01: INTRO

操作系统：在硬件和应用之间的软件层

“操作系统”并没有严格的唯一定义，是一个相对概念：相对“应用”而言，操作系统也可包含运行在用户态的框架（framework）
例如：SSL 库、Dalvik（Java 虚拟机）是否属于操作系统?

- 对 Android 来说，属于操作系统框架层；App 属于应用
- 对 Linux 来说，不属于操作系统；hello 属于应用

我们课程中的操作系统：以 hello 作为典型应用

> “操作系统是管理硬件资源、控制程序运行、改善人机界面和为应用软件提供支持的一种系统软件。” 《计算机百科全书(第2版)》

> “操作系统将有限的、离散的资源，高效地抽象为无限的、连续的资源。” 《操作系统：原理与实现》

02: ASM-ARM

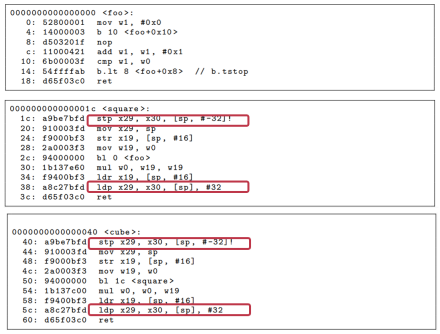
基地址寻址

- $[rb]$
- 基地址加偏移量模式（偏移量寻址）$[rb, \text{offset}]$
- 前索引寻址（寻址操作前更新基地址）
  $[rb, \text{offset}]! = rb+ = Offset;\ \ M[rb]$
- 后索引寻址（寻址操作后更新基地址）$[rb], \text{offset} = \ \ M[rb]; rb+ = \text{Offset}$

条件码（Conditional Code, CC）

条件码是一组标志位的统称，条件码由 PSTATE 寄存器维护 N（Negative）、Z（Zero）、C（Carry）、V（Overflow）条件码保留之前相关指令的执行状态，这种指令包括：带有 s 后缀的算术或逻辑运算指令（如 subs、adds）比较指令 cmp：操作数之差；例如 cmp x0, x1 cmn：操作数之和；例如 cmn x0, x1 tst ：操作数相与；例如 tst x0, x1
第一类：通过 s 后缀数据处理指令隐式设置 adds Rd, Rn, Op2 等价于 C 语言中的: t = a + b, C：当运算产生进位时被设置, Z：当 t 为 0 时被设置, N：当 t 小于 0 时被设置, V：当运算产生有符号溢出时被设置 (a>0 && b>0 && t<0) || (a<0 && b<0 && t>=0)
第二类：通过比较指令 cmp 显式设置 cmp src1, src2, 计算 src1 - src2，但不存储结果，只改变条件码 C：当减法产生借位时被设置 Z：当两个操作数相等时被设置 N：当 Src1 小于 Src2 时被设置 V：当运算产生有符号溢出时被设置 (a>0 && b<0 && (a-b)<0) || (a<0 && b>0 && (a-b)>0)

- N: 当用两个补码表示的带符号数进行运算时，N=1 表示运算的结果为负数；N=0 表示运算的结果为正数或零.
- Z: Z=1 表示运算的结果为零，Z=0 表示运算的结果非零。
- C: 可以有 4 种方法设置 C 的值加法运算（包括 CMN）：当运算结果产生了进位时（无符号数溢出），C=1，否则 C=0。减法运算（包括 CMP）：当运算时产生了借位时（无符号数溢出），C=0，否则 C=1。对于包含移位操作的非加/减运算指令，C 为移出值的最后一位。对于其它的非加/减运算指令，C 的值通常不会改变。
- V: 可以有 2 种方法设置 V 的值：对于加减法运算指令，当操作数和运算结果为二进制的补码表示的带符号数时，V=1 表示符号位溢出对于其它的非加/减运算指令，V 的值通常不会改变。



注意-1：x30就是LR寄存器

注意-2：同时保存了x29

函数栈帧

栈桢：函数在运行期间使用的一段内存
生命周期：从被调用到返回前
作用：存放其局部状态，包括：存放返回地址，存放上一个栈桢的位置存放局部变量，…

通过寄存器调用参数

调用者使用 x0 - x7 寄存器传递前 8 个参数，被调用者使用 x0 寄存器传递返回值

寄存器使用约定

调用者保存的寄存器包括 X9 - X15
调用者在调用前按需（仅考虑自己是否需要）进行保存，调用者在被调用者返回后恢复这些寄存器的值，被调用者可以随意使用这些寄存器调用后的值可能发生改变.
被调用者保存的寄存器包括 X19 - X28
被调用者在使用前进行保存，被调用者在返回前进行恢复，调用者视角：这些寄存器的值在函数调用前后不会改变

