
哈爾濱工業大學（深圳）

HARBIN INSTITUTE OF TECHNOLOGY, SHENZHEN

# 操作系统 (Operating System)

# xv6原理简析：进程管理

操作系统课程组
哈尔滨工业大学（深圳）

2022年秋季

Email：xiawen@hit.edu.cn

# xv6进程管理



- 和很多操作系统一样，xv6使用进程来代表一个运行程序的实例，内核通过管理进程来管理程序的运行。进程是实现隔离的重要手段。它可以被动态的**创建(fork)和退出(exit\wait)**，它可以被内核**调度**来实现分时占用CPU，也可以被**阻塞和唤醒**。
- 与Linux等现代操作系统相比，xv6的进程管理模块很简单：
  - 内核不支持线程
  - 不支持信号
  - 只实现了管道这种进程间通信方式
  - 只支持一种调度器，而且调度算法简单直接

- 进程管理相关代码主要位于kernel目录下：
  - proc.c/h、sleeplock.c/h、spinlock.c/h、exec.c

# xv6进程的基本概念



- xv6中代表进程控制块的数据结构为**struct proc** *(/kernel/proc.h:85)*，里面存放着内核管理进程所需要的全部信息，包括：
  - **自旋锁**：保护PCB的锁，保证互斥访问PCB
  - **状态信息**：state, killed, xstate
  - **chan**：用于进程的阻塞和唤醒
  - **文件管理相关**：文件描述符表ofile，当前目录cwd
  - **用户地址相关**：用户页表pagetable，用户程序大小sz
  - **上下文**：用户态/内核态切换上下文trapframe，
    进程之间的切换上下文context
  - 还有进程号pid、内核栈地址kstack等

```
85  // Per-process state
86  struct proc {
87    struct spinlock lock;
88
89    // p->lock must be held when
90    enum procstate state;
91    struct proc *parent;
92    void *chan;
93    int killed;
94    int xstate;
95    int pid;
96
97    // these are private to the p
98    uint64 kstack;
99    uint64 sz;
100   pagetable_t pagetable;
101   struct trapframe *trapframe;
102   struct context context;
103   struct file *ofile[NOFILE];
104   struct inode *cwd;
105   char name[16];
106 };
```

- 给进程分配资源就是把资源记录在PCB中（比如文件描述符），进程的状态转移就是修改PCB的state字段。可以说，管理进程就是管理PCB。

# xv6进程的基本概念



- **进程表**：所有进程控制块都放在一个全局数组（常驻内核区里面的数据段），即进程表（*kernel/proc.c:11）中*。

```
struct proc proc[NPROC];
```

- **allocproc函数**（*kernel/proc.c:108*）负责在进程表中分配一个空闲位置来保存PCB。

- 如果要获取该CPU正在运行的进程PCB，就需要借助cpu表了。

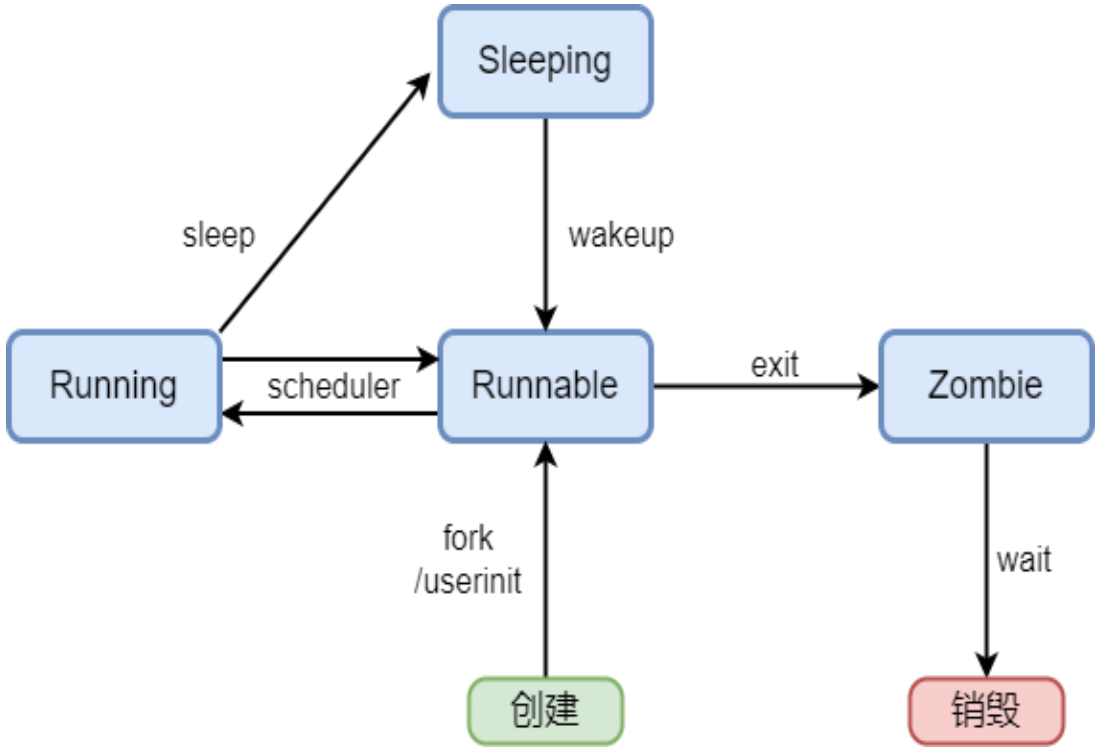
- CPU**表**：CPU表中包含着各个CPU的运行信息（包含正在运行的进程）。在OS启动时，会将cpu号存储到**tp寄存器**里（*kernel/start.c:50*），编译器会保证内核代码不会乱动tp寄存器。所以在内核中，我们可以通过tp寄存器的值作为下标访问cpu表来获得当前cpu的信息，进而获取当前正在执行的进程（详见myproc函数，*proc.c:82)*

```
struct cpu cpus[NCPU];
```

