第8章 异常控制流

信号和非本地跳转 Signals and Nonlocal Jumps

100076202： 计算机系统导论

任课教师：

宿红毅　张艳　黎有琦　李秀星

原作者：

Randal E. **Bryant and** David R. O'Hallaron

上次课复习 Review from last lecture

- 异常 **Exceptions**
  - 需要非标准控制流的事件 Events that require nonstandard control flow
  - 由外部（中断）或内部（陷阱和故障）生成 Generated externally (interrupts) or internally (traps and faults)

- 进程 **Processes**
  - 在任何给定时间，系统都有多个活动进程 At any given time, system has multiple active processes
  - 一次只能在任何单个内核上执行一个进程 Only one can execute at a time on any single core
  - 每个进程似乎都可以完全控制处理器+专用内存空间 Each process appears to have total control of processor + private memory space

复习（续） Review (cont.)

- **创建进程 Spawning processes**
  - 调用fork　Call `fork`
  - 一次调用，两次返回 One call, two returns
- **进程完成 Process completion**
  - 调用exit　Call `exit`
  - 调用一次，不返回 One call, no return
- **回收和等待进程 Reaping and waiting for processes**
  - 调用wait或waitpid　Call `wait` or `waitpid`
- **加载和运行程序 Loading and running programs**
  - 调用execve（或变种）　Call `execve` (or variant)
  - 调用一次，（正常）不返回 One call, (normally) no return

execve：加载并运行程序
execve：Loading and Running Programs

- `int execve(char *filename, char *argv[], char *envp[])`
- 加载并在当前进程运行：**Loads and runs in the current process:**
  - 可执行文件**filename** Executable file **filename**
    - 可以是目标文件或以"#!解释器"开始的脚本文件 Can be object file or script file beginning with `#!interpreter` (e.g., `#!/bin/bash`)
  - 参数列表argv ...with argument list **argv**
    - 按照规则argv[0]为文件名 By convention **argv[0]==filename**
  - 和环境变量列表envp ...and environment variable list **envp**
    - "名字=值"串 "name=value" strings (e.g., `USER=droh`)
    - `getenv`, `putenv`, `printenv`
- **覆盖代码、数据和栈 Overwrites code, data, and stack**
  - 维持PID、打开文件和信号上下文 Retains PID, open files and signal context
- 调用一次，从不返回 **Called once and never returns**
  - 除非如果发生错误 ...except if there is an error

异常控制流存在系统每个层次
ECF Exists at All Levels of a System

- 异常 **Exceptions**
  - 硬件和操作系统内核软件
  - Hardware and operating system kernel software
- 进程上下文切换 **Process Context Switch**
  - 硬件时钟和内核软件
  - Hardware timer and kernel software

Previous Lecture
以前的课

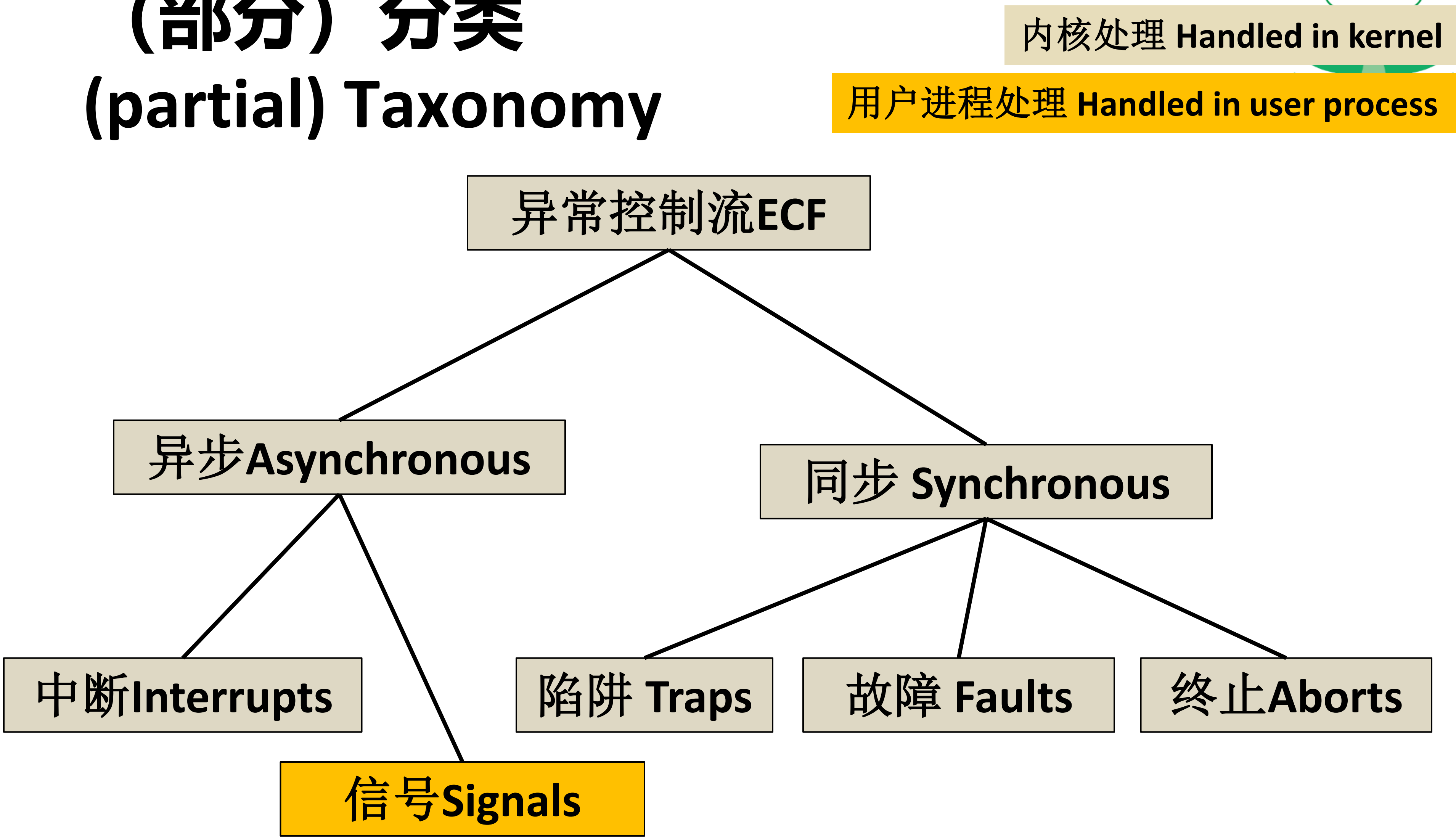
- 信号 **Signals**
  - 内核软件和应用软件
  - Kernel software and application software

This Lecture
本次课

- 非局部跳转 **Nonlocal jumps**
  - 应用代码 Application code

教材和补充幻灯片
Textbook and supplemental slides

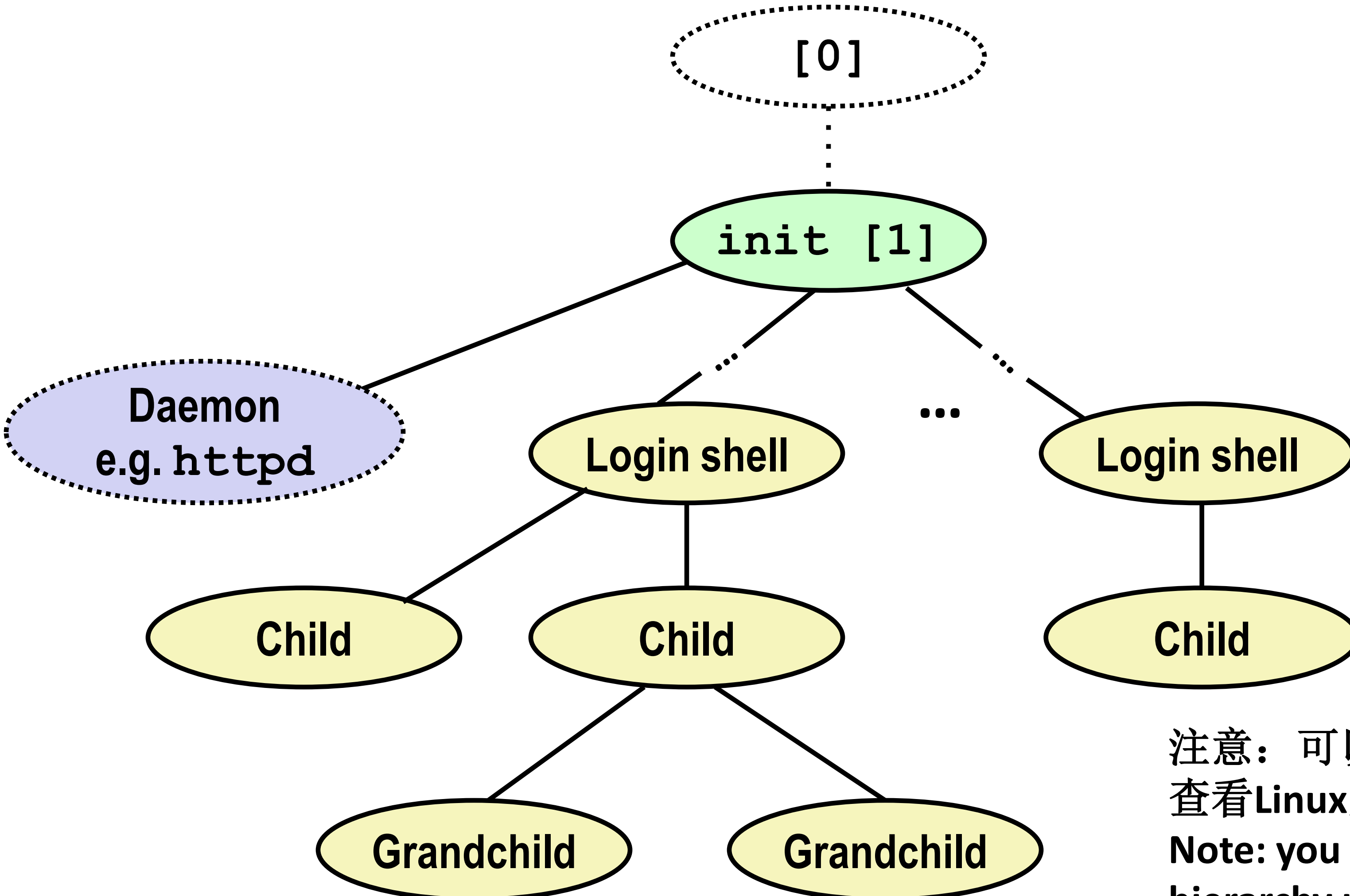
（部分）分类
(partial) Taxonomy



议题

- 外壳 **Shells**
- 信号 Signals
- 非局部跳转 Nonlocal jumps

Linux进程树 Linux Process Hierarchy



注意：可以用pstree命令查看Linux系统的进程树

Note: you can view the hierarchy using the Linux `pstree` **command**

Shell程序 Shell Programs

- **Shell是按照用户要求运行程序的应用程序 A *shell* is an application program that runs programs on behalf of the user**
  - `sh` 最早的 Original Unix shell (Stephen Bourne, AT&T Bell Labs, 1977)
  - `csh/tcsh` BSD Unix C shell
  - `bash` 默认的 "Bourne-Again" Shell (default Linux shell)
- **简单shell Simple shell**
  - 教材p753页处描述 Described in the textbook, starting at p. 753
  - 一个非常基础的shell实现 Implementation of a very elementary shell
  - 目的 Purpose
    - 理解当输入了命令后究竟发生了什么事情 Understand what happens when you type commands
    - 理解进程控制操作的使用和操作 Understand use and operation of process control operations

