

第8章 异常控制流II: ——信号与非本地跳转

- 教 师： 郑贵滨
- 计算机科学与技术学院
- 哈尔滨工业大学

异常控制流发生在系统的所有层次

■ 异常

- 硬件和操作系统内核程序

■ 进程上下文切换

- 硬件定时器和内核程序

■ 信号

- 内核程序和应用程序

■ 非本地跳转

- 应用程序

以前的内容

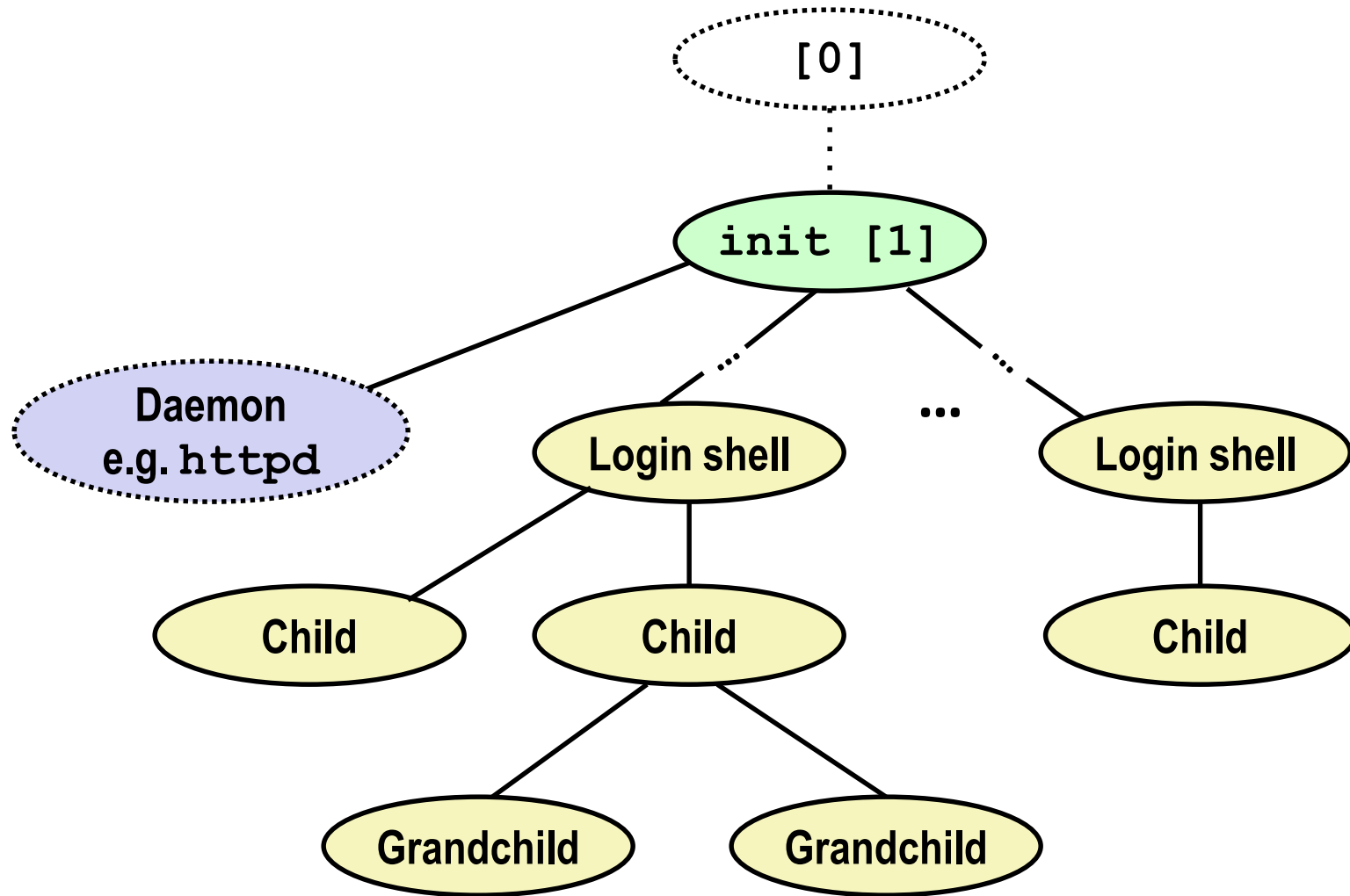
本节内容

课本和补充资料

主要内容

- Shells
- 信号
- 非本地跳转

Linux 进程体系



Note: `ps tree` 命令查看进程树

Shell 程序

- **shell** 是一个交互型应用级程序，代表用户运行其他程序
 - **sh** 最早的shell (Stephen Bourne, AT&T Bell Labs, 1977)
 - **csh** C Shell, 語法有點類似C語言
 - **tcsh** 整合C Shell, 提供更多的功能
 - **bash** 缺省的Linux shell (Bourne Again SHell, bash)
是基于GNU的增强版本sh

```
int main() {
    char cmdline[MAXLINE]; /* command line */
    while (1) {
        /* read */
        printf("> ");
        Fgets(cmdline, MAXLINE, stdin);
        if (feof(stdin))
            exit(0);
        /* evaluate */
        eval(cmdline);
    }
}
```

shellex.c

shell 执行一系列的
读/求值 步骤

读步骤读取用户的
命令行

求值步骤解析命令，
代表用户运行

一个简单的Shell程序：eval函数

```

void eval(char *cmdline){
    char *argv[MAXARGS]; /* Argument list execve() */
    char buf[MAXLINE]; /* Holds modified command line */
    int bg; /* Should the job run in bg or fg? */
    pid_t pid; /* Process id */
    strcpy(buf, cmdline);
    bg = parseline(buf, argv);
    if (argv[0] == NULL)
        return; /* Ignore empty lines */
    if (!builtin_command(argv)) {
        if ((pid = Fork()) == 0) { /* Child runs user job */
            if (execve(argv[0], argv, environ) < 0) {
                printf("%s: Command not found.\n", argv[0]);
                exit(0);
            }
        }
    }
}

```

Parseline函数解析以空格分隔的命令行参数，并构造传递给execve的argv向量

builtin_command函数检查第一个命令行参数是否是一个内置的shell命令

```

        /* Parent waits for foreground job to terminate */父进程等待前台子进程结束
        if (!bg) {
            int status;
            if (waitpid(pid, &status, 0) < 0)
                unix_error("waitfg: waitpid error");
            else
                printf("%d %s", pid, cmdline);
        }
        return;
    } //void eval

```

//shellex.c

简单shell例子的问题

- 在这个例子中shell可以正确等待和回收前台作业
- 但是后台作业呢？
 - 后台作业终止时会成为僵死进程
 - 永远不会被回收，因为shell通常不会终止
 - 将导致内存泄漏

怎么办？

■ 解决办法: 异常控制流

- 在后台进程完成时内核将中断正常处理程序提醒我们
- 在Unix里这种提醒机制叫作信号

主要内容

- Shells
- 信号
- 非本地跳转

Linux信号

- *signal* 就是一条小消息，它通知进程系统中发生了一个某种类型的事件
 - 类似于异常和中断
 - 从内核发送到（有时是在另一个进程的请求下）一个进程
 - 信号类型是用小整数ID来标识的(1-30)
 - 信号中唯一的信息是它的ID和它的到达

