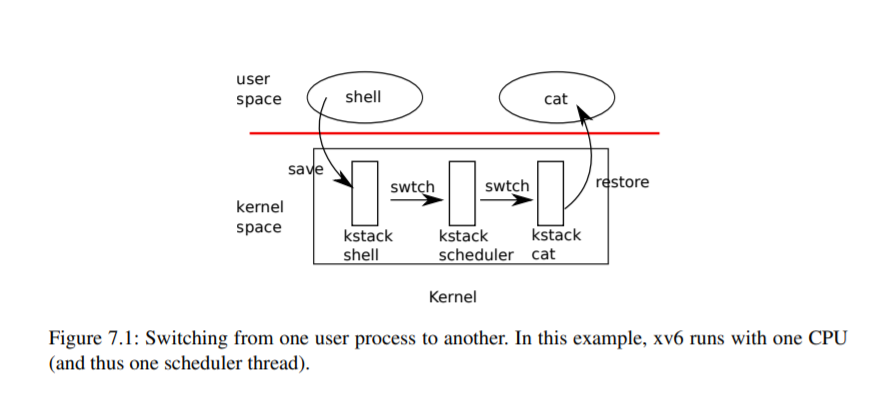
Scheduling

任何操作系统运行的进程数量都可能超过计算机的CPU数量，因此需要制定一个方案，在各进程之间分时共享CPU。理想情况下，这种共享对用户进程是透明的。一种常见的方法是通过将进程复用到硬件CPU上，给每个进程提供它有自己的虚拟CPU的假象。本章解释xv6如何实现这种复用。

### 7.1 Multiplexing

xv6通过在两种情况下将每个CPU从一个进程切换到另一个进程来实现复用。首先，xv6的sleep和wakeup机制在进程等待设备或管道I/O，或等待子进程退出，或在sleep系统调用中等待时进行切换。其次，xv6周期性地强制切换，以应对长时间的计算进程。这种复用造成了每个进程都有自己的CPU的假象，就像xv6使用内存分配器和硬件页表造成每个进程都有自己的内存的假象一样。

实现复用会带来一些挑战。首先，如何从一个进程切换到另一个进程？虽然上下文切换的想法很简单，但实现起来却很难。第二，如何对用户进程透明的强制切换？xv6采用一般的方式，用定时器中断来驱动上下文切换。第三，许多CPU可能会在进程间并发切换，需要设计一个锁来避免竞争。第四，当进程退出时，必须释放进程的内存和其他资源，但它自己不能做到这一切，因为它不能释放自己的内核栈，同时又在使用内核栈。第五，多核机器的每个内核必须记住它正在执行的进程，这样系统调用就会修改相应进程的内核状态。最后，sleep和wakeup允许一个进程放弃CPU和睡眠等待事件，并允许另一个进程唤醒第一个进程。需要注意避免会导致唤醒通知丢失的竞争。Xv6试图尽可能简单地解决这些问题，但尽管如此，写出来代码还是很棘手。



### 7.2 Code: Context switching

图7.1概述了从一个用户进程切换到另一个用户进程所涉及的步骤：用户内核转换（系统调用或中断）到旧进程的内核线程，context切换到当前CPU的调度器线程，context切换到新进程的内核线程，以及trap返回到用户级进程。xv6调度器在每个CPU上有一个专门的线程(保存的寄存器和栈)，因为调度器在旧进程的内核栈上执行是不安全的：其他核心可能会唤醒进程并运行它，而且在两个不同的核心上使用相同的栈将是一场灾难。在本节中，我们将研究在内核线程和调度线程之间切换的机制。

从一个线程切换到另一个线程，需要保存旧线程的CPU寄存器，并恢复新线程之前保存的寄存器；stack指针和pc被保存和恢复，意味着CPU将切换stack和正在执行的代码。

函数swtch执行内核线程切换的保存和恢复。swtch并不直接知道线程，它只是保存和恢复寄存器组，称为contexts。当一个进程要放弃CPU的时候，进程的内核线程会调用swtch保存自己的context并返回到调度器context。每个context都包含在一个struct context(kernel/proc.h:2)中，它本身包含在进程的struct proc或CPU的struct cpu中。Swtch接受两个参数：struct context \*old和struct context \*new。它将当前的寄存器保存在old中，从new中加载寄存器，然后返回。

