操作系统思辨报告2

2019211315班 2019211243 谭恩浩

> 请说明操作系统创建内核进程的方法，内核进程创建用户进程的方法，用户进程使用系统调用并从系统调用返回用户进程的方法，内核进程切换的方法，用户进程切换的方法，请比较hariboteOS，xv6，uCore三种自制系统的做法。

hariboteOS

hariboteOS并没有明确区分内核态和用户态，对系统的保护是区分了操作系统栈和用户栈，用户程序无法使用操作系统的段地址。用户程序进行的系统调用也仅仅是直接使用操作系统的函数，并没有“委托”给操作系统去执行，而是自己直接调用，缺乏安全性。

进程切换

hariboteOS中没有线程的概念，进程被称为“任务”，以下也用“任务”来代表进程进行叙述。

当向CPU发出任务切换的指令时，CPU会先把寄存器中的值全部写入内存中，这样做是为了当以后切换回这个程序的时候，可以从中断的地方继续运行。接下来， 为了运行下一个程序，CPU会把所有寄存器中的值从内存中读取出来，这样就完成了一次切换。hariboteOS也没有进程控制块PCB的概念，而用了TSS结构（Task Status Segment）来储存寄存器的值和任务相关信息。TSS共包含26个int成员，总计104字节。其中，使用扩展指针指令寄存器`EIP`寄存器来记录下一条需要执行的指令位于内存中哪个地址。

hariboteOS可以调用`mt_taskswitch`函数进行任务切换。首先用`mt_init`函数初始化`mt_timer`和`mt_tr`的值，并将计时器设置为0.02秒之后。之后用`mt_taskswitch`函数按照当前的mt_tr变量的值计算出下一个`mt_tr`的值，将计时器（`mt_timer`）重新设置为0.02秒之

后，并进行任务切换。其中，任务寄存器`TR`（即`mt_tr`）的作用是让CPU记住当前正在运行哪一个任务。当进行任务切换的时候，`TR`寄存器的值也会自动变化。

转换到汇编，实际上是使用`JMP`指令进行任务切换。`JMP`指令分为两种，只改写扩展指针指令寄存器`EIP`的称为near模式，同时改写扩展指针指令寄存器`EIP`和代码段寄存器`CS`的称为far模式。如果一条`JMP`指令所指定的目标地址段不是可执行的代码，而是TSS的话，CPU就不会执行通常的改写`EIP`和`CS`的操作，而是将这条指令理解为任务切换。也就是说，CPU会切换到目标TSS所指定的任务。任务结束后，切换回的操作与切换出的操作几乎相同，CPU会根据TSS中的值，回到之前的任务`JMP`指令处继续进行。完成进程之间的切换。

uCore

内核线程

内核线程是一种特殊的进程，内核线程与用户进程的区别有两个：内核线程只运行在内核态而用户进程会在在用户态和内核态交替运行；所有内核线程直接使用共同的uCore内核内存空间，不需为每个内核线程维护单独的内存空间而用户进程需要维护各自的用户内存空间从内存空间占用情况这个角度上看，可以把线程看作是一种共享内存空间的轻量级进程。

创建内核进程

在uCore OS启动后，已经对整个内核内存空间进行了管理， 通过设置页表建立了内核虚拟空间，uCore OS内核中的所有线程都不需要再建立各自的页表，只需共享这个内核虚拟空间就可以访问整 个物理内存。

创建第0个内核线程idleproc（即操作系统开始运行时所创建的第一个进程）

当前的执行上下文可以看成是uCore内核(也可看做是内核进程)中的一个内核线程的上下文。为此，uCore通过给当前执行的上下文分配一个进程控制块以及对它进行相应初始化，将其打造成第0个内核线程`idleproc`。需要注意以后的其他线程的内核栈都需要通过分配获得，因为uCore启动时设置的内核栈直接分配给`idleproc`使用了。

创建第 1 个内核线程 initproc（真正意义上“做工作”的第一个进程）

第0个内核线程主要工作是完成内核中各个子系统的初始化，然后就通过执行`cpu_idle`函数开始过退休生活了，所以uCore接下来还需创建其他进程来完成各种工作，于是通过调用`kernel_thread`函数创建了一个内核线程函数`init_main`，`init_main`函数是`initproc`的主

体。`init_main`的工作就是创建特定的其他内核线程或用户进程。其中，`do_fork`是创建线程的主要函数。该函数进行分配并初始化进程控制块、内核栈，设置中断帧和上下文等工作。需要注意的是，设置initproc的进程上下文时，设置了`initproc`的执行现场中主要的两个信息:上次停止执行时的下一条指令地址和上次停止执行时的堆栈地址，由于`initproc`还没有执行过，这其实就是`initproc`实际执行的第一条指令地址和堆栈指针。`initproc`此时处在`proc_list`队列中，为就绪状态。

在uCore执行完`proc_init`函数后，就创建好了两个内核线程：`idleproc`和`initproc`。`idleproc`将通过执行`cpu_idle`函数让出CPU，给其它内核线程执行，在这里只有两个进程，于是CPU被转交给`initproc`。接下来就可以根据需要创建其他的内核进程或用户进程了。

创建用户进程

第一个用户进程是由第二个内核线程`initproc`通过把应用程序执行码覆盖到`initproc`的用户虚拟内存空间来创建的。这里涉及几个函数，uCore通过`do_execve`函数来完成用户进程的创建工作，并通过`load_icode`函数给用户进程建立一个能够让用户进程正常运行的用户环境。`do_execve`函数的主要工作流程是：首先为加载新的执行码做好用户态内存空间清空准备，然后加载应用程序执行码到当前进程的新创建的用户态虚拟空间中。第二步正是通过`load_icode`函数实现的。

