信号和非本地跳转

Signals and Nonlocal Jumps

本章内容

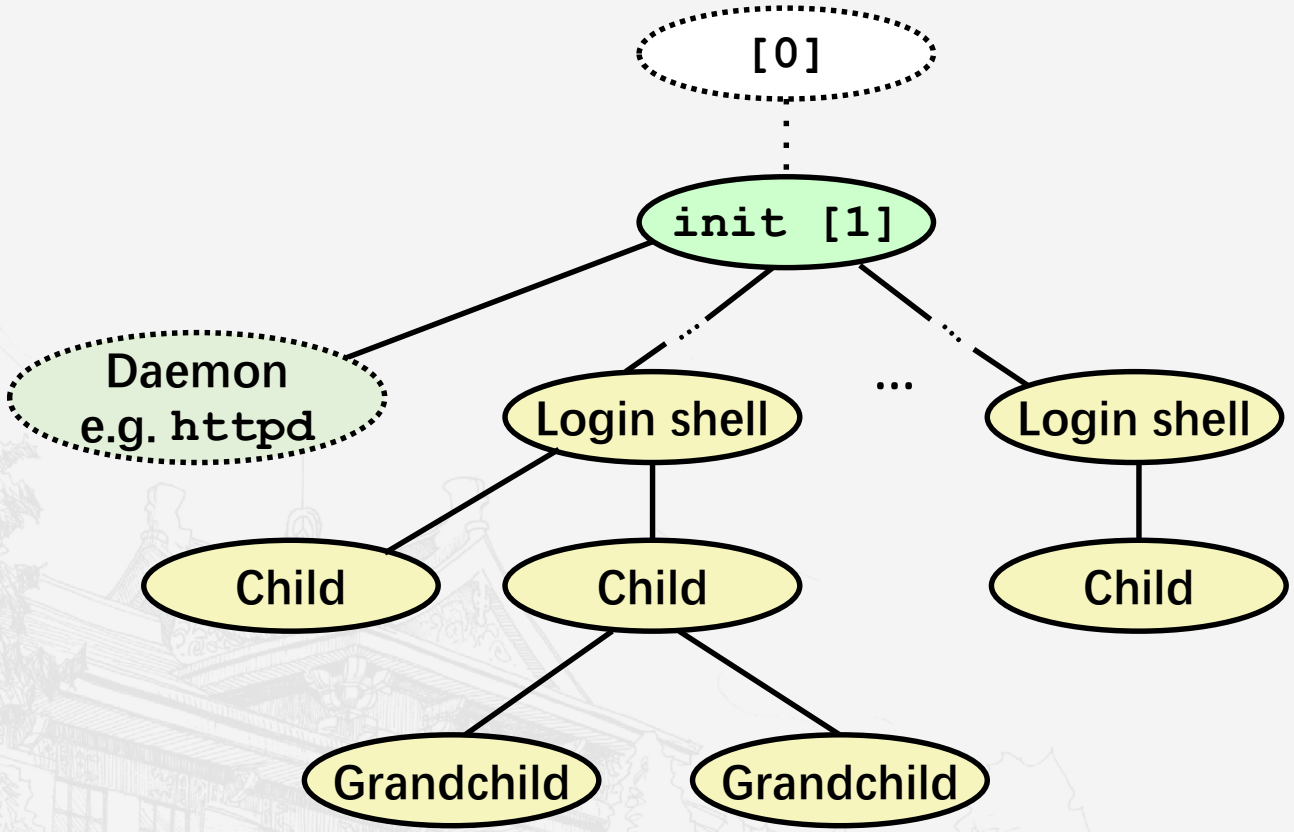
Topic

- 命令解释器
- 信号
- 非本地跳转

命令解释器

Command Interpreter

Linux下进程的层次结构



在Linux下可以试用pstree命令查看进程的层次结构

命令解释器

Command Interpreter

Shell程序

- shell （外壳），为用户提供操作界面的软件，为用户运行应用程序的程序。
- 常见的shell
  - sh　　原始的 Unix Shell（Stephen Bourne, AT&T Bell Labs, 1977）
  - csh/tcsh BSD Unix C shell
  - bash　“Bourne-Again” Shell（Linux默认使用的shell）
  - dash　Debian Almquist shell，是从 NetBSD 派生而来的轻量级 shell，专门为 Debian 设计
  - zsh　macOS默认使用的shell

命令解释器

Command Interpreter

一个简单的shell程序的实现

```c
int main()
{
    /* command line */
    char cmdline[MAXLINE];

    while (1) {
        /* read */
        printf("> ");
        Fgets(cmdline, MAXLINE, stdin);
        if (feof(stdin))
            exit(0);

        /* evaluate */
        eval(cmdline);
    }
}
```

```c
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);
            }
        }

        /* Parent waits for foreground job to terminate */
        if (!bg) {
            int status;
            if (waitpid(pid, &status, 0) < 0)
                unix_error("waitfg: waitpid error");
        }
        else
            printf("%d %s", pid, cmdline);
    }
    return;
}
```

程序中存在的问题

- 示例 的shell程序能够正确等待并回收前台作业
- 但是后台作业呢?
  - 当它们终止时会成为僵尸
  - 永远不会被收割，因为 shell（通常）不会终止
  - 将创建一个内存泄漏，可能使内核耗尽内存

使用异常控制流解决这个问题

- 解决方案：异常控制流
  - 内核将中断常规处理以在后台进程完成时提醒我们
  - 在类Unix 系统中，这种提醒机制称为**信号**

本章内容

Topic

- 命令解释器
- 信号
- 非本地跳转

信号

Signals

- **信号**是一个简短的消息，通知进程系统中发生了某种类型的事件
  - 类似于异常和中断
  - 从内核发送（有时是由另一个进程的请求触发）到一个进程
  - 信号类型由小整数 ID（1-30）进行标识
  - 信号中唯一的信息是它的 ID 和它到达的事实

