

# TDDDB68/TDDDE47

## Lesson 1

Felipe Boeira

Based on previous slides from the course

# Contents

- Overview of the labs
- General information about Pintos
- System calls
- Lab 0/Lab 1 details
- Debugging

# Administration

Webreg:

- If you still could not register, send me an email!

# General information about the labs

- The labs are based on Pintos: an educational OS (developed at Stanford University)
- Pintos is written in C and well documented
- The labs are about adding functionality to Pintos
- Pintos is around 7 500 lines of code (LOC)

# General information about the labs

- The complicated parts of the labs are understanding how pintos is structured and what you should do: you do not need to write a lot of code
- In other words, try to read a lot!
- A proper understanding of C will save you a lot of time debugging!
- You **must** work on **non-scheduled** time as well! These labs tend to be time-consuming

# General information about the labs

- If you pass the labs on time(**2020-03-13**), then you earn **4 bonus points** on the exam
- This offer is only available to new students
- The final deadline is **2020-03-30** in Webreg
- The procedure of handing in the labs
  - [gitlab.liu.se](https://gitlab.liu.se)

# Lab 0

- Setting up Pintos
- Linked lists
- Information about how to debug Pintos

# Lab 1

- The first real lab
- Single user process
- Implement a number of system calls:
  - Reading from and writing to the console
  - Creating, reading from and writing to files
  - Exit a process and halt the machine
- Tends to take some time since you have to familiarise yourself with Pintos



# Lab 2

- Multiple user processes
- Implement another system call: Sleep
- Sleep delays execution of the calling process by the given number of milliseconds
- Synchronisation is now necessary
- This lab tends to take the least amount of time

# Lab 3

- Multiple user processes
- Implement the exec system call
  - Exec: Let processes start execute programs in a child process
- Create parent-child relationship
- This lab together with the next two tend to take a lot of time to finish!

# Lab 4

- Programs cannot have arguments yet - fix it!
- Setup the user space program stack with arguments according to the x86 convention
- This lab requires careful understanding of memory layout and pointer arithmetic

# Lab 5



- Multiple user processes
- Implement the system call Wait
  - Wait: Let processes wait for their children to finish executing
- Use or extend the parent-child relationship you have already created

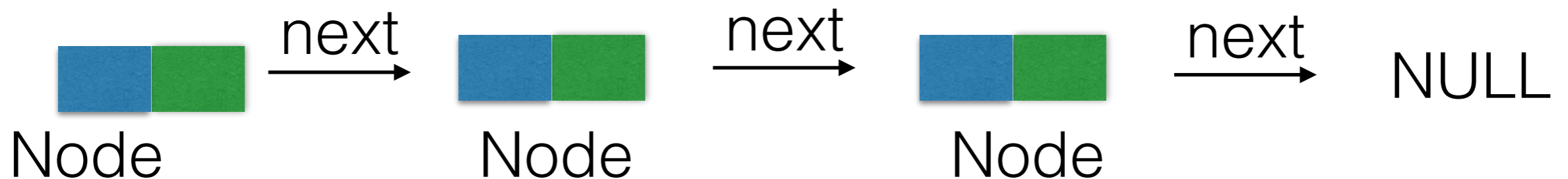
# Lab 6

- If several processes write to the same file, they will overwrite each other's content arbitrarily
- Make sure that no order of system call, or internal calls, leads to an invalid state (open, close, write, read, and so on)
- Synchronise the file system! (readers/writers algorithm, and more)
- This lab tends to take about as much time as lab 1

# A closer look at Lab 0

- Linked list is a simple data structure to dynamically store data
- Singly linked list example:

```
struct Node {  
    int data;   
    struct Node* next;   
};
```



