操作系统　第四章　进程管理

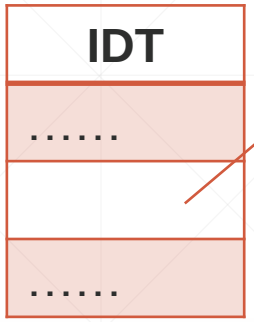
4.2 中断与调度

同济大学计算机系

一、4种不同的中断 与 调度系统的关系

- 外部中断
  - 外设中断
  - 时钟中断
- 系统调用
- 异常

1、外设中断的响应过程




```
void 中断入口程序()
{
    SaveContext();
    SwitchToKernel();

    CallHandler(中断处理程序);

    例行调度

    RestoreContext();
    Leave();
    InterruptReturn();
}
```

```
void 中断处理程序()
{     // 以读操作为例
    接收外设数据，送入系统缓存
    EOI
    唤醒等待IO操作结束的进程 PB
    if ( PB.p_pri < CurPri )
        RunRun++
}
```

- 现运行进程响应外设中断，执行中断处理程序
- 如果中断处理程序有唤醒优先权更高的进程，RunRun++
- 返回中断入口程序，例行调度

外设中断 与 调度

- 先前态核心态，不管，恢复中断现场，返回。核心态不调度。
- 先前态用户态 & RunRun非0，现运行进程执行Swtch( )，把处理器让给中断处理程序唤醒的进程。现运行进程响应中断被剥夺，运行 → 就绪。
- 被唤醒的进程从Swtch( )返回，恢复运行。

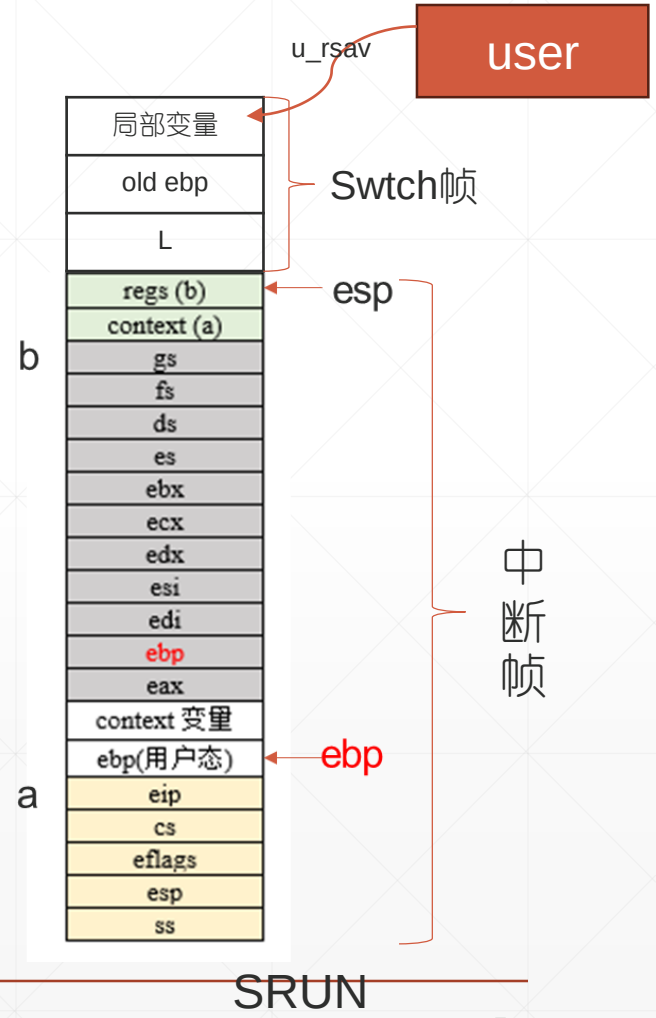
```
struct pt_context *context;
__asm__ __volatile__ ("    movl %%ebp, %0; addl $0x4, %0 " : "+m" (context) );

if( context->xcs & USER_MODE )  /*先前为用户态*/
{
        while(true)
        {
                X86Assembly::CLI();         /* 关中断 */

                if(Kernel::Instance().GetProcessManager().RunRun > 0)    /* 判断 RunRun 标识
                {
                        X86Assembly::STI();         /* 开中断 */
                        Kernel::Instance().GetProcessManager().Swtch();   // 现运行进程放弃CPU
                }
                else
                {
                        break;    /* 如果runrun == 0，则退栈回到用户态继续执行应用程序 */
                }
        }
}
```

