第8章 异常控制流II：
——信号与非本地跳转

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

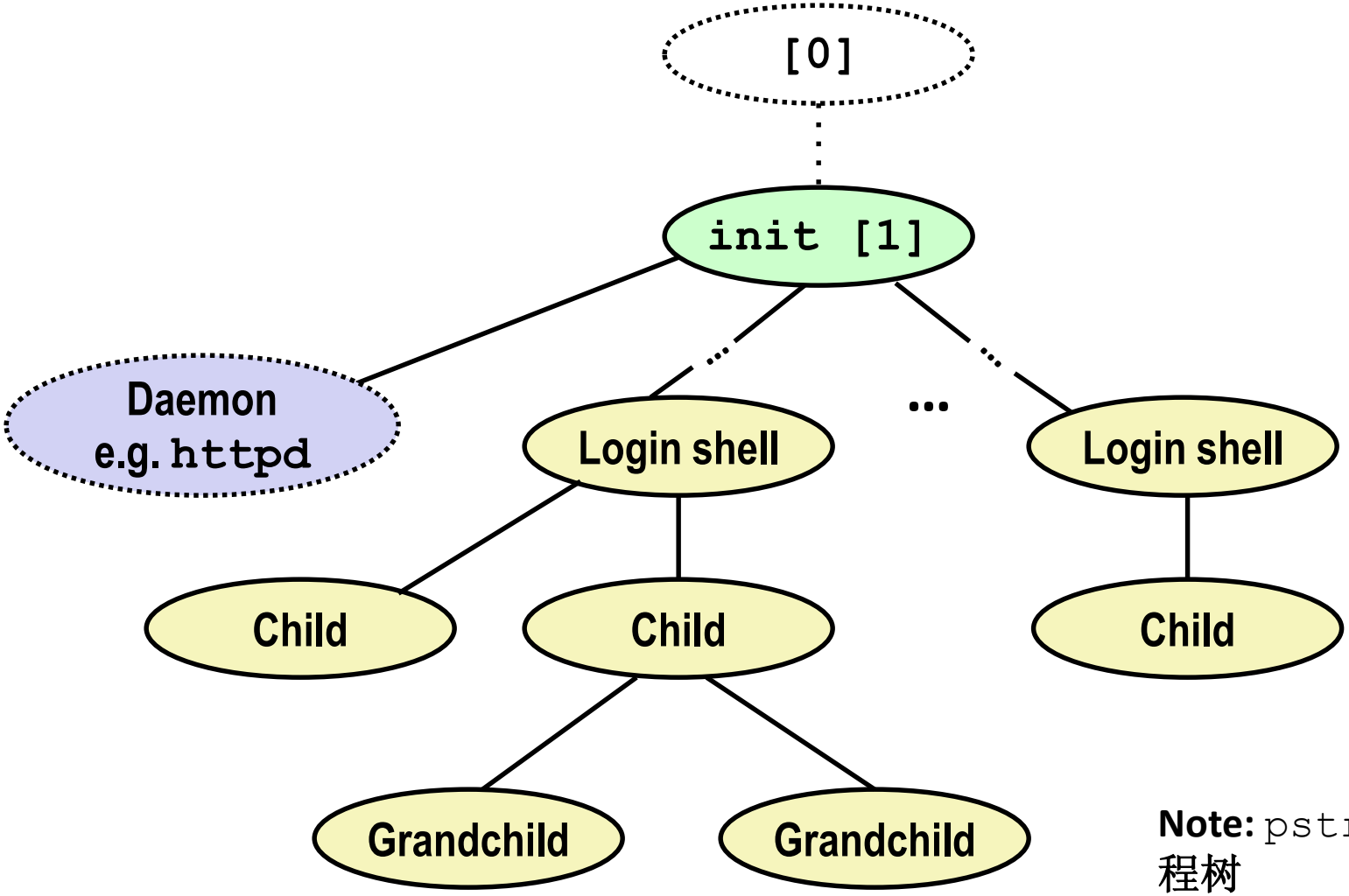
本节内容

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: `pstree` 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/tcsh** 变种
  - **bash** 变种、默认的Linux shell

```
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);
         }
      }

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

```
/* If first arg is a builtin command, run it and return true */
int builtin_command(char **argv)
{
    if (!strcmp(argv[0], "quit")) /* quit command */
        exit(0);
    if (!strcmp(argv[0], "&"))    /* Ignore singleton & */
        return 1;
    return 0;                     /* Not a builtin command */
}
```

shellex.c

简单shell例子的问题