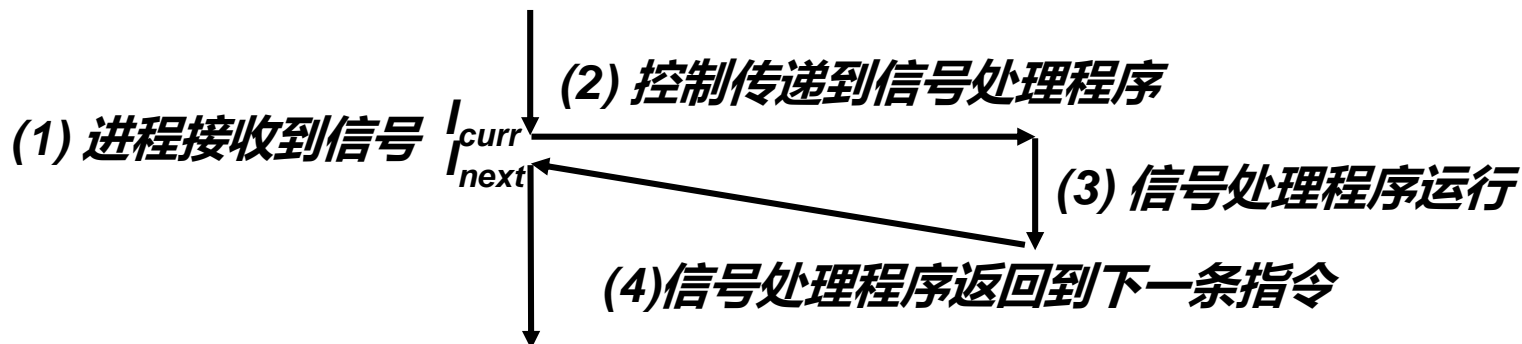
ID	名称	默认行为	相应事件
2	SIGINT	终止	来自键盘的中断
9	SIGKILL	终止	杀死程序(该信号不能被捕获不能被忽略)
11	SIGSEGV	终止	无效的内存引用（段故障）
14	SIGALRM	终止	来自alarm函数的定时器信号
17	SIGCHLD	忽略	一个子进程停止或者终止

信号术语：发送信号

- 内核通过更新目的进程上下文中的某个状态，**发送**（**递送**）一个信号给目的进程
- 发送信号可以是如下原因之一：
 - 内核检测到一个系统事件如除零错误(SIGFPE)或者子进程终止(SIGCHLD)
 - 一个进程调用了`kill`系统调用，显式地请求内核发送一个信号到目的进程
 - ✓ 一个进程可以发送信号给它自己

信号术语: 接收信号

- 当目的进程被内核强迫以某种方式对信号的发送做出反应时，它就**接收**了信号
- 反应的方式:
 - **忽略**这个信号(do nothing)
 - **终止**进程(with optional core dump)
 - 通过执行一个称为信号处理程序 (***signal handler***) 的用户层函数**捕获**这个信号
 - 类似于响应异步中断而调用的硬件异常处理程序



信号术语: 待处理信号和阻塞信号

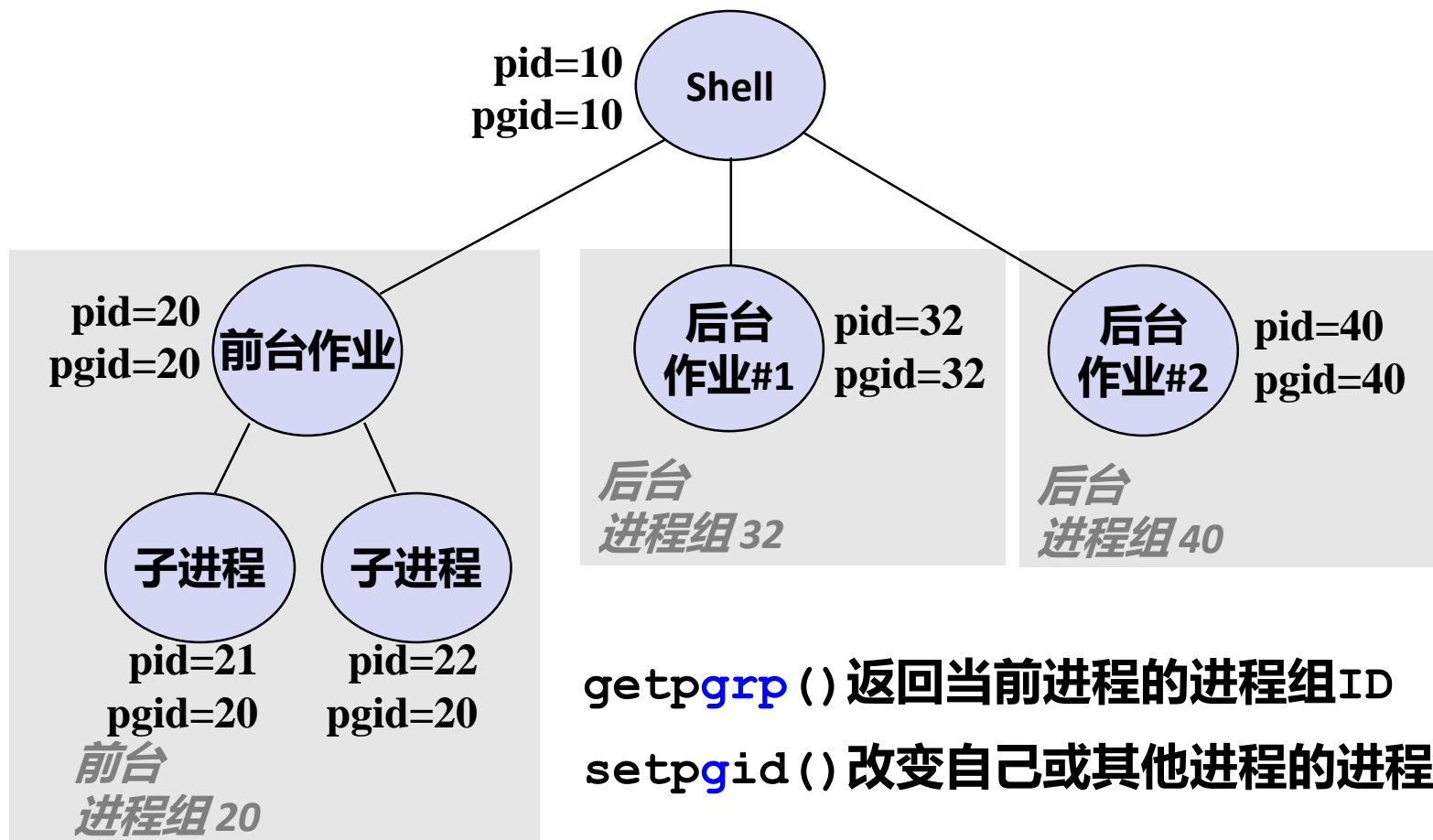
- 一个发出而没有被接收的信号叫做待处理信号 (*pending*)
 - 任何时刻, 一种类型至多只有一个待处理信号
 - Important: 信号不会排队等待
 - 如果一个进程有一个类型为k的待处理信号, 那么任何接下来发送到这个进程的类型为k的信号都会被丢弃
- 一个进程可以选择 **阻塞** 接收某种信号
 - 阻塞的信号仍可以被发送, 但不会被接收, 直到进程取消对该信号的阻塞
- 一个待处理信号最多只能被接收一次