- **现运行进程被剥夺**，断点在 指令 L，是Swtch( )的返回地址。
- 很久以后，当该进程再次成为优先级最高的进程时，会被系统选中。
- 作为新运行进程，从Swtch返回，回到中断入口程序：弹出所有用户态寄存器，恢复应用程序执行现场。IRET返回用户态继续执行应用程序

```
void 中断入口程序()
{
    SaveContext();
    SwitchToKernel();

    CallHandler(中断处理程序);

    例行调度

    RestoreContext();
    Leave();
    InterruptReturn();
}
```

```
struct pt_context *context;
__asm__ __volatile__ ("    movl %%ebp, %0; addl $0x4, %0 " : "+m" (context) );

if( context->xcs & USER_MODE )  /*先前为用户态*/
{
L:  while(true)
    {
        X86Assembly::CLI();        /* 关中断 */

        if(Kernel::Instance().GetProcessManager().RunRun > 0)    /* 判
        {
            X86Assembly::STI();        /* 开中断 */
            Kernel::Instance().GetProcessManager().Swtch();   // 玉
        }
        else
        {
            break;    /* 如果runrun == 0，则退栈回到用户态继续执
        }
    }
}
```

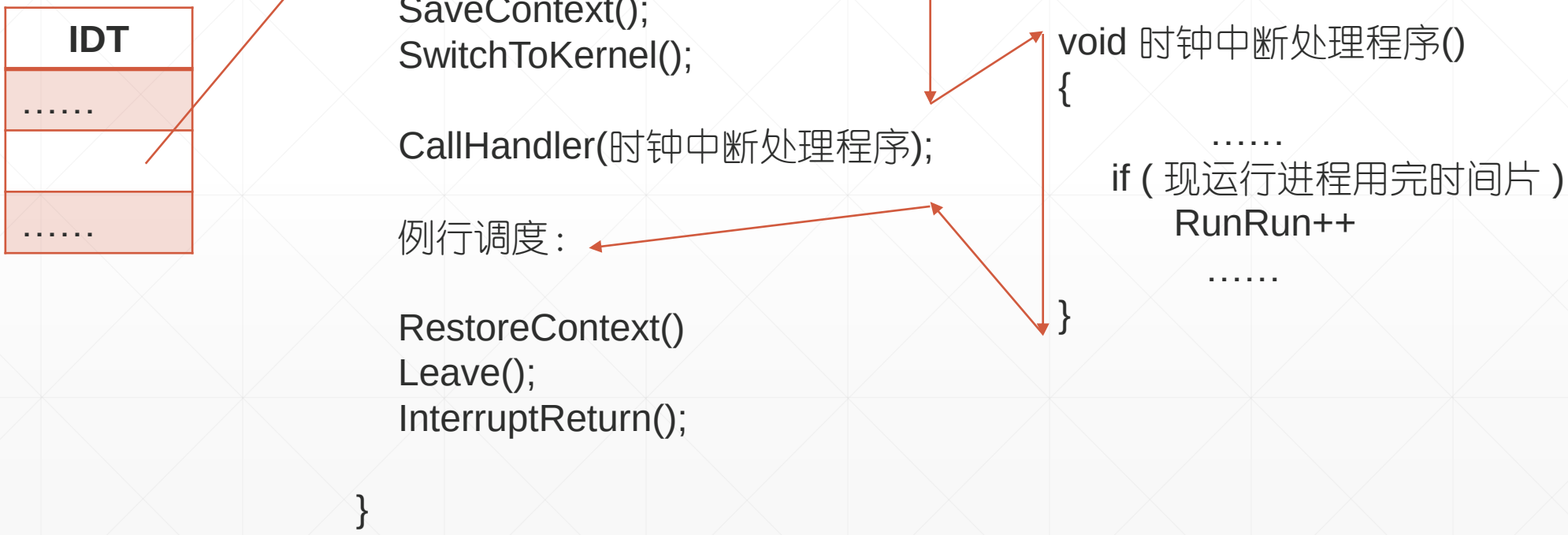
1 2 3 4



例 1

- T时刻，现运行进程PA 执行系统调用。响应外设中断，唤醒一个睡眠进程PB。问，PB进程何时上台运行?

