切换执行环境

引入

采用隔离的保护机制时，需要创建一个隔离的执行环境，涉及到上下文切换等一系列知识点。

流程梳理

一个独立的执行环境需要什么?

- 内存空间（数据）——创建分配管理页、堆栈等
- 执行空间（代码）——切换执行流
- 其它切换伴随的保护机制——中断等

首先，简单介绍一下几种常见的上下文切换。他们对于内存的管理和执行流的管理具体是如何操作的。然后介绍一下内核修改过程中可能会触及到的保护机制以及相应的处理方式。最后总结一下创建自己的隔离的执行环境需要怎么操作。

上下文切换

我们都知道，Linux 是一个**多任务操作系统**，它支持远大于 CPU 数量的任务同时运行。当然，这些任务实际上并不是真的在同时运行，而是因为系统在很短的时间内，将 CPU 轮流分配给它们，造成多任务同时运行的错觉。

而在每个任务运行前，CPU 都需要知道任务从哪里加载、又从哪里开始运行，也就是说，需要系统事先帮它设置好 **CPU 寄存器和程序计数器**（Program Counter，PC）。

CPU 寄存器，是 CPU 内置的容量小、但速度极快的内存。而程序计数器，则是用来存储 CPU 正在执行的指令位置、或者即将执行的下一条指令位置。它们都是 CPU 在运行任何任务前，必须的依赖环境，因此也被叫做 **CPU 上下文**。

CPU 上下文切换，就是先把前一个任务的 CPU 上下文（也就是 CPU 寄存器和程序计数器）保存起来，然后加载新任务的上下文到这些寄存器和程序计数器，最后再跳转到程序计数器所指的新位置，运行新任务。

问题来了——我猜肯定会有人说，CPU 上下文切换无非就是更新了 CPU 寄存器的值嘛，但这些寄存器，本身就是为了快速运行任务而设计的，为什么会影响系统的 CPU 性能呢?

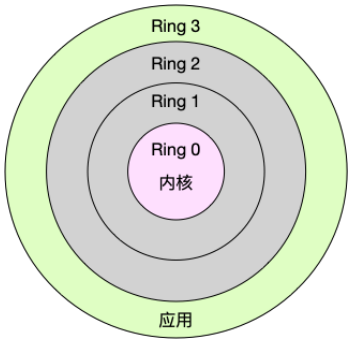
在回答这个问题前，不知道你有没有想过，操作系统管理的这些"**任务**"到底是什么呢?

- 任务
    - **进程/线程**——最常见的任务
    - 硬件通过触发信号，会导致**中断处理程序**的调用，也是一种常见的任务

所以，根据任务的不同，CPU 的上下文切换就可以分为几个不同的场景，也就是**进程上下文切换**、**线程上下文切换**以及**中断上下文切换**。

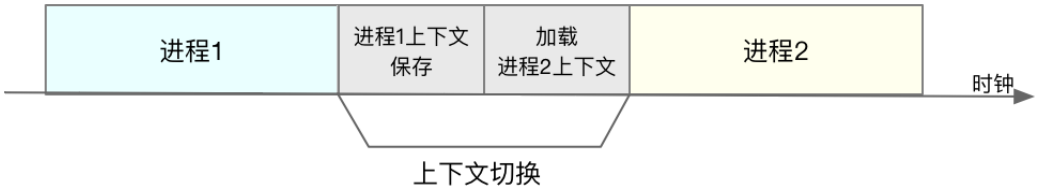
进程上下文切换

- cpu特权等级
  - 内核空间（Ring 0）具有最高权限，可以直接访问所有资源；
  - 用户空间（Ring 3）只能访问受限资源，不能直接访问内存等硬件设备，必须通过系统调用陷入到内核中，才能访问这些特权资源。



- 系统调用（用户态与内核态切换）
  - 从用户态到内核态的转变，需要通过**系统调用**来完成。**比如，**当我们查看文件内容时，就需要多次系统调用来完成：首先调用 open() 打开文件，然后调用 read() 读取文件内容，并调用 write() 将内容写到标准输出，最后再调用 close() 关闭文件。
  - 系统调用的过程有没有**发生 CPU 上下文的切换**呢？是的——CPU 寄存器里原来用户态的指令位置，需要先保存起来。接着，为了执行内核态代码，CPU 寄存器需要更新为内核态指令的新位置。最后才是跳转到内核态运行内核任务。而系统调用结束后，CPU 寄存器需要**恢复**原来保存的用户态，然后再切换到用户空间，继续运行进程。所以，一次系统调用的过程，其实是发生了**两次 CPU 上下文切换**。
  - 系统调用过程中，并**不会涉及到虚拟内存等进程用户态的资源**，也不会切换进程。所以，**系统调用过程通常称为特权模式切换，而不是上下文切换**。但实际上，系统调用过程中，CPU 的上下文切换还是无法避免的。
- 进程上下文切换
  - 进程上下文切换和系统调用区别：
    - 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。所以，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**。

