XV6 进程调度 阅读报告

本次代码阅读主要对 XV6 中的进程调度机制进行调研。主要涉及的代码有：proc.c, proc.h，swtch.S。下面我将逐个剖析，以阐释 XV6 的进程调度机制。

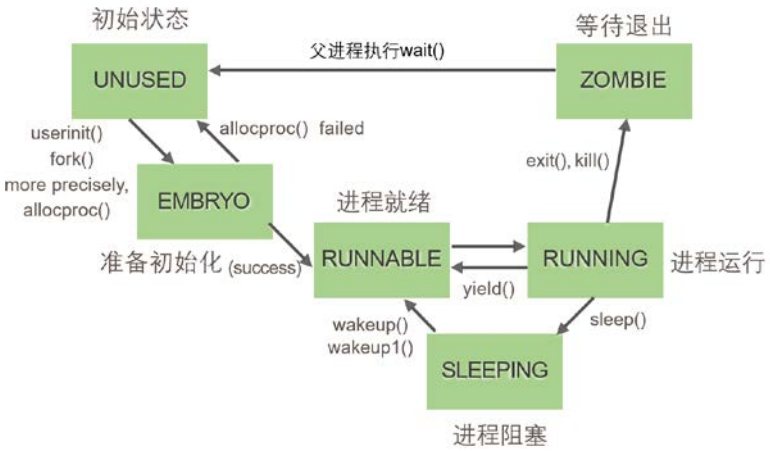
一、进程调度的时机

首先，系统启动时，当 main 函数执行 userinit 函数创建完第一个用户级进程后，会在 mpmain 函数中调用 scheduler 函数进行一次进程调度操作，如下图所示：

```
17  int
18  main(void)
19  {
20    kinit1(end, P2V(4*1024*1024)); // phys page allocator
21    kvmalloc();      // kernel page table
22    mpinit();        // collect info about this machine
23    lapicinit();
24    seginit();       // set up segments
25    cprintf("\ncpu%d: starting xv6\n\n", cpu->id);
26    picinit();       // interrupt controller
27    ioapicinit();    // another interrupt controller
28    consoleinit();   // I/O devices & their interrupts
29    uartinit();      // serial port
30    pinit();         // process table
31    tvinit();        // trap vectors
32    binit();         // buffer cache
33    fileinit();      // file table
34    iinit();         // inode cache
35    ideinit();       // disk
36    if(!ismp)
37      timerinit();   // uniprocessor timer
38    startothers();   // start other processors
39    kinit2(P2V(4*1024*1024), P2V(PHYSTOP)); // must come after startothers()
40    userinit();      // first user process
41    // Finish setting up this processor in mpmain.
42    mpmain();
43  }
```

```
56  static void
57  mpmain(void)
58  {
59    cprintf("cpu%d: starting\n", cpu->id);
60    idtinit();       // load idt register
61    xchg(&cpu->started, 1); // tell startothers() we're up
62    scheduler();     // start running processes
63  }
```

接下来让我们来回顾一下进程状态间的转换关系：



在进程模型的阅读报告中，我们分析了每个进程操作的具体流程，我们会发现，exit、yield、sleep 这三个函数中会调用 sched 函数来进行进程调度。这分别对应了以下三种情况：

1.进程执行完毕，需要进行调度，运行其他就绪进程（exit 函数）

2.进程主动让出 CPU，需要执行调度来选择其他进程（yield 函数）

3.进程遇到事件（如等待 I/O）而阻塞，为了充分利用 CPU，需要调度选择其他进程（sleep 函数）

二、进程调度的流程

在分析完进程调度的时机后，我们来分析一下进程调度的过程。对于单个 CPU 来说，XV6 的进程调度主要由 sched 和 scheduler 函数来完成。sched 函数执行一些检查工作，然后调用 scheduler 函数来完成新进程选择和上下文切换的操作。

