# 系统调用完整的执行过程

## Part 1，会睡的系统调用

**情景分析：T 时刻现运行进程 PB 用户态执行图 1 所示应用程序，read 函数。2s 后用户键盘输入字符 a。**

```
main（ ）
{
    ……
    char local ;
    read(0, &local, 1);   // 读标准输入，键盘
    ……
}
```

图 1、读键盘的应用程序 demoSyscall

### 宏观的认识:

1、 PB 执行 read 系统调用，读键盘输入、会入睡。低优先权睡眠，p_stat = SWAIT。
2、 用户输入的 a 字符，由内核送给应用程序。针对本例，read 系统调用将用户键入的字符 a，赋值、送给用户栈中的局部变量 local。
3、 read 系统调用从调用到返回，执行了 2s。其间，几乎所有时间睡眠、等待用户输入。PB 睡觉的 2s，CPU 没闲着，会执行其它进程。

有趣的问题：是不是每次执行程序 demoSyscall，read 系统调用都会执行 2s？

### 调度细节

**时刻 T**

1、 PB 模式切换，用户态进入核心态。
2、 PB 核心态执行 read 系统调用的上半段：读 tty 输入缓存，空的、没有字符可供读取，所以 PB sleep( &tty 输入缓存，TTIPRI ) 入睡放弃 CPU，SWAIT 低优先权睡眠。这里发生了进程切换，CPU 的现运行进程不再是 PB。

**时刻 T+2**

3、 键盘中断是 IO 完成信号，通知系统用户有输入。收到这个异步信号，系统（内核）做

3 件事:

3.1　收数据。读键盘控制器（外设芯片），得用户敲击键盘输入的字符 a；送 tty 输入缓存（这是内核空间中为每个外设配置的缓冲区）。

3.2　tty 有新数据啦，唤醒 PB。

3.3　PB 被调度程序选中执行，读 tty 输入缓存，得字符 a，送用户空间，局部变量 c。read 系统调用完成，PB 返回用户态。注：tty 输入缓存又空了。

**3.1~3.3 是 T+2 时刻，系统发生的与这次 read 系统调用执行相关的中断响应、调度和系统调用过程。**

其中，3.1 和 3.2　是键盘中断处理程序做的。执行键盘中断处理程序的是 T+2 时刻的现运行进程。是谁不重要，总之是被键盘中断的那个进程 PX 核心态执行键盘中断处理程序，唤醒了 PB。

3.3 是 PB 进程自己做的。执行 read 系统调用下半部。这个操作，PB 核心态运行。

**最后，系统调用返回时的调度**

注：系统调用返回前，Trap 会调用 Setpri 计算 PB 执行应用程序时的优先数。RunRun++会触发调度。如果 PB 不是优先级最高的就绪进程，会放弃 CPU，成为就绪进程。恢复用户态现场的操作（下节，现场恢复 2）可能会延迟至下次 PB 被调度程序选中。

总结：本例，系统调用 read 的执行分两半，上半段时刻 T 执行，在入睡前；下半段时刻 T+2 执行，在唤醒后。系统调用执行期间，共有 4 次调度过程。第一次，PB 入睡放弃 CPU。第二次，唤醒之后，PB 被选中执行。第三次，系统调用返回用户态被剥夺。第四次，作为普通就绪态进程被选中执行。

多道系统有完善的状态保护、恢复机制，能够保证历经模式切换和进程切换后，应用程序的执行状态依旧保持连贯。下面观察系统调用执行过程，系统如何保护程序状态。

## 现场保护、恢复细节

1、现场保护 1：时刻 T。

PB 执行系统调用入口函数，将用户态寄存器保存在自己的核心栈底。

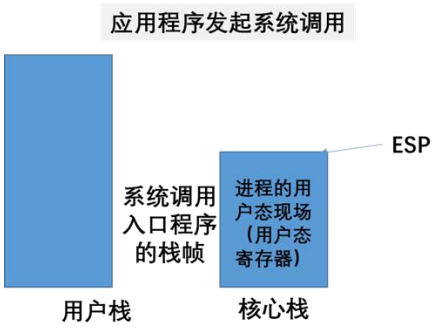


图 2

2、现场保护 2：入睡放弃 CPU。

PB 入睡，sleep( )→swtch( )，将寄存器 ESP、EBP，也就是 swtch 栈帧定位指针，存入自己的 user 结构。PB 成为睡眠态进程

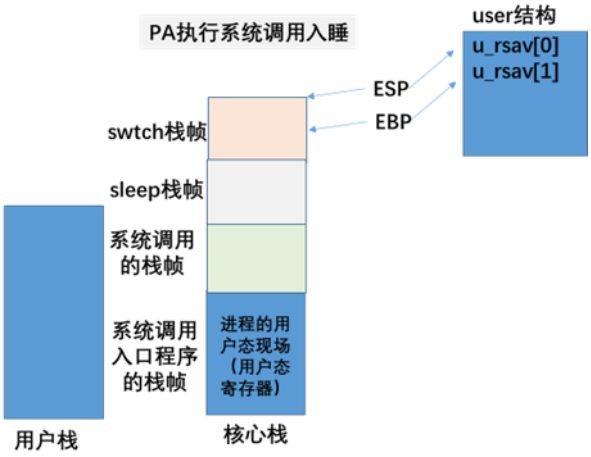


图 3

3、现场恢复 1：时刻 T+2。

PB 被中断处理程序唤醒，调度程序 select 选中后，swtch( )从它的 user 结构中取出时刻 T 保存的 swtch 栈顶定位指针，并赋值 ESP 和 EBP。