```
/* If first arg is a builtin command, run it and return true */
int builtin_command(char **argv)
{
    if (!strcmp(argv[0], "quit")) /* quit command */
        exit(0);
    if (!strcmp(argv[0], "&"))    /* Ignore singleton & */
        return 1;
    return 0;                     /* Not a builtin command */
}
```

- 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
  - ***Windows下称为消息***

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)　　/现代64类信号
  - 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（CTRL-C）
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

序号	名称	默认行为	相应事件
1	SIGHUP	终止	终端线挂断
2	SIGINT	终止	来自键盘的中断
3	SIGQUIT	终止	来自键盘的退出
4	SIGILL	终止	非法指令
5	SIGTRAP	终止并转储内存①	跟踪陷阱
6	SIGABRT	终止并转储内存①	来自 abort 函数的终止信号
7	SIGBUS	终止	总线错误
8	SIGFPE	终止并转储内存①	浮点异常
9	SIGKILL	终止②	杀死程序
10	SIGUSR1	终止	用户定义的信号 1
11	SIGSEGV	终止并转储内存①	无效的内存引用（段故障）
12	SIGUSR2	终止	用户定义的信号 2
13	SIGPIPE	终止	向一个没有读用户的管道做写操作
14	SIGALRM	终止	来自 alarm 函数的定时器信号
15	SIGTERM	终止	软件终止信号
16	SIGSTKFLT	终止	协处理器上的栈故障
17	SIGCHLD	忽略	一个子进程停止或者终止
18	SIGCONT	忽略	继续进程如果该进程停止
19	SIGSTOP	停止直到下一个 SIGCONT②	不是来自终端的停止信号
20	SIGTSTP	停止直到下一个 SIGCONT	来自终端的停止信号