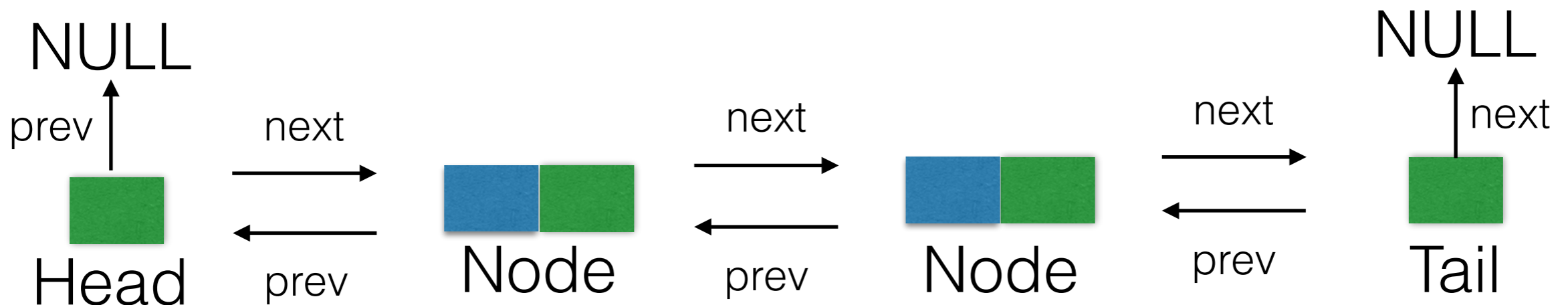
# A closer look at Lab 0

- Doubly linked lists are similar, they also point to the previous element
- Pintos implementation example:

```
struct list_elem {
    struct list_elem *prev; /* Previous list element. */
    struct list_elem *next; /* Next list element. */ };

struct list {
    struct list_elem head; /* List head. */
    struct list_elem tail; /* List tail. */};
```

```
struct Node {
    int data;           
    struct list_elem elem;           
};
```



# A closer look at Lab 1

- Only one user process at a time - no concurrency!
- Suppose a user process wants to open a file, then it:
  1. Calls the function `int open(const char *file);`
  2. The function `open` puts the arguments on the stack, together with the syscall no.;
  3. Produces an interrupt to switch from user mode to supervisor mode;
  4. The interrupt handler then looks at the interrupt no. and delegates it to the appropriate subhandler, in this case, the syscall handler;

Already implemented!

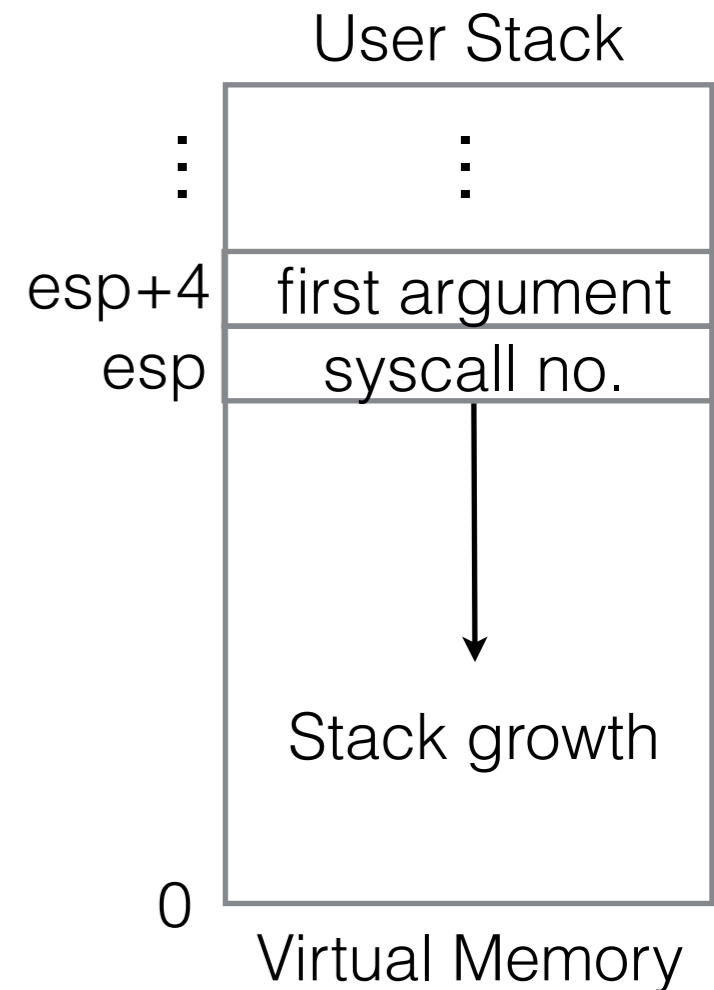


# A closer look at Lab 1

lib/user/syscall.[h|c] - The syscall wrapper

```
/* Invokes syscall NUMBER, passing argument ARG0, and returns the
   return value as an `int'. */
#define syscall1(NUMBER, ARG0) \
    ({ \
        int retval; \
        asm volatile \
            ("pushl %[arg0]; pushl %[number]; int $0x30; addl $8, %%esp" \
             : "=a" (retval) \
             : [number] "i" (NUMBER), \
               [arg0] "g" (ARG0) \
             : "memory"); \
        retval; \
    })
```

```
int \
open (const char *file) \
{ \
    return syscall1 (SYS_OPEN, file); \
}
```



# A closer look at Lab 1

- The syscall handler then (in supervisor mode):
  1. Reads the syscall no. to decide the type of syscall (write, read, open, close, and so on).
  2. Based on the type, the handler reads the correct number of arguments from the stack, and performs the syscall.
- For example, the handler does not get the arguments to this syscall directly: **bool** create(**const char** \*file, **unsigned** size)  
To get them you have to read them from the stack: f->esp
- Note that the actual string is NOT on the stack. The stack only has a pointer to the first character of the string.
- To return the result of the syscall, set this register: f->eax

This is the assignment!