让我们跟随一个进程通过swtch进入scheduler。我们在第4章看到，在中断结束时，有一种情况是usertrap调用yield。yield又调用sched，sched调用swtch将当前上下文保存在p->context中，并切换到之前保存在cpu->scheduler中的调度器上下文（kernel/proc.c:509）。

Swtch(kernel/swtch.S:3)只保存callee-saved寄存器，caller-saved寄存器由调用的C代码保存在堆栈上(如果需要)。Swtch知道struct context中每个寄存器字段的偏移量。它不保存pc。相反，swtch保存了ra寄存器[[1]](#footnote-1)，它保存了swtch被调用的返回地址。现在，swtch从新的context中恢复寄存器，新的context中保存着前一次swtch所保存的寄存器值。当swtch返回时，它返回到被恢复的ra寄存器所指向的指令，也就是新线程之前调用swtch的指令。此外，它还会在新线程的栈上返回。

### 7.3 Code: Scheduling

上一节研究了swtch的低级细节，现在我们把swtch作为一个给定的条件，研究从一个进程的内核线程通过调度器切换到另一个进程。调度器以CPU特殊线程的形式存在，每个线程运行调度器函数。这个函数负责选择下一步运行哪个进程。一个想要放弃CPU的进程，必须获取自己的进程锁p->lock，释放它所持有的其他锁，更新自己的状态（p->state），然后调用sched。Yield (kernel/proc.c:515)遵循这个惯例，就像我们稍后要研究的sleep和exit一样。Sched对这些条件进行双重检查(kernel/proc.c:499-504)，然后对这些条件进行暗示：因为锁被持有，所以中断应该被禁止。最后，sched调用swtch保存p->context中的当前上下文，并在cpu->scheduler中切换到调度器上下文。调度器继续for循环，找到一个要运行的进程，切换到它，然后循环重复。

我们刚刚看到xv6在调用swtch的过程中持有p->lock：swtch的调用者必须已经持有锁，而锁的控制权则传递给切换到的代码。这种约定对于锁来说是常见的，通常获得锁的线程也要负责释放锁。对于上下文切换来说，有必要打破这个约定，因为p->lock保护了进程的状态和context字段上的不变量，而这些不变量在swtch中执行时是不真实的。如果p->lock在swtch过程中不被持有，可能会出现问题的一个情况：在yield将其状态设置为RUNNABLE之后，但在swtch使其停止使用自己的内核栈之前，不同的CPU可能会决定运行这个进程。结果就是两个CPU运行在同一个栈上，这是不正确的。

一个内核线程在sched中放弃它的CPU，并且总是切换到调度器的同一个位置，而调度器（几乎）总是切换到之前调用sched的某个内核线程。因此，如果把xv6切换线程的行号打印出来，就会观察到下面的结果：(kernel/proc.c:475)，(kernel/proc.c:509)，(kernel/proc.c:475)，(kernel/proc.c:509)，等等。在两个线程之间发生这种风格化切换的程序有时被称为coroutine；在这个例子中，sched和调度器是彼此的coroutines。

当一个新的进程第一次被调度时，它开始于forkret (kernel/proc.c:527)。Forkret的存在是为了释放p->lock，否则，新进程需要从usertrapret开始执行。

调度器(kernel/proc.c:457)运行了一个简单的循环：找到一个要运行的进程，运行它，直到它yield，重复。调度器在进程表上循环寻找一个可运行的进程，一个有p->state == RUNNABLE的进程。一旦找到一个进程，它就会设置每个CPU当前进程c->proc，将该进程标记为RUNNING，然后调用swtch开始运行它(kernel/proc.c:470- 475)。

