第8章 进程管理

进程管理关系到在多个进程之间共享处理机和主存，这些进程可被视为是这些资源的竞争者。

在初启使用资源的进程时，或者由于其他原因，经常要作出重新分配资源的决定。

8.1 进程切换

一个进程调用swtch(2178)，swtch又调用“ retu ”(0740)可挂起自身，也就是释放处理机。

例如若一个进程已到达它不能超过的某个点，这时它就要调用“ sleep ”(2066)，而sleep则调用“ swtch ”。

另外，一个在核心态下运行的进程，当它将要转入用户态之前，会测试变量“ runrun ”，如果其值非0，则意味着更高优先权的进程已为运行准备就绪。此时该核心态进程也将调用“ swtch ”。

“ swtch ”在“ proc ”表中搜索其“ p_stat ”等于“ SRUN ”，其“ p_flag ”中设置了“ SLOAD ”位的各进程，并从中选择其“ p_pri ”最小的进程，然后将控制转移给被选中的进程。

对于每一个进程的“ p_pri ”值，系统要经常重新进行计算，为此目的使用的过程是“ setpri ”(2156)。显然，“ setpri ”使用的算法具有重要影响。

调用“ sleep ”挂起自身的进程可由另一个进程唤醒而返回到准备运行状态。在进行中断处理时经常发生这种情况，此时处理中断的进程直接调用“ setrun ”(2134)或先调用“ wakeup ”(2113)，然后由“ wakeup ”调用“ setrun ”。

8.2 中断

应当注意一个硬件中断 (请参见第9章)并不会直接造成对“ swtch ”或其等效例程的调用。一个硬件中断使一个用户进程转变为一核心进程，正如上面刚刚提到的，在它处理完中断后可能不返回到用户态而是调用“ swtch ”。

若一核心态进程被中断，那么在中断处理结束后，该核心态进程一定回到它被中断的点继续运行。这一点对理解UNIX如何避免很多与“ 临界区 ”(critical sections)相关的陷井是很重要的。本章最后部分将对临界区进行讨论。

8.3 程序交换

一般而言，内存容量并不足以让所有进程的映像同时驻留其中，于是某些数据段就是被“ 换出 ”(swapped out)，写到磁盘上一个被称之为磁盘交换区(disk swapping area)的区域中。

存放在磁盘期间，进程映像相对而言是不可存取的，而且确实是不可执行的。因此，驻在内存中的进程映像集应当定期更改，其方法是将内存中的某些进程映像换出，将存放在磁盘交换区中的某些进程映像换进。

关于进程映像换进换出的很多决策都是由过程“ sched ”(1940)作出的，在第14章将对此过程进行详细分析。

“ sched ”是由＃0进程执行的。＃ 0进程完成其初始任务后起两种作用。一种作用是调度进程 (scheduler)，也就是一个正常的核心态进程；另一种是在不知不觉之中担任“ swtch ”的中间进程 (已在第7章中讨论了 swtch)。因为过程“ sched ”决不终止，所以核心态＃ 0进程也决不会完成其任务，随之而来的结果是：不会有在用户态下运行的＃ 0进程。

8.4 作业

在UNIX中没有作业的概念，至少是没有传统批处理系统中作业的概念。

任一进程在任一时刻都可创建 (fork)它自身的一个副本，而且基本上没有多少延迟，因而也就创建了一个新作业的等价物。 UNIX中没有作业调度、作业类之类的活动和概念。

8.5 汇编语言过程

下面3个过程是用汇编语言编写的，而且在处理机优先级设置为 7级时运行。这些过程并不遵循一般的过程入口约定，在进入和退出这些过程时，不影响 r5和r6(环境和栈指针)。

正如在上一章中已提及的，“ savu ”和“ retu ”两者组合起来起合作例程跳转的作用。第 3个例程是“ aretu ”，若其后跟随一条“ return ”语句，则起非本地“ goto ”的作用。

8.6 savu(0725)

此过程由“ newproc ”(1889，1905)、“ swtch ”(2189，2281)、“ expand ”(2284)、“ trap1 ”(2846)和“ xswap ”(4476，4477)调用。

将r5和r6的值存放到一个数组中，该数组的地址作为一个参数传递至“ savu ”。

8.7 retu(0740)

此过程由“ swtch ”(2193，2228)和“ expand ”(2294)调用。

它复位第7个核心态段地址寄存器，然后从“ u.u_rsav ”最新可存取副本中复位 r6和r5(注意，u.u_rsav位于“ u ”的开始处)。

8.8 aretu(0734)

此过程由“ sleep ”(2106)和“ swtch ”(2242)调用。它从作为参数传递过来的地址处重装

r6和r5。

8.9 swtch(2178)

