



操作系统

第2章第3讲 系统调用

文艳军（教授）
计算机学院

回顾

一. 中断/异常的处理过程

栈的切换，态的切换，中断向量表，
中断处理程序，`iret`

二. 演示：除零异常的响应

`idiv`指令，`divide_error`

三. 独学&讨论

`Interrupt Procedures`，`LIDT`指令

系统调用

例:

- 创建文件 `fd=creat(name,···)`
- 打开文件 `fd=open(name,···)`
- 读文件 `n=read(fd,buffer,nbyte)*`

系统调用

```
move_to_user_mode();
```

```
if (!fork()) { /* we count on this going ok */  
    (void)mysignal(SIGALRM, SIG_IGN);
```

目录

- 一. 自陷指令
- 二. 系统调用的处理过程
- 三. 演示：fork系统调用
- 四. 操作系统的运行模式
- 五. 独学&讨论

一. 自陷指令 (trap指令)

- 引发一种特殊的异常，可用于完成系统调用功能

```
<bochs:14> u/2  
0000692b: (  
00006930: (  
<bochs:15> █
```

```
): mov eax, 0x00000002  
int 0x80
```

IA32中的陷入指令

```
move_to_user_mode();
```

```
if (!fork()) { /* we count on this going ok */  
    (void)mysignal(SIGALRM, SIG_IGN);
```

```
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; \  
}
```

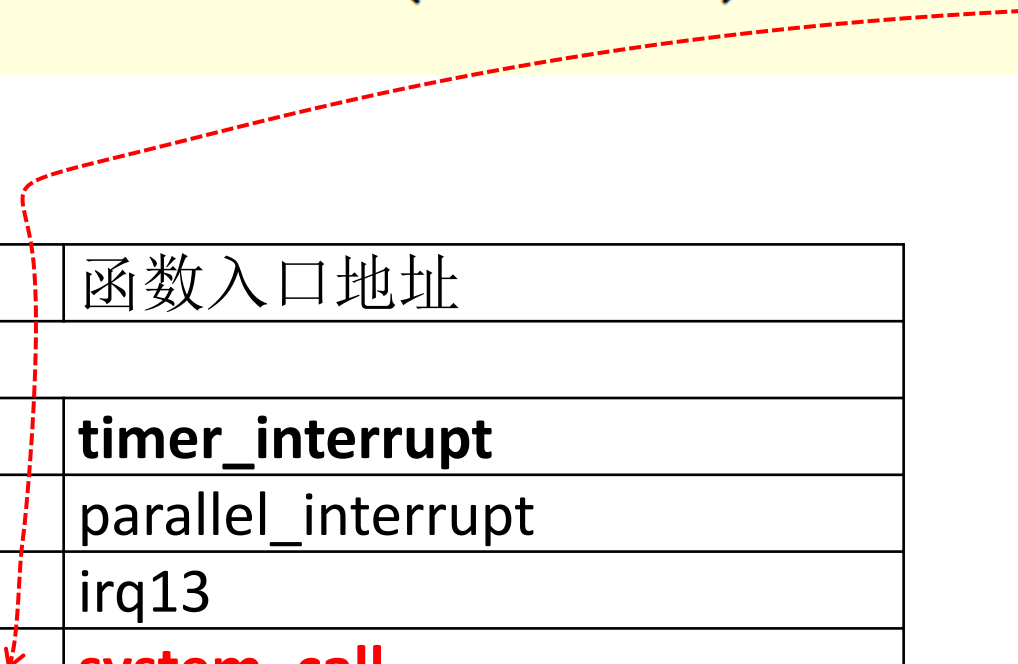
软中断指令

#define	__NR_setup	0
#define	__NR_exit	1
#define	__NR_fork	2
#define	__NR_read	3
#define	__NR_write	4
#define	__NR_open	5
#define	__NR_close	6

eax寄存器传递
参数和结果

```
(0) [0x0000000006930] 000f:00006930 (unk. ctxt): int 0x80
<bochs:16> █
```

位置	子位置	函数入口地址
[54c0, 5cc0)	idt	
	idt[0x20]	timer_interrupt
	idt[0x28]	parallel_interrupt
	idt[0x2d]	irq13
	idt[0x80]	system_call
	idt[0x81]	display_interrupt

