

6.s081

Intro to OS

Lecture Notes

Kevin Yang

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

1 Lecture 1

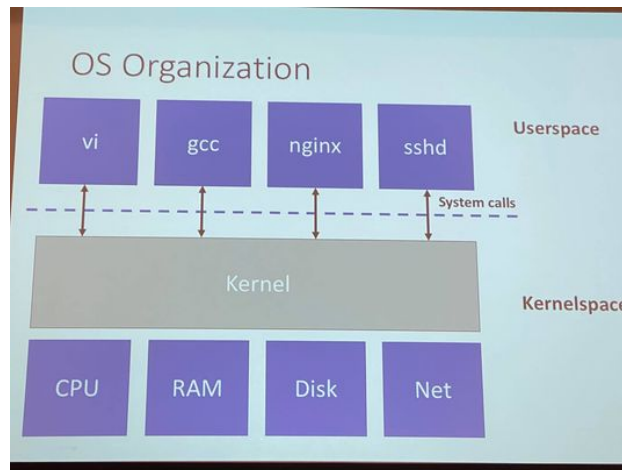
1.1 What is the purpose of an OS?

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 really 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 <code>n</code> 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 <code>n</code> 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 <code>n</code> bytes from <code>buf</code> to file descriptor <code>fd</code> ; returns <code>n</code> .
<code>int read(int fd, char *buf, int n)</code>	Read <code>n</code> bytes into <code>buf</code> ; returns number read; or 0 if end of file.
<code>int close(int fd)</code>	Release open file <code>fd</code> .
<code>int dup(int fd)</code>	Return a new file descriptor referring to the same file as <code>fd</code> .
<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 <code>*st</code> .
<code>int stat(char *file, struct stat *st)</code>	Place info about a named file into <code>*st</code> .
<code>int link(char *file1, char *file2)</code>	Create another name (<code>file2</code>) for the file <code>file1</code> .
<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

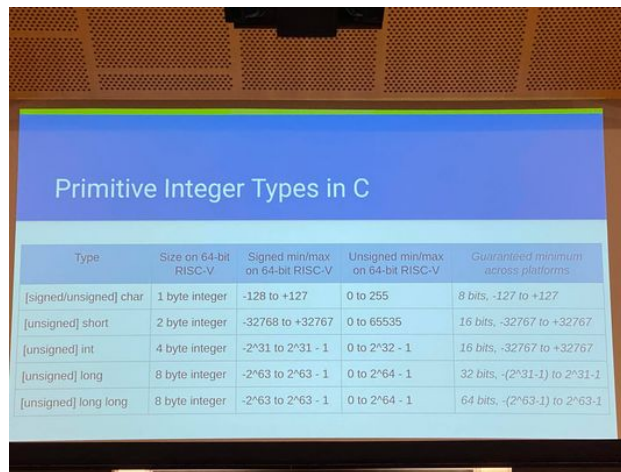
```
x = 10.5  
y = "hello"
```

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

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

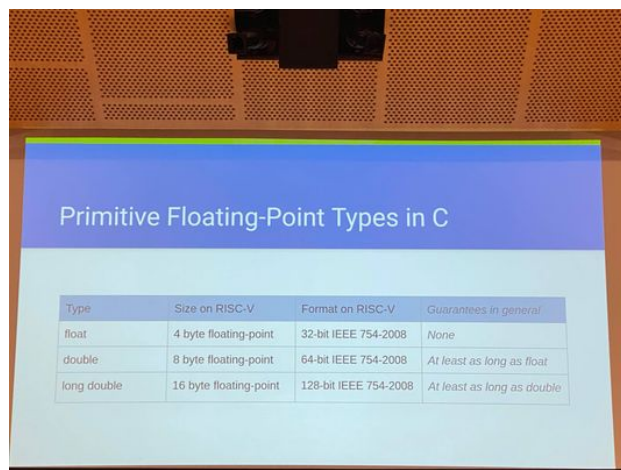
- In python, the type of a variable is stored in memory with the variable. In C, only the bits are stored and each variable is backed by a *memory region*. No variable type information is kept past the compiler.