      因此，进程的上下文切换就比系统调用时多了一步：在保存当前进程的内核状态和 CPU 寄存器之前，需要先**把该进程的虚拟内存、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。
    - 如下图所示，保存上下文和恢复上下文的过程并不是“免费”的，需要内核在 CPU 上运行才能完成。



  - 开销

        - 根据 Tsuna 的测试报告，每次上下文切换都需要几十纳秒到数微秒的 CPU 时间。这个时间还是相当可观的，特别是在进程上下文切换次数较多的情况下，很容易导致 CPU 将大量时间耗费在寄存器、内核栈以及虚拟内存等资源的保存和恢复上，进而大大缩短了真正运行进程的时间。
        - Linux 通过 TLB（Translation Lookaside Buffer）来管理虚拟内存到物理内存的映射关系。当虚拟内存更新后，**TLB 也需要刷新，内存的访问也会随之变慢**。特别是在多处理器系统上，缓存是被多个处理器共享的，刷新缓存不仅会影响当前处理器的进程，还会影响共享缓存的其他处理器的进程。
    - 何时发生上下文切换——**进程调度**的时候

      Linux 为每个 CPU 都维护了**一个就绪队列**，将活跃进程（即正在运行和正在等待 CPU 的进程）按照优先级和等待 CPU 的时间排序，然后选择最需要 CPU 的进程，也就是优先级最高和等待 CPU 时间最长的进程来运行。

      1. 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。
      2. 进程在**系统资源不足**（比如内存不足）时，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。
      3. 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。
      4. 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。
      5. 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。

线程上下文切换

- 线程与进程最大的区别在于，**线程是调度的基本单位，而进程则是资源拥有的基本单位**。说白了，所谓内核中的任务调度，实际上的调度对象是线程；而进程只是给线程提供了虚拟内存、全局变量等资源。
- 线程的上下文切换其实就可以分为两种情况：

  第一种， 前后两个线程属于**不同进程**。此时，因为资源不共享，所以切换过程就跟进程上下文切换是一样。

  第二种，前后两个线程属于**同一个进程**。此时，因为虚拟内存是共享的，所以在切换时，虚拟内存这些资源就保持不动，只需要切换线程的私有数据、寄存器等不共享的数据。

中断上下文切换

为了快速响应硬件的事件，**中断处理会打断进程的正常调度和执行**，转而调用中断处理程序，响应设备事件。

- 跟进程上下文不同，中断上下文切换并不涉及到进程的用户态
    - 所以，即便中断过程打断了一个正处在用户态的进程，也不需要保存和恢复这个进程的虚拟内存、全局变量等用户态资源。中断上下文，其实只包括内核态中断服务程序执行所必需的状态，包括 CPU 寄存器、内核堆栈、硬件中断参数等。
- **对同一个 CPU 来说，中断处理比进程拥有更高的优先级**，所以中断上下文切换并不会与进程上下文切换同时发生。同样道理，由于中断会打断正常进程的调度和执行，所以大部分中断处理程序都**短小精悍**，以便尽可能快的执行结束。

- 中断和调度器逻辑不属于任何进程(它们也不是进程，它们只是一段在某个时间内会被CPU执行的代码，一般被称作**控制流或执行流**）。
    - 中断响应硬件事件，与进程是并行的概念。调度器是管理进程的，概念上就比进程高一个层次。
    - 那么问题来了：这两个控制流运行时使用的是哪里的堆栈？中断借用当前进程(Linux的current或xv6的proc)的内核栈。调度器使用自己独立的内核堆栈。同时因为每个CPU都有一个自己的调度器，所以调度器的堆栈在系统中有多个，是per-CPU结构。

进程切换

为了控制进程的执行，内核必须有能力刮起正在CPU上运行的进程，并恢复以前挂起的某个进程的执行，这种行为被称为进程切换（process switch）、任务切换（task switch）或上下文切换（context switch）。

硬件上下文

- 进程恢复执行前必须装入寄存器的一组数据称为**硬件上下文（hardware context）**。硬件上下文是进程可执行上下文的一个子集，进程硬件上下文的一部分存放在**TSS段**，剩余部分存放在**内核态的堆栈**中。
    - 早期的Linux版本利用80x86体系结构所提供的硬件支持，并通过**far jmp指令**[far jmp即修改cs寄存器，也修改eip寄存器，而简单的jmp指令只修改eip寄存器]跳到next进程TSS描述符的选择符来执行进程切换。当执行这条指令时，CPU通过自动保存原来的硬件上下文，装入新的硬件上下文来执行硬件上下文切换。但基于一下原因，Linux2.6使用**软件执行进程切换**：**【代码】**
        - 通过一组mov指令逐步执行切换，这样能较好地控制所装入数据的合法性。尤其是，这使检查ds和es段寄存器的值成为可能，这些值有**可能被恶意用户伪造**。当用单独的far jmp指令时，不可能进行这类检查。
        - 旧方法和新方法所需时间大致相同。然而，尽管当前的切换代码还**有改进的余地**，却不能对硬件上下文进行优化。
    - 进程切换至发生在内核态，**用户态进程使用的所有寄存器**内容都保存在内核态堆栈上 包括ss和esp这对寄存器的内容（存储用户态堆栈指针的地址）。

