

第 2 讲：操作系统与系统结构和程序设计语言

第一节：从 OS 角度看计算机系统

向勇、陈渝

清华大学计算机系

xyong,yuchen@tsinghua.edu.cn

2020 年 5 月 5 日

① 第一节：从 OS 角度看计算机系统

- 隔离
- 虚拟内存
- 特权模式/中断

再看计算机系统

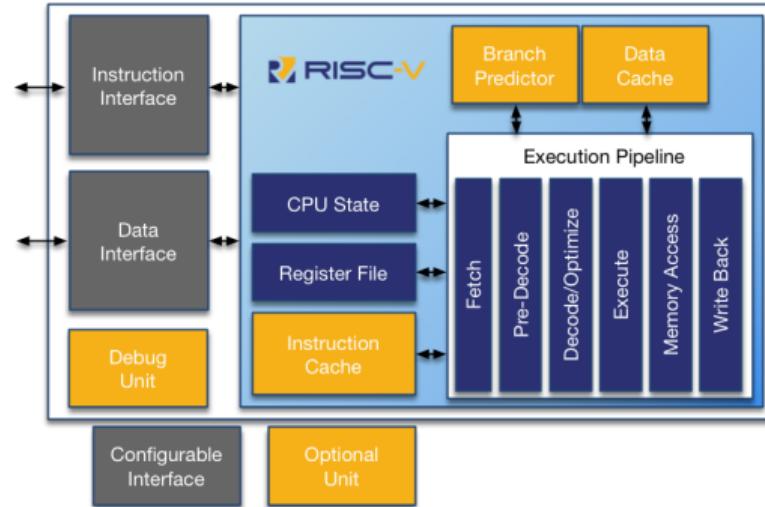


图: 再看计算机系统 – 从 OS 角度

- 程序调用 `ssize_t read(int fd, void *buf, size_t count);` 会发生什么？
- 我们可以在应用程序中直接调用内核的函数吗？
- 我们可以在内核中使用应用程序普通的函数调用吗？
- 程序调用的特征
 - 好处：执行很快；
 - 好处：灵活-易于传递和返回复杂数据类型；
 - 好处：程序员熟悉的机制，...
 - 坏处：应用程序不可靠，可能有恶意，有崩溃的风险

隔离：什么是隔离？

- 强制隔离以避免对整个系统的可用性/可靠性/安全影响
- 运行的程序通常是是隔离的单元
- 防止程序 X 破坏或监视程序 Y
 - 读/写内存，使用 100% 的 CPU，更改文件描述符
- 防止进程干扰操作系统
- 防止恶意程序、病毒、木马和 bug
 - 错误的过程可能会试图欺骗硬件或内核

隔离：主要的隔离方法？

- 地址空间 address spaces
 - 一个程序仅寻址其自己的内存
 - 每个程序若无许可，则无法访问不属于自己的内存
- CPU 硬件中的特权模式/中断机制
 - 防止应用程序访问设备和敏感的 CPU 寄存器
 - 例如地址空间配置寄存器
 - 例如打断一直占用 CPU 的应用程序