信号术语: 待处理位/阻塞位

- 内核为每个进程维护着待处理位向量 (pending) 和 阻塞位向量 (blocked)
 - **pending**: 待处理信号的集合
 - 若传送了一个类型为k的信号, 内核会设置pending中的第k位
 - 若接收了一个类型为k的信号, 内核将清除pending中的第k位
 - **blocked**: 被阻塞信号的集合
 - 通过 **sigprocmask** 函数设置和清除
 - 也称信号掩码

发送信号: 进程组

- 每个进程只属于一个进程组



用 /bin/kill 程序发送信号

- /bin/kill程序可以向另外的进程或进程组发送任意的信号

■ Examples

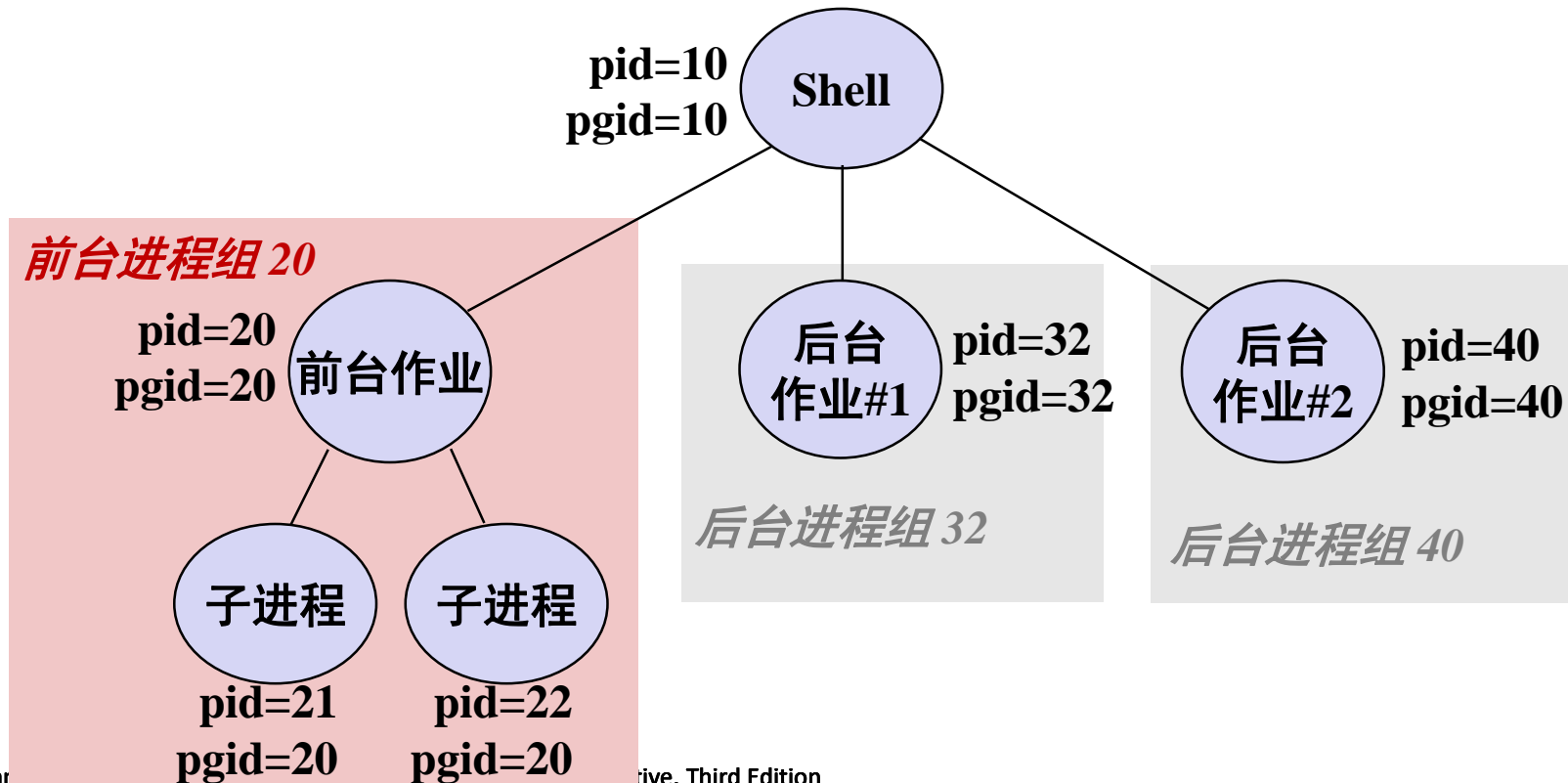
- /bin/kill -9 24818
发送信号9 (SIGKILL) 给进程 24818
- /bin/kill -9 -24817
发送信号SIGKILL给进程组24817中的每个进程
(负的PID会导致信号被发送到进程组PID中的每个进程)

```
linux> ./forks 16
Child1: pid=24818 pgrp=24817
Child2: pid=24819 pgrp=24817
```

```
linux> ps
  PID TTY          TIME CMD
24788 pts/2        00:00:00 tcsh
24818 pts/2        00:00:02 forks
24819 pts/2        00:00:02 forks
24820 pts/2        00:00:00 ps
linux> /bin/kill -9 -24817
linux> ps
  PID TTY          TIME CMD
24788 pts/2        00:00:00 tcsh
24823 pts/2        00:00:00 ps
linux>
```


从键盘发送信号

- 输入 ctrl-c /ctrl-z 会导致内核发送一个 SIGINT /SIGTSTP信号到前台进程组中的每个作业
 - SIGINT – 默认情况是终止前台作业
 - SIGTSTP – 默认情况是停止（挂起）前台作业



ctrl-c /ctrl-z 示例

STAT (进程状态) 图例:

第一个字母:

S: 休眠

T: 停止

R: 运行

第二个字母:

s: 会话领导进程(session leader process)

+: 前台进程组

执行“man ps” 查看详细内容

```
bluefish> ./forks 17
```

```
Child: pid=28108 pgrp=28107
```

```
Parent: pid=28107 pgrp=28107
```

```
<types ctrl-z>
```

```
Suspended
```

```
bluefish> ps w
```

PID	TTY	STAT	TIME	COMMAND
-----	-----	------	------	---------

27699	pts/8	Ss	0:00	-tcsh
-------	-------	----	------	-------

28107	pts/8	T	0:01	./forks 17
-------	-------	---	------	------------

28108	pts/8	T	0:01	./forks 17
-------	-------	---	------	------------

28109	pts/8	R+	0:00	ps w
-------	-------	----	------	------

```
bluefish> fg
```

```
./forks 17
```

```
<types ctrl-c>
```

```
bluefish> ps w
```

