

# 第8章 异常控制流——异常和进程

## Exceptional Control Flow——Exceptions and Processes

- 教 师： 郑贵滨
- 计算机科学与技术学院
- 哈尔滨工业大学

# 主要内容

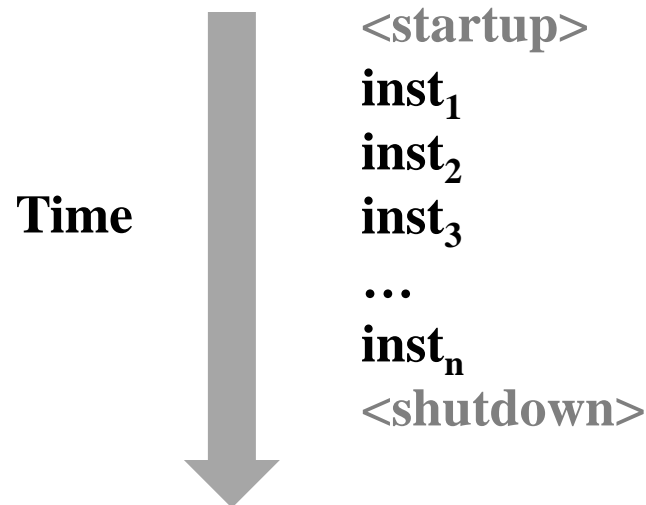
- 异常控制流(Exceptional Control Flow)
- 异常(Exceptions)
- 进程(Processes)
- 进程控制(Processes Control)

# 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)

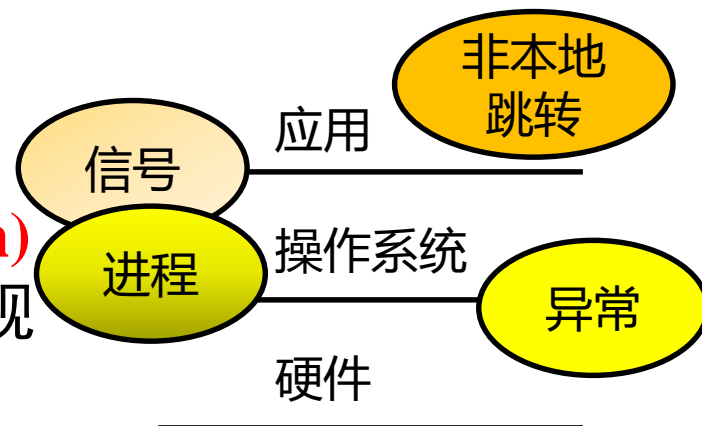
- 改变控制流的两种机制:
  - 跳转和分支(Jumps and branches)
  - 调用和返回(Call and return)能够对(由程序变量表示的)程序状态的变化做出反应
- 不足：难以对系统状态的变化做出反应
  - 磁盘或网络适配器的数据到达
  - 除零错误
  - 用户的键盘输入( Ctrl-C )
  - 系统定时器超时上述系统变化不能用程序变量表示
- 现代系统通过使控制流发生突变对这些情况做出反应，称为“异常控制流”

# 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, 跨越函数之间的控制权跳转

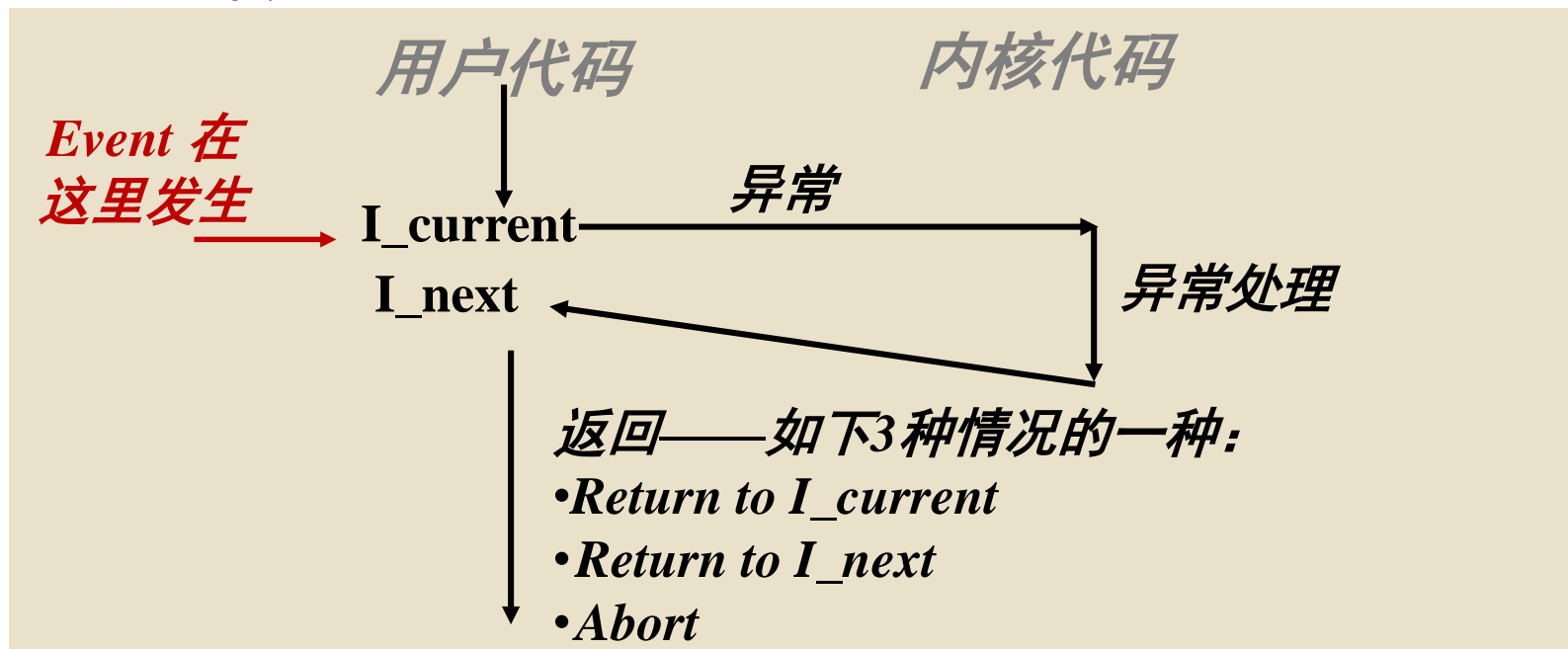


# 主要内容

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

# 异常(Exceptions)

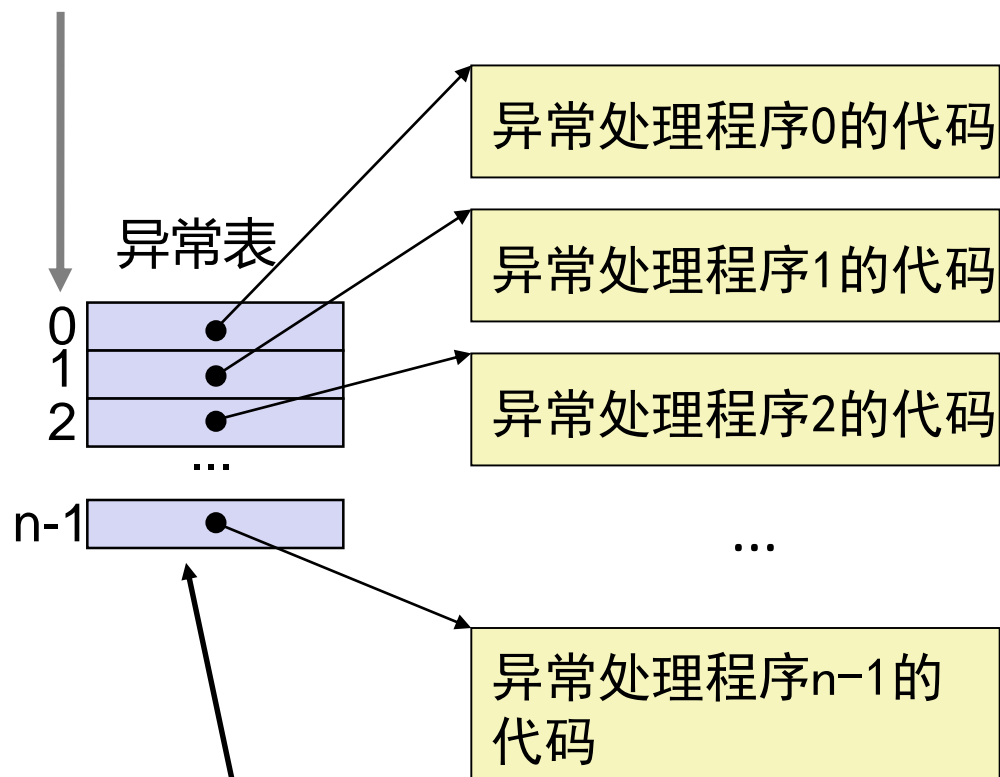
- **异常**是指为响应某个事件将控制权转移到操作系统内核中的情况
  - 内核指操作系统常驻内存的部分
  - 事件示例：被零除、算术运算溢出、缺页、I/O请求完成、键盘输入



# 异常处理

## 硬件 + 软件的配合

异常号



- 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, 若异常  $k$  发生, 则异常  $k$  的处理程序立刻被调用

