

# 第8章 异常控制流 I——异常和进程

Exceptional Control Flow——Exceptions and Processes

# 主要内容

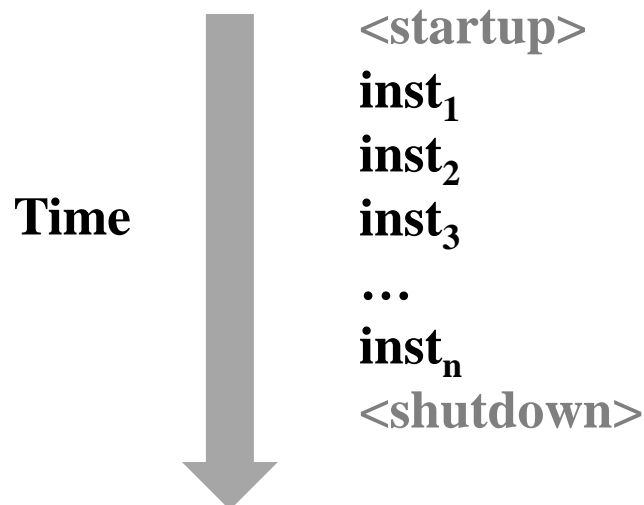
- 异常控制流(Exceptional Control Flow)
- 异常(Exceptions)
- 进程(Processes)
- 进程控制(Processes Control)

# 控制流(Control Flow)

## ■ 处理器只做一件事

- 处理器从加电到断电，处理器只是简单地读取和执行一个指令序列(一次执行一条指令)
- 这个指令序列就是处理器的控制流 ( *control flow* or *flow of control* )

*Physical control flow*



# 改变控制流(Altering the Control Flow)

## ■ 改变控制流的两种机制:

- 跳转和分支(Jumps and branches)
- 调用和返回(Call and return)

能够对(由程序变量表示的)**程序状态**的变化做出反应

## ■ 对**系统状态**的变化做出反应

- 磁盘或网络适配器的数据到达
- 除零错误
- 用户的键盘输入( Ctrl-C )
- 系统定时器超时

*上述系统变化不能用程序变量表示*

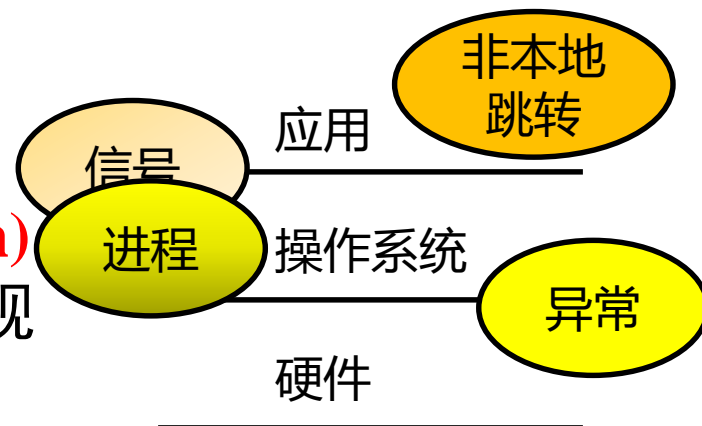
## ■ 现代计算机系统需要针对“控制流发生突变”的情况做出反应，称为“异常控制流”机制

# 异常控制流(Exceptional Control Flow)

- 发生在计算机系统的所有层次
- 低层机制(硬件层)
  - 1. **异常 (Exceptions)**
    - 硬件检测到的事件会触发控制转移到异常处理程序
    - 操作系统和硬件共同实现

- 高层机制

- 2. **进程切换(Process context switch)**
  - 通过操作系统和硬件定时器实现
- 3. **信号(Signals)**
  - 操作系统实现
- 4. **非本地跳转(Nonlocal jumps): setjmp() and longjmp()**
  - C运行库实现

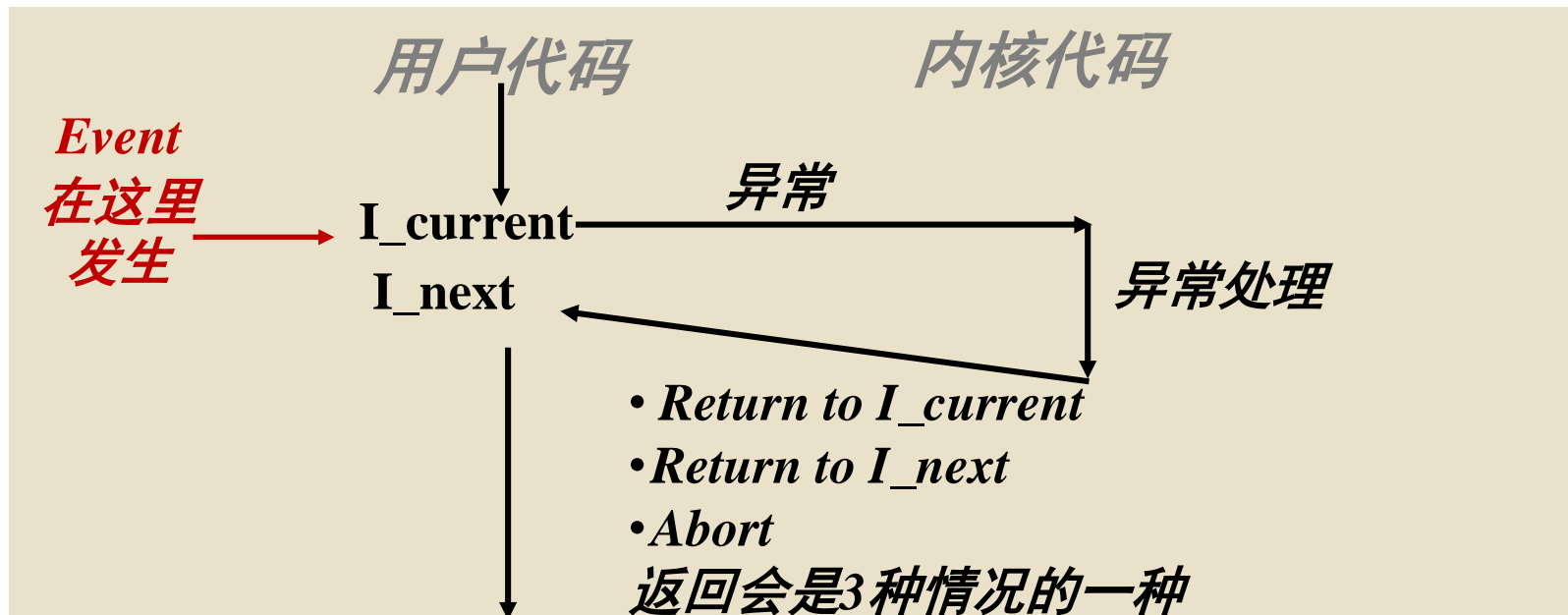


# 主要内容

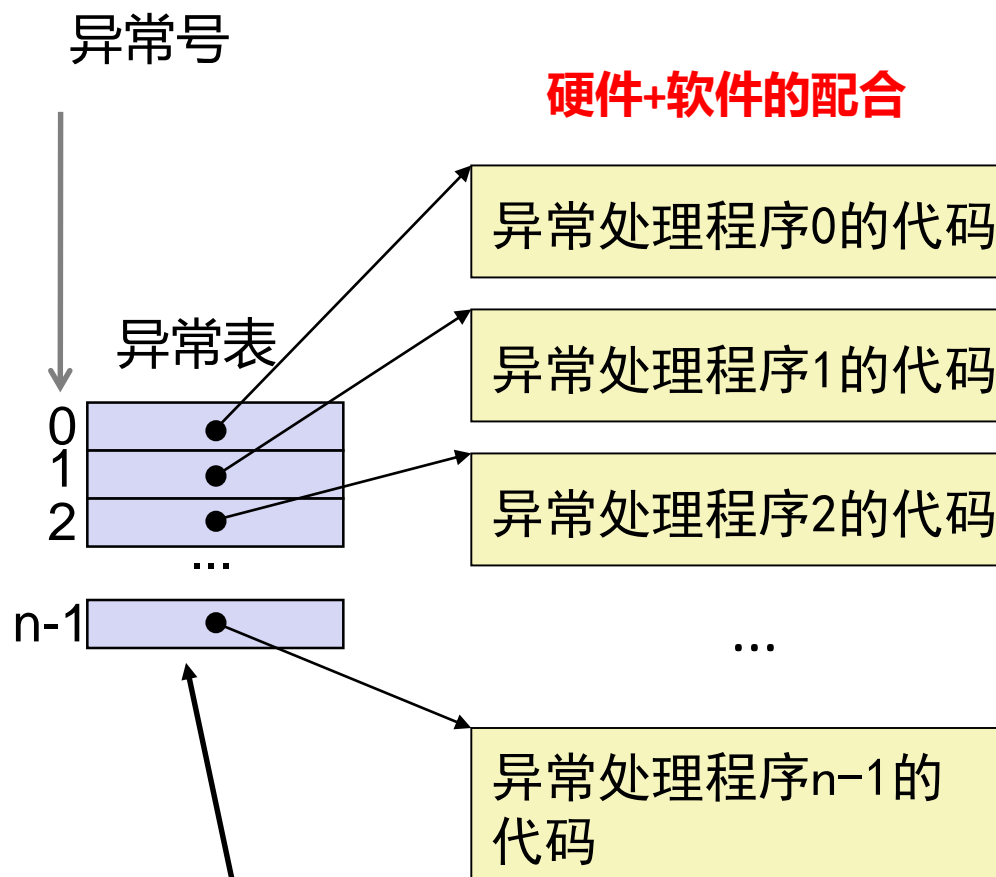
- 异常控制流(Exceptional Control Flow)
- 异常(Exceptions)
- 进程(Processes)
- 进程控制(Processes Control)

# 异常(Exceptions)

- **异常**是指为响应某个事件将控制权转移到操作系统内核中的情况
  - 内核指操作系统常驻内存的部分
  - 事件示例：被零除、算术运算溢出、缺页、I/O请求完成、键盘输入



# 异常处理



- 每种类型的事件有一个唯一的异常号(Exception numbers ) $k$
- 异常号 $k$ 是到异常表的索引(又名中断向量)
- 任何时候异常 $k$ 发生, 则异常 $k$ 的处理程序立刻被调用

异常表 (中断向量表) 是一张跳转表, 表目 $k$ 包含异常 $k$ 的处理程序代码的地址



# 异常的分类

类别	原因	异步 / 同步	返回行为
中断	来自 I/O 设备的信号	异步	总是返回到下一条指令
陷阱	有意的异常	同步	总是返回到下一条指令
故障	潜在可恢复的错误	同步	可能返回到当前指令
终止	不可恢复的错误	同步	不会返回

**异步：**指处理器之外的事件引起的，和处理器执行的指令没有时间上的关系

**同步：**指处理器中指令执行引起的异常

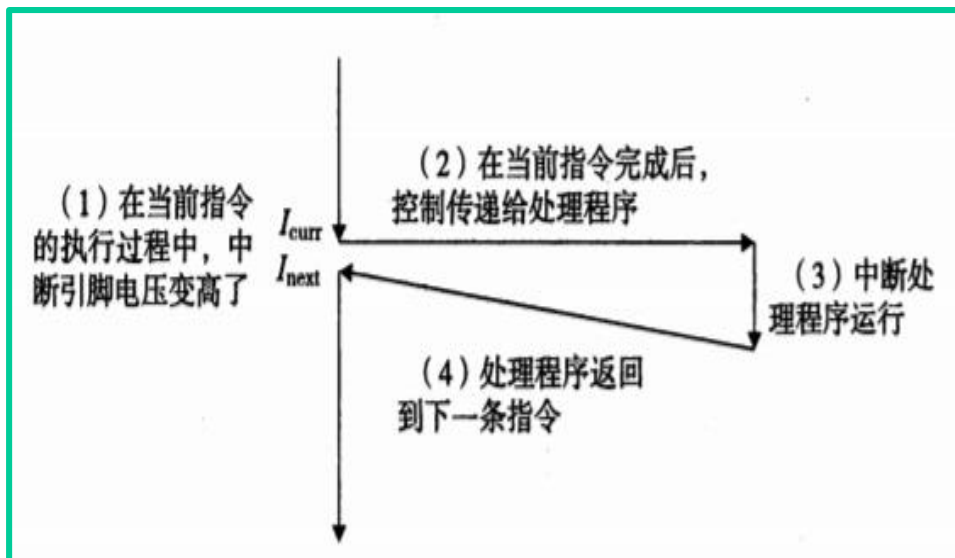
**陷阱 (Trap)** 是一种特殊的异常 (Exception)，它是由程序显式触发的，通常用于请求操作系统的服务或处理某些特定事件。陷阱也被称为 软件中断 (Software Interrupt) 或 系统调用 (System Call)。

# 异步异常 Asynchronous Exceptions (中断 Interrupts)

- 处理器外部I/O设备引起
  - 由处理器的中断引脚指示
  - 中断处理程序返回到下一条指令处

## ■ Examples:

- 时钟中断(Timer interrupt)
  - 定时器芯片每隔几毫秒触发一次中断
  - 内核从用户程序取回控制权
- 外部设备的I/O中断(I/O interrupt from external device)
  - 键盘上敲击一个 Ctrl-C
  - 网络数据包到达
  - 磁盘数据的到达

