TDDB68 2015
Lesson 3
Pintos
Assignments (2), 3 & 4

Erik Hansson
erik.hansson@liu.se


Remember

• Pass assignments on time to get bonus points on the exam
• Soft deadlines:
  – Lab 00, 0 – Should be finished now
  – Lab 1 – Finished now
  – Lab 2 – February 12. Tomorrow
  – Lab 3 – March 3.
  – Lab 4 – March 11.
• FINAL DEADLINE (bonus): March 17. (webreg)
  – Finish before the last lab session.
  – Correction after this session is up to your assistants goodwill.

Lab 2: Recall

• Concerns
  – Threads
  – 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!
}
```

Plan your time

• Plan your time with your labpartner
• Some weeks have more scheduled lab time than others.
  • Example:
    – Week 7: 2 hours scheduled lab
    – Week 8: 6 hours scheduled lab
    – Week 9: 6 hours scheduled lab
  – Work outside scheduled time...
  • 36 hours scheduled labs < 3 hp (ECTS)
Lab 2: Recall 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

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 ...

Lab 2: Summary
(or what your are supposed to do)

- Each thread should be able to sleep and be woken up
  - Semaphore
  - Wake-up-time
- Keep track of sleeping threads using shared pintos list
  - You can use a sorted list
  - Protected the list (disable/enable interrupts)
    (Explain why protection is needed)
- When a timer interrupt occurs
  - Check if sleeping threads should be woken up
    - Remove
- Start by looking in /devices/timer.c

Lab 2: Tests

- 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
Lab 3: General Description

- Lab 3: “Execution, termination and synchronization of user programs”
  - A: Execution of several user programs
  - B: Handling program arguments
  - C: Termination of ill-behaving user programs
  - And: Synchronization of shared data structures
- Lab 3 is tricky!
  - Labs 1&2: difficult to understand, easy to implement
  - Lab 3: more implementation

Lab 3: overview

- Part A
  - Multiple user programs
  - New system calls: exec & wait
  - Extended system call from Lab 2: exit

- Part B
  - User program arguments
  - cp foo bar : two arguments

- Part C
  - Make the kernel robust to ill-behaving user programs
  - Example: evil user program:
    create( (char *) 0, 1);

Lab 3: overview

Exec & wait example:

```c
src/examples/shell.c

int main (void) { /* Simplified */
  for (;;) {
    char command[80];
    printf("---");
    read_line (command, sizeof command);
    /* Execute command. */
    if (strcmp (command, "exit") == 0)
      break;
    else {
      pid_t pid = exec (command);
      if (pid != PID_ERROR) {
        printf ("exit code %d\n", wait (pid));
      }
    }
  }
```
Lab 3: Exit

syscall_handler() {
    ...  
save the exit code
    thread_exit()
    process_exit()
    ...  
    process_exit()
    ...  
clean up:
}

printf("%s: exit(%d)\n", thread-name, thread-exit-value)

This is needed for testing purposes (gmake check).

At exit, do:

Part A

Lab 3: Exec

- pid_t exec (const char *cmd_line);
- Runs the executable whose name is given in cmd_line,
  - returns the new process’s program id (pid)
- Must return (pid) -1, if the program cannot load or run for any reason

They’re both int. You can make them a one-to-one mapping, so that the same values in both identify the same process, or you can use a more complex mapping. Up to You!

 pid = process ID (user space)
 tid = thread ID (kernel space)

Lab 3: Exec

Starting the child

Problem: If load fails (pid) -1 should be returned
The loading of the binary is done in the child,
hence the parent does not know at fork-time if loading will work
Need synchronization!
Lab 3: Wait

- `int wait (pid_t pid);`
- Returns the child exit code.
- Child has exited → `wait` returns without delay.
- Child has not exited → parent waits for the child.

Lab 3: Situations with Wait

- Child exits before the parent and:
  - parent calls `wait()` afterwards

![Diagram of child exiting and parent waiting](image1)

`wait()` returns child’s exit value without waiting

Lab 3: Situations with Wait

- Parent exits without calling `wait()` while the child is still running

![Diagram of parent exiting and child continuing](image2)

Exit code will never be used

Lab 3: Situations with Wait

- Child exits before the parent and:
  - parent will exit without calling `wait()`.

![Diagram of child exiting and parent exiting](image3)

You should keep child’s exit value until the parent exits (since the child doesn’t know if the parent calls `wait()` later on)
Lab 3: Situations with Wait

- Parent calls `wait()` before the child exits.

![Diagram showing parent and child processes with wait and exit functions]

**Reference counting**

"poor man's garbage collector"

- Parent – Child needs a new data structure
- Who will free this memory depends on who exits last
- Many ways to implement this. Suggestion: you can use reference counting:
  ```c
  struct child_status{
    int exit_status;
    /* ... */
    int ref_cnt;
  }
  
  /* Initialize it */
  struct child_status * cs = malloc ...;
  cs->ref_cnt = 2; /* both parent and child live */

