

操作系统 第二阶段 调度

二、案例学习 Unix V6++ 的中断和调度子系统

3、系统调用与异常

Part 1

系统调用

- 系统调用是为内核保留的一个中断。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。Unix V6++和Linux，中断号是0x80。
- 系统调用是软件中断。是CPU执行int指令或syscall指令向自己发送的中断请求。 Unix V6++和Linux，用int 0x80指令。
- 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。
 - 用户空间的钩子函数在标准C库。Unix V6++，src/lib/Lib_V6++.a；源代码在src/lib/src。
 - 内核空间的入口函数，

Unix V6++在 src/interrupt/SystemCall.cpp，统一定义为：

```
int SystemCall::Sys_*****()
```

入口地址，登记在系统调用入口表，参见 src/include/SystemCall.h 和 src/interrupt/SystemCall.cpp。

```
class SystemCall
{
    .....
    static const unsigned int SYSTEM_CALL_NUM = 64;
    static SystemCallTableEntry m_SystemEntranceTable[SYSTEM_CALL_NUM];
}
```

钩子函数

- 应用程序调用钩子函数执行系统调用。
- 钩子函数为系统调用 (1) 准备系统调用号和入口参数 (2) 引发中断 (3) 接收系统调用的返回结果。
 - 执行成功, 返回非负值。
 - 执行失败, 返回 -1。
 - 每个应用程序有一个全局变量 `errno`, 是错误码, 钩子函数用它存放系统调用的报错信息。正整数表示出错, 0 表示执行成功。
- 运行在 i386 架构中的操作系统。陷入内核前, 用 `EAX` 寄存器存放系统调用号, 其余通用寄存器存放系统调用的入口参数 (`Unix V6++`, `EBX`, `ECX`, `EDX`, `ESI` 和 `EDI`) ; 执行完毕, `EAX` 存放系统调用的返回值。



例1

20#系统调用 getpid

```
#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( );  // 应用程序执行getpid( )，获取执行该程序的进程的pid号
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 );
}
```

```
int getpid()
{
1:  int res;
2:  __asm__ __volatile ( "int $0x80":"=a"(res):"a"(20) ); // 负责执行系统调用的内联汇编函数
3:  if ( res >= 0 )
4:      return res;
5:  else
6:  {
7:      errno = -res ;
8:      return -1;
9:  }
}
```

2、指令部：
陷入内核

1、输入部：eax =
getpid系统调用号20

3、输出部：
系统调用的返回值（进程pid#）→ eax



例2: read系统调用的钩子函数

```
int main( int argc, char* argv[] )
{
    if(argc!=3)
        printf("Usage: copy oldfile newfile\n");
    int oldFile = open(argv[1],O_RDONLY);
    int newFile = open(argv[2],O_WRONLY|O_CREAT,S_IRUSR|S_IWUSR);

    char c;
    while( read(oldFile,&c,1) == 1 )
        write(newFile,&c,1);

    printf("Copy Done\n");

    close(oldFile);
    close(newFile);
}
```

```
int read(int fd, char* buf, int nbytes)
```

```
{
```

```
    int res;
```

```
    __asm__ __volatile__ ("int $0x80":"=a"(res):"a"(3),"b"(fd),"c"(buf),"d"(nbytes));
```

```
    if ( res >= 0 )
```

```
        return res;
```

```
    return -1;
```

```
}
```

2、陷入内核

1、输入部: **eax** = read系统调用号3
ebx = 第1个参数, 要访问的文件oldFile
ecx = 第2个参数, &c
edx = 第3个参数, 1

3、输出部:
read系统调用读入的字节数 → **eax**

内核：系统调用入口函数

```
void SystemCall::SystemCallEntrance()
{
    SaveContext();
    SwitchToKernel();
    CallHandler(SystemCall, Trap);

    例行调度：
    if(RunRun)
        Swtch( );

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

pt_regs
pt_context
GS
FS
DS
ES
EBX
ECX
EDX
ESI
EDI
EBP
EAX
SystemCall() 局部变量区
old ebp
EIP (用户态)
CS (0x1b)
EFLAGS
ESP
SS (0x23)



内核：系统调用处理函数

```
void SystemCall::Trap(struct pt_regs* regs, struct pt_context* context)
```

- 1、取系统调用号（中断栈帧中用户态寄存器EAX, 3），user结构中用u_ar0指针记住这个位置。
- 2、用系统调用号为下标（3），访问系统调用入口表m_SystemEntranceTable，得入口函数首地址和参数数量（3）。
- 3、取系统调用参数，存入user结构 u_arg数组。
- 4、trap1() 调用入口函数(Sys_read)。系统调用执行期间，
 - 执行成功的返回值存入u_ar0指向的单元
 - 失败的错误码取负存入u_ar0指向的单元
- 5、SetPri(),
 计算进程返回用户态之后的优先数，睡过的进程RunRun++
- 6、返回 // 信号会打断系统调用，后面说

```
12 SystemCallTableEntry SystemCall::m_SystemEntranceTable[SYSTEM_CALL_NUM] =
13 {
14     { 0, &Sys_NullSystemCall },      /* 0 = indir */
15     { 1, &Sys_Rexit },                /* 1 = rexit */
16     { 0, &Sys_Fork },                 /* 2 = fork */
17     { 3, &Sys_Read },                 /* 3 = read */
18     { 3, &Sys_Write },                /* 4 = write */
19     { 2, &Sys_Open },                 /* 5 = open */
20 }
```

u_ar0 →

trap1() 栈帧
trap()局部变量区
SystemCall()栈基地址
trap()返回地址
pt_regs
pt_context
GS
FS
DS
ES
EBX
ECX
EDX
ESI
EDI
EBP
EAX
SystemCall()局部变量区
old ebp
EIP (用户态)
CS (0x1b)
EFLAGS
ESP
SS (0x23)



u_arg数组

第1个参数
第2个参数
.....
.....
第5个参数

Trap() 源代码注释

