Unix V6++系统调用

同济大学计算机系操作系统讲义　　邓蓉　　2023-11-15

系统调用是 0x80h（128#）中断。应用程序执行系统调用，促使执行它的进程陷入内核，执行语义确切的内核服务逻辑。

每个系统调用独用一个系统调用号，用户空间有一个钩子函数，内核空间有一个入口函数。钩子函数在系统的标准函数库，Unix V6++系统把它们定义在 src/lib 目录下，分布在多个 C 源文件里。Unix V6++采用静态链接技术，gcc 将钩子函数和应用程序源文件链在一起，生成可执行程序。Unix V6++采用单内核结构，所以系统调用入口函数和其它内核函数链接在一起，是内核（kernel 程序）中的一个普通子函数。

下面，我们以 20 号系统调用 getpid( )为例，逐步展开，介绍系统调用的源代码和具体的执行过程。最后，用一张简图总结系统调用与调度子系统之间的关系。

1、用户空间的钩子函数

应用程序调用钩子函数执行系统调用。代码 1，第 1 行。

系统调用成功，返回非负值。本例，getpid 系统调用获得现运行进程的 pid 号。

系统调用失败，返回 -1。全局变量 errno 是出错码，钩子函数用它来存放系统调用的出错信息。正整数表示出错，0 表示执行成功。

```
#include <stdio.h>
#include <stdlib.h>    // src/lib/include/stdlib.h。定义有 errno

int main1(int argc, char* argv[])
{
1:    int i = getpid( );
2:    if( i >= 0 )
3:          printf( " My pid is %d\n ", i );
4:    else
5:          printf( " Fail System Call ! return value = %d，  errno = %d \n ", i , errno );
}
```

代码 1：应用程序 getpid.c

钩子函数为系统调用准备参数、引发中断，最后接收系统调用的返回结果。本例，定义在 src/lib/src/sys.c 中的库函数 getpid( )，代码如下：

```
int getpid()
{
1:    int res;
2:    __asm__ volatile ( "int $0x80":"=a"(res):"a"(20) ) ;
3:    if ( res >= 0 )
4:          return res ;
```

```
5:    else
6:    {
7:          errno = -res ;
8:          return -1;
9:    }
}
```

代码 2：钩子函数 getpid( )与应用程序 getpid.c 链接在一起，运行在用户态为系统调用准备参数、引发中断，最后接收系统调用的返回结果。

- 语句 2。输入部 "a"(20)，将 20 写入 EAX 寄存器，这是在为内核提供系统调用号；汇编指令部 int 0x80，向 CPU 发送 0x80 号中断请求，激活内核执行系统调用；系统调用执行完毕后，CPU 返回用户态，执行输出部 "=a"(res) 的操作，取出内核存放在 EAX 寄存器中的系统调用返回值，赋给变量 res。
- if 分支，系统调用成功返回，返回值是进程的 pid 号。else 分支将出错码赋给全局变量 errno，返回 -1。

上面介绍的系统调用 getpid 没有入口参数。下面我们看执行 open 系统调用的内联汇编语句，这是 5#系统调用，有 2 个参数：pathname 和 mode。执行系统调用前，钩子函数将入口参数存入寄存器 EBX 和 ECX。

```
__asm__volatile("int $0x80":"=a"(res):"a"(5),"b"(pathname),"c"(mode));
```

所有系统调用使用寄存器的方式是一致的。

- 执行前 EAX 存放系统调用号，EBX、ECX、EDX、ESI 和 EDI 分别存放系统调用的 5 个参数。
- 执行完毕后，内核会将系统调用的返回值存入 EAX。

以下是内核中，系统调用的实现细节。

2、 类 SystemCall

```
class SystemCall
{
public:
    /*系统调用入口表的大小*/
    static const unsigned int SYSTEM_CALL_NUM = 64;

public:
    static void SystemCallEntrance();
    static void Trap(struct pt_regs* regs, struct pt_context* context);
    static void Trap1(int (*func)());

private:
    /* 空函数，0#系统调用不存在 */
```

```
    static int Sys_NullSystemCall();

    /*    1 = rexit    count = 0  */
    static int Sys_Rexit();

    /*    2 = fork     count = 0  */
    static int Sys_Fork();

    /*    3 = read     count = 2  */
    static int Sys_Read();

    ……  已实现的，共计 48 个系统调用

private:
    /* 系统调用入口表 */
    static SystemCallTableEntry m_SystemEntranceTable[SYSTEM_CALL_NUM];
};
```