1.调度入口：sched 函数

这个函数用于进入调度进程的上下文。首先，函数进行一些必备条件的检查，包括：（1）获取进程表的互斥锁（2）pushcli 的深度为 1，（3）当前进程不是正在运行的 RUNNING 状态（进程状态已切换）（4）中断被 disable 住。当上述条件全部满足时，程序调用 swtch 函数进行上下文切换，切换完毕后恢复原 CPU 状态。此时，函数进入了调度程序的上下文，可以准备开始执行调度了。

```
292  void
293  sched(void)
294  {
295    int intena;
296
297    if(!holding(&ptable.lock))
298      panic("sched ptable.lock");
299    if(cpu->ncli != 1)
300      panic("sched locks");
301    if(proc->state == RUNNING)
302      panic("sched running");
303    if(readeflags()&FL_IF)
304      panic("sched interruptible");
305    intena = cpu->intena;
306    swtch(&proc->context, cpu->scheduler);
307    cpu->intena = intena;
308  }
```

上下文切换的过程主要是利用汇编语言保存被调用者保存的寄存器，切换栈空间以后，恢复待调度进程的调用者保存寄存器。XV6 的进程上下文包括 4 个被调用者保存寄存器的值，以及程序计数器的值。虽然 swtch 不负责储存程序计数器，但是会有其他函数来处理%eip 的保存问题。

```
44  struct context {
45    uint edi;
46    uint esi;
47    uint ebx;
48    uint ebp;
49    uint eip;
50  };
```

```
292  void
293  sched(void)
294  {
295    int intena;
296
297    if(!holding(&ptable.lock))
298      panic("sched ptable.lock");
299    if(cpu->ncli != 1)
300      panic("sched locks");
301    if(proc->state == RUNNING)
302      panic("sched running");
303    if(readeflags()&FL_IF)
304      panic("sched interruptible");
305    intena = cpu->intena;
306    swtch(&proc->context, cpu->scheduler);
307    cpu->intena = intena;
308  }
```

2.调度算法的实现：scheduler 函数

这个函数相当于每个 CPU 中的进程调度器。这个函数不会返回，它反复执行循环，寻找可执行的进程，然后切换上下文和地址空间，执行这个进程。

具体来说，函数首先会调用 sti 函数激活处理器的中断接收功能，然后获取当前 CPU 的进程表锁，之后遍历进程表，寻找第一个可运行的进程（状态处于 RUNNABLE），然后按顺序进行：用户虚拟内存空间的切换、进程状态的切换（由 RUNNABLE 变成 RUNNING）、内核虚拟地址空间的切换。这些执行完毕后，便会转到对应进程的上下文来继续执行。执行完毕后，如果再次切换到调度函数，调度函数的上下文应处在刚刚执行完 switchkvm 的状态。这时，scheduler 会把局部进程对象指针 proc 清零，释放进程表锁以后，继续寻找可运行进程。可以看到，XV6 的调度机制是 Round Robbin 思路的。

这里，进程表锁实现了调度过程的原子化，可以解决竞争问题。

```
257  void
258  scheduler(void)
259  {
260    struct proc *p;
261
262    for(;;){
263      // Enable interrupts on this processor.
264      sti();
265
266      // Loop over process table looking for process to run.
267      acquire(&ptable.lock);
268      for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
269        if(p->state != RUNNABLE)
270          continue;
271
272        // Switch to chosen process.  It is the process's job
273        // to release ptable.lock and then reacquire it
274        // before jumping back to us.
275        proc = p;
276        switchuvm(p);
277        p->state = RUNNING;
278        swtch(&cpu->scheduler, proc->context);
279        switchkvm();
280
281        // Process is done running for now.
282        // It should have changed its p->state before coming back.
283        proc = 0;
284      }
285      release(&ptable.lock);
286
287    }
288  }
```