PID	TTY	STAT	TIME	COMMAND
-----	-----	------	------	---------

27699	pts/8	Ss	0:00	-tcsh
-------	-------	----	------	-------

28110	pts/8	R+	0:00	ps w
-------	-------	----	------	------

用 kill 函数发送信号

```

void fork12()
{
    pid_t pid[N];
    int i;
    int child_status;

    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0) {
            /* Child: Infinite Loop */
            while(1) ;
        }

    for (i = 0; i < N; i++) {
        printf("Killing process %d\n",
            pid[i]);
        kill(pid[i], SIGINT);
    }
}

```

```

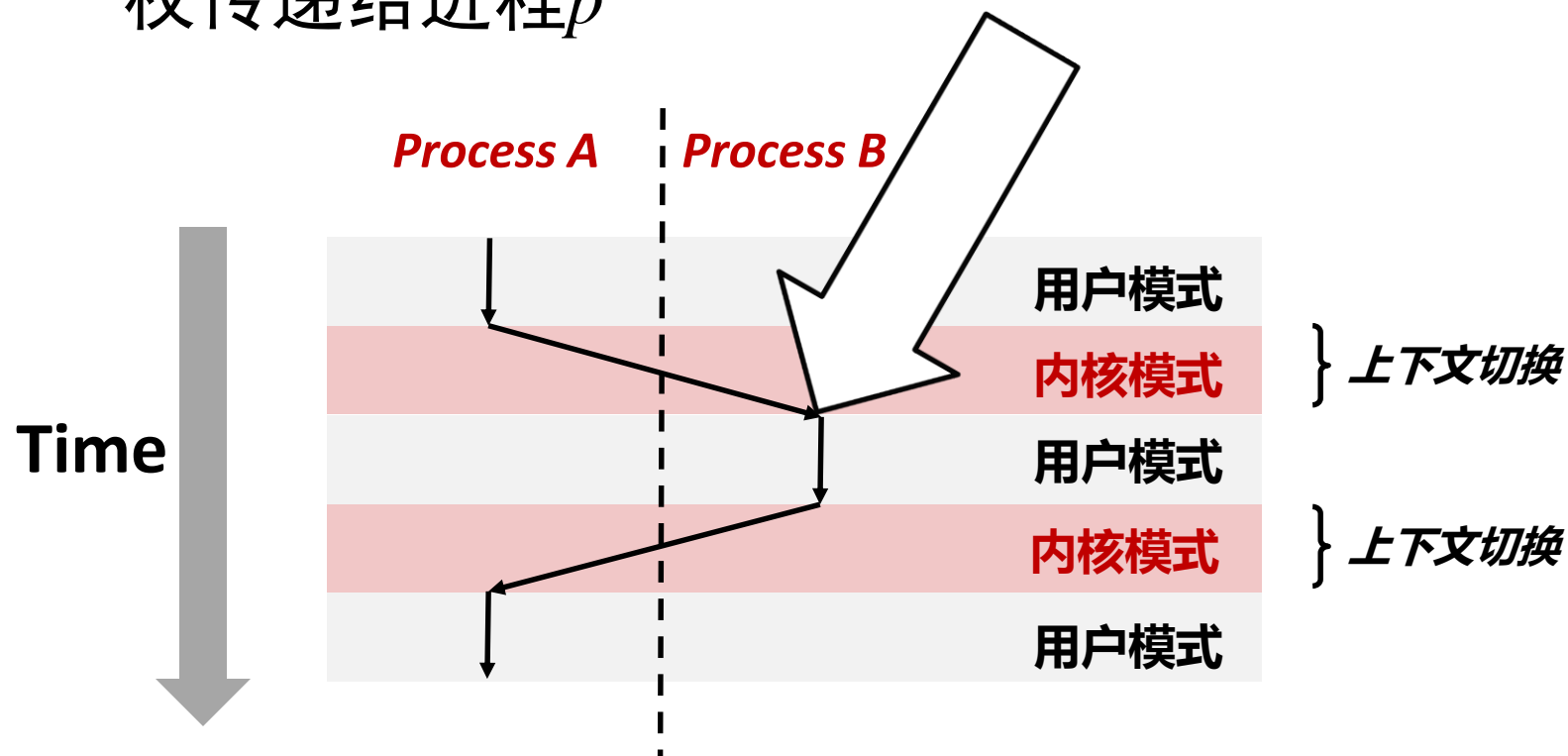
    for (i = 0; i < N; i++) {
        pid_t wpid = wait(&child_status);
        if (WIFEXITED(child_status))
            printf("Child %d terminated
                with exit status %d\n",
                    wpid,
                    WEXITSTATUS(child_status));
        else
            printf("Child %d terminated
                abnormally\n", wpid);
    }
}

```

forks.c

接收信号

- 假设内核正在从异常处理程序返回，并准备将控制权传递给进程 p



接收信号

- 假设内核正在从异常处理程序返回，并准备将控制权传递给进程

p

 - 内核检查 $\text{pnb} = \text{pending} \ \& \ \sim\text{blocked}$
 - 进程

p

的未被阻塞的待处理信号的集合
 - If ($\text{pnb} == 0$) //如果集合为空
 - 将控制传递到 p 的逻辑控制流中的下一条指令
 - else//不为空
 - 选择集合

pnb

中最小的非零位

k

，强制

p

接收信号

k
 - ✓ 收到信号会触发进程

p

采取某种行为
 - 对所有的非零

k

，重复上述操作
 - 控制传递到

p

的逻辑控制流中的下一条指令

默认行为

- 每个信号类型都有一个预定义默认行为，是下面中的一种：
 - 进程终止
 - 进程停止（挂起）直到被SIGCONT信号重启
 - 进程忽略该信号

设置信号处理程序

■ `signal`函数

■ 函数原型:

```
handler_t *signal(int signum, handler_t  
*handler)
```

■ 功能:

可以修改和信号相关联的默认行为

■ `handler`的不同取值:

- `SIG_IGN`: 忽略类型为`signum`的信号
- `SIG_DFL`: 类型为 `signum`的信号行为恢复为默认行为
- `handler` 是用户自定义函数的地址，这个函数称为信号处理程序

信号处理...

- 只要进程接收到类型为 `signum` 的信号就会调用信号处理程序
- 设置信号处理程序：使用 `signal` 函数，设定指定信号的处理程序(地址)，从而改变默认行为
- 捕获信号：调用信号处理程序称为捕获信号
- 处理信号：执行信号处理程序称为处理信号
- 当处理程序执行 `return` 时，控制会传递到控制流中被信号接收所中断的指令处

