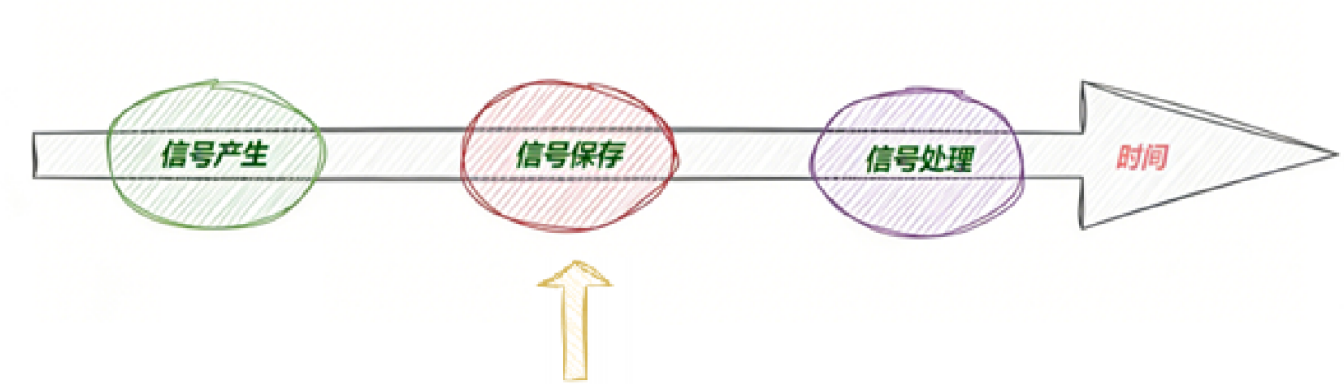
Linux进程信号（下）

1. 保存信号

当前阶段

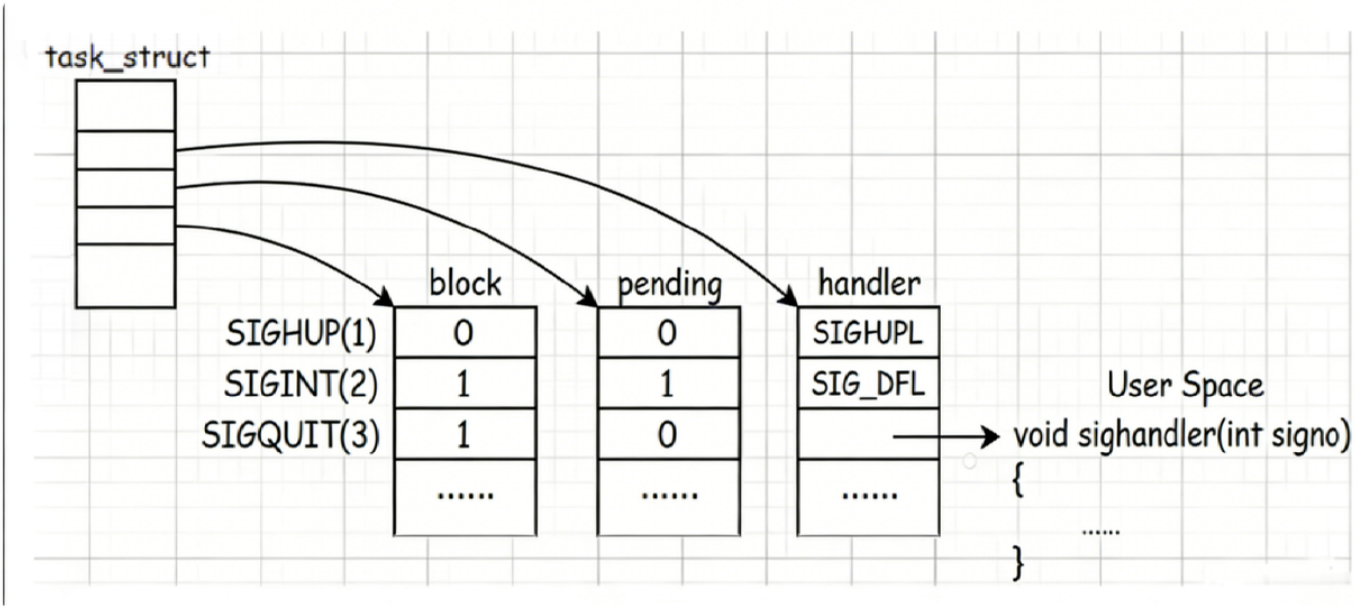


1.1 信号其他相关常见概念

> · 实际执行信号的处理动作称为信号递达(Delivery)
> · 信号从产生到递达之间的状态,称为信号未决(Pending)。
> · 进程可以选择阻塞 (Block)某个信号。
> · 被阻塞的信号产生时将保持在未决状态,直到进程解除对此信号的阻塞,才执行递达的动作.
> · 注意,阻塞和忽略是不同的,只要信号被阻塞就不会递达,而忽略是在递达之后可选的一种处理动作。

1.2 在内核中的表示

信号在内核中的表示意图



· 每个信号都有两个标志位分别表示阻塞(block)和未决(pending),还有一个函数指针表示处理动作。信号产生时,内核在进程控制块中设置该信号的未决标志,直到信号递达才清除该标志。在上图的例子中,SIGHUP信号未阻塞也未产生过,当它递达时执行默认处理动作。

· SIGINT信号产生过,但正在被阻塞,所以暂时不能递达。虽然它的处理动作是忽略,但在没有解除阻塞之前不能忽略这个信号,因为进程仍有机会改变处理动作之后再解除阻塞。

· SIGQUIT信号未产生过,一旦产生SIGQUIT信号将被阻塞,它的处理动作是用户自定义函数sighandler。

如果在进程解除对某信号的阻塞之前这种信号产生过多次,将如何处理?POSIX.1允许系统递送该信号一次或多次。Linux是这样实现的:常规信号在递达之前产生多次只计一次,而实时信号在递达之前产生多次可以依次放在一个队列里。本章不讨论实时信号。

代码块

```
// 内核结构 2.6.18
struct task_struct
{
    ...
    /* signal handlers */
    struct sighand_struct *sighand;
    sigset_t blocked
    struct sigpending pending;
    ...
}

struct sighand_struct
{
    atomic_t count;
    struct k_sigaction action[_NSIG]; // #define _NSIG 64
```

```
16      spinlock_t siglock;
17  };
18
19  struct __new_sigaction
20  {
21      __sighandler_t sa_handler;
22      unsigned long sa_flags;
23      void (*sa_restorer)(void); /* Not used by Linux/SPARC */
24      __new_sigset_t sa_mask;
25  };
26  struct k_sigaction
27  {
28      struct __new_sigaction sa;
29      void __user *ka_restorer;
30  };
31
32  /* Type of a signal handler. */
33  typedef void (*__sighandler_t)(int);
34
35  struct sigpending
36  {
37      struct list_head list;
38      sigset_t signal;
39  };
```

1.3 sigset_t

> 从上图来看,每个信号只有一个bit的未决标志, 非0即1, 不记录该信号产生了多少次,阻塞标志也是这样
>
> 表示的。因此, 未决和阻塞标志可以用相同的数据类型sigset_t来存储, , 这个类型
>
> 可以表示每个信号的“有效”或“无效”状态, 在阻塞信号集中“有效”和“无效”的含义是该信号
>
> 是否被阻塞, 而在未决信号集中“有效”和“无效”的含义是该信号是否处于未决状态。下一节将详
>
> 细介绍信号集的各种操作。阻塞信号集也叫做当前进程的 这里的“屏
>
> 蔽”应该理解为阻塞而不是忽略。

🏝️📚 · 类似权限哪里的 umask

1.4 信号集操作函数

sigset_t类型对于每种信号用一个bit表示“有效”或“无效”状态, 至于这个类型内部如何存储这些bit则依赖于系统实现, 从使用者的角度是不必关心的, 使用者只能调用以下函数来操作sigset_ t变量,而不

应该对它的内部数据做任何解释, 比如用printf直接打印sigset_t变量是没有意义的。

代码块

```
#include <signal.h>
int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
int sigaddset(sigset_t *set, int signo);
int sigdelset(sigset_t *set, int signo);
int sigismember(const sigset_t *set, int signo);
```

> · 函数sigemptyset初始化set所指向的信号集,使其中所有信号的对应bit清零,表示该信号集不包含任何有效信号。
>
> · 函数sigfillset初始化set所指向的信号集,使其中所有信号的对应bit置位,表示 该信号集的有效信号包括系统支持的所有信号。
>
> · 注意,在使用sigset_ t类型的变量之前,一定要调 用sigemptyset或sigfillset做初始化,使信号集处于确定的状态。初始化sigset_t变量之后就可以在调用sigaddset和sigdelset在该信号集中添加或删除某种有效信号。

这四个函数都是成功返回0,出错返回-1。sigismember是一个布尔函数,用于判断一个信号集的有效信号中是否包含 某种 信号,若包含则返回1,不包含则返回0,出错返回-1。

1.4.1 sigprocmask

调用函数 sigprocmask 可以读取或更改进程的信号屏蔽字(阻塞信号集)。

代码块

```
#include <signal.h>
int sigprocmask(int how, const sigset_t *set, sigset_t *oset);
返回值:若成功则为0,若出错则为-1
```

> 如果oset是非空指针,则读取进程的当前信号屏蔽字通过oset参数传出。如果set是非空指针,则 更改进程的信 号屏蔽字,参数how指示如何更改。如果oset和set都是非空指针,则先将原来的信号屏蔽字备份到oset里,然后 根据set和how参数更改信号屏蔽字。假设当前的信号屏蔽字为mask,下表说明了how参数的可选值。

SIG_BLOCK	set包含了我们希望添加到当前信号屏蔽字的信号，相当于mask=mask|set
SIG_UNBLOCK	set包含了我们希望从当前信号屏蔽字中解除阻塞的信号，相当于mask=mask&~set
SIG_SETMASK	设置当前信号屏蔽字为set所指向的值，相当于mask=set

如果调用sigprocmask解除了对当前若干个未决信号的阻塞,则在sigprocmask返回前,至少将其中一个信号递达。

1.4.2 sigpending

代码块

```
#include <signal.h>
int sigpending(sigset_t *set);

读取当前进程的未决信号集,通过set参数传出。
调用成功则返回0,出错则返回-1
```