三、进程调度的触发

上面我们分析了进程的调度机制，下那么接下来我们来分析一下进程调度触发的时机。对于进程操作而言，有以下三种情况会触发调度

1.exit

exit 退出当前进程时，会先清除当前进程的打开文件表，然后调用 wakeup1 函数唤醒其父进程，再把当前进程的子进程挂载到 initproc 进程上，最后设置当前进程状态为 ZOMBIE。这些处理执行完毕后，函数最后调用 sched 函数，进行进程的上下文切换，调度其他进程运行，并不再返回。

```
166  void
167  exit(void)
168  {
169    struct proc *p;
170    int fd;
171
172    if(proc == initproc)
173      panic("init exiting");
174
175    // Close all open files.
176    for(fd = 0; fd < NOFILE; fd++){
177      if(proc->ofile[fd]){
178        fileclose(proc->ofile[fd]);
179        proc->ofile[fd] = 0;
180      }
181    }
182
183    iput(proc->cwd);
184    proc->cwd = 0;
185
186    acquire(&ptable.lock);
187
188    // Parent might be sleeping in wait().
189    wakeup1(proc->parent);
190
191    // Pass abandoned children to init.
192    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
193      if(p->parent == proc){
194        p->parent = initproc;
195        if(p->state == ZOMBIE)
196          wakeup1(initproc);
197      }
198    }
199
200    // Jump into the scheduler, never to return.
201    proc->state = ZOMBIE;
202    sched();
203    panic("zombie exit");
204  }
```

2.yield

这个函数会让当前进程主动释放 CPU，把当前进程设置为 RUNNABLE 状态，然后调用 sched 函数，让出 CPU 给其他进程来执行。当进程因时钟中断进入 trap 函数时，如果进程运行达到时间片最大时长，则会自动调用 yield 函数让出 CPU。

```
311  void
312  yield(void)
313  {
314    acquire(&ptable.lock);  //DOC: yieldlock
315    proc->state = RUNNABLE;
316    sched();
317    release(&ptable.lock);
318  }
```

```
if(proc && proc->state == RUNNING && tf->trapno == T_IRQ0+IRQ_TIMER)
  yield();
```

3.sleep

这个函数会把设置睡眠时长，并把当前进程设置为 SLEEPING 的阻塞状态，然后调用 sched 函数进行进程调度。当函数因外部中断或者系统调用需要等待响应或者执行信号处理函数时，对应的函数便会调用 sleep 来进行进程的切换。当到达睡眠时长时，时钟中断会在 trap 函数中唤醒这个进程。我们可以在 wait，sys_sleep，iderw，bget，ilock，begin_trans，pipewrite，piperead，consoleread 等处理函数中看到 sleep 被调用。

```
342  void
343  sleep(void *chan, struct spinlock *lk)
344  {
345    if(proc == 0)
346      panic("sleep");
347
348    if(lk == 0)
349      panic("sleep without lk");
350
351    // Must acquire ptable.lock in order to
352    // change p->state and then call sched.
353    // Once we hold ptable.lock, we can be
354    // guaranteed that we won't miss any wakeup
355    // (wakeup runs with ptable.lock locked),
356    // so it's okay to release lk.
357    if(lk != &ptable.lock){  //DOC: sleeplock0
358      acquire(&ptable.lock);  //DOC: sleeplock1
359      release(lk);
360    }
361
362    // Go to sleep.
363    proc->chan = chan;
364    proc->state = SLEEPING;
365    sched();
366
367    // Tidy up.
368    proc->chan = 0;
369
370    // Reacquire original lock.
371    if(lk != &ptable.lock){  //DOC: sleeplock2
372      release(&ptable.lock);
373      acquire(lk);
374    }
375  }
```

四、小结

本次代码阅读中，我调研了 XV6 的进程调度机制。XV6 的 CPU 进程调度大致可以总结如下：CPU 在启动时，会启动一个 schedule 进程来进行进程调度的准备。当遇到对应的条件（exit、sleep、yield 调用）时，进程会首先调用 sched 进行调度前的必要检查，然后会把上下文切换到调度进程，调度进程选择好下一个要执行的进程后，便会切换到对应的上下文中，执行新进程。XV6 中的调度算法是 Round Robbin 类的：每次从先前的位置开始，遍历进程表，选择下一个可运行的进程。当时间片用尽，或者进程因遇到其他中断/系统调用而阻塞后，便会进行进程调度操作。除此之外，为了保证每次操作的原子性，XV6 还为进程表添加了锁，以解决竞争问题。