2、时钟中断的响应过程


IDT
.....
.....
void 时钟中断入口程序()
{
SaveContext();
SwitchToKernel();
CallHandler(时钟中断处理程序);
例行调度：
RestoreContext()
Leave();
InterruptReturn();
}
void 时钟中断处理程序()
{
.....
if (现运行进程用完时间片)
RunRun++
.....
}

时钟中断 与 调度

- 现运行进程时间片未用完，中断处理结束后，继续执行被中断任务。
- 时间片用完
  - 先前态核心态。继续被中断任务。
  - 先前态用户态，被剥夺，处理器执行等待时间最久的应用程序。
- 被剥夺的现运行进程就绪，随着等待时间增加，它的优先权会上升。时间片轮转一轮后，它会得到运行机会，恢复用户态寄存器，继续执行应用程序。

```
struct pt_context *context;
__asm__ __volatile__ ("    movl %%ebp, %0; addl $0x4, %0 " : "+m" (context) );

if( context->xcs & USER_MODE )  /*先前为用户态*/
{
        while(true)
        {
                X86Assembly::CLI();         /* 关中断 */

                if(Kernel::Instance().GetProcessManager().RunRun > 0)     /* 判断 RunRun 标识
                {
                        X86Assembly::STI();         /* 开中断 */
                        Kernel::Instance().GetProcessManager().Swtch();   // 现运行进程放弃CPU
                }
                else
                {
                        break;    /* 如果runrun == 0，则退栈回到用户态继续执行应用程序 */
                }
        }
}
```

3、系统调用

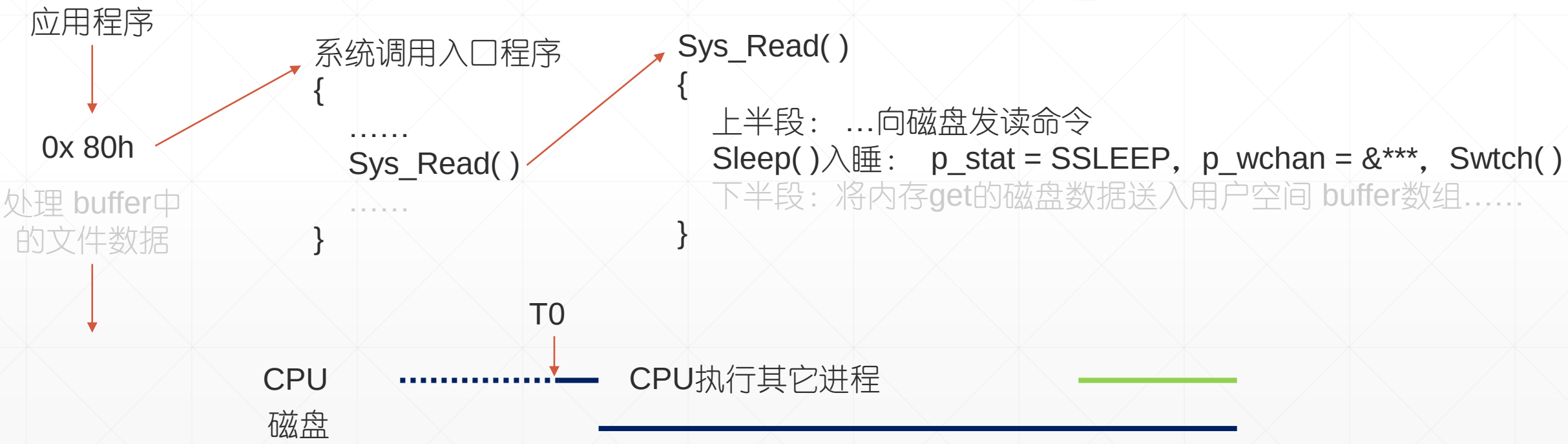
```
#include <fcntl.h>
char buffer[2048];
int version = 1;

    ......

copyOperation (old,new)
      int old,new;
{
      int count;
N:    while ((count=read(old,buffer,sizeof(buffer)))>0)
            write(new, buffer, count);
}
```

应用程序执行 read 系统调用。负责执行这个程序的进程陷入内核，读取磁盘文件数据，将其送入用户空间，buffer数组。完成后，系统调用结束。进程返回用户态，执行应用程序处理 buffer数组中的文件数据，把它写进另一个文件 new。

