

6.s081

Intro to OS

Lecture Notes

Kevin Yang

9/8/21-?/?/??

1 Lecture 1

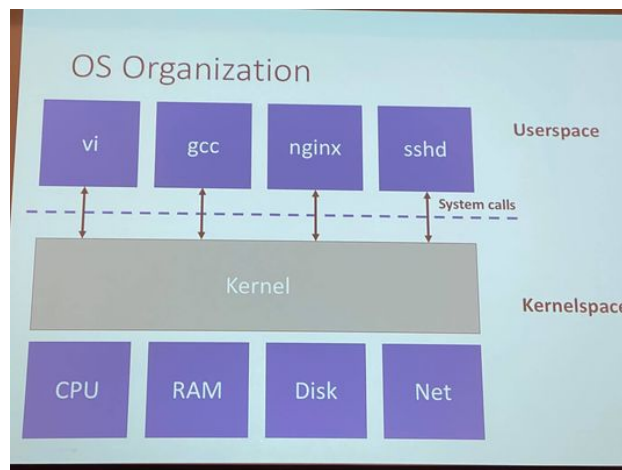
1.1 What is the purpose of an OS?

1. Abstraction

- Hides hardware details for portability and convenience
- Must not get in the way of high performance
- Must support a wide range of applications

2. Multiplexing

- Allows multiple application to share hardware
- Isolation to contain bugs and provide security
- Sharing to allow cooperation



1.2 OS abstractions

- Process (a running program)
 - Instructions
 - Memory Storage/Allocation
- Memory allocation

- File descriptor
- File names and directories
 - Namespaces
- Access control and quotas
- Many others: users, IPC, network sockets, time, etc.

1.3 User ↔ Kernel Interface

- Primarily system calls
- Examples:
 - `fd = open("out", 1)`
 - `len = write(fd, "hello how\n", 6)`
 - `pid = fork();`
- Look and behave like function calls but they aren't. They are switching between user and kernel space and directly call things in the hardware

1.4 Why OSES are interesting

- Unforgiving to build: debugging is hard
- Design tensions:
 - Efficiency vs Portability/Generality
 - Powerful vs Simple
 - Flexible vs Secure
- Challenge: good orthogonality, feature interactions
- Varied uses from smartbulbs to supercomputers
- Evolving HW: NVRAM, multicore, 200Gbit networks

1.5 Take this course if you:

- What to understand how computers reallyh work from an engineering perspective
- Want to build future system infrastructure
- Want to solve bugs and security problems
- Care about performance

1.6 Logistics

- Course Website
 - <https://pdos.csail.mit.edu/6.s081>
 - Schedule, course policies, lab assignments, etc
 - Videos and notes of 2020 lectures
- Piazza
 - <https://piazza.com/mit/fall2021/6s081>
 - Announcements and discussions
 - Ask questions about labs and lecture

1.7 Lectures

1. OS concepts
2. Case studies of xv6 — a small simple OS
3. Lab background and solutions
4. OS papers
 - Submit a question before each lecture
 - Resource: x6 book

1.8 Labs

- Goal: Hands-on experience
- Three types of labs:
 1. Systems programming: due next week
 2. OS primitives: e.g. thread scheduler
 3. OS extensions: e.g. networking driver

1.9 Collaboration

- Feel free to ask and discuss questions about lab assignments in class or Piazza
- Discussion is great
 - But all solutions(code and written work) must be your own
 - Acknowledge ideas from others
- Do not post solutions on Github etc

1.10 Covid-19 and in-person learning

- Masks are **required**; must be worn correctly
- If you have symptoms or test positive...
 - Don't attend class, contact us right away
 - We will work with you to provide course materials

1.11 Grading

- 70% labs, based on the same tests you will run
- 20% lab checkoff meetings
 - We will ask questions about randomly selected labs during office hours
- 10% homework and class/piazza participation

1.12 Back to system calls

- Will use xv6, the same OS you'll build labs on
- xv6 is similar to UNIX or Linux, but way simpler
 - Why? So you can understand the entire thin
- Why UNIX?
 1. Clean design, widely used: Linux, OSx, Windows(mostly)
- xv6 runs on Risc-V, like 6.004
- You will use Qemu to run xv6 (emulation)