21	SIGTTIN	停止直到下一个 SIGCONT	后台进程从终端读
22	SIGTTOU	停止直到下一个 SIGCONT	后台进程向终端写
23	SIGURG	忽略	套接字上的紧急情况
24	SIGXCPU	终止	CPU 时间限制超出
25	SIGXFSZ	终止	文件大小限制超出
26	SIGVTALRM	终止	虚拟定时器期满
27	SIGPROF	终止	剖析定时器期满
28	SIGWINCH	忽略	窗口大小变化
29	SIGIO	终止	在某个描述符上可执行 I/O 操作
30	SIGPWR	终止	电源故障

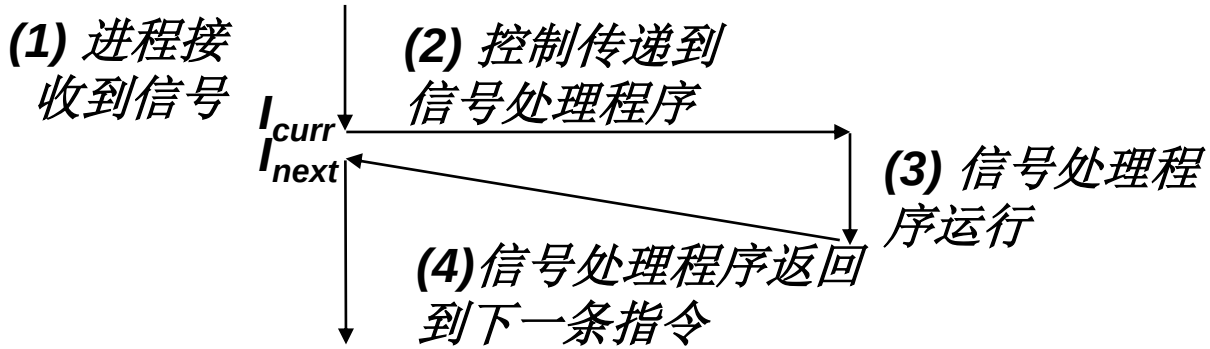
信号术语：发送信号

- 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 ----软件异常处理程序
    - 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

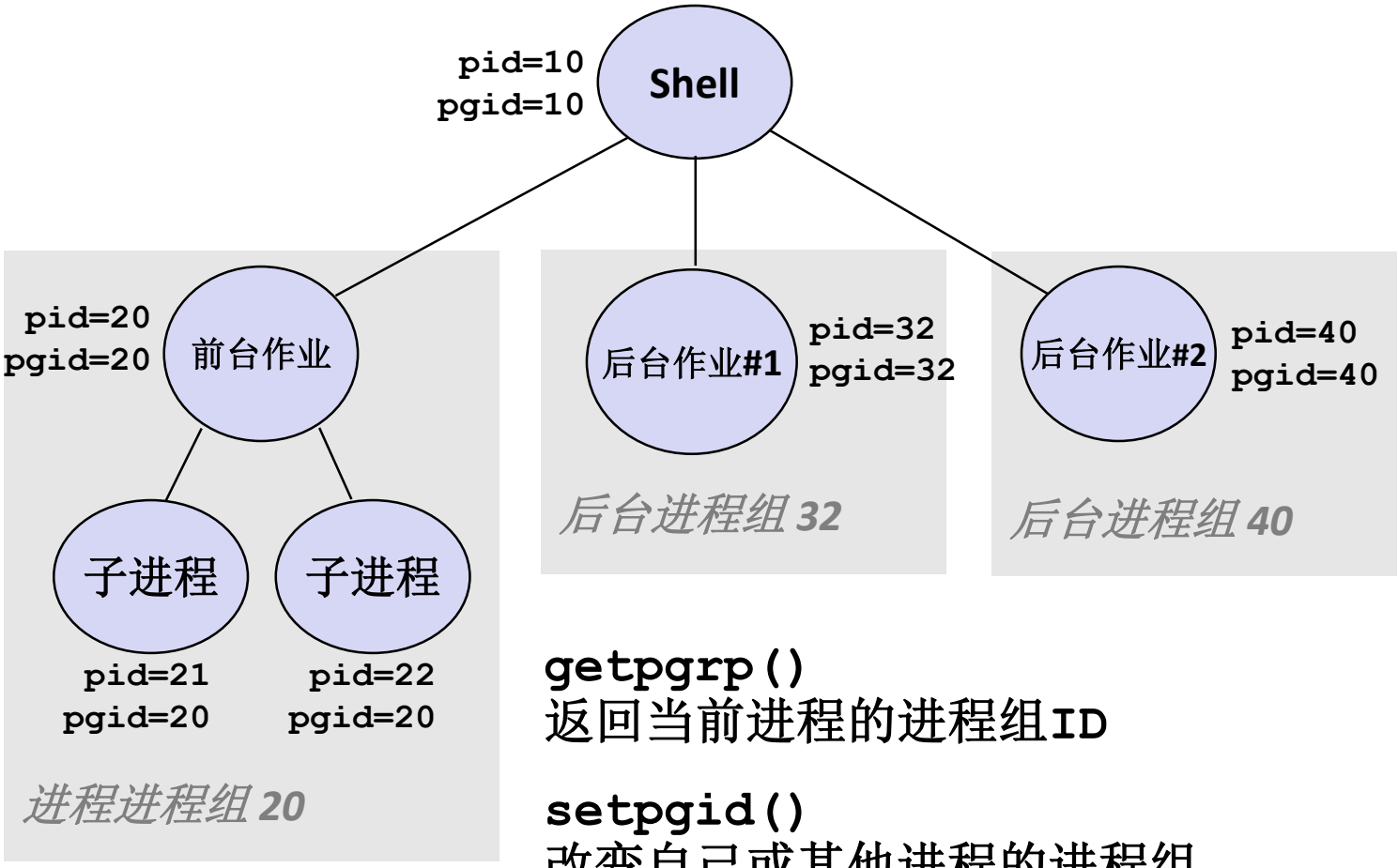
- 一个发出而没有被接收的信号叫做待处理信号（*pending*）
  - 任何时刻，一种类型（1-30）至多只有一个待处理信号
  - Important: 信号不会排队等待
    - 如果一个进程有一个类型为k的待处理信号，那么任何接下来发送到这个进程的类型为k的信号都会被丢弃
- 一个进程可以选择*阻塞*接收某种信号
  - 阻塞的信号仍可以被发送，但不会被接收，直到进程取消对该信号的阻塞
- 一个待处理信号最多只能被接收一次

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mask

发送信号: 进程组

- 每个进程只属于一个进程组



getpgrp()
返回当前进程的进程组ID

setpgid()
改变自己或其他进程的进程组

用 /bin/kill 程序发送信号

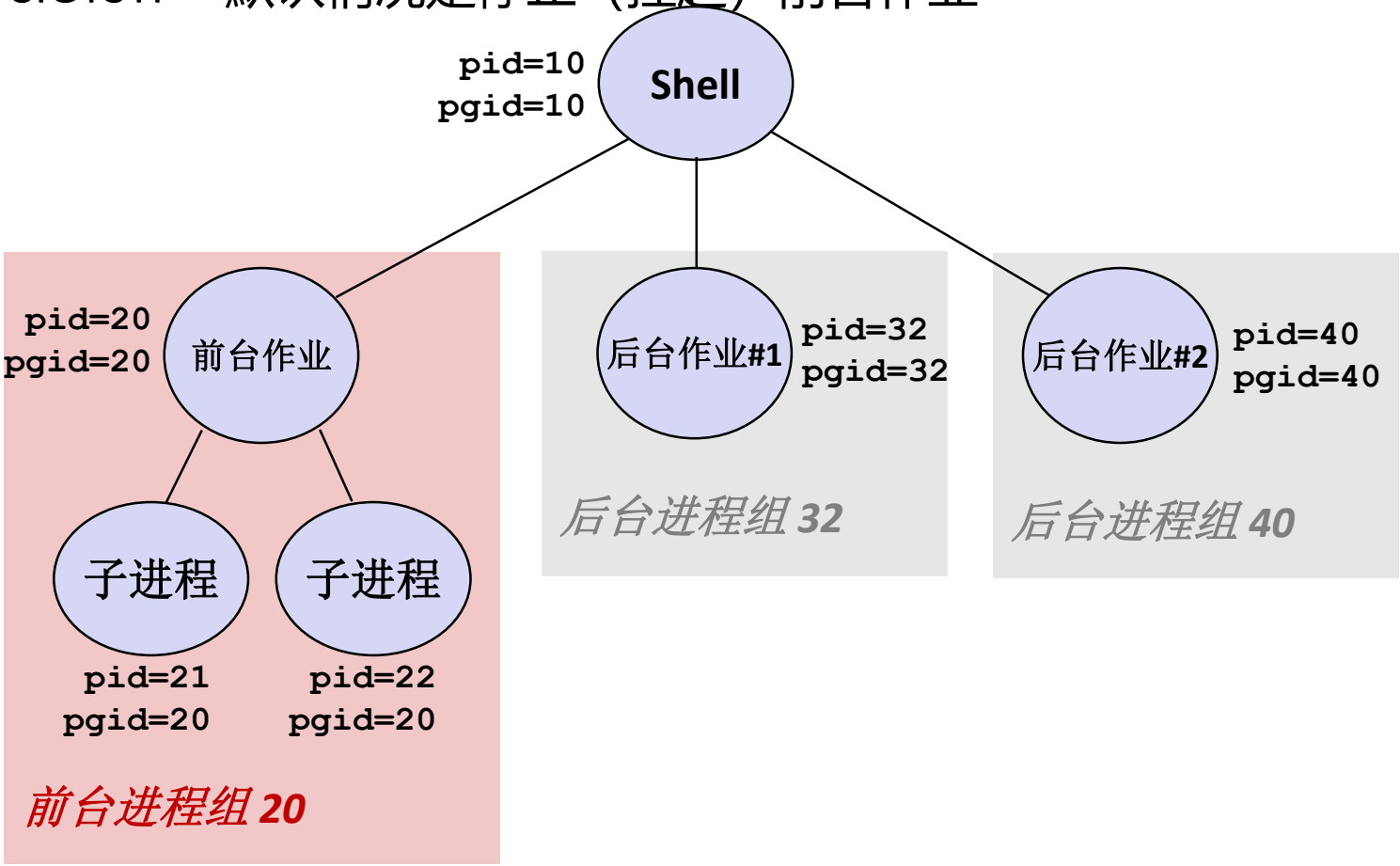
- /bin/kill程序可以向另外的进程或进程组发送任意的信号
- Examples
  - /bin/kill -9 24818
    发送信号9（SIGKILL）给进程24818
  - /bin/kill -9 -24817
    发送信号SIGKILL给进程组24817中的每个进程

（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



Example of `ctrl-c` and `ctrl-z`

