一个新进程的诞生（五）通过 fork 看一次系统调用

Original 闪客 低并发编程 2022-02-27 16:30

收录于合集
#操作系统源码 43 #一个新进程的诞生 8

本系列作为 你管这破玩意叫操作系统源码 的第三大部分，讲述了操作系统第一个进程从无到有的诞生过程，这一部分你将看到内核态与用户态的转换、进程调度的上帝视角、系统调用的全链路、fork 函数的深度剖析。

不要听到这些陌生的名词就害怕，跟着我一点一点了解他们的全貌，你会发现，这些概念竟然如此活灵活现，如此顺其自然且合理地出现在操作系统的启动过程中。

本篇章作为一个全新的篇章，需要前置篇章的知识体系支撑。

第一部分 进入内核前的苦力活
第二部分 大战前期的初始化工作

当然，没读过的也问题不大，我都会在文章里做说明，如果你觉得有困惑，就去我告诉你的相应章节回顾就好了，放宽心。

------- 第三部分目录 -------

------- 正文开始 -------

书接上回，上回书咱们说到，我们通过自己设计了一遍进程调度，又看了一次 Linux 0.11 的进程调度的全过程。有了这两回做铺垫，我们下一回就该非常自信地回到我们的主流程！

```
void main(void) {
    ...
    move_to_user_mode();
    if (!fork()) {
        init();
    }
    for(;;) pause();
}
```

也就是这个 fork 函数干了啥？

这个 fork 函数稍稍绕了点，我们看如下代码。

```
static _inline _syscall0(int,fork)

#define _syscall0(type,name) \
type name(void) \
{ \
long __res; \
__asm__ volatile ("int $0x80" \
    : "=a" (__res) \
    : "0" (__NR_##name)); \
if (__res >= 0) \
    return (type) __res; \
errno = -__res; \
return -1; \
}
```

别急，我把它变成稍稍能看得懂的样子，就是这样。

```
#define _syscall0(type,name) \
type name(void) \
{ \
    volatile long __res; \
    _asm { \
        _asm mov eax,__NR_##name \
        _asm int 80h \
        _asm mov __res,eax \
    } \
    if (__res >= 0) \
        return (type) __res; \
    errno = -__res; \
    return -1; \
}
```

所以，把宏定义都展开，其实就相当于**定义了一个函数。**

```
int fork(void) {
     volatile long __res;
    _asm {
        _asm mov eax,__NR_fork
        _asm int 80h
        _asm mov __res,eax
    }
    if (__res >= 0)
        return (void) __res;
    errno = -__res;
    return -1;
}
```

仅此而已。

具体看一下 fork 函数里面的代码，又是讨厌的内联汇编，不过上面我已经变成好看一点的样子了，而且不用你看懂，听我说就行。

关键指令就是一个 0x80 号软中断的触发，**int 80h**。

其中还有一个 eax 寄存器里的参数是 **__NR_fork**，这也是个宏定义，值是 **2**。

OK，还记得 0x80 号中断的处理函数么？这个是我们在 第18回 | 大名鼎鼎的进程调度就是从这里开始的 **sched_init** 里面设置的。

```
set_system_gate(0x80, &system_call);
```

看这个 system_call 的汇编代码，我们发现这么一行。

```
_system_call:
    ...
    call [_sys_call_table + eax*4]
    ...
```

刚刚那个值就用上了，eax 寄存器里的值是 2，所以这个就是在这个 **sys_call_table** 表里找下标 2 位置处的函数，然后跳转过去。

那我们接着看 sys_call_table 是个啥。

```
fn_ptr sys_call_table[] = { sys_setup, sys_exit, sys_fork, sys_read,
  sys_write, sys_open, sys_close, sys_waitpid, sys_creat, sys_link,
  sys_unlink, sys_execve, sys_chdir, sys_time, sys_mknod, sys_chmod,
  sys_chown, sys_break, sys_stat, sys_lseek, sys_getpid, sys_mount,
  sys_umount, sys_setuid, sys_getuid, sys_stime, sys_ptrace, sys_alarm,
  sys_fstat, sys_pause, sys_utime, sys_stty, sys_gtty, sys_access,
  sys_nice, sys_ftime, sys_sync, sys_kill, sys_rename, sys_mkdir,
  sys_rmdir, sys_dup, sys_pipe, sys_times, sys_prof, sys_brk, sys_setgid,
  sys_getgid, sys_signal, sys_geteuid, sys_getegid, sys_acct, sys_phys,
  sys_lock, sys_ioctl, sys_fcntl, sys_mpx, sys_setpgid, sys_ulimit,
  sys_uname, sys_umask, sys_chroot, sys_ustat, sys_dup2, sys_getppid,
  sys_getpgrp, sys_setsid, sys_sigaction, sys_sgetmask, sys_ssetmask,
  sys_setreuid, sys_setregid
};
```