Swtch 栈帧的返回地址会引导 PB 从 sleep 函数返回，执行 read 系统调用的下半部。图 4。

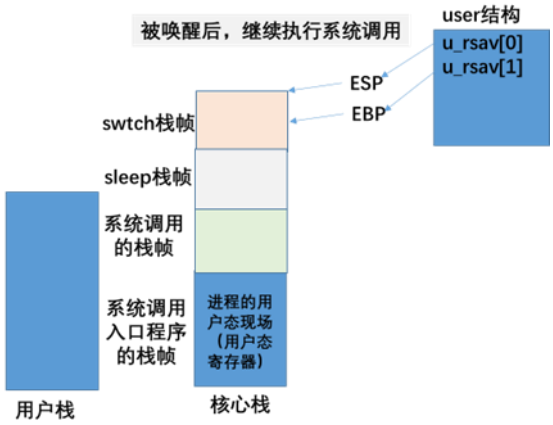


图 4

4、现场恢复 2：read 系统调用返回时，系统调用的入口函数会弹出核心栈底保存的用户态现场。图 5。返回用户态后，PB 从 read 系统调用的钩子函数返回，继续执行应用程序。

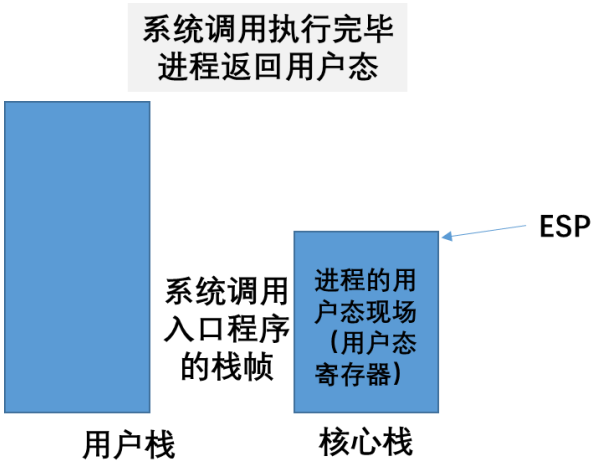


图 5

# Part 2，不会睡的系统调用

getpid( )等系统调用，不会导致进程入睡。执行期间，进程保持运行态。

系统调用执行期间，2 次模式切换。基本不会有进程调度，除非系统调用发生在整数秒，进程时间片用尽。

# Part3，就绪进程：图像和被选中恢复执行的细节

1、 图 4 是第二类就绪进程的图像。这些进程是被唤醒的睡眠进程。被选中运行时，从 Swtch 返回，自 Swtch 栈帧返回地址处的那条指令 X 开始，执行系统调用下半部。