虚拟内存

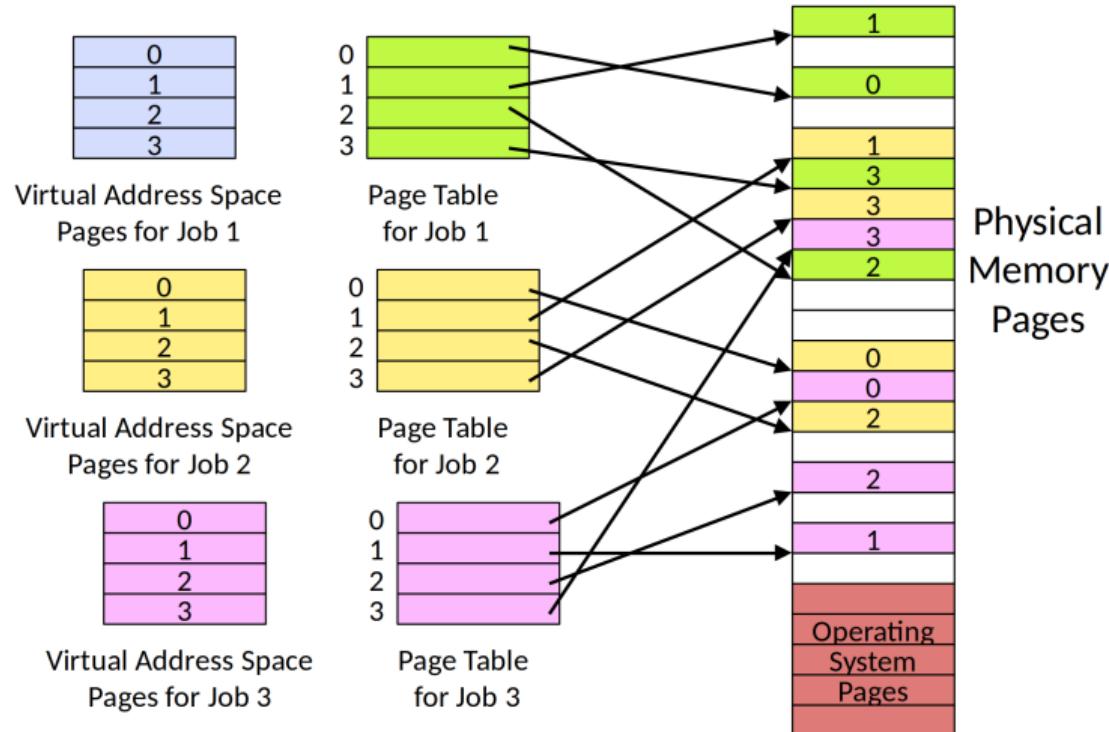


图: 虚拟内存

虚拟内存

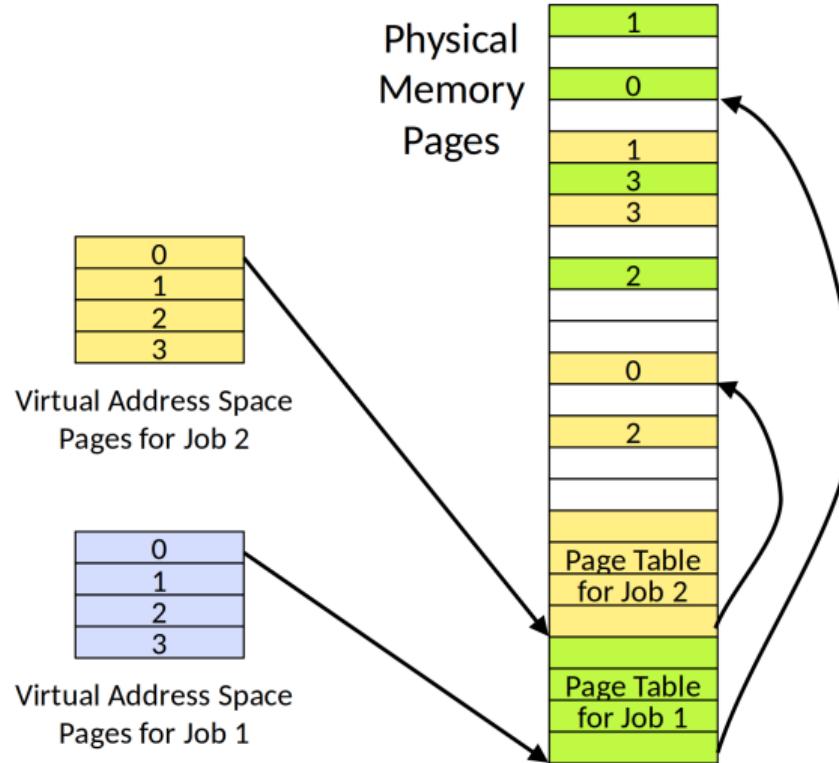


图: 页表

虚拟内存

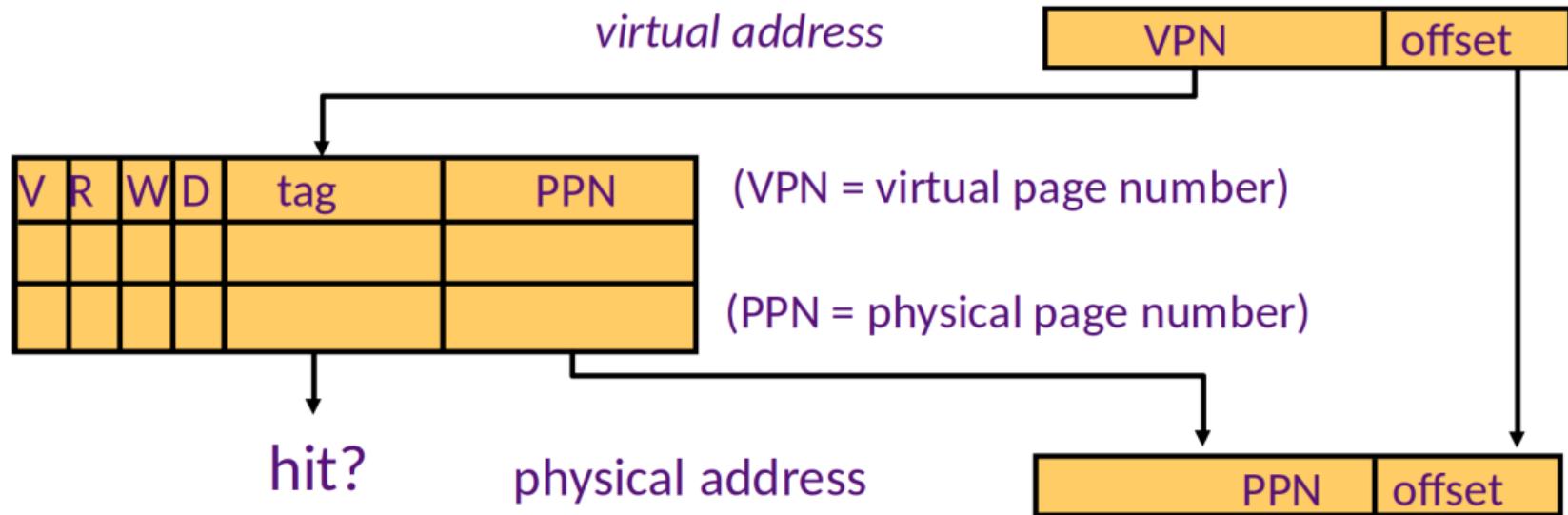


图: TLB

虚拟内存

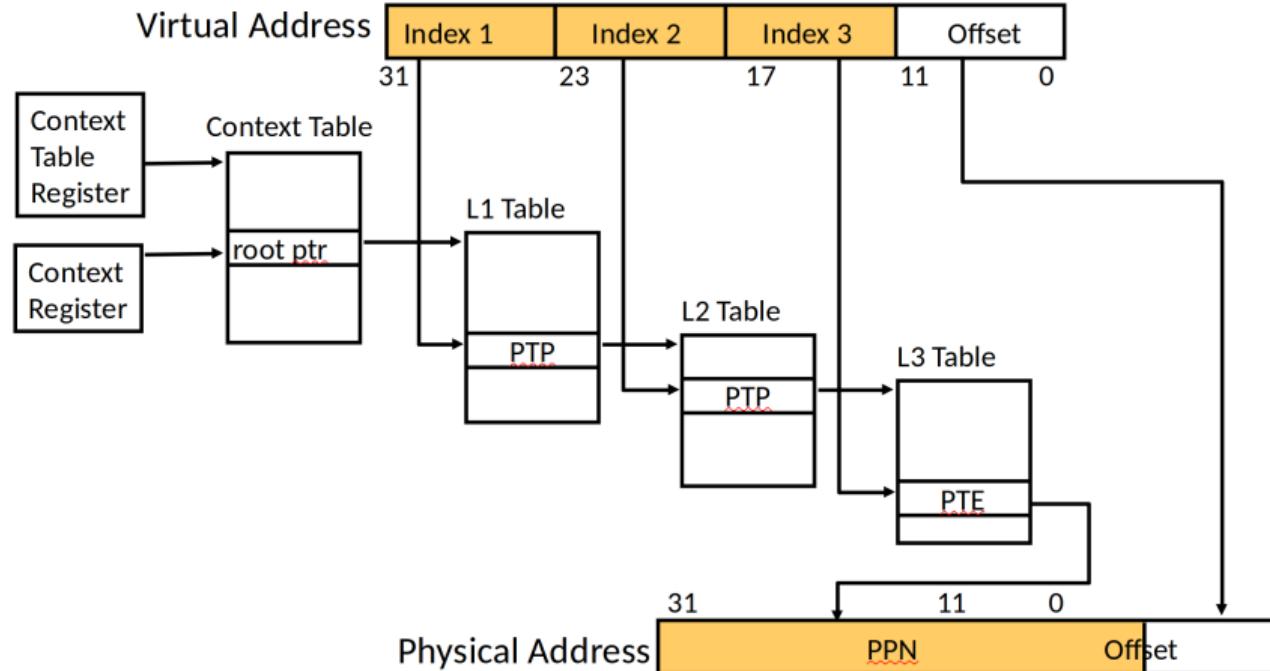


图: MMU 处理 TLB Missing

虚拟内存

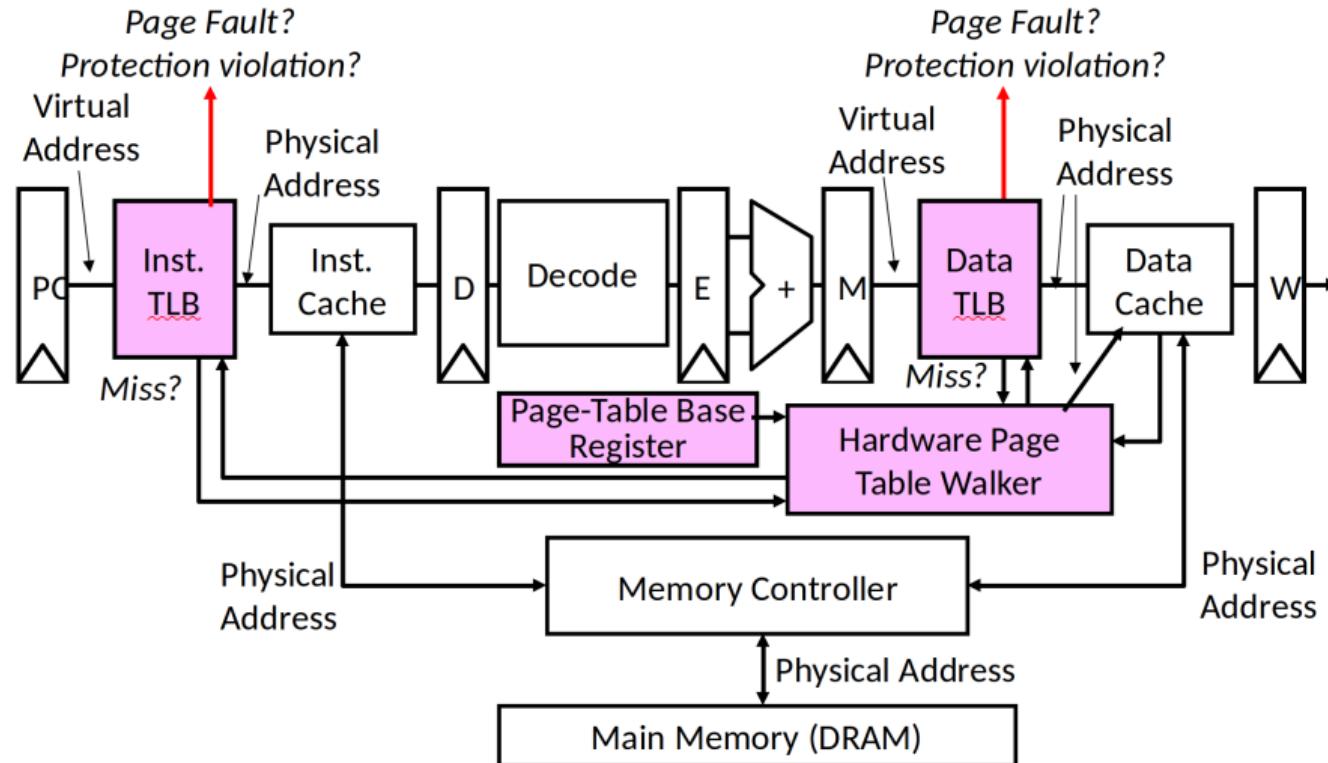


图: 带 MMU/TLB 的计算机系统

特权模式

- CPU 硬件支持不同的特权模式
 - Kernel Mode vs User Mode
 - Kernel Mode 可以执行 User Mode 无法执行的特权操作
 - 访问外设
 - 配置地址空间（虚拟内存）
 - 读/写特殊系统级寄存器
 - OS kernel 运行在 Kernel Mode
 - 应用程序运行在 User Mode
 - 每个重要的微处理器都有类似的用户/内核模式标志

中断机制

- CPU 硬件支持中断/异常的处理
- 中断是异步发生，是来自处理器外部的 I/O 设备的信号的结果。
 - 硬件中断不是由任何一条专门的 CPU 指令造成，从这个意义上它是异步的。
- 硬件中断的异常处理程序通常称为中断处理程序（interrupt handle）
 - I/O 设备通过向处理器芯片的一个引脚发信号，并将异常号放到系统总线上，以触发中断；
 - 在当前指令执行完后，处理器从系统总线读取异常号，保存现场，切换到 Kernel Mode；
 - 调用中断处理程序，当中断处理程序完成后，它将控制返回给下一条本来要执行的指令。
- Timer 可以稳定定时地产生中断
 - 防止应用程序死占着 CPU 不放
 - 让 OS kernel 能周期性地进行资源管理