看到没，就是各种函数指针组成的一个数组，说白了就是个系统调用函数表。

那下标 2 位置处是啥？从第零项开始数，第二项就是 **sys_fork** 函数！

至此，我们终于找到了 fork 函数，通过系统调用这个中断，最终走到内核层面的函数是什么，就是 sys_fork。

```
_sys_fork:
    call _find_empty_process
    testl %eax,%eax
    js 1f
    push %gs
    pushl %esi
    pushl %edi
    pushl %ebp
    pushl %eax
    call _copy_process
    addl $20,%esp
1:  ret
```

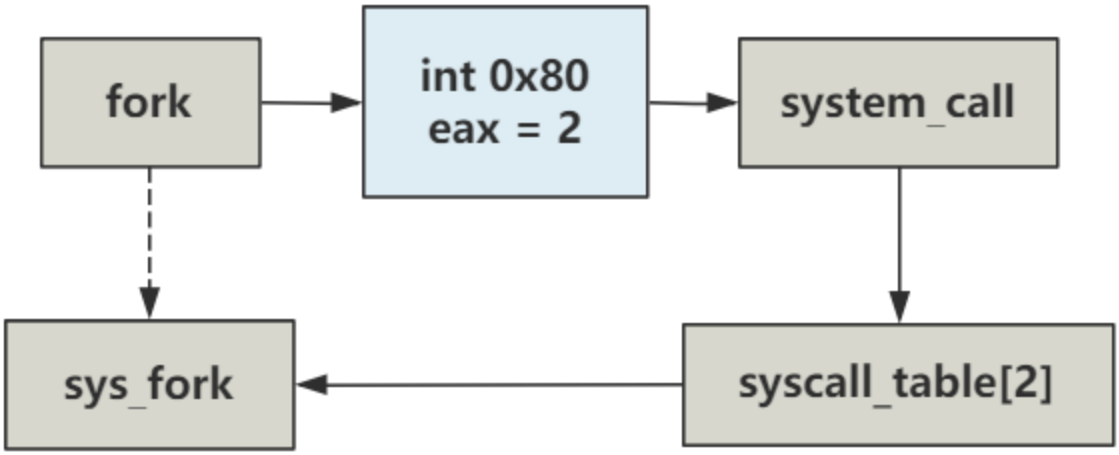
至于这个函数是什么，我们下一讲再说。

从这讲的探索我们也可以看出，操作系统通过**系统调用**，提供给用户态可用的功能，都暴露在 **sys_call_table** 里了。

系统调用统一通过 **int 0x80** 中断来进入，具体调用这个表里的哪个功能函数，就由 **eax** 寄存器传过来，这里的值是个数组索引的下标，通过这个下标就可以找到在 sys_call_table 这个数组里的具体函数。

同时也可以看出，用户进程调用内核的功能，可以直接通过写一句 int 0x80 汇编指令，并且给 eax 赋值，当然这样就比较麻烦。

所以也可以直接调用 fork 这样的包装好的方法，而这个方法里本质也是 int 0x80 以及 eax 赋值而已。



本讲就借着这个机会，讲讲系统调用的玩法，你学会了么？

那我们再多说两句，刚刚定义 fork 的系统调用模板函数时，用的是 **syscall0**，其实这个表示参数个数为 0，也就是 sys_fork 函数并不需要任何参数。

所以其实，在 unistd.h 头文件里，还定义了 syscall0 ~ syscall3 一共四个宏。

```
#define _syscall0(type,name)
#define _syscall1(type,name,atype,a)
#define _syscall2(type,name,atype,a,btype,b)
#define _syscall3(type,name,atype,a,btype,b,ctype,c)
```