“swtch”由“trap”(0770，0791)、“sleep”(2084，2093)、“expand”(2287)、“exit”(3256)、“stop”(4027)和“xalloc”(4480)调用。

“swtch”是一个非常特殊的过程，它分三段执行，一般而言，涉及3个不同的核心态进程。其中，第1和第3个进程分别被称为“退休”(retiring)进程和“上升”(arising)进程。第2个进程总是#0进程，它也可以是“退休”或“上升”进程。

注意，“swtch”使用的变量都是寄存器型、全局或静态(全局存放)变量。

2184：静态“proc”结构指针“p”规定了搜索“proc”数组的起点，搜索的目的是找到下一个要激活的进程。将“p”定义为静态变量就可以在两次调用“swtch”之间保持其值，也就保持了搜索的偏置值。若“p”值为null，则将其值设置为“proc”数组的起始地址。这只在第1次调用“swtch”时才发生此种情况。

2189：调用“savu”(0725)保存环境和栈指针。

2193：“retu”(0740)复位r5和r6，而最重要的是将核心态地址寄存器#6设置为调度进程的数据段地址。

2195：第2段从此开始：

从此行开始到2224行的代码只由核心态进程#0执行。其中包含了两层嵌套循环，除非找到一可运行进程，否则不会从其中退出。

在系统没有准备就绪且映像在内存的进程时，处理机消耗其大部分时间执行 2220行。只有发生中断时(例如时钟中断)才会终止执行此行代码。

2196：清“runrun”标志，该标志指示一个较当前进程具有更高优先权的进程已为运行准备就绪。“swtch”将寻找最高优先权进程。

2224：将“上升”进程的优先数存放在全局变量“curpri”中，以便将来引用和比较。

2228：对“retu”的另一次调用，它将r5、r6和第7个核心态地址寄存器设置为与“上升”进程相适应的值。

2229：第3段从此开始：

“sureg”(1739)用为“上升”进程存储的原型值复位用户态硬件段寄存器组。

2230：从此行开始的注释并不令人鼓午。在本章将结束处，我们还会返回到此处。

2247：经仔细检查后，你会发现没有一个调用“swtch”的过程直接检查此行的返回值。只有调用“newproc”的过程才对此值有兴趣。所创建子进程的第一次激活与此值有关。

8.10 setpri(2156)

2161：按下面的公式计算进程优先数：

优先数=min{127,(使用的时间+PUSER+p_nice)}

其中：

1) 使用的时间=进程最近一次换入以来累计使用的CPU时间(滴答)除以16，亦即使用时间的测量单位是约1/3秒。(当讨论时钟中断时，还将对此进行讨论。)

2) PUSER==100。

3)“p_nice”是计算进程优先数的偏置值。p_nice值通常为正，因此减少了进程的有效优先权。

注意，UNIX中的一条易于使人迷惑的惯例是：优先数(p_pri)愈小，优先权愈高。于是优先数－10的优先权高于优先数100的优先权。

2156：如果刚计算优先数进程的优先权低于当前进程，则设置再调度标志。

2165行测试的含意令人惊讶，特别是与2141行进行比较时。我们将此留给读者思考，以找到为什么这不是一个错误的理由。

8.11 sleep(2066)

当一个核心态进程要将自己挂起时调用此过程。(在代码中大约有30处调用此过程。)此过程有两个参数：

- 要进入睡眠状态的原因。
- 一优先数，该进程以后被唤醒后起作用。

若此优先数为负，则一个信号的到达不会使该进程不进入睡眠状态。第13章将讨论“信号”。

2070：保存当前处理机状态，包括处理机优先级和前状态信息。

2072：若优先数非负，则测试是否接收到信号。

2075：从此开始一个小的临界区，其中改变进程状态，调用此过程的两个参数则存放在通常可存取的单元中(在proc结构中)。

因为这几个信息字段可被“wakeup”(2113)查询和更改，所以这一代码段是临界区，“wakeup”经常由中断处理程序调用。

2080：若“runin”标志非0，则调度进程(#0进程)正在睡眠，以等待将另一进程换进内存。

2084：对“swtch”的调用提供了一个未知长度的延迟，(亦即从此次放弃处理机到下一次被再次切换占用处理机之间的时间长度是不确定的。)在此期间可能发生相关的外部事件。因此对“issig”(2085)的第二次测试是合适的。

2087：对于负优先数“睡眠”，进程典型地等待系统表空间的释放，为了加快此种活动的进展速度，不允许一“信号”的发生。

8.12 wakeup(2133)

此过程与“sleep”互补。它简单地搜索进程集，寻找由于指定的原因(作为参数“chan”给出)而睡眠的所有进程，并调用“setrun”分别激活它们。

8.13 setrun(2134)

