IPADS
INSTITUTE OF PARALLEL AND DISTRIBUTED SYSTEMS

上海交通大学
SHANGHAI JIAO TONG UNIVERSITY

操作系统的硬件运行环境

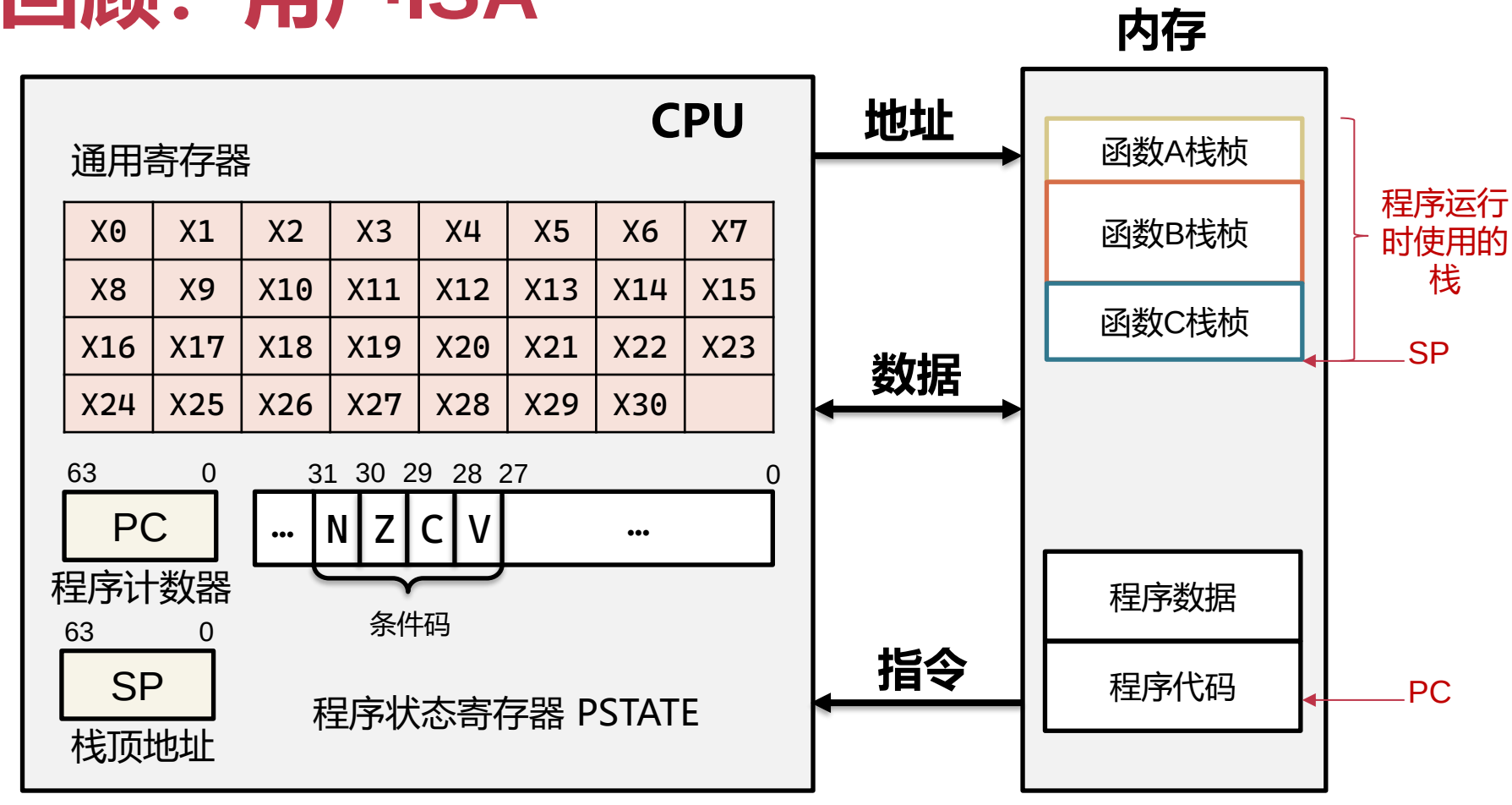
上海交通大学

https://www.sjtu.edu.cn

版权声明

- 本内容版权归**上海交通大学并行与分布式系统研究所**所有
- 使用者可以将全部或部分本内容免费用于非商业用途
- 使用者在使用全部或部分本内容时请注明来源
  - 内容来自：上海交通大学并行与分布式系统研究所+材料名字
- 对于不遵守此声明或者其他违法使用本内容者，将依法保留追究权
- 本内容的发布采用 Creative Commons Attribution 4.0 License
  - 完整文本：https://creativecommons.org/licenses/by/4.0/legalcode

回顾：用户ISA


内存
CPU
地址
数据
指令
通用寄存器
X0 X1 X2 X3 X4 X5 X6 X7
X8 X9 X10 X11 X12 X13 X14 X15
X16 X17 X18 X19 X20 X21 X22 X23
X24 X25 X26 X27 X28 X29 X30
63 0
PC
程序计数器
63 0
SP
栈顶地址
31 30 29 28 27 0
… N Z C V …
条件码
程序状态寄存器 PSTATE
函数A栈桢
函数B栈桢
函数C栈桢
程序运行时使用的栈
SP
程序数据
程序代码
PC

回顾：函数调用

- ✓ 调用被调用者：**bl**指令
- ✓ 返回到调用者：**ret**指令
- ✓ 传递数据：**寄存器**与**栈**
- ✓ 寄存器使用约定：**调用者保存、被调用者保存**
- ✓ 局部变量：存在函数**栈桢中**

回顾：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：若使用调用者保存的寄存器（如x9），是否能够避免保存?