看都能看出来，其实 **syscall1** 就表示有**一个参数**，**syscall2** 就表示有**两个参数**。

哎，就这么简单。

那这些参数放在哪里了呢？总得有个约定的地方吧？

我们看一个今后要讲的重点函数，**execve**，是一个通常和 fork 在一起配合的变身函数，在之后的进程 1 创建进程 2 的过程中，就是这样玩的。

```
void init(void) {
    ...
    if (!(pid=fork())) {
        ...
        execve("/bin/sh",argv_rc,envp_rc);
        ...
    }
}
```

当然我们的重点不是研究这个函数的作用，仅仅把它当做研究 syscall3 的一个例子，因为它的宏定义就是 **syscall3**。

```
execve("/bin/sh",argv_rc,envp_rc);

_syscall3(int,execve,const char *,file,char **,argv,char **,envp)

#define _syscall3(type,name,atype,a,btype,b,ctype,c) \
type name(atype a,btype b,ctype c) { \
    volatile long __res; \
    _asm { \
        _asm mov eax,__NR_##name \
        _asm mov ebx,a \
        _asm mov ecx,b \
        _asm mov edx,c \
        _asm int 80h \
        _asm mov __res,eax\
    } \
    if (__res >= 0) \
        return (type) __res; \
    errno = -__res; \
    return -1; \
}
```

可以看出，**参数 a 被放在了 ebx 寄存器，参数 b 被放在了 ecx 寄存器，参数 c 被放在了 edx 寄存器。**

我们再打开 system_call 的代码，刚刚我们只看了它的关键一行，就是去系统调用表里找函数。

```
_system_call:
    ...
    call [_sys_call_table + eax*4]
    ...
```