  /* When parent or child is done with cs: */
  cs->ref_cnt--; /* Needs to be protected by a lock */
  if (cs->ref_cnt == 0) { free(cs); }
  ```

Lab 3: Exec, Exit and Wait

- `exit()`; // the child exits with the exit code 0
- `wait()` // the parent waits for its child...

Lab 3: Part B

- Sometimes we want to pass arguments to programs
  - `insult -s 17`
- In C programs: `argc` and `argv`
  - `argv[0]`: name of the program ("insult")
  - `argv[1]`: first argument ("-s")
  - `argv[2]`: second argument ("-17")
  - `argc`: number of arguments plus one (3)
  - `argv[argc] = NULL`: Required by the C standard

**Your task:** put arguments on the stack when a program is loaded.

Note that a parent can have several children!
Lab 3: Program arguments

- Arguments to a program should be placed within single quotes ('...') on the Pintos command line:
  ```
  pintos --qemu -- run 'insult -s 17'
  ```
- Or in the command line string to a syscall:
  ```
  exec("insult -s 17");
  ```

Lab 3: Part B

- Every time you do a function call a stack frame is pushed on the stack.
  - Done by the caller.
- The very first function of a program is never really called.
- Arguments to the program are put on the stack by the system (your code).
  - First the data,
  - Then the first stack frame

Lab 3: The first stack frame

Lab 2 STEP 3

- into userprog/process.c, find setup_stack()
  - *esp = PHYS_BASE;  \[\text{CHANGE IT BACK!}\]
  - change to *esp = PHYS_BASE - 42;
- Pintos Manual section 3.5.1 Program Startup Details
- Check src/lib/user/entry.c

```c

void start (int argc, char *argv[])
{
    exit (main (argc, argv));
}
```

Example

```c

void start (int argc, char *argv[])
{
    exit (main (argc, argv));
}
```

Lab 3 (B): Program arguments

- **STEP 1**: Break the string in to parts
  - Use `strtok_r()` in `lib/string.(ch)`;
  - Example `exec("insult -s 17 ");`
  - char `s[] = "insult -s 17 "; const char delim[] = " ";`
  - char *token, *save_ptr;
  - for \{token = strtok_r (s, delim, &save_ptr);\}
    - token != NULL;
    - token = strtok_r (NULL, delim, &save_ptr);
      - printf (ʽ"%s" ʽ, token);
  - Output:
    ʽinsult ʽ-s ʽ17ʽ

*Warning: the string s is modified*
Lab 3 (B): Program arguments

- **STEP 2:** Put the string parts on the stack

Necessary details about setting up the stack for this task you can find in section of 3.5.1

Program Startup Details in Pintos documentation.

- Have a look at `load()` and `setup_stack()` in process.c
- There is some inactive debug code there that you can use to print the stack (#define STACK_DEBUG)

<table>
<thead>
<tr>
<th>Address</th>
<th>Name</th>
<th>Data</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xbfffffd</td>
<td>argv[2]</td>
<td>&quot;17\0&quot;</td>
<td>char[3]</td>
</tr>
<tr>
<td>0xbfffffa</td>
<td>argv[1]</td>
<td>&quot;-s\0&quot;</td>
<td>char[3]</td>
</tr>
<tr>
<td>0xbfffff3</td>
<td>argv[0]</td>
<td>&quot;insult\0&quot;</td>
<td>char[7]</td>
</tr>
<tr>
<td>0xbfffff0</td>
<td>word-align</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0xbfffff6c</td>
<td>argv[3]</td>
<td>0</td>
<td>char *</td>
</tr>
<tr>
<td>0xbfffff68</td>
<td>argv[2]</td>
<td>0xbfffffd</td>
<td>char *</td>
</tr>
<tr>
<td>0xbfffff6e</td>
<td>argv[1]</td>
<td>0xbfffffa</td>
<td>char *</td>
</tr>
<tr>
<td>0xbfffff60</td>
<td>argv[0]</td>
<td>0xbfffff3</td>
<td>char *</td>
</tr>
<tr>
<td>0xbfffff68</td>
<td>argv</td>
<td>0xbfffff6c</td>
<td>char*[4]</td>
</tr>
<tr>
<td>0xbfffff64</td>
<td>argc</td>
<td>3</td>
<td>int</td>
</tr>
<tr>
<td>0xbfffff60</td>
<td>return address</td>
<td>unused</td>
<td>void(*)()</td>
</tr>
</tbody>
</table>

Lab 3 : Testing

- When Part B is finished you can run
  - `gmake check`

- The following tests Lab 1:
  - `halt`, `wait`, `create-null`, `create-arg`, `create-arg1`, `open-arg1`, `open-arg2`, `open-arg3`, `open-arg4`, `write-arg1`, `write-arg2`, `write-arg3`, `write-arg4`, `read-arg1`, `read-arg2`, `read-arg3`, `read-arg4`

- Most of the `exec-`* and `wait-`* tests (and lab1) should pass when you have finished Part A&B

- Run a single test (from `userprogs/build`):
  - `gmake tests/userprogs/halt_result`

- The rest when Part C is finished (around 60 tests in total)

Lab 3 : Part C

- **Part C:** Making the kernel robust.
  - Nothing a user-program does should cause the kernel to crash, panic or fail an assertion!
  - Example evil user program passes NULL for filename:
    ```c
    create( (char *) 0, 1);
    ```
  - Or worse, writing to kernel space:
    ```c
    read( STDIN_FILENO, Oxc0000000, 666 );
    ```
  - Be paranoid
    - All pointers from user programs into the kernel must be checked.
      - stack pointer, string, buffer
Lab 3: Example: bad stack pointer

- A user program controls its own stackpointer:
  ```
  asm volatile (  
    "movl $0x0, %esp;
    int $0x30" :: );
  ```

- What will happen in kernel?
- syscall_handler will look at the stack to find out which syscall it is:
  ```
  syscall_handler(...) {
    ...
    switch (*esp) // *(0)
  ```

- The kernel will crash if you don't check esp.

### Bad string argument

- User program:
  ```
  create((char*)NULL, 17);
  ```

- syscall_handler will call:
  ```
  filesys_create((char*)arg0, arg1);
  ```

- But, arg0 is NULL
  ```
  - filesys_create will crash the kernel
  ```

- You must:
  - First check stack pointer
  - Then check the pointer that is on the stack
  - Check all pages until you read '0'

### A non-terminated string

- In this program, the beginning of the string is valid