小思考

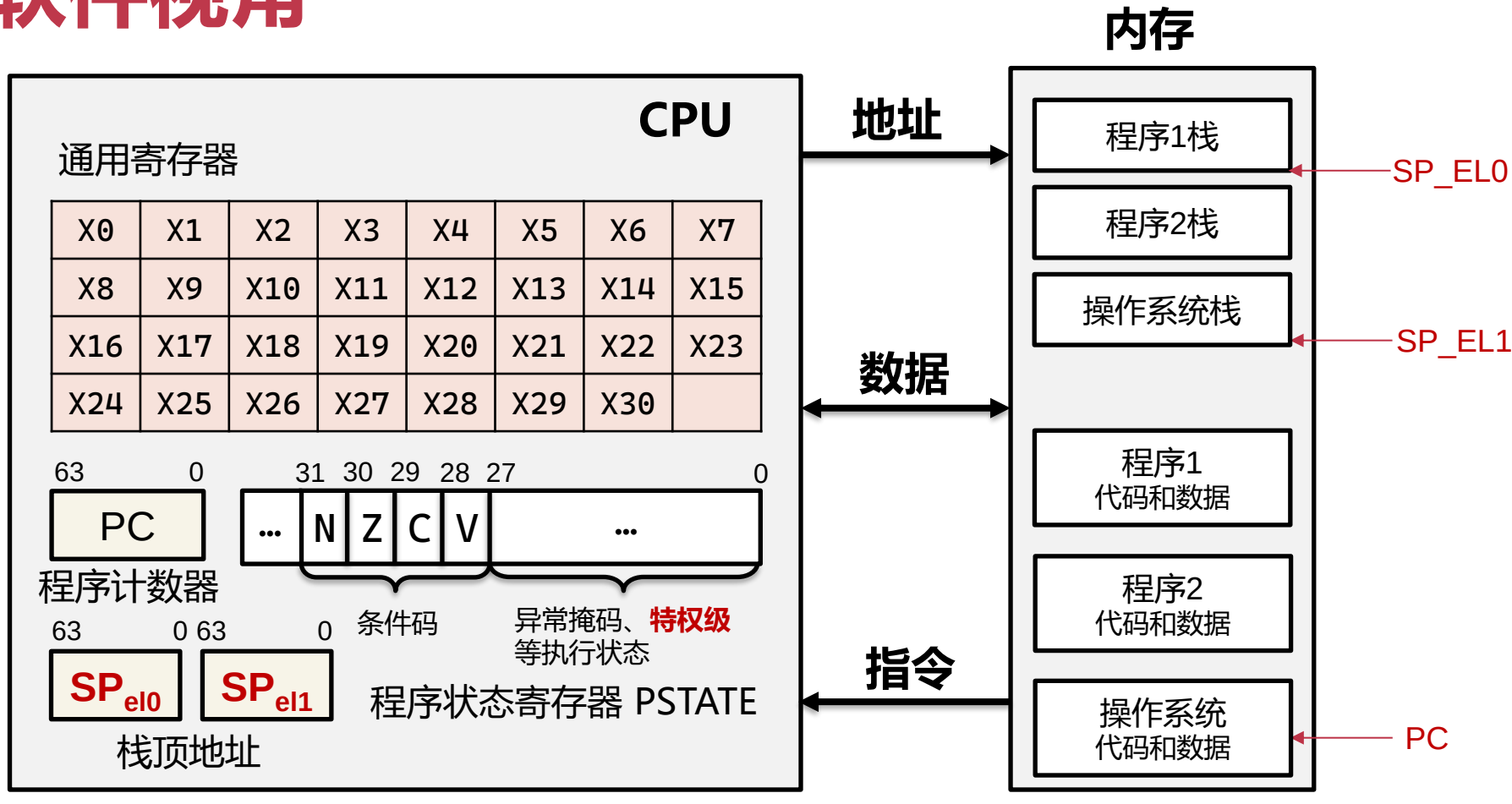
- **当操作系统运行时，代码和数据存放在哪里？**
  - 操作系统用栈吗？

```
640  static void *____cache_alloc_node(struct kmem_cache *, gfp_t, int);
641  static void *alternate_node_alloc(struct kmem_cache *, gfp_t);
642
643  static struct alien_cache *__alloc_alien_cache(int node, int entries,
644                                                 int batch, gfp_t gfp)
645  {
646          size_t memsize = sizeof(void *) * entries + sizeof(struct alien_cache);
647          struct alien_cache *alc = NULL;
648
649          alc = kmalloc_node(memsize, gfp, node);
650          if (alc) {
651                  kmemleak_no_scan(alc);
652                  init_arraycache(&alc->ac, entries, batch);
653                  spin_lock_init(&alc->lock);
654          }
655          return alc;
656  }
657
658  static struct alien_cache **alloc_alien_cache(int node, int limit, gfp_t gfp)
659  {
660          struct alien_cache **alc_ptr;
661          int i;
662
663          if (limit > 1)
664                  limit = 12;
665          alc_ptr = kcalloc_node(nr_node_ids, sizeof(void *), gfp, node);
666          if (!alc_ptr)
667                  return NULL;
668
669          for_each_node(i) {
670                  if (i == node || !node_online(i))
671                          continue;
672                  alc_ptr[i] = __alloc_alien_cache(node, limit, 0xbaadf00d, gfp);
673                  if (!alc_ptr[i]) {
674                          for (i--; i >= 0; i--)
675                                  kfree(alc_ptr[i]);
676                          kfree(alc_ptr);
677                          return NULL;
678                  }
679          }
680          return alc_ptr;
681  }
```

