第44回 | 进程的阻塞与唤醒

Original 闪客 低并发编程 2022-07-17 17:30 Posted on 山东

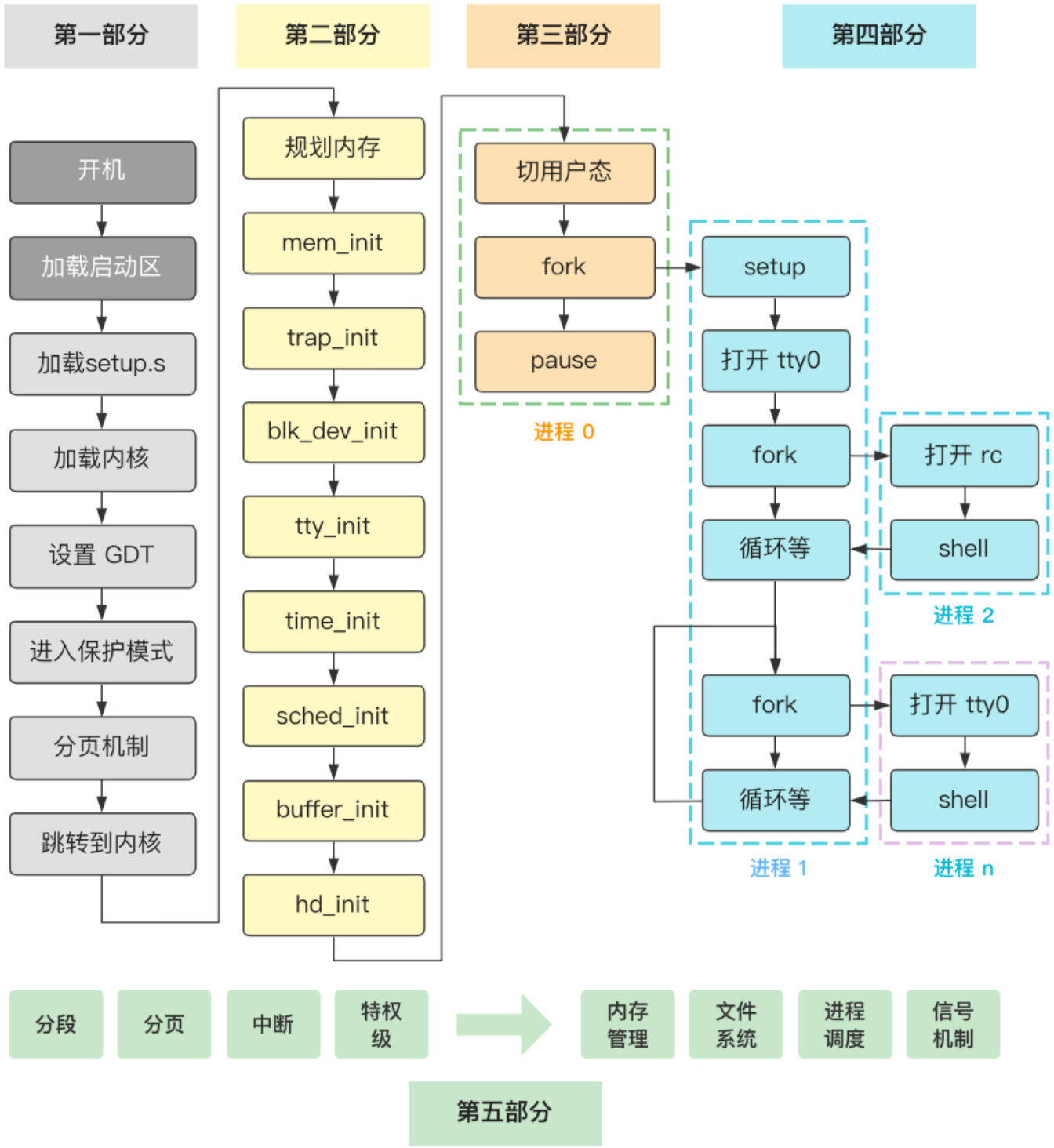
收录于合集
#操作系统源码 52 #一条shell命令的执行 8

新读者看这里，老读者直接跳过。

本系列会以一个读小说的心态，从开机启动后的代码执行顺序，带着大家阅读和赏析 Linux 0.11 全部核心代码，了解操作系统的技术细节和设计思想。

本系列的 GitHub 地址如下，希望给个 star 以示鼓励（文末**阅读原文**可直接跳转，也可以将下面的链接复制到浏览器里打开）
https://github.com/sunym1993/flash-linux0.11-talk