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

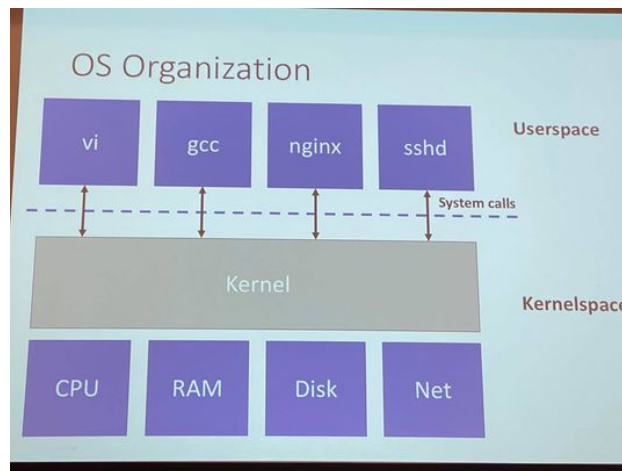
2.17 Strings

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.

3 Lecture 3: OS Organization



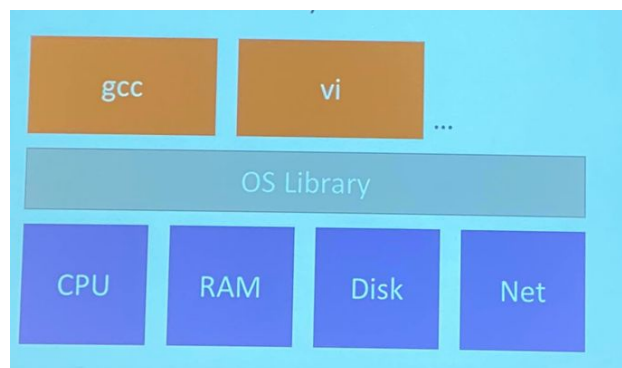
3.1 Multiplexing

- Must handle multiple applications
- Needs isolation between them
- But must share resources too

3.2 Strawman Solution

- Applications use hardware directly
- OS acts as a library

Malicious applications can use up all the CPU's computing power and not allow other users to accept inputs. A programmer could have no malicious intentions but they can have a bug or have an exploit that others can use to cause malicious attacks.



3.3 Problem: Can't multiplex

- Each app must periodically give up hardware
- But weak isolation
 - App forgets to give up -> nothing else runs
 - App has end-less loop -> nothing else runs

- Can't even kill misbehaving app from another app
- This scheme is sometimes used in practice
 - Called *cooperative* scheduling

3.4 Bigger problem: Memory isolation

- All apps share physical memory
 - One app can overwrite another app's memory
 - One app can overwrite the OS library
- No security

3.5 UNIX Interface

- Processes (instead of cores): `fork()`
 - OS transparently allocates cores. This allows the OS to decide which/when process runs. To the process, it appears that the core was never given up.
 - * Save + Restore registers
 - Enforces that processes give them up
 - * Periodically re-allocates cores. Guarantees system's response rate even if a process is not cooperating.
- Memory (instead of physical memory): `exec()`, `brk()`
 - Each process has its own memory
 - OS decides where to place app in memory. Resolves issue that one program can access another program's memory through abstraction.
 - OS enforces isolation between apps
 - OS stores image in file system (loaded with `exec`)

`exec()` moves memory in order for it to be executed

`brk()` asks the OS for more memory

Will go over these later

Question: How does `sbrk()/mmap()` work?

Answer: Every process has an address space. The location is where the memory you are trying to read and write to is located. For a program, you have the program code in memory, then the program heap. `sbrk()` changes size/location of the heap.

- Files instead of disk blocks: `open()`, `read()`, `write()`
 - OS provides convenient names
 - OS allows files to be shared by apps and users
- Pipes instead of shared physical memory: `pipe()`
 - OS buffers data
 - OS stops sender if it transmits too fast

3.6 OS must be defensive

- An app shouldn't be able to crash the OS
- An app shouldn't be able to break out of isolation
- Need strong isolation between apps and OS

Solution: use CPU hardware isolation
Using privileges

3.7 CPU provides user/kernel mode

- Kernel mode: can execute "privileged" instructions
 - changing back to user mode
 - programming a timer chip
 - controlling virtual memory
- User mode: can't execute "privileged" instructions
 - if it tries, it faults to kernel

Plan is to run kernel in kernel mode , apps in user mode
RISC-V uses a third mode called M-mode(monitor mode) that is not used.