异常表是一张跳转表, 表目  $k$  包含异常  $k$  的处理程序代码的地址

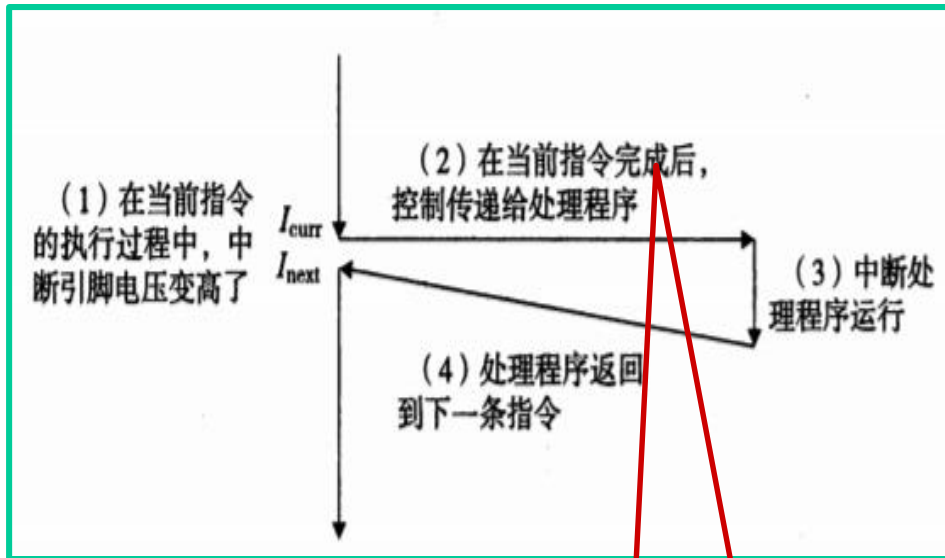


# 异步异常 Asynchronous Exceptions (中断 Interrupts)

- 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



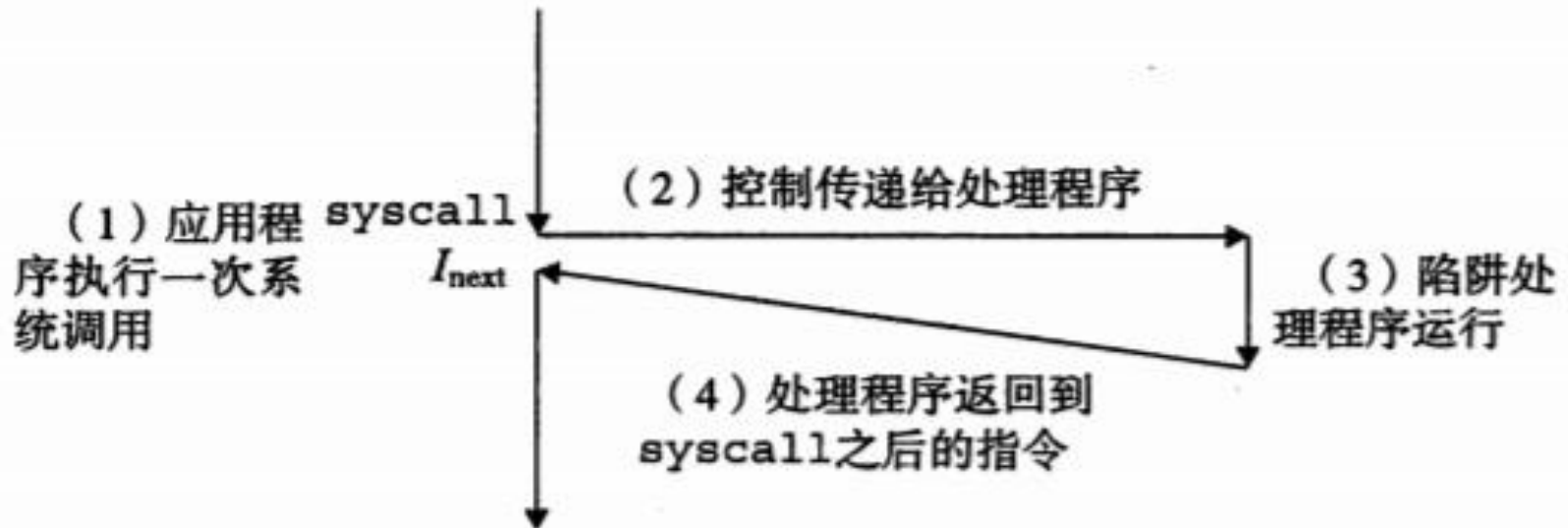
跳转时机：当前指令完成后

# 同步异常(Synchronous Exceptions)

## ■ 同步异常——执行指令产生的结果

### ■ 陷阱(Traps)

- 有意的，**执行指令的结果**（发生时间可预知）
- 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)

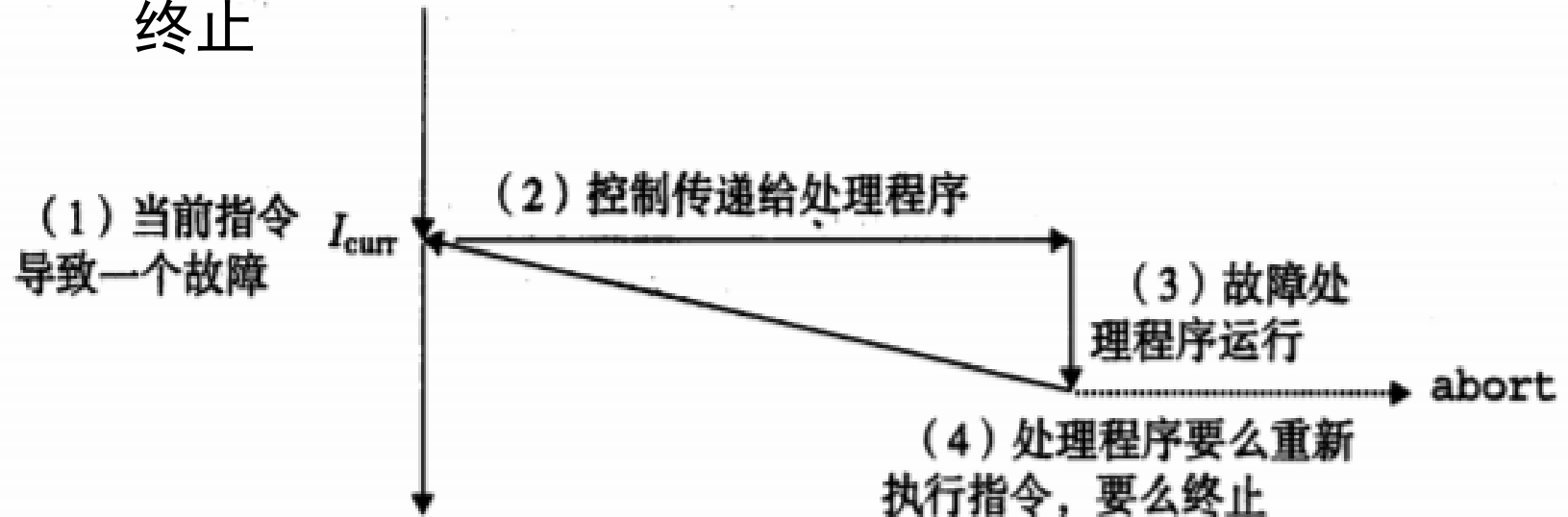
## ■ 同步异常——执行指令产生的结果...

### ■ 故障(Faults)

- 不是有意的，但可能被修复

Examples: 缺页故障(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)

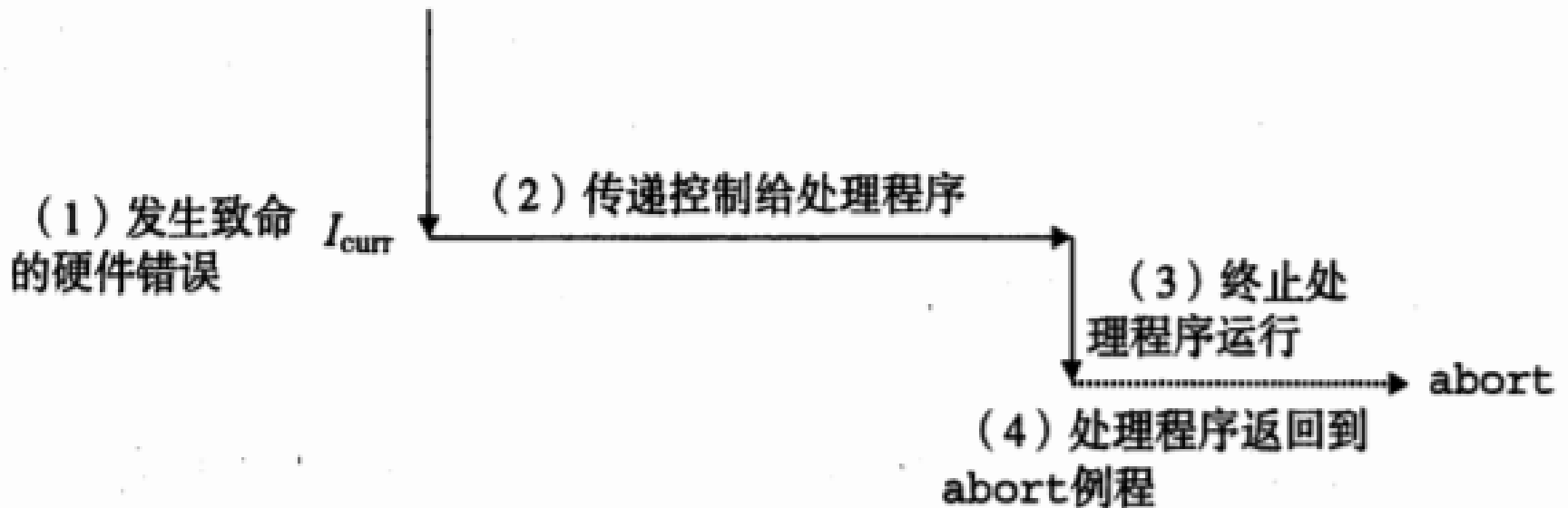
## ■ 同步异常——执行指令产生的结果...

