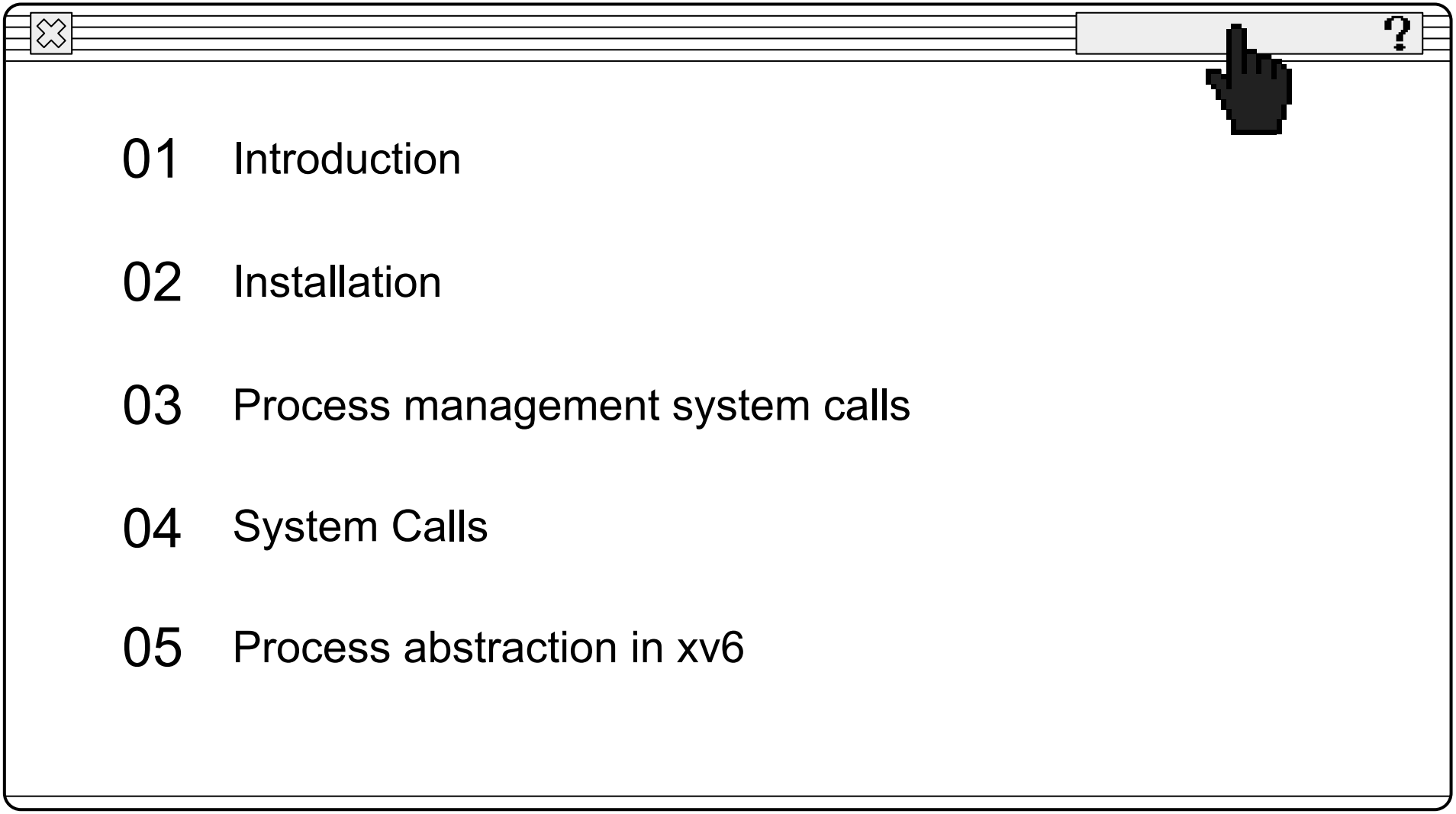
OSN Tutorial-3

An Introduction to xv6



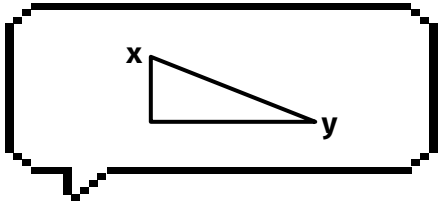
01 Introduction

02 Installation

03 Process management system calls

04 System Calls

05 Process abstraction in xv6



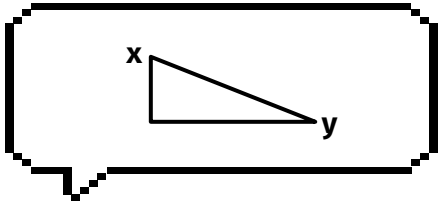
01

Introduction

It’s not all that, or is it?

xv-6 : Developed by MIT :)

In simple words, you can think of xv6 as a "toy" operating system that helps people, especially students studying computer science, learn how real operating systems function. xv6 provides the basic interfaces introduced in the Unix OS and also mimics Unix’s internal design. By studying xv6, students can grasp fundamental concepts like process management, memory management, file systems, and how different parts of an operating system interact with each other.