3.8 CPU provides virtual memory

- Page tables translate virtual address to physics
- Defines what physical memory an app can access
- OS sets up page table so each app can only access its own memory

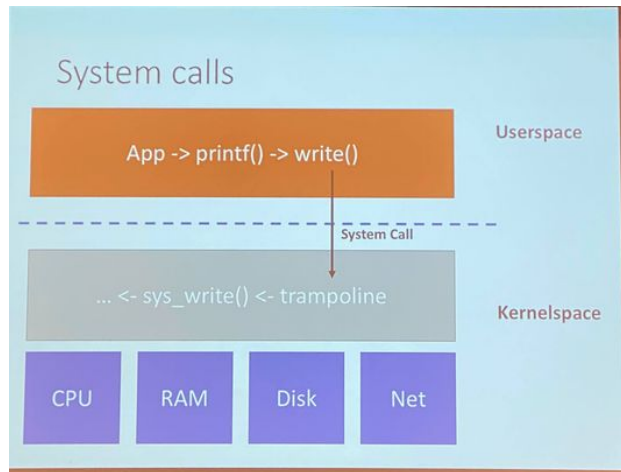
Memory is divided into page sized chunks and uses a page table to link physical memory pages to virtual memory pages.

3.9 System calls

- Apps need to communicate with kernel
- Solution add instruction to change mode in controlled way
 - ecall on RISC-V
 - Enters kernel at pre-agreed instruction address

A trampoline is a piece of code that sets up something that allows another code to run. It sets the CPU in a suspicious state so it is ready to call `sys_write`.

System calls are indeed a fault. The trampoline handles the fault. If we have a memory fault, it can also call the trampoline.



3.10 Kernel is trusted computing base

- Kernel must be correct
 - Bugs could allow user apps to circumvent isolation
- Kernel must treat user apps as suspect
 - Each system call argument must be validated
 - User/kernel mode transitions must be set up correctly
- Require a security mindset
 - A bug could be a security exploit

3.11 Is Isolation possible without HW support

- imagine no kernel/user mode or virtual memory
- Yes! Use a strongly typed programming language
 - singularity OS
- Compiler is then the TCB
- But HW support is the most common plan

3.12 Monolithic Kernels

- OS runs in kernel space
- xv6 does this, so does Linux
- Kernel interface == system call interface
- Kernel is one big program with everything (filesystem, drivers, memory management)
- Pros
 - Easy for subsystem to cooperate
 - * one cache for file system and virtual memory
 - Good performance

- Cons
 - Interactions are complex, leads to bugs
 - No isolation within

3.13 Microkernels

- Runs OS services as ordinary user services
 - a server provides the file system
- kernel implements minimal mechanism to run services in user space
 - Processes with memory
 - Interprocess communication
- Pros: more isolation, more fault tolerance
- Cons: less performance, high complexity

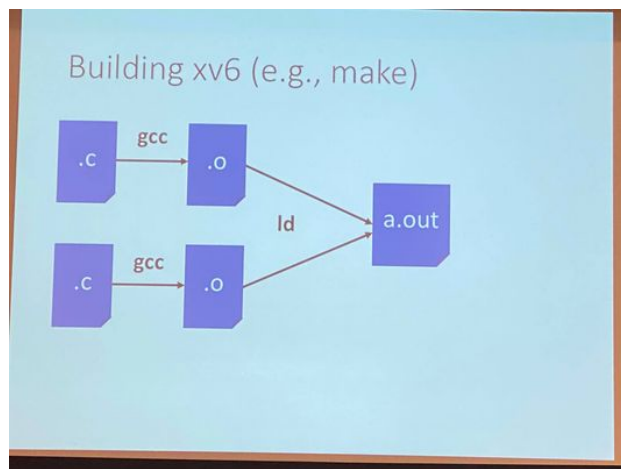
Windows WSL is a micro kernel. These are very complex since a component can break and it would be hard to fix since they are not unified under the kernel such as device drivers.