理解调度代码结构的一种方法是，它执行一组关于每个进程的invariant，并且每当这些invariant不为真时，就持有p->lock。一个invariant是，如果一个进程正在运行，定时器中断的yield必须能够安全地切换进程；这意味着CPU寄存器必须持有进程的寄存器值（即swtch没有将它们移到上下文中），并且c->proc必须指向进程。另一个invariant是，如果一个进程是RUNNABLE的，那么对于一个空闲的CPU调度器来说，运行它必须是安全的；这意味着p->context必须持有进程的寄存器（也就是说，它们实际上并不在真实的寄存器中），没有CPU在进程的内核栈上执行，也没有CPU的c->proc指向进程。请注意，当p->lock被持有时，这些属性往往不为真。

维护上述invariant是xv6经常在一个线程中获取p->lock，然后在另一个线程中释放的原因，例如在yield中获取，在schedululer中释放。一旦yield开始修改一个正在运行的进程的状态，使其成为RUNNABLE，锁必须一直保持，直到invariant被恢复：最早正确的释放点是在调度器（运行在自己的堆栈上）清除c->proc之后。同样，一旦调度器开始将一个RUNNABLE进程转换为RUNNING，锁就不能被释放，直到内核线程完成运行（在swtch之后，例如在yield中）。

p->lock也保护其他的东西：exit和wait之间的相互作用，避免丢失wakeup的机制（见第7.5节），以及避免进程退出和其他进程读写其状态之间的竞争（例如，退出系统调用查看p->pid并设置p->killed (kernel/proc.c:611)）。值得思考的是，是否可以将p->lock的不同功能拆分开来，这样既清晰，也可能提高性能。

### 7.4 Code: mycpu and myproc

Xv6经常需要一个指向当前进程proc结构的指针。在单核处理器上，可以用一个全局变量指向当前的proc结构。这在多核机器上是行不通的，因为每个核都执行不同的进程。解决这个问题的方法是利用每个核都有自己的一组寄存器的事实；我们可以使用其中的一个寄存器来帮助查找每个核的信息。

Xv6为每个CPU维护了一个cpu结构(kernel/proc.h:22)，它记录了当前在该CPU上运行的进程(如果有的话)，为CPU的调度线程保存的寄存器，以及管理中断禁用所需的嵌套自旋锁的计数。函数mycpu(kernel/proc.c:60)返回一个指向当前CPU结构cpu的指针。RISC-V对其CPU进行编号，给每个CPU一个hartid。Xv6确保每个CPU的hartid在内核中被存储在该CPU的tp寄存器中。这使得mycpu可以使用tp对cpu结构的数组进行索引，从而找到正确的cpu。

确保一个CPU的tp始终保持CPU的hartid是有一点复杂的。mstart在CPU启动序列的早期设置tp寄存器，此时CPU处于机器模式(kernel/start.c:46)。usertrapret 保存在trampoline页中，因为用户进程可能会修改tp寄存器。最后，当从用户空间进入内核时，uservec会恢复保存的tp(kernel/trampoline.S:70)。编译器保证永远不使用tp寄存器。如果RISC-V允许xv6直接读取当前的hartid会更方便，但这只允许在机器模式下，不允许在supervisor 模式下使用。

cpuid和mycpu的返回值是很易错的：如果定时器中断，导致线程yield，然后转移到不同的CPU上，之前返回的值将不再正确。为了避免这个问题，xv6要求调用者禁用中断，只有在使用完返回的cpu结构后才启用中断。

myproc(kernel/proc.c:68)函数返回当前CPU上运行的进程的proc结构指针。myproc禁用中断，调用mycpu，从cpu结构中获取当前进程指针(c->proc)，然后启用中断。即使启用了中断，myproc的返回值也可以安全使用：如果定时器中断将调用进程移到了不同的CPU上，它的proc结构指针将保持不变。

### 7.5 Sleep and wakeup

调度和锁有助于让一个进程对另一个进程的不可见，但到目前为止，我们还没有任何抽象来帮助进程有意地进行交互。人们发明了许多机制来解决这个问题。Xv6使用了一种叫做sleep和wakeup的机制，它允许一个进程sleep等待事件，而另一个进程在事件发生后将其wakeup。睡眠和唤醒通常被称为序列协调或条件同步机制。