ID	Name	Default Action	Corresponding Event
2	SIGINT	Terminate	User typed ctrl-c
9	SIGKILL	Terminate	Kill program (cannot override or ignore)
11	SIGSEGV	Terminate	Segmentation violation
14	SIGALRM	Terminate	Timer signal
17	SIGCHLD	Ignore	Child stopped or terminated

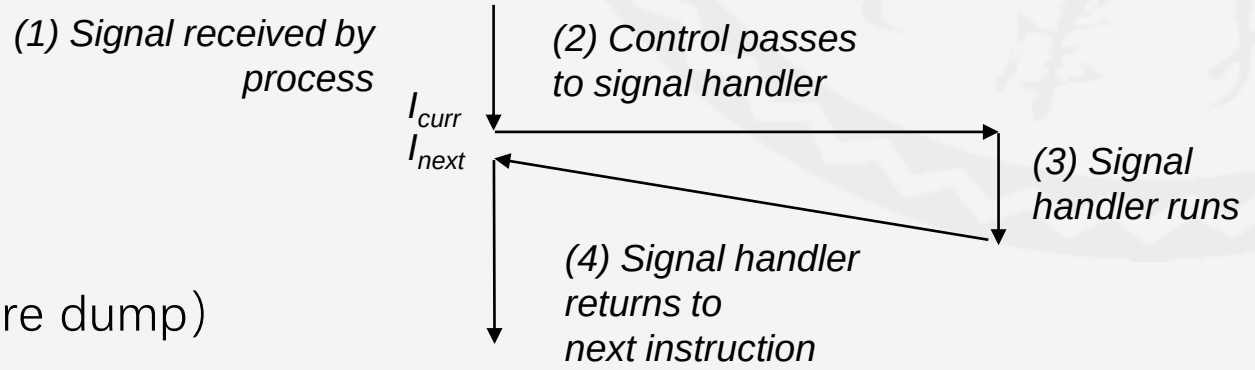
发送信号

- 内核通过更新目标进程的上下文中的某些状态向目标进程发送（传递）信号
- 内核发送信号的原因有以下几种：
  - 内核检测到系统事件，如：除零错误（SIGFPE）或子进程的终止（SIGCHLD）
  - 另一个进程调用了 kill 系统调用，显式请求内核向目标进程发送信号

接收信号

- 当内核强制目标进程对信号的传递做出某种反应时，目标进程会收到一个信号

- 一些可能的响应方式：
  - **忽略**信号（什么都不做）
  - **终止**进程（可以生成核心转储core dump）
  - 通过执行称为信号处理程序的用户级函数**捕获**信号
    - 类似于硬件异常处理程序在响应异步中断时被调用



挂起（Pending）和阻塞（Blocked）信号

- 如果信号已发送但尚未接收，则该信号处于挂起状态
  - 每种类型的信号最多只能有一个挂起的信号
  - 重要的是：信号不是排队的
  - 如果一个进程有一个类型为 k 的挂起信号，那么发送到该进程的后续类型为 k 的信号将被丢弃
- 进程可以阻塞接收某些信号
  - 被阻塞的信号可以被传递，但直到信号解除阻塞时才会被接收
- 挂起的信号最多只能接收一次

挂起和阻塞位

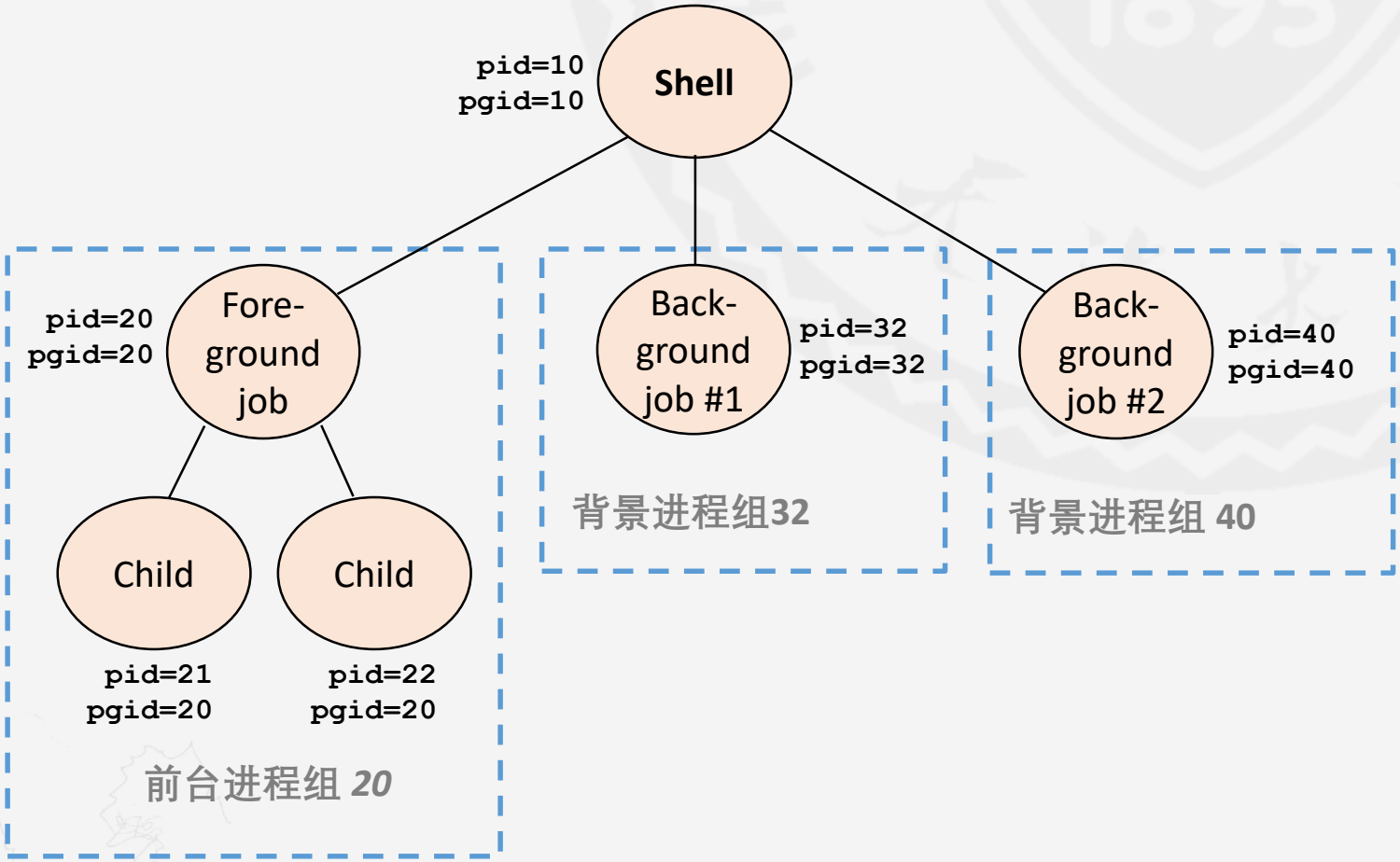
- 在每个进程的上下文中，都会维护标识信号挂起和阻塞的位向量
- 挂起位向量：表示挂起信号的集合
  - 当传递类型为 k 的信号时，内核将挂起中的位 k 设置为 1
  - 当接收到类型为 k 的信号时，内核将挂起中的位 k 清零
- 阻塞位向量：表示阻塞信号的集合
  - 可以使用 sigprocmask 函数设置和清除
  - 也称为信号屏蔽