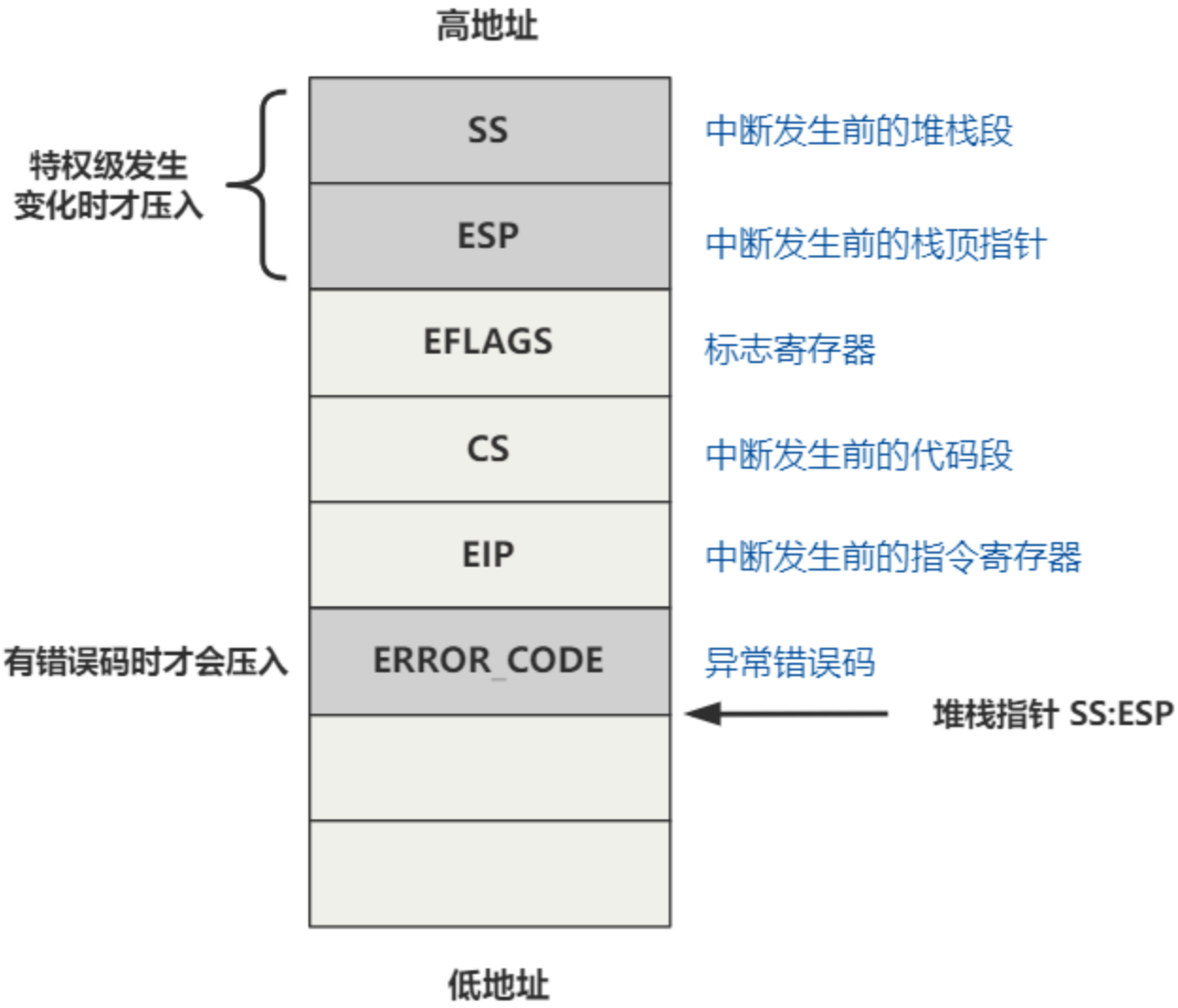
我们再看看全貌。

```
_system_call:
    cmpl $nr_system_calls-1,%eax
    ja bad_sys_call
    push %ds
    push %es
    push %fs
    pushl %edx
    pushl %ecx      # push %ebx,%ecx,%edx as parameters
    pushl %ebx      # to the system call
    movl $0x10,%edx     # set up ds,es to kernel space
    mov %dx,%ds
    mov %dx,%es
    movl $0x17,%edx     # fs points to local data space
    mov %dx,%fs
    call _sys_call_table(,%eax,4)
    pushl %eax
    movl _current,%eax
    cmpl $0,state(%eax)     # state
    jne reschedule
    cmpl $0,counter(%eax)       # counter
    je reschedule
ret_from_sys_call:
    movl _current,%eax      # task[0] cannot have signals
    cmpl _task,%eax
    je 3f
    cmpw $0x0f,CS(%esp)     # was old code segment supervisor ?
    jne 3f
    cmpw $0x17,OLDSS(%esp)      # was stack segment = 0x17 ?
    jne 3f
    movl signal(%eax),%ebx
    movl blocked(%eax),%ecx
    notl %ecx
    andl %ebx,%ecx
    bsfl %ecx,%ecx
    je 3f
    btrl %ecx,%ebx
    movl %ebx,signal(%eax)
    incl %ecx
    pushl %ecx
    call _do_signal
```

```
    call _do_signal
    popl %eax
3:  popl %eax
    popl %ebx
    popl %ecx
    popl %edx
    pop %fs
    pop %es
    pop %ds
    iret
```

又被吓到了是不是？

别怕，我们只关注压栈的情况，还记不记得在 一个新进程的诞生（二）从内核态到用户态 讲中，我们聊到触发了中断后，CPU 会自动帮我们做如下压栈操作。



因为 system_call 是通过 int 80h 这个软中断进来的，所以也属于中断的一种，具体说是属于特权级发生变化的，且没有错误码情况的中断，所以在这之前栈已经被压了 **SS、ESP、EFLAGS、CS、EIP** 这些值。

接下来 system_call 又压入了一些值，具体说来有 **ds、es、fs、edx、ecx、ebx、eax。**

如果你看源码费劲，得不出我上述结论，那你可以看 system_call.s 上面的注释，Linus 作者已经很贴心地给你写出了此时的堆栈状态。

```
/*
 * Stack layout in 'ret_from_system_call':
 *
 *   0(%esp) - %eax
 *   4(%esp) - %ebx
 *   8(%esp) - %ecx
 *   C(%esp) - %edx
 *  10(%esp) - %fs
 *  14(%esp) - %es
 *  18(%esp) - %ds
 *  1C(%esp) - %eip
 *  20(%esp) - %cs
 *  24(%esp) - %eflags
 *  28(%esp) - %oldesp
 *  2C(%esp) - %oldss
 */
```

