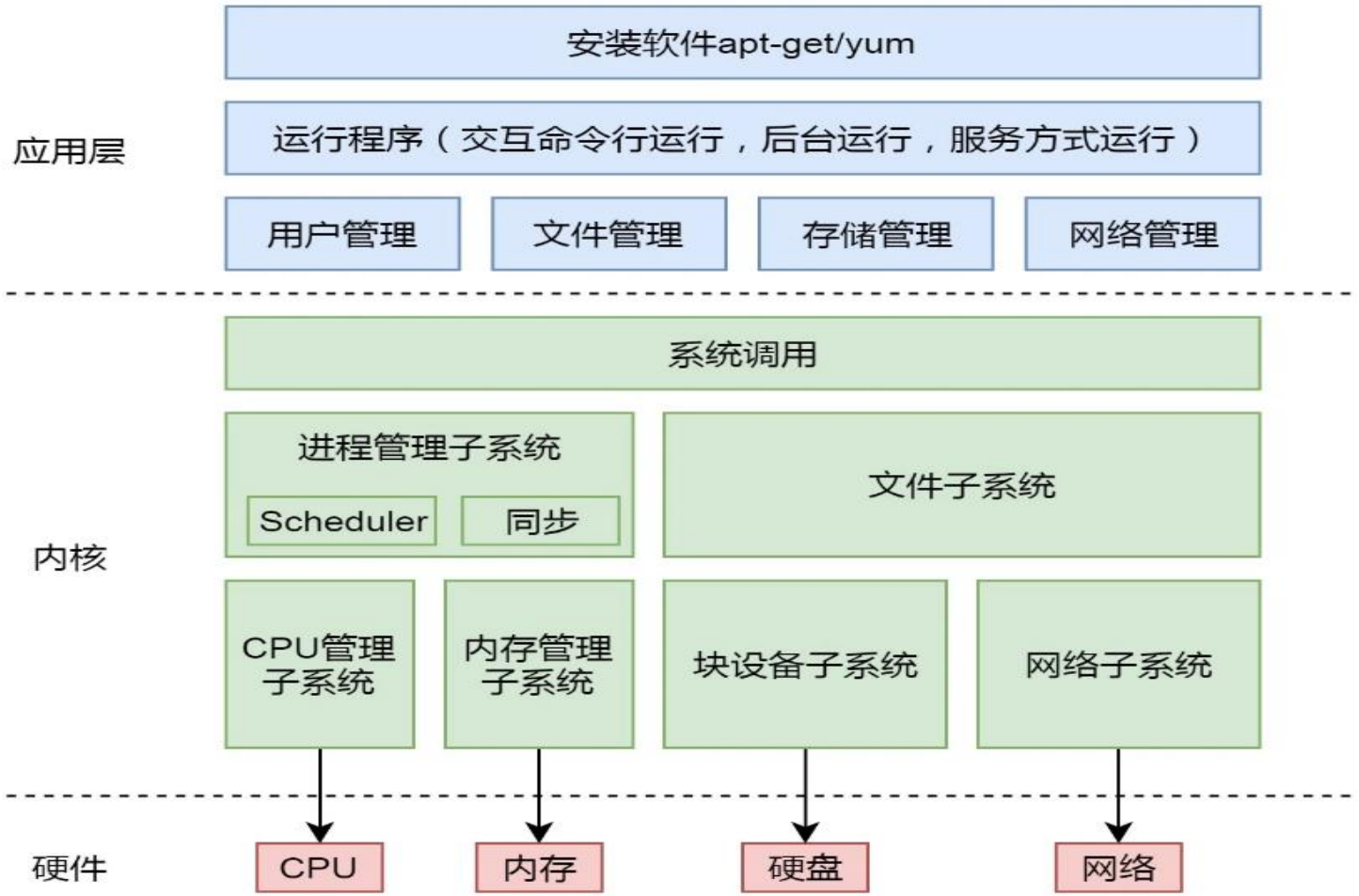
操作系统

By 公众号@极客重生

大师兄Alex

操作系统宏观视角



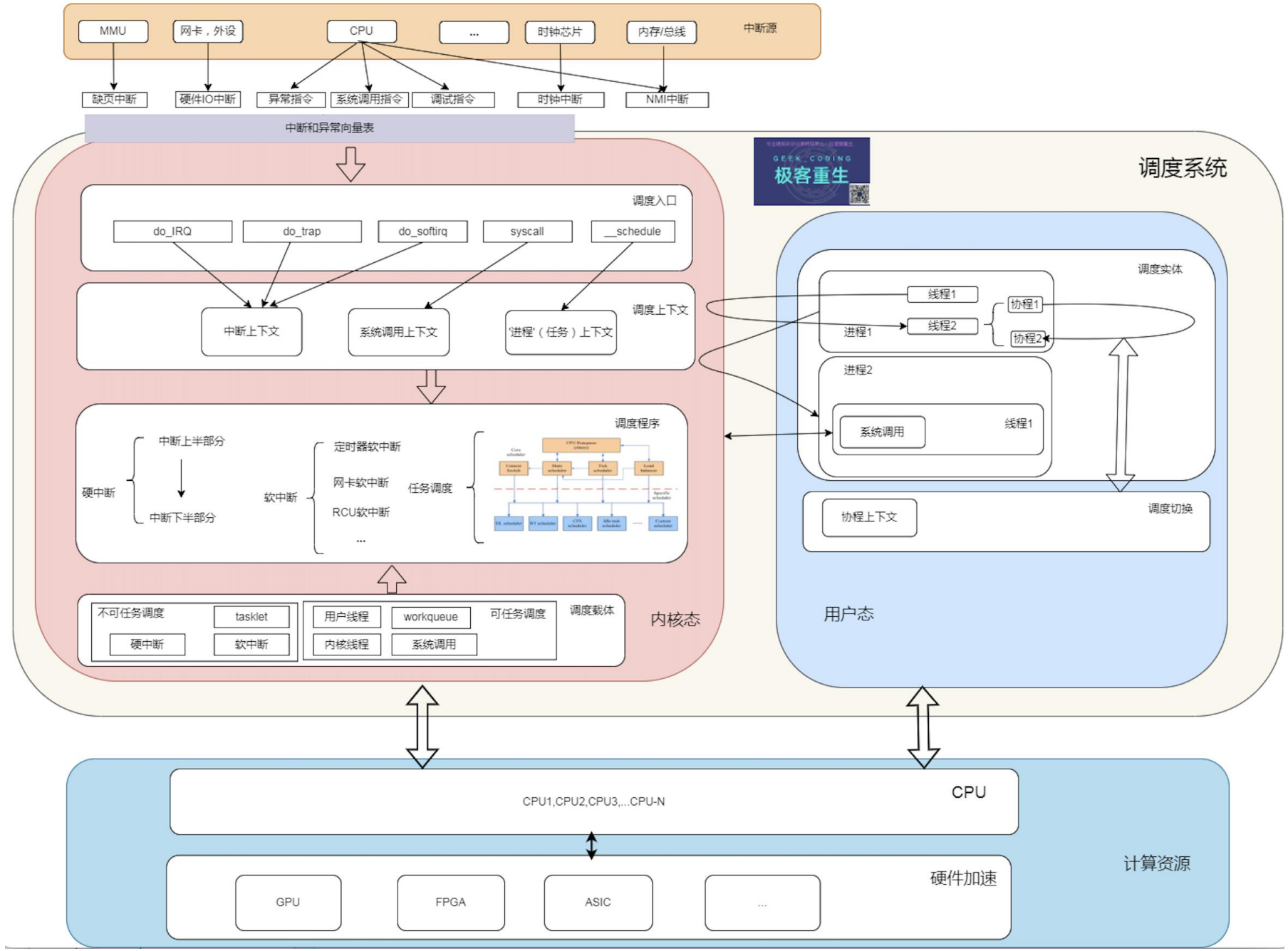
Linux操作系统

操作系统各大子系统

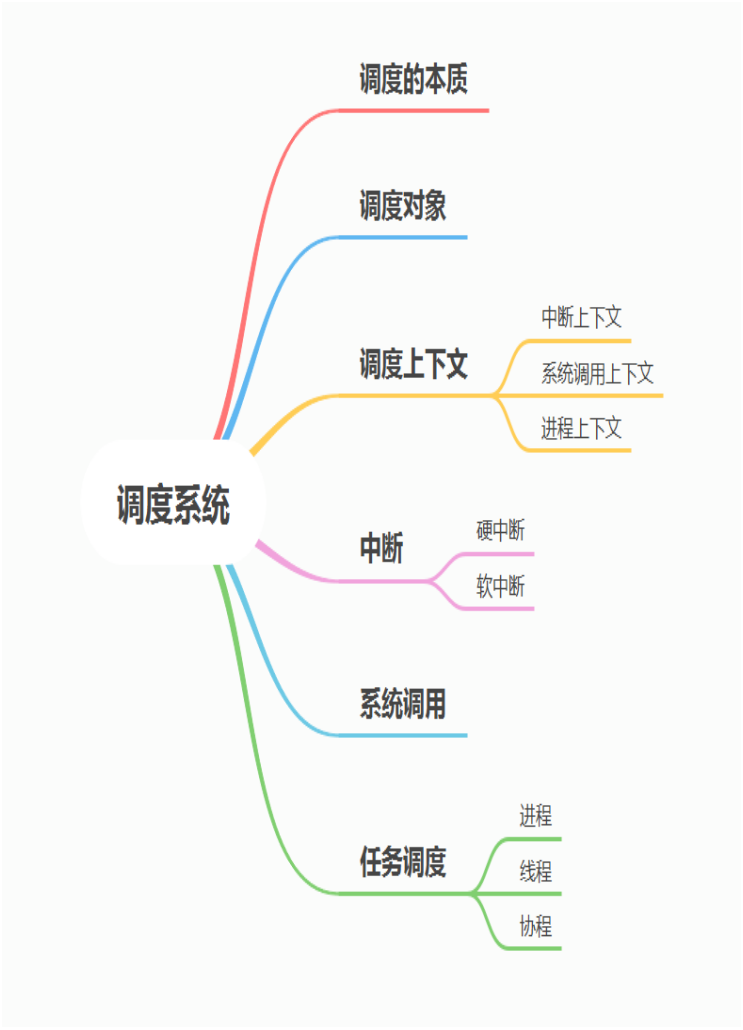
云计算开发	IaaS：虚拟化，计算，网络，存储 PaaS：虚拟化，容器，计算，网络，存储
Linux后台服务器开发	高性能，高并发
Linux嵌入式开发	内核裁剪，各种硬件接口适配，性能优化，手机，Iot
Linux SRE方向	运维，解决Linux稳定性问题。