```
void SystemCall::Trap(struct pt_regs* regs, struct pt_context* context)
{
```

User& u = Kernel::Instance().GetUser(); // 0、现运行进程的user结构

.....

u.u_ar0 = ®s->eax; // 1、存返回值的地方

u.u_error = User::NOERROR; // 2、系统调用出错码，清0

SystemCallTableEntry *callp = &m_SystemEntranceTable[regs->eax]; // 3、callp→系统调用表中的元素

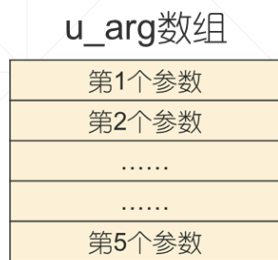
unsigned int * syscall_arg = (unsigned int *)®s->ebx; // 4、中断栈帧中第一个参数的首地址

for(unsigned int i = 0; i < callp->count; i++) // 5、根据入口表登记的参数数量，传参

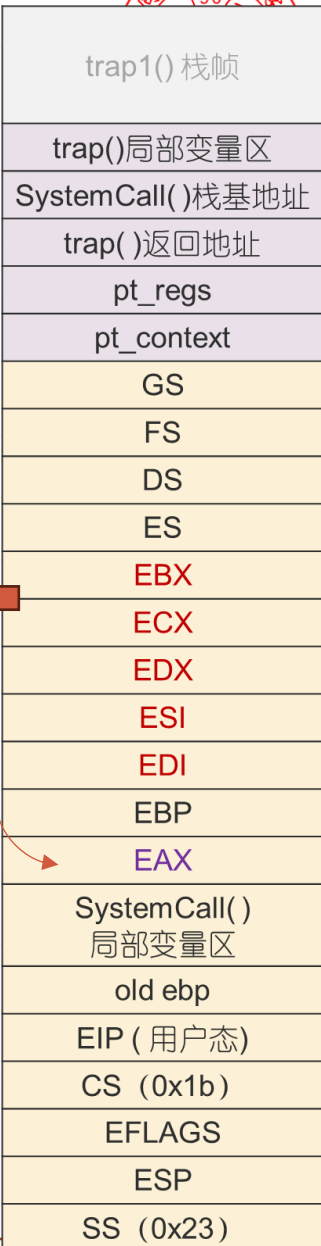
```
{
    u.u_arg[i] = (int)(*syscall_arg++);
}
```

u.u_dirp = (char *)u.u_arg[0]; // 文件系统目录搜索要用的参数， open/create系统调用

u.u_arg[4] = (int)context; // exec系统调用要用的参数



u_ar0





Trap() 源代码注释

```
Trap1(callp->call); // 6、从入口函数进，执行系统调用，read，getpid .....
.....

if( User::NOERROR != u.u_error )
{ // 7、出错嘛？User::NOERROR是0，表示没出错
  regs->eax = -u.u_error; // 出错码取负，存入中断栈帧，带回用户态
} // 错误码定义，参见PPT

.....

u.u_procp->SetPri(); // 8、计算、设置执行应用程序的优先数
}
```

从u_arg数组取入口参数，
将运行结果写入u.u_ar0，
如果出错，错误码写u.u_error。

u_ar0 →

trap1() 栈帧
trap()局部变量区
SystemCall()栈基地址
trap()返回地址
pt_regs
pt_context
GS
FS
DS
ES
EBX
ECX
EDX
ESI
EDI
EBP
EAX
SystemCall() 局部变量区
old ebp
EIP (用户态)
CS (0x1b)
EFLAGS
ESP
SS (0x23)

u_arg数组



第1个参数
第2个参数
.....
.....
第5个参数

回用户态，恢复应用程序使用的寄存器值

- EAX是返回值，或出错码
- EBX、ECX、EDX、ESI、EDI，没有派用处
- 其余寄存器恢复系统调用前的原值

```
int getpid()
{
1:  int res;
2:  __asm__ volatile ( "int $0x80": "=a"(res): "a"(20) ); // 负责执行系统调用的内联汇编函数
3:  if ( res >= 0 )
4:      return res ;
5:  else
6:  {
7:      errno = -res ;
8:      return -1;
9:  }
}
```

2、陷入内核

1、输入部：eax = getpid系统调用号20

3、输出部：系统调用的返回值（进程pid#）→ eax

钩子函数

u_ar0 →

trap1() 栈帧
trap()局部变量区
SystemCall() 栈基地址
trap() 返回地址
pt_regs
pt_context
GS
FS
DS
ES
EBX
ECX
EDX
ESI
EDI
EBP
EAX
SystemCall() 局部变量区
old ebp
EIP (用户态)
CS (0x1b)
EFLAGS
ESP
SS (0x23)

u_arg数组

第1个参数
第2个参数
.....
.....
第5个参数



附录：User.h中定义的系统调用出错码