小思考

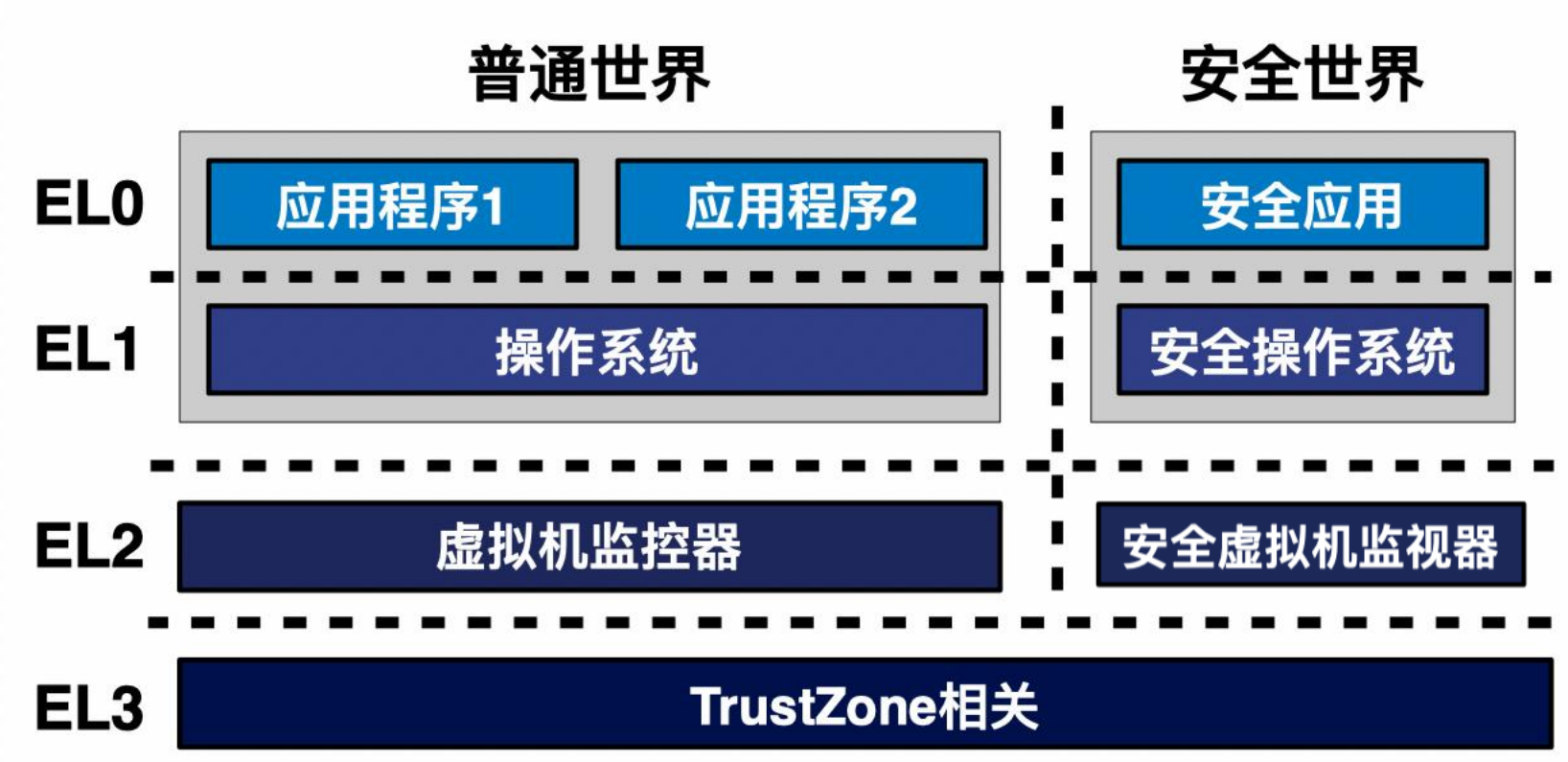
- **当操作系统运行时，代码和数据存放在哪里?**
  - 操作系统用栈吗?
- **操作系统会使用应用运行时使用的通用寄存器吗?**
  - 条件码呢?
- **操作系统运行和应用运行时有什么区别吗?**

软件视角


内存
CPU
通用寄存器
X0 X1 X2 X3 X4 X5 X6 X7
X8 X9 X10 X11 X12 X13 X14 X15
X16 X17 X18 X19 X20 X21 X22 X23
X24 X25 X26 X27 X28 X29 X30
63 0
PC
程序计数器
31 30 29 28 27 0
… N Z C V …
条件码
异常掩码、特权级
等执行状态
63 0 63 0
SP el0
SP el1
栈顶地址
程序状态寄存器 PSTATE
地址
数据
指令
程序1栈
程序2栈
操作系统栈
程序1
代码和数据
程序2
代码和数据
操作系统
代码和数据
SP_EL0
SP_EL1
PC

特权级别

ARMv8.4特权级 (Exception Level)



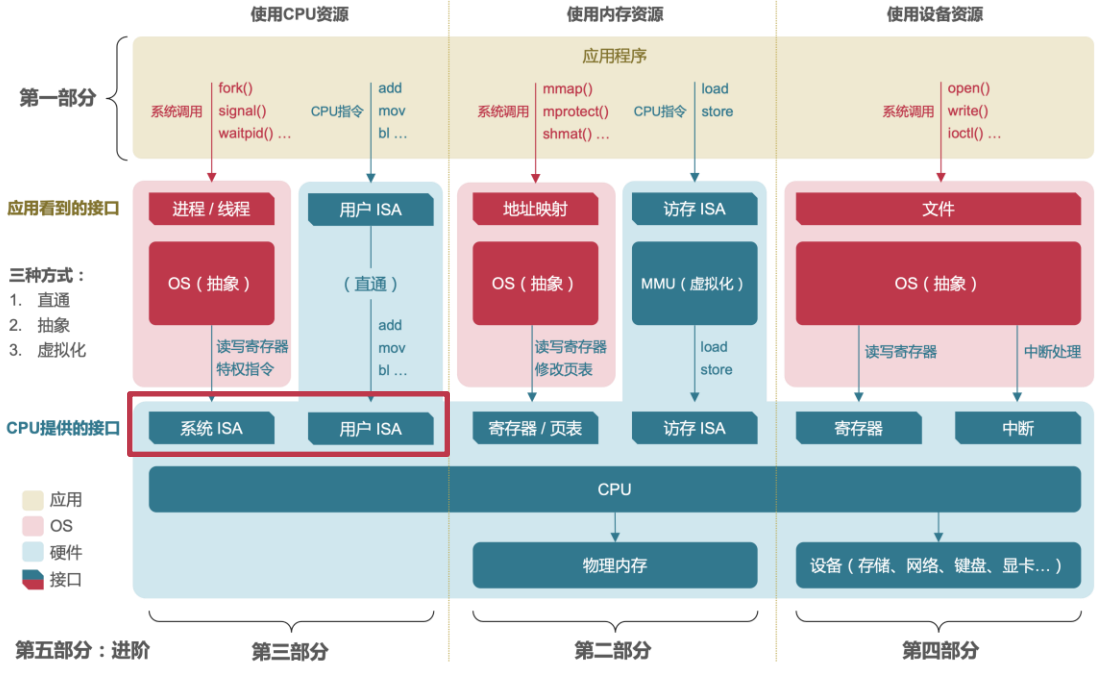
系统状态寄存器：PSTATE

- **抽象进程状态信息（PSTATE）**
  - 条件码 (Condition flags)
    - NZCV
  - 执行状态 (Execution state controls)
    - **CurrentEL：CPU当前特权级别**
  - 异常掩码 (Exception mask bits)
    - DAIF
  - 访问控制 (Access control bits)
    - 例如PAN（Privileged Access Never）