简单shell示例 Simple Shell Example

```
linux> ./shellex
> /bin/ls -l csapp.c
-rw-r--r-- 1 bryant users 23053 Jun 15  2015 csapp.c
> /bin/ps
  PID TTY          TIME CMD
31542 pts/2    00:00:01 tcsh
32017 pts/2    00:00:00 shellex
32019 pts/2    00:00:00 ps
> /bin/sleep 10 &
32031 /bin/sleep 10 &
> /bin/ps
 PID TTY          TIME CMD
31542 pts/2    00:00:01 tcsh
32024 pts/2    00:00:00 emacs
32030 pts/2    00:00:00 shellex
32031 pts/2    00:00:00 sleep
32033 pts/2    00:00:00 ps
> quit
```

必须给出程序的全路径名 Must give full pathnames for programs

后台运行程序 Run program in background

Sleep正在后台运行
Sleep is running in background

简单shell实现
Simple Shell Implementation

- **基本循环 Basic loop**
  - 从命令行读一行 Read line from command line
  - 执行请求的操作 Execute the requested operation
    - 内置命令（仅实现一个命令是**quit**）Built-in command (only one implemented is **quit**)
    - 从文件加载和执行程序 Load and execute program from file

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

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

        /* evaluate */
        eval(cmdline);
    }
   ...                                 shellex.c
```

执行的过程就是一系列读/求值的步骤 Execution is a sequence of read/evaluate steps

简单的Shell eval函数 Simple Shell `eval` Function

```
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;
}
```

shellex.c

简单的Shell eval函数 Simple Shell eval Function

```
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);
```

Parseline函数将buf解析成argv并返回是否输入行以&结尾

parseline will parse 'buf' into 'argv' and return whether or not input line ended in '&'

shellex.c

简单的Shell eval函数 Simple Shell eval Function

```
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 */
```

忽略空行

Ignore empty lines.

shellex.c

简单的Shell eval函数 Simple Shell eval Function

```
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)) {
```

如果是“内置”命令，那么在这个程序此处处理它。否则创建进程(fork) /执行(exec) 在argv[0]中指定的程序

If it is a ‘built in’ command, then handle it here in this program. Otherwise fork/exec the program specified in argv[0]

shellex.c

简单的Shell eval函数 Simple Shell eval Function

```
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 */
```

创建子进程/Create child

shellex.c

简单的Shell eval函数 Simple Shell eval Function

```
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);
            }
        }
```

启动argv[0].
记住execve仅在出错时返回
Start **argv[0]**.
Remember **execve** only returns on error.

shellex.c

简单的Shell eval函数 Simple Shell eval Function

```
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");
        }
```

如果子进程在前台运行，等待直到子进程完成
If running child in foreground, wait until it is done.

shellex.c

简单的Shell eval函数 Simple Shell eval Function

```
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 termin
       if (!bg) {
            int status;
            if (waitpid(pid, &status, 0) < 0)
                unix_error("waitfg: waitpid error");
        }
        else
            printf("%d %s", pid, cmdline);
    }
    return;
}
```

如果子进程在后台运行，打印pid并继续做其它事情

If running child in background, print pid and continue doing other stuff.

简单的Shell eval函数 Simple Shell `eval` Function