```
enum ErrorCode
{
    NOERROR = 0,      /* No error */
    EPERM = 1,        /* Operation not permitted */
    ENOENT = 2,        /* No such file or directory */
    ESRCH = 3,         /* No such process */
    EINTR = 4,         /* Interrupted system call */
    EIO = 5,           /* I/O error */
    ENXIO = 6,         /* No such device or address */
    E2BIG = 7,         /* Arg list too long */
    ENOEXEC = 8,       /* Exec format error */
    EBADF = 9,         /* Bad file number */
    ECHILD = 10,       /* No child processes */
    EAGAIN = 11,       /* Try again */
    ENOMEM = 12,       /* Out of memory */
    EACCES = 13,       /* Permission denied */
    EFAULT = 14,       /* Bad address */
    ENOTBLK = 15,      /* Block device required */
    EBUSY = 16,        /* Device or resource busy */
    EEXIST = 17,       /* File exists */
    EXDEV = 18,        /* Cross-device link */
    ENODEV = 19,       /* No such device */
    ENOTDIR = 20,      /* Not a directory */
    EISDIR = 21,       /* Is a directory */
    EINVAL = 22,       /* Invalid argument */
    ENFILE = 23,       /* File table overflow */
    EMFILE = 24,       /* Too many open files */
    ENOTTY = 25,       /* Not a typewriter (terminal) */
    ETXTBSY = 26,      /* Text file busy */
    EFBIG = 27,        /* File too large */
    ENOSPC = 28,       /* No space left on device */
    EPIPE = 29,        /* Illegal seek */
    EROFS = 30,        /* Read-only file system */
    EMLINK = 31,       /* Too many links */
    EPIPE = 32,        /* Broken pipe */
    ENOSYS = 100,
    //EFAULT = 106
};
```

慢系统调用 和 快系统调用

- 快系统调用，指不会入睡的系统调用或访问磁盘的系统调用。内核有把握，在有限时间内完成系统调用服务。若入睡，`p_stat=SSLEEP`。
- 其余是慢系统调用。内核无法控制这些系统调用的运行时长。进程会睡很久，有可能永远醒不了。若入睡，`p_stat= SWAIT`。
 - `scanf/getchar`
 - 读网络数据包
 - `sleep(1000000)`



Part 2 异常和信号

一、异常

- 当前指令无法正常执行时，CPU会给自己发中断信号，激活内核处理异常事件。
Exception，异常。

产生异常的原因	具体的语义	默认处理方式
应用程序有错	执行非法指令，运算溢出，访问非法内存	用信号杀死出错的现运行进程
需要为现运行进程扩展内存	本条指令访问的数据，或下条要执行的指令不在内存	注1
内核有错	-	panic，系统停摆，等待管理员关机、修复、重启
硬件故障	-	用信号杀死现运行进程（前，用户态）或 panic（前，核心态）

注1：虚拟存储器，请求调页。连续存储管理方式的 Unix V6++，堆栈自动扩展

注2：这张表不包含3#断点异常

异常的产生

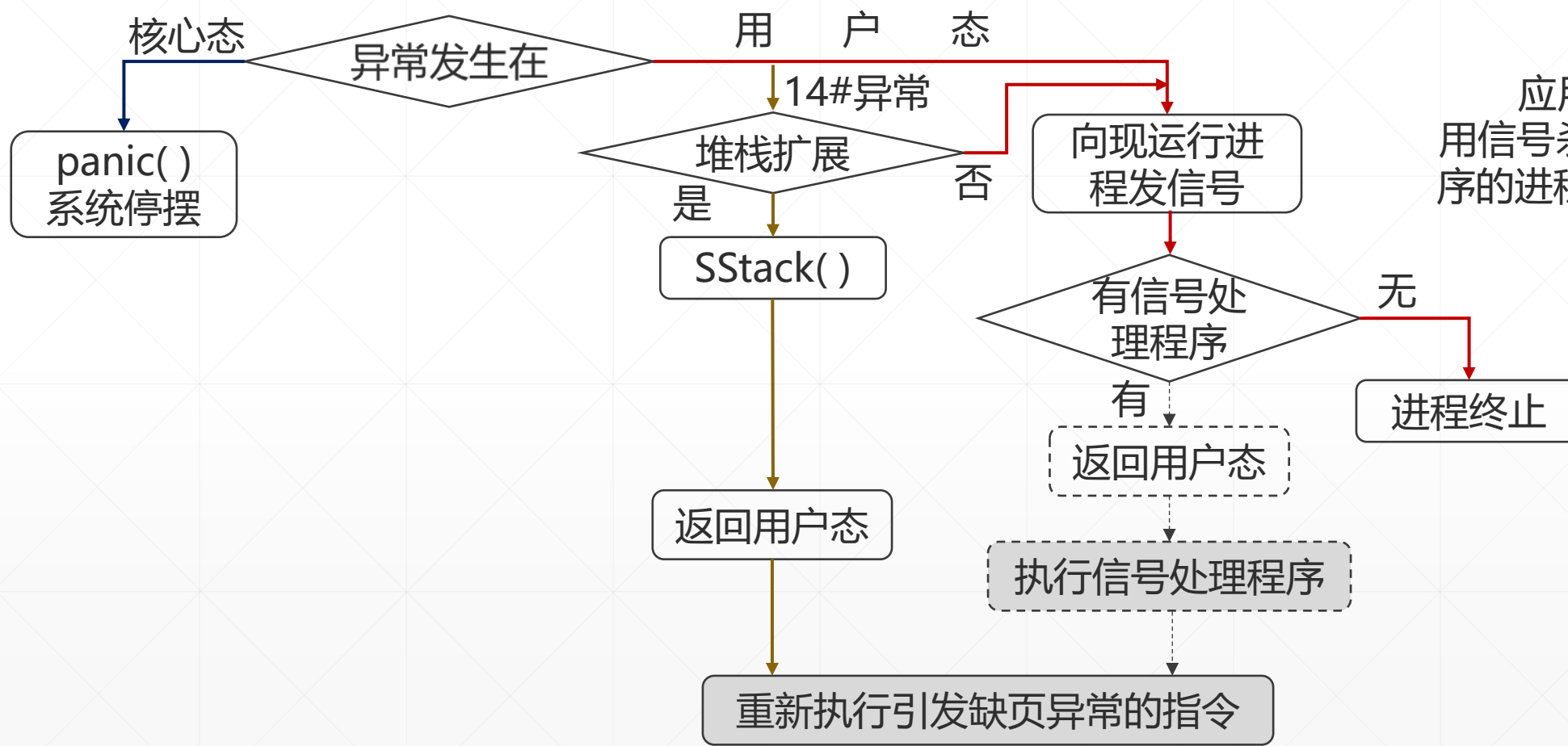
#	Exception
0	Divide error
1	Debug
2	NMI
3	Breakpoint
4	Overflow
5	Bounds check
6	Invalid opcode
7	Device not available
8	Double fault
9	Coprocessor segment overrun
10	Invalid TSS
11	Segment not present
12	Stack exception
13	General protection
14	Page Fault
15	Intel reserved
16	Floating-point error
17	Alignment check
18	Machine check
19	SIMD floating point

例1：CPU执行C语句 `c = a/b;` // 变量b值为0
CPU执行单元抛出 0#异常(Divide Error)，表示除法出错。

例2：MMU页级单元实施地址映射和内存保护操作时，无法给出当前虚地址对应的物理地址，抛出 14#缺页异常(Page Fault)。具体而言：

- 1、线性地址高10位是页表号，用它查页目录，获得映射所需页表所在的页框号，中10位是页号，用它查页表，得映射所需PTE。
 - PDE(页目录项) 或 PTE为空，抛出14#缺页异常
- 2、用PTE中的prot，识别非法内存访问
 - CPU用户态执行 (CPL=11)，prot S，抛出14#缺页异常
 - CPU执行内存写操作，prot RO，抛出14#缺页异常
 - CPU取指令，prot ~X，抛出14#缺页异常
- 3、物理地址 $PA = base \text{ 拼接 } offset$

异常产生后，内核予以干预 (Unix V6++)



应用程序有bug
用信号杀死执行该应用程序的进程。程序异常终止。

每个异常，对应一个确切的信号

#	Exception	Signal
0	Divide error	SIGFPE
1	Debug	SIGTRAP
2	NMI	None
3	Breakpoint	SIGTRAP
4	Overflow	SIGSEGV
5	Bounds check	SIGSEGV
6	Invalid opcode	SIGILL
7	Device not available	SIGSEGV
8	Double fault	SIGSEGV
9	Coprocessor segment overrun	SIGFPE
10	Invalid TSS	SIGSEGV
11	Segment not present	SIGBUS
12	Stack exception	SIGBUS
13	General protection	SIGSEGV
14	Page Fault	SIGSEGV
15	Intel reserved	None
16	Floating-point error	SIGFPE
17	Alignment check	SIGBUS
18	Machine check	None
19	SIMD floating point	SIGFPE

异常处理程序发出的信号

异常发生时，异常处理程序将这个信号发送给现运行进程，用信号杀死出错的进程

例1：CPU执行C语句 $z = x/y$ ；// 变量y值为0

0#异常，SIGFPE信号（浮点运算错误）

例2：现运行进程非法内存访问

14#缺页异常，SIGSEGV信号（段错误，Segment Fault）。



异常处理流程 (0#异常)

现运行进程用户态执行C语句: $c = a/b$; // 变量b, 值为0
CPU执行单元抛出0#异常