用信号处理程序捕获SIGINT信号

```
void sigint_handler(int sig) /* SIGINT handler */
{
    printf("So you think you can stop the bomb with ctrl-c, do you?\n");
    sleep(2);
    printf("Well...\n");
    fflush(stdout);
    sleep(1);
    printf("OK. :-)\n");
    exit(0);
}

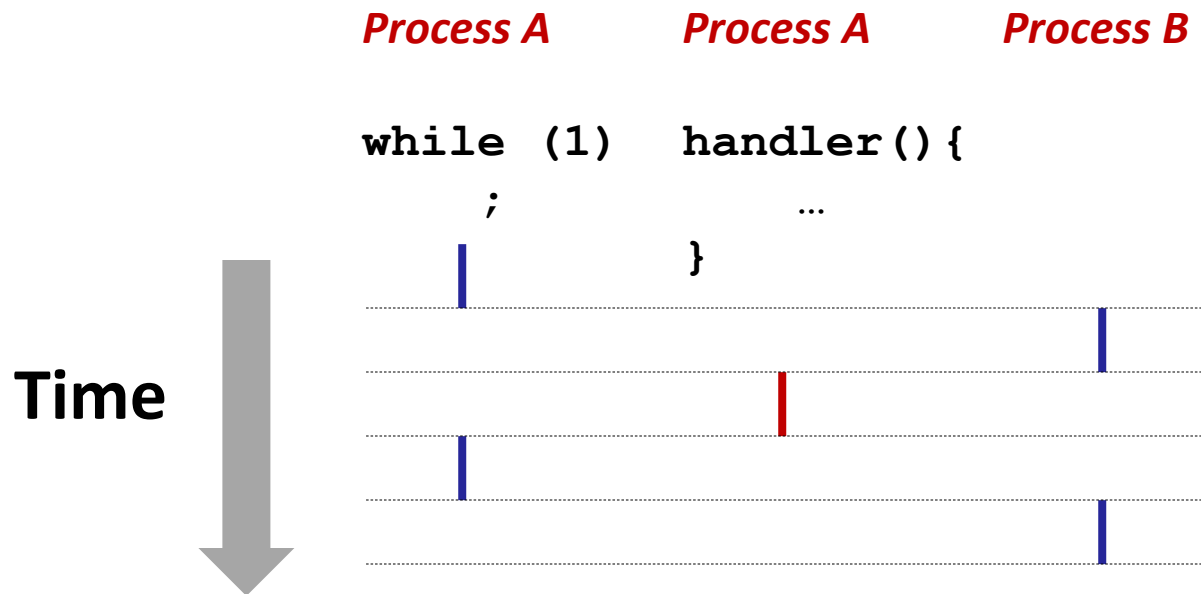
int main()
{
    /* Install the SIGINT handler */
    if (signal(SIGINT, sigint_handler) == SIG_ERR)
        unix_error("signal error");

    /* Wait for the receipt of a signal */
    pause(); //
    return 0;
}
```

