进程控制

Process Control

本章内容

Topic

- 处理系统调用的错误
- 获取进程ID
- 创建和终止进程
- 回收子进程
- 进程休眠
- 加载并运行程序

处理系统调用的错误

System Call Error Handling

- 在错误发生时，Linux 系统调用通常返回 -1，并设置全局变量 errno 来指示原因。
- 务必要做到：
  - 检查每个系统级函数的返回状态。
  - 唯一的例外是那些少数返回 void 的函数。
- 示例：

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

Linux下的C语言代码特点：返回值用于传递执行的状态和简单结果，复杂的返回数据常通过传入的参数指针进行传递

处理系统调用的错误

System Call Error Handling

错误报告函数

- 可以使用一个错误报告函数来简化一些操作

```
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");
```

处理系统调用的错误

System Call Error Handling

包装一下错误处理

- 通过使用 Stevens 风格的错误处理包装函数，我们进一步简化了代码：

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

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

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

Stevens风格，函数原型相同，首字母由小写变为大写，表示函数经过了包装。

Linux中，命名风格常用小写字母和下划线的组合

W. R. Stevens, B. Fenner, and A. M. Rudoff. *Unix Network Programming: The Sockets Networking API, Third Edition*, volume 1. Prentice Hall, 2003.

本章内容

Topic

- 处理系统调用的错误
- 获取进程ID
- 创建和终止进程
- 回收子进程
- 进程休眠
- 加载并运行程序

获取进程ID

Obtaining Process IDs

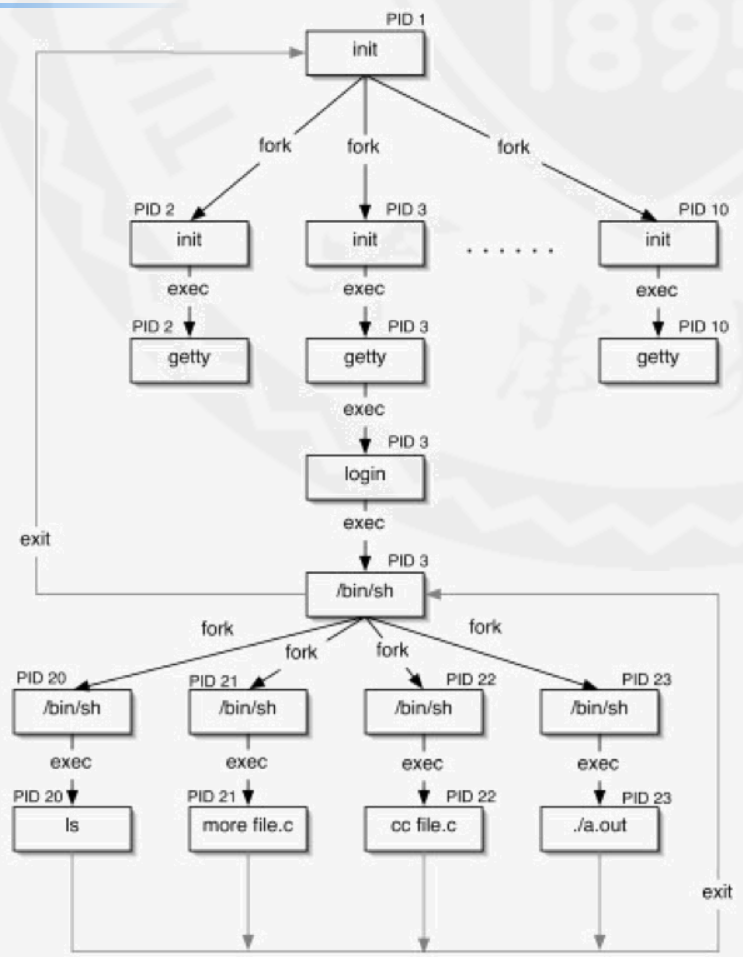
- `pid_t getpid(void)`
  - 获取当前进程的ID
- `pid_t getppid(void)`
  - 获取父进程的ID