```
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
```

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: session leader
+: 前台进程组

执行“**man ps**” 查看详细内容

用 kill 函数发送信号

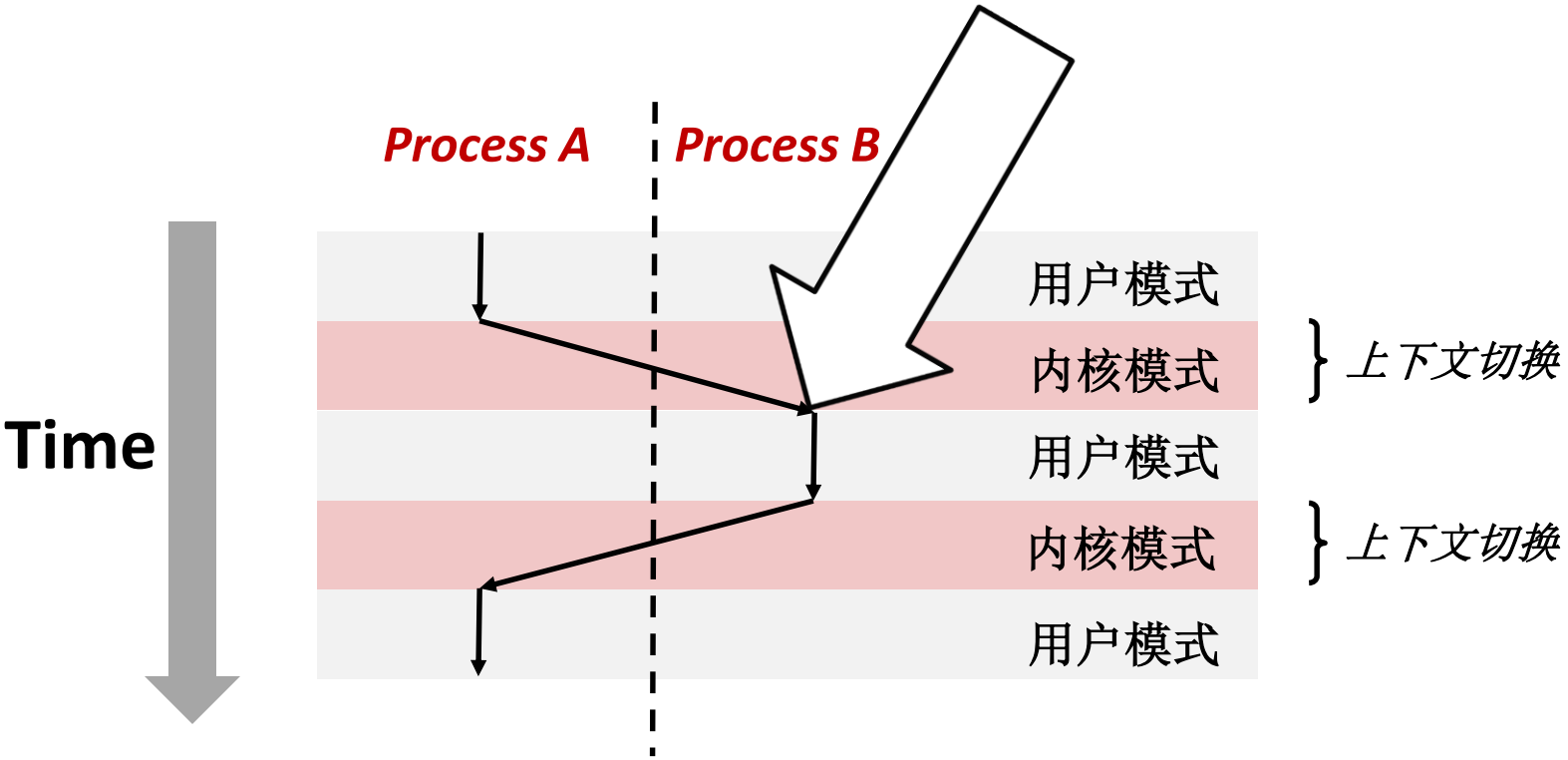
```
void fork12()
{
    pid_t pid[N];
    int i;
    int child_status;