### ■ 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)到致命的硬件错误

- 中止当前程序



# 陷阱 (Traps)——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

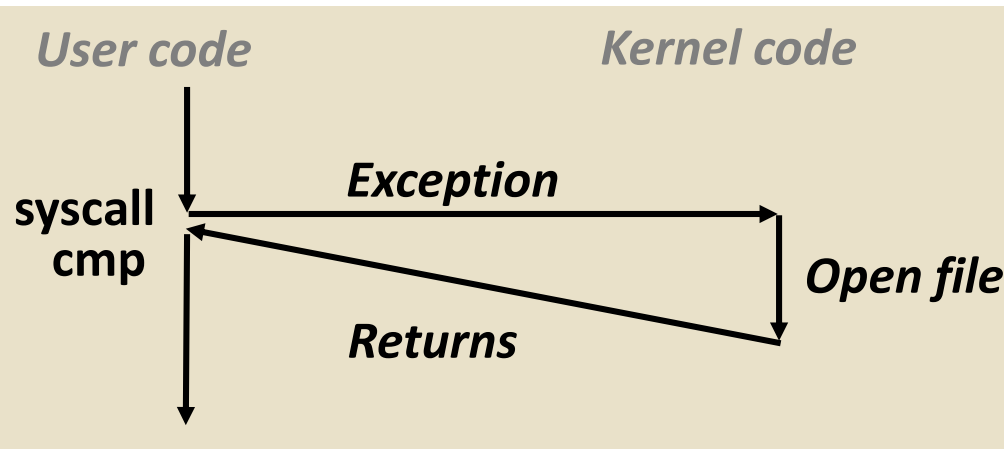
# 陷阱 (Traps)——系统调用的例子: 打开文件

- 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 $0xfffffffffff001,%rax
...
e5dfa:  c3               retq
  
```



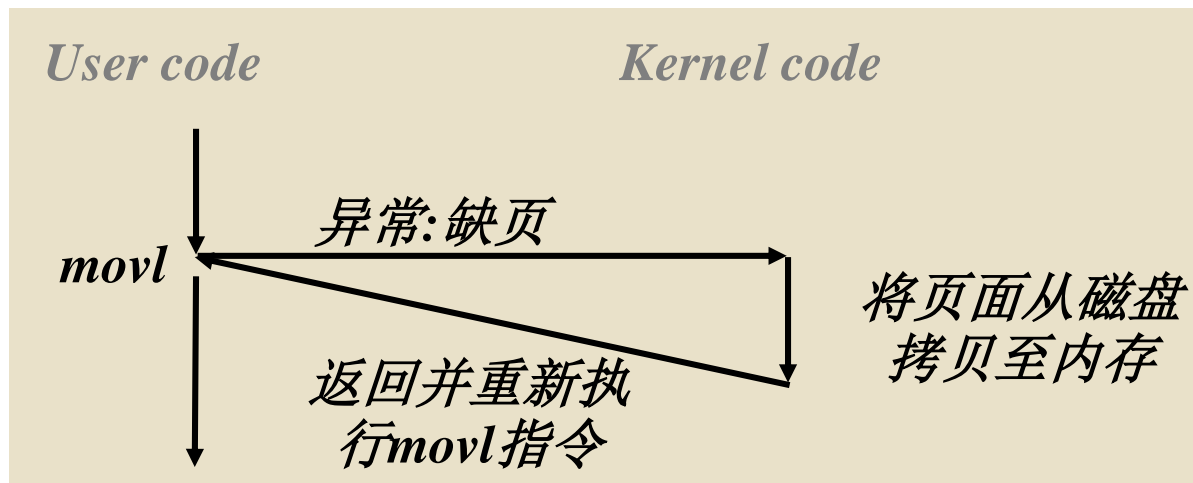
- `%rax` 包含系统调用号
- 其他参数(至多6个)依次在 `%rdi`, `%rsi`, `%rdx`, `%r10`, `%r8`, `%r9`
- 返回值在 `%rax`
- 负数返回值表明发生了错误, 对应于负的 `errno`

# 故障的例子：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;
}
```

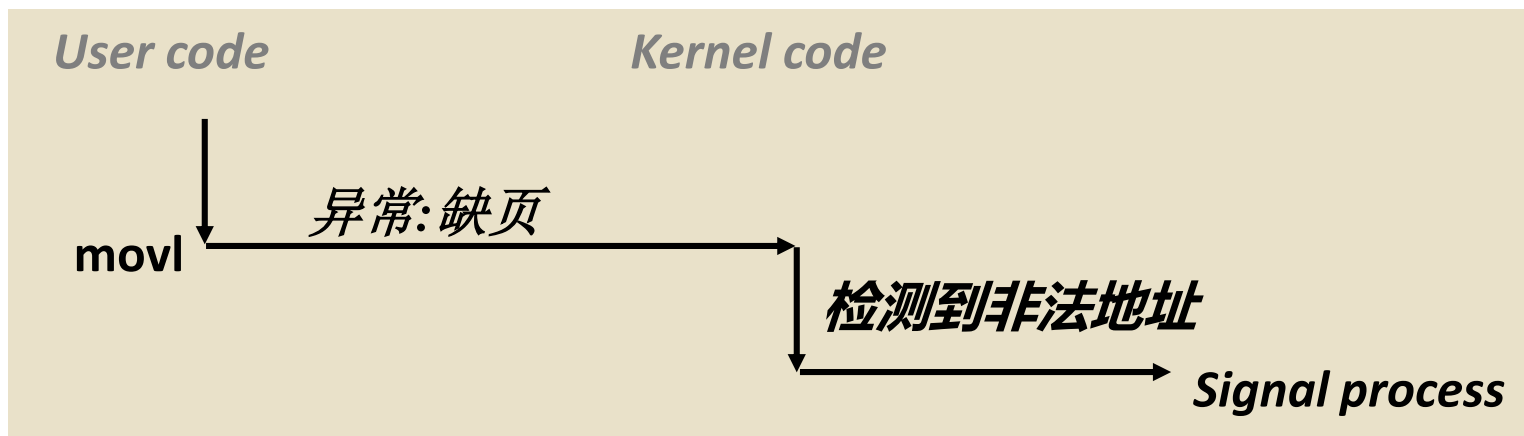
80483b7: c7 05 10 9d 04 08 0d movl \$0xd, 0x8049d10



# 故障的例子：非法内存引用

```
int a[1000];  
main ()  
{  
    a[5000] = 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“段故障” (segmentation fault)退出

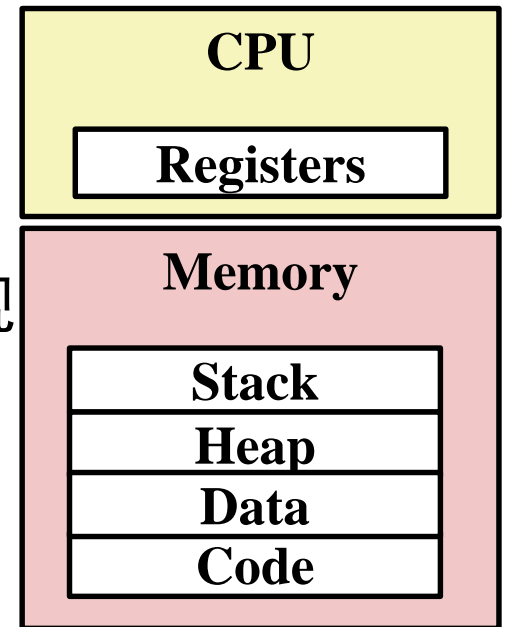


# 主要内容

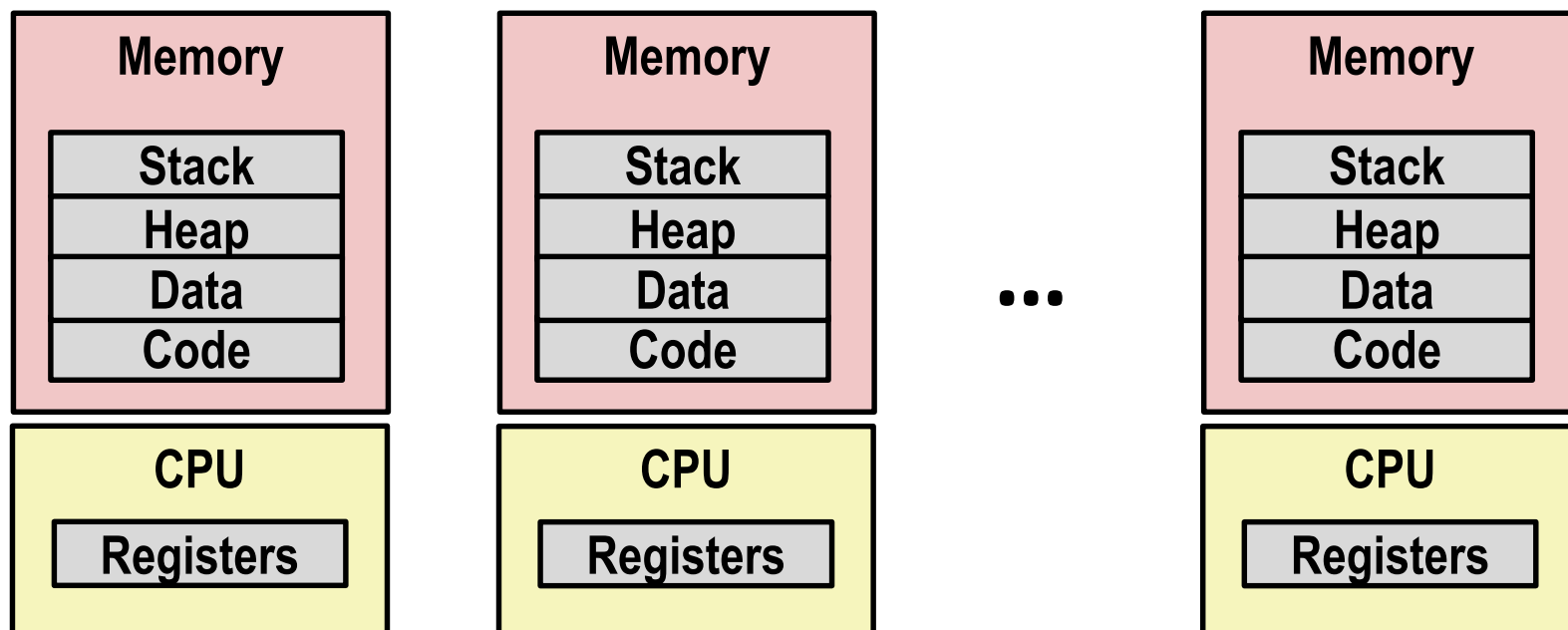
- 异常控制流(Exceptional Control Flow)
- 异常(Exceptions)
- 进程(Processes)
- 进程控制(Processes Control)

# 进程(Processes)

- 定义:一个正在运行的程序的实例(A *process* is an instance of a running program)
  - 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  
由OS内核通过**上下文切换**机制实现
  - **私有地址空间 (Private address space)**
    - 每个程序似乎独占地使用内存系统  
由OS内核的**虚拟内存**机制实现



# 多重处理的假象



- 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

```

Terminal File Edit View Search Terminal Help
top - 20:58:14 up 5:23, 1 user, load average: 1.34, 0.45, 0.20
Tasks: 185 total, 2 running, 183 sleeping, 0 stopped, 0 zombie
%Cpu(s): 42.3 us, 23.7 sy, 33.7 ni, 0.0 id, 0.0 wa, 0.0 hi, 0.3 si, 0.0 st
KiB Mem : 2036780 total, 117528 free, 1305012 used, 614240 buff/cache
KiB Swap: 2094076 total, 2049068 free, 45008 used. 503420 avail Mem