第 2 讲：操作系统与系统结构和程序设计语言

第二节：从 OS 角度看 RISC-V

向勇、陈渝

清华大学计算机系

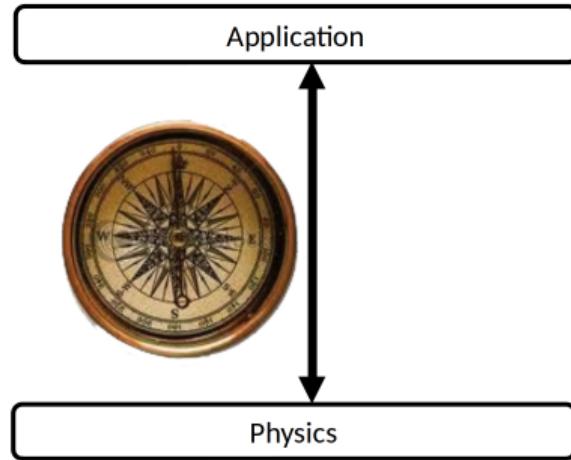
xyong,yuchen@tsinghua.edu.cn

2020 年 5 月 5 日

① 第二节：从 OS 角度看 RISC-V

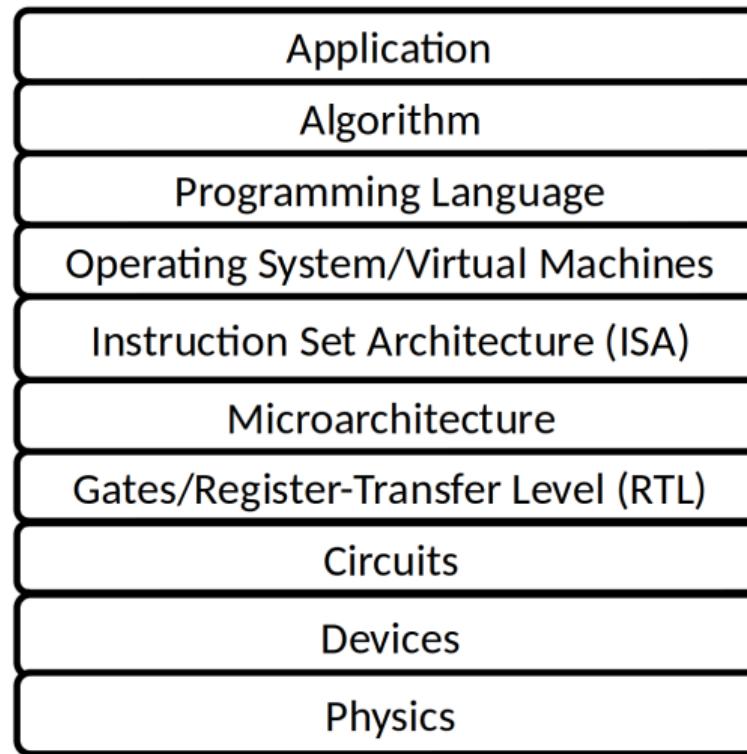
- Why RISC-V
- RISC-V 特权架构

Why RISC-V: 计算机系统 [CS-152 Berkeley]

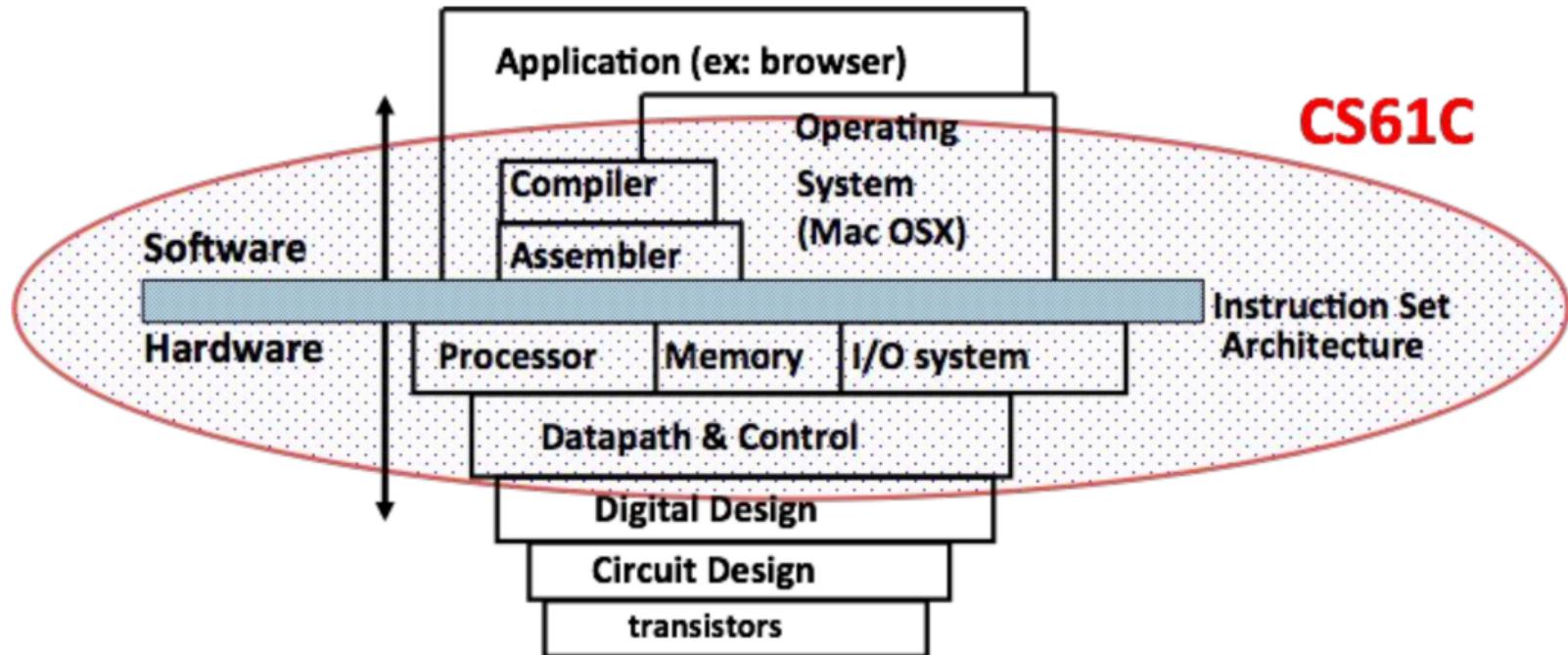


广义的定义, 计算机系统 (computer architecture) 是一种抽象层次的设计, 用于实现可有效使用现有制造技术的信息处理应用。 [CS-152 Berkeley]