read 系统调用的执行细节 1

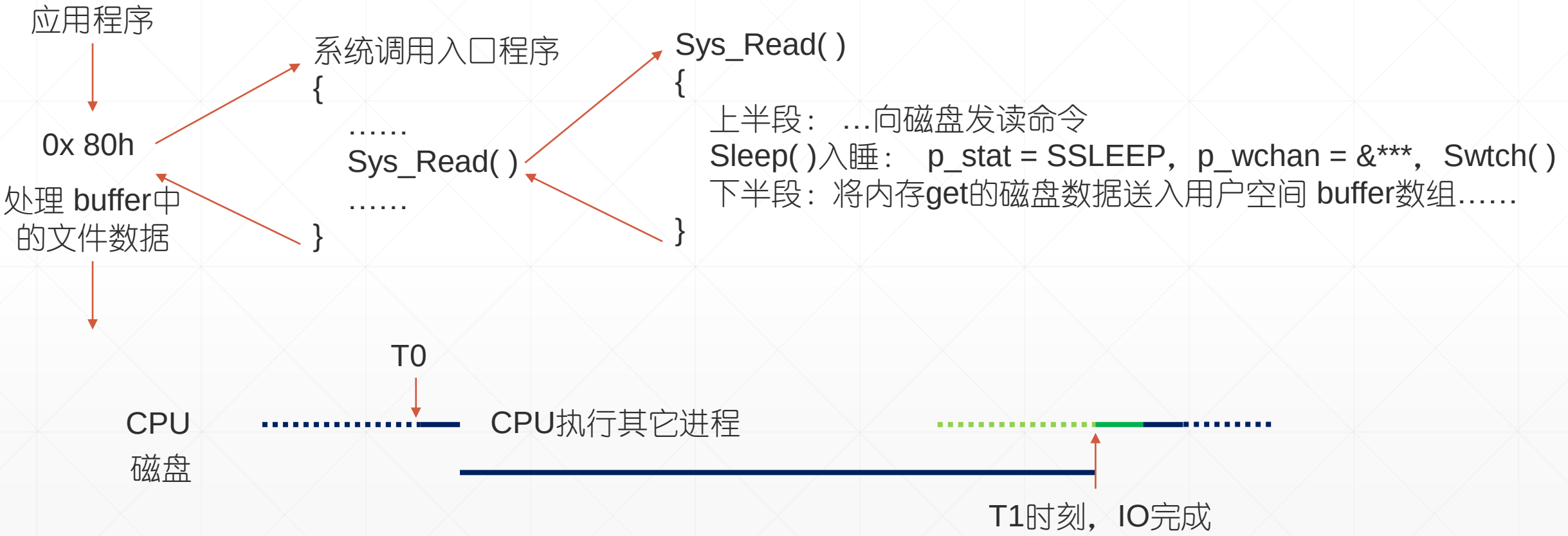
T0 时刻，PA 执行应用程序，子程序 CopyOperation 执行 read 系统调用读 old 文件数据。

```
copyOperation (old,new)
      int old,new;
{
      int count;
N:    while ((count=read(old,buffer,sizeof(buffer)))>0)
            write(new, buffer, count);
}
```



read 系统调用的执行细节 2

T1时刻，磁盘IO完成。现运行进程PB用户态运行，执行showStack( )函数。


应用程序
0x 80h
处理 buffer中
的文件数据
系统调用入口程序
{
……
Sys_Read( )
……
}
Sys_Read( )
{
上半段：…向磁盘发读命令
Sleep( )入睡： p_stat = SSLEEP，p_wchan = &***，Swtch( )
下半段：将内存get的磁盘数据送入用户空间 buffer数组……
}
T0
CPU
磁盘
CPU执行其它进程
T1时刻，IO完成

4、异常

CPU无法顺利执行当前指令，给自己发中断信号。叫做异常。

1、指令不对
非法操作码
CPU用户态，执行了特权指令
运算溢出 或 除数是0

2、debug的程序走到断点

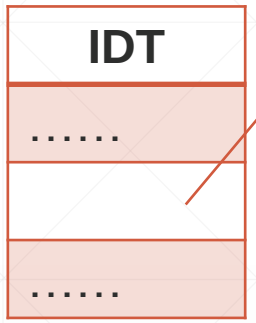
断点处，gdb会插入int 3指令。
被调试的程序走到断点，就会执行 int 3 指令，抛出3#异常。

3、缺页
- 野指针：指针引用了未经分配的内存单元，PTE为空
- 写只读页：赋值操作，PTE是RO
- 应用程序访问内核页面：CPL==11，PTE是S
- 堆栈扩展：栈溢出，ESP所在的PTE为空
- 虚拟存储器请求调页：PTE P为0的合法页面

} 非法内存访问

特权指令 包括：
- in，out 指令
- STI，CLI指令
- 除 通用寄存器 和 EFLAGS 外，访问CPU其它寄存器的指令。

异常处理过程




```
void 异常入口程序()
{
    SaveContext();
    SwitchToKernel();

    CallHandler(时钟中断处理程序);

    例行调度：

    RestoreContext()
    Leave();
    InterruptReturn();

}
```

```
void 异常处理程序()  // 非14#缺页异常
{
    异常发生在核心态？
    Y：panic // 输出警告信息，死循环
    N：向现运行进程发信号
       应用程序有定义信号处理程序？
       没有（默认），执行exit终止
       有，返回用户态执行信号处理程序
}
```

异常转化为信号，信号杀死应用程序

应用程序可以直接读磁盘文件，不执行系统调用吗?