实例：保存寄存器

```c
int square(int n)
{
  return n * n;
}

int cube(int n)
{
  return n * square(n);
}
```

```
square:
    mul     w0, w0, w0
    ret
cube:
    stp     x29, x30, [sp, -32]!
    mov     x29, sp
    str     x19, [sp, 16]
    mov     w19, w0
    bl      square
    mul     w0, w0, w19
    ldr     x19, [sp, 16]
    ldp     x29, x30, [sp], 32
    ret
```

cube作为被调用者（比如main函数调用cube），在使用x19前需要在栈上保存它

问1：若使用调用者保存的寄存器（如x9），是否能够避免保存?
问2：x19（w19）是用来保存x0（w0）的值，为什么不直接把x0存在栈上?

优化选项 Og

```
<cube>:
    stp     x29, x30, [sp, #-32]!
    mov     x29, sp
    str     x19, [sp, #16]
    mov     w19, w0
    bl      0 <square>
    mul     w0, w0, w19
    ldr     x19, [sp, #16]
    ldp     x29, x30, [sp], #32
    ret
```

优化选项 O0 （无优化）

```
<cube>:
    stp     x29, x30, [sp, #-32]!
    mov     x29, sp
    str     w0, [sp, #28]
    ldr     w0, [sp, #28]
    bl      0 <square>
    mov     w1, w0
    ldr     w0, [sp, #28]
    mul     w0, w1, w0
    ldp     x29, x30, [sp], #32
    ret
```

1）因为他既是caller也是callee 所以需要不能避免
2）把x0存在栈上会导致没有必要的对栈的读取 更多的访存开销

03: System-ISA

局部变量

函数局部变量存储在函数栈帧中

为什么不直接把局部变量存储在寄存器（寄存器快）？

- 寄存器数量有限
- 数组和结构体等复杂数据结构
- 局部变量可能需要寻址 (如 &a)

局部变量的分配：在分配栈帧时被一起分配，**局部变量的释放**：在返回前释放栈帧时释放. 局部变量通过 SP 相对地址引用 (例如 ldr x1, [sp, #8])

栈上局部变量的例子

```
caller:
    stp   x29, x30, [sp, -32]!
    mov   x29, sp
    mov   x0, 534
    str   x0, [sp, 24]
    mov   x0, 1057
    str   x0, [sp, 16]
    add   x1, sp, 16
    add   x0, sp, 24
    bl    swap_add
```

caller的栈帧

+24	534 (arg1)
+16	1057 (arg2)
+8	X30
0	X29

sp

```
long caller():
    long arg1 = 534;
    long arg2 = 1057;
    long sum = swap_add(&arg1, &arg2);
    long diff = arg1 - arg2;
    return sum * diff;
```

参数小于8个，x0、x1传参即可

```
swap_add:
    ldr   x3, [x0]
    ldr   x2, [x1]
    str   x2, [x0]
    str   x3, [x1]
    add   x0, x3, x2
    ret
```

拿到x0处的值

交换后存回去

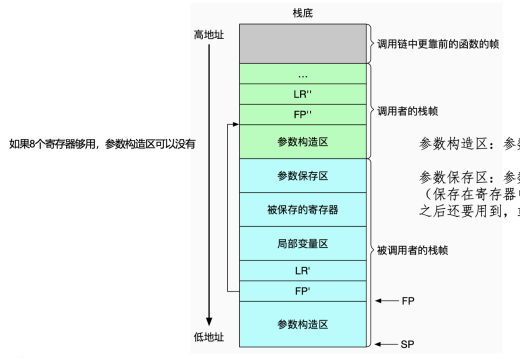
caller的栈帧

+24	534 (arg1)
+16	1057 (arg2)
+8	X30
0	X29

sp

```
long swap_add(long *xp, long *yp):
    long x = *xp;
    long y = *yp;
    *xp = y;
    *yp = x;
    return x + y;
```