本回的内容属于第五部分。



你会跟着我一起，看着一个操作系统从啥都没有开始，一步一步最终实现它复杂又精巧的设计，读完这个系列后希望你能发出感叹，原来操作系统源码就是这破玩意。

以下是**已发布文章**的列表，详细了解本系列可以先从开篇词看起。

开篇词

第一部分 进入内核前的苦力活

------- 正文开始 -------

新建一个非常简单的 info.txt 文件。

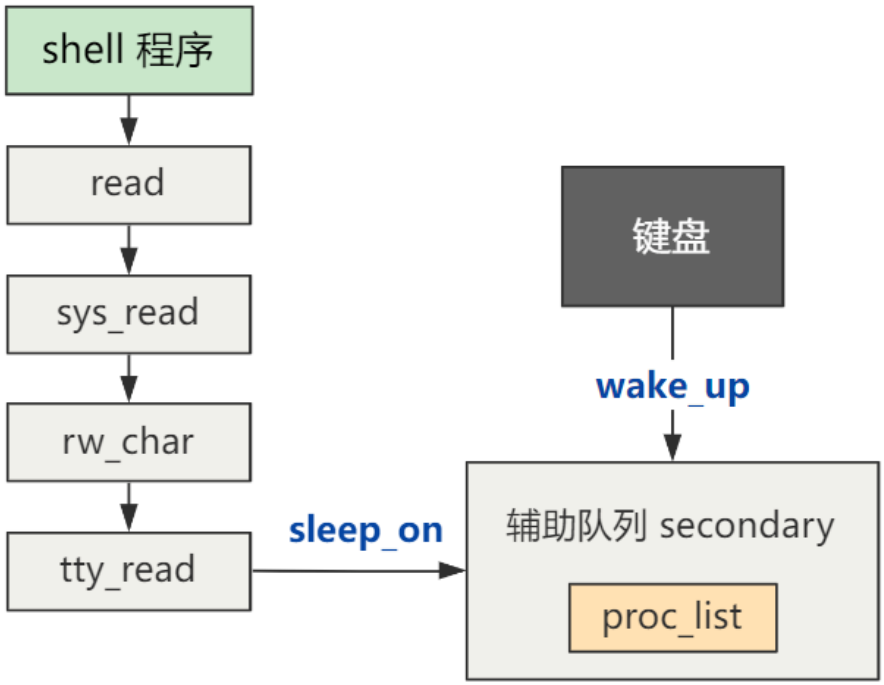
```
name:flash
age:28
language:java
```

在命令行输入一条十分简单的命令。

```
[root@linux0.11] cat info.txt | wc -l
3
```

这条命令的意思是读取刚刚的 info.txt 文件，输出它的行数。

在上一回中，我们分析了一下 shell 进程是如何读取你的命令的，流程如下图。



当然，这里的 **sleep_on** 和 **wake_up** 是进程的阻塞与唤醒机制的实现，我们没有展开讲解。

那我们今天，就详细看看这块的逻辑。

首先，表示进程的数据结构是 task_struct，其中有一个 **state** 字段表示进程的状态，它在 Linux 0.11 里有五种枚举值。

```
// shed.h
#define TASK_RUNNING 0      // 运行态
#define TASK_INTERRUPTIBLE 1    // 可中断等待状态。
#define TASK_UNINTERRUPTIBLE 2  // 不可中断等待状态
#define TASK_ZOMBIE 3       // 僵死状态
#define TASK_STOPPED 4      // 停止
```

当进程首次被创建时，也就是 fork 函数执行后，它的初始状态是 0，也就是运行态。

```
// system_call.s
_sys_fork:
    ...
    call _copy_process
    ...

// fork.c
int copy_process(...) {
    ...
    p->state = TASK_RUNNING;
    ...
}
```

只有当处于运行态的进程，才会被调度机制选中，送入 CPU 开始执行。

```
// sched.c
void schedule (void) {
    ...
    if ((*p)->state == TASK_RUNNING && (*p)->counter > c) {
        ...
        next = i;
    }
    ...
    switch_to (next);
}
```

以上我简单列出了关键代码，基本可以描绘进程调度的大体框架了，不熟悉的朋友还请回顾下第23回 | 如果让你来设计进程调度。

所以，使得一个进程阻塞的方法非常简单，并不需要什么魔法，只需要将其 **state** 字段，变成 **非 TASK_RUNNING** 也就是非运行态，即可让它暂时不被 CPU 调度，也就达到了阻塞的效果。

同样，唤醒也非常简单，就是再将对应进程的 state 字段变成 TASK_RUNNING 即可。

Linux 0.11 中的阻塞与唤醒，就是 sleep_on 和 wake_up 函数。

其中 sleep_on 函数将 state 变为 TASK_UNINTERRUPTIBLE。

```
// sched.c
void sleep_on (struct task_struct **p) {
    struct task_struct *tmp;
    ...
    tmp = *p;
    *p = current;
    current->state = TASK_UNINTERRUPTIBLE;
    schedule();
    if (tmp)
        tmp->state = 0;
}
```

而 wake_up 函数将 state 变回为 TASK_RUNNING，也就是 0。

```
// sched.c
void wake_up (struct task_struct **p) {
    (**p).state = 0;
}
```

是不是非常简单?