- 不行。
- 在多道系统中，应用程序不可以直接访问磁盘文件，这是为了保护系统资源；不可以访问内核页面。也就是，不能直接 call 内核子程序，不能读写内核变量；更不能执行特权指令向磁盘发IO命令。
- 所以，应用程序想要读取磁盘文件数据，必需执行 int指令 或 syscall指令，借助中断机制提升CPU特权级，让进程在read系统调用执行期间拥有访问内核、控制系统完备的权力：它可以执行特权指令向外设发IO命令，可以控制CPU 和 所有芯片的运行模式。。。

例 2

- **某程序员使用Word写文档，同时运行一个程序打印网上下载的文件。已知：网卡中断优先级>键盘中断优先级>打印机中断优先级。**
- **T 时刻，现运行进程word用户态执行，执行拼写检查任务。系统收到打印机发来的中断请求。中断响应，执行打印机中断处理程序。**
- **打印机中断处理程序执行时，系统收到网卡发来的中断请求。网卡中断优先级高，立即响应，发生中断嵌套。**

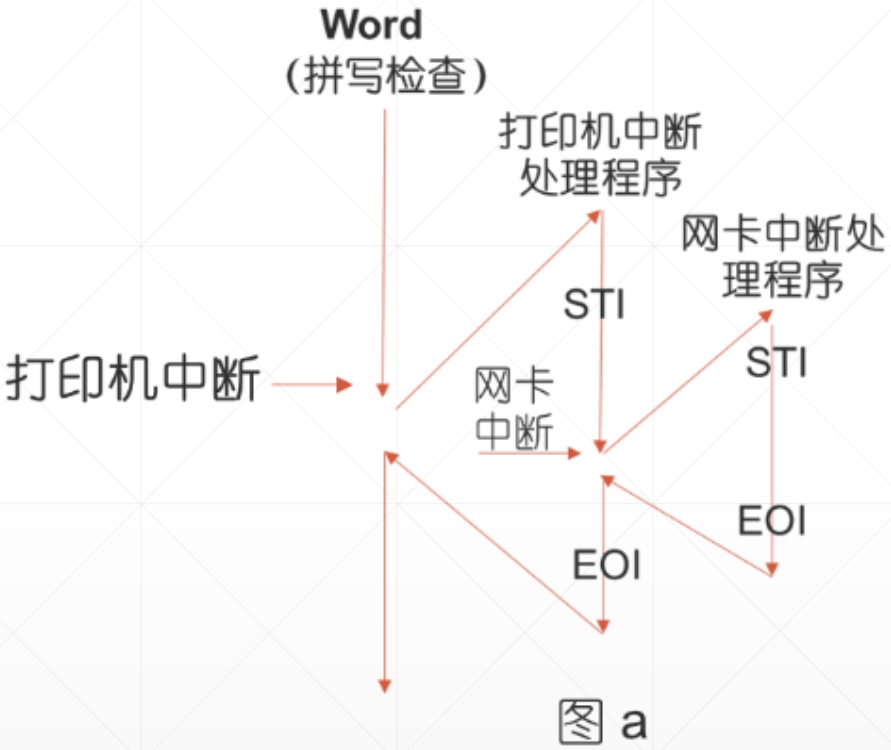


图 a

问：1、执行拼写检查的word进程什么时候放弃CPU？
2、word进程放弃CPU后，调度系统状态如何？
后续，CPU执行的任务序列？

操作系统　第四章　进程管理

4.3 进程的优先级

同济大学计算机系

六、进程调度 —— 进程的优先级

Unix V6++使用动态优先级调度算法

- 每个进程的Process结构记录进程的优先数 p_pri ∈ [-100, 255]。优先数越小，优先级越高。现运行进程让出CPU后，系统选p_pri最小的、图像全部在内存的就绪进程（回看 select 函数）
- 合理设置进程优先级，有利于提升系统效率，公平对待所有应用程序。总的策略是：
  - 独占使用的系统资源，持有者进程优先级高。目的在于缩短资源使用的平均时长，提高资源的周转速度。
  - 马上会发出IO请求的进程优先级高。这样可以提高CPU和外设的并行程度。
  - 应用程序时间片轮转。现运行进程连续运行一个时间片后，让出CPU。非现运行进程，排队时间很久以后，优先级上升。这样，现运行进程让出CPU后，排队时间最久的进程将获得运行机会。