信号

Signals

进程组

- 每个进程属于且仅属于一个进程组
- `int getpgid(int pid);`
  返回当前的进程组
- `int setpgid(int pid, int pgid);`
  设置进程组
- 更多见教材



使用/bin/kill发送信号

- /bin/kill 程序可以向一个进程和一个进程组发送某些信号
- 例如：
  - /bin/kill –9 24818
    - 向24818进程发送SIGKILL信号
  - /bin/kill –9 –24817
    - 向24817进程组中的每个进程发送SIGKILL信号

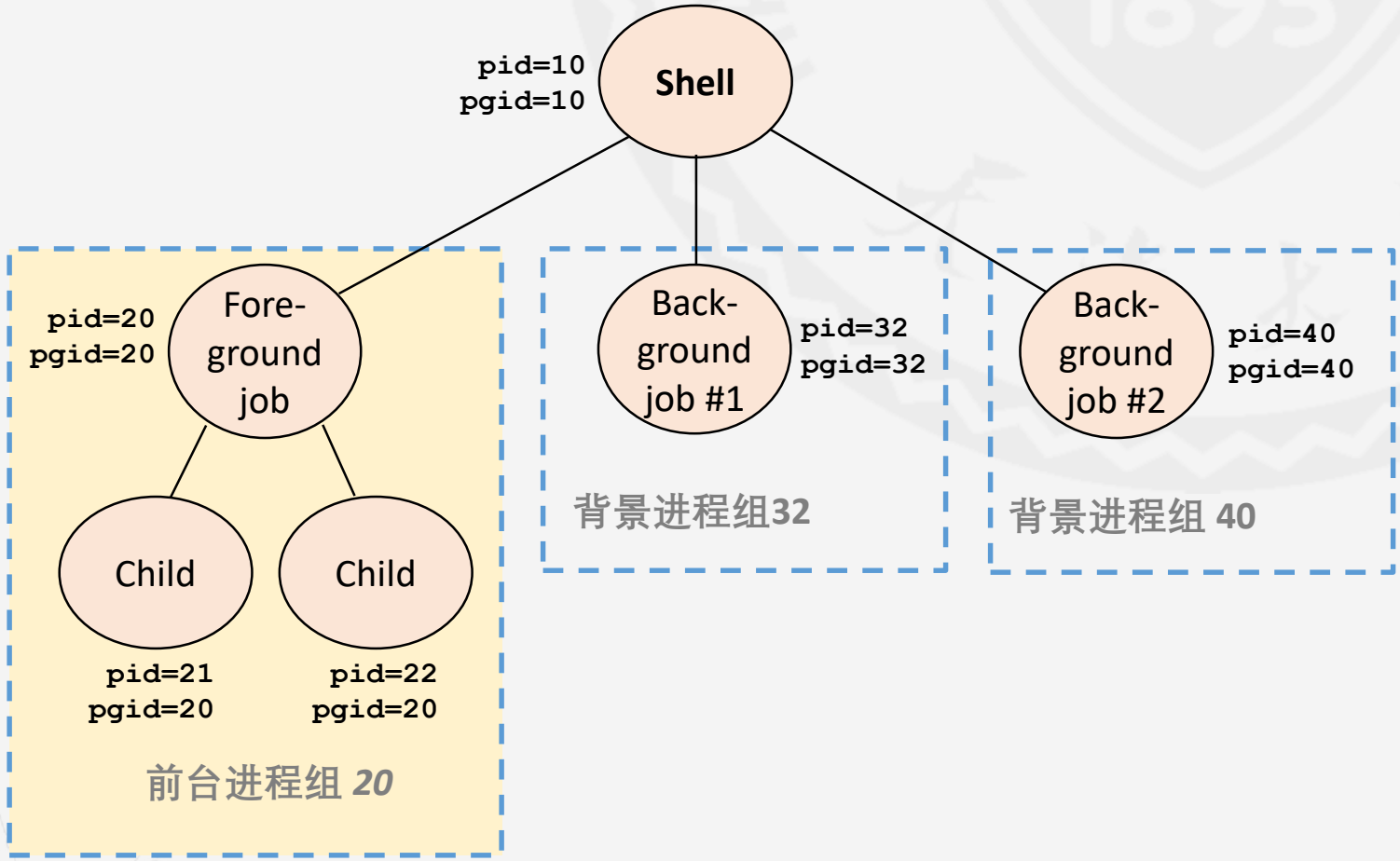
```
linux> ./forks 16
Child1: pid=24818 pgrp=24817
Child2: pid=24819 pgrp=24817

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>
```