```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 termina
       if (!bg) {
            int status;
            if (waitpid(pid, &status, 0) < 0)
                unix_error("waitfg: waitpid error");
        }
        else
            printf("%d %s", pid, cmdline);
    }
    return;
}
```

shellex.c

哎呀。此代码有问题。

Oops. *There is a problem with this code.*

简单Shell程序存在的问题
Problem with Simple Shell Example

- **Shell设计成无限循环运行 Shell designed to run indefinitely**
  - 不应该积累不需要的资源 Should not accumulate unneeded resources
    - 内存 Memory
    - 子进程 Child processes
    - 文件描述符 File descriptors
- 例子**shell**只能等待并回收前台作业 **Our example shell correctly waits for and reaps foreground jobs**
- 后台作业怎么办？ **But what about background jobs?**
  - 终止后变成僵尸 Will become zombies when they terminate
  - 由于shell不会终止，所以永远不会被回收 Will never be reaped because shell (typically) will not terminate
  - 会造成系统内存泄露并耗尽内核内存 Will create a memory leak that could run the kernel out of memory

可以利用ECF解决 ECF to the Rescue!

- **解决方案：异常控制流 Solution: Exceptional control flow**
  - 在后台进程处理完成后，内核打断正常处理流程并提醒我们 The kernel will interrupt regular processing to alert us when a background process completes
  - Unix系统中这种提醒的机制是信号 In Unix, the alert mechanism is called a ***signal***

议题

- 外壳 Shells
- 信号 Signals
- 非局部跳转 Nonlocal jumps

信号 Signals

- 信号是一条小消息，用来通知一个进程某种类型的事件在系统中发生了 **A *signal* is a small message that notifies a process that an event of some type has occurred in the system**
  - 类似于异常和中断 Akin to exceptions and interrupts
  - 由内核发送给一个进程（有时是根据另一个进程的请求）Sent from the kernel (sometimes at the request of another process) to a process
  - 信号的类型是用1-30的小整型标识 Signal type is identified by small integer ID's (1-30)
  - 信号的唯一信息就是这个ID以及信号达到的事实 Only information in a signal is its ID and the fact that it arrived

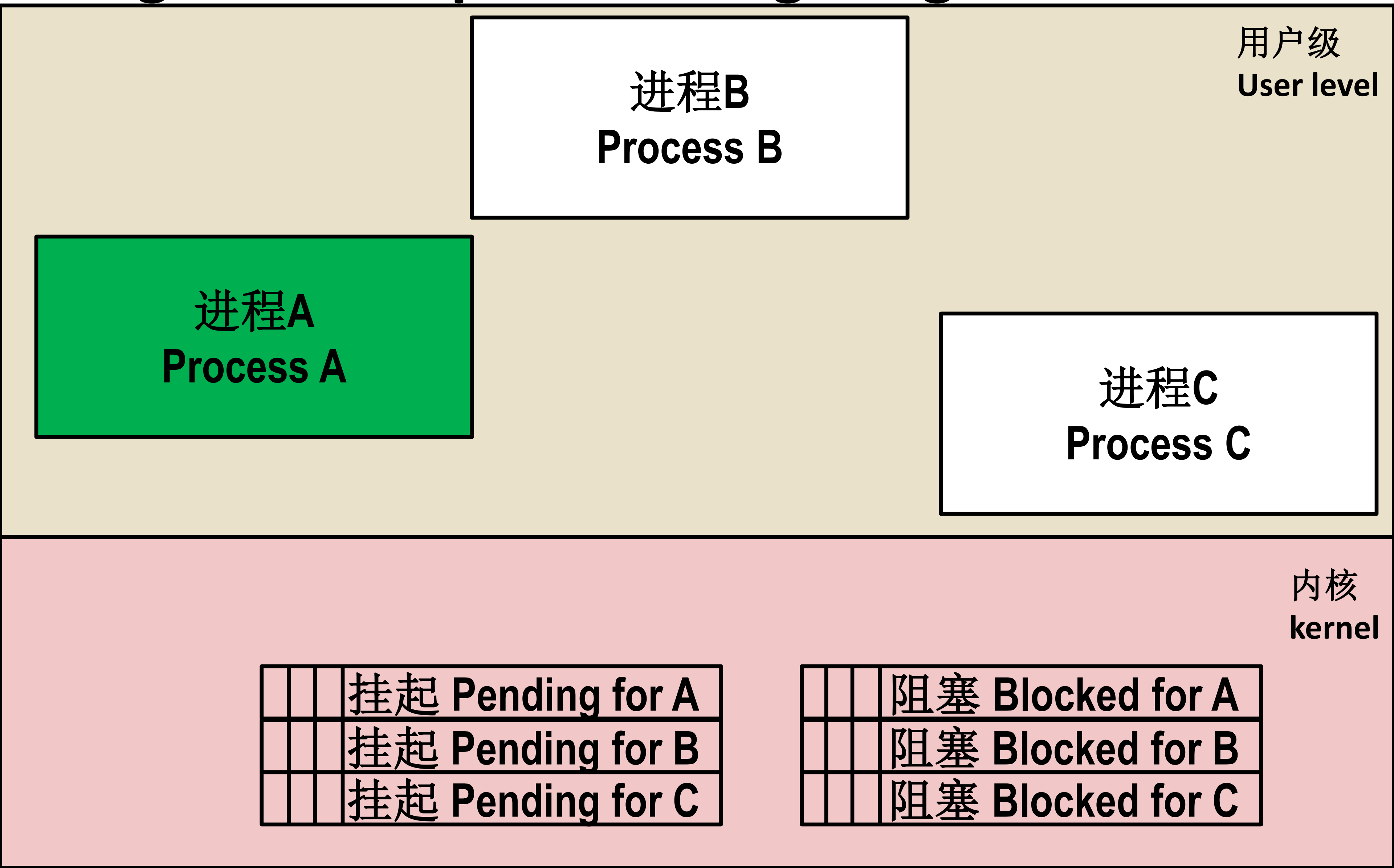
ID	*Name*	*Default Action*	*Corresponding Event*
2	SIGINT	Terminate	用户输入ctrl-c User typed ctrl-c
9	SIGKILL	Terminate	杀死程序（不能覆盖或被忽略）Kill program (cannot override or ignore)
11	SIGSEGV	Terminate	段错误 Segmentation violation
14	SIGALRM	Terminate	时钟信号 Timer signal
