第8章 异常控制流

异常和进程 Exceptions and Processes

100076202： 计算机系统导论

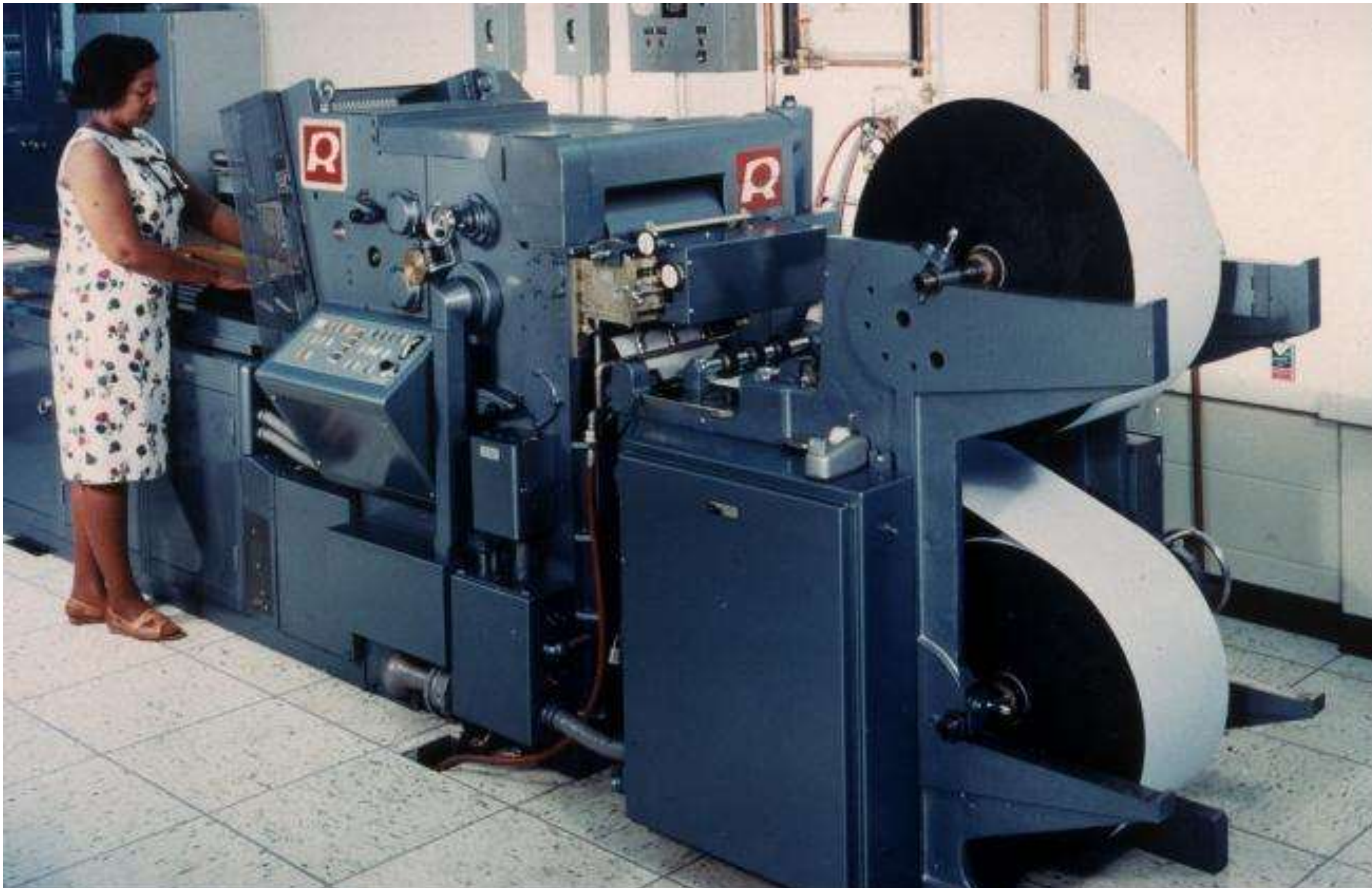
任课教师：

宿红毅　张艳　黎有琦　颜珂

原作者：

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

打印机过去常常着火
Printers Used to Catch on Fire



高度异常控制流

Highly Exceptional Control Flow

```c
static int lp_check_status(int minor)
{
        int error = 0;
        unsigned int last = lp_table[minor].last_error;
        unsigned char status = r_str(minor);
        if ((status & LP_PERRORP) && !(LP_F(minor) & LP_CAREFUL))
                /* No error. */
                last = 0;
        else if ((status & LP_POUTPA)) {
                if (last != LP_POUTPA) {
                        last = LP_POUTPA;
                        printk(KERN_INFO "lp%d out of paper\n", minor);
                }
                error = -ENOSPC;
        } else if (!(status & LP_PSELECD)) {
                if (last != LP_PSELECD) {
                        last = LP_PSELECD;
                        printk(KERN_INFO "lp%d off-line\n", minor);
                }
                error = -EIO;
        } else if (!(status & LP_PERRORP)) {
                if (last != LP_PERRORP) {
                        last = LP_PERRORP;
                        printk(KERN_INFO "lp%d on fire\n", minor);
                }
                error = -EIO;
        } else {
                last = 0; /* Come here if LP_CAREFUL is set and no
                             errors are reported. */
        }

        lp_table[minor].last_error = last;

        if (last != 0)
                lp_error(minor);

        return error;
}
```

https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/drivers/char/lp.c?h=v5.0-rc3

内容提纲

- 异常控制流 Exceptional Control Flow　　CSAPP 8
- 异常 Exceptions　　CSAPP 8.1
- 进程 Processes　　CSAPP 8.2
- 进程控制 Process Control　　CSAPP 8.3-8.4

控制流 Control Flow

- **处理器只做一件事 Processors do only one thing:**
  - 从开机到关机，每个CPU核只是读入和执行（解释）指令序列，每次一条 From startup to shutdown, each CPU core simply reads and executes (interprets) a sequence of instructions, one at a time *
  - 这个序列就是CPU的*控制流* This sequence is the CPU's *control flow* (or *flow of control*)

物理控制流 Physical control flow

Time

<startup> //开机
$inst_1$
$inst_2$
$inst_3$
…
$inst_n$
<shutdown> //关机

*从外部体系结构视角看（内部来看，CPU可以使用并行乱序执行）
* Externally, from an architectural viewpoint (internally, the CPU may use parallel out-of-order execution)

改变控制流 Altering the Control Flow

- **目前：两种改变控制流的机制：Up to now: two mechanisms for changing control flow:**
  - 跳转分支指令 Jumps and branches
  - 调用和返回指令 Call and return

  反应*程序状态*的变化 React to changes in ***program state***
- **对有用的系统来说还不够： Insufficient for a useful system: 难以反应*系统状态*的改变 Difficult to react to changes in *system state***
  - 从磁盘或者网络适配器获取的数据到达 Data arrives from a disk or a network adapter
  - 指令除零 Instruction divides by zero
  - 用户键盘按下了Ctrl-C User hits Ctrl-C at the keyboard
  - 系统定时器超时 System timer expires
- **系统需要“异常控制流”处理机制 System needs mechanisms for “exceptional control flow”**

异常控制流 Exceptional Control Flow

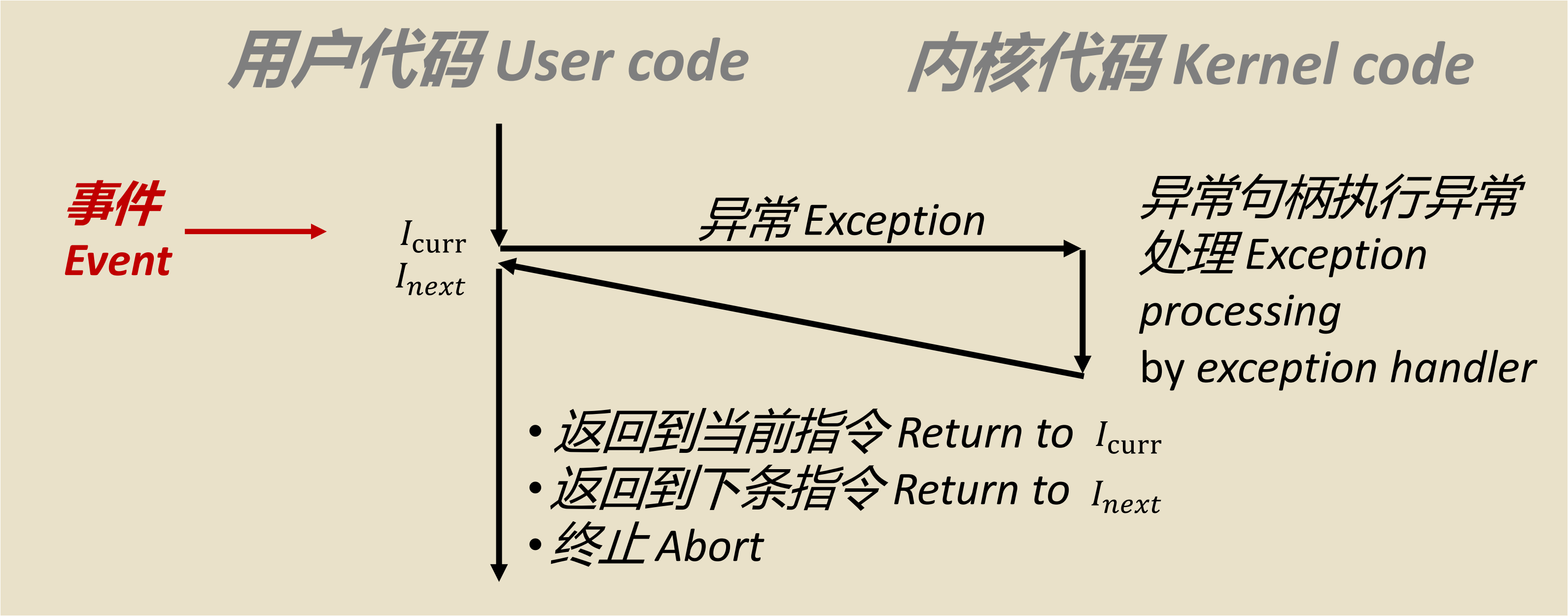
- **存在计算机系统的每个层次 Exists at all levels of a computer system**
- **低层次机制 Low level mechanisms**
  - 1.异常 **Exceptions**
    - 为响应系统事件改变控制流（例如系统状态改变） Change in control flow in response to a system event (i.e., change in system state)
    - 硬件和OS软件组合实现 Implemented using combination of hardware and OS software
- **高层次机制 Higher level mechanisms**
  - 2. 进程上下文切换 **Process context switch**
    - 硬件定时器和OS软件实现 Implemented by OS software and hardware timer
  - 3. 信号 **Signals**
    - OS软件实现 Implemented by OS software
  - 4. 非局部跳转 **Nonlocal jumps**: `setjmp()` and `longjmp()`
    - C运行时库实现 Implemented by C runtime library

内容提纲

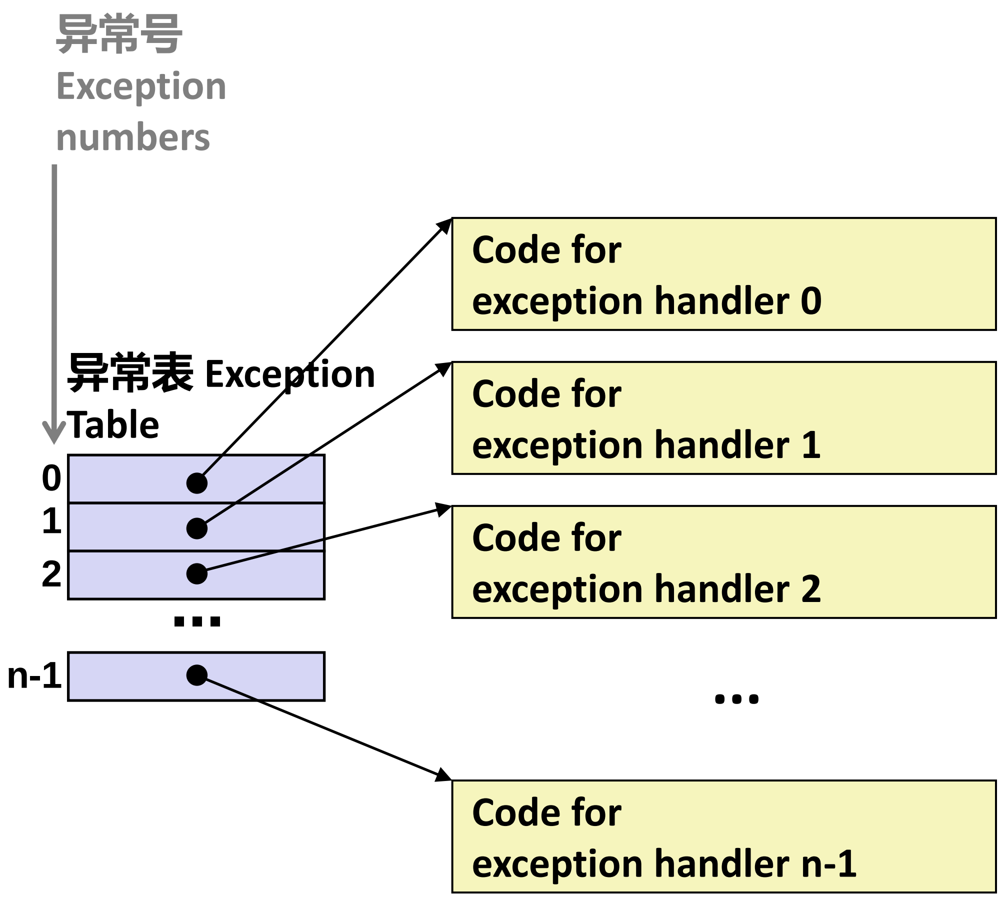
- 异常控制流 Exceptional Control Flow
- **异常 Exceptions**
- 进程 Processes
- 进程控制 Process Control

异常 Exceptions

- **异常是为了响应某些事件（即处理器状态改变）而将控制流转移到OS内核 An *exception* is a transfer of control to the OS *kernel* in response to some *event* (i.e., change in processor state)**
  - 内核是操作系统的内存驻留 Kernel is the memory-resident part of the OS
  - 事件举例：除零，算术溢出，缺页，I/O请求完成，键入Ctrl+C Examples of events: Divide by 0, arithmetic overflow, page fault, I/O request completes, typing Ctrl-C

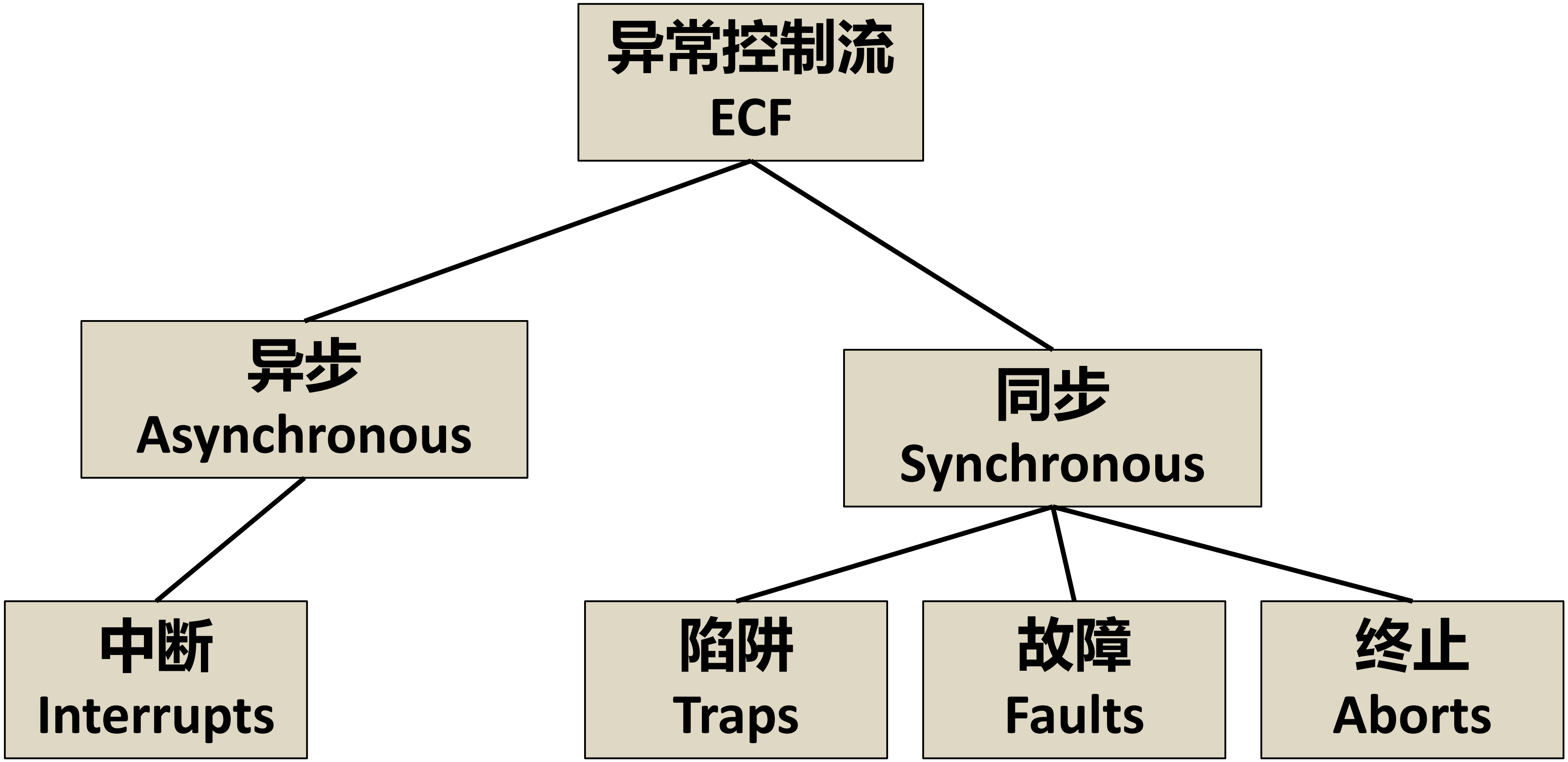


异常表 Exception Tables



- 每个事件类型有惟一的异常编号k Each type of event has a unique exception number k
- 用k做为异常表的索引（即中断向量） k = index into exception table (a.k.a. interrupt vector)
- 每次发生异常k时，就会调用句柄k（句柄就是异常处理程序指针） Handler k is called each time exception k occurs

(部分）分类 (partial) Taxonomy



异步异常（中断）
Asynchronous Exceptions (Interrupts)

- **由处理器外部事件引起 Caused by events external to the processor**
  - 通过设置处理器的中断引脚来指示有中断请求到达 Indicated by setting the processor's *interrupt pin*
  - 中断处理程序返回后执行下一条指令 Handler returns to "next" instruction
- **举例 Examples:**
  - 时钟中断 Timer interrupt
    - 每隔大约几ms，外部时钟芯片触发一个中断 Every few ms, an external timer chip triggers an interrupt
    - 将控制权从用户程序切换到内核 Used by the kernel to take back control from user programs
  - 外部设备的I/O中断 I/O interrupt from external device
    - 键盘键入Ctrl-C Hitting Ctrl-C at the keyboard
    - 网络有一个包抵达 Arrival of a packet from a network
    - 从磁盘有数据抵达 Arrival of data from a disk

同步异常 Synchronous Exceptions

- **指令执行结果导致的异常事件 Caused by events that occur as a result of executing an instruction:**
  - ***陷入/陷阱 Traps***
    - 人为的 Intentional
    - 例如：**系统调用**、断点、特殊指令等 Examples: ***system calls***, breakpoint traps, special instructions
    - 控制流返回到下一条指令 Returns control to “next” instruction
  - ***故障 Faults***
    - 不是有意的但是大概率可恢复 Unintentional but possibly recoverable
    - 例如：缺页异常（可恢复）、保护异常（不可恢复）、浮点异常 Examples: page faults (recoverable), protection faults (unrecoverable), floating point exceptions
    - 重新执行故障（“当前”）指令或者终止执行 Either re-executes faulting (“current”) instruction or aborts
  - ***终止 Aborts***
    - 非故意且不可恢复 Unintentional and unrecoverable
    - 例如：非法指令、校验错、机器检查 Examples: illegal instruction, parity error, machine check
    - 终止当前程序执行 Aborts current program

系统功能调用 System Calls

- **每个x86-64系统调用都有一个唯一的ID编号 Each x86-64 system call has a unique ID number**
- **例如： Examples:**

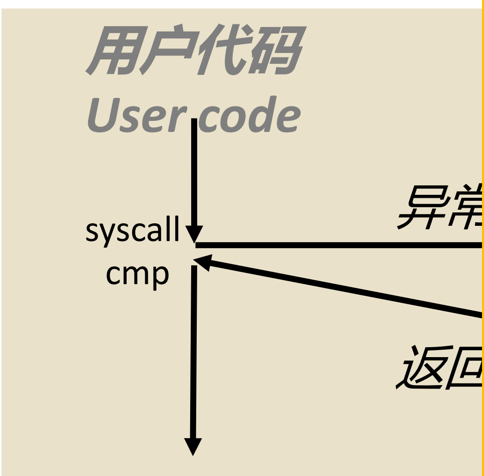
编号 Number	名字 Name	描述 Description
0	`read`	读文件 Read file
1	`write`	写文件 Write file
2	`open`	打开文件 Open file
3	`close`	关闭文件 Close file
4	`stat`	获取有关文件的信息 Get info about file
57	`fork`	创建进程 Create process
59	`execve`	执行一个程序 Execute a program
60	`_exit`	终止进程 Terminate process
62	`kill`	发送信号给进程 Send signal to process

系统调用举例：打开文件

System Call Ex

- 用户调用：open
- 调用__open函数，
function, which in

```
00000000000e5d70 <__
...
e5d79:   b8 02 00 00 00
e5d7e:   0f 05            sys
e5d80:   48 3d 01 f0 ff ff
...
e5dfa:   c3               retq
```



几乎和函数调用类似 Almost like a function call

- **转换控制 Transfer of control**
- **返回时执行下条指令 On return, executes next instruction**
- **使用调用规则传递参数 Passes arguments using calling convention**
- **返回值在%rax中 Gets result in `%rax`**

一个重要的差异 One Important exception!