获取进程ID

Obtaining Process IDs

进程的层次结构

- 在Linux中，所有进程都是由其他进程启动的。
- 这被称为父/子关系。
- 进程"init"是所有用户进程的祖先进程。它在系统启动时由内核创建。
- 进程形成一个层次结构，称为“进程树”。
- Windows没有进程层次的概念。



本章内容

Topic

- 处理系统调用的错误
- 获取进程ID
- 创建和终止进程
- 回收子进程
- 进程休眠
- 加载并运行程序

创建和终止进程

Creating and Terminating Processes

- 从程序员的角度来看，我们可以将进程看作处于以下三种状态之一：
  - 运行中
    - 进程正在执行，或者等待执行，并最终将由内核调度（即被选择执行）。
  - 停止
    - 进程的执行被暂停，直到进一步通知（信号）才会被调度。
  - 终止
    - 进程永久停止。

终止进程

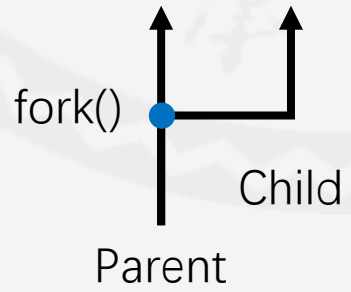
- 进程因以下三个原因之一而终止：
  - 收到默认操作是终止的**信号**
  - 从main函数返回
  - 调用 exit （系统调用）
- **`void exit(int status)`**
  - 终止进程并返回状态`status`
  - 惯例：正常返回状态是 `0`，错误时是非零。
  - 另一种显式设置退出状态的方法是从`main`函数返回一个整数值。
- exit 被一旦被调用，就会结束进程，函数不会返回

创建和终止进程

Creating and Terminating Processes

创建进程

- 父进程通过调用 fork 来创建一个新的运行中的子进程。
- `int fork(void)`
  - 对于子进程，fork 返回 0，对于父进程，返回子进程的 PID。
  - 子进程几乎与父进程相同：
    - 子进程获得父进程虚拟地址空间的相同（但是独立的）副本。
    - 子进程获得父进程打开的文件描述符的相同副本。
    - 子进程有不同于父进程的 PID。
- fork 很有趣（而且经常令人困惑），因为它被调用一次但返回两次。



创建和终止进程

Creating and Terminating Processes

fork 示例

```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);
}
```

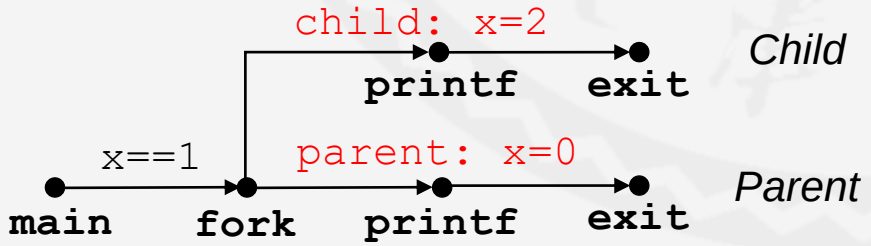
fork.c

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

- 调用一次，返回两次
- 并发执行
  - 无法预测父进程和子进程的执行顺序
- 相同但是独立的地址空间
  - 当 fork 在父进程和子进程中返回时，x 的值为 1
  - 对 x 的后续更改是相互独立的
- 共享打开文件
  - stdout 在父进程和子进程中是相同的。

进程图

- 进程图是描述并发程序中语句执行排序的工具：
  - 每个顶点表示一个语句的执行
  - a -> b 表示 a 发生在 b 之前
  - 边可以用变量的当前值标记
  - printf 顶点可以用输出标记
  - 每个图以一个没有入边的顶点开始