栈的全貌



特权级

特权级的必要性

一台计算机上同时运行多个应用程序，如何保证不同应用间的隔离？如果所有的应用均能完全控制硬件计算资源，则会导致混乱例如：某个应用希望关机，某个应

用希望格式化硬盘因此必须先让应用降权，不允许直接改变全局的系统状态例如：中断是否打开方案：必须要有不同的权限级——至少两种权限低权限：不允许改变全局系统状态，用来运行应用高权限：集中运行能改变全局系统状态的操作，形成了操作系统特权操作：操作设备（读取文件、发送网络包…）、调整 CPU 频率、提供进程间通信…

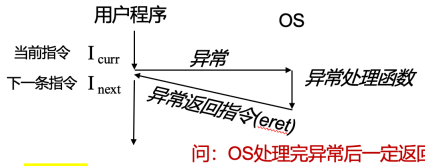
用户ISA 系统ISA

	寄存器		EL0	EL1	描述
	通用寄存器	$X0 \sim X30$	√	√	
内核想访问啥都行	特殊寄存器	PC	√	√	程序计数器
		SP_EL0	√	√	用户栈寄存器
		SP_EL1		√	内核栈寄存器
		$PSTATE$	√[7]	√	状态寄存器
这四个要掌握	系统寄存器	ELR_EL1		√	异常链接寄存器
		$SPSR\_EL1$		√	保存的状态寄存器
		$VBAR\_EL1$		√	异常向量表基地址
		ESR_EL1		√	异常症状寄存器

系统状态寄存器：PSTATE

何时发生特权级切换：发生异常的时候

注意 OS 处理完异常后不一定返回：例如：杀死，收到别人的网络包，时间到了切换别的程序执行.



问：OS处理完异常后一定返回到被打断执行的用户程序吗？

- **同步异常**
  - 执行当前指令触发异常
    - 第一类：用户程序**主动发起**：**svc指令**（OS利用eret指令返回）
    - 第二类：非主动，例如用户程序意外访问空指针：普通ldr指令（OS“杀死”出错程序）
- **异步异常**
  - CPU收到中断信号
    - 从外设发来的中断，例如屏幕点击、鼠标、收到网络包
    - CPU时钟中断，例如定时器超时

异常处理函数

异常处理函数属于操作系统的一部分，是运行在内核态的代码

异常处理函数完成异常处理后，将通过下述操作之一转移控制权：

- 回到发生异常时正在执行的指令
- 回到发生异常时的下一条指令
- 切换到其它进程执行

异常向量表：CPU 找到异常处理函数

操作系统内核预先在一张表，表中准备好不同类型异常的处理函数

- 基地址存储在 VBAR_EL1 寄存器中
- 系统寄存器

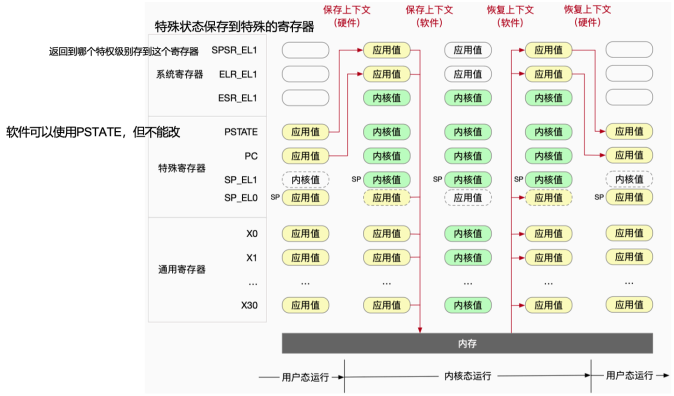
CPU 在异常发生时自动跳转到相应处理函数

- 同步异常：主动下陷 svc、指令执行出错
- 异步异常：中断（IRQ、FIQ）、SError

CPU 执行逻辑

- **CPU的执行逻辑很简单**
  1. 以PC的值为地址从内存中获取一条指令并执行
  2. PC+=4，goto 1（简化，未表示跳转/函数调用）
- **执行过程中可能发生两种情况**
  1. 指令执行出现异常，比如svc、缺页（同步异常）
  2. 外部设备触发中断（异步异常）
- **这两种情况在ARM平台均称为「异常」**
  - 均会导致CPU陷入内核态，并根据异常向量表找到对应的处理函数执行
  - 处理函数执行完后，执行流需要恢复到之前被打断的地方继续运行
    - 注意与x86相关概念的区别