代码 3：SystemCall 类定义，src/include/SystemCall.h

类 SystemCall 实现系统调用。代码 3。其中，

- SystemCallEntrance( )是系统调用总入口函数；它的起始地址登记在 IDT[0x80]。
- Trap( )是系统调用处理函数，所有系统调用都从这里进。该函数根据用户提供的系统调用号查阅系统调用入口表：m_SystemEntranceTable，调用具体的系统调用入口函数。
- Trap1( )是 Trap( )的辅助函数，系统调用执行前，设置一个长跳转的返回点。
- 其余，以 Sys_为前缀的，是系统已经实现的所有系统调用的入口函数。对的，每个系统调用的具体操作从这里开始。如果要添加新的系统调用，这里要加定义。
- 系统调用入口表 m_SystemEntranceTable，有 SYSTEM_CALL_NUM = 64 个表项。每个表项是一个类型为 SystemCallTableEntry 的元素，定义有该系统调用的参数数量和入口函数地址。如果添加新的系统调用，这里要加表项。

```
struct SystemCallTableEntry
{
    unsigned int   count;    // 系统调用的参数个数
    int   (*call)();         // 系统调用入口函数的地址
};
```

这是 Unix V6++系统调用表的值，可以一窥 Unix 核心系统调用[2]。没有 0#系统调用。**1#exit 进程终止；2#fork 创建子进程；3#read 读文件；4#write 写文件；5#open 打开文件；6#close 关闭文件；7#wait 父进程等待子进程终止，应用程序可以用它实现任务之间的前驱后继关系；8#creat 创建新文件；9#link 给已有文件一个新名字；10#unlink 删除文件；11# exec 进程加载应用程序、从 main 程序入口开始执行；12#chdir 改变进程的当前工作目录；13#time get 系统时间，就是计算机系统内部时刻，通常又叫做墙上时间(wall clock time)；14#mknod 添加外设，为其创建一个特殊文件（源码有误，u_arg[1]应该改为 u_arg[0]）；15#chmod 改变文件的访问权限；16#chown 改变文件所有者；17#brk 改变进程图像的尺寸，支持库函数 malloc 和 free；18#stat get 未打开文件的属性；19#seek 移**

动文件读写指针；20# getpid，获取进程 pid；21#mount，安装子文件系统，比如 U 盘上的文件系统；22#unmount，使用完毕卸载子文件系统；23#setuid 设置进程 uid，用于验明用户身份后设置 shell 进程的 uid；24#getuid 获得有效用户 uid 和真实用户 uid；25#stime 超级用户设置系统时间；26#ptrace，还没写 ；27#nosys ；28#fstat get 一个使用过程中的文件（打开文件）的属性；29#trace 师弟们在玩；30# 没这个系统调用；31#stty 过时了【1】；32#gtty 过时了【1】；33#nosys；34#nice 设置进程静态优先数的基础值；35#sleep(seconds) 进程睡眠 seconds 秒；36#sync 同步文件系统更新，将内存中所有的改动刷回磁盘；37#kill 向目标进程发信号，最常用的命令 kill -9 pid：杀死进程 pid；38#switch，获得系统中总的进程切换次数（POSIX 里是否定义存疑）；39#pwd，get 进程的当前工作目录；40# 没这个系统调用；41#dup，分配一个新的文件描述符去访问一个已经打开的文件；42#pipe，建立一个进程通信管道；43#times，get 进程用户态执行时长，核心态执行时长；44#prof，get 程序剖分图，观察各个指令区间的执行时长，有利于优化系统时确定程序瓶颈；45# 没这个系统调用；46#setgid，拥有超级用户权限的进程设置自己的 gid 和 real gid；47#getgid，get 进程的 gid 和 real gid；48#signal，为进程设置信号处理方式。 49~63，系统调用号空着，留给开发者扩展 Unix API。

【1】设置、get PDP-11 tty 终端端口属性，比如工作波特率，擦除字符……stty 会清空字符缓存中的数据，之后重置 tty 端口属性。

【2】不同的系统，支持数量不等的系统调用。相同功能的系统调用，在不同的系统中编号不同。所以，一般而言，为一个系统开发的应用无法运行在另一个系统上。windows 应用无法运行在 Unix/Linux 系统（反过来也一样），最重要的原因就在于此。为了提高应用程序的可移植性，Unix/Linux 社区制定 POSIX 标准。该标准严格定义所有系统调用，包括每个系统调用的编号，参数列表，系统调用的功能语义和每个参数的具体含义。

