Operating Systems

Moritz M. Konarski

November 2, 2020

Contents

Class 07.09.2020

Introduction and Syllabus

- follow rules: mute yourself, use your proper name
- for today:
  - go through syllabus
  - use first lecture
- we will study the internals about operating systems – generally some GNU linux distribution
- know the cost of various things that you use
- how are OS utilities made, how they interact
- different architectures and how they influence or limit what we can do
- generally stuff that is asked in interviews and studied in higher years

- recreation of popular UNIX utilities
- 2 larger labs – some we will do together, others will be individual
- final project: develop a "toy" file system and tools associated: inspect, defragment
- we need decent knowledge of C to complete the course
- all the prerequisites are courses I haven't done
- supplemental reading is not necessary, but gives insight into how things are done in Windows or Mac OS compared to Linux
- grade:
  - 30% practice tasks
  - 30% practice final
  - 30% course project
  - 5% participation
  - done with tasks list and the likes
  - exams done with coding interviews
- for all the rules see the syllabus
- some of the projects will modify a linux kernel
- he has been teaching this course for like 10 years now, it's probably the course he has invested the most amount of time into

First Question

Name OSs

- Windows (family of OSs)
- Mac OS
- GNU/Linux (the kernel, distros are e.g. Ubuntu, Fedora, Arch …)
  - Linux the just the kernel – the central part of the OS
  - GNU is an organization that maintains a collection of free open-source software
- FreeBSD
- iOS
- Android (also based on Linux – not GNU)

Slides

Purpose of the OS

- to run programs (but not needed), memory management, storage management, manage hardware – **manage hardware and time resources**
- provide abstractions for programmers – e.g. printing to the screen
- general scheme:
  1. Hardware – does the actual work
  2. Kernel – very direct access to hardware
  3. Applications – access to hardware through kernel abstractions
  4. User
- things to manage
  - CPU
  - Memory
  - Disks
  - Other peripheral devices (GPU, …)

- manage the ideas of processes and threads, files etc for the devices that don't actually know what they are – they only know streams of data
- shell as a pretty direct access to the kernel and utilities – generally stands for access and not just cli, e.g. Windows Explorer is a type of shell

Additional Information

- `symbian` was super popular in Nokia and Sony-Ericsson
- `FreeDOS` is a OS that basically only allows you to access a disk – many notebooks ship with it because it is simpler and cheaper than Windows or most other OSs
- `ReactOS` is an open source implementation of the Windows kernel that will then be compatible with all the programs and drivers
- `KolibriOS` is super small – the idea is that it can fit and start from a 1.4 MB floppy disk
- `TempleOS` – a OS developed by 1 person with certain mental problems – he thought God was talking to him and telling him to do that

Class 11.09.2020

General Terminal Commands

- we will cover an important topic
- and we'll work on our server
- they were working on `true` and `false`
- then move on to `yes` and `echo`
- read the two sets of slides that are on `Practice.md`
- explanations of how `cd`, `mv`, `cp`, `ls`, `rm`, `mkdir`, `pwd`, `cat`, `touch`, `less`, `nano` work – super basic stuff
- `touch`ing an existing file will change its modification date – useful for make and stuff to force recompilation
- `readline` is a C library for interactive cli input
- read book as outlined in presentations
- `bash` supports scripting, who'd have thought
- in C, `0` is `false`, `1` is `true`
- we can use our own `./true` in bash
- write `yes` – done
- basic stuff about bools, stdio, argc, argv

System Calls

- system calls are channels of communication
- memory and resources are divided into kernel space and user space
- different levels of access
  1. kernel
  2. drivers
  3. drivers
  4. applications
- this secures and improves programs
- kernel interface is the bridge between kernel space and user space
- puts (application) → write (C library) → SYSCALL (kernel)
- programs cannot do whatever they want, their environment is limited
- the kernel controls what you can do
- using `gdb` to analyse the syscall `write`

