异常与进程
Exceptions and Processes

课 程 名 ： 计算机系统
主 讲 人 ： 孟文龙

本课内容

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

控制流

- 处理器只做一件事：
  - 从加电到断电，CPU 的所有工作仅限于读取并执行一个指令序列，一次处理一条指令
  - 这个指令序列就是该处理器的控制流*control flow*

实际的控制流

<开机>

$inst_1$

$inst_2$

$inst_3$

...

$inst_n$

<关机>

```c
#include <stdio.h>

int main() {
    printf("开机\n");       // 模拟开机
    printf("inst1\n");     // 指令1
    printf("inst2\n");     // 指令2
    printf("inst3\n");     // 指令3
    // ... 可以继续添加更多“指令”
    printf("关机\n");       // 模拟关机
    return 0;
}
```

控制流的改变

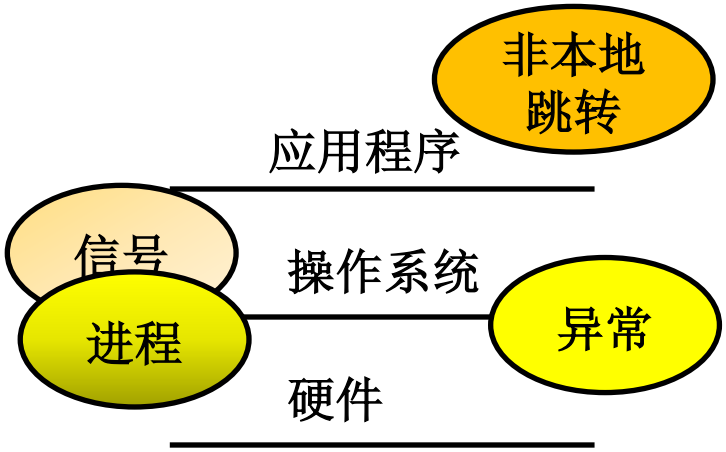
- 目前为止，改变控制流的两种机制：
  - 无条件跳转 `jmp` 和条件转移
  - 调用 `call` 和返回 `return`
- 仅有这些还不足以造就一台真正的计算机，因其难以对**系统状态**的变化做出反应：
  - 由磁盘或网卡发来了数据
  - 出现被零除错误
  - 用户在键盘上敲击Ctrl-C
  - 系统定时器超时

这些系统变化无法用程序变量表示

- 现代计算机系统需要针对“控制流发生突变”的情况做出反应，称为**异常控制流**机制ECF

异常控制流Exception Control Flow

- 发生在计算机系统的所有层次
- 底层（硬件层次）机制
  - 1. 异常
  - 硬件检测到的事件会触发控制转移到异常处理程序
    - 由硬件和操作系统共同实现
- 高层机制
  - 2. 进程上下文切换
    - 由操作系统和硬件定时器实现
  - 3. 信号(Ctrl-C)
    - 操作系统实现
  - 4. 非本地跳转：`setjmp()` 和 `longjmp()`
    - 由 C 运行时库实现

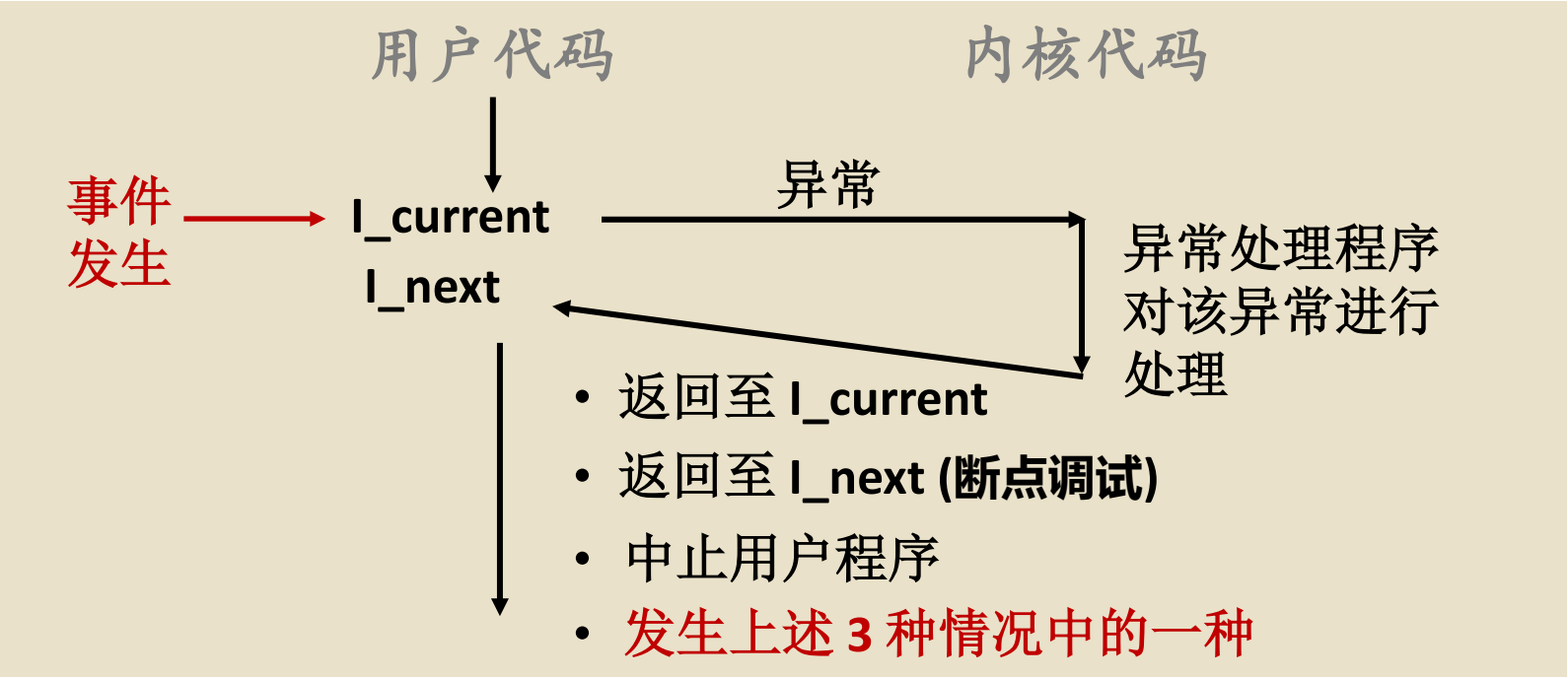


本课内容

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

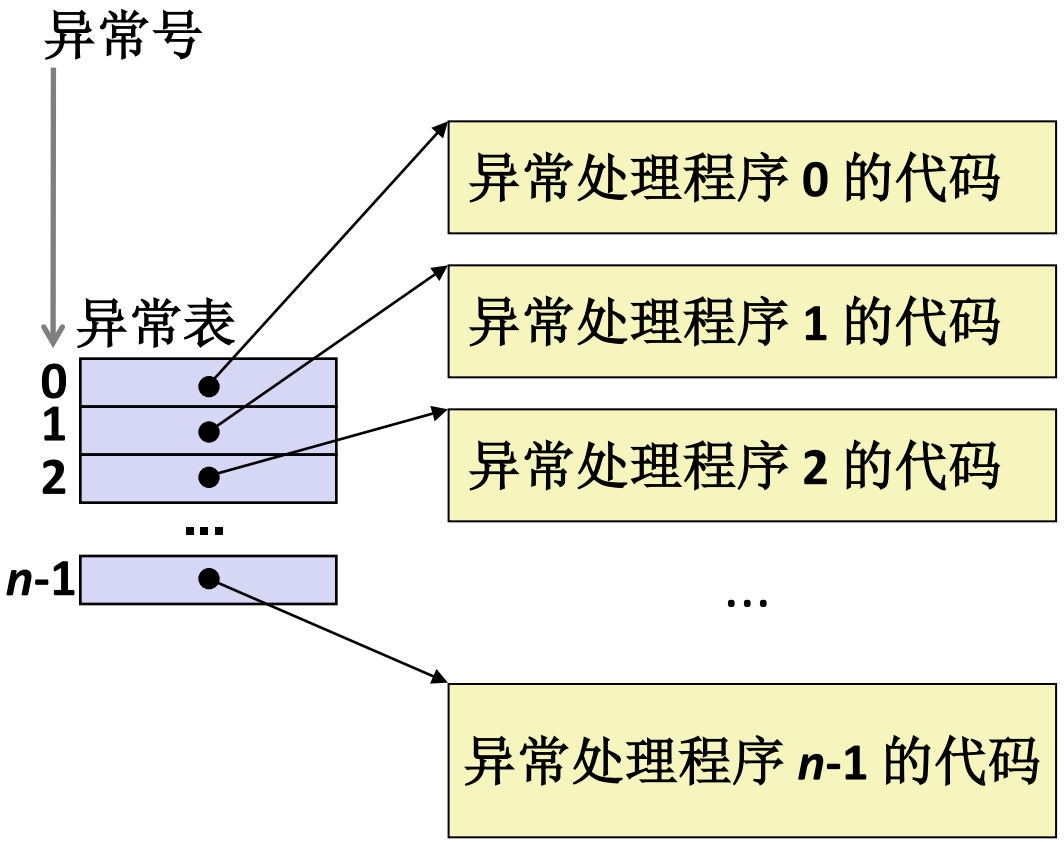
异常

- **异常**是指为响应处理器状态的变化而引发的**控制流的突变**（控制权转移至操作系统内核）
  - 内核指的是操作系统常驻内存的部分
  - 典型的事件：被零除、算术运算溢出、缺页、键入 Ctrl-C、I/O 请求完成