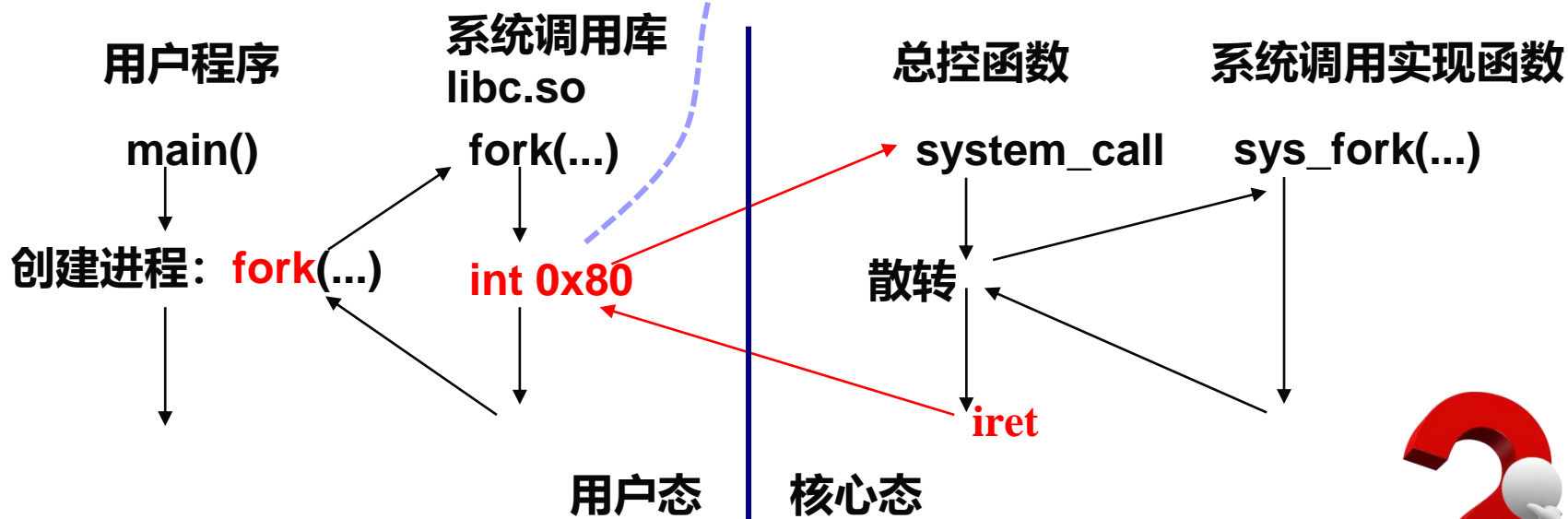


目录

- 一. 自陷指令
- 二. 系统调用的处理过程
- 三. 演示：fork系统调用
- 四. 操作系统的运行模式
- 五. 独学&讨论

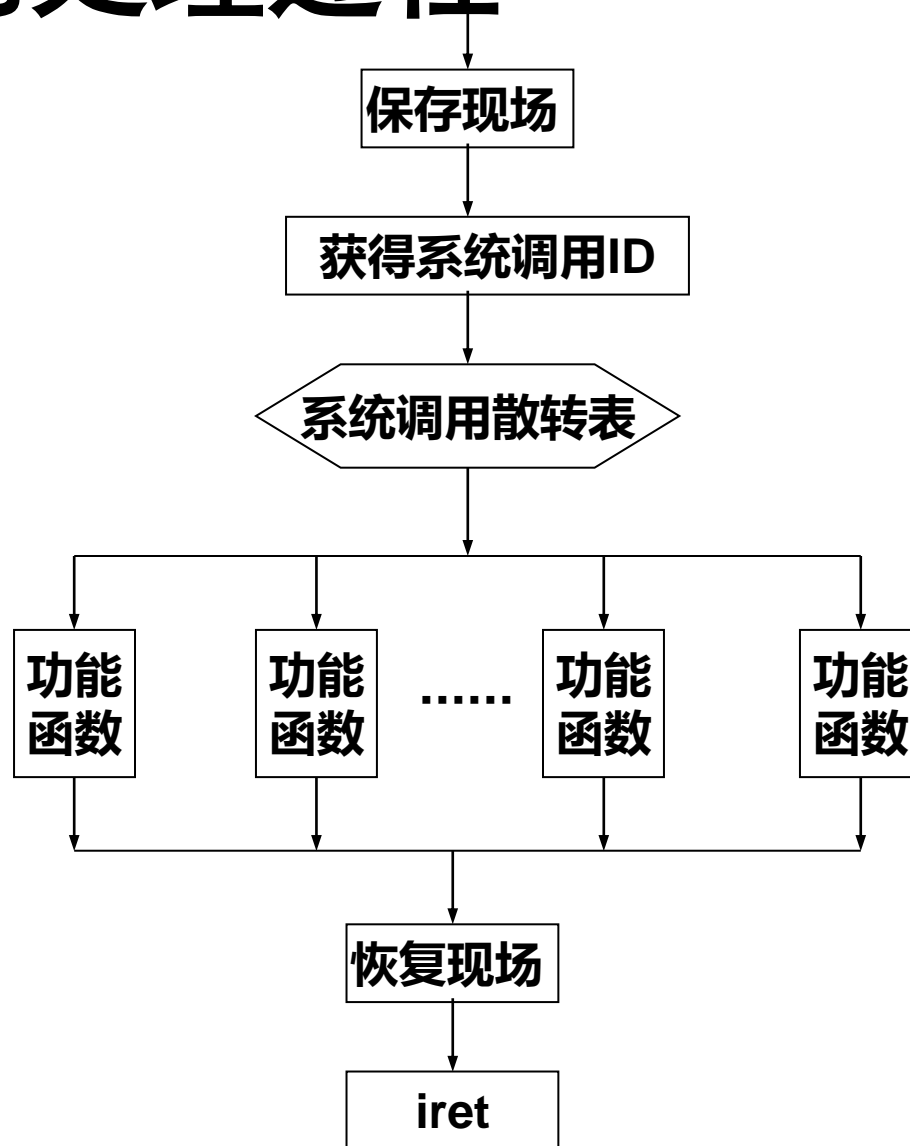
二. 系统调用的处理过程

IA32中的陷入指令，参数传递通过**约定的寄存器**



二. 系统调用的处理过程

- 系统调用总入口
处理程序



二. 系统调用的处理过程

系统调用ID:

```
_system_call:
    cmpl $nr_system_calls-1,%eax
    ja bad_sys_call
    push %ds
    push %es
    push %fs
    pushl %edx
    pushl %ecx          # push %ecx,%ecx,%edx as param
    pushl %ebx          # to 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)
```

系统调用散转表

0	sys_setup()
1	sys_exit()
2	sys_fork()
3	sys_read()
4	sys_write()
5	sys_open()
.....	