信号

Signals

从键盘发送信号

- 按下 Ctrl-C（Ctrl-Z）会导致内核向**前台进程组**中的每个作业发送 SIGINT（SIGTSTP）信号
  - SIGINT - 默认操作是终止每个进程
  - SIGTSTP - 默认操作是停止（暂停）每个进程



ctrl-c和ctrl-z示例

```
void fork17()
{
    if (fork() == 0) {
      printf("Child: pid=%d pgrp=%d\n",
             getpid(), getpgrp());
    } else {
      printf("Parent: pid=%d pgrp=%d\n",
             getpid(), getpgrp());
    }
    while(1);
}
```

```
linux> ./forks 17
Child: pid=28108 pgrp=28107
Parent: pid=28107 pgrp=28107
<types ctrl-z>
Suspended
linux> 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
linux> fg
./forks 17
<types ctrl-c>
linux> ps w
  PID TTY      STAT   TIME COMMAND
27699 pts/8    Ss     0:00 -tcsh
28110 pts/8    R+     0:00 ps w
```

STAT（进程状态）标识：

第一个字母：

S：休眠

T：停止

R：运行

第二个字母：

s：会话发起者

+：前台进程组

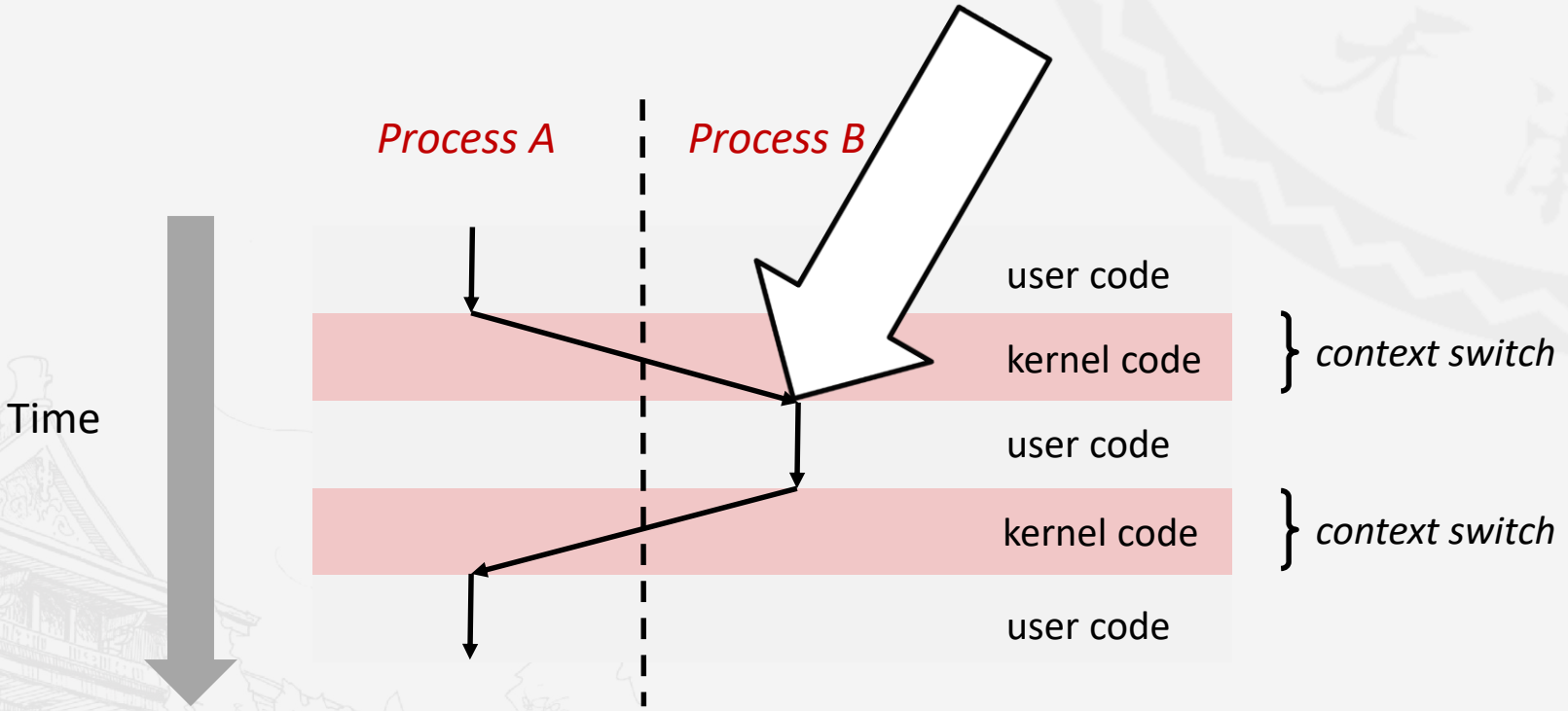
更多细节请参见 "man ps"。

使用kill系统调用发送信号

```
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);
    }
}
```

