异常控制流

100076202： 计算机系统导论

任课教师：

计卫星　　宿红毅　　张艳

原作者：

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

内容提纲

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

控制流/Control Flow

- **处理器只做一件事/Processors do only one thing:**
  - 从开机到关机，CPU只是读入和执行（解释）指令序列，每次一条
    From startup to shutdown, a CPU 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

时间/Time

<startup>
$\textbf{inst}_1$
$\textbf{inst}_2$
$\textbf{inst}_3$
…
$\textbf{inst}_n$
<shutdown>

改变控制流/Altering the Control Flow

- **目前的两种方式/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

  是对程序状态改变的响应/React to changes in ***program state***
- **对有用系统来说还不够/Insufficient for a useful system:**
  难以对系统状态的改变进行响应/Difficult to react to changes in *system state*
  - 从磁盘或者网络获取的数据到达/Data arrives from a disk or a network adapter
  - 指令除零/Instruction divides by zero
  - 用户按下了Ctrl-C/User hits Ctrl-C at the keyboard
  - 时钟超时触发/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

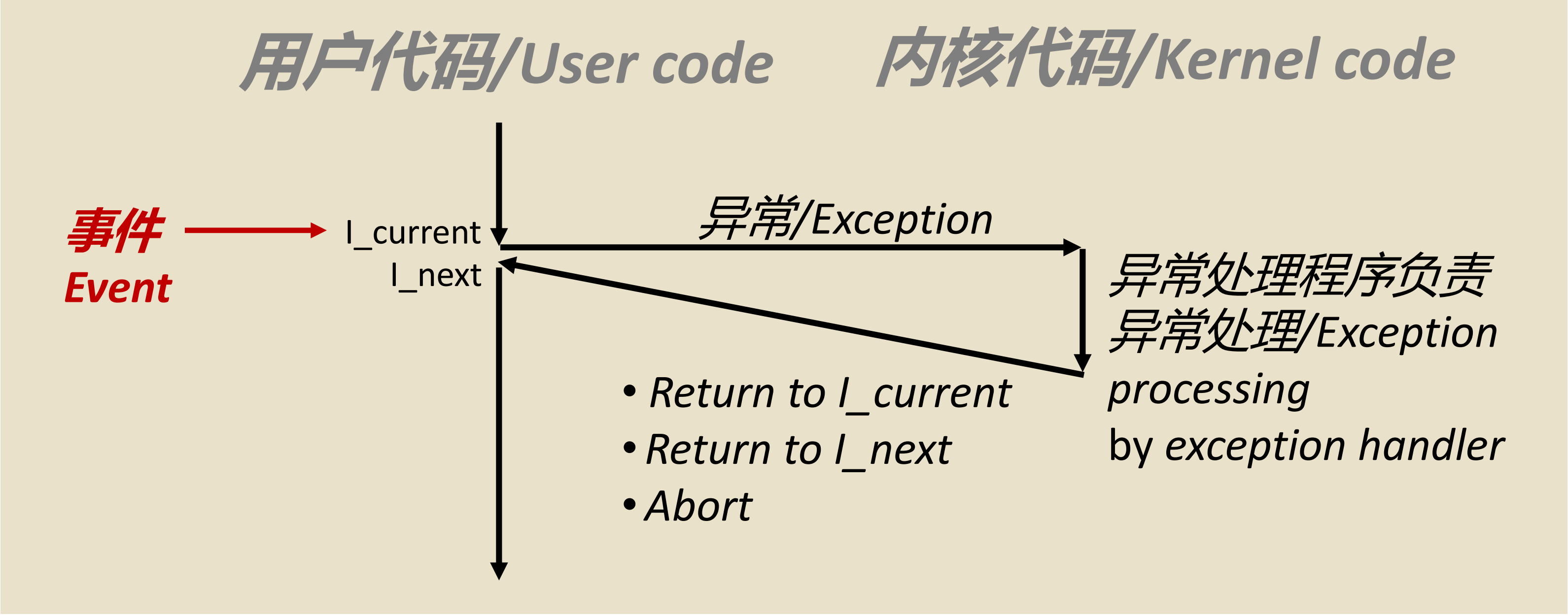
- **存在系统的每个层次 / Exists at all levels of a computer system**
- **低层次机制 / Low level mechanisms**
  - 1.异常 / **Exceptions**
    - 为响应系统事件改变控制流（例如，系统状态改变）/ Change in control flow in response to a system event (i.e., change in system state)
    - 硬件和OS软件组合实现 / Implemented using combination of hardware and OS software
- **高层次机制/Higher level mechanisms**
  - 2. 进程上下文切换 / **Process context switch**
    - 硬件时钟和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

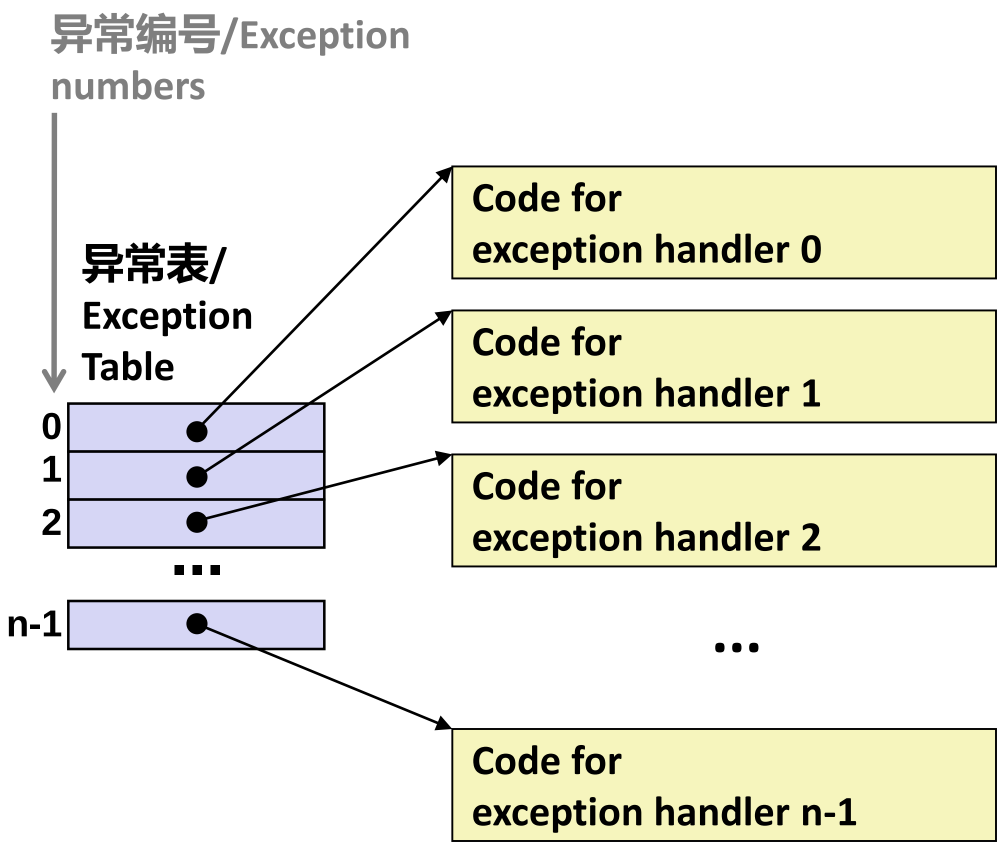
- 异常控制流 Exceptional Control Flow
- **异常 Exceptions**
- 进程 Processes
- 进程控制 Process Control

异常/Exceptions

- **异常是为了响应某些事件而将控制转移到OS内核（例如，处理器状态改变）/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：除零错误、算术溢出、缺页中断、I/O请求完成、Ctrl-C输入/Ctrl+C Examples of events: Divide by 0, arithmetic overflow, page fault, I/O request completes, typing Ctrl-C



异常表格 Exception Tables



- **每个事件类型有一个编号**
  Each type of event has a unique exception number k
- k = index into exception table (a.k.a. interrupt vector)
- Handler k is called each time exception k occurs

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**
  - 通过处理器的中断管脚给出/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
    - 大约几毫秒，外部时钟芯片触发/Every few ms, an external timer chip triggers an interrupt
    - 将控制权从用户切换到内核/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
    - 键盘输入Ctrl-C/Hitting Ctrl-C at the keyboard
    - 网络数据包到达/Arrival of a packet from a network
    - 磁盘数据到达/Arrival of data from a disk

