2020 年秋操作系统 xv6 源码阅读报告 3

进程调度

黎善达

18000129610@pku.edu.cn

2020 年 11 月 4 日

1 关键代码阅读与分析

xv6 源码中，涉及进程调度的主要文件包括 `proc.h`，`proc.c`，`swtch.S`；同时涉及一些其它文件中的宏和函数。在上一次有关进程模型的报告中，已经对这三个文件中的数据结构和函数进行了逐个分析。本次报告将再次阅读并更深入地分析与进程调度有关的内容，其它内容则会略去。

1.1 `proc.h`

该文件的主要内容是下列结构的定义：

- 结构 `cpu`，用于刻画一个 CPU 的状态信息。对进程调度而言，这里有一行代码值得注意：

  ```
  1   struct context *scheduler; // swtch() here to enter scheduler
  ```

  此处是调度进程的上下文。在 xv6 中，每个 CPU 都**有一个专门的调度器线程**负责调度；换言之，调度不是在其他进程的内核线程上进行的。
- 结构 `context`，即上下文。熟悉这一结构对理解 `swtch` 函数（见1.3）十分重要。
- 枚举类型 `procstate`，即进程状态。可以看出，**xv6 中的进程共有 6 种状态**。
- 结构 `proc`，用于刻画一个进程。这是进程模型中最核心的数据结构。
- 因为先前调用了 `yield` 函数，所以需要再次检查当前进程是否被杀死：若当前进程的权限为 `DPL_USER` 且 `killed` 位已被置 1，则退出程序。

1.2 `proc.c`

该文件包含了与进程相关的基本函数。

该文件首先声明了进程表 ptable，它包含了一个自旋锁，和一个含有 NPROC 个 proc 结构的数组。宏 NPROC 在 param.h 中被定义为 64，即支持的最大进程数是 64。我们将看到，这里的自旋锁 ptable.lock 在进程调度中对**确保调度的正确性、维护进程表、避免竞争**等方面起到了重要作用。

下面重点分析该文件中与进程调度相关的函数，并指出其中值得注意的细节。

1.2.1 void scheduler(void)

该函数是各个 CPU 的调度程序，它在一个专门的调度器线程中运行，CPU 结构体中有专门的空间存储该线程的上下文（即 struct context *scheduler）。该函数不会返回，而是先进行一些所必须的设置，然后通过 swtch 函数在被调度进程的内核线程和调度器线程不断交换控制。

具体来说，该函数执行一个永不终止的循环。在一个循环体中，首先需要开中断（其原因稍后解释），获得进程表的锁；其次按次序遍历进程表，寻找状态为 RUNNABLE 的进程。

若找到了这样的进程，则调用 switchuvm 进行内存的切换，将被找到的进程状态改为 RUNNING，将当前 CPU 进程改为被找到的进程，用 swtch 函数进行上下文切换（同时相当于将控制权转给被找到的进程）。由此，被找到的进程被调度上 CPU。

若没有找到这样的进程，则释放进程表锁，再次进入循环体。

现在考虑被找到（即被调度上 CPU）的进程让出 CPU，则该进程在让出 CPU 时必然会通过 swtch 函数进行上下文切换。由于调度该进程时，我们已经通过执行 swtch 函数将调度器线程的上下文保存在了 &cpu->scheduler 中，所以此时即将让出 CPU 的进程调用 swtch 函数完毕后，将进入 scheduler 函数的 switchkvm() 语句（详见下面所附代码），即切换进入内核内存。随后，该线程将当前 CPU 上运行的进程的指针清空，再次执行循环体以寻找下一个需要调度的进程。

涉及调度器线程和被调度进程切换的代码如下：

```
1    // Switch to chosen process. It is the process's job
2    // to release ptable.lock and then reacquire it
3    // before jumping back to us.
4    proc = p;
5    switchuvm(p);
6    p->state = RUNNING;
7    swtch(&cpu->scheduler, proc->context);
8    switchkvm();
```

最后对本函数中的一些细节进行说明。

（1）在循环体执行开始时开中断。这是为了避免如下情况：调度器线程遍历进程表发现未找到状态为 RUNNABLE 的进程可供调度，但这是因为无法运行的进程都在等待 I/O。假如调度器不周期性地允许

中断，那么这些进程等待的 I/O 将永远无法到达，而调度器也永远无法找到可供调度的进程。

（2）在循环体执行结束时释放进程表锁而在下一次遍历进程表前再次获得之。这一操作使得调度器进程周期性释放进程表锁。同样，假如调度器不如此做，那么一个不断循环而始终未找到可供调度进程的调度器将一直持有进程表锁，这导致其他 CPU 无法执行上下文切换或任何和进程相关的系统调用，从而不可能将某个进程标记为 RUNNABLE 并让当前调度器顺利找到可供调度的进程了。因此，这一操作是为了避免死锁。