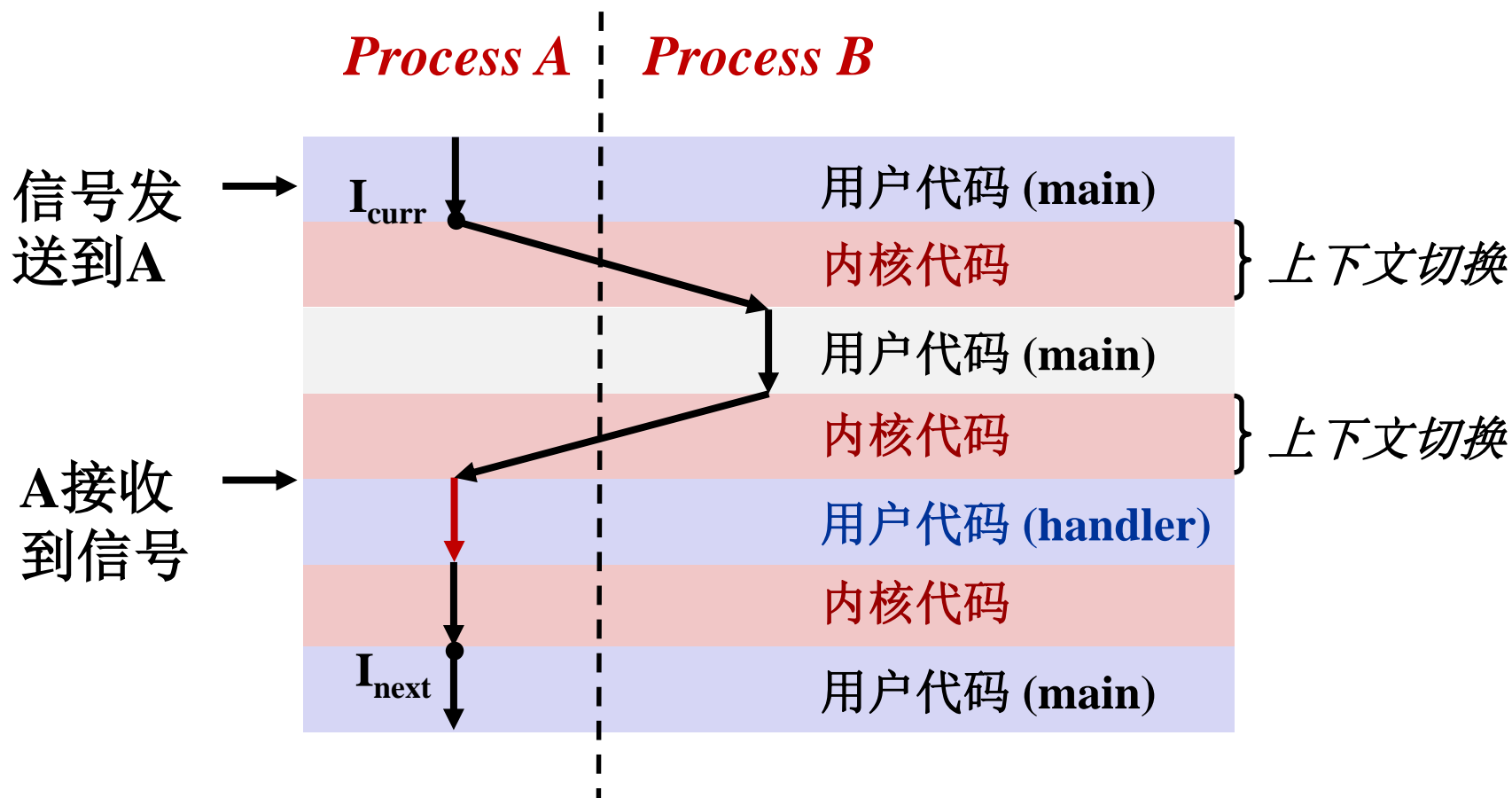
sigint.c

作为并发流的信号处理程序

- 信号处理程序是与主程序同时运行的独立逻辑流（不是进程）

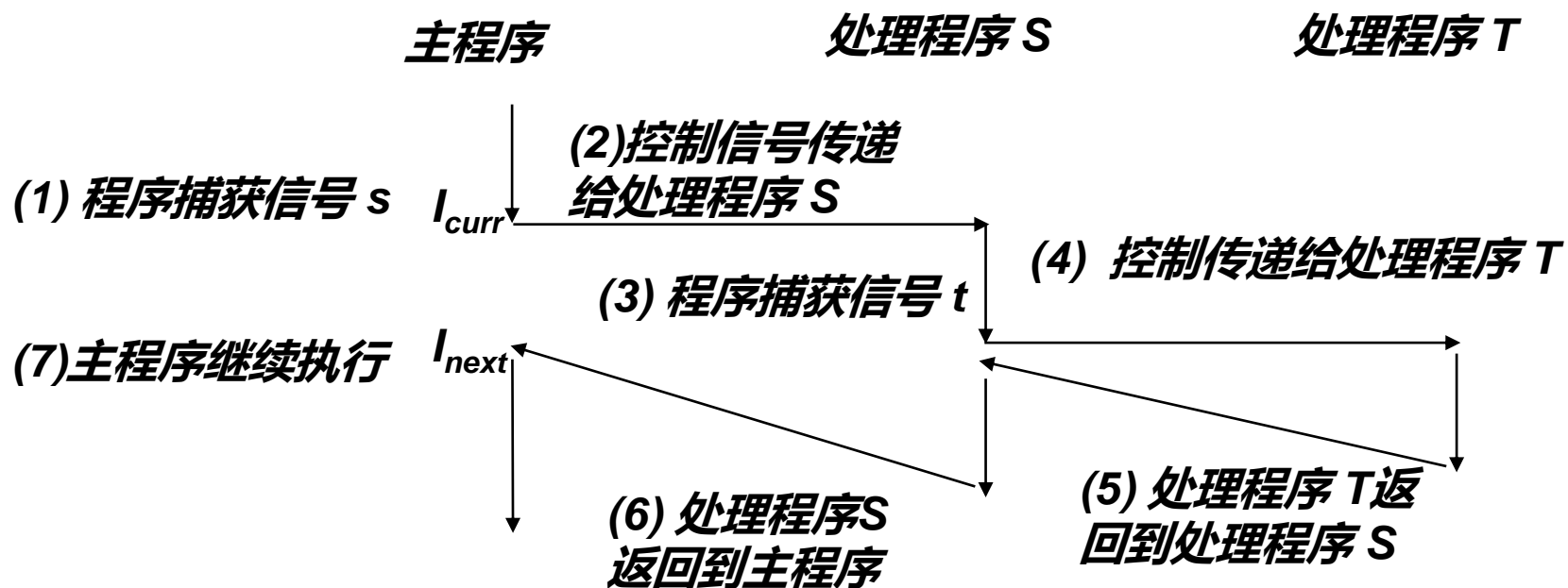


另一个角度看作为并发流的信号处理程序



嵌套的信号处理程序

- 信号处理程序可以被其他信号处理程序中中断



阻塞和解除阻塞信号

■ 隐式阻塞机制

- 内核默认阻塞与当前正在处理信号类型相同的待处理信号
- 如，一个SIGINT 信号处理程序不能被另一个 SIGINT信号中断（此时另一个SIGINT信号被阻塞）

■ 显示阻塞和解除阻塞机制

- `sigprocmask` 函数及其辅助函数可以明确地阻塞/解除阻塞选定的信号
- 辅助函数——信号集合操作
 - `sigemptyset`: 初始化set为空集合
 - `sigfillset`: 把每个信号都添加到set中
 - `sigaddset`: 把指定的信号`signum`添加到set
 - `sigdelset`: 从set中删除指定的信号`signum`

临时阻塞接收信号

```
sigset_t mask, prev_mask;
```

```
Sigemptyset(&mask);
```

```
Sigaddset(&mask, SIGINT);
```

```
/* Block SIGINT and save previous blocked set */
```

```
Sigprocmask(SIG_BLOCK, &mask, &prev_mask); //增加阻塞mask中的  
信号
```

```
/* Code region that will not be interrupted by SIGINT */
```

```
... ..
```

```
/* Restore previous blocked set, unblocking SIGINT */
```

```
Sigprocmask(SIG_SETMASK, &prev_mask, NULL);
```

```
//解除对mask中信号的阻塞
```

```
Sigprocmask(SIG_UNBLOCK, &mask, &prev_mask);
```

```
//仅仅阻塞mask中的信号
```

```
Sigprocmask(SIG_SETMASK, &mask, &prev_mask);
```

安全的信号处理

- 信号处理程序很麻烦，因为它们和主程序：
 - 并发地运行
 - 共享全局数据结构——共享的数据结构可能被破坏
- 第12章 详细讲述并发编程
- 这里仅给出一些原则

编写处理程序的原则

- G0: 处理程序尽可能简单
 - e.g., 简单设置全局标志并立即返回
- G1: 在处理程序中只调用异步信号安全的函数
 - **printf, sprintf, malloc, and exit are not safe!**
- G2: 保存和恢复 `errno`
 - 确保其他处理程序不会覆盖当前的 `errno`
- G3: 阻塞所有信号，保护对共享全局数据结构的访问
 - 避免可能的冲突
- G4: 用 `volatile` 声明全局变量
 - 强迫编译器从内存中读取引用的值
- G5: 用 `sig_atomic_t` 声明标志
 - *原子型标志*: 只适用于单个的读或者写，不适用 `flag++` 或 `flag=flag+10` 这样的更新(e.g. `flag = 1`, not `flag++`)
 - 采用这种方式声明的标志不需要类似其他全局变量的保护

异步信号安全

- **异步信号安全**的函数：函数要么是可重入的（如只访问局部变量）要么不能被信号处理程序中中断
- 可重入函数(Reentrant function):不引用任何共享数据
- Posix保证安全的 117 个异步信号安全的函数
 - Source: “man 7 signal”
 - 常见的函数包括:
 - `_exit`, `write`, `wait`, `waitpid`, `sleep`, `kill`
 - 常见的函数里不包括:
 - `printf`, `sprintf`, `malloc`, `exit`
 - `write` 函数是信号处理程序中**唯一安全的输出函数**

开发安全的输出函数

- 在处理程序中使用来自 `csapp.c` 的可重入 SIO (安全的 I/O 库)
 - `ssize_t sio_puts(char s[])` /* 输出一个字符串 */
 - `ssize_t sio_putl(long v)` /* 输出一个 long 类型数 */
 - `void sio_error(char s[])` /* 打印一条错误信息并终止 */

```
void sigint_handler(int sig) /*sigint.c中SIGINT处理程序的一个安全版本*/
{
    Sio_puts("So you think you can stop the bomb with ctrl-c, do
you?\n");
    sleep(2);
    Sio_puts("Well...\n");
    sleep(1);
    Sio_puts("OK. :-)\n");
    _exit(0);
}
```

sigintsafe.c

正确的信号处理

- 待处理信号是不排队的
 - 对每种信号类型，
pending位向量只有1位与之对应
 - 因此每种类型最多只能有
1个未处理信号
- 不能用信号来对其他进程中发生的事件计数，
如子程序的终止

```
whaleshark> ./forks 14
Handler reaped child
23240
```

```
Handler reaped child
```

```
int ccount = 0;
void child_handler(int sig) {
    int olderrno = errno;
    pid_t pid;
    if ((pid = wait(NULL)) < 0)
        Sio_error("wait error");
    ccount--;
    Sio_puts("Handler reaped child ");
    Sio_putl((long)pid);
    Sio_puts(" \n");
    sleep(1);
    errno = olderrno;
}
void fork14() {
    pid_t pid[N];
    int i;
    ccount = N;
    Signal(SIGCHLD, child_handler);

    for (i = 0; i < N; i++) {
        if ((pid[i] = Fork()) == 0) {
            Sleep(1);
            exit(0); /* Child exits */
        }
    }
    while (ccount > 0) ; /* Parent spins */
}
```

