

第二章 Linux的信号处理

信号

- 信号是更高层次的异常。允许进程和内核打断其它进程
- 信号是一个整数。它是一条小消息，通知进程发生了某种类型的事件

编号	信号名称	默认的信号处理方式	解释
0	SIGNUL		未收到信号
1	SIGHUP	Abort	控制tty断开连接（挂断）
2	SIGINT	Abort	来自键盘的中断（用户在键盘上按CTRL_C）
3	SIGQUIT	Dump	来自键盘的退出（TTY键盘上按CTRL_\\）
4	SIGILL	Dump	非法指令（异常）
5	SIGTRAP	Dump	跟踪断点（遇到debug断点，用于调试）
6	SIGABRT	Dump	异常结束（使进程流产）
6	SIGIOT	Dump	等价于SIGABRT
7	SIGBUS	Abort	访问内存失败
8	SIGFPE	Dump	算术运算或浮点处理出错
9	SIGKILL	Abort	强迫进程终止（不可屏蔽）
	
13	SIGPIPE	Abort	向无读者的管道（管道读端已关闭）写
	

定义在user.h中

信号机制使用的数据结构 (Unix V6++)

- 进程收到的信号

Process[i].**p_sig**

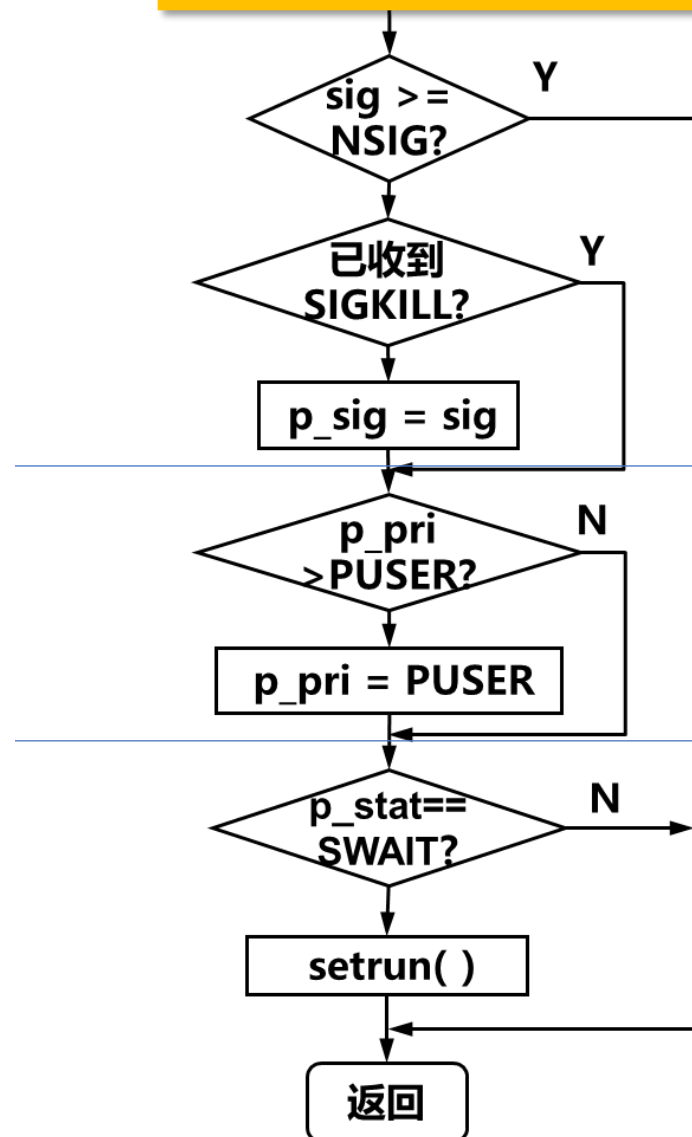
- 信号处理方式

user.**u_signal**[i]

- 0 系统默认方式处理
- 1 忽略
- 偶数 信号处理程序的入口地址

信号发送

信号的发送:
Process::PSignal(int sig)



1、发信号

2、调高信号接收进程优先权，
至用户态进程的最高优先权

3、如果进程在低优先权睡眠
状态收到信号，唤醒它

内核直接发信号

程序运行时出错：
将异常转化为信号

```
.....  
if ( (context->xcs & USER_MODE) == USER_MODE )  
{  
    current->PSignal(Signal_Value);  
    if ( current->IsSig() )  
        current->PSig(context);  
}  
.....
```