- **由内核执行 Executed by Kernel**
- **不同的优先权 Different set of privileges**
- **以及其它不同：And other differences:**
  - **例如："函数"的"地址"是在%rax中 E.g., "address" of "function" is in `%rax`**
  - **使用错误号 Uses `errno`**
  - **等 Etc.**

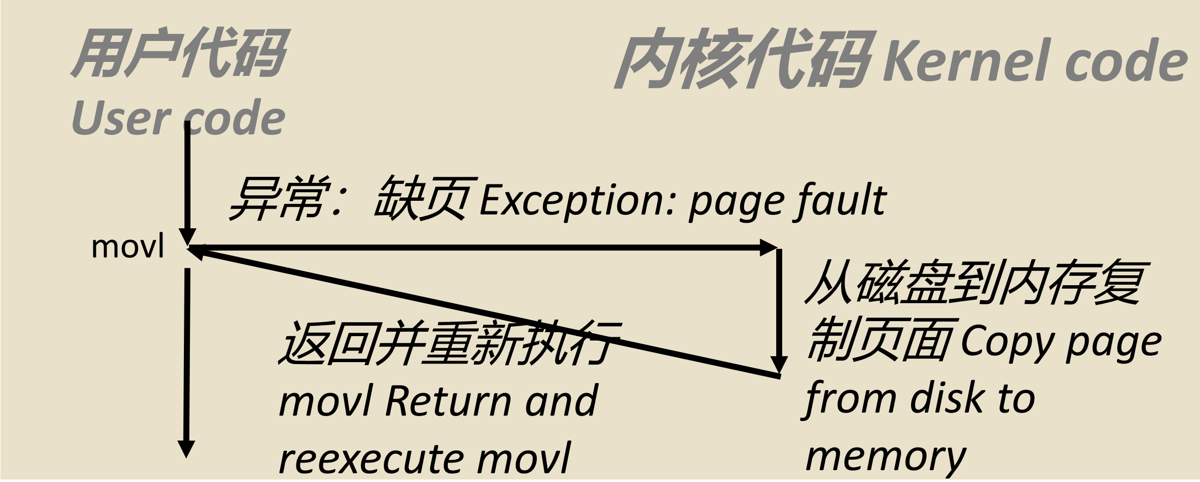
negative `errno`

故障举例：缺页异常 Fault Example: Page Fault

- 用户写内存 User writes to memory location
- 对应的页面在磁盘上 That portion (page) of user's memory is currently on disk

```
int a[1000];
main ()
{
    a[500] = 13;
}
```

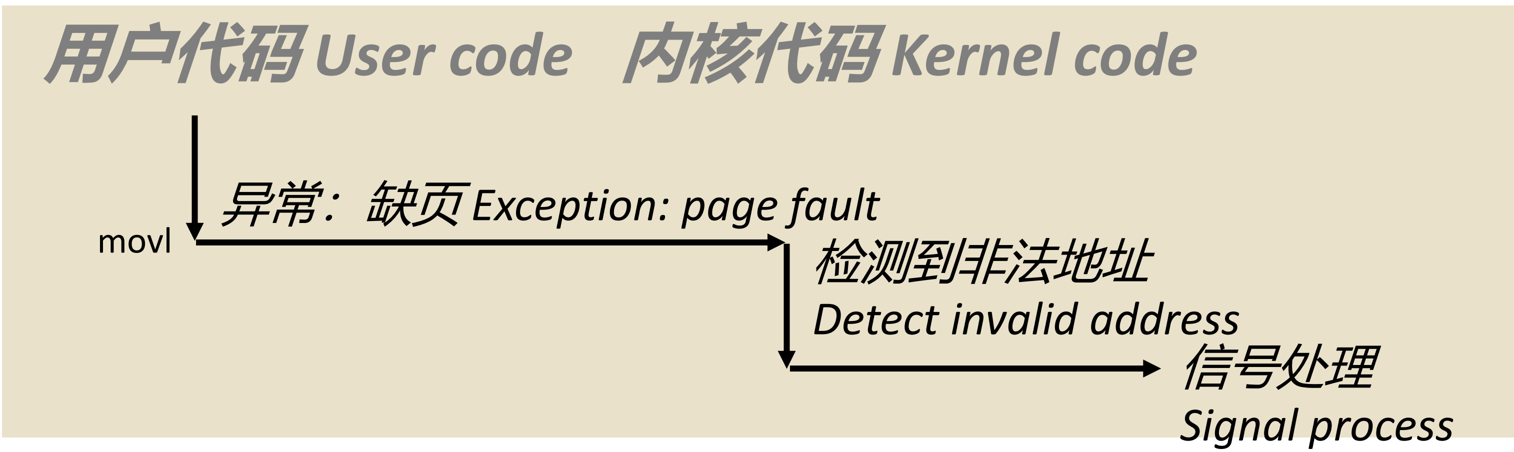
```
80483b7:        c7 05 10 9d 04 08 0d    movl    $0xd,0x8049d10
```



故障举例：非法内存引用
Fault Example: Invalid Memory Reference

```
int a[1000];
main ()
{
    a[5000] = 13;
}
```

```
80483b7:        c7 05 60 e3 04 08 0d    movl    $0xd,0x804e360
```



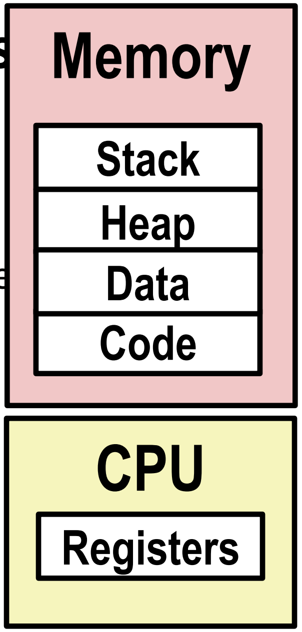
- 发送SIGSEGV信号给用户进程 Sends **SIGSEGV** signal to user process
- 用户进程会“段错误”异常退出 User process exits with “segmentation fault”

内容提纲

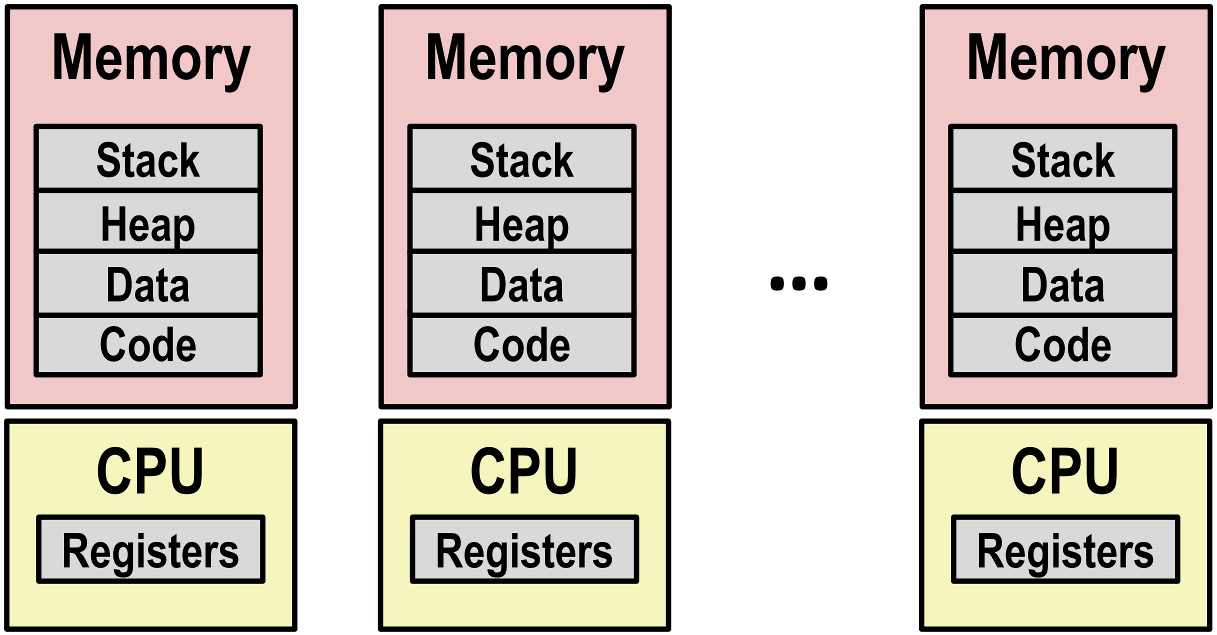
- 异常控制流 Exceptional Control Flow
- 异常 Exceptions
- 进程 Processes
- 进程控制 Process Control

进程 Processes

- **定义：进程是程序的一次执行(运行程序的实例)**
  Definition: A *process* is an instance of a running program.
  - 计算机科学最重要的概念之一 One of the most profound ideas in computer science
  - 与“程序”或“处理器”不同 Not the same as “program” or “processor”
- **进程为每个程序提供了两个关键抽象 Process provides each program with two key abstractions:**
  - ***逻辑控制流 Logical control flow***
    - 每个程序看起来独占CPU Each program seems to have exclusive use of the CPU
    - 内核支持的上下文切换 Provided by kernel mechanism called *context switching*
  - ***私有地址空间Private address space***
    - 每个程序看起来独占主存空间 Each program seems to have exclusive use of main memory.
    - 内核支持的虚拟内存 Provided by kernel mechanism called *virtual memory*



多进程幻象： Multiprocessing: The Illusion



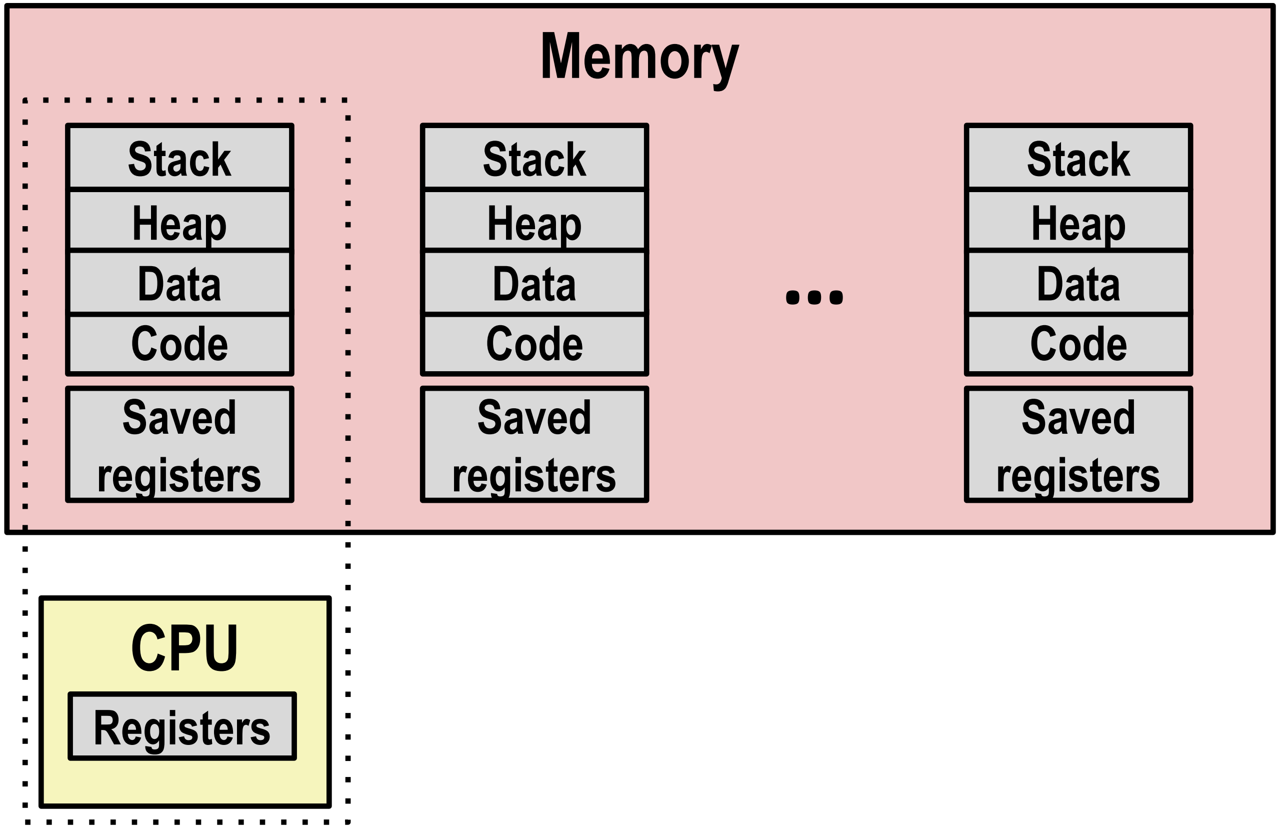
- **计算机同时运行很多进程 Computer runs many processes simultaneously**
  - 单个或多个用户的应用 Applications for one or more users
    - Web浏览器、邮件客户、编辑器。。。 Web browsers, email clients, editors, …
  - 后台任务 Background tasks
    - 监视网络和I/O设备 Monitoring network & I/O devices

多进程举例 Multiprocessing Example

```
xterm
Processes: 123 total, 5 running, 9 stuck, 109 sleeping, 611 threads          11:47:07
Load Avg: 1.03, 1.13, 1.14  CPU usage: 3.27% user, 5.15% sys, 91.56% idle
SharedLibs: 576K resident, 0B data, 0B linkedit.
MemRegions: 27958 total, 1127M resident, 35M private, 494M shared.
PhysMem: 1039M wired, 1974M active, 1062M inactive, 4076M used, 18M free.
VM: 280G vsize, 1091M framework vsize, 23075213(1) pageins, 5843367(0) pageouts.
Networks: packets: 41046228/11G in, 66083096/77G out.
Disks: 17874391/349G read, 12847373/594G written.

PID    COMMAND      %CPU TIME     #TH   #WQ  #PORT #MREG RPRVT  RSHRD  RSIZE  VPRVT  VSIZE
99217- Microsoft Of 0.0  02:28.34 4     1    202   418   21M    24M    21M    66M    763M
99051  usbmuxd      0.0  00:04.10 3     1    47    66    436K   216K   480K   60M    2422M
99006  iTunesHelper 0.0  00:01.23 2     1    55    78    728K   3124K  1124K  43M    2429M
84286  bash         0.0  00:00.11 1     0    20    24    224K   732K   484K   17M    2378M
84285  xterm        0.0  00:00.83 1     0    32    73    656K   872K   692K   9728K  2382M
55939- Microsoft Ex 0.3  21:58.97 10    3    360   954   16M    65M    46M    114M   1057M
54751  sleep        0.0  00:00.00 1     0    17    20    92K    212K   360K   9632K  2370M
54739  launchdadd   0.0  00:00.00 2     1    33    50    488K   220K   1736K  48M    2409M
54737  top          6.5  00:02.53 1/1   0    30    29    1416K  216K   2124K  17M    2378M
54719  automountd   0.0  00:00.02 7     1    53    64    860K   216K   2184K  53M    2413M
54701  ocspd        0.0  00:00.05 4     1    61    54    1268K  2644K  3132K  50M    2426M
54661  Grab         0.6  00:02.75 6     3    222+  389+  15M+   26M+   40M+   75M+   2556M+
54659  cookied      0.0  00:00.15 2     1    40    61    3316K  224K   4088K  42M    2411M
```

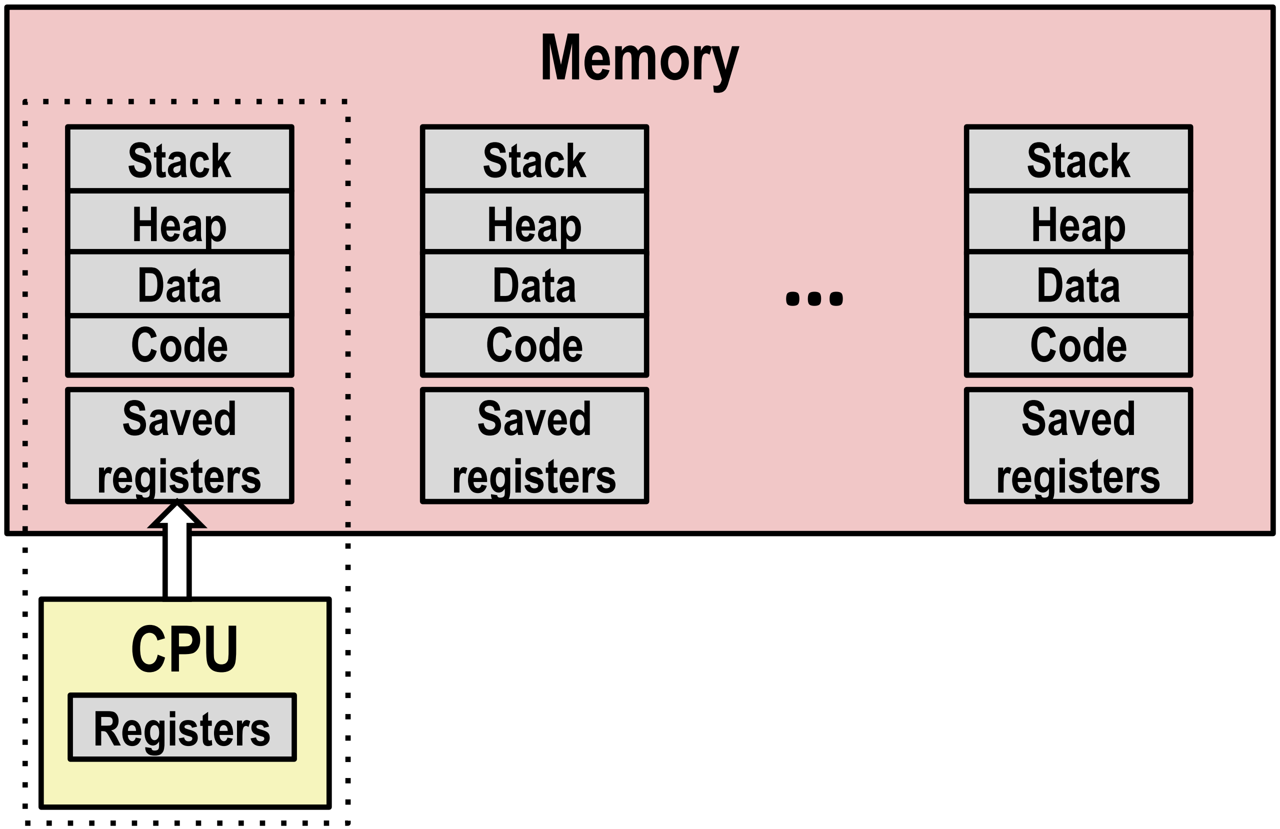
- 在Mac计算机上运行程序"top"命令 Running program "top" on Mac
  - 系统有123个进程，5个是活跃状态 System has 123 processes, 5 of which are active
  - 使用进程ID(PID)标识 Identified by Process ID (PID)