  PID USER      PR  NI   VIRT   RES   SHR  S  %CPU  %MEM     TIME+ COMMAND
14136 linda    20   0 2097040 245048 121328 R  19.6  12.0   0:04.35 Web Content
8663  linda    20   0 1249912 269268 42056 S  16.9  13.2  16:21.32 compiz
14018 linda    20   0 2238548 283732 112924 S  15.3  13.9   0:09.75 firefox
7845  root     20   0 487720 116352 32812 S   5.0   5.7   1:37.50 Xorg
8730  linda    20   0 1004620 145128 22464 S   3.7   7.1   0:06.37 gnome-software
  33 root     20   0      0      0      0 S   1.0   0.0   0:00.67 kswapd0
   6 root     20   0      0      0      0 S   0.3   0.0   0:00.53 ksoftirqd/0
  194 root     0 -20      0      0      0 S   0.3   0.0   0:00.56 kworker/0:1H
7665  root     20   0 277628 5564 5436 S   0.3   0.3   0:00.35 accounts-daemon
7796  root     20   0 283672 7976 5828 S   0.3   0.4   0:00.21 polkitd
7894  nobody   20   0 54608 3224 3096 S   0.3   0.2   0:00.08 dnsmasq
9048  linda    20   0 599748 24352 20332 S   0.3   1.2   0:00.44 update-notifier
13894 linda    20   0 43540 3572 2924 R   0.3   0.2   0:00.63 top
   1 root     20   0 119632 4316 2852 S   0.0   0.2   0:02.11 systemd