异常表exception tables

硬件 + 软件的配合



- 每一类事件有唯一的异常号 *k*
- 异常号 *k* 为异常表的索引
  - 异常表（又称中断向量表、中断描述符表等）为一跳转表，表目 *k* 包含异常 *k* 的处理程序的起始地址
- 找寻异常表：ETBR异常表基址寄存器（抽象名称）

异常号 (k)	计算公式	得到的地址	存放内容
4	ETBR + 4 × (表项大小)	ETBR + 4 × (表项大小)	处理程序4的入口地址

- 每当异常 *k* 发生，异常 *k* 的处理程序立刻被调用

异步异常（中断）

- **处理器外部事件引起**
  - 通过设置处理器的中断引脚提示
  - 中断处理程序返回至下一条指令处

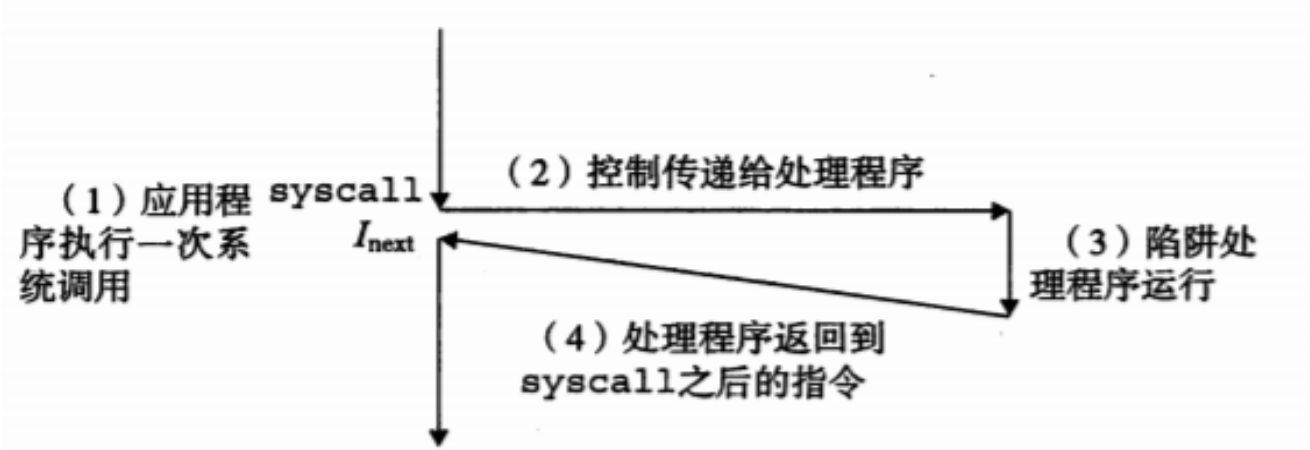
  中断通常发生在一条指令执行完毕后，处理器检测到中断信号，然后暂停后续指令的执行，转而去执行中断服务程序（ISR）。

- **例如：**
  - 时钟中断
    - 外部定时器芯片每隔几毫秒触发一次中断
    - 内核借此从用户程序收回控制权
  - 外部设备的 I/O 中断
    - 用户在键盘上敲击 Ctrl-C
    - 网络数据包到达
    - 磁盘数据到达

同步异常

- **因指令执行而产生的结果，包括 3 类：**
  - **陷阱traps**
    - 主动、有意的（故此得名）

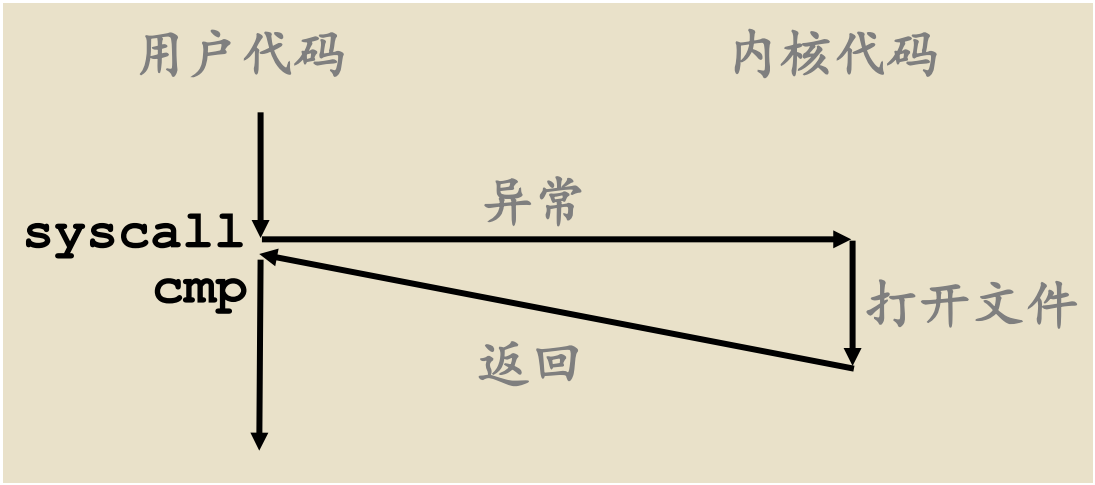
      例：系统调用、断点陷阱、特殊指令
    - 用户程序和 OS 内核之间的接口
    - 陷阱处理程序执行完，控制返回至下一条指令



系统调用举例：打开文件

- 用户调用函数：open(filename, options)
- 调用 __open 函数，该函数借助系统调用指令syscall实现

```c
#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>

int main() {
    int fd = open("test.txt", O_RDONLY);
    if (fd == -1) {
        printf("打开文件失败\n");
        return 1;
    }
    printf("打开文件成功，fd=%d\n", fd);
    close(fd);
    return 0;
}
```



系统调用system call

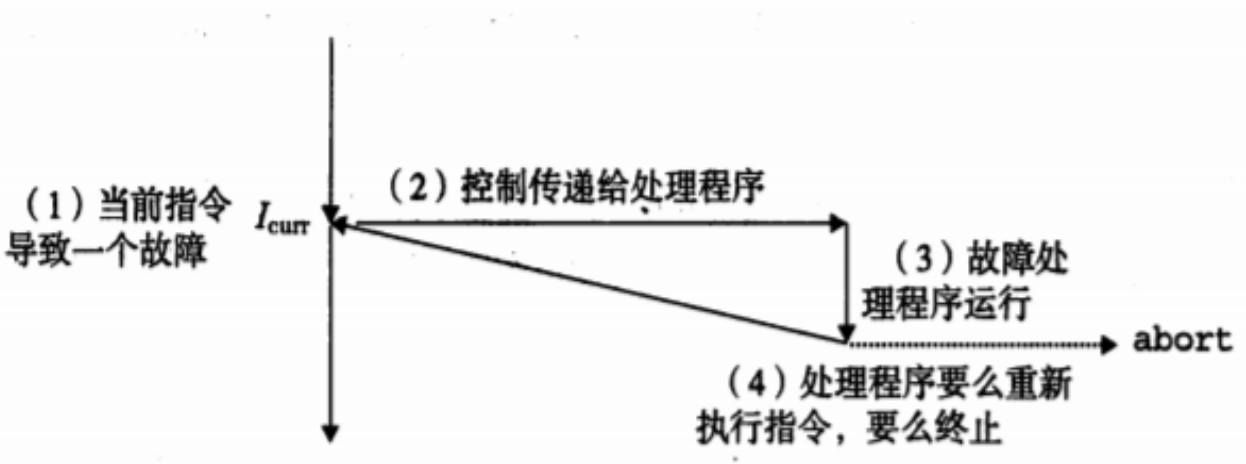
- 每一个 Linux 系统调用有唯一的 ID
- 例如：

编号	名称	说明
0	read	读文件
1	write	写文件
2	open	打开文件
3	close	关闭文件
4	stat	获取文件信息
57	fork	创建进程
59	execve	执行程序
60	_exit	终止进程
62	kill	向进程发信号