多进程的真像 Multiprocessing: The (Traditional) Reality



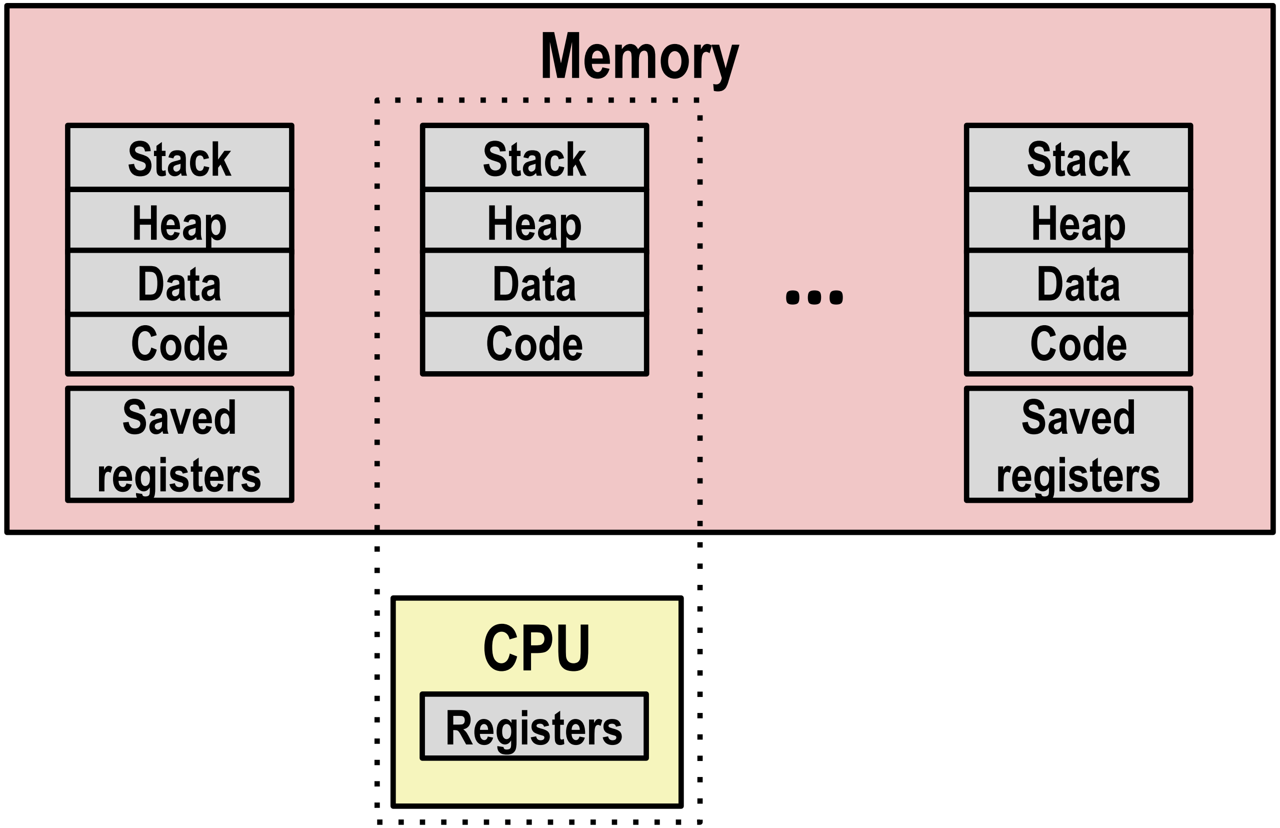
- **单个处理器并发执行多个进程 Single processor executes multiple processes concurrently**
  - 进程交替执行（多任务） Process executions interleaved (multitasking)
  - 地址空间由虚拟内存系统管理 Address spaces managed by virtual memory system (later in course)
  - 非激活进程的寄存器值存储在内存中 Register values for nonexecuting processes saved in memory

多进程真像 Multiprocessing: The (Traditional) Reality



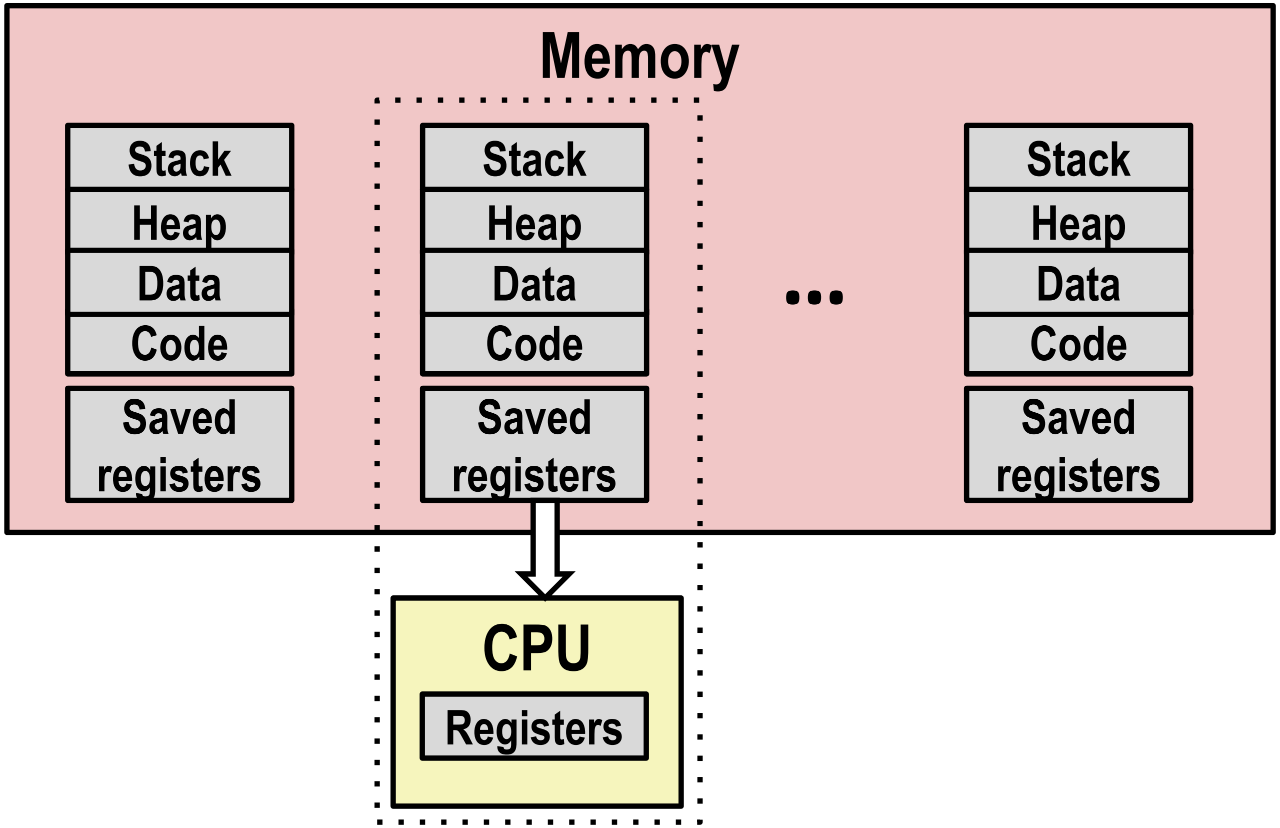
- 将当前寄存器存储在内存里 Save current registers in memory

多进程真像 Multiprocessing: The (Traditional) Reality



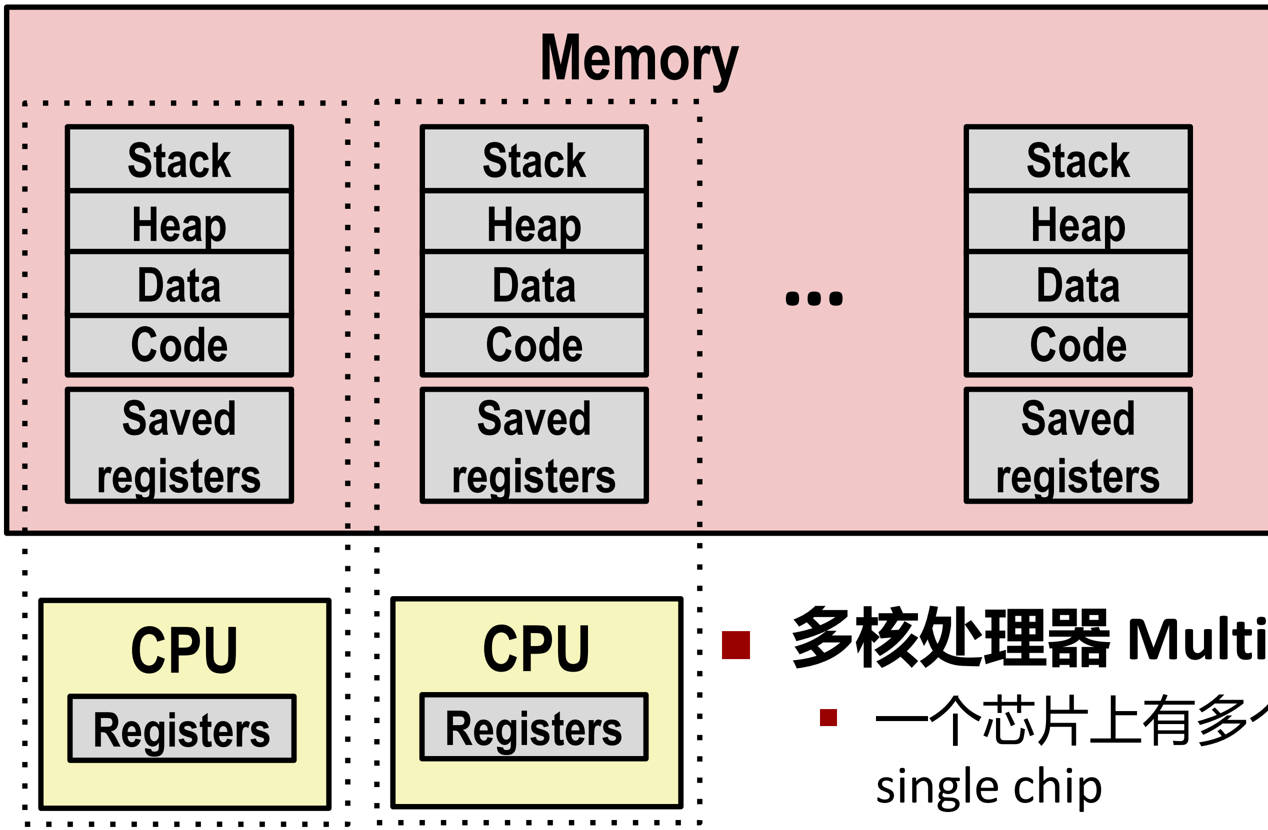
- 调度下一个进程执行 Schedule next process for execution

多进程真像 Multiprocessing: The (Traditional) Reality



- 加载保存的寄存器并切换地址空间（上下文切换）Load saved registers and switch address space (context switch)

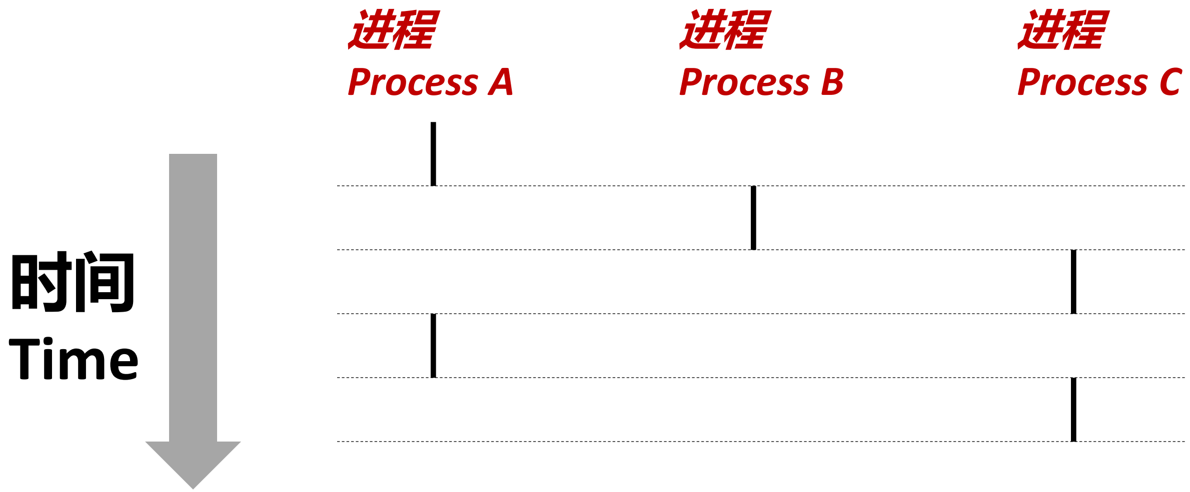
多进程的真像 Multiprocessing: The (Modern) Reality



- **多核处理器 Multicore processors**
  - 一个芯片上有多个CPU Multiple CPUs on single chip
  - 共享主存储器（以及部分cache） Share main memory (and some of the caches)
  - 每个可以执行一个独立进程 Each can execute a separate process
    - 由内核完成处理器到核心的调度 Scheduling of processors onto cores done by kernel

并发进程 Concurrent Processes

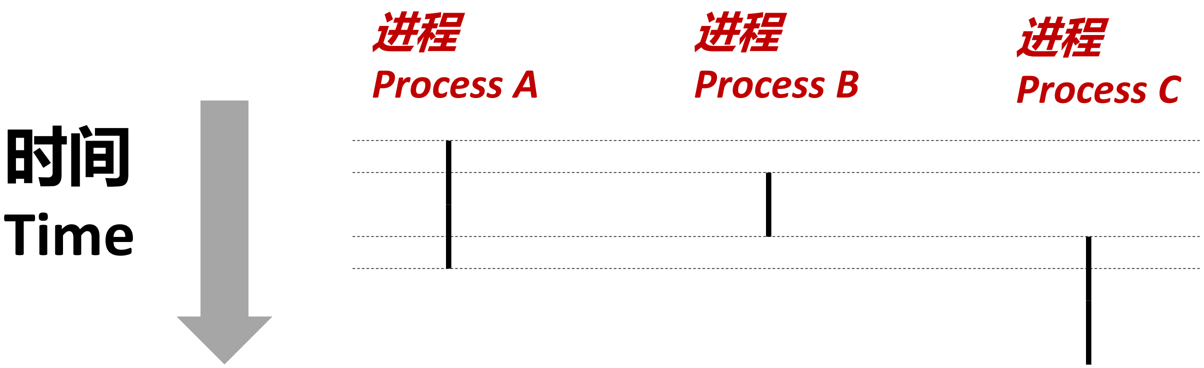
- 每个进程是一个逻辑控制流 Each process is a logical control flow.
- 两个进程并发运行如果在时间上重叠 Two processes *run concurrently* (*are concurrent)* if their flows overlap in time
- 否则是顺序执行 Otherwise, they are *sequential*
- 例如（运行在单核上） Examples (running on single core):
  - 并发：Concurrent: A & B, A & C
  - 顺序：Sequential: B & C



并发进程的用户视图
User View of Concurrent Processes

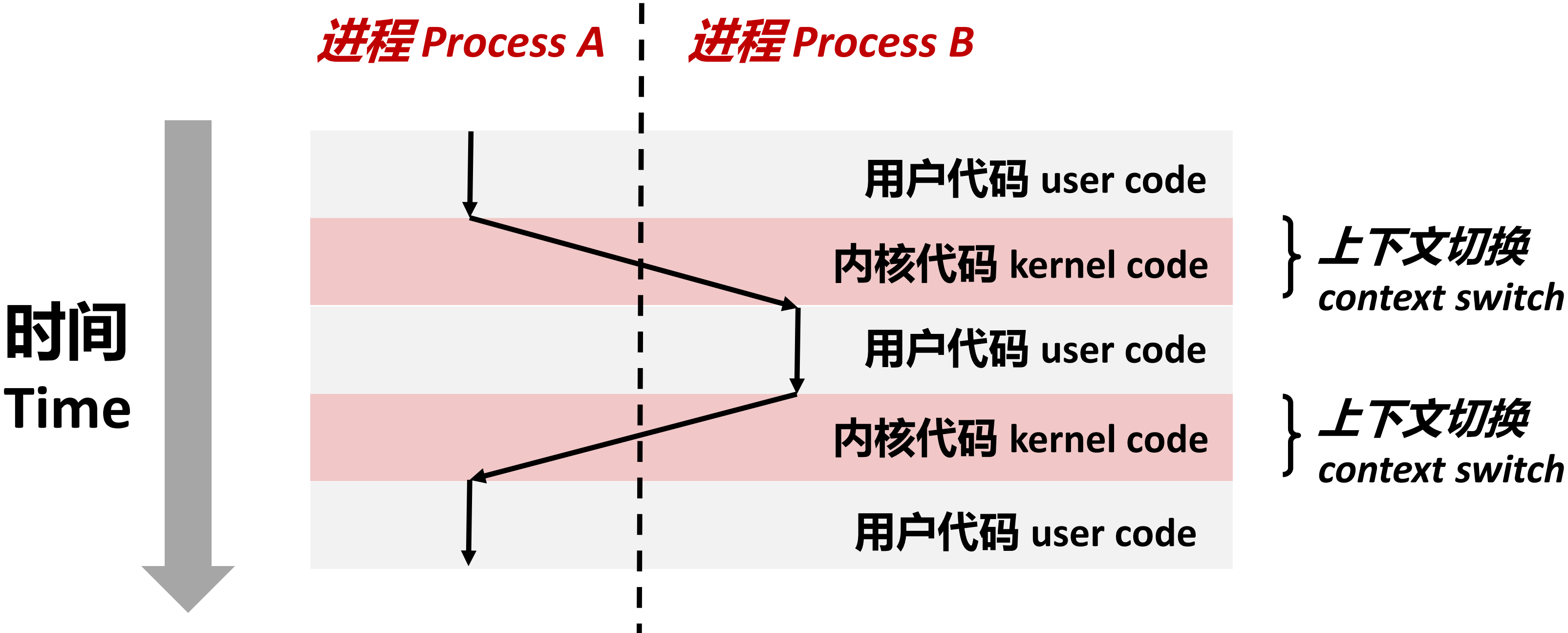
- **并发进程的控制流在时间上是物理上不相交的** Control flows for concurrent processes are physically disjoint in time

- **然而，我们可以将并发进程视为彼此并行运行** However, we can think of concurrent processes as running in parallel with each other



上下文切换 Context Switching

- **进程由称为内核的共享内存驻留操作系统代码块管理 Processes are managed by a shared chunk of memory-resident OS code called the *kernel***
  - 重点：内核不是一个独立的进程，而是作为某些现存进程的一部分运行 Important: the kernel is not a separate process, but rather runs as part of some existing process.
- **上下文切换使得控制流从一个进程切换到另一个进程 Control flow passes from one process to another via a *context switch***



内容提纲

- 异常控制流 Exceptional Control Flow
- 异常 Exceptions
- 进程 Processes
- **进程控制 Process Control**

系统功能调用错误处理 System Call Error Handling

- **出错时，Linux系统函数返回-1并通过全局变量errno设置错误编号指明原因 On error, Linux system-level functions typically return -1 and set global variable `errno` to indicate cause.**
- **硬性规定：Hard and fast rule:**
  - 你必须检查每个系统函数返回状态 You must check the return status of every system-level function
  - 返回值为void的函数除外 Only exception is the handful of functions that return `void`
- **例如 Example:**
- **#include <errno.h>**

```
if ((pid = fork()) < 0) {
    fprintf(stderr, "fork error: %s\n", strerror(errno));
    exit(0);
}
```

错误报告函数 Error-reporting functions

- 使用错误报告函数可以简化一些工作 Can simplify somewhat using an *error-reporting function*:

```
void unix_error(char *msg) /* Unix-style error */
{
   fprintf(stderr, "%s: %s\n", msg, strerror(errno));
   exit(0);
}
```

```
if ((pid = fork()) < 0)
   unix_error("fork error");
```

注意：退出时返回0
Note: csapp.c exits with 0.

- 但是，必须考虑应用。当出现问题时退出并不总是合适的 But, must think about application. Not alway appropriate to exit when something goes wrong.

