Concurrent programming and Operating Systems

Lesson 1

Dag Jönsson

LINKÖPING UNIVERSITY

LINKÖPING UNIVERSITY

WebReg

Deadline 2024-01-19

Use the Teams room if you haven't found someone to work with
Send me an email if you are unable to register!
dag.jonsson@liu.se

Bonus

- If you have passed all labs by **2024-03-08** you get 3 bonus points on the exam
- Only available for students taking the course for the first time
- Final hard deadline is **2023-03-26**! Need to have all pass in Webreg.
- Hand in through LiUs Gitlab

"Deadlines"

- Individual labs do not have deadlines
- "Soft deadlines", recommended pace

Demo and hand in

Oral examaniation after each assignment

After demonstration: make any corrections, commit, branch, push, email

```
git checkout -b labX
git push --set-upstream origin labX
git checkout main # continue working on main
```

Note: origin might be something else for you, you can just try a `git push` on the new branch to get some help

Pintos

- Labs are based on Pintos; an educational OS developed at Standford University
- Written in C and is well documented
- Around 7 500 lines of code (LOC)
- The labs are about adding functionality to Pintos

Pintos

- Complication comes from reading and understanding code
- Fairly small amount of actual code will be written
- Good understanding of C will save a lot of time when debugging
- **Need** to work on the labs on **non-scheduled** time as well
- There are preparatory questions in most labs, do take the time to actually answer these

Pintos

- While working on the labs, prefer to use the Linux machines on LiU
- A VM is available, (user and password: pintos) (slightly out of date)
- Possible to to make it work on your own machine if you use Linux (or WSL), but you need to figure out the details
- Prefer to use a simple editor, i.e. emacs, vim or VS Code

LINKÖPING UNIVERSITY

Lab 0

- Getting to know C and pointers
- Single linked list
- Setting up Pintos and git
- How to debug with GDB, both outside and inside Pintos

Lab 1

- Implement argument passing to programs
- Setup of the stack for a userspace program according to the x86 convention
- Requires solid understanding of memory layout and pointer arithmetic
- Solutions are usually around 30-50 LOC

Lab 2

- Single user process
- First iteration of a system call handler
- 12 system calls to be implemented
- Afterwards, your OS should be able to:
  - Read from and write to the console
  - Create, remove, read from, and write to files
  - Exit a process and halt the machine
  - Sleep a process for given amount of time
- Usually takes a bit of time since you need to familiarize yourself with the file structure
- Solutions are usually around 160-200 LOC

Lab 3

- Multiple system threads
- Synchronisation is now required
- This lab usually takes the least amount of time
- Solutions are usually around 40-60 LOC

Lab 4

- Multiple user processes
- Another system call to implement: exec
- exec allows a process start the execution of child processes
- Solutions are usually around 50-100 LOC

Lab 5

- Multiple user processes
- Implement yet another system call: wait
- wait: Let a process wait for one of the children to finish executing
- Create parent-child relationship
- Validation of arguments given by the user
- Solutions are usually around 50-70 LOC

Lab 6

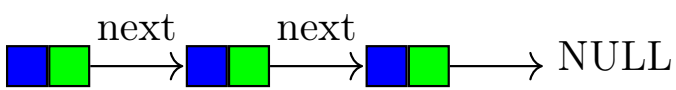
- Multiple processes
- Synchronisation of the filesystem
- Make sure that no order of system calls, or internal calls, leads to an invalid state (open, close, write, read, and so on)
- Tends to take about as much time as lab 2
- Solutions are usually around 40-50 LOC

LINKÖPING UNIVERSITY

Lab 0: Introductionary

Linked list is a simple data structure to dynamically store data

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



GDB

- Small problems to practice the basics of debugging
- Not exhaustive, only introductionary

Memory layout

- Memory is split between kernel and userspace
- Userspace is from address 0 up to `PHYS_BASE` (`0xc0000000`)
- Kernel space occupies the rest, 1 GB of memory reserved
- Userspace programs may not write or read from kernel space

The stack

- Every program has it's own stack (slice of the total user space)
- Arguments when calling programs are pushed on the stack by the OS
- Certain rules to follow (calling convention)