Special-purpose register	PSTATE fields
NZCV	N, Z, C, V
DAIF	D, A, I, F
CurrentEL	EL
SPSel	SP
PAN	PAN
UAO	UAO
DIT	DIT
SSBS	SSBS

用户ISA与系统ISA

- **用户ISA**
  - 通用寄存器
  - （用户）栈寄存器
  - 条件码寄存器
  - 运算指令等
- **系统ISA**
  - 系统寄存器
  - 系统指令



用户态（EL0）与内核态（EL1）

- **用户态（User-mode）**
  - 只能使用用户 ISA
- **内核态（Kernel-mode）**
  - 可以同时使用系统 ISA 和用户ISA
- **操作系统往往同时包含内核态与用户态的代码**
  - 如：Unix包含内核态的kernel 与 用户态的 shell

AArch64 中常见寄存器在不同特权级的可见情况

用户ISA 系统ISA

寄存器		EL0	EL1	描述
通用寄存器	$X0 \sim X30$	√	√	
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
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

EL0与EL1

特权级切换

用户态和内核态之间的控制流跳转

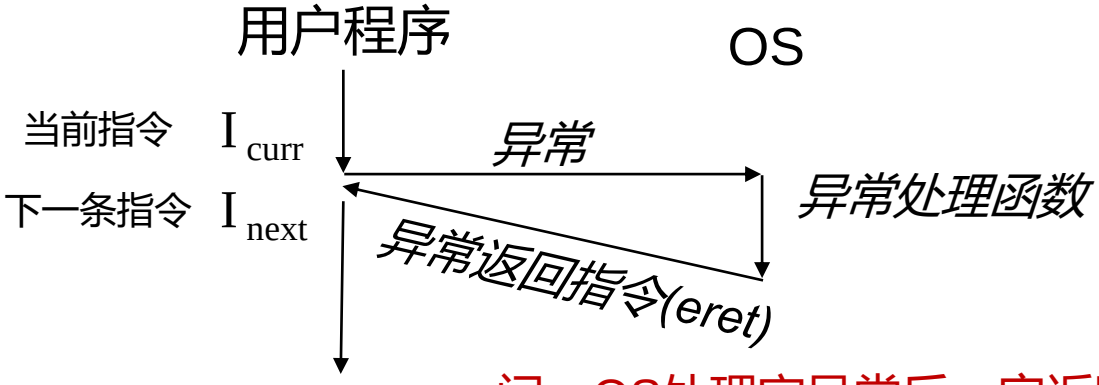
- **当CPU处于用户态（EL0）执行应用程序时，如何改变CPU控制流从用户态进入内核态?**
  - 已知的两种改变控制流的方式：
    - 跳转指令，如 b
    - 过程调用与返回指令，如 bl 和 ret
  - 这两种方式只能在同一种模式之间跳转
  - 需要新的指令（**在控制流跳转的同时进行特权级切换**）：
    - **svc/eret**

特权级切换的必要性

- **操作系统的职责之一：**
  - 服务应用、管理应用

- **特权级切换的必要性：**
  - 将CPU控制权移交给内核
  - 服务：应用程序向操作系统请求服务
  - 管理：操作系统能够切换不同应用程序执行
    - 否则，错误/恶意程序死循环怎么办

何时发生特权级切换：发生异常



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

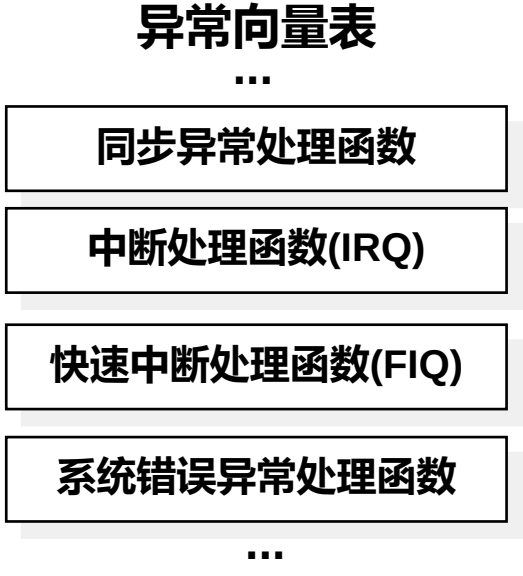
- **属于操作系统的一部分**
  - 运行在内核态的代码
- **异常处理函数完成异常处理后，将通过下述操作之一转移控制权：**
  - 回到发生异常时正在执行的指令
  - 回到发生异常时的下一条指令
  - 不返回/切换到其它进程执行

思考：什么情况返回到当前指令？
什么情况返回到下一条指令？

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中设置不同异常的处理函数**
  - 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



小结：CPU的执行逻辑

- **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/segmentation fault（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