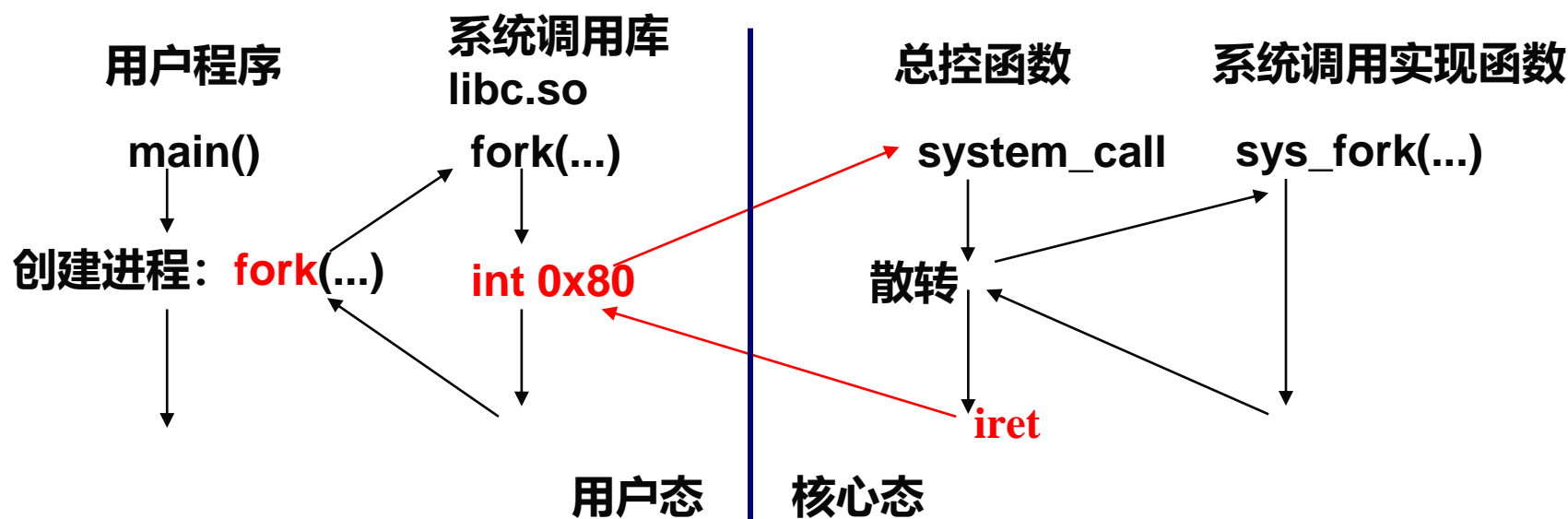
二. 系统调用的处理过程

fork系统调用函数
入口，索引为2

```
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, sys_sigsuspend, sys_sigpending, sys_sethostname,  
sys_setrlimit, sys_getrlimit, sys_getrusage, sys_gettimeofday,  
sys_settimeofday, sys_getgroups, sys_setgroups, sys_select, sys_symlink,  
sys_lstat, sys_readlink, sys_uselib };
```

二. 系统调用的处理过程

■ 思考：如何增加一个系统调用？



目录

- 一. 自陷指令
- 二. 系统调用的处理过程
- 三. 演示：fork系统调用
- 四. 操作系统的运行模式
- 五. 独学&讨论

三. 演示

■ 内容:

fork 系统调用

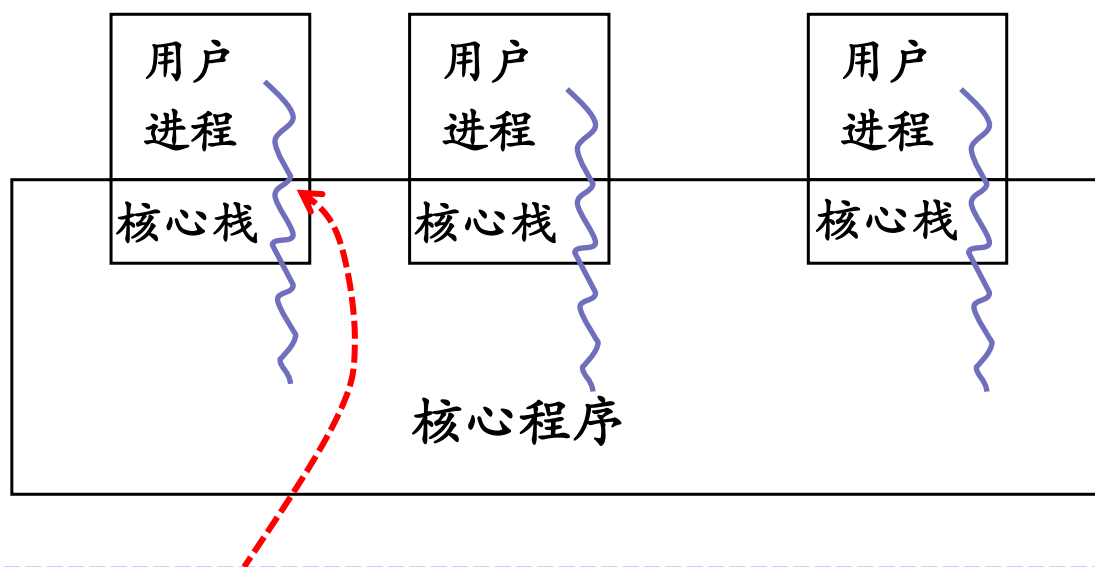
位置: `main.c:145, fork()`

目录

- 一. 自陷指令
- 二. 系统调用的处理过程
- 三. 演示：fork系统调用
- 四. 操作系统的运行模式
- 五. 独学&讨论

四. 操作系统运行模式

- ① 内核嵌入在用户进程中运行模式：
每个进程有核心栈(如Linux)