exit()为C库函数

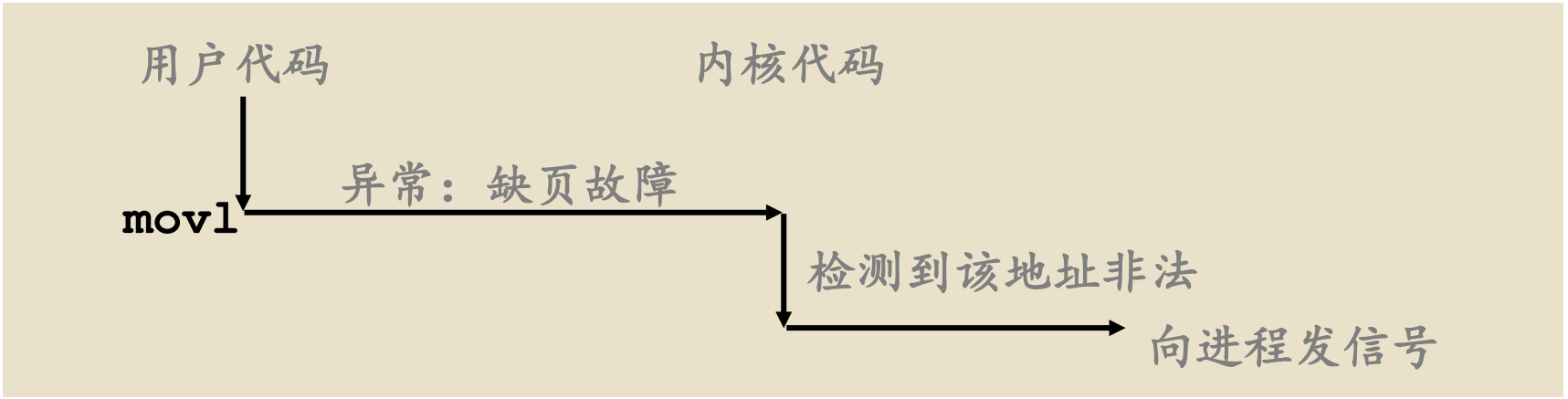
同步异常（续）

- **故障faults**
  - 非有意的，可修正或无法修正
  - 例：缺页（可修正）、保护故障（不可恢复）、浮点异常
  - 要么重新执行引起故障的当前指令（已修复），要么终止该程序