用户态/内核态切换时的处理器状态变化



1. **将发生异常事件的指令地址保存在ELR_EL1中**
2. **将异常事件的原因保存在ESR_EL1**
   - 例如，是执行svc指令导致的，还是访存缺页导致的
3. **将处理器的当前状态（即PSTATE）保存在SPSR_EL1**
4. **栈寄存器不再使用SP_EL0（用户态栈寄存器），开始使用SP_EL1**
   - 内核态栈寄存器，需要由操作系统提前设置
5. **修改PSTATE寄存器中的特权级标志位，设置为内核态**
6. **找到异常处理函数的入口地址，并将该地址写入PC，开始运行操作系统**
   - 根据VBAR_EL1寄存器保存的异常向量表基地址，以及发生异常事件的类型确定

为什么操作系统不能直接使用应用程序在用户态的栈呢？

会往用户态泄露信息。用户态篡改内容，操作系统也会受到影响。

PC 寄存器的值必须由处理器保存，否则当操作系统开始执行时，PC 将被覆盖，从而不知道该返回到哪里。

栈的切换也必须由硬件完成，否则操作系统有可能使用用户态的栈，导致安全问题

eret：从内核态返回到用户态

1. 将 SPSR_EL1 中的处理器状态写入 PSTATE 中，处理器状态也从 EL1 切换到 EL0
2. 栈寄存器不再使用 SP_EL1，开始使用 SP_EL0 **注意：SP_EL1 的值并没有改变**，下一次下陷时，操作系统依然会使用这个内核栈
3. 将 ELR_EL1 中的地址写入 PC，并执行应用程序代码

操作系统在切换过程中的任务

- **主要任务：将属于应用程序的 CPU 状态保存到内存中**
  - 用于之后恢复应用程序继续运行
- **应用程序需要保存的运行状态称为处理器上下文**
  - 处理器上下文（Processor Context）：应用程序在完成切换后恢复执行所需的最小处理器状态集合
  - 处理器上下文中的寄存器具体包括：
    - 通用寄存器 X0-X30
    - 特殊寄存器，主要包括PC、SP和PSTATE
    - 系统寄存器，包括页表基地址寄存器等

如果寄存器放不下参数怎么办？

寄存器放不下，只能通过内存传；将参数放在内存中，将指针放在寄存器中传给内核，内核通过指针访问相关参数. 存在安全的隐患.

04: SYSCALL

系统调用优化：VDSO

系统调用的时延不可忽略: 尤其是调用非常频繁的情况，系统调用实际执行逻辑很简单，但是需要耗费几百上千个时钟周期。

如何降低系统调用的时延？特权级切换造成的时间开销，如果没有模式切换，那么就不需要保存恢复状态

gettimeofday

VDSO：很多系统调用就是为了读一个状态，读取一个信息的代码本身很简单，但是因为system call所以非常慢。
解决办法：直接把要读的数据放到用户这。

- **内核定义**
  - 在编译时作为内核的一部分
- **用户态运行**
  - 将gettimeofday的代码加载到一块与应用共享的内存页
  - 这个页称为：vDSO
    - Virtual Dynamic Shared Object
  - Time 的值同样映射到用户态空间（只读）
    - 只有在内核态才能更新这个值
- **Q：和以前的gettimeofday相比有什么区别?**

系统调用优化：Flex-SC

如何进一步降低系统调用的时延？不仅仅是 gettimeofday(). ” 时间都去哪儿了？”

1. 大部分是用来做状态的切换保存和恢复状态 + 权限的切换
2. Cache pollution

是否有可能在不切换状态的情况下实现系统调用?

Flexible System Call

一种新的 syscall 机制

引入 **ssystem call page** ，由 user & kernel 共享

User threads 可以将系统调用的请求 push 到 system call page, kernel threads 会从 system call page poll system call 请求

Exception-less syscall 将系统调用的调用和执行解耦，可分布到不同的 CPU 核

1、宏内核（Monolithic Kernel）

整个系统分为内核与应用两层:

- 内核：运行在特权级，集中控制所有计算资源
- 应用：运行在非特权级，受内核管理，使用内核服务

优点: 宏内核拥有丰富的沉淀和积累, 拥有巨大的统一的社区和生态, 针对不同场景优化了 30 年.