作业控制：

用户希望提前终止应用程序运行时，按ctrl+c按键。系统将SIGINT信号送给与这个终端相关的所有进程。

读写 PIPE 和 socket：

写进程执行写操作时，发现读端已关闭，向自己发送一个SIGPIPE信号

应用程序 使用kill系 统调用 发信号

```
2 void ProcessManager::Kill()
3 {
4     User& u = Kernel::Instance().GetUser();
5     int pid = u.u_arg[0];
6     int signal = u.u_arg[1];
7     bool flag = false;
8
9     for ( int i = 0; i < ProcessManager::NPROC; i++ )
10    {
11        /* 不允许发送信号给进程自身 */
12        if ( u.u_procp == &process[i] )
13        {
14            continue;
15        }
16        /* 不是信号的接收方目标进程，继续搜寻 */
17        if ( pid != 0 && process[i].p_pid != pid )
18        {
19            continue;
20        }
21        /* pid为0，则将信号发送至与发送进程同一终端的所有进程，0#进程不包括在内 */
22        if ( pid == 0 && (process[i].p_ttyp != u.u_procp->p_ttyp || i == 0 ) )
23        {
24            continue;
25        }
26        /* 除非是超级用户，否则要求发送、接收进程u.uid相同，即不可给其它用户进程发送信号 */
27        if ( u.u_uid != 0 && u.u_uid != process[i].p_uid )
28        {
29            continue;
30        }
31        flag = true;
32        /* 信号发送给满足条件的目标进程 */
33        process[i].PSignal(signal);
34    }
35    if ( false == flag )
36    {
```

信号处理 Psig()

```
void Process::PSig(struct pt_context* pContext)
{
    int signal = this->p_sig;
    this->p_sig = 0;

    if ( u.u_signal[signal] != 0 )
    {
        unsigned int old_eip = pContext->eip;
        pContext->eip = u.u_signal[signal];
        pContext->esp -= 4;          // -8
        int* pInt = (int *)pContext->esp;
        *pInt = old_eip;           // 接下去: pInt++; *pInt = signal;

        u.u_signal[signal] = 0;
        return;
    }

    /* u.u_signal[n]为0, 采用默认方式处理信号, 进程终止 */
    u.u_procp->Exit();
}
```

信号处理时机

- 进程下次返回用户态前夕
- 用户态执行时收到信号
 - 系统调用执行前
(这个系统调用就不执行了)
- 下一个整数秒时刻，时钟中断处理程序

```
void SystemCall::Trap( )
{
    User& u = Kernel::Instance().GetUser();

    /* 新加进的代码。判断有无接收到信号，如接收到信号则
       进行响应 */
    if ( u.u_procp->IsSig() )
    {
        u.u_procp->PSig(context);
        u.u_error = User::EINTR;
        regs->eax = -u.u_error;
        return;
    }
    ..... 调用Trap1() 执行系统调用 ..... }
```

```
void Time::Clock( )
{
    .....
    /* 如果中断前为用户态，则考虑进行信号处理 */
    if ( (context->xcs & USER_MODE) == USER_MODE )
    {
        if ( current->IsSig() )
            current->PSig(context);
    }
}
```


信号处理时机

- 进程下次返回用户态
- 执行系统调用时收到信号
 - 系统调用返回时刻处理信号
 - 快系统调用
磁盘IO结束后
 - 慢系统调用
接收信号的时刻

```
void SystemCall::Trap( )
{
    ..... 调用Trap1() 执行系统调用;
    /* 系统调用返回用户态前, 重算当前进程优先数 */
    u.u_procp->SetPri();
    if ( u.u_procp->IsSig() )
        u.u_procp->PSig(context);
}
```

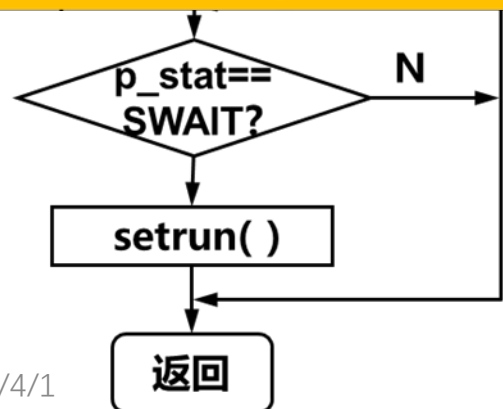
系统调用被信号打断, 失败返回
系统调用的返回值是 -1;
系统调用出错码 error == EINTR

