Concurrent programming and Operating Systems Lesson 1

Dag Jönsson

LINKÖPING UNIVERSITY

General lab information

General lab information

- Work in groups of 2
  - Discussion groups
- Exemptions to working in group?
- If you aren’t signed up in WebReg, email me
- Demo and hand in

General lab information

- Assessments
  - Correct memory management
  - No undefined behaviour
  - No synchronization errors
- ”Deadlines”
  - Soft deadlines (recommended pace) in TimeEdit
- Deadline
  - 2025-03-28

General lab tips

- Read before you code
  - Including source code
- Write/draw before you code
  - How does the data need to flow?
  - What needs to be done in what order?
- Keep solutions simple
- Work outside of scheduled lab hours
- Try and answer the prep. questions in labs

Lab overview

Pintos

- Educational OS developed at Stanford University
- C and Assembly, well documented
- 7 500 LOC
- Exists:
  - Device drivers, filesystem
  - Userspace with small standard library
  - Simple scheduler, interrupt support
- Labs: Add functionality

Lab environment

- Linux is required, prefer LiU machines
- VM is available, very out of date
- Own machine?
  - Linux? At minimum gcc, make, qemu needed
  - WSL?
  - Mac OS X?
- Editor?
  - Emacs, vim, VSCode

Lab 0

- Single linked list
  - Doesn’t have to be perfect
- Trying debugging tool
  - Not required to use GDB in the labs, but good option
- Running Pintos and debug with GDB

Lab 1 – Command line

- Single user process
- Setting up stack for main(int argc, char** argv)
  - X86 convention
- Remember popping and pushing from/to the stack?
- Solid knowledge about memory layout and pointer arithmetic
- About 30-50 LOC

Lab 2 – Basic System calls

- Single user process
- Handle system calls in kernelspace
- Need to familiarize yourself with the file structure
- About 160-200 LOC

Lab 3 – Basic Synchronization

- Multiple system threads
- Synchronisation is now required
- About 40-60 LOC

Lab 4 – exec

- Multiple user processes
- One more systemcall: exec
  - Allow a program to run another program
- About 50-100 LOC

Lab 5 – wait and exit

- Multiple user processes
- One more systemcall! wait
  - Let a program wait on a child process
- Validate arguments from userspace
  - Make sure the kernel doesn’t crash because of user
- About 50-70 LOC

Lab 6 – File system

- Multiple processes
- Synchronize the filesystem
  - Allow several read and write operations to interleave
  - Filesystem needs to always be in a valid state
- Usually takes about the same time as Lab 2
- About 40-50 LOC

Total LOC

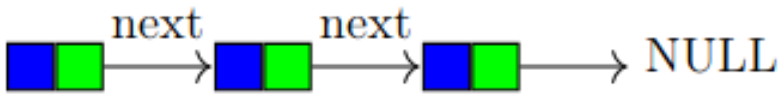
- Lab 1: 30-50 LOC
- Lab 2: 160-200 LOC
- Lab 3: 40-60 LOC
- Lab 4: 50-100 LOC
- Lab 5: 50-70 LOC
- Lab 6: 40-50 LOC
- Total: 370 – 530 LOC
  - Not that much!

Lab 0

Lab 0: Single linked list

- Simple linked list to store dynamically allocated data

```
1 struct Node {
2   int data;
3   struct Node* next;
4 }
```