下面用刚学的几个函数做个实验。程序如下:

代码块

```
#include <iostream>
#include <unistd.h>
#include <cstdio>
#include <sys/types.h>
#include <sys/wait.h>

void PrintPending(sigset_t &pending)
{
    std::cout << "curr process[" << getpid() << "]pending: ";
    for (int signo = 31; signo >= 1; signo--)
```

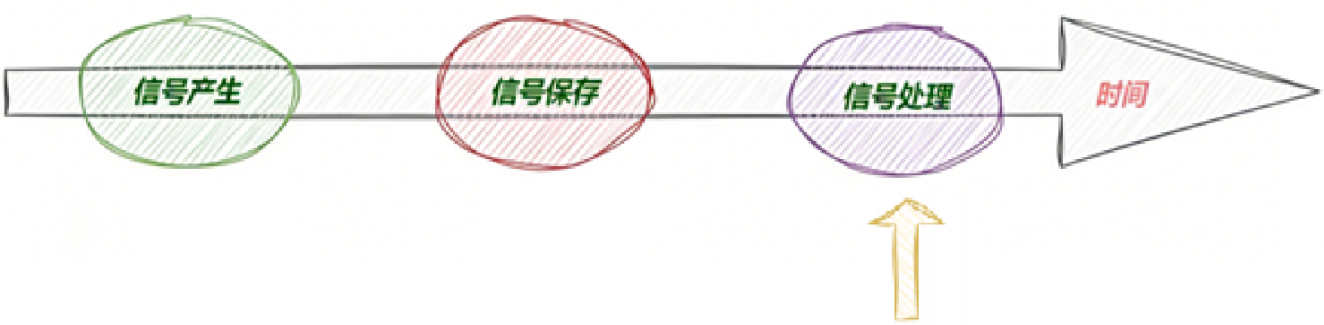
```
11      {
12          if (sigismember(&pending, signo))
13          {
14              std::cout << 1;
15          }
16          else
17          {
18              std::cout << 0;
19          }
20      }
21
22      std::cout << "\n";
23  }
24  void handler(int signo)
25  {
26      std::cout << signo << " 号信号被递达!!!" << std::endl;
27
28      std::cout << "------------------------------" << std::endl;
29      sigset_t pending;
30
31      sigpending(&pending);
32      PrintPending(pending);
33      std::cout << "------------------------------" << std::endl;
34  }
35  int main()
36  {
37      // 0. 捕捉2号信号
38      signal(2, handler); // 自定义捕捉
39      //signal(2, SIG_IGN); // 忽略一个信号
40      //signal(2, SIG_DFL); // 信号的默认处理动作
41
42      // 1. 屏蔽2号信号
43      sigset_t block_set, old_set;
44      sigemptyset(&block_set);
45      sigemptyset(&old_set);
46      sigaddset(&block_set, SIGINT); // 我们有没有修改当前进行的内核block表呢？？？
47
48      // 1.1 设置进入进程的Block表中
49      sigprocmask(SIG_BLOCK, &block_set, &old_set); // 真正的修改当前进行的内核
    block表，完成了对2号信号的屏蔽!
50      int cnt = 15;
51
52      while (true)
53      {
54          // 2. 获取当前进程的pending信号集
55          sigset_t pending;
56          sigpending(&pending);
```

```
57
58          // 3. 打印pending信号集
59          PrintPending(pending);
60          cnt--;
61
62          // 4. 解除对2号信号的屏蔽
63          if (cnt == 0)
64          {
65              std::cout << "解除对2号信号的屏蔽!!!" << std::endl;
66              sigprocmask(SIG_SETMASK, &old_set, &block_set);
67          }
68
69          sleep(1);
70      }
71  }
72  $ ./run
73  curr process[448336]pending: 00000000000000000000000000000000
74  curr process[448336]pending: 00000000000000000000000000000000
75  ^Ccurr process[448336]pending: 00000000000000000000000000000010
76  curr process[448336]pending: 00000000000000000000000000000010
77  curr process[448336]pending: 00000000000000000000000000000010
78  curr process[448336]pending: 00000000000000000000000000000010
79  curr process[448336]pending: 00000000000000000000000000000010
80  curr process[448336]pending: 00000000000000000000000000000010
81  curr process[448336]pending: 00000000000000000000000000000010
```

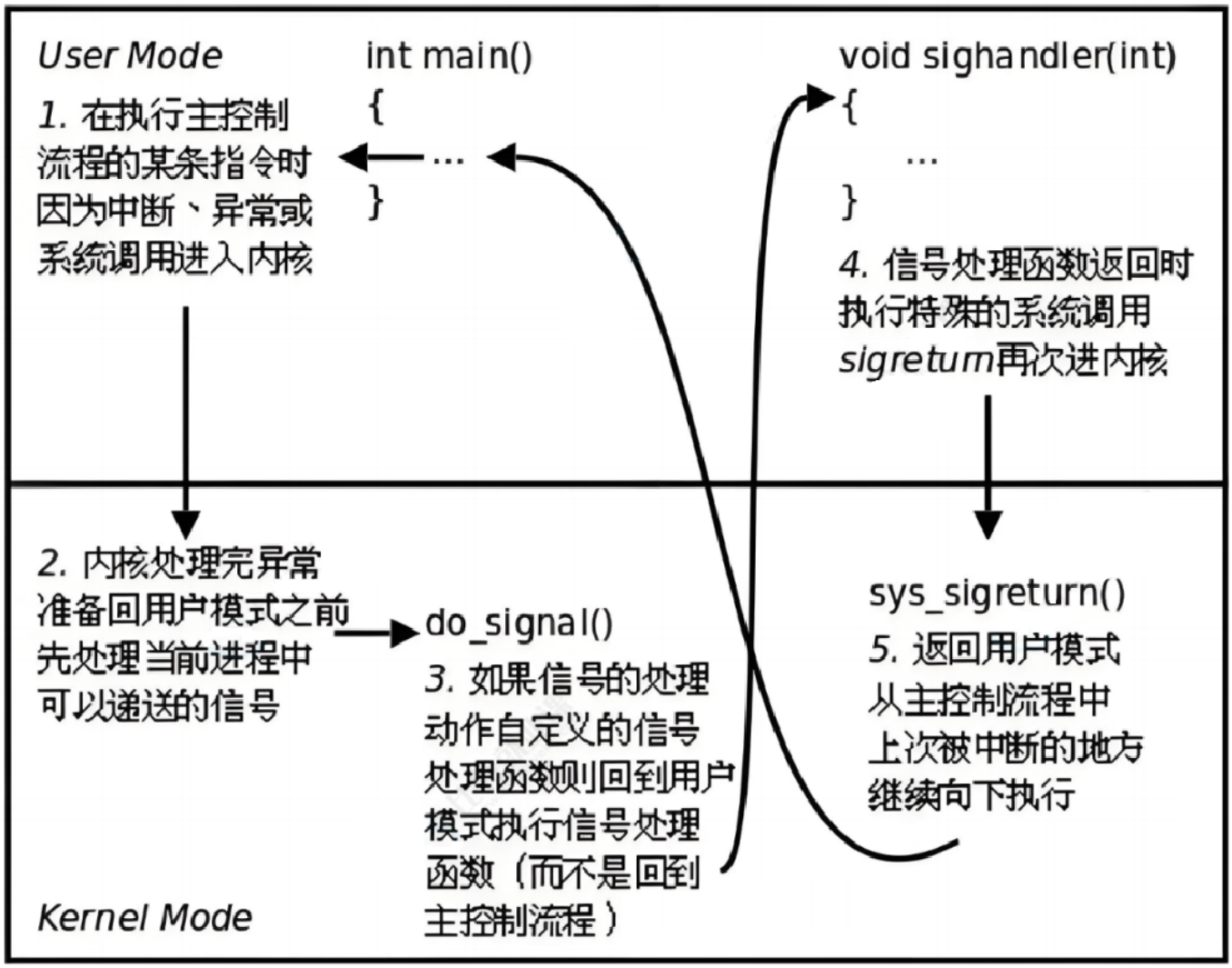
程序运行时,每秒钟把各信号的未决状态打印一遍,由于我们阻塞了SIGINT信号,按Ctrl-C将会 使SIGINT信号处于未决状态,按Ctrl-\仍然可以终止程序,因为SIGQUIT信号没有阻塞。

2. 捕捉信号

当前阶段



2.1 信号捕捉的流程



如果信号的处理动作是用户自定义函数,在信号递达时就调用这个函数,这称为捕捉信号。

由于信号处理函数的代码是在用户空间的,处理过程比较复杂,举例如下:

· 用户程序注册了 SIGQUIT 信号的处理函数 sighandler 。

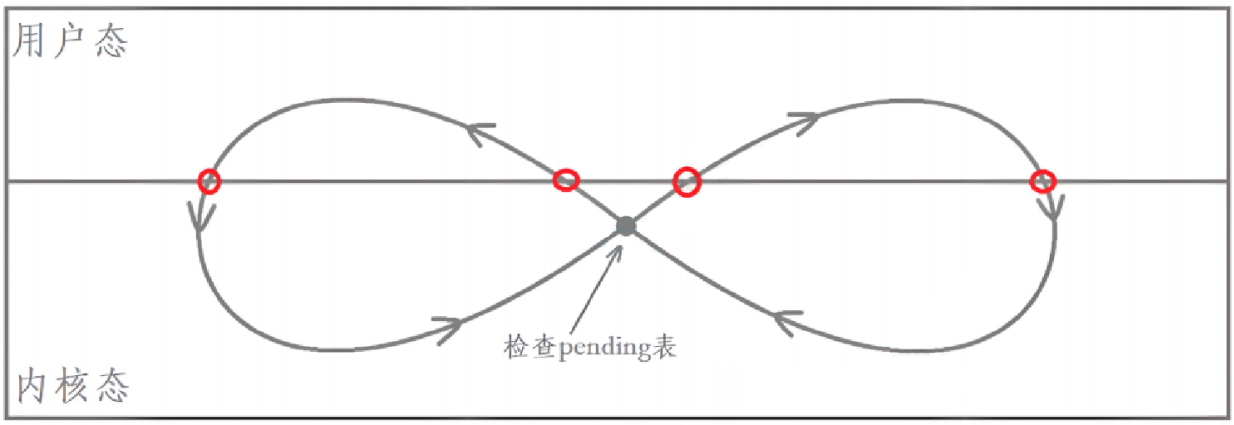
· 当前正在执行 main 函数,这时发生中断或异常切换到内核态。

· 在中断处理完毕后要返回用户态的 main 函数之前检查到有信号 SIGQUIT 递达。

· 内核决定返回用户态后不是恢复 main 函数的上下文继续执行,而是执行 sighandler 函数, sighandler 和 main 函数使用不同的堆栈空间,它们之间不存在调用和被调用的关系,是两个独立的控制流程。