同步异常/Synchronous Exceptions

- **指令执行导致的异常事件/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、保护异常、浮点异常/Examples: page faults (recoverable), protection faults (unrecoverable), floating point exceptions
    - 重新执行或者终止执行/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、校验错误、机器检查/Examples: illegal instruction, parity error, machine check
    - 终止当前程序执行/Aborts current program

系统调用/System Calls

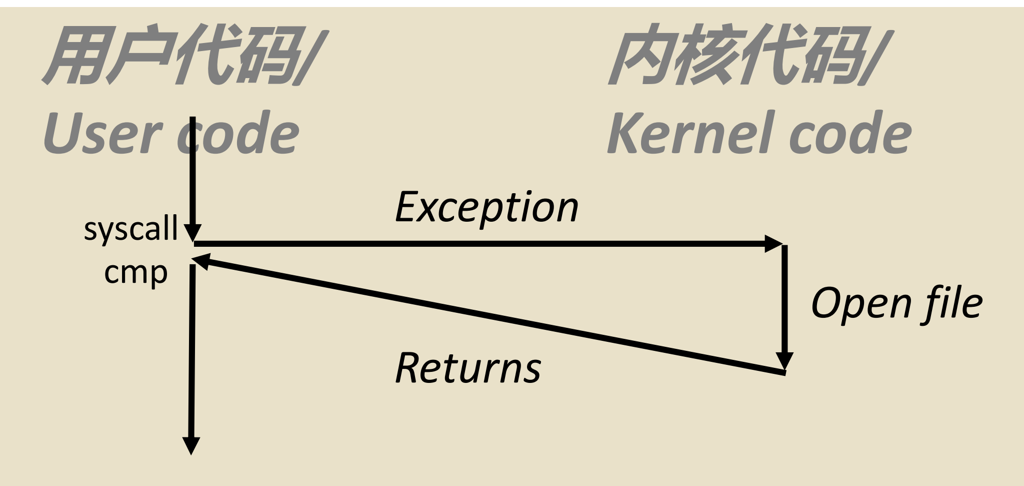
- **每个x86-64系统调用都有一个唯一的ID编号/Each x86-64 system call has a unique ID number**
- **例如：/Examples:**

编号 /Number	名称 /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`	获取文件信息/Get info about file
57	`fork`	创建进程/Create process
59	`execve`	执行程序/Execute a program
60	`_exit`	终止进程/Terminate process
62	`kill`	给进程发送信号/Send signal to process

系统调用举例：打开文件/System Call Example: Opening File

- 调用接口/User calls: **`open(filename, options)`**
- 调用__open函数，会执行系统调用指令syscall / Calls **`__open`** function, which invokes system call instruction **`syscall`**

```
00000000000e5d70 <__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  $0xfffffffffffff001,%rax
...
e5dfa:   c3                  retq
```



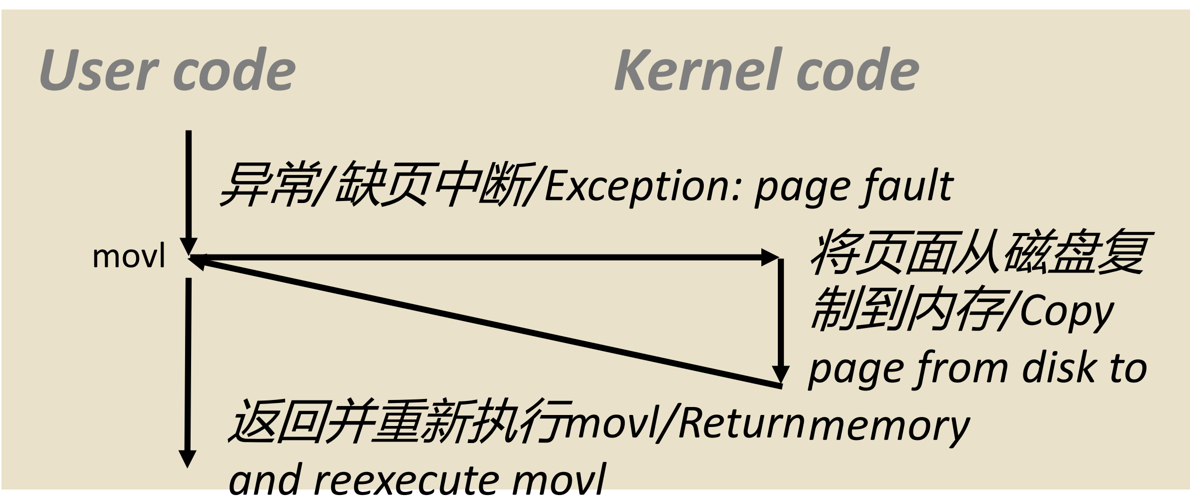
- `%rax`包含了系统调用编号/`%rax` contains syscall number
- 其他的参数在/Other arguments in `%rdi`, `%rsi`, `%rdx`, `%r10`, `%r8`, `%r9`
- 返回值在%rax/Return value in `%rax`
- 负数表示出错对应errno/Negative value is an error corresponding to negative `errno`

故障举例：缺页异常 /Fault Example: Page Fault

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

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

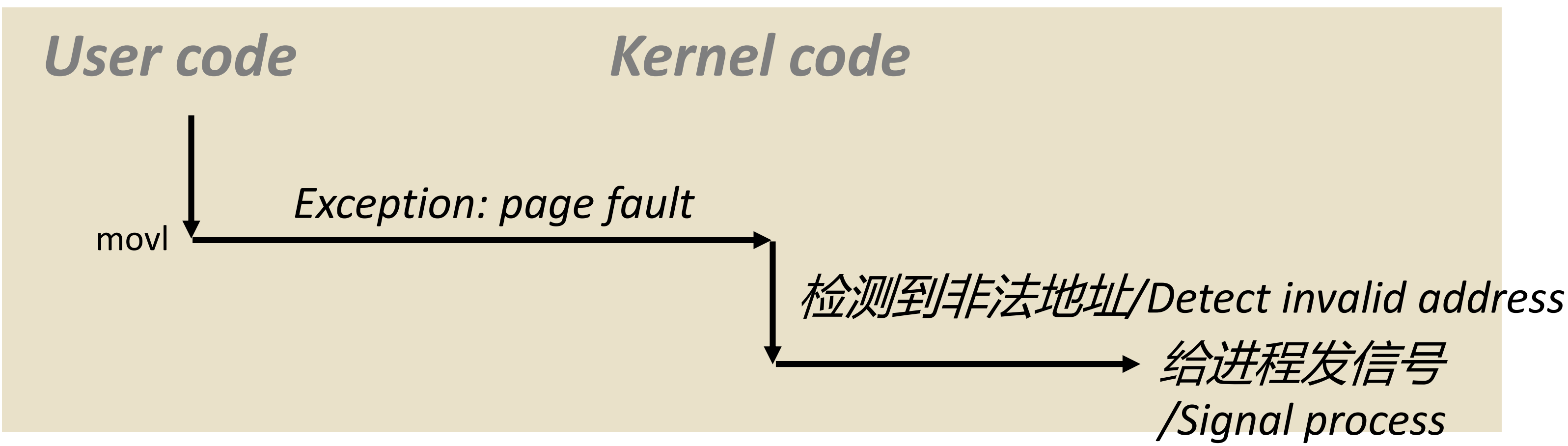
```
80483b7:        c7 05 10 9d 04 08 0d    movl    $0xd,0x8049d10
```