被信号打断的系统调用

```
void SystemCall::Trap( )
{
    ..... 调用Trap1() 执行系统调用;
    /* 系统调用返回用户态前, 重算当前进程优先数 */
L1:  u.u_procp->SetPri();
    if ( u.u_procp->IsSig() )
        u.u_procp->PSig(context);
}
```

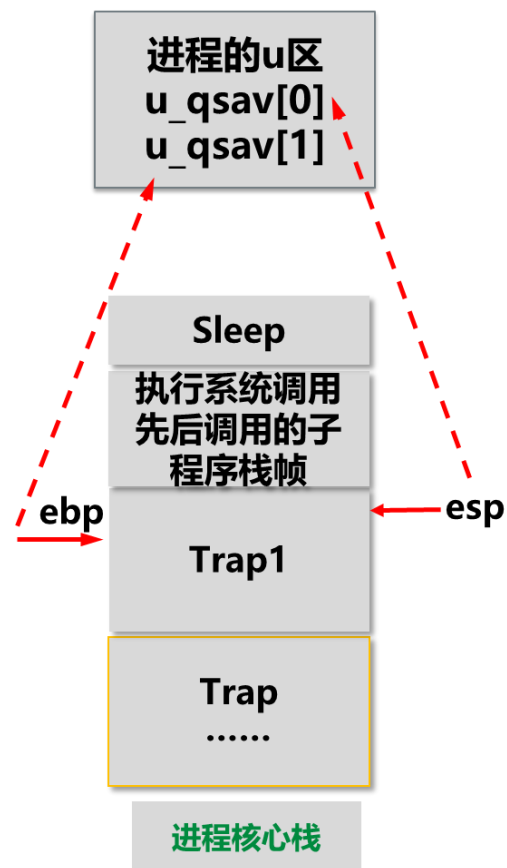
```
Void SystemCall::Trap1(int (*func)())
{
    .....
    u.u_intflg = 1;
    SaveU(u.u_qsav);
    func();
    u.u_intflg = 0;
}
```

信号的发送: Process::PSignal(int sig)



sleep函数, 低优先权睡眠分支

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



例

从文件描述符 fd 指向的文件中读 n 个字节。只有读到 EOF，函数的返回值才会小于 n；其余，无论出现什么情况，函数一定会读入 n 个字节。

```
nleft = n;
while(nleft != 0) {
    count = read(fd, bufp, nleft);    // 略去出错处理
    if (count == 0)                    // EOF
        break;
    nleft -= count;    bufp += count;
}
```

磁盘文件

n = 10K
文件长9K字节

socket

缓存大小：8K字节
n = 10K
socket传送的数据 9K字节

例

```
1  ssize_t rio_readn(int fd, void *usrbuf, size_t n)
2  {
3      size_t nleft = n;
4      ssize_t nread;
5      char *bufp = usrbuf;
6
7      while (nleft > 0) {
8          if ((nread = read(fd, bufp, nleft)) < 0) {
9              if (errno == EINTR) /* Interrupted by sig handler return */
10                 nread = 0;      /* and call read() again */
11              else
12                 return -1;      /* errno set by read() */
13          }
14          else if (nread == 0)
15              break;             /* EOF */
16          nleft -= nread;
17          bufp += nread;
18      }
19      return (n - nleft);        /* Return >= 0 */
20 }
```

信号的pending

- 收到信号oldSig，要过会才能处理
- 还没处理，收到新信号newSig
 - Unix V6++ Process[i].p_sig
 - oldSig被覆盖了
 - Linux
 - 若oldSig ≠ newSig，
newSig不会覆盖oldSig
 - 若oldSig = newSig，
信号处理程序只能执行一次