为了说明这一点，让我们考虑一个叫做semaphore[4]的同步机制，它协调生产者和消费者。semaphore维护一个计数并提供两个操作。V操作（针对生产者）增加计数。P操作（针对消费者）等待，直到计数非零，然后将其递减并返回。如果只有一个生产者线程和一个消费者线程，而且它们在不同的CPU上执行，编译器也没有太过激进的优化，那么这个实现是正确的。

struct semaphore

{

  struct spinlock lock;

  int count;

};

void V(struct semaphore \*s)

{

  acquire(&s->lock);

  s->count += 1;

  release(&s->lock);

}

void P(struct semaphore \*s)

{

  while (s->count == 0)

    ;

  acquire(&s->lock);

  s->count -= 1;

  release(&s->lock);

}

上面的实现是代价很大。如果生产者很少行动，消费者将把大部分时间花在while循环中，希望得到一个非零的计数。消费者的CPU可以通过反复轮询s->count来找到比忙碌等待更有成效的工作。避免忙碌的等待需要一种方法，让消费者让出CPU，只有在V增量计数后才恢复。

这里是朝着这个方向迈出的一步，虽然我们会看到它是不够的。让我们想象一对调用，sleep和wakeup，其工作原理如下。Sleep(chan)对chan进行睡眠，称为等待通道。Sleep使调用进程进入睡眠状态，释放CPU进行其他工作。Wakeup(chan)唤醒所有在chan上sleep的进程（如果有的话），使它们的sleep调用返回。如果没有进程在chan上等待，则wakeup不做任何事情。我们更改semaphore实现，以使用sleep和wakeup。

void V(struct semaphore \*s)

{

  acquire(&s->lock);

  s->count += 1;

  wakeup(s);

  release(&s->lock);

}

void P(struct semaphore \*s)

{

  while (s->count == 0)

    sleep(s);

  acquire(&s->lock);

  s->count -= 1;

  release(&s->lock);

}

P现在放弃了CPU而不是一直检测，这是一个不错的改进。然而，事实证明，用这个接口设计sleep和wakeup并不明确，因为存在可能丢失wakeup的问题。假设执行P时s->count == 0。当P在sleep之前，V在另一个CPU上运行：它将s->count改为非零，并调用wakeup，wakeup发现没有进程在sleep，因此什么也不做。现在P继续执行：它调用sleep并进入睡眠状态。这就造成了一个问题：P正在sleep，等待一个已经发生的V调用。除非我们运气好，生产者再次调用V，否则消费者将永远等待，即使计数是非零。

这个问题的根源在于，P只有在s->count == 0时才会sleep的invariant 被V在错误的时刻运行所违反。保护这个invariant 的一个不正确的方法是移动P中的锁的获取，这样它对计数的检查和对sleep的调用是原子的：

void V(struct semaphore \*s)

{

  acquire(&s->lock);

  s->count += 1;

  wakeup(s);

  release(&s->lock);

}

void P(struct semaphore \*s)

{

  acquire(&s->lock);

  while (s->count == 0)

    sleep(s);

  s->count -= 1;

  release(&s->lock);

}

人们可能希望这个版本的P能够避免丢失的唤醒，因为锁会阻止V在s->count == 0和sleep之间执行。它做到了这一点，但它也会死锁。P在sleep时保持着锁，所以V将永远阻塞在等待锁的过程中。

我们将通过改变sleep的接口来修正前面的方案：调用者必须将条件锁传递给sleep，这样在调用进程被标记为sleeping并在sleep channel上等待后，它就可以释放锁。锁将强制并发的V等待到P完成使自己进入sleeping状态，这样wakeup就会发现sleeping的消费者并将其唤醒。一旦消费者再次被唤醒，sleep就会重新获得锁，然后再返回。我们新的正确的睡眠/唤醒方案可以使用如下。

void V(struct semaphore \*s)

{

  acquire(&s->lock);

  s->count += 1;

  wakeup(s);

  release(&s->lock);

}

void P(struct semaphore \*s)

{

  acquire(&s->lock);

  while (s->count == 0)

    sleep(s, &s->lock);

  s->count -= 1;

  release(&s->lock);