故障举例：非法内存引用

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

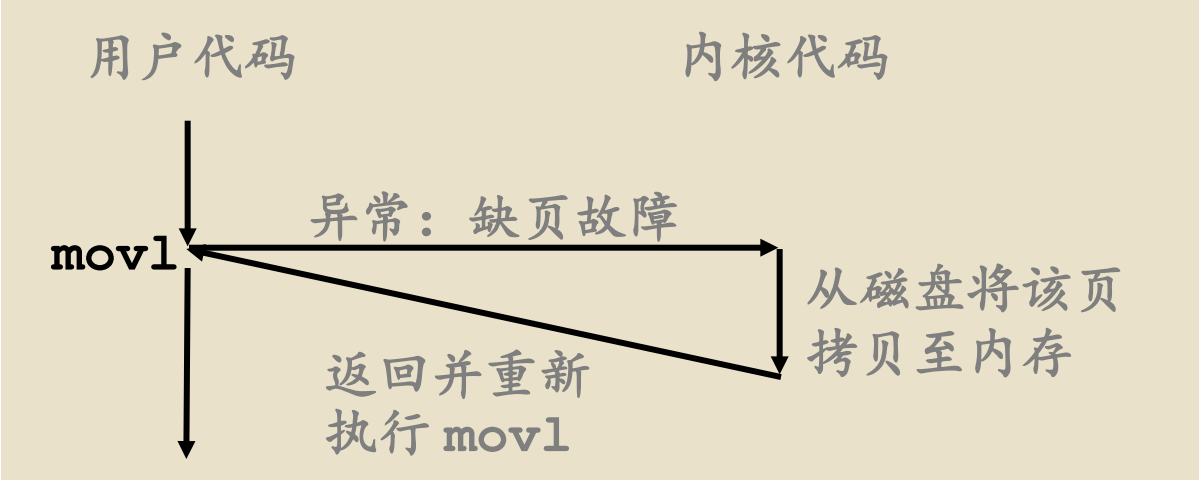


- OS 发送 SIGSEGV 信号给用户进程（不尝试恢复）
- 用户进程以“段故障”（segmentation fault）退出

故障举例：缺页故障page Fault

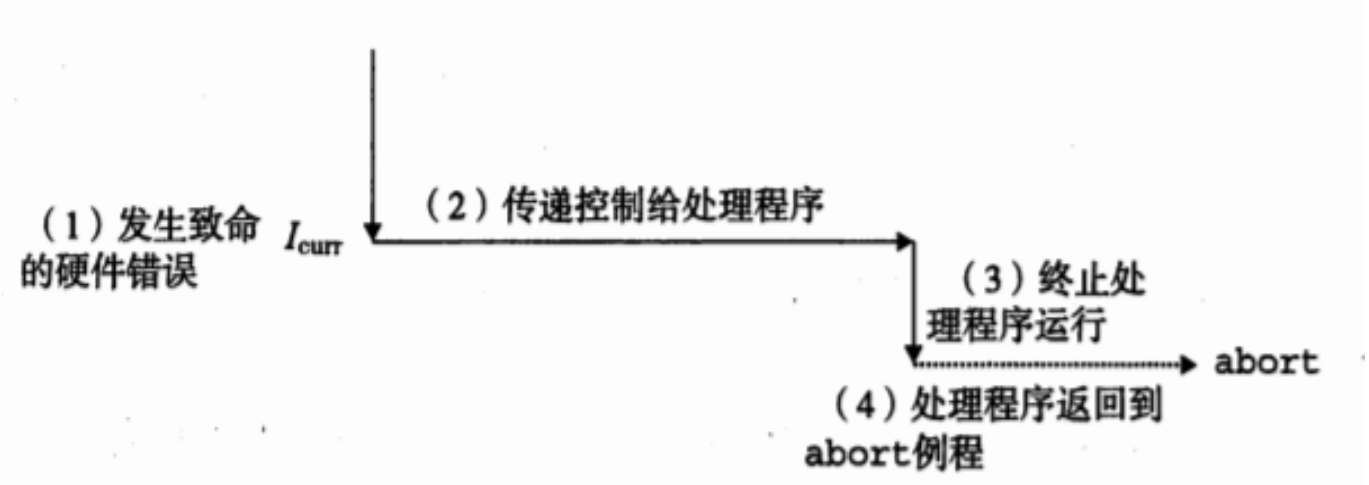
- 用户访问内存地址
- 该地址对应的物理页不在内存，而在磁盘

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



同步异常（续）

- **终止aborts**
  - 非有意，由不可修正的致命错误造成
  - 例：非法指令、奇偶校验错、机器检查
  - 中止当前程序

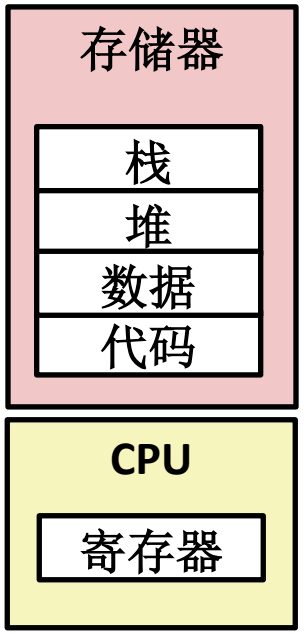


本课内容

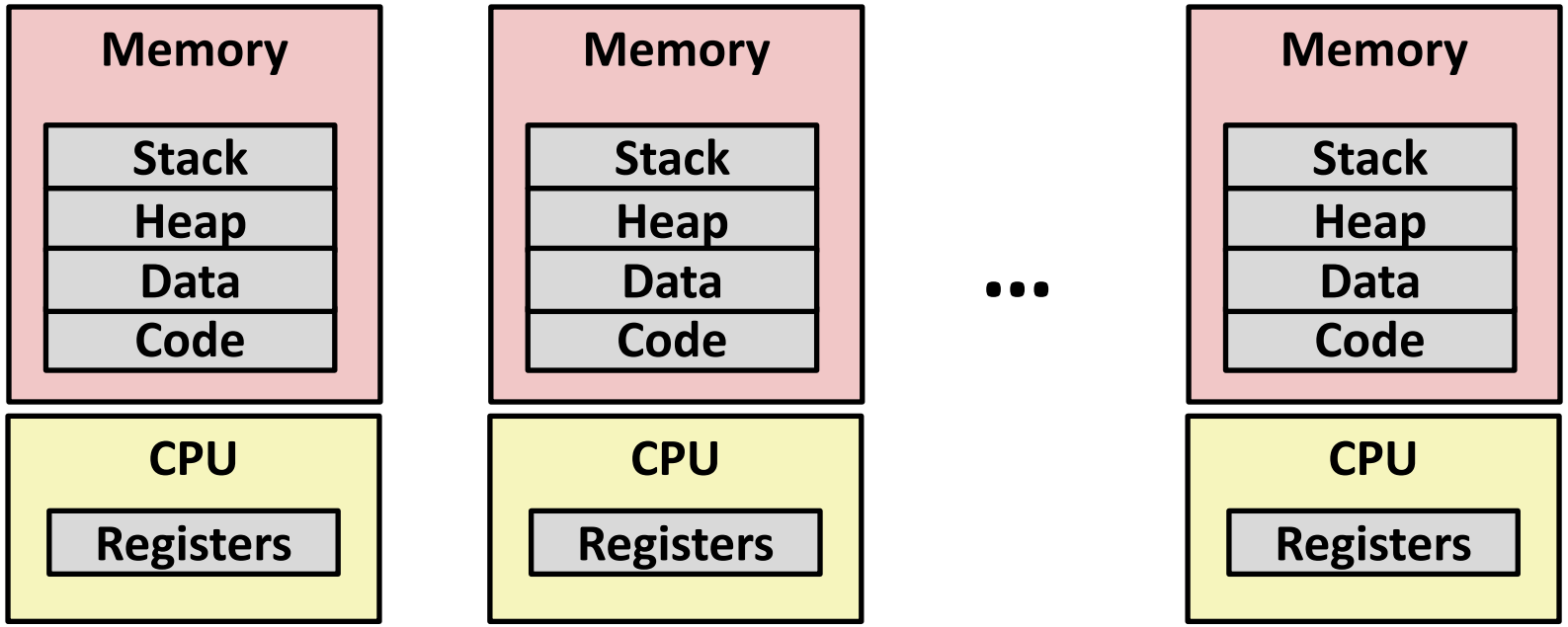
- 异常控制流Exceptional Control Flow
- 异常Exceptions
- **进程Processes**
- 进程控制Processes Control

进程process

- 定义： A *process* is an instance of a program. 一个执行中的程序
- 进程对应用程序做了两个关键抽象：
  - 逻辑控制流logical control flow
    - 每个运行中的程序都似乎独占着 CPU
    - 通过 OS 内核的上下文切换机制实现
  - 私有地址空间private address space
    - 每个运行中的程序都似乎独占着存储器
    - 通过 OS 内核的虚拟内存机制提供



多重处理：假象



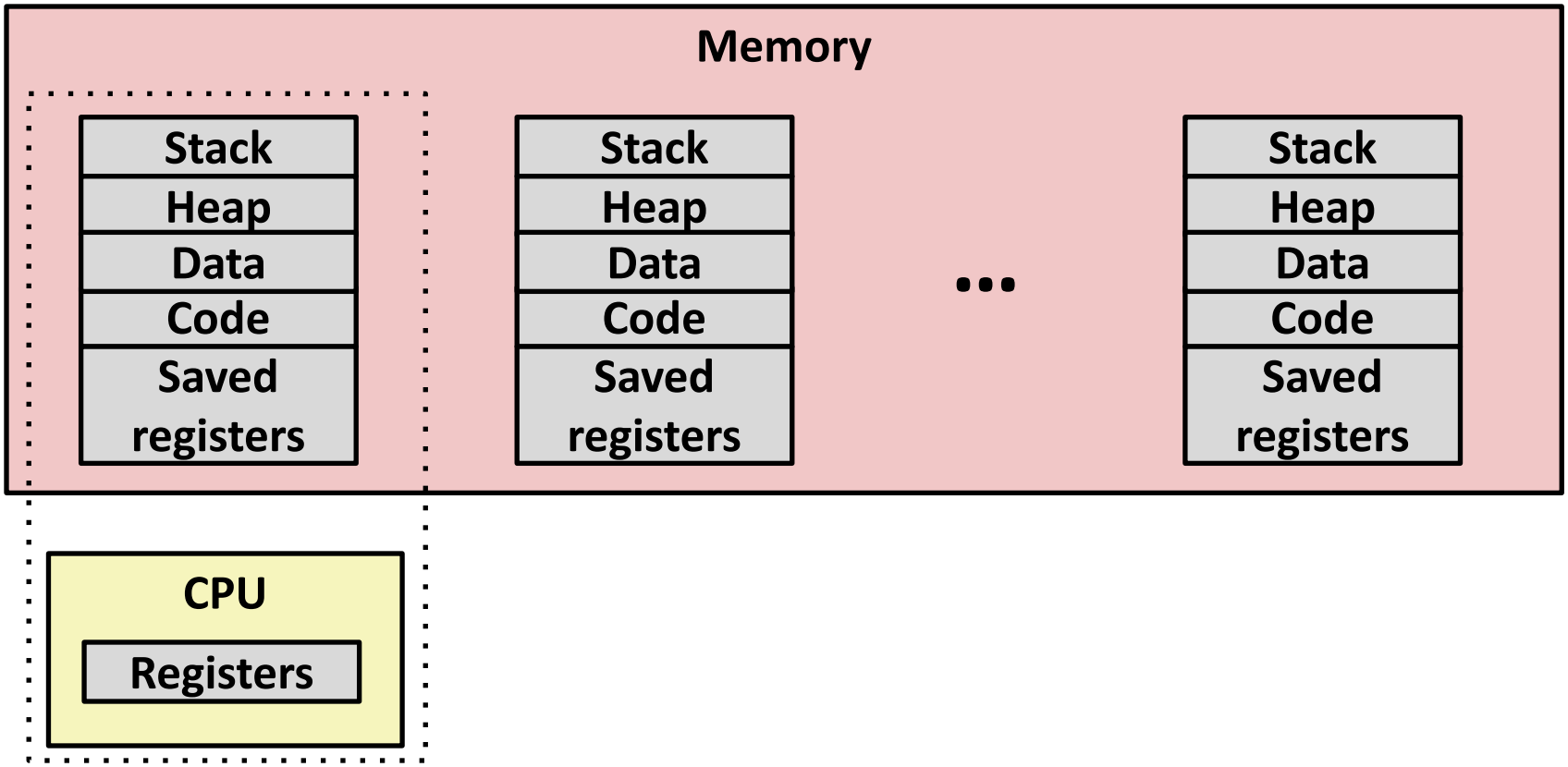
- 计算机可以同时运行很多进程
  - 单个用户（或多个用户）的各种应用程序：
    - Web 浏览器、email 客户端、编辑器、...
  - 后台任务：
    - 对网络的监测、对各种 I/O 设备的监测

多重处理示例

```
top - 21:49:19 up 1 day,  1:40,  1 user,  load average: 0.00, 0.00, 0.00
Tasks: 100 total,   1 running,  99 sleeping,   0 stopped,   0 zombie
%Cpu(s):  0.1 us,  0.1 sy,  0.0 ni, 99.8 id,  0.0 wa,  0.1 hi,  0.0 si,  0.0 st
MiB Mem :   3428.3 total,   2757.5 free,    187.6 used,    483.3 buff/cache
MiB Swap:      0.0 total,      0.0 free,      0.0 used.   2996.2 avail Mem

    PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
    928 root      20   0  138884  11208   9640 S   0.3   0.3   0:19.32 hostwatch
    950 root      20   0 2115128  27588  18396 S   0.3   0.8   4:57.65 hostguard
   2023 root      20   0  223024   2696   2108 S   0.3   0.1   0:32.26 wrapper
      1 root      20   0  103184  11444   8764 S   0.0   0.3   0:01.88 systemd
      2 root      20   0       0      0      0 S   0.0   0.0   0:00.66 kthreadd
      3 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 rcu_gp
      4 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 rcu_par_gp
      6 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 kworker/0:0H-kblockd
      8 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 mm_percpu_wq
      9 root      20   0       0      0      0 S   0.0   0.0   0:00.00 ksoftirqd/0
     10 root      20   0       0      0      0 I   0.0   0.0   0:01.90 rcu_sched
     11 root      20   0       0      0      0 I   0.0   0.0   0:00.00 rcu_bh
     12 root      rt   0       0      0      0 S   0.0   0.0   0:00.06 migration/0
     13 root      20   0       0      0      0 S   0.0   0.0   0:00.00 cpuhp/0
     14 root      20   0       0      0      0 S   0.0   0.0   0:00.00 cpuhp/1
     15 root      rt   0       0      0      0 S   0.0   0.0   0:00.06 migration/1
     16 root      20   0       0      0      0 S   0.0   0.0   0:00.01 ksoftirqd/1
     18 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 kworker/1:0H-kblockd
```