接收信号的实现

- 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。



接收信号的实现（2）

- 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。
- 内核计算 pnb = pending & ~blocked
  - 这是进程 p 的挂起非阻塞信号集合
- 如果 (pnb == 0)
  - 将控制传递给进程 p 的逻辑流中的下一条指令
- 否则
  - 选择 pnb 中最小的非零位 k，并强制进程 p 接收信号 k
  - 接收信号触发 p 中的某些操作
  - 对于 pnb 中的所有非零位 k 重复上述过程
  - 将控制传递给进程 p 的逻辑流中的下一条指令

默认的信号处理方式

- 每种信号类型都有一个预定义的默认操作，为以下三者之一：
  - 进程终止
  - 进程暂停，直到收到 SIGCONT 信号重新启动
  - 进程忽略该信号

处理信号

- handler_t *signal(int signum, handler_t *handler)
  - signal函数可以将信号signum与信号处理函数handler进行关联，以取代默认的处理方式
- handler的值：
  - SIG_IGN：忽略类型为 signum 的信号
  - SIG_DFL：收到类型为 signum 的信号时恢复默认操作
  - 否则，handler 是用户级别信号处理函数的地址
    - 当进程接收到类型为 signum 的信号时，调用该函数
    - 执行处理程序被称为“捕获”或“处理”信号
    - 当处理程序执行其返回语句时，控制返回到被信号中断的进程控制流中的指令

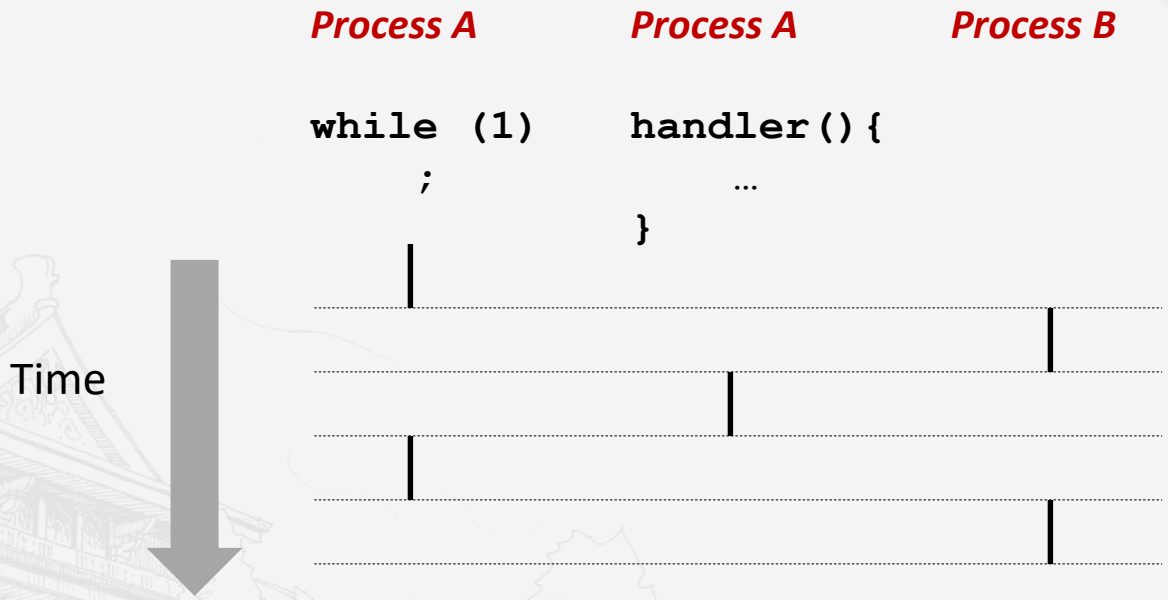
信号处理示例

```c
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...");
    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;
}
```