· sighandler 函数返回后自动执行特殊的系统调用 sigreturn 再次进入内核态。

· 如果没有新的信号要递达,这次再返回用户态就是恢复 main 函数的上下文继续执行了。



2.2 sigaction

代码块

```
1   #include <signal.h>
2   int sigaction(int signo, const struct sigaction *act, struct sigaction *oact);
```

> · sigaction函数可以读取和修改与指定信号相关联的处理动作。调用成功则返回0,出错则返回- 1。signo是指定信号的编号。若act指针非空,则根据act修改该信号的处理动作。若oact指针非空, 则通过oact传出该信号原来的处理动作。act和oact指向sigaction结构体:
>
> · 将sa_handler赋值为常数SIG_IGN传给sigaction表示忽略信号,赋值为常数SIG_DFL表示执行系统默认动作,赋值为一个函数指针表示用自定义函数捕捉信号,或者说向内核注册了一个信号处理函数,该函数返回值为void,可以带一个int参数,通过参数可以得知当前信号的编号,这样就可以用同一个函数处理多种信号。显然,这也是一个回调函数,不是被main函数调用,而是被系统所调用。

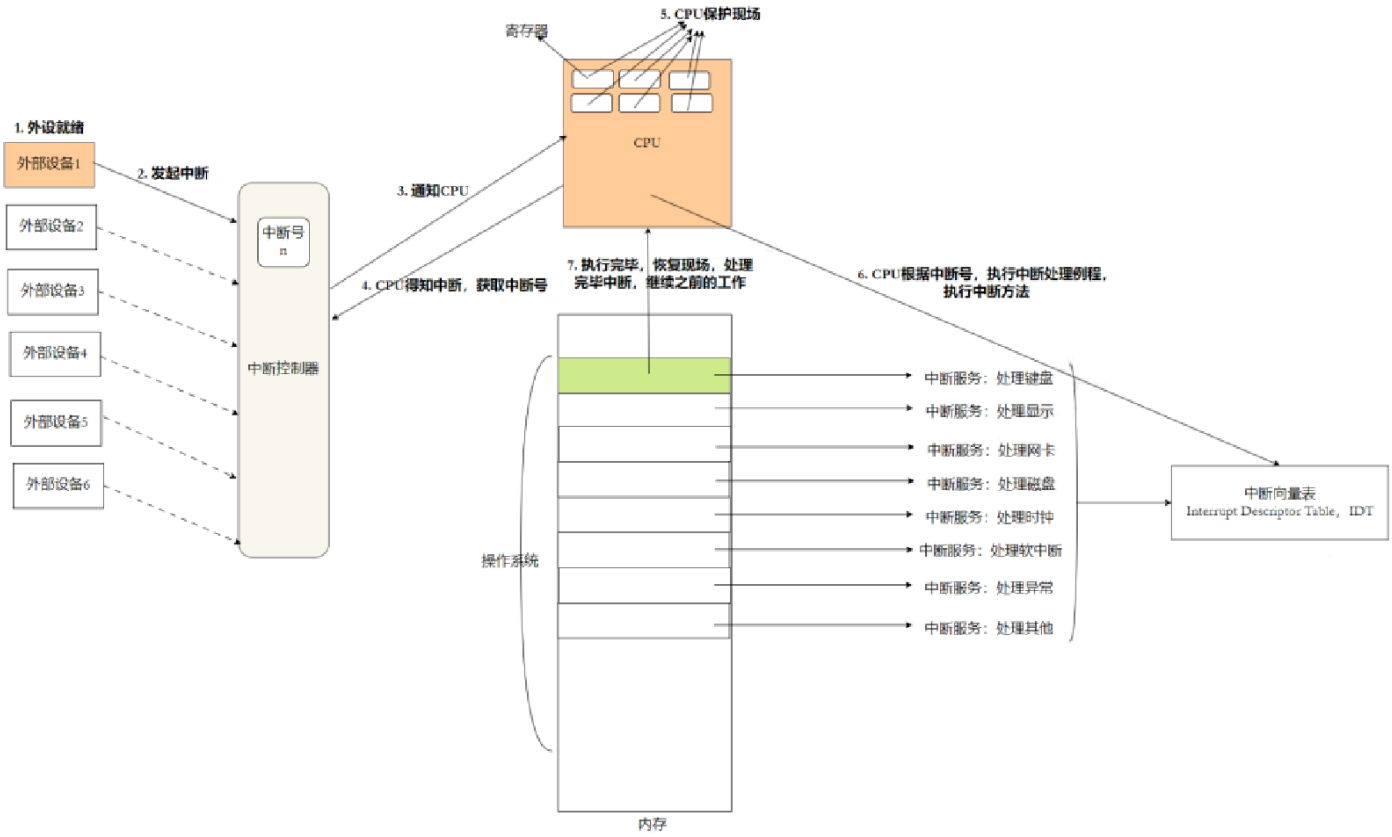
当某个信号的处理函数被调用时,内核自动将当前信号加入进程的信号屏蔽字,当信号处理函数返回时自动恢复原来的信号屏蔽字,这样就保证了在处理某个信号时,如果这种信号再次产生,那么 它会被阻塞到当前处理结束为止。 如果在调用信号处理函数时,除了当前信号被自动屏蔽之外,还希望自动屏蔽另外一

些信号,则用sa_mask字段说明这些需要额外屏蔽的信号,当信号处理函数返回时自动恢复原来的信号屏

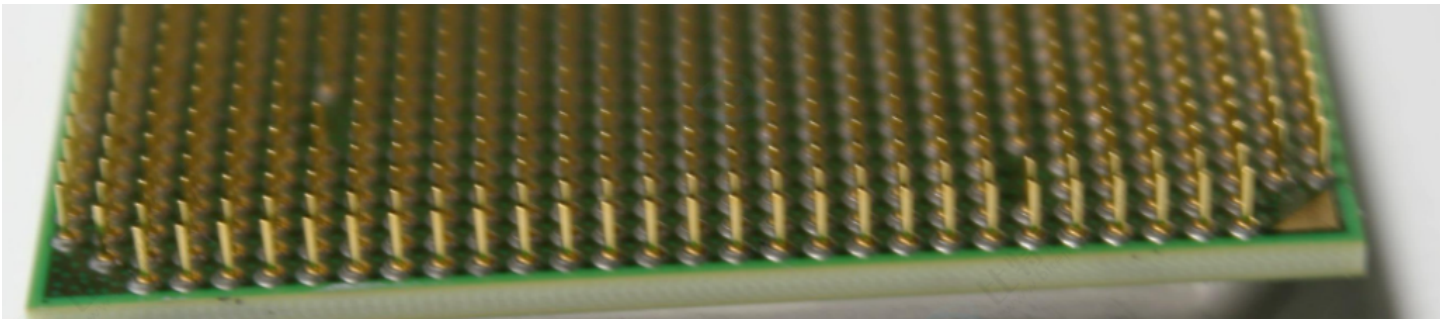
蔽字。 sa_flags字段包含一些选项,本章的代码都把sa_flags设为0,sa_sigaction是实时信号的处理函数,本章不详细解释这两个字段,有兴趣的同学可以在了解一下。

2.3 穿插话题 - 操作系统是怎么运行的

2.3.1 硬件中断



- 中断向量表就是操作系统的一部分，启动就加载到内存中了
- 通过外部硬件中断，操作系统就不需要对外设进行任何周期性的检测或者轮询
- 由外部设备触发的，中断系统运行流程，叫做硬件中断



代码块

```
//Linux内核0.11源码
void trap_init(void)
{
    int i;
    set_trap_gate(0,÷_error);// 设置除操作出错的中断向量值。以下雷同。
    set_trap_gate(1,&debug);
    set_trap_gate(2,&nmi);
    set_system_gate(3,&int3); /* int3-5 can be called from all */
    set_system_gate(4,&overflow);
    set_system_gate(5,&bounds);
    set_trap_gate(6,&invalid_op);
```