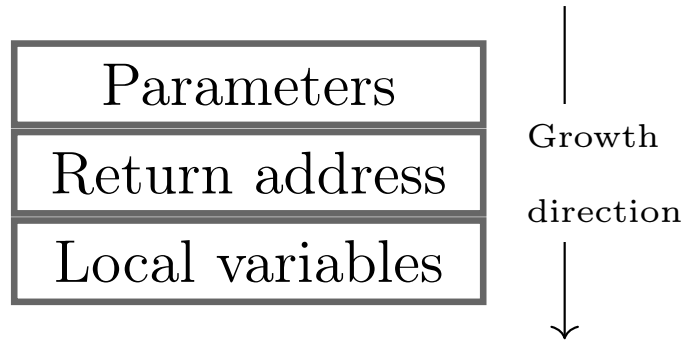
The stack

Suppose we run `binary -s 17` The parameters of the main function of the C progam are `int argc` and `char **argv`. So in this example:

```
1  argc = 3
2  argv[0] = "binary\0"
3  argv[1] = "-s\0"
4  argv[2] = "17\0"
5  argv[3] = NULL
```

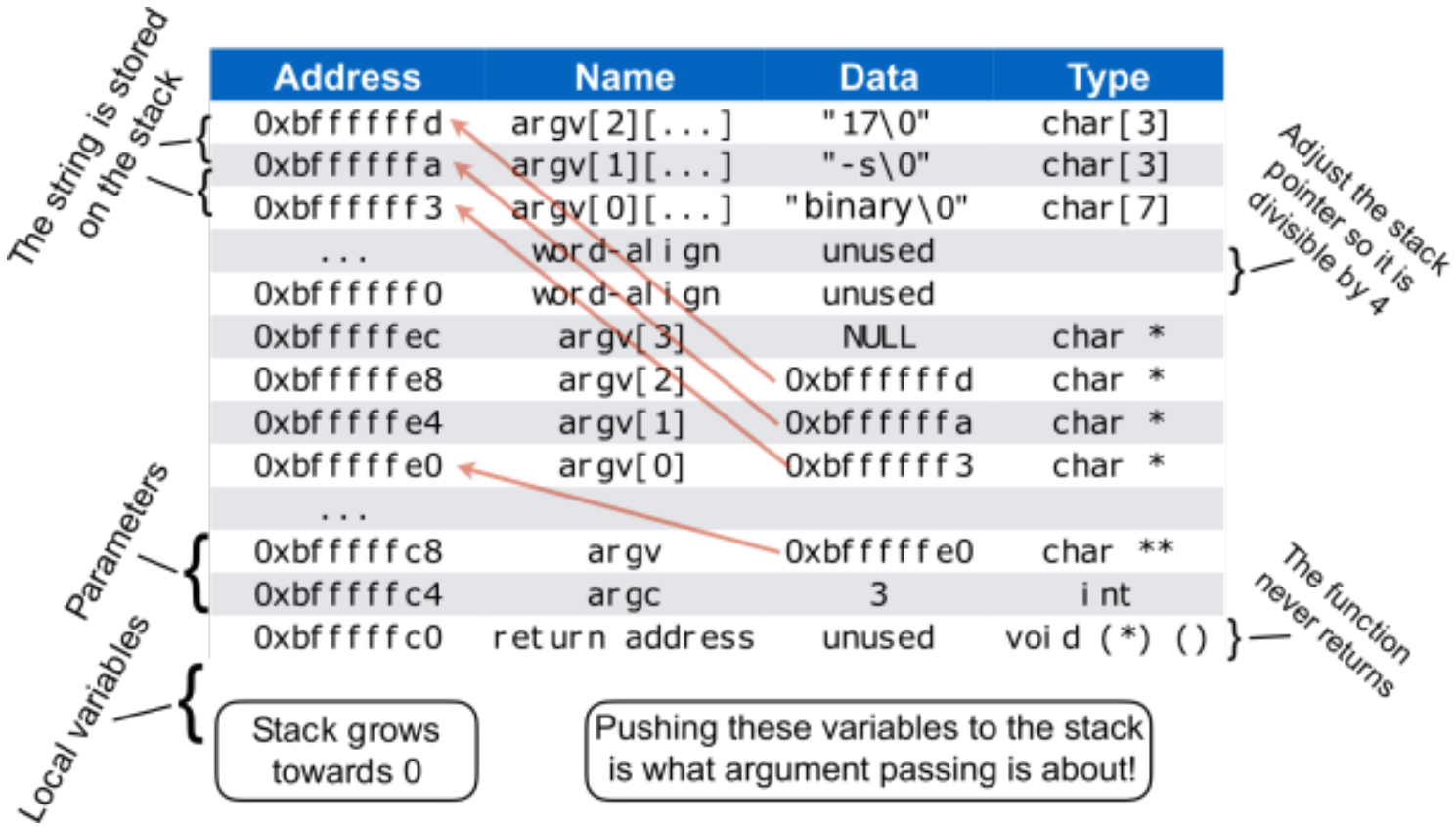
The stack

- Every time you do a function call, a stack frame is created:



- The `main` function is never really called, but the layout is the same
- The parameters and the return address of the stack frame are pushed onto the stack by the operating system

The stack


The string is stored on the stack
Address
Name
Data
Type
0xbfffffffd
argv[2][...]
"17\0"
char[3]
0xbfffffffa
argv[1][...]
"-s\0"
char[3]
0xbfffffff3
argv[0][...]
"binary\0"
char[7]
...
word-align
unused
0xbfffffff0
word-align
unused
Adjust the stack pointer so it is divisible by 4
0xbffffffec
argv[3]
NULL
char *
0xbffffffe8
argv[2]
0xbfffffffd
char *
0xbffffffe4
argv[1]
0xbfffffffa
char *
0xbffffffe0
argv[0]
0xbfffffff3
char *
...
Parameters
0xbffffffc8
argv
0xbffffffe0
char **
0xbffffffc4
argc
3
int
Local variables
0xbffffffc0
return address
unused
void (*) ()
The function never returns
Stack grows towards 0
Pushing these variables to the stack is what argument passing is about!

Interrupts

- Two kinds of interrupts, **software** and hardware
- Software interrupts: triggered by software, often to get the kernel to do something (system calls)
- Sometimes called "internal interrupt"
- Interrupt frame: A snapshot of the process state at the time of the interrupt

Interupt frame

- Declared in `threads/interrupt.h`
- Contains the values that were in the CPU registers at time of interrupt
- Registers of interest for you:
  - `esp` - The stack pointer
  - `eax` - Return register

Pintos boot

- Boot process is defined in `threads/init.c`