}

P持有s->lock会阻止了V在P检查c->count和调用sleep之间试图wakeup它。但是，请注意，我们需要sleep来原子地释放s->lock并使消费者进程进入sleeping状态。

### 7.6 Code: Sleep and wakeup

让我们看看sleep (kernel/proc.c:548) 和 wakeup (kernel/proc.c:582) 的实现。其基本思想是让sleep将当前进程标记为SLEEPING，然后调用sched释放CPU；唤醒则寻找一个在给定的等待通道上睡觉的进程，并将其标记为RUNNABLE。sleep和wakeup的调用者可以使用任何方便的数字作为chan。Xv6经常使用参与等待的内核数据结构的地址。

sleep获取p->lock (kernel/proc.c:559)。现在进入睡眠状态的进程同时持有p->lock和lk。在调用者(在本例中为P)中，持有lk是必要的：它保证了没有其他进程(在本例中，运行V的进程)可以调用wakeup(chan)。现在sleep持有p->lock，释放lk是安全的：其他进程可能会调用wakeup(chan)，但wakeup会等待获得p->lock，因此会等到sleep完成将进程放入睡眠状态，使wakeup不会错过sleep的进程。

有一个复杂情况：如果lk和p->lock是同一个锁，那么sleep如果试图获取p->lock，就会和自己死锁。但是如果调用sleep的进程已经持有p->lock，那么它就不需要再做任何事情来避免错过一个并发的wakeup。当wait (kernel/proc.c:582)调用sleep并持有p->lock时，就会出现这种情况。

现在sleep持有p->lock，而没有其他的锁，它可以通过记录sleep chan，将进程状态设置SLEEPING，并调用sched(kernel/proc.c:564-567)来使进程进入睡眠状态。稍后我们就会明白为什么在进程被标记为SLEEPING之前，p->lock不会被释放（由调度器）。

在某些时候，一个进程将获得条件锁，设置sleeper正在等待的条件，并调用wakeup(chan)。重要的是，wakeup是在持有条件锁[[2]](#footnote-2)的情况下被调用的。Wakeup循环浏览进程表（kernel/proc.c:582）。它获取每个被检查的进程的p->lock，因为它可能会修改该进程的状态，也因为p->sleep确保sleep和wakeup不会相互错过。当wakeup发现一个进程处于状态为SLEEPING并有一个匹配的chan时，它就会将该进程的状态改为RUNNABLE。下一次调度器运行时，就会看到这个进程已经准备好运行了。

为什么sleep和wakeup的锁规则能保证sleep进程不会错过wakeup？sleep进程从检查条件之前到标记为SLEEPING之后的这段时间里，持有条件锁或它自己的p->lock或两者都持有。调用唤醒的进程在唤醒的循环中持有这两个锁。因此，waker要么在消费者检查条件之前使条件为真；要么waker的wakeup严格地在wakeup线程被标记为SLEEPING之后检查它。 那么wakeup就会看到这个sleep进程，并将其唤醒（除非有其他事情先将其wakeup）。

有时会出现多个进程在同一个chan上sleep的情况；例如，有多个进程从管道中读取数据。调用一次wakeup就会把它们全部唤醒。其中一个进程将首先运行，并获得sleep参数传递的锁，（在管道的情况下）读取任何数据都会在管道中等待。其他进程会发现，尽管被唤醒了，但没有数据可读。从他们的角度来看，唤醒是虚假的，他们必须再次睡眠。出于这个原因，sleep总是在一个检查条件的循环中被调用。

如果两次使用sleep/wakeup不小心选择了同一个通道，也不会有害：它们会看到虚假的唤醒，如上所述的循环可以容忍这个问题。睡眠/唤醒的魅力很大程度上在于它既是轻量级的（不需要创建特殊的数据结构来充当睡眠通道），又提供了一层间接性（调用者不需要知道他们正在与哪个具体的进程交互）。

### 7.7 Code: Pipes

一个使用sleep和wakeup来同步生产者和消费者的更复杂的例子是xv6的管道实现。我们在第1章看到了管道的接口：写入管道一端的字节被复制到内核缓冲区，然后可以从管道的另一端读取。未来的章节将研究管道如何支持文件描述符，但我们现在来看一下pipewrite和piperead的实现吧。