17	SIGCHLD	Ignore	子进程停止或者终止 Child stopped or terminated

信号概念：发送一个信号
Signal Concepts: Sending a Signal

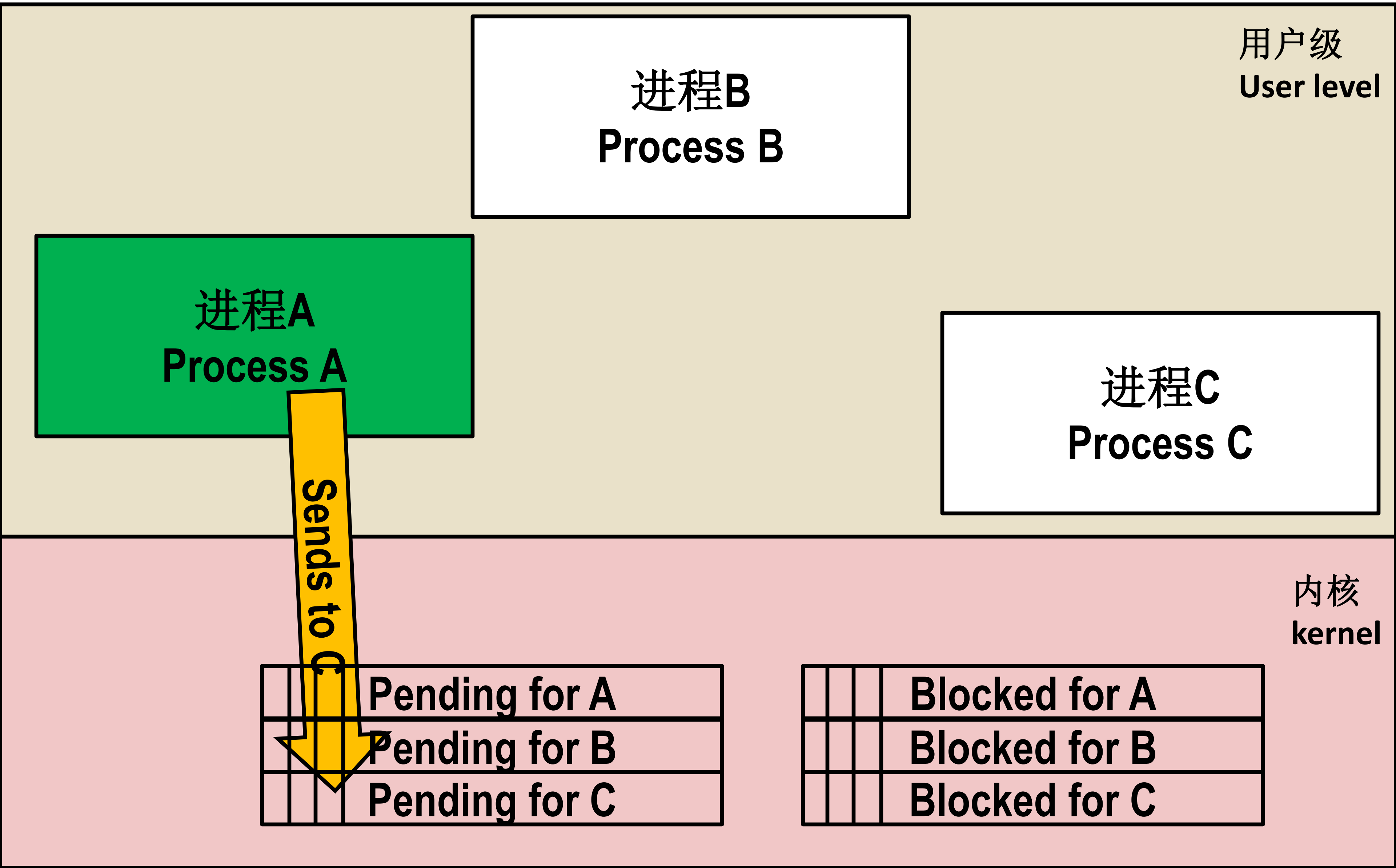
- 内核通过更新目标进程上下文的某些状态来*发送*（传递）一个信号给*目标进程* **Kernel *sends* (delivers) a signal to a *destination process* by updating some state in the context of the destination process**

- 内核发送信号是由于以下原因之一 **Kernel sends a signal for one of the following reasons:**
  - 内核侦测到除零错误（SIGFPE）或者子进程终止（SIGCHLD）等系统事件 Kernel has detected a system event such as divide-by-zero (SIGFPE) or the termination of a child process (SIGCHLD)
  - 另外一个进程调用了kill系统调用显式请求内核发送一个信号给目标进程 Another process has invoked the `kill` system call to explicitly request the kernel to send a signal to the destination process

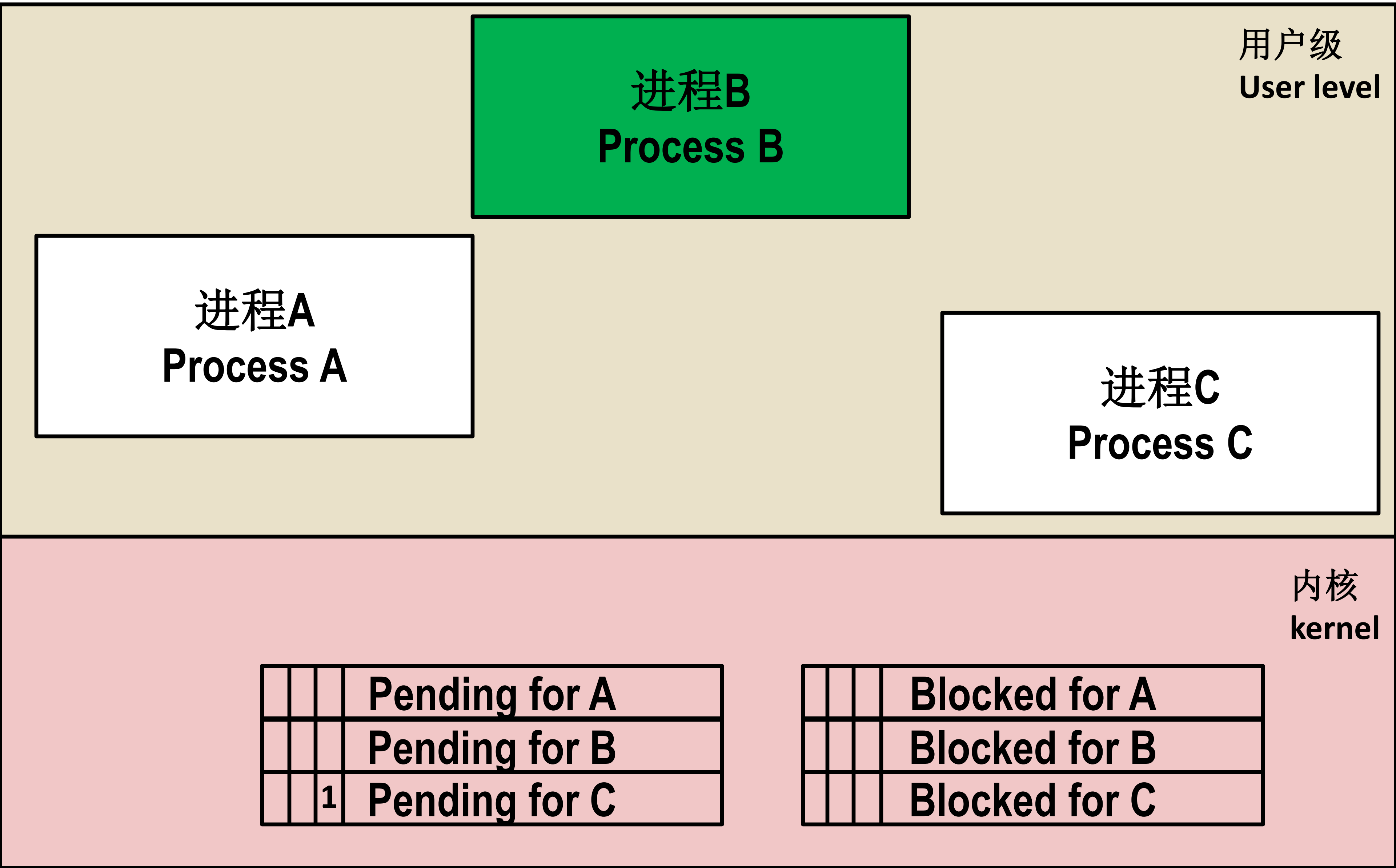
信号概念：发送一个信号
Signal Concepts: Sending a Signal



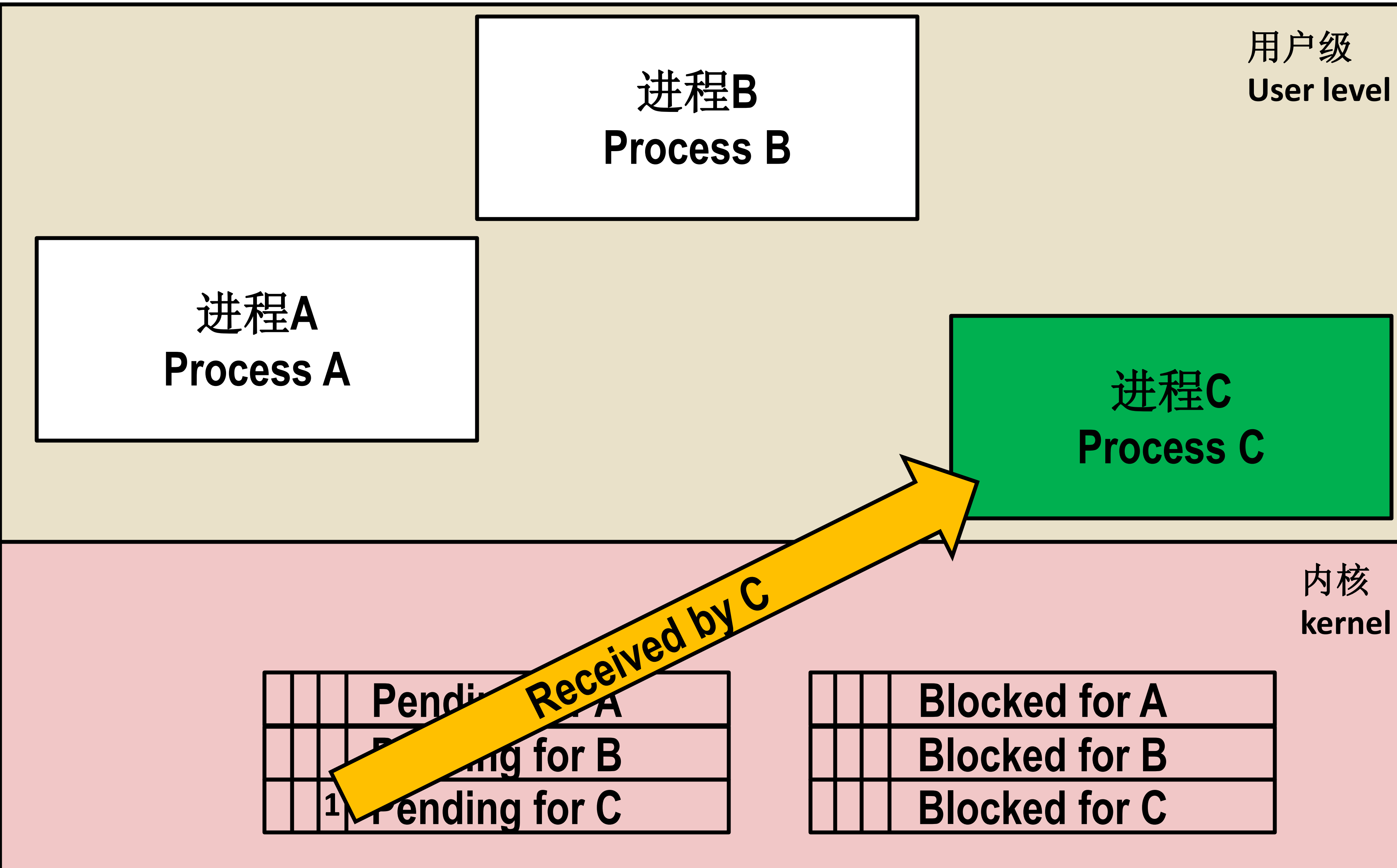
信号概念：发送一个信号
Signal Concepts: Sending a Signal



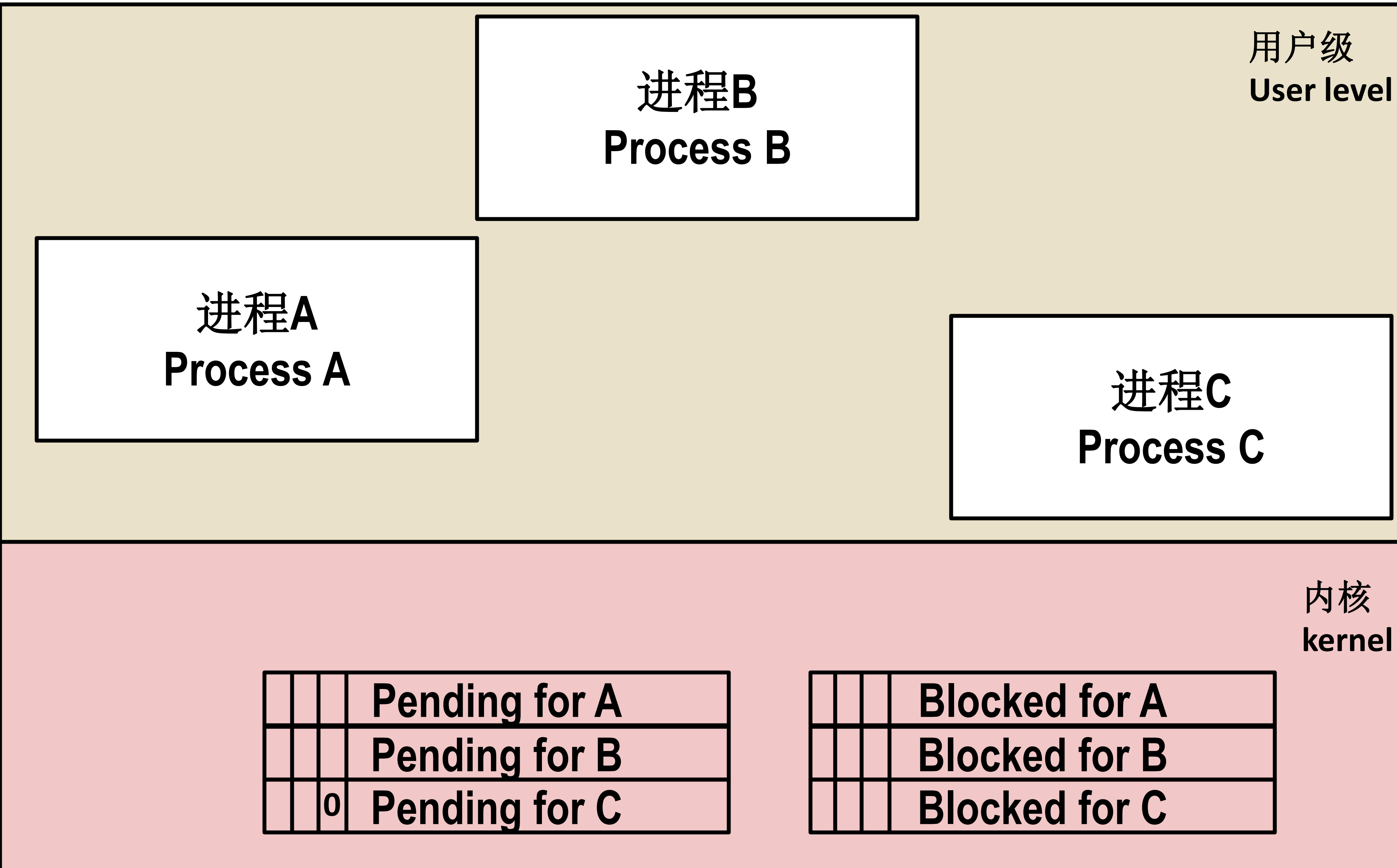
信号概念：发送一个信号
Signal Concepts: Sending a Signal



信号概念：发送一个信号Signal
Concepts: Sending a Signal

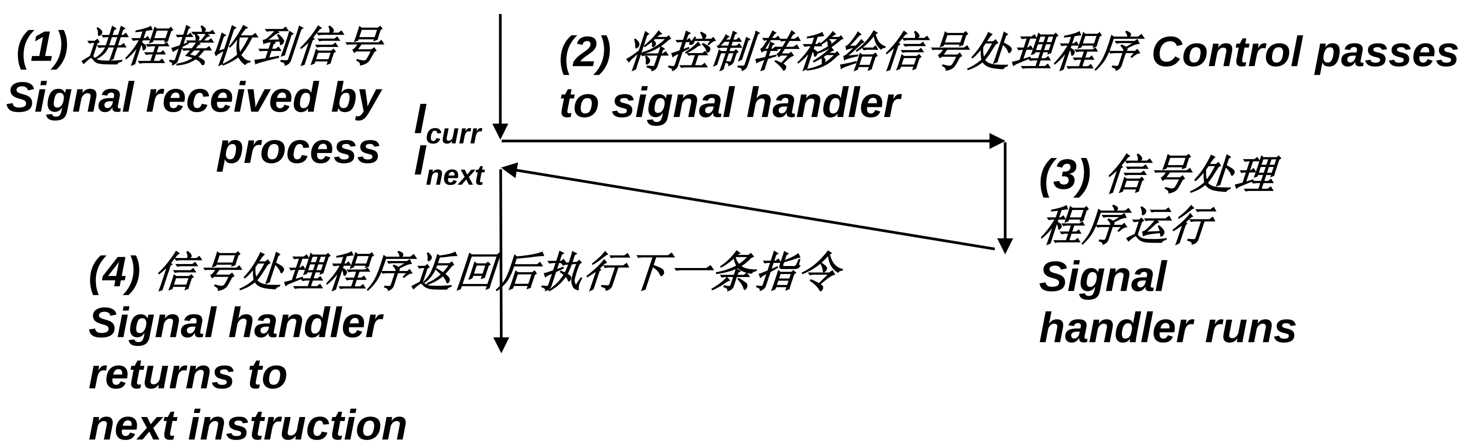


信号概念：发送一个信号Signal
Concepts: Sending a Signal



信号概念：接收一个信号
Signal Concepts: Receiving a Signal

- 目标进程***接收***信号是由于系统内核强制其对某个信号的发送做出响应
  A destination process *receives* a signal when it is forced by the kernel to react in some way to the delivery of the signal
- 可能的响应方式 **Some possible ways to react:**
  - ***忽略***信号（什么也不做） ***Ignore*** the signal (do nothing)
  - ***终止进程***（可以选择对信息转储） ***Terminate*** the process (with optional core dump)
  - ***调用***用户级***信号处理函数***对信号进行处理 ***Catch*** the signal by executing a user-level function called ***signal handler***
    - 类似于硬件异常处理函数对异步中断的响应 Akin to a hardware exception handler being called in response to an asynchronous interrupt:



信号概念：挂起或者阻塞的信号
Signal Concepts: Pending and Blocked Signals

- 已经发送但是没有被接收的信号处于*挂起*状态 **A signal is *pending* if sent but not yet received**
  - 任何特定类型的信号最多有一个挂起的 There can be at most one pending signal of any particular type
  - 重要：信号不排队 Important: Signals are not queued