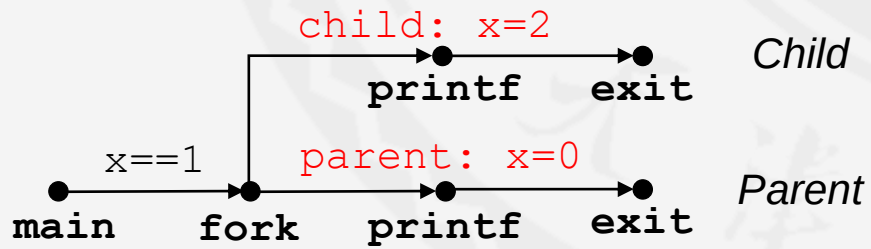


创建和终止进程

Creating and Terminating Processes

进程图示例

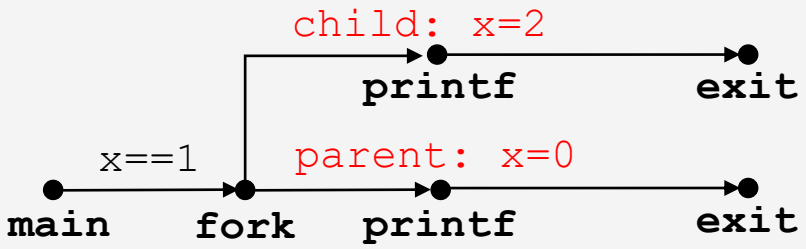
```
int main()
{
    pid_t pid;
    int x = 1;

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

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



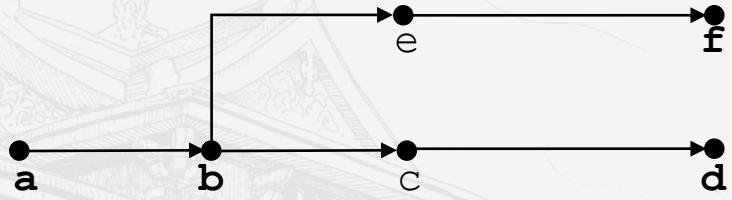
- 对于单处理器计算机，进程图所有顶点的拓扑排序表示程序中语句的一个可行的全序排列
  - 所有边缘都从左到右指向的顶点的总排序

进程图中结点排序

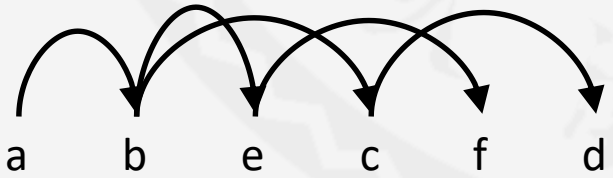
原始进程图



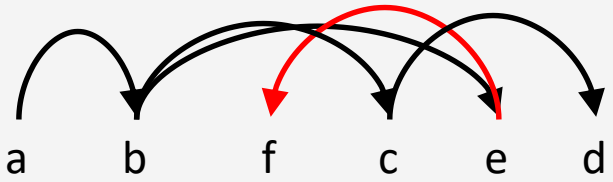
简化标记后



可行的排序



不可行的排序

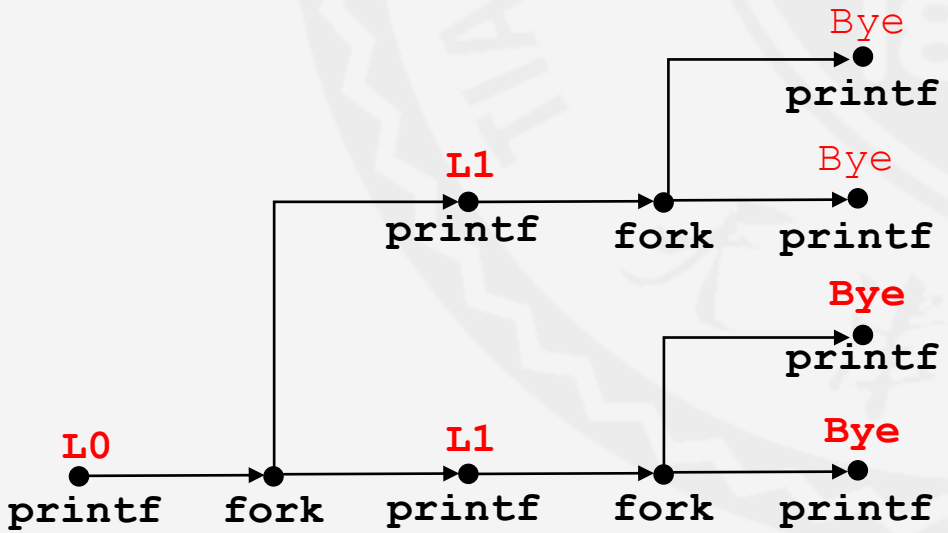


创建和终止进程

Creating and Terminating Processes

fork 示例：两次连续的 fork

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



Feasible output:

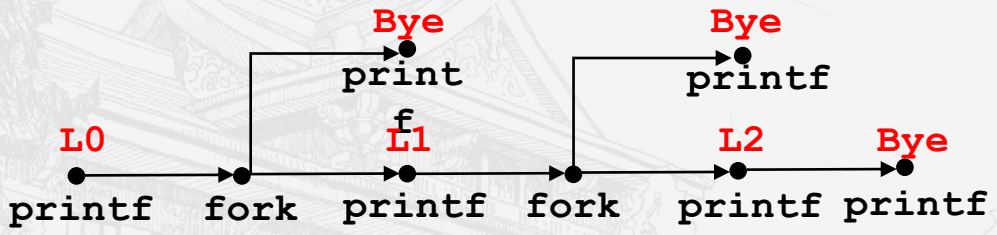
L0
L1
Bye
Bye
L1
Bye
Bye

Infeasible output:

L0
Bye
L1
Bye
L1
Bye
Bye

fork 示例：在父进程中嵌套执行fork

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

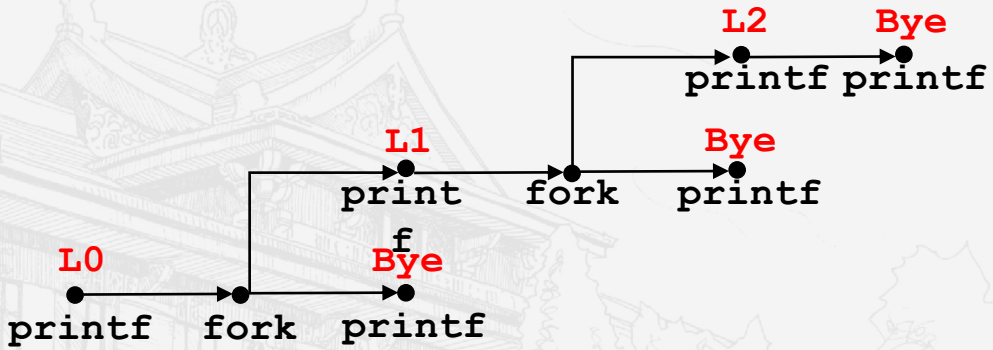


Feasible output:
L0
L1
Bye
Bye
L2
Bye

Infeasible output:
L0
Bye
L1
Bye
Bye
L2

fork 示例：在子进程中嵌套执行fork

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



Feasible output:
L0
Bye
L1
L2
Bye
Bye

Infeasible output:
L0
Bye
L1
Bye
Bye
L2

本章内容

Topic

- 处理系统调用的错误
- 获取进程ID
- 创建和终止进程
- 回收子进程
- 进程休眠
- 加载并运行程序

回收子进程

Reaping Child Processes

- 目标
  - 当进程终止时，它仍然消耗系统资源
    - 例子：退出状态，各种操作系统表
  - 被称为“僵尸”（Zombie）
    - 半活又半死
- 回收
  - 由父进程对终止的子进程执行（使用 wait 或 waitpid）
  - 父进程获得退出状态信息
  - 内核然后删除僵尸子进程
- 如果父进程不进行回收呢?
  - 如果任何父进程终止而不对子进程进行回收，则被遗弃的子进程将由 init 进程（pid == 1）回收
  - 因此，只需要在长时间运行的进程中进行显式的回收
    - 例如，shell 和服务器

回收子进程

Reaping Child Processes

僵尸进程