进程优先级的确定方式

- UNIX V6++进程的优先数是动态变化的。除非正在执行系统调用，否则，

  p-pri = min { 127 , 进程的静态优先数 + （p_cpu/16）}

- 进程的静态优先数 = PUSER(100) + p_nice。

- 进程执行系统调用入睡时，设置与其睡眠原因相匹配的优先数。持有资源竞争越激烈，进程优先级越高。
  - SSLEEP，p_pri∈[-100,0 ）。进程入睡等待磁盘IO结束，等待使用 磁盘高速缓存 或 内存Inode。
  - SWAIT，p_pri ∈[0, 256 ）。进程入睡等待其它事件。
- 唤醒后，维持睡眠优先数不动。直至下次入睡设置新的优先数，或进程执行完系统调用下半部，返回用户态。

Unix V6进程优先数 p_pri

高优先级睡眠
p_stat = SSLEEP

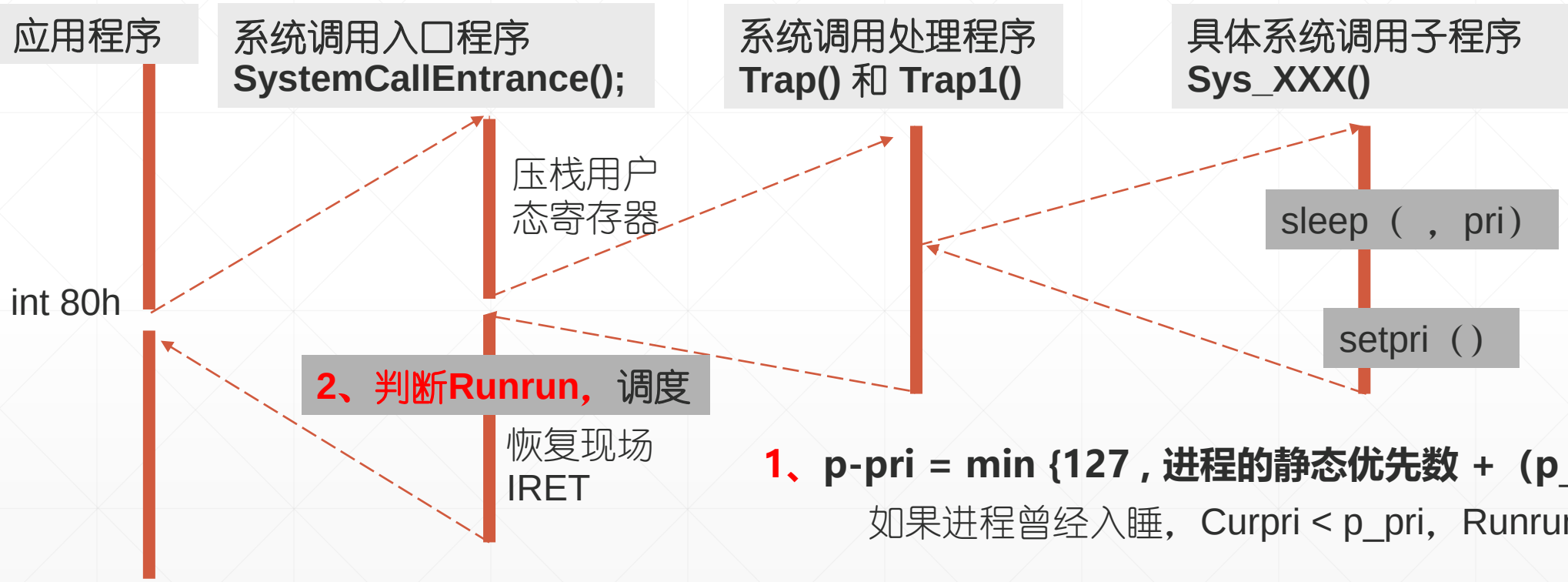
唤醒后，进程优先级非常高，系统调用能够得到立即执行

低优先级睡眠
p_stat = SWAIT

唤醒后，进程优先级高，系统调用能够也能得到立即执行

序号	名称	优先数值	
1	PSWP	-100	内核线程sched (0#进程) swap in，swap out
2	PINOD	-90	
3	PRIBIO	-50	读写磁盘文件的 read/write 系统调用
4	EXPRI	-1	
5	PPIPE	1	读写管道的 read/write 系统调用
6	TTIPRI	10	scanf,getc，读键盘的read 系统调用
7	TTOPRI	20	printf,putc，写屏幕的write 系统调用
8	PWAIT	40	Wait，父进程等待子进程终止的系统调用
9	PSLEP	90	sleep(seconds)系统调用
10	PUSER	100	应用程序优先数的基数