进程PCB中存放信号的数据结构

Unix V6++

```
int  p_sig;
```

Linux

信号位图 signal

31				...					1	0
0	1	1	0	0	0		
0	0	0	0	0	0	0	0

例：用信号处理程序回收终止子进程的PCB

```
2
3 void handler1(int sig)
4 {
5     int olderrno = errno;
6
7     if ((waitpid(-1, NULL, 0)) < 0)
8         sio_error("waitpid error");
9     Sio_puts("Handler reaped child\n");
10    Sleep(1);
11    errno = olderrno;
12 }
13
14 int main()
15 {
16     int i, n;
17     char buf[MAXBUF];
18
19     if (signal(SIGCHLD, handler1) == SIG_ERR)
20         unix_error("signal error");
```

```
22     /* Parent creates children */
23     for (i = 0; i < 3; i++) {
24         if (Fork() == 0) {
25             printf("Hello from child %d\n", (int)getpid());
26             exit(0);
27         }
28     }
29
30     /* Parent waits for terminal input and then processes it */
31     if ((n = read(STDIN_FILENO, buf, sizeof(buf))) < 0)
32         unix_error("read");
33
34     printf("Parent processing input\n");
35     while (1)
36         ;
37
38     exit(0);
39 }
```

会少回收一个子进程的PCB

```
linux> ./signal1
Hello from child 14073
Hello from child 14074
Hello from child 14075
Handler reaped child
Handler reaped child
CR
Parent processing input
```

正确的代码

```
1 void handler2(int sig)
2 {
3     int olderrno = errno;
4
5     while (waitpid(-1, NULL, 0) > 0) {
6         Sio_puts("Handler reaped child\n");
7     }
8     if (errno != ECHILD)
9         Sio_error("waitpid error");
10    Sleep(1);
11    errno = olderrno;
12 }
```

插曲
信号处理程序
要保护errno

Linux信号的屏蔽

Why ? ?