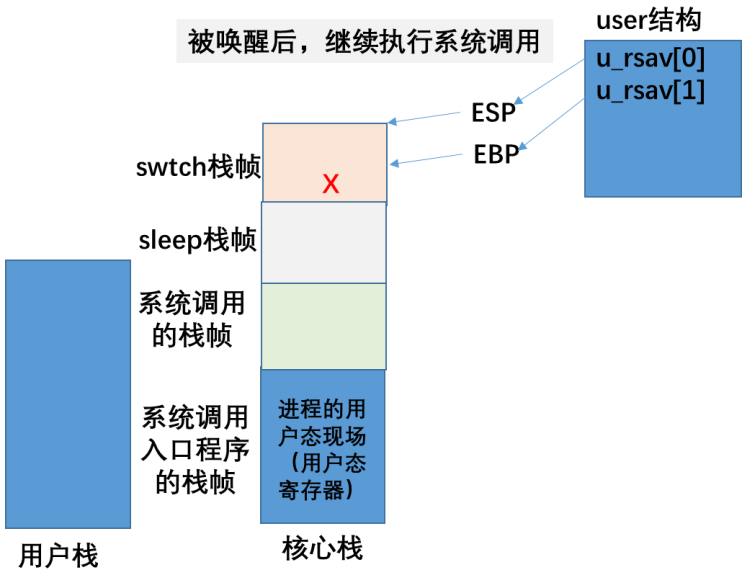


图 4（重复）

如果唤醒的是高优先级睡眠进程，X 是 Sleep 函数的 leave，ret。进程 Sleep 函数返回，继续执行系统调用。

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

        Kernel::Instance().GetProcessManager().Swtch();
  X }
}
```

2、第一类就绪进程，被选中运行时，恢复用户态现场，返回用户态执行应用程序。图 6 是这类进程的图像。

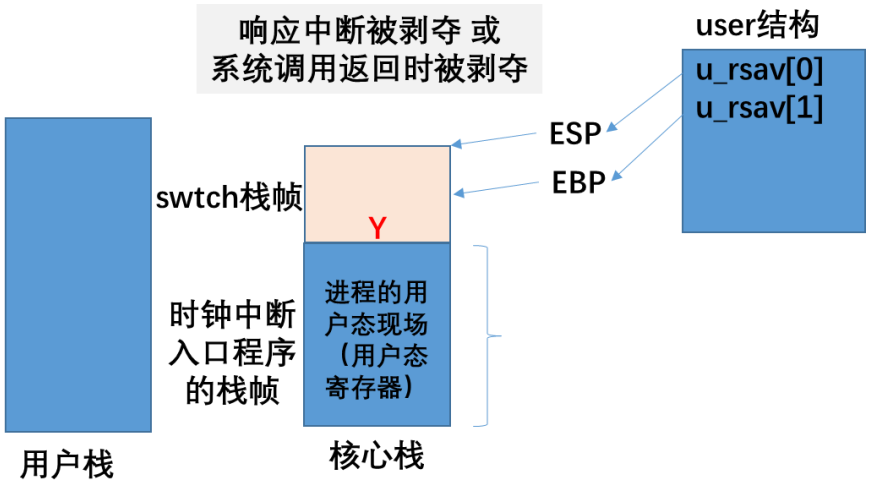


图 6

进程被选中，执行 Swtch 栈帧返回处的指令 Y。这条指令在中断/异常入口函数的这个位置。随后，恢复用户态执行现场，进程返回用户态执行应用程序。

```
void Time::***Entrance()
{
 SaveContext();              /* 保存中断现场 */

 SwitchToKernel();       /* 进入核心态 */

 CallHandler(Time, Clock);          /* 调用时钟中断处理子程序 */

 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*/
        {


      Y：  while(true)
            {
                X86Assembly::CLI();

                if(Kernel::Instance().GetProcessManager().RunRun > 0)
                {
                    X86Assembly::STI();
                    Kernel::Instance().GetProcessManager().Swtch();
                }
                else
                {
                    break;
                }
            }
        }

        RestoreContext();       /* 恢复现场 */

        Leave();                /* 手工销毁栈帧 */

        InterruptReturn();      /* 退出中断 */
    }
```

# Part4，快系统调用、慢系统调用、高优先权睡眠态和低优先权睡眠态

快系统调用，访问磁盘的系统调用。进程如果入睡，SSLEEP 高优先权睡眠。
其余所有系统调用，慢系统调用。进程如果入睡，SWAIT 低优先权睡眠。

快系统调用的执行过程，计算机系统能够完全控制。因为访问的是本地资源，硬盘里的数据。慢系统调用的执行过程，计算机系统完全控制不了。比如，图 1 中，read 系统调用会执行多久？不知道，取决于用户什么时候敲键盘。再有，读网络包的系统调用 recv，何时返回，取决于网络拥塞程度和通信对端另一台计算机。

快系统调用和慢系统调用的处理过程是不一样的。最显著的区别在于对信号的处理方式。快系统调用，如其名很快很快能够运行结束，执行过程不受信号影响；完成后再处理信号不迟。而慢系统调用就不一样了，它，很有可能会睡很久，等它完成再处理信号，太晚了，会拖垮

整个分布式系统。

因此，为了提高进程响应信号的速度，慢系统调用入睡前会检查信号，如果有收到，不睡了，系统调用立即返回用户态处理信号；另外，信号会终止低优先权睡眠状态，收到信号的进程不再继续执行系统调用，也是立即返回用户态去处理信号。这就是 Unix V6++，sleep 函数中低优先权睡眠状态分支中处理信号的 2 处，如图 7。

```
void Process::Sleep(unsigned long chan, int pri)
{
    ……
    if ( pri > 0 )
    {
        if ( this->IsSig() )
        {
        /* return 确保 aRetU()跳回到 SystemCall::Trap1()之后立刻执行 ret 返回指令 */
            aRetU(u.u_qsav);
            return;
        }
        X86Assembly::CLI();
        this->p_wchan = chan;
        this->p_stat = Process::SWAIT;
        this->p_pri = pri;
        X86Assembly::STI();
    ……
        Kernel::Instance().GetProcessManager().Swtch();

        /* 被唤醒之后再次检查信号 */
        if ( this->IsSig() )
        {
        /* return 确保 aRetU()跳回到 SystemCall::Trap1()之后立刻执行 ret 返回指令 */
            aRetU(u.u_qsav);
            return;
        }
    }
    ……
}
```

图 7

被信号打断的系统调用是失败的，因为它没有成功 IO 数据给应用程序，错误号 ErrorNo = EINTR。一般，涉及 IO 操作的系统调用，被信号打断后，系统会自动重启；其余被信号打断的慢系统调用，比如 sleep(seconds)，无需自动重启。这就是信号能够提前终止应用程序定时器的原因。