3.14 xv6 case study

- Monolithic kernel
 - UNIX system calls are the kernel interface
- Source code reflects OS organization
 - user/
 - kernel/
- Kernel has several parts
 - kernel/defs.h, kernel/proc.c, kernel/fs.c, etc

Goal: simple, easily readable/understandable

3.15 Building xv6



3.16 Risc-V (emulated) computer

- A very simple board, eg no display
- Risc-V processor with N cores
- RAM (128 MB)
- Supports interrupts (PLIC, CLINT)
- Supports UART (serial port)
 - Xv6 uses this to provide a console (out)
 - Xv6 uses this to get keyboard inputs (in)
- Support E1000 network card (over PCIe)

3.17 Why develop on qemu

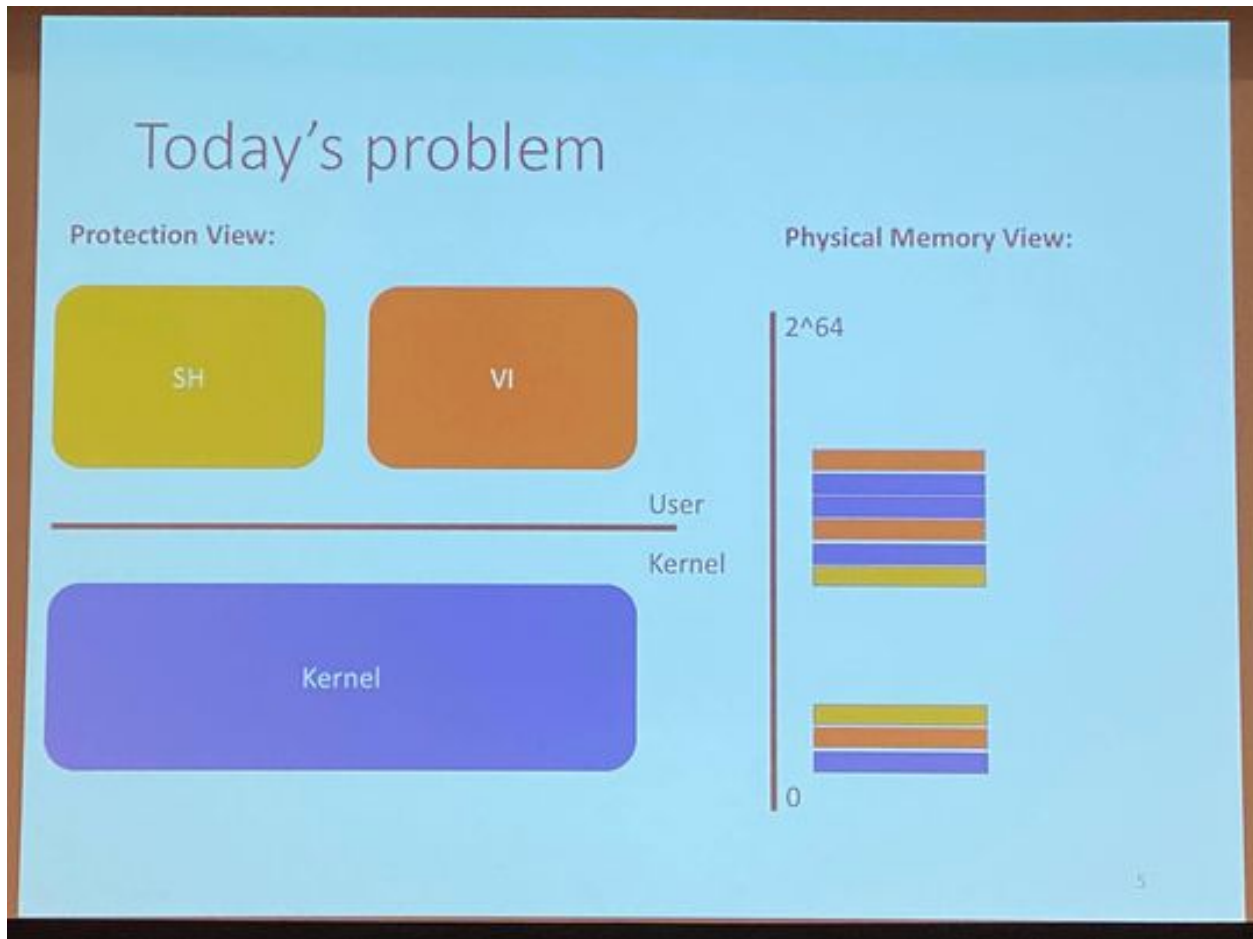
- More convenient than using real hardware
- Qemu emulates several types of computers
 - We use the "virt" one
 - Close to the SiFive board but with virtio for disk