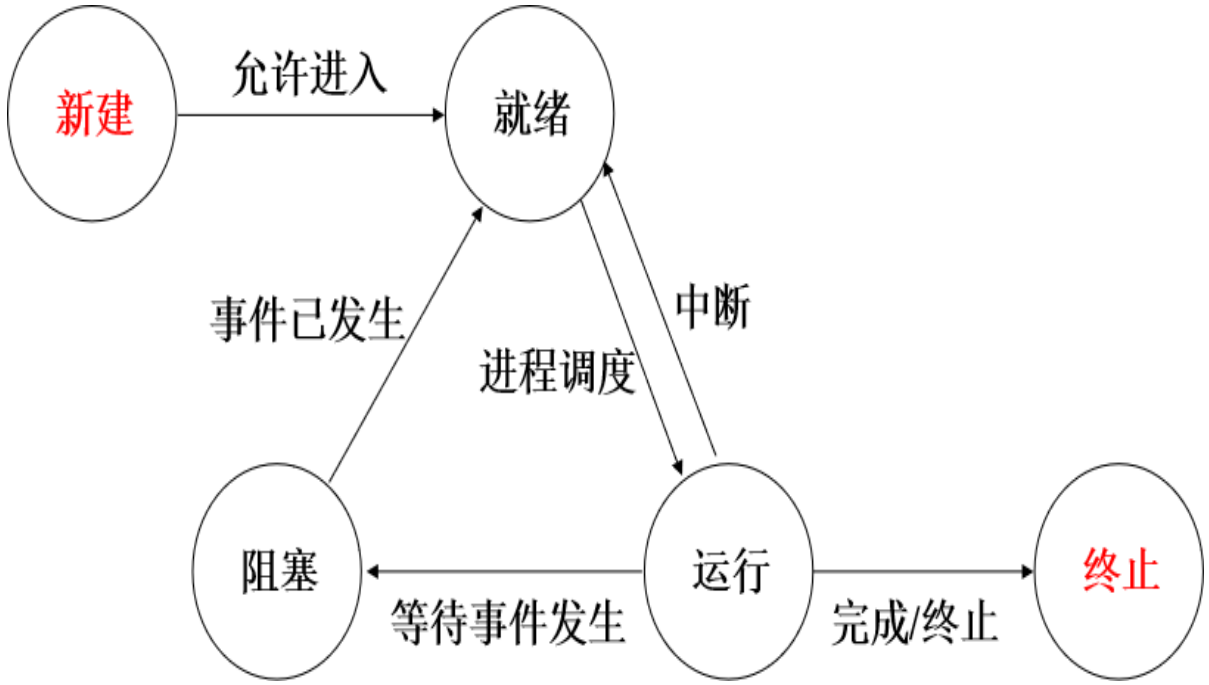
# 进程的状态



xv6里面的进程有4种状态：**阻塞SLEEPING，可运行RUNNABLE（就绪），正在运行RUNNING，僵尸状态ZOMBIE。**




**XV6进程状态**




**进程五状态（教材通用）**

# 进程的创建、退出、回收



- xv6中，除了初始进程是由内核调用userinit创建的，**其它进程都是由用户使用fork系统调用创建的。**
- **fork**（*kernel/proc.c：279*）-->**进程创建**：主要就是将parent进程的各种信息（**用户地址空间内存及页表**，用户上下文trapframe，文件描述符等）复制给新进程。parent进程会通过修改子进程PCB中的trapframe→a0来使子进程的fork系统调用返回0。