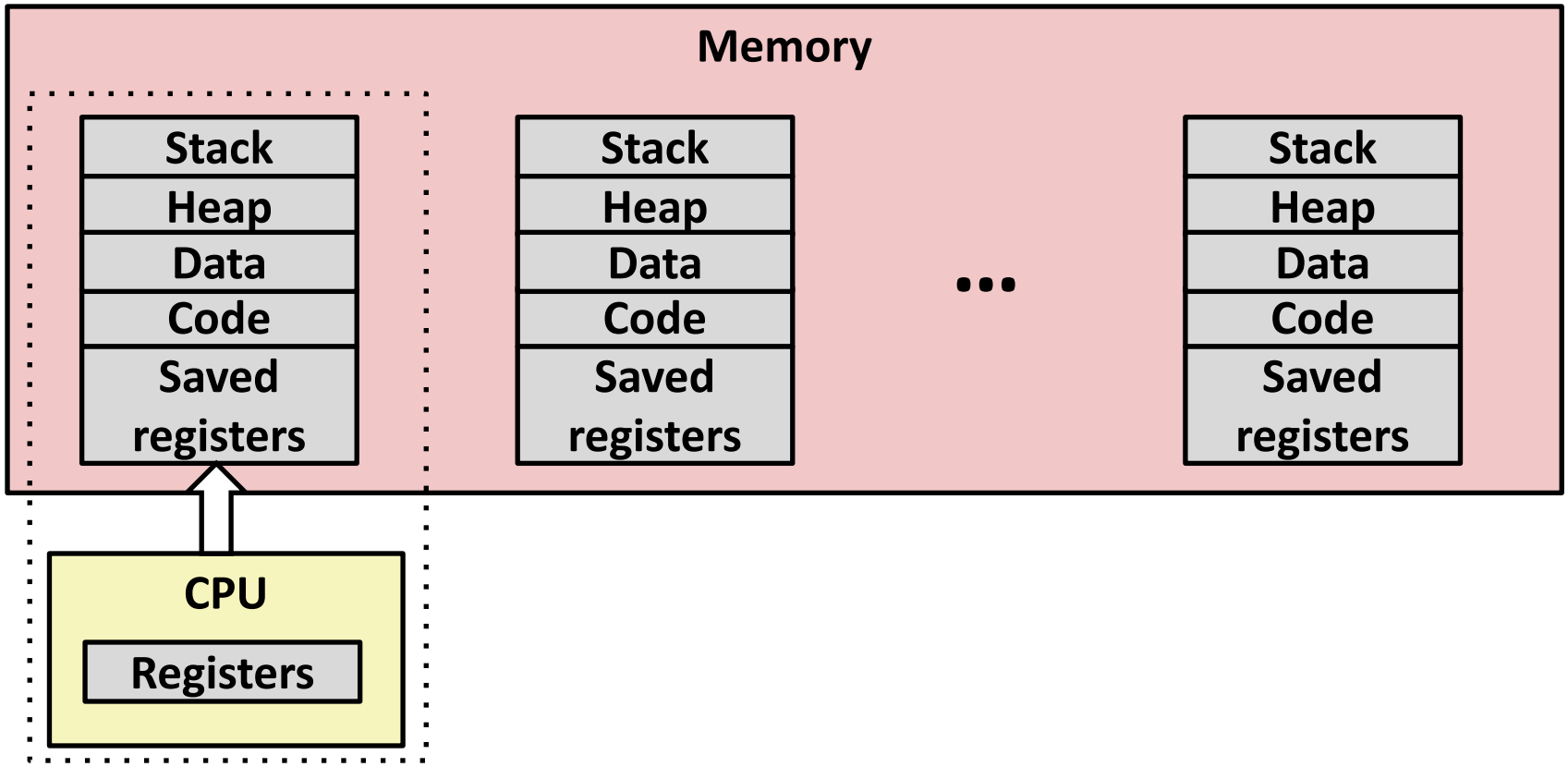
- 在 openEuler 上运行 top 程序
  - 有 100 个进程，正在运行的有 1 个，休眠 99 个
  - 以进程 ID（PID）进行区分

多重处理：真相



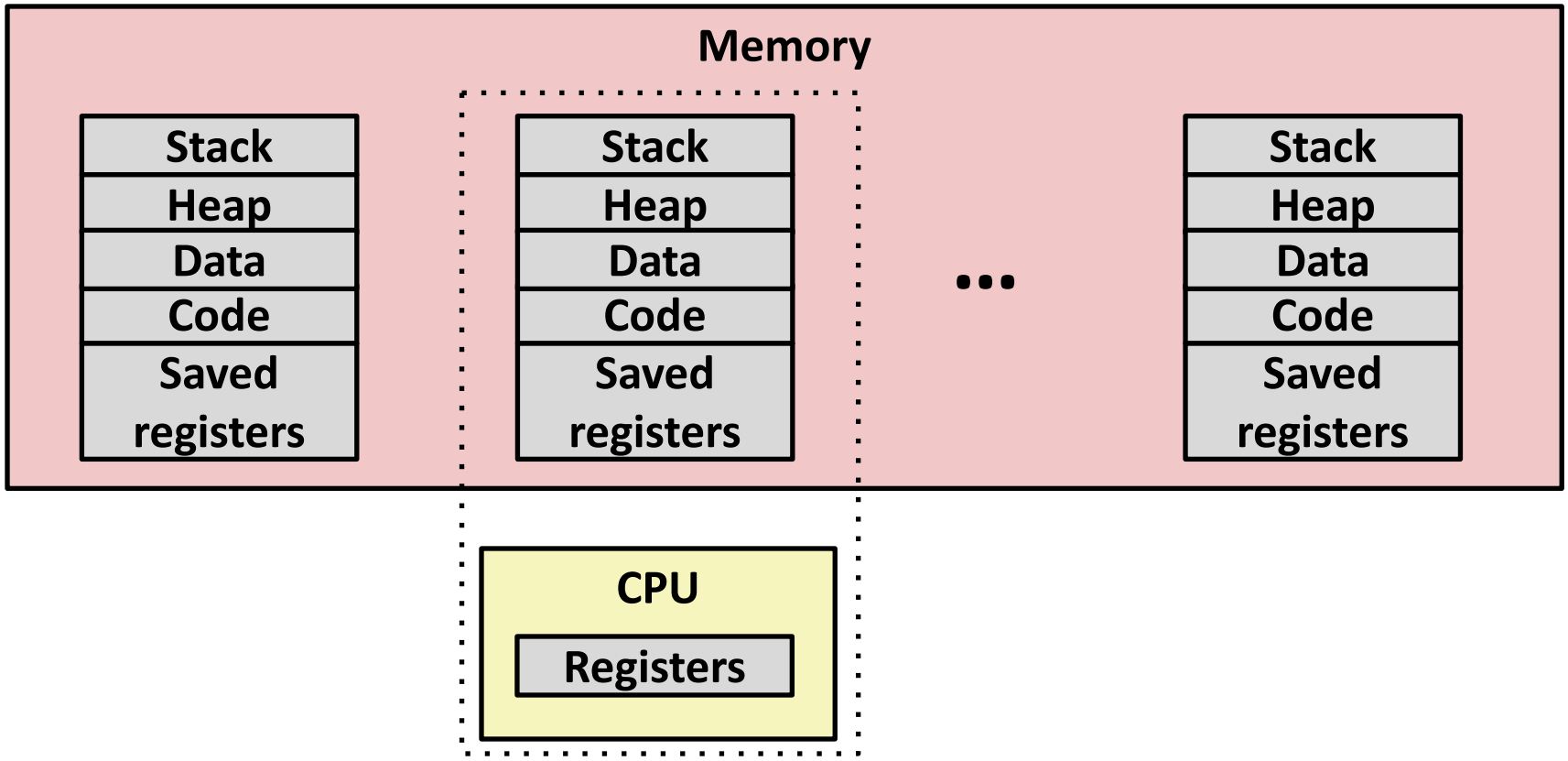
- 单处理器在并发地执行多个进程
  - 进程交错执行（多任务multitasking）
  - 各地址空间由虚拟内存系统进行管理（后文介绍）
  - 当前未执行的进程，其寄存器值保存在内存中