错误处理包装器 Error-handling Wrappers

- **通过使用Stevens[1]风格的错误处理包装器，我们进一步简化了向您展示的代码： We simplify the code we present to you even further by using Stevens-style error-handling wrappers:**

```
pid_t Fork(void)
{
  pid_t pid;

  if ((pid = fork()) < 0)
    unix_error("Fork error");
  return pid;
}
```

```
pid = Fork();
```

- **而不是您在实际应用程序中通常要做这件事情 NOT what you generally want to do in a real application**

[1]例如，在“Unix网络编程：套接字网络API” [1]e.g., in “UNIX Network Programming: The sockets networking API“ W. Richard Stevens

获得进程PID Obtaining Process IDs

- `pid_t getpid(void)`
  - 返回当前进程的PID Returns PID of current process

- `pid_t getppid(void)`
  - 返回父进程的PID Returns PID of parent process

创建和终止进程
Creating and Terminating Processes

从程序员的角度，可以认为一个进程处于3种状态之一 From a programmer's perspective, we can think of a process as being in one of three states

- **运行 Running**
  - 进程或者正在执行，或者等待被执行并最终由内核调度（即被选择执行） Process is either executing, or waiting to be executed and will eventually be *scheduled* (i.e., chosen to execute) by the kernel
- **停止 Stopped**
  - 进程执行被挂起，直到被触发重新调度执行 Process execution is *suspended* and will not be scheduled until further notice (next lecture when we study signals)
- **终止 Terminated**
  - 进程永远停止运行 Process is stopped permanently

进程终止 Terminating Processes

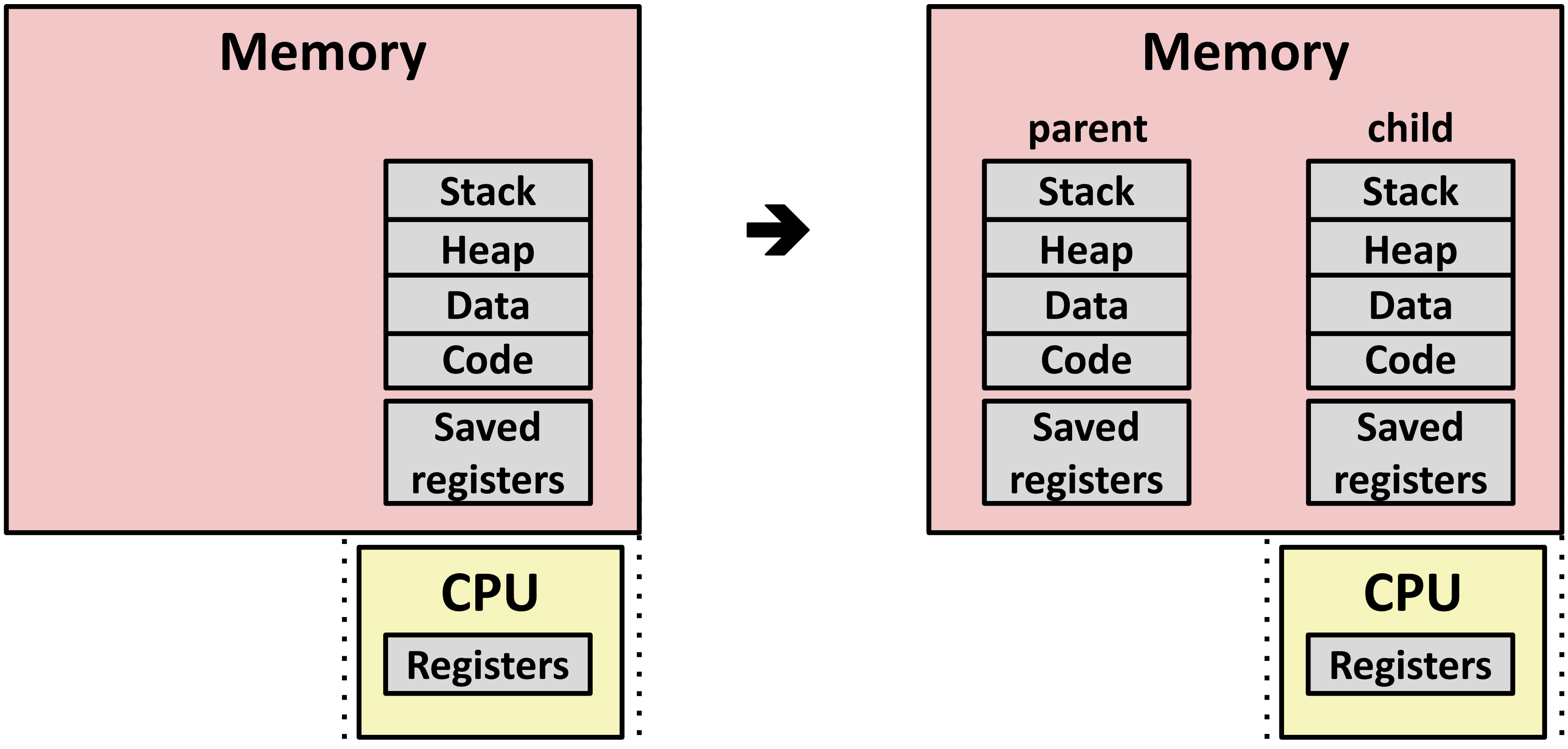
- **进程由于以下三个原因之一终止 Process becomes terminated for one of three reasons:**
  - 收到默认动作是终止的信号 Receiving a signal whose default action is to terminate (next lecture)
  - 从main函数返回 Returning from the `main` routine
  - 调用exit函数 Calling the `exit` function
- **`void exit(int status)`**
  - 终止退出状态为status Terminates with an *exit status* of `status`
  - 规则：正常返回状态为0，出错为非0 Convention: normal return status is 0, nonzero on error
  - 另一种显式设置退出状态的方式是从main函数返回一个整数值 Another way to explicitly set the exit status is to return an integer value from the main routine
- **`exit`调用一次，但从不返回 `exit` is called once but never returns.**

创建进程 Creating Processes

- ***父进程通过调用fork创建一个新的运行子进程 Parent process*** **creates a new running *child process* by calling `fork`**
- **`int fork(void)`**
  - 返回0给子进程，子进程的PID给父进程 Returns 0 to the child process, child's PID to parent process
  - 子进程和父进程几乎是一样的 Child is *almost* identical to parent:
    - 子进程获得与父进程的虚拟地址空间同样的拷贝（但是是分开的）Child get an identical (but separate) copy of the parent's virtual address space.
    - 子进程获得与父进程打开文件描述符同样的拷贝 Child gets identical copies of the parent's open file descriptors
    - 子进程与父进程有不同的PID Child has a different PID than the parent

- **`fork`很有意思（通常也令人费解），因为它调用一次，但返回两次 `fork` is interesting (and often confusing) because it is called *once* but returns *twice***

fork的概念视图 Conceptual View of `fork`



- **做完全的执行状态拷贝 Make complete copy of execution state**
  - 指定一个为父进程一个为子进程 Designate one as parent and one as child
  - 恢复父进程或子进程的执行 Resume execution of parent or child

重新审视fork函数
The `fork` Function Revisited

- **虚拟存储器和内存映射解释了fork如何为每个进程提供私有的虚拟地址空间 VM and memory mapping explain how `fork` provides private address space for each process.**
- **为了给新进程创建虚拟地址 To create virtual address for new process:**
  - 创建与当前**`mm_struct`**、**`vm_area_struct`**和页表精确一致的拷贝 Create exact copies of current **`mm_struct`**, **`vm_area_struct`**, and page tables.
  - 设置两个进程对每个页具有只读权限 Flag each page in both processes as read-only
  - 设置两个进程对每个**`vm_area_struct`**都是私有COW Flag each **`vm_area_struct`** in both processes as private COW
- **返回时，每个进程具有精确的虚拟内存拷贝 On return, each process has exact copy of virtual memory.**
- **后续的写操作使用COW机制创建新页面 Subsequent writes create new pages using COW mechanism.**

fork举例 fork Example

```
int main(int argc, char** argv)
{
    pid_t pid;
    int x = 1;

    pid = Fork();
    if (pid == 0) {   /* Child */
        printf("child : x=%d\n", ++x);
        return 0;
    }

    /* Parent */
    printf("parent: x=%d\n", --x);
    return 0;
}
```

fork.c

- 调用一次，返回两次 Call once, return twice
- 并发执行 Concurrent execution
  - 不能预测父进程和子进程的执行顺序 Can't predict execution order of parent and child

```
linux> ./fork
parent: x=0
child : x=2
```

```
linux> ./fork
child : x=2
parent: x=0
```

```
linux> ./fork
parent: x=0
child : x=2
```

```
linux> ./fork
parent: x=0
child : x=2
```

fork举例 fork Example

```
int main(int argc, char** argv)
{
    pid_t pid;
    int x = 1;

    pid = Fork();
    if (pid == 0) {  /* Child */
        printf("child : x=%d\n", ++x);
        return 0;
    }

    /* Parent */
    printf("parent: x=%d\n", --x);
    return 0;
}
```

```
linux> ./fork
parent: x=0
child : x=2
```

- 调用一次，返回两次 Call once, return twice
- 并发执行 Concurrent execution
  - 不能预测父进程和子进程的执行顺序 Can't predict execution order of parent and child
- 重复但是分开的地址空间 Duplicate but separate address space
  - x的值为1，当fork在父子进程返回 x has a value of 1 when fork returns in parent and child
  - 后续对x的改变是独立的 Subsequent changes to x are independent
- 共享打开的文件 Shared open files
  - 标准输出对父子进程是相同的 stdout is the same in both parent and child

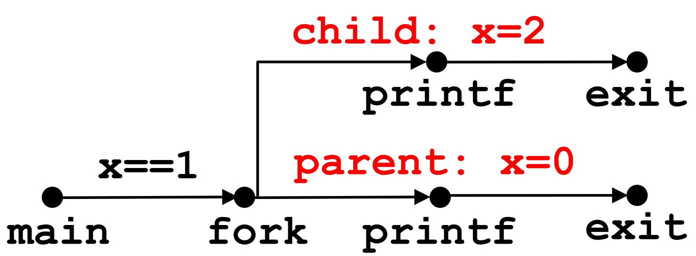
使用进程图描述fork
Modeling `fork` with Process Graphs

- ***进程图*是一个有用的工具，它可以捕获并发程序中语句的偏序 A *process graph* is a useful tool for capturing the partial ordering of statements in a concurrent program:**
  - 每个顶点都是语句的执行 Each vertex is the execution of a statement
  - a->b表示a发生在b之前 a -> b means `a` happens before b
  - 可以用变量的当前值标记边 Edges can be labeled with current value of variables
  - 可以用输出标记printf顶点 `printf` vertices can be labeled with output
  - 每个图都以一个没有输入边的顶点开始 Each graph begins with a vertex with no inedges
- **进程图的任何*拓扑排序*都对应于一种可行的全排序 Any *topological sort* of the graph corresponds to a feasible total ordering.**
  - 所有边从左向右指向的顶点的全排序 Total ordering of vertices where all edges point from left to right

进程图举例 Process Graph Example

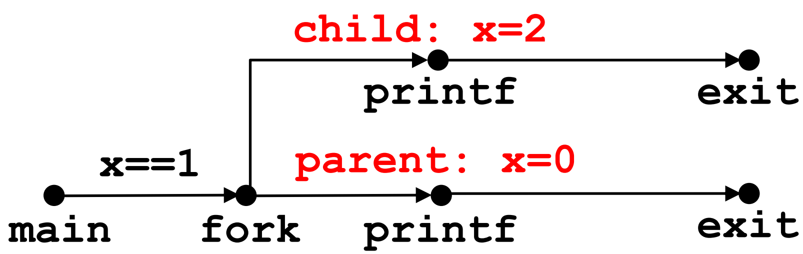
```
int main()
{
    pid_t pid;
    int x = 1;

    pid = Fork();
    if (pid == 0) {  /* Child */
        printf("child : x=%d\n", ++x);
               exit(0);
    }

    /* Parent */
    printf("parent: x=%d\n", --x);
    exit(0);
}
```

fork.c

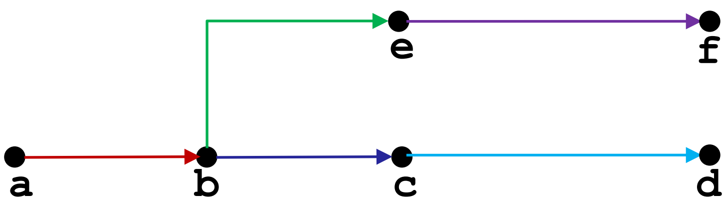


解释进程图 Interpreting Process Graphs

- 原始图 Original graph:

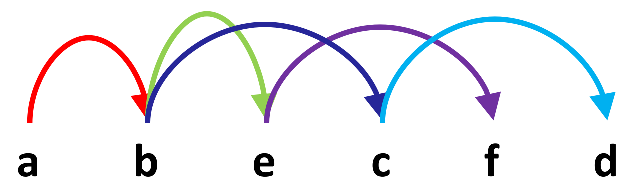


- 重新标记的图 Relabled graph:



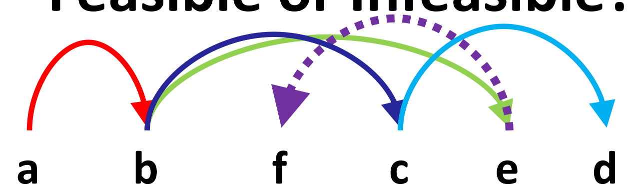
可行的全排序

Feasible total ordering:



可行还是不可行?

Feasible or Infeasible?



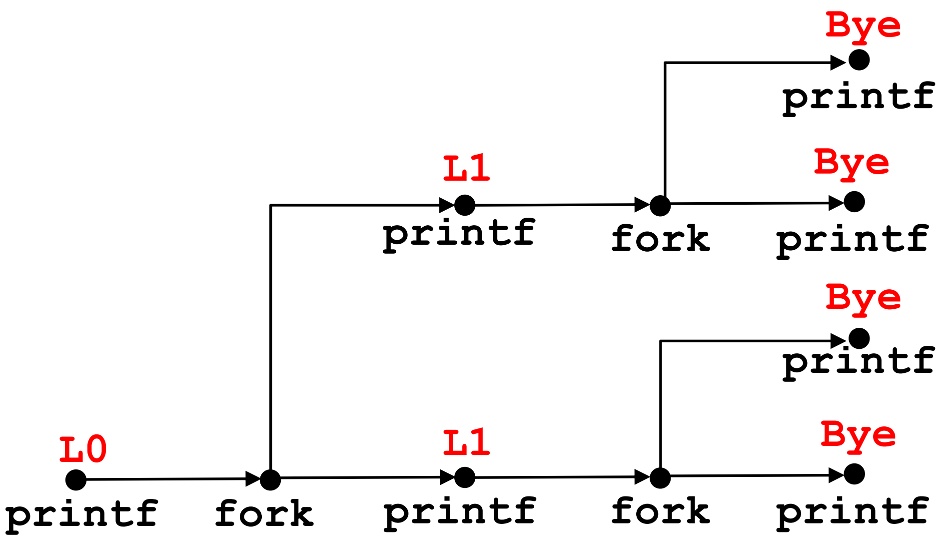
不可行：不是一种拓扑排序

Infeasible: not a topological sort

fork举例：两个连续的fork
fork Example: Two consecutive forks

```
void fork2()
{
    printf("L0\n");
    fork();
    printf("L1\n");
    fork();
    printf("Bye\n");
}
```

forks.c



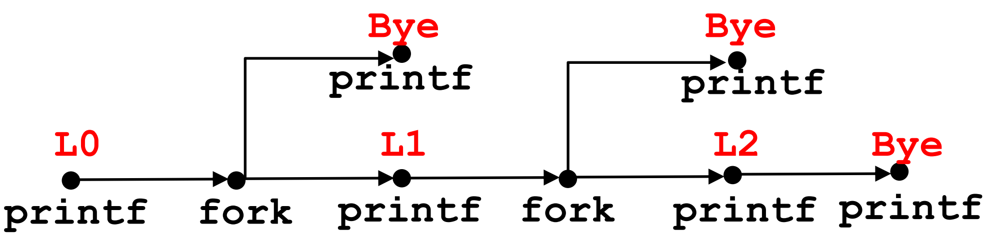
可能的输出
Feasible output:
L0
L1
Bye
Bye
L1
Bye
Bye

不可能的输出
Infeasible output:
L0
Bye
L1
Bye
L1
Bye
Bye

fork举例：父类进程中的嵌套forks
fork Example: Nested forks in parent

```
void fork4()
{
    printf("L0\n");
    if (fork() != 0) {
        printf("L1\n");
        if (fork() != 0) {
            printf("L2\n");
        }
    }
    printf("Bye\n");
}
```

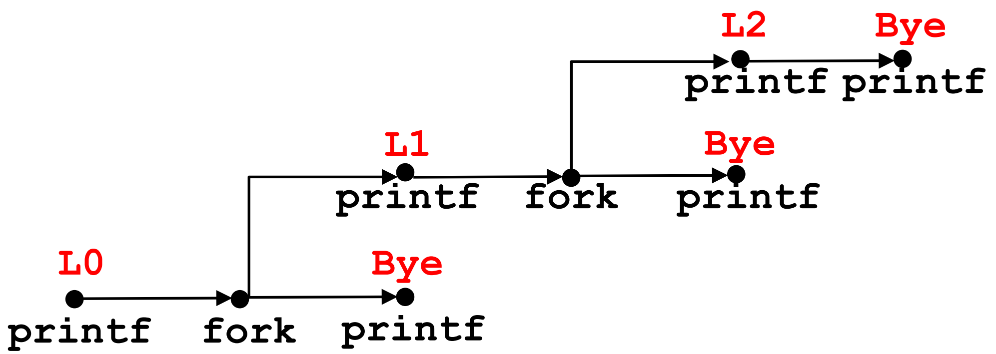
forks.c



可能的输出 Feasible output:	不可能的输出 Infeasible output:
L0	L0
L1	Bye
Bye	L1
Bye	Bye
L2	Bye
Bye	L2

fork举例：子进程中的嵌套forks
fork Example: Nested forks in children

```
void fork5()
{
    printf("L0\n");
    if (fork() == 0) {
        printf("L1\n");
        if (fork() == 0) {
            printf("L2\n");
        }
    }
    printf("Bye\n");
}
```
forks.c



可能的输出
Feasible output:
L0
Bye
L1
L2
Bye
Bye

不可能的输出
Infeasible output:
L0
Bye
L1
Bye
Bye
L2

回收子进程 Reaping Child Processes

- **思想 Idea**
  - 进程终止后仍然消耗系统资源 When process terminates, it still consumes system resources
    - 例如：退出状态，各种OS表格 Examples: Exit status, various OS tables
  - 称为“僵尸”进程 Called a “zombie”
    - 活着的尸体，半生半死 Living corpse, half alive and half dead
- **回收 Reaping**
  - 父类进程对终止的子进程操作（使用wait或waitpid） Performed by parent on terminated child (using `wait` or `waitpid`)
  - 父类进程持有退出状态信息 Parent is given exit status information
  - 内核随后删掉僵尸子进程 Kernel then deletes zombie child process

回收子进程 Reaping Child Processes

- **如果父类进程没有回收会怎么样？What if parent doesn't reap?**
  - 如果任何父类进程终止没有回收子进程，则该孤儿子进程由init进程（pid==1）回收 If any parent terminates without reaping a child, then the orphaned child will be reaped by **`init`** process (pid == 1)
    - 除非ppid==1，此时需要重启 Unless ppid == 1! Then need to reboot…
  - 所以只需要显式回收长时间运行的进程 So, only need explicit reaping in long-running processes
    - 例如外壳程序和服务器程序 e.g., shells and servers

僵尸举例 Zombie Example

```
void fork7() {
    if (fork() == 0) {
        /* Child */
        printf("Terminating Child, PID = %d\n", getpid());
        exit(0);
    } else {
        printf("Running Parent, PID = %d\n", getpid());
        while (1)
            ; /* Infinite loop */
    }
}
```

```
linux> ./forks 7 &
[1] 6639
Running Parent, PID = 6639
Terminating Child, PID = 6640
linux> ps
  PID TTY          TIME CMD
 6585 ttyp9    00:00:00 tcsh
 6639 ttyp9    00:00:03 forks
 6640 ttyp9    00:00:00 forks <defunct>
 6641 ttyp9    00:00:00 ps
linux> kill 6639
[1]    Terminated
linux> ps
  PID TTY          TIME CMD
 6585 ttyp9    00:00:00 tcsh
 6642 ttyp9    00:00:00 ps
```

- **ps显示子进程为“defunct”（即僵尸）** **ps** shows child process as “defunct” (i.e., a zombie)
- 杀死父进程允许子进程由init进程回收 Killing parent allows child to be reaped by **init**

非终止子进程举例

Non-terminating Child Example

```
void fork8()
{
    if (fork() == 0) {
        /* Child */
        printf("Running Child, PID = %d\n",
               getpid());
        while (1)
            ; /* Infinite loop */
    } else {
        printf("Terminating Parent, PID = %d\n",
               getpid());
        exit(0);
    }
}
```

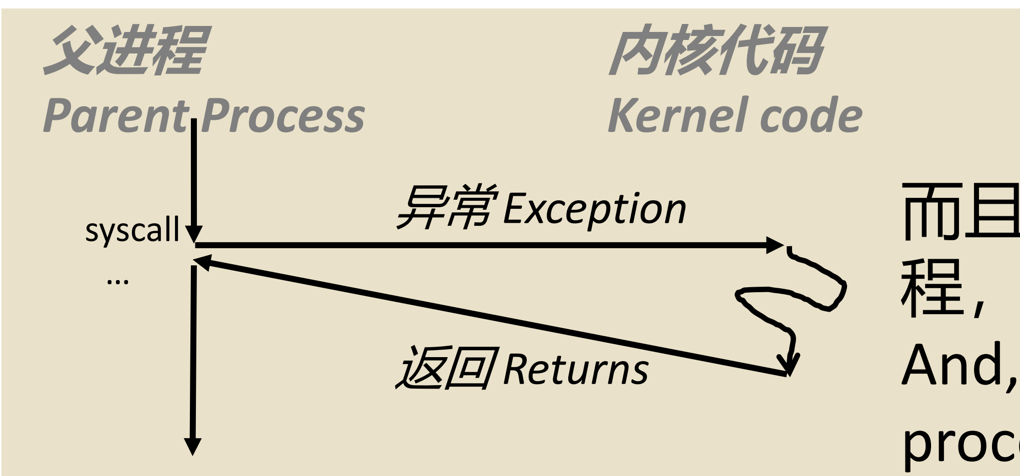
```
linux> ./forks 8
Terminating Parent, PID = 6675
Running Child, PID = 6676
linux> ps
  PID TTY          TIME CMD
 6585 ttyp9    00:00:00 tcsh
 6676 ttyp9    00:00:06 forks
 6677 ttyp9    00:00:00 ps
linux> kill 6676
linux> ps
  PID TTY          TIME CMD
 6585 ttyp9    00:00:00 tcsh
 6678 ttyp9    00:00:00 ps
```