```
void Exception:: DivideErrorEntrance()  
{  
    SaveContext();  
  
    SwitchToKernel();  
  
    CallHandler(Exception, DivideError);  
  
    RestoreContext();  
  
    Leave();  
  
    InterruptReturn();  
}  
  
void Exception:: DivideError(struct pt_regs* regs, struct pt_context* context)  
{  
    User& u = Kernel::Instance().GetUser(); // user, 指向现运行进程的user结构  
    Process* current = u.u_procp; // current, 指向现运行进程的Process结构  
  
    if ( (context->xcs & USER_MODE) == USER_MODE ) // 异常发生在用户态  
    {  
        current->PSignal(User::SIGFPE); // 向现运行进程发SIGFPE信号  
        if ( current->IsSig() )  
            current->PSig(context); // 信号处理  
    }  
    else // 异常发生在核心态  
    {  
        Utility::Panic("Divide Exception!"); // panic, 系统停摆  
    }  
}
```



二、Unix 信号处理

- 信号用来控制进程执行过程。
 - 杀死进程
 - 内核（异常处理程序）用信号终止出错的应用程序
 - 用户，用信号终止应用程序。
 - 合作进程的异步通信手段
 - 发送进程向接收进程发一个信号，接收进程暂停原先任务，优先执行信号处理程序，与发送进程交互。


```
static const int NSIG = 32; /* 信号个数 */

/* p_sig中接受到的信号定义 */
static const int SIGNUL = 0; /* No Signal Received */
static const int SIGHUP = 1; /* Hangup (kill controlling terminal) */
static const int SIGINT = 2; /* Interrupt from keyboard */
static const int SIGQUIT = 3; /* Quit from keyboard */
static const int SIGILL = 4; /* Illegal instruction */
static const int SIGTRAP = 5; /* Trace trap */
static const int SIGABRT = 6; /* use abort() API */
static const int SIGBUS = 7; /* Bus error */
static const int SIGFPE = 8; /* Floating point exception */
static const int SIGKILL = 9; /* Kill(can't be caught or ignored) */
static const int SIGUSR1 = 10; /* User defined signal 1 */
static const int SIGSEGV = 11; /* Invalid memory segment access */
static const int SIGUSR2 = 12; /* User defined signal 2 */
static const int SIGPIPE = 13; /* Write on a pipe with no reader, Broken pipe */
static const int SIGALRM = 14; /* Alarm clock */
static const int SIGTERM = 15; /* Termination */
static const int SIGSTKFLT = 16; /* Stack fault */
static const int SIGCHLD = 17; /* Child process has stopped or exited, changed */
static const int SIGCONT = 18; /* Continue executing, if stopped */
static const int SIGSTOP = 19; /* Stop executing */
static const int SIGTSTP = 20; /* Terminal stop signal */
static const int SIGTTIN = 21; /* Background process trying to read, from TTY */
static const int SIGTTOU = 22; /* Background process trying to write, to TTY */
static const int SIGURG = 23; /* Urgent condition on socket */
static const int SIGXCPU = 24; /* CPU limit exceeded */
static const int SIGXFSZ = 25; /* File size limit exceeded */
static const int SIGVTALRM = 26; /* Virtual alarm clock */
static const int SIGPROF = 27; /* Profiling alarm clock */
static const int SIGWINCH = 28; /* Window size change */
static const int SIGIO = 29; /* I/O now possible */
static const int SIGPWR = 30; /* Power failure restart */
static const int SIGSYS = 31; /* invalid sys call */
```

- 实现上，信号是一个小整数，每个信号有确切的语义。Unix V6++支持32种传统的Unix信号，符合国际标准POSIX。例：
 - ctrl+c 终止正在执行的应用程序。键盘中断处理程序会给相关进程发SIGINT，2#信号。
 - kill -9 pid。向PID#为pid的进程发SIGKILL，9#信号。
 - 现运行进程执行除数为0的除法指令，得到一个SIGFPE，8#信号。
 - 现运行进程非法内存访问，得到一个SIGSEGV，11#信号。
 - 子进程终止时，父进程会得到一个SIGCHLD，17#信号。

Unix V6++的信号处理机制

- 每个进程有自己私有的信号处理方式。
- User结构, `u_signal`数组, 32个元素。 `u_signal[i]`是*i*#信号的处理方式。
 - 0, *i*#信号会杀死进程
 - -1, 忽略。*i*#信号对进程没有影响
 - & 信号处理程序。回用户态, 执行信号处理程序。
- 进程的信号处理方式, 由其执行的应用程序设置。
 - `exec`系统调用加载应用程序时, `u_signal`数组清0。任何信号均可以杀死该进程。
 - 应用程序可以使用`signal`系统调用为信号设置信号处理方式。
- Process结构, `p_sig`字段是进程收到的信号。
 - 没有信号需要处理, `p_sig == 0`
 - 收到SIGINT信号, `p_sig == 2`



与异常相关的信号处理过程（除数为0）

```
void Exception:: DivideError(struct pt_regs* regs, struct pt_context* context)
{
    User& u = Kernel::Instance().GetUser();           // user, 指向现运行进程的user结构
    Process* current = u.u_procp;                     // current, 指向现运行进程的Process结构

    if ( (context->xcs & USER_MODE) == USER_MODE ) // 异常发生在用户态
    {
        current->PSignal(User::SIGFPE); // current -> p_sig = 8
        if ( current->IsSig() ) // current -> p_sig 非0, 有信号需要处理
            current->PSig(context); // 信号处理
    }
    else // 异常发生在核心态
    {
        Utility::Panic("Divide Exception!");
    }
}
```