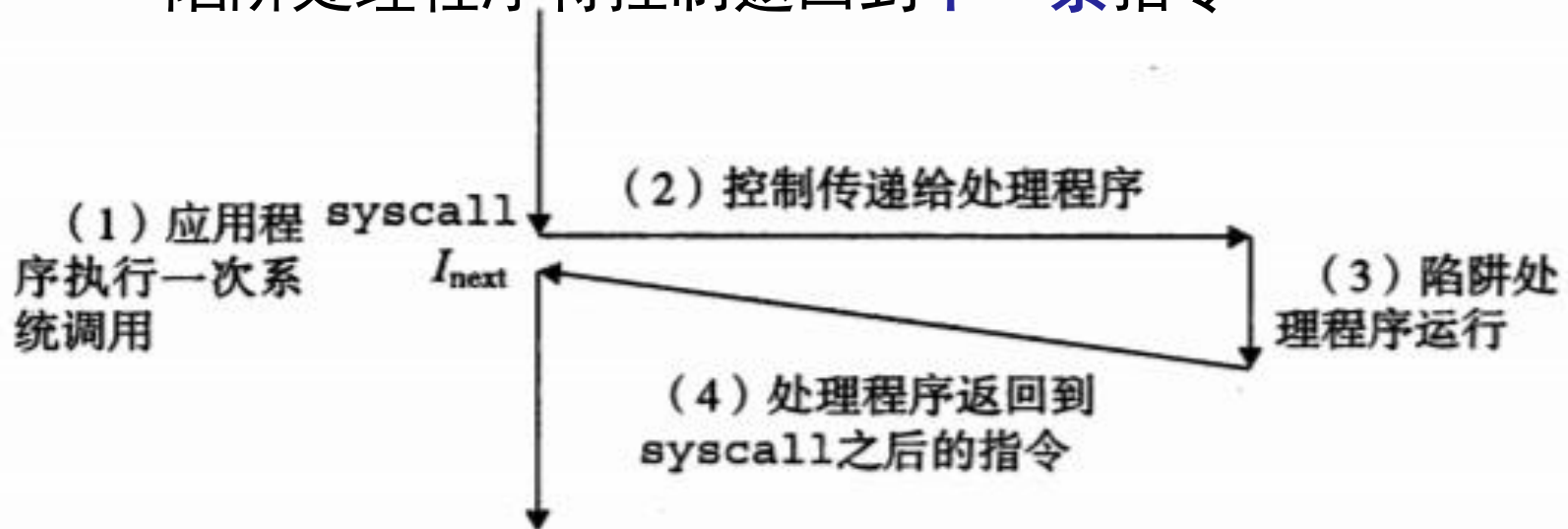


# 同步异常(Synchronous Exceptions)

## ■ 执行指令产生的结果:

### ■ 陷阱(Traps)

- 有意的，执行指令的结果
- Examples: 系统调用(System Call), 用户程序和内核之间的一个接口
- 陷阱处理程序将控制返回到**下一条**指令

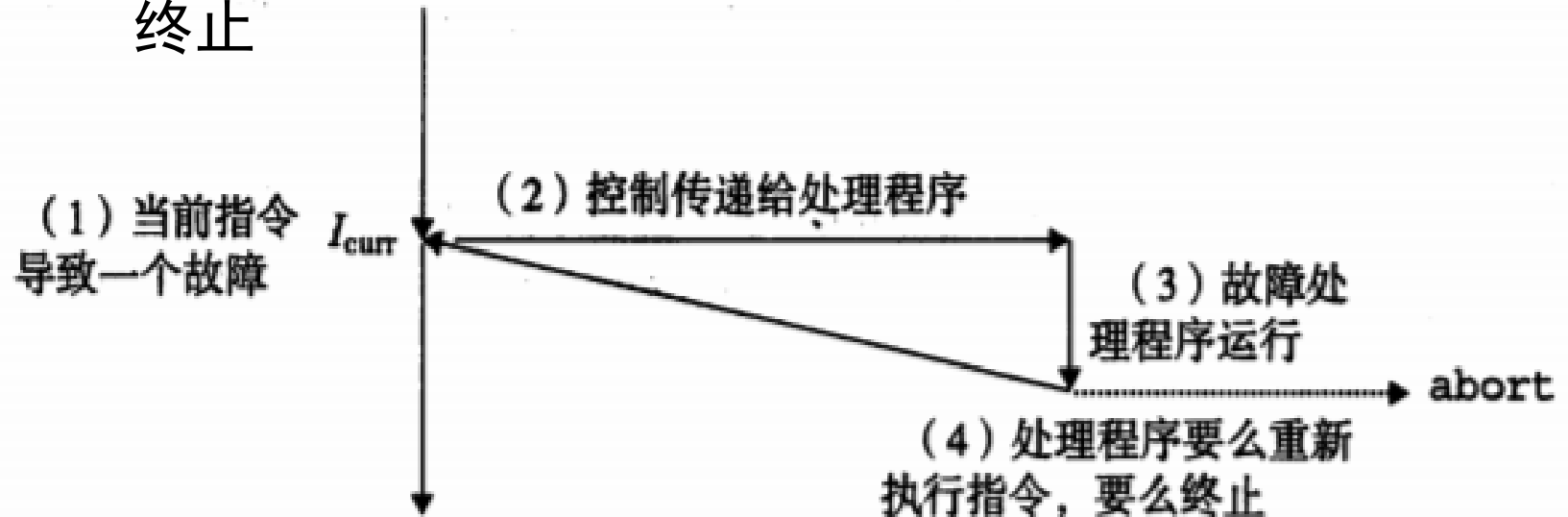


# 同步异常(Synchronous Exceptions)

## ■ 执行指令产生的结果:

### ■ 故障(Faults)

- 不是有意的，但可能被修复
- Examples: 缺页(可恢复), 保护故障(protection faults, 不可恢复), 浮点异常(floating point exceptions)
- 处理程序要么重新执行引起故障的指令(已修复), 要么终止

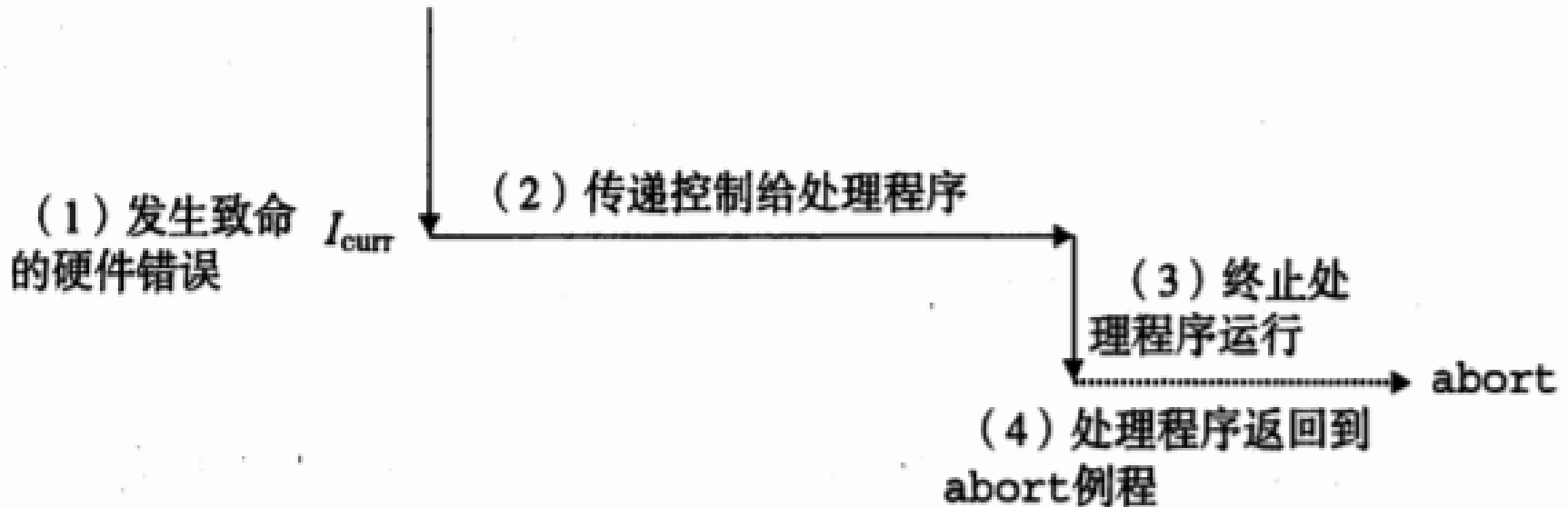


# 同步异常(Synchronous Exceptions)

## ■ 执行指令产生的结果:

### ■ **终止 (Aborts)**

- 非故意，不可恢复的致命错误造成
- Examples: 非法指令, 奇偶校验错误(parity error), 机器检查(machine check)
- 中止当前程序



# 系统调用(System Call)

每个x86-64系统调用有一个唯一的ID号

■ Examples(Linux系统调用):

<i>Number</i>	<i>Name</i>	<i>Description</i>
0	read	Read file
1	write	Write file
2	open	Open file
3	close	Close file
4	stat	Get info about file
57	fork	Create process
59	execve	Execute a program
60	_exit	Terminate process
62	kill	Send signal to process

# 系统调用的例子: 打开文件

- 用户调用函数: `open(filename, options)`
- 调用 `__open` 函数, 该函数使用系统调用指令 `syscall`

0000000000e5d70 <\_\_open>:

...

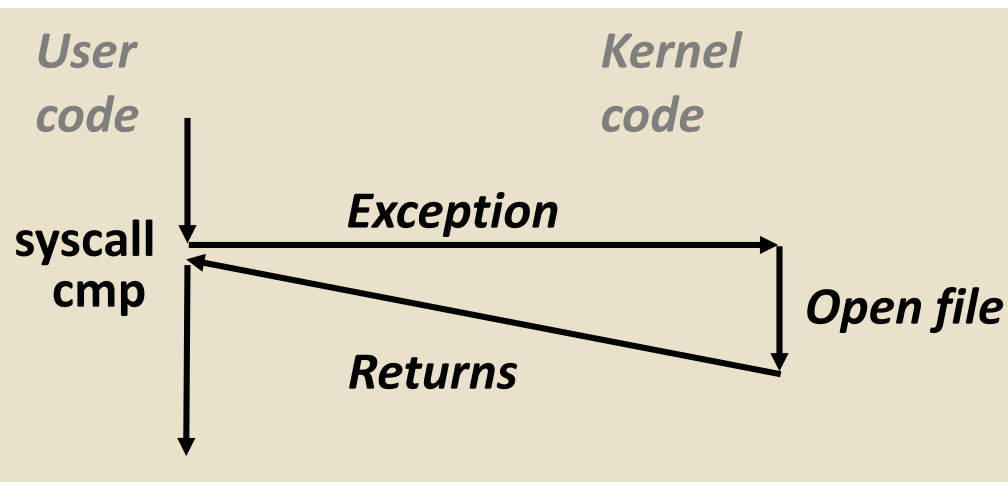
e5d79: b8 02 00 00 00    `mov $0x2,%eax # open is syscall #2`

e5d7e: 0f 05            `syscall        # Return value in %rax`

e5d80: 48 3d 01 f0 ff ff   `cmp $0xffffffffffff001,%rax`

...

e5dfa: c3                `retq`



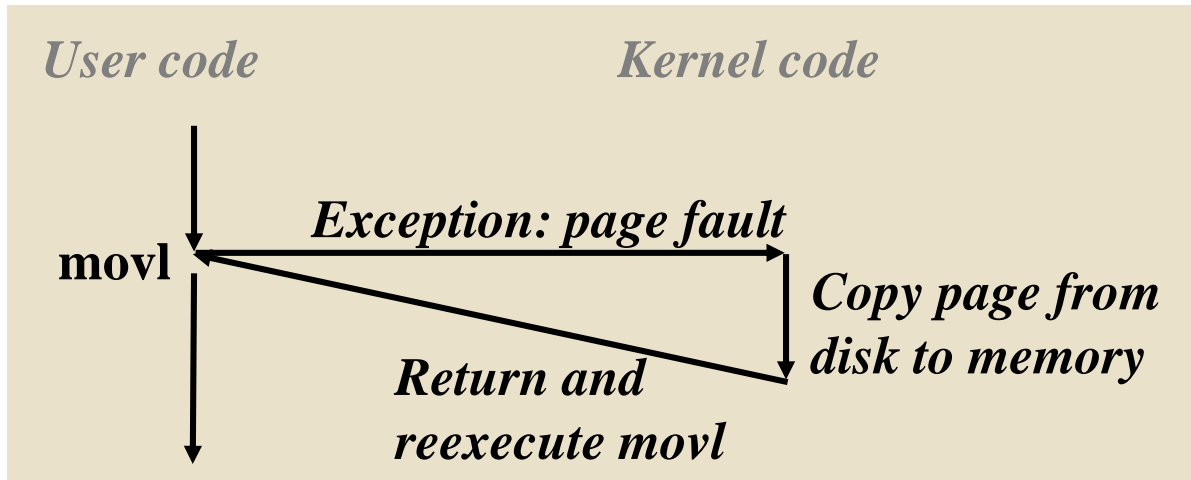
- `%rax` 包含系统调用号
- 其他参数(至多6个)依次在 `%rdi, %rsi, %rdx, %rcx, %r8, %r9`
- 返回值在 `%rax`
- 负数返回值表明发生了错误, 对应于负的 `errno`