为异常处理，操作系统需要做

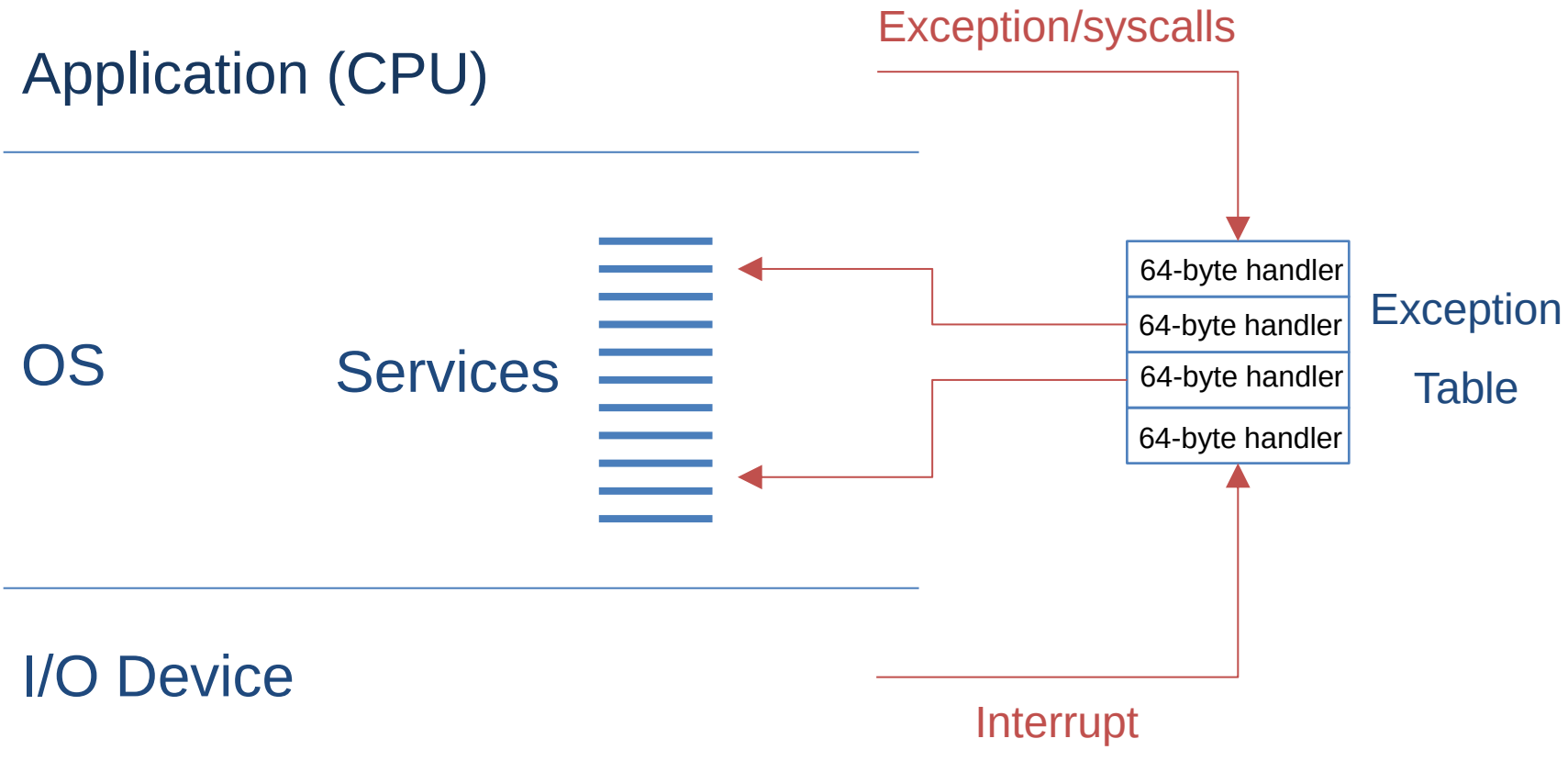
一、实现对异常向量表的设置

- 系统初始化的重要工作之一：在开启中断和启动第一个应用之前
- **msr vbar_el1**, x0 (内核态才能使用的指令和访问的寄存器)

二、实现对不同异常（中断）的处理函数

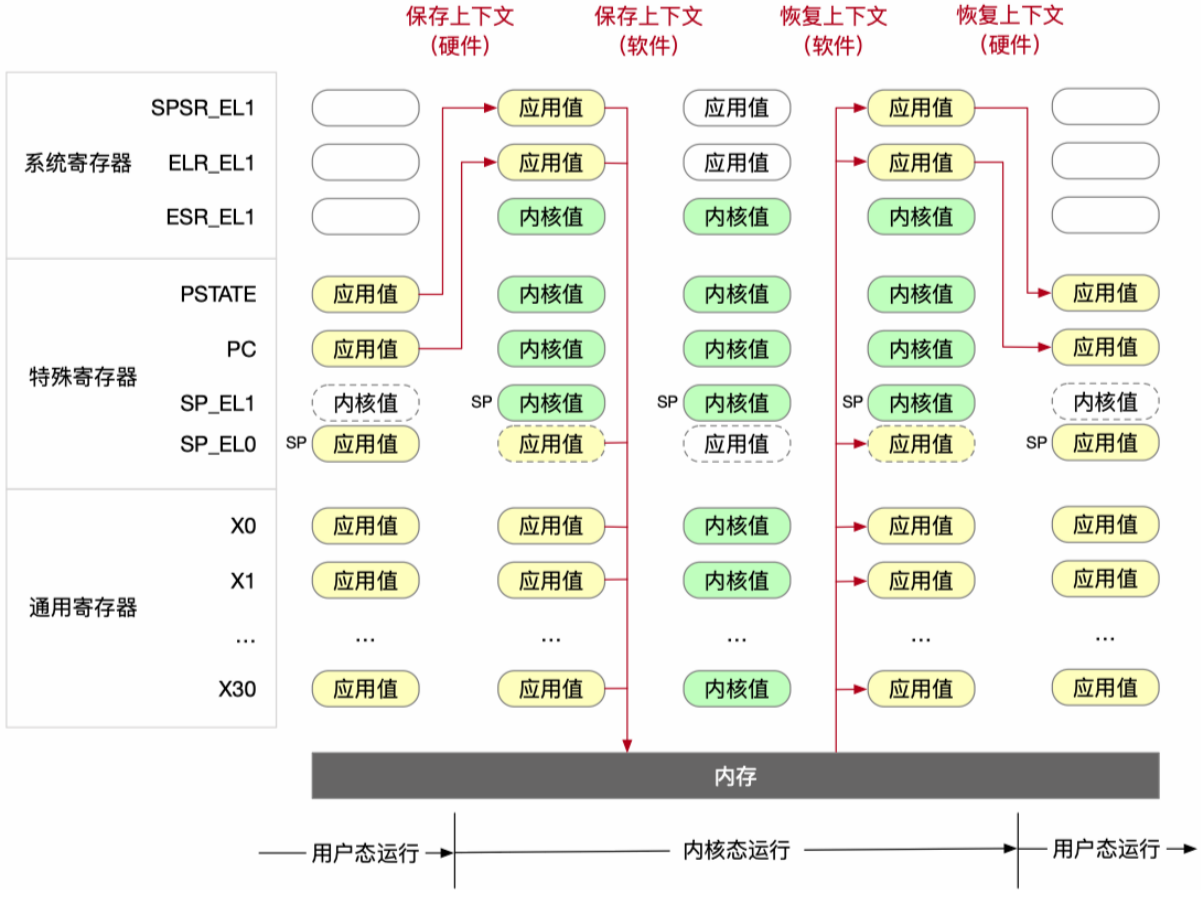
- 处理应用程序出错的情况：如访问空指针
  - Q：内核如果自己运行出错怎么办？
- 一类特殊的同步异常：系统调用，由应用主动触发
  - Q：如何识别出是系统调用？ **mrs x1, esr_el1**
- 处理来自外部设备的中断：如收取网络包、获取键盘输入等