```
void Process::PSig(struct pt_context* pContext)
{
    User& u = Kernel::Instance().GetUser();
    int signal = this->p_sig;
    this->p_sig = 0;

    if ( u.u_signal[signal] != 0 )
        .....
    else
        exit( ); // 进程终止
}
```


除0，进程终止

```
#include <unistd.h>
#include <stdio.h>
int
main(void)
{
    int a,b,c;

    for ( ; ; )
    {
        printf("输入被除数\n");
        scanf("%d", &a);

        printf("输入除数\n");
        scanf("%d", &b); ← 输入0,
                           进程终止

        c=a/b;

        printf("%d / %d = %d\n",a,b,c);
    }
}
```

```
drong@ubuntu: ~/Desktop/ostest
drong@ubuntu:~/Desktop/ostest$ gcc noException.c
drong@ubuntu:~/Desktop/ostest$ ./a.out
输入被除数
10
输入除数
2
10 / 2 = 5
输入被除数
2
输入除数
0
Floating point exception (core dumped)
drong@ubuntu:~/Desktop/ostest$
```

0#异常 → SIGFPE信号 → 现运行进程执行exit()函数终止

应用程序设置信号处理方式后，收到信号进程不会终止



```
noException.c x divide0.c x ctrlc2.c x exception
#include <signal.h>
#include <unistd.h>
#include <stdio.h>

static void sig_dyzero(int);

int
main(void)
{
    int a,b;
    int c;

    if (signal(SIGFPE, sig_dyzero) == SIG_ERR)
        printf("can't catch divide by zero error!");

    for ( ; ; )
    {
        printf("输入被除数\n");
        scanf("%d", &a);

        printf("输入除数\n");
        scanf("%d", &b);

        c=a/b;
        printf("%d / %d = %d\n",a,b,c);
    }
}
```

```
static void
sig_dyzero(int signo)    /* argument is signal number */
{
    printf("Divide by zero!\n");
}
```

u_signal[8] = sig_dyzero

```
void Process::PSig(struct pt_context* pContext)
{
    User& u = Kernel::Instance().GetUser();
    int signal = this->p_sig;
    this->p_sig = 0;

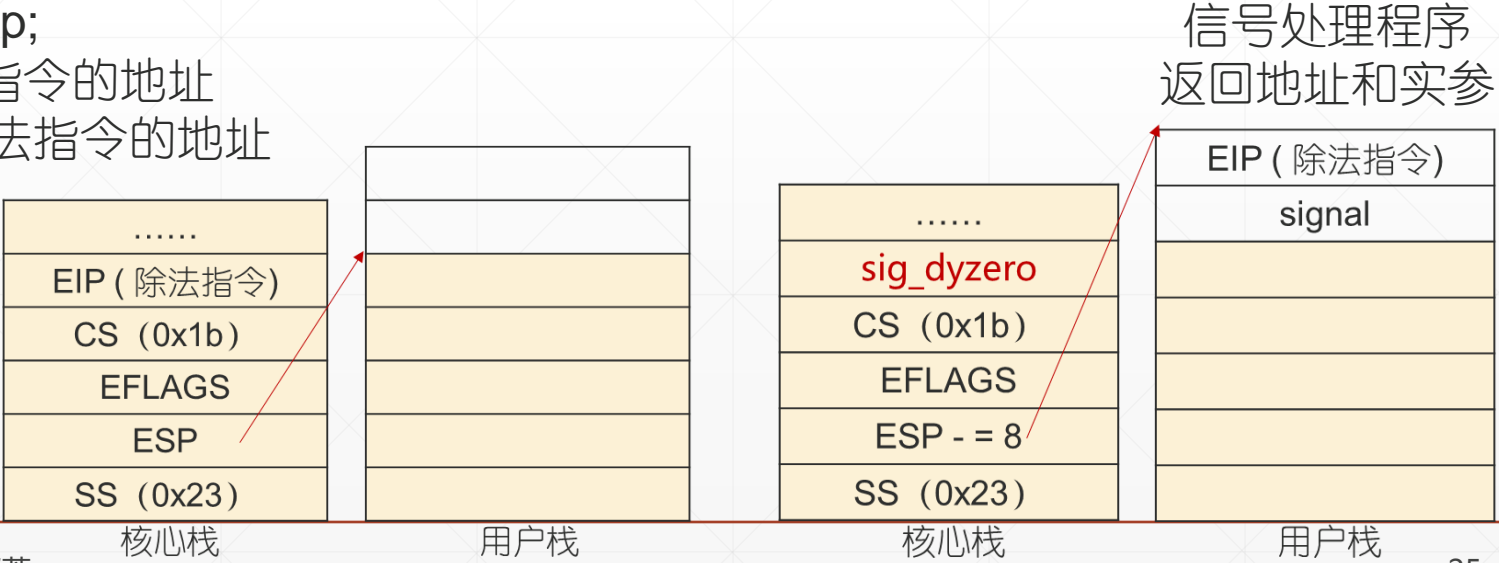
    if ( u.u_signal[signal] != 0 )
        .....
    else
        exit(); // 进程终止
}
```



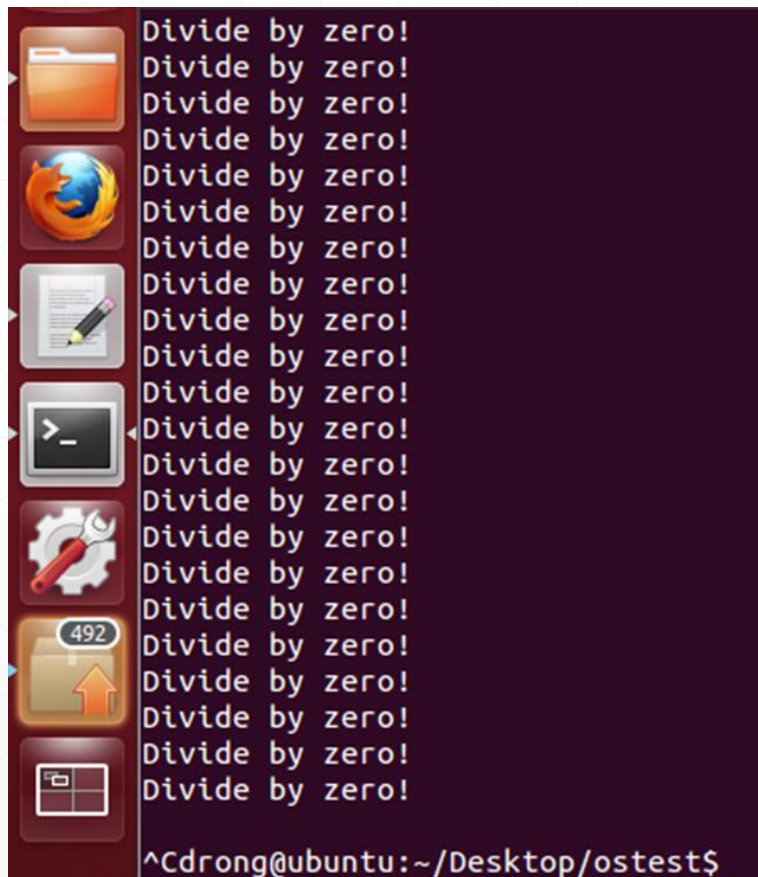
```
void Process::PSig(struct pt_context* pContext)
{
    User& u = Kernel::Instance().GetUser();
    int signal = this->p_sig;
    this->p_sig = 0;