Class 14.09.2020

- how OSs work
- look at how OSs were developed, history, different kinds
- talking about unikernels and 'simple' ways to build your own OS
- what is a kernel?
  - we need system programs
  - kernel does the basic stuff
  - all of the stuff
  - hybrid kernels, monolithic kernels, micro-kernels

Lecture 18.09.2020

- continue work on the labs
- syscalls – what are they, why are they needed, etc
- next week we will get a graded assignment – specifics will
- syscalls are a direct way to communicate with the OS
- the kernel acts as a security barrier – keeps people from doing stuff they should not

Unix programs

- echo – just print supplied text
- many simple things that are really good at the one thing they do
- then you can use these to build more complicated programs
- we are now stripping away the stuff from the print action
- down to write, then to syscalls, then to assembly

Class 21.09.2020

- last class we discussed what an OS is, now we will be moving onto history
- how did we get to OSs?

History

1945 to 1955

- Zuse machines from Germany
- manual programming using binary or even cables
- then came punch cards that made stuff easier – they were helpers for feeding information to the system
- they used mechanical relays and later vacuum tubes
- no programming languages, not even assembly languages

1955 to 1965

- transistors became a thing and that made them smaller and slightly cheaper
- now programming languages and assembly languages became more popular
- one of the first and still active languages is FORTRAN – many scientific computing elements depend on it
- precursors were batch and job control languages – they controlled submission and execution of programs on computers

1965 to 1980

- integrated circuits – miniaturization and power efficiency
- cheaper computers, more people using them – first major operating systems were created
- IBM OS/360 was the first time a laid-out instruction architecture was implemented – they really planned stuff out such that all machines in the series could run each others programs

- MULTICS was a first timer OS where files were organized in a hierarchical file system that could have many different levels – it also allowed for concurrency and thus multiple users through terminals
- MULTICS treated most things like files – mounted devices etc are all writable memory locations
- UNIX features were taken from MULTICS and implemented
- Ken Thompson and Dennis Ritchie et al. did this – some of the most influential people ever
- UNIX was just given out for free, you only had to pay for the tape it came on
- then someone created BSD from UNIX
- CTSS -> MUTLICS -> UNIX
- first PCs – didn't really have resource sharing anymore
- then came Disk Operating Systems (DOS), from Microsoft, Apple, IBM
- XEROX PARK was one of the most influential research projects
- XEROX Alto etc
- Stalman wrote all the utilities because he wanted to – Torvalds wrote the kernel – so this is what happened

Class 25.09.2020

- further work on command line programs
- graded project will be introduced on Monday
- we wrote `cat` using syscalls and used `strace` to find out what calls the actual program uses
- first major thing will be a lot of fun if you believe him
- we have to look at a toy shell and play with it

Class 28.09.2020

- we now have a private repository where we can put our code
- introduction to gihub classroom, canvas

Project 1

Part 1

Until next Monday 23:00 1. install Debian amd64 in QEMU see here 2. install Debian aarch64 in QEMU see here 3. Download and start mis64el Debian distro in QEMU see here 4. Download and start riscv64 Debian distro in QEMU see here

Part 2

- work on particular source files inside of all these systems

Class 02.10.2020

- use the repo toksaitov/syscall-project to find all the data
- download the data and use it to install the operating systems

- our standard libraries do very little actually, the kernel of the os does most of the work when it comes to printing etc
- the scripts are kinda finicky, if you don't follow the instructions they wont work because they depend on specific directories
- `syscall` is way more expensive than a simple function call – function calls are just push return address to stack and then jump to func, a syscall needs the kernel to do stuff, it then also does
- write `touch` program
-

Class 05.10.2020

- examples of how to submit files etc to our repository

Part 2

- test the shell after writing each part of assembly
- read the code and understand it, he will ask about it
- we can also write a C program and then compile it and see what C does and copy it