- 调度子系统
- 内存子系统
- IO子系统
- 网络子系统
- 驱动（设备管理）子系统

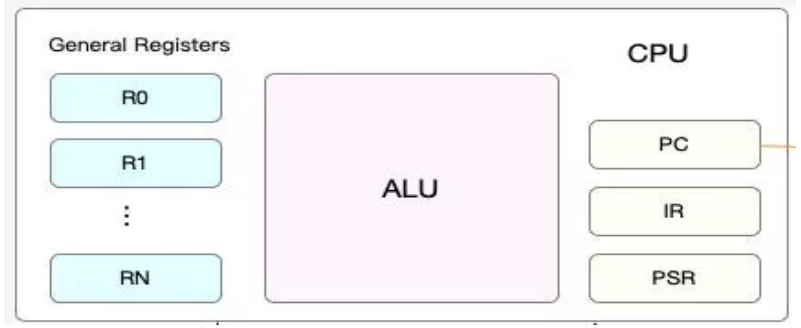
调度系统全景图


MMU
网卡，外设
CPU
...
时钟芯片
内存/总线
中断源
缺页中断
硬件IO中断
异常指令
系统调用指令
调试指令
时钟中断
NMI中断
中断和异常向量表
调度系统
GEEK CODING
极客重生
调度入口
do_IRQ
do_trap
do_softirq
syscall
__schedule
调度上下文
中断上下文
系统调用上下文
'进程'（任务）上下文
调度程序
硬中断
中断上半部分
中断下半部分
软中断
定时器软中断
网卡软中断
RCU软中断
...
任务调度
不可任务调度
tasklet
硬中断
软中断
可任务调度
用户线程
workqueue
内核线程
系统调用
调度载体
内核态
调度实体
线程1
线程2
进程1
协程1
协程2
进程2
系统调用
线程1
协程上下文
调度切换
用户态
CPU1,CPU2,CPU3,...CPU-N
CPU
GPU
FPGA
ASIC
...
硬件加速
计算资源