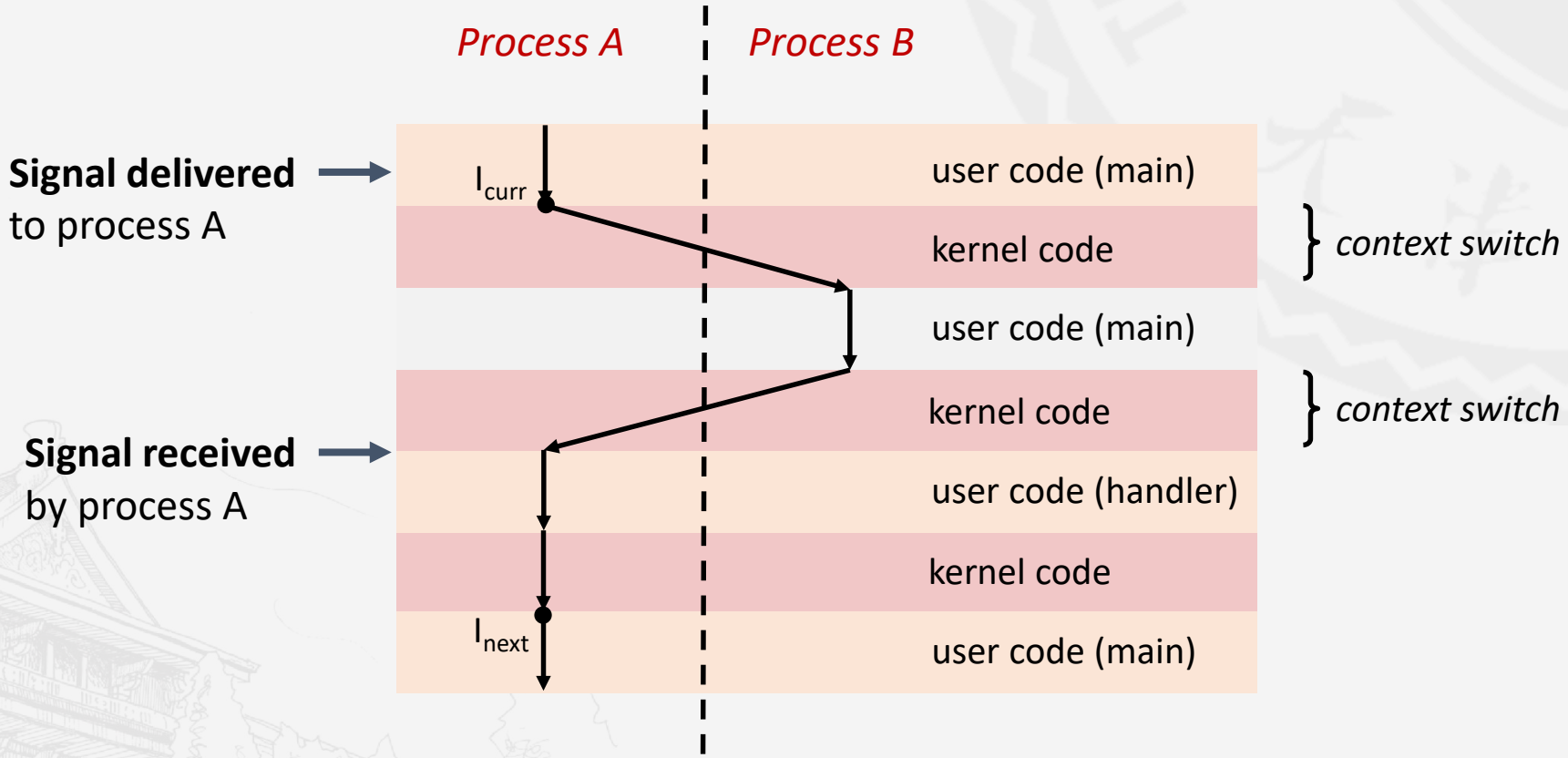
sigint.c

在并发流中信号处理程序的位置

- 信号处理程序是一个独立的逻辑流（不是进程），与主程序并发运行。

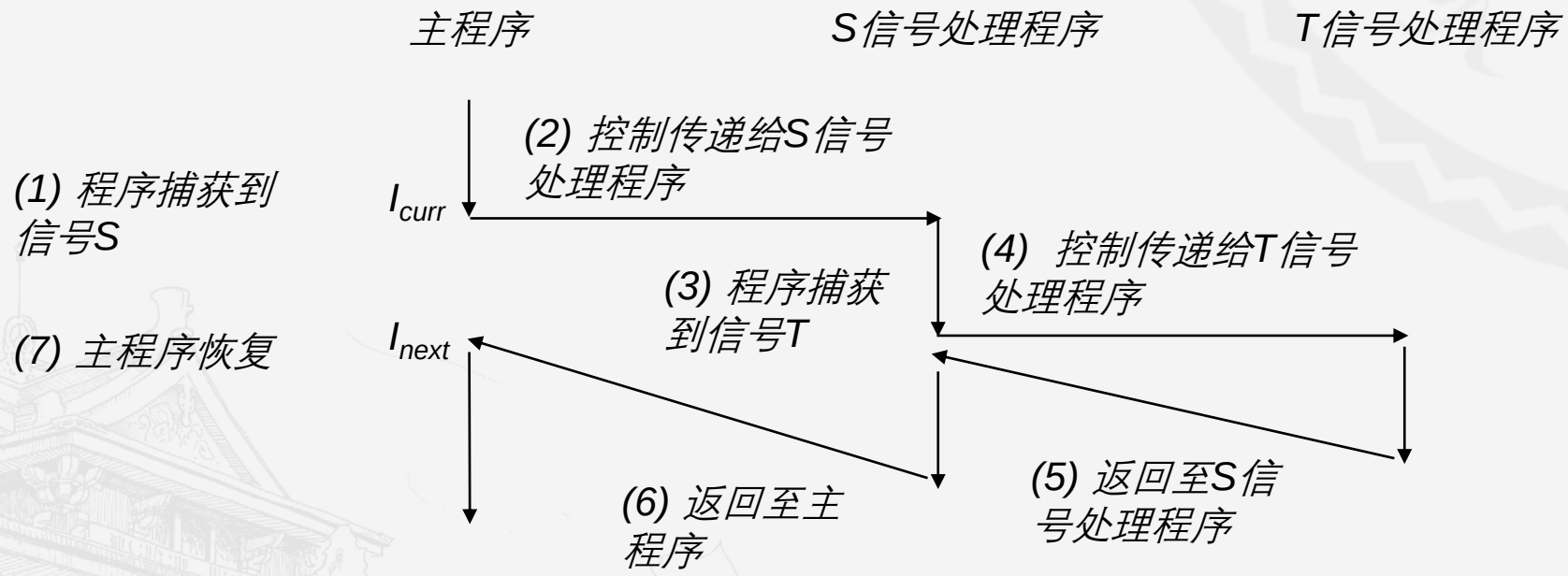


对信号处理程序的另一种视角



信号的嵌套处理

- 信号处理程序可能会被其他信号处理程序打断



信号的阻塞和解除阻塞

- 隐式阻塞机制
  - 内核会阻塞当前正在处理的类型的任何挂起信号
  - 例如，SIGINT处理程序无法被另一个SIGINT中断。
- 显式阻塞和解除阻塞机制
  - sigprocmask函数