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
```



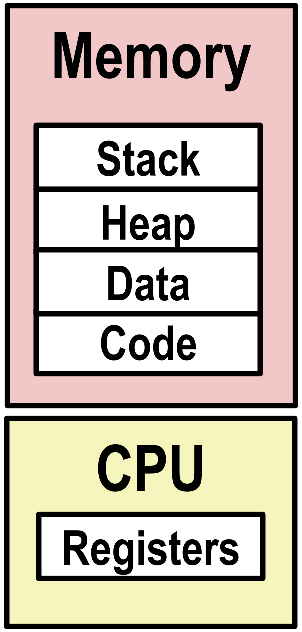
- 发送SIGSEG信号给用户进程/Sends **`SIGSEGV`** signal to user process
- 用户进程会异常退出/User process exits with “segmentation fault”

内容提纲

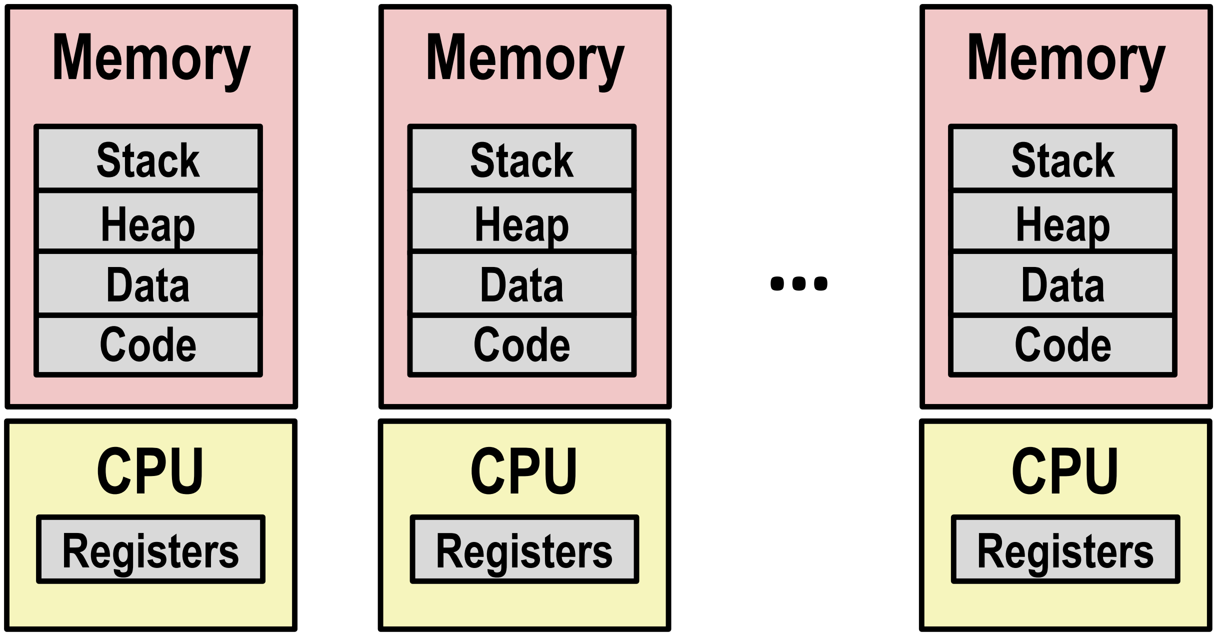
- 异常控制流 Exceptional Control Flow
- 异常 Exceptions
- 进程 Processes
- 进程控制 Process Control

进程/Processes

- **定义：进程是程序的运行的实例/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
  - 与程序和处理器不同/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.
    - 系统支持的虚拟内存/Provided by kernel mechanism called *virtual memory*



多进程幻象/Multiprocessing: The Illusion



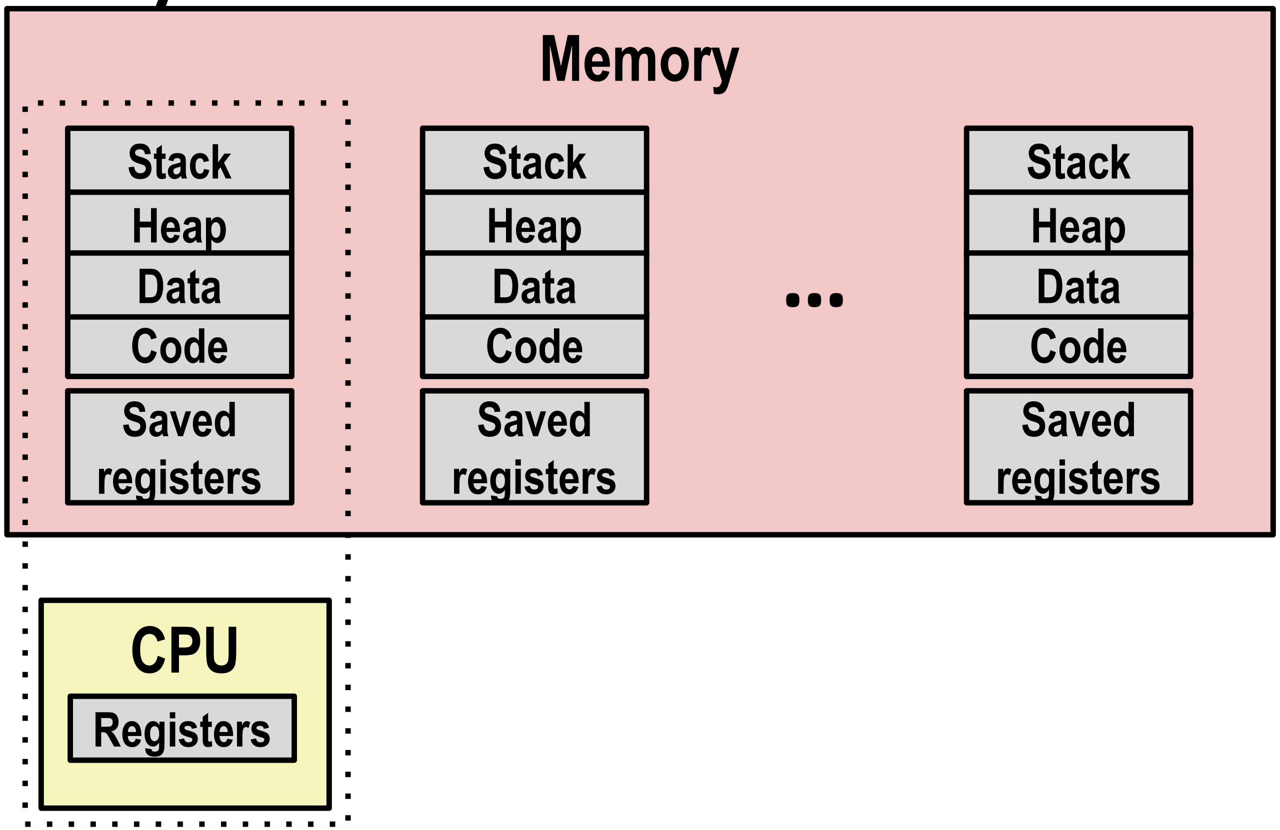
- **计算机同时运行很多进程/Computer runs many processes simultaneously**
  - 单个或多个用户的应用/Applications for one or more users
    - 网页浏览器、邮件客户端、编辑器/Web browsers, email clients, editors, …
  - 后台任务/Background tasks
    - 监控网络和I/O设备/Monitoring network & I/O devices

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
```

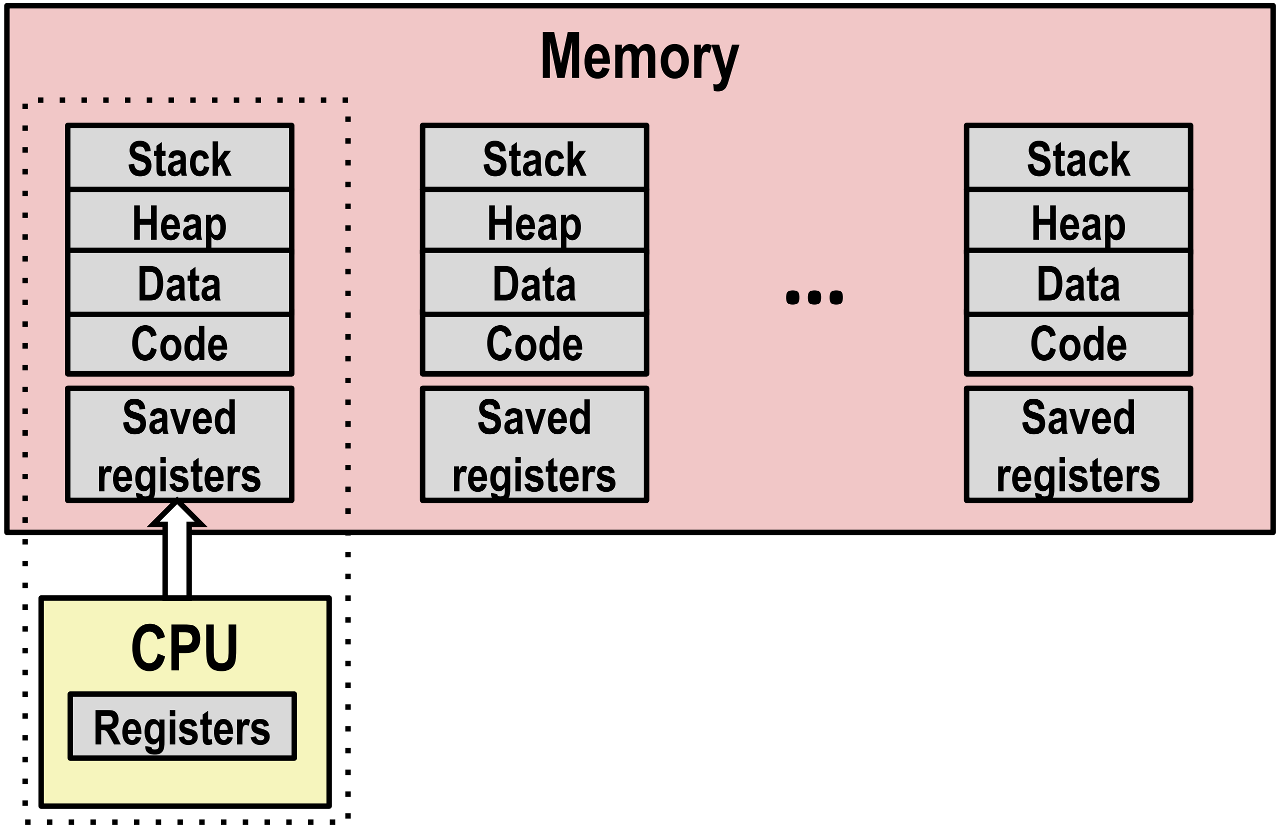
- 运行top命令 Running program “top” on Mac
  - 系统有123个进程，5个是活跃状态/System has 123 processes, 5 of which are active
  - 使用PID标识/Identified by Process ID (PID)