调度系统

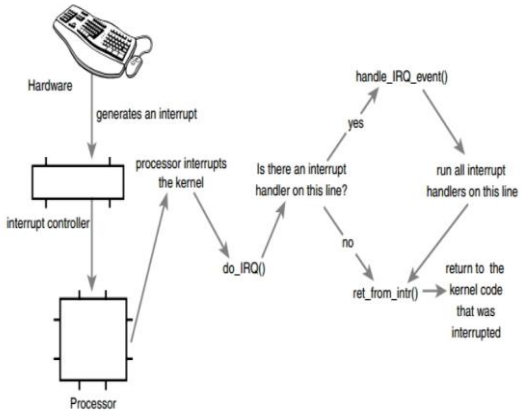


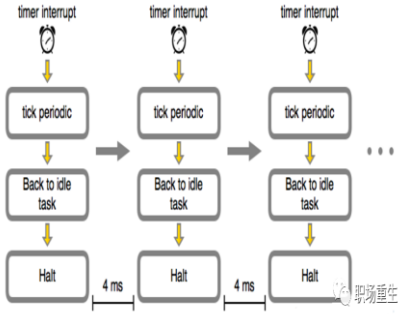
调度上下文



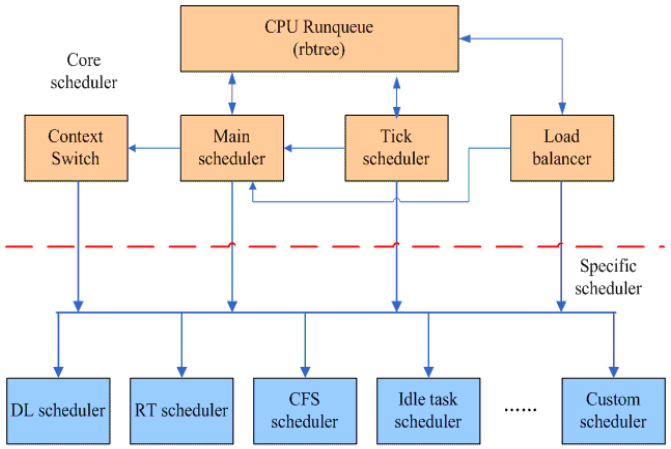
- 中断上下文
- 进程上下文
- 系统调用上下文
- 协程上下文（非内核）

CPU调度的本质





Linux任务调度框架

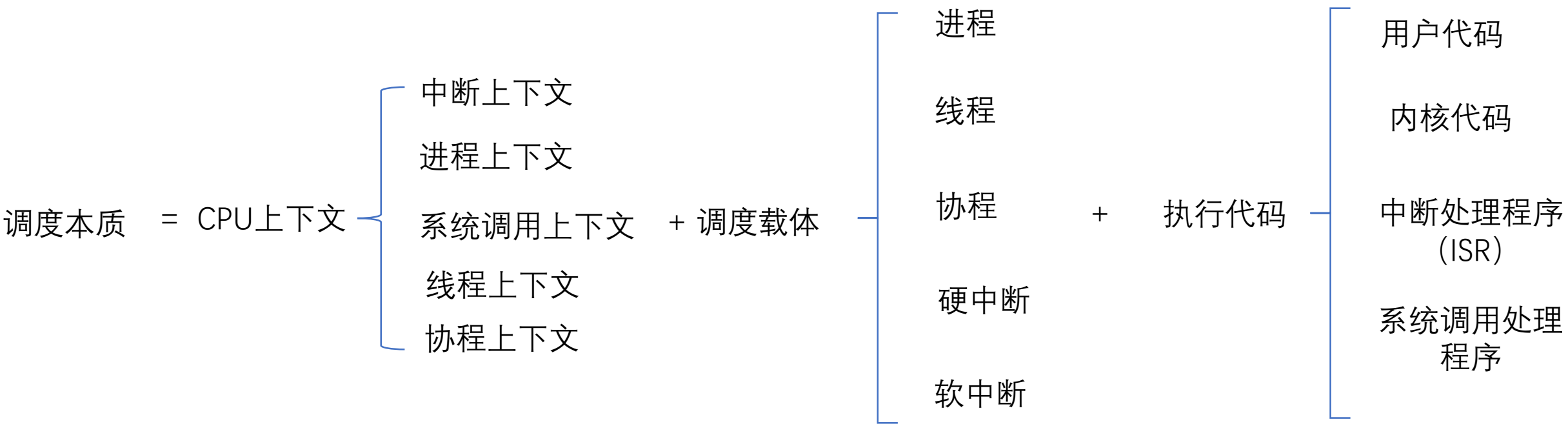


调度入口

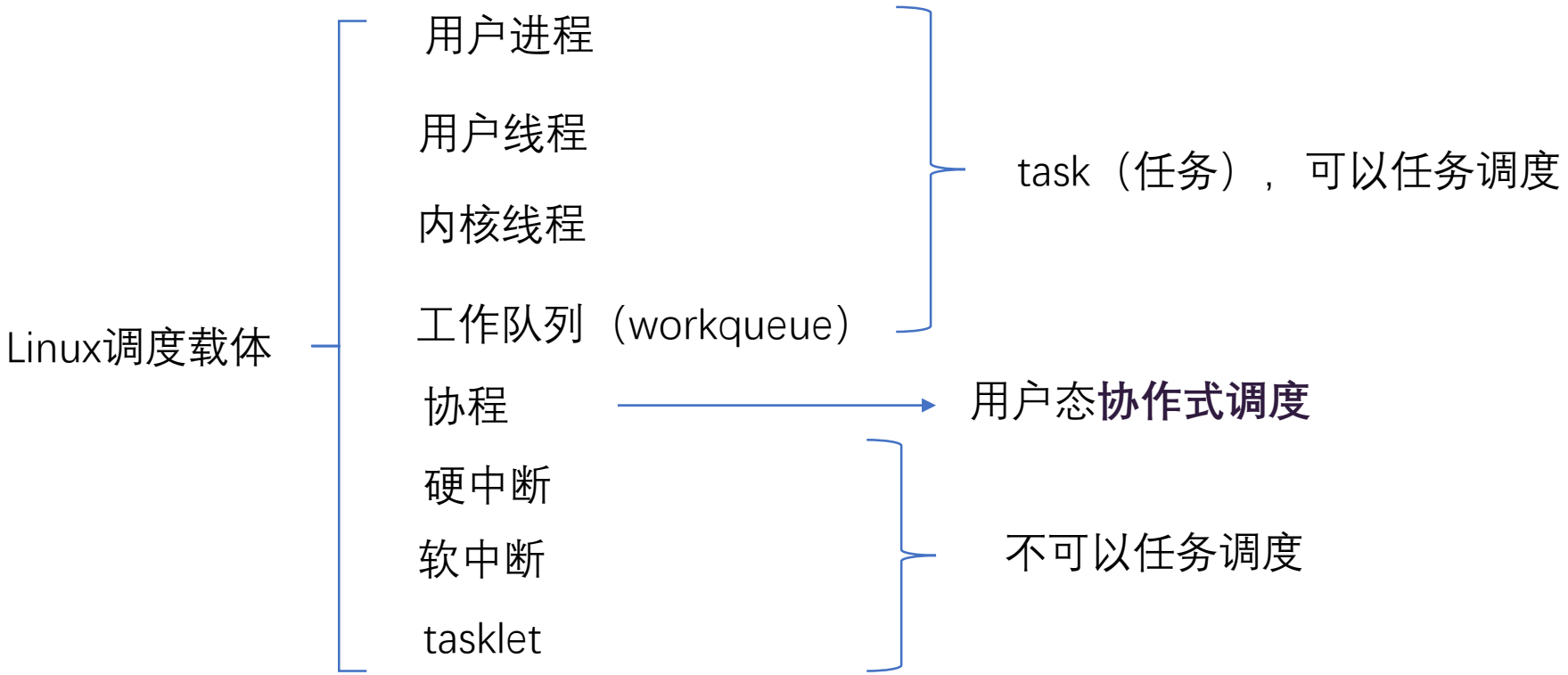
- 硬中断
- 软中断
- 系统调用
- 指令异常

调度本质

调度本质 = CPU上下文（执行环境） + 调度载体 + 执行代码



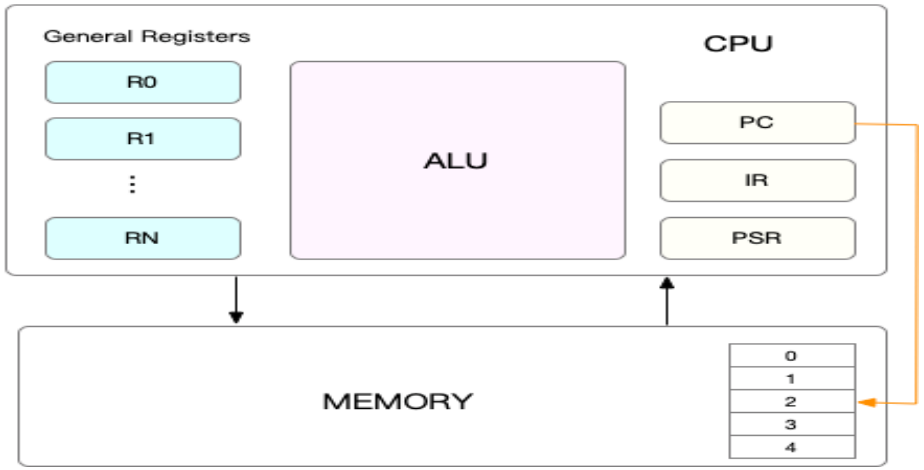
调度载体（抽象化的代码执行对象）

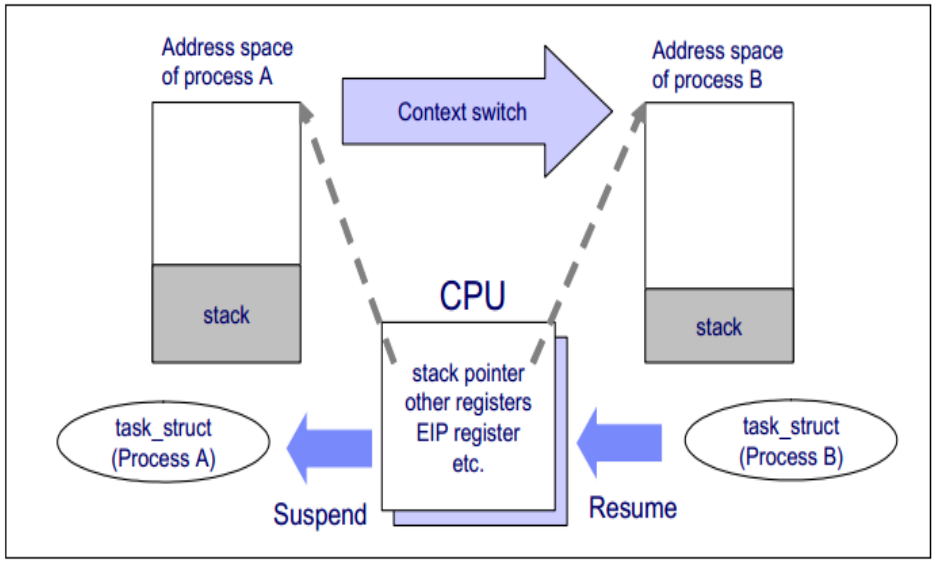


优先级顺序：硬中断 > 软中断 > 进程 > 线程 > 协程

一般切换代价： 协程 < 线程 < 系统调用 < 进程

调度上下文（抽象化代码执行上下文环境）





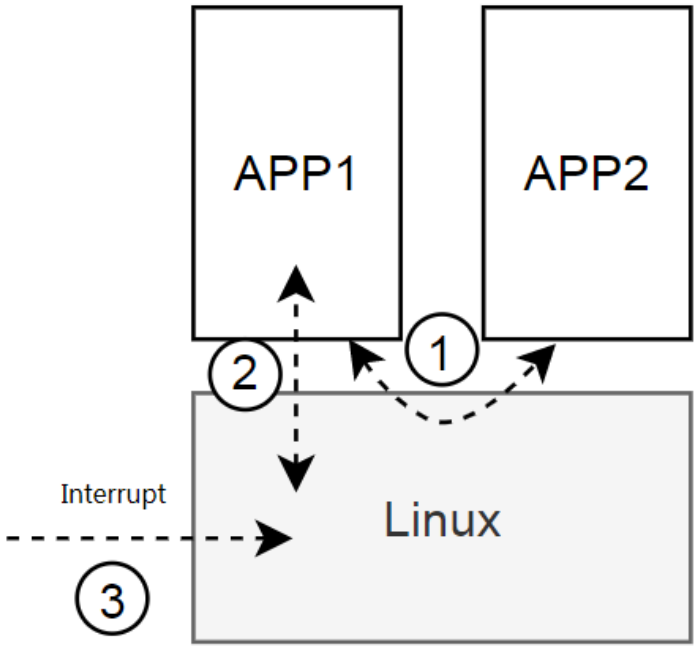
现代CPU+操作系统，其设计目标主要是为了完美高效的实现一个多任务系统，多任务系统的三个核心特征是：权限分级、数据隔离和任务切换。以X86_64架构为例，权限分级通过CPU的多模式机制和分段机制实现，数据隔离通过分页机制实现，任务切换通过中断机制和任务机制（X85 TR/TSS寄存器）实现。

CPU上下文（Context）是从CPU角度引出的，简单来说，上下文是程序（进程/中断）**运行时所需要的寄存器的最小集合**，这类寄存器代表着程序运行所需的资源，比如Intel X86架构中的通用寄存器、程序计数器PC、CR3页目录基址寄存器等。

上下文切换（Context Switching）是指程序从一种状态切换到另一种状态（例如从用户态切换到内核态），或者一个程序切换到另一个程序（例如进程切换），导致相关寄存器值的变化行为。具体来说，这种变化是指旧程序上下文相关寄存器的值被保存在内存中，新程序上下文相关寄存器的值从内存中被加载到物理寄存器中。

上下文切换

根据任务的不同，通常有进程上下文切换、系统调用上下文（从用户态到内核态的切换）、中断上下文切换三种类型。



上下文切换有性能损失吗？如何减少损失？

一. 进程上下文

不考虑多核和超线程技术，一个CPU Pipeline在任意时刻只能有一个进程在运行，进程本身是系统资源分配和调度的基本单位，为了提高CPU资源利用率，在一个进程A被阻塞时，操作系统会主动调度执行另一个进程B。在进程切换的时候，需要把上下文相关的寄存器的值保存在内存中并写入新的进程B的值，从而实现保护现场、切换上下文。

由于不同的进程采用两套不同的地址翻译映射关系，进行进程切换时需要更新页表映射寄存器、刷新TLB的。具体来说，对于AARCH64架构，除了通用寄存器以外，还需要切换TTBR0_EL1寄存器（Translation Table Base Register，页表基址）；对于X86架构，切换CR3寄存器，该寄存器用于存放页目录基址地址，不同进程CR3不同，所以也需要保存和恢复；对于RISC-V架构，页表基址地址存储在Supervisor模式的SATP寄存器的PNN位，所以也需要暂存当前进程的SATP寄存器。