宏内核的结构性缺陷安全性与可靠性问题：模块之间没有很强的隔离机制

实时性支持：系统太复杂导致无法做最坏情况时延分析

系统过于庞大而阻碍了创新：Linux 代码行数已经过 2 千万

宏内核难以满足的应用场景

向上向下的扩展：很难去剪裁/扩展一个宏内核系统支持从 KB 级别到 TB 级别的场景 **硬件异构性**：很难长期支持一些定制化的方式去解决一些特定问题 **功能安全**：一个广泛共识：Linux 无法通过汽车安全完整性认证（ASIL-D）**信息安全**：单点错误会导致整个系统出错，而现在有数百个安全问题（CVE）**确定性时延**：Linux 花费 10+ 年合并实时补丁，目前依然不确定是否能支持确定性时延

2、微内核系统架构

设计原则：最小化内核功能

- 将操作系统功能移到用户态，称为” 服务”（Server）
- 在用户模块之间，使用消息传递机制通信

微内核优缺点

优点易于扩展：直接添加一个用户进程即可为操作系统增加服务

易于移植：大部分模块与底层硬件无关

更加可靠：在内核模式运行的代码量大大减少

更加安全：即使存在漏洞，服务与服务之间存在进程粒度隔离

更加健壮：单个模块出现问题不会影响到系统整体

缺点性能较差：内核中的模块交互由函数调用变成了进程间通信

生态欠缺：尚未形成像 Linux 一样具有广泛开发者的社区

重用问题：重用宏内核操作系统提供兼容性，带来新问题

06:VM-ADDR-TRANSLATION

分页机制的特点

物理内存离散分配; 任意虚拟页可以映射到任意物理页, 大大降低对物理内存连续性的要求; 主存资源易于管理，利用率更高; 按照固定页大小分配物理内存, 能大大降低外部碎片和内部碎片; 被现代处理器和操作系统广泛采用

页表基地址寄存器

AARCH64:

两个页表基地址寄存器：TTBR0_EL1 & TTBR1_EL1

MMU 根据虚拟地址第 63 位选择使用哪一个

以 Linux 为例

应用程序（地址首位为 0）：使用 TTBR0_EL1

操作系统（地址首位为 1）：使用 TTBR1_EL1

X86_64

一个寄存器：CR3（Control Register 3）

页表使能

机器上电会先进入物理寻址模式，系统软件需配置控制寄存器使能页表从而进入虚拟寻址模式

AARCH64, SCTLR_EL1, System Control Register, EL1

第 0 位（M 位）置 1，即在 EL0 和 EL1 权限级使能页表

X86_64, CR4，第 31 位（PG 位）置 1，即使能页表

页表是内存，需要初始化，初始化不是硬件做的，是操作系统做的。

绝大部分情况下，使用虚拟地址（只要页表开了）

TLB：缓存页表项

多级页表的设计是典型的用时间换空间的设计

- 优点：压缩页表大小
- 缺点：增加了访存次数（逐级查询）,1 次内存访问，变成了 5 次内存访问

TLB 位于 CPU 内部，是页表的缓存

- Translation Lookaside Buffer
- 缓存了虚拟页号到物理页号的映射关系
- 在地址翻译过程中，MMU 首先查询 TLB; TLB 命中，则不再查询页表, TLB 未命中，再查询页表

TLB 刷新

TLB 使用虚拟地址索引, 当 OS 切换进程页表时需要全部刷新

AARCH64 上内核和应用程序使用不同的页表分别存在 TTBR0_EL1 和 TTBR1_EL1 , 使用的虚拟地址范围不同系统调用过程不用切换.

降低开销

硬件特性 ASID（AARCH64）：Address Space ID

OS 为不同进程分配 8 位或 16 位 ASID，ASID 的位数由 TCR_EL1 的第 36 位（AS 位）决定，OS 负责将 ASID 填写在 TTBR0_EL1 的高 8 位或高 16 位.

TLB 的每一项也会缓存 ASID, 地址翻译时，硬件会将 TLB 项的 ASID 与 TTBR0_EL1 的 ASID 对比, 若不匹配，则 TLB miss.

使用了 ASID 之后切换页表（即切换进程）后，不再需要刷新 TLB，提高性能.

修改页表映射后，仍需刷新 TLB.