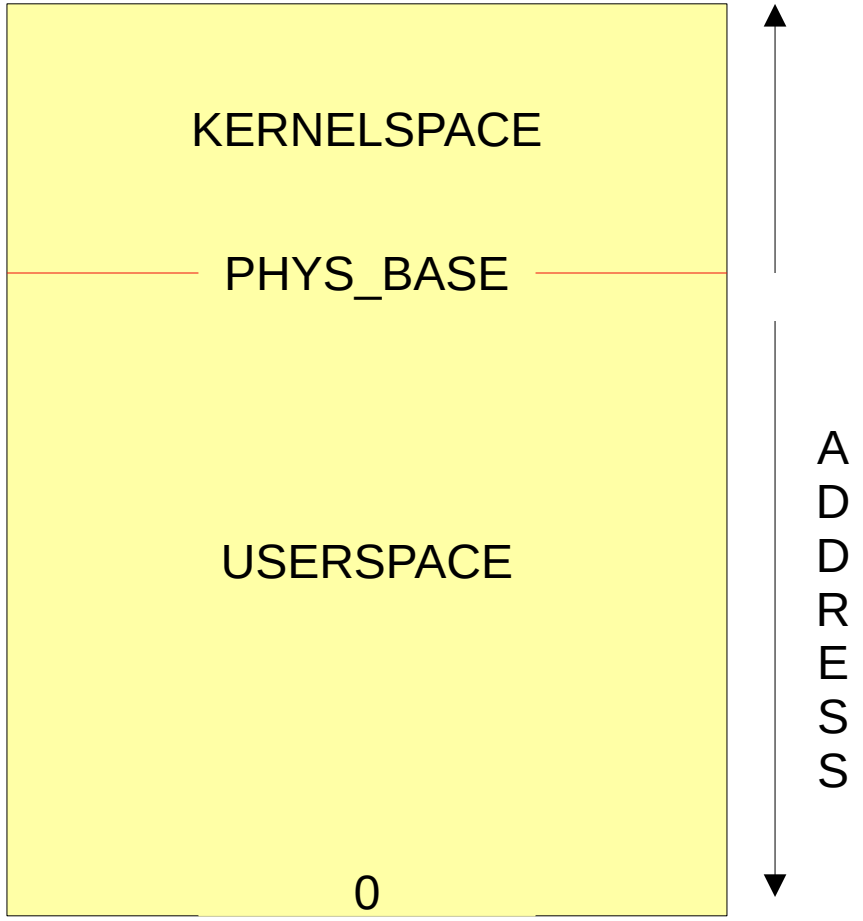
Lab 0: GDB

- Small problems
  - Practice debugging
  - Try a (potentially) new tool
- Not exhaustive, only introductionary

Lab 1

Memory layout

- Split between kernel and userspace
- Addresspace:
  - Userspace: `]PHYS_BASE, 0]`, grows → `0`
  - Kernelspace: `[PHYS_BASE, MAX_MEM]`



The stack

- Every program has its own stack
- Consider the command line **`ls -la .`**
  - Where are is it stored? By who? Why? How is it used?
- The OS is responsible for setting up the stack
- Rules that need to be followed (x86 convention)

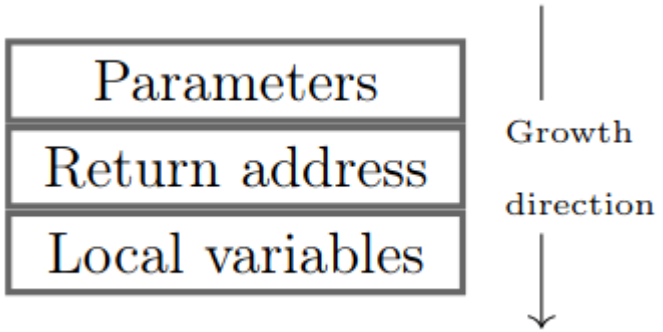
The stack

- Consider the following program:
- Running the program as:
  `./a.out Hello`
- What will the first line print?
- The second? And the last?

```
int main(int argc, char** argv) {
    printf("%s", argv[0]);
    printf("%s", argv[1]);
    printf("%s", argv[argc]);
}
```