# Fault Example:缺页故障(Page Fault)

- 用户写内存地址(虚拟地址)
- 该地址对应的物理页不在内存，在磁盘中

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

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

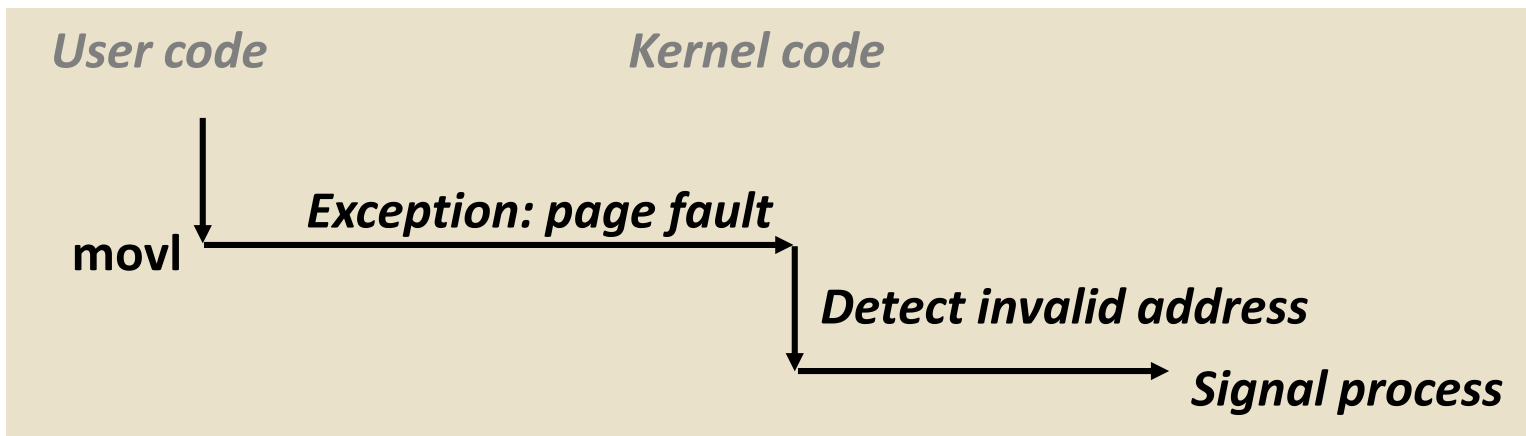




# Fault Example:非法内存引用

```
int a[1000];  
main ()  
{  
    a[500000] = 13;  
}
```

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



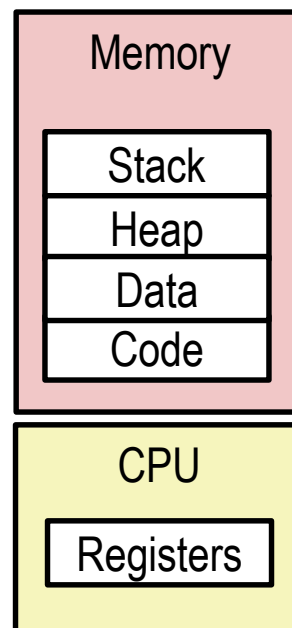
- OS发送 **SIGSEGV** 信号给用户进程(不尝试恢复)
- 用户进程以“段错误”(segmentation fault)退出

# 主要内容

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

# 进程(Processes)

- 定义: A **process** is an instance of a running program.  
一个执行中程序的实例
  - 计算机科学最深刻的概念之一
  - 不同于“程序”或“处理器”
- 进程提供给应用程序两个关键抽象
  - **逻辑控制流 (Logical control flow)**
    - 每个程序似乎独占地使用CPU
    - 通过OS内核的上下文切换机制提供
  - **私有地址空间 (Private address space)**
    - 每个程序似乎独占地使用内存系统
    - OS内核的虚拟内存机制提供



# “进程” 的 “上下文”

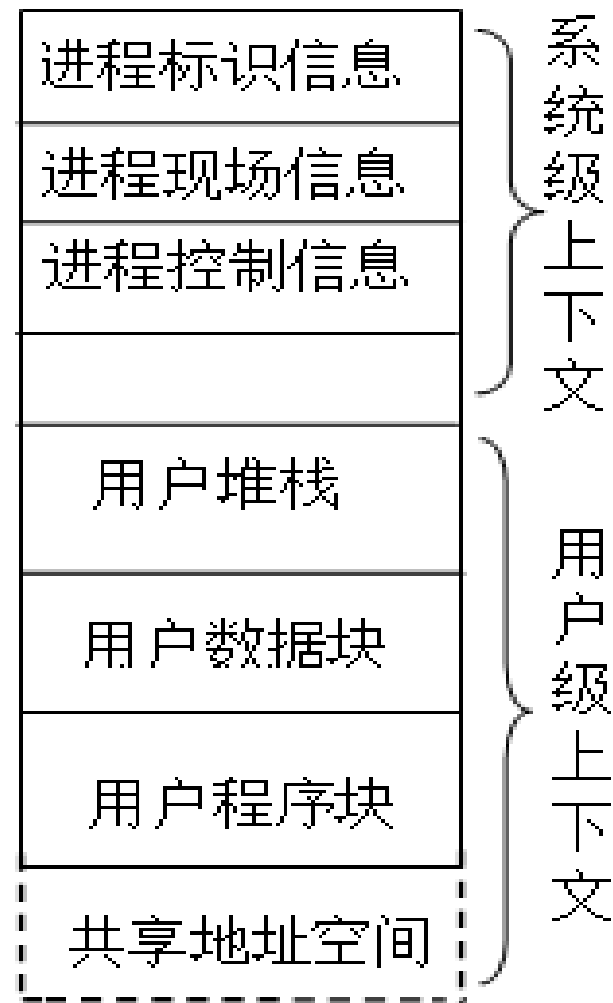
- 进程的物理实体（代码和数据等）和支持进程运行的环境合称为**进程的上下文**。具体有：

1) 由进程的程序块、数据块、运行时的堆和用户栈（两者通称为用户堆栈）等组成的用户空间信息被称为**用户级上下文**；

2) 由进程标识信息、进程现场信息、进程控制信息和系统内核栈等组成的内核空间信息被称为**系统级上下文**；

3) 处理器中各寄存器的内容被称为**寄存器上下文**（也称**硬件上下文**），即进程的现场信息。在进行进程上下文切换时，操作系统把换下进程的寄存器上下文保存到系统级上下文中的现场信息位置。

- 用户级上下文地址空间和系统级上下文地址空间一起构成了一个**进程的整个存储器映像**



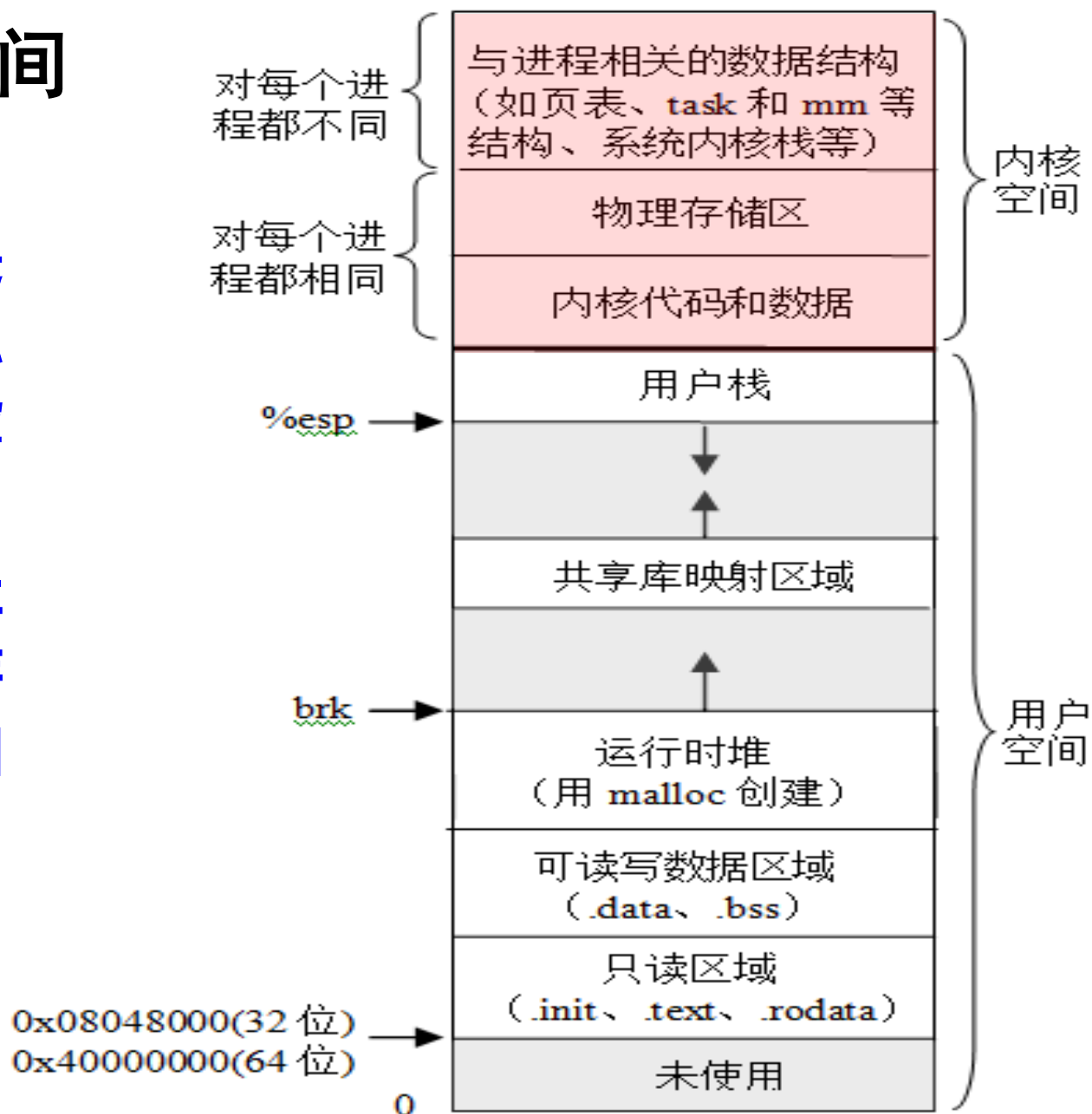
## 进程的存储器映像

# 见教材511页

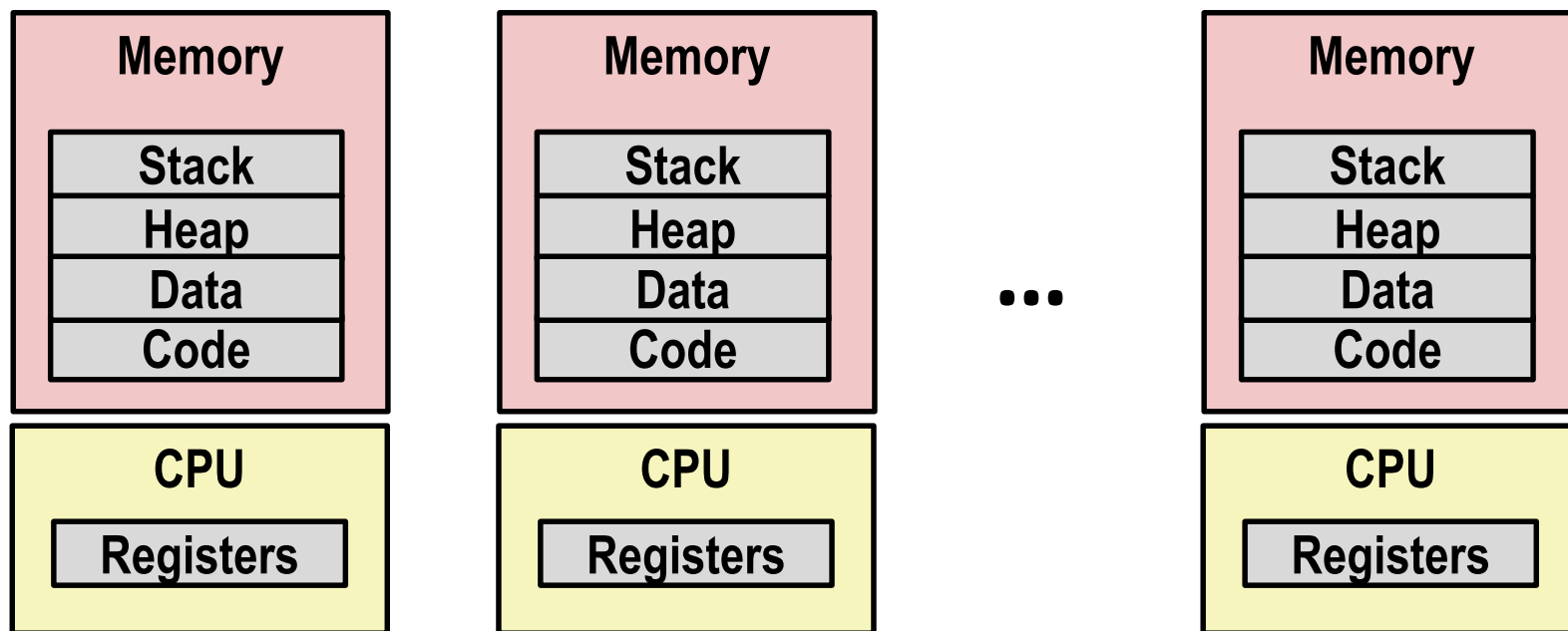
- 内核为每个进程维持一个上下文（context）。上下文就是内核重新启动一个被抢占的进程所需的状态。它由一些对象的值组成，这些对象包括通用目的寄存器、浮点寄存器、程序计数器、用户栈、状态寄存器、内核栈和各种内核数据结构，比如描述地址空间的页表、包含有关当前进程信息的进程表，以及包含进程已打开文件的信息的文件表。

# 进程的地址空间

- Linux平台下，每个（用户）进程具有独立的私有地址空间（虚拟地址空间）
- 每个进程的地址空间划分（即存储映像）布局相同（如右图）



# 多重处理:假象(Multiprocessing: The Illusion☺)



- 计算机同时运行许多进程
  - 单/多用户的应用程序
    - Web 浏览器、email客户端、编辑器 ...
  - 后台任务(Background tasks)
    - 监测网络和 I/O 设备

# 多重处理的例子

The image shows a Linux terminal window on the left and a Windows Task Manager window on the right, illustrating system resource usage.