- 子进程仍然活着，尽管父进程已经终止 Child process still active even though parent has terminated
- 必须显式杀死子进程，否则子进程将会永远一直在运行 Must kill child explicitly, or else will keep running indefinitely

wait：与子进程同步
wait: Synchronizing with Children

- **父进程通过调用wait函数回收子进程Parent reaps a child by calling the `wait` function**

- **`int wait(int *child_status)`**
  - 挂起当前进程直到其子进程之一终止 Suspends current process until one of its children terminates
  - 用syscall实现 Implemented as syscall



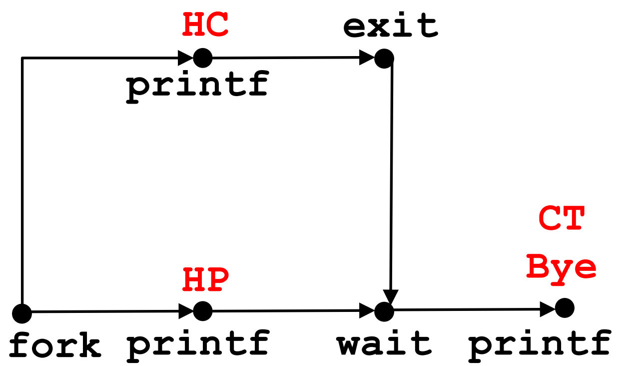
而且，潜在地其它用户进程，包括父进程的子进程
And, potentially other user processes, including a child of parent

`wait`：与子进程同步
`wait`: Synchronizing with Children

- **父进程通过调用wait函数回收子进程 Parent reaps a child by calling the `wait` function**
- **`int wait(int *child_status)`**
  - 挂起当前进程直到其子进程之一终止 Suspends current process until one of its children terminates
  - 返回值是终止子进程的PID Return value is the **`pid`** of the child process that terminated
  - 如果**`child_status`**不为空，那么它指向的整数将会设置为一个值，以指示子进程终止的原因和退出状态： If **`child_status != NULL`**, then the integer it points to will be set to a value that indicates reason the child terminated and the exit status:
    - 使用wait.h中宏定义进行检查 Checked using macros defined in `wait.h`
      - `WIFEXITED`, `WEXITSTATUS`, `WIFSIGNALED`, `WTERMSIG`, `WIFSTOPPED`, `WSTOPSIG`, `WIFCONTINUED`
      - 参见教材了解详情 See textbook for details

wait：与子进程同步
wait: Synchronizing with Children

```
void fork9() {
    int child_status;

    if (fork() == 0) {
        printf("HC: hello from child\n");
                exit(0);
    } else {
        printf("HP: hello from parent\n");
        wait(&child_status);
        printf("CT: child has terminated\n");
    }
    printf("Bye\n");
}
```
forks.c



可能的输出
Feasible output(s):
HC
HP
CT
Bye

不可能的输出
Infeasible output:
HP
CT
Bye
HC

另一个wait的例子
Another wait Example

- 如果多个子进程终止，将会以任意顺序进行 If multiple children completed, will take in arbitrary order
- 可以使用宏WIFEXITED和WEXITSTATUS获取有关退出状态的信息 Can use macros WIFEXITED and WEXITSTATUS to get information about exit status

```
void fork10() {
    pid_t pid[N];
    int i, child_status;

    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0) {
            exit(100+i); /* Child */
        }
    for (i = 0; i < N; i++) { /* Parent */
        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 terminate abnormally\n", wpid);
    }
}
```

forks.c

waitpid：等待特定进程

waitpid: Waiting for a Specific Process

- `pid_t waitpid(pid_t pid, int &status, int options)`
  - 挂起当前进程直到指定进程终止 Suspends current process until specific process terminates
  - 各种选项（参见教材） Various options (see textbook)

```
void fork11() {
    pid_t pid[N];
    int i;
    int child_status;

    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0)
            exit(100+i); /* Child */
    for (i = N-1; i >= 0; i--) {
        pid_t wpid = waitpid(pid[i], &child_status, 0);
        if (WIFEXITED(child_status))
            printf("Child %d terminated with exit status %d\n",
                   wpid, WEXITSTATUS(child_status));
        else
            printf("Child %d terminate abnormally\n", wpid);
    }
}
```

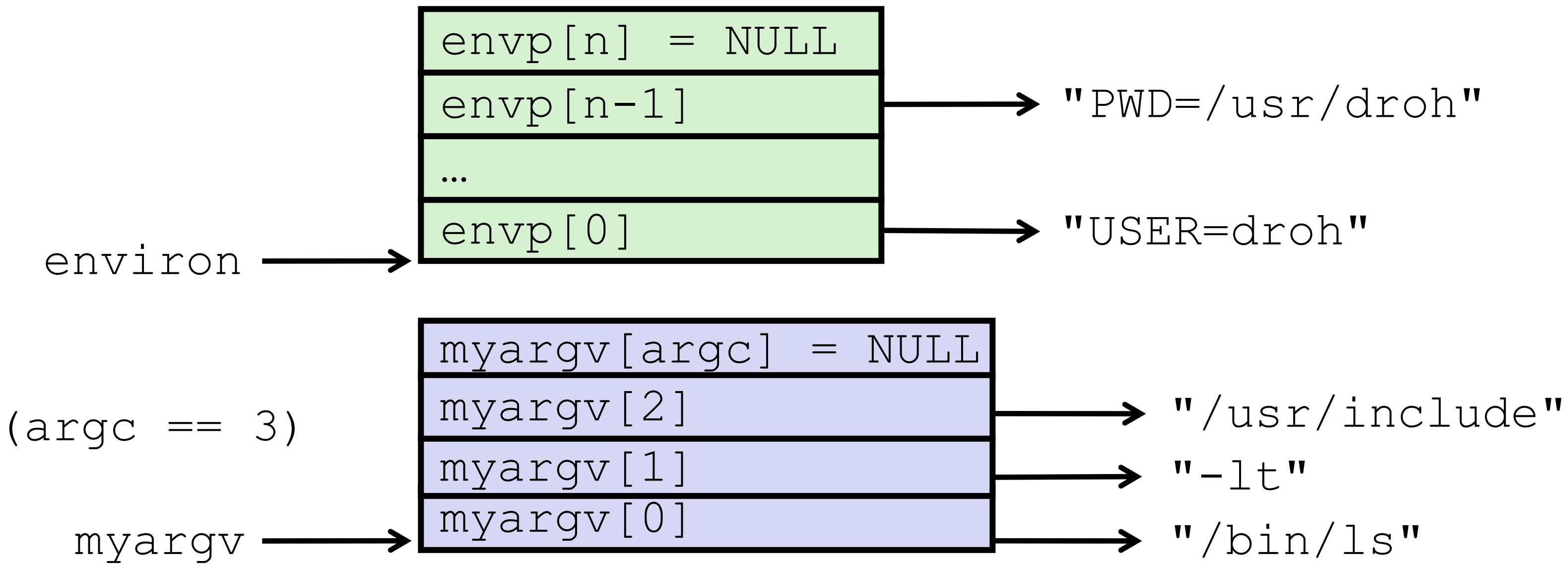
forks.c

execve：加载运行程序
execve：Loading and Running Programs

- `int execve(char *filename, char *argv[], char *envp[])`
- **在当前进程加载和运行 Loads and runs in the current process:**
  - 可执行文件文件名filename   Executable file **`filename`**
    - 目标代码文件或者以"#! 解释器"开始的脚本文件 Can be object file or script file beginning with `#!interpreter` (e.g., `#!/bin/bash`)
  - …带有参数列表argv …with argument list **`argv`**
    - 按照约定第一个参数为文件名 By convention **`argv[0]==filename`**
  - …带有环境变量列表envp   …and environment variable list **`envp`**
    - "名字=值"串 "name=value" strings (e.g., `USER=droh`)
    - `getenv, putenv, printenv`
- **覆盖代码、数据和堆栈 Overwrites code, data, and stack**
  - 保持PID、打开文件和信号上下文 Retains PID, open files and signal context
- **调用一次而且从不返回 Called once and never returns**
  - …除非如果有错误 …except if there is an error

Execve举例 execve Example

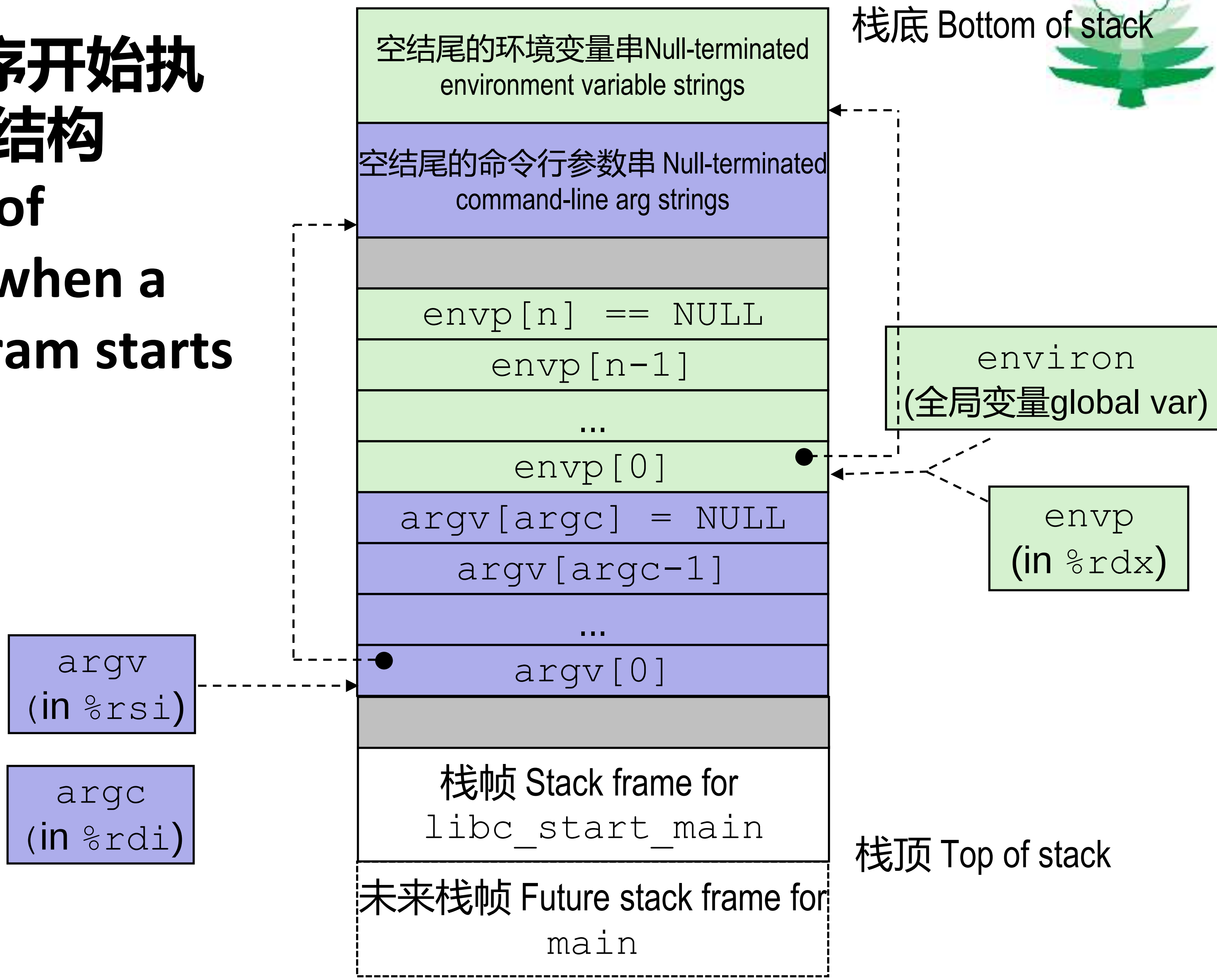
- **使用当前环境在子进程中执行 Execute** `"/bin/ls -lt /usr/include"` **in child process using current environment:**



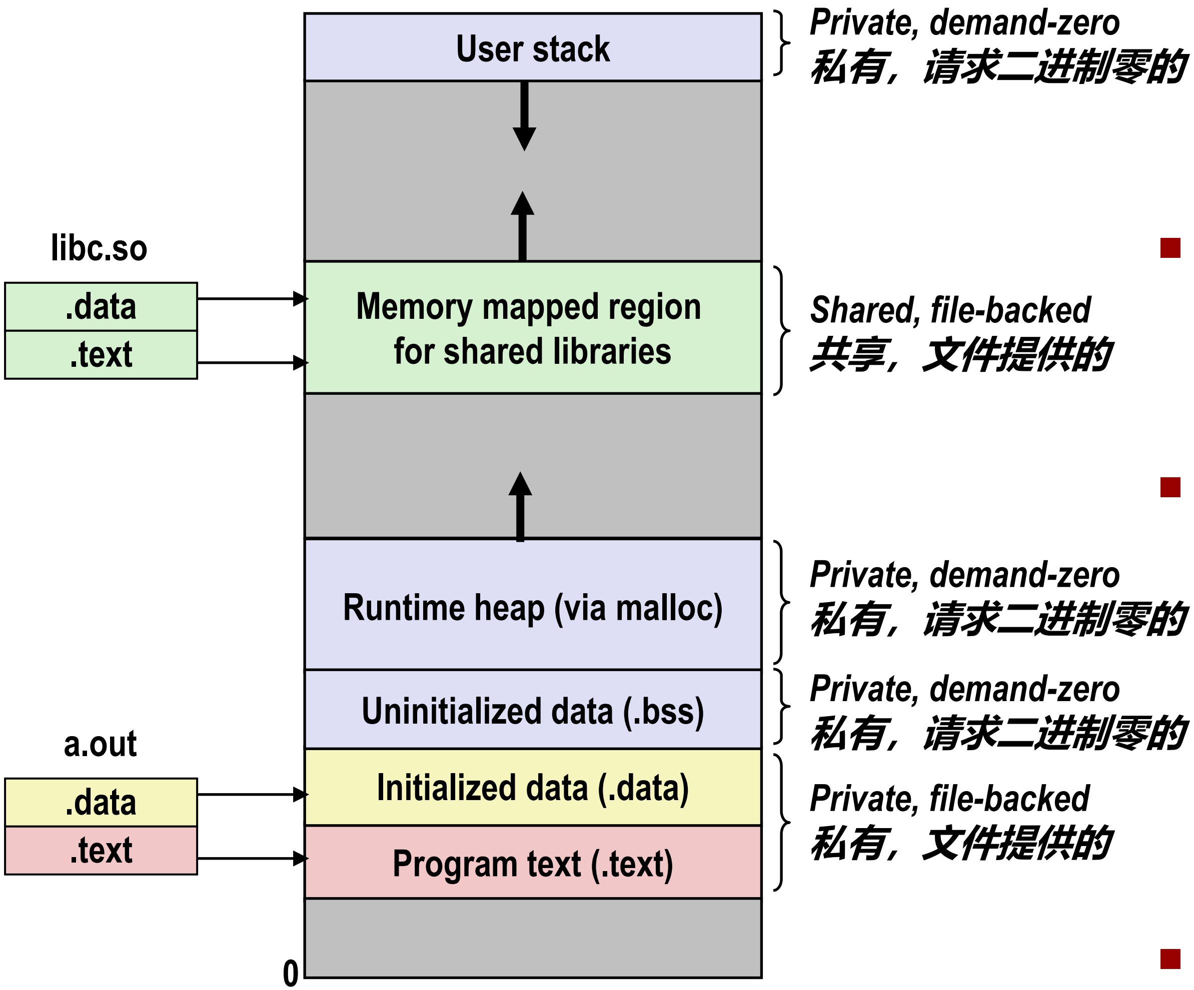

```c
if ((pid = Fork()) == 0) {   /* Child runs program */
    if (execve(myargv[0], myargv, environ) < 0) {
        printf("%s: Command not found.\n", myargv[0]);
        exit(1);
    }
}
```

一个新程序开始执行时的栈结构

Structure of the stack when a new program starts



重新审视execve函数
The execve Function Revisited



- **要使用execve在当前进程加载和运行一个新程序a.out To load and run a new program `a.out` in the current process using `execve`:**
- **释放老区域的`vm_area_struct`和页表 Free `vm_area_struct`'s and page tables for old areas**
- **为新区域创建`vm_area_struct`和页表 Create `vm_area_struct`'s and page tables for new areas**
  - 程序和初始化后的数据由目标文件提供 Programs and initialized data backed by object files.
  - `.bss`和栈由匿名文件提供 `.bss` and stack backed by anonymous files.
- **设置PC为.text中的入口点 Set PC to entry point in `.text`**
  - Linux将陷入需要的代码和数据页 Linux will fault in code and data pages as needed.

总结 Summary

- **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）产生 Generated externally (interrupts) or internally (traps and faults)

- **进程 Processes**
  - 任意时刻，系统有多个活动进程 At any given time, system has multiple active processes
  - 尽管在单核上每个时刻只能执行一个进程 Only one can execute at a time on a single core, though
  - 每个进程看起来独占处理器和私有内存空间 Each process appears to have total control of processor + private memory space

总结（续） Summary (cont.)

- **生成新进程 Spawning processes**
  - 调用fork Call `fork`
  - 一次调用，两次返回 One call, two returns
- **结束进程 Process completion**
  - 调用exit Call `exit`
  - 一次调用，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`
- **加载运行程序 Loading and running programs**
  - 调用execve（或变种） Call `execve` (or variant)
  - 一次调用，（正常）不返回 One call, (normally) no return

使fork更不确定
Making `fork` More Nondeterministic

- 问题 Problem
  - Linux调度器不会产生很多运行间差异 Linux scheduler does not create much run-to-run variance
  - 在非确定性程序中隐藏潜在的竞争条件 Hides potential race conditions in nondeterministic programs
    - 例如，fork是先返回到子进程，还是返回到父进程？ E.g., does `fork` return to child first, or to parent?
- 解决方案 Solution
  - 创建库例程的自定义版本，沿不同分支插入随机延迟 Create custom version of library routine that inserts random delays along different branches
    - 例如， fork父进程和子进程 E.g., for parent and child in `fork`
  - 使用运行时库打桩使程序使用特殊版本的库代码 Use runtime interpositioning to have program use special version of library code

延迟变化的fork Variable delay fork

```
/* fork wrapper function */
pid_t fork(void) {
    initialize();
    int parent_delay = choose_delay();
    int child_delay = choose_delay();
    pid_t parent_pid = getpid();
    pid_t child_pid_or_zero = real_fork();
    if (child_pid_or_zero > 0) {
        /* Parent */
        if (verbose) {
            printf(
"Fork.  Child pid=%d, delay = %dms.  Parent pid=%d, delay = %dms\n",
                   child_pid_or_zero, child_delay,
                   parent_pid, parent_delay);
            fflush(stdout);
        }
        ms_sleep(parent_delay);
    } else {
        /* Child */
        ms_sleep(child_delay);
    }
    return child_pid_or_zero;
}
```

myfork.c

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 颜珂

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

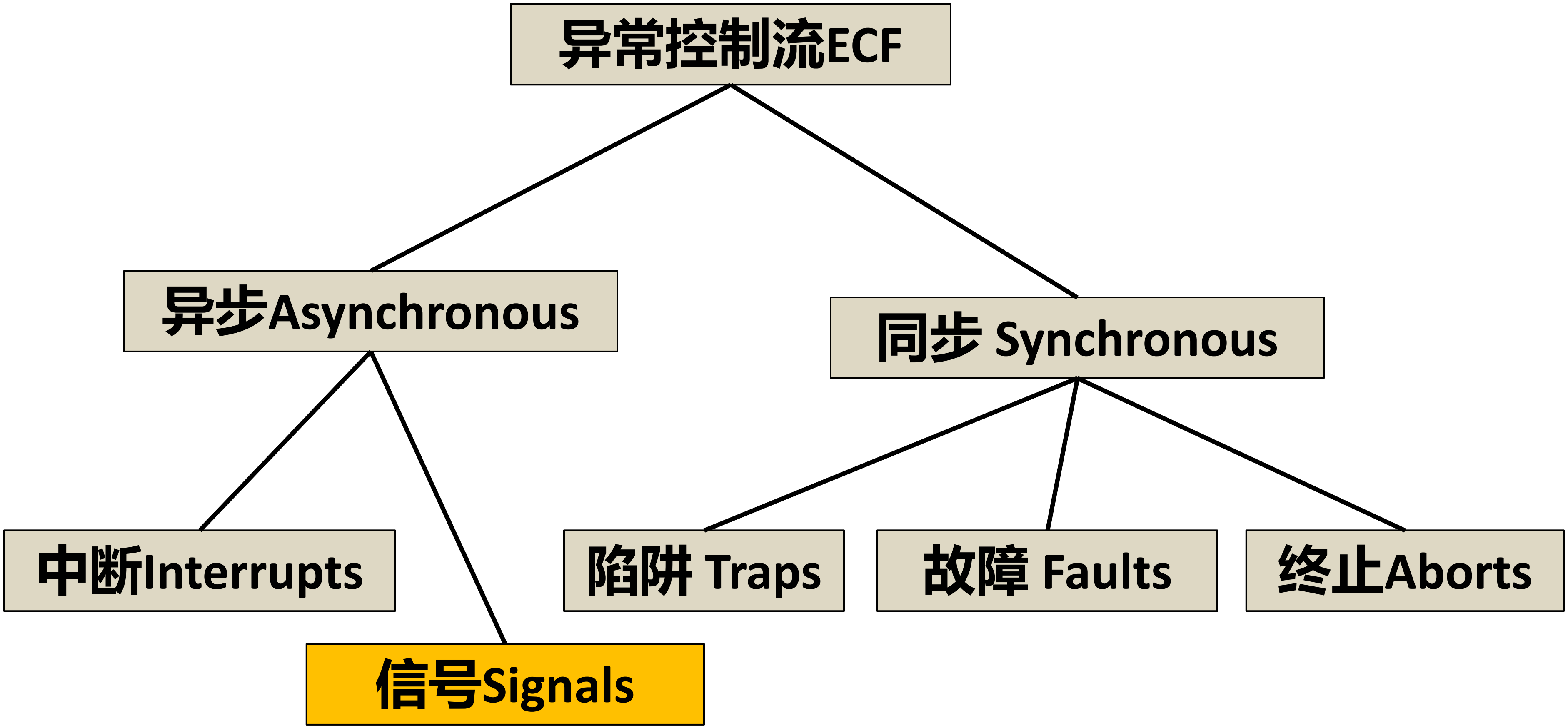
- **非局部跳转 Nonlocal jumps**
  - 应用代码 Application code