TLB 与多核

OS 修改页表后，需要刷新其它核的 TLB 吗？需要，因为一个进程可能在多个核上运行

OS 如何知道需要刷新哪些核的 TLB？操作系统知道进程调度信息

OS 如何刷新其他核的 TLB?

x86_64: 发送 IPI 中断某个核，通知它主动刷新

AARCH64: 可在 local CPU 上刷新其它核 TLB 调用的 ARM 指令：TLBI ASIDE1IS

虚拟内存机制的优势

高效使用物理内存，使用 DRAM 作为虚拟地址空间的缓存

简化内存管理，每个进程看到的是统一的线性地址空间

更强的隔离与更细的权限控制，一个进程不能访问属于其他进程的内存用户程序不能够访问特权更高的内核信息不同内存页的读、写、执行权限可以不同

不同进程互不干扰，仿佛独占所有内存

绝大部分地址段均可用，除了顶部的内核地址区域

08:VM-MANAGE

从实验内核看系统初始化

设置 CPU 异常级别为 EL1

设置页表并开启虚拟内存机制:

TTBR0_EL1: 虚拟地址 = 物理地址

TTBR1_EL1: 虚拟地址 = 物理地址 + OFFSET

设置异常向量表

每个异常向量表项跳转到对应的异常处理函数

处理异常前保存进程上下文、返回进程前恢复其上下文

合法虚拟地址信息的记录方式

VMA(VM AREA，每一进程都会有的数据结构，或者叫 VM Region)。除了起始地址和初始地址，还要加上文件的信息（elf 的信息，“后台” 文件）

Problems

Q1.AArch64 页表中有一个 nG(notGlobal) 位，当 nG 位是 0 的时候，表示该映射所对应的 TLB 缓存项对所有 ASD 有效，反之当 nG 位是 1 的时候，表示对应的 TLB 缓存项只对当前 ASID 有效。请思考哪些情况下 nG 位应设置为 0，哪些情况下应设置为 1，两种情况下设置错了分别会有什么后果?(4)

在 AArch64 页表中，nG（notGlobal）位用于控制 TLB（Translation Lookaside Buffer）缓存项的全局性。以下是 nG 位设置为 0 或 1 的一些常见情况，以及可能的后果：**nG 位设置为 0 的情况：**当 nG 位设置为 0 时，表示 TLB 缓存项对所有 ASID（Address Space Identifier）有效。这在所有进程都需要访问同一内存映射的情况下很有用，比如操作系统内核的内存映射。在这种情况下，使得所有的进程都可以使用相同的 TLB 条目，节省了 TLB 空间并提高了效率。如果错误地在非全局映射上设置了 nG 位为 0，可能会导致安全性问题和性能问题。不同的进程可能会错误地共享同一 TLB 条目，导致一个进程可能访问到另一个进程的内存空间，这可能引发安全问题。此外，频繁的 TLB 刷新也可能降低系统的性能。**nG 位设置为 1 的情况：**当 nG 位设置为 1 时，表示 TLB 缓存项只对当前 ASID 有效。这在每个进程有自己独立的内存映射的情况下很有用，比如用户空间的内存映射。在这种情况下，每个进程都有自己的 TLB 条目，可以防止进程之间的内存访问冲突。如果错误地在全局映射上设置了 nG 位为 1，可能会导致性能问题。由于全局映射需要被所有进程共享，如果每个进程都有自己的 TLB 条目，那么会浪费大量的 TLB 空间，并且在进程切换时需要频繁的 TLB 刷新，这可能降低系统的性能。总的来说，nG 位的设置需要根据内存映射的具体情况进行合理选择，以保证系统的性能和安全性。

Q2. 现代 CPU 中至少有两个特权级，分别用来运行内核和用户程序。为什么要这样设计？现代 CPU 通常设计有多个特权级（也称为保护模式或特权模式）来满足操作系统和应用程序之间的隔离、安全性和资源管理的需求。至少包括两个特权级是为了区分内核（或操作系统）和用户程序的运行环境。以下是为什么需要这样设计的主要原因：1. **安全性和隔离**：内核通常运行在最高特权级，拥有对硬件资源的完全访问权限，这是因为内核需要管理和控制整个系统的资源。用户程序在较低的特权级运行，限制其直接访问硬件资源。这种设计确保了用户程序无法直接修改内核数据结构或执行特权指令，从而提供了操作系统和用户程序之间的隔离，增强了系统的安全性。

2. **错误隔离**：由于用户程序可能存在缺陷或错误，通过特权级的隔离可以防止错误的用户程序影响到整个系统的稳定性。例如，用户程序可能会意外地尝试访问非法的内存地址，但由于它在较低的特权级运行，这种错误行为会被操作系统捕获并进行相应的错误处理，而不会导致整个系统崩溃。