- Initializes submodules (threads, memory, file system etc.)
- Executes any given user program with `process_execute()`, defined in `userprog/process.c`

process_execute(), start_process()

- Creates a thread for the new process
- Hands over execution to new thread with `start_process()`
- `start_process()` allocates memory in user space, load the binary, create an empty stack, and if successful, hand the execution over to userspace
- Difference between thread and process in Pintos?

`thread` struct

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

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 resoure it represents
- In Pintos, FD 0 and 1 are reserverd for `stdin` and `stdout`

LINKÖPING UNIVERSITY

Lab 1: Command line

- Currently, Pintos does not support arguments to *programs*
- Implement the necessary code to make sure the arguments are passed on correcly
- First steps: Familarize yourself with how `start_process()` works

Lab 1: String tokenization

- You get a string, such as "binary -s 17\0", and you need to split it up in smaller parts.
  Helpful functions found in `lib/string.[c|h]`
  `char* strtok_r(char *, const char *, char **)`
  `void* memcpy(void*, const void*, size_t)`
- Read the comment above the definition for an example of usage.
- After `strtok_r` is run on the string, it will look like: `"binary\0-s\017\0"`
- You need to save a pointer to every word!
- Read the Pintos documentation 3.5 for another description

Lab 1: Command line

"Where should our code go?" - What function within `process.c` has access to both the stack pointer and the `*cmd_line`? And when is the stack actually available?

Hint: `start_process()` creates an interrupt frame which holds a pointer to the newly created stack

Remember, double pointer needs to be dereferenced twice to read/write the value pointed at, dereference once to access the pointer to the value.