  *(注：可以使用copy-on-write来大幅提高fork速度，可以看xv6官方实验Lab: Copy-on-Write Fork)*
- **exit** (kernel/proc.c:346)-->**进程退出**：xv6进程exit时只是关闭了文件，并且设置退出码xstate。调用reparent将子进程交给初始进程收养。其它资源的回收是由父进程负责的。
- **wait**(kernel/proc.c:390)-->**回收进程**：
  - wait会遍历进程表，找到状态为ZOMBIE的子进程。
  - 如果找到ZOMBIE子进程，会通过freeproc回收子进程占用资源，完成wait系统调用。
  - 如果没有找到ZOMBIE子进程，父进程就会阻塞，等待某个子进程exit时唤醒自己，唤醒后重复上述过程（所以wait是阻塞的）。

# kill函数的实现（系统调用）



- 用户程序可以通过kill来杀死指定pid的进程
  - kill（kernel/proc.c:618）会遍历进程表，通过pid找到目标进程
  - 将目标进程PCB的**killed**字段置1，如果目标进程被阻塞了，就将其state强制改为**RUNNABLE（就绪）**
  - 进程并不会被立刻杀死，进程在内核态返回用户态时（即发生在usertrap函数内部，比如系统调用返回）会检查自己的killed字段是否被置为1，如果是，就主动执行**exit**
  - **注意这里实现了类似信号（kill-9）的功能**

```
617  int
618  kill(int pid)
619  {
620    struct proc *p;
621
622    for(p = proc; p < &proc[NPROC]; p++){
623      acquire(&p->lock);
624      if(p->pid == pid){
625        p->killed = 1;
626        if(p->state == SLEEPING){
627          // Wake process from sleep().
628          p->state = RUNNABLE;
629        }
630        release(&p->lock);
631        return 0;
632      }
633      release(&p->lock);
634    }
635    return -1;
636  }
```

kernel/proc.c

# 进程调度



- 操作系统通过进程调度使得各进程分时共享CPU。进程被调度的原因有很多，比如时钟中断、被阻塞、主动让出CPU（调用yield函数）等等。
- 进程的调度需要解决几个问题：
  - 怎么选择下一个要执行的进程，即**调度算法**。
  - 进程调度应该由谁执行？（**调度器线程**）
  - 怎么从当前进程切换到要执行进程，即**进程间上下文的切换**。
- 这涉及到几个重要函数：
  - swtch：负责上下文切换
  - scheduler：进程调度核心函数，包含进程调度算法
  - sched：调度器scheduler的入口，会调用swtch切换到调度器线程

# 进程调度：上下文切换



- swtch函数负责进程上下文切换
  - swtch函数是由汇编语言编写的。它会将当前CPU的寄存器保存到context old中，并将context new加载到当前CPU的寄存器中。通过保存和恢复寄存器实现了上下文切换。

```
void swtch(struct context* old, struct context* new);
```

  - 进程p通过如下调用切换到调度器线程

```
swtch(&p->context, &mycpu()->context);
```

(proc.c:501 sched函数内部)