二.系统调用上下文

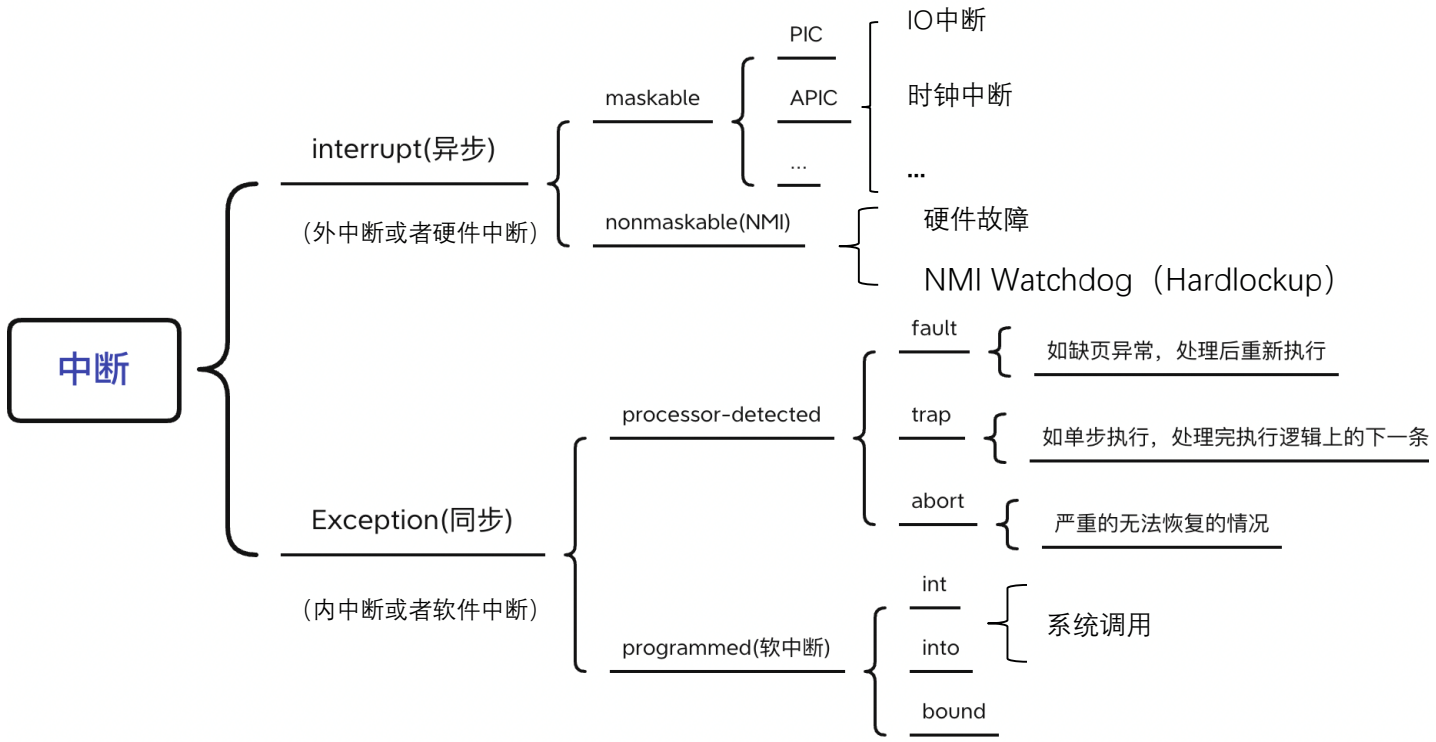
广义上这里的指的是特权模式的切换，如X86不同的Ring，以从用户态陷入内核态的系统调用为例，因为用户态和内核态特权模式的代码对系统资源的访问权限不同，也需要切换部分上下文，如程序计数器、栈指针、页表基址地址等。

需要注意的是，对于系统调用时页表切换，不同架构有所不同。X86架构中，应用程序和操作系统共用一套页表，操作系统是把自己映射到应用程序页表的高地址部分，从而系统调用时无须切换页表，避免了TLB的冲刷；

三. 中断上下文

中断处理函数运行在特殊的上下文中，叫做中断上下文。当CPU处理一个中断时，如按键盘时中断控制器会给CPU发送信号执行中断处理程序，此时不管当前CPU是在运行一个进程，甚至还是在处理另一个中断，会切换到新中断的上下文中执行，此时需要将原始程序的执行状态进行保存，在去执行新的中断处理程序，中断处理完成之后，恢复先前被打断的原始程序的执行。

中断上下文—中断



中断分类：

中断可分为同步（*synchronous*）中断和异步（*asynchronous*）中断：

- 同步中断是当指令执行时由 CPU 控制单元主动产生，之所以称为同步，是因为只有在一条指令执行完毕后 CPU 才会发出中断，而不是发生在代码指令执行期间，比如系统调用，根据 Intel 官方资料，同步中断称为异常（*exception*），异常可分为故障（*fault*）、陷阱（*trap*）、终止（*abort*）三类。
- 异步中断是指由其他硬件设备依照 CPU 时钟信号随机产生，即意味着中断能够在指令之间发生，例如键盘中断，异步中断被称为中断（*interrupt*），中断可分为可屏蔽中断（*Maskable interrupt*）和非屏蔽中断（*Nomaskable interrupt*）。

1. 非屏蔽中断(Non-maskable interrupts,即NMI)：就像这种中断类型的字面意思一样，这种中断是不可能被CPU忽略或取消的。NMI是在单独的中断线路上进行发送的，它通常被用于关键性硬件发生的错误，如内存错误，风扇故障，温度传感器故障等。
2. 可屏蔽中断（Maskable interrupts）：这些中断是可以被CPU忽略或延迟处理的。当缓存控制器的外部针脚被触发的时候就会产生这种类型的中断，而中断屏蔽寄存器就会将这样的中断屏蔽掉。我们可以将一个比特位设置为0，来禁用在此针脚触发的中断。

中断代码运行于内核空间，中断上下文即运行中断代码所需要CPU上下文环境（寄存器），硬件传递过来的**硬件中断参数**，内核需要保存的一些其他环境（主要是当前被打断执行的进程或其他中断环境），这些一般都保存在中断栈中（x86是独立的，其他可能和内核栈共享，这和具体处理架构密切相关），在中断结束后，进程仍然可以从原来的状态恢复运行。

CPU处理优先级：kernel线程 < 时钟中断 < NMI中断

中断上下文—中断处理

中断处理

中断处理上半场（top half）

CPU 硬件处理：

- CPU检查如果当前是用户态，会切换到内核态， CPU自动将用户态SS、rsp、rflags、CS、rip压入栈中。
- CPU根据寄存器 IDTR 记录的基址和中断控制器的寄存器 ISR 提供的中断向量号找到 IDT 表中对应的 Interrupt Gate 表项，然后跳转到相应的中断向量地址（这些中断处理函数地址是操作系统初始化时候放好的（内存)），将CS:rip指向中断描述符中的函数入口地址）。
- 在中断处理函数里面会保存部分参数寄存器，然后会跳到操作系统内部 do_IRQ入口函数处理。
- CPU同时会屏蔽该中断。

中断处理函数在：arch/x86/kernel/entry_64.S。

OS 处理：关中断条件下的中断处理 do_IRQ
(arch/x86/kernel/irq.c)

CPU处理：中断返回ret_from_intr：

- 如果是在内核态，就跳转到retint_kernel执行，这里会根据内核是否打开抢占进行不同的处理：
- 如果内核不可抢占，那就恢复寄存器后返回到被中断的上下文继续执行；如果是可抢占的，那就可以进行调度。
- 如果是在用户态，就进行调度及信号相关的判断和处理；处理完成并恢复寄存器后，
  便通过iretq指令返回被中断的上下文继续执行。