System call	Description
<code>int fork()</code>	Create a process, return child's PID.
<code>int exit(int status)</code>	Terminate the current process; status reported to <code>wait()</code> . No return.
<code>int wait(int *status)</code>	Wait for a child to exit; exit status in <code>*status</code> ; returns child PID.
<code>int kill(int pid)</code>	Terminate process PID. Returns 0, or -1 for error.
<code>int getpid()</code>	Return the current process's PID.
<code>int sleep(int n)</code>	Pause for n clock ticks.
<code>int exec(char *file, char *argv[])</code>	Load a file and execute it with arguments; only returns if error.
<code>char *sbrk(int n)</code>	Grow process's memory by n bytes. Returns start of new memory.
<code>int open(char *file, int flags)</code>	Open a file; flags indicate read/write; returns an fd (file descriptor).
<code>int write(int fd, char *buf, int n)</code>	Write n bytes from buf to file descriptor fd; returns n.
<code>int read(int fd, char *buf, int n)</code>	Read n bytes into buf; returns number read; or 0 if end of file.
<code>int close(int fd)</code>	Release open file fd.
<code>int dup(int fd)</code>	Return a new file descriptor referring to the same file as fd.
<code>int pipe(int p[])</code>	Create a pipe, put read/write file descriptors in <code>p[0]</code> and <code>p[1]</code> .
<code>int chdir(char *dir)</code>	Change the current directory.
<code>int mkdir(char *dir)</code>	Create a new directory.
<code>int mknod(char *file, int, int)</code>	Create a device file.
<code>int fstat(int fd, struct stat *st)</code>	Place info about an open file into *st.
<code>int stat(char *file, struct stat *st)</code>	Place info about a named file into *st.
<code>int link(char *file1, char *file2)</code>	Create another name (file2) for the file file1.
<code>int unlink(char *file)</code>	Remove a file.

In UNIX, for std: Use `make qemu` to run xv6 emulation. `-smp` tag controls number of multiprocessors.

- 0 input
- 1 output
- 2 errors

`read` loads a keyboard text buffer in the kernel space which is then sent into the user space's program when enter is pressed. A program like `copy` will then write to the kernel using the user's input.

`open` will open a file based on the path provided. It takes flags such as `O_WRONLY` or `O_CREATE`. `write` is used to write to a certain file by sending in a string and the number of chars in the string.

The shell like a very simple programming language that helps you chain together other instructions and programs using things like pipes and other commands. Command shells are `bash`, etc.

`fork` creates a completely identical process with copied over memory and instructions. It uses the return code `pid`, a unique number (process identifier), to differentiate between the parent and the child. It is a single system call that is called once but is returned twice. If the `pid` is 0 then it is a child. Can cause a race condition since both processes output to the same console.

`exec` tells the kernel to run another program/instruction by loading another binary code into the console. This replaces the existing program binary code so a new fork is able to run something new.

The program runs `wait` really fast when `exec` is called on the child so there are much less race issues. It provides a status of whether the process succeeded or failed. `exec` jumps to a new instruction and clears away everything else in the forked program.

The `exit` system call takes the child status and delivers it to the parent as it waits. 0 is a success and 1 is a failure. If you have multiple forks that execs, it will return the first status return rather than use unique pids.

`fds[2]` is used to set up two of file descriptors. These FDs are used in `pipe` that is used to read/write from/to. The text written into the pipe is stored in a buffer that the kernel maintains for each pipe. Using pids, you are able to use pipes to communicate between two processes.

2 Lecture 2

Introduction to C

2.1 Why use C?

Why we might not:

- C is old and complicated, with subtle behaviors and sharp edges
- Lots of recent work building OSes in newer languages. Rust, Go, Java, etc.

However:

- Used everywhere in OS engineering. Many (not all) real systems are in C
- Supported everywhere on (nearly) every platform
- Forces you to gain a better understanding of the underlying machine

2.2 C vs Python: types, variables, and values

- In any language like Python, a value has a type, and a variable can contain any value of any type

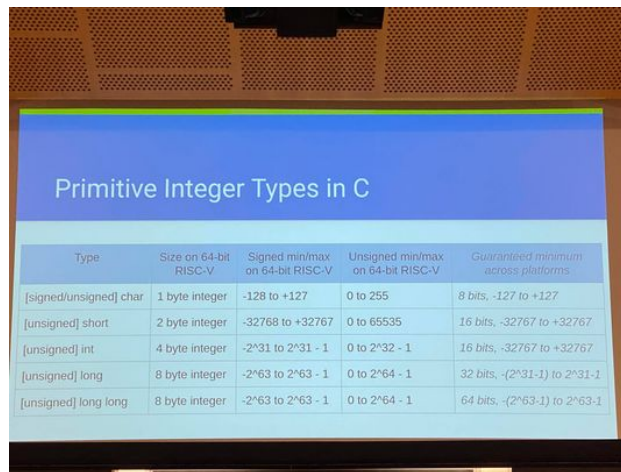
```
x = 10.5
y = "hello"
```

- In C, values do not store any type information. All type information is stored in variables.

```
int x = 10;
char * y = "hello, world!";
```

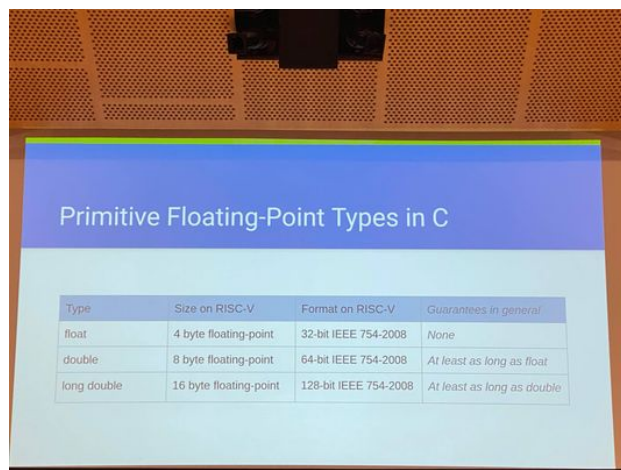
- In python, the type of a variable is stored in memory with the variable. In C, only the bits are stored and each variable is backed by a *memory region*. No variable type information is kept past the compiler.