多重处理：真相（续）



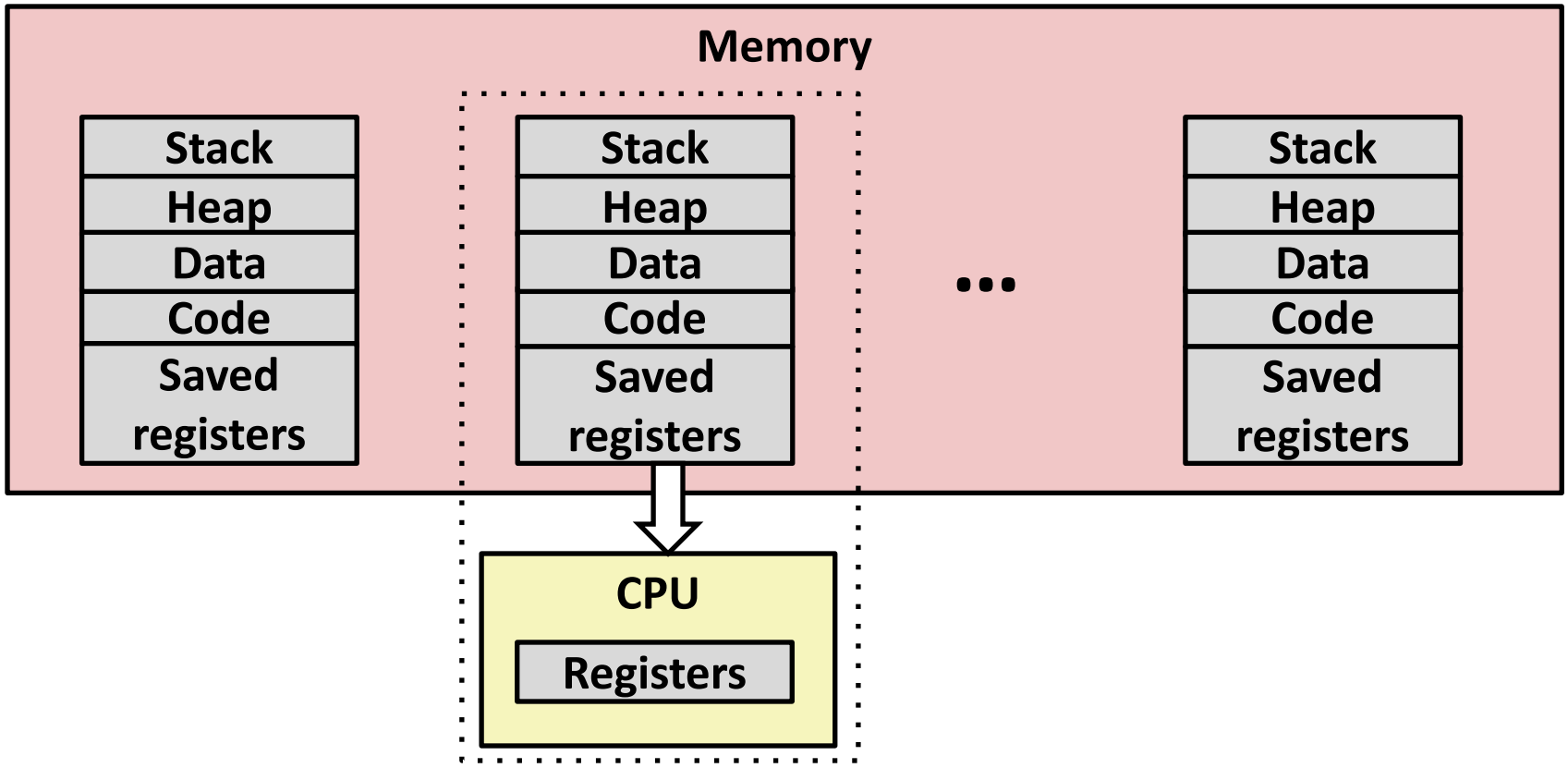
- 将寄存器的当前值保存至内存

多重处理：真相（续）



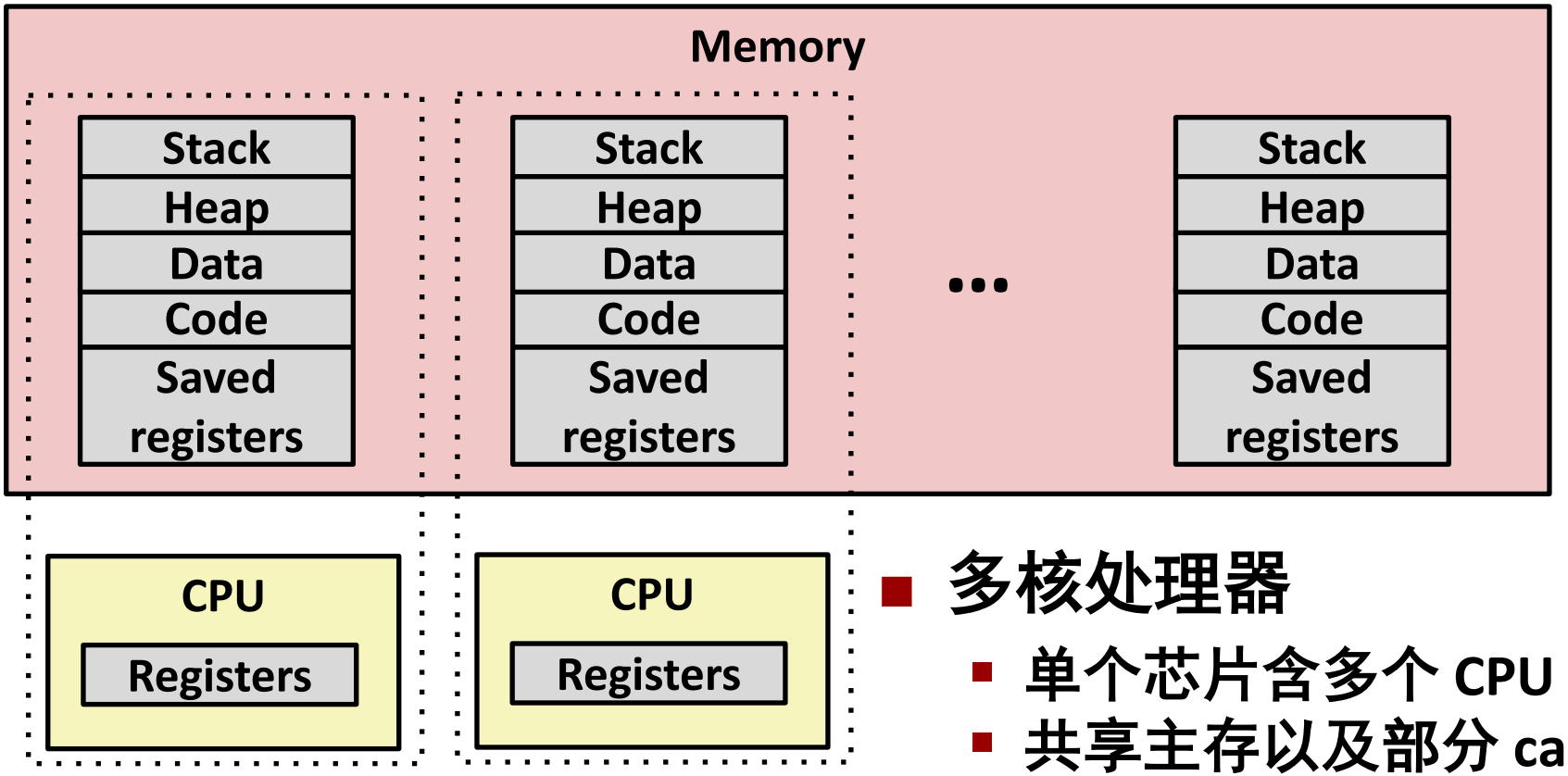
- 调度下一个进程执行

多重处理：真相（续）



- 加载保存的各寄存器值，并切换地址空间（上下文切换）

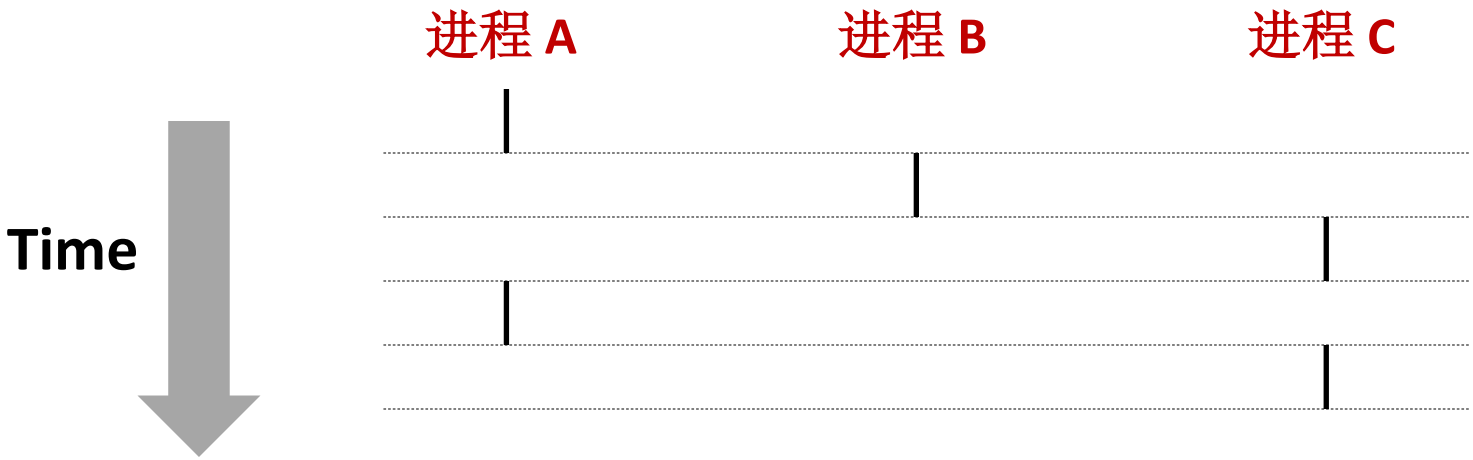
多重处理：真相（现代计算机）



- 多核处理器
  - 单个芯片含多个 CPU
  - 共享主存以及部分 cache
  - 每个 CPU 核可单独执行一个进程
    - OS 内核负责处理器核的调度