任务状态段 TSS

tss_struct结构描述TSS的格式，init_tss数组为系统上每个不同的CPU存放一个TSS。每个TSS有她自己8字节的任务状态段描述符（TSSD）。

- thread字段
    - 在每次进程切换时，被替换进程的硬件上下文必须保存在别处，**不能像Intel原始设计那样把它保存在TSS中**，因为Linux为每个处理器而不是为每个进程使用TSS。
    - 因此，每个进程描述符包含一个类型为**thread_struct的thread字段**，只要进程被切换出去，内核就把其硬件上下文保存在这个结构中。这个数据结构包含的字段涉及大部分CPU寄存器，但**不包括诸如eax、ebx等等这些通用寄存器**，它们的值保留在内核堆栈中。

执行进程切换

进程切换可能发生在精心定义的点：schedule()函数，这里我们关注内核如何执行一个进程切换。

从本质上说，每个进程切换由两步组成：

1. **切换页全局目录**以安装一个新的地址空间。【第九章】
2. **切换内核态堆栈和硬件上下文**，因为硬件上下文提供了内核执行新进程所需要的所有信息，包含CPU寄存器。

- switch_to宏

  进程切换的第二步由switch_to宏执行。它是内核中与硬件关系最密切的例程之一

  - 首先，该宏有3个参数，prev，next和last。
    - prev和next是输入参数，分别表示被替换进程和新进程描述符的地址在内存中的位置。
    - 那last呢？在任何进程切换中，到三个进程而不是两个。假设内核决定暂停A而激活B，那么在schedule函数中，prev指向A而next指向B。 当切换回A后，就必须暂停另外一个进程C。而LAST则指向C进程。【P110】

```
include/asm-i386/system.h
/**
 * 进程切换时，切换内核态堆栈和硬件上下文。
 * prev-被替换的进程
 * next-新进程
 * last-在任何进程切换中，到三个进程而不是两个。假设内核决定暂停A而激活B，那么在
schedule函数中，prev指向A而next指向B。
 *       当切换回A后，就必须暂停另外一个进程C。而LAST则指向C进程。
 */
#define switch_to(prev,next,last) do {          \
  unsigned long esi,edi;            \
  /**
   * 在真正执行汇编代码前，已经将prev存入eax，next存入edx中了。
   * 没有搞懂gcc汇编语法，反正结果就是这样。
   * 应该是"2" (prev), "d" (next)这句的副作用。
   */

        /**
         * 保存eflags和ebp到内核栈中。必须保存是因为编译器认为在switch_to结束
前，
         * 它们的值应当保持不变。
         */
  asm volatile("pushfl\n\t"         \
        "pushl %%ebp\n\t"           \
        /**
         * 把esp的内容保存到prev->thread.esp中
         * 这样该字段指向prev内核栈的栈顶。
         */
        "movl %%esp,%0\n\t"  /* save ESP */     \
```

```
            /**
             * 将next->thread.esp装入到esp.
             * 此时，内核开始在next的栈上进行操作。这条指令实际上完成了从prev到next
的切换。
             * 由于进程描述符的地址和内核栈的地址紧挨着，所以改变内核栈意味着改变当前
进程。
             */
            "movl %5,%%esp\n\t"  /* restore ESP */ \
            /**
             * 将标记为1f的地址存入prev->thread.eip.
             * 当被替换的进程重新恢复执行时，进程执行被标记为1f的那条指令。
             */
            "movl $1f,%1\n\t"    /* save EIP */    \
            /**
             * 将next->thread.eip的值保存到next的内核栈中。
             * 这样，_switch_to调用ret返回时，就会跳转到next->thread.eip执行。
             * 这个地址一般情况下就会是1f.
             */
            "pushl %6\n\t"   /* restore EIP */ \
            /**
             * 注意，这里不是用call，是jmp，这样，上一条语句中压入的eip地址就可以执
行了。
             */
            "jmp __switch_to\n"         \
            /**
             * 到这里，进程A再次获得CPU。它从栈中弹出ebp和eflags。
             */
            "1:\t"            \
            "popl %%ebp\n\t"          \
            "popfl"             \
            :"=m" (prev->thread.esp),"=m" (prev->thread.eip),  \
            /* last被作为输出参数，它的值会由eax赋给它。 */
             "=a" (last),"=S" (esi),"=D" (edi)      \
            :"m" (next->thread.esp),"m" (next->thread.eip),  \
             "2" (prev), "d" (next));         \
} while (0)
```