**Terminal Window (top command output):**

```
top - 20:58:14 up 5:23, 1 us
Tasks: 185 total, 2 running,
%Cpu(s): 42.3 us, 23.7 sy, 33.
KiB Mem : 2036780 total, 11
KiB Swap: 2094076 total, 204
```

PID	USER	PR	NI	VIRT
14136	linda	20	0	2097040
8663	linda	20	0	1249912
14018	linda	20	0	2238548
7845	root	20	0	487720
8730	linda	20	0	1004620
33	root	20	0	0
6	root	20	0	0
194	root	0	-20	0
7665	root	20	0	277628
7796	root	20	0	283672
7894	nobody	20	0	54608
9048	linda	20	0	599748
13894	linda	20	0	43540
1	root	20	0	119632
2	root	20	0	0

**Windows Task Manager (Performance tab):**

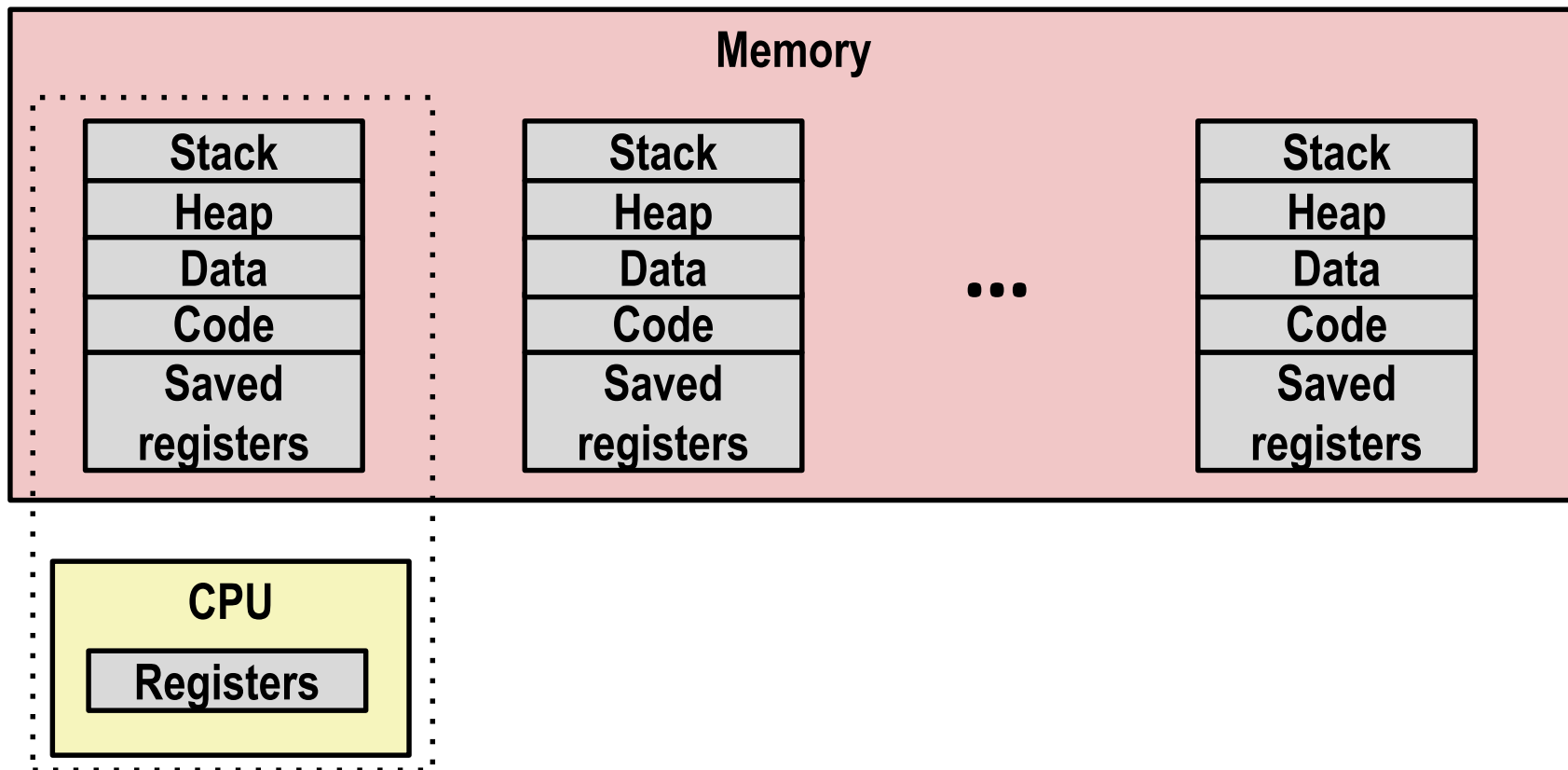
名称	CPU	内存	磁盘	网络
应用 (8)				
> Foxmail 7.2 (32 位)	0%	15.9 MB	0.1 MB/秒	0.1 Mbps
> Microsoft PowerPoint (32 位)...	0%	145.7 MB	0 MB/秒	0 Mbps
> Microsoft Word (32 位)	0%	40.5 MB	0 MB/秒	0 Mbps
> VMware Workstation (32 位)...	0%	58.5 MB	0 MB/秒	0 Mbps
> Windows 资源管理器 (3)	0.1%	59.8 MB	0 MB/秒	0 Mbps
> 画图	0%	45.4 MB	0 MB/秒	0 Mbps
> 任务管理器	0.2%	18.7 MB	0 MB/秒	0 Mbps
> 搜狗高速浏览器 (32 位)	0%	34.4 MB	0 MB/秒	0 Mbps
后台进程 (101)				
> 64-bit Synaptics Pointing Enh...	0%	2.1 MB	0 MB/秒	0 Mbps
> Active Protection System	0%	0.2 MB	0 MB/秒	0 Mbps
> Adobe Acrobat Update Servi...	0%	2.3 MB	0 MB/秒	0 Mbps
> Application Frame Host	0%	12.4 MB	0 MB/秒	0 Mbps

## ■ 运行“top”

- 有185个进程，正在运行的有2个
- 以进程ID (PID)相区分

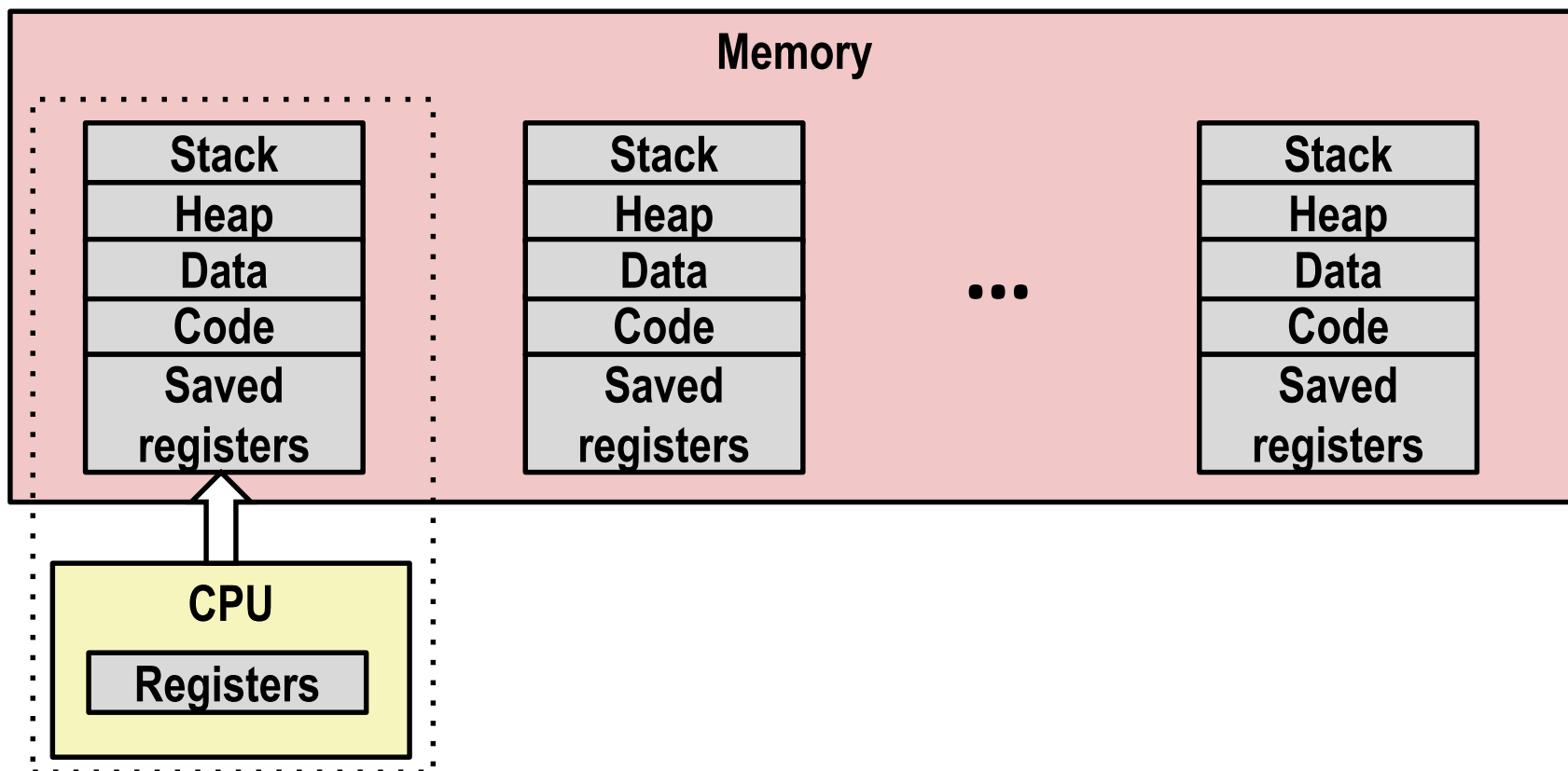


# 多重处理的真相



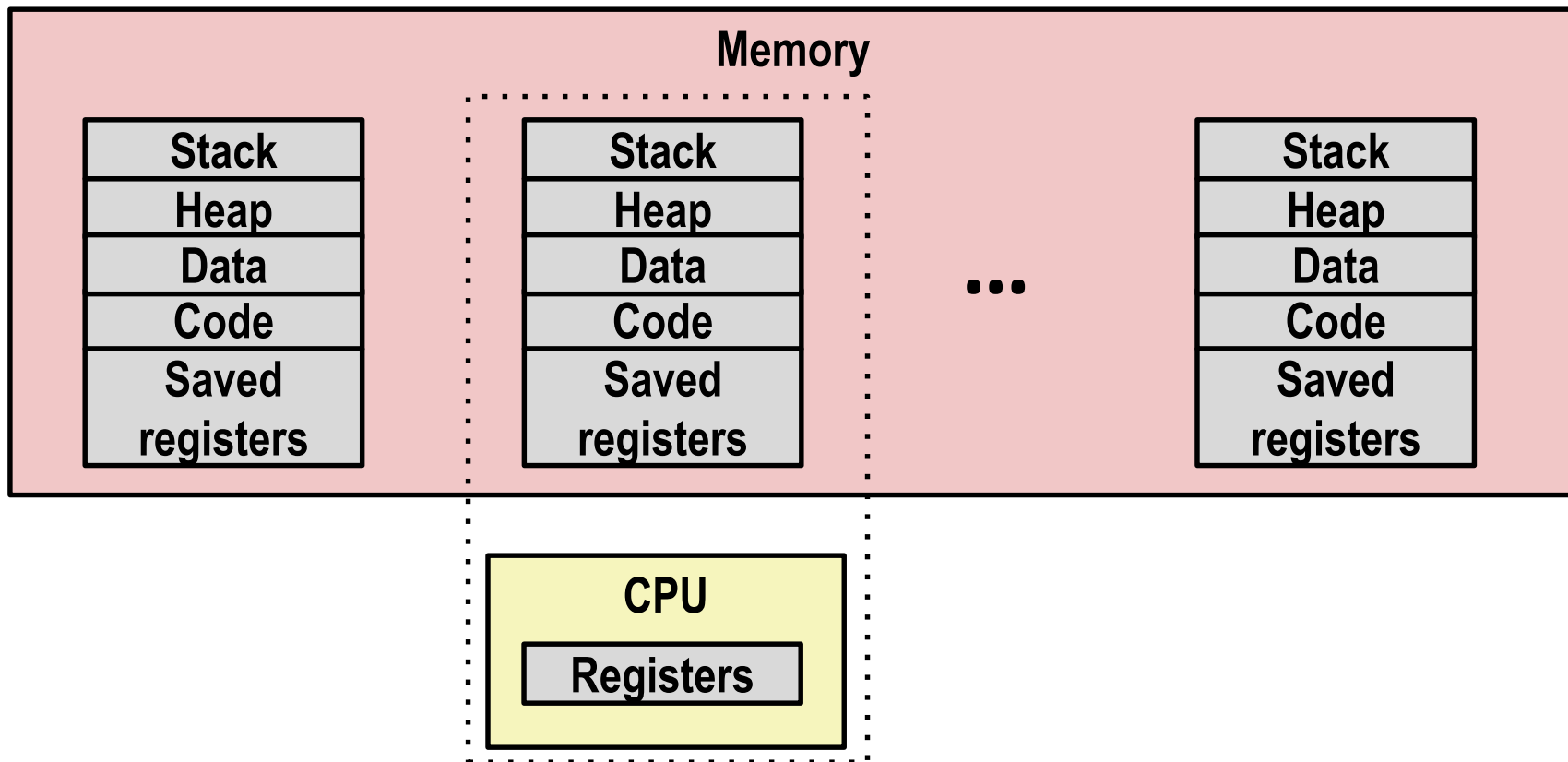
- **单处理器在并发地执行多个进程**
  - 进程交错执行(multitasking)
  - 地址空间由虚拟内存系统管理 (later in course)
  - 未执行进程的寄存器值保存在内存中