    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);
    }
}
```

如果pid大于零，那么kill函数发送信号号码sig给进程pid。如果pid等于零，那么kill发送信号sig给调用进程所在进程组中的每个进程，包括调用进程自己。如果pid小于零，kill发送信号sig给进程组|pid|（pid的绝对值）中的每个进程。

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 `pnb = pending & ~blocked`
    - 进程*p* 的未被阻塞的待处理信号的集合
  - If (`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* 处理信号*k* *(清0)*
      - 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
    - 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

- 可以使用 **signal**函数修改和信号**signum**相关联的默认行为:
  - **handler_t *signal(int signum, handler_t *handler)**
- **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
  - 否则，**handler** 就是用户定义的函数的地址，这个函数称为信号处理程序
    - 只要进程接收到类型为 **signum**的信号就会调用*信号处理程序*
    - 将处理程序的地址传递到**signal**函数从而改变默认行为，这叫作设置信号处理程序
    - 调用*信号处理程序*称为捕获信号
    - 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

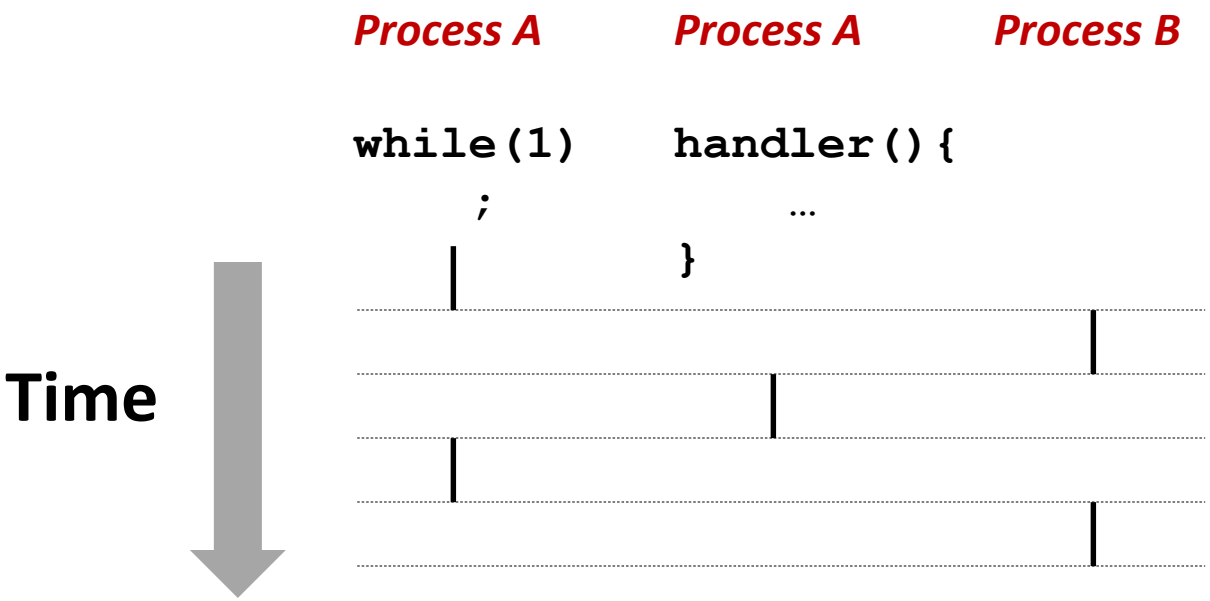
用信号处理程序捕获SIGINT信号

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

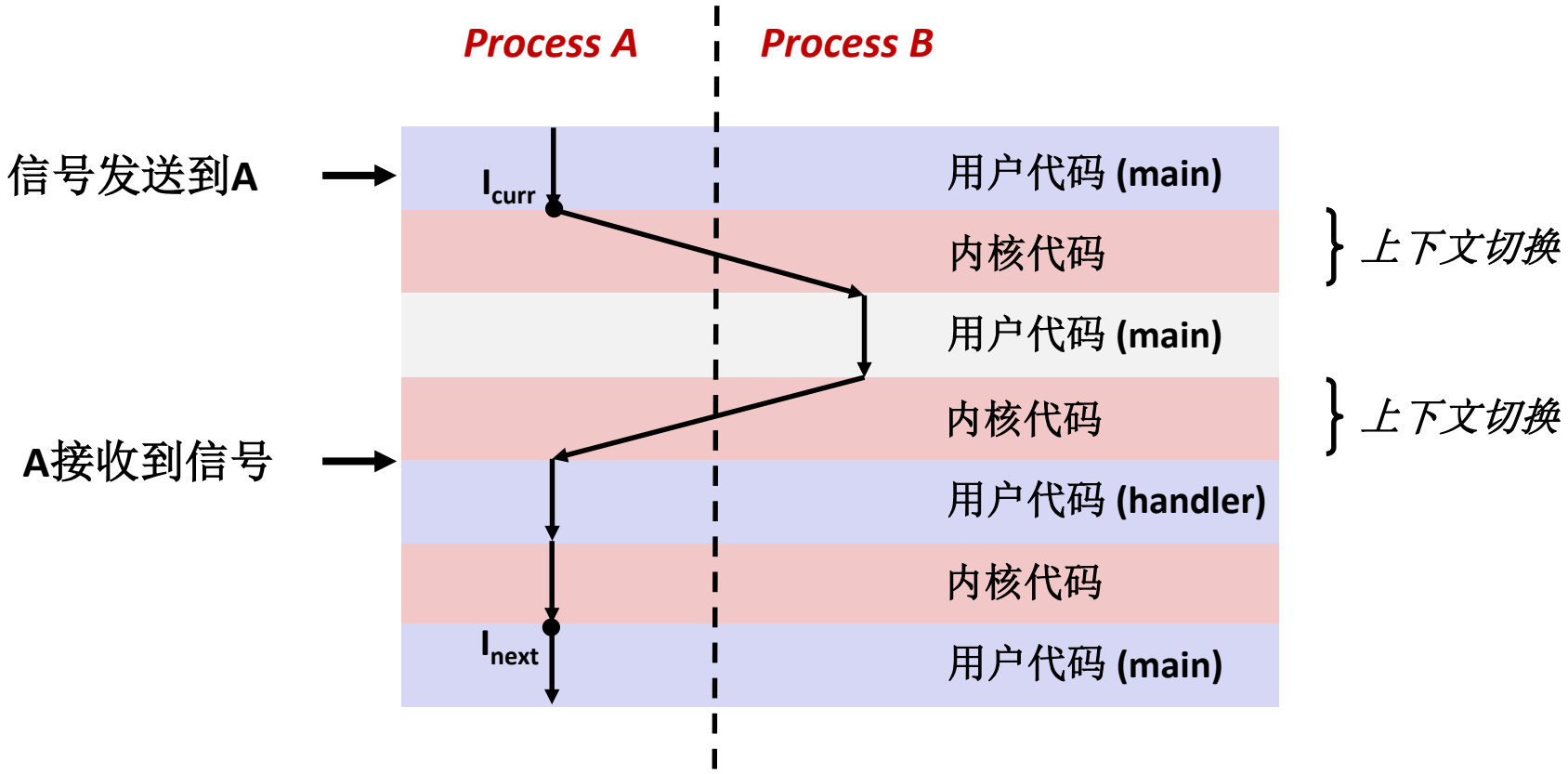
sigint.c

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）

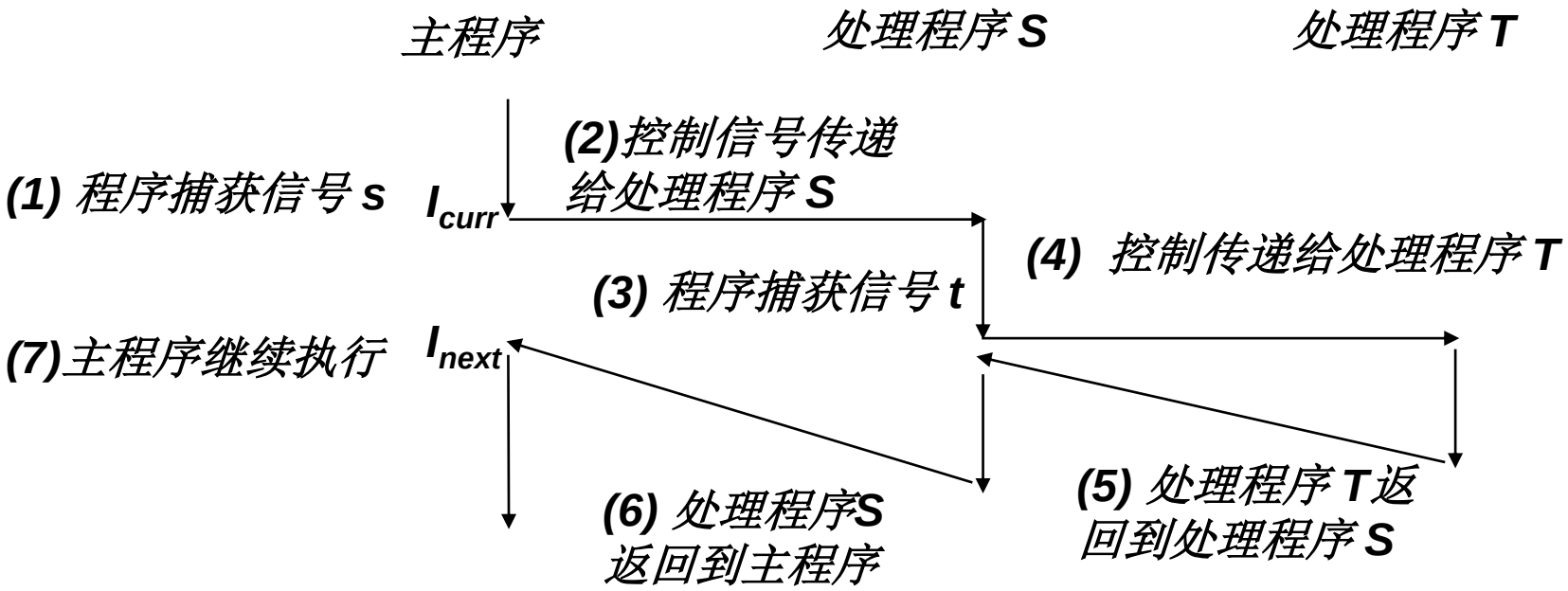


另一个角度看作为并发流的信号处理程序



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）

- 显式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：`SIG_ BLOCK/UNBLOCK/SET_MASK`
  - 辅助函数