```
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
```

```c
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

ps 显示子进程状态为 defunct，即僵尸

终止主线程后，子线程将被init线程回收

回收子进程

Reaping Child Processes

未终止的子进程示例

```
linux> ./forks8
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
```

父进程已经终止，子进程仍然处于活动状态

必须显示的结束子进程，否则会一直运行下去

```c
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

回收子进程

Reaping Child Processes

wait：实现子进程的同步（1）

- 父进程可以通过wait系统调用来回收子进程
- `int wait(int *child_status)`
  - 挂起当前进程，直到其某个子进程（之一）终止。
  - 返回值是终止的子进程的 pid。
  - 如果 child_status 不为 NULL，则指针内的整数将被设置为子进程终止原因和退出状态的值：
    - 这些值在 wait.h 中定义
      - WIFEXITED、WEXITSTATUS、WIFSIGNALED、WTERMSIG、WIFSTOPPED、WSTOPSIG、WIFCONTINUED
    - 详细信息请参阅教材

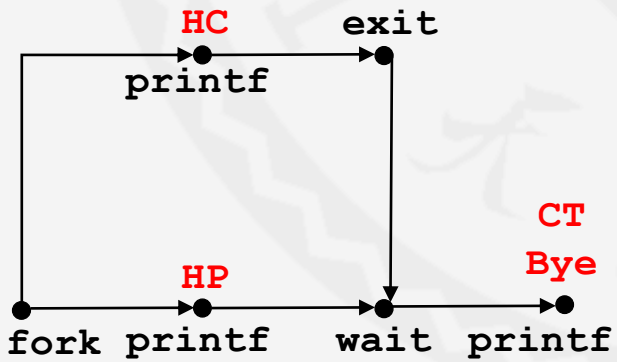
回收子进程

Reaping Child Processes

wait：实现子进程的同步（2）

```
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

回收子进程

Reaping Child Processes

另一个wait的示例

- 如果多个子进程完成，将按任意顺序进行处理。
- 可以使用宏 WIFEXITED 和 WEXITSTATUS 获取有关退出状态的信息。

```c
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

回收子进程

Reaping Child Processes

waitpid: 等待某个特定的进程

- `pid_t waitpid(pid_t pid, int &status, int options)`
  - 挂起当前进程，直到特定的进程终止。
  - 有各种选项（请参阅教材）

```c
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

本章内容

Topic

- 处理系统调用的错误
- 获取进程ID
- 创建和终止进程
- 回收子进程
- 进程休眠
- 加载并运行程序

进程休眠

Putting Processes to Sleep

将进程挂起一段指定时间

- `unsigned int sleep(unsigned int secs);`
  - secs单位：秒
  - 返回值：剩余休眠的秒数
    - 当休眠至指定时间后，函数返回，返回值为0
    - 在休眠时被信号打断，函数返回，返回值为剩余的秒数