当然 sleep_on 函数除了改变 state 状态之外，还有些难理解的操作，我们先试着来分析一下。

当首次调用 sleep_on 函数时，比如 tty_read 在 secondary 队列为空时调用 sleep_on，传入的 *p 为 NULL，因为此时还没有等待 secondary 这个队列的任务。

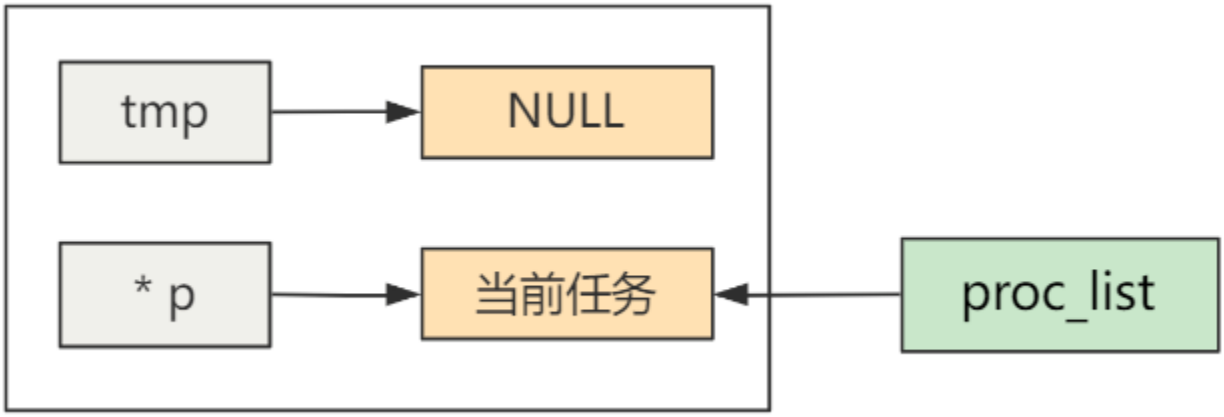
```
struct tty_queue {
    ...
    struct task_struct * proc_list;
};

struct tty_struct {
    ...
    struct tty_queue secondary;
};

int tty_read(unsigned channel, char * buf, int nr) {
    ...
    sleep_if_empty(&tty->secondary);
    ...
}

static void sleep_if_empty(struct tty_queue * queue) {
    ...
    interruptible_sleep_on(&queue->proc_list);
    ...
}
```