02

Installation

MAC users beware

Linux

Follow simple, easy to install instructions :)

Mac

Didn’t you know OSN Assignments didn’t work on Mac before :)

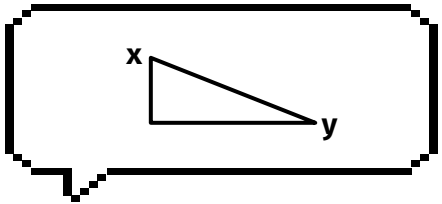
But the TAs worked hard to make it work for you guys this time

1. Use your hotspots, DON’T use the IIIT network.
2. Install brew: https://brew.sh/
3. brew install riscv-tools
4. brew install qemu

Test your Installations

Ensure that the following commands work:

1. riscv64-unknown-elf-gcc --version
2. qemu-system-riscv64 --version
3. Move to the xv-6 code directory, and run make qemu.



02

Process Abstraction

Oh no, here we go again

PROCESS ABSTRACTION

- The OS is responsible for concurrently running multiple processes.
- OS maintains all the information about an active process in the process control block (PCB).
- PCB is declared as the struct proc in xv6 (*kernel/proc.h*)

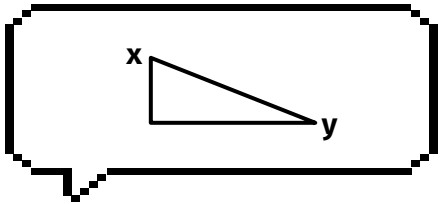
```
1. struct proc {
2.   struct spinlock lock;
3.
4.   // p->lock must be held when using these:
5.   enum procstate state;        // Process state
6.   void *chan;                  // If non-zero, sleeping on chan
7.   int killed;                  // If non-zero, have been killed
8.   int xstate;                  // Exit status to be returned to parent's wait
9.   int pid;                     // Process ID
10.
11.  // wait_lock must be held when using this:
12.  struct proc *parent;         // Parent process
13.
14.  // these are private to the process, so p->lock need not be held.
15.  uint64 kstack;               // Virtual address of kernel stack
16.  uint64 sz;                   // Size of process memory (bytes)
17.  pagetable_t pagetable;       // User page table
18.  struct trapframe *trapframe; // data page for trampoline.S
19.  struct context context;      // swtch() here to run process
20.  struct file *ofile[NOFILE];  // Open files
21.  struct inode *cwd;           // Current directory
22.  char name[16];               // Process name (debugging)
23. };
```

Kernel Stack

- Every process in xv6 is assumed to have both a user stack and a kernel stack associated with it. So when we trap into the kernel, we switch from using the user stack to the kernel stack (different for each process).
- OS does not trust the user stack which is why we have a separate kernel stack to handle system calls.
- The state is saved to (and restored from) the kernel stack.

Page Table

- Every instruction or data item in the memory image of a process has an address
  - Virtual address (starting from 0)
  - Physical address
- A page table is simply a mapping from the virtual address to the actual physical address in the memory.



03

System Calls

We are not a broken record, I swear

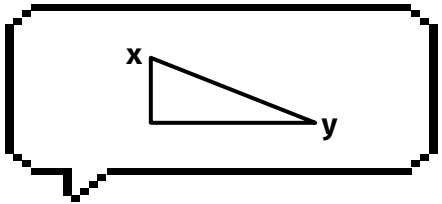
Process System Calls: Shell

- When the shell boots up, it starts the `init` process (first user process).
- Init forks shell (another user process, which prompts for input).
- Shell executes user commands as follows:
  - Shell reads command for terminal
  - Shell forks child
  - When child runs, it calls exec()
  - Shell (parent) waits for child to terminate
  - The process repeats.
- Show: *ls* does an *ls* of the user/ directory and not the main dir (do not copy for your shell!)
- Some commands (eg. cd) have to be run by the parent itself and not the child.
  - Such commands are directly executed by the shell without forking a child.
  - Not implemented in xv6 (assignment question incoming?)

Implementing syscalls

- *user/user.h*: This file has all the signatures for syscalls.
- Each call also has a user interface in the *user/* directory.
- *user/usys.S*: contains assembly code for all the syscalls (generated using *usys.pl*).
- The syscall invokes a special trap instruction (interrupt in x86 architecture), which causes a jump to the kernel code that handles the system call.
    - The state is saved onto the kernel stack. (*kernel/trampoline.S*) → trapframe.

- *kernel/trap.c:* Contains the trap handler.
- The *usertrap()* function is where the main part of the code resides. It calls the syscall() function in *kernel/syscall.c*.
- After servicing the syscall, it calls the *usertrapret()* function which restores the user state.
- *kernel/syscall.h*: Header file containing syscall numbers.
- *kernel/syscall.c:* Contains the generic code for the system calls.
- *kernel/sysproc.c*: Contains the implementation of various system calls.
- *Makefile*: Modify the UPROGS to add the interface for the newly implemented syscall.



04

Process Management System Calls

Fork it, I’mma sleep!

fork()

- *kernel/proc.c:*
  - Allocates memory for a new process.
  - Copies user memory from parent to child.
  - Handles user registers and file descriptors.
  - Sets the parent of the process.
  - Declares the process RUNNABLE.
- Tip: Do not forget to make changes to fork() while implementing syscalls in case you make use of any new variables in your code (inside struct proc).

exit()

- *kernel/proc.c:*
  - Closes all open files.
  - Assigns all its children to the init process.
  - Informs parent that it is exiting.
  - Marks itself as ZOMBIE and invokes the scheduler.

- Note: Find out about wait() and exec() yourself!

Thank you (for now)