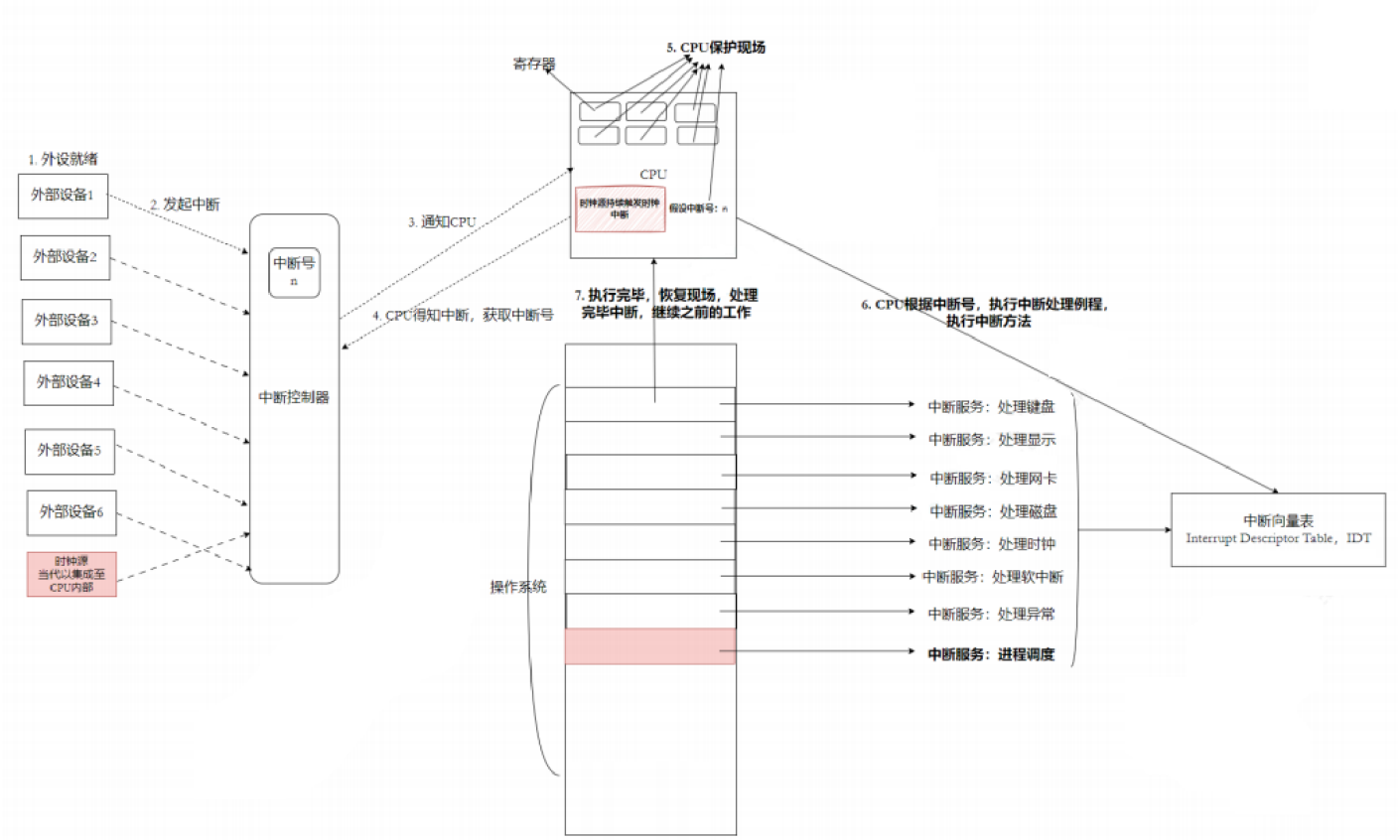
```
12      set_trap_gate(7,&device_not_available);
13      set_trap_gate(8,&double_fault);
14      set_trap_gate(9,&coprocessor_segment_overrun);
15      set_trap_gate(10,&invalid_TSS);
16      set_trap_gate(11,&segment_not_present);
17      set_trap_gate(12,&stack_segment);
18      set_trap_gate(13,&general_protection);
19      set_trap_gate(14,&page_fault);
20      set_trap_gate(15,&reserved);
21      set_trap_gate(16,&coprocessor_error);
22      // 下面将int17-48 的陷阱门先均设置为reserved，以后每个硬件初始化时会重新设置自己的
    陷阱门。
23      for (i=17;i<48;i++)
24      set_trap_gate(i,&reserved);
25      set_trap_gate(45,&irq13);// 设置协处理器的陷阱门。
26      outb_p(inb_p(0x21)&0xfb,0x21);// 允许主8259A 芯片的IRQ2 中断请求。
27      outb(inb_p(0xA1)&0xdf,0xA1);// 允许从8259A 芯片的IRQ13 中断请求。
28      set_trap_gate(39,¶llel_interrupt);// 设置并行口的陷阱门。
29  }
30
31  void rs_init (void)
32  {
33      set_intr_gate (0x24, rs1_interrupt); // 设置串行口1 的中断门向量(硬件IRQ4信
    号)。
34      set_intr_gate (0x23, rs2_interrupt); // 设置串行口2 的中断门向量(硬件IRQ3信
    号)。
35      init (tty_table[1].read_q.data); // 初始化串行口1(.data 是端口号)。
36      init (tty_table[2].read_q.data); // 初始化串行口2。
37      outb (inb_p (0x21) & 0xE7, 0x21); // 允许主8259A 芯片的IRQ3，IRQ4 中断信号请
    求。
38  }
```

2.3.2 时钟中断

问题：

· 进程可以在操作系统的指挥下，被调度，被执行，那么操作系统自己被谁指挥，被谁推动执行呢？

· 外部设备可以触发硬件中断，但是这个是需要用户或者设备自己触发，有没有自己可以定期触发的设备？



· 这样，操作系统不就在硬件的推动下，自动调度了么！！！

代码块

```
// Linux 内核0.11
// main.c
sched_init(); // 调度程序初始化(加载了任务0 的tr, ldtr) (kernel/sched.c)

// 调度程序的初始化子程序。
void sched_init(void)
{
    ...
    set_intr_gate(0x20, &timer_interrupt);
    // 修改中断控制器屏蔽码，允许时钟中断。
    outb(inb_p(0x21) & ~0x01, 0x21);
    // 设置系统调用中断门。
    set_system_gate(0x80, &system_call);
    ...
}

// system_call.s
_timer_interrupt:
        ...
;// do_timer(CPL)执行任务切换、计时等工作，在kernel/shched.c,305 行实现。
        call _do_timer ;// 'do_timer(long CPL)' does everything from

// 调度入口
```

```
24  void do_timer(long cpl)
25  {
26      ...
27      schedule();
28  }
29
30  void schedule(void)
31  {
32      ...
33      switch_to(next); // 切换到任务号为next 的任务，并运行之。
34  }
```

2.3.3 死循环

如果是这样，操作系统不就可以躺平了吗？对，操作系统自己不做任何事情，需要什么功能，就向中断向量表里面添加方法即可.操作系统的本质：就是一个死循环！

代码块

```
1   void main(void) /* 这里确实是void，并没错。 */
2   {                   /* 在startup 程序(head.s)中就是这样假设的。 */
3       ...
4       /*
5       * 注意!! 对于任何其它的任务，'pause()'将意味着我们必须等待收到一个信号才会返
6       * 回就绪运行态，但任务0（task0）是唯一的意外情况（参见'schedule()'），因为任
7       * 务0 在任何空闲时间里都会被激活（当没有其它任务在运行时），
8       * 因此对于任务0'pause()'仅意味着我们返回来查看是否有其它任务可以运行，如果没
9       * 有的话我们就回到这里，一直循环执行'pause()'。
10      */
11      for (;;)
12          pause();
13  } // end main
```

· 这样，操作系统，就可以在硬件时钟的推动下，自动调度了.

· 所以，什么是时间片？CPU为什么会有主频？为什么主频越快，CPU越快？主频可以作为OS调度执行速度的参考之一

2.3.4 软中断

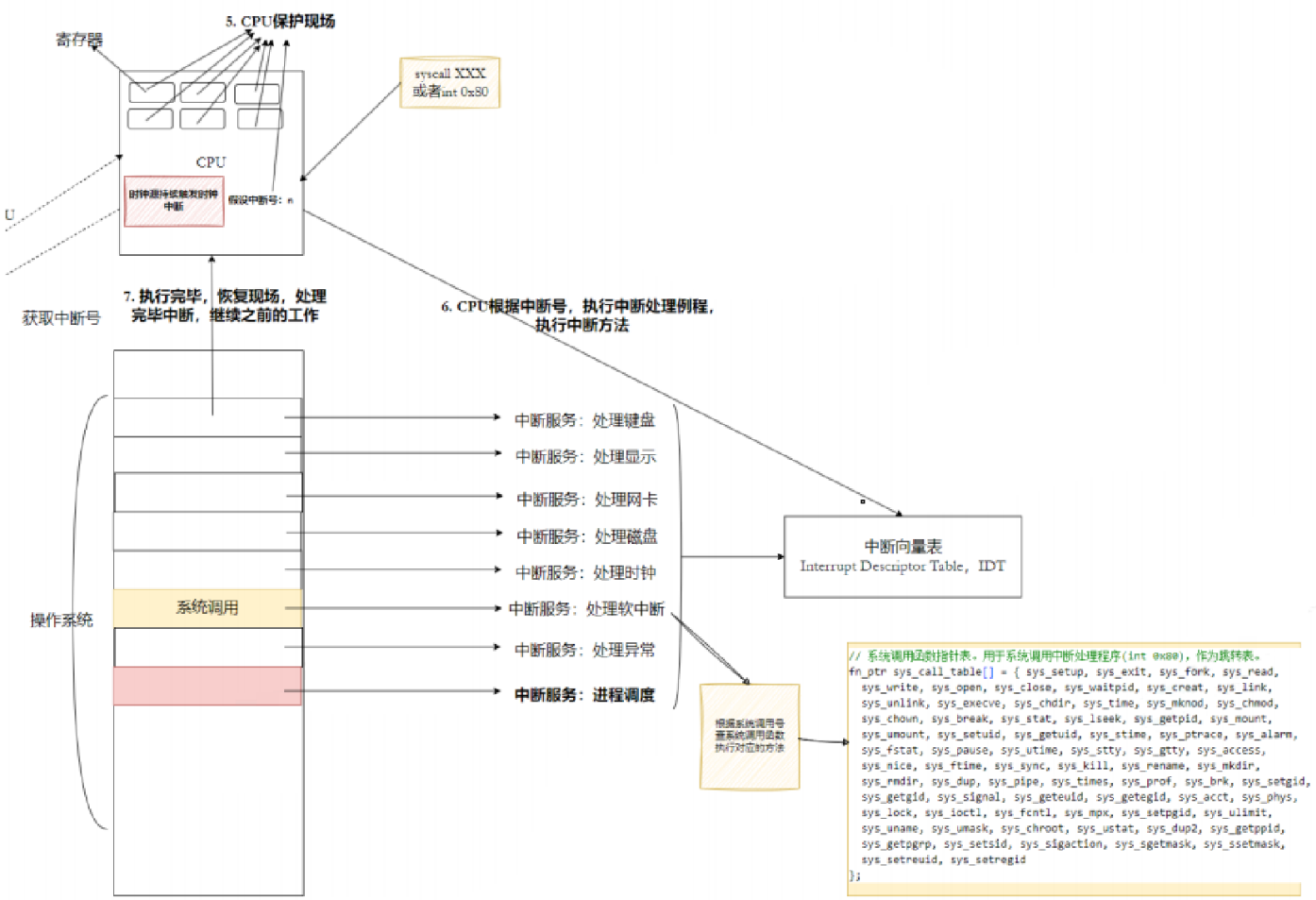
· 上述外部硬件中断，需要硬件设备触发。

· 有没有可能，因为软件原因，也触发上面的逻辑？有！

· 为了让操作系统支持进行系统调用，CPU也设计了对应的汇编指令(int 或者 syscall),可以让CPU内

部触发中断逻辑。

所以：



问题：

· 用户层怎么把系统调用号给操作系统？ - 寄存器(比如EAX)