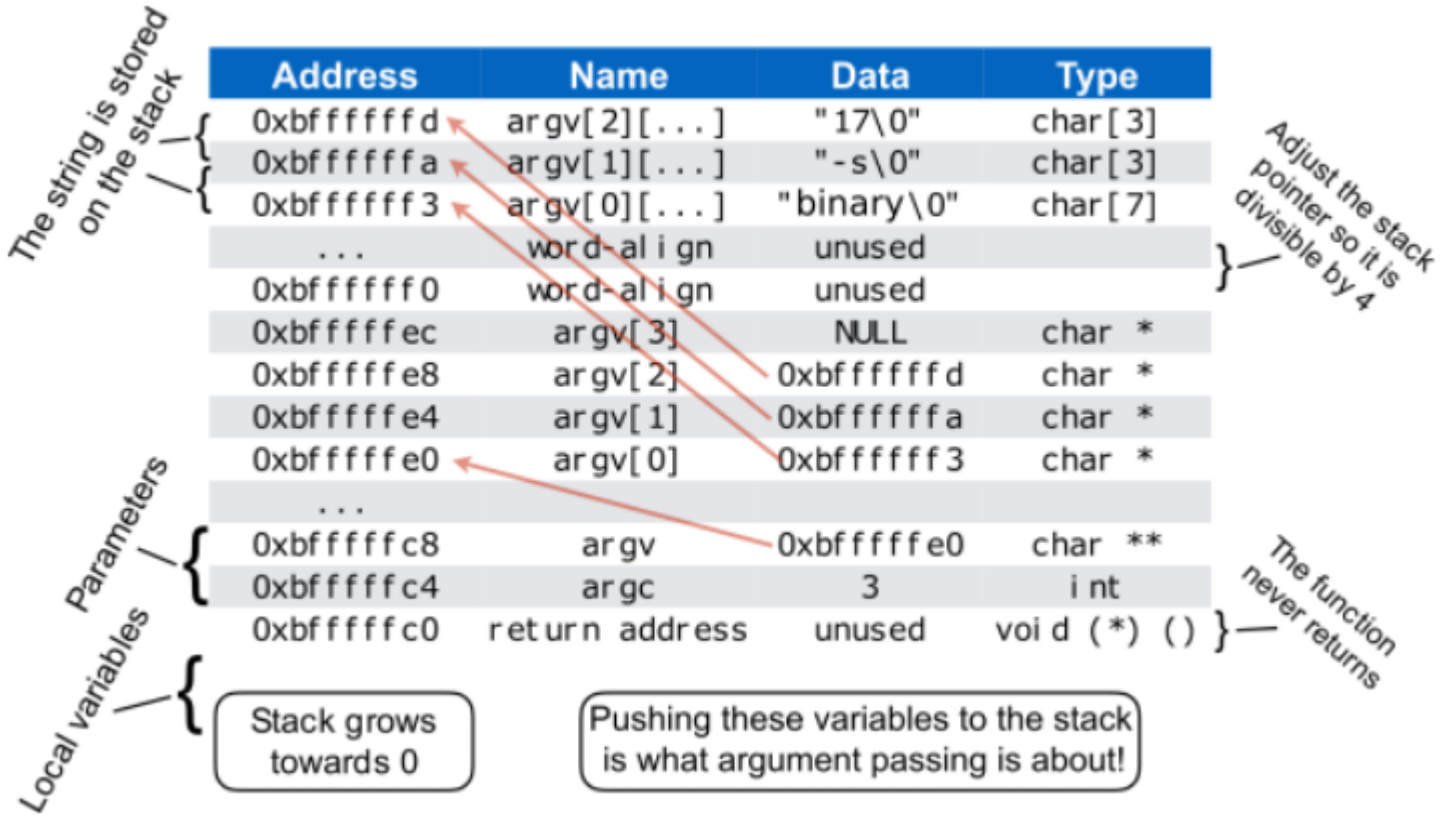
The stack

- Every time a function is called, a stack frame is created:



- But we don’t call the main function?
- The argument values and return adress is pushed to the stack, by the OS

The stack



Address	Name	Data	Type
0xbfffffd	argv[2][...]	"17\0"	char[3]
0xbfffffa	argv[1][...]	"-s\0"	char[3]
0xbffffff3	argv[0][...]	"binary\0"	char[7]
...	word-align	unused	
0xbffffff0	word-align	unused	
0xbffffec	argv[3]	NULL	char *
0xbffffe8	argv[2]	0xbfffffd	char *
0xbffffe4	argv[1]	0xbfffffa	char *
0xbffffe0	argv[0]	0xbffffff3	char *
...			
0xbffffc8	argv	0xbffffe0	char **
0xbffffc4	argc	3	int
0xbffffc0	return address	unused	void (*) ()

Pintos boot

- Defined in **`threads/init.c`**
- Initializes submodules (threads, memory, file system, etc)
- Executes a given userprogram with **`process_execute()`**, defined in **`userprog/process.c`**

`process_execute()`, `start_process()`

- **`T0:`** Tries to create a thread for the new process
  - If success: Hands over execution to the new thread, have it start in **`start_process()`**
- **`T1:`** will try and allocate resources, load binary and initialize the stack
  - If successful: Hand over execution to the userprogram, starting in **`main()`**
- Difference between thread and process in Pintos?