指令在：arch/x86/kernel/entry_64.S

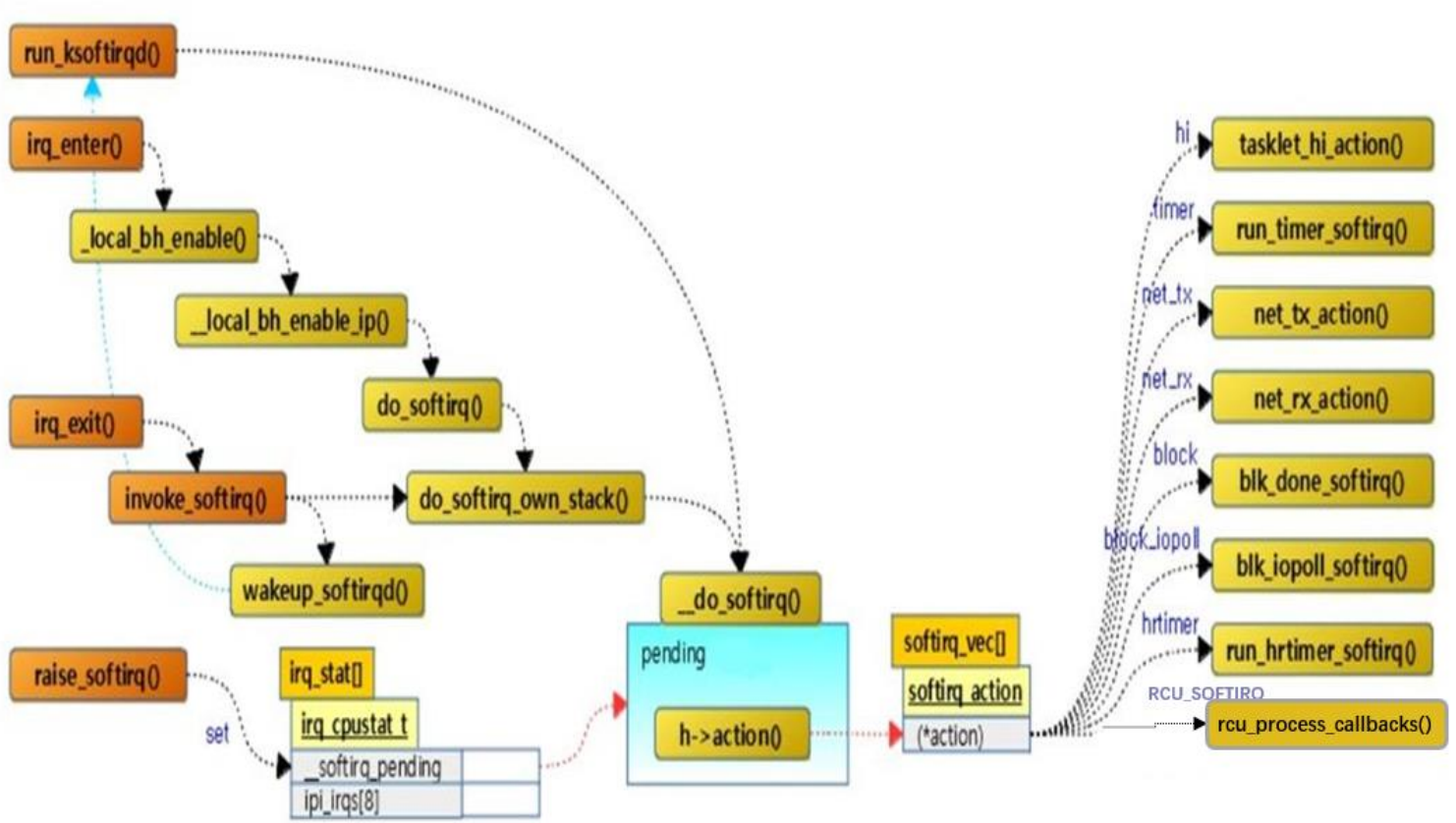
中断处理下半场（bottom half）

软中断（中断耗时部分延后处理）

Linux中断处理为什么两会有个部分？

中断上下文— 中断下半场

中断下半场
- Softirq --> ksoftirqd
- tasklet
- workqueue



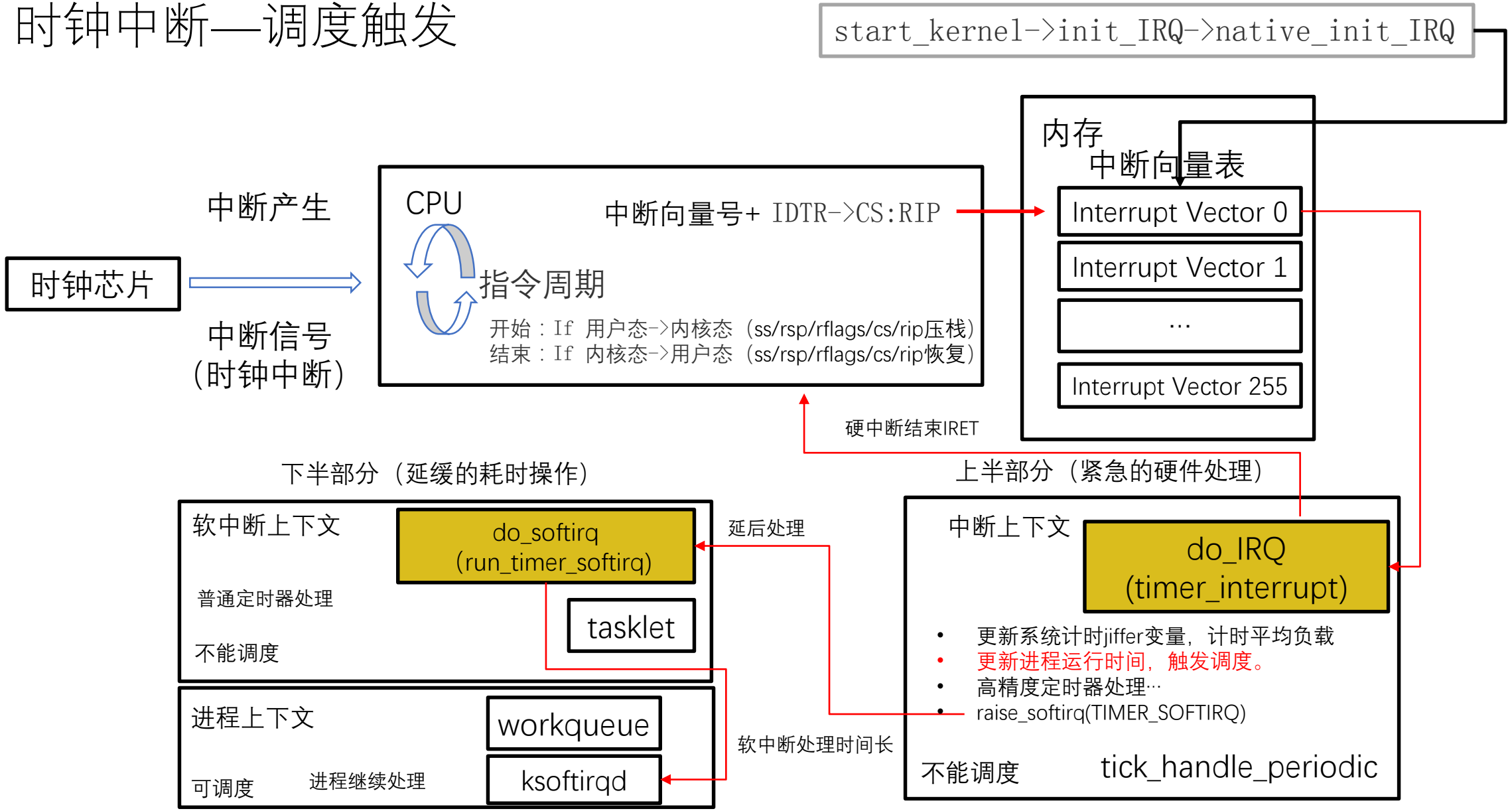
Softirq处理

上面各自适合什么场景？

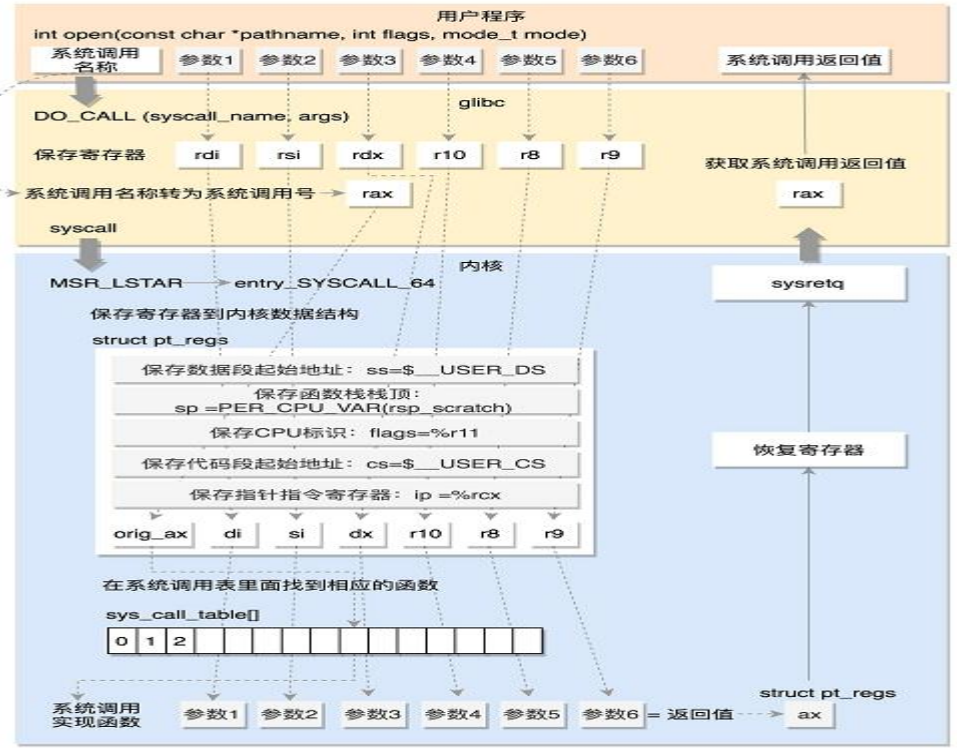
软中断的调度时机？

1. do_irq完成I/O中断时调用irq_exit。
2. 系统使用I/O APIC，在处理完本地时钟中断时。
3. local_bh_enable，即开启本地软中断时。
4. ksoftirqd/n线程被唤醒时。

时钟中断—调度触发


start_kernel->init_IRQ->native_init_IRQ
内存
中断向量表
Interrupt Vector 0
Interrupt Vector 1
…
Interrupt Vector 255
中断产生
时钟芯片
中断信号
(时钟中断)
CPU
中断向量号+ IDTR->CS:RIP
指令周期
开始：If 用户态->内核态（ss/rsp/rflags/cs/rip压栈）
结束：If 内核态->用户态（ss/rsp/rflags/cs/rip恢复）
硬中断结束IRET
下半部分（延缓的耗时操作）
上半部分（紧急的硬件处理）
软中断上下文
do_softirq
(run_timer_softirq)
普通定时器处理
tasklet
不能调度
延后处理
中断上下文
do_IRQ
(timer_interrupt)
更新系统计时jiffer变量，计时平均负载
更新进程运行时间，触发调度。
高精度定时器处理…
raise_softirq(TIMER_SOFTIRQ)
进程上下文
workqueue
ksoftirqd
可调度
进程继续处理
软中断处理时间长
不能调度
tick_handle_periodic