   2 root     20   0      0      0      0 S   0.0   0.0   0:00.00 kthreadd
  
```

任务管理器

文件(F) 选项(O) 查看(V)

进程 性能 应用历史记录 启动 用户 详细信息 服务

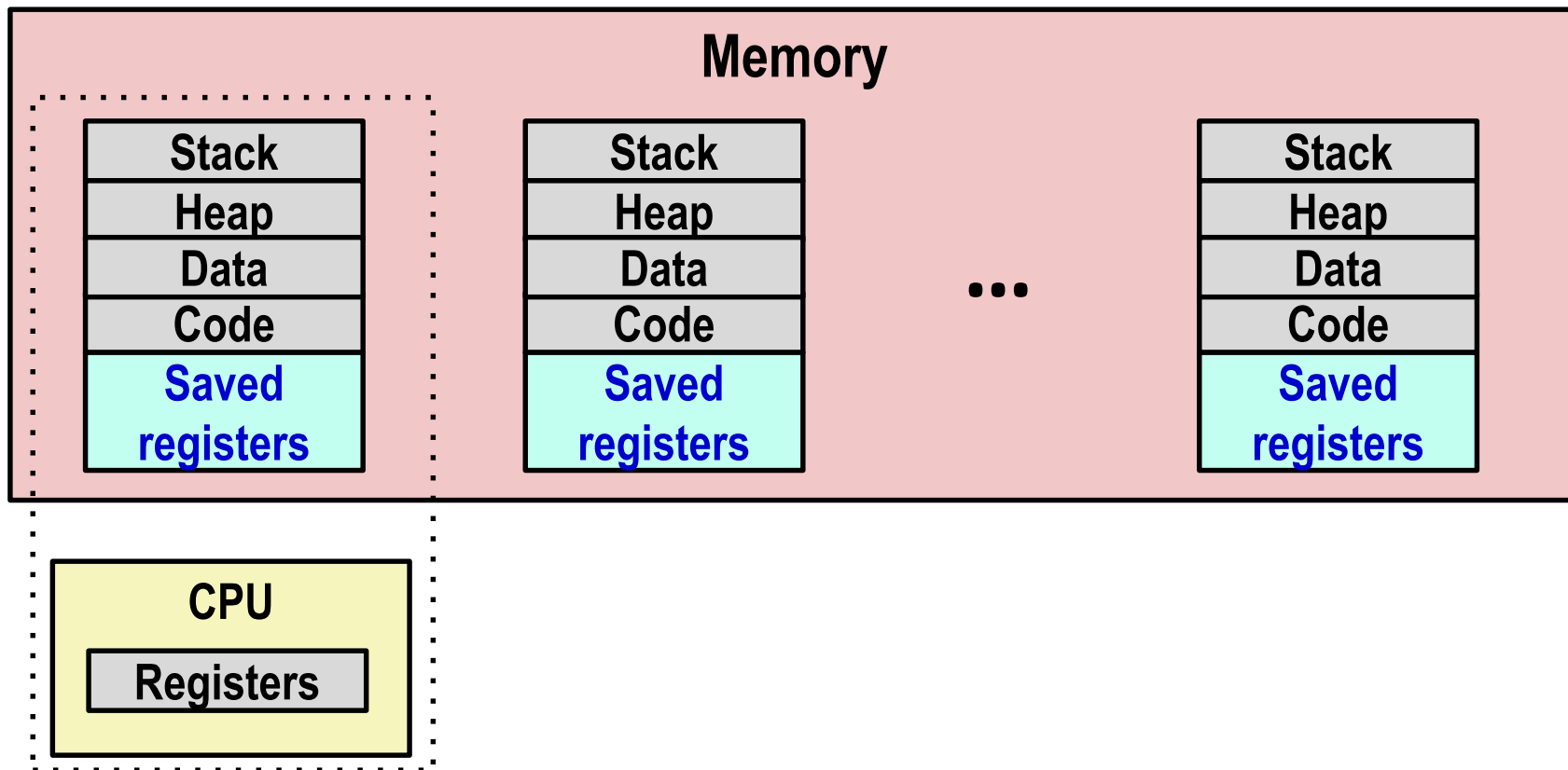
名称	10% CPU	34% 内存	0% 磁盘	0% 网络
应用 (8)				
> Foxmail 7.2 (32 位)	0%	15.9 MB	0.1 MB/秒	0.1 Mbps
> Microsoft PowerPoint (32 位)...	0%	145.7 MB	0 MB/秒	0 Mbps
> Microsoft Word (32 位)	0%	40.5 MB	0 MB/秒	0 Mbps
> VMware Workstation (32 位)...	0%	58.5 MB	0 MB/秒	0 Mbps
> Windows 资源管理器 (3)	0.1%	59.8 MB	0 MB/秒	0 Mbps
	0%	45.4 MB	0 MB/秒	0 Mbps
	0.2%	18.7 MB	0 MB/秒	0 Mbps
	0%	34.4 MB	0 MB/秒	0 Mbps
	0%	2.1 MB	0 MB/秒	0 Mbps
	0%	0.2 MB	0 MB/秒	0 Mbps
	0%	2.3 MB	0 MB/秒	0 Mbps
	0%	12.4 MB	0 MB/秒	0 Mbps

结束任务(E)

## ■ 运行“top”

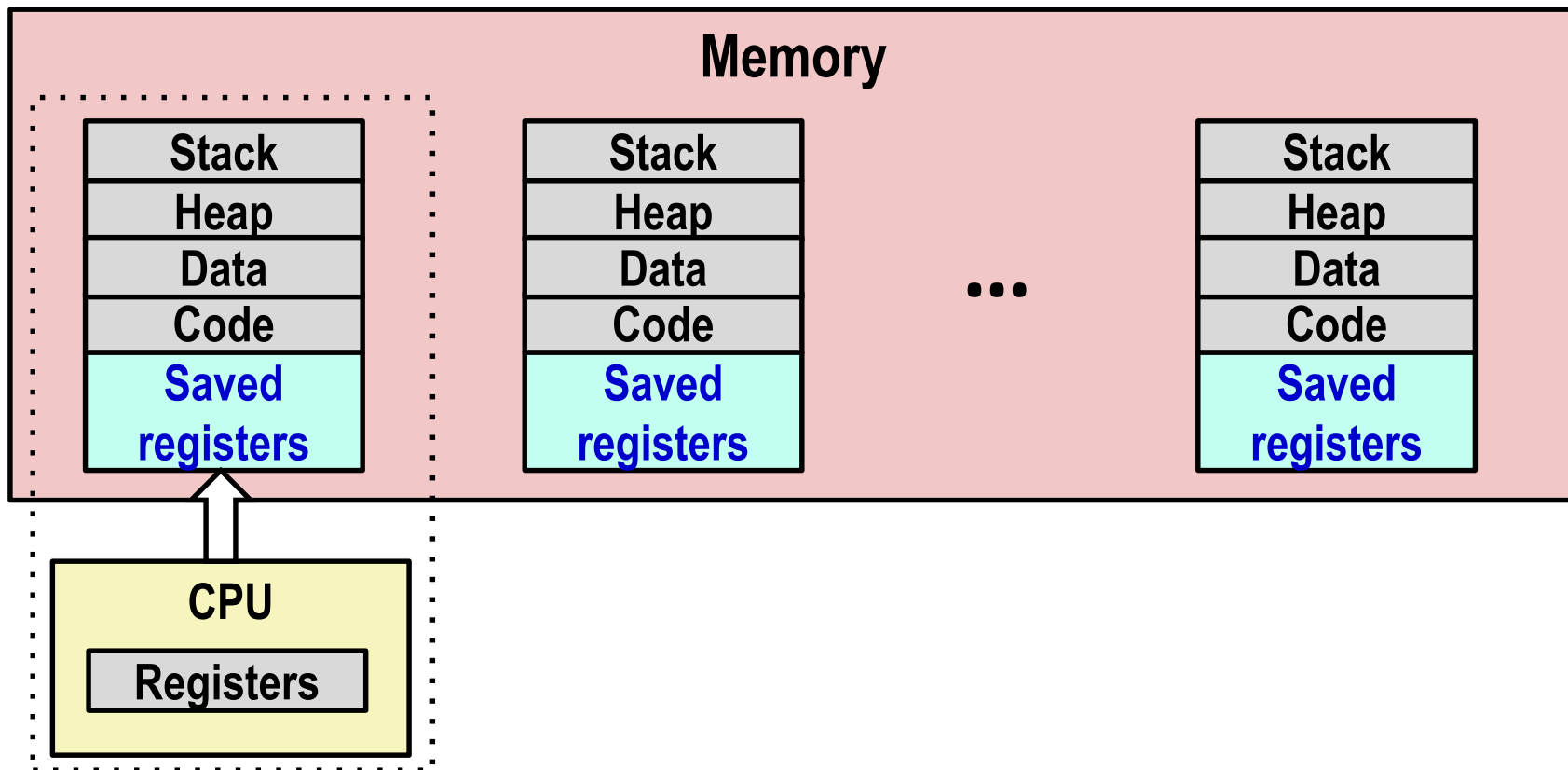
- 有185个进程，正在运行的有2个
- 以进程ID (PID)相区分

# 多重处理的真相



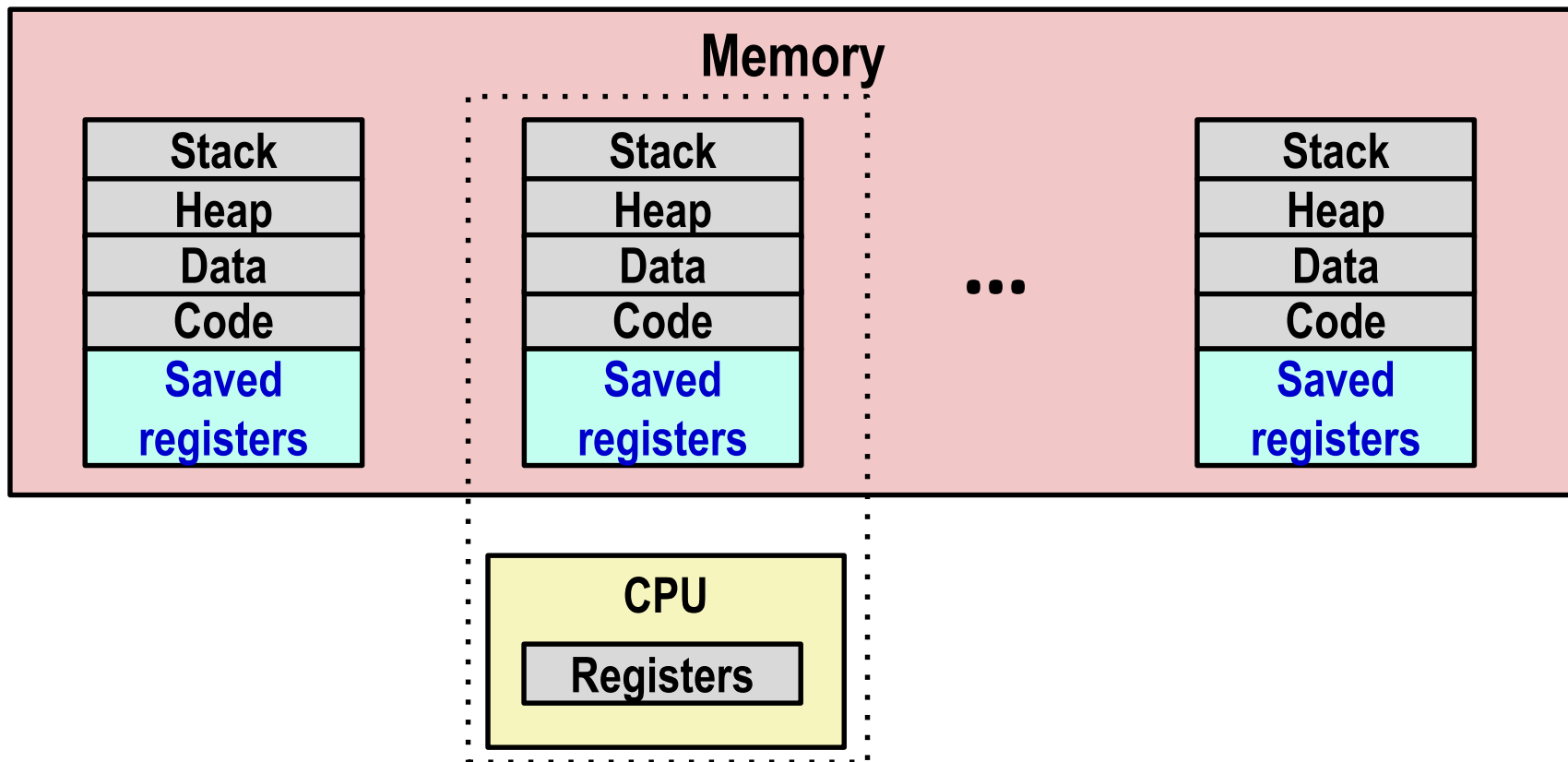
- 单处理器在并发地执行多个进程
  - 进程交错执行(multitasking)
  - 地址空间由虚拟内存系统管理
  - 未执行进程的寄存器值保存在内存中

# 多重处理的真相



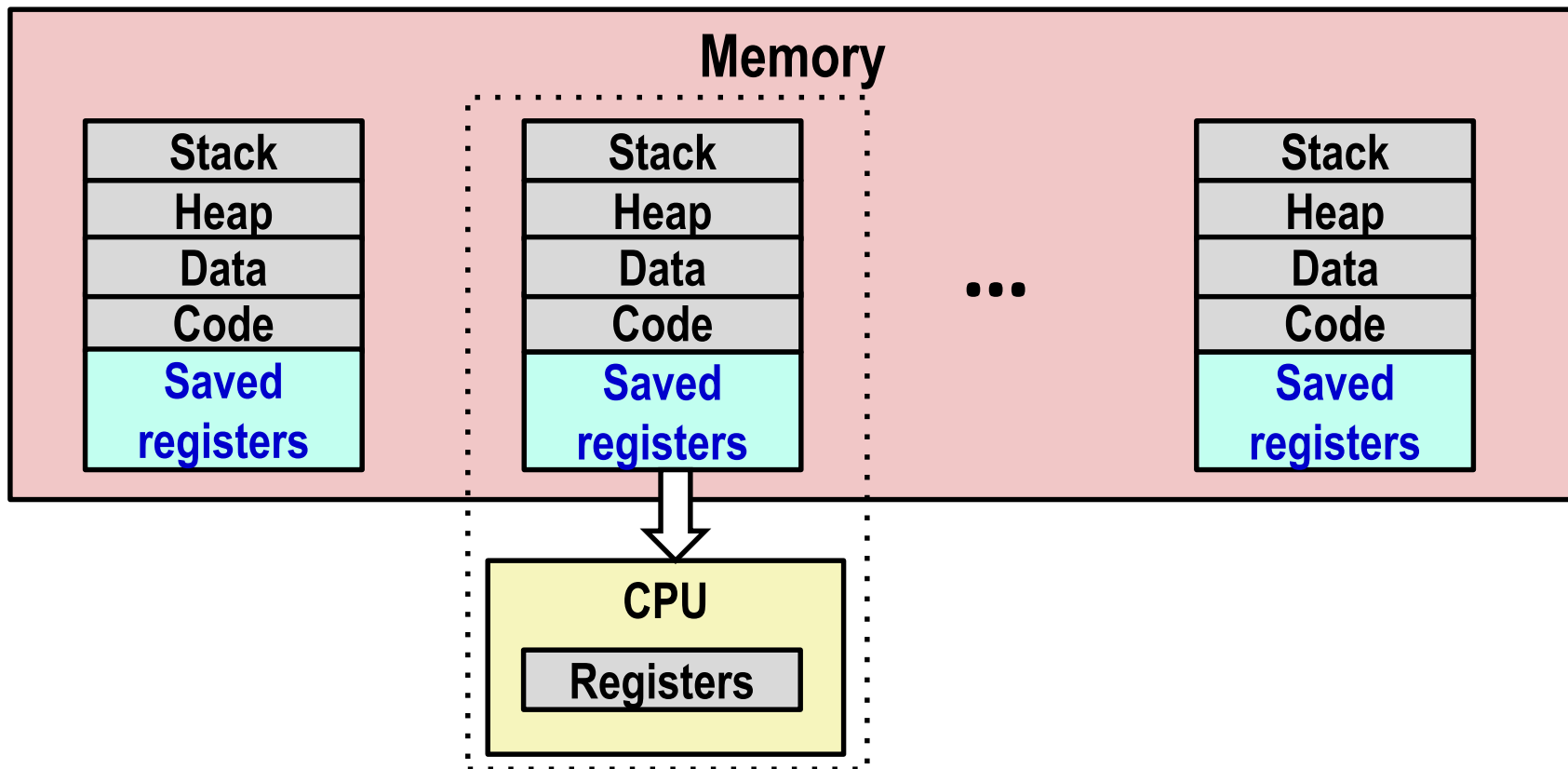
- 寄存器当前值保存到内存

# 多重处理的真相



- 调度下一个进程执行

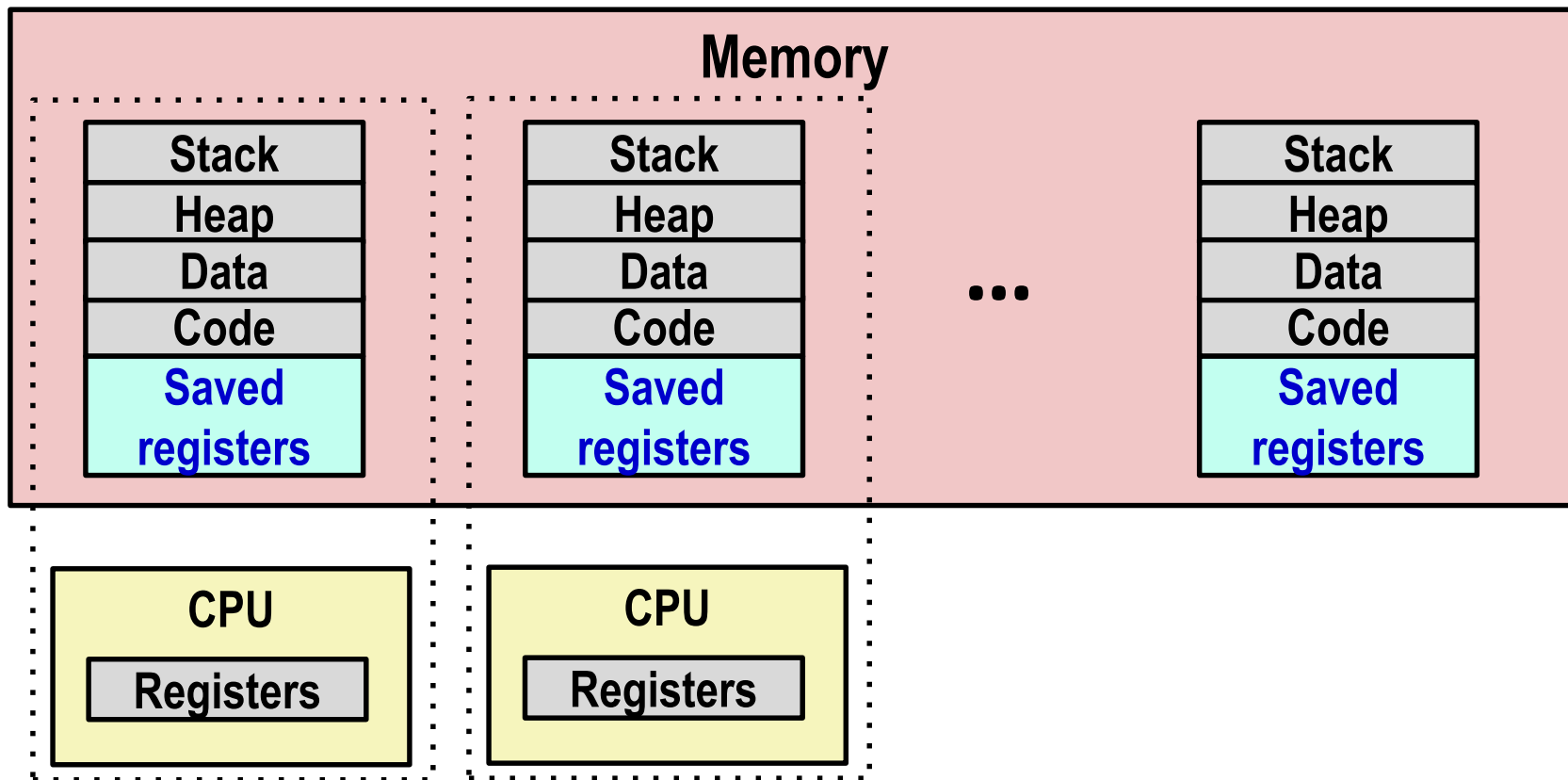
# 多重处理的真相



- 上下文切换: 装载保存的寄存器、切换地址空间
- context switch: Load saved registers and switch address space