```c
#define PGS 4096 /* page size */
#define PMASK 0xffffffff
static char inbss;

int main (int argc, char ** argv)
{
    char * bss_page = (char*) ((uint)(4*inbss) & PMASK);
    memset (bss_page, 'a', PGS );
    create (bss_page+PGS-5, 1024);
    exit(0);
}
```

- Just checking the pointer is not enough

### Lab 3: Pointer paranoia

#### Memory validation (simple method)

- A valid pointer into a syscall:
  - is below PHYS_BASE in VM
  - is associated with a page of physical memory: use `pagedir_get_page()`

- Kill the user program if there is an error!
  - exit code -1
Lab 3: Pointer paranoia
Memory validation (tricky method)

• Another way to validate memory is mentioned in the manual
  - Check that the pointer is below PHYS_BASE
  - Dereference the pointer (*ptr) and then take care of page faults
• This is how it is done in real operating systems, it is faster
• Suggestion: Only do it this way if you want the extra challenge
• See Pintos Manual section 3.1.5 for more details!

Lab 4: General Description

• Lab 4: The File System
  - Lecture 9
  - Course book chapter 11.
  - Synchronization of read-write operations
    • One writer writes at a time
    • Many readers can read
  • See Readers-Writers Problem
    - Lecture 4.
    - Course book (9th ed.) section 6.7.2
  • Additional system calls to work with files
  • Creating and removing files without destroying the file system

Synchronize access to the directory and the free_map

Lab 4: Files (1)

• filesys/file.[h|c] - operations on files. A file object represents an open file.
• filesys/filesys.[h|c] - operations on the file system.
• filesys/directory.[h|c] - operations on directories.
• filesys/inode.[h|c] - the most important part of the implementation related to the file system.
  - An inode object represents an individual file (e.g. several open files fd1, fd2, fd3 may belong to one inode "student.txt").

Lab 4: Files (2)

• devices/disk.[h|c] - implementation of the low-level access to the disk-drive.
• filesys/free-map.[h|c] - implementation of the map of free disk sectors.
Lab 4: Reading/Writing

- Requirements
  - Several readers should be able to read from a file at the same time.
  - Reading should be forbidden if the file content is being changed by a writer.
  - Only one writer can write to a file at a time.

Lab 4: Additional System Calls

- `void seek (int fd, unsigned position)`
  - Sets the current seek position of `fd`
- `unsigned tell (int fd)`
  - Get the seek position of `fd`.
- `int filesize (int fd)`
  - Returns the file size of the open file `fd`.
- `bool remove (const char *file_name)`
  - Removes the file `file_name`.
  - Open files are not deleted from the file system before they are closed.

Lab 4: Create and Remove

- Creating and removing of files must not lead to destructive consequences to the file system
- Create and remove are writing operations on the directories (filesys/directory[.h|.c])
- To make your life easier, reading & writing operations on the directory can be synchronized by a single lock.

Lab 4: Final Tests

- For testing your readers-writers algorithm, we provide the following user programs: `pfs.c`, `pfs_reader.c`, `pfs_writer.c`
- Run `pfs` several (100??) times, check result each time...
- These programs emulate several readers and writers accessing the same file.
  - 2 Writers: repeatedly fill `file1` with a letter (new letter each time).
  - 3 Readers: repeatedly read `file1` and check that all letters are the same.
  - If the readers see that not all the letters are the same they are seeing a half finished write; fail
Remember: Debugging

- Using DDD
- `printf`
  - Do not forget to remove them when running `gmake check`
- backtrace

This is the last slide

- Contact me if you have ideas on how to improve the course?
  - labs
  - lessons
- Please, finish the labs **before** the last lab session.