    - 如有某个进程有一个类型为k的信号挂起，则后续发给该进程的k类信号被直接抛弃 If a process has a pending signal of type k, then subsequent signals of type k that are sent to that process are discarded

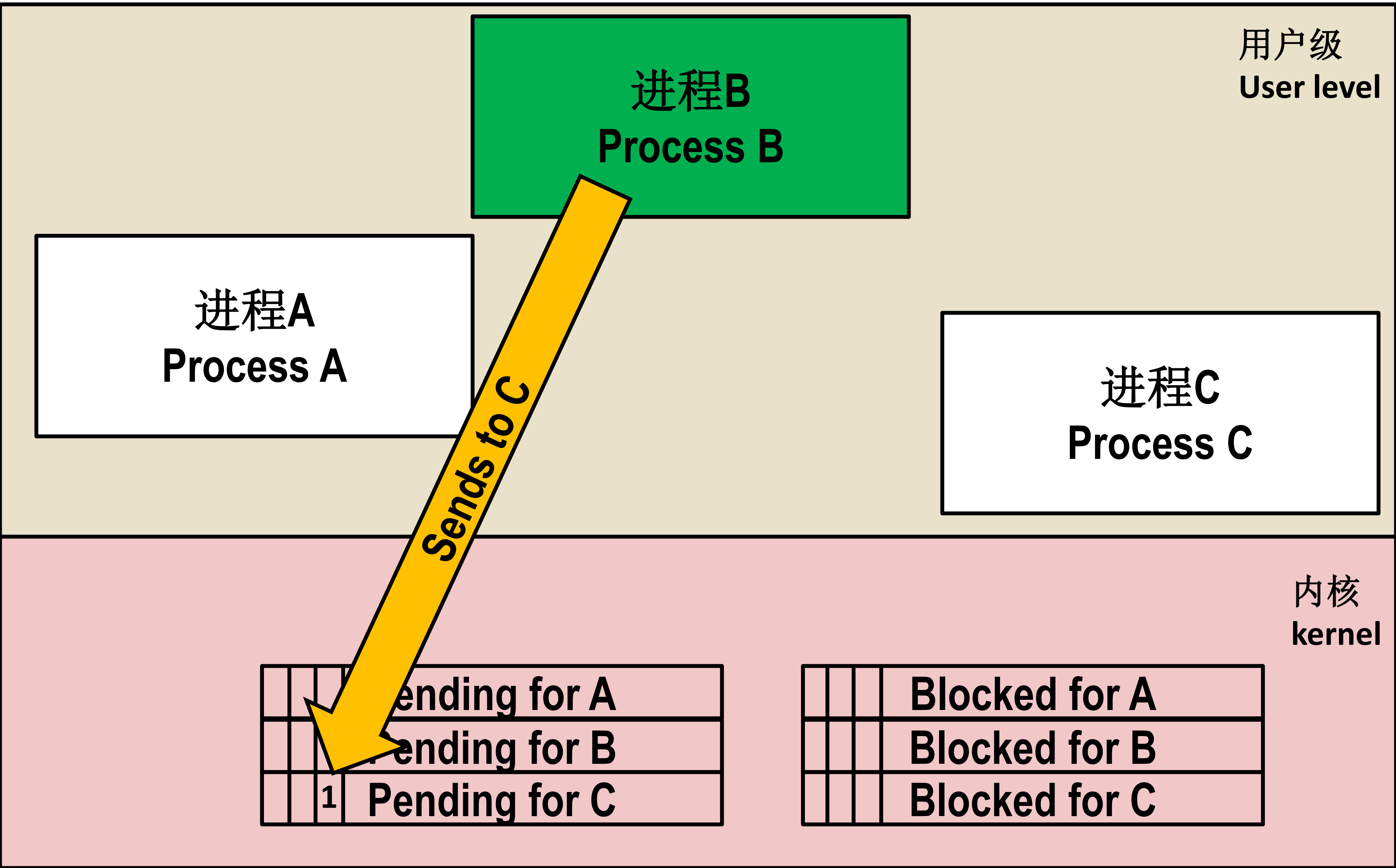
- 一个进程会*阻塞*某种特定类型信号的接收 **A process can *block* the receipt of certain signals**
  - 阻塞的信号可以发送，但是在解除阻塞前不会被接收 Blocked signals can be delivered, but will not be received until the signal is unblocked
  - 有些信号不能被阻塞（SIGKILL, SIGSTOP）或者仅当其它进程发送（SIGSEGV、SIGILL等）时被阻塞 Some signals cannot be blocked (SIGKILL, SIGSTOP) or can only be blocked when sent by other processes (SIGSEGV, SIGILL, etc)

- 挂起的信号最多被接收一次 **A pending signal is received at most once**

信号概念：挂起/阻塞位
Signal Concepts: Pending/Blocked Bits

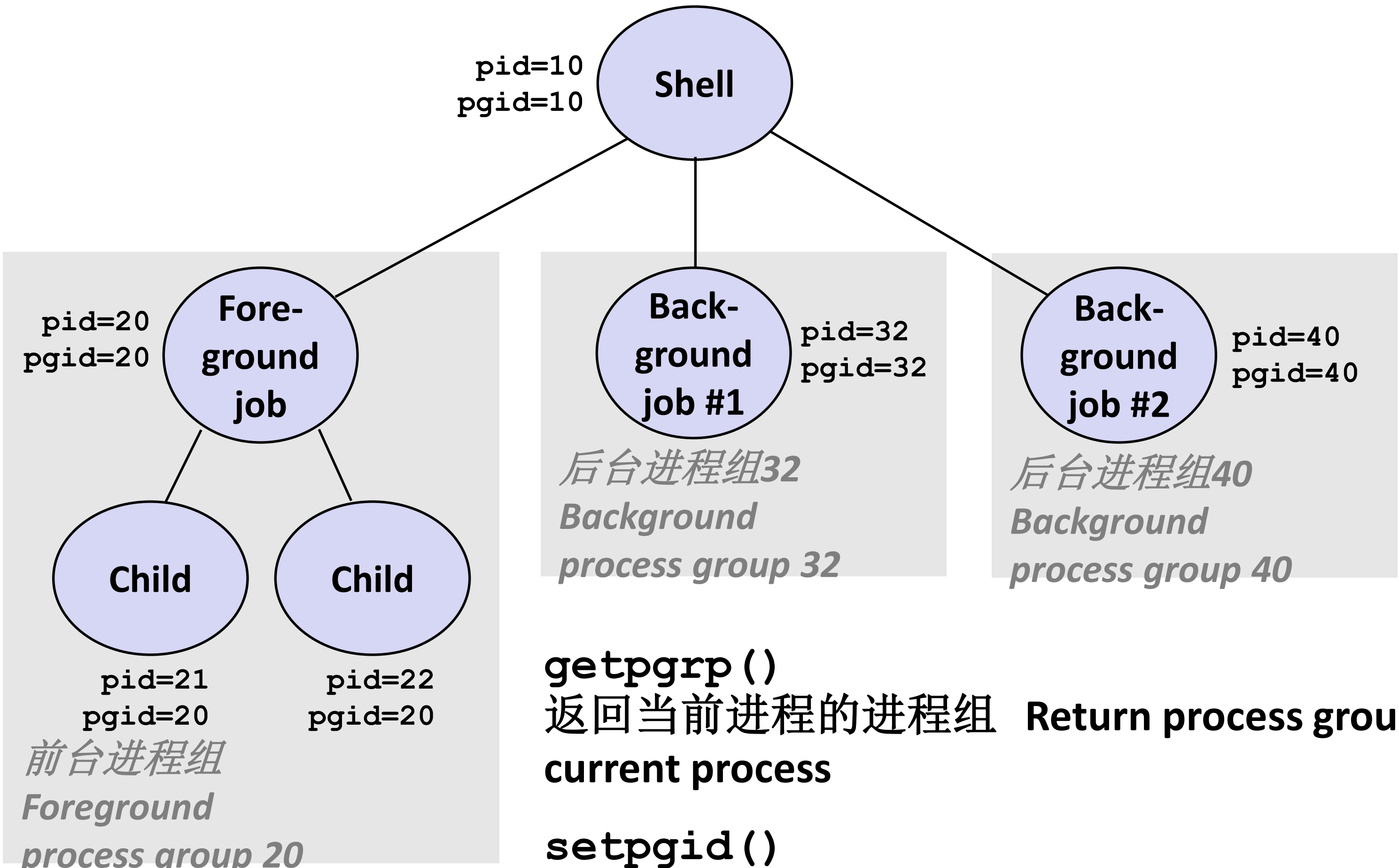
- 内核在每个进程的上下文维护一个挂起和阻塞的比特向量 **Kernel maintains `pending` and `blocked` bit vectors in the context of each process**
  - **挂起：表示挂起的信号集合** **`pending`**: represents the set of pending signals
    - 当发送了一个k类型的信号时系统设置第k个比特位 Kernel sets bit k in **`pending`** when a signal of type k is delivered
    - 当类型k的信号被接收后系统会将第k个比特位清零 Kernel clears bit k in **`pending`** when a signal of type k is received
  - **阻塞：表示阻塞的信号集合** **`blocked`**: represents the set of blocked signals
    - 可以使用**`sigprocmask`**函数设置或者清除 Can be set and cleared by using the **`sigprocmask`** function
    - 也称为信号掩码 Also referred to as the *signal mask*.

信号概念：发送信号
Signal Concepts: Sending a Signal



发送信号：进程组
Sending Signals: Process Groups

- 每个进程只属于一个进程组 **Every process belongs to exactly one process group**



`getpgrp()`
返回当前进程的进程组 **Return process group of current process**

`setpgid()`
修改当前进程的进程组（细节见教材） **Change process group of a process (see text for details)**

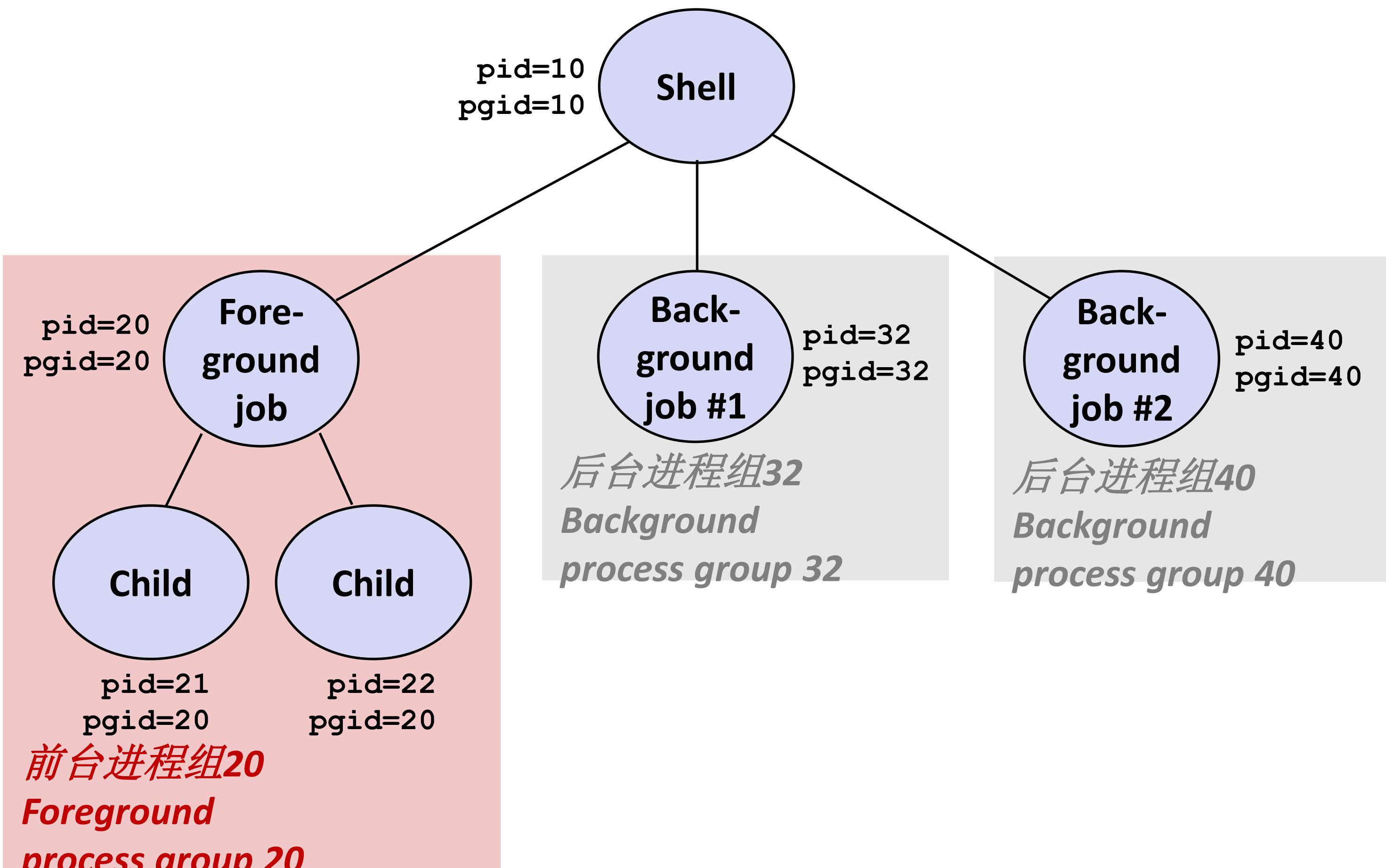
通过/bin/kill程序发送信号
Sending Signals with `/bin/kill` Program

- `/bin/kill`程序可以发送任意信号给一个进程或者进程组 `/bin/kill` program sends arbitrary signal to a process or process group
- 例如 Examples
  - `/bin/kill -9 24818` **发送SIGKILL给进程24818** Send SIGKILL to process 24818
  - `/bin/kill -9 -24817` **发送SIGKILL给进程组的每个进程** Send SIGKILL to every process in process group 24817