qemu is a C program that faithfully implements a RISC-V processor

4 Lecture 3

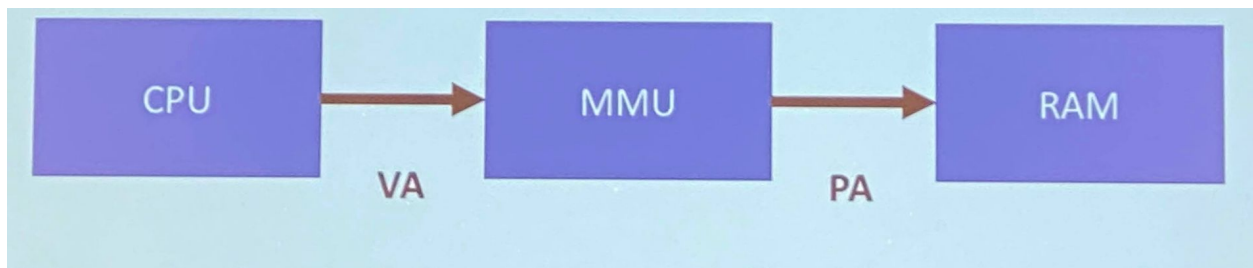
Outline

- Address spaces
- Risc-V paging hardware
- Case study: xv6



4.1 Solution: Introduce a level of indirection

- Plan: software can only read and write to virtual memory
- Only kernel can program MMU
- MMU has a **page table** that maps virtual addresses to physical
- Some virtual addresses restricted to kernel-only



4.2 Virtual Memory in Risc-V

- Supports different addressing modes:
 - Sv32, Sv39, Sv48 → number of virtual addr bits

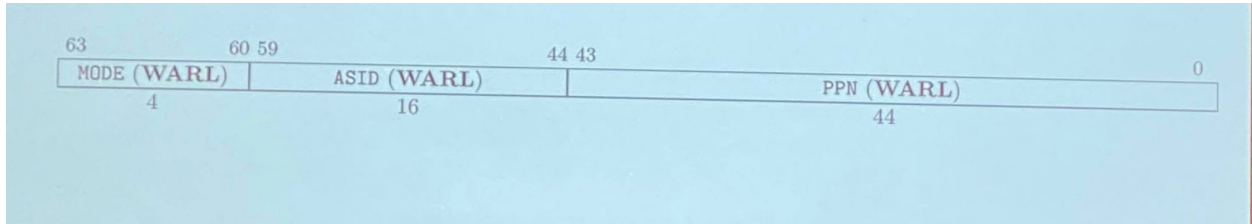
- This class will use Sv39 (3-level page table)

- **SFENCE.VMA** tells CPU to check page tbl updates
- **satp** register points to page root (set w/CSRW)

It tells the MMU where the page table is, in physical memory using physical addr

This is the only thing that is in physical addr since everything else uses the page table to map virtual addr

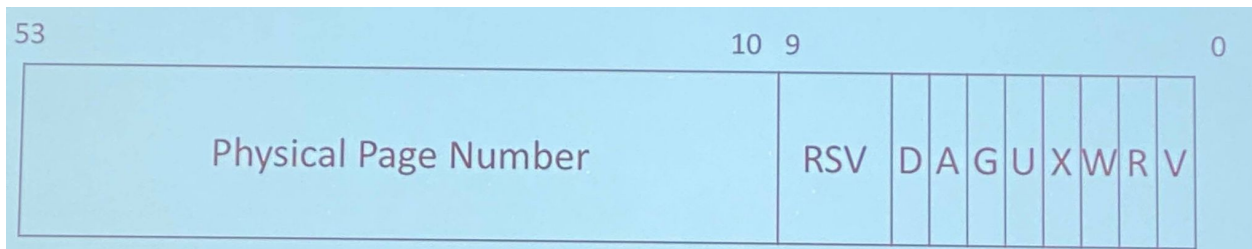
Some CPUs don't need as much address bits or need alot of address bits so the various modes handles this.