2.3 Primitive Integer Types in C



Type	Size on 64-bit RISC-V	Signed min/max on 64-bit RISC-V	Unsigned min/max on 64-bit RISC-V	Guaranteed minimum across platforms
[signed/unsigned] char	1 byte integer	-128 to +127	0 to 255	8 bits, -127 to +127
[unsigned] short	2 byte integer	-32768 to +32767	0 to 65535	16 bits, -32767 to +32767
[unsigned] int	4 byte integer	-2^{31} to $2^{31} - 1$	0 to $2^{32} - 1$	16 bits, -32767 to +32767
[unsigned] long	8 byte integer	-2^{63} to $2^{63} - 1$	0 to $2^{64} - 1$	32 bits, $-(2^{31}-1)$ to $2^{31}-1$
[unsigned] long long	8 byte integer	-2^{63} to $2^{63} - 1$	0 to $2^{64} - 1$	64 bits, $-(2^{63}-1)$ to $2^{63}-1$

2.4 Primitive Floating Point Types in C



Type	Size on RISC-V	Format on RISC-V	Guarantees in general
float	4 byte floating-point	32-bit IEEE 754-2008	None
double	8 byte floating-point	64-bit IEEE 754-2008	At least as long as float
long double	16 byte floating-point	128-bit IEEE 754-2008	At least as long as double

2.5 Defining primitive variables in C

```
int value_1;  
int value_2 = 83;  
float value_3 = 125.0;  
char value_4, value_5=3, value_6=0xFF
```

The declaration of a variable simply tells the compiler to set out the needed amount of the memory. You need to initialize the variable with a number or there is no guarantee of what the variable is. The layout of variables in memory is not guaranteed.

2.6 Endianness

How do we write out a number like 0x12345678 in memory?

Big Endian:

0x12, 0x34, 0x56, 0x78

or Little Endian:

0x78, 0x56, 0x34, 0x12

The endianness depends on the platform but our RISC-V platform is little endian.

2.7 Types of memory in C

- Stack Memory
 - Local variables allocated within function. This memory is destroyed and may be reused
 - Not initialized by default. Will reflect whatever happened to be in that piece of memory
- Static Memory
 - Variables declared outside any function, variables declared with `static`
 - A single copy is stored at a predefined non changing address
 - Initialized to 0 by default
- Heap memory
 - Explicitly allocated (`malloc`) and deallocated

2.8 Key topic in C: memory safety

- Use-after-free
- Double-free
- Uninitialized memory
- Buffer overflow
- Memory leak
- Type confusion

2.9 Key topic in C: pointers

We represent regions of memory in C using *pointers*:

```
int value_1 = 6828;
int *pointer_to_value_1 = &value_1;
*pointer_to_value_1 = 6081;
printf("%d\n", value_1); // prints 6081, not 6828
```

Pointers are references that describe the location of an underlying piece of memory.

2.10 Pointers are integers in disguise

Pointers are integers that specify the address where a region of memory starts.
You can have pointers to pointers!

2.11 Another data type: arrays

Arrays are basically a continuous section of memory that is divided into multiple variables of the same type. The compiler handles the indexing and memory locating for the array.

- Arrays are not lists. They have a fixed length and are not resizable
- Laid out sequentially in memory
- Arrays are 0-indexed, not 1-indexed. Unlike python, you cannot use negative indices.

2.12 Pointer Arithmetic

- Dereferencing the nth element of an array (`array[n]`) is the same as dereferencing the memory n plus the array pointer
- Taking a reference to the nth element of an array is the same as adding n to the array pointer
- Pointer arithmetic multiplies by the size of the underlying data type

2.13 Casting between primitive types

You can cast between different integer types but you may lose precision such as between float and int. You can also cast between pointers and integers.

Casting can cause memory unsafety.

2.14 What size are pointers

Depending on the type of the pointer's variable, the size can change.

2.15 Unusual data type: void

- Lack of data type
- Useful in return types and parameters of functions
- Can't declare a variable as void
- Can define a void * pointer.

2.16 Definitions vs Declarations

- You must declare a variable before it can be used because C needs to know its type or type signature
- You must define each variable or function exactly once in your code base

2.17 Strings

Strings are arrays of chars

2.18 Stopped paying attention and decided to do Down 4 Across

Just read the slides. For the C review.