  - 调度器线程通过如下调用切换到进程p

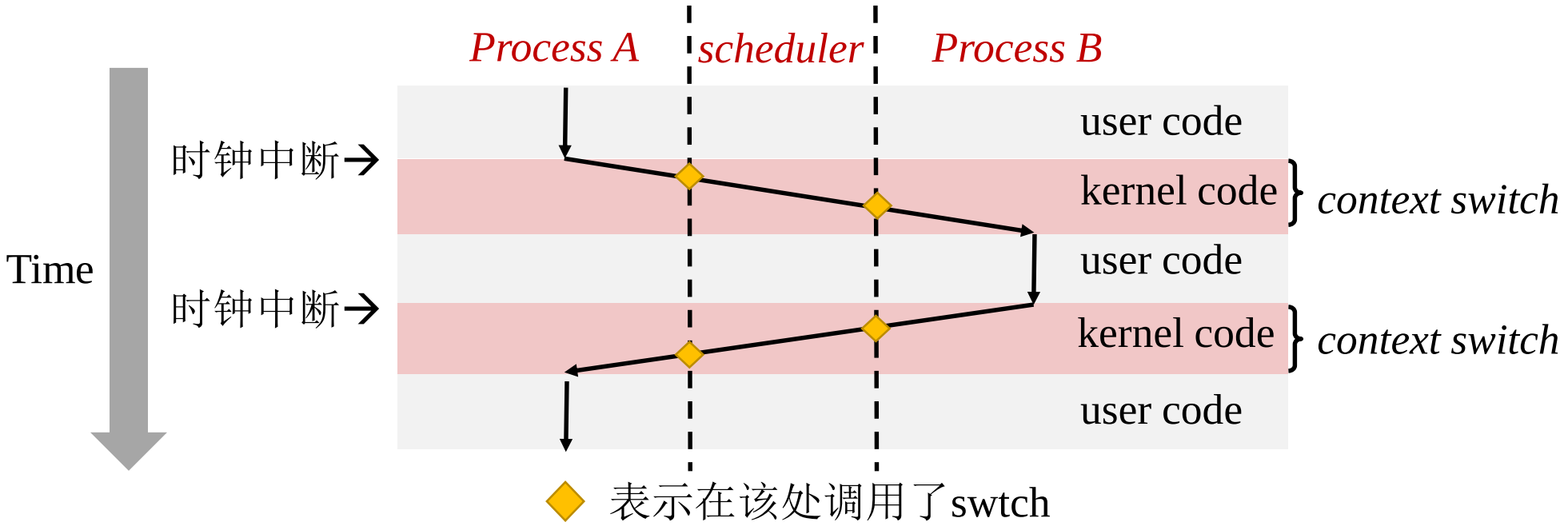
```
swtch(&c->context, &p->context);
```

(proc.c:476 scheduler函数内部)

# 进程调度:上下文切换



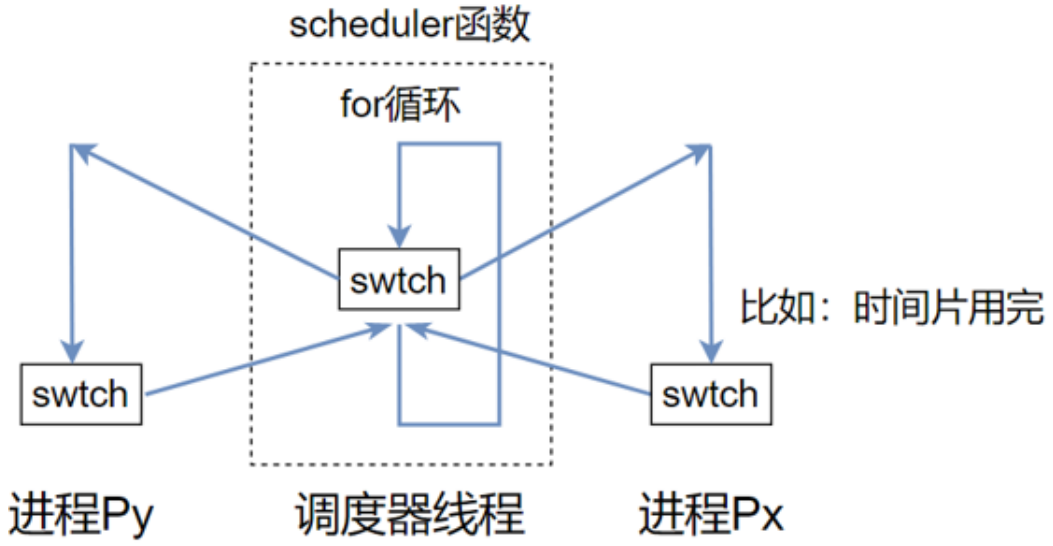
- 上下文切换
  - 控制流的切换通过上下文切换进行
  - 如图例所示：进程A时间片用完，被时钟中断打断，进入内核。在内核中，进程A通过swtch切换到调度器控制流，调度器通过swtch切换到进程B，完成了两个进程间的上下文切换。进程B切换到进程A同理。



# 进程调度：scheduler



- 进程调度该由谁执行：**scheduler thread（xv6 book的叫法）**
- OS启动后，每个CPU完成初始化后就进入scheduler函数，这个过程的控制流不属于任何进程。为了方便，可以称这些控制流属于**调度器线程**（scheduler thread），调度器线程进入scheduler后就一直在for循环里面不断进行进程调度。
- **如前文所述，当进行进程调度的时候（比如时间片用完），要先由旧进程切换到scheduler线程上下文，由scheduler线程执行进程调度算法找到新进程，再切换到新进程。**