Class 09.10.2020

- we will continue to talk about project 1
- he gives an example of how to call `read` from x86
- this is kinda boring
- to actually use your syscalls – ish_syscalls.h and take the appropriate `#define` out of the `#if` block and then test the function

Class 13.10.2020

- don't wait till the last moment to start the project, it might take a while
- don't put unnecessary stuff into our private submission repo, it's only for the code we are supposed to submit
- he might actually give us the machines we had to create in the first part to allow us to use them and focus on the second part

Toy Shell

Fork Syscall

- it forks the process into either:
  - parent, then wait for child to finish
  - child, execute, exit to parent
- implementation of a simple fork syscall
- on AARCH64 we have to use 'clone
- use gdb to figure out what happens

- use `set follow-fork-mode <parent/child>`
- `catch syscall` will catch any system call – we can then examine it
- `c` to continue program
- `layout asm` to show the assembly – then look for the syscall and copy the values
- you don't need to use all registers for clone, just give it zero
- we need to get a thread ID and use the `mrs` call that gives the register
- move `tpidr_el0` to the appropriate register

```c
int main(int argc, char *argv[])
{
    pid_t pid = fork();
    if (pid > 0) {
        puts("We are in the parent process!");
        int child_exit_status;
        waitpid(pid, &child_exit_status, 0);
    } else if (pid == 0) {
        puts("We are in the child process!");
    } else {
        puts("Failed to create a new process...");
    }

    return 0;
}
```

Figure 1: PID test program

Class 16.10.2020

- QA session

Class 23.10.2020

Questions about MIPS64EL

- I cannot clobber `a4` in my fork system call. Why? – Confusion in gcc about which registers are available in 32 and 64 bit?
- Do we need to do anything about `a3` being the error indication (if `-1`) (number is in `v0`) or can we ignore it?
- watch the office hours https://man7.org/linux/man-pages/man2/syscall.2.html

Notes

-

Figure 2: Tricky syscalls

Midterm Notes

Labs

- to see the return code: `echo $?`
- for syscall numbers `/usr/include/asm/unistd_64.h`

`true`

- returns 0 (C false)

`false`

- returns 1 (C true)

`yes`

- prints `y` or provided arguments separated by spaces forever
- use print, strlen

```
long strlen(const char *s) {
    long i;
    for (i = 0; s[i] != '\0'; i++) ;
    return i;
}
```

`echo`

- print space-separated arguments, ending in a newline

`cat`

- `openat` for file
- for each argument:

```
ssize_t bytes_read;
while((bytes_read = read(fd, buff, BUFF_SIZE)) > 0) {
    write(1, buff, bytes_read);
}
close(fd);
```

`pwd`

- `getcwd` syscall

`touch`

- if file does not exist: create empty file
  - `openat` with arguments – `O_WRONLY|O_CREAT|O_NOCTTY|O_NONBLOCK`
- if file exists – update time stamp
  - `openat` with arguments – `O_WRONLY|O_CREAT|O_NOCTTY|O_NONBLOCK`

mkdir

- `mkdir` with `0777` as argument
- user, group, other

mv

- `renameat2(AT_FDCWD, "test/", AT_FDCWD, "test-2/", RENAME_NOREPLACE) = 0`
- this is done for any type
- move all but the last file into the last location

rm

- for `rm -rf`

```
unlinkat(AT_FDCWD, "test-2/", AT_REMOVEDIR) = 0
```

- for `rm`

```
unlinkat(AT_FDCWD, "touch_2.c", 0) = 0
```

cp

- if `cp -r`
  - check if dir exists with fsstatat
    - if not, `mkdir`
  - for each file in the directory – open the file, read from it, then write to it
  - all with `openat`
- if just `cp`
  - stat both files to check existance
  - read from one file, write to the other one

ls

- `openat(AT_FDCWD, ".", O_RDONLY|O_NONBLOCK|O_CLOEXEC|O_DIRECTORY) = 3`
- then `fstat(3, {st_mode=S_IFDIR|0755, st_size=4096, ...}) = 0`
- `getdents64(3, 0x55dc15be8420 /* 9 entries */, 32768) = 224`
- then simply write the `names[]` to `stdout`