```
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>
```

通过键盘发送信号 Sending Signals from the Keyboard

- **输入ctrl-c(ctrl-z)会导致系统内核发送一个SIGINT (SIGTSTP) 信号给前台进程组的每个作业 Typing ctrl-c (ctrl-z) causes the kernel to send a SIGINT (SIGTSTP) to every job in the foreground process group.**
  - SIGINT – default action is to terminate each process 默认终止每个进程
  - SIGTSTP – default action is to stop (suspend) each process 默认停止（挂起）每个进程



ctrl-c和ctrl-z示例
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
```

进程状态STAT标记 **STAT (process state) Legend:**

First letter 第一个字母:

S: sleeping 睡眠
T: stopped 停止
R: running 运行

Second letter 第二个字母:

s: session leader 会话首领
+: foreground proc group 前台进程组

参见**“man ps”**了解更多细节
See “man ps” for more details

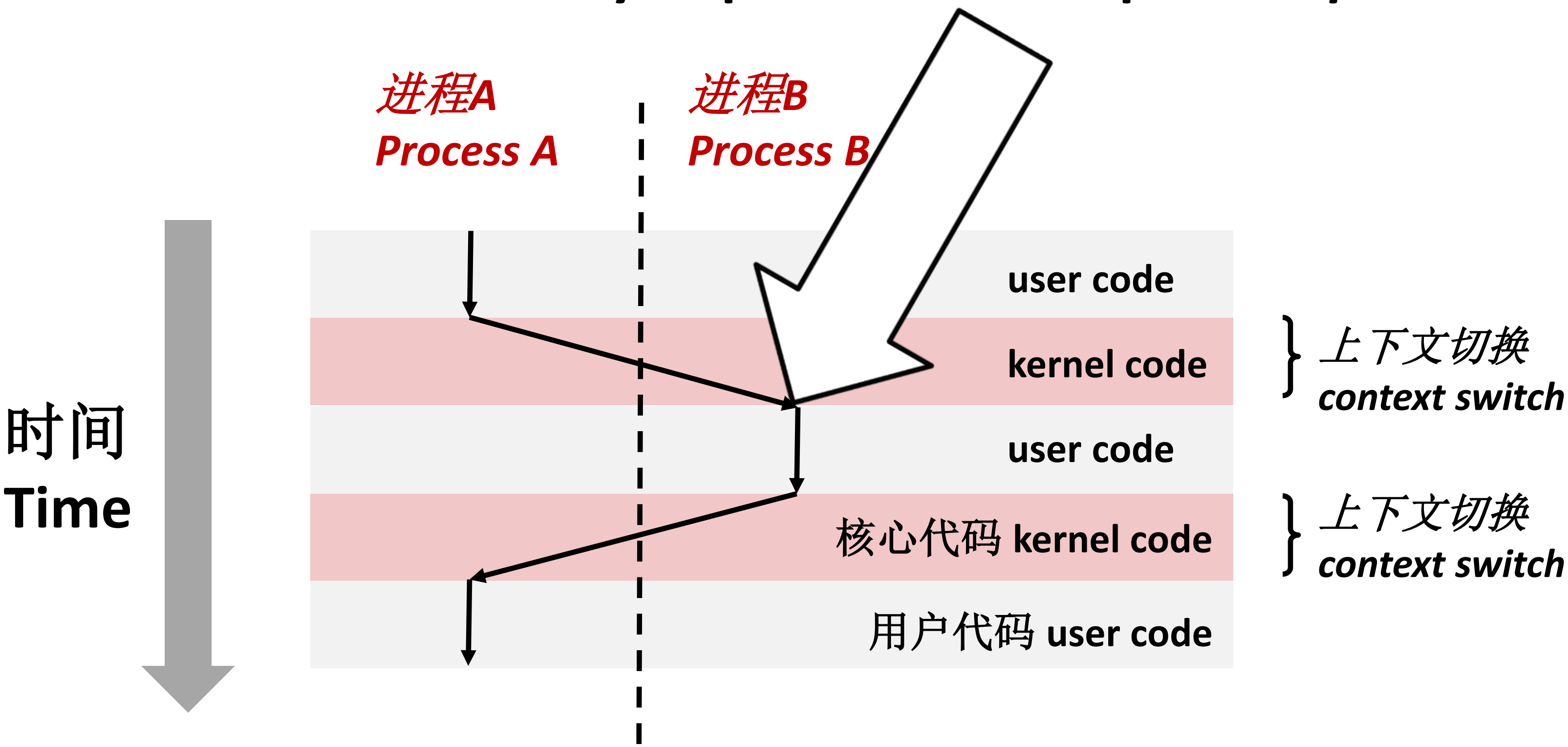
通过kill函数发送信号
Sending Signals with `kill` Function

```
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 Receiving Signals

- 假设内核正从异常处理函数返回，并准备把控制权传递给进程p　Suppose kernel is returning from an exception handler and is ready to pass control to process *p*



接收信号 Receiving Signals

- **假设内核正从异常处理函数返回，并准备把控制权传递给进程p Suppose kernel is returning from an exception handler and is ready to pass control to process *p***
- 内核计算 **Kernel computes `pnb = pending & ~blocked`**
  - 进程p挂起但非阻塞信号的集合 The set of pending nonblocked signals for process *p*
- 如果集合为空 **If (`pnb == 0`)**
  - 将控制权交给进程p逻辑流的下一条指令 Pass control to next instruction in the logical flow for *p*
- 否则 **Else**
  - 选择pnb中最低非0位k并强制进程p接收信号k Choose least nonzero bit *k* in **`pnb`** and force process *p* to ***receive*** signal *k*
  - 信号的接收触发了p的某些动作 The receipt of the signal triggers some ***action*** by *p*
  - 对pnb中每个非0位k重复上述过程 Repeat for all nonzero *k* in **`pnb`**
  - 将控制权交给进程p逻辑流的下一条指令 Pass control to next instruction in logical flow for *p*

默认动作 Default Actions

- 每种类型的信号有一个预定义的*默认动作*，可能是如下中的一个 **Each signal type has a predefined *default action*, which is one of:**
  - 终止进程 The process terminates
  - 停止进程，直到接收到SIGCONT时重启 The process stops until restarted by a SIGCONT signal
  - 进程忽略掉该信号 The process ignores the signal

安装信号处理程序 Installing Signal Handlers

- 函数signal修改接收信号signum对应的默认行为 The `signal` function modifies the default action associated with the receipt of signal `signum`:
  - `handler_t *signal(int signum, handler_t *handler)`

- 信号处理程序handler的不同值 Different values for `handler`:
  - SIG_IGN: ignore signals of type `signum` 忽略signum类型的信号
  - SIG_DFL: revert to the default action on receipt of signals of type `signum` 接收到signum类型的信号时按照默认动作处理
  - 否则handler是用户级信号处理程序的地址 Otherwise, `handler` is the address of a user-level ***signal handler***
    - 当进程接收到类型为signum的信号时调用 Called when process receives signal of type `signum`
    - 称为安装信号处理程序 Referred to as ***"installing"*** the handler
    - 执行信号处理程序称为捕获或处理该信号 Executing handler is called ***"catching"*** or ***"handling"*** the signal
    - 当信号处理程序执行返回语句时，控制权交给进程接收到信号时被打断控制流中指令 When the handler executes its return statement, control passes back to instruction in the control flow of the process that was interrupted by receipt of the signal

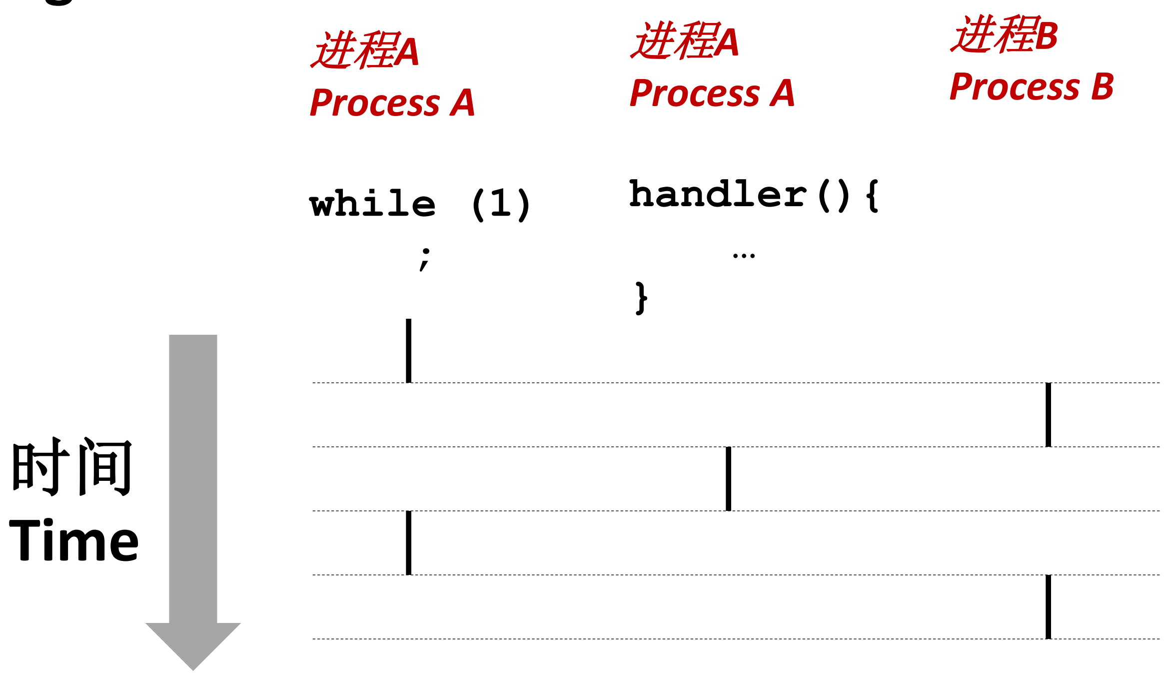
信号处理例子 Signal Handling Example