用户进程的用户环境搭建完毕后，`initproc`将按产生系统调用的函数调用路径原路返回，执行中断返回指令`iret`后，将切换到用户进程的第一条语句位置开始执行。

使用系统调用并从系统调用返回

系统调用的实现，从硬件层面上看，需要硬件能够支持在用户态的用户进程通过某种机制切换到内核态。在uCore初始化函数`kern_init`中调用了`idt_init`函数来初始化中断描述符表，并设置一个特定中断号的中断门，专门用于用户进程访问系统调用，由`ide_init`函数完成。

在执行加载中断描述符表`lidt`指令前，专门设置了一个特殊的中断描述符`idt[T_SYSCALL]`，它的特权级设置为`DPL_USER`，中断向量处理地址在`__vectors[T_SYSCALL]`处。这样建立好这个中断描述符后，一旦用户进程执行`INT T_SYSCALL`后，由于此中断允许用户态进程产生， 所以CPU就会从用户态切换到内核态，保存相关寄存器，并跳转到`__vectors[T_SYSCALL]`处开始执行。

在操作系统中初始化好系统调用相关的中断描述符、中断处理起始地址等后，还需在用户态的应用程序中初始化好相关工作，简化应用程序访问系统调用的复杂性。为此在用户态

建立了一个中间层，即简化的`libc`实现，在`user/libs/ulib.[ch]`和`user/libs/syscall.[ch]`中完成了对访问系统调用的封装。用户态最终的访问系统调用函数是`syscall`。

在操作系统完成用户请求的服务后，操作系统按调用关系的路径原路返回到。然后开始根据当前进程的中断帧内容做恢复执行现场操作。把`trapframe`的一部分内容保存到寄存器内容。恢复寄存器内容结束后，调整内核堆栈指针到 中断帧的`tf_eip`处。执行`IRET`指令后，CPU根据内核栈的情况回复到用户态，并把`EIP`指向`tf_eip`的值。这样整个系统调用就执行完毕了。

进程切换

uCore系统没有明确区分内核进程的切换和用户进程的切换，故放在一起统一解释。

首先在执行某进程A的用户代码时，出现了一个`trap`（例如是一个外设产生的中断），这个时候就会从进程A的用户态切换到内核态（过程 1），并且保存好进程A的`trapframe；`
当内核态处理中断时发现需要进行进程切换时，uCore要通过`schedule`函数选择下一个将占用CPU执行的进程（即进程B），然后会调用`proc_run`函数，`proc_run`函数进一步调用`switch_to`函数，切换到进程B的内核态（过程 2），继续进程B上一次在内核态的操作，并通过`iret`指令，最终将执行权转交给进程B的用户空间（过程 3）。
当进程B由于某种原因发生中断之后（过程 4），会从进程B的用户态切换到内核态，并且保存好进程B的`trapframe`；当内核态处理中断时发现需要进行进程切换时，即需要切换到进程A， uCore再次切换到进程A（过程 5），会执行进程A上一次在内核调用`schedule`函数返回后的下一行代码，这行代码还是在进程A的上一次中断处理流程中。最后当进程A的中断处理完毕的时候，执行权又会交还给进程A的用户代码（过程 6）。这就是在只有两个进程的情况下，进程切换间的大体流程。

XV-6

XV-6系统和uCore系统几乎是使用同一种策略进行内核进程的创建、用户进程的创建、系统调用和进程切换。现仅调一部分简单叙述

创建用户进程

使用`userinit`函数创建第一个进程，`userinit`函数首先调用`allocproc`函数，`allocproc`的工作是在页表中分配一个槽，并初始化进程的状态，为其内核线程的运行做准备。需要注意的是，`userinit`仅仅在创建第一个进程时被调用，而`allocproc`创建每个进程时都会被调用。`allocproc`为新进程设置好一个特别准备的内核栈和一系列内核寄存器，使得进程

第一次运行时会"返回"到用户空间。第一个进程内存中的初始内容是由`initcode.S`汇编得到的。

系统调用并返回

我们可以在用户空间调用`T_SYSCALL`中断。每个中断的处理过程在`vectors.S`中实现，压入不同参数过后执行`alltraps`函数。在`alltraps`函数中准备参数`trapframe`，然后调用`trap`函数。当中断是`T_SYSCALL`时执行函数`syscall`，`syscall`根据`eax`寄存器中的系统调用id，确定具体函数，执行相应函数，把执行结果放入`eax`寄存器中。

服务结束后，按原路径返回，恢复现场，回到用户态。

进程切换

当前进程通过调用`yield`函数，进行进程切换。`yield`函数调用`sched`函数，`sched`函数启动`swtch`函数完成进程切换。保存当前进程上下文和换入新进程上下文的工作由`swtch`函数完成。进程切换主要有两部分：切换全局页表项和切换内核堆栈和硬件上下文。其中，切换器是一个死循环，该循环不断在进程表中扫描，选择一个RUNNABLE的进程进行调度。

需要注意的是，对于进程切换的代码都是被所有进程共用的，代码本身不属于某一个特定的进程，所以判定当前在哪一个进程不是通过看执行的代码是哪个进程的，而是通过`esp`指向哪个进程的堆栈来判定的。`esp`指向了其它进程的堆栈，当前进程即被挂起，等待若干时间，当`esp`指针再次指回这个进程的堆栈时，这个进程又重新开始运行。