（3）scheduler 函数调用 swtch 函数时仍持有 ptable.lock，被调度的进程负责释放之，且需要在再次进入 scheduler 函数前获得之。该部分源码中，有几处细节与此呼应：其一是如同 allocproc 函数所设计的，一个新进程第一次被调度时将从 forkret 函数开始运行，此处该进程会释放 ptable.lock；其二是在 sched 函数中所设计的，通过 swtch 进入 scheduler 前将再次确认已获得 ptable.lock，这体现了不同功能模块的协同配合。

1.2.2 void sched(void)

该函数调用 swtch，从被调度程序回到 scheduler 函数。该函数，如等；同时，该函数还保存 sched 函数是内核线程让出处理器（即进入调度器线程）唯一的最终出口。

该函数首先进行一系列检查，确保当前进程有进入 scheduler 函数的条件：已获得进程表锁、进程状态不是 RUNNING、中断嵌套恰有一层、中断已被关闭。若检查条件未满足，则系统将调用 panic 报告错误信息并崩溃。其次，该函数保存当前 CPU 的中断使能信息，因为中断使能信息实际上是当前进程的性质而非 CPU 的性质。这两步操作完成后，调用 swtch 函数，进行从当前进程到调度器的切换。

当一个从 sched 进入调度器的进程再次被调度上 CPU 中，它被保存的即将执行的第一行代码是 cpu->intena = intena，即将当前 CPU 的中断使能恢复为先前保存的该进程的中断使能。

1.2.3 void yield(void)

该函数使当前进程让出 CPU。具体而言，该函数首先获得进程表锁，其次将自身状态改为 RUNNABLE，然后调用 sched 函数使系统重新调度进程，达到让出 CPU 的目的。

当调用 yield 函数让出 CPU 的进程再次被调度上 CPU 时，将会从 shed 函数返回，然后释放 ptable.lock：这符合 scheduler 函数中的设计。

1.2.4 void forkret(void)

如同 allocproc 函数所设置的，一个进程的第一次被调度将会进入此处，这里将 scheduler 函数中获得的进程表锁释放（这与 scheduler 函数的设计呼应），然后返回。返回地址也已经被 allocproc 函数设置好了：即 trapret 函数，这使得当前进程“回到”用户空间。

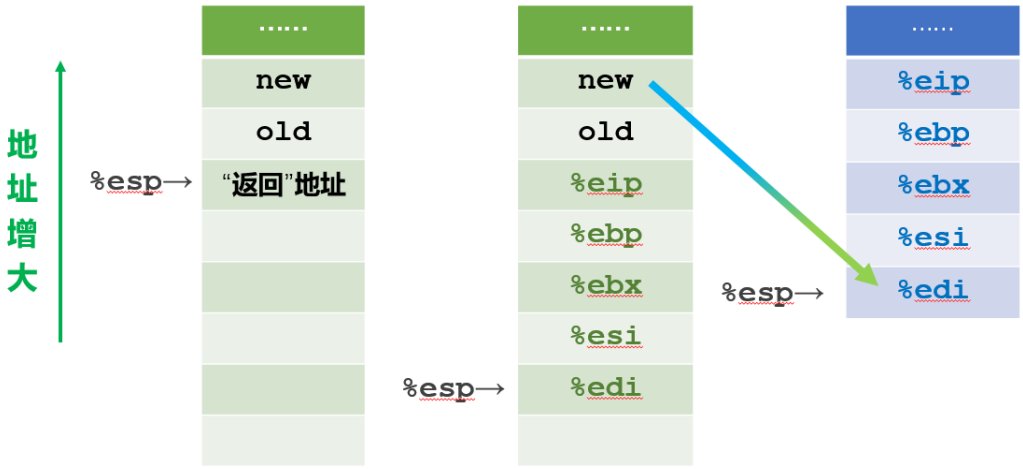
1.3 swtch.S

该文件为汇编代码，对应函数 `void swtch(struct context **old, struct context *new)`，主要行为是进行上下文切换。这一段代码虽然简短，但设计十分巧妙。函数接收的两个参数，参数 `old` 是一个用于存放 `struct context *` 类型变量的地址，稍后将会在其中存入旧进程上下文的指针；参数 `new` 是指向新进程上下文的指针。

下面展示汇编代码并逐行说明如下：

```
1    .globl swtch
2    swtch:
3      movl 4(%esp), %eax
4      movl 8(%esp), %edx
5
6      # Save old callee-saved registers
7      pushl %ebp
8      pushl %ebx
9      pushl %esi
10     pushl %edi
11
12     # Switch stacks
13     movl %esp, (%eax)
14     movl %edx, %esp
15
16     # Load new callee-saved registers
17     popl %edi
18     popl %esi
19     popl %ebx
20     popl %ebp
21     ret
```