```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;
}
```

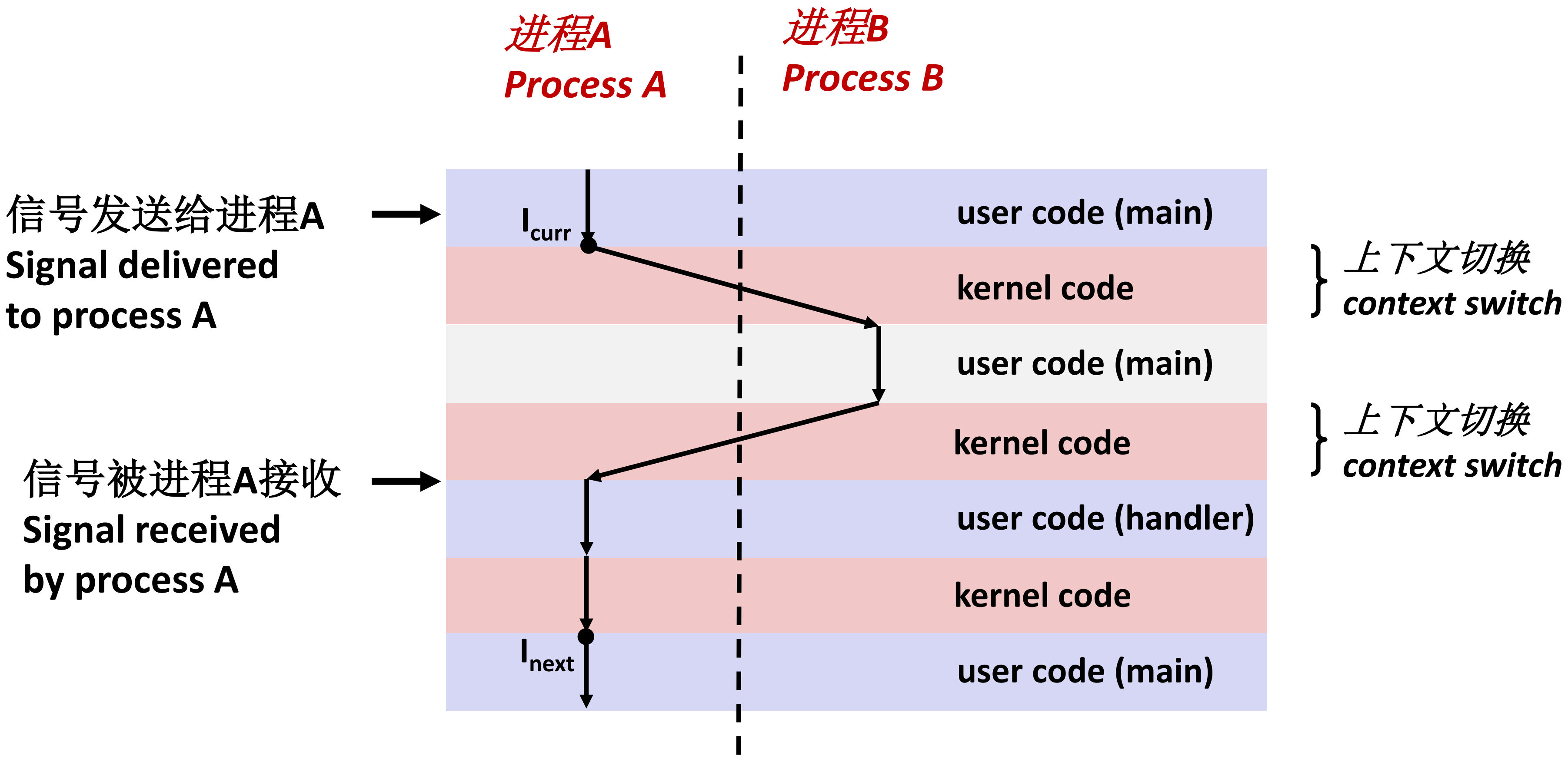
sigint.c

信号处理程序作为并发控制流
Signals Handlers as Concurrent Flows

- 每个信号处理程序都是一个独立的逻辑控制流（非进程），与主程序并发执行 **A signal handler is a separate logical flow (not process) that runs concurrently with the main program**

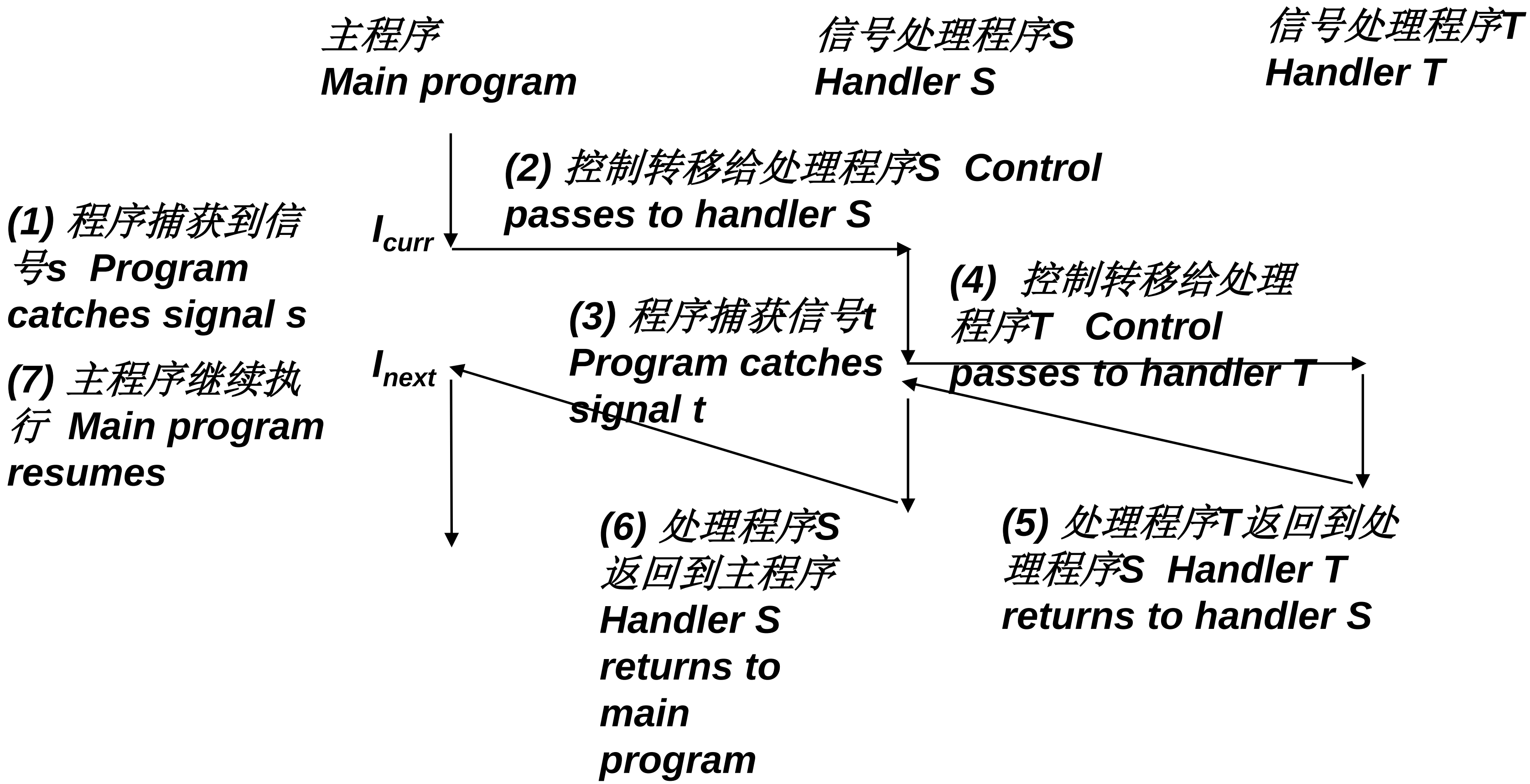


信号处理程序作为并发控制流的另一个视图
Another View of Signal Handlers as Concurrent Flows



嵌套信号处理 Nested Signal Handlers

- 信号处理程序可能被另一个信号处理程序打断
  Handlers can be interrupted by other handlers



阻塞和解除信号阻塞
Blocking and Unblocking Signals

- 隐式阻塞机制 **Implicit blocking mechanism**
  - 内核会阻塞当前正在被处理的任何挂起信号类型 Kernel blocks any pending signals of type currently being handled.
  - 例如SIGINT信号处理程序不能被另一个SIGINT打断 E.g., A SIGINT handler can’t be interrupted by another SIGINT

- 显式阻塞和解除阻塞机制 **Explicit blocking and unblocking mechanism**
  - `sigprocmask`函数 `sigprocmask` function

- 支持函数 **Supporting functions**
  - `sigemptyset` – Create empty set 创建一个空的集合
  - `sigfillset` – Add every signal number to set 对集合设置每个信号编号
  - `sigaddset` – Add signal number to set 对集合设置某个信号编号
  - `sigdelset` – Delete signal number from set 将信号编号从集合删除

临时阻塞信号
Temporarily Blocking Signals

```
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
Safe Signal Handling

- 信号处理程序比较复杂，是因为他们是和主程序并发运行的，并且共享同样的全局数据结构 **Handlers are tricky because they are concurrent with main program and share the same global data structures.**
  - 共享数据结构更容易被破坏 Shared data structures can become corrupted.

- 我们在这学期后面讨论并发的问题 **We'll explore concurrency issues later in the term.**

- 现在只给一些有助避免麻烦的提示 **For now here are some guidelines to help you avoid trouble.**

编写安全处理程序的提示
Guidelines for Writing Safe Handlers

- **G0: 信号处理程序越简单越好 Keep your handlers as simple as possible**
  - 例如，设置全局标记后返回 e.g., Set a global flag and return
- **G1: 在信号处理程序中只调用异步信号安全的函数 Call only async-signal-safe functions in your handlers**
  - `printf`, `sprintf`, `malloc`, and `exit` are not safe! 这些都不安全
- **G2: 进入和退出时保存和恢复errno Save and restore `errno` on entry and exit**
  - 以便其它的信号处理程序不会覆盖你的errno值 So that other handlers don't overwrite your value of `errno`
- **G3:临时阻塞所有的信号后再访问共享数据结构 Protect accesses to shared data structures by temporarily blocking all signals.**
  - 避免可能的破坏 To prevent possible corruption
- **G4: 将全局变量声明为volatile Declare global variables as `volatile`**
  - 避免编译器将其存储在寄存器中 To prevent compiler from storing them in a register
- **G5: 将全局标记声明为volatile `sig_atomic_t` Declare global flags as `volatile sig_atomic_t`**
  - *flag*只读或只写的变量（例如flag=1，不是flag++） *flag*: variable that is only read or written (e.g. flag = 1, not flag++)
  - 按照这种方式声明的flag变量不需要像其他全局变量那样保护 Flag declared this way does not need to be protected like other globals

异步信号安全 Async-Signal-Safety

- 如果一个函数是可重入的（例如所有变量存储在栈帧，CS:APP3e 12.7.2）或者不可以被信号打断的则将其称为*异步信号安全async-signal-safe* Function is *async-signal-safe* if either reentrant (e.g., all variables stored on stack frame, CS:APP3e 12.7.2) or non-interruptible by signals.
- Posix中有117个函数是异步信号安全async-signal-safe Posix guarantees 117 functions to be async-signal-safe
  - 来源：man命令 Source: "`man 7 signal`"
  - 在其中的常见函数包括： Popular functions on the list:
    - `_exit, write, wait, waitpid, sleep, kill`
  - 并不在其中得常见函数 Popular functions that are **not** on the list:
    - `printf, sprintf, malloc, exit`
    - 不幸的事实：write是唯一异步信号安全async-signal-safe输出函数 Unfortunate fact: `write` is the only async-signal-safe output function

安全格式化输出：选项#1
Safe Formatted Output: Option #1

- 在信号处理程序中使用csapp.c的可重入的SIO（安全I/O库）
  Use the reentrant SIO (Safe I/O library) from `csapp.c` in your handlers
  - `ssize_t sio_puts(char s[]) /* Put string */`
  - `ssize_t sio_putl(long v)   /* Put long */`
  - `void sio_error(char s[])   /* Put msg & exit */`

```
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

安全格式化输出：选项#2
Safe Formatted Output: Option #2

- 使用新的且改进的可重入**`sio_printf`!** **Use the new & improved reentrant `sio_printf`!**
  - 处理printf受限类的格式串 Handles restricted class of `printf` format strings
    - 识别： Recognizes: **`%c %s %d %u %x %%`**
    - 大小指定符： Size designators '**`l`**' and '**`z`**'

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

正确的信号处理
Correct Signal Handling

```
volatile 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 */
        ;
}
```

N == 5

这段代码不正确！
This code is incorrect!

- 挂起的信号是不排队的 **Pending signals are not queued**
  - 对每个信号类型，只用一个比特位来标识是否有信号被挂起 For each signal type, one bit indicates whether or not signal is pending…
  - 因此每种最多有一个挂起的信号 …thus at most one pending signal of any particular type.
- 不可以使用信号对事件计数，例如子进程终止等 **You can't use signals as**

```
whaleshark> ./forks 14
Handler reaped child 23240
Handler reaped child 23241
...(hangs)
```

正确信号处理 Correct Signal Handling

- 必须等待所有终止的子进程 **Must wait for all terminated child processes**
  - 将wait放入到循环中以回收所有终止的子进程 Put `wait` in a loop to reap all terminated children

```
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;
}
```

```
whaleshark> ./forks 15
Handler reaped child 23246
Handler reaped child 23247
Handler reaped child 23248
Handler reaped child 23249
Handler reaped child 23250
whaleshark>
```

可移植的信号处理
Portable Signal Handling

- 不同的Unix版本有不同的信号处理语义 **Ugh! Different versions of Unix can have different signal handling semantics**
  - 一些早期的系统在捕获到信号后会恢复默认动作 Some older systems restore action to default after catching signal
  - 有些被中断的系统调用会返回errno == EINTR Some interrupted system calls can return with errno == EINTR
  - 有的系统并不阻塞正在被处理的信号类型 Some systems don't block signals of the type being handled
- 解决方案：**sigaction** **Solution: `sigaction`**