进入内核时 **只有态的切换**，没有进程的切换。

内核嵌入在用户进程中运行模式

进程1:

```
main() {  
    pause();  
}
```

`system_call()`

```
{  
    []sys_pause();  
}
```

进程2:

```
main() {  
    pause();  
}
```

`system_call()`

```
{  
    []sys_pause();  
}
```

□ □ □

用户态

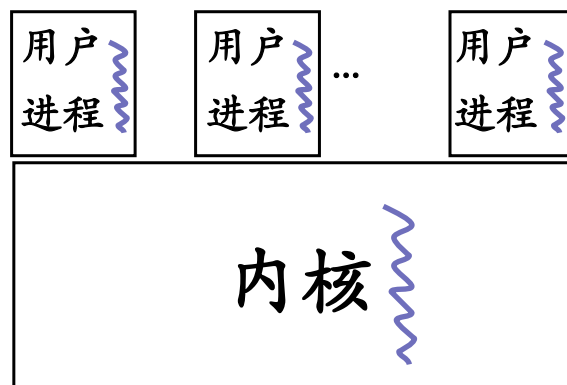
核心态

□ □ □

四. 操作系统的运行模式

② 独立内核模式：内核可看作一个特殊进程。（如seL4）

优点：便于保证
内核的正确性



缺点：内核的并发运行困难

独立内核模式

进程1:

```
main() {  
    pause{  
        发出请求;  
        等待结果;  
    }  
}
```

进程2:

```
main() {  
    pause{  
        发出请求;  
        等待结果;  
    }  
}
```

□ □ □

用户态

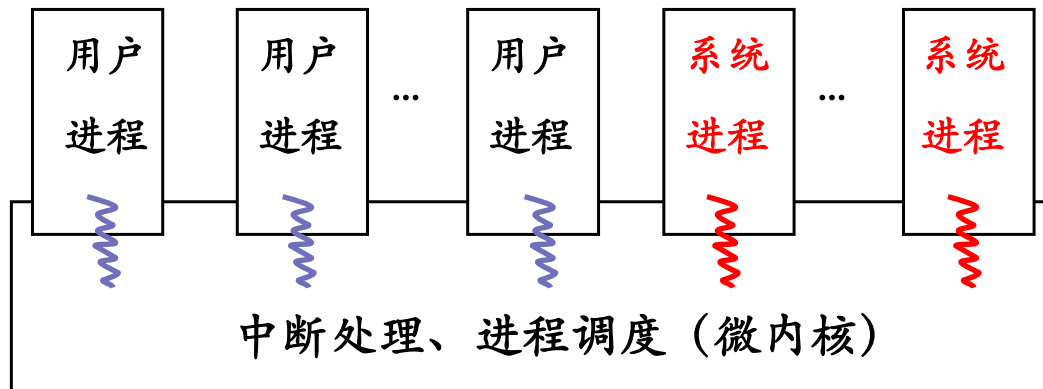
```
system_call() {  
    while(1) {  
        取一个请求;  
        处理请求{  
            []sys_pause();  
        }  
    }  
}
```

□ □ □

核心态

四. 操作系统运行模式

- ③ 微内核模式：内核的很多功能由用户态运行的**系统进程**实现(如Mach、Minix3)



缺点：开销大

优点：容错性好

独学&讨论

学习内核源码，回答下列问题：

- **idt**是在哪定义的？
- **IDTR**是什么时候设置的？
- **idt[0x20]**和**idt[0x80]**是什么时候设置的？

位置	子位置	函数入口地址
[54c0, 5cc0)	idt	
	idt[0x20]	timer_interrupt
	idt[0x28]	parallel_interrupt
	idt[0x2d]	irq13
	idt[0x80]	system_call
	idt[0x81]	display_interrupt