Why RISC-V: 计算机系统的抽象层次 [CS-152 Berkeley]



Why RISC-V: 软硬件接口 [CS-152 Berkeley]

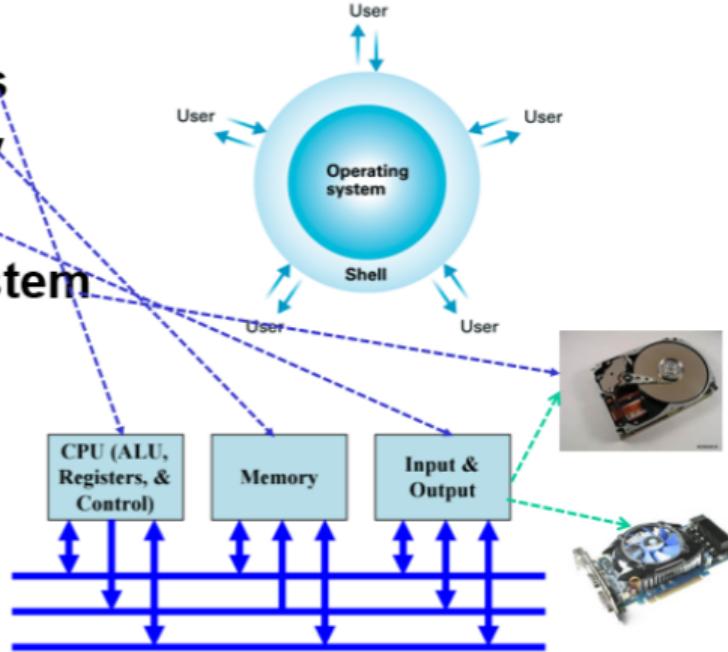


Why RISC-V: os 与体系结构的关系

Kernel

- Process
- Memory
- Device
- File System
- ...

- Memory
- ALU
- Control Unit
- I/O System



Why RISC-V: 主流 CPU 指令集比较 [Waterman2017]



x86

Designer	Intel, AMD
Bits	16-bit, 32-bit and 64-bit
Introduced	1978 (16-bit), 1985 (32-bit), 2003 (64-bit)
Design	CISC
Type	Register-memory
Encoding	Variable (1 to 15 bytes)
Endianness	Little



ARM architectures

Designer	ARM Holdings
Bits	32-bit, 64-bit
Introduced	1985; 31 years ago
Design	RISC
Type	Register-Register
Encoding	AArch64/A64 and AArch32/A32 use 32-bit instructions, T32 (Thumb-2) uses mixed 16- and 32-bit instructions. ARMv7 user-space compatibility ^[1]
Endianness	Bi (little as default)

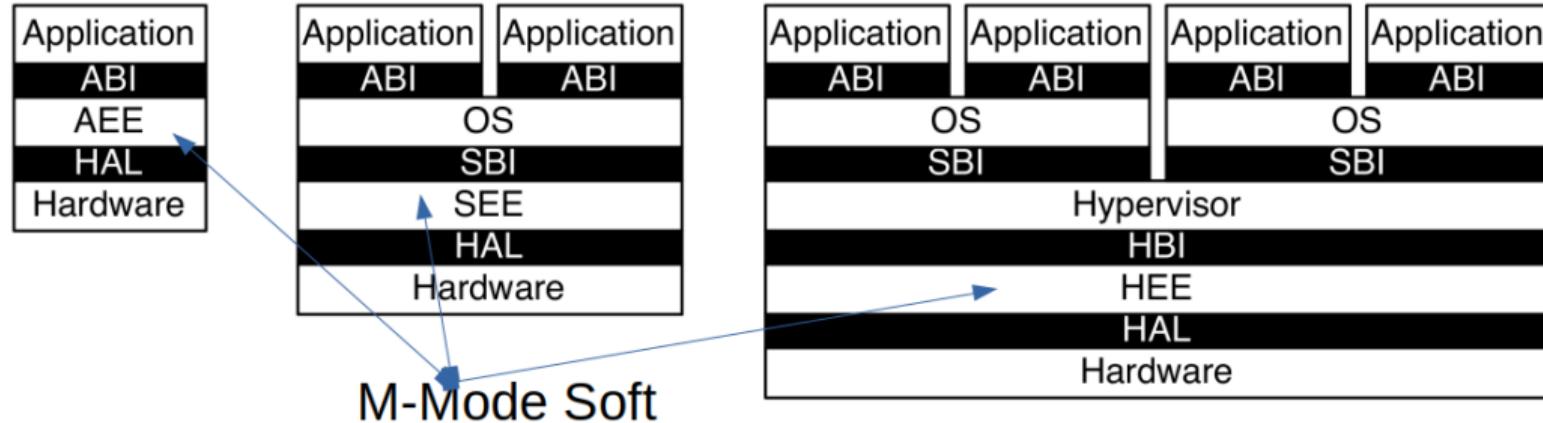


RISC-V

Designer	University of California, Berkeley
Bits	32, 64, 128
Introduced	2010
Version	2.2
Design	RISC
Type	Load-store
Encoding	Variable
Branching	Compare-and-branch
Endianness	Little

ISA	Pages	Words	Hours to read	Weeks to read
RISC-V	236	76,702	6	0.2
ARM-32	2736	895,032	79	1.9
x86-32	2198	2,186,259	182	4.5

RISC-V 特权架构：隔离



- 不同软件层有清晰的硬件隔离
- AEE: Application Execution Environment
- ABI: Application Binary Interface
- **MODE – U: User | S: Supervisor | H: Hypervisor | M: Machine**