```
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

同步控制流避免竞争
Synchronizing Flows to Avoid Races

- 简单shell的SIGCHLD处理程序 **SIGCHLD handler for a simple shell**
  - 当运行临界代码时阻塞所有信号 Blocks all signals while running critical code

```
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 (pid != 0 && errno != ECHILD)
        sio_error("waitpid error");
    errno = olderrno;
}
```

procmask1.c

同步控制流避免竞争
Synchronizing Flows to Avoid Races

- 简单的shell程序有个不易发现的同步问题，因为其假设父进程先于子进程　Simple shell with a subtle synchronization error because it assumes parent runs before child

```
int main(int argc, char **argv)
{
    int pid;
    sigset_t mask_all, prev_all;
    int n = N;  /* N = 5 */
    sigfillset(&mask_all);
    signal(SIGCHLD, handler);
    initjobs(); /* Initialize the job list */

    while (n--) {
        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

没有竞争问题的修正shell程序
Corrected Shell Program Without Race

```
int main(int argc, char **argv)
{
    int pid;
    sigset_t mask_all, mask_one, prev_one;
    int n = N; /* N = 5 */
    sigfillset(&mask_all);
    sigemptyset(&mask_one);
    sigaddset(&mask_one, SIGCHLD);
    signal(SIGCHLD, handler);
    initjobs(); /* Initialize the job list */

    while (n--) {
        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

显式等待信号
Explicitly Waiting for Signals

- 信号处理程序显式等待SIGCHLD信号的到来 **Handlers for program explicitly waiting for SIGCHLD to arrive**

```c
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;
}

void sigint_handler(int s)
{
}
```

waitforsignal.c

显式等待信号 Explicitly Waiting for Signals

```
int main(int argc, char **argv) {
    sigset_t mask, prev;
    int n = N; /* N = 10 */
    signal(SIGCHLD, sigchld_handler);
    signal(SIGINT, sigint_handler);
    sigemptyset(&mask);
    sigaddset(&mask, SIGCHLD);

    while (n--) {
        sigprocmask(SIG_BLOCK, &mask, &prev); /* Block SIGCHLD */
        if (fork() == 0) /* Child */
            exit(0);
        /* Parent */
        pid = 0;
        sigprocmask(SIG_SETMASK, &prev, NULL); /* Unblock SIGCHLD */

        /* Wait for SIGCHLD to be received (wasteful!) */
        while (!pid)
            ;
        /* Do some work after receiving SIGCHLD */
        printf(".");
    }
    printf("\n");
    exit(0);
}
```

类似于shell等待一个前台的作业终止
Similar to a shell waiting
for a foreground job to terminate.

waitforsignal.c

显式等待信号 Explicitly Waiting for Signals

```
while (!pid)
    ;
```

- **程序是对的，但是太浪费资源 Program is correct, but very wasteful**
  - 程序忙于等待循环 Program in busy-wait loop

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

- **可能存在竞争 Possible race condition**
  - 在检查pid和开始暂停之间，可能接收信号 Between checking pid and starting pause, might receive signal

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

- **安全，但是很慢 Safe, but slow**
  - 会占用1秒钟才能响应 Will take up to one second to respond

使用sigsuspend等待信号
Waiting for Signals with sigsuspend

- `int sigsuspend(const sigset_t *mask)`

- 等价于原子版本（无中断可能）的：　**Equivalent to atomic (uninterruptable) version of:**

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

使用sigsuspend等待信号
Waiting for Signals with sigsuspend

```
int main(int argc, char **argv) {
    sigset_t mask, prev;
    int n = N; /* N = 10 */
    signal(SIGCHLD, sigchld_handler);
    signal(SIGINT, sigint_handler);
    sigemptyset(&mask);
    sigaddset(&mask, SIGCHLD);
    while (n--) {
        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(".");
    }
    printf("\n");
    exit(0);
}
```

sigsuspend.c

议题

- 外壳 Shells
- 信号 Signals
- **非局部跳转 Nonlocal jumps**
  - 参见教材和附加的幻灯片 Consult your textbook and additional slides

总结 Summary

- **信号提供进程级异常处理 Signals provide process-level exception handling**
  - 可以从用户程序产生 Can generate from user programs
  - 可以声明信号处理程序定义处理效果 Can define effect by declaring signal handler
  - 编写信号处理函数的时候要特别小心 Be very careful when writing signal handlers

- **非局部跳转给出了进程内部的异常控制流 Nonlocal jumps provide exceptional control flow within process**
  - 遵守栈相关的原则 Within constraints of stack discipline

附加的幻灯片 Additional slides

非局部跳转
Nonlocal Jumps: `setjmp/longjmp`

- 将控制转移到任意位置的强大（但比较危险）用户级机制 **Powerful (but dangerous) user-level mechanism for transferring control to an arbitrary location**
  - 受控的打破call/return规则的方式 Controlled to way to break the procedure call / return discipline
  - 通常用于错误恢复和信号处理 Useful for error recovery and signal handling

- `int setjmp(jmp_buf j)`
  - 必须在longjmp之前调用 Must be called before longjmp
  - 给出后续longjmp对应的返回位置 Identifies a return site for a subsequent longjmp
  - 一次调用，返回一次或者多次 Called **once**, returns **one or more** times

- 实现 **Implementation:**
  - 通过将当前寄存器上下文、栈指针和PC值存储在jmp_buf中记住当前位置 Remember where you are by storing the current ***register context***, ***stack pointer***, and ***PC value*** in `jmp_buf`
  - 返回0 Return 0

`setjmp/longjmp` (续 cont)

- `void longjmp(jmp_buf j, int i)`
  - 含义 Meaning:
    - 从setjmp返回，再次被跳转缓冲区j记住 return from the **`setjmp`** remembered by jump buffer **`j`** again ...
    - 这次返回i而不是0 ... this time returning **`i`** instead of 0
  - setjmp之后调用 Called after **`setjmp`**
  - 一次调用但是从不返回 Called **once**, but **never** returns

- **`longjmp`实现 `longjmp` Implementation:**
  - 从跳转缓冲区j中恢复寄存器上下文（栈指针、基指针、PC值） Restore register context (stack pointer, base pointer, PC value) from jump buffer **`j`**
  - 将返回值寄存器%eax设置为i Set **`%eax`** (the return value) to **`i`**
  - 跳转到跳转缓冲j中PC指定的位置 Jump to the location indicated by the PC stored in jump buf **`j`**

setjmp/longjmp Example 示例

- 目标：从深度嵌套的函数直接返回最开始的调用者
- Goal: return directly to original caller from a deeply-nested function

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

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

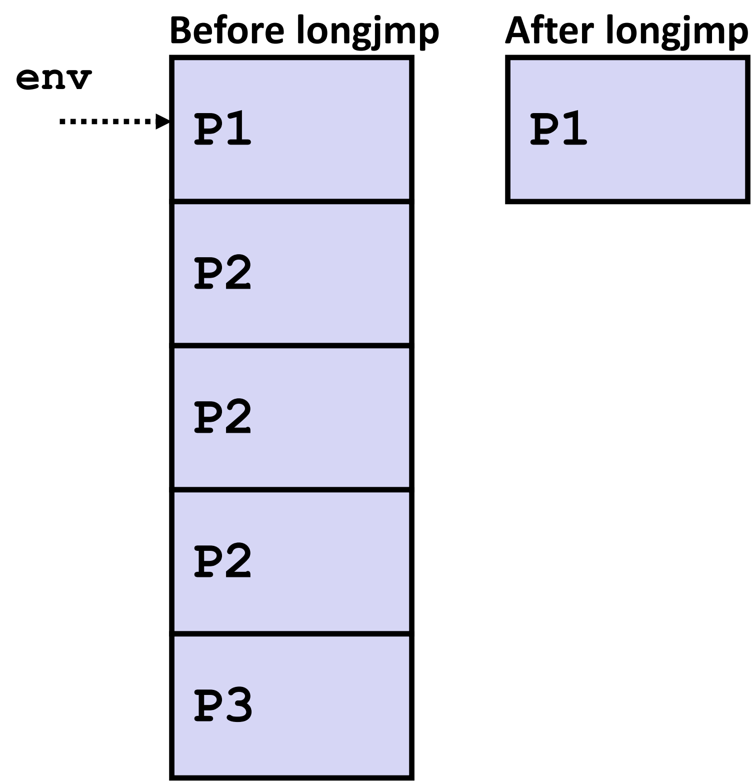
setjmp/longjmp 示例/Example (续/cont)

```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);
}
```

非局部跳转的限制
Limitations of Nonlocal Jumps

- 基于栈原理工作 **Works within stack discipline**
  - 只能跳转到已经调用但是还没有完成的函数 Can only long jump to environment of function that has been called but not yet completed

```
jmp_buf env;

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

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

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

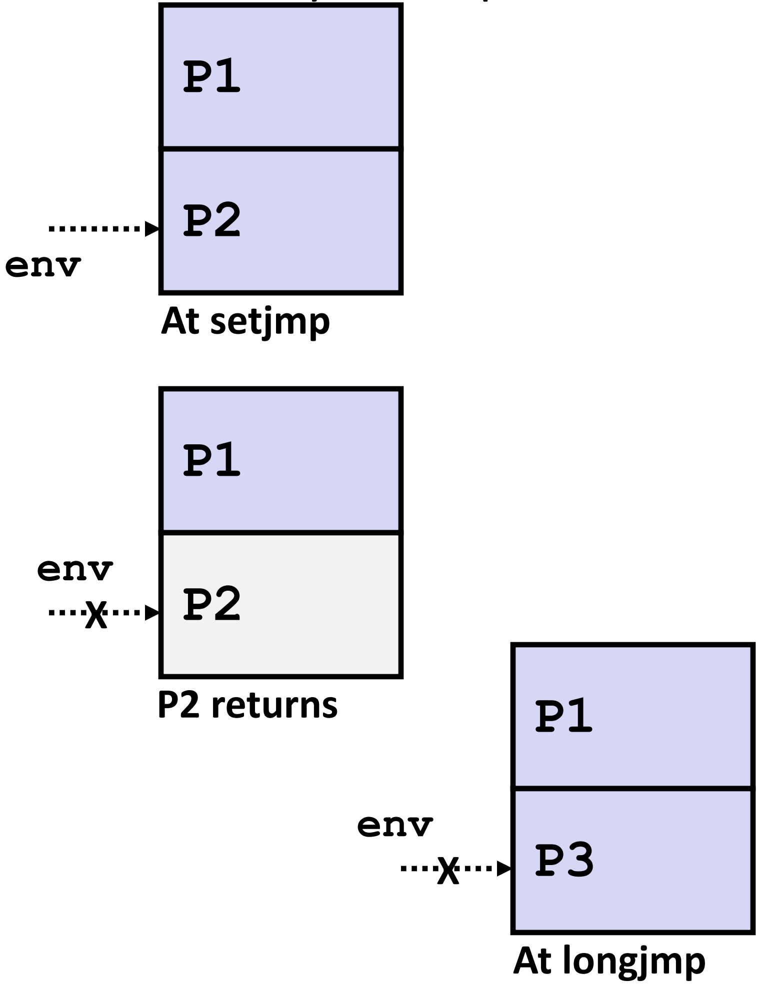


非局部跳转的限制（续）

Limitations of Long Jumps (cont.)

- 基于栈原理工作 **Works within stack discipline**
  - 只能跳转到已经调用但是还没有完成的函数/Can only long jump to environment of function that has been called but not yet completed

```
jmp_buf env;

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

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

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



整合在一起：程序在按下ctrl-c或d时重启
Putting It All Together: A Program That Restarts Itself When `ctrl-c`'d

```
#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

```
greatwhite> ./restart
starting
processing...
processing...
processing...
restarting
processing...
processing...
restarting
processing...
processing...
processing...
```

Ctrl-c

Ctrl-c