每个管道由一个结构体 **pipe**表示，它包含一个锁和一个数据缓冲区。**nread**和**nwrite**两个字段统计从缓冲区读取和写入的字节总数。缓冲区呈环形：在**buf[PIPESIZE-1]**之后写入的下一个字节是**buf[0]**。计数不呈环形。这个约定使得实现可以区分满缓冲区(**nwrite == nread+PIPESIZE**)和空缓冲区(**nwrite == nread**)，但这意味着对缓冲区的索引必须使用**buf[nread % PIPESIZE]**，而不是使用**buf[nread]**(nwrite也是如此)。

假设对**piperead**和**pipewrite**的调用同时发生在两个不同的CPU上。**Pipewrite** (kernel/pipe.c:77)首先获取管道的锁，它保护了计数、数据和相关的invariant。然后，**Piperead** (kernel/pipe.c:103)也试图获取这个锁，但是不会获取成功。它在acquire(kernel/spinlock.c:22)中循环，等待锁的到来。当**piperead**等待时，**pipewrite**会循环写，依次将每个字节添加到管道中(kernel/pipe.c:95)。在这个循环中，可能会发生缓冲区被填满的情况(kernel/pipe.c:85)。在这种情况下，pipewrite调用wakeup来提醒所有睡眠中的reader有数据在缓冲区中等待，然后在**&pi->nwrite**上**sleep**，等待reader从缓冲区中取出一些字节。**Sleep**函数内会释放**pi->lock**，然后pipwrite进程睡眠。

现在**pi->lock**可用了，piperead设法获取它并进入它的临界区：它发现pi->nread != pi->nwrite (kernel/pipe.c:110) (pipewrite进入睡眠状态是由于pi->nwrite == pi->nread+PIPESIZE (kernel/pipe.c:85))，所以它进入for循环，将数据从管道中复制出来(kernel/pipe.c:117)，并按复制的字节数增加nread。现在又可写了，所以 piperead 在返回之前调用 wakeup (kernel/pipe.c:124) 来唤醒所有sleep的写入者。Wakeup找到一个在&pi->nwrite上sleep的进程，这个进程正在运行pipewrite，但在缓冲区填满时停止了。它将该进程标记为RUNNABLE。

管道代码对reader和writer使用单独的sleep chan（pi->nread和pi->nwrite）；这可能会使系统在有多个读取器和写入器等待同一个管道的情况下更有效率。管道代码在循环内sleep，检查sleep条件；如果有多个reader 和 writer，除了第一个被唤醒的进程外，其他进程都会看到条件仍然是假的，然后再次sleep。

### 7.8 Code: Wait, exit, and kill

sleep和wakeup可以用于许多种等待。在第1章中介绍的一个有趣的例子是，一个子进程的退出和其父进程的等待之间的交互。在子进程退出的时候，父进程可能已经在等待中sleep了，也可能在做别的事情；在后一种情况下，后续的wait调用必须观察子进程的退出，也许是在它调用exit之后很久。xv6在wait观察到子进程退出之前，记录子进程退出的方式是让exit将调用进程设置为ZOMBIE状态，在那里停留，直到父进程的wait注意到它，将子进程的状态改为UNUSED，复制子进程的退出状态，并将子进程的进程ID返回给父进程。如果父进程比子进程先退出，父进程就把子进程交给init进程，而init进程则持久的调用wait；这样每个子进程都有一个“父进程”来清理。主要的实现挑战是父进程和子进程的wait和exit，以及exit和exit之间可能出现竞争和死锁的情况。

Wait使用调用进程的p->lock作为条件锁，以避免唤醒丢失，它在开始时获取该锁（kernel/proc.c:398）。然后它扫描进程表。如果它发现一个处于ZOMBIE状态的子进程，它释放这个子进程的资源和它的proc结构，将子进程的退出状态复制到提供给wait的地址(如果它不是0)，并返回子进程的ID。如果wait找到了子进程但没有一个退出，它调用sleep等待其中一个子进程退出(kernel/proc.c:445)，然后再次扫描。这里，在sleep中释放的条件锁是等待进程的p->锁，也就是上面提到的特殊情况。请注意，wait经常持有两个锁；它在试图获取任何子锁之前，会先获取自己的锁；因此xv6的所有锁都必须遵守相同的锁顺序（父进程的锁，然后是子进程的锁），以避免死锁。