教材和补充幻灯片
Textbook and supplemental slides

（部分）分类
(partial) Taxonomy

内核处理 Handled in kernel

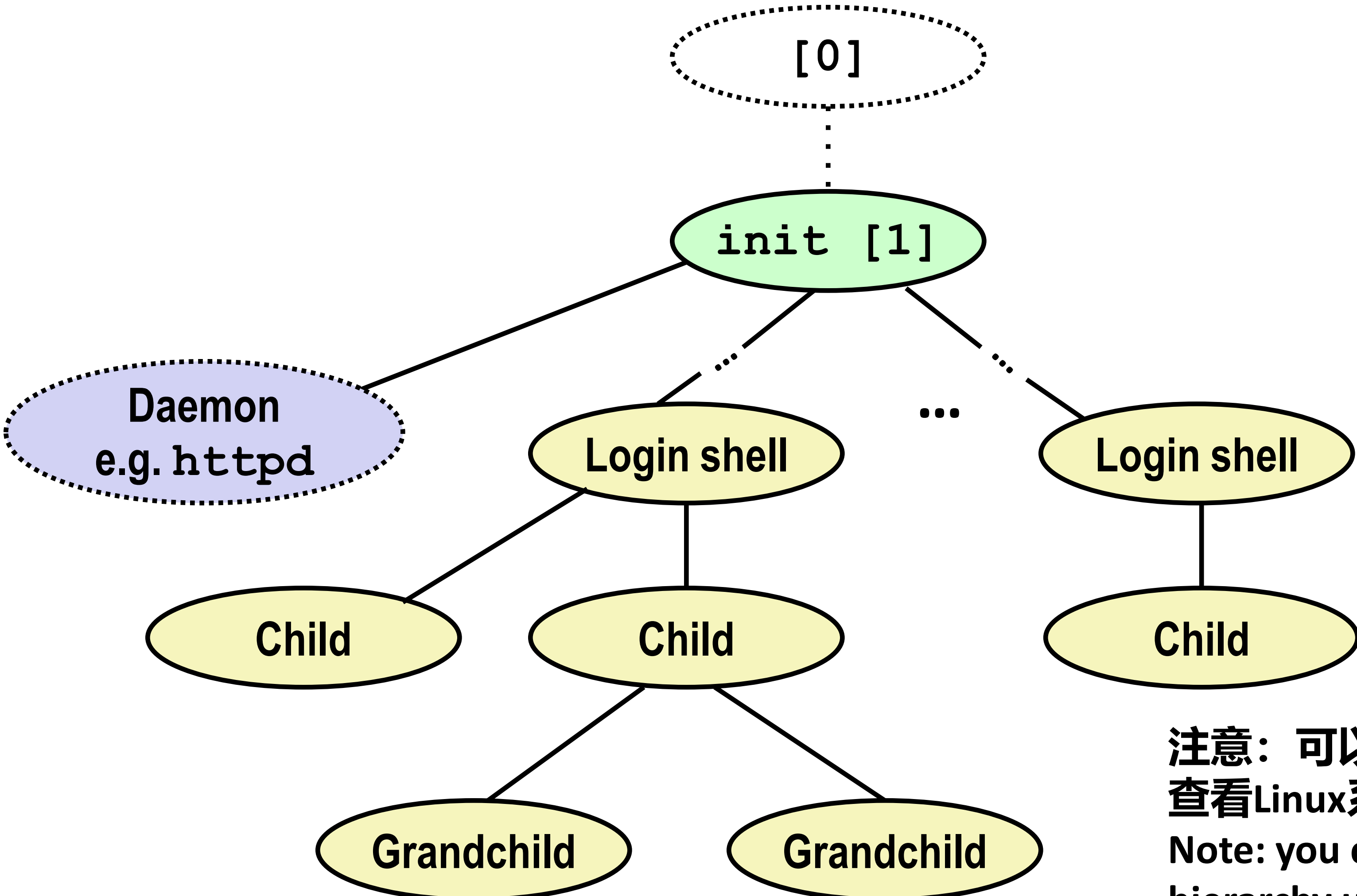
用户进程处理 Handled in user process



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

```
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

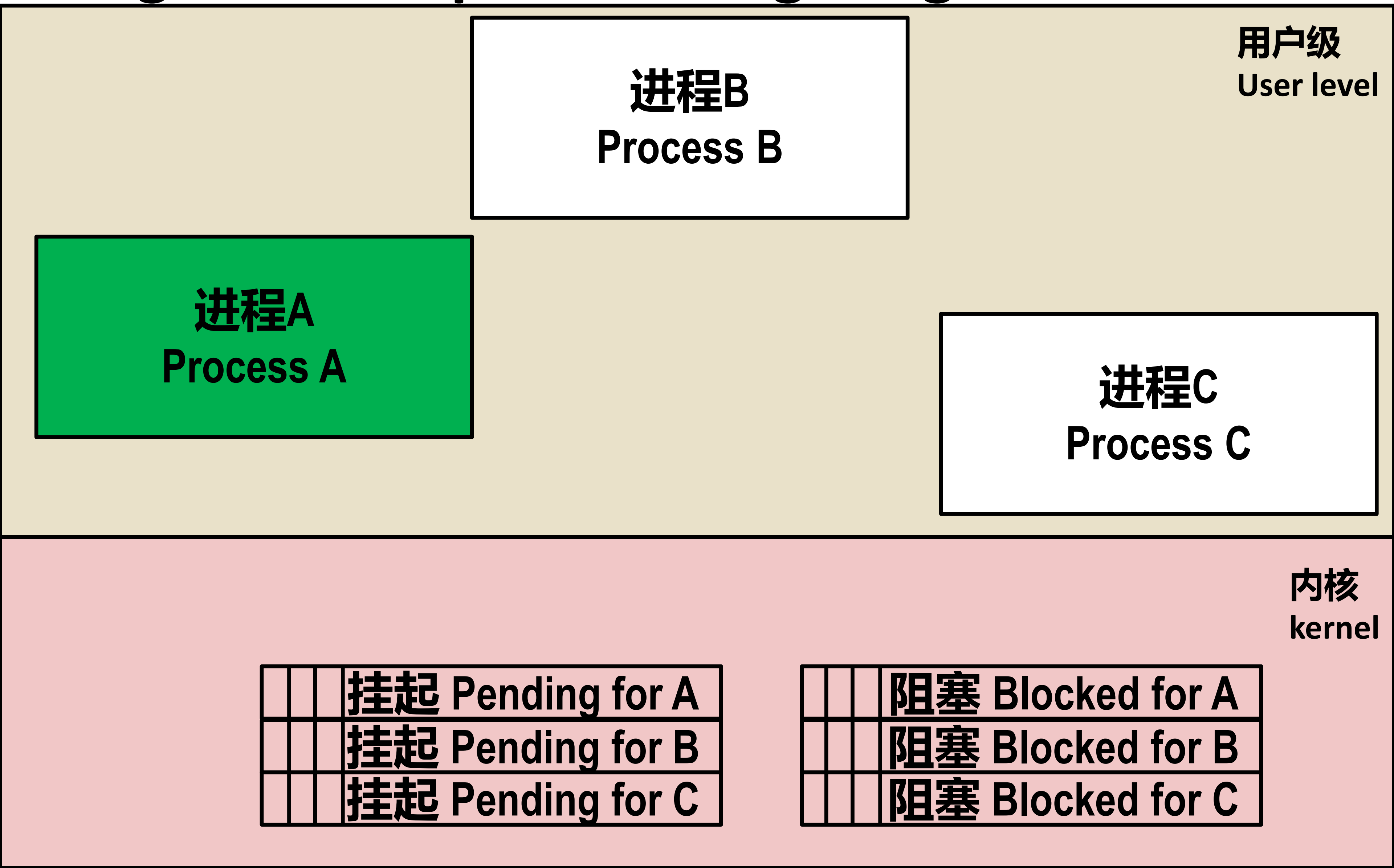
ID	*Name*	*Default Action*	*Corresponding Event*
2	SIGINT	Terminate	用户输入ctrl-c User typed ctrl-c
9	SIGKILL	Terminate	杀死程序（不能覆盖或被忽略）Kill program (cannot override or ignore)
11	SIGSEGV	Terminate	段错误 Segmentation violation
14	SIGALRM	Terminate	时钟信号 Timer signal
17	SIGCHLD	Ignore	子进程停止或者终止 Child stopped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

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

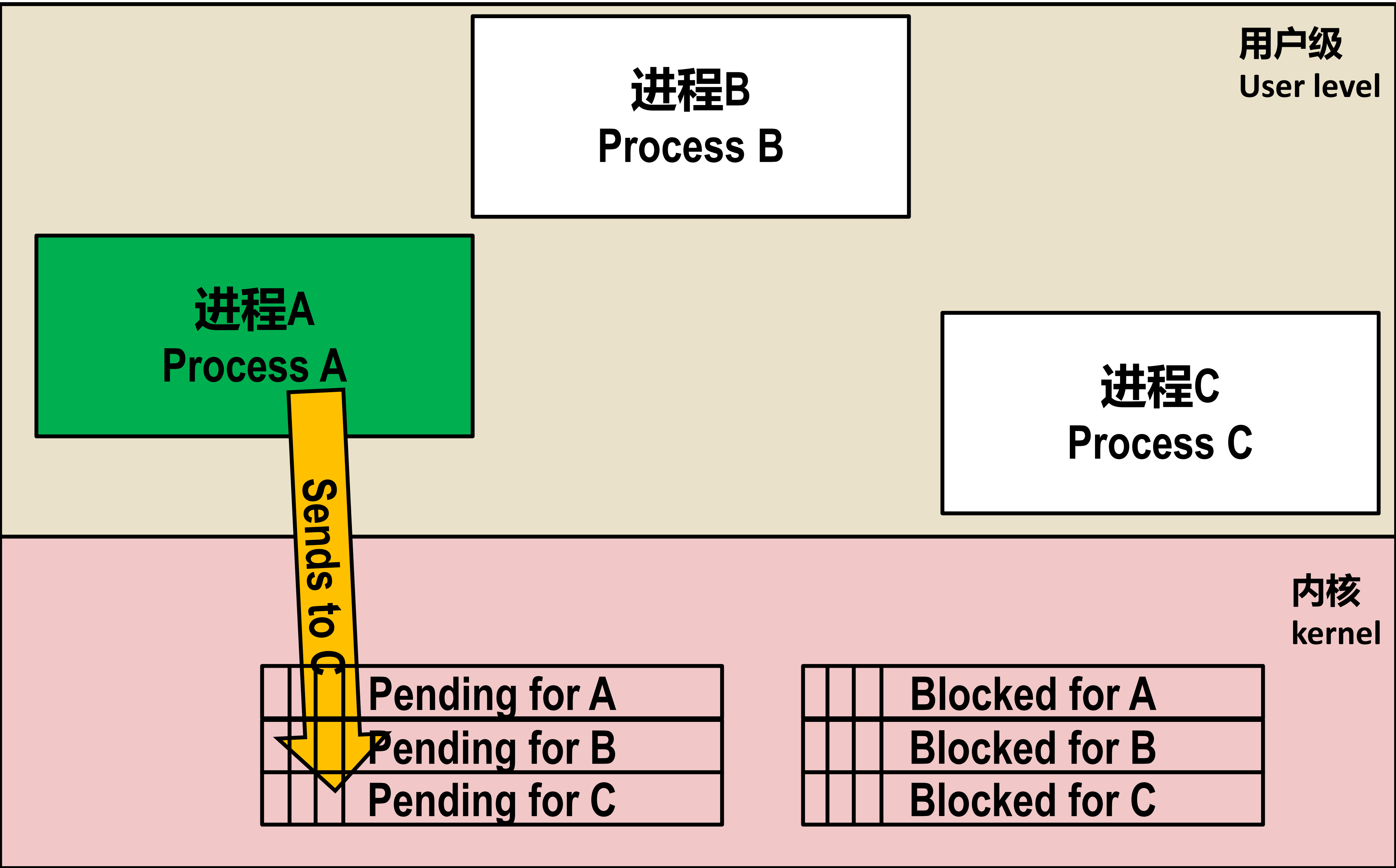
- **内核通过更新目标进程上下文的某些状态来*发送*（传递）一个信号给*目标进程* Kernel *sends* (delivers) a signal to a *destination process* by updating some state in the context of the destination process**

- **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

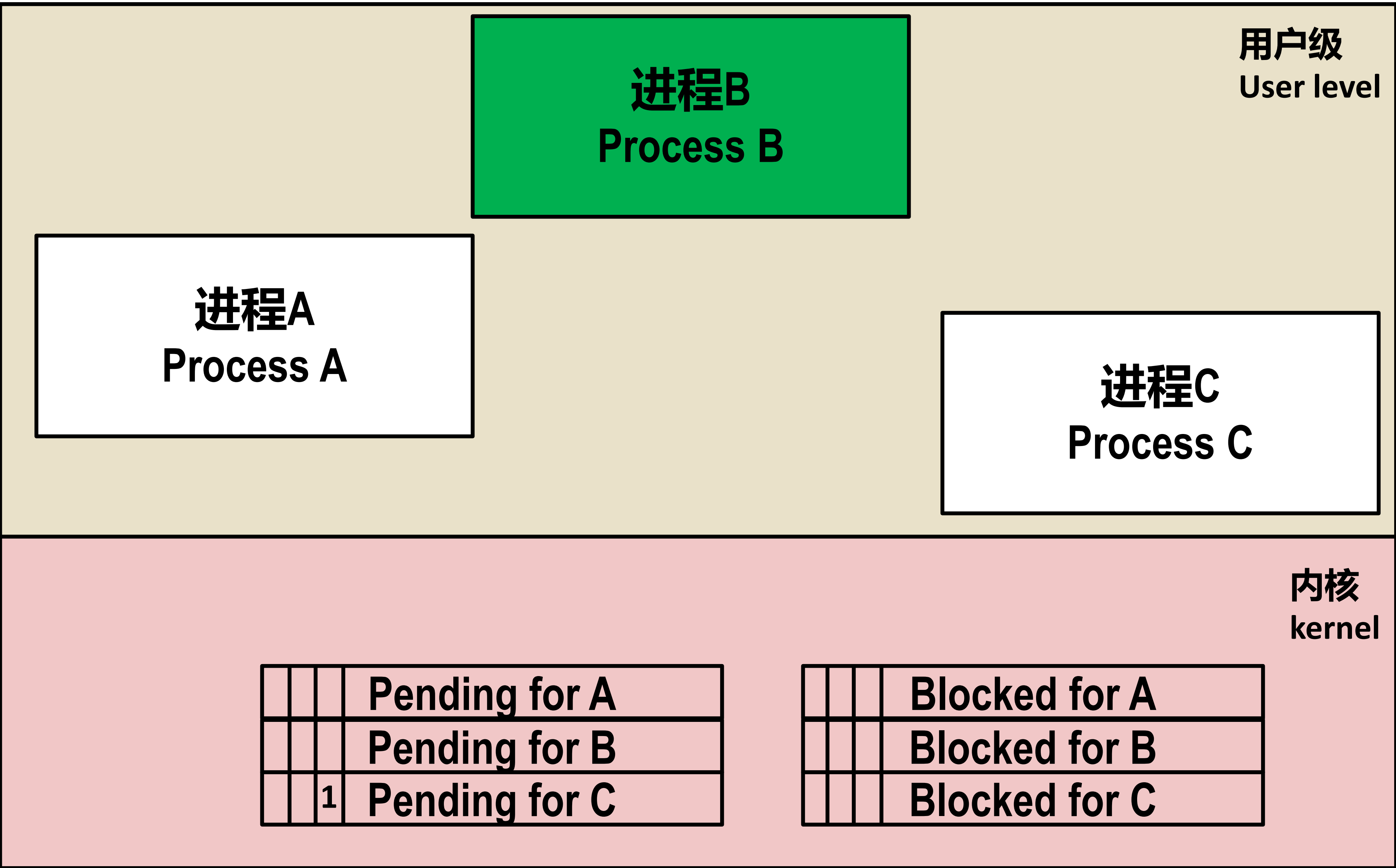
信号概念：发送一个信号
Signal Concepts: Sending a Signal



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

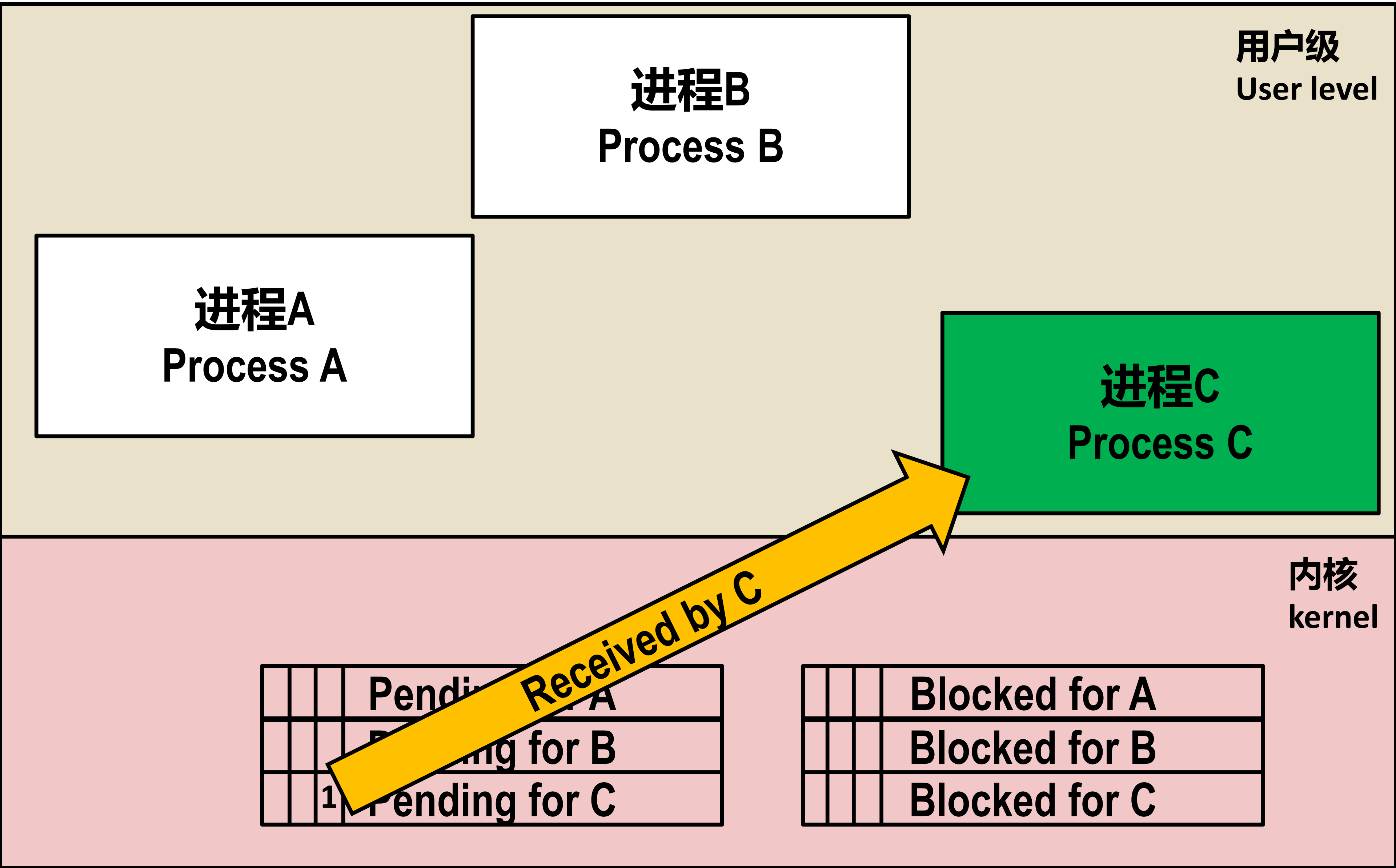


信号概念：发送一个信号
Signal Concepts: Sending a Signal

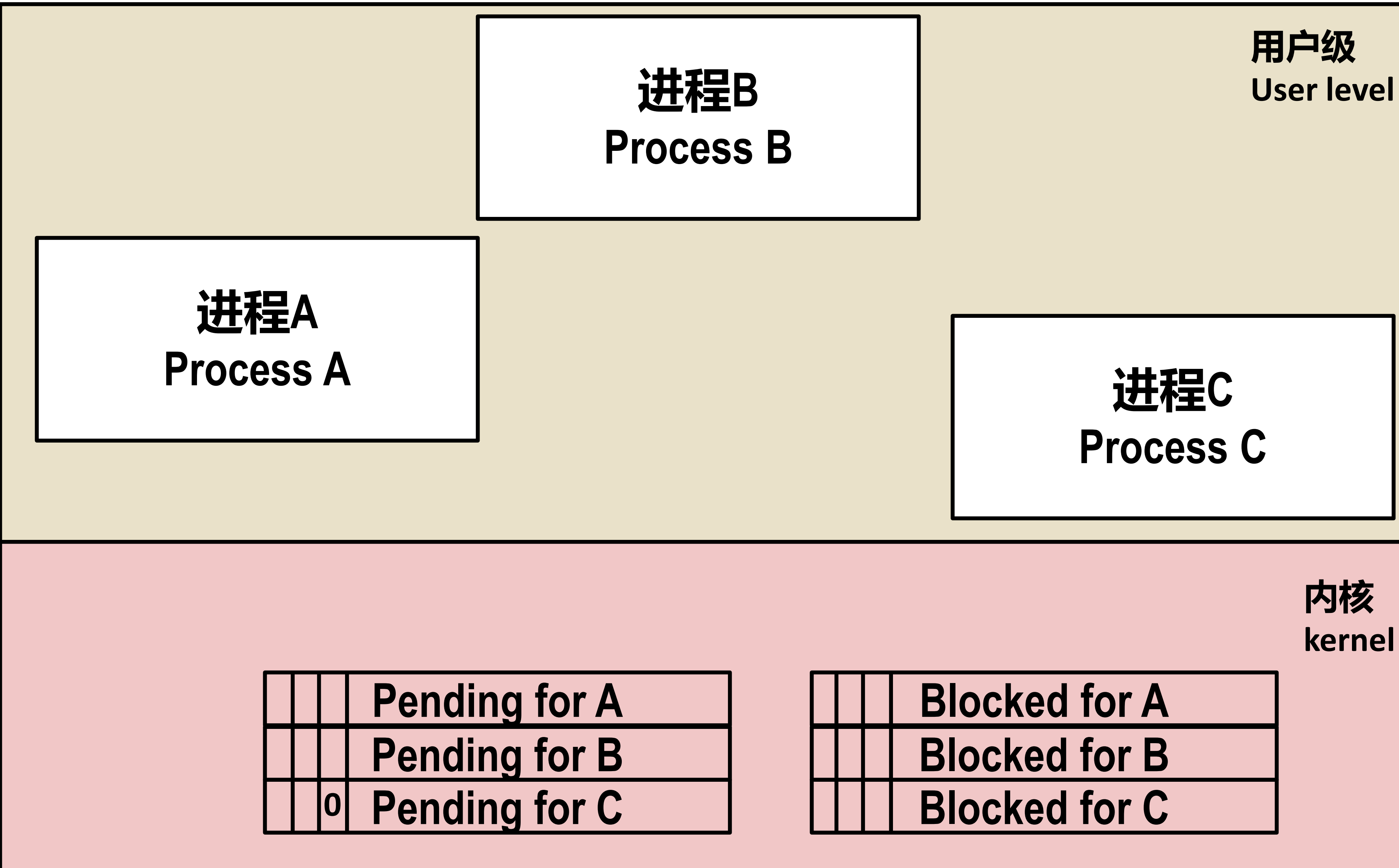