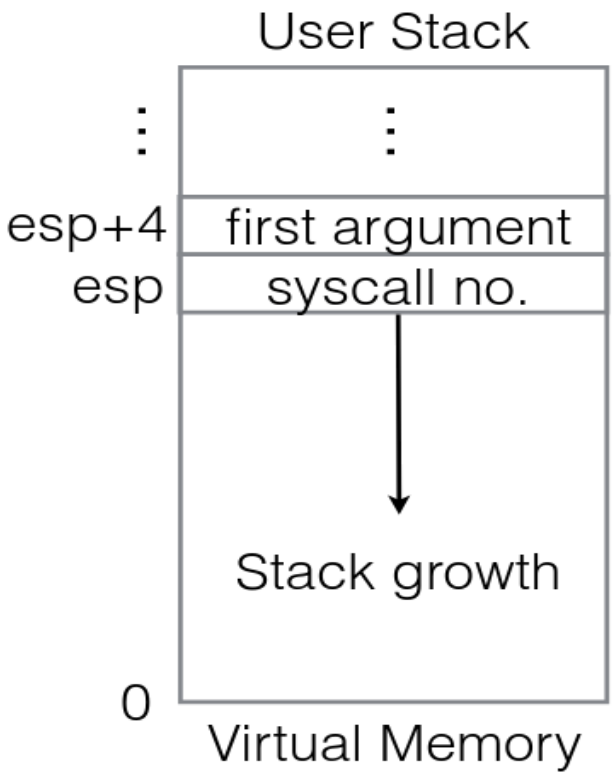
LINKÖPING UNIVERSITY

Lab 2: Syscalls

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

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    }
```



This is what you need to implement

- The syscall handler then (in kernel mode) does
  1. Reads the syscall number to decide what syscall was made (write, read, open, and so on)
  2. Based on what syscall was made, the handler reads the correct number of arguments from the stack, and then performs the syscall
- The handler does not get the arguments for the syscall directly, but it has to extract them from the stack: `f->esp`
- Note that some arguments are just pointers, strings for example are passed as pointers to the first character of the string.
- If the syscall is expected to return some value, this needs to be stored in the `f->eax` register.

Files that should be studied:

- `lib/user/syscall.[h|c]` - The syscall wrapper
- `lib/syscall-nr.h` - Syscall numbers
- `threads/interrupt.[h|c]` - Important structures
- `filesys/filesys.[h|c]` - Pintos file system

Files that should be modified:

- `userprog/syscall.[h|c]` - Implement syscall handler here
- `userprog/process.[h|c]` - If you need to clean anything up when a process is shutting down
- `threads/thread.[h|c]` - Expand current structures if needed

- Currently, the syscall handler kills every calling process
- The handler must do the things that we discuess earlier
- `f->esp` is the stack of the calling process
- The syscall number is at the top, after that are the arguments, if any
- Every syscall has its own syscall number: use it to decide the number of arguments
- Pintos currently doesn't implement FDs, you need to figure out a strategy

Some things to keep in mind when working on the lab

- Pretty much all functionality is already implemented, you task is putting it together
- Every user process should be able to have at least 128 files open at the same time
- It's **dangerous** to assume that the arguments are valid! Example of things you need to handle:
  - Given FD is not associated with any file
  - Invalid buffer size (for example -1)
  - Too many files opened
- You do not need to validate pointers yet! This will be revisited in lab 5.

LINKÖPING UNIVERSITY

FAQ

- Use `thread_current()` to get the thread struct for the calling process.
- The functions `filesys_open(char *)` opens a file, and the function `file_close(file *)` closes it
- The function `init_thread(...)` initialises every thread, while 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.

- Run `lab2test` to test your solution. It will
  - Create files
  - Open files
  - Read and write from the console
  - Try to use bad FDs
- If you want to rerun the test, remove any files created by the test first
  `pintos -- rm test0 rm test1 rm test2`
- Passing `lab2test` 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 5

- In total, you will implement 14 system calls
- Linux has around 460 system calls, depending on architecture
- Windows has more than 2000 system calls

LINKÖPING UNIVERSITY

Debugging

- Read Appendix E: Debugging tools in the Pintos documentation
- If you get "Kernel Panic", then try the `backtrace` tool
- `free` sets the bytes to `0xcc`: If you see these values, then something likely freed the memory
- Commit often! It's fairly common to accidently break Pintos in obscure ways, and often it's easier to just revert back to a working version and redo the changes.

If you get something like this:

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

Then type this (when standing in the `build` folder):

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

You should get something like 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)
```

Dag Jönsson
dag.jonsson@liu.se

www.liu.se

LINKÖPING
UNIVERSITY