· 操作系统怎么把返回值给用户？- 寄存器或者用户传入的缓冲区地址

· 系统调用的过程，其实就是先int 0x80、syscall陷入内核，本质就是触发软中断，CPU就会自动执行系统调用的处理方法，而这个方法会根据系统调用号，自动查表，执行对应的方法

· 系统调用号的本质：数组下标！

代码块

```
// sys.h
// 系统调用函数指针表。用于系统调用中断处理程序(int 0x80)，作为跳转表。
extern int sys_setup (); // 系统启动初始化设置函数。 (kernel/blk_drv/hd.c,71)
extern int sys_exit (); // 程序退出。 (kernel/exit.c, 137)
extern int sys_fork (); // 创建进程。 (kernel/system_call.s, 208)
extern int sys_read (); // 读文件。 (fs/read_write.c, 55)
extern int sys_write (); // 写文件。 (fs/read_write.c, 83)
extern int sys_open (); // 打开文件。 (fs/open.c, 138)
extern int sys_close (); // 关闭文件。 (fs/open.c, 192)
```

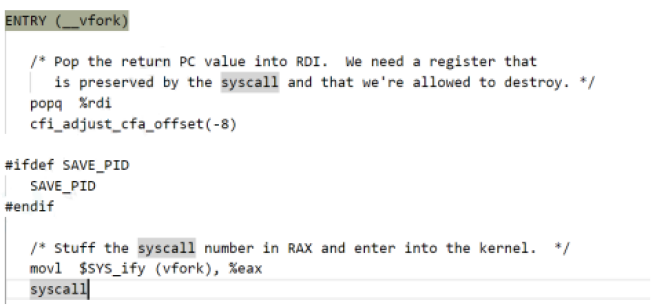
```
10  extern int sys_waitpid (); // 等待进程终止。 (kernel/exit.c, 142)
11  extern int sys_creat (); // 创建文件。 (fs/open.c, 187)
12  extern int sys_link (); // 创建一个文件的硬连接。 (fs/namei.c, 721)
13  extern int sys_unlink (); // 删除一个文件名(或删除文件)。 (fs/namei.c, 663)
14  extern int sys_execve (); // 执行程序。 (kernel/system_call.s, 200)
15  extern int sys_chdir (); // 更改当前目录。 (fs/open.c, 75)
16  extern int sys_time (); // 取当前时间。 (kernel/sys.c, 102)
17  extern int sys_mknod (); // 建立块/字符特殊文件。 (fs/namei.c, 412)
18  extern int sys_chmod (); // 修改文件属性。 (fs/open.c, 105)
19  extern int sys_chown (); // 修改文件宿主和所属组。 (fs/open.c, 121)
20  extern int sys_break (); // (-kernel/sys.c, 21)
21  extern int sys_stat (); // 使用路径名取文件的状态信息。 (fs/stat.c, 36)
22  extern int sys_lseek (); // 重新定位读/写文件偏移。 (fs/read_write.c, 25)
23  extern int sys_getpid (); // 取进程id。 (kernel/sched.c, 348)
24  extern int sys_mount (); // 安装文件系统。 (fs/super.c, 200)
25  extern int sys_umount (); // 卸载文件系统。 (fs/super.c, 167)
26  extern int sys_setuid (); // 设置进程用户id。 (kernel/sys.c, 143)
27  extern int sys_getuid (); // 取进程用户id。 (kernel/sched.c, 358)
28  extern int sys_stime (); // 设置系统时间日期。 (-kernel/sys.c, 148)
29  extern int sys_ptrace (); // 程序调试。 (-kernel/sys.c, 26)
30  extern int sys_alarm (); // 设置报警。 (kernel/sched.c, 338)
31  extern int sys_fstat (); // 使用文件句柄取文件的状态信息。(fs/stat.c, 47)
32  extern int sys_pause (); // 暂停进程运行。 (kernel/sched.c, 144)
33  extern int sys_utime (); // 改变文件的访问和修改时间。 (fs/open.c, 24)
34  extern int sys_stty (); // 修改终端行设置。 (-kernel/sys.c, 31)
35  extern int sys_gtty (); // 取终端行设置信息。 (-kernel/sys.c, 36)
36  extern int sys_access (); // 检查用户对一个文件的访问权限。(fs/open.c, 47)
37  extern int sys_nice (); // 设置进程执行优先权。 (kernel/sched.c, 378)
38  extern int sys_ftime (); // 取日期和时间。 (-kernel/sys.c,16)
39  extern int sys_sync (); // 同步高速缓冲与设备中数据。 (fs/buffer.c, 44)
40  extern int sys_kill (); // 终止一个进程。 (kernel/exit.c, 60)
41  extern int sys_rename (); // 更改文件名。 (-kernel/sys.c, 41)
42  extern int sys_mkdir (); // 创建目录。 (fs/namei.c, 463)
43  extern int sys_rmdir (); // 删除目录。 (fs/namei.c, 587)
44  extern int sys_dup (); // 复制文件句柄。 (fs/fcntl.c, 42)
45  extern int sys_pipe (); // 创建管道。 (fs/pipe.c, 71)
46  extern int sys_times (); // 取运行时间。 (kernel/sys.c, 156)
47  extern int sys_prof (); // 程序执行时间区域。 (-kernel/sys.c, 46)
48  extern int sys_brk (); // 修改数据段长度。 (kernel/sys.c, 168)
49  extern int sys_setgid (); // 设置进程组id。 (kernel/sys.c, 72)
50  extern int sys_getgid (); // 取进程组id。 (kernel/sched.c, 368)
51  extern int sys_signal (); // 信号处理。 (kernel/signal.c, 48)
52  extern int sys_geteuid (); // 取进程有效用户id。 (kenrl/sched.c, 363)
53  extern int sys_getegid (); // 取进程有效组id。 (kenrl/sched.c, 373)
54  extern int sys_acct (); // 进程记帐。 (-kernel/sys.c, 77)
55  extern int sys_phys (); // (-kernel/sys.c, 82)
56  extern int sys_lock (); // (-kernel/sys.c, 87)
```

```
57  extern int sys_ioctl (); // 设备控制。 (fs/ioctl.c, 30)
58  extern int sys_fcntl (); // 文件句柄操作。 (fs/fcntl.c, 47)
59  extern int sys_mpx (); // (-kernel/sys.c, 92)
60  extern int sys_setpgid (); // 设置进程组id。 (kernel/sys.c, 181)
61  extern int sys_ulimit (); // (-kernel/sys.c, 97)
62  extern int sys_uname (); // 显示系统信息。 (kernel/sys.c, 216)
63  extern int sys_umask (); // 取默认文件创建属性码。 (kernel/sys.c, 230)
64  extern int sys_chroot (); // 改变根系统。 (fs/open.c, 90)
65  extern int sys_ustat (); // 取文件系统信息。 (fs/open.c, 19)
66  extern int sys_dup2 (); // 复制文件句柄。 (fs/fcntl.c, 36)
67  extern int sys_getppid (); // 取父进程id。 (kernel/sched.c, 353)
68  extern int sys_getpgrp (); // 取进程组id，等于getpgid(0)。(kernel/sys.c, 201)
69  extern int sys_setsid (); // 在新会话中运行程序。 (kernel/sys.c, 206)
70  extern int sys_sigaction (); // 改变信号处理过程。 (kernel/signal.c, 63)
71  extern int sys_sgetmask (); // 取信号屏蔽码。 (kernel/signal.c, 15)
72  extern int sys_ssetmask (); // 设置信号屏蔽码。 (kernel/signal.c, 20)
73  extern int sys_setreuid (); // 设置真实与/或有效用户id。 (kernel/sys.c,118)
74  extern int sys_setregid (); // 设置真实与/或有效组id。 (kernel/sys.c, 51)
75  // 系统调用函数指针表。用于系统调用中断处理程序(int 0x80)，作为跳转表。
76  fn_ptr sys_call_table[] = { sys_setup, sys_exit, sys_fork, sys_read,
77  sys_write, sys_open, sys_close, sys_waitpid, sys_creat, sys_link,
78  sys_unlink, sys_execve, sys_chdir, sys_time, sys_mknod, sys_chmod,
79  sys_chown, sys_break, sys_stat, sys_lseek, sys_getpid, sys_mount,
80  sys_umount, sys_setuid, sys_getuid, sys_stime, sys_ptrace, sys_alarm,
81  sys_fstat, sys_pause, sys_utime, sys_stty, sys_gtty, sys_access,
82  sys_nice, sys_ftime, sys_sync, sys_kill, sys_rename, sys_mkdir,
83  sys_rmdir, sys_dup, sys_pipe, sys_times, sys_prof, sys_brk, sys_setgid,
84  sys_getgid, sys_signal, sys_geteuid, sys_getegid, sys_acct, sys_phys,
85  sys_lock, sys_ioctl, sys_fcntl, sys_mpx, sys_setpgid, sys_ulimit,
86  sys_uname, sys_umask, sys_chroot, sys_ustat, sys_dup2, sys_getppid,
87  sys_getpgrp, sys_setsid, sys_sigaction, sys_sgetmask, sys_ssetmask,
88  sys_setreuid, sys_setregid
89  };
90  // 调度程序的初始化子程序。
91  void sched_init(void)
92  {
93      ...
94      // 设置系统调用中断门。
95      set_system_gate(0x80, &system_call);
96  }
97
98  _system_call:
99          cmp eax,nr_system_calls-1 ;// 调用号如果超出范围的话就在eax 中置-1 并退出。
100         ja bad_sys_call
101         push ds ;// 保存原段寄存器值。
102         push es
103         push fs
```

```
104             push edx ;// ebx,ecx,edx 中放着系统调用相应的C 语言函数的调用参数。
105             push ecx ;// push %ebx,%ecx,%edx as parameters
106             push ebx ;// to the system call
107             mov edx,10h ;// set up ds,es to kernel space
108             mov ds,dx ;// ds,es 指向内核数据段(全局描述符表中数据段描述符)。
109             mov es,dx
110             mov edx,17h ;// fs points to local data space
111             mov fs,dx ;// fs 指向局部数据段(局部描述符表中数据段描述符)。
112     ;// 下面这句操作数的含义是: 调用地址 = _sys_call_table + %eax * 4。参见列表后的说明。
113     ;// 对应的C 程序中的sys_call_table 在include/linux/sys.h 中，其中定义了一个包括72个
114     ;// 系统调用C 处理函数的地址数组表。
115             call [_sys_call_table+eax*4]
116             push eax ;// 把系统调用号入栈。
117             mov eax,_current ;// 取当前任务（进程）数据结构地址??eax。
118     ;// 下面97-100 行查看当前任务的运行状态。如果不在就绪状态(state 不等于0)就去执行调度程
        序。
119     ;// 如果该任务在就绪状态但counter[??]值等于0，则也去执行调度程序。
120             cmp dword ptr [state+eax],0 ;// state
121             jne reschedule
122             cmp dword ptr [counter+eax],0 ;// counter
123             je reschedule
124     ;// 以下这段代码执行从系统调用C 函数返回后，对信号量进行识别处理。
125     ret_from_sys_call:
```