符合 POSIX 标准的内核和应用程序相互兼容。为任何一个平台开发的应用程序原则上可以运行在任何符合 POSIX 标准的平台。这不仅可以提高应用程序的可移植性，还能为融入生态的内核发展提供足够的灵活度：只要遵循接口规范，内核架构和实现细节不受任何限制。该标准的出现极大促进了系统核心技术的发展，催生了 Linux 和 MacOS 等众多开源、闭源产品，为 Unix 发展奠定了坚实基础。

Unix V6++的核心系统调用基本符合 POSIX 标准，请读者对照参考链接：https://www.jianshu.com/p/8da8cbcce815

```
SystemCallTableEntry SystemCall::m_SystemEntranceTable[SYSTEM_CALL_NUM] =
{
    { 0, &Sys_NullSystemCall },         /* 0 = indir   */
    { 1, &Sys_Rexit },                  /* 1 = exit    */
    { 0, &Sys_Fork    },                /* 2 = fork    */
    { 3, &Sys_Read    },                /* 3 = read    */
    { 3, &Sys_Write   },                /* 4 = write   */
    { 2, &Sys_Open    },                /* 5 = open    */
    { 1, &Sys_Close   },                /* 6 = close   */
    { 1, &Sys_Wait    },                /* 7 = wait    */
    { 2, &Sys_Creat   },                /* 8 = creat   */
    { 2, &Sys_Link},                    /* 9 = link    */
    { 1, &Sys_UnLink},                  /* 10 = unlink*/
    { 3, &Sys_Exec    },                /* 11 = Exec   */
    { 1, &Sys_ChDir   },                /* 12 = chdir  */
    { 0, &Sys_GTime   },                /* 13 = time   */
```

```
{ 3, &Sys_MkNod },              /* 14 = mknod     */
{ 2, &Sys_ChMod  },             /* 15 = chmod     */
{ 3, &Sys_ChOwn  },             /* 16 = chown     */
{ 1, &Sys_SBreak},              /* 17 = sbreak    */
{ 2, &Sys_Stat },               /* 18 = stat      */
{ 3, &Sys_Seek     },           /* 19 = seek   */
{ 0, &Sys_Getpid},              /* 20 = getpid    */
{ 3, &Sys_Smount  },            /* 21 = mount     */
{ 1, &Sys_Sumount       },      /* 22 = umount    */
{ 1, &Sys_Setuid    },          /* 23 = setuid*/
{ 0, &Sys_Getuid    },          /* 24 = getuid    */
{ 1, &Sys_Stime          },     /* 25 = stime */
{ 3, &Sys_Ptrace    },          /* 26 = ptrace    */
{ 0, &Sys_Nosys     },          /* 27 = nosys */
{ 2, &Sys_FStat     },          /* 28 = fstat   */
{ 1, &Sys_Trace     },          /* 29 = trace   */
{ 0, &Sys_NullSystemCall },     /* 30 = smdate; inoperative */
{ 2, &Sys_Stty },               /* 31 = stty    */
{ 2, &Sys_Gtty},                /* 32 = gtty    */
{ 0, &Sys_Nosys     },          /* 33 = nosys */
{ 1, &Sys_Nice      },          /* 34 = nice    */
{ 1, &Sys_Sslep     },          /* 35 = sleep   */
{ 0, &Sys_Sync      },          /* 36 = sync    */
{ 2, &Sys_Kill   },             /* 37 = kill      */
{ 0, &Sys_Getswit},             /* 38 = switch*/
{ 1, &Sys_Pwd},                 /* 39 = pwd     */
{ 0, &Sys_Nosys     },          /* 40 = nosys */
{ 1, &Sys_Dup},                 /* 41 = dup     */
{ 1, &Sys_Pipe      },          /* 42 = pipe    */
{ 1, &Sys_Times     },          /* 43 = times */
{ 4, &Sys_Profil},              /* 44 = prof    */
{ 0, &Sys_Nosys     },          /* 45 = nosys */
{ 1, &Sys_Setgid},              /* 46 = setgid*/
{ 0, &Sys_Getgid},              /* 47 = getgid*/
{ 2, &Sys_Ssig },               /* 48 = signal */
{ 0, &Sys_Nosys     },          /* 49 = nosys */
{ 0, &Sys_Nosys     },          /* 50 = nosys */
{ 0, &Sys_Nosys     },          /* 51 = nosys */
{ 0, &Sys_Nosys     },          /* 52 = nosys */
{ 0, &Sys_Nosys     },          /* 53 = nosys */
{ 0, &Sys_Nosys     },          /* 54 = nosys */
{ 0, &Sys_Nosys     },          /* 55 = nosys */
{ 0, &Sys_Nosys     },          /* 56 = nosys */
{ 0, &Sys_Nosys     },          /* 57= nosys  */
```

```
    { 0, &Sys_Nosys    },                /* 58 = nosys */
    { 0, &Sys_Nosys    },                /* 59 = nosys */
    { 0, &Sys_Nosys    },                /* 60 = nosys */
    { 0, &Sys_Nosys    },                /* 61 = nosys */
    { 0, &Sys_Nosys    },                /* 62 = nosys */
    { 0, &Sys_Nosys    },                /* 63 = nosys */
};
```