# 多重处理的真相



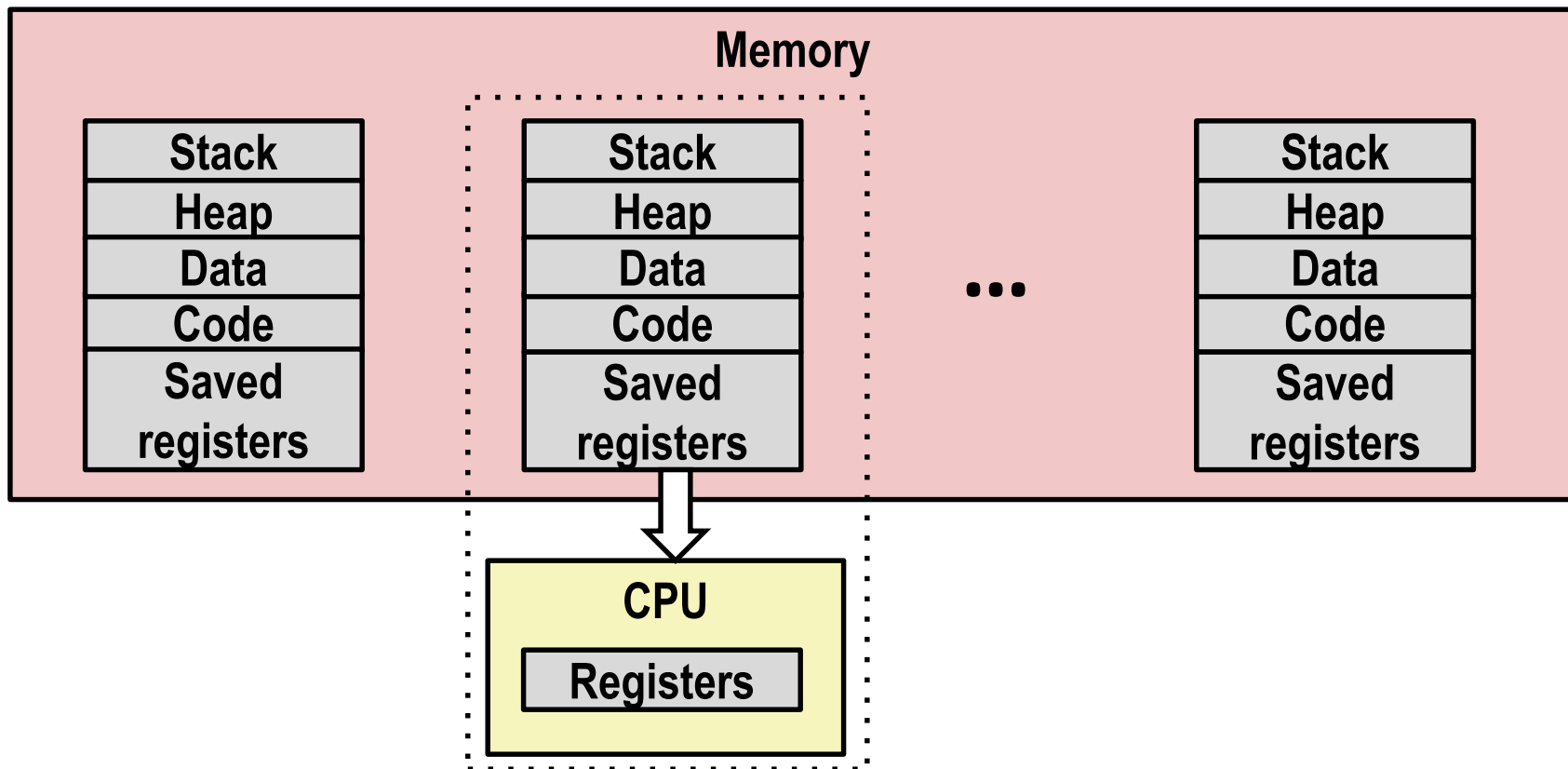
- 寄存器当前值保存到内存

# 多重处理的真相



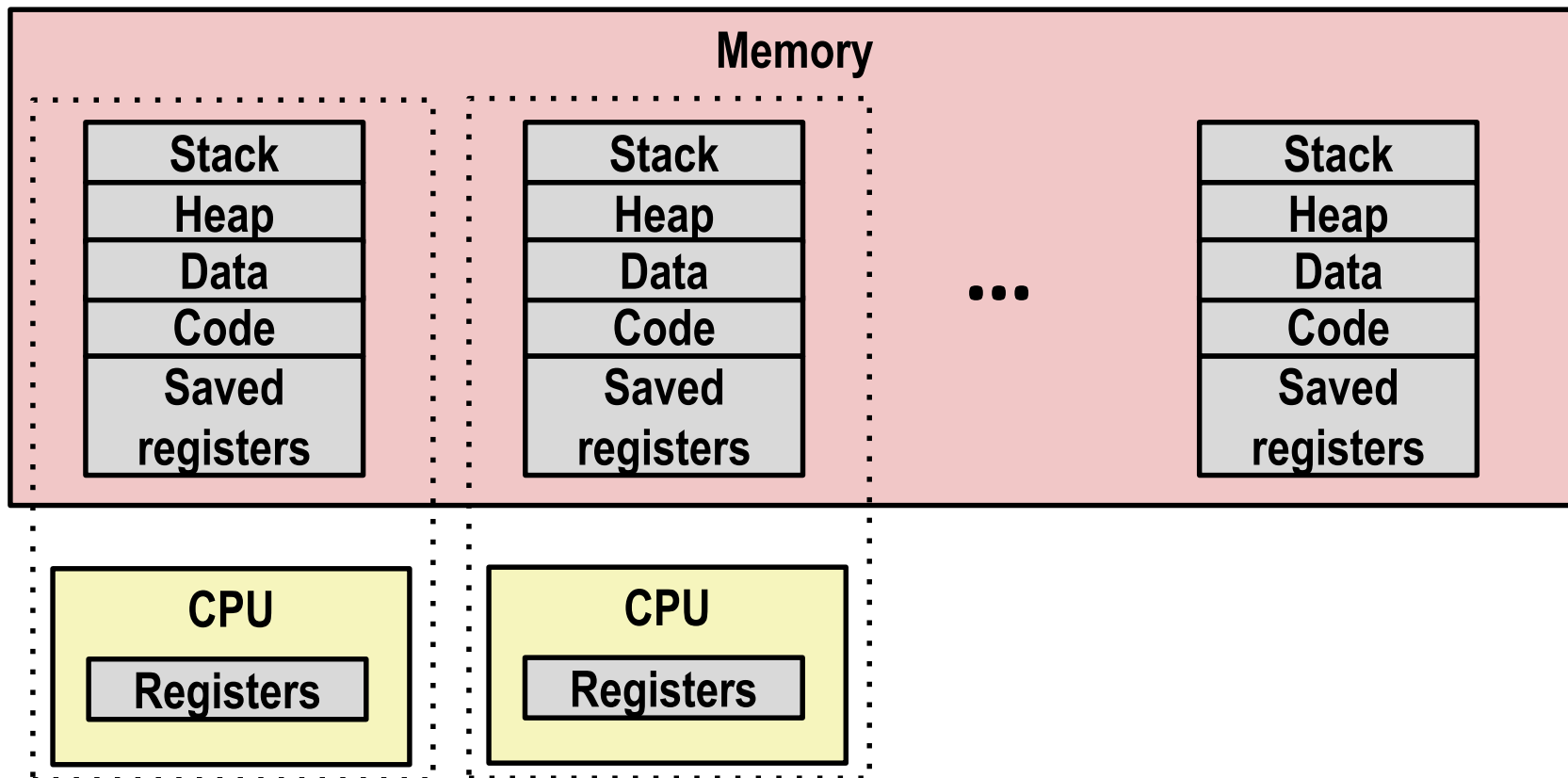
- 调度下一个进程执行

# 多重处理的真相



- 加载保存的寄存器组，并切换地址空间-上下文切换

# 多重处理的真相

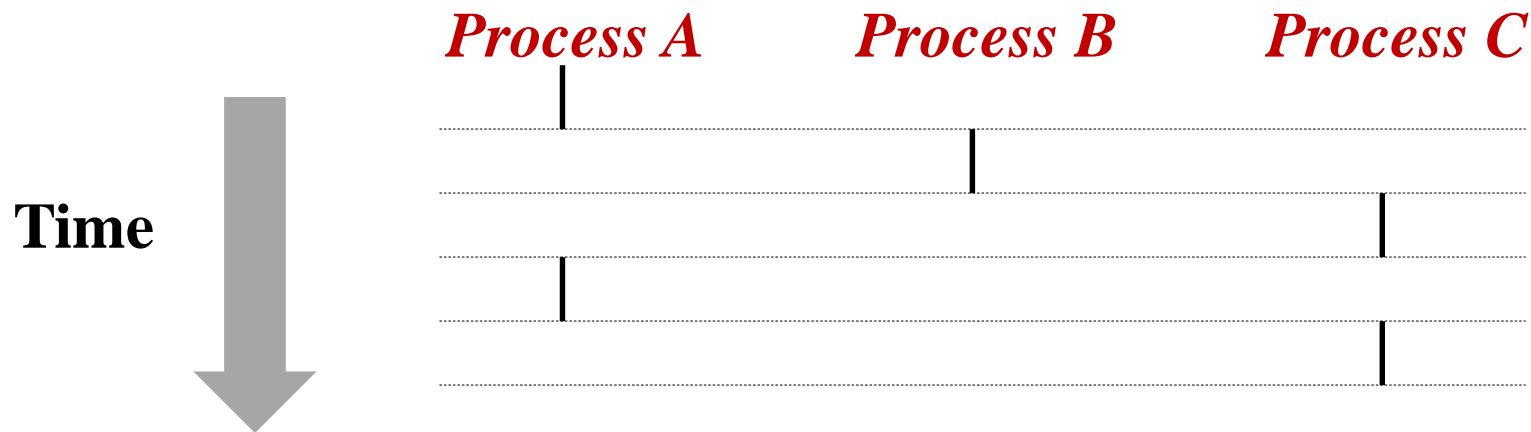


## ■ 多核处理器

- 单个芯片有多个CPU
- 共享主存、有的还共享cache
- 每个核可以执行独立的进程 kernel负责处理器的内核调度

# 并发进程(Concurrent Processes)

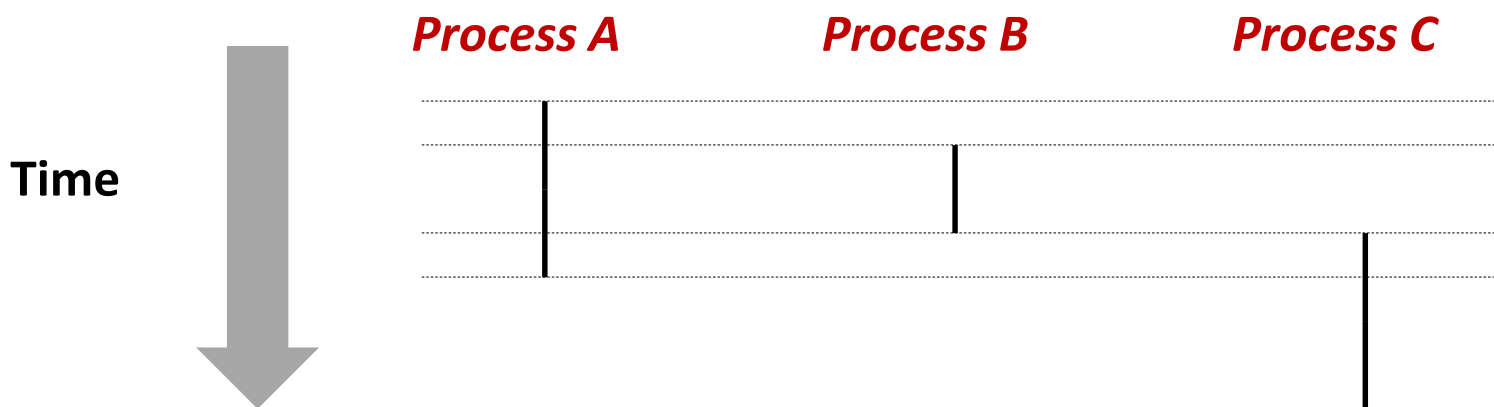
- 每个进程是个逻辑控制流
- 如果两个逻辑流在时间上有重叠，则称这两个进程是并发的(并发进程)
- 否则他们是顺序的