· 可是为什么我们用的系统调用，从来没有见过什么 int 0x80 或者 syscall 呢？都是直接调用上层的函数的啊？

· 那是因为Linux的gnu C标准库，给我们把几乎所有的系统调用全部封装了。

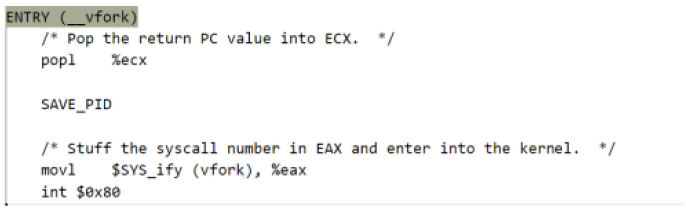


```
ENTRY (__vfork)

   /* Pop the return PC value into RDI.  We need a register that
      is preserved by the syscall and that we're allowed to destroy. */
   popq  %rdi
   cfi_adjust_cfa_offset(-8)

#ifdef SAVE_PID
   SAVE_PID
#endif

   /* Stuff the syscall number in RAX and enter into the kernel.  */
   movl  $SYS_ify (vfork), %eax
   syscall
```



```
ENTRY (__vfork)
   /* Pop the return PC value into ECX.  */
   popl    %ecx

   SAVE_PID

   /* Stuff the syscall number in EAX and enter into the kernel.  */
   movl    $SYS_ify (vfork), %eax
   int $0x80
```

· #define SYS_ify(syscall_name) __NR_##syscall_name： 是一个宏定义，用于将系统调用的名称转换为对应的系统调用号。比如： SYS_ify(open) 会被展开为 __NR_open

· 而系统调用号，不是 glibc 提供的，是内核提供的，内核提供系统调用入口函数 man 2 syscall ，或者直接提供汇编级别软中断命令 int or syscall ，并提供对应的头文件或者开发入口，让上层语言的设计者使用系统调用号，完成系统调用过程

```
源代码路径：linux-2.6.18\linux-2.6.18\include\asm-x86_64\unistd.h
/* at least 8 syscall per cacheline */
#define __NR_read 0
__SYSCALL(__NR_read, sys_read)
#define __NR_write 1
__SYSCALL(__NR_write, sys_write)
#define __NR_open 2
__SYSCALL(__NR_open, sys_open)
#define __NR_close 3
__SYSCALL(__NR_close, sys_close)
#define __NR_stat 4
__SYSCALL(__NR_stat, sys_newstat)
#define __NR_fstat 5
__SYSCALL(__NR_fstat, sys_newfstat)
#define __NR_lstat 6
__SYSCALL(__NR_lstat, sys_newlstat)
#define __NR_poll 7
__SYSCALL(__NR_poll, sys_poll)
#define __NR_lseek 8
__SYSCALL(__NR_lseek, sys_lseek)
#define __NR_mmap 9
__SYSCALL(__NR_mmap, sys_mmap)
#define __NR_mprotect 10
__SYSCALL(__NR_mprotect, sys_mprotect)
#define __NR_munmap 11
__SYSCALL(__NR_munmap, sys_munmap)
#define __NR_brk 12
__SYSCALL(__NR_brk, sys_brk)
#define __NR_rt_sigaction 13
__SYSCALL(__NR_rt_sigaction, sys_rt_sigaction)
#define __NR_rt_sigprocmask 14
__SYSCALL(__NR_rt_sigprocmask, sys_rt_sigprocmask)
#define __NR_rt_sigreturn 15
__SYSCALL(__NR_rt_sigreturn, stub_rt_sigreturn)

...
```

或者部分版本在glibc中，库函数调用实现方式：

代码块

```
# define INTERNAL_SYSCALL_NCS(name, err, nr, args...) \
({ \
    unsigned long int resultvar; \
    LOAD_ARGS_#nr (args) \
    LOAD_REGS_#nr \
```

```
6       asm volatile ( \
7       "syscall\n\t" \
8       : "=a" (resultvar) \
9       : "0" (name) ASM_ARGS_#nr : "memory", "cc", "r11", "cx"); \
10      (long int) resultvar; })
```

2.3.5 缺页中断？内存碎片处理？除零野指针错误？

代码块

```
1   void trap_init(void)
2   {
3       int i;
4
5       set_trap_gate(0,÷_error);// 设置除操作出错的中断向量值。以下雷同。
6       set_trap_gate(1,&debug);
7       set_trap_gate(2,&nmi);
8       set_system_gate(3,&int3); /* int3-5 can be called from all */
9       set_system_gate(4,&overflow);
10      set_system_gate(5,&bounds);
11      set_trap_gate(6,&invalid_op);
12      set_trap_gate(7,&device_not_available);
13      set_trap_gate(8,&double_fault);
14      set_trap_gate(9,&coprocessor_segment_overrun);
15      set_trap_gate(10,&invalid_TSS);
16      set_trap_gate(11,&segment_not_present);
17      set_trap_gate(12,&stack_segment);
18      set_trap_gate(13,&general_protection);
19      set_trap_gate(14,&page_fault);
20      set_trap_gate(15,&reserved);
21      set_trap_gate(16,&coprocessor_error);
22      // 下面将int17-48 的陷阱门先均设置为reserved，以后每个硬件初始化时会重新设置自己的
    陷阱
23      门。
24      for (i=17;i<48;i++)
25      set_trap_gate(i,&reserved);
26      set_trap_gate(45,&irq13);// 设置协处理器的陷阱门。
27      outb_p(inb_p(0x21)&0xfb,0x21);// 允许主8259A 芯片的IRQ2 中断请求。
28      outb(inb_p(0xA1)&0xdf,0xA1);// 允许从8259A 芯片的IRQ13 中断请求。
29      set_trap_gate(39,¶llel_interrupt);// 设置并行口的陷阱门。
30  }
```

· 缺页中断？内存碎片处理？除零野指针错误？这些问题，全部都会被转换成为CPU内部的软中断，然后走中断处理例程，完成所有处理。有的是进行申请内存，填充页表，进行映射的。有的是用来

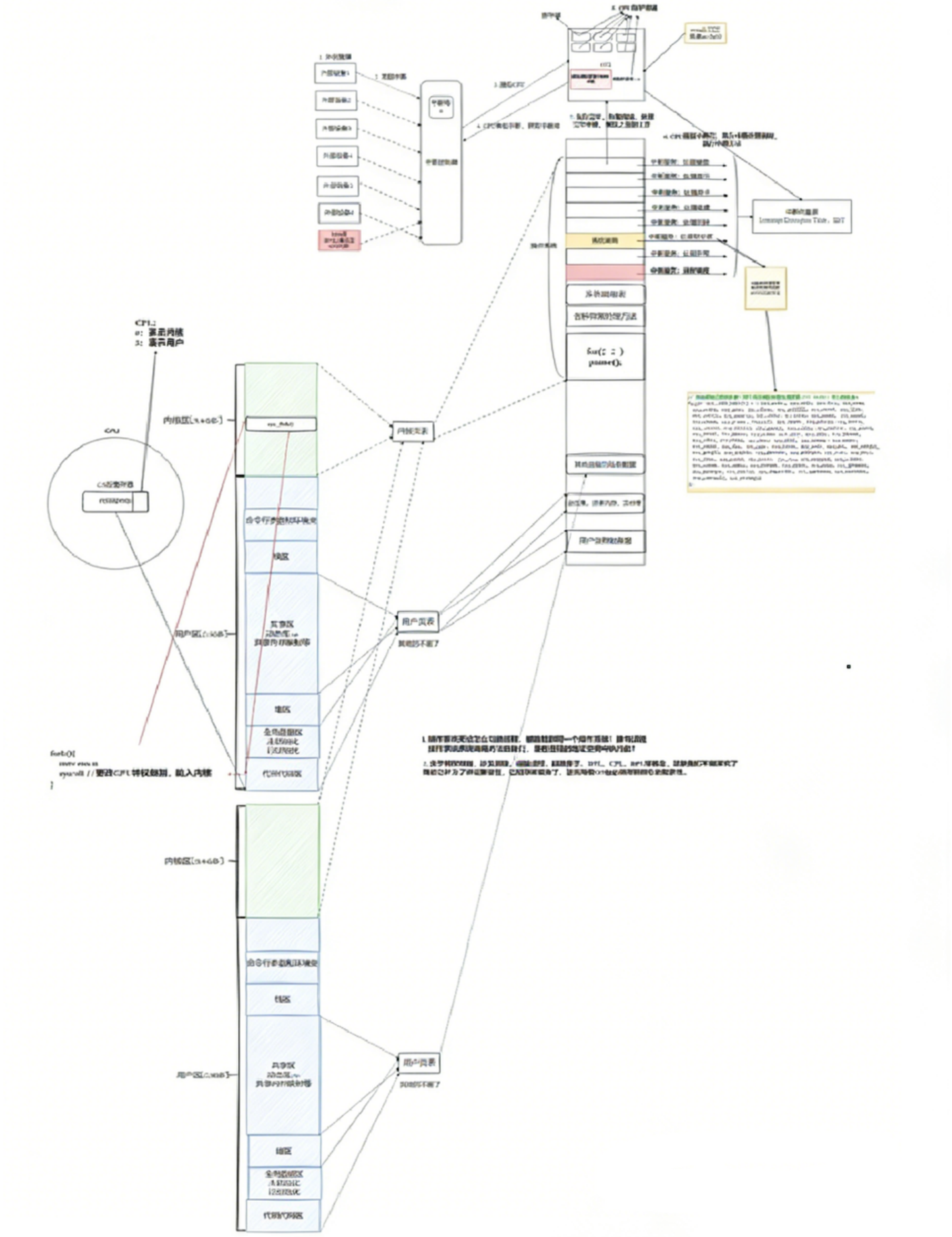
处理内存碎片的，有的是用来给目标进行发送信号，杀掉进程等等。

❤️ 📌 所以：· 操作系统就是躺在中断处理例程上的代码块！

· CPU内部的软中断，比如int 0x80或者syscall，我们叫做 陷阱

· CPU内部的软中断，比如除零/野指针等，我们叫做 异常。（所以，能理解“缺页异常”为什么这么叫了吗？）

2.4 如何理解内核态和用户态



结论：

· 操作系统无论怎么切换进程，都能找到同一个操作系统！换句话说操作系统系统调用方法的执行，

是在进程的地址空间中执行的!