    if ( u.u_signal[signal] != 0 ) // sig_dyzero
    {
        .....
        unsigned int old_eip = pContext->eip; // 除法指令的eip
        pContext->eip = u.u_signal[signal]; // 返回用户态，执行信号处理函数
        pContext->esp -= 8; // 用户栈顶上移（向低地址端）8字节
        int* plnt = (int *)pContext->esp;
        *plnt = old_eip; // 存入除法指令的地址
        *(plnt+4) = signal; // 存入除法指令的地址
        .....
        return;
    }
    else
        exit( );
}
```

```
static void
sig_dyzero(int signo)      /* argument is signal number */
{
    printf("Divide by zero!\n");
}
```



习题1 debug



```
noException.c x divide0.c x ctrlc2.c x exception
#include <signal.h>
#include <unistd.h>
#include <stdio.h>

static void sig_dyzero(int);

int
main(void)
{
    int a,b;
    int c;

    if (signal(SIGFPE, sig_dyzero) == SIG_ERR)
        printf("can't catch divide by zero error!");

    for ( ; ; )
    {
        printf("输入被除数\n");
        scanf("%d", &a);

        printf("输入除数\n");
        scanf("%d", &b);

        c=a/b;
        printf("%d / %d = %d\n",a,b,c);
    }
}
```

```
static void
sig_dyzero(int signo)    /* argument is :
{
    printf("Divide by zero!\n");
}
```

要求

编程

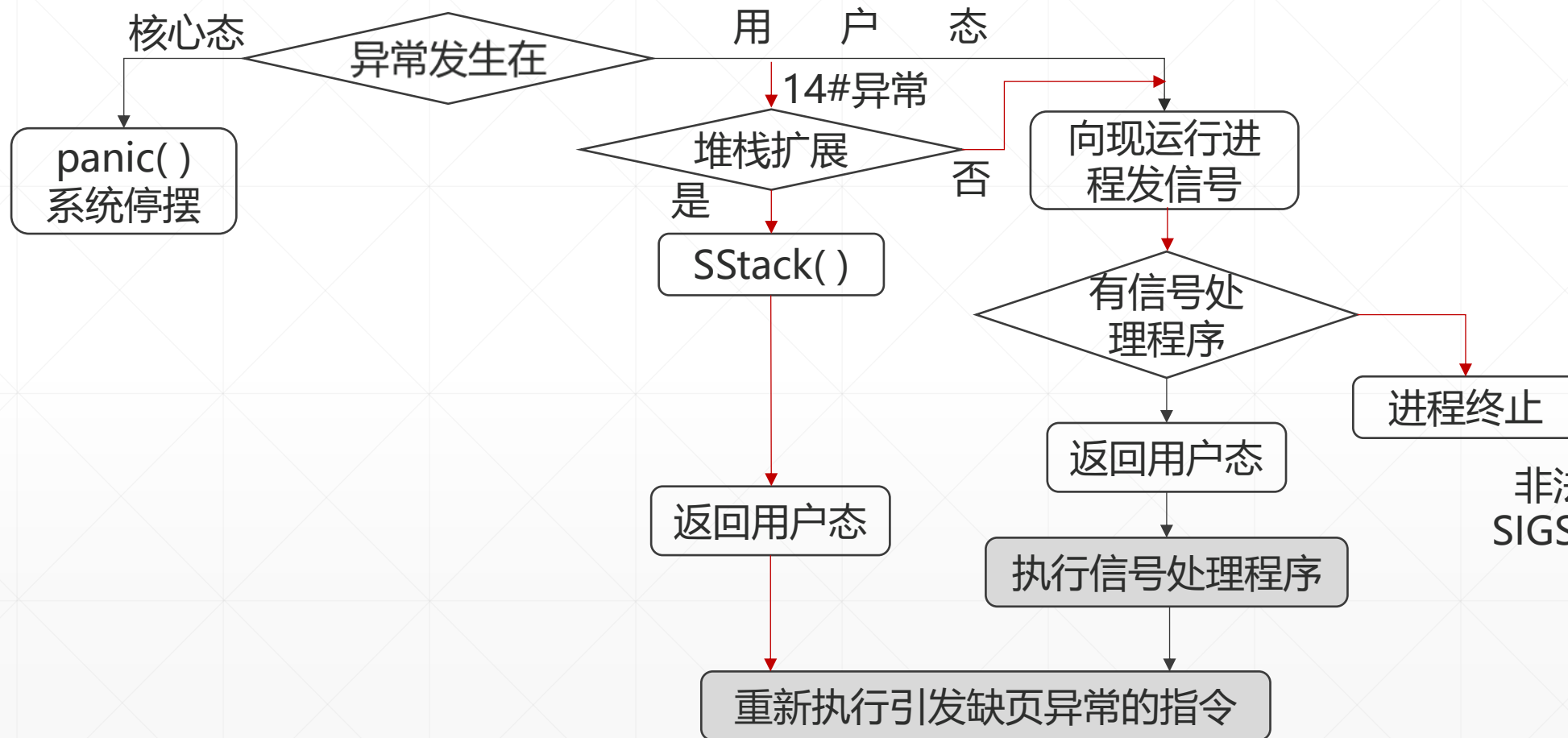
- 1、右图输出
- 2、ctrl+c 杀不死这个进程

想办法终止你改过的这个应用程序
(不可以关terminal)