- 其他函数
  - sigemptyset - 创建空集合
  - sigfillset - 将全部信号添加到集合中
  - sigaddset - 将某个信号添加到集合中
  - sigdelset - 从集合中删除信号

临时阻塞信号

```
sigset_t mask, prev_mask;

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

/* Block SIGINT and save previous blocked set */
Sigprocmask(SIG_BLOCK, &mask, &prev_mask);

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

/* Restore previous blocked set, unblocking SIGINT */
Sigprocmask(SIG_SETMASK, &prev_mask, NULL);
```

安全的信号处理

- 信号处理程序的可靠实现是一件很棘手的事情
  - 因为它们与主程序并发运行并共享相同的全局数据结构
  - 共享的数据结构可能会被损坏
- 后续课程中会讨论如何解决并发问题
- 以下给出一些建议，以实现更可靠安全的信号处理

编写安全信号处理程序的原则

1. 保持处理程序尽可能简单
   - 例如，设置一个全局标志并返回
2. 在处理程序中只调用异步信号安全的函数
   - printf、sprintf、malloc和exit是不安全的!
3. 在进入和退出时保存和恢复errno
   - 以防其他处理程序覆盖errno的值
4. 通过临时阻塞所有信号来保护对共享数据结构的访问，以防止可能的损坏
5. 将全局变量声明为volatile，以防止编译器将它们存储在寄存器中
6. 将全局标志声明为volatile sig_atomic_t
   - 标志：仅被读取或写入的变量（例如，flag = 1，而不是flag++）
   - 以这种方式声明的标志不需要像其他全局变量一样受到保护。

异步信号安全的函数

- 如果函数是可重入的（例如，所有变量都存储在堆栈帧上）或者不会被信号中断，则该函数是异步信号安全的
- POSIX保证有117个函数是异步信号安全的
  - Linux下使用“man 7 signal”命令可以查看
  - 列表中包含的一些常用函数：
    - _exit、write、wait、waitpid、sleep、kill
  - 列表中未包含的一些常用函数：
    - printf、sprintf、malloc、exit
- 不幸的是，write是唯一一个异步信号安全的输出函数

在信号处理函数中安全的输出

- 在处理程序中可以使用可重入的SIO（安全I/O库）实现安全输出（csapp.c）
  - `ssize_t sio_puts(char s[]) /* Put string */`
  - `ssize_t sio_putl(long v)   /* Put long */`
  - `void sio_error(char s[])   /* Put msg & exit */`

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

sigintsafe.c

```c
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 */

}
```

forks.c

这个代码是错误的

正确的信号处理（1）

- 挂起的信号不会排队
  - 对于每种信号类型，都有一个标志位指示该信号是否挂起.
  - 因此，每种特定类型的信号最多只能有一个挂起信号。
- 不能使用信号来计算事件，比如子进程的终止。

正确的信号处理（2）

- 必须等待所有已终止的子进程
  - 使用循环将 wait 放在其中，以收集所有已终止的子进程。

```
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;
}
```

wait的参数 status 用来保存被收集进程退出时的状态，如果不关心子进程是如何退出的，而只想把该僵死进程销毁，则可以设置参数为 NULL

如果成功，wait（）会返回被收集的子进程的进程 ID

如果没有子进程，则会失败，此时 wait（）返回-1，同时 errno 被置为 ECHILD

可移植的信号处理

- 不同版本的Unix可能具有不同的信号处理语义
  - 一些旧系统在捕获信号后会将操作还原为默认值
  - 一些被中断的系统调用可能会返回 errno == EINTR
  - 一些系统不会阻塞正在处理的信号类型的信号
- 解决方案：使用 sigaction

```c
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);
}
```

csapp.c

同步流以避免并发错误（1）

- 右面是一个带有微妙的同步错误的Shell实现

```
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

同步流以避免并发错误（2）

- SIGCHLD信号处理函数

错误原因：父进程不一定在子进程之前恢复运行；deletejob可能先于addjob

```
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

修正后

```c
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);
}
```

procmask2.c

显示地等待信号（1）

```c
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) {
       /* Block SIGCHLD */
       Sigprocmask(SIG_BLOCK, &mask, &prev);
       if (Fork() == 0) /* Child */
         exit(0);
       /* Parent */
       pid = 0;
       /* Unblock SIGCHLD */
       Sigprocmask(SIG_SETMASK, &prev, NULL);
       /* Wait for SIGCHLD to be received (wasteful!) */
       while (!pid);
       /* Do some work after receiving SIGCHLD */
       printf(".");
    }
    exit(0);
}
```

waitforsignal.c

```c
volatile sig_atomic_t pid;

void sigchld_handler(int s)
{
    int olderrno = errno;
    /* Main is waiting for nonzero pid */
    pid = Waitpid(-1, NULL, 0);
    errno = olderrno;
}

void sigint_handler(int s)
{
}
```

waitforsignal.c

类似于等待一个前台任务结束

显示地等待信号（2）

- 逻辑是正确的，但是性能很差（忙等待浪费CPU计算资源）
- 一些其他的方案

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

SIGCHLD如果在pause前到达，则pause无法返回

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

响应不实时

- 解决方案：使用 sigsuspend

使用sigsuspend等待信号

- `int sigsuspend(const sigset_t *mask)`
- 等效于不可中断版本的：（假设前两行代码中间不可打断）

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

使用sigsuspend等待信号

```c
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);
}
```

sigsuspend.c

本章内容

Topic

- 命令解释器
- 信号
- 非本地跳转

非本地跳转

Nonlocal jumps

setjmp/longjmp (1)