系统调用上下文




```
SYSCALL_DEFINE3(open, const char __user *, filename, int,
flags, umode_t, mode) {
  if (force_o_largefile()) flags |= O_LARGEFILE;
        return do_sys_open(AT_FDCWD, filename, flags, mode);
}
```

fs/open.c

系统调用OPEN流程

用户态进入内核态的方式还有哪些？

从用户态切换内核态

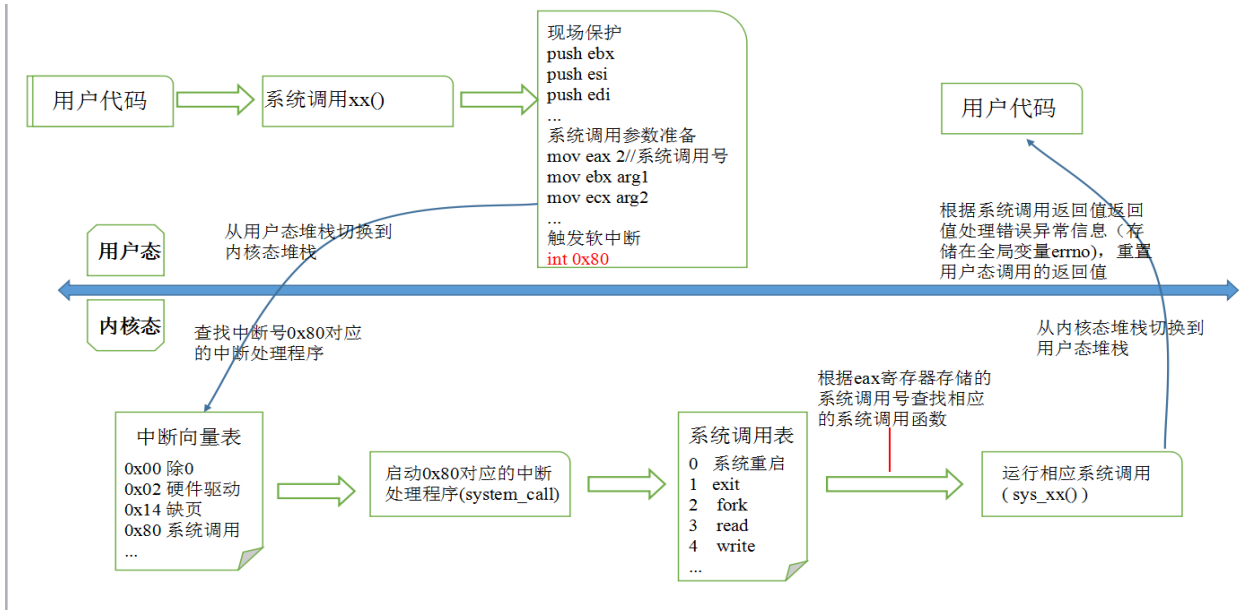
- 切换时先保存CPU寄存器中的用户态指令
- 栈切换到内核栈
- 再重新更新内核态指令位置
- 最后跳转到内核态运行内核任务
- 当系统调用结束后需要恢复原来保存的用户态

Linux 用户态和内核态区别？

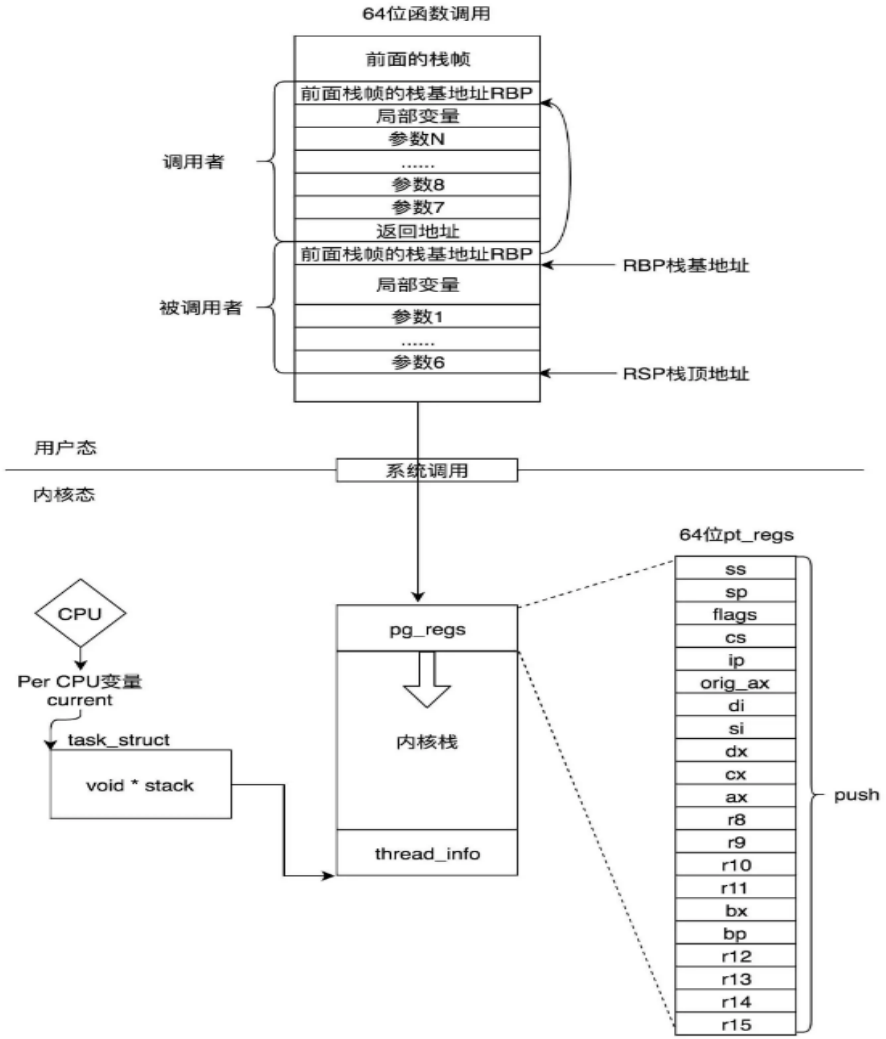
Linux所谓的**用户态**和**内核态**，是Linux为了有效实现CPU的权限分级和数据隔离的目标而出现的，是通过组合CPU的分段机制+分页机制而形成的。本质是对CPU提供的功能的一层封装抽象。

进程既可以在用户空间运行，又可以在内核空间中运行。进程在用户空间运行时，被称为进程的用户态，而陷入内核空间的时候，被称为进程的内核态。

系统调用上下文



系统调用简化流程



你了解哪些可以加速系统调用技术？

1 快速系统调用指令(syscall/sysret)
2 用户空间系统调用(vdso)
...

进程上下文

进程的上下文包含哪些内容？

进程既可以在用户空间运行，又可以在内核空间中运行。进程在用户空间运行时，被称为进程的用户态，而陷入内核空间的时候，被称为进程的内核态。

进程的上下文不仅包括了虚拟内存、栈、全局变量等用户空间的资源，还包括了内核堆栈、寄存器等内核空间的状态。

进程和线程区别和联系？

进程上下文—任务调度

标记进程是否调度

时钟中断

更新进程运行时间，标记调度

task

进程唤醒，标记调度

设置任务可调度

TIF_NEED_RESCHED

真正抢占点

- 主动让出

- 主动调用sleep
- 主动调用schedule
- 或者进程阻塞

- 被动抢占

- **从系统调用返回用户态**
  syscall_return_sloowpath调用schedule
- **从中断返回用户态**
  do_IRQ后retint_user调用schedule
- **内核启动可抢占**（内核开启抢占）
  preempt_enable调用schedule
- **从中断返回内核态**
  do_IRQ后retint_kernel调用schedule

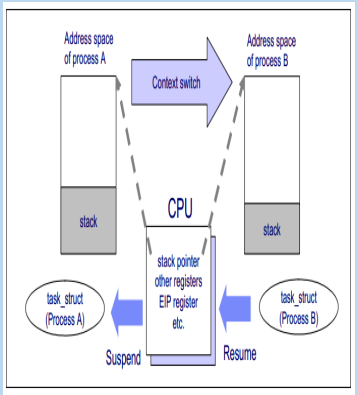
调度入口

```
schedule() {
    pick_next_task() {
        ...
        RT调度类
        在线  CFS调度类
        IDLE
    }
    context_switch();
}
```