正确的信号处理

■ 必须回收所有终止的子进程

- 方法：将 `wait` 放入 `while` 循环，从而回收所有终止的子进程

```
void child_handler2(int sig)
{
    int olderrno = errno;
    pid_t pid;
    while ((pid = wait(NULL)) > 0) {
        ccount--;
        Sio_puts("Handler reaped child ");
        Sio_putl((long)pid);
        Sio_puts(" \n");
    }
    if (errno != ECHILD)
        Sio_error("wait error");
    errno = olderrno;
}
```

```
linux> ./forks 15
```

```
Handler reaped child 23246
```

```
Handler reaped child 23247
```

```
Handler reaped child 23248
```

```
Handler reaped child 23249
```

```
Handler reaped child 23250
```

```
linux>
```

可移植的信号处理

- 不同的Unix版本有不同的信号处理语义
 - 一些老系统在信号被捕获后就恢复为该信号的默认行为
 - 一些被中断的系统调用在信号处理程序返回时也立即返回，并将 `errno` 设置为 `EINTR`
 - 一些系统不阻塞与处理程序同类型的信号
- 解决办法: **sigaction** 函数，可明确指定信号处理语义

```
handler_t *Signal(int signum, handler_t *handler){
    struct sigaction action, old_action;

    action.sa_handler = handler;
    sigemptyset(&action.sa_mask); /* Block sigs of type being handled */
    action.sa_flags = SA_RESTART; /* Restart syscalls if possible */

    if (sigaction(signum, &action, &old_action) < 0)
        unix_error("Signal error");
    return (old_action.sa_handler);
}
```

同步流以避免竞争（并发错误）

- 具有细微同步错误的简单shell，因为它假定父进程在子进程之前运行

```
int main(int argc, char **argv){
    int pid;
    sigset_t mask_all, prev_all;
    Sigfillset(&mask_all);
    Signal(SIGCHLD, handler);
    initjobs(); /* Initialize the job list */
    while (1) {
        if ((pid = Fork()) == 0) { /* Child */
            Execve("/bin/date", argv, NULL);
        }
        Sigprocmask(SIG_BLOCK, &mask_all, &prev_all); /* Parent */
        addjob(pid); /* Add the child to the job list */
        Sigprocmask(SIG_SETMASK, &prev_all, NULL);
    }
    exit(0);
}
```

procmask1.c

同步流以避免竞争（并发错误）

■ 一个简单shell中的SIGCHLD 处理程序

```

void handler(int sig)
{
    int olderrno = errno;
    sigset_t mask_all, prev_all;
    pid_t pid;

    Sigfillset(&mask_all);
    while ((pid = waitpid(-1, NULL, 0)) > 0) { /* Reap child */
        Sigprocmask(SIG_BLOCK, &mask_all, &prev_all);
        deletejob(pid); /* Delete the child from the job list */
        Sigprocmask(SIG_SETMASK, &prev_all, NULL);
    }
    if (errno != ECHILD)
        Sio_error("waitpid error");
    errno = olderrno;
}

```

procmask1.c

正确的Shell 程序：消除竞争/避免并发错误

```

int main(int argc, char **argv){
    int pid;
    sigset_t mask_all, mask_one, prev_one;

    Sigfillset(&mask_all);
    Sigemptyset(&mask_one);
    Sigaddset(&mask_one, SIGCHLD);
    Signal(SIGCHLD, handler);
    initjobs(); /* Initialize the job list */

    while (1) {
        Sigprocmask(SIG_BLOCK, &mask_one, &prev_one); /* Block SIGCHLD */
        if ((pid = Fork()) == 0) { /* Child process */
            Sigprocmask(SIG_SETMASK, &prev_one, NULL); /* Unblock SIGCHLD */
            Execve("/bin/date", argv, NULL);
        }
        Sigprocmask(SIG_BLOCK, &mask_all, NULL); /* Parent process */
        addjob(pid); /* Add the child to the job list */
        Sigprocmask(SIG_SETMASK, &prev_one, NULL); /* Unblock SIGCHLD */
    }
    exit(0);
}

```


如何实现等待(某事件)的功能？

- 程序需要等待某个事件/信号的出现、处理结束等
例：等待子进程结束并回收，shell等待前台作业结束...
 - 方法1：程序显式地等待SIGCHLD信号到达

```
volatile sig_atomic_t pid;
```

```
void sigchld_handler(int s)
```

```
{
```

```
    int olderrno = errno;
```

```
    pid = Waitpid(-1, NULL, 0); /* Main is waiting for nonzero pid */
```

```
    errno = olderrno;
```

```
}
```

waitforsignal.c

显式地等待信号...

```
int main(int argc, char **argv) {
    sigset_t mask, prev;
    Signal(SIGCHLD, sigchld_handler);
    Signal(SIGINT, sigint_handler);
    Sigemptyset(&mask);
    Sigaddset(&mask, SIGCHLD);
    while (1)
    {
        Sigprocmask(SIG_BLOCK, &mask, &prev); /*显式地阻塞 SIGCHLD
*/
        if (Fork() == 0) /* Child */
            exit(0);
        /* Parent */
        pid = 0;
        Sigprocmask(SIG_SETMASK, &prev, NULL); /* 显式地解除阻塞
SIGCHLD */

        while (!pid) ; /* Wait for SIGCHLD to be received
(wasteful!) */
        /* Do some work after receiving SIGCHLD */
        printf(".");
    }
}
```

类似于一个等待前台
作业终止的shell

显式地等待信号

- 程序正确，但循环很浪费
- 改进:

```
while (!pid) /* Race! */  
    pause();
```

在while测试之后、pause之前
收到SIGCHLD信号，
则pause会永远睡眠！！！！

```
while (!pid) /* Too slow! */  
    sleep(1);
```

在while测试之后、sleep之前
收到SIGCHLD信号，
也要睡眠1秒钟之后才能继续

- 真正的解决方法—— **sigsuspend**

用 `sigsuspend` 等待信号

- `int sigsuspend(const sigset_t *mask)`
- 等价于下述代码的原子(不可中断的) 版本:

```
sigprocmask(SIG_SETMASK, &mask, &prev);  
pause();  
sigprocmask(SIG_SETMASK, &prev, NULL);
```

用 sigsuspend等待信号

```

int main(int argc, char **argv) {
    sigset_t mask, prev;
    Signal(SIGCHLD, sigchld_handler);
    Signal(SIGINT, sigint_handler);
    Sigemptyset(&mask);
    Sigaddset(&mask, SIGCHLD);
    while (1) {
        Sigprocmask(SIG_BLOCK, &mask, &prev); /* Block SIGCHLD */
        if (Fork() == 0) /* Child */
            exit(0);
        /* Wait for SIGCHLD to be received */
        pid = 0;
        while (!pid)
            Sigsuspend(&prev);
        /* Optionally unblock SIGCHLD */
        Sigprocmask(SIG_SETMASK, &prev, NULL);
        /* Do some work after receiving SIGCHLD */
        printf(".");
    }
    exit(0);
}

```

主要内容

- Shells
- 信号
- 非本地跳转
 - 参考书本

非本地跳转: `setjmp/longjmp`

- 强大的（但危险的）用户级机制，将控制转移到任意位置
 - 控制转移时不遵守调用/返回规则
 - 对错误恢复和信号处理程序有好处
- `int setjmp(jmp_buf envbuf)`
 - 必须在`longjmp`之前被调用
 - 在`envbuf`中保存当前调用环境，供后续 `longjmp`使用
 - 被调用一次，返回多次
- 执行结果
 - 在`envbuf`中保存当前调用环境：包括寄存器、栈指针和程序计数器
 - 返回 0
 - 返回值不能给变量赋值： `rc = setjmp(env); //Error`

setjmp/longjmp (cont)