Wait会查看每个进程的np->parent来寻找它的子进程。它使用 np->parent 而不持有 np->lock，这违反了共享变量必须受锁保护的通常规则。但是np有可能是当前进程的祖先，在这种情况下，获取np->lock可能会导致死锁，因为这违反了上面提到的顺序。在这种情况下，在没有锁的情况下检查np->parent似乎是安全的；一个进程的父字段只有它的父字段才会被改变，所以如果np->parent==p为真，除非当前进程改变它，否则该值就不会改变。

Exit (kernel/proc.c:333)记录退出状态，释放一些资源，将所有子进程交给init进程，在父进程处于等待状态时唤醒它，将调用进程标记为zombie，并永久放弃CPU。最后的序列有点棘手。退出的进程必须持有父进程的锁，同时将其状态设置为ZOMBIE并唤醒父进程，因为父进程的锁是条件锁，可以防止在等待中丢失wakeup。子代也必须持有自己的p->lock，否则父进程可能会看到它的状态为ZOMBIE，并在它还在运行时释放它。锁的获取顺序对避免死锁很重要：因为wait在子锁之前获取父锁，所以exit必须使用相同的顺序。

Exit 调用了一个专门的唤醒函数 wakeup1，它只唤醒父函数，而且只有父进程在wait中sleep的情况下才会唤醒(kernel/proc.c:598)。在将父进程的状态设置为ZOMBIE之前，子进程唤醒父进程可能看起来并不正确，但这是安全的：尽管wakeup1可能会导致父进程运行，但wait中的循环不能检查子进程，直到子进程的p->lock被调度器释放为止，所以wait不能查看退出的进程，直到exit将其状态设置为ZOMBIE之后(kernel/proc.c:386)。

exit允许一个进程自行终止，而kill（kernel/proc.c:611）则允许一个进程请求另一个进程终止。如果让kill直接摧毁进程，那就太复杂了，因为相应进程可能在另一个CPU上执行，也许正处于更新内核数据结构的敏感序列中。因此，kill的作用很小：它只是设置进程的p->killed，如果它在sleep，则wakeup它。最终，进程会进入或离开内核，这时如果p->killed被设置，usertrap中的代码会调用exit。如果进程在用户空间运行，它将很快通过进行系统调用或因为定时器（或其他设备）中断而进入内核。

如果进程处于睡眠状态，kill调用wakeup会使进程从睡眠中返回。这是潜在的危险，因为正在等待的条件可能不为真。然而，xv6对sleep的调用总是被包裹在一个while循环中，在sleep返回后重新测试条件。一些对sleep的调用也会在循环中测试p->killed，如果设置了p->killed，则放弃当前活动。只有当这种放弃是正确的时候才会这样做。例如，管道读写代码如果设置了killed标志就会返回；最终代码会返回trap，trap会再次检查标志并退出。

一些xv6 sleep循环没有检查p->killed，因为代码处于多步骤系统调用的中间，而这个调用应该是原子的。virtio驱动(kernel/virtio\_disk.c:242)就是一个例子：它没有检查p->killed，因为磁盘操作可能是一系列写操作中的一个，而这些写操作都是为了让文件系统处于一个正确的状态而需要的。一个在等待磁盘I/O时被杀死的进程不会退出，直到它完成当前的系统调用和usertrap看到killed的标志。