3、陷入内核后，系统调用的执行过程

响应 0x80 号中断请求，调用 Trap( )程序，为具体的系统调用准备参数，随后调用 Trap1( )程序，这个程序设置一个长跳转点之后调用中断入口程序 Sys_***，开始真正的系统调用执行过程。

复杂的系统调用过程，子程序嵌套调用的程度会比较深，逐级传入入口参数、送回系统调用返回值的方法会非常麻烦，所以，Unix V6++的设计是使用全局变量：u_arg 数组存放系统调用的入口参数，u_ar0 指针指向的内存单元存放系统调用的返回值。

先看 Trap( )程序的源代码和执行过程。依然以 getpid 系统调用为例。

第一步：准备。

```
1:  User& u = Kernel::Instance().GetUser();
    /* u_ar0 定位栈帧中的用户态寄存器。系统调用出错码清 0 */
2:  u.u_ar0 = &regs->eax;
3:  u.u_error = User::NOERROR;
```

u_ar0 是全局指针变量，指向存放系统调用返回值的内存单元。Unix V6++用 EAX 寄存器存放系统调用的返回值，所以 u_ar0 = &regs->eax。u_error 是系统调用出错码。系统调用执行前，对它清 0。

第二步：确定系统调用号和入口表项。

```
    /* callp 定位系统调用对应的入口表项 */
4:  SystemCallTableEntry *callp = &m_SystemEntranceTable[regs->eax];
```

regs->eax 的值是系统调用号，Trap 用它作为下标查系统调用入口表。

getpid 系统调用，这个值是 20。

第三步：为系统调用准备参数。

```
    /* 将系统调用参数存入 u.u_arg 数组 */
5:  unsigned int * syscall_arg = (unsigned int *)&regs->ebx;
6:  for( unsigned int i = 0; i < callp->count; i++ )
7:      u.u_arg[i] = (int)(*syscall_arg++);
```

Unix V6++利用 user 结构中的 u_arg 数组存放系统调用参数。第 5~7 行，将核心栈中保存的应用程序传入参数（EBX、ECX，EDX，ESI 和 EDI）转存进 u_arg 数组。具体传送的参数数量，记录在系统调用入口表项：SystemCallTableEntry 结构里。

getpid 系统调用没有入口参数，这一步跳过。

第四步：启动系统调用入口程序

```
    /* 调用系统调用处理子程序，如 fork(), read()等等 */
10：Trap1(callp->call);        /* 执行期间，若被信号打断，系统调用会置出错码 User::EINTR */
```

调用 Trap1( )函数启动系统调用入口程序。Trap1( )函数的细节后面介绍。

我们的例子，getpid 系统调用传入 Trap1( )的实参是入口函数 Sys_Getpid( )的地址。

第五步：执行系统调用

系统调用执行期间，

- 从 u_arg 数组取用系统调用的参数
- 将返回值存入 u_ar0 指向的用户态寄存器 EAX。
- 将出错码写入全局变量 u_error。非 0 值是系统调用的出错码，0 表示系统调用成功。