· 关于特权级别，涉及到段，段描述符，段选择子，DPL，CPL，RPL等概念， 而现在芯片为了保证兼容性，已经非常复杂了，进而导致OS也必须得照顾它的复杂性，这块我们不做深究了。

· 用户态就是执行用户[0,3]GB时所处的状态

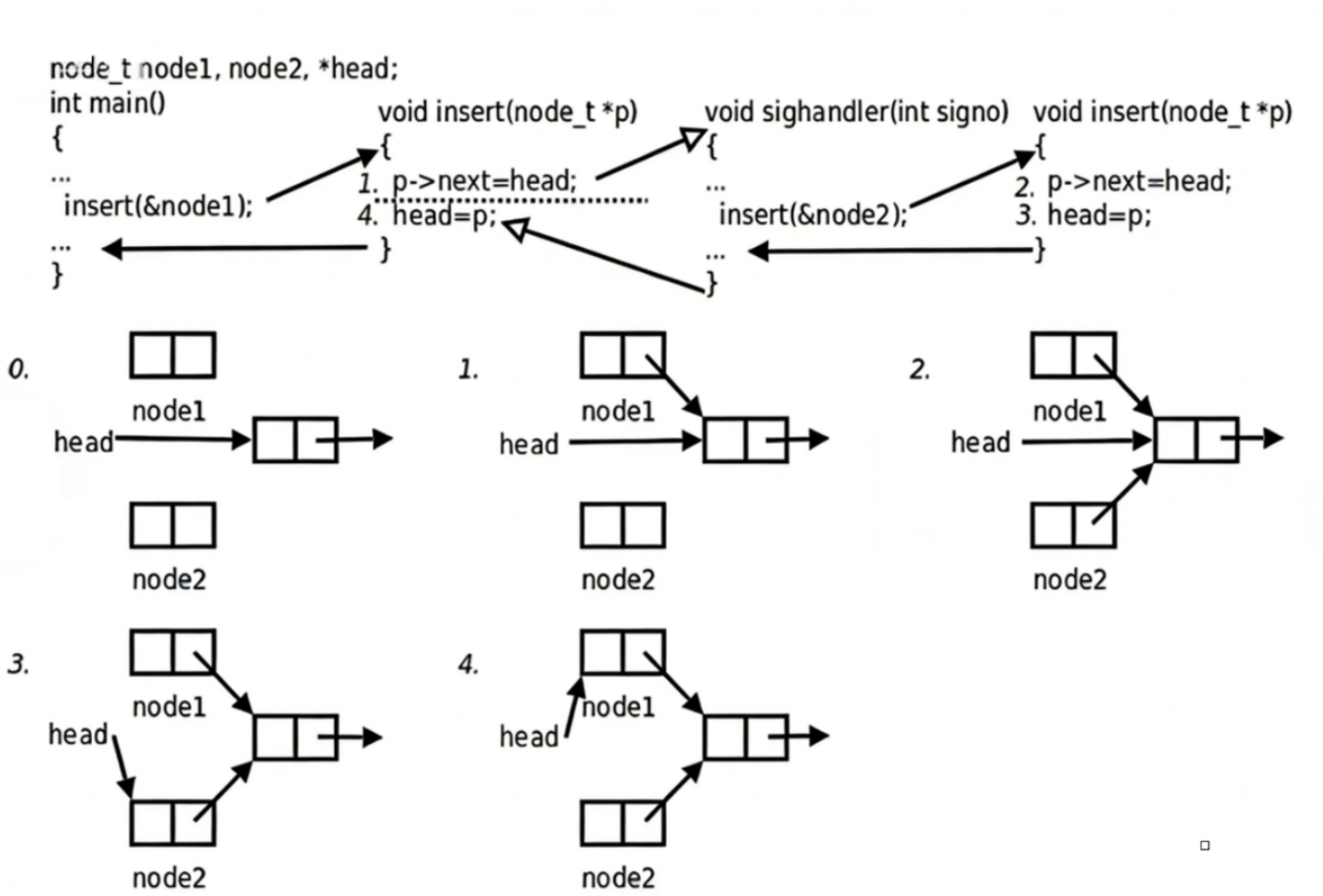
· 内核态就是执行内核[3,4]GB时所处的状态

· 区分就是按照CPU内的CPL决定，CPL的全称是Current Privilege Level，即当前特权级别。

· 一般执行 int 0x80 或者 syscall 软中断，CPL会在校验之后自动变更(怎么校验看学生反映)

· 这样会不会不安全？？

3. 可重入函数



> · main函数调用insert函数向一个链表head中插入节点node1,插入操作分为两步,刚做完第一步的时候,因为硬件中断使进程切换到内核,再次回用户态之前检查到有信号待处理,于是切换 到sighandler函数,sighandler也调用insert函数向同一个链表head中插入节点node2,插入操作的两步都做完之后从sighandler返回内核态,再次回到用户态就从main函数调用的insert函数中继续往下执行,先前做第一步之后被打断,现在继续做完第二步。结果是,main函数和sighandler先后 向

> 链表中插入两个节点,而最后只有一个节点真正插入链表中了。
>
> · 像上例这样,insert函数被不同的控制流程调用,有可能在第一次调用还没返回时就再次进入该函数,这称为重入,insert函数访问一个全局链表,有可能因为重入而造成错乱,像这样的函数称为 不可重入函数,反之,如果一个函数只访问自己的局部变量或参数,则称为可重入(Reentrant) 函数。想一下,为什么两个不同的控制流程调用同一个函数,访问它的同一个局部变量或参数就不会造成错乱?
>
> 如果一个函数符合以下条件之一则是不可重入的:
>
> · 调用了malloc或free,因为malloc也是用全局链表来管理堆的。
>
> · 调用了标准I/O库函数。标准I/O库的很多实现都以不可重入的方式使用全局数据结构。

4. volatile

· 该关键字在C当中我们已经有所涉猎，今天我们站在信号的角度重新理解一下

代码块

```
[hb@localhost code_test]$ cat sig.c
1#include <stdio.h>
#include <signal.h>

int flag = 0;
void handler(int sig)
{
    printf("chage flag 0 to 1\n");
    flag = 1;
}

int main()
{
    signal(2, handler);
    while(!flag);
    printf("process quit normal\n");
    return 0;
}

[hb@localhost code_test]$ cat Makefile
sig:sig.c
    gcc -o sig sig.c #-O2
.PHONY:clean
clean:
    rm -f sig
[hb@localhost code_test]$ ./sig
^Cchage flag 0 to 1
process quit normal
```

> 标准情况下，键入 CTRL-C ,2号信号被捕捉，执行自定义动作，修改 flag=1 ， while 条件不满足, 退出循环，进程退出

代码块

```
[hb@localhost code_test]$ cat sig.c
#include <stdio.h>
#include <signal.h>

int flag = 0;
void handler(int sig)
{
    printf("chage flag 0 to 1\n");
    flag = 1;
}

int main()
{
    signal(2, handler);
    while(!flag);
    printf("process quit normal\n");
    return 0;
}

[hb@localhost code_test]$ cat Makefile
sig:sig.c
    gcc -o sig sig.c -O2
.PHONY:clean
clean:
    rm -f sig
[hb@localhost code_test]$ ./sig
^Cchage flag 0 to 1
^Cchage flag 0 to 1
^Cchage flag 0 to 1
```

> 优化情况下，键入 CTRL-C ，2号信号被捕捉，执行自定义动作，修改 flag=1 ，但是 while 条
>
> 件依旧满足,进程继续运行！但是很明显flag肯定已经被修改了，但是为何循环依旧执行？很明显，
>
> while 循环检查的 flag，并不是内存中最新的 flag，这就存在了数据二异性的问题。 while 检

> 测的 `flag` 其实已经因为优化，被放在了CPU寄存器当中。如何解决呢？很明显需要 `volatile`

代码块

```
[hb@localhost code_test]$ cat sig.c
#include <stdio.h>
#include <signal.h>

volatile int flag = 0;
void handler(int sig)
{
    printf("chage flag 0 to 1\n");
    flag = 1;
}

int main()
{
    signal(2, handler);
    while(!flag);
    printf("process quit normal\n");
    return 0;
}

[hb@localhost code_test]$ cat Makefile
sig:sig.c
    gcc -o sig sig.c -O2
.PHONY:clean
clean:
    rm -f sig
[hb@localhost code_test]$ ./sig
^Cchage flag 0 to 1
process quit normal
```

> · volatile 作用：保持内存的可见性，告知编译器，被该关键字修饰的变量，不允许被优化，对该变量的任何操作，都必须在真实的内存中进行操作

5. SIGCHLD信号

进程一章讲过用wait和waitpid函数清理僵尸进程,父进程可以阻塞等待子进程结束,也可以非阻塞地查询是否有子进程结束等待清理(也就是轮询的方式)。采用第一种方式,父进程阻塞了就不 能处理自己的工作了;采用第二种方式,父进程在处理自己的工作的同时还要记得时不时地轮询一 下,程序实现复杂。

其实,子进程在终止时会给父进程发SIGCHLD信号,该信号的默认处理动作是忽略,父进程可以自 定义

SIGCHLD信号的处理函数,这样父进程只需专心处理自己的工作,不必关心子进程了,子进程终止时会通知父进程,父进程在信号处理函数中调用wait清理子进程即可。

请编写一个程序完成以下功能:父进程fork出子进程,子进程调用exit(2)终止,父进程自定 义SIGCHLD信号的处理函数, 在其中调用wait获得子进程的退出状态并打印。