进程休眠

Putting Processes to Sleep

将进程挂起直至收到信号

- `int pause(void);`
  - 调用后进程休眠，当进程收到信号后函数返回
  - 返回值：-1，且errno被设置为EINTR

本章内容

Topic

- 处理系统调用的错误
- 获取进程ID
- 创建和终止进程
- 回收子进程
- 进程休眠
- 加载并运行程序

加载并运行程序

Loading and Running Programs

Execve系统调用

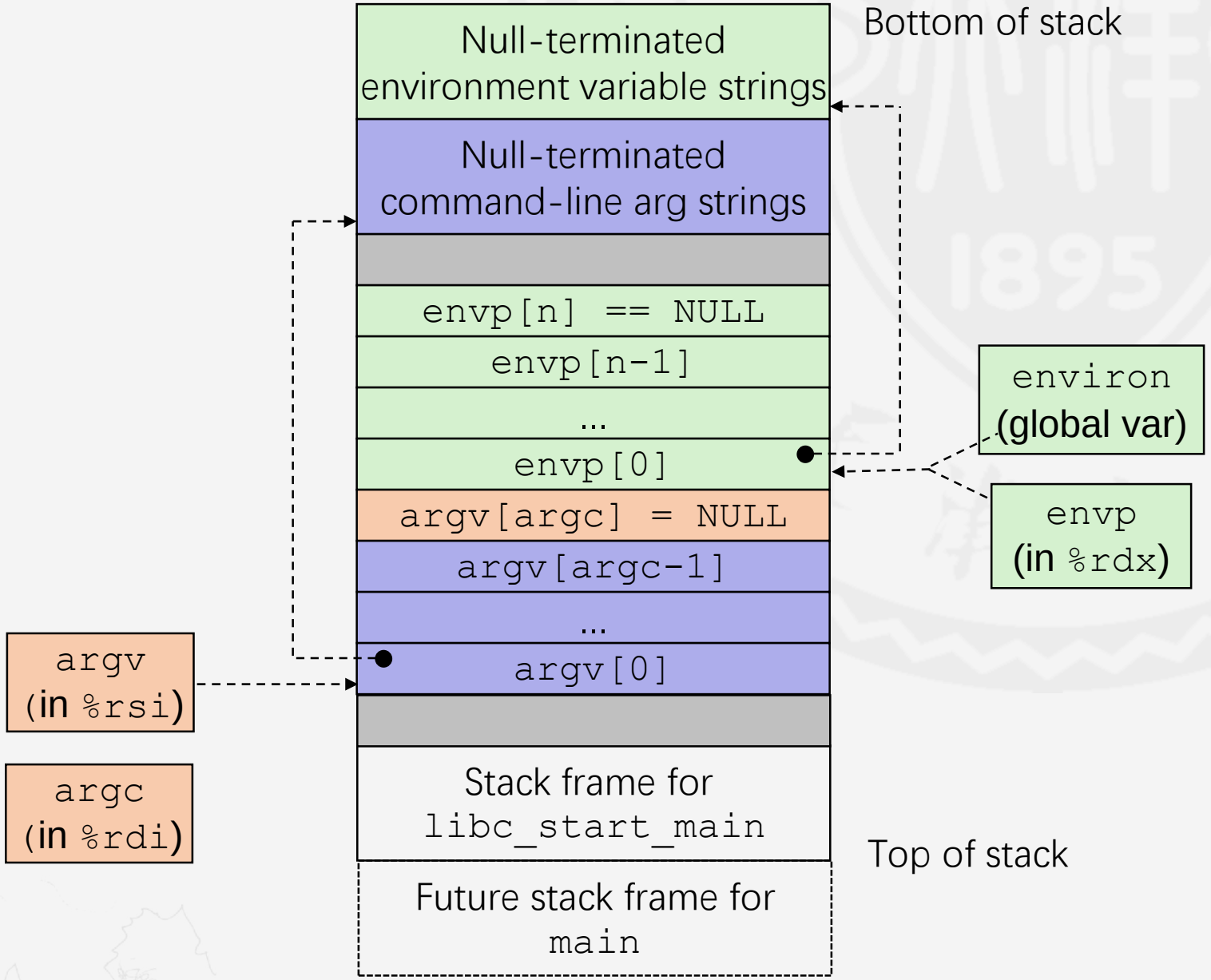
- `int execve(char *filename, char *argv[], char *envp[])`
- 加载并在当前进程中运行
  - filename: 可执行文件路径
    - 可以是可执行目标文件，也可以是以#!interpreter开头的脚本文件（例如：#!/bin/bash）
  - argv:参数列表
    - 按照惯例 argv[0]==filename
  - envp: 环境变量列表
    - “name=value” 字符串（例如，USER=droh）
    - 访问环境变量的函数：getenv、putenv、printenv