RISC-V 特权模式架构：模式组合

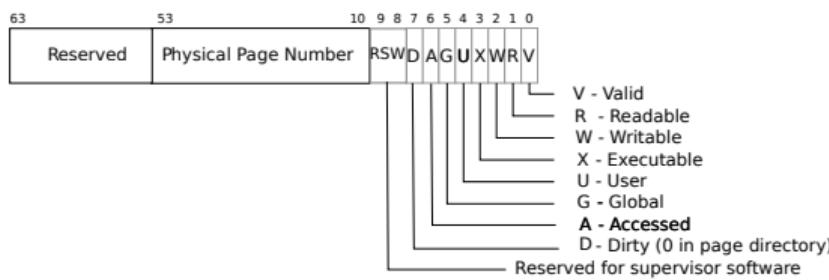
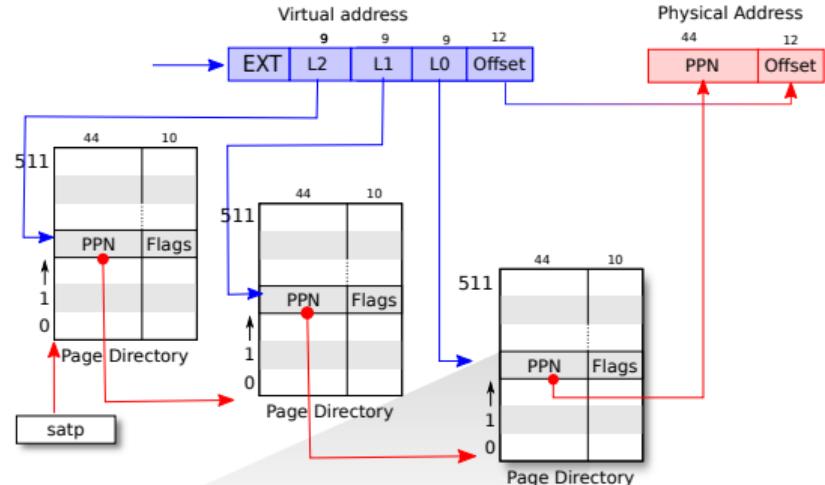
0	00	User/Application	U
1	01	Supervisor	S
2	10	Hypervisor	H
3	11	Machine	M

- M, S, U for systems running Unix-like general operating systems

RISC-V 特权模式架构：控制状态寄存器

- 设置 CSR(控制状态寄存器) 实现隔离
 - 防止应用程序访问设备和敏感的 CPU 寄存器
 - 例如地址空间配置寄存器
- 强制隔离以避免对整个系统的可用性/可靠性/安全影响
 - 运行的程序通常是隔离的单元
 - 防止恶意程序、病毒、木马破坏或监视应用程序或干扰操作系统
 - 读/写内存: mstatus/satp CSR
 - 使用 100% 的 CPU: mstatus/stvec CSR
 - mstatus/satp/stvec CSR 页表异常处理

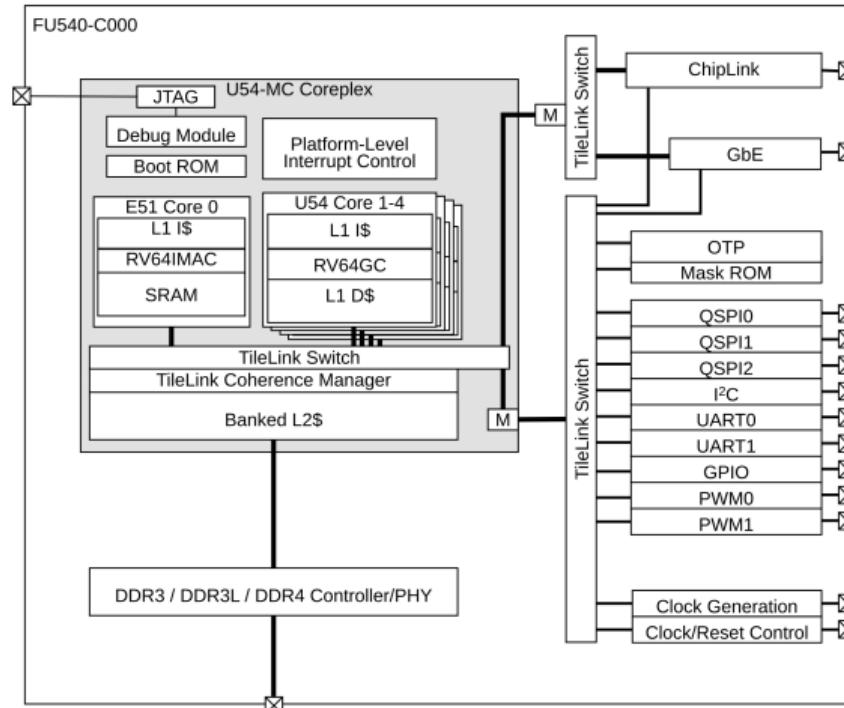
RISC-V 虚拟内存



- 有虚拟内存的好处/坏处
 - 灵活的不连续内存分配
 - 多个运行程序的地址空间相互隔离/共享
 - 进一步隔离应用程序与 OS 内核
 - 多了访问页表的开销

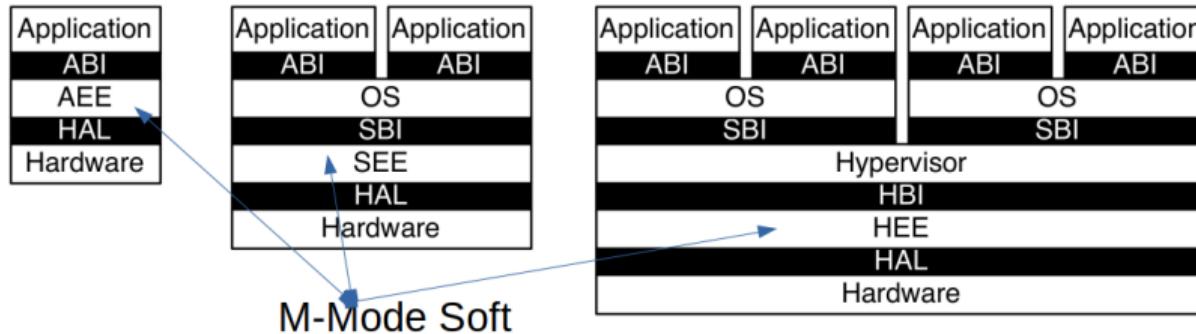
RISC-V 中断机制

- 中断是异步发生，是来自处理器外部的 I/O 设备的信号的结果。
- Timer 可以稳定定时地产生中断
 - 防止应用程序死占着 CPU 不放，让 OS Kernel 能得到执行权...