# 多重处理的真相

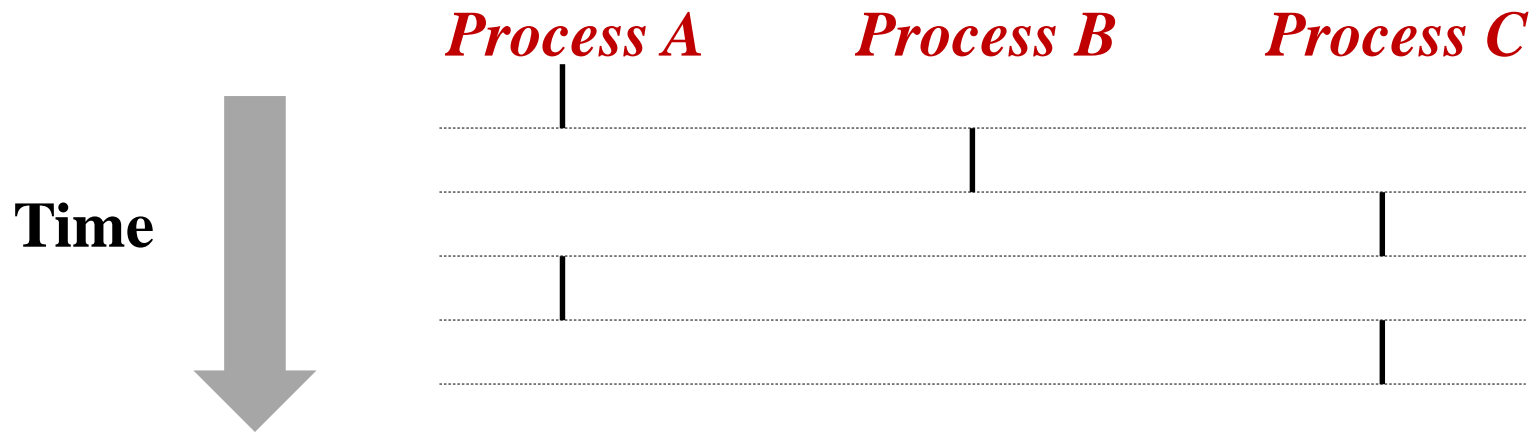


## ■ 多核处理器

- 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)

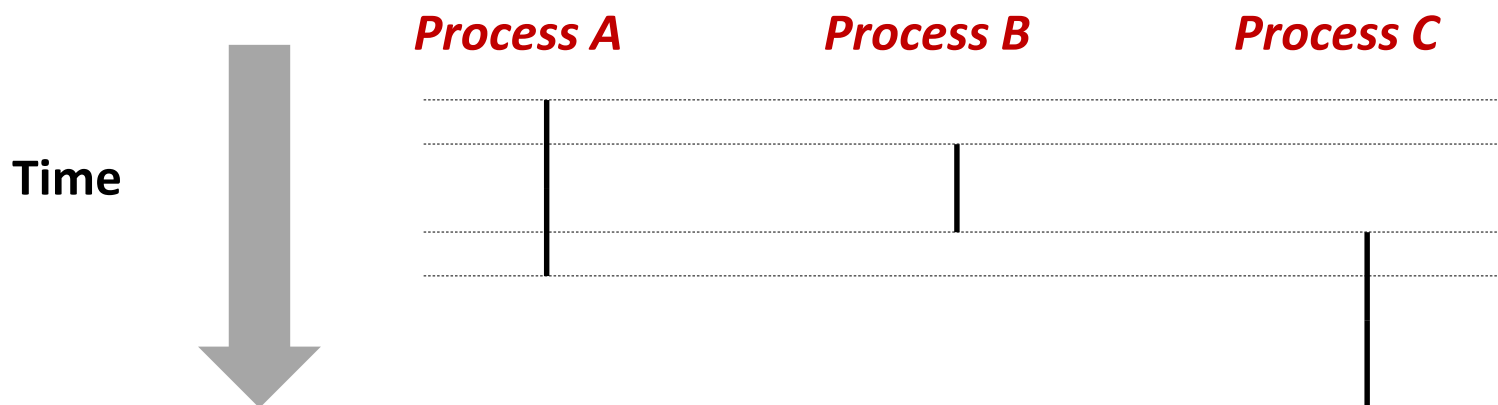
- 每个进程是个逻辑控制流
- 如果两个逻辑流在时间上有重叠，则称这两个进程是并发的(并发进程)
- 否则他们是顺序的



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

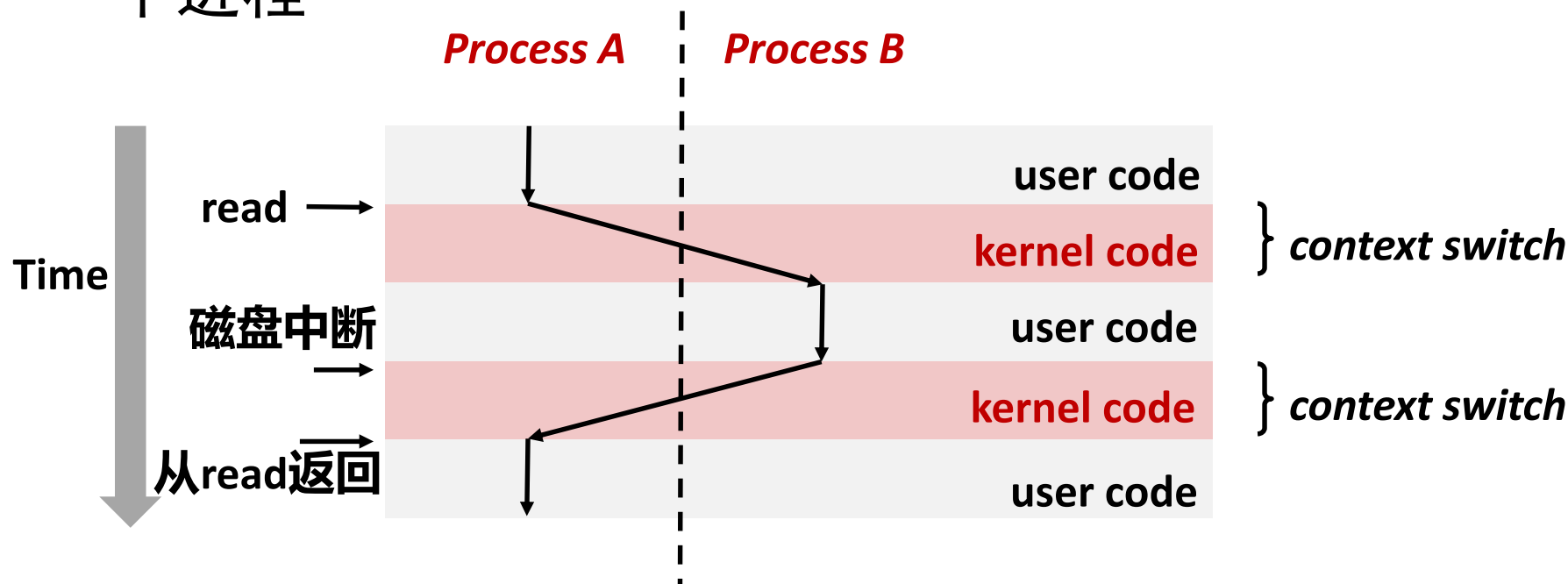
# 用户角度看并发进程

- 并发进程的控制流，在时间上是物理不相交的
- 然而，可将并发的进程看做是并行运行的



# 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
- 通过**上下文切换**，控制流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)

- 使用错误处理包装函数，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

- **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

从程序员的角度看，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(选中去执行)

## ■ 停止Stopped

- 进程被挂起且不会被调度，直至收到新的通知(如：信号 SIGCONT)

## ■ 终止Terminated：进程永远地停止（结束）了

# 终止进程

- 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
- 程序运行过程中，**exit** 函数只能被调用一次，且不返回到调用函数中。

# 创建进程

- 父进程通过调用**fork**函数创建一个新的、处于运行状态的子进程
- **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
    - 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()                                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);
}