`thread` struct

- Declared in **`threads/thread.h`**
- Well documented in the source files
- Keep track of kernel resources allocated for a thread/process
- Used throughout the lab series

Lab 1: Command line

- Currently, Pintos does not setup the stack correctly
- Your task:
  - Write code to setup the stack correctly
  - Make sure the correct filename is loaded (and set the thread name)
- Initial steps: Familarize yourself with
  `userprog/process.c : start_process()`
- See Pintos documentation 3.5 80x86 Calling Convention for another explanation

Lab 1: String tokenization

- `start_process()` will get a C-String, e.g. "`binary -s 17\0`", you need to process this before pushing it to the stack.
  - Helpful functions in `lib/string.h`

```
char* strtok_r(char*, const char*, char**)
void* memcpy(void*, const void*, size_t)
```

- These functions are documented in their implementation: `lib/string.c`
  - You might find other useful functions there as well.

Lab 1: String tokenization

- Where to put our code?
  - **`userprog/process.c : start_process()`**
  - **Hint: `start_process()`** creates an interrupt frame which holds a pointer to the stack. Make sure the stack is initialized before putting anything on it.
- Remember, double pointers need to be dereferenced twice to get at the actual value. Deref once to change the pointer to the value.

Lab 2

Interrupts and systemcalls

- Two groups of interrupts in Pintos
  - External and internal
- Systemcalls - internal interrupt
- Interrupts -> interrupt frame

Interrupt frame

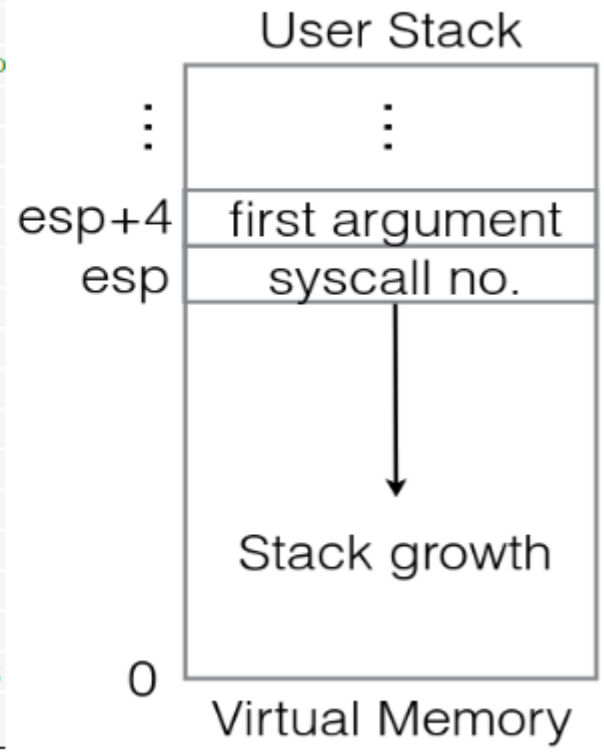
- Declared in **`threads/interrupt.h`**
- Snapshot of the CPU registers at interrupt
  - Used to restore the CPU registers once the interrupt is served
- Registers of interest to you:
  - **`esp`** – The stack pointer
  - **`eax`** – Return register

Lab 2: syscalls

- You only need to think about 1 process
- Suppose a user process want to open a file, it has to:
  - Call the function int open(const char* file)
  - It will push the arguments to the stack, and add in the correct syscall number
  - Cause an interal interrupt and switch execution over to kernel mode, specifially interrupt handler
  - It will look at the interrupt number, and delegate the servicing of the interrupt to the syscall handler. **`userprog/syscall.c : syscall_handler()`**
- All of the above is already implemented and works as expected!