RISC-V 中断机制

- 中断是异步发生，是来自处理器外部的 I/O 设备的信号的结果。
- Timer 可以稳定定时地产生中断
 - 防止应用程序死占着 CPU 不放，让 OS Kernel 能得到执行权...
 - 由高特权模式下的软件获得 CPU 控制权
 - 也可由高特权模式下的软件授权低特权模式软件处理中断



第 2 讲：操作系统与系统结构和程序设计语言

第三节：Rust 语言与系统编程

向勇、陈渝

清华大学计算机系

xyong,yuchen@tsinghua.edu.cn

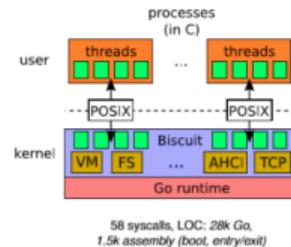
2020 年 5 月 5 日

① 第三节：Rust 语言与系统编程

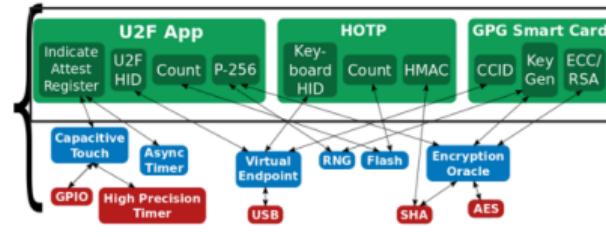
- 系统编程语言
- 高级语言编译到机器指令

系统编程语言：对当前编程语言的理解

- 系统编程语言 (system language) 定义现在似乎没有特别严谨和一致的定义
- 系统编程语言用于构建控制底层计算机硬件的软件系统，并提供由用于构建应用程序和服务的更高级应用程序编程语言使用的软件平台。
- 开发操作系统的系统编程语言其实很多；还离不开汇编语言。



MIT 用 Go 语言开发了 Biscuit OS
缺少对实际应用的优化，OSDI'2018



Stanford 用 RUST 语言开发了 tock OS
缺少对面向通用系统应用的支持，SOSP'2017



系统编程语言： Why Rust

- CS140e: Stanford 实验性课程, Rust OS for Raspi3
- RVirt: MIT RISC-V Hypervisor
- Writing an OS in Rust —BlogOS: 详尽的 Rust OS 教程
- 产业界: 蚂蚁金服: Occlum – Facebook: Libra

系统编程语言：Rust 的主要特性

- 内存 + 线程安全
- 高级语言特性
- 成熟的工具链
- 友好的助教 + 社区（OS 示例代码 + 文档）生态
- 学习 Rust 的入门门槛比较高（认清你自己，量力而行）

高级语言编译到机器指令： C/Rust 代码

```
//C CODE
int sum_to(int n) {
    int acc = 0;
    for (int i = 0; i < n; i++) {
        acc += i;
    }
    return acc;
}
```

```
//Rust CODE
fn sum_to(n:i32)->i32 {
    let mut acc = 0;
    for i in 0..n {
        acc += i;
    }
    return acc
}
```

高级语言编译到机器指令：汇编代码

```
# sum_to(n)
# expects argument in a0
# returns result in a0
sum_to:
    mv t0, a0          # t0 <- a0
    li a0, 0           # a0 <- 0
loop:
    add a0, a0, t0      # a0 <- a0 + t0
    addi t0, t0, -1     # t0 <- t0 - 1
    bnez t0, loop       # if t0 != 0: pc <- loop
ret
```

- 有限的抽象
 - 无类型的位置参数 – 没有局部变量 – 仅寄存器

高级语言编译到机器指令：RISC-V 寄存器

表: RISC-V 寄存器描述

Register	ABI Name	Description	Saver
x0	zero	Hard-wired zero	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5-7	t0-2	Temporaries	Caller
x8	s0/fp	Saved register/frame pointer	Callee
x9	s1	Saved register	Callee
x10-11	a0-1	Function arguments/return values	Caller
x12-17	a2-7	Function arguments	Caller
x18-27	s2-11	Saved registers	Callee
x28-31	t3-6	Temporaries	Caller

高级语言编译到机器指令：机器指令

- 机器甚至看不到汇编代码
- 查看机器指令的二进制编码
 - 每条指令：16 位或 32 位
 - 例如 ‘mv t0, a0’ 编码为 0x82aa
 - 从 asm 不太完全一对一编码 (但是很接近)
- 另一个函数将如何调用 sum_to?

```
1 main:  
2   li  a0, 10          # a0 <- 10  
3   call sum_to
```

高级语言编译到机器指令：函数调用

- 函数调用的语义是什么？

```
call label :=  
## ra <- address of next instruction  
ra <- pc + 4      ;  
## jump to label  
pc <- label      ;
```

- 机器不理解标签
- 换为相对于 PC 的跳转或绝对跳转

高级语言编译到机器指令：函数返回

- 函数返回 (return) 的语义是什么??

```
ret := pc <- ra
```

看看汇编代码：demo1.S

```
(gdb) file user/_demo1
```

```
(gdb) break main
```

```
(gdb) continue
```

```
(gdb) layout split
```

```
(gdb) stepi
```

```
(gdb) info registers
```

```
(gdb) p $a0
```

```
(gdb) advance 18
```

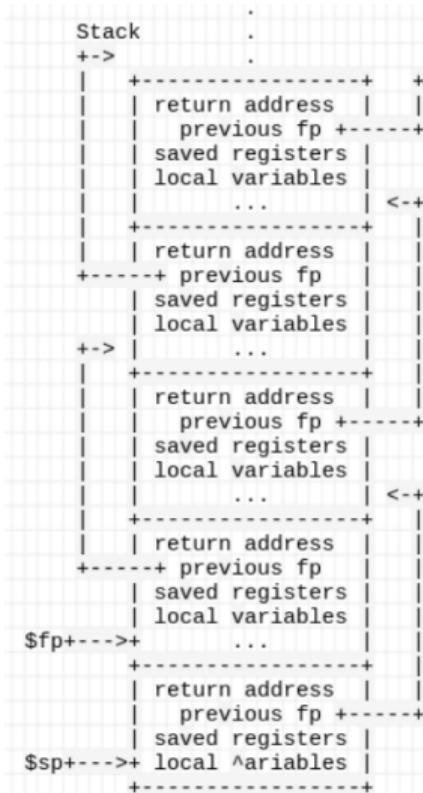
```
(gdb) si
```

```
(gdb) p $a0
```

高级语言编译到机器指令：调用约定

- 如何传递参数?
 - a0, a1, ..., a7, 放在堆栈上
- 值如何返回?
 - a0, a1
- 谁保存寄存器?
 - 指定为已保存的呼叫者或被呼叫者
 - ra 可以是保存被呼叫者的寄存器吗
- 汇编代码应遵循此约定
- GCC 生成的 C 代码遵循此约定
- RUST 代码遵循此约定
- 这意味着各种代码之间都可以互操作