```

```

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
- 复制但独立的地址空间
  - fork返回时，x在父进程和  
子进程中都为1
  - 之后，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
  - 在父、子进程中， stdout  
是相同的

# 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

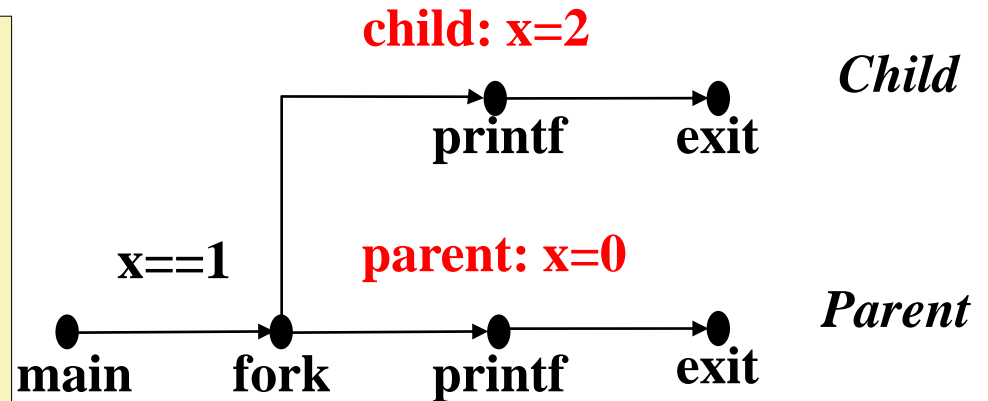
# 进程图

*fork.c*

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

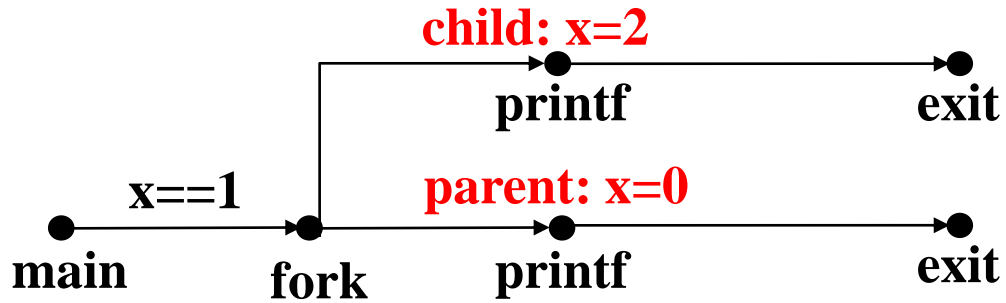
    pid = Fork();
    if (pid == 0) { /* Child */
        printf("child : x=%d\n", ++x);
        exit(0);
    }

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

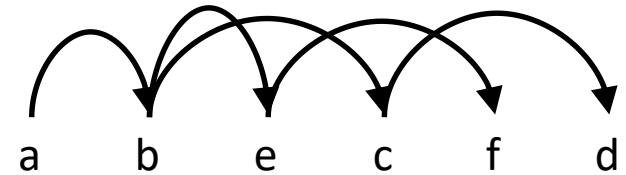


# 进程图的解读

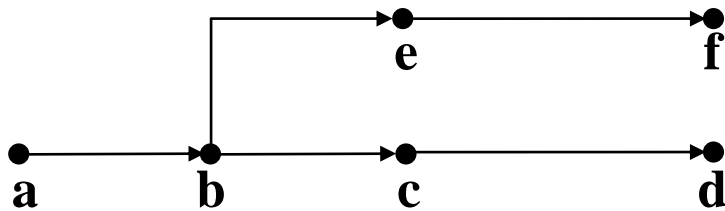
## ■ Original graph:



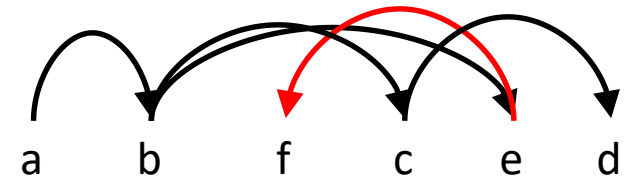
## 可行的全序排列



## ■ Relabled graph:



## 不可行的全序排列

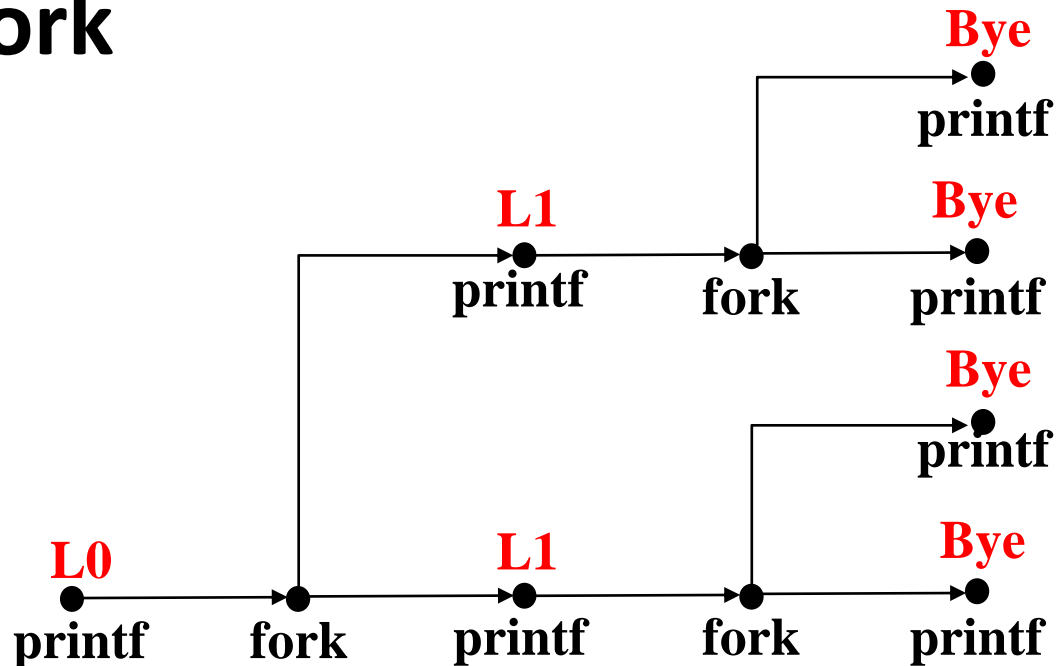




# 两个连续的fork

*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

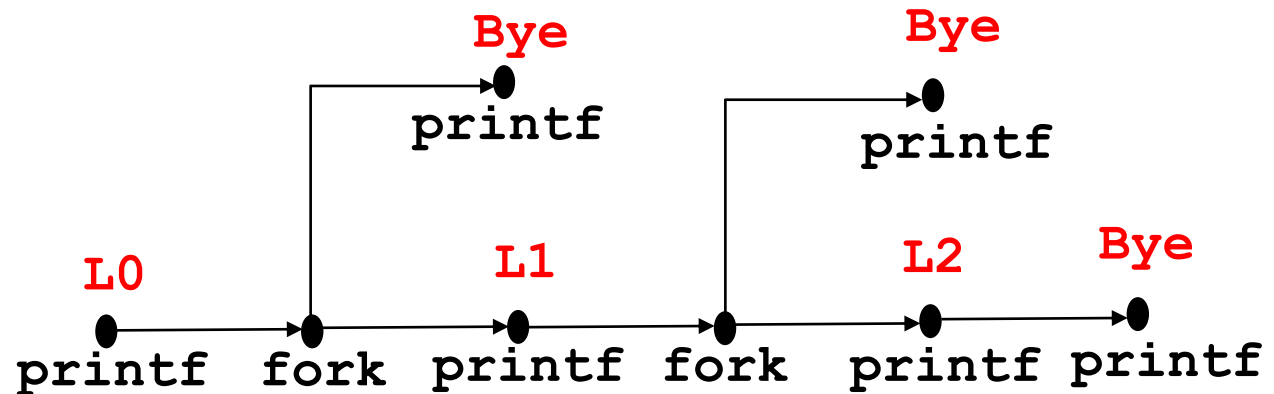
不可能的输出:

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

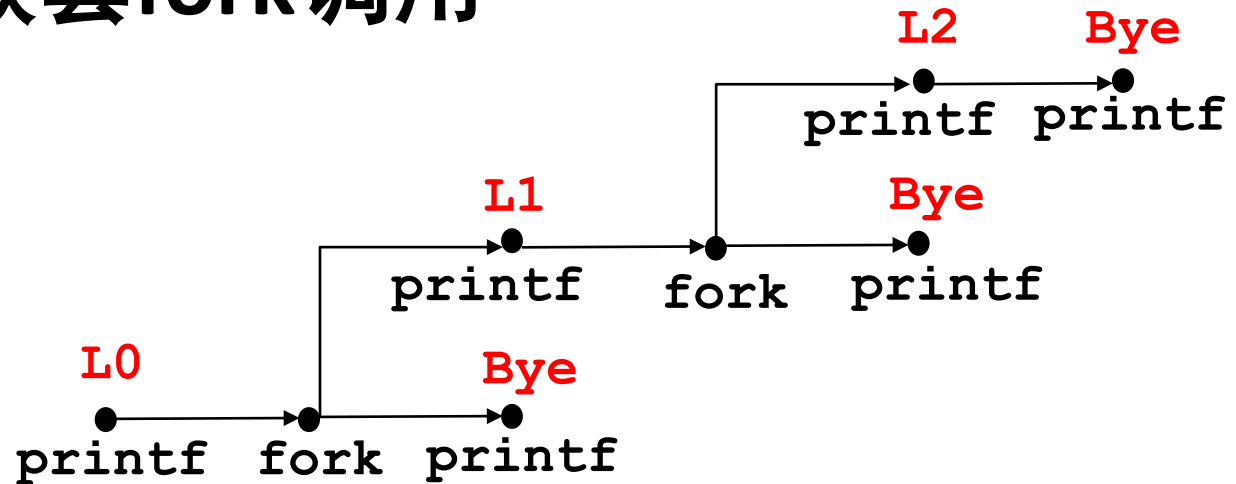
不可能的输出:

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

- 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

## ■ 回收 (Reaping)

- 父进程执行回收(使用函数wait或waitpid)
- 父进程收到子进程的退出状态
- 内核删掉僵死子进程

# 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> ./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命令显示：子进程的状态标记为“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	ttyp9	00:00:00	tcsh
<b>6676</b>	<b>ttyp9</b>	<b>00:00:06</b>	<b>forks</b>
6677	ttyp9	00:00:00	ps

父进程终止，但子进程仍处于活动状态