多进程(传统)真像/Multiprocessing: The (Traditional) Reality



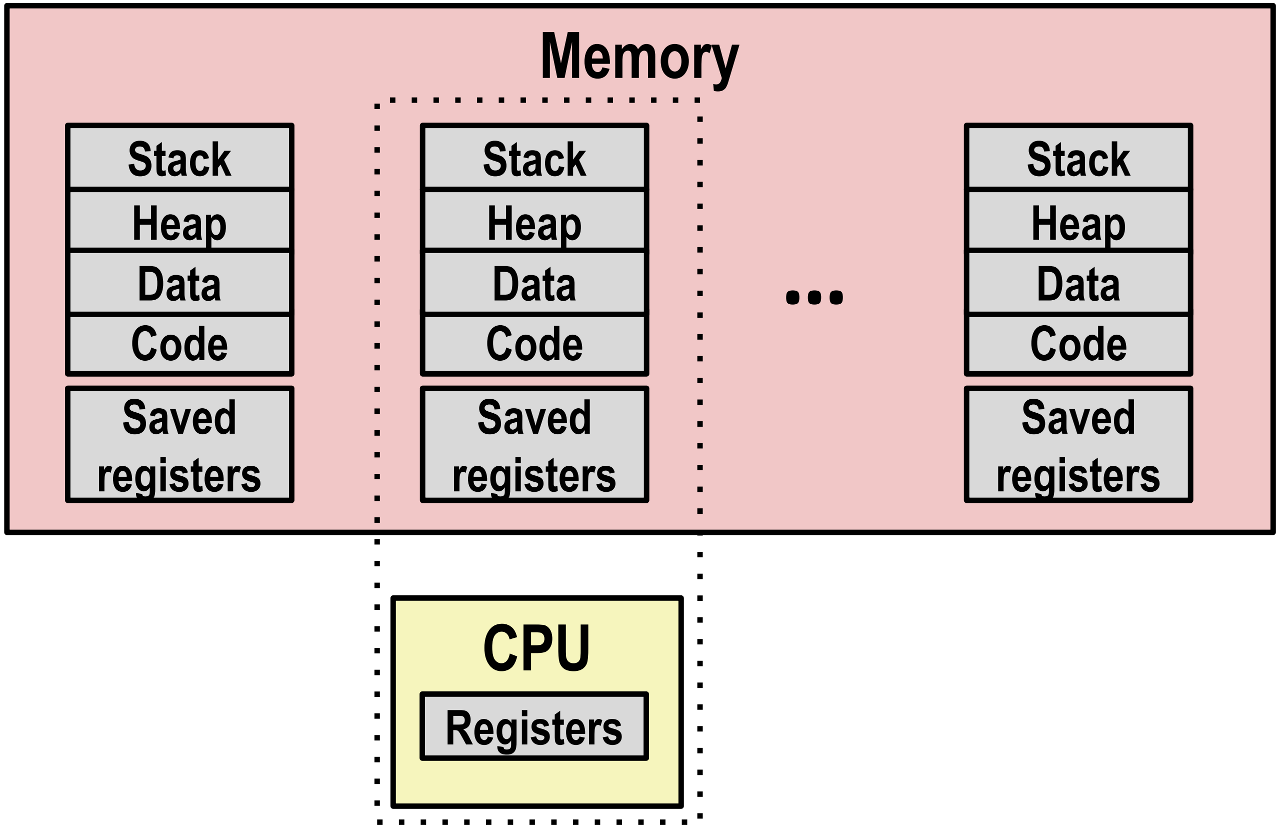
- **单个处理器并发执行多个进程/Single processor executes multiple processes concurrently**
  - 进程交替执行（多任务）/ Process executions interleaved (multitasking)
  - 地址空间由虚拟内存系统管理 /Address spaces managed by virtual memory system (later in course)
  - 非激活进程的寄存器值存储在内存中 /Register values for nonexecuting processes saved in memory

多进程真像/Multiprocessing: The (Traditional) Reality



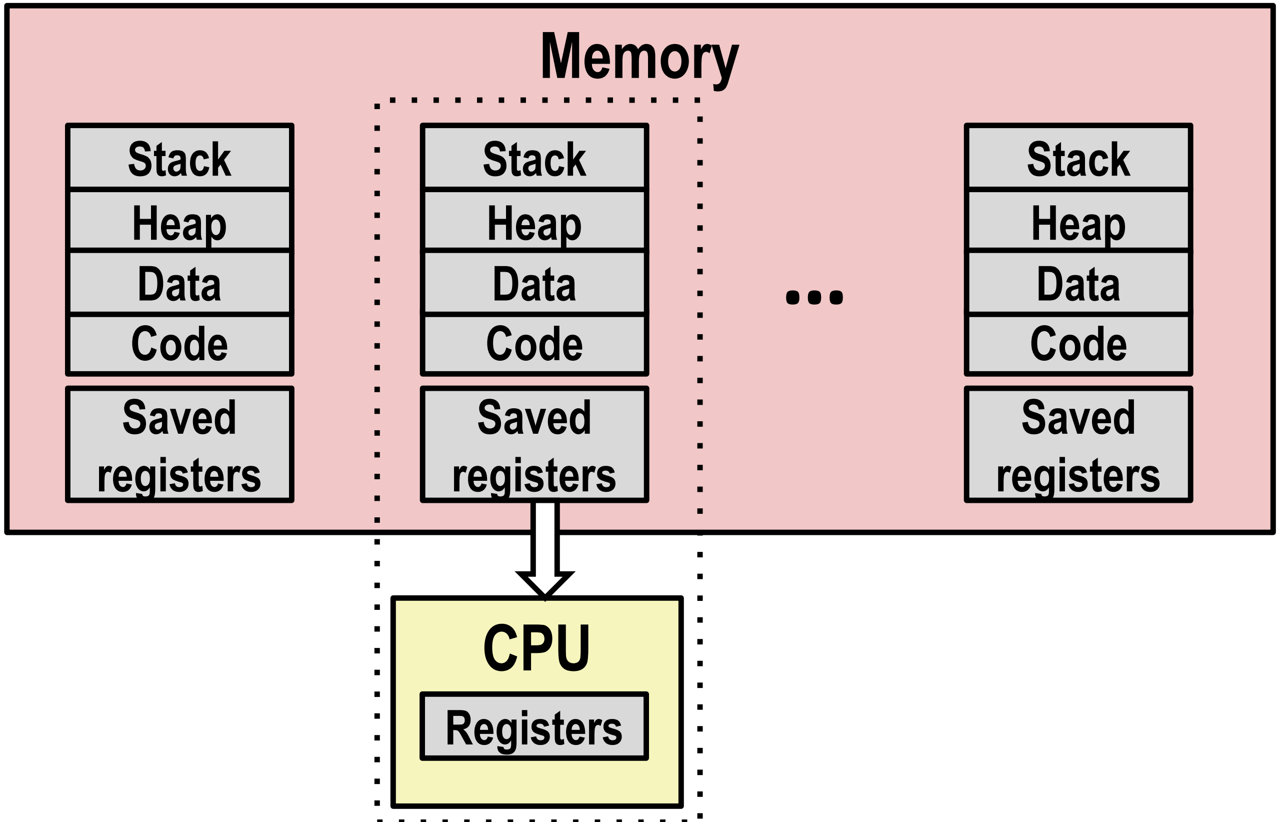
- 将当前寄存器存储在内存里/Save current registers in memory