高级语言编译到机器指令：理解 stack



第 2 讲：操作系统与系统结构和程序设计语言

第四节：RISC-V CPU 启动

向勇、陈渝

清华大学计算机系

xyong,yuchen@tsinghua.edu.cn

2020 年 5 月 5 日



- RISC-V

- CPU: RV64 ISA with MMU/TLB
- MEM: ROM, RAM, IO-MEM, etc
- I/O: Timer, UART
- Interrupt: PLIC(Platform-Level Interrupt Controller) – Device
- Interrupt: CLINT(Core Local Interruptor) – Timer, IPI

RISC-V CPU



- RISC-V CPU 启动过程
 - 初始化 CPU/Regs
 - 初始化内存
 - 初始化基本外设
 - 执行 ROM 中固化的代码
- 出处：<https://github.com/qemu/qemu>

```
/* Default Reset Vector address */
#define DEFAULT_RSTVEC      0x1000
static void riscv_any_cpu_init(Object *obj)
{
    CPURISCVState *env = &RISCV_CPU(obj)->env;
    set_misa(env, RVXLEN | RVI | RVM | RVA | RVF | RVD | RVC | RVU);
    set_priv_version(env, PRIV_VERSION_1_11_0);
    set_resetvec(env, DEFAULT_RSTVEC);
}

static void riscv_cpu_reset(CPUState *cs)
{
    RISCVCPU *cpu = RISCV_CPU(cs);
    RISCVCPUClass *mcc = RISCV_CPU_GET_CLASS(cpu);
    CPURISCVState *env = &cpu->env;

    mcc->parent_reset(cs);
#ifndef CONFIG_USER_ONLY
    env->priv = PRV_M;
    env->mstatus &= ~(MSTATUS_MIE | MSTATUS_MPRV);
    env->mcause = 0;
    env->pc = env->resetvec;
#endif
    cs->exception_index = EXCP_NONE;
    env->load_res = -1;
    set_default_nan_mode( val: 1, &env->fp_status);
}
```

RISC-V CPU



- RISC-V CPU 启动过程-初始化内存

```
static const struct MemmapEntry {
    hwaddr base;
    hwaddr size;
} virt_memmap[] = {
    [VIRT_DEBUG] = { .base: 0x0, .size: 0x100 },
    [VIRT_MROM] = { .base: 0x1000, .size: 0x11000 },
    [VIRT_TEST] = { .base: 0x100000, .size: 0x1000 },
    [VIRT_CLINT] = { .base: 0x2000000, .size: 0x10000 },
    [VIRT_PLIC] = { .base: 0xc000000, .size: 0x4000000 },
    [VIRT_UART0] = { .base: 0x10000000, .size: 0x100 },
    [VIRT_VIRTIO] = { .base: 0x10001000, .size: 0x1000 },
    [VIRT_FLASH] = { .base: 0x20000000, .size: 0x4000000 },
    [VIRT_DRAM] = { .base: 0x80000000, .size: 0x0 },
    [VIRT_PCIE_MMIO] = { .base: 0x40000000, .size: 0x40000000 },
    [VIRT_PCIE_PIO] = { .base: 0x03000000, .size: 0x00010000 },
    [VIRT_PCIE_ECAM] = { .base: 0x30000000, .size: 0x10000000 },
};
```

```
static void riscv_virt_board_init(MachineState *machine)
{
    const struct MemmapEntry *memmap = virt_memmap;
    RISCVVirtState *s = RISCV_VIRT_MACHINE(machine);
    MemoryRegion *system_memory = get_system_memory();
    MemoryRegion *main_mem = g_new(MemoryRegion, n_structs: 1);
    MemoryRegion *mask_rom = g_new(MemoryRegion, n_structs: 1);
    char *PLIC_HART_CONFIG;
    size_t plic_hart_config_len;
    target_ulong start_addr = memmap[VIRT_DRAM].base;

    .....
    sifive_clint_create(memmap[VIRT_CLINT].base,
        memmap[VIRT_CLINT].size, smp_cpus,
        SIP_BASE: SIFIVE_SIP_BASE, timecmp_base: SIFIVE_TIMECMP_BASE, time_base: SIFIVE_TIME_BASE);

    .....
    serial_mm_init(system_memory, memmap[VIRT_UART0].base,
        it_shift: 0, irq: qdev_get_gpio_in(DEVICE(obj: s->plic), n: UART0_IRQ), baudbase: 399193,
        chr: serial_hd(i: 0), end: DEVICE_LITTLE_ENDIAN);

    virt_flash_create(s);
```

RISC-V CPU 启动过程-ROM 初始化代码

```
[VIRT_MROM] = { .base: 0x1000, .size: 0x11000 },
    uint32_t reset_vec[8] = {
        0x00000297, /* 1: auipc t0, %pcrel_hi(dtb) */
        0x02028593, /*      addi   a1, t0, %pcrel_lo(1b) */
        0xf1402573, /*      csrr   a0, mhartid */
#if defined(TARGET_RISCV32)
        0x0182a283, /*      lw     t0, 24(t0) */
#elif defined(TARGET_RISCV64)
        0x0182b283, /*      ld     t0, 24(t0) */
#endif
        0x00028067, /*      jr     t0 */
        0x00000000,
        start_addr, /* start: .dword */
        0x00000000, /* dtb: */
    };
    target_ulong start_addr = memmap[VIRT_DRAM].base;

```

[VIRT_DRAM] = { .base: 0x80000000, .size: 0x0 },