- 强大但危险的用户级机制，用于将控制权传递到任意位置
  - 打破了“过程调用/返回规则”的受控方式
  - 用于错误恢复和信号处理
- `int setjmp(jmp_buf j)`
  - 必须在调用 `longjmp` 之前调用
  - 为随后的 `longjmp` 标识返回位置
  - 仅调用一次，可以返回多次
- 实现：
  - 通过将当前寄存器上下文、栈指针和 PC 值存储在 `jmp_buf` 中来记住当前位置
  - 返回 `0`

非本地跳转

Nonlocal jumps

setjmp/longjmp (2)

- `void longjmp(jmp_buf j, int i)`
  - 返回至通过setjmp存储的j所在的位置
  - 返回的值为 `i`，而不是 `0`
  - 在 `setjmp` 之后调用
  - 仅调用一次，但不会返回
- `longjmp` 实现：
  - 从跳转缓冲 `j` 恢复寄存器上下文（堆栈指针、基址指针、PC 值）
  - 将 `%eax`（返回值）设置为 `i`
  - 跳转到跳转缓冲 `j` 中存储的 PC 指示的位置

非本地跳转

Nonlocal jumps

setjmp/longjmp 示例

```c
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);
}
```

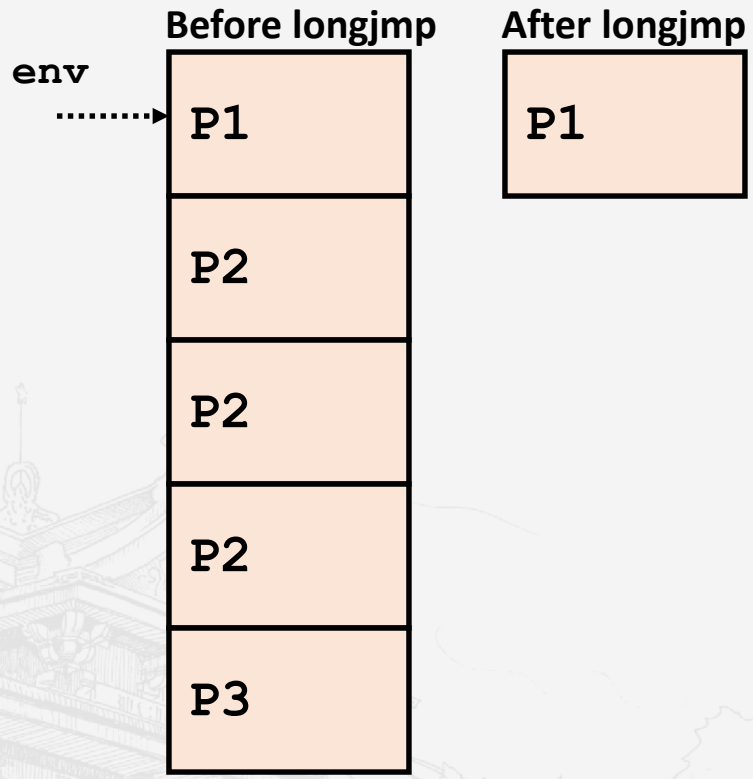
```c
/* Deeply nested function foo */
void foo(void)
{
    if (error1)
        longjmp(buf, 1);
    bar();
}

void bar(void)
{
    if (error2)
        longjmp(buf, 2);
}
```

非本地跳转

Nonlocal jumps

非本地跳转的限制 (1)

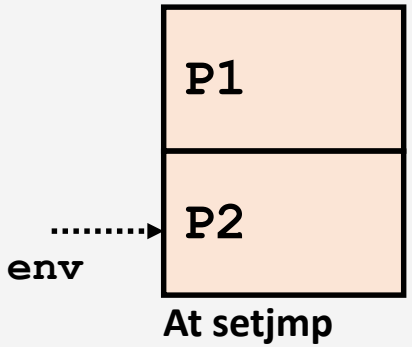
- 在堆栈规则内有效
  - 只能长跳转到已调用但尚未完成的函数环境中

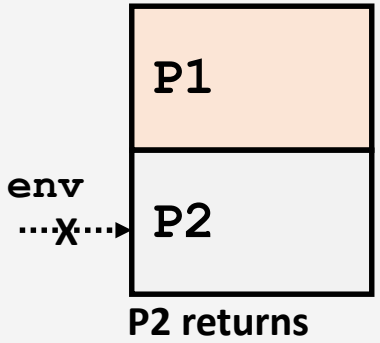


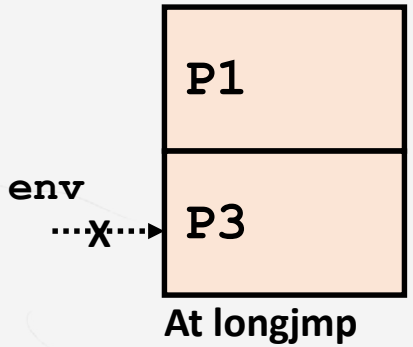

```
jmp_buf env;

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

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

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

非本地跳转

Nonlocal jumps







非本地跳转的限制 (2)

```
jmp_buf env;

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

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

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

非本地跳转

Nonlocal jumps

```c
#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 */
}
```

restart.c

综合：一个在按下 ctrl-c 时重新启动自身的程序

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

小结

- 信号提供进程级别的异常处理
  - 可以由用户程序生成
  - 可以通过声明信号处理程序定义其效果
  - 在编写信号处理程序时要非常小心
- 非本地跳转在进程内提供异常控制流
  - 在堆栈规则的限制内