3. **资源管理**：内核需要执行许多关键的管理任务，如任务调度、内存管理和设备驱动管理等。这些任务需要对硬件进行直接操作。通过将内核运行在高特权级，可以确保内核能够有效地管理这些资源，而不被用户程序干扰。

4. **系统调用和中断处理**：用户程序经常需要请求操作系统提供的服务，例如文件操作、网络访问等。这是通过系统调用实现的。当用户程序发起系统调用时，会发生特权级的切换，从低特权级切换到高特权级，然后操作系统在高特权级下执行相关的服务，执行完毕后再切换回用户程序的特权级。此外，中断处理也通常在高特权级下执行。

5. **性能和效率**：特权级的设计也考虑到性能和效率。例如，由于内核运行在高特权级，它可以直接访问硬件资源，而不需要进行额外的检查或中介操作，这有助于提高操作的效率。综上所述，特权级的设计是为了确保操作系统的稳定性、安全性和效率，同时为用户程序提供一个受限但简单、清晰的执行环境。

Q3. 立即映射和延迟映射的区别？

立即映射（Eager Mapping）：在这种策略中，当分配给进程的内存页被创建或分配时，立即在页表中为这些页创建映射。这种策略的好处是，一旦内存分配完成，进程就可以立即访问这些内存，无需等待映射的创建。然而，缺点是这可能会创建大量不必要的映射，因为并非所有分配的内存都会被立即或实际使用。

延迟映射（Lazy Mapping）：在这种策略中，内存页的映射不是在内存分配时立即创建的，而是在进程首次尝试访问这些页时创建。这种策略的好处是，可以减少不必要的映射，节省页表的空间，因为只有实际被使用的内存页才会被映射。然而，缺点是进程首次访问内存时可能会因为需要创建映射而产生延迟。

Q4. SLAB 和伙伴系统的区别？伙伴系统最小的分配单位是一个物理页 (4KB), 但是大多数情况下, 内核需要分配的内存大小通常是几十个字节或几百个字节, 远远小于一个物理页的大小。如果仅仅使用伙伴系统进行内存分配, 会出现严重的内部碎片问题, 从而导致内存资源利用率降低。于是, 操作系统开发人员设计了另外一套内存分配机制用于分配小内存, 该机制经过演化就形成 SLAB 分配器。

总的来说，伙伴系统更适合于大块内存的管理，特别是当内存的分配和释放大小变化较大时。而 SLAB 分配器则针对小而频繁的内存分配进行了优化，特别是对于需要快速分配和释放大量相同大小内存块的场景。
首先,SLUB 分配器是为了满足操作系统 (频繁的) 分配小对象的需求, 其依赖于伙伴系统进行物理页的分配。简单来说,SLUB 分配器做的事情是把伙伴系统分配的大块内存进一步细分成小块内存进行管理。一方面由于操作系统频繁分配的对象大小相对比较固定, 另一方面为了避免外部碎片问题, 所以 SLUB 分配器只分配固定大小的内存块.
Q5．使用大页的好处？使用大页的好处显而易见, 一方面它能够减少 TLB 缓存项的使用, 从而有机会提高 TLB 命中率, 另一方面它可以减少页表的级数, 从而提升查询页表的效率。不过, 过度的大页使用也会有弊端, 一方面应用程序可能未使用整个大页而造成物理内存资源浪费, 另一方面大页的使用也会增加操作系统管理内存的复杂度,Linux 中就存在与大页相关的漏洞。**好处案例：**提供 API 允许应用程序进行显示的大页分配透明大页（Transparent Huge Pages) 机制
Q6．对内核来说：已映射的地址不一定正在使用（需要通过 kmalloc 才能用）正在使用的地址通常已映射（例外：vmalloc）
当我们谈论操作系统内核中的内存管理时，有两个重要的概念需要理解：地址映射和地址使用。**1．已映射的地址不一定正在使用：**- 当我们说一个地址已经被映射，这意味着在虚拟内存和物理内存之间已经建立了一种关联。即，一个虚拟地址已经被指定对应到某块物理内存上。- 但是，仅仅映射并不意味着这块内存正在被 actively 使用。例如，在 Linux 内核中，可以预先分配一大块连续的虚拟地址空间，并在页表中为其分配相应的物理内存。这个时候，这些地址虽然已经映射，但除非有进程或内核模块通过分配函数（如 ‘kmalloc‘）显式请求使用这些内存，否则它们只是预留下来并不被使用。- ‘kmalloc‘ 是 Linux 内核中用于分配物理连续的内存空间的函数。当调用 ‘kmalloc‘ 时，内核会在已映射的虚拟地址空间中找到一块尚未使用的空间，并标记为正在使用，然后返回这块内存的地址。**2．正在使用的地址通常已映射：**- 在大多数情况下，一个正在被使用的虚拟地址一定是已经映射到物理内存的。换句话说，当一个程序或内核模块请求一块内存并使用它时，这个虚拟地址肯定是有对应的物理地址的，否则就无法进行读写操作。- 但有一种例外情况，那就是通过 ‘vmalloc‘ 分配的内存。‘vmalloc‘ 用于分配虚拟连续但物理上可能不连续的内存空间。当使用 ‘vmalloc‘ 分配内存时，内核只是预留了一块虚拟地址空间，但不会立即分配对应的物理内存，这部分的物理内存只有在实际访问时才会被按需分配（按需分页）。所以，有些正在使用的地址（通过 ‘vmalloc‘ 分配的）在刚开始可能并没有被实际映射到物理内存上。总的来说，虚拟内存的映射和使用在操作系统中是一个动态的过程，根据需求进行按需分配和映射。