```
drong@ubuntu: ~/Desktop/ostest
drong@ubuntu:~/Desktop/ostest$ gcc exception4.c
drong@ubuntu:~/Desktop/ostest$ ./a.out
输入被除数
10
输入除数
4
10 / 4 = 2
输入被除数
5
输入除数
0
Divide by zero!
输入被除数
9
输入除数
0
Divide by zero!
输入被除数
█
```



三、Unix V6++的缺页异常 (Page Fault)



非法内存访问，用信号
SIGSEGV杀死现运行进程



```
void Exception::PageFault(struct pt_regs* regs, struct pte_context* context)
{
    User& u = Kernel::Instance().GetUser();    // 现运行进程的user结构
    Process* current = u.u_procp;              // Process对象
    MemoryDescriptor& md = u.u_MemoryDescriptor; // 内存描述符

    unsigned int cr2;
    __asm__ __volatile__( " mov %%cr2, %0" : "=r"(cr2) ); // cr2是产生缺页的线性地址（虚地址）

    if( (context->xcs & USER_MODE) == USER_MODE)
    {
        if( cr2 < MemoryDescriptor::USER_SPACE_SIZE - md.m_StackSize && cr2 >= context->esp - 8
            && md.m_DataSize + md.m_StackSize + PageManager::PAGE_SIZE < MemoryDescriptor::USER_SPACE_SIZE - md.m_DataStartAddress )
            current->SStack(); // 堆栈扩展
        else
        {
            Diagnose::Write("Invalid MM access");
            current -> PSignal(User::SIGSEGV);
            if ( current->IsSig() )
                current->PSig( (pt_context *)&context->eip );
        } // 段错误（Segment Fault），用信号SIGSEGV杀死缺页的现运行进程
    } // 应用程序缺页
    else
        Utility::Panic("Page Fault in Kernel Mode."); // 内核缺页，不可能。有bug，停摆。
}
```

例2：MMU页级单元实施地址映射和内存保护操作时，无法给出当前虚地址对应的物理地址，抛出14#缺页异常(Page Fault)。具体而言：

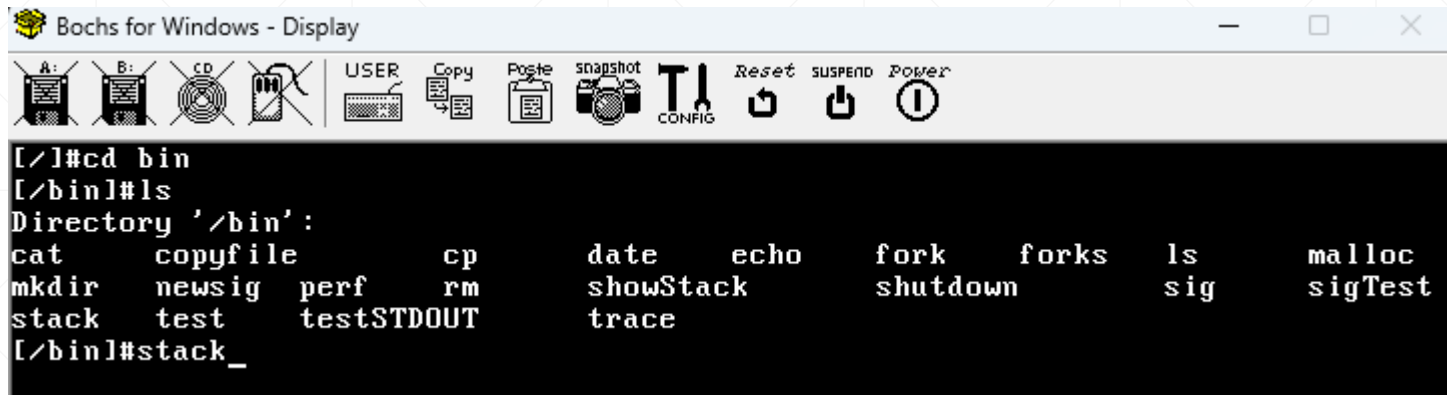
1、线性地址高10位是页表号，用它查页目录，获得映射所需页表所在的页框号，中10位是页号，用它查页表，得映射所需PTE。

- PDE(页目录项) 或 PTE为空，抛出14#缺页异常
- 2、用PTE中的prot，识别非法内存访问
- CPU用户态执行（CPL=11），prot S，抛出14#缺页异常
- CPU执行内存写操作，prot RO，抛出14#缺页异常
- CPU取指令，prot ~X，抛出14#缺页异常

3、物理地址 PA = base 拼接 offset

习题2

- 运行应用程序stack，观察堆栈扩展过程。



```
[/]#cd bin
[/bin]#ls
Directory '/bin':
cat      copyfile    cp          date      echo      fork      forks      ls         malloc
mkdir    newsig      perf        rm         showStack shutdown   sig        sigTest
stack    test        testSTDOUT trace

[/bin]#stack_
```

- 条件 $cr2 < \text{MemoryDescriptor::USER_SPACE_SIZE} - md.m_StackSize \ \&\& \ cr2 \geq \text{context->esp} - 8$ 用来判断缺页是否因为堆栈扩展，请解释这个判断条件的合理性。

- 为什么把条件改为

$cr2 < \text{MemoryDescriptor::USER_SPACE_SIZE} - md.m_StackSize \ \&\& \ cr2 \geq \text{context->esp}$

stack程序运行一半会挂掉。

四、其它

1、异常有3种类型

- fault。现运行进程出错。
 - 压栈的EIP是引发异常的当前指令。异常处理程序执行完毕后，如果现运行进程未被杀死，重新执行引发异常的那条指令。
 - 0#，14#是fault。
- trap。被调试的程序走到断点。3#异常。
 - 压栈的EIP是下条指令。内核会让被调试进程暂停。暂停后，后者等待下个调试命令。
- abort。运行环境致命错误，引发异常的进程不可能继续运行。
 - 内核或硬件出错。现运行进程终止，内核panic。EIP不必压。

2、用宏展开所有异常的入口和处理程序（不包括14#）

SystemCall.cpp

```
#define IMPLEMENT_EXCEPTION_ENTRANCE(Exception_Entrance, Exception_Handler) \
void Exception::Exception_Entrance() \
{ \
    SaveContext(); \
    SwitchToKernel(); \
    CallHandler(Exception, Exception_Handler); \
    RestoreContext(); \
    Leave(); \
    InterruptReturn(); \
}
```