Kernel Project

get_pids.c

- see file

get_task_info.c

- see file

task_info.h

- see file

Presentations

Introduction

- most fundamental system program is the operating system
- it manages the computer's resources and provides a base for application development
- Hardware → Kernel → Applications → User
- it does resource management and machine abstraction
- resources:
  - CPU
  - memory
  - disks
  - other devices
- machine abstraction:
  - processes
  - threads
  - files
  - shell
- process, files systems, device abstraction can be modeled as trees
- system calls hide the complexity of the real machine
- single-tasking, multi-tasking, single-user, multi-user are what they sounds like
- real-time operating system – should not have a time delay on the input
- distributed – separate nodes that are networked
- embedded os – specific tasks for non-pcs
- library os – single address-space machine that has the bare minimum of stuff for programs to run and they are then compiled and run directly on hardware without an os
- microkernel: bare minimum runs in kernel space, the rest in user mode
- hybrid kernel: mix of things – Windows, OS X
- monolithic kernel: everything but applications run in kernel mode – Linux

History

1945 to 1955

- plug boards, punch cards

1955 to 1965

- transistors
- assembly languages
- programming languages
- jobs, job control systems, batch systems (jobs without user input than can be run as resources permit)

1965 to 1980

- integrated circuits
- major os projects: IBM OS/360, CTSS, MULTICS
- multi threading
- time sharing

- UNIX 1970, Ken Thompson, Dennis Ritchie
- CTSS -> Multics -> UNIX
- first in asm, then later in C

1980 to 1990

- large scale integration circuits – CPUs
- microcomputers
- PCs
- DOS (apple, microsoft,…)
- GNU, BSD

1990 to present

- internet
- mobile boom
- iot

Boot Process

1. power up
2. internal firmware
   - BIOS
   - UEFI
   - custom stuff
3. boot loader
   - multiple stages
   - MBR, GPT
4. kernel
   - CPU and hardware setup
5. initialization service
   - init, upstart, systemd, launchd
6. user space programs
   - login, shell

Processses

- executable format: structure of an exe
- produced by a linker
- format header, program header, then links to code, global and static data
- common formats: PE, ELF, Mach-O
- each process has: pid, state, priority, cpu state (registers, program counter, stack pointer), virtual memory (memory map), opened files, code, data, parent id, working directory, exit status,…
- the kernel maintains a list of all the process descriptors
- process states: running, ready, blocked for scheduling
- context switching to give all exes an equal amount of time
- processes can be created and exit when they are done
- io devices can interrupt, the clock can interrupt

Context Switching

1. save current cpu state
2. load a CPU state from next executable
3. run that one

Threads

- lightweight process that shares some elements with its parent process
- e.g. memory address space, opened files, global data, …
- they tend to have a private stack, registers, state
- same data structure for processes and threads

Scheduling

- either compute-bound or IO-bound processes
- the time-bottleneck is either computation or IO
- compute-bound: long CPU bursts, little IO
- IO-bound: short CPU bursts, a lot of IO
- a scheduler can start and stop these processes to make a efficient use of the system resources
- preemptive scheduling – mostly by priority
- non-preemptive goes by what processes finish when
- fairness and balance are common goals

Batch Scheduler

- maximize throughput
- minimize turnabout
- maximize CPU utilization
- first-come first-served
- shortest job first
- shortest remaining time next

Interactive Scheduler

- minimize response time
- round-robin scheduling
- shortest process next
- lottery scheduling
- fair-share scheduling

Real-time Scheduler

- meet deadlines
- hard real time vs soft real time:
  - hard real time have deadlines that must be met
  - soft real time has a certain tolerance but still not desirable
- periodic or non-periodic events