并发进程concurrent processes

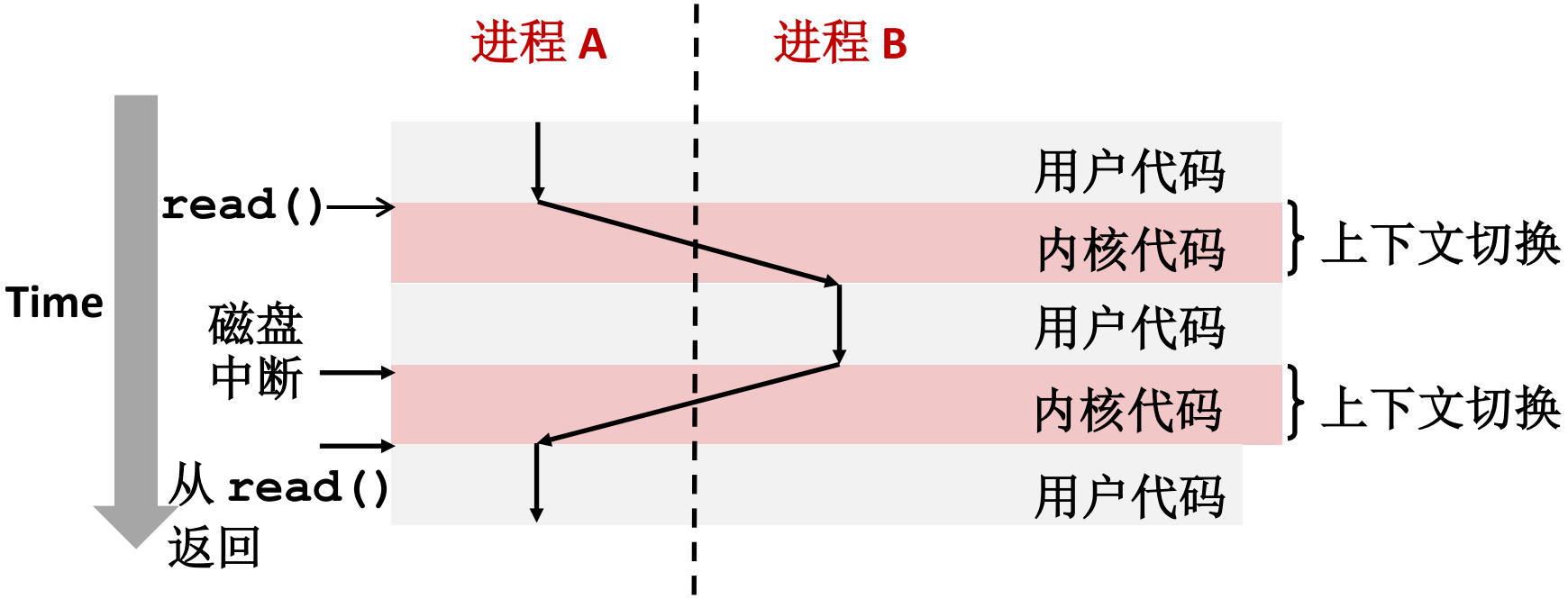
- 每个进程都是一个逻辑控制流
- 如果两个进程的逻辑流在执行期间有交错，则称这两个进程是并发concurrent的
- 否则称它们是顺序sequential的



单核 CPU 示例：A 和 B，A 和 C 均为并发关系
B 和 C 是顺序关系

上下文切换Context Switching

- 进程由一段常驻内存的，共用的 OS 代码（内核）进行管理
  - 注意：内核不是作为一个单独进程，而是作为现有进程的一部分而运行
- 控制流通过上下文切换由一个进程传递至另一进程



本课内容

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

系统调用错误处理

- 当调用 Linux 系统级函数发生错误时，通常返回 -1，并设置全局整型变量 errno 以标示出错原因
- 硬性规则：
  - 每次调用系统级函数，都必须检查其返回状态
  - 少数返回值为空void的函数除外
- 例：

```
if ((pid = fork()) < 0) {
    fprintf(stderr, "fork error: %s\n",
        strerror(errno));
    exit(1);
}
```

  - strerror()将错误编号转换为相应字符串
  - 但会导致代码冗长

报错函数

- 通过定义如下的错误报告函数，能够在一定程度上简化代码：

```
void unix_error(char *msg)      /* Unix 风格的错误 */
{
    fprintf(stderr, "%s: %s\n", msg,
            strerror(errno));
    exit(1);
}
```

则可将对 fork() 的调用缩减至 2 行：

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

错误处理包装函数error-handling wrapper

- 通过使用错误处理包装函数能够进一步简化代码：

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

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

则可将对 fork()的调用缩减至 1 行：

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

获取进程 ID

- 每个进程都有唯一的非零正整数标识，称作进程 ID（PID）
- `pid_t getpid(void)`
  - 返回本进程的 PID
- `pid_t getppid(void)`
  - 返回本进程的父进程的 PID
- `pid_t` 在 Linux 系统中由 `int` 型 `typedef` 得到

创建和终止进程

- 从程序员的角度，可以认为进程总是处于如下 3 种状态之一：
  - 运行态running
    - 进程正在被 CPU 执行（执行态）
    - 正在等待被 CPU 执行，且终将被 OS 内核调度（就绪态）
  - 停止（暂停、挂起）态stopped/paused/suspended
    - 进程的执行被挂起，在收到新的信号之前不会被调度
  - 终止态terminated
    - 进程永远地停止了，但仍占据着资源

创建进程

- 父进程通过调用 `fork` 函数创建一个新的，处于运行态的子进程
- `int fork(void)`
  - 对子进程返回 0，对父进程返回子进程的 PID
  - 新创建的子进程几乎与父进程完全相同：
    - 子进程其虚拟地址空间在内容和地址布局上与父进程一致，但是相互独立的拷贝（包括代码段、数据段、堆、 共享库以及用户栈）
    - 子进程还得到父进程调用 `fork` 时所有打开文件的文件描述符的拷贝（即父进程打开的文件，子进程也能访问）
    - 子进程与父进程的 PID 不同
- `fork` 函数：调用 1 次，返回 2 次

fork 示例

```
父进程
  |
  |-- fork() 调用
  |
  +--> 父进程继续运行，fork()返回子进程PID
  |
  +--> 子进程开始运行，fork()返回0
```

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

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

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

fork.c

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

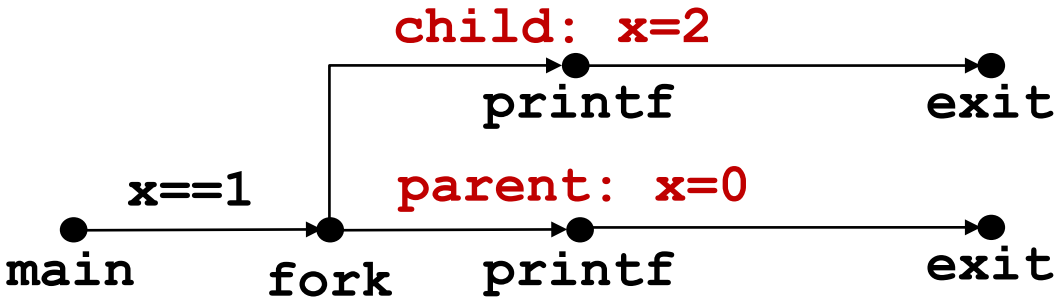
- 调用 1 次，返回 2 次
- 并发执行
  - 无法预测父进程与子进程的执行顺序
- 相同但独立的地址空间
  - fork 返回时，x 在父进程和子进程中都为 1
  - x 为局部变量
  - 接下来，父进程和子进程对 x 所做的任何改变都是独立的
- 共享打开的文件
  - stdout 文件在父子进程是相同的