4.1.2 进程操作

- getpid():返回当前进程PID，getppid():返回父进程PID，类型为pid-t。
- exit()终止进程并返回上一个status，无返回值
- fork()创建（复制）新进程，返回值在父进程中为字进程PID，字进程为0
- execve(file,argv,env)加载运行文件，成功无返回，失败返回-1

4.1.3 僵尸进程

已经终止没被回收的进程，等待父进程回收（将exit状态传给父进程），没回收将被系统初始化时|核创建的init进程回收，init进程PID为1。

4.1.4 waitpid

pid-t waitpid(pid-tt pid, int *status, int option)

- 参数pid设置为子进程pid则指定等待回收该子进程，设置-1则包含全部子进程
- option=0则等待集合中任意子进程终止（如果有子进程在调用前就已经终止则立刻返回）。
- status指针用于带回子线程的终止状态信息。
- 返回值为子进程pid，若无子进程返回-1，errno = ECHILD，等待被中断返回-1，errno = EINTR

4.1.5 内存使用

- 协商划分方案：内存总大小不够，缺乏隔离，程序编译难
- 分时服用方案: 写入与读取损失性能，物理内存容量不够

目标：彼此隔离，具有统一性（易于开发），内存总量突破物理内存总量限制。
虚拟内存：所有应用共享统一、连续虚拟内存，CPU按照OS的规则进行地址翻译。（可以不做映射）应用程序所使用的虚拟地址彼此隔离。
优势：提升内存资源利用率（按需映射），突破物理内存限制（可以暂存在磁盘）
进程的虚拟内存布局:书P75。

4.2 文件

IO设备均被抽象为文件，输入输出为读写文件。文件类型：普通文件、目录（文件名到文件的映射）、套接字。文件有标识符file descripter(fd)，文件操作：Open(filename, flags, mode), 成功返回新文件标识符，出错则-1
close(fd)，成功返回0，失败返回-1。read/write, 成功返回读/写字节数，失败返回-1，读遇到EOF返回0

6 启动

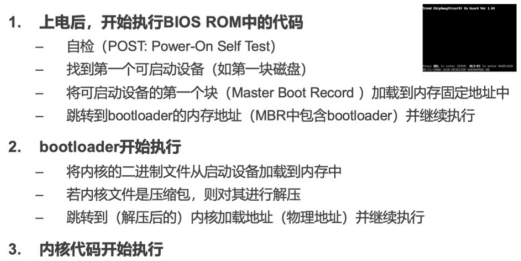


Figure 1: bootloader

内核函数的位置：CPU从预定义的RAM地址读取第一行代码,是物理地址（还没配置映射）

- 设置当前CPU异常级别至EL1
- 设置栈，调用C函数（C函数中可能有函数调用，所以必须用到栈）
- 跳转到高虚拟地址运行start-kernel, 此时在C函数中已经配置好了页表，内核态运行在高地址

C函数：页表初始化（设置TTBR0与TTBR1页表，后启动地址翻译）C函数：异常向量表初始化（每个异常向量表项跳转到对应的异常处理函数，注意上下文的保存与恢复）