通过 **tmp = *p** 和 ***p = current** 两个赋值操作，此时：

tmp = NULL

*p = 当前任务

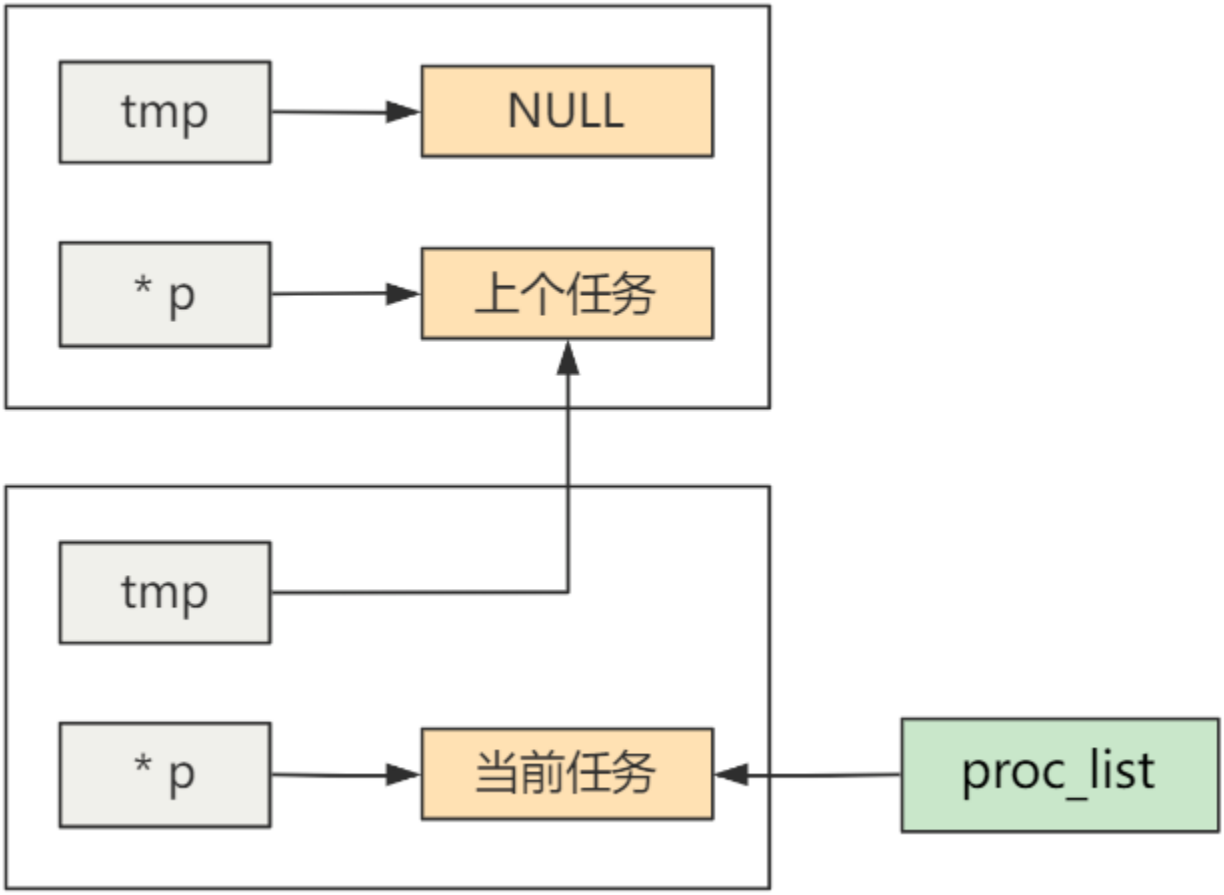
同时也使得 proc_list 指向了当前任务的 task_struct。



当有另一个进程调用了 tty_read 读取了同一个 tty 的数据时，就需要再次 sleep_on，此时携

带的 *p 就是一个指向了之前的"当前任务"的结构体。

那么经过 tmp = *p 和 *p = current 两个赋值操作后，会变成这个样子。



也就是说，通过每一个当前任务所在的代码块中的 tmp 变量，总能找到上一个正在同样等待一个资源的进程，因此也就形成了一个链表。

那么，当某进程调用了 wake_up 函数唤醒 proc_list 上指向的第一个任务时，改任务变会在 sleep_on 函数执行完 schedule() 后被唤醒并执行下面的代码，把 tmp 指针指向的上一个任务也同样唤醒。

```
// sched.c
void sleep_on (struct task_struct **p) {
    struct task_struct *tmp;
    ...
    tmp = *p;
    *p = current;
    current->state = TASK_UNINTERRUPTIBLE;
    schedule();
    if (tmp)
        tmp->state = 0;
}
```

永远记住，唤醒其实就是把 state 变成 0 而已。

而上一个进程唤醒后，和这个被唤醒的进程一样，也会走过它自己的 sleep_on 函数的后半段，把它的上一个进程，也就是上上一个进程唤醒。

那么上上一个进程，又会唤醒上上上一个进程，上上上一个进程，又会...

看懂了没，通过一个 wake_up 函数，以及上述这种 tmp 变量的巧妙设计，我们就能制造出唤醒的一连串连锁反应。

当然，唤醒后谁能优先抢到资源，那就得看调度的时机以及调度的机制了，对我们来说相当于听天由命了。

OK，现在我们的 shell 进程，通过 read 函数，中间经过了层层封装，以及后面经过了阻塞与唤醒这一番折腾后，终于把键盘输入的字符们，成功由 tty 中的 secondary 队列，读取并存放与 buf 指向的内存地址处。

```
[root@linux0.11] cat info.txt | wc -l
```

接下来，就该解析并执行这条命令了。

```
// xv6-public sh.c
int main(void) {
    static char buf[100];
    // 读取命令
    while(getcmd(buf, sizeof(buf)) >= 0){
        // 创建新进程
        if(fork() == 0)
            // 执行命令
            runcmd(parsecmd(buf));
        // 等待进程退出
        wait();
    }
}
```

也就是上述函数中的 **runcmd** 命令。

欲知后事如何，且听下回分解。

------- 关于本系列 -------

本系列的开篇词看这，开篇词

本系列的番外故事看这，让我们一起来写本书？也可以直接无脑加入星球，共同参与这场旅行。