**示例 (单核CPU) : A & B, A & C是并发关系  
B & C是顺序关系**

# 用户角度看并发进程

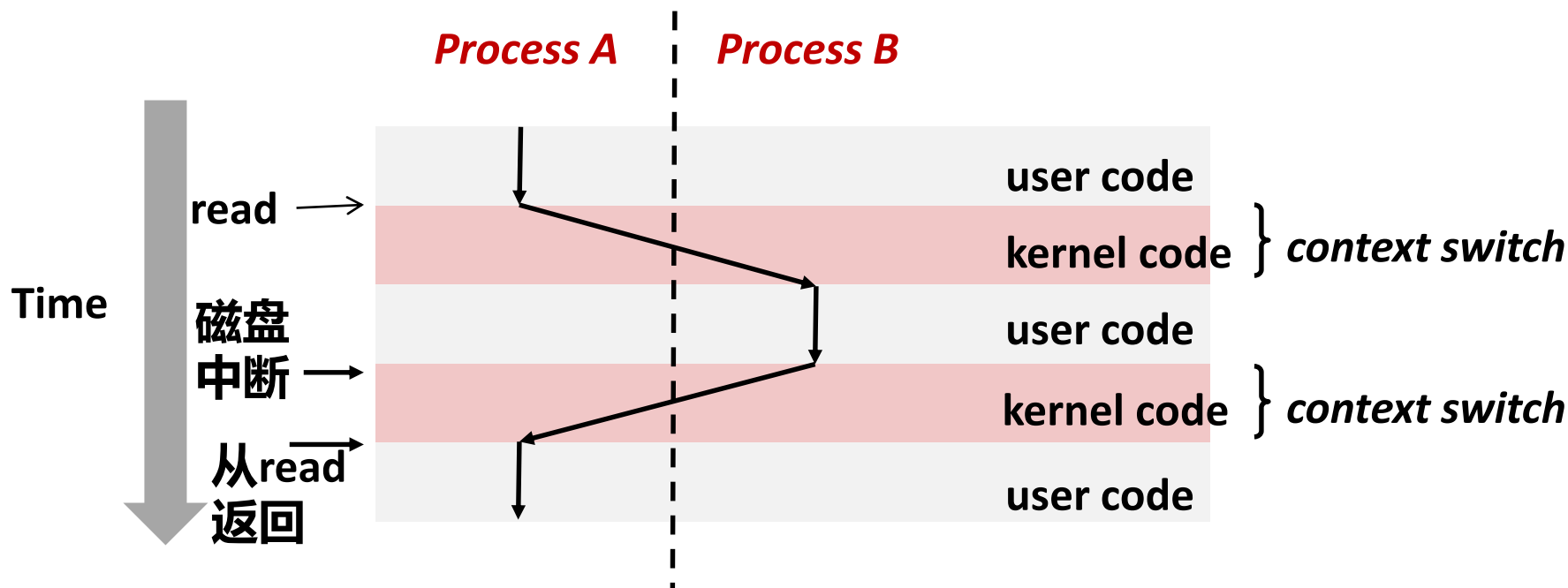
- 并发进程的控制流物理上是不相交的
- 然而，我们可以认为并发进程是并行运行的



- 进程内是串行、进程间是并行—乱序？-看OS

# 上下文切换 (Context Switching)

- 进程由常驻内存的操作系统代码块(称为**内核**)管理
  - ★内核不是一个单独的进程，而是作为现有进程的一部分运行
- 控制流通过**上下文切换**从一个进程传递到另一个进程





# 主要内容

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

# 系统调用错误的处理

- 当Linux系统级函数遇到错误时，通常返回-1并设置全局整数变量 **errno** 来标示出错原因.
- 硬性规定：
  - 必须检查每个系统级函数的返回状态
  - 只有少数是返回空的函数
- Example:

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

# 报错函数

- 通过定义下面的错误报告函数，能够在某种程度上简化代码：

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

对fork的调用：从4行缩减到2行

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

# 错误处理包装函数(Error-handling Wrappers)

- 通过使用错误处理包装函数可以更进一步简化代码:

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

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

对fork的调用缩减到1行

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

# 获取进程ID (Obtaining Process IDs)

- `pid_t getpid(void)`
  - 返回当前进程的PID
- `pid_t getppid(void)`
  - 返回父进程的PID

# 创建和终止进程

从程序员的角度，我们可以认为进程总是处于下面三种状态之一

## ■ 运行Running

- Process is either executing, or waiting to be executed and will eventually be *scheduled* (i.e., chosen to execute) by the kernel
- 进程要么在CPU上执行，要么在等待被执行且最终会被操作系统内核调度

## ■ 停止/暂停/挂起/Stopped/Paused/Hanged

- 进程的执行被挂起且不会被调度，直到收到新的信号

## ■ 终止Terminated

- 进程永远地停止了，**但仍占资源！**

# 终止 (terminate) 进程

- 进程会因为三种原因终止：
  - 收到一个信号，该信号的默认行为是终止进程
  - 从主程序返回
  - 调用exit函数
    - `void exit(int status)`
      - 以`status`退出状态来终止进程
      - 常规的：正常返回状态为0，错误为非零
      - 另一种设置退出状态的方法是从主程序中返回一个整数值
    - `exit` 函数不返回到下一行

# 创建进程

- 父进程通过调用**fork**函数创建一个新的运行的子进程
- **int fork(void)**
  - 子进程返回0，父进程返回子进程的PID
  - 新创建的子进程几乎但不完全与父进程相同：
    - 子进程得到与父进程虚拟地址空间相同的(但是独立的)一份**副本**（代码、数据段、堆、共享库以及用户栈）
    - 子进程获得与父进程任何打开文件描述符相同的副本
    - 子进程有不同于父进程的PID
- **fork**函数：被调用一次，却返回两次



# fork 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*

```

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

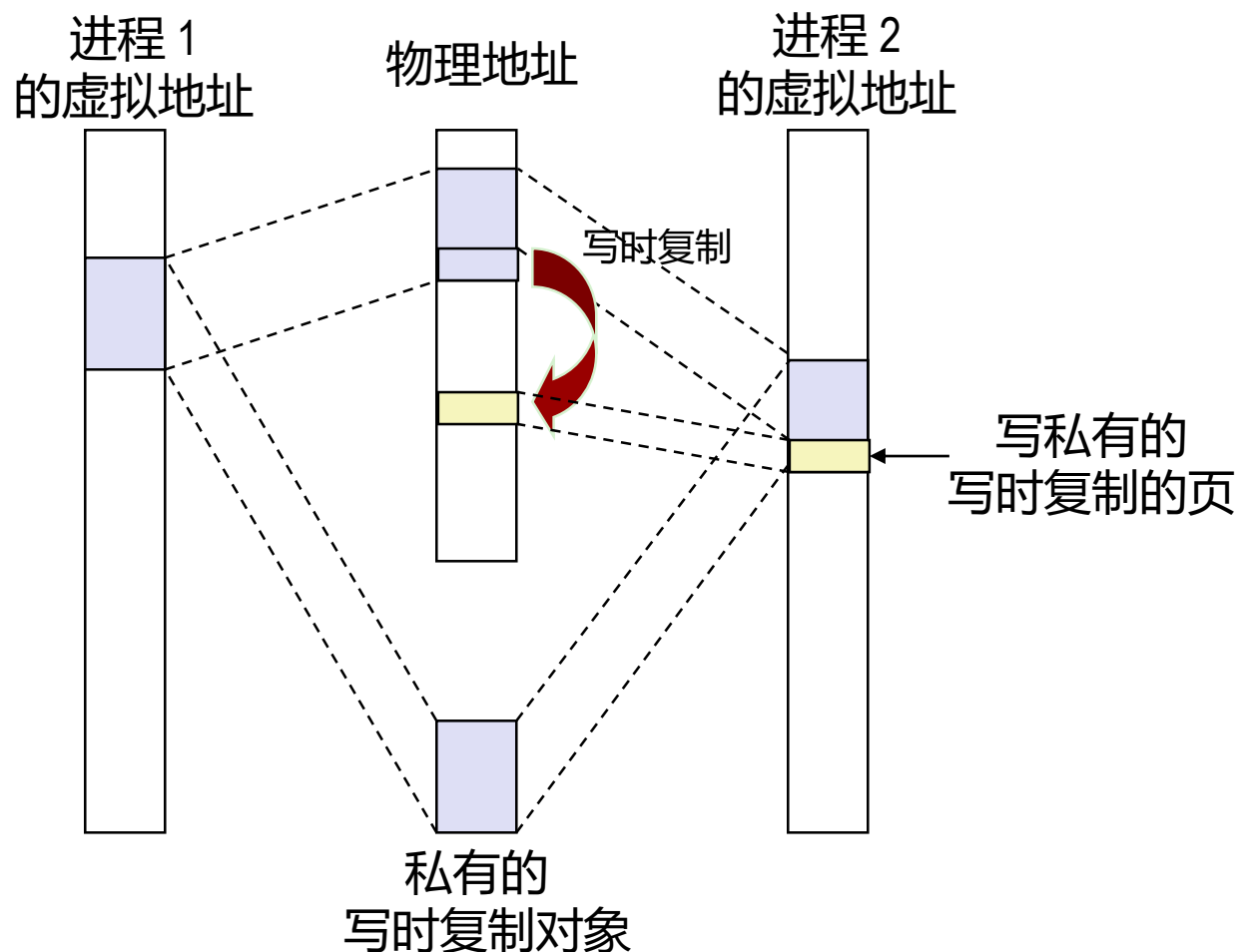
```

- 调用一次，返回两次
- 并发执行
  - 不能预测父进程与子进程的  
执行顺序
- 相同但是独立的地址空间
  - fork返回时 (第6行), x在  
父进程和子进程中都为1
  - x为局部变量-reg/stack
  - 后面, 父进程和子进程对x  
所做的任何改变都是独立的
- 共享文件
  - stdout文件在父、子进程是  
相同的

# fork 函数与虚拟内存

- 虚拟内存和内存映射解释了fork函数如何为每个新进程提供私有的虚拟地址空间.
- 为新进程创建虚拟内存
  - 创建当前进程的mm\_struct, vm\_area\_struct和页表的原样副本
  - 两个进程中的每个页面都标记为只读
  - 两个进程中的每个区域结构 (vm\_area\_struct) 都标记为私有的写时复制 (COW)
- 在新进程中返回时, 新进程拥有与调用fork进程相同的虚拟内存
- 随后的写操作通过写时复制机制创建新页面

# 共享对象： 私有的写时复制（Copy-on-write）对象



- 写私有页的指令触发保护故障
- 故障处理程序创建这个页面的一个新副本，更新PTE条目，且可写
- 故障处理程序返回时重新执行写指令
- 尽可能地延迟拷贝（创建副本）充分利用物理内存

# 用进程图(Process Graph)刻画fork

- 进程图是捕获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 $a \rightarrow b$ 表示语句 a 发生在语句 b 之前
  - 边上可以标记信息如变量的当前值
  - `printf` 语句的顶点可以标记上printf的输出
  - 每张图从一个没有入边的顶点开始
- 图的任何拓扑排序对应于程序中语句的一个可行的全序排列.
  - 所有顶点的总排序, 这些顶点的每条边都是从左到右的

# 进程图

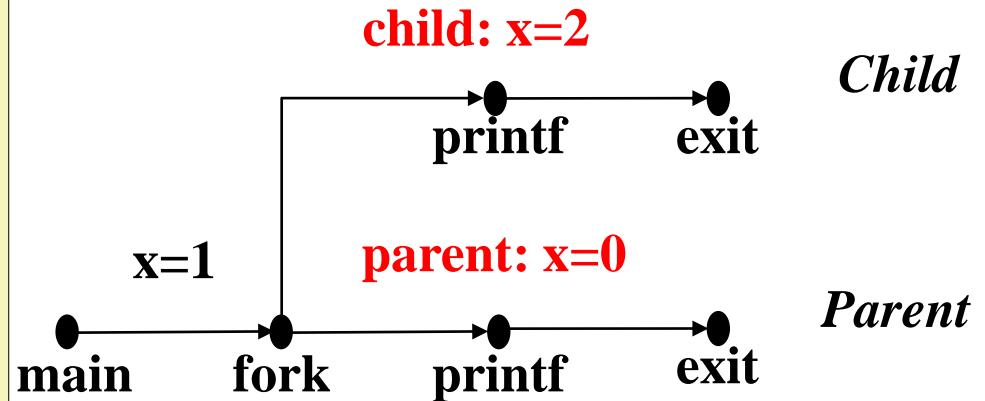
```

int main()                                fork.c
{
    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);
}