4.3 Virtual Memory in Risc-V(Sv39)

Virtual addresses are divided into 4-KB "pages" Virtual addresses for Sv39 are split into a 27-bit page number and a 12-bit offset.

4.4 Page table entries (PTE)



- **Physical page number:** Identifies 44-bit physical page location; MMU replaces virtual bits with these physical bits
- **U:** If set, userspace can access this virtual address
- **W:** If set, the CPU can write to this virtual address
- **V:** If set, an entry for this virtual address exists
- **RSV:** Ignored by MMU

V-bit is very crucial for the lab since it tell sth eMMU is the entry is present or not. If it is cleared, if the address is accessed throw a fault and ignore everything in this address.

RSV-bits are used by the kernel to store meta data on the page table entry. It primarily reads this when there is a fault. Since it is really common for Oses to read meta data during a fault, it is much easier to store the meta data into the page table instead of using two. Real kernels such as Linux will also have a separate data structure.

4.5 Strawman: Store PTEs in an array

GET_PTE(va) = &ptes[va << 12]

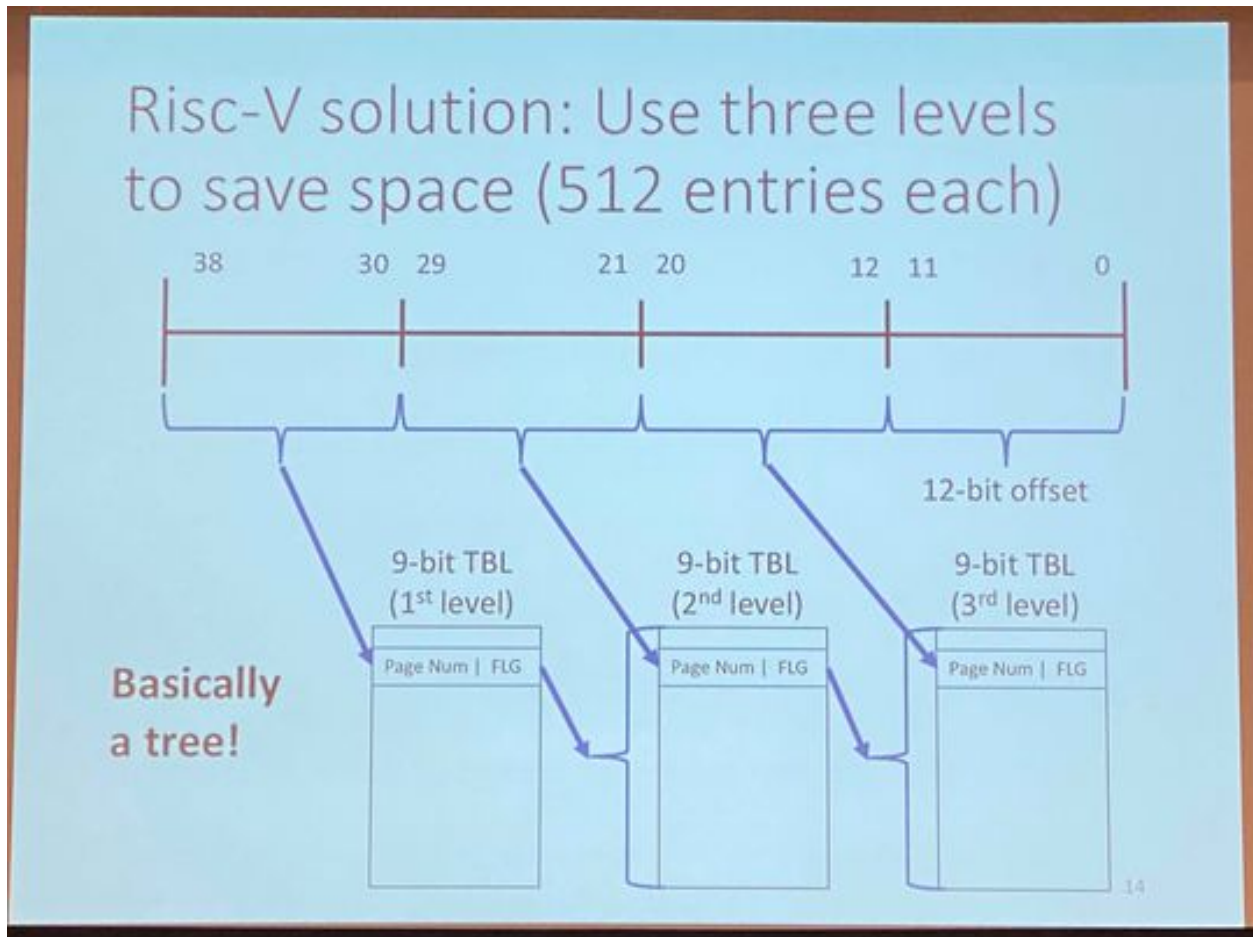
How large is the array?