- blocked
- 例： 进程屏蔽30#信号

31		...			1		0		
0	1	0	0	0	0	
0	0	0	0	0	0	0

- Signal

31		...			1		0		
0	1	1	0	0	0	
0	0	0	0	0	0	0

- 30#信号pending, 直至进程解除对30#信号的屏蔽

Linux支持信号处理的主要数据结构

sigpending不为 0，进程有等待处理的非屏蔽信号：

sigpending != 0的充要条件：

OR $0 < i < 64$ (signal[i] & NOT blocked[i])为真

信号处理方式：**signal_struct** 中的 **action**

_sa_handler，信号处理函数的入口地址

sa_mask，信号掩码

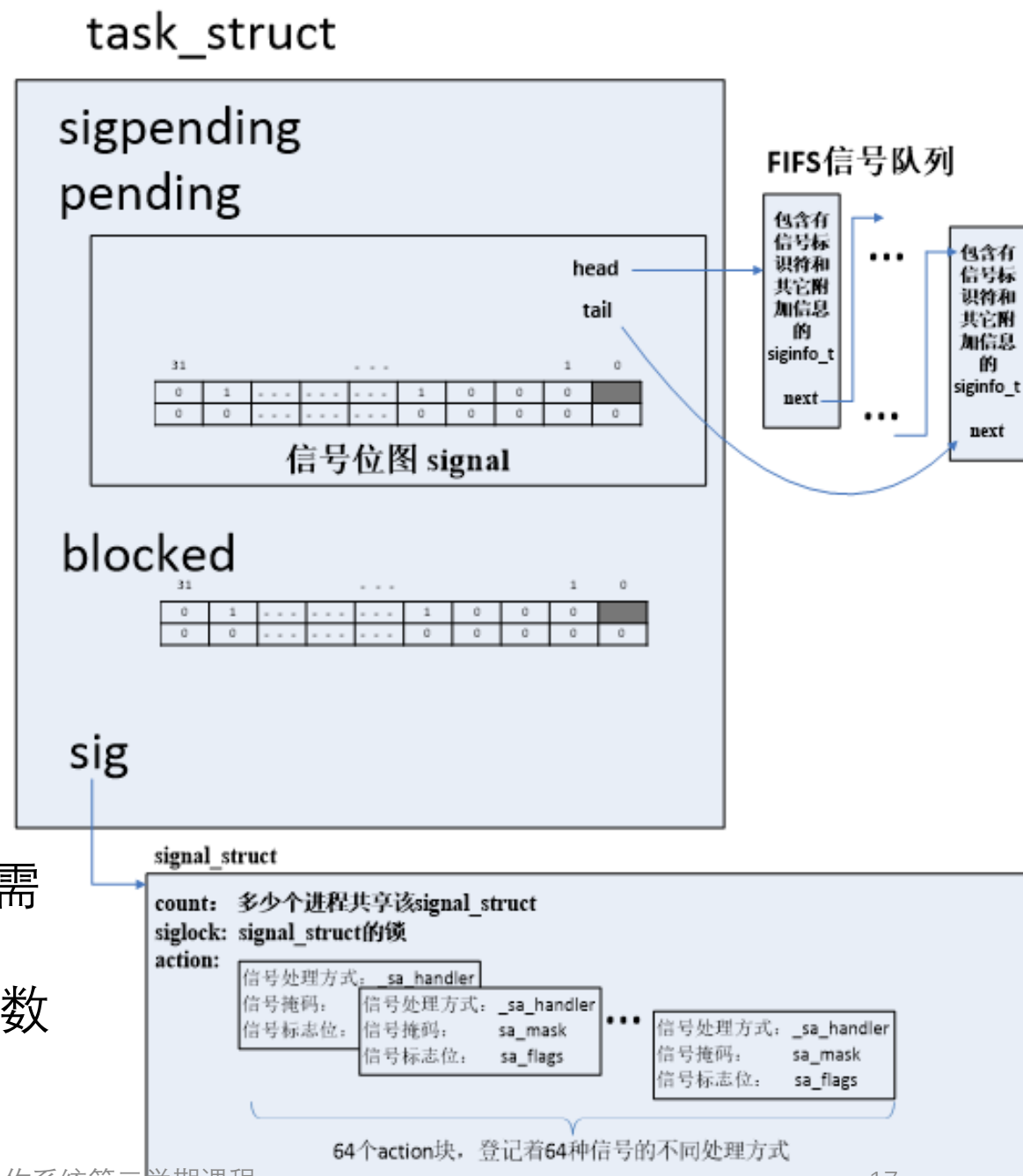
sa_flags，标志位集合。表示处理信号时，系统需要注意些什么。比如：

- SA_SIGINFO：实时信号 1，要喂给信号处理函数待处理数据，用3个入口参数的_sa_sigaction函数
- SA_RESTART：自动重启系统调用

• 2019/4/1

同济大学计算机系 邓蓉

操作系统第二学期课程



阻塞和解除阻塞信号

- 隐式阻塞机制：信号处理程序执行期间，屏蔽当前处理的信号
- 显式阻塞机制：sigprocmask函数……

用这些函数可以禁信号处理函数的执行，还可以等待信号处理函数执行完毕

```
int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);
```

sigprocmask 函数改变当前阻塞的信号集合(8.5.1 节中描述的 blocked 位向量)。具体的行为依赖于 how 的值：
信号屏蔽位图

SIG_BLOCK：把 set 中的信号添加到 blocked 中(blocked=blocked | set)。

SIG_UNBLOCK：从 blocked 中删除 set 中的信号(blocked=blocked &~set)。

SIG_SETMASK：block=set。

如果 oldset 非空，那么 blocked 位向量之前的值保存在 oldset 中。

例：这段代码禁SIGINT信号

```
1      sigset_t mask, prev_mask;
2
3      Sigemptyset(&mask);
4      Sigaddset(&mask, SIGINT);
5
6      /* Block SIGINT and save previous blocked set */
7      Sigprocmask(SIG_BLOCK, &mask, &prev_mask);
8      ∴ // Code region that will not be interrupted by SIGINT
9
10     /* Restore previous blocked set, unblocking SIGINT */
11     Sigprocmask(SIG_SETMASK, &prev_mask, NULL);
```


编写信号处理程序的安全规则

出错的时候，就会错得不可预测和不可重复，这样是很难调试的。一定要防患于未然！

- G0. 处理程序要尽可能简单。避免麻烦的最好方法是保持处理程序尽可能小和简单。例如，处理程序可能只是简单地设置全局标志并立即返回；所有与接收信号相关的处理都由主程序执行，它周期性地检查(并重置)这个标志。
- G1. 在处理程序中只调用异步信号安全的函数。所谓异步信号安全的函数(或简称安全的函数)能够被信号处理程序安全地调用，原因有二：要么它是可重入的(例如只访问局部变量，见 12.7.2 节)，要么它不能被信号处理程序中断。图 8-33 列出了 Linux 保证安全的系统级函数。注意，许多常见的函数(例如 printf、sprintf、malloc 和 exit)都不在此列。