用进程图描述 fork

- 进程图是刻画并发程序中语句偏序的有用工具
  - 每个顶点 a 对应一条语句的执行
  - 有向边 a → b 表示语句 a 发生在语句 b 之前
  - 边上可以标记变量的当前值
  - printf 语句对应的顶点可以标记输出的内容
  - 每张图从一个没有入边的顶点开始
- 图中顶点的任意一个拓扑排序对应着程序中语句的一个可能的执行顺序
  - 所有顶点的总排序，每条边都是从左到右

解释进程图

- 原进程图：

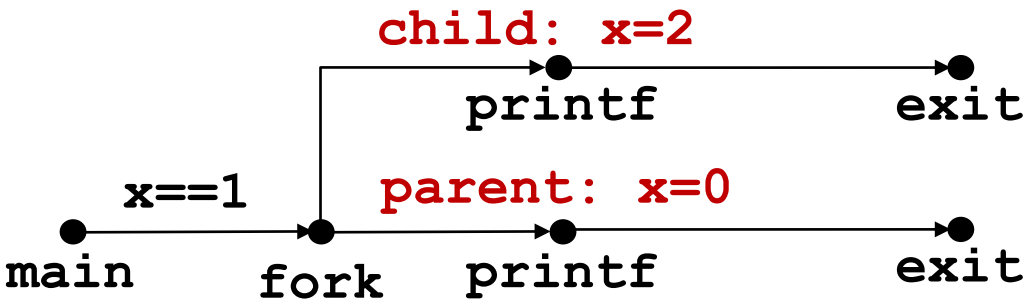



```c
int main()
{
    pid_t pid;
    int x = 1;

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

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

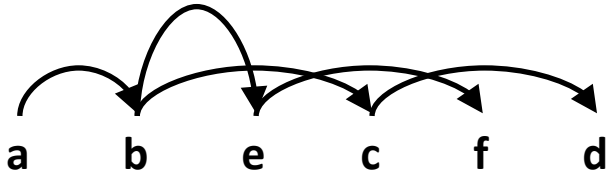
fork.c

解释进程图

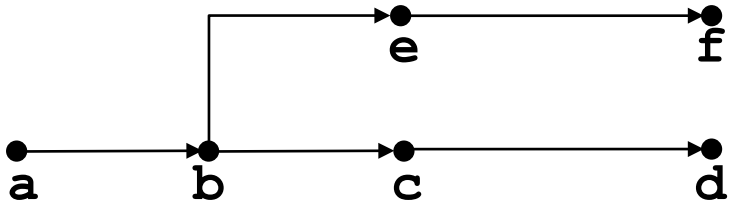
- **原进程图：**



可行的全序排列



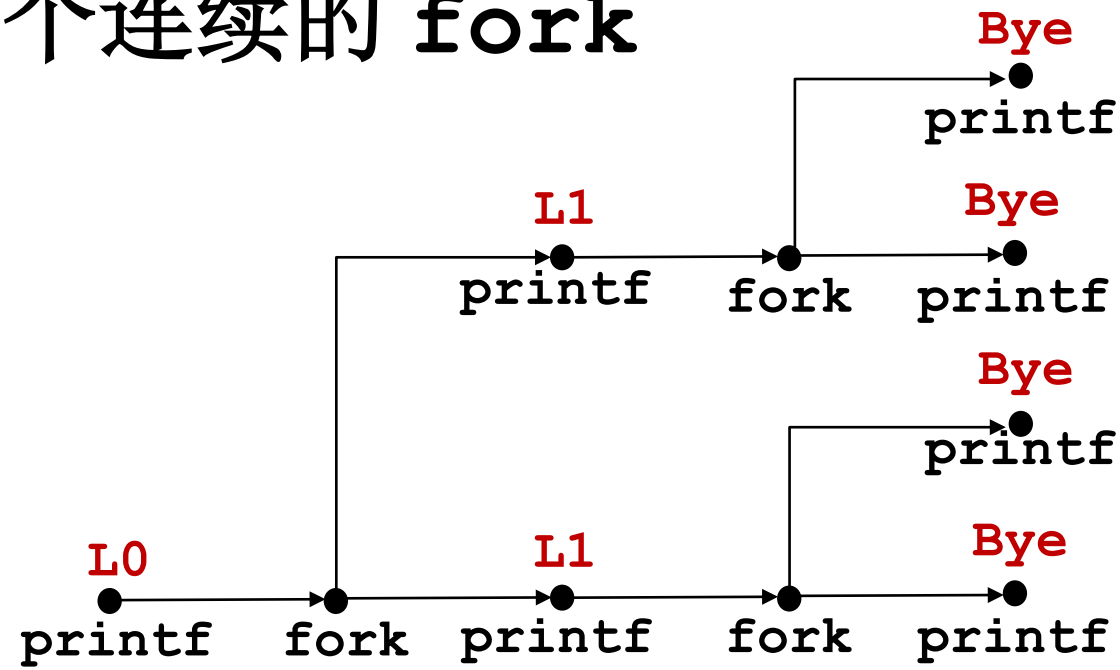
- **重新标记后的进程图：**



不可行的全序排列

fork 示例：两个连续的 fork

```
void fork2()
{
    printf("L0\n");
    Fork();
    printf("L1\n");
    Fork();
    printf("Bye\n");
}
```
forks.c



可能的输出：

L0
L1
Bye
Bye
L1
Bye
Bye

不可能的输出：

L0
Bye
L1
Bye
L1
Bye
Bye

fork 示例：父进程中 fork 的嵌套调用

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

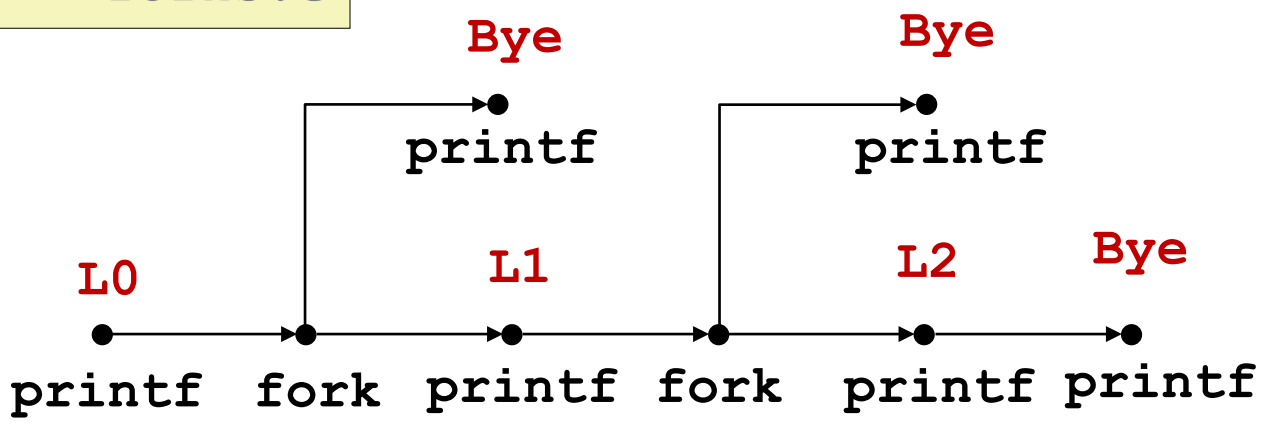
forks.c

可能的输出：

L0
L1
Bye
Bye
L2
Bye

不可能的输出：

L0
Bye
L1
Bye
Bye
L2



终止进程

- 导致进程终止的原因有 3 种：
  - 接收到一个信号（如 SIGKILL、SIGTERM 等），该信号的默认行为是终止进程
  - `main()` 返回
  - 调用 `_exit` 或 `exit` 函数
- `void exit(int status)`
  - 以退出状态 `status` 终止本进程
  - 通常以返回状态为 0 表示正常终止，非零表示出错
- `exit` 函数：调用 1 次，返回 0 次（不返回）

回收子进程

- 思想——为什么要回收？——与 `fork` 创建相反！
  - 进程终止后，仍然在消耗系统资源
    - 例：退出状态、OS 的各种表（占用内存）
  - 称为僵尸zombie进程
  - 僵尸进程占用内存资源、打开的 I/O 资源等
- 回收reaping
  - 由父进程执行对终止态的子进程的回收（通过 `wait` 或 `waitpid` 系统调用）
  - 父进程接收到子进程的退出状态信息
  - 内核接着删除僵尸子进程，从系统中删除其所有痕迹

回收子进程（续）

- 若父进程未回收子进程，如何处理？
  - 如果父进程没有回收它的子进程就终止了，内核将安排 `init` 进程作为养父回收它们
  - `init` 进程在系统启动时创建，其 PID 为 1，从不终止
  - `init` 进程是所有进程的祖先
  - 只有长时间运行的进程需要显式地对其子进程 进行回收
    - 例：shell、服务器程序