# 进程调度



- 使用全局进程表查找可用进程
  - struct proc proc[NPROC]; （kernel/proc.c:14）类似单队列调度
  - 通过简单的单队列管理，天然地实现负载均衡
- scheduler函数（kernel/proc.c:445）负责进程调度
  - 从当前进程开始，循环遍历进程表，第一个找到的可执行进程就是将要调度的进程
  - 完全不考虑亲和性（让同一个进程在同一个CPU内跑）
  - 进程退出或让出CPU（调用yield函数）后会进入scheduler
- **使用时间片轮转调度**
  - 时间片约0.1秒，即每0.1秒触发时钟中断（kernel/start.c:63）
  - 时钟中断强制当前用户进程调用yield函数，让出CPU（kernel/trap.c:156）
  - yield将进行上下文切换，并回到调度函数scheduler（kernel/proc.c:524）

# 进程调度：调度算法

- xv6的进程调度算法在scheduler函数中：从**当前进程开始，循环遍历进程表，第一个找到的可执行进程就是将要调度的进程。**所有的进程都具有相同的优先级。

kernel/proc.c

```
456  void
457  scheduler(void)
458  {
459    struct proc *p;
460    struct cpu *c = mycpu();
461
462    c->proc = 0;
463    for(;;){
464      // Avoid deadlock by ensuring that devices can interrupt.
465      intr_on();
466
467      int found = 0;
468      for(p = proc; p < &proc[NPROC]; p++) {
469        acquire(&p->lock);
470        if(p->state == RUNNABLE) {
471          // Switch to chosen process.  It is the process's job
472          // to release its lock and then reacquire it
473          // before jumping back to us.
474          p->state = RUNNING;
475          c->proc = p;
476          swtch(&c->context, &p->context);
477
478          // Process is done running for now.
479          // It should have changed its p->state before coming back.
480          c->proc = 0;
481
482          found = 1;
483        }
484        release(&p->lock);
485      }
486      if(found == 0) {
487        intr_on();
488        asm volatile("wfi");
489      }
490    }
491  }
```

# Sleep and Wakeup



**进程在等待一个可能长时间都不会发生的事件的时候，可以让出CPU**，等待被其它进程唤醒。在xv6中，睡眠和唤醒是通过sleep（proc.c:555）和wakeup(proc.c:589)函数实现的。

- 睡眠：sleep(chan)即在chan上睡眠，chan被称为**等待通道(wait chan)**。chan的作用其实就是让唤醒进程知道应该唤醒哪些进程。sleep的第二个参数是一个自旋锁，在这里也被称为条件锁，其作用是避免“唤醒丢失”。

- 唤醒：wakeup(chan)，它会扫描进程表，找到在chan上睡眠的进程，通过将其状态修改为RUNNABLE来唤醒它。

```
void sleep(void *chan, struct spinlock *lk);

void wakeup(void *chan);
```

# 睡眠锁



- xv6基于sleep和wakeup函数实现了睡眠锁sleeplock，与自旋锁不同，如果尝试获取一个被占有的睡眠锁，进程会陷入睡眠，而不是像自旋锁那样“忙等待”。
- 睡眠锁其实就是一个二元信号量，通常用来保护一个会被长久占用的资源，比如bcache、inode等。
- acquiresleep将睡眠锁的内存地址作为唤醒通道chan，这样releasesleep中的wakeup就可以通过chan找到在对应睡眠锁上阻塞的进程。

```
21  void
22  acquiresleep(struct sleeplock *lk)
23  {
24    acquire(&lk->lk);
25    while (lk->locked) {
26      sleep(lk, &lk->lk);
27    }
28    lk->locked = 1;
29    lk->pid = myproc()->pid;
30    release(&lk->lk);
31  }
32
33  void
34  releasesleep(struct sleeplock *lk)
35  {
36    acquire(&lk->lk);
37    lk->locked = 0;
38    lk->pid = 0;
39    wakeup(lk);
40    release(&lk->lk);
41  }
```

kernel/sleeplock.c

# 总结



- xv6的进程管理模块的框架和算法非常简单。它不支持线程，仅有管道这种进程通信方式，而且简化了相关系统调用的接口，各种函数(fork, wait等)的实现也简单直接。
- 但是代码中也有不少难以理解的细节：比如sleep, wakeup, scheduler等函数中各种自旋锁的获取和释放到底有什么讲究，还有sched，swtch，scheduler是怎么配合来实现进程切换的 等等。思考清楚这些细节将对理解xv6乃至其它操作系统的进程管理模块和操作系统设计大有帮助。*（可以读相关代码和看xv6 book）。*