issuing “gmake”

```c
#include <syscall.h>
int main (void){
    halt ();  //shuts down the system
}
```
Lab 1: How Do I Start? (2)

• STEP 2: Dealing with Pintos disk
  - Pintos uses a simulated hard-drive
  - Can hold up to 16 files
  - Filenames are up to 14 chars long
  - To create it, do in userprog/build:
    pintos-mkdisk fs.dsk 2
  - This will create a file $fs\text{.dsk}$ with a 2MB simulated disk in the directory.
  - Format the simulated disk:
    pintos --qemu -- -f -q
Lab 1: How Do I Start? (3)

• Put a file on the disk:
  pintos --qemu -p ../../examples/halt -a halt -- -q

• To examine the disk:
  pintos --qemu -- ls
  pintos --qemu -- cat filename
  pintos --qemu -- rm file1 rm file2 ls

• Example: to run halt:
  - pintos --qemu -p ../../examples/halt -a halt -- -q
  - pintos --qemu -- run halt
Lab 1: How Do I Start? (4)

• STEP 3:
  - into userprog/process.c, find setup_stack()
  - *esp = PHYS_BASE;
  - change to *esp = PHYS_BASE - 12;

• STEP 4:
  - For now, make process_wait() an infinite loop or the kernel might power off too quickly
  - Because the kernel calls process_wait on the first program.
  - (recompile pintos: gmake)
Lab 1: Shutdown Example

• The simplest user program:

• halt.c:

    #include <syscall.h>
    int main (void){
        halt (); //shuts down the system
    }

• Hint1: Halt is simple; do it first.
  – No arguments.
  – No return value.

• Hint2: Do the console part of the Write system call next so you can use printf in user programs
Lab 1: Create Example

- 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 ...
Lab 1: Create Example

- To implement the system calls, which operate with files, look at `filesys/filesys.[h|c]` and `filesys/file.[h|c]`.
- Everything is already done there for you! Just call the functions.
- But what if the user program passes a bad pointer? e.g.: `create(NULL, 0);`  
  - the kernel would crash when reading the filename!
  - you will fix this in Lab 3.
Lab 1
Systems calls at a high level

**user_program.c**

```c
bool flag;
flag = create("file.txt", 1000);
```

**Pintos kernel**

```c
bool create(const char *file, unsigned initial_size) {
    return syscall2(SYS_CREATE, file, initial_size);
}
```

**syscall_handler(...)**

// Your code here!

**Pintos/src/lib/user/syscall.c**

**System call wrapper**

```c
bool create (const char *file, unsigned initial_size) {
    return syscall2(SYS_CREATE, file, initial_size);
}
```

**Pintos/src/userprog/syscall.c**

**Calling corresponding function in the wrapper**

**Calling corresponding exception in the kernel**

**Return result back to the user program**
The syscall handler (1)

- Currently the syscall handler kills the thread

- You must:
  - Get the syscall number
    - Remember \texttt{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
  \begin{verbatim}
  ptr + i == &(ptr[i])
  \end{verbatim}

\begin{verbatim}
static void syscall_handler (struct intr_frame *f UNUSED)
{
    printf ("system call!\n");
    thread Exit ();
}
\end{verbatim}
The syscall handler (2)

```c
static void syscall_handler (struct intr_frame *f UNUSED)
{
    printf("system call!\n");
    thread_exit ();
    //Get syscall number from stack!

    //HALT:
    //shutdown the machine

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

    //....
}
```
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
The Pintos bitmap by example

```c
#include "lib/kernel/bitmap.h"
#define FOOSIZE 128
static struct bitmap * foomap;
...
{
    /* Allocate memory for our bitmap */
    foomap = bitmap_create (FOOSIZE);
    if (foomap == NULL)
        PANIC ("Map is too big");
    /* Set 0th bit to 1 */
    bitmap_mark (foomap, 0);
    /* Find a bit which is 0 and mark it */
    int fnd = bitmap_scan_and_flip (foomap, 0, 1, 0);
    /* Free the bitmap (otherwise a leak) */
    bitmap_release (foomap);
}
```
Lab 1: Read System Call

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

- Reads `size` bytes from the file open as `fd` into `buffer`.
  - Returns the number of bytes actually read,
  - or -1 if the file could not be read.