2140：将进程状态设置为“SRUN”。“swtch”和“sched”将把此进程视为可再次执行的

候选者。

2141：若该进程的优先数低于当前进程，亦即它比当前进程更重要，则设置重新调度标志“ runrun ”，以便以后引用。

2143：若“ sched ”正睡眠，等待一个进程“ 换入 ”，而且刚被唤醒的进程位于磁盘上，则唤醒“ sched ”。

因为“ sched ”是唯一以“ chan ”为“ &runout ”调用“ sleep ”的过程，2145行可以代换成下列形式的递归调用：

```
setrun(&proc〔0〕);
```

或者代换成更好一些的下列代码：

```
rp=&proc〔0〕;
goto sr;
```

其中，sr是要插在2139行开始处的标号。

8.14 expand(2268)

在本过程开始处(2251)的注释对此过程作了几乎全部必要的说明，唯一需要补充之处与内存不足时要进行的换出操作有关。

注意，“ expand ”并不特别关注用户数据区或栈区的内容。

2277：若扩充实际上是收缩，则从高地址端释放多余的存储区。

2281：调用“ savu ”将r5和r6的值存入“ u.u_rsav ”。

2283：如无足够内存可供使用…

2284：环境指针和栈指针再次存放至“ u.u_ssav ”中。但是，应当注意：因为没有进入新过程，栈也并未增长，所以存放到“ u.u_ssav ”中的值与2281行的相同。

2285：“ xswap ”(4368)将其第1个参数所指定进程的内存映像复制到磁盘上。

因为第2个参数值非0，所以将该进程数据段占用的内存区释放回可用空间列表中 (即coremap中)。

但是，在“ swtch ”中再次调用“ retu ”之前，仍继续使用内存中的同一区。

注意：在已经复制了磁盘映像后，“ swtch ”中在2189行所调用的“ savu ”将新值存入“ u.u_rsav ”(因为内存映像已被正式放弃，所以这并无实际用处)。

2286：在该进程“ proc ”结构的“ p_flag ”中增加设置“ SSWAP ”标志。(proc结构不会被换出，因此SSWAP标志随时可以检查。)

2287：调用“ swtch ”，于是仍在其原占用区运行的进程将自身挂起。因为调用“ xswap ”使“ SLOAD ”标志被清除，所以“ swtch ”不会立即再激活此进程。

只有该进程在磁盘上的映像再次被复制到内存中后，才可能再次激活它。“ swtch ”执行的“ return ”，将控制转移调用“ expand ”的过程。

8.15 再回到swtch

当一个进程再次被激活，也就是变成了“ swtch ”中的“ 上升 ”进程时将发生什么呢？

2228：从“ u.u_rsav ”中恢复栈和环境指针 (注意，指向u的指针也是指向u.u_rsav的指针(0415))，但是……

2240：若内存映像已被换出(例如被expand换出)……

2242：栈和环境指针的值已经不再能继续使用，从“ u.u_ssav ”中复位它们的值。

问题是如果在2284行存入“ u.u_ssav ”中的值与2281行存入“ u.u_rsav ”的值相同，那么它们又怎样会变成不同了的呢？

对此问题的回答需要细致的分析、推敲，也有相当难度，因此才会有 2238行的注释“ 并不期望你能理解这一类。”请先将“ xswap ”分析透彻，而最后的答案将可在第 15章中找到。现在你可能急于弄清楚这一问题，那么请立即加入“ 2238 ”俱乐部。

8.16 临界区

若2个或多个进程对同一数据集进行操作，那么这些进程的组合输出将依赖于它们的相对同步(依赖于它们对同一数据集操作的前后顺序)。这通常是非常不希望发生的，应想方设法加以避免。解决方法一般是在由每个进程执行的代码中定义“ 临界区 ” (组织临界区是程序员的责任)。程序员应当保证在任一时刻不会有几个进程同时执行存取特定数据集的代码段。

在UNIX中用户进程不共享数据，所以并不会产生这种形式的冲突。但是核心态进程会存取多种系统数据，于是会产生冲突。

在UNIX中，中断并不直接造成进程的改变。只有核心态进程显式调用“ sleep ”，才可能将它们自己挂起在一个临界区中，此时确实需要引进一显式的锁变量。甚至在这种情况下，对锁变量的测试和设置操作也无须作为一个原子操作执行。

呈序执行的。为了保护其输出会受到某些中断处理影响的
机优先级暂时提升到足够高，这样就可以推迟中断，当该
尤先级。当然，有一些中断处理程序应当遵循的约定，我

策略只对单处理机系统有效，对多处理机系统则是完全不