- __switch_to()函数

```
arch/i386/kernel/process.c
/**
 * __switch_to函数执行大多数进程切换的工作。
 * 进程切换的工作开始于switch_to宏，但是它的主要工作还是由__switch_to完成。
 * 这个函数是寄存器传参的函数。在switch_to宏中，参数已经保存在eax和edx中了.
 */
struct task_struct fastcall * __switch_to(struct task_struct *prev_p,
struct task_struct *next_p)
{
```

```
  struct thread_struct *prev = &prev_p->thread,
         *next = &next_p->thread;
  /**
   * 通过读取current_thread_info()->cpu,获得当前进程在哪个CPU上运行.
   * 因为在schedule函数中已经调用了禁用抢占,所以这里可以直接使用smp_processor_id()
   */
  int cpu = smp_processor_id();
  struct tss_struct *tss = &per_cpu(init_tss, cpu);

  /* never put a printk in __switch_to... printk() calls wake_up*()
indirectly */
  /**
   * __unlazy_fpu宏有选择的保存FPU\MMX\XMM寄存器的内容.
   * 它可能会延后保存这些寄存器的内容.
   */
  __unlazy_fpu(prev_p);

  /*
   * Reload esp0, LDT and the page table pointer:
   */
  /**
   * 把next_p->thread.esp0装入本地CPU的TSS的esp0字段.
   * 任何由sysenter汇编指令产生的从用户态到内核态的特权级转换将把这个地址复制到esp寄存
器.
   */
  load_esp0(tss, next);

  /*
   * Load the per-thread Thread-Local Storage descriptor.
   */
  /**
   * 将next_p进程使用的线程局部存储(TLS)段装入本地CPU的全局描述符表.
   */
  load_TLS(next, cpu);

  /*
   * Save away %fs and %gs. No need to save %es and %ds, as
   * those are always kernel segments while inside the kernel.
   */
  /**
   * 把fs和gs段寄存器的内容分别存放在prev_p->thread.fs和prev_p->thread.gs中.
   */
  asm volatile("movl %%fs,%0":"=m" (*(int *)&prev->fs));
  asm volatile("movl %%gs,%0":"=m" (*(int *)&prev->gs));

  /*
   * Restore %fs and %gs if needed.
   */
  /**
```

```
    * 不管是prev还是next,只要他们使用了fs和gs,那么,都需要将next中的fs,gs更新到段寄存器.
    * 即使next并不使用fs,但是只要prev使用了,也需要更新.这样可以防止next通过fs,gs访问prev的数据.
    */
  if (unlikely(prev->fs | prev->gs | next->fs | next->gs)) {
    /**
     * loadsegment可能会装载一个无效的段寄存器.CPU可能会产生一个异常.
     * 但是loadsegment会采用代码修正技术来处理这种情况.
     */
    loadsegment(fs, next->fs);
    loadsegment(gs, next->gs);
  }

  /*
   * Now maybe reload the debug registers
   */
  /**
   * 用debugreg数组的内容dr0..dr7中的6个调试寄存器.这允许定义四个断点区域.
   */
  if (unlikely(next->debugreg[7])) {
    loaddebug(next, 0);
    loaddebug(next, 1);
    loaddebug(next, 2);
    loaddebug(next, 3);
    /* no 4 and 5 */
    loaddebug(next, 6);
    loaddebug(next, 7);
  }

  /**
   * 如果必要,更新TSS中的IO位图.当next或者prev有其自己的定制IO权限位图时必须这么做.
   */
  if (unlikely(prev->io_bitmap_ptr || next->io_bitmap_ptr))
    /**
     * handle_io_bitmap并不立即更新位图,而是采用一种懒模式的方法.
     */
    handle_io_bitmap(next, tss);

  /**
   * return产生的汇编指令是movl %edi, %eax,ret.
   * 这里有保护eax和返回地址的问题.请仔细理解.
   * 除了需要理解switch_to宏中的jmp指令外,对于没有产生切换,而是第一次开始执行的进程.
   * 它并不会跳回switch_to,而是找到ret_from_fork函数的超始地址.
   */
  return prev_p;
}
```

中断和异常

中断描述符表 IDT

IDT是一个系统表，它与每一个中断或异常向量相联系，每一个向量在表中有**相应的中断或异常处理程序的入口地址**。内核在允许中断发生前，必须适当的初始化IDT。

IDT的格式与GDT和LDT非常相似，表中的每一项对应一个中断或异常向量，每个向量由**8个字节**组成，因此，最多需要**256x8=2048字节**来存放IDT（80x86体系结构限制了只能使用256个向量。其中32个留给CPU，可用向量空间有224个向量）。

idtr CPU寄存器使**IDT可以位于内存的任何地方**，它指定IDT的线性基地址及其限制（最大长度）。在允许中断之前，必须用lidt汇编指令初始化idtr。