```

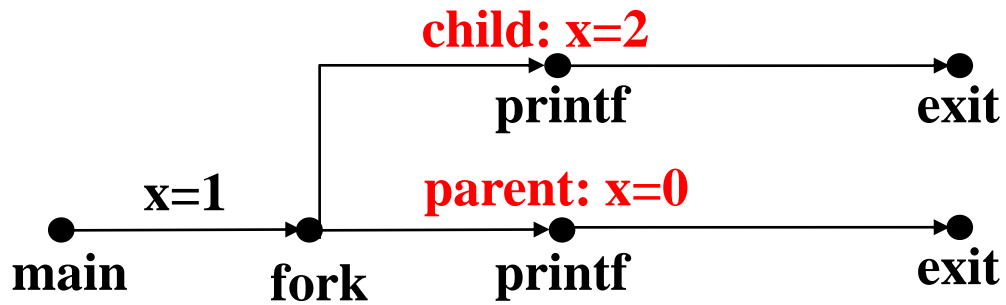


## 用进程图(Process Graph)刻画fork

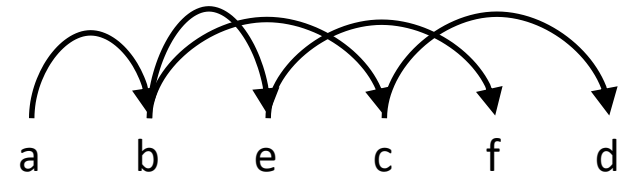
- 进程图是捕获并发程序中语句偏序的有用工具:
  - 每个顶点a对应一条语句的执行
  - 有向边a -> b 表示语句 a 发生在语句 b 之前
  - 边上可以标记信息如变量的当前值
  - printf 语句的顶点可以标记上printf的输出
  - 每张图从一个没有入边的顶点开始
- 图的任何拓扑排序对应于程序中语句的一个可行的全序排列.
  - 所有顶点的总排序, 这些顶点的每条边都是从左到右的

# 解释进程图

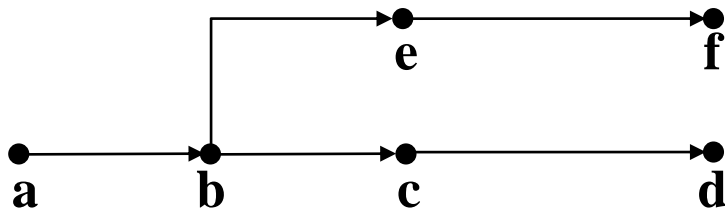
## ■ Original graph:



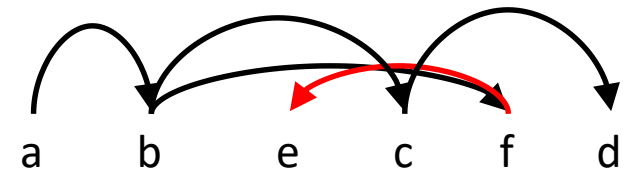
## 可行的全序排列



## ■ Relabled graph:



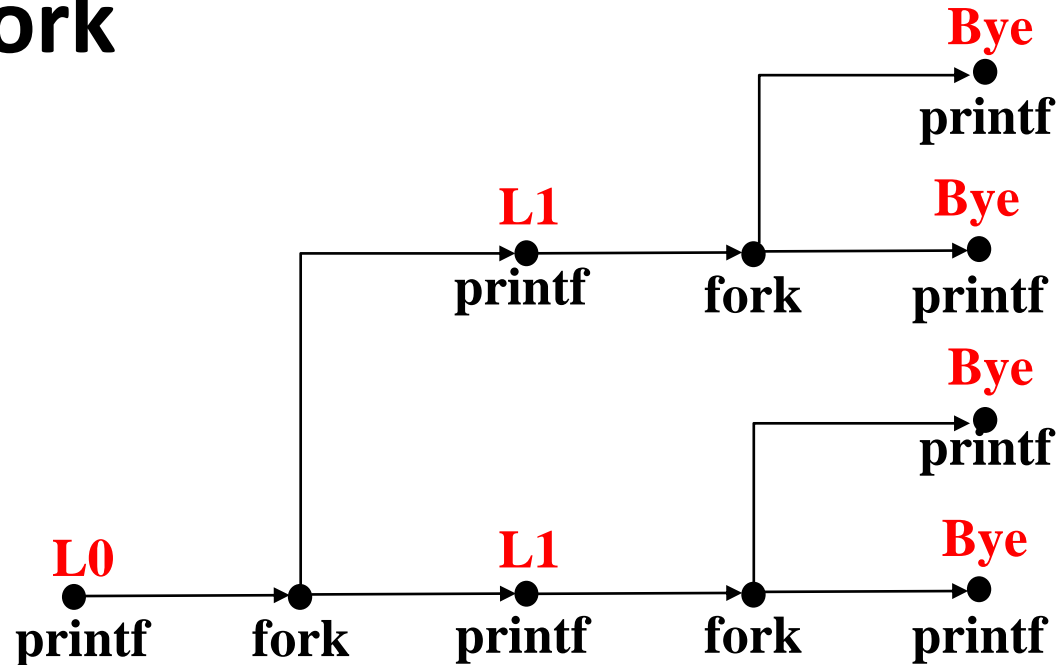
## 不可行的全序排列



## 两个连续的fork

***`forks.c`***

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



## 可能的输出:

L0  
L1  
Bye  
Bye  
L1  
Bye  
Bye

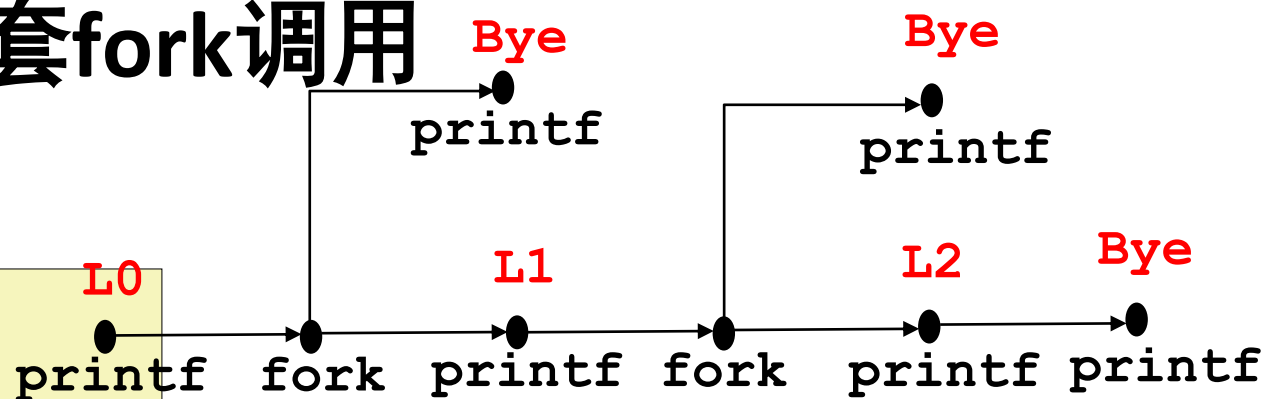
## 不可能的输出:

L0  
Bye  
L1  
Bye  
L1  
Bye  
Bye

# 父进程中的嵌套fork调用

*forks.c*

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



可能的输出:

L0  
L1  
Bye  
Bye  
L2  
Bye

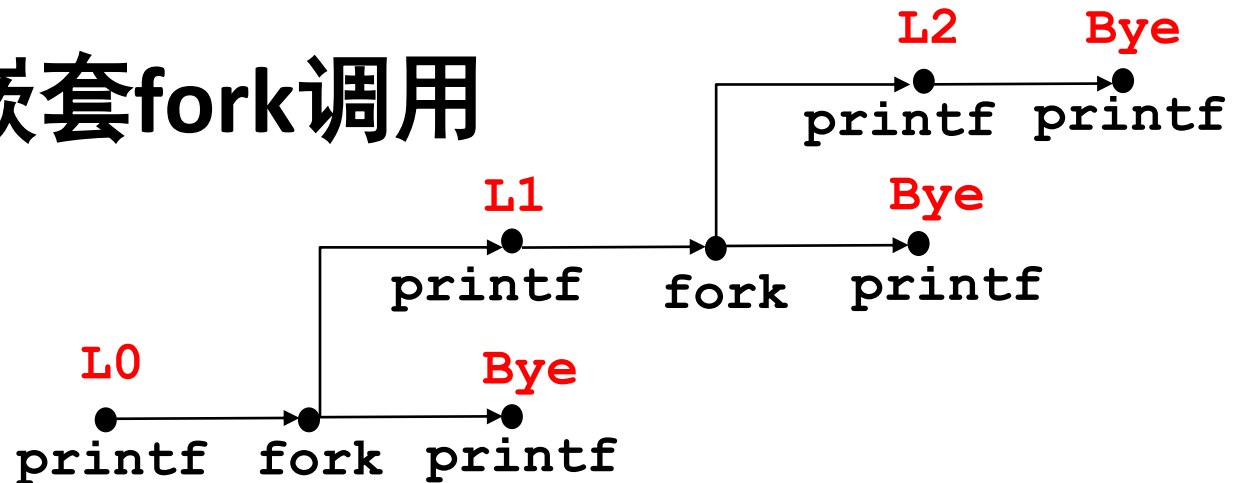
不可能的输出:

L0  
Bye  
L1  
Bye  
Bye  
L2



# 子进程中的嵌套fork调用

*forks.c*



```

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

```

可能的输出:

L0  
Bye  
L1  
L2  
Bye  
Bye

不可能的输出:

L0  
Bye  
L1  
Bye  
Bye  
L2

# 回收子进程(Reaping Child Processes)

- 想法-为什么回收？ --与fork创建相反！
  - 当进程终止时，它仍然消耗系统资源
    - Examples: Exit status, various OS tables(占用内存)
  - 称为“僵尸zombie”进程
    - 活着的尸体，半生半死
  - 僵尸进程占用内存资源、打开的IO资源等
- 如何回收 (Reaping)
  - 父进程执行回收操作( using wait or waitpid )
  - 父进程收到子进程的退出状态
  - 内核删掉僵死子进程、从系统中删除掉它的所有痕迹

**阅读：为什么程序退出了还提示内存被占用**

<https://docs.pingcode.com/ask/297738.html?p=297738>

# 回收子进程(Reaping Child Processes)

- 父进程不回收子进程的后果：
  - 如果父进程没有回收它的僵死子进程就终止了，内核安排 init进程去回收它们（init进程PID为1，系统启动时创建，不会终止，是所有进程的祖先）
  - 长时间运行的进程应当主动回收它们的僵死子进程
    - e.g., shells and servers

# 僵死进程

```
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 */  
    }  
}
```

*forks.c*

```
linux> ./forks7 &
```

```
[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命令显示的子进程标记为“defunct”即，僵死进程

■ 杀死父进程，从而让init回收子进程

# 非终止子进程

```
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);
    }
}
```

*forks.c*

```
linux> ./forks 8
Terminating Parent, PID = 6675
Running Child, PID = 6676
linux> ps
  PID TTY          TIME CMD
 6585 tttyp9      00:00:00 tcsh
 6676 tttyp9      00:00:06 forks
 6677 tttyp9      00:00:00 ps
linux> kill 6676
linux> ps
  PID TTY          TIME CMD
 6585 tttyp9      00:00:00 tcsh
 6678 tttyp9      00:00:00 ps
```

父进程终止，但子进程仍处于活动状态

必须明确地杀死子进程，否则将无限持续地运行

# 僵尸进程与孤儿进程

- **孤儿进程：**一个父进程退出，而它的一个或多个子进程还在运行，那么那些子进程将成为孤儿进程。孤儿进程将被init进程(进程号为1)所收养，并由init进程对它们完成状态收集工作。
- **僵尸进程：**一个进程使用fork创建子进程，如果子进程退出，而父进程并没有调用wait或waitpid获取子进程的状态信息，那么子进程的进程描述符仍然保存在系统中。这种进程称之为僵尸进程。

# 与子进程同步: wait/waitpid

- 父进程通过wait/waitpid函数回收子进程

`wait(&status)` 同 `waitpid(-1, &status, 0)`

- `int wait(int *child_status)`

- **挂起**当前进程的执行直到它的一个子进程终止/停止
- 返回已终止/停止子进程（**可能很多-是个集合**）的 `pid`
- 如 `child_status != NULL`, 则在该指针指向的整型量中写入关于终止原因和退出状态的信息):

- 用 `wait.h` 头文件中定义的宏来检查

`WIFEXITED, WEXITSTATUS,`

`WIFSIGNALED, WTERMSIG, WSTOPSIG,`

`WIFSTOPPED, WIFCONTINUED`

# 子进程退出状态信息

## ■ 主要的几个宏

- `WIFEXITED`: 如果子进程通过调用`exit`或者一个返回(`return`)正常终止, 就返回真
- `WEXITSTATUS`: 返回一个正常终止的子进程的退出状态。只有在`WIFEXITED()`返回为真时, 才会定义这个状态
- 还有`WIFSIGNALED`、`WTERMSIG`等详见书P517



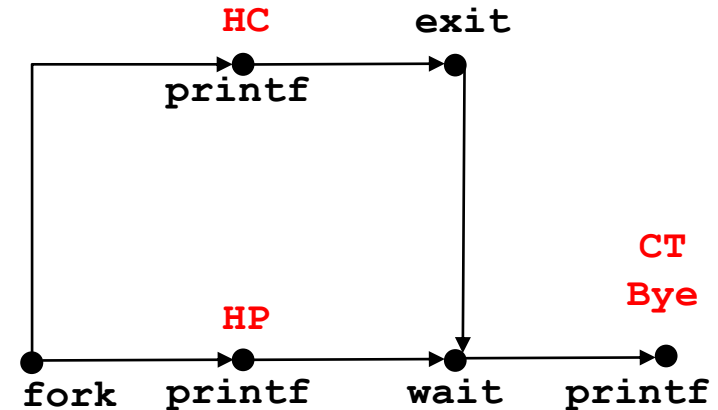
# 与子进程同步：wait示例1