■ `void longjmp(jmp_buf envbuf, int i)`

■ 含义

- 从**envbuf**中恢复调用环境，并触发 **setjmp** 返回
- 返回数值**i**

■ 在**setjmp**之后被调用

■ 被调用一次，从不返回

■ **longjmp** 的执行

- 从**envbuf**中恢复寄存器内容（栈指针、基址指针、程序计数器）
- 返回值**i**在 **%eax**中
- 跳转至保存在**envbuf**中的PC所指示的位置

setjmp/longjmp Example

- 目标:从深层嵌套函数调用中直接返回

```
/* Deeply nested function foo */  
void foo(void)  
{  
    if (error1)  
        longjmp(buf, 1);  
    bar();  
}  
  
void bar(void)  
{  
    if (error2)  
        longjmp(buf, 2);  
}
```

setjmp/longjmp Example (cont)

```
jmp_buf buf;
int error1 = 0;
int error2 = 1;
void foo(void), bar(void);

int main()
{
    switch(setjmp(buf)) {
        case 0:
            foo();
            break;
        case 1:
            printf("Detected an error1 condition in foo\n");
            break;
        case 2:
            printf("Detected an error2 condition in foo\n");
            break;
        default:
            printf("Unknown error condition in foo\n");
    }
    exit(0);
}
```

非本地跳转的局限

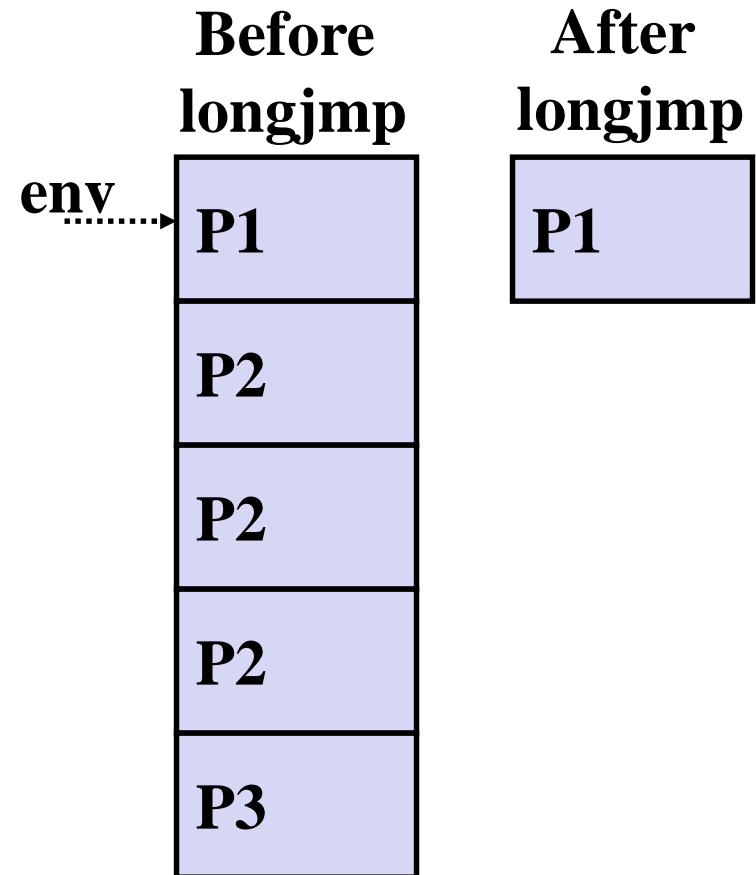
- 工作在堆栈规则下
 - 只能跳到被调用但尚未完成的函数环境中

```

jmp_buf env;
P1(){
  if (setjmp(env)) {
    /* Long Jump to here */
  } else {
    P2();
  }
}

P2(){ ... P2(); ... P3(); }

P3(){ longjmp(env, 1);}
  
```



非本地跳转的局限(cont.)

- 工作在堆栈规则下
 - 只能跳到被调用但尚未完成的函数环境里

```

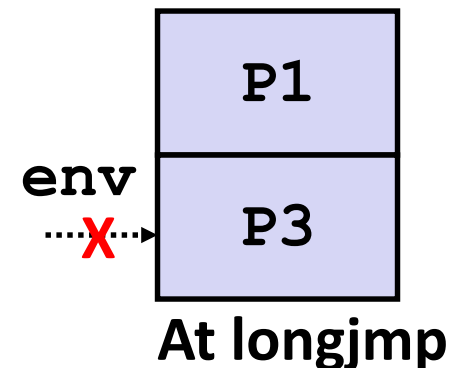
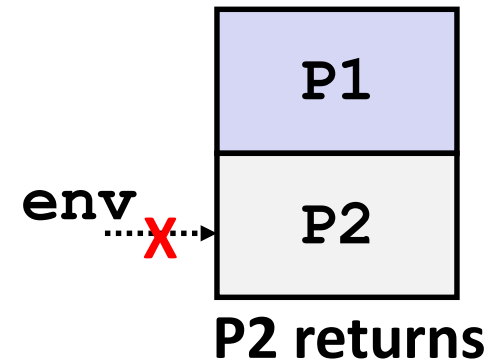
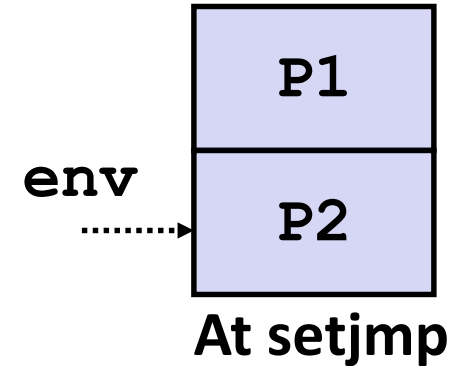
jmp_buf env;

P1(){ P2(); P3();}

P2(){
  if (setjmp(env)) {
    /* Long Jump to here */
  }
}

P3(){ longjmp(env, 1);}

```



综合：利用ctrl-c来重启自身的程序

```
#include "csapp.h"
sigjmp_buf buf;
void handler(int sig)
{
    siglongjmp(buf, 1);
}
int main()
{
    if (!sigsetjmp(buf, 1)) {
        Signal(SIGINT, handler);
        Sio_puts("starting\n");
    }
    else
        Sio_puts("restarting\n");

    while(1) {
        Sleep(1);
        Sio_puts("processing...\n");
    }
    exit(0); /* Control never reaches here */
}
```

```
greatwhite> ./restart
starting
processing...
processing...
processing...
restarting ← Ctrl-c
processing...
processing...
restarting ← Ctrl-c
processing...
processing...
processing...
```

restart.c

小结

- 信号提供了进程级的异常处理
 - 可由用户程序产生
 - 可以通过设定信号处理程序，重新定义信号的行为
 - 要小心书写信号处理程序
- 非本地跳转提供进程内的异常控制流（用户级）
 - 在堆栈规则内