事实上,由于UNIX 的历史原因,要想不产生僵尸进程还有另外一种办法:父进程调用sigaction将SIGCHLD的处理动作置为SIG_IGN,这样fork出来的子进程在终止时会自动清理掉,不会产生僵尸进程,也不会通知父进程。系统默认的忽略动作和用户用sigaction函数自定义的忽略 通常是没有区别的,但这

是一个特例。此方法对于Linux可用,但不保证在其它UNIX系统上都可 用。请编写程序验证这样做不会产生僵尸进程。

测试代码

代码块

```c
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
void handler(int sig)
{
    pid_t id;
    while( (id = waitpid(-1, NULL, WNOHANG)) > 0)
    {
        printf("wait child success: %d\n", id);
    }
    printf("child is quit! %d\n", getpid());
}

int main()
{
    signal(SIGCHLD, handler);
    pid_t cid;
    if((cid = fork()) == 0)
    {
        //child
        printf("child : %d\n", getpid());
        sleep(3);
        exit(1);
    }

    while(1)
    {
        printf("father proc is doing some thing!\n");
```

```
29            sleep(1);
30        }
31        return 0;
32    }
```

6. 附录

用户态和内核态

- `CPU 指令集` ：是 CPU 实现软件指挥硬件执行的媒介，具体来说每一条汇编语句都对应了一条 `CPU 指令` ，而非常非常多的 `CPU 指令` 在一起，可以组成一个、甚至多个集合，指令的集合叫 `CPU 指令集` 。
- `CPU 指令集` 有权限分级，大家试想， `CPU 指令集` 可以直接操作硬件的，要是因为指令操作的不规范，造成的错误会影响整个计算机系统的。好比你写程序，因为对硬件操作不熟悉，导致操作系统内核、及其他所有正在运行的程序，都可能会因为操作失误而受到不可挽回的错误，最后只能重启计算机才行。
    - 对开发人员来说是个艰巨的任务，还会增加负担，同时开发人员在这方面也不被信任，所以操作系统内核直接屏蔽开发人员对硬件操作的可能，都不让你碰到这些 `CPU 指令集` 。

针对上面的需求，硬件设备商直接提供硬件级别的支持，做法就是对 `CPU 指令集` 设置了权限，不同级别权限能使用的 `CPU 指令集` 是有限的，以 Inter CPU 为例，Inter把 `CPU 指令集` 操作的权限由高到低划为4级：

- ring 0：权限最高，可以使用所有 `CPU 指令集`
- ring 1
- ring 2
- ring 3：权限最低，仅能使用常规 CPU 指令集 ，不能使用操作硬件资源的 CPU 指令集 ，比如 IO 读写、网卡访问、申请内存都不行

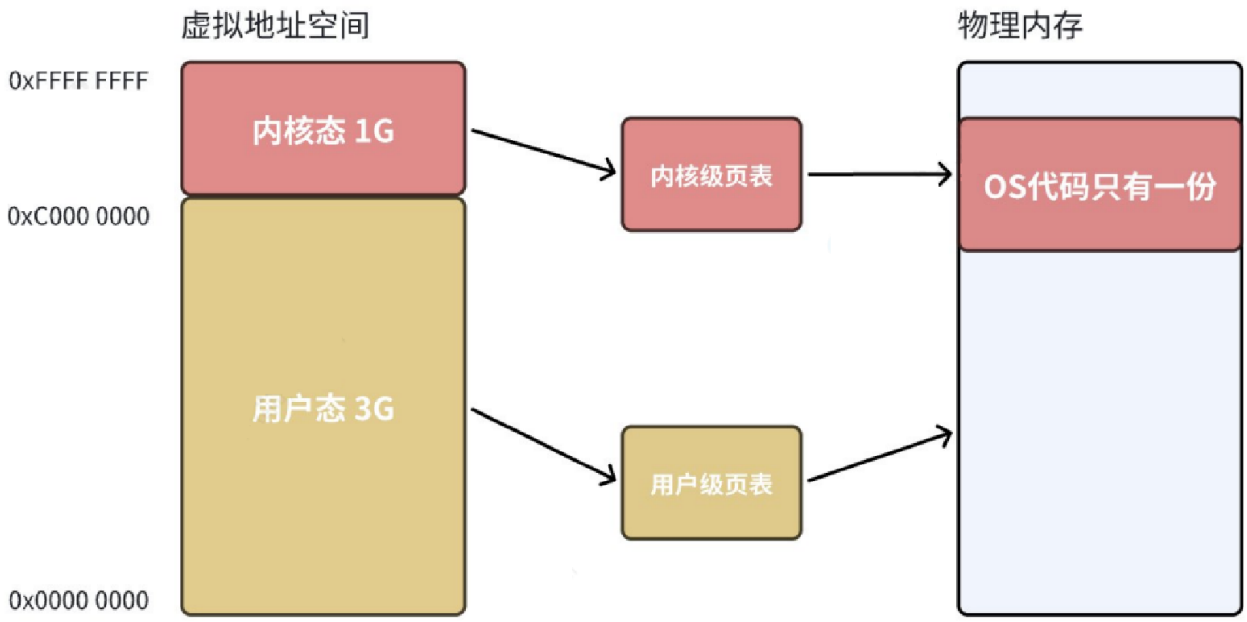
要知道的是，Linux系统仅采用ring 0 和 ring 3 这2个权限。**CPU中有一个标志字段，标志着线程的运行状态，用户态为3，内核态为0。**

- ring 0被叫做内核态，完全在操作系统内核中运行
    - 执行内核空间的代码，具有ring 0保护级别，有对硬件的所有操作权限，可以执行所有 `CPU 指令集` ，访问任意地址的内存，在内核模式下的任何异常都是灾难性的，将会导致整台机器停机
- ring 3被叫做用户态，在应用程序中运行
    - 在用户模式下，具有ring 3保护级别，代码没有对硬件的直接控制权限，也不能直接访问地址的内存，程序是通过调用系统接口(System Call APIs)来达到访问硬件和内存，在这种保护模式下，即时程序发生崩溃也是可以恢复的，在电脑上大部分程序都是在，用户模式下运行的

低权限的资源范围较小，高权限的资源范围更大，所以用户态与内核态的概念就是CPU 指令集权限的区别。

我们通过指令集权限区分用户态和内核态，还限制了内存资源的使用，操作系统为用户态与内核态划分了两块内存空间，给它们对应的指令集使用。

在内存资源上的使用，操作系统对用户态与内核态也做了限制，每个进程创建都会分配虚拟空间地址，以Linux32位操作系统为例，它的寻址空间范围是 **4G** （2的32次方），而操作系统会把虚拟控制地址划分为两部分，一部分为内核空间，另一部分为用户空间，高位的 **1G** （从虚拟地址 0xC0000000 到 0xFFFFFFFF）由内核使用，而低位的 **3**G （从虚拟地址 0x00000000 到 0xBFFFFFFF）由各个进程使用。



- 用户态：只能操作 0-3G 范围的低位虚拟空间地址
- 内核态： 0-4G 范围的虚拟空间地址都可以操作，尤其是对 3-4G 范围的高位虚拟空间地址必须由内核态去操作
    - 3G-4G 部分大家是共享的（指所有进程的内核态逻辑地址是共享同一块内存地址），是内核态的地址空间，这里存放在整个内核的代码和所有的内核模块，以及内核所维护的数据。
    - 在内核运行的过程中，会涉及内核栈的分配，内核的进程管理的代码会将内核栈创建在内核空间中，当然相应的页表也会被创建。

用户态与内核态的切换

什么情况会导致用户态到内核态切换？？

> - 系统调用 ： 用户态进程主动切换到内核态的方式，用户态进程通过系统调用向操作系统申请资源完成工作，例如 fork（）就是一个创建新进程的系统调用。

    - 操作系统提供了中断指令int 0x80来主动进入内核，这是用户程序发起的调用访问内核代码的唯一方式。调用系统函数时会通过内联汇编代码插入int 0x80的中断指令，内核接收到int 0x80中断后，查询中断处理函数地址，随后进入系统调用。
- 异常： 当 CPU 在执行用户态的进程时，发生了一些没有预知的异常，这时当前运行进程会切换到处理此异常的内核相关进程中，也就是切换到了内核态，如缺页异常
- 中断： 当 CPU 在执行用户态的进程时，外围设备完成用户请求的操作后，会向 CPU 发出相应的中断信号，这时 CPU 会暂停执行下一条即将要执行的指令，转到与中断信号对应的处理程序去执行，也就是切换到了内核态。如硬盘读写操作完成，系统会切换到硬盘读写的中断处理程序中执行后边的操作等。

切换时 CPU 需要做什么？？

- 当某个进程中要读写 IO ，必然会用到 ring 0 级别的 CPU 指令集 。而此时 CPU 的指令集操作权限只有 ring 3，为了可以操作ring 0 级别的 CPU 指令集 ， CPU 切换指令集操作权限级别为ring 0（可称之为提权），CPU再执行相应的ring 0 级别的 CPU 指令集 （内核代码）。
- 代码发生提权时，CPU 是需要切换栈的！！前面我们提到过，内核有自己的内核栈。CPU 切换栈是需要栈段描述符（ss寄存器）和栈顶指针（esp寄存器），这两个值从哪里来?
    - CPU通过一个段寄存器（tr）确定 TSS（任务状态段，struct TSS） 的位置。在TSS结构中存在这么一个 SS0 和 ESP0。提权的时候，CPU就从这个TSS里把SS0和ESP0取出来，放到 ss 和 esp 寄存器中。

切换流程

1. 从用户态切换到内核态时，首先用户态可以直接读写寄存器，用户态操作CPU，将寄存器的状态保存到对应的内存中，然后调用对应的系统函数，传入对应的用户栈地址和寄存器信息，方便后续内核方法调用完毕后，恢复用户方法执行的现场。
2. 从用户态切换到内核态需要提权，CPU 切换指令集操作权限级别为 ring 0。
3. 提权后，切换内核栈。然后开始执行内核方法，相应的方法栈帧时保存在内核栈中。
4. 当内核方法执行完毕后，CPU切换指令集操作权限级别为 ring 3，然后利用之前写入的信息来恢复用户栈的执行。

从上述流程可以看出用户态切换到内核态的时候，会牵扯到用户态现场信息的保存以及恢复，还要进行一系列的安全检查，还是比较耗费资源的。