There are $2^{27} * 8$ bytes = 1 Gigabyte.

This is not a reasonable implementation since it takes up so much space.

4.6 Risc-V solution: Use three levels to save space

Use 3-leveled table each with 9-bit tables that point to the next table. It basically is a tree with 512 entries each.



4.7 What about recursive mapping?

What would happen if in a 9-bit page table, there is an entry that points to itself?
It would just return a page table entry at the very end.

The page table entries are sparse so not everything may be filled up.

4.8 How do we program the MMU?

- satp register is a pointer to the current page table
- Hardware walks page table tree to find PTEs
- Recently used PTEs cached in TLB

The CPU's MMU calls the satp to find the first level 9-bit table and then walks through all three levels. Using caching, modern CPUs stores copies of the most recent data in the TLB(translation localized buffer) to prevent the CPU from incurring the big (4x) cost of reading all the page tables.

4.9 More about flags

X	W	R	Meaning
0	0	0	Pointer to next level of page table.
0	0	1	Read-only page.
0	1	0	<i>Reserved for future use.</i>
0	1	1	Read-write page.
1	0	0	Execute-only page.
1	0	1	Read-execute page.
1	1	0	<i>Reserved for future use.</i>
1	1	1	Read-write-execute page.

- If U is cleared, only the kernel can access
- What happens if flag permission is violated?
 - We get a page fault!

4.10 Handling stale entries in TLB

- Since TLB is a cache, might contain stale entries
 - When PTEs are removed
 - PTE flags change
 - When switching page roots (satp)
- Risc-V provides instruction to flush TLB
 - **LFENCE.VMA**: flushes entire TLB or specific VA
- **G** flag prevents FLB flushes of a PTE

In xv6, we simply flush all of the TLB.

4.11 What about segmentation?

- Base and bounds
- Why not use this instead of paging?
- Paging seems to be favored but debate is ongoing
- Really powerful features enabled by paging

Provides really powerful memory isolation.
Way more computationally efficient.
Could end up with a lot of unused space - very prone to fragmentation.
Needs a lot of memory space to grow