```
#define IMPLEMENT_EXCEPTION_HANDLER(Exception_Handler, Error_Message, Signal_Value) \
void Exception::Exception_Handler(struct pt_regs* regs, struct pt_context* context) \
{ \
    User& u = Kernel::Instance().GetUser(); \
    Process* current = u.u_procp; \
    if ( (context->xcs & USER_MODE) == USER_MODE ) \
    { \
        current->PSignal(Signal_Value); \
        if ( current->IsSig() ) \
            current->PSig(context); \
    } // 异常发生在用户态 \
    else \
    { \
        Utility::Panic(Error_Message); \
    } // 异常发生在核心态，计算机停摆。输出报错info: Error_Message, 等待重启 \
}
```

向现运行进程发这个异常对应的信号 Signal_Value

信号处理：无信号处理函数，进程终止有，返回用户态，执行信号处理函数

异常入口程序名

异常处理程序名

异常处理程序名

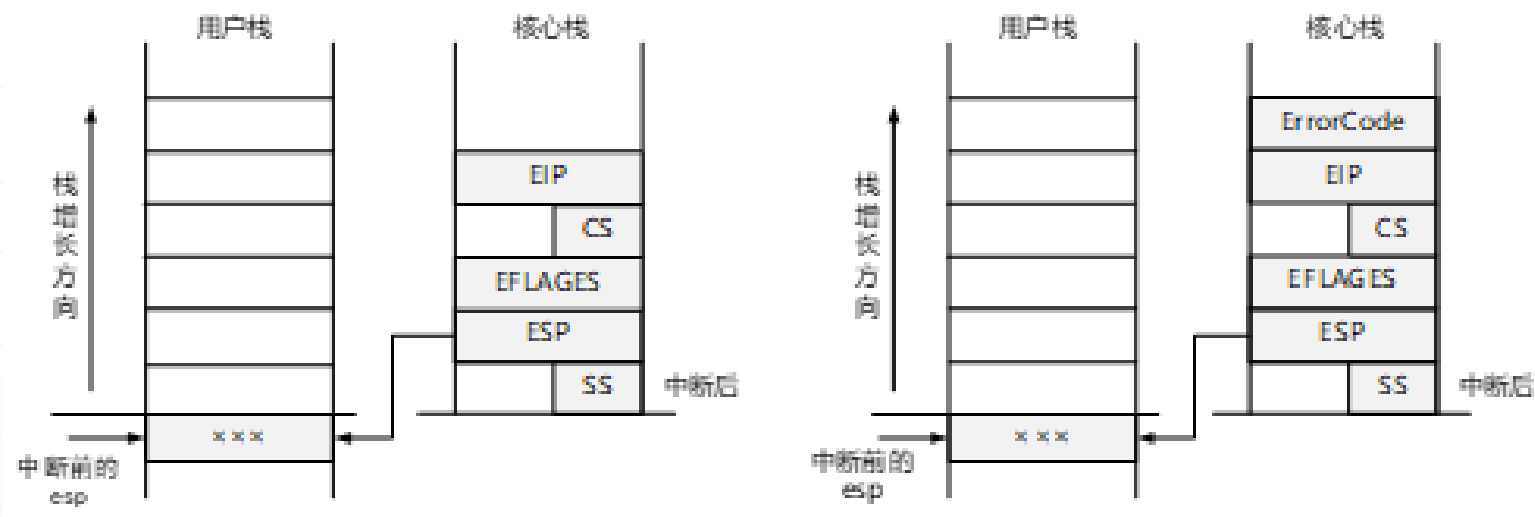
描述异常的报错 info

异常对应的信号

IMPLEMENT_EXCEPTION_ENTRANCE(DivideErrorEntrance, DivideError)

IMPLEMENT_EXCEPTION_HANDLER(DivideError, "Divide Exception!", User::SIGFPE)

3、有些异常，硬件会压出错码 error code



比如，14#异常 PageFault。
入口程序 IRET 前，要注意 pop ErrorCode。

4、信号、低优先级睡眠 和 被信号打断的系统调用

- 信号会打断低优先级睡眠状态。

```

405 void Process::PSignal( int signal )
406 {
407     if ( signal >= User::NSIG )
408     {
409         return;
410     }
411     /* 如果已经接收到SIGKILL信号，则忽略后续信号 */
412     if ( this->p_sig != User::SIGKILL )
413     {
414         this->p_sig = signal;
415     }
416     /* 若进程的优先数大于PUSER(100)，则将其设置为PUSER */
417     if ( this->p_pri > ProcessManager::PUSER )
418     {
419         this->p_pri = ProcessManager::PUSER;
420     }
421     /* 若进程的处于低优先级睡眠，则将其唤醒 */
422     if ( this->p_stat == Process::SWAIT )
423     {
424         this->SetRun();
425     }
426 }
427 }

```

```

void Process::Sleep(unsigned long chan, int pri)
{
    .....
    if ( pri > 0 )
    {
        if ( this->IsSig() )
        {
            aRetU(u.u_qsav);
            return;
        }
        低优先级睡眠过程：..... Swch() .....
        if ( this->IsSig() )
        {
            aRetU(u.u_qsav);
            return;
        }
    }
    else
    {
        ..... 高优先级睡眠 .....
    }
}

```

如果有信号，不睡了，
返回 trap1()

如果有信号，不睡了，
返回 trap1()

执行系统调用

```
void SystemCall::Trap( )
{
    .....
    Trap1(callp->call);
    if ( u.u_intflg != 0 )
        u.u_error = User::EINTR;
    .....
}
```

```
void SystemCall::Trap1(int (*func)())
{
    User& u = Kernel::Instance().GetUser();

    u.u_intflg = 1;
    SaveU(u.u_qsav);
    func(); // 系统调用入口函数
    u.u_intflg = 0;
}
```

```
21 #define SaveU(u_sav) \
22     __asm__ __volatile__( \
23         "movl %%esp, %0\n\t" \
24         "movl %%ebp, %1\n\t" \
25         : "+m"(u_sav[0]), "+m"(u_sav[1]) \
26         );
```

收到信号
Switch返回

```
if ( this->IsSig() )
{
    aRetU(u.u_qsav);
    return;
}
```

```
57 #define aRetU(u_sav) \
58     __asm__ __volatile__( \
59         "movl %0, %%esp;" \
60         "movl %1, %%ebp;" \
61         : \
62         : "m"(u_sav[0]), "m"(u_sav[1]) );
```

系统调用失败返回
错误码EINTR，被信号打断