操作系统异常处理示意图



内核态与用户态的切换

用户态/内核态切换时的处理器状态变化



Q：操作系统的运行状态（寄存器）需要保存吗？

处理器（硬件）在切换过程中的任务

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，还是访问地址出错导致的
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

思考题

- 为什么操作系统不能直接使用应用程序在用户态的栈呢?

处理器的这些操作都是必要的吗?

- **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

- **栈的切换是否也必须由硬件完成?**
  - 不一定!
  - 原则：避免操作系统使用用户态应用程序的栈
  - 软件实现：在异常处理函数的第一行指令就换栈

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**

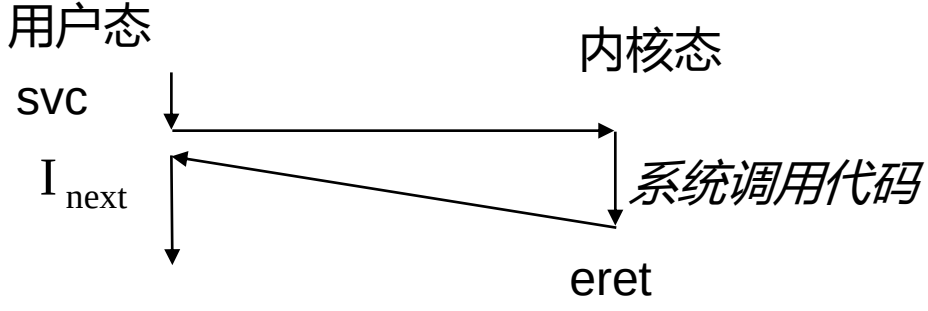
操作系统（软件）在切换过程中的任务

- **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

系统调用

系统调用

- **硬件提供了一对指令svc/eret指令来在用户态、内核态之间切换**
- **系统调用**
  - 用户与操作系统之间，类似于过程调用的接口
  - 通过受限的方式访问内核提供的服务



系统调用例子

```
# hello world
1 int main()
2 {
3     write(1, "hello, world\n", 13);
4      _exit(0);
5 }
```

AArch64下常见的Linux的系统调用

编号	名称	描述.	编号	名称	名称.
17	getcwd	Get current working directory	129	kill	Send signal to a process
23	dup	Duplicate a file descriptor	172	getpid	Get process ID
56	openat	Open a file	214	brk	Set the top of heap
57	close	Close a file	215	munmap	Unmap a file from memory
63	read	Read a file	220	clone	Create a process
64	write	Write a file	221	execve	Execute a program
80	fstat	Get file status	222	mmap	Map a file into memory
93	_exit	Terminate the process	260	wait4	Wait for process to stop

系统调用例子

```
1     .section .rodata
2     .LC0:
3           .string "hello, world\n"
4     .text
5     .align 2
6     .global main
7     .type main, %function
8 main:
```

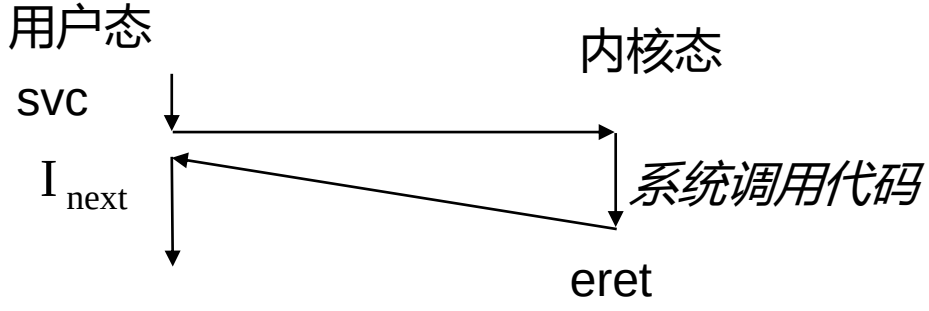
系统调用例子

```
  First, call write(1, "hello, world\n", 13)
9       movq   x8, #0x40          write is system call 64
10      movq   x0, #0x1           Arg1:stdout has descriptor 1
11      adrp   x3, .LC0
12      add x1,x3,:lo12:.LC0      Arg2:Hello world string
12      movq   x2, #0xd           Arg3:string length
13      svc                       Make the system call
  Next, call exit(0)
14      movq   x8, #0x5d          _exit is system call 93
15      movq   x0, #0x0           Arg1:exit status is 0
16      svc                       Make the system call
```

系统调用的参数传递和返回值（软件约定）

- **最多允许8个参数**
  - x0-x7寄存器
  - x8用于存放系统调用编号

- **返回值存放于x0寄存器中**



系统调用返回值与errno

- **库函数API**
  - 出错时返回-1，并设置全局变量errno为具体的错误值

- **系统调用ABI：通过寄存器向应用传递返回值**
  - 出错时设置为 -errno
  - 库对系统调用的 wrapper code 会将系统调用的返回值转换为库函数形式的返回值

Q: 如果寄存器放不下参数怎么办?