系统调用执行完毕，返回用户态前，恢复进程的用户态优先权



1、p-pri = min {127 , 进程的静态优先数 + （p_cpu/16）}

如果进程曾经入睡，Curpri < p_pri，Runrun++

例 2 续

- **某程序员使用Word写文档，同时运行一个程序打印网上下载的文件。已知：网卡中断优先级>键盘中断优先级>打印机中断优先级。**
- **T 时刻，现运行进程word用户态执行，执行拼写检查任务。系统收到打印机发来的中断请求。中断响应，执行打印机中断处理程序。**
- **打印机中断处理程序执行时，系统收到网卡发来的中断请求。网卡中断优先级高，立即响应，发生中断嵌套。**

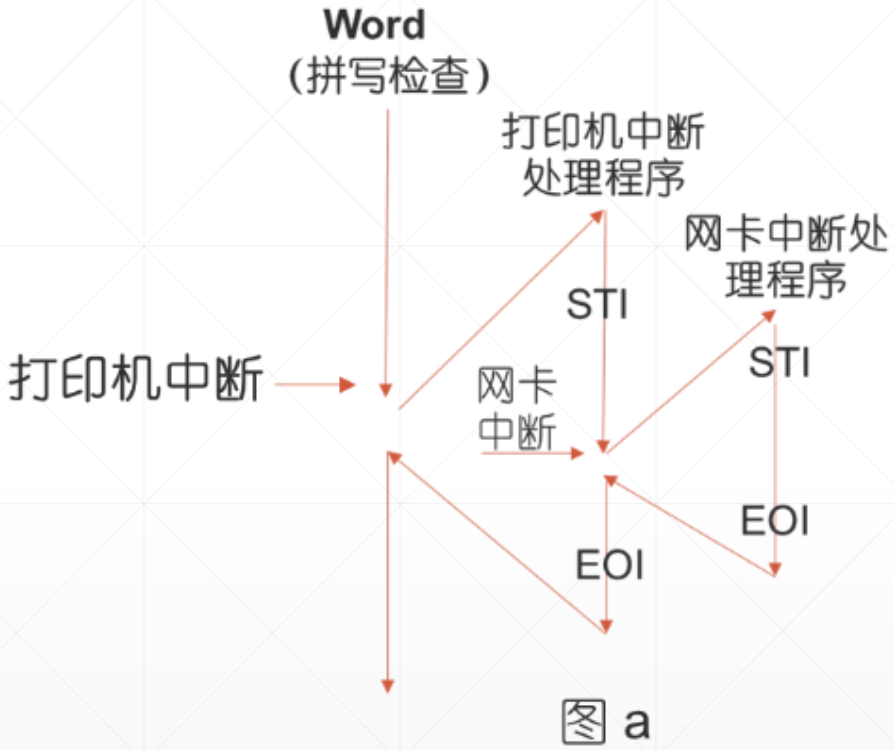
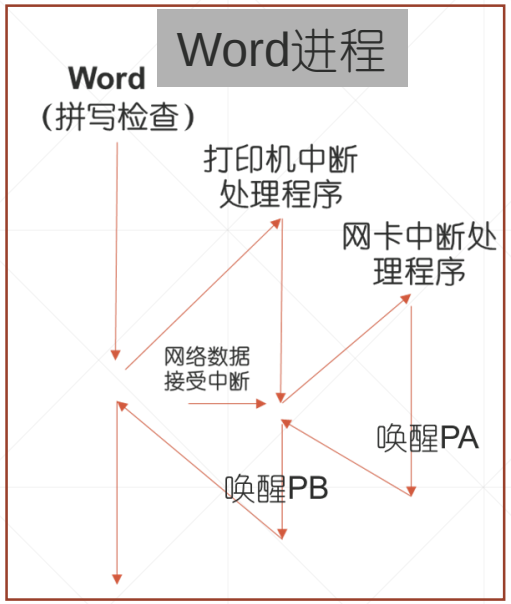