IDT包含三种类型的描述符，注意40-43位的Type字段的值表示描述符的类型。【图】

- **任务门**（task gate）

  当中断信号发生时，必须取代当前进程的那个进程的TSS选择符存放在任务门中。

- **中断门**（interrupt gate）

  包含段选择符和中断或异常处理程序的段内偏移量。当控制权转移到一个适当的段时，处理器清IF标志，从而关闭将来会发生的可屏蔽中断。

- **陷阱门**（Trap gate）

  与中断门相似，只是控制权传递到一个适当的段时处理器不修改IF标志。

Linux利用中断门处理中断，利用陷阱门处理异常，“Double fault”异常是唯一由任务门处理的异常，它表示一种内核错误。

中断和异常的硬件处理

当执行了一条指令之后，**cs和eip这对寄存器包含下一条将要执行的指令的逻辑地址**。在处理那条指令之前，**控制单元**会检查在运行前一条指令时是否已经发生了一个中断或异常。如果发生了一个中断或异常，那么控制单元执行下列操作：

1. 确定与中断或异常关联的向量i（0<=i<=255）。
2. 读由idtr寄存器指向的IDT表中的第i项。
3. 从gdtr寄存器获得GDT的基地址，并在GDT中查找，以**读取IDT表项中的选择符所标识的段描述符**。这个描述符指定中断或异常处理程序所在段的基地址。
4. **确信中断是由授权的（中断）发生源发出的。**
   - 首先将当前特权级CPL（存放在cs寄存器的第两位）与短描述符（存放在GDT中）的描述符特权级DPL比较，如果CPL小于DPL，就产生一个“General protection”异常，因为**中断处理程序的特权不能低于引起中断的程序的特权**。
   - 对于编程异常，则做进一步的安全检查：比较CPL与处于IDT中的门描述符的DPL，如果DPL小于CPL，就产生一个“General protection”异常。这最后一个检查可以**避免用户应用程序访问特殊的陷阱门或中断门**。
5. **检查是否发生了特权级的变化**，也就是说，CPL是否不同于所选择的段描述符的DPL。如果是，控制单元必须开始使用与新的特权级相关的栈。通过执行以下步骤来做到这点：

    1. 读tr寄存器，以访问运行进程的TSS段。
    2. 用与新特权级相关的栈段和栈指针的正确值装载ss和esp寄存器。这些值可以在TSS中找到。
    3. 在新的栈中保存ss和esp以前的值，这些值定义了与旧特权级相关的栈的逻辑地址。
6. 如果故障已经发生，用引起异常的指令地址装载cs和eip寄存器，从而使得这条指令能再次被执行。
7. 在**栈中保存eflags、cs和eip的内容**。
8. 如果异常产生了一个硬件错误码，则将它保存在栈中。
9. **装载cs和eip寄存器**，其值分别是**IDT表中第i项门描述符的段选择符**和**偏移量字段**。这些值给出了**中断或者异常处理程序的第一条指令的逻辑地址**。

控制单元所执行的最后一步就是跳转到中断或者异常处理程序。换句话说，**处理完中断信号后**，控制单元所执行的指令就是**被选中处理程序的第一条指令**。

中断或异常被处理完后，**相应的处理程序必须产生一条iret指令，把控制权转交给被中断的进程**，这将迫使控制单元：

1. 用保存在栈中的值装载cs、eip或eflags寄存器。如果一个硬件出错码曾被压入栈中，并且在eip内容的上面，那么，执行iret指令钱必须先弹出这个硬件出错码。
2. 检查处理程序的CPL是否等于cs中最低两位的值（这意味着被中断的进程与处理程序运行在同一特权级）。如果是，iret终止执行；否则，转入下一步。
3. 从栈中装载ss和esp寄存器，因此，返回到与旧特权级相关的栈。
4. 检查ds、es、fs及gs段寄存器的内容，如果其中一个寄存器包含的选择符是一个段描述符，而且其DPL小于CPL，那么，清相应的段寄存器。控制单元这么做是为了禁止用户态的程序（CPL=3）利用内核以前所用的段寄存器（DPL=0）。如果不清这些寄存器，怀有恶意的用户态程序就可能利用它们来访问内核地址空间。

中断和异常处理程序的嵌套执行

每个中断或异常都会引起一个内核控制路径，或者说**代表当前进程在内核态执行单独的指令序列**。

内核控制路径可以任意嵌套；一个中断处理程序可以被另一个中断处理程序“中断”，因此引起内核控制路径的嵌套执行【图】。其结果是，对中断进行处理的内核控制路径，其最后一部分指令并不总能使当前进程返回到用户态：如果嵌套深度大于1，这些指令将执行上次被打断的内核控制路径，此时的CPU依然运行在内核态。