<code>_Exit</code>	<code>fexecve</code>	<code>poll</code>	<code>sigqueue</code>
<code>_exit</code>	<code>fork</code>	<code>posix_trace_event</code>	<code>sigset</code>

errno是库函数为所有进程提供的全局变量，用来存放系统调用的出错码

- G2. 保存和恢复 errno。许多 Linux 异步信号安全的函数都会在出错返回时设置 errno。在处理程序中调用这样的函数可能会干扰主程序中其他依赖于 errno 的部分。解决方法是在进入处理程序时把 errno 保存在一个局部变量中，在处理程序返回前恢复它。注意，只有在处理程序要返回时才有此必要。如果处理程序调用 `_exit` 终止该进程，那么就不需要这样做了。
- G3. 阻塞所有的信号，保护对共享全局数据结构的访问。如果处理程序和主程序或其他处理程序共享一个全局数据结构，那么在访问(读或者写)该数据结构时，你的处理程序和主程序应该暂时阻塞所有的信号。这条规则的原因是从主程序访问一个数据结构 *d* 通常需要一系列的指令，如果指令序列被访问 *d* 的处理程序中断，那么处理程序可能会发现 *d* 的状态不一致，得到不可预知的结果。在访问 *d* 时暂时阻塞信号保证了处理程序不会中断该指令序列。

- G4. 用 `volatile` 声明全局变量。考虑一个处理程序和一个 `main` 函数，它们共享一个全局变量 `g`。处理程序更新 `g`，`main` 周期性地读 `g`。对于一个优化编译器而言，`main` 中 `g` 的值看上去从来没有变化过，因此使用缓存在寄存器中 `g` 的副本来满足对 `g` 的每次引用是很安全的。如果这样，`main` 函数可能永远都无法看到处理程序更新过的值。

可以用 `volatile` 类型限定符来定义一个变量，告诉编译器不要缓存这个变量。例如：

```
volatile int g;
```

`volatile` 限定符强迫编译器每次在代码中引用 `g` 时，都要从内存中读取 `g` 的值。一般来说，和其他所有共享数据结构一样，应该暂时阻塞信号，保护每次对全局变量的访问。

- G5. 用 `sig_atomic_t` 声明标志。在常见的处理程序设计中，处理程序会写全局标志来记录收到了信号。主程序周期性地读这个标志，响应信号，再清除该标志。对于通过这种方式来共享的标志，C 提供一种整型数据类型 `sig_atomic_t`，对它的读和写保证会是原子的（不可中断的），因为可以用一条指令来实现它们：

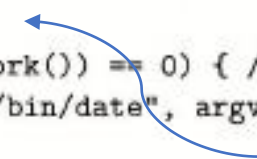
```
volatile sig_atomic_t flag;
```

异常控制流实例 1

父进程创建一个子进程后，将子进程的 PID 塞入一个列表。子进程终止后，SIGCHLD处理程序回收子进程PCB & 将子进程PID从列表中删除

```
1  /* WARNING: This code is buggy! */
2  void handler(int sig)
3  {
4      int olderrno = errno;
5      sigset_t mask_all, prev_all;
6      pid_t pid;
7
8      Sigfillset(&mask_all);
9      while ((pid = waitpid(-1, NULL, 0)) > 0) { /* Reap a zombie child */
10         Sigprocmask(SIG_BLOCK, &mask_all, &prev_all);
11         deletejob(pid); /* Delete the child from the job list */
12         Sigprocmask(SIG_SETMASK, &prev_all, NULL);
13     }
14     if (errno != ECHILD)
15         Sio_error("waitpid error");
16     errno = olderrno;
17 }
```

```
19 int main(int argc, char **argv)
20 {
21     int pid;
22     sigset_t mask_all, prev_all;
23
24     Sigfillset(&mask_all);
25     Signal(SIGCHLD, handler);
26     initjobs(); /* Initialize the job list */
27
28     while (1) {
29         if ((pid = Fork()) == 0) { /* Child process */
30             Execve("/bin/date", argv, NULL);
31         }
32         Sigprocmask(SIG_BLOCK, &mask_all, &prev_all); /* Parent process
33         addjob(pid); /* Add the child to the job list */
34         Sigprocmask(SIG_SETMASK, &prev_all, NULL);
35     }
36     exit(0);
37 }
```



```

18  int main(int argc, char **argv)
19  {
20      int pid;
21      sigset_t mask_all, mask_one, prev_one;
22
23      Sigfillset(&mask_all);
24      Sigemptyset(&mask_one);
25      Sigaddset(&mask_one, SIGCHLD);
26      Signal(SIGCHLD, handler);
27      initjobs(); /* Initialize the job list */
28
29      while (1) {
30          Sigprocmask(SIG_BLOCK, &mask_one, &prev_one); /* Block SIGCHLD */
31          if ((pid = Fork()) == 0) { /* Child process */
32              Sigprocmask(SIG_SETMASK, &prev_one, NULL); /* Unblock SIGCHLD */
33              Execve("/bin/date", argv, NULL);
34          }
35          Sigprocmask(SIG_BLOCK, &mask_all, NULL); /* Parent process */
36          addjob(pid); /* Add the child to the job list */
37          Sigprocmask(SIG_SETMASK, &prev_one, NULL); /* Unblock SIGCHLD */
38      }
39      exit(0);
40  }