    - `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);

    /* 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

- 信号处理程序很麻烦是因为它们和主程序并发地运行并共享全局数据结构
  - 共享数据结构可能被破坏
- 第12章会详细讲述并发编程
- 这里仅给出一些原则

编写处理程序的原则(阅读教材P536)

- **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，让主进程去判断处理
- **G1:** 在处理程序中只调用异步信号安全的函数 **P534（OS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++)—读写过程中不响应中断或信号
  - 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

- 函数是*异步信号安全的*指函数要么是可重入的（如只访问局部变量，见12.7.2节）要么不能被信号处理程序中断
- Posix保证安全的 117 个异步信号安全的函数
  - Source: “`man 7 signal`”
  - 常见的安全的函数**包括:**
    - `_exit, write, wait, waitpid, sleep, kill`
    - `write` 函数是信号处理程序中唯一安全的输出函数
  - **不包括:**
    - **`printf, sprintf, malloc, exit`**

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

将控制直接从一个函数转移到另一个当前正在执行的函数，而不需要经过正常的调用-返回序列（调用栈）

- `int setjmp(jmp_buf j)`
  - 必须在longjmp之前被调用
  - 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 j 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

setjmp/longjmp (cont)

- **void longjmp(jmp_buf j, int i)**
  - 含义:
    - 从缓冲区**j**中恢复调用环境，并触发 **setjmp** 返回非零的返回值 **i**
  - 在setjmp之后被调用
  - 被调用一次，从不返回

- **longjmp** 的执行:
  - 从缓冲区**j**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
  - 跳转至保存在缓冲区 **j** 中的PC所指示的位置

setjmp/longjmp Example (cont)

- 目标:从深层嵌套函数调用中直接返回