```

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:

HC  
HP  
CT  
Bye

不可能的输出:

HP  
CT  
Bye  
HC

# 与子进程同步：wait示例2

- 子进程完成结束的顺序是任意的（没有固定的顺序）
- 可用宏函数WIFEXITED和WEXITSTATUS 获取进程的退出状态信息

```
void fork10() { //forks.c  
    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);  
    }  
}
```

# waitpid: 等待特定进程

- `pid_t waitpid(pid_t pid, int &status, int options)`
  - 挂起当前进程直到指定进程终止才返回，有多种选项

```
void fork11() { //forks.c
    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);
    }
}
```

## pid\_t waitpid(pid\_t pid,int &statusp,int options)

- pid=-1 等待集合是由父进程所有的子进程组成
- pid=n 等待集合是某个单独的子进程
- Options:0 等待（某一个）子进程**终止并回收**
  - WNOHANG: 如果等待集合存在子进程没有终止，就立即返回0。父进程可继续其他工作
  - WUNTRACED:挂起当前进程，直到等待集合中的任一进程终止或停止，则返回其pid。用于检查
  - WCONTINUED:挂起当前进程，直到等待集合中的任一进程从运行到终止或一个停止的进程收到SIGCONT而恢复运行。
  - 可以组合WNOHANG|WUNTRACED 如果等待集合任一子进程没有终止返回0，如果有一个终止或停止则返回其pid。

# execve : 加载并运行程序

- **int execve(char \*filename, char \*argv[], char \*envp[])**
- 在当前进程中载入并运行程序: loader加载器函数
  - **filename**:可执行文件
    - 目标文件或脚本(用#!指明解释器,如 #!/bin/bash)
  - **argv**:参数列表,惯例: **argv[0]==filename**
  - **envp**: 环境变量列表
    - "name=value" strings (e.g., USER=droh)
    - getenv, setenv, unsetenv, printenv
- Loader删除进程现有的虚拟内存段, 创建一组新的段(栈与堆初始化为0),并将虚拟地址空间中的页映射到可执行文件的页大小的片 chunk, 新的代码与数据段被初始化为可执行文件的内容, 然后跳到\_start..... 除了一些头部信息, 实际没读文件, 直到缺页中断
- 覆盖当前进程的代码、数据、栈
  - 保留: 有相同的PID, 继承已打开的文件描述符和信号上下文
- Called **once** and **never** returns(调用一次并从不返回)
  - ...除非有错误, 例如: 指定的文件不存在

`execve()` 是一个系统调用，用于加载并运行一个新的程序。当调用 `execve()` 时，操作系统会替换当前进程的地址空间，加载新的程序，并从新程序的入口点开始执行。这个过程涉及到多个步骤，包括加载程序的可执行文件、设置新的地址空间、初始化环境变量等。

### `execve()` 的工作原理

#### 1. 加载可执行文件：

- `execve()` 会读取指定的可执行文件（通常是 ELF 格式），解析其头部信息，确定程序的入口点和所需的内存布局。
- 操作系统会根据可执行文件的段信息（如代码段、数据段等），将程序的各个部分加载到内存中。

#### 2. 设置新的地址空间：

- 当前进程的地址空间会被清空，包括代码段、数据段、堆和栈。
- 新的地址空间会被设置，包括加载可执行文件的代码段和数据段，以及初始化堆和栈。

#### 3. 初始化环境变量：

- `execve()` 会根据传入的参数（如环境变量）设置新进程的环境变量。
- 新进程的环境变量会从父进程继承，或者通过 `execve()` 的参数显式设置。

#### 4. 设置程序入口点：

- 操作系统会设置程序的入口点，通常是 `_start` 函数（由 C 标准库提供）。
- `_start` 函数会初始化程序的运行时环境，包括设置栈帧、初始化全局变量等，然后调用 `main()` 函数。

### 缺页中断

当 `execve()` 加载新的程序时，操作系统会将可执行文件的各个段加载到内存中。这个过程可能会触发缺页中断（Page Fault），但这些缺页中断是正常的，因为操作系统需要将磁盘上的数据加载到内存中。

### 加载过程中的缺页中断

#### 1. 映射文件到内存：

- 操作系统会将可执行文件的各个段（如代码段、数据段）映射到进程的地址空间。
- 这些段最初可能只是映射到虚拟内存中，而没有实际加载到物理内存中。

#### 2. 触发缺页中断：

- 当进程尝试访问这些映射的内存区域时，如果对应的物理页面尚未加载到内存中，就会触发缺页中断。
- 操作系统会处理缺页中断，将所需的页面从磁盘加载到物理内存中。

#### 3. 继续执行：

- 一旦页面被加载到内存中，缺页中断处理程序会更新页表，标记该页面为已加载。
- 进程可以继续执行，访问所需的内存区域。

# 新程序启动后的函数栈结构

execve加载了filename之后，它调用书7.9节（P484）中描述的启动代码。启动代码设置栈，并将控制传递给新程序的主函数。

主函数的形式：

```
int main(int argc, char **argv,
char **envp)
```

函数参数

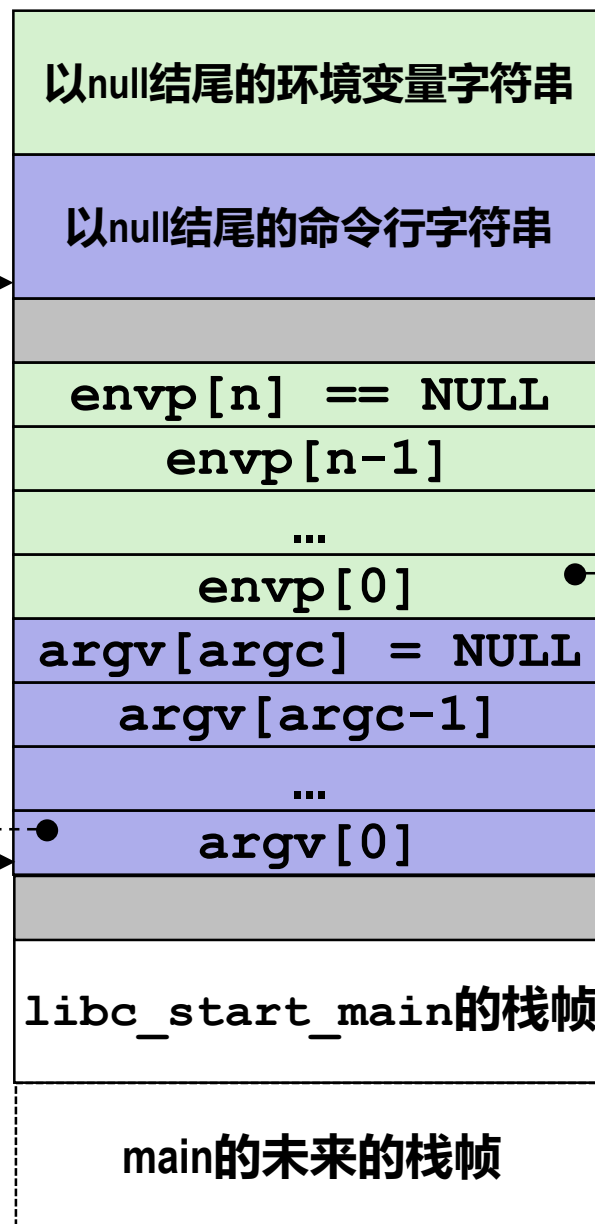
可执行目标文件的名字

参数列表

argv  
(in %rsi)

argc  
(in %rdi)

给出argv[]数组中非空指针的数量



栈底

指向envp[]数组  
中的第一个条目

environ  
(全局变量)

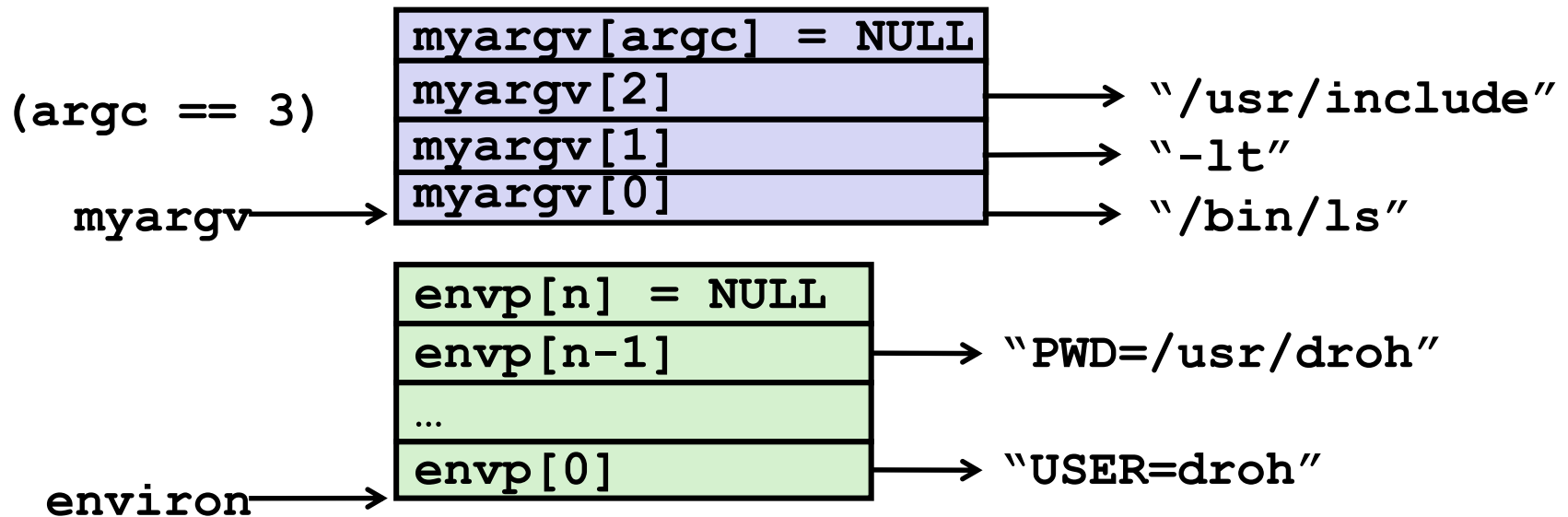
envp  
(in %rdx)

环境变量列表

栈顶

# execve 示例

- 在子进程中用当前的环境执行 “/bin/ls -lt /usr/include”



```

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

```



# 习题

某 C 程序(64 位模式)的 main 函数参数 argv 地址为 0x0000413433323110, 其内容如下:

0x0000413433323110: 30 31 32 33 34 41 00 00 33 31 32 33 34 41 00 00

0x0000413433323120: 35 31 32 33 34 41 00 00 00 00 00 00 00 00 00 00

0x0000413433323130: 31 43 00 30 00 32 42 00 38 00 31 31 32 32 00 30

0x0000413433323140: 32 33 00 61 41 00 31 00 32 00 33 00 31 00 00 31

请写出 程序名: \_\_\_\_\_, 本程序的参数个数\_\_\_\_\_

按顺序写出各个参数为\_\_\_\_\_

提示: int main(int argc, char \*argv[]); 字符 0、A、a 的 ASCII 为 0x30、0x41、0x61

**答案:**

1C

2

0 2B

Argv 变量 (二级指针) 指向一个以 null 结尾的指针数组, 其中每个指针都指向一个参数字符串

注意: 数据的大小端顺序和字符串在内存中的顺序区别 (书 P34)

# 总结

## ■ 异常Exceptions

- 需要非常规控制流的事件
- 外部产生——中断
- 内部产生——陷阱和故障

## ■ 进程Processes

- 任何给定的时间，系统中都有多个活动进程
- 但是，在单个内核上，一个时刻只能有一个进程执行
- 每个进程**似乎**完全拥有处理器和私有内存空间（的控制）

# 总结(cont.)

## ■ 创建进程

- `fork`: 1次调用、2次返回

## ■ 进程退出

- `exit`: 1次调用、0次返回

## ■ Reaping and waiting for processes

- Call `wait` or `waitpid`

## ■ 加载和运行程序

- `execve` (或`exec`函数的其他变体)
- 一次调用, 0次返回 (如没有错误)

# 习题

- 1.内核为每个进程维持一个上下文，不属于进程上下文的是（ ）
  - A.寄存器    B.进程表    C.文件表    D.调度程序
- 2.关于异常处理后返回的叙述，错误的叙述是()
  - A.中断处理结束后，会返回到下一条指令执行
  - B.故障处理结束后，会返回到下一条指令执行
  - C.陷阱处理结束后，会返回到下一条指令执行
  - D.终止异常，不会返回
- 1.D      2. B

*Hope you  
enjoyed  
the  
CSAPP  
course!*