（1） 不会导致进程入睡的系统调用

getpid 系统调用的入口是 Sys_Getpid( )。这个系统调用不会导致进程入睡，它很简单，读取现运行进程的 pid 存入系统调用栈帧，完成后就可以返回了。

```
int SystemCall::Sys_Getpid()
{
    User& u = Kernel::Instance().GetUser();
    u.u_ar0[User::EAX] = u.u_procp->p_pid;
    return 0; /* GCC likes it！这个返回值没有意义。*/
}
```

□

（2）可能导致进程入睡的系统调用

复杂的系统调用可能导致进程入睡。入睡的进程，被唤醒后会继续执行系统调用。系统调用是内核任务，其优先级理应高于应用程序。为了做到这点，Unix V6++为每种入睡原因设置一个小于 100 的固定的优先数值，见表 1。入睡时，sleep 函数把它写入现运行进程的 p_pri。唤醒后，进程用这个优先数竞争 CPU。

优先数越小，进程优先级越高。0#进程优先数最小，唤醒后能立即使用 CPU。

表 1：UNIX V6++中定义的睡眠进程拥有的优先数

序号	入睡原因	优先数值
1	0#进程等待执行换入、换出操作	PSWP(-100)
2	进程等待磁盘空闲资源分配释放的权限或内存inode的使用权限	PINOD (-90)
3	磁盘输入输出操作	PRIBIO (-50)
4	进程等待exec系统调用的执行资格	EXPRI (-1)
5	管道输入输出操作	PPIPE (1)
6	终端输入操作（scanf）	TTIPRI (10)
7	终端输出操作（printf）	TTOPRI (20)
8	父进程等待子进程终止	PWAIT (40)
9	进程设置定时器之后入睡	PSLEP (90)

10	不是睡眠原因，是应用程序优先数的基值	PUSER (100)

以 sleep 函数为分界点，可以把系统调用分成上半段和下半段。调用 Sleep 函数之前是上半段，之后是下半段。如图：

```
系统调用( )
{
    上半段;
    Sleep（睡眠原因，入睡优先数）;
    下半段;
}
```

Sleep 函数会改变 p_pri，所以，下半段执行时，进程的优先级会提升至入睡优先数。Unix V6 的这种设计等价于为每个会入睡的系统调用设置了一个优先权，系统调用使用的资源，竞争越强烈，优先权越高；影响面越广，优先权越高。如此，并发执行的各个系统调用，就可以在保证内核正常运转的前提下，以提高系统运行效率为目标等级森严地排队运行。下节介绍时钟中断，我们会看到一个综合的例子。

有关系统调用执行的内容就先介绍到这里。

现在系统调用执行完毕，我们返回 Trap( )函数。

第六步：Trap( ) 返回用户态

系统调用返回用户态前，Trap( )需要做 4 方面的处理。

（1）第 11 行 判断系统调用执行期间，是否出现信号打断低优先权睡眠过程的情况。如果有，u_intflg 标识是 1，系统调用并没有完成既定任务，这一点需要明确知会系统和应用程序：第 12 行将出错码 EINTR 赋给 u_error 变量。

```
    /* u.u_intflg == 0，系统调用正常结束。
     * u.u_intflg != 0，慢系统调用被信号打断。设置出错码 User::EINTR。*/
11：if ( u.u_intflg != 0 )
12：        u.u_error = User::EINTR;
```

getpid 系统调用不会导致进程入睡，这一步跳过。

（2）第 13、14 行。为出错的系统调用准备返回值。

```
    /* 用户态寄存器 EAX 用来存放系统调用的返回值。如果出错，系统调用错误号取负值存入 EAX */
13：if( User::NOERROR != u.u_error )
14：        regs->eax = -u.u_error;
```

getpid 系统调用不会失败（因为进程不可能没有 pid 的）。这一步跳过。

（3）第 15、16 行。信号处理。

```
    /* 如果系统调用执行期间，进程收到信号。无论其是否被信号打断，返回用户态前，进行信号处理 */
```

```
15:  if ( u.u_procp->IsSig() )
16:      u.u_procp->Psig(context);
```

getpid 系统调用执行期间，若收到信号，在这里处理。

（4）第 17 行。调用 Setpri( ) 计算、设置现运行进程返回用户态之后的优先数。

```
    /* 计算、恢复现运行进程返回用户态之后的优先数。执行系统调用期间临时提高的优先权得以复原 */
17:  u.u_procp->SetPri();
```