小结

一. 自陷指令： **int 0x80**

二. 系统调用的处理过程

总控函数、散转表、iret

三. 演示：fork系统调用

四. 操作系统的运行模式

独立内核，在用户进程中，微内核

五. 独学&讨论

作业

■ 课堂练习2.3的第2关

- ☐ 命令 `ls` 执行的系统调用

■ 课后作业2.3的第4~5关

- ☐ 增加系统调用 `getjiffies`

- ☐ 使用新系统调用

Linux 0.11

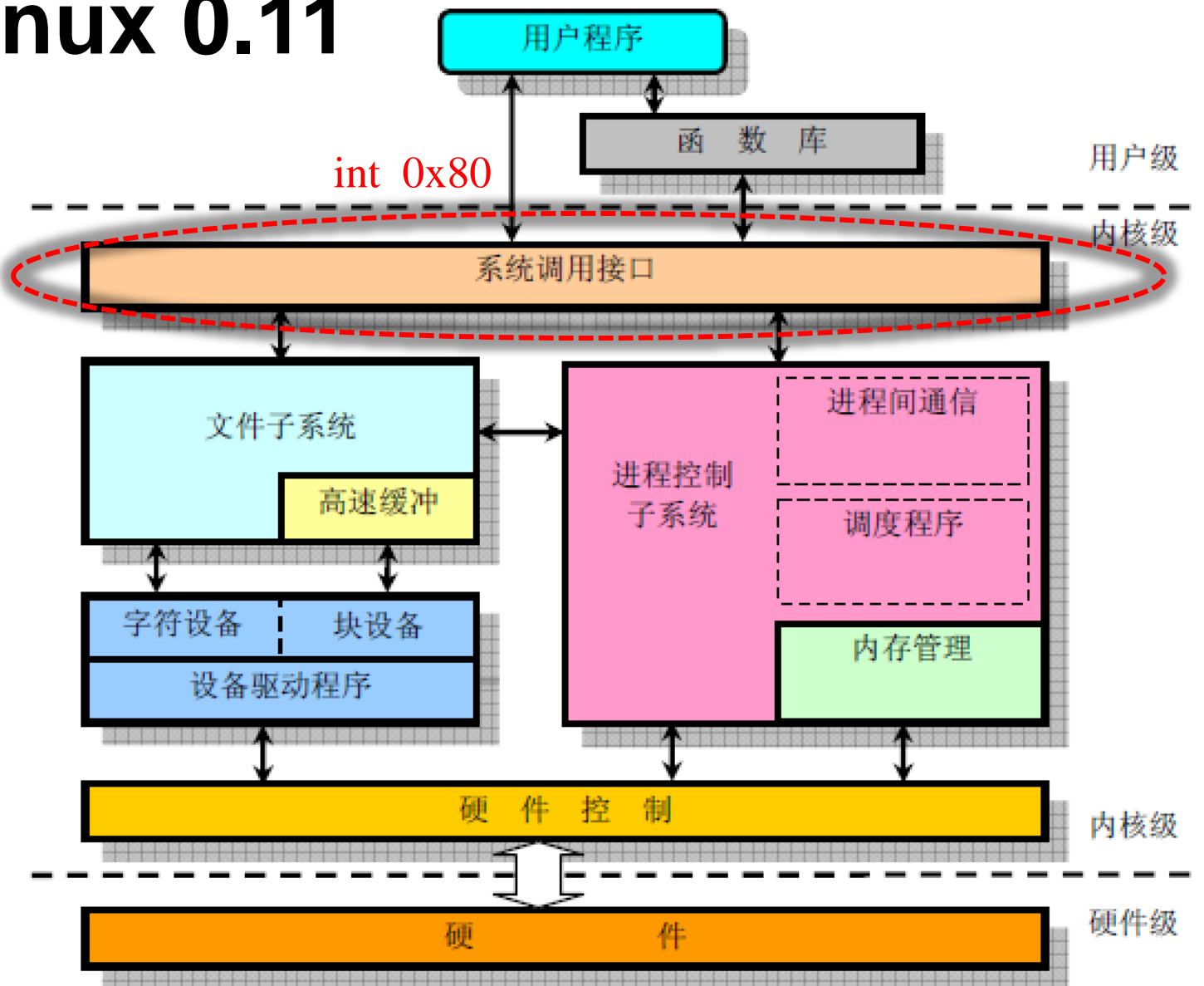


图 2-4 内核结构框图

```

sched_init();
buffer_init(buffer_memory_end);
hd_init();
floppy_init();
sti(); /* wyj */
move_to_user_mode();

```

```

if (!fork()) { ← /* we count on this going ok */
    (void)mysignal(SIGALRM, SIG_IGN);
    for(;;) {
        task1(); /* wyj */
        alarm(1);
        pause(); ←
    }
    /* init(); */
}
/*

```

NOTE!! For any other task 'pause()' would mean we have to get a signal to awaken, but task0 is the sole exception (see 'schedule()') as task 0 gets activated at every idle moment (when no other tasks can run). For task0 'pause()' just means we go check if some other task can run, and if not we return here.

```

for(;;) {
    task0(); /* wyj */
    pause();
}

```

} ? end main ?