多进程真像 Multiprocessing: The (Traditional) Reality



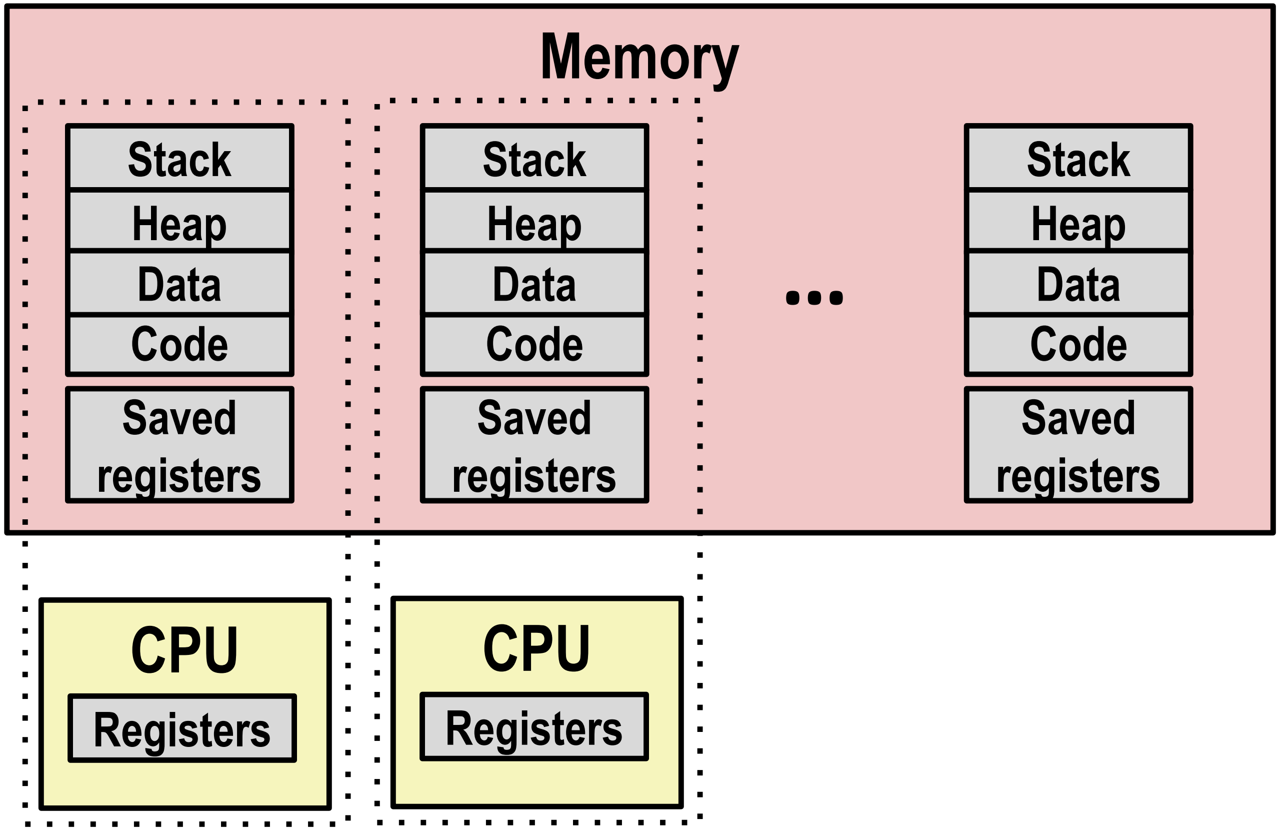
- 调度下一个进程执行/Schedule next process for execution

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)

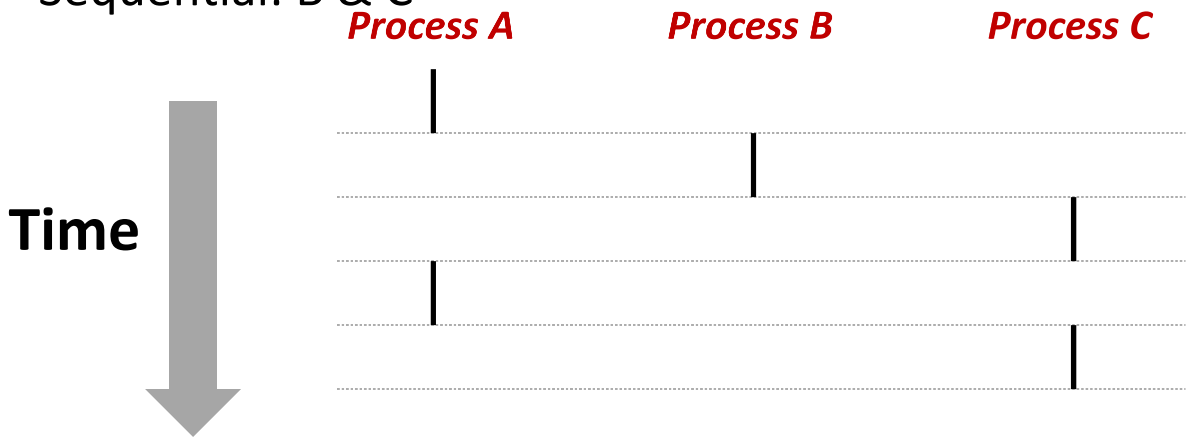
多进程真像 Multiprocessing: The (Modern) Reality



- **多处理器/Multicore processors**
  - 一个芯片上有多个CPU/Multiple CPUs on single chip
  - 共享主存储器（部分Cache）/Share main memory (and some of the caches)
  - 每个可以执行一个进程/Each can execute a separate process
    - 调度是由系统内核完成的/Scheduling of process onto cores done by kernel

并发进程/Concurrent Processes

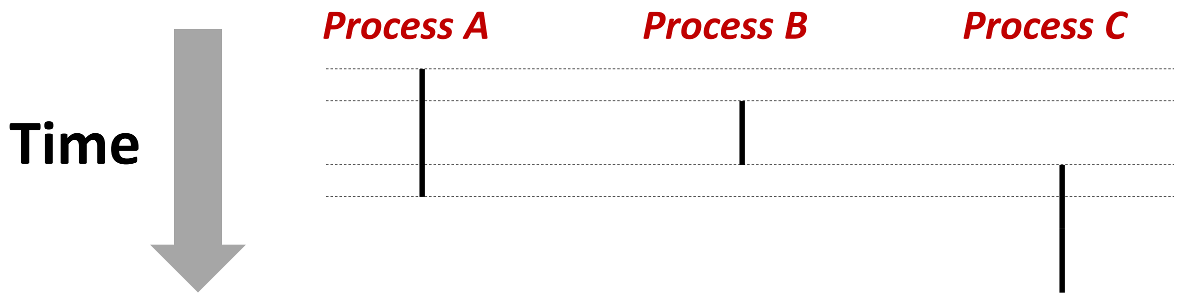
- **每个进程是一个逻辑控制流/Each process is a logical control flow.**
- **两个进程并发运行如果在时间上重叠/Two processes *run concurrently* (*are concurrent)* if their flows overlap in time**
- **否则是顺序执行/Otherwise, they are *sequential***
- **例如 (在单个核上运行) /Examples (running on single core):**
  - Concurrent: A & B, A & C
  - Sequential: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