信号概念：发送一个信号Signal

Concepts: Sending a Signal



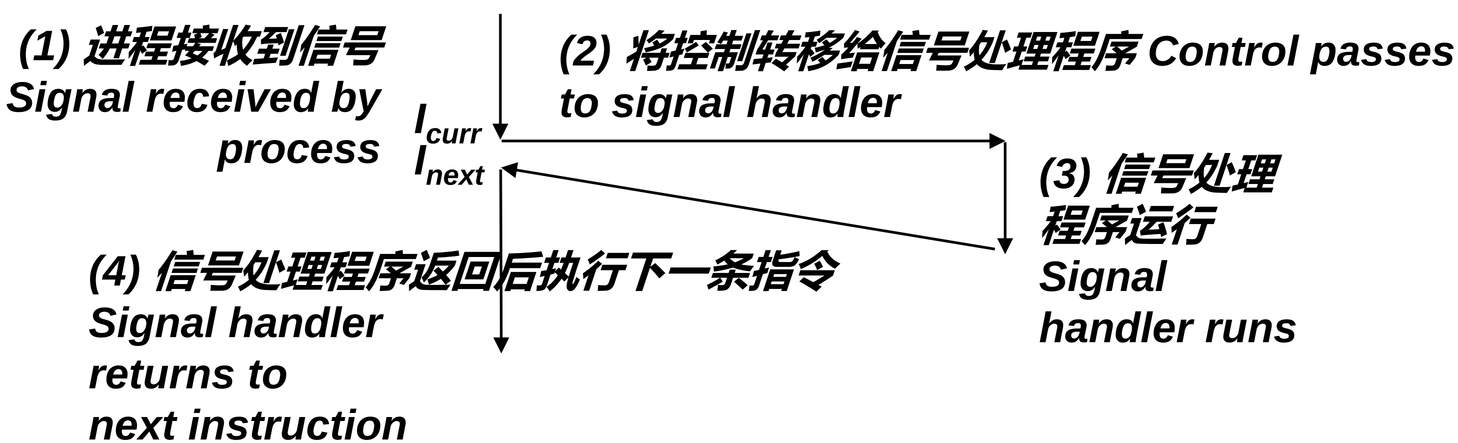
信号概念：发送一个信号Signal
Concepts: Sending a Signal



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

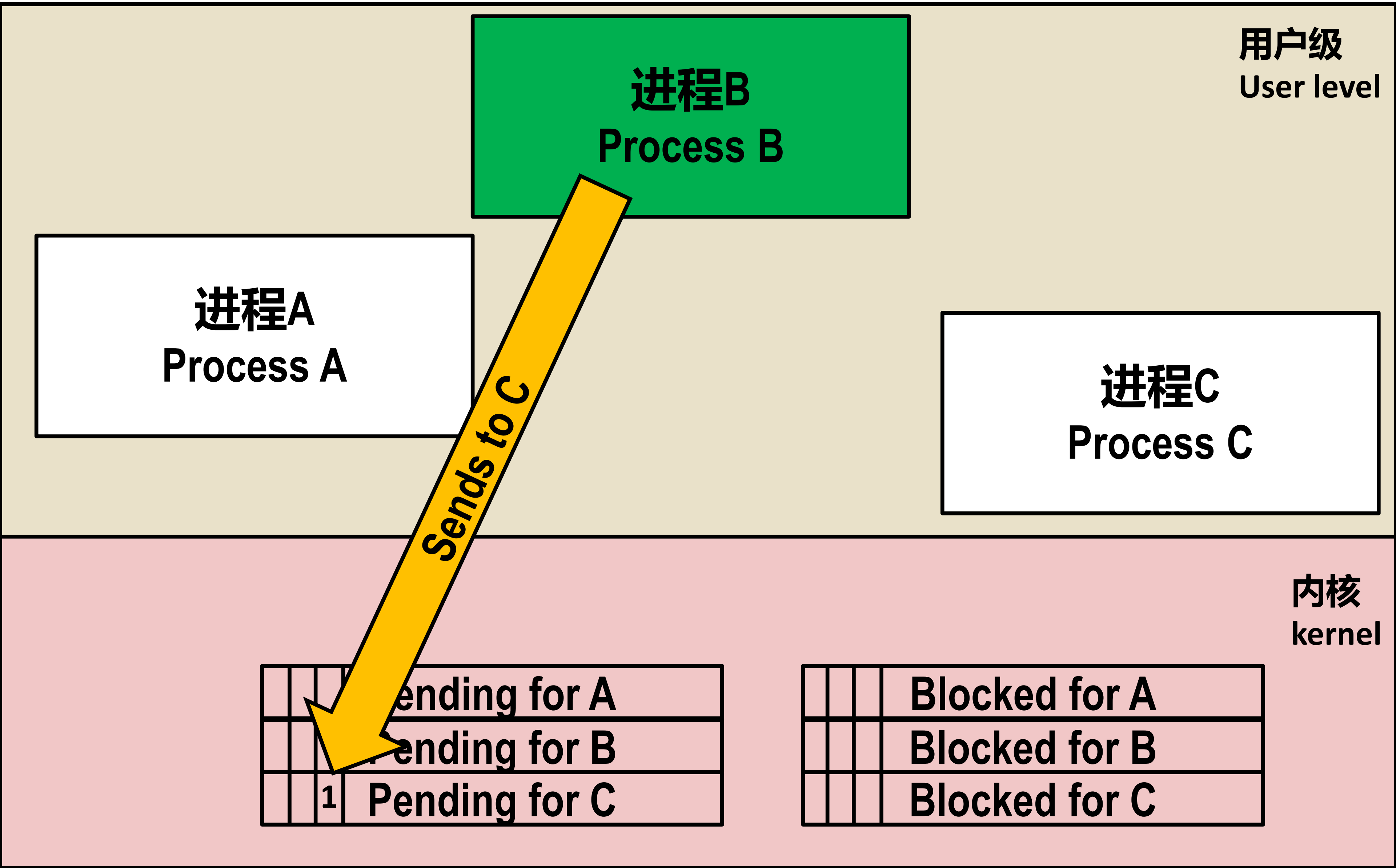
- **一个进程会*阻塞*某种特定类型信号的接收 A process can *block* the receipt of certain signals**
  - 阻塞的信号可以发送，但是在解除阻塞前不会被接收 Blocked signals can be delivered, but will not be received until the signal is unblocked
  - 有些信号不能被阻塞（SIGKILL, SIGSTOP）或者仅当其它进程发送（SIGSEGV、SIGILL等）时被阻塞 Some signals cannot be blocked (SIGKILL,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

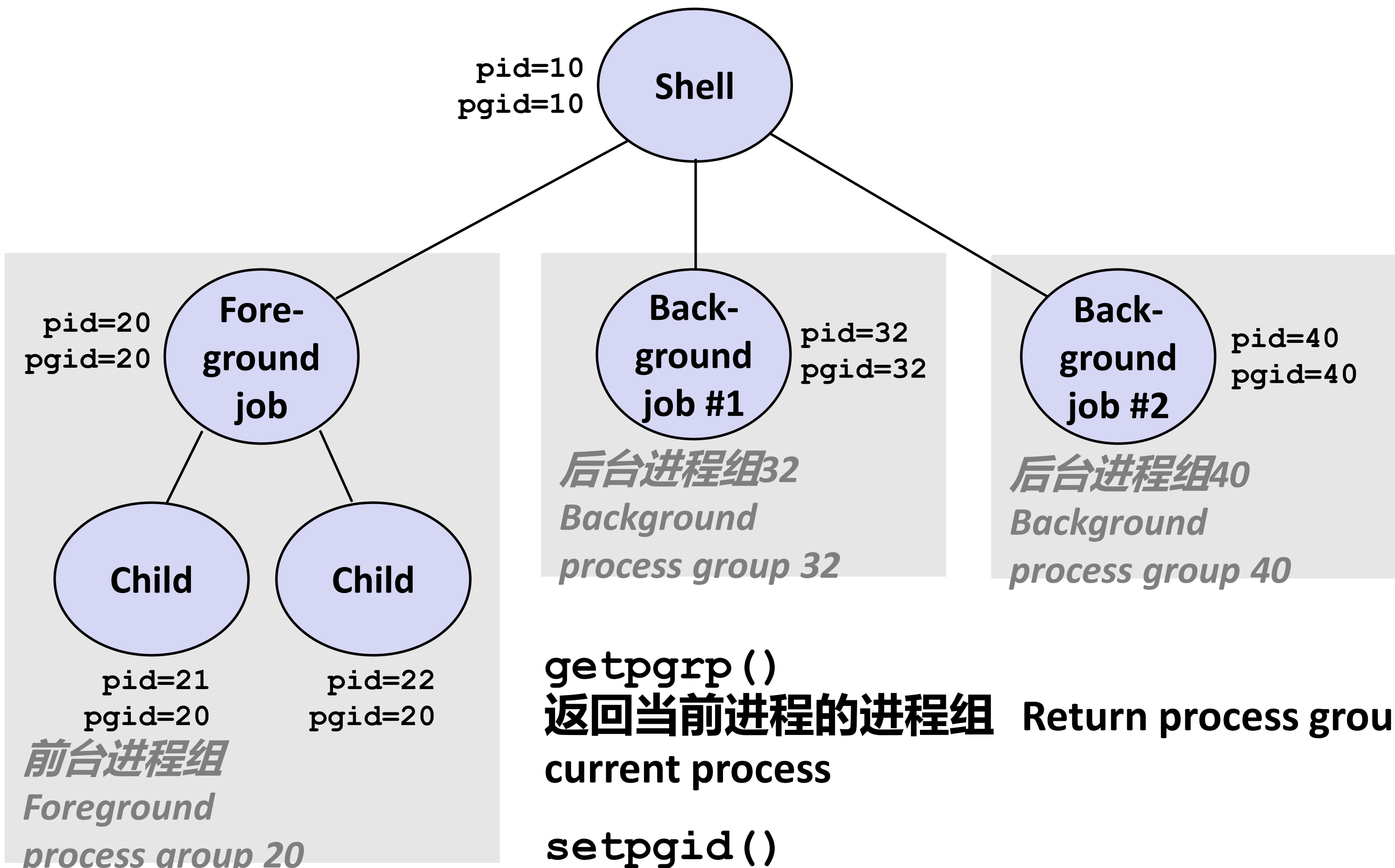
- **内核在每个进程的上下文维护一个挂起和阻塞的比特向量 Kernel maintains `pending` and `blocked` bit vectors in the context of each process**
  - **挂起：表示挂起的信号集合** `pending`: represents the set of pending signals
    - 当发送了一个k类型的信号时系统设置第k个比特位 Kernel sets bit k in `pending` when a signal of type k is delivered
    - 当类型k的信号被接收后系统会将第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)

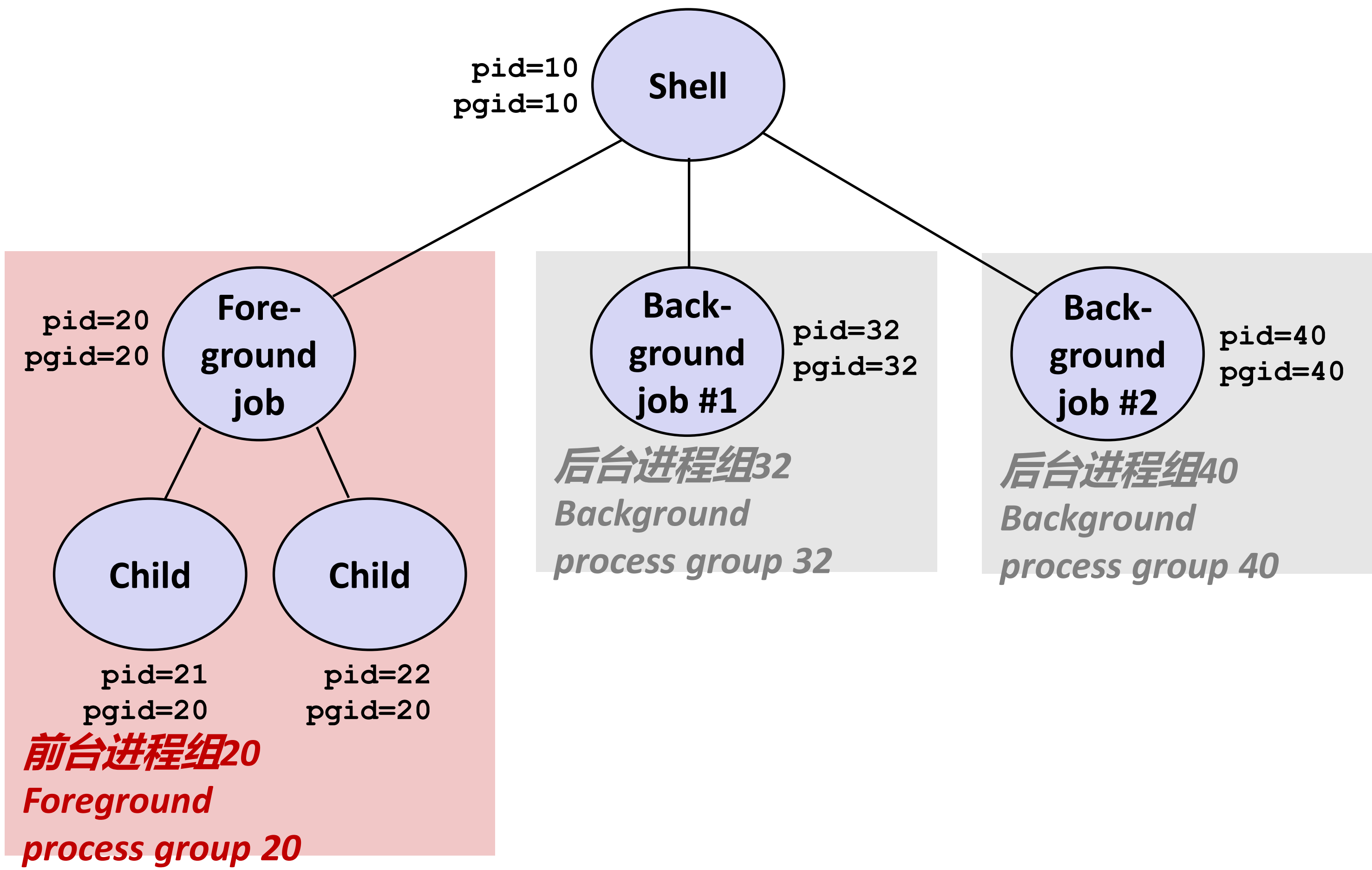
通过/bin/kill程序发送信号
Sending Signals with `/bin/kill` Program

- `/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

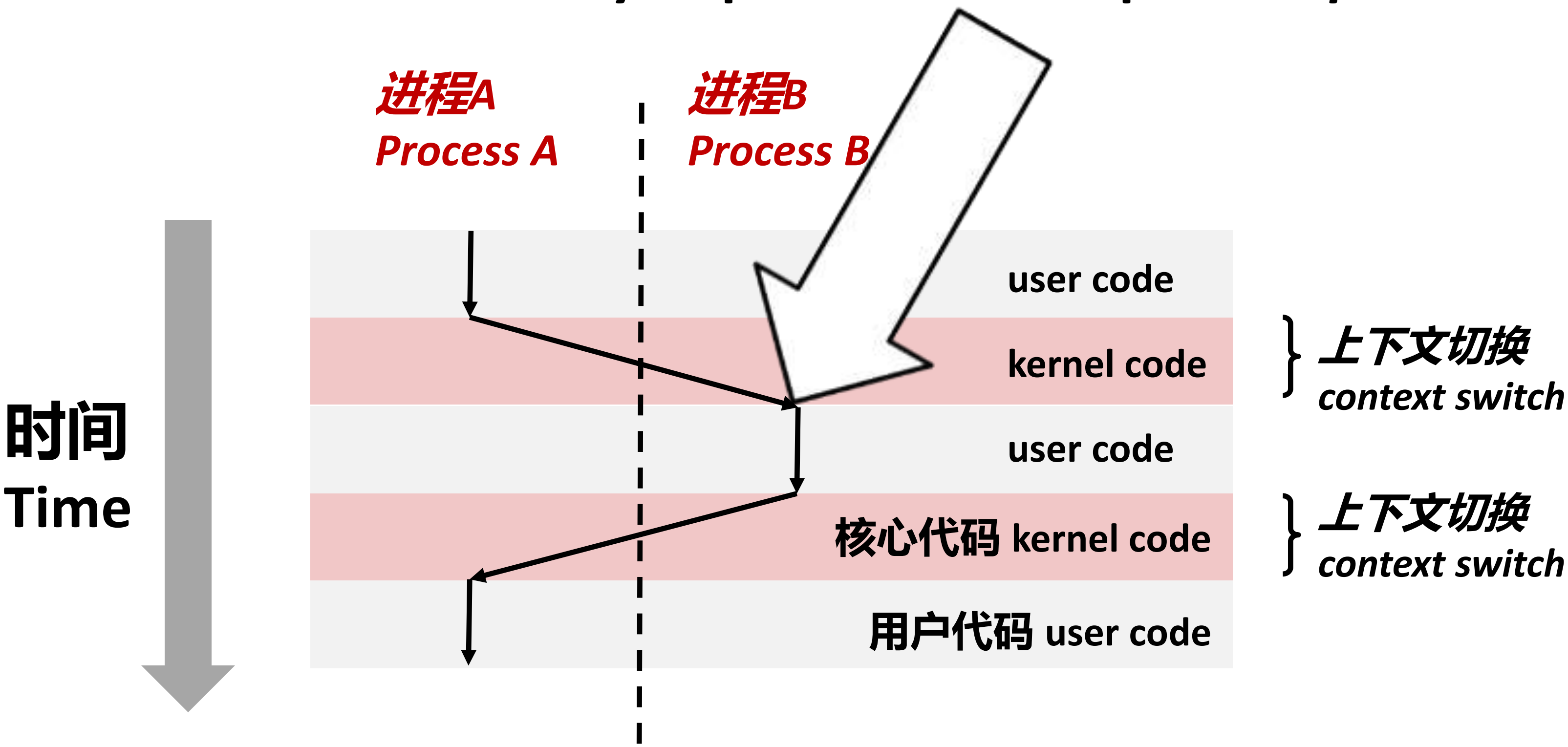
通过kill函数发送信号
Sending Signals with `kill` Function