中断处理

- 为中断处理程序保存寄存器的值
    - 当CPU接收一个中断是，就开始执行相应的中断处理程序代码，该代码的地址存放在IDT的相应门中（参见前面“中断和异常的硬件处理”）。
    - 保存寄存器是中断处理程序做的第一件事情。IRQn中断处理程序的地址开始存在interrupt[n]中，然后复制到IDT相应表项的中断门中。

```
arch/i386/kernel/entry.S
/*
 * Build the entry stubs and pointer table with
 * some assembler magic.
 */
```

```
.data
ENTRY(interrupt)
.text

vector=0
ENTRY(irq_entries_start)
.rept NR_IRQS
  ALIGN
1:  pushl $vector-256
  jmp common_interrupt
.data
  .long 1b
.text
vector=vector+1
.endr

  ALIGN
common_interrupt:
  SAVE_ALL
  movl %esp,%eax
  call do_IRQ
  jmp ret_from_intr

/**
 * 处理器间的中断处理程序的汇编语言代码是由BUILD_INTERRUPT宏产生的。
 * 它保存寄存器，从栈顶压入向量号减256的值，然后调用高级C函数（其名字就是低级处理程序的名字加前缀smp_）。
 * 如，CALL_FUNCTION_VECTOR类型的处理器间中断的低级处理程序是call_function_interrupt，它调用smp_call_function_interrupt的高级处理程序。
 */
#define BUILD_INTERRUPT(name, nr) \
ENTRY(name)       \
  pushl $nr-256;      \
  SAVE_ALL      \
  movl %esp,%eax;     \
  call smp_/**/name;    \
  jmp ret_from_intr;
```

- 调用do_IRQ()函数执行与一个中断相关的所有中断服务例程

```
arch/i386/kernel/irq.c
/*
 * do_IRQ handles all normal device IRQ's (the special
 * SMP cross-CPU interrupts have their own specific
 * handlers).
 */
/**
```

```
 * do_IRQ执行与一个中断相关的所有中断服务例程.
 */
fastcall unsigned int do_IRQ(struct pt_regs *regs)
{
  /* high bits used in ret_from_ code */
  int irq = regs->orig_eax & 0xff;
#ifdef CONFIG_4KSTACKS
  union irq_ctx *curctx, *irqctx;
  u32 *isp;
#endif

  /**
   * irq_enter增加中断嵌套计数
   */
  irq_enter();
#ifdef CONFIG_DEBUG_STACKOVERFLOW
  /* Debugging check for stack overflow: is there less than 1KB free?
*/
  {
    long esp;

    __asm__ __volatile__("andl %%esp,%0" :
          "=r" (esp) : "0" (THREAD_SIZE - 1));
    if (unlikely(esp < (sizeof(struct thread_info) + STACK_WARN))) {
      printk("do_IRQ: stack overflow: %ld\n",
        esp - sizeof(struct thread_info));
      dump_stack();
    }
  }
#endif

#ifdef CONFIG_4KSTACKS

  /**
   * 如果中断栈使用不同的的栈,就需要切换栈.
   */
  curctx = (union irq_ctx *) current_thread_info();
  irqctx = hardirq_ctx[smp_processor_id()];

  /*
   * this is where we switch to the IRQ stack. However, if we are
   * already using the IRQ stack (because we interrupted a hardirq
   * handler) we can't do that and just have to keep using the
   * current stack (which is the irq stack already after all)
   */
  /**
   * 当前在使用内核栈,而不是硬中断请求栈.就需要切换栈
   */
  if (curctx != irqctx) {
```

```
    int arg1, arg2, ebx;

    /* build the stack frame on the IRQ stack */
    isp = (u32*) ((char*)irqctx + sizeof(*irqctx));
    /**
     * 保存当前进程描述符指针
     */
    irqctx->tinfo.task = curctx->tinfo.task;
    /**
     * 把esp栈指针寄存器的当前值存入irqctx的thread_info(内核oops时使用)
     */
    irqctx->tinfo.previous_esp = current_stack_pointer;

    /**
     * 将中断请求栈的栈顶装入esp,isp即为中断栈顶
     * 调用完__do_IRQ后,从ebx中恢复esp
     */
    asm volatile(
      "       xchgl   %%ebx,%%esp      \n"
      "       call    __do_IRQ         \n"
      "       movl    %%ebx,%%esp      \n"
      : "=a" (arg1), "=d" (arg2), "=b" (ebx)
      :  "0" (irq),   "1" (regs),  "2" (isp)
      : "memory", "cc", "ecx"
    );
  } else/* 否则,发生了中断嵌套,不用切换 */
#endif
    __do_IRQ(irq, regs);

  /**
   * 递减中断计数器并检查是否有可延迟函数
   */
  irq_exit();

  /**
   * 结束后,会返回ret_from_intr函数.
   */
  return 1;
}
```

内存与TLB

TLB

在多处理系统中，每个CPU都有自己的TLB，这叫做该CPU的本地TLB。**当CPU的cr3被修改时，硬件自动使本地TLB中的所有项都无效**，这是因为新的一组页表被启用而TLB指向的是旧数据。