- **Fd 0 == STDIN_FILENO**
  Should read from the keyboard using `input_getc()`
  (see: devices/input.h).
Lab 1: Write System Call

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

- Writes `size` bytes from `buffer` to the open file `fd`
- Returns the number of bytes actually written or -1 if the file could not be written
- The expected behavior is to write as many bytes as possible up to end-of-file

- Fd 1==STDOUT_FILENO
  use **putbuf ()**
  (check lib/kernel/stdio.h and lib/kernel/console.c)
Lab 1: Exit System Call

- **void exit (int status);**

- Terminates the current user program, returning *status* to the kernel.

- By convention, a *status* of 0 indicates success and nonzero values indicate errors.

- This system call will be extended in Lab 3, for now you won't use *status*.

- **Important:** Free resource in `thread_exit()` function located in `process.c`.

CLOSE ALL OPEN FILES
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)
Subversion

- `svn up` #get latest version from repository
- `svn commit` #record changes in repository
- `svn st` #show which files are modified and new
- `svn diff` #show difference between working copy and most recently checked in revision
- `svn -r 17 diff` #same, but compared to revision 17
- `svn -r 17 -x -wp diff` #ignore changes in whitespace and include context information
- `svn log` #show commit log
- `svn -v log` #also list which files are modified
(Advanced subversion usage)

• Ignore this if you only do labs on a single IDA account

• Starting a subversion daemon
  - cd svnrepository/conf
  - Add users and passwords in authz and passwd
  - Edit svnserve.conf to not allow anonymous access, and uncomment some lines...
  - Login to astmatix.ida.liu.se using ssh
  - svnserve -d -r $HOME/svnrepository --listen-port 12345
  - Pick a port number no one else is using (>1024)
  - Now, on another computer:
  - svn co svn://astmatix.ida.liu.se:12345/pintos pintos

Astmatix reboots every morning...
Debugging tips

• Read Appendix E in the documentation.

• `ASSERT(mypointer != NULL)`

• Bytes in memory that have been freed are set to `0xcc`
  
  - If a pointer `== 0xcccccccccccc` something probably freed the memory.

• If you get a “Kernel Panic” use the `backtrace` tool...
The backtrace tool can make call stacks useful. Read "Backtraces" in the "Debugging Tools" chapter of the Pintos documentation for more information.

QEMU: Terminated

```
2 mater@mina1.ida.liu.se ~/pintos/pintos/src/threads> backtrace
Call stack: 0xc01062e4 0xc01012f8 0xc010142b 0xc01030eb 0xc0108215 0xc0108093 0xc0107fde 0xc0100458 0xc01004f4 0xc0100090.
```

```
0xc01062e4: debug_panic (..././lib/kernel/debug.c:34)
0xc01012f8: intr_handler (..../threads/interrupt.c:377)
0xc010142b: intr_entry (threads/intr-stubs.S:38)
0xc01030eb: timer_sleep (...../devices/timer.c:101)
0xc0108215: test_sleep (..../threads/alarm-wait.c:101)
0xc0108093: test_alarm_single (....s/threads/alarm-wait.c:19)
0xc0107fde: run_test (....tests/threads/tests.c:59)
0xc0100458: run_task (..../threads/init.c:270)
0xc01004f4: run_actions (..../threads/init.c:319)
0xc0100090: main (..../threads/init.c:124)
```
• It is possible to access your home directory and run programs remotely using ssh, if you have a ssh client installed. Do the following in a terminal window:

  • `ssh -Y your-liu-id@astmatix.ida.liu.se`

  • The `-Y` flag allows you to forward windows for graphical applications as well.

• You can also mount your home directory as a directory on your local machine, in Linux in a terminal:

  1. Create a new directory, for example astmatix:
     
     `mkdir astmatix`

  2. `sshfs -o idmap=user your-liu-id@astmatix.ida.liu.se:/home/your-liu-id ./astmatix`
General advice

• Work hard

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

• Plan your time together with your lab partner!

• 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 3.
• In 1 week from now
• Lab 2