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

```
1
2   /* Invokes syscall NUMBER, passing no
3   arguments, and returns
4   the return value as an `int'. */
5   #define syscall1(NUMBER)
6   ({
7     int retval;
8     asm volatile("pushl %[number];
9     int $0x30; addl $4, %%esp"
10    : "=a"(retval)
11    : [number] "i"(NUMBER)
12    : "memory");
13    retval;
14    })
15
16    int open (const char *file) {
17      return syscall1 (SYS_OPEN, file);
18    }
```



Lab 2: syscalls

- Your task is to implement the **`syscall_handler()`** (kernel mode)
  - Read the syscall number from the stack (defined in **`lib/syscall-nr.h`**)
  - Decide on how many arguments to extract from the stack, based on the syscall number
  - Service the syscall, potentially returning a value to the userprog (usermode)
- The handler has to extract the values manually from the stack (**`f->esp`**)
  - Why?
- Note that some of the arguments are pointers
- Return value? Store it in the **`f->eax`** register

Lab 2: File descriptors (FD)

- A process unique non-negative integer that represents abstract input/output resources
- For example: files, consoles, network sockets, etc
- Userprograms only knows about FDs
- Special FDs:
  - **`0 – stdin`**
  - **`1 - stdout`**

Lab 2: File handling

- You need to figure out a strategy for FDs
- Remember:
  - They need to be unique for the given process
  - The FD represent a resource allocated to the process
  - Where to store the actual resource? How?
  - If a file is opened several times, how many FDs?

Lab 2: Files

- You should read through the following files:
  - `lib/user/syscall.[h|c]` – The syscall wrapper
  - `lib/syscall-nr.h` – Syscall numbers
  - `threads/interrupt.h` – Important structure!
  - `filesys/filesys.[h|c]` – High level functions for the file system
- Modify:
  - `userprog/syscall.[h|c]` – Implement syscall handler
  - `userprog/process.[h|c]` – Clean up any resources on exit here
  - `threads/thread.[h|c]` – Any resources allocated for the thread goes here

Lab 2: Final tips

- Currently, the **`syscall_handler()`** kills any calling program
  - Remove this to avoid confusion later
- **`printf()`** does not work in userprograms until the **`write`** syscall is implemented.
  - **`printf()`** does work if you are in kernel mode though!
- **`f→esp`** is pointing to the stack of the calling process.
  - Specifically the top of the stack
- Traversing the stack means you are increasing the memory address.

Lab 2: Final tips

- Most of the actual functionally is already there
  - You just need to call it correctly
- Any given process should be able to open 128 files
- Verify any values from userspace
  - Is the given FD associated with a resource?
  - Is the given buffer size reasonable? (>= 0)
- Don’t validate pointers (yet!)
  - You can assume that the pointers themselves are correct for now

FAQ and general tips (again!)

- Use **`thread_current()`** to get the current thread struct for the calling process/thread.
- The function **`filesys_open()`** opens a file, while **`file_close()`** closes a file.
- **`init_thread()`** is used to initialize a singular thread. **`thread_init()`** initializes the thread module (once, during boot). If you need to initialize some values in the thread struct, do it in **`init_thread()`**.
- Structure your code for readability!
  - You will very likely revist your solution in later labs.
    - Think about your future selves!
  - Add more functions if you feel it helps
    - But consider if they need to be global or local only

Debugging

- Read Appendix E. Debugging tools in the Pintos documentation
- If you get ”Kernel Panic”, you can try and use the `backtrace` tool
- Free sets the bytes to `0×cc`: If you see these values the memory accessed is very likely freed
- Commit often!
  - Sometimes it easier to revert to a working version instead of solving the issue.

- If you get something like this:

```
Call stack: 0xc0106eff 0xc01102fb 0xc010dc22 0xc010cf67
0xc0102319 0xc010325a 0x804812c 0x8048a96 0x8048ac8
```

- Then try the backtrace tool:

```
backtrace kernel.o 0xc0106eff 0xc01102fb 0xc010dc22 0xc010cf67
0xc0102319 0xc010325a 0x804812c 0x8048a96 0x8048ac8
```

- You should get:

```
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)
```

Dag Jönsson
dag.jonsson@liu.se

LINKÖPING UNIVERSITY