看，就是 CPU 中断压入的 5 个值，加上 system_call 手动压入的 7 个值。

所以之后，中断处理程序如果有需要的话，就可以从这里取出它想要的值，包括 CPU 压入的那五个值，或者 system_call 手动压入的 7 个值。

比如 **sys_execve** 这个中断处理函数，一开始就取走了位于栈顶 0x1C 位置处的 EIP 的值。

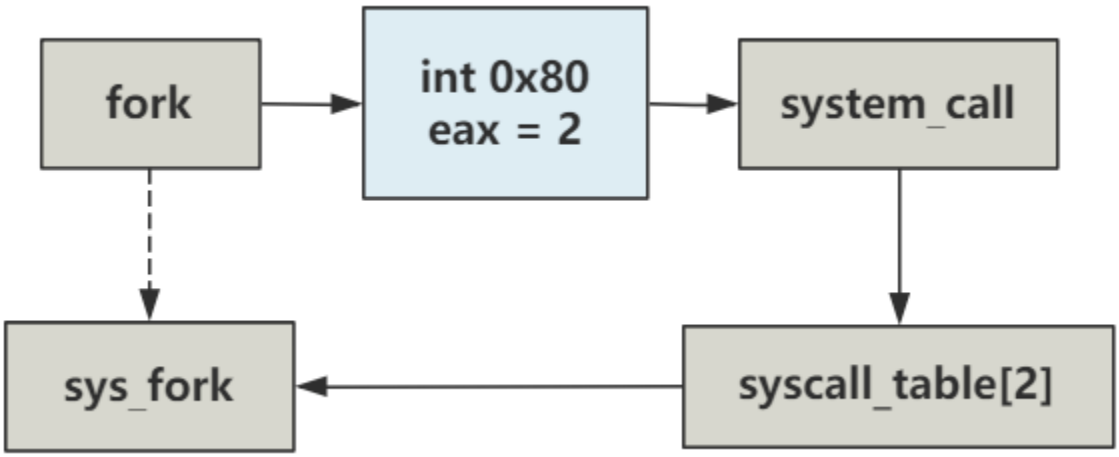
```
EIP = 0x1C
_sys_execve:
    lea EIP(%esp),%eax
    pushl %eax
    call _do_execve
    addl $4,%esp
    ret
```

随后在 **do_execve** 函数中，又通过 C 语言函数调用的约定，取走了 **filename，argv，envp** 等参数。

```
int do_execve(
        unsigned long * eip,
        long tmp,
        char * filename,
        char ** argv,
        char ** envp) {
    ...
}
```