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

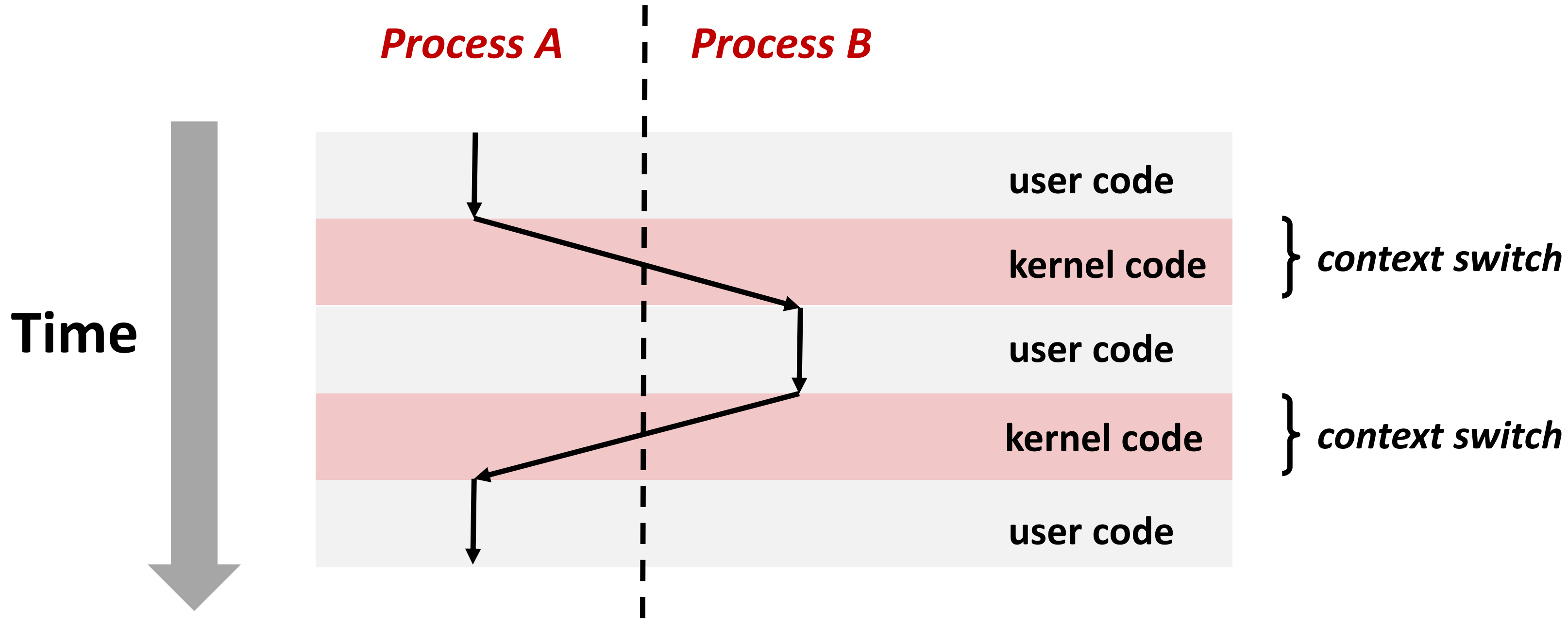
- **每个并发进程的控制流在时间上并不重叠/Control flows for concurrent processes are physically disjoint in time**

- **但是我们可以认为所有的并发进程都是并行执行的/However, we can think of concurrent processes as running in parallel with each other**



上下文切换/Context Switching

- **进程是由操作系统内核管理的/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**

系统调用错误处理/System Call Error Handling

- **出错时，系统函数返回-1并通过全局变量errno给出出错原因/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:**
  - 每次系统调用要检查调用结果/You must check the return status of every system-level function
  - 返回值为void的除外/Only exception is the handful of functions that return `void`
- **例如/Example:**

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

错误报告函数/Error-reporting functions

- **可以使用错误报告函数简化处理/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");
```

错误处理包装函数/Wrappers Error-handling Wrappers

- **采用Stevens-style 的错误处理方式简化代码/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();
```

Stevens-style： https://www.superfrink.net/docs/debugging-intro.html

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三种状态之一/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，最终会被内核调度/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终止 Process becomes terminated for one of three reasons:**
  - 收到一个信号，这个信号的默认多种是终止/ Receiving a signal whose default action is to terminate (next lecture)
  - 从主函数返回 Returning from the `main` routine
  - 调用exit函数 Calling the `exit` function
- **`void exit(int status)`**
  - 以某个状态status终止/Terminates with an *exit status* of `status`
  - 惯例：返回0表示正常退出，其他表示出错/Convention: normal return status is 0, nonzero on error
  - 也可以通过main函数的返回值显式设置进程退出状态/Another way to explicitly set the exit status is to return an integer value from the main routine
- **`exit`只会被调用一次且会不返回/`exit` is called once but never returns.**

创建进程/Creating Processes

- ***父进程通过fork创建一个新的进程 /Parent process* creates a new running *child process* by calling `fork`**
- `int fork(void)`
  - 对子进程返回0，对父进程返回子进程的PID/Returns 0 to the child process, child’s PID to parent process
  - 子进程几乎和父进程是一样的/Child is *almost* identical to parent:
    - 子进程具有和父进程一样但是独立的虚拟地址空间/Child get an identical (but separate) copy of the parent’s virtual address space.
    - 子进程会有和父进程一样的文件描述符拷贝/Child gets identical copies of the parent’s open file descriptors
    - 子进程和父进程的PID不同/Child has a different PID than the parent
- **`fork`是很有意思的（让人困惑的），因为只调用了一次但是返回两次/`fork` is interesting (and often confusing) because it is called *once* but returns *twice***

fork举例/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
```

- 一次调用，两次返回/Call once, return twice
- 并发执行/Concurrent execution
  - 不能预测父进程和子进程之间的执行顺序/Can't predict execution order of parent and child
- 重复的但是独立的地址空间/Duplicate but separate address space
  - `fork`时x的值是1/`x` has a value of 1 when fork returns in parent and child
  - 后续对x的修改都是独立的/Subsequent changes to `x` are independent
- 共享打开的文件/Shared open files
  - `stdio`在子进程和父进程中都是一样的/`stdout` is the same in both parent and child

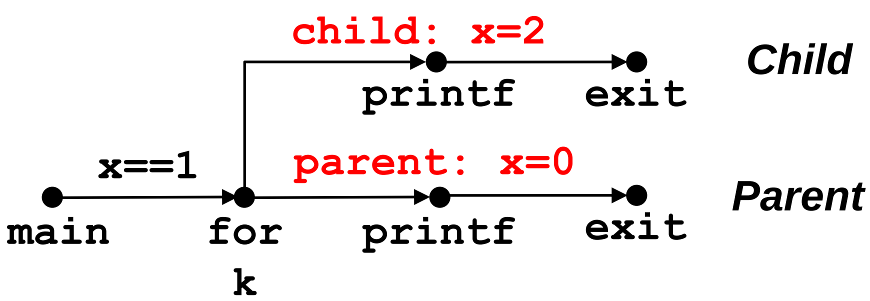
使用进程图描述fork /Modeling `fork` with Process Graphs

- **进程图是描述并发程序中语句偏序关系的有用工具/A *process graph* is a useful tool for capturing the partial ordering of statements in a concurrent program:**
  - 每个顶点是一个语句/Each vertex is the execution of a statement
  - a -> b 表示a在b之前发生/a -> b means `a` happens before b
  - 每个边可以用当前值或者变量进行标注/Edges can be labeled with current value of variables
  - `printf`节点可以用标注为`output`/`printf` vertices can be labeled with output
  - 每个图的开始节点没有入边/Each graph begins with a vertex with no inedges
- **图的任何拓扑排序都对应一个可行的全局序/Any *topological sort* of the graph corresponds to a feasible total ordering.**
  - 全局序节点之间的边从左指向右/Total ordering of vertices where all edges point from left to right

进程图举例/Process Graph Example

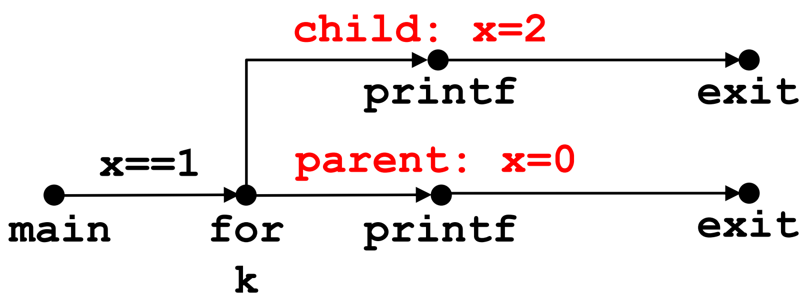
```c
int main()
{
    pid_t pid;
    int x = 1;