- 如果系统调用曾经导致进程入睡，临时提高的进程优先权得以复原，至值>=100，与进程返回用户态后执行应用程序的身份相匹配。
- getpid 系统调用不会导致进程入睡，所以进程的优先权基本不会得到提升。这种情况下，Setpri( )，更多地是在考量现运行进程的连续运行时长。如果时间片用完，RunRun++，让出 CPU。

Trap( )程序和源代码分析

```
void SystemCall::Trap(struct pt_regs* regs, struct pt_context* context)
{
1:   User& u = Kernel::Instance().GetUser();

     /* u_ar0 定位栈帧中的用户态寄存器。系统调用出错码清 0 */
2:   u.u_ar0 = &regs->eax;
3:   u.u_error = User::NOERROR;

     /* callp 定位系统调用对应的入口表项 */
4:   SystemCallTableEntry *callp = &m_SystemEntranceTable[regs->eax];

     /* 将系统调用参数存入 u.u_arg 数组 */
5:   unsigned int * syscall_arg = (unsigned int *)&regs->ebx;
6:   for( unsigned int i = 0; i < callp->count; i++ )
7:       u.u_arg[i] = (int)(*syscall_arg++);

     /* open，creat 系统调用需要使用 u.u_dirp 指针，指向 文件路径名 参数 */
8:   u.u_dirp = (char *)u.u_arg[0];
     /* exec()系统调用需要使用 u_arg[4]，访问核心栈硬件保护现场 */
9:   u.u_arg[4] = (int)context;

     /* 调用系统调用处理子程序，如 fork(), read()等等 */
10:  Trap1(callp->call);      /* 执行期间，若被信号打断，系统调用会置出错码 User::EINTR */

     /* u.u_intflg == 0，系统调用正常结束。
      * u.u_intflg != 0，慢系统调用被信号打断。设置出错码 User::EINTR。*/
11:  if ( u.u_intflg != 0 )
```

```
12:         u.u_error = User::EINTR;

    /* 用户态寄存器 EAX 用来存放系统调用的返回值。如果出错，系统调用错误号取负值存入 EAX */
13:  if( User::NOERROR != u.u_error )
14:         regs->eax = -u.u_error;

    /* 如果系统调用执行期间，进程收到信号。无论其是否被信号打断，返回用户态前，进行信号处理 */
15:  if ( u.u_procp->IsSig() )
16:         u.u_procp->Psig(context);

    /* 计算、恢复现运行进程返回用户态之后的优先数。执行系统调用期间临时提高的优先权得以复原 */
17:  u.u_procp->SetPri();
}
```

辅助函数 Trap1( )

考虑到慢系统调用可能会被信号打断，Trap1( )在执行系统调用前，用 SaveU 宏为进程设置了一个长跳转返回点：Trap 的第 11 行，系统调用的返回点。(a)子图。

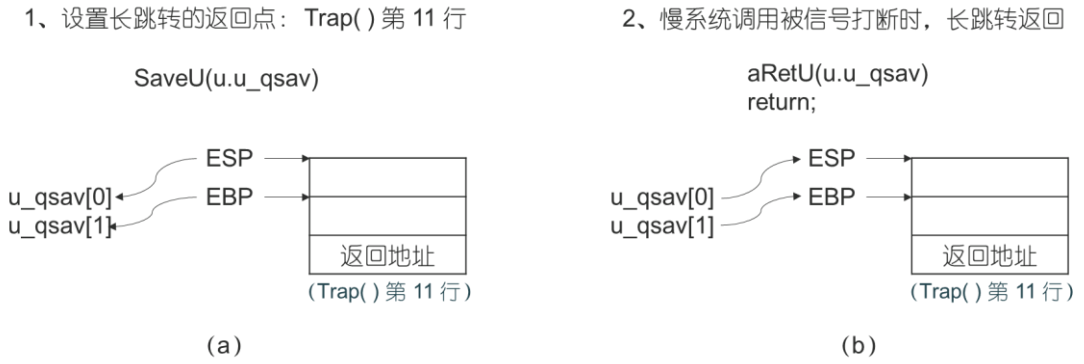
被信号唤醒的慢系统调用，执行 aRetU 宏从 u_qsav 中恢复 Trap1 栈帧（ESP 和 EBP）。随后，在该栈帧内执行 return 语句，长跳转返回 Trap( )函数、返回用户态执行信号处理函数，(b)子图。