### 7.9 Real world

xv6 调度器实现了一个简单的调度策略，它依次运行每个进程。这种策略被称为round robin。真正的操作系统实现了更复杂的策略，例如，允许进程有优先级。这个想法是，一个可运行的高优先级进程将被调度器优先于一个可运行的低优先级进程。这些策略可能会很快变得复杂，因为经常有相互竞争的目标：例如，操作者可能还想保证公平性和高吞吐量。此外，复杂的策略可能会导致不尽人意的交互，如优先级倒置和convoys。当低优先级和高优先级进程共享一个锁时，就会发生优先级倒置，当低优先级进程获得锁时，就会阻止高优先级进程的进展。当许多高优先级进程都在等待一个获得共享锁的低优先级进程时，就会形成一个长长的等待进程的车队；一旦convoys形成，就会持续很长时间。为了避免这类问题，在复杂的调度器中需要额外的机制。

sleep和wakeup是一种简单有效的同步方法，但还有很多其他的方法。在所有这些方法中，第一个挑战是避免我们在本章开头看到的丢失wakeup问题。最初的Unix内核的sleep只是禁用了中断，这已经足够了，因为Unix运行在单CPU系统上。因为xv6运行在多处理器上，所以它增加了一个显式的sleep锁。FreeBSD 的 msleep 采用了同样的方法。Plan 9的sleep使用了一个回调函数，它在进入睡眠前保持调度锁的情况下运行；这个函数的作用是在最后一刻检查sleep情况，以避免wakeup丢失。Linux内核的sleep使用一个显式的进程队列，称为等待队列，而不是等待通道；队列有自己的内部锁。

在wakeup过程中扫描整个进程列表，寻找有匹配的chan的进程，效率很低。一个更好的解决方案是用一个数据结构代替sleep和wakeup中的chan，该结构上存放着sleep的进程列表，比如Linux的等待队列。Plan 9的sleep和wakeup将该结构称为rendezvous point或Rendez。许多线程库将同一个结构称为条件变量；在这种情况下，睡眠和唤醒的操作被称为wait和signal。所有这些机制都有相同的机制：sleep条件被sleep期间原子地释放的锁保护。

wakeup唤醒了所有在某个特定chan上等待的进程，可能很多进程都在等待这个特定chan。操作系统会调度所有这些进程，它们会争相检查sleep条件。以这种方式行事的进程有时被称为thundering herd，最好避免这种情况。大多数条件变量都有两个唤醒的基元：信号（signal），唤醒一个进程；广播（broadcast），唤醒所有等待的进程。

Semaphores通常用于同步。count通常对应于类似于管道缓冲区中可用的字节数或一个进程拥有的僵尸子进程的数量。使用显式计数作为抽象的一部分，可以避免丢失wakeup的问题：有一个显式的计数，说明已经发生的唤醒次数。该计数还避免了虚假wakeup和thundering herd问题。

终止进程和清理进程在xv6中引入了很多复杂性。在大多数操作系统中，它甚至更加复杂，因为，例如，被杀进程可能trap在内核中sleep，而解除它的堆栈需要很多仔细的编程。许多操作系统使用显式的异常处理机制来解除堆栈，比如longjmp。此外，还有其他一些事件可以导致一个sleep的进程被唤醒，即使它正在等待的事件还没有发生。例如，当一个Unix进程处于睡眠状态时，另一个进程可能会向它发送一个信号。在这种情况下，该进程将从中断的系统调用中返回，值为-1，错误代码设置为EINTR。应用程序可以检查这些值并决定做什么。Xv6不支持信号，也就不会出现这种复杂性。

Xv6对kill的支持并不完全令人满意：有些sleep循环可能应该检查p->killed。一个相关的问题是，即使是检查p->killed的sleep循环，在sleep和kill之间也会有一个竞争；后者可能会设置p->killed，并试图在被杀进程的循环检查p->killed之后，但在它调用sleep之前唤醒被杀进程。如果这个问题发生了，被杀进程不会注意到p->killed，直到它所等待的条件发生。这可能会晚很多（例如，当virtio驱动返回一个被杀进程正在等待的磁盘块时），也可能永远不会发生（例如，如果被杀进程正在等待来自控制台的输入，但用户没有键入任何输入）。

真正的操作系统会在常数时间内用显式的free list来寻找free的proc结构，而不是在allocproc中进行线性时间的搜索；xv6为了简单起见，使用了线性扫描的方式。

1. 函数return地址 [↑](#footnote-ref-1)
2. 严格地说，如果只是在acquire之后wakeup就足够了(也就是说，可以在release之后调用wakeup)。 [↑](#footnote-ref-2)