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

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

fork.c

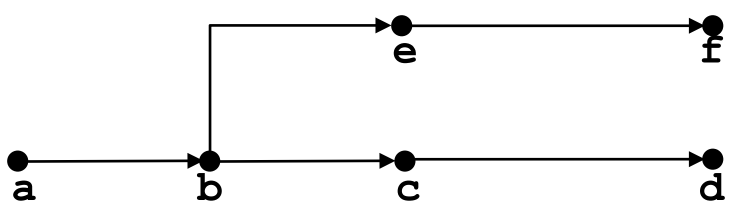


解释进程图/Interpreting Process Graphs

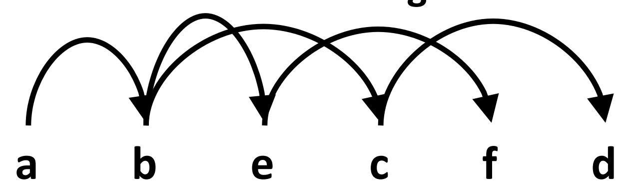
- **原图/Original graph:**



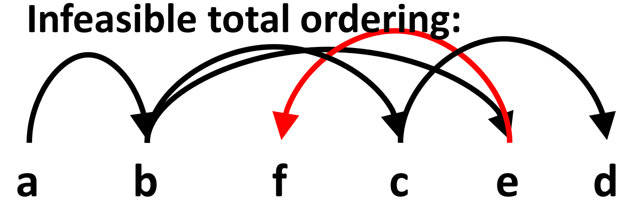
- **标注后的图/Relabled graph:**



可能的全局序/
Feasible total ordering:

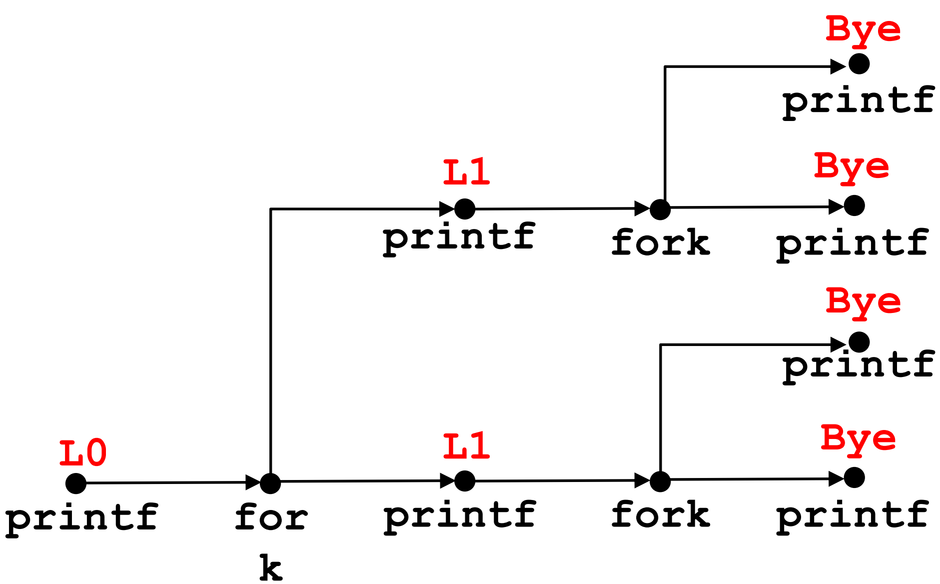


不可能的全局序/
Infeasible total ordering:



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

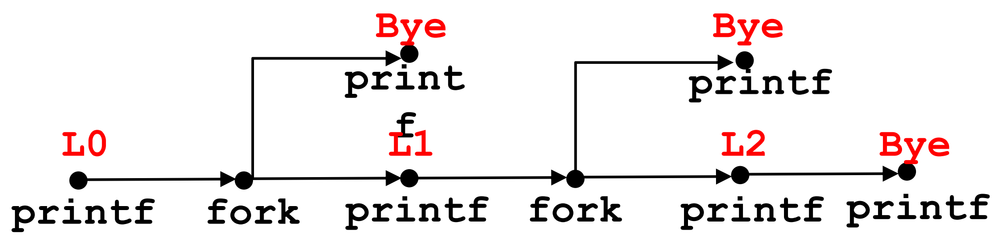


可能的输出/
Feasible output:
L0
L1
Bye
Bye
L1
Bye
Bye

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

```c
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:

L0
L1
Bye
Bye
L2
Bye

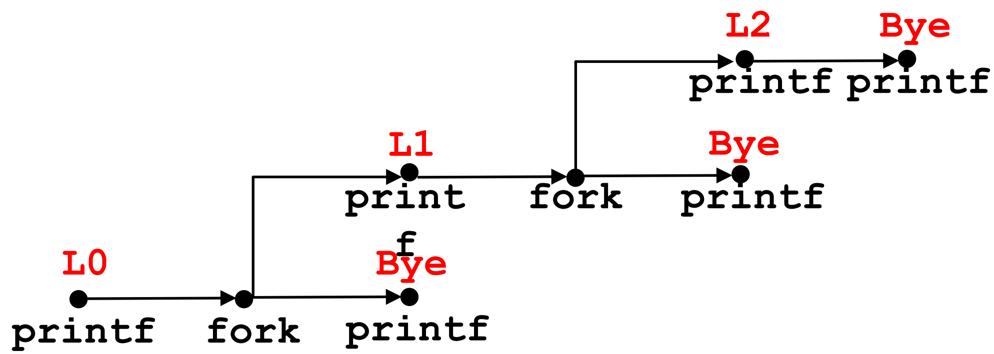
不可能的输出/ Infeasible output:

L0
Bye
L1
Bye
Bye
L2

fork举例，子进程嵌套fork/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

捕获子进程/Reaping Child Processes

- **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
  - 僵尸 Called a “zombie”
    - Living corpse, half alive and half dead
- **捕获 Reaping**
  - 父类进程等待子进程终止/Performed by parent on terminated child (using `wait` or `waitpid`)
  - 父类进程获得退出状态信息/Parent is given exit status information
  - 内核删掉僵尸子进程 Kernel then deletes zombie child process
- **如果父类进行没有回收会怎么样？What if parent doesn’t reap?**
  - 如果父类不回收，则由init进程回收 If any parent terminates without reaping a child, then the orphaned child will be reaped by **`init`** process (pid == 1)
  - 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
    - e.g., shells and servers

僵尸进程举例/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 */
    }
}
```

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
```

- **使用ps命令显式子进程是"defunct"/ps** shows child process as "defunct" (i.e., a zombie)
- 杀死父进程允许子进程通过init被重新捕获/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);
    }
}
```