# A closer look at Lab 1

Files to study:

- `lib/user/syscall.[h|c]` - The syscall wrapper
- `userprog/syscall.[h|c]` - Implement syscalls here!
- `threads/thread.[h|c]` - Implement something here!
- `threads/interrupt.[h|c]` - Important structures
- `lib/syscall-nr.h` - Syscall numbers
- `filesystem/filesys.[h|c]` - Pintos file system

# A closer look at Lab 1

- Currently, the syscall handler kills every calling process
- The handler must do the things that we discussed earlier
- `f->esp` is the stack of the calling process
- The syscall number is at the top, then the arguments
- Every syscall has its own syscall number: use it to decide the number of arguments

# A closer look at Lab 1

## File descriptors (FD)

- A FD is a non-negative integer that represents abstract input/output resources
- Input/output resources are, for example, files, consoles, network sockets, and so on
- The user processes only knows about FDs, and the OS knows what concrete resource it represents
- In the labs, FD 0 and 1 are reserved for the console (stdin/stdout)

# A closer look at Lab 1

What to think about when implementing:

- **create** - Create a file. Return true if a file was created, false otherwise. Hint: Use already implemented functions.
- **open** - Open a file. Return a FD to the user process. How do we decide on a FD and how do we map it to an opened file? Every process has its *own* collection of opened files. FD values 0 and 1 are reserved for the console. Hint: Modify the struct thread to track file descriptors.
- **close** - Close the file associated with the given FD. Disassociate the FD with the file. Hint: Use already implemented functions.
- **exit** - Kill the process. Deallocate all of its resources (eg. files). We revisit this syscall in Lab 3. Hint: Free resources in `thread_exit`.

# A closer look at Lab 1

What to think about when implementing:

- **read** - Read the file associated with the given FD. The user process gives a buffer (a piece of memory) in which the read bytes are written to. Return the number of read bytes. Hint: Use already implemented functions. Use **input\_getc** to read from the console.
- **write** - Write to the file associated with the given FD. The user process gives a buffer with the content that should be written. Return the number of written bytes. Hint: Use already implemented functions. Use **putbuf** to write to the console (check `lib/kernel/stdio.h` and `lib/kernel/console.c`).
- **halt** - Shutdown the machine (halts the processor). Hint: Use already implemented functions.

# A closer look at Lab 1

What to think about when implementing all of them:

- Every user process should be able to have at least 128 files opened at the same time
- It is *dangerous* to assume that the arguments are valid! We will revisit this topic in future labs
- Special cases: What happens if the arguments of the syscall are invalid? Such as NULL pointers, invalid buffer size, FDs with no associated file, and if the process has opened too many files



# A closer look at Lab 1

## Frequently Asked Questions:

- Use the function `thread_current()` to get the thread structure of the calling process
- The function `filesys_open(char *)` opens a file, and the function `file_close(file *)` closes it
- The function `init_thread(...)` initialises every thread, whilst the function `thread_init(...)` initialises the thread module (once, when Pintos starts up). If you need to do some initialisation for every thread, modify the former function (at the end of it)

# A closer look at Lab 1

- Run `lab1test2` to test your code. It will:
  - Create files
  - Open files
  - Read and write from the console
  - Try to use bad FDs
- Gotchas:
  - Remove all the files on the virtual hard drive before running it again:  
`pintos --qemu -- rm test0 rm test1 rm test2`
- Passing `lab1test` does NOT mean that you have finished the lab! You must ensure that there are no special cases
- Your implementation will be tested more thoroughly in Lab 3

# System call overview

- 13 system calls to be implemented
- Linux: Around 320 system calls
- Windows: More than 1000 system calls

# Debugging

- Read Appendix E: Debugging tools in the documentation
- `ASSERT(p != NULL)`
- If you get "Kernel Panic", then try the `backtrace` tool
- `free` sets bytes to `0xcc`: If you see bytes with this value, then something likely freed the memory
- Use versioning and commit frequently. It is common to break Pintos in very obscure ways, and it is often easier to go back to a working version and redo the changes

# Debugging

Backtrace example:

- If you were to get this:  
Call stack: 0xc0106eff 0xc01102fb 0xc010dc22 0xc010cf67  
0xc0102319 0xc010325a 0x804812c 0x8048a96 0x8048ac8
- Then type this:  
backtrace kernel.o 0xc0106eff 0xc01102fb 0xc010dc22 0xc010cf67  
0xc0102319 0xc010325a 0x804812c 0x8048a96 0x8048ac8
- To get this:  
0xc0106eff: debug\_panic (lib/debug.c:86)  
0xc01102fb: file\_seek (filesys/file.c:405)  
0xc010dc22: seek (userprog/syscall.c:744)  
0xc010cf67: syscall\_handler (userprog/syscall.c:444)  
0xc0102319: intr\_handler (threads/interrupt.c:334)  
0xc010325a: intr\_entry (threads/intr-stubs.S:38)  
...

Questions?