```
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

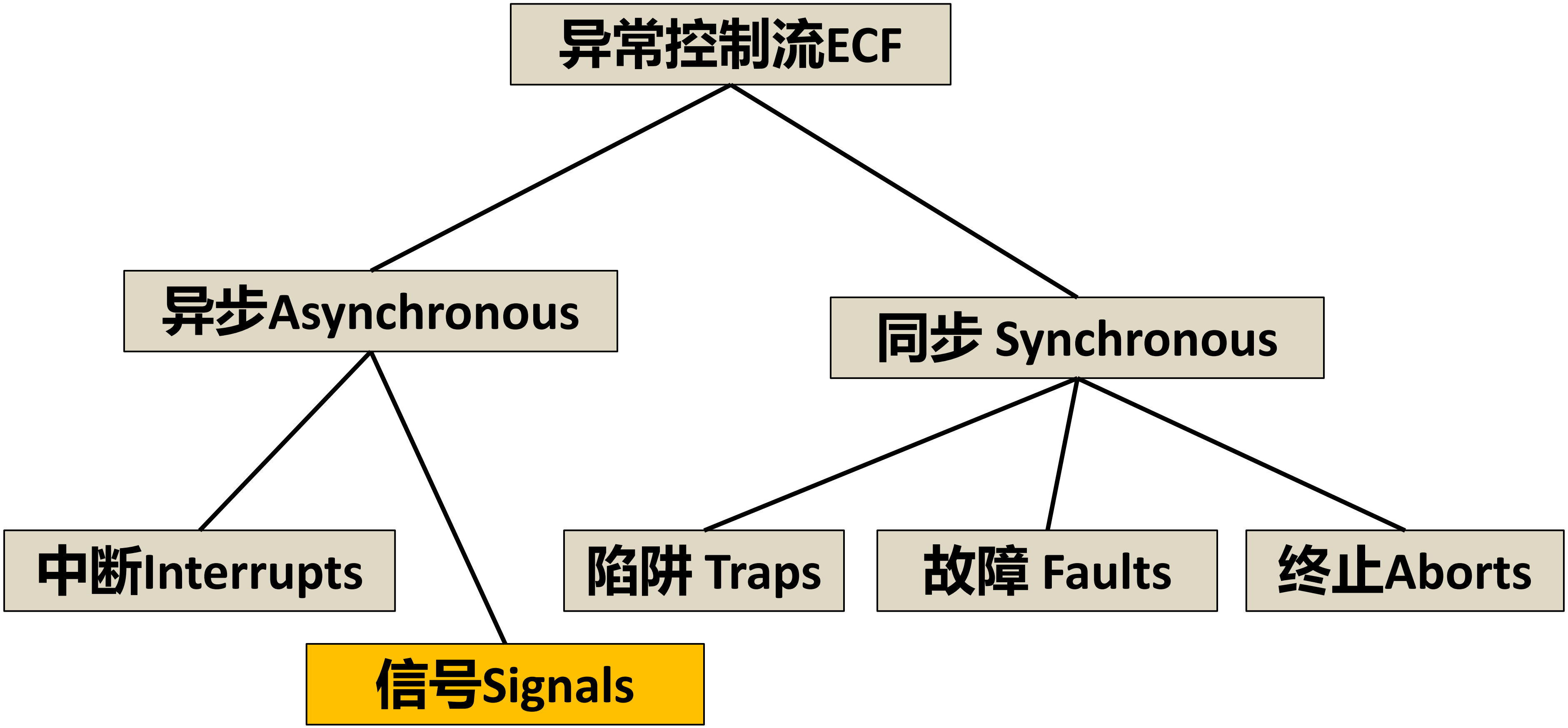
回顾

- **异常控制流**
- **异常的分类**
- **进程**
  - 进程在真实是怎么实现的?
  - 它提供了什么抽象?
  - 进程图会画吗?
  - 如何判断一个输出是多进程可以可行的结果?
  - 并发运行和并行运行如何理解?
  - 几个重要的系统调用
    - fork、wait、waitpid、execve、sleep、pause、kill

（部分）分类
(partial) Taxonomy

内核处理 Handled in kernel

用户进程处理 Handled in user process



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

```
#include <signal.h>
```

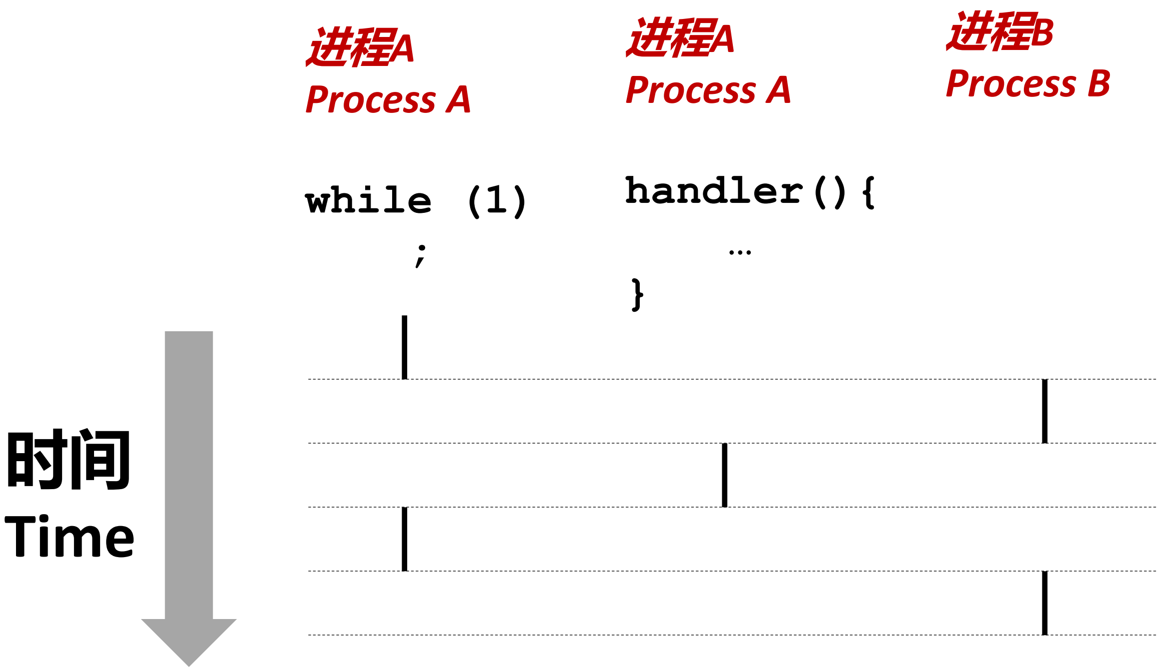
信号处理例子 Signal Handling Example

```
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;
}
```

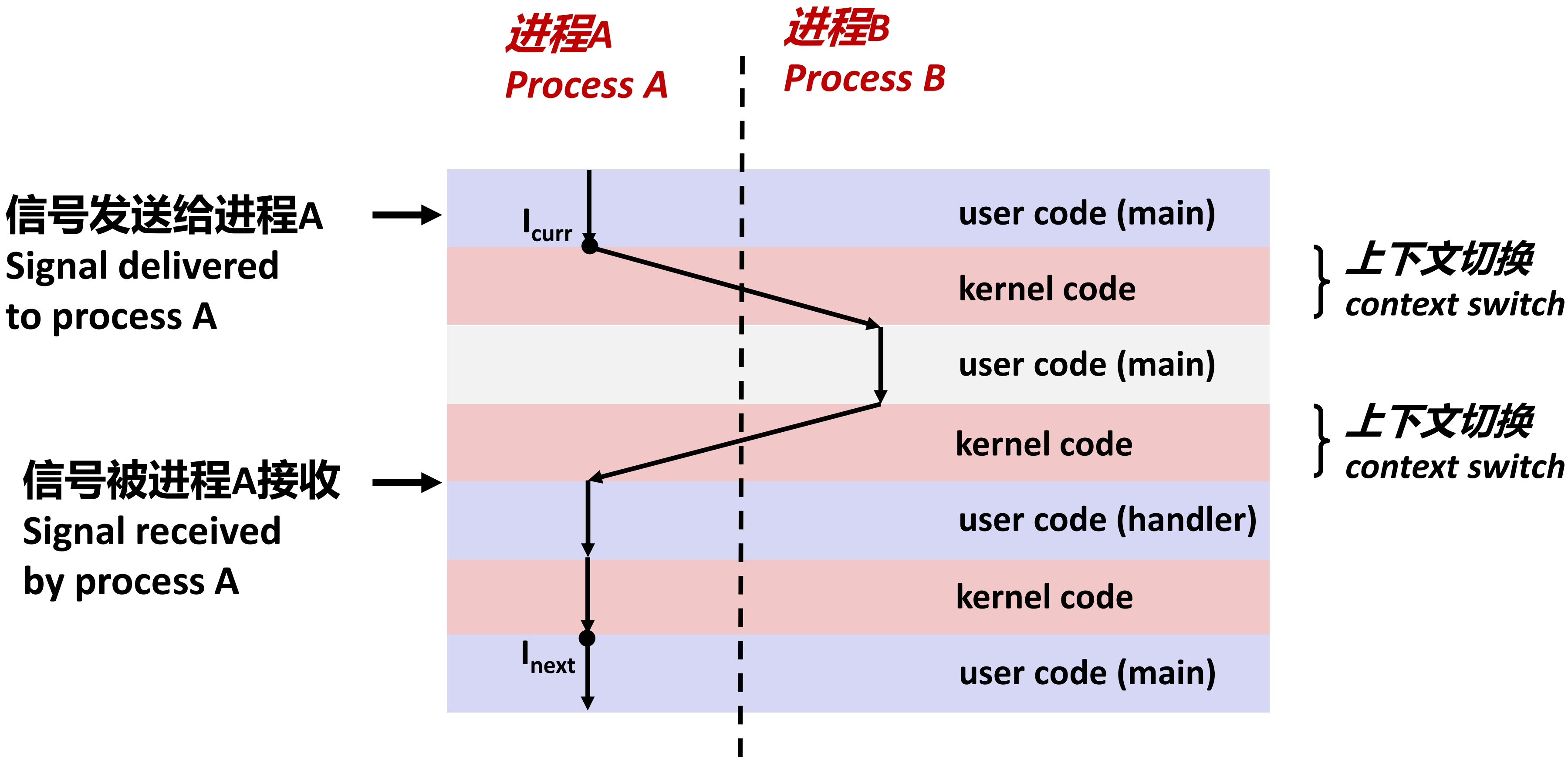
sigint.c

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

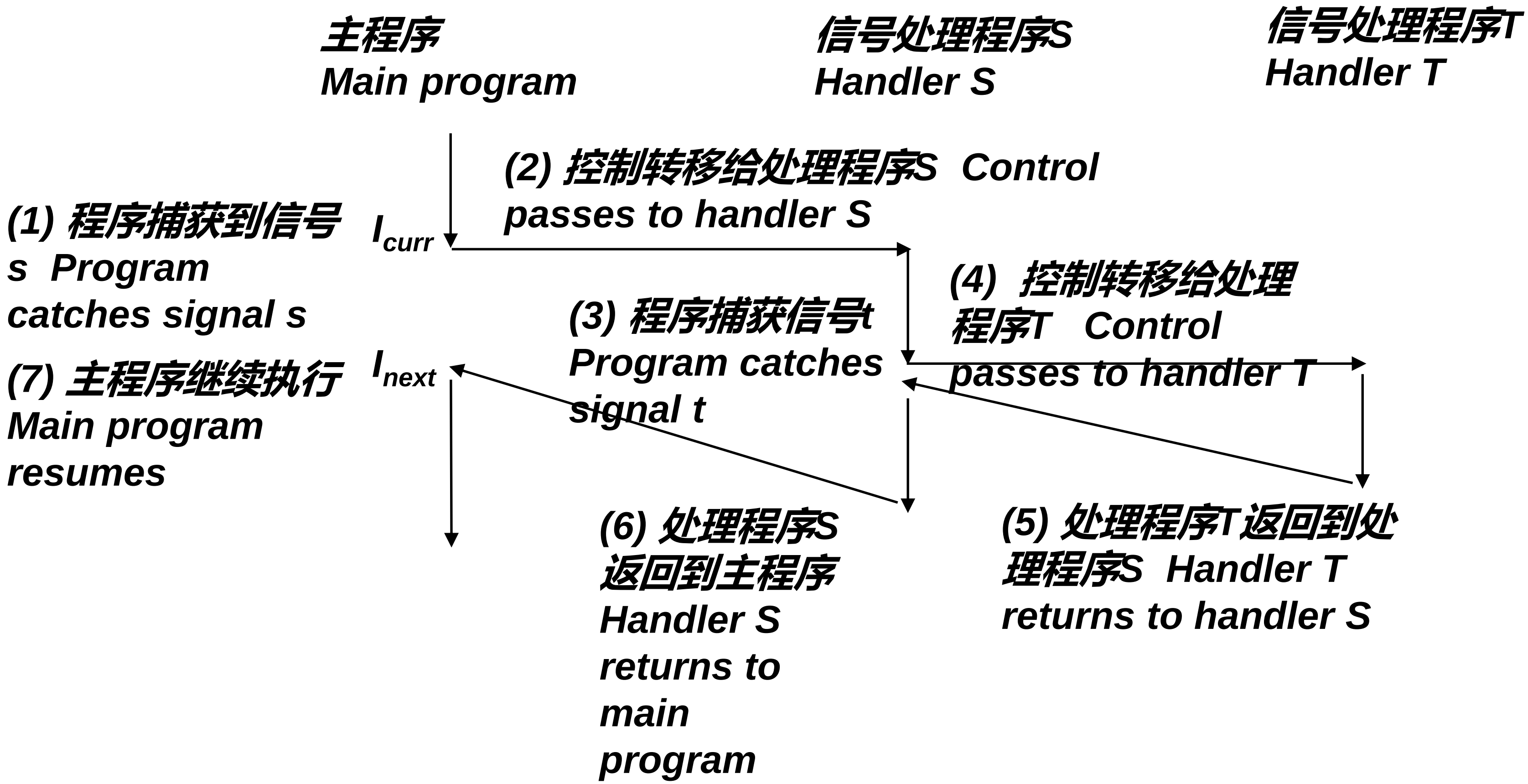


信号处理程序作为并发控制流的另一个视图
Another View of Signal Handlers as Concurrent Flows



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`
  - 常见的函数并不在其中　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;
    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

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 (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**

```
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;
    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);
        /* 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
- **Solution: `sigsuspend`**

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;
    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(".");
    }
    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

对比总结 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
  - 可以声明并实现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

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
  - 识别出后续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中来记住当前位置 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);
}
```

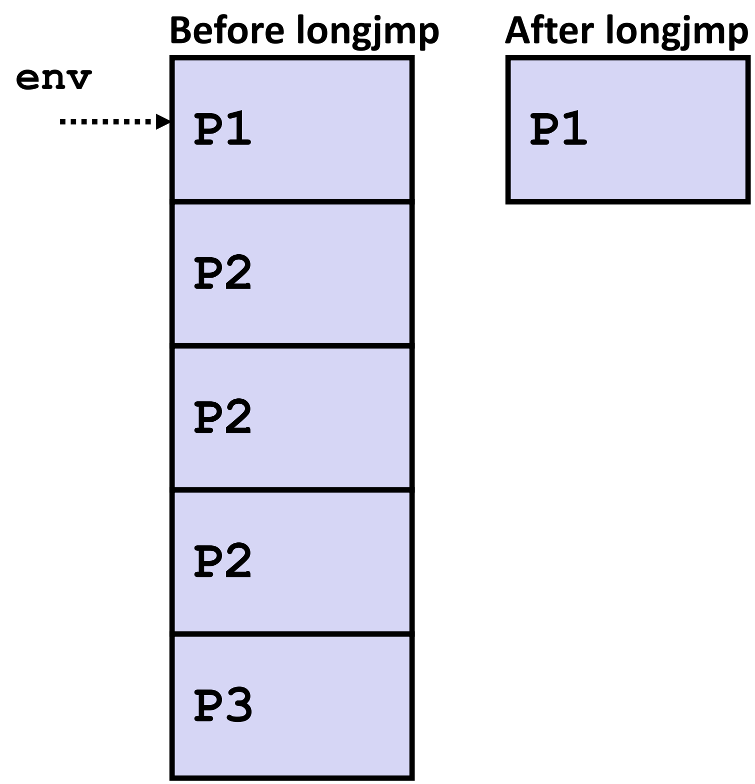
setjmp/longjmp 示例/Example (续/cont)

```
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

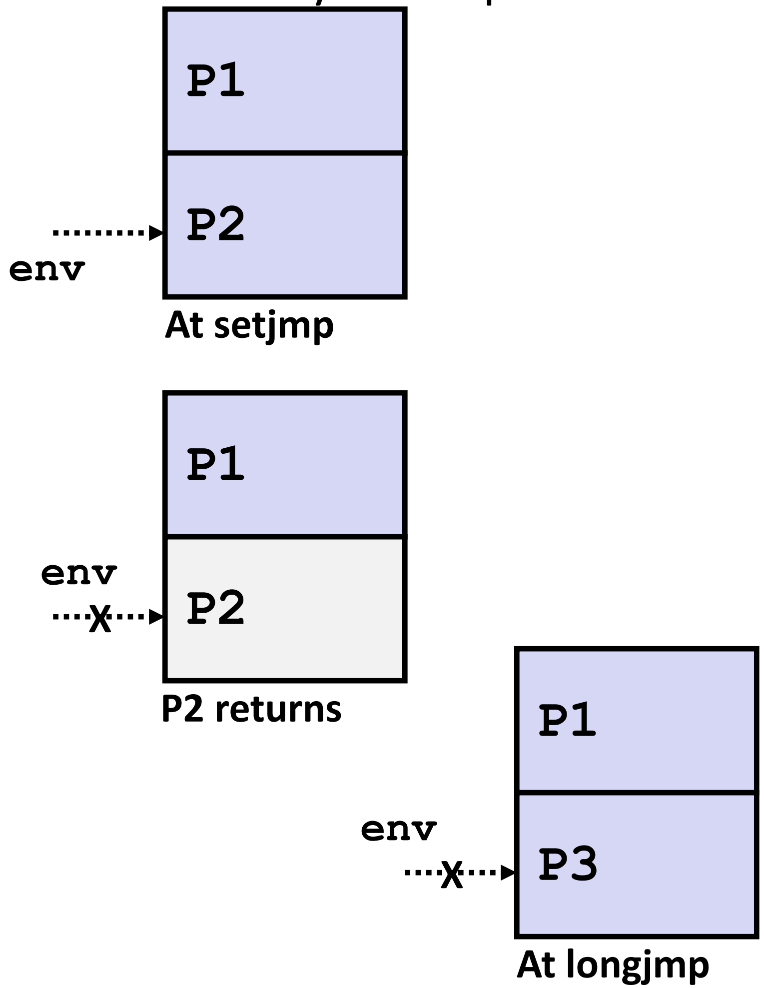
```
jmp_buf env;

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

```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

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