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
 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
```

- 父进程终止后子进程仍然活跃/Child process still active even though parent has terminated
- 必须显式杀死子进程，否则子进程将永远运行下去/Must kill child explicitly, or else will keep running indefinitely

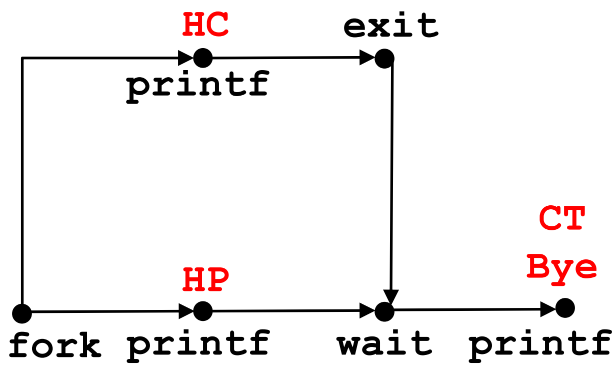
`wait`：子进程同步/`wait`: Synchronizing with Children

- **父进程通过wait函数重新对接子进程/Parent reaps a child by calling the `wait` function**
- `int wait(int *child_status)`
  - 挂起当前进程直到其中一个子进程退出/Suspends current process until one of its children terminates
  - 返回的是终止的子进程的PID/Return value is the **`pid`** of the child process that terminated
  - 如果**`child_status`**不为空，该指针指向的值表示的是子进程终止和退出的原因/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:
    - 可以使用wait.h中定义的下列宏进行检查/Checked using macros defined in `wait.h`
      - `WIFEXITED`, `WEXITSTATUS`, `WIFSIGNALED`, `WTERMSIG`, `WIFSTOPPED`, `WSTOPSIG`, `WIFCONTINUED`
      - See textbook for details

wait：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



Feasible output:
HC
HP
CT
Bye

Infeasible output:
HP
CT
Bye
HC

另一个wait的例子/Another wait Example

- 多个子进程可能按照任意顺序完成/If multiple children completed, will take in arbitrary order
- 可以使用WIFEXITED 和WEXITSTATUS获取退出的信息/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)`
  - 挂起当前进程直到特定进程终止/Suspends current process until specific process terminates
  - 很多不同选项（见教材）/Various options (see textbook)

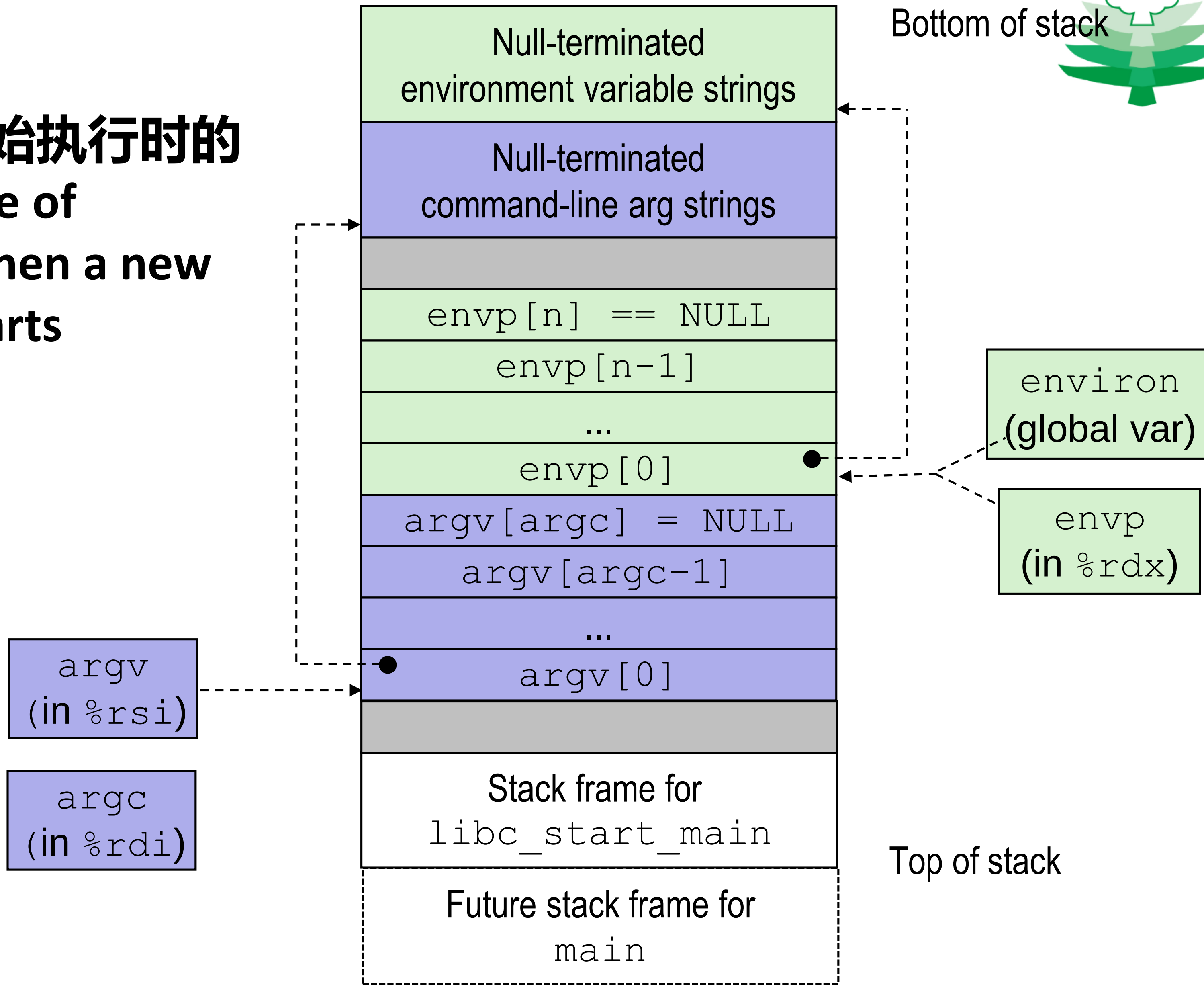
```
void fork11() {
    pid_t pid[N];
    int i;
    int child_status;

    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

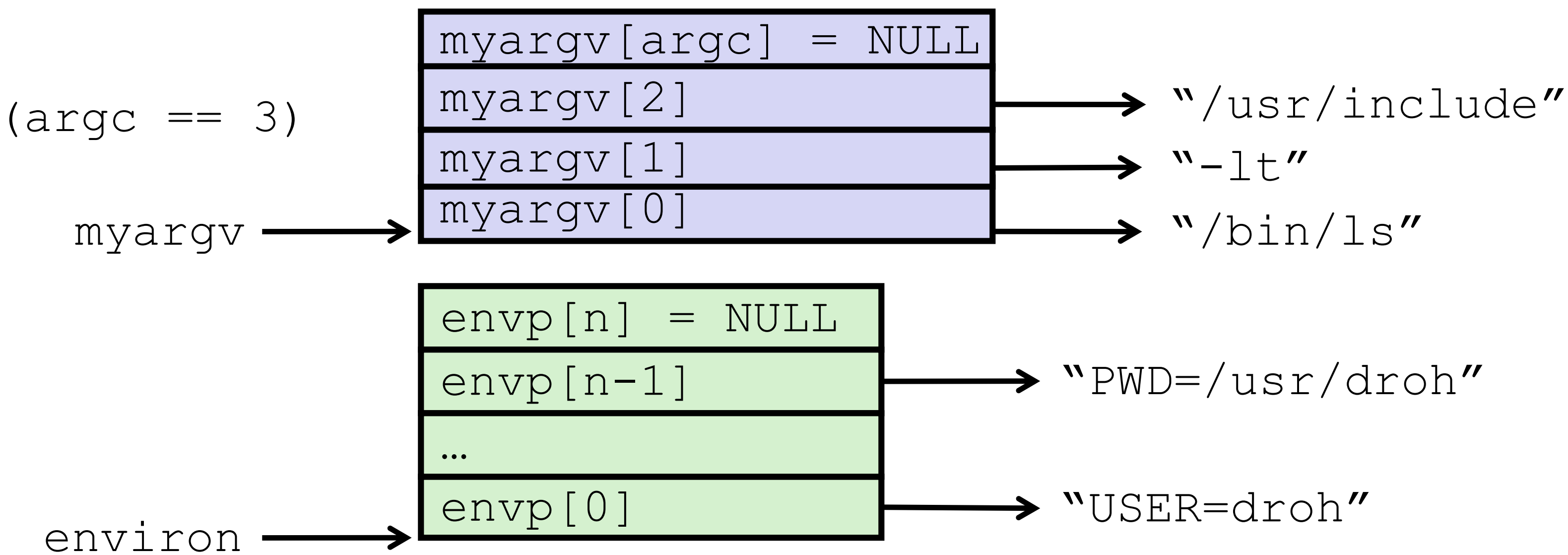
- `int execve(char *filename, char *argv[], char *envp[])`
- **在当前进程加载运行/Loads and runs in the current process:**
  - filename: Executable file **`filename`** / **可执行文件名字**
    - 目标代码文件或者以`#!interpreter`开头的脚本文件 Can be object file or script file beginning with `#!interpreter` (e.g., `#!/bin/bash`)
  - 带有参数列表/ …with argument list **`argv`**
    - 按惯例argv[0]是第一个参数filename/By convention **`argv[0]==filename`**
  - 带有环境变量列表/ …and environment variable list **`envp`**
    - "name=value" strings (e.g., `USER=droh`)
    - `getenv, putenv, printenv`
- **覆盖代码、数据和堆栈/Overwrites code, data, and stack**
  - 保留PID、打开的文件和信号上下文/Retains PID, open files and signal context
- **一次调用，并无返回/Called once and never returns**
  - …除非有错误/…except if there is an error

一个程序开始执行时的栈/Structure of the stack when a new program starts



execve举例/execve Example

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




```
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 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）生成 / Generated externally (interrupts) or internally (traps and faults)

- 进程/Processes
  - 任意时刻，系统有多个进程/At any given time, system has multiple active processes
  - 每个时刻单个核只能执行一个进程/Only one can execute at a time on a single core, though
  - 每个进程看起来是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/Summary (cont.)

- **生成新进程/Spawning processes**
  - 调用fokr/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，无返回/One call, no return
- **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/Call `execve` (or variant)
  - 一次调用，一般情况下无返回/One call, (normally) no return