Concurrent programming and Operating Systems

Lesson 1

Dag Jönsson

LINKÖPING UNIVERSITY

LINKÖPING UNIVERSITY

Webreg

Deadline 2023-01-20

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 **2023-03-08** you get 3 bonus points on the exam
- Only available for students taking the course for the first time
- Final hard deadline is **2023-03-28**! Need to have all pass in Webreg.
- Hand in through LiUs Gitlab

"Deadlines"

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

Hand in

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

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

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
- Having a 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
- There is a VM you can download and run if you want to (user and password: pintos)
- You might be able to make it work on your own machine if you use Linux, but this is not supported by us
- Don't use an IDE, use a simple editor of your choice (emacs, vim, VS Code, etc)

Lab 0

- Getting to know C and pointers
- Linked lists, your own implementation and Pintos implementation
- Setting up Pintos
- How to debug Pintos with GDB

Lab 1

- Single user process
- Adding first iteration of a system call handler
- 7 system calls to be implemented
- Afterwards, your OS should be able to:
  - Read from and write to the console
  - Create, read from, and write to files
  - Exit a process and halt the machine
- Usually takes a bit of time since you need to familiarize yourself with Pintos
- Solutions are usually around 160-200 LOC

Lab 2

- Multiple system threads
- One more system call: sleep
- sleep delays execution of the calling thread by given number of milliseconds
- Synchronisation is now required
- This lab usually takes the least amount of time
- Solutions are usually around 40-60 LOC

Lab 3

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

Lab 4

- 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 40-50 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
- Use or extend the parent-child relationship your created in lab 3.
- Validation of arguments given by the user
- Solutions are usually around 50-70 LOC

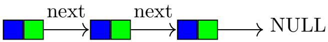
Lab 6

- Multiple processes/threads
- 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)
- 4 more syscalls; seek, tell, filesize, and remove
- Tends to take about as much time as lab 1
- Solutions are usually around 40-50 LOC

Lab 0 in detail

Linked list is a simple data structure to dynamically store data

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



Doubly linked lists are similar, but they also keep track of the previous node

Pintos implements a generic doubly linked list

```
struct list_elem {
  struct list_elem *prev;
  struct list_elem *next;
};
struct list {
  struct list_elem head;
  struct list_elem tail;
};
struct Node {
  int data;
  struct list_elem elem;
};
```

Lab 1 in detail

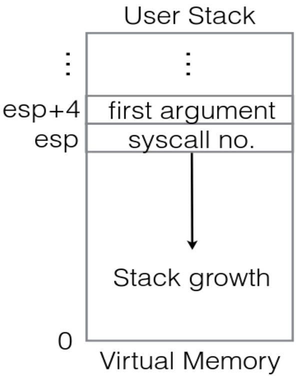
- There is only one user process at a time - no concurrency.
- Suppose a user process want to open a file, then it: **Already implemented!**
  1. Calls the function `int open(const char *file)`
  2. The function `open` puts the arguments on the stack, together with the syscall number.
  3. Produces an 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

```c

/* Invokes syscall NUMBER, passing no
   arguments, and returns
   the return value as `int'. */
#define syscall1(NUMBER)
  ({
    int retval;
    asm volatile("pushl %[number];
      int $0x30; addl $4, %%esp"
      : "=a"(retval)
      : [number] "i"(NUMBER)
      : "memory");
    retval;
  })

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



This is the assignment

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

Files that should be modified:

- userprog/syscall.[h|c] - Implement syscall handler here
- 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

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`

The syscalls

- **halt** - Shutdown the machine (halts the processor).
  Hint: Use already implemented functions.
- **exit** - Exit the current process. Deallocate all of the thread's allocated resources (eg. files). This will be revisted in later labs.
  Hint: Free resources in `thread_exit`.

- **create** - Create a file, return true if successful, false otherwise.
  Hint: Use already implemented functions
- **open** - Open a file. Returns the FD assigned to the file. Every process have their *own* collection of opened files.
  Hint: Modify the thread struct to keep track of its FDs.
- **close** - Close the file associated with the given FD.
  Hint: Use already implemented functions.

- **read** - Read from the file associated with the given FD. The user process gives a buffer (piece of memory) in which the read bytes should be written to. Returns the number of bytes read.
  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 (study `lib/kernel/stdio.h` and `lib/kernel/console.c`).

Some things to keep in mind when working on the labs

- 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
  - Too many files opened
- You do not need to validate pointers yet! This will be revisited in lab 5.

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 `lab1test2` 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 `lab1test2` 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

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

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