物理内存布局

在初始化阶段，内核必须建立一个物理地址映射来**指定哪些物理地址范围对内核可用而哪些不可用**（或者因为他们映射硬件设备I/O的共享内存，或者因为相应的页框含有BIOS数据）。

内核将下列页框记为**保留**：

- 在不可用的物理地址范围内的页框。
- **含有内核代码和已初始化的数据结构**的页框。

保留页框的页绝不能被动态分配或交换到磁盘上。

一般来说，Linux内核安装在RAM中从物理地址0x00100000开始的地方，也就是第二个MB。为什么内核没有安装在**RAM第一个MB**开始的地方？因为PC体系结构有几个独特的地方必须考虑到。例如：

- 页框0由BIOS使用，**存放加电自检期间检查到的系统硬件配置**。因此，很多膝上型电脑（笔记本）的BIOS甚至在系统初始化后还将数据写到该页框。
- 物理地址**从0x000a0000到0x000fffff地址范围通常留给BIOS例程**，并且映射ISA图形卡上的内部内存。这个区域就是所有IBM兼容PC上从640KB到1MB之间著名的洞：物理地址存在但被保留，对应的页框不能由操作系统使用。
- 第一个MB内的其他页框可能由特定计算机模型保留。例如，IBM Thinkpad把0xa0页框映射到0x9f页框。

Linux2.6的前768个页框（3MB）【图】

我们假设内核需要小于3MB的RAM：

- 符号`_text`对应与物理地址0x0010000，表示内核代码第一个字节的地址，内核代码的结束位置由另一个类似的符号`_etext`表示。
- 内核数据分为两组：初始化过的数据和没有初始化的数据。
  - 初始化过的数据在`_etext`后开始，在`_edata`结束。
  - 紧接着就是为初始化的数据并以`_end`结束。
- 图中出现的符号并没有在Linux源码中定义，他们是编译内核时产生的，可以在System.map文件中找到这些符号的线性地址，System.map时编译内核以后所创建的。【代码】

进程页表

进程的线性地址空间分为两部分：

- 从0x00000000到0xbfffffff的线性地址，无论进程运行在**用户态还是内核态**都可以寻址。
- 从0xc0000000到0xffffffff的线性地址，只有**内核态**的进程才能寻址。

当进程运行在用户态时，他产生的线性地址小于0xc0000000；当进程运行在内核态时，他执行内核代码，所产生的地址大于等于0xc0000000.但是，在某些情况下，内核为了检索或存放数据必须访问用户态线性地址空间。

页全局目录的第一部分表项映射的线性地址小于0xc0000000，具体大小依赖于特定进程。相反，剩余的表项对所有进程来说都应该是相同的，他们等于主内核页全局目录的相应表项。

怎么切换内核态的？代码？如何判定当前运行在内核态？

内核页表

内核维持着一组自己使用的页表，驻留在所谓的**主内核页全局目录**（master kernel Page Global Directory）中。**系统初始化后，这组页表还从未被任何进程或任何内核线程直接使用**；更确切地说，主内核页全局目录的最高目录项部分作为参考模型，为系统中每个普通进程对应的页全局目录项提供参考模型。

- 那么内核如何确保对主内核页全局目录的**修改能传递到进程实际使用的页全局目录**中？【第八章 非连续内存区的线性地址】
  - copy_mm中对pgd里面pmd的设置具体在哪？
  - 只要处于内核态的一个进程为“高端”线性地址（高于TASK_SIZE，等于PAGE_OFFSET 通常为0xc0000000）修改了页表项，那么，他就也应当**更新系统中所有进程页表集合中的相应表项**。事实上，触及所有进程的页表集合是相当费时的操作，因此，Linux采用一种**延迟方式**。
  - 这种延迟方式：每当一个高端地址被重新映射时（一般是通过vmalloc()或vfree()），内核就更新被定位在`swapper_pg_dir`主内核页全局目录中的常规页表集合。这个页全局目录由**主内存描述符（master memory descriptor）的pgd**字段所指向，而主内存描述符存放于`init_mm`变量。swapper内核线程在初始化阶段使用init_mm。但是一旦初始化完成，swapper再不使用这个内存描述符。
  - **vmalloc 给内核分配非连续内存区——修改内核使用的页表项**：`map_vm_area`

```
mm/vmalloc.c
/**
 * 将线性地址和页框对应起来
 * area-指向内存区的vm_struct描述符的指针
 * prot-已分配页框的保护位，它总是被置为0x63，对应着
present,accessed,read/write及dirty.
 * pages-指向一个指针数组的变量的地址。该指针数组的指针指向页描述符。
 */
int map_vm_area(struct vm_struct *area, pgprot_t prot, struct page
***pages)
{
  /**
   * 首先将内存区的开始和末尾的线性地址分配给局部变量address和end
   */
  unsigned long address = (unsigned long) area->addr;
  unsigned long end = address + (area->size-PAGE_SIZE);
  unsigned long next;
  pgd_t *pgd;
  int err = 0;
  int i;

  /**
```

```
    * 使用pgd_offset_k来获得主内核页全局目录中的目录项。该目录项对应于内存区起始线性地址。
    */
   pgd = pgd_offset_k(address);
   /**
    * 获得内核页表自旋锁。
    */
   spin_lock(&init_mm.page_table_lock);
   /**
    * 此循环为每个页框建立页表项。
    */
   for (i = pgd_index(address); i <= pgd_index(end-1); i++) {
     /**
      * 调用pud_alloc来为新内存区创建一个页上级目录。并把它的物理地址写入内核页全局目录的合适表项。
      */
     pud_t *pud = pud_alloc(&init_mm, pgd, address);
     if (!pud) {
       err = -ENOMEM;
       break;
     }
     next = (address + PGDIR_SIZE) & PGDIR_MASK;
     if (next < address || next > end)
       next = end;
     /**
      * map_area_pud函数为页上级目录所指向的所有页表建立对应关系。
      */
     if (map_area_pud(pud, address, next, prot, pages)) {
       err = -ENOMEM;
       break;
     }

     address = next;
     pgd++;
   }

   spin_unlock(&init_mm.page_table_lock);
   flush_cache_vmap((unsigned long) area->addr, end);
   return err;
}
```