具体这个函数的详细流程和作用，将会在第四部分的 shell 程序装载章节讲到。

今天你只需要记住**一次系统调用的流程和原理**，就可以了，把下图印在脑子里。



之后很多函数都会像今天的 fork 一样，走一遍系统调用的流程，到时候我就不再展开了。

所以前面的底子打得越好，后面你将会学得越爽。

欲知后事如何，且听下回分解。

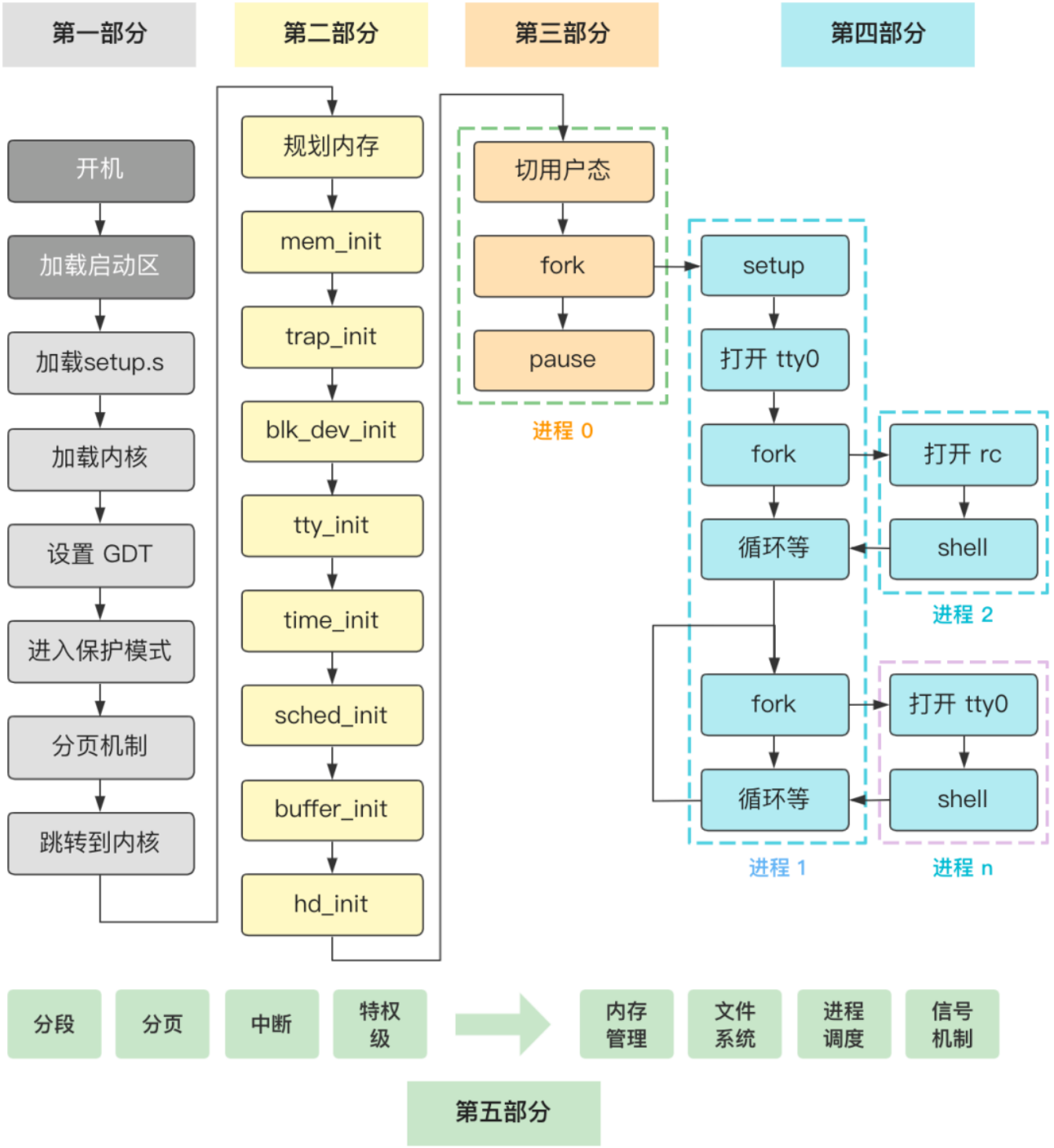
------- 关于本系列的完整内容 -------

本系列的开篇词看这
闪客新系列！你管这破玩意叫操作系统源码

本系列的扩展资料看这（也可点击**阅读原文**），这里有很多有趣的资料、答疑、互动参与项目，持续更新中，希望有你的参与。
https://github.com/sunym1993/flash-linux0.11-talk

本系列的番外故事看这
让我们一起来写本书？

本系列全局视角



最后，祝大家都能追更到系列结束，只要你敢持续追更，并且把每一回的内容搞懂，我就敢让你在系列结束后说一句，我对 Linux 0.11 很熟悉。

公众号更新系列文章不易，阅读量越来越低，希望大家多多传播，不方便的话点个小小的**在看**我也会很开心，我相信星火燎原的力量，谢谢大家咯。

另外，本系列**完全免费**，希望大家能多多传播给同样喜欢的人，同时给我的 GitHub 项目点个 star，就在**阅读原文**处，这些就足够让我坚持写下去了！我们下回见。

低并发编程
战略上藐视技术，战术上重视技术
175篇原创内容

Official Account

收录于合集 #操作系统源码 43

上一篇
让我们一起来写本书?

下一篇
一个新进程的诞生（六）fork 中进程基本信息的复制

Read more

People who liked this content also liked

LabVIEW操作MySQL数据库(4)-编程实例
虚拟仪器技术及应用

xenomai内核解析--双核系统调用(二)--应用如何区分xenomai/linux系统调用或服务
Linux阅码场

在Stata中调用Python的三种方式
经管学苑