- **寄存器放不下，只能通过内存传参**
  - 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

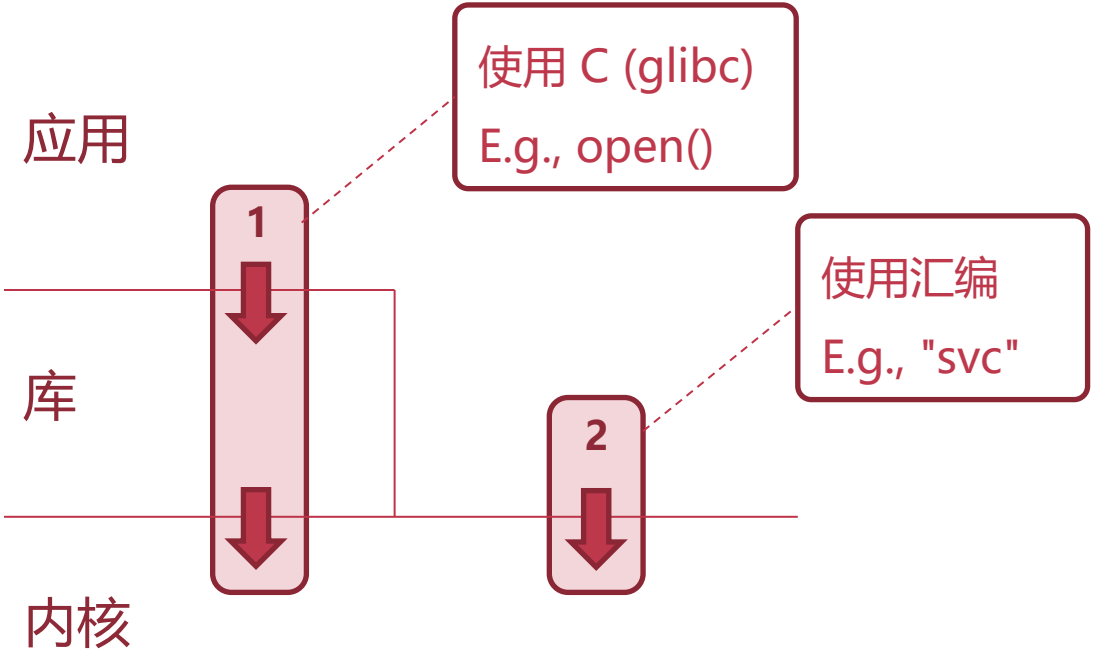
如何跟踪系统调用?

```
int main() {
  write(1, "Hello world!\n", 13);
}
```

```
$ strace -o hello.out ./hello
```

```
execve(“./hello2”, [“./hello2”], [/* 59 vars */]) = 0
…
brk(0) = 0xca9000
brk(0xcaa1c0) = 0xcaa1c0
arch_prctl(ARCH_SET_FS, 0xca9880) = 0
brk(0xccb1c0) = 0xccb1c0
brk(0xccc000) = 0xccc000
write(1, "Hello world!\n", 13) = 13
exit_group(13) = ?
```

程序员角度看系统调用



Virtual Dynamic Shared Object

VDSO

动机

- **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

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

gettimeofday

- **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?**

vDSO的共享页在哪儿?

```
$ ldd `which bash`
  linux-vdso.so.1 => (0x00007fff667ff000)
  libtinfo.so.5 => /lib/x86_64-linux-gnu/libtinfo.so.5 (0x00007f623df7d000)
  libdl.so.2 => /lib/x86_64-linux-gnu/libdl.so.2 (0x00007f623dd79000)
  libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f623d9ba000)
  /lib64/ld-linux-x86-64.so.2 (0x00007f623e1ae000)
```

The source can be found in arch/x86/vdso/vclock_gettime.c

Flexible System Call Scheduling with Exception-Less System Calls, OSDI’10

FLEX-SC

动机

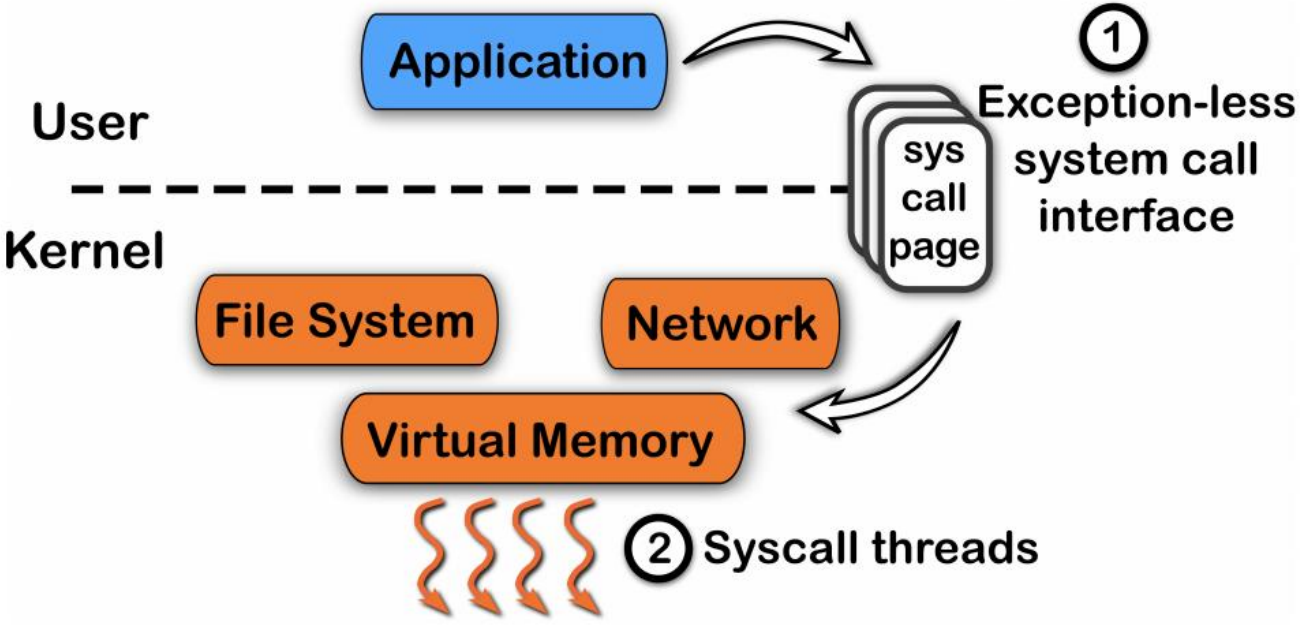
- **如何进一步降低系统调用的时延?**
  - 不仅仅是 gettimeofday()
- **"时间都去哪儿了? "**
  - 大部分是用来做状态的切换
    - 保存和恢复状态 + 权限的切换
  - Cache pollution
- **是否有可能在不切换状态的情况下实现系统调用?**