```


异常控制流实例 2

1、Linux shell 创建一个前台作业后，等待子进程终止，输出\$提示符……

2、ctrl+c 信号不能杀死 shell 进程。

```
16 int main(int argc, char **argv)
17 {
18     sigset_t mask, prev;
19
20     Signal(SIGCHLD, sigchld_handler);
21     Signal(SIGINT, sigint_handler);
22     Sigemptyset(&mask);
23     Sigaddset(&mask, SIGCHLD);
24
25     while (1) {
26         Sigprocmask(SIG_BLOCK, &mask, &prev); /* Block SIGCHLD */
27         if (Fork() == 0) /* Child */
28             exit(0);
29
30         /* Parent */
31         pid = 0;
32         Sigprocmask(SIG_SETMASK, &prev, NULL); /* Unblock SIGCHLD */
33
34         /* Wait for SIGCHLD to be received (wasteful) */
35         while (!pid)
36             ;
37
38         /* Do some work after receiving SIGCHLD */
39         printf("$");
40     }
41     exit(0);
42 }
```

1、while (!pid)

pause();

2、while (!pid)

sleep(1);

```
2
3 volatile sig_atomic_t pid;
4
5 void sigchld_handler(int s)
6 {
7     int olderrno = errno;
8     pid = waitpid(-1, NULL, 0);
9     errno = olderrno;
10 }
11
12 void sigint_handler(int s)
13 {
14 }
```

判断 pid 的值和入睡该一次完成，中间不能插入信号处理程序
∴ 语句32应该插在while判断和进程入睡之后。也就是，26行~进程入睡，期间所有操作，屏蔽SIGCHLD信号。进程睡着之后，取消SIGCHLD的屏蔽。

用sigsuspend函数等待信号处理函数终止

int sigsuspend(const sigset_t *mask) 睡，置block是原子操作

```
16 int main(int argc, char **argv)
17 {
18     sigset_t mask, prev;
19
20     Signal(SIGCHLD, sigchld_handler);
21     Signal(SIGINT, sigint_handler);
22     Sigemptyset(&mask);
23     Sigaddset(&mask, SIGCHLD);
24
25     while (1) {
26         Sigprocmask(SIG_BLOCK, &mask, &prev); /* Block SIGCHLD */
27         if (Fork() == 0) /* Child */
28             exit(0);
29
30         /* Parent */
31         pid = 0;
32         Sigprocmask(SIG_SETMASK, &prev, NULL); /* Unblock SIGCHLD */
33
34         /* Do some work after receiving SIGCHLD */
35         printf(".");
36         pause();
37
38         /* Do some work after receiving SIGCHLD */
39         printf(".");
40     }
41     exit(0);
42 }
```

```
2
3 volatile sig_atomic_t pid;
4
5 void sigchld_handler(int s)
6 {
7     int olderrno = errno;
8     pid = waitpid(-1, NULL, 0);
9     errno = olderrno;
10 }
11
12 void sigint_handler(int s)
13 {
14 }
```

sigsuspend(&prev); // 入睡、prev→block原子操作，一次完成
//也就是这个函数打包了pause和语句32，
//unblock了SIGCHLD信号