如 `swtch.S` 节选代码第二行所示，在进入 `swtch` 函数时，栈的情况如下图左侧所示：



因此，第 3、4 行代码的行为是在栈指针%esp 还未发生改变时，将参数 old 和 new 存入寄存器%eax 和%edx 中。

第 7 10 行代码的行为是保存旧进程的上下文。注意到，调用 swtch 的函数在执行 call swtch 指令时，会将下一条指令的地址压栈（即“返回”地址），这里的地址恰好是该进程再次被调度上 CPU 后需要执行的第一条指令的地址，因此它对应的就是旧进程上下文中的程序计数器%eip。如上图中间所示，程序计数器%eip，连同刚刚压栈的 4 个寄存器，恰好构成了一个 context 结构，而此时的栈指针就是指向该结构的指针。这里，第 7-10 行代码弹栈的顺序是不能更改的：该顺序与 context 结构中变量声明的顺序对应。

第 13、14 行代码进行栈转换。由前面的分析，转换前的栈指针就是指向旧进程的 context 结构的指针，而用于保存该指针的参数 old 位于寄存器%eax 中，故有第 13 行指令 movl %esp, (%eax)；指向新进程上下文的指针存放在%edx 中，将其复制作为新的栈指针，进入上图右侧所示的新的栈。在图中，新旧进程的栈与寄存器以不同颜色区分。

第 17 21 行代码的行为是恢复新进程的上下文。首先按次序进行 4 次弹栈，回复被调用者保存寄存器，然后调用 ret 指令进行跳转，使得程序计数器%eip 变为栈中存放的新进程应继续执行的指令，从而完成了上下文切换。这里对指令 ret 的使用十分巧妙。

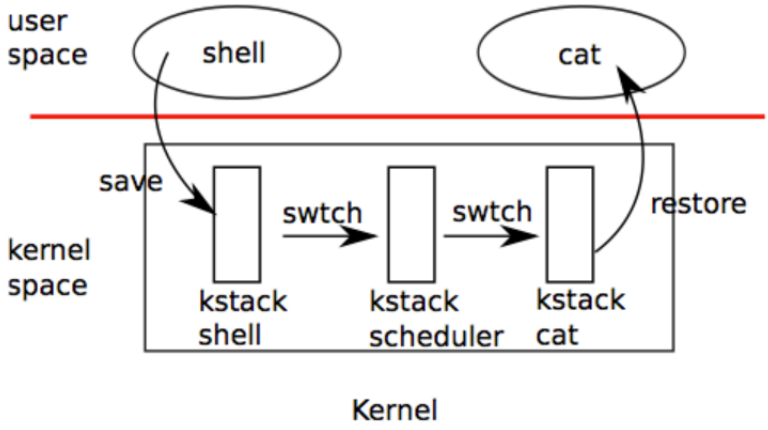
总而言之，该函数保存了旧进程的栈指针、程序计数器和通用寄存器，恢复了新进程的栈指针、程序计数器和通用寄存器，达到了在两个进程的内核栈间切换、在两段指令间切换的效果。

2 小结

本次阅读中，关乎进程调度的最核心的函数是 scheduler 和 swtch。这两个函数中，swtch 更为底层，主要完成上下文切换的具体细节，对“线程”事实上一无所知；而 scheduler 更为宏观，将 swtch 函数作为一个功能，实现宏观层面的调度。

下面从机制和策略两个角度评价 xv6 的进程调度。

机制层面，xv6 的进程调度流程如下图所示：



在 xv6 中，每个 CPU 都**有一个专门的调度器线程**负责调度；换言之，调度不是在其他进程的内核线程上进行的。因此，进行一次进程调度，将进行“内核线程 A→ 调度器线程 → 内核线程 B”的转换。调度器线程某种程度上拥有“上帝视角”，即了解其他所有进程的情况，同时拥有属于自己的栈，从而不必依赖于其他进程就可以对其他进程进行操作。通过底层的 `swtch` 实现上下文切换的具体工作，通过 `scheduler` 完成调度器线程与其他线程的切换，通过进程表锁 `ptable.lock` 确保安全性和正确性，xv6 构建了一个完善的进程调度机制。

策略层面，xv6 的进程调度算法十分朴素，调度器总是在进程表中寻找第一个处于 `RUNNABLE` 状态的进程。这可能造成的问题包括：（1）没有区分不同任务之间的重要性（优先级），可能影响系统性能和用户体验；（2）没有根据任务特点进行进程调度，在吞吐量等方面可能表现较差；（3）可能出现饥饿现象，尤其是在进程表中排序靠后的进程在这一算法中被调度的机会较小。此外，这部分许多功能的实现往往采用*遍历进程表*的方式，虽然代码显得简单易读，但会带来 $O(n)$ 的时间复杂度（其中，n 为进程数），从而导致效率的下降。