Linux 调度器框架

主要就是调度器、策略及优先级。调度就是按照某种算法，从进程的就绪队列中选取一个进程放到cpu上运行。目的是协调各进程对cpu等资源的合理使用。

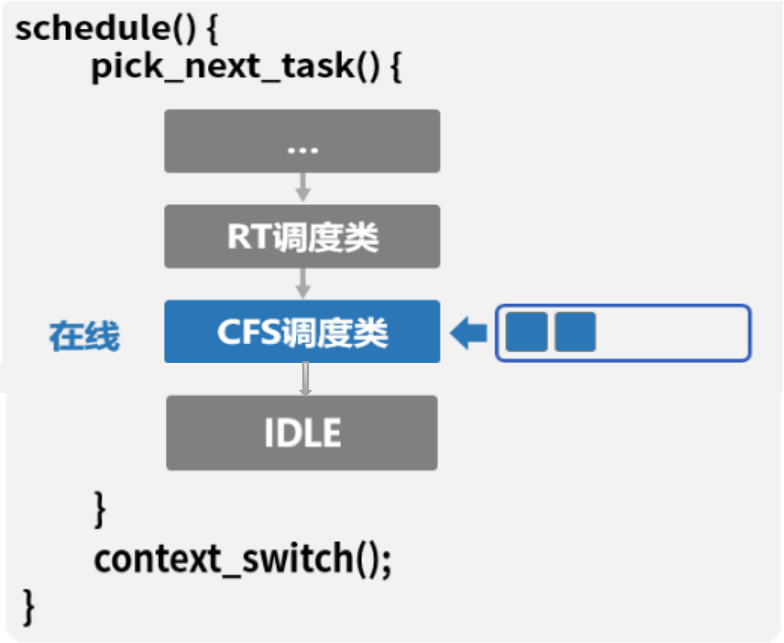
调度切换

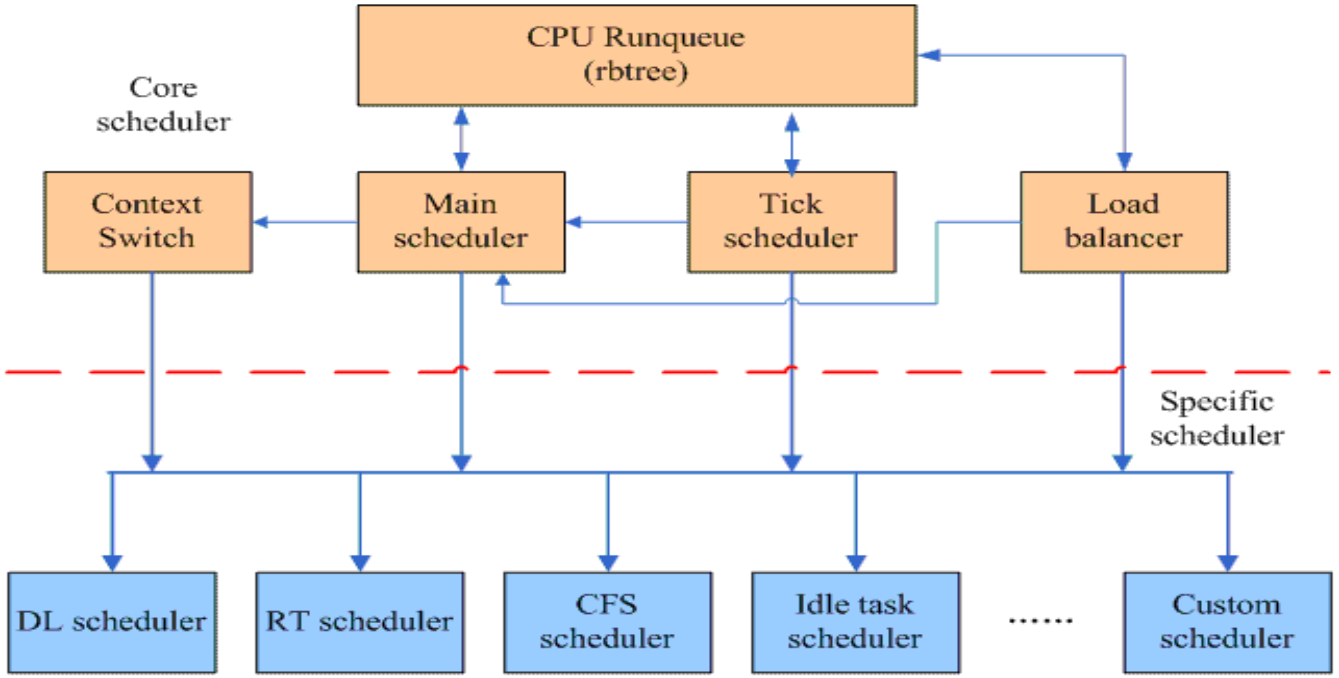


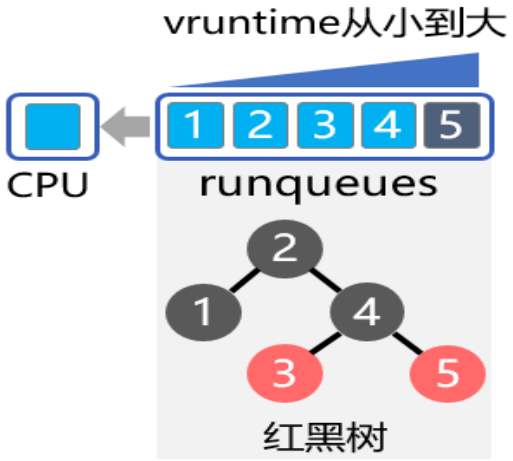
进程上下文切换：把CPU当前任务切换为下一个任务运行

非抢占式内核主要用于服务器等对吞吐量要求较高的场景，而抢占式内核主要用于嵌入式设备和桌面等对响应要求较高的场景。

进程上下文—任务调度



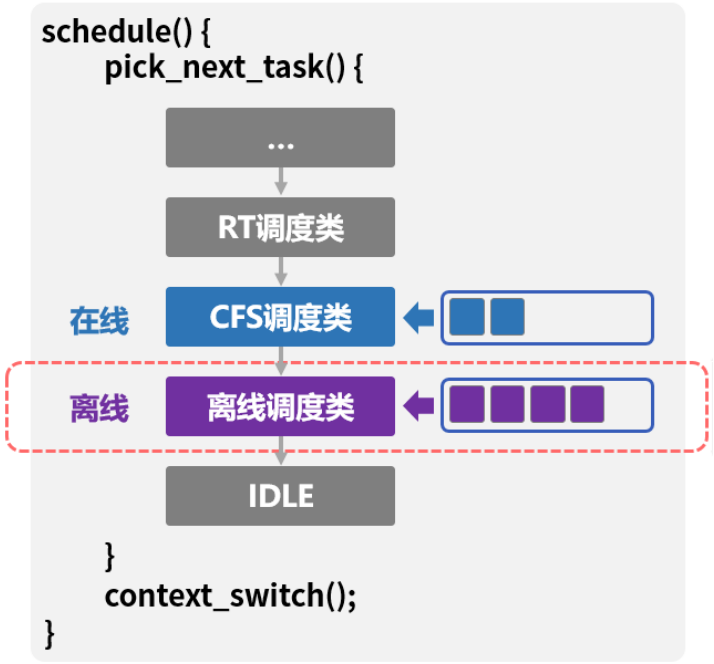


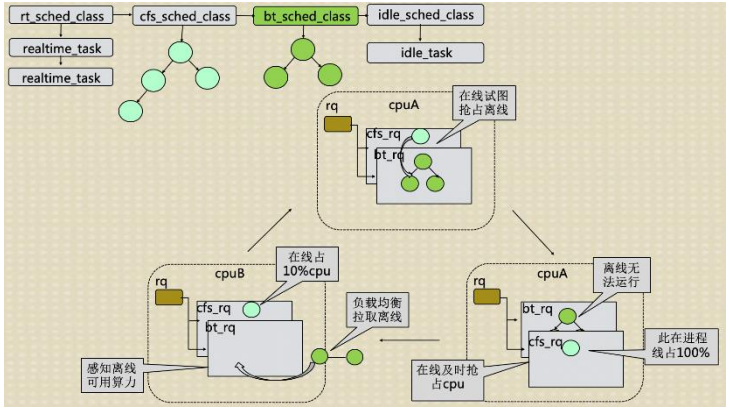


CFS调度策略在选择任务执行时的基本逻辑很简单，一般情况就是选择位于运行队列 *(run queue)* 上的第一个任务。在运行队列中，各个任务按照 *vruntime* 值的大小从小到大排列(用红黑树维护以vruntime为排序条件)，即vruntime越小的进程，越靠近队首。由于任务的调度实体（sched_entity）会频繁地进出运行队列。因此运行队列中任务的调度实体按照红黑树方式进行组织。