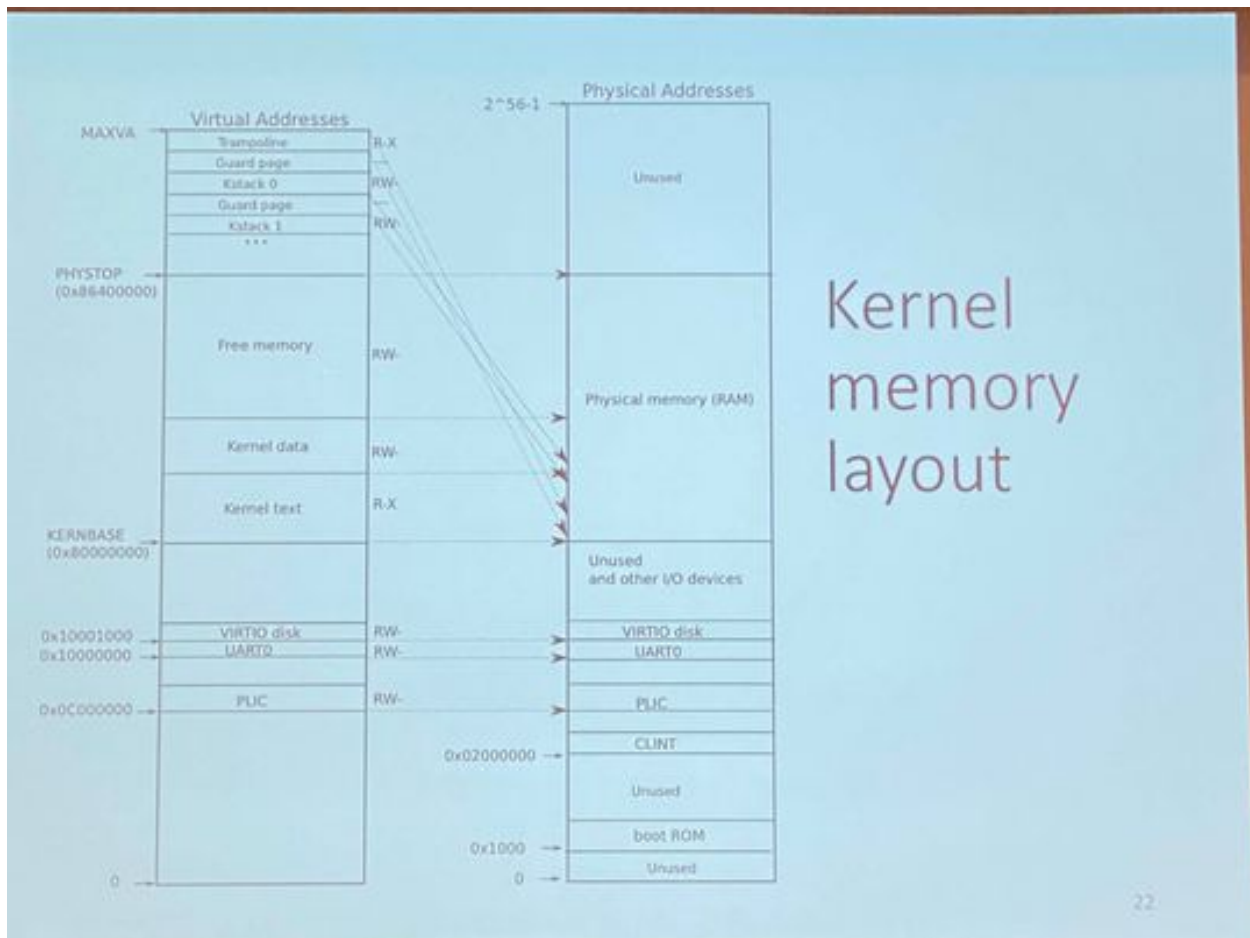
4.12 Why use virtual memory while in kernel?

- Isolation benefits for userspace are clear
- Practical Reasons
 - Hard (expensive) to turn off paging for each system call
 - Hard to deal with system call arguments that straddle page boundaries
Takes memory pointer as an argument such as a path name. If it is long it could be spread over multiple pages. It could be virtually continuous but physically not continuous to the kernel without a translation.
 - Difficult for kernel to support many different hardware physical address layouts
- Reducing fragmentation
 - The kernel needs to allocate memory too

4.13 Paging is powerful

- Copy-on-write: Focus on upcoming lab
- Enables many use cases
 - Lazy memory allocation
 - Runtime system optimizations
 - Topic of upcoming lecture

4.14 Kernel Memory Layout



The trampoline is at the top of the virtual addresses and is exactly mapped in the same location across two different page tables allows the code to run continuously even when switching cpu configurations.

The kernel is laid out on purpose to 1 to 1 map stuff like RAM.

There are some addresses that are hardware addresses that borrow the addresses for communication. Can be seen at the bottom.

4.15 Processes in xv6

- Each process has its own page table
- Set satp to new page table root when switching processes
- Kernel's trampoline mapping exists in same location in each page table

The kernel has its own page table.

4.16 How do processes allocate memory?

xv6 uses `void * sbrk(int n)` to map more memory. The `n == 0` means nothing happens. Negative is free memory. Positive is add memory.