如此设计，低优先权睡眠不会耽搁进程执行信号处理程序。好棒！ □

void SystemCall::Trap1(int (*func)())

```
{
1:   User& u = Kernel::Instance().GetUser();

2:   u.u_intflg = 1;          //设置长跳转标识
3:   SaveU(u.u_qsav);         //设置长跳转的返回点：Trap1 的第 11 行
4:   func();                  //调用系统调用处理函数
5:   u.u_intflg = 0;          //系统调用成功，清理长跳转标识
}
```



图：系统调用执行前设置长跳转返回点。

4、返回用户态后

回到钩子函数引发系统调用的那条内联汇编语句（如果系统调用被信号打断，要先跑完信号处理程序），执行输出部，从 EAX 寄存器中取出内核送回的系统调用返回值，将正确的执行结果返回给用户 或者 返回-1 表示系统调用失败，errno 装出错码。

最后返回执行系统调用的应用程序。返回值-1 表示系统调用失败，如果出错原因是被信号打断（errno==EINTR），根据需要重启系统调用。一般，涉及 IO 操作的系统调用要重启；其余系统调用，比如设置定时器的系统调用 sleep，不需要。

应用程序中涉及重启系统调用的代码片段如下，以读取网络链接的 Linux 应用为例：

```
while ( (count = read ( fd, &array, 4096)) <= 0 )
{    // 未能成功读取数据
     if ( count == -1 )
     {
          if ( errno == EINTR )  // 原因是（1）被信号打断。
               continue ;    // 重启系统调用
          else
               return -1 ;  //  其它原因，系统调用失败，返回 -1
     }
     else
          return 0 ;        // 原因是（2）通信对端关闭链接。EOF，后面没数据了，返回 0，read 系统调用成功返回。
}
return count ;              //      成功读取网络数据，返回读入的字节数。
```

总结：系统调用是内核与应用程序的接口。入口参数来自核心栈底系统调用帧，是用户态寄存器的值。

（1）系统调用与普通子程序调用有何不同？

普通子程序调用。调用者和被调用者属于同 1 个程序，32 位编译器用 call 指令调用子程序，用实参压栈的方式传递参数，用 ret 指令实现子程序返回。子程序调用返回过程中，除 ESP、EBP 和 EIP 之外，其余寄存器值不变，开销很小。

系统调用。调用者和被调用者分属 2 个不同的程序，它们运行在不同的处理器特权级，使用 2 个不同的堆栈，无法用实参压栈的方式传递参数。Unix V6++实现系统调用的方式如下，应用程序执行指令 int 0x80、借由中断硬件调用内核子程序，用寄存器，借由全局变量 u_arg 数组向后者传参。系统调用执行期间，需要保存、恢复整套用户态寄存器，经常还会引发进程切换，相较普通子程序调用，开销巨大。

为了减小系统调用开销，Unix 系统标准库函数在用户空间设置数据缓存区。对读操作，每执行一次系统调用，驱动内核送入大块数据，填满用户缓冲区供应用程序慢慢享用；执行写操作时，应用程序可以慢慢向缓存输出，待缓存集满，一次输出全部数据。这种方式可以平摊系统调用开销，减少 IO 操作次数。参见习题。

（2）系统调用相关的调度过程

进程执行系统调用期间，如果需要入睡，执行 sleep( )主动放弃 CPU，完成运行→阻塞的状态变迁。这是第一次调度。等待事件发生后，也就是睡眠原因解除后，睡眠进程会被

现运行进程唤醒。唤醒后的进程就绪，并不能立即执行。需要和其它可能存在的需要执行系统调用下半段的进程一起，在现运行进程即将返回用户态前夕得到运行机会。这是第二次调度。

系统调用执行完毕，进程返回用户态前夕可能会被抢占，将 CPU 让给同级或低优先级的系统调用，抑或是高优先级的其它应用程序。发生运行→就绪的状态变迁。这是可能的第 3 次调度。被抢占的系统调用，下次进程得到运行机会时，恢复现场，返回用户态。这是可能的第 4 次调度。

getpid 系统调用执行时，进程经历几次调度?

一般情况下，不会经历调度。执行系统调用期间，现运行进程如果时间片到，2 次调度。