```
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);
}
```

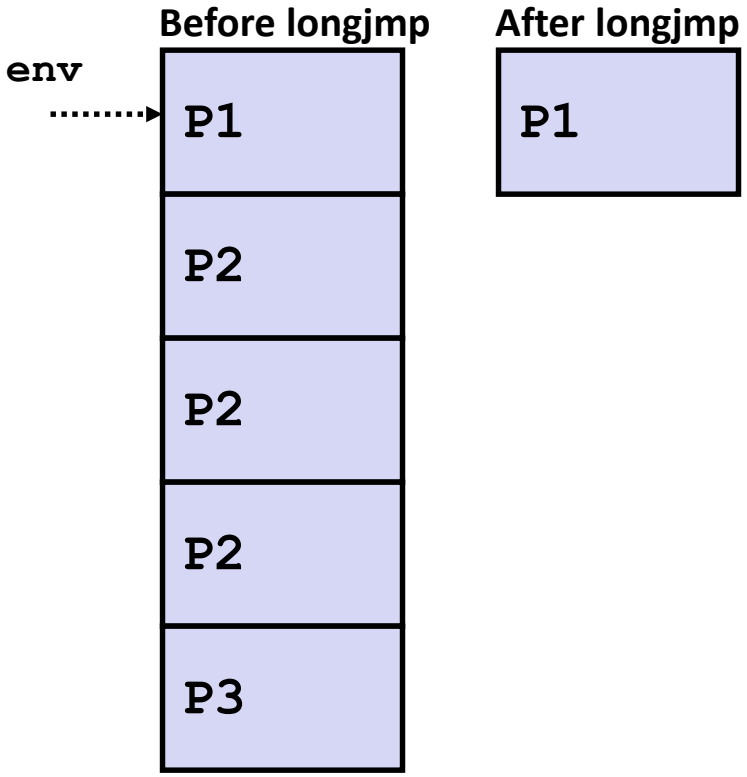
```
/* Deeply nested function foo */
void foo(void)
{
   if (error1)
      longjmp(buf, 1);
   bar();
}