- 注意，`map_vm_area()`并不触及当前进程的页表。因此，当内核态的进程访问非连续内存区时，缺页发生，因为**该内存区所对应的进程页表中的表项为空**。然而，缺页处理程序要检查这个缺页线性地址是否在主内核页表中（也就是init_mm.pgd页全局目录和它的子页表）。一旦处理程序发现一个主内核页表含有这个线性地址的非空项，就把它的值拷贝到相应的进程页表项中，并恢复进程的正常执行（详细查阅 缺页异常处理程序）。

- 内核**如何初始化自己的页表**
  - 第一个阶段，内核**创建一个有限的地址空间**，包括内核的代码段和数据段、初始页表和用于存放动态数据结构的共128KB大小的空间。这个最小限度的地址空间仅够将内核装入RAM和对其初始化的核心数据结构。
  - 第二个阶段，内核充分利用剩余的RAM并适当的建立分页表。
- 临时内核页表
  - 临时页全局目录是在内核编译过程中静态的初始化的，而临时页表是由`startup_32()`汇编语言函数（定义于`arch/i386/kernel/head.S`）初始化的。
  - 临时页全局目录放在`swapper_pg_dir`变量中，临时页表在pg0变量处开始存放，紧接在内核未初始化的数据段后面。

固定映射的线性地址

fix-mapped linear address ……

处理TLB

处理器不能自动同步他们自己的TLB高速缓存，因为**决定线性地址和物理地址之间映射何时不再有效的是内核，而不是硬件**。

- 使TLB无效的宏：
  - `__flush_tlb()`，将cr3寄存器的当前值重新写回cr3，使用对象：`flush_tlb`，`flush_tlb_m`，`flush_tlb_range`。
  - `__flush_tlb_global()`，通过清除cr4的PGE标志禁用全局页，将cr3寄存器的当前值重新写回cr3，并再次设置PGE标志，使用对象：`flush_tlb_all`，`flush_tlb_kernel_range`。
  - `__flush_tlb_single(addr)`， 以addr为参数执行invlpg汇编语言指令，使用对象：`flush_tlb_page`。
  - 这些独立于系统结构的使TLB无效的方法非常简单地扩展到了多处理器系统上。
- 一般来说，任何进程切换都会暗示着更换活动页表集。相对于过期页表，本地TLB表项必须被刷新；这个过程在**内核把新的页全局目录的地址写入cr3控制寄存器时会自动完成**。
- 懒惰TLB（lazy TLB）模式……cpu_tlbstate变量。

保护机制

…

实践

- 创建页表 alloc_page，得到独立空间的页表的pgd

- 初始化页表 init_mapping，设置各entry映射的值
- 保存当前cr3的值，加载新的cr3

- 考虑在独立空间执行的时候 发生中断、异常或上下文切换的情况。这些情况的处理需要切换回原来的内核空间。因此要中止（abort）独立执行环境 并恢复之前cr3。
    - 在中断处理函数开始的位置加入start_abort()，结束的位置加入finish_abort()，主要是恢复cr3的一系列操作。
- 考虑在独立空间执行的时候 发生页错误的情况。需要切换回原来的内核空间。
    - 记录相关的错误日志
- 考虑自己的四级页表管理，独立空间的页表缓冲区 与内核页表的track
    - init_backend 缓冲 同步
    - set_pud、pud_alloc、pud_offset、copy_pud_range、clear_pud_range（p4d、pud、pmd、pte等的封装）
    - 管理映射 mapping_list（添加、删除）
        - 考虑页对齐、overmap 信息泄露warning
- 考虑多cpu
- 考虑PCID

> 基础篇：经常说的 CPU 上下文切换是什么意思？（上）

2020.12.4 by jn