CFS的vruntime += 处理器运行时间 * nice对应的权重

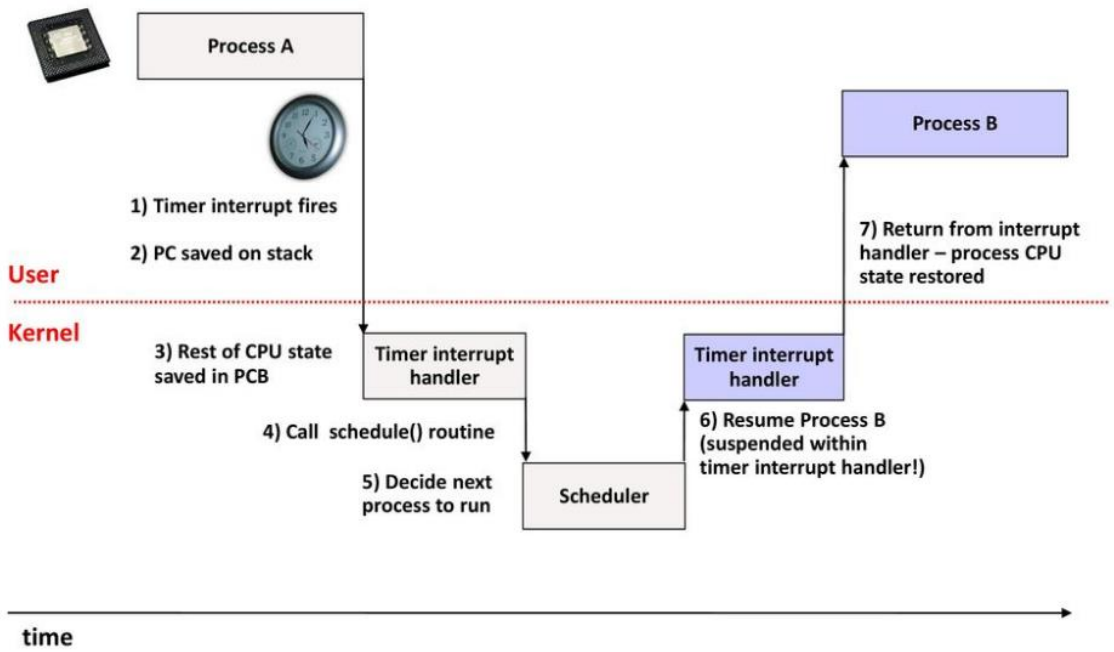
进程上下文—实现特殊任务调度算法






```c
struct sched_class {
    // 调度类是一个链表，按照优先顺序排列，next执行下一个调度类
    const struct sched_class *next;
    // 添加任务
    void (*enqueue_task) (struct rq *rq, struct task_struct
*p, int flags);
    // 移除任务
    void (*dequeue_task) (struct rq *rq, struct task_struct
*p, int flags);
    // 校验是否当前任务应该被抢占
    void (*check_preempt_curr)(struct rq *rq, struct
task_struct *p, int flags);
    // 或取下一个待执行的任务
    struct task_struct * (*pick_next_task)(struct rq *rq,
                                           struct task_struct *prev,
                                           struct rq_flags *rf);
    void (*put_prev_task)(struct rq *rq, struct task_struct
*p);
    void (*set_curr_task)(struct rq *rq);
    // 时钟中断处理
    void (*task_tick)(struct rq *rq, struct task_struct *p,
int queued);
    void (*switched_from)(struct rq *this_rq, struct
task_struct *task);
    void (*switched_to)  (struct rq *this_rq, struct
task_struct *task);
  ......
};
const struct sched_class bt_sched_class = {
    .next                 = &idle_sched_class,
    .enqueue_task         = enqueue_task_bt,
    .dequeue_task         = dequeue_task_bt,
    .check_preempt_curr   = check_preempt_wakeup_bt,
    .pick_next_task       = pick_next_task_bt,
    .put_prev_task        = put_prev_task_bt,
    .set_curr_task        = set_curr_task_bt,
    .task_tick            = task_tick_bt,
    .switched_from        = switched_from_bt,
    .switched_to          = switched_to_bt,
    ......
};
```

进程上下文—任务切换



进程的上下文切换：在保存内核态资源（当前进程的内核状态和 CPU 寄存器）之前，需要先把该进程的用户态资源（虚拟内存、栈等）保存下来；而加载了下一进程的内核态后，还需要刷新进程的虚拟内存和用户栈。

- **进程地址空间切换（具体做什么？）**
- **处理器状态（CPU上下文）切换**

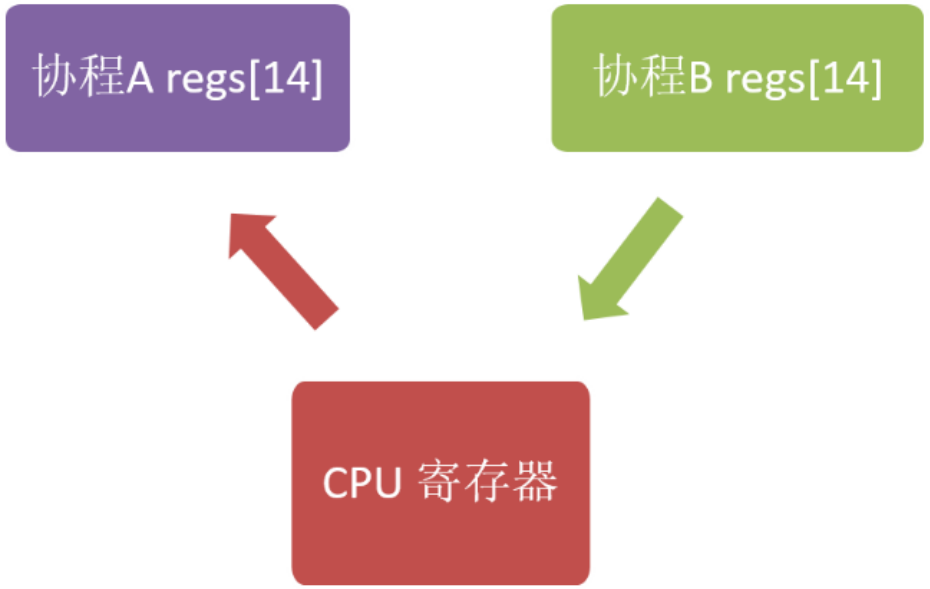
进程是由内核来管理和调度的，进程的切换只能发生在内核态。

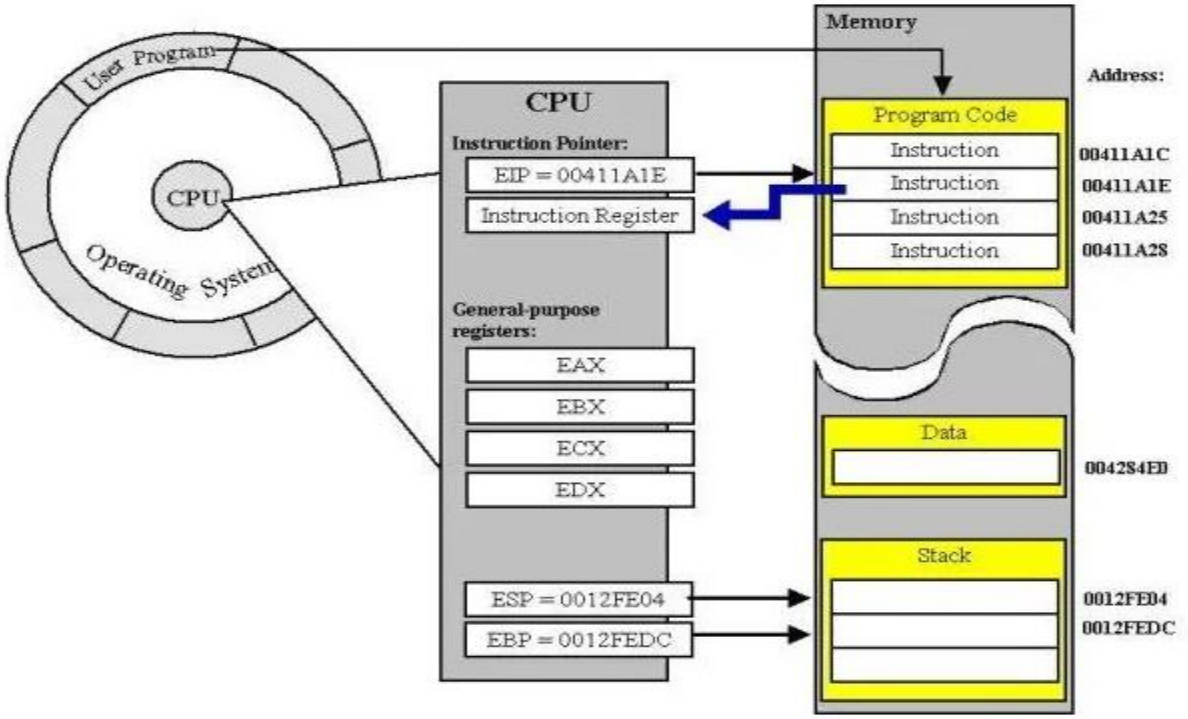
普通用户进程、普通用户线程、内核线程切换的差别？

有哪些情况可以触发任务切换？

- 时间片耗尽：为了保证所有进程可以得到公平调度，CPU 时间被划分为一段段的时间片，这些时间片再被轮流分配给各个进程。这样，当某个进程的时间片耗尽了，就会被系统挂起，切换到其它正在等待 CPU 的进程运行。
- 进程IO阻塞：要等到资源满足后才可以运行，这个时候进程也会被挂起，并由系统调度其他进程运行。
- 主动退出：当进程通过睡眠函数 sleep 这样的方法将自己主动挂起时，自然也会重新调度。
- 抢占开启：当有优先级更高的进程运行时，为了保证高优先级进程的运行，当前进程会被挂起，由高优先级进程来运行。
- 中断优先：发生硬件中断时，CPU上的进程会被中断挂起，转而执行内核中的中断服务程序。

协程上下文





协程上下文是什么?

协程切换只涉及基本的CPU上下文切换（CPU寄存器），完全在用户空间进行，没有模式切换，所以比线程切换要小，开源libco的协程切换的汇编代码，也就是二十来条汇编指令，

协程切换代价如何？　一般切换代价：协程 < 线程 < 系统调用 < 进程

协程有哪些缺点？