- 借鸡生蛋
  - 覆盖当前进程代码、数据和堆栈
    - 保留 PID、打开文件和信号上下文
  - 调用后不会返回
    - …除非发生错误

加载并运行程序

Loading and Running Programs

新程序启动时
栈的结构

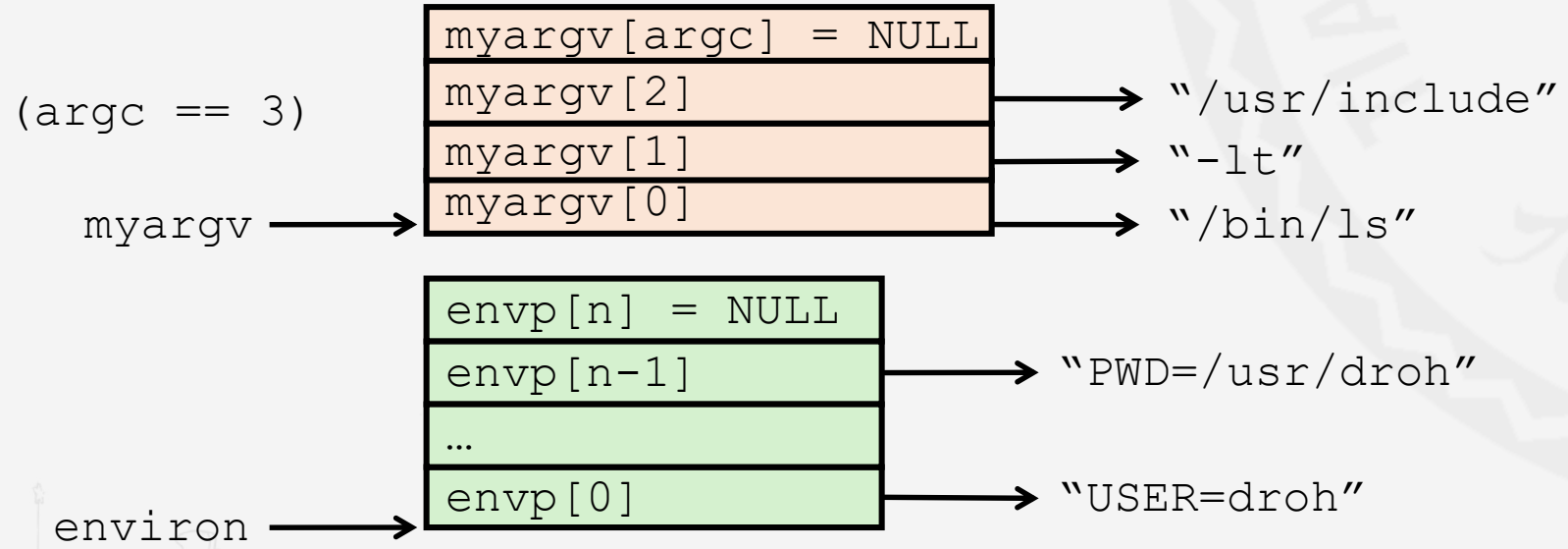


加载并运行程序

Loading and Running Programs

Execve示例

在子进程中使用当前环境变量执行 “/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);
    }
}
```

小结

- 生成进程
  - 调用 `fork`
  - 一次调用，两次返回
- 进程完成
  - 调用 `exit`
  - 调用后不会返回
- 回收和等待进程
  - 调用 `wait` 或 `waitpid`
- 进程休眠
  - `sleep` 休眠指定时间，或等到信号
  - `pause` 休眠至等到信号
- 加载和运行程序
  - 调用 `execve`（或变体）
  - 调用后（通常）没有返回