```
linux> kill 6676
```

必须明确地杀死子进程，否则将无限持续地运行

```
linux> ps
```

PID	TTY	TIME	CMD
6585	ttyp9	00:00:00	tcsh
6678	ttyp9	00:00:00	ps

# 与子进程同步: `wait`

## ■ 父进程通过`wait`函数回收子进程

`int wait(int *statusp)`

`wait(&status)`等价于: `waitpid(-1,&status,0)`

- 挂起当前进程的执行直到它的一个子进程终止
- 返回已终止子进程的 `pid`
- 如 `child_status != NULL`, 则在该指针指向的整型量表明子进程终止原因和退出状态信息:
  - 用`wait.h`头文件中定义的宏函数来检查, 例如:  
`WIFEXITED(status)`、`WEXITSTATUS(status)`、  
`WIFSIGNALED(status)`、`WTERMSIG(status)`、  
`WIFSTOPPED(status)`、`WSTOPSIG(status)`、  
`WIFCONTINUED(status)`



# 与子进程同步：wait示例1

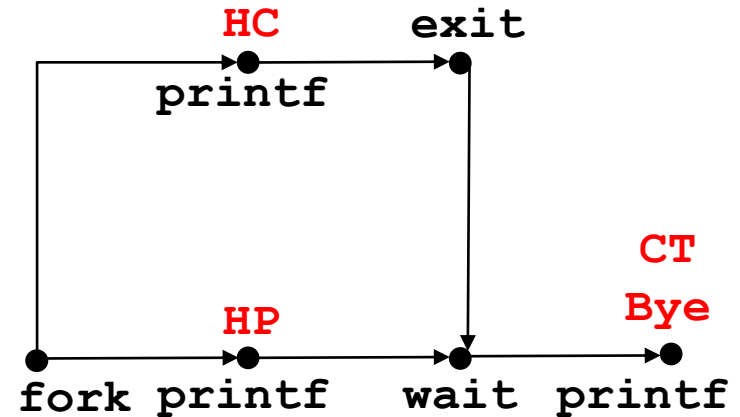
```

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() {
    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

- `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
  - `pid`  
`pid > 0`: 等待集合是该`pid`指定的进程  
`pid = -1`: 所有子进程  
`pid < -1`时, 等待ID为`|pid|`的进程组中的任何子进程  
`pid = 0`时, 等待同一个进程组中的任何子进程
  - `wait(&status)` 等价于 `waitpid(-1, &status, 0)`;
  - `option`: 可以是0, 或者以下选项的(或运算`|`)组合
    - ✓ `WNOHANG`: 若无子进程结束, 也会立即返回(返回值0)。不会挂起当前进程。
    - ✓ `WUNTRACED`: 若等待集合中的一个进程被停止, 也返回。  
(`waitpid`默认只返回已终止的子进程)
    - ✓ `WCONTINUED`: 若等待集合中的一个进程接收到`SIGCONT`而从停止状态重新开始, 也返回。

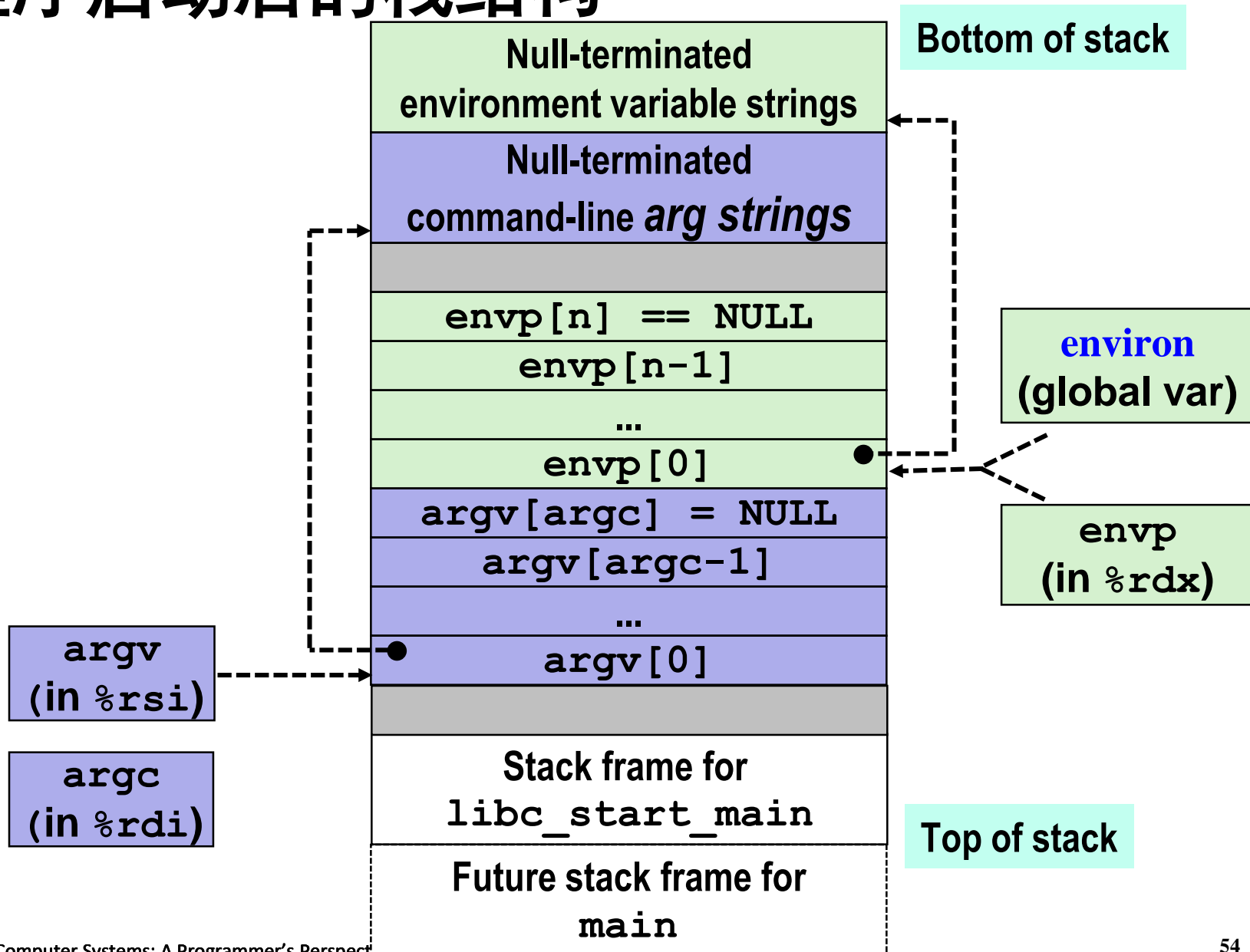
# waitpid: 等待特定进程

```
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);
    } }
```

## execve : 加载并运行程序

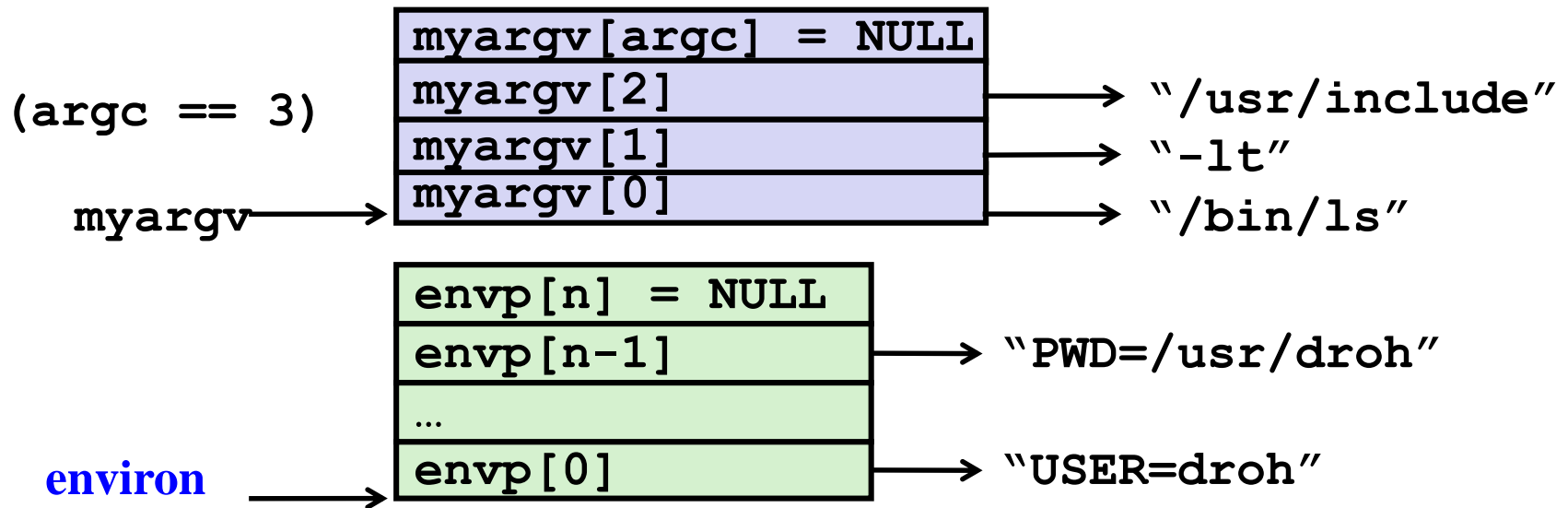
- **int execve(char \*filename, char \*argv[], char \*envp[])**
- 在当前进程中载入并运行程序:
  - **filename**: 可执行文件
    - 目标文件或脚本(用#!指明解释器,如 #!/bin/bash)
  - **argv**: 参数列表, 惯例: **argv[0]==filename**
  - **envp**: 环境变量列表
    - "name=value" strings (e.g., USER=droh)
    - getenv, putenv, printenv
- **覆盖当前进程的代码、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

# 新程序启动后的栈结构



# 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);
    }
}

```

# 总结

## ■ 异常Exceptions

- 需要使用非常规控制流的事件
- 外部产生——中断
- 内部产生——陷阱、故障

## ■ 进程Processes

- 任何给定的时间，系统中都有多个活动的进程
- 但是，在单个内核上，一个时刻只能执行一个进程
- 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

## ■ 进程的结束

- `exit`: 1次调用、0次返回

## ■ 回收和等待进程

- 调用函数`wait`或`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)