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

1. main函数首先调用setjmp保存当前的调用环境（函数返回值为0）
2. 然后调用函数foo
3. foo函数依次调用函数bar

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

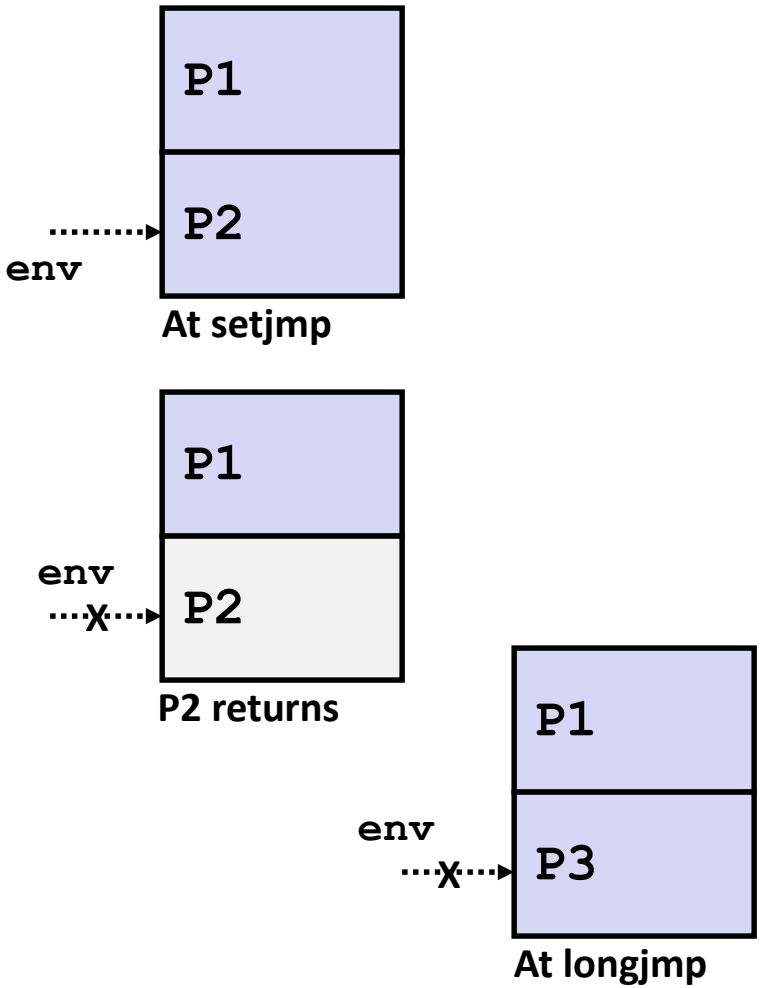
```
jmp_buf env;

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);
}
```



小结

- 信号提供了进程级的异常处理
  - 可由用户程序产生
  - 可以通过信号处理程序的声明定义信号的行为
  - 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

习题

- 1.非本地跳转中的setjmp函数调用一次，返回_____次。
- 2.子程序运行结束会向父进程发送_____信号。
- 3.进程加载函数execve，如调用成功则返回_____次。
- 4. 异步信号安全的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断，包括I/O函数（　）
  - A. printf　　B. sprintf　　C. write　　D. malloc

- 1.多
- 2.SIGCHLD
- 3.0
- 4.C

Hope you enjoyed the CSAPP course!