Flexible System Call

- **一种新的syscall机制**
  - 引入 **system call page**，由 user & kernel 共享
  - 应用程序可以将系统调用的请求 **push** 到 system call page
  - 内核会从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

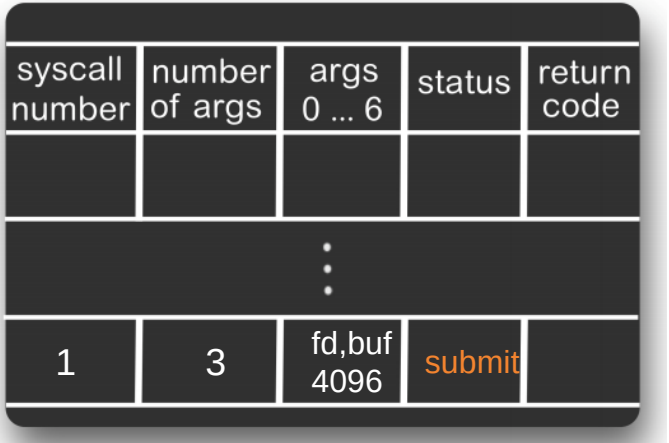
System Call的另一种方法



Exception-less System Call

```
write(fd, buf, 4096);
```

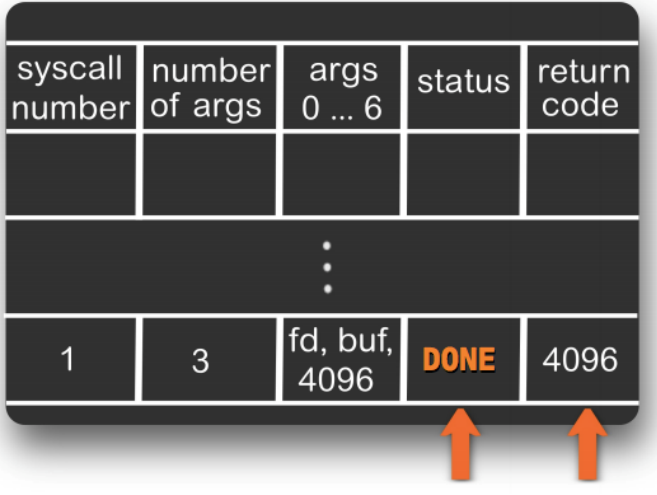
```
entry = free_syscall_entry();

/* write syscall */
entry->syscall = 1;
entry->num_args = 3;
entry->args[0] = fd;
entry->args[1] = buf;
entry->args[2] = 4096;
entry->status = SUBMIT;

while (entry->status != DONE)
   do_something_else();

return entry->return_code;
```



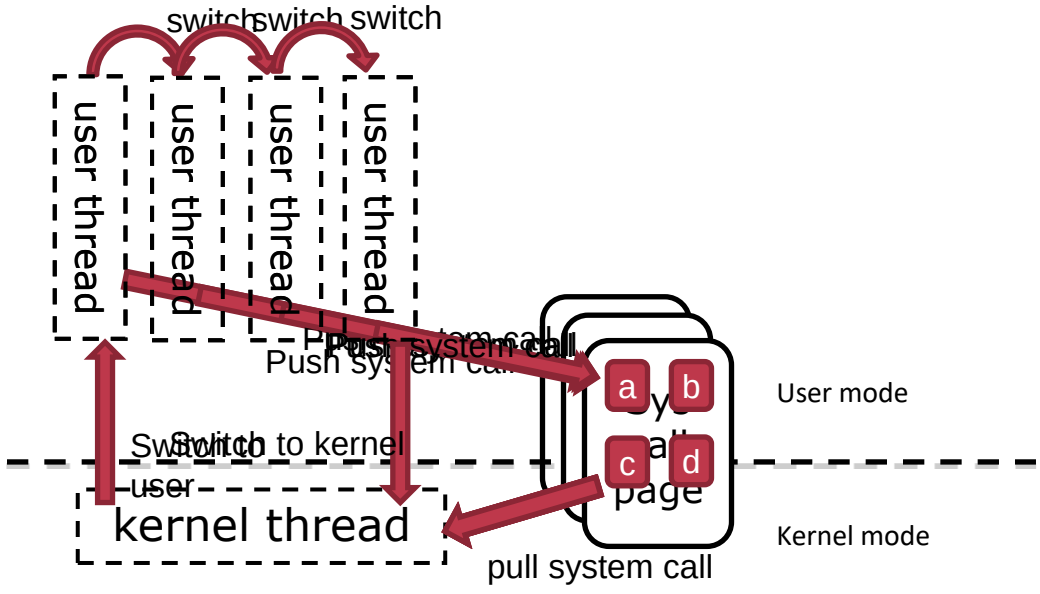
Kernel填充syscall的返回值

```
write(fd, buf, 4096);
```

```
entry = free_syscall_entry();

/* write syscall */
entry->syscall = 1;
entry->num_args = 3;
entry->args[0] = fd;
entry->args[1] = buf;
entry->args[2] = 4096;
entry->status = SUBMIT;

while (entry->status != DONE)
   do_something_else();

return entry->return_code;
```



FlexSC示例



FlexSC: Flexible System Call Scheduling with Exception-Less System Calls

小结

- **特权级EL0、EL1**
- **特权级切换**
  - 同步异常与异步异常（中断）
  - 异常处理
    - 异常处理函数表
    - 系统寄存器：vbar_el1、esr_el1、elr_el1
    - 栈切换：sp_el1和sp_el0
- **系统调用（system call）**
  - 一种特殊的同步异常
  - svc + eret