图 a

问：3、执行拼写检查的word进程放弃CPU后，CPU执行的任务序列 ？

答：



收网络包的系统调用 **recv（PA进程）**

{

上半段 ……

sleep（ &接受网络数据的缓存，-60）

下半段 ……

}

操作打印机的系统调用 *** **（PB进程）**

{

上半段 ……

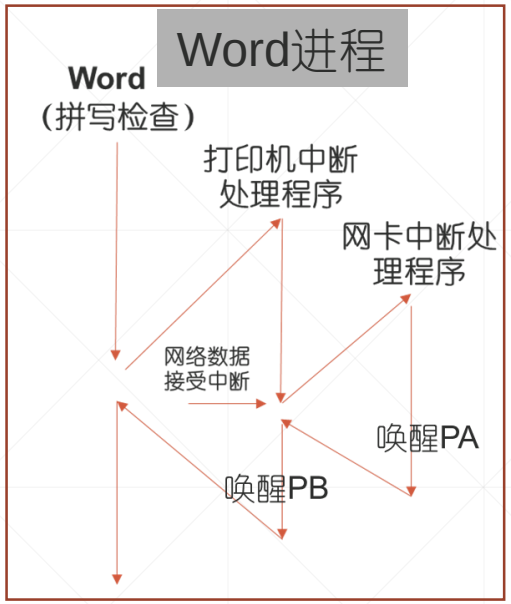
sleep（ &接受网络数据的缓存，20）

下半段 ……

}

1、被中断的现运行进程word（p_pri >=100）完成所有中断处理后，放弃CPU，PA进程（ p_pri < 100）上台

- 调度系统状态：　RunRun == 0，Curpri = -60
- PA进程执行系统调用下半段（用TCP/IP协议栈处理网络包，送用户空间）
- recv系统调用执行完毕，PA执行setpri( ) 恢复 p_pri（>=100）； p_pri >Curpri，Runrun++
- 返回用户态前，PA执行swtch( )放弃CPU。PB进程上台。




```
PA进程，接受网络数据的系统调用
{
    上半段 ……
    sleep（ &接受网络数据的缓存，-60）
    下半段 ……
}
```

```
PB进程，操作打印机的系统调用
{
    上半段 ……
    Sleep(&接受打印机数据的缓存，20)
    下半段 ……
}
```

2、PB执行系统调用，完成后放弃CPU。

3、PA、PB、word进程，被调度到的那个进程 IRET 返回用户态执行应用程序。

4、随后，这3个应用程序（用户态的PA、PB、word进程）轮流使用CPU。

直至某个进程执行新的系统调用入睡。

进程入睡，基本工作过程

1、根据优先数判断进程需要进入高优先权睡眠状态还是低优先权睡眠状态
2、设置睡眠原因
3、修改调度状态
4、设置优先数
5、放弃CPU

备注：进入低优先权睡眠状态之前要做额外处理

chan 睡眠原因，　pri 是优先数

```
void Process::Sleep(unsigned long chan, int pri)
{
    ……
    if ( pri > 0 )
    {      ……
            X86Assembly::CLI();
            this->p_wchan = chan;
            this->p_stat = Process::SWAIT;
            this->p_pri = pri;
            X86Assembly::STI();
      ……Kernel::Instance().GetProcessManager().Swtch();
    }
    else
    {
            X86Assembly::CLI();
            this->p_wchan = chan;
            this->p_stat = Process::SSLEEP;
            this->p_pri = pri;
            X86Assembly::STI();
            Kernel::Instance().GetProcessManager().Swtch();
    }
```

进程唤醒，基本工作过程

- ProcessManager::WakeUpAll(chan) 函数唤醒所有因 p_wchan == chan 而入睡的进程。

```
void ProcessManager::WakeUpAll(unsigned long chan)
{
    for(int i = 0; i < ProcessManager::NPROC; i++)
    {
        if( this->process[i].IsSleepOn(chan) )  // process[i]. p_wchan == chan ?
        {
            this->process[i].SetRun();
        }
    }
}
```

进程唤醒，基本工作过程

- Process::SetRun( )，恢复进程的就绪状态。

```
void Process::SetRun()
{
    ProcessManager& procMgr = Kernel::Instance().GetProcessManager();

    /* this是被唤醒进程的Process对象（结构） */
    this->p_wchan = 0;          // 清除睡眠原因
    this->p_stat = Process::SRUN;    // 变就绪
    if ( this->p_pri < procMgr.CurPri )
    {
        procMgr.RunRun++;
    }
        ……
}
```