Linking II & ECF: Exceptions & Processes

刘昕垚 杨斯淇 许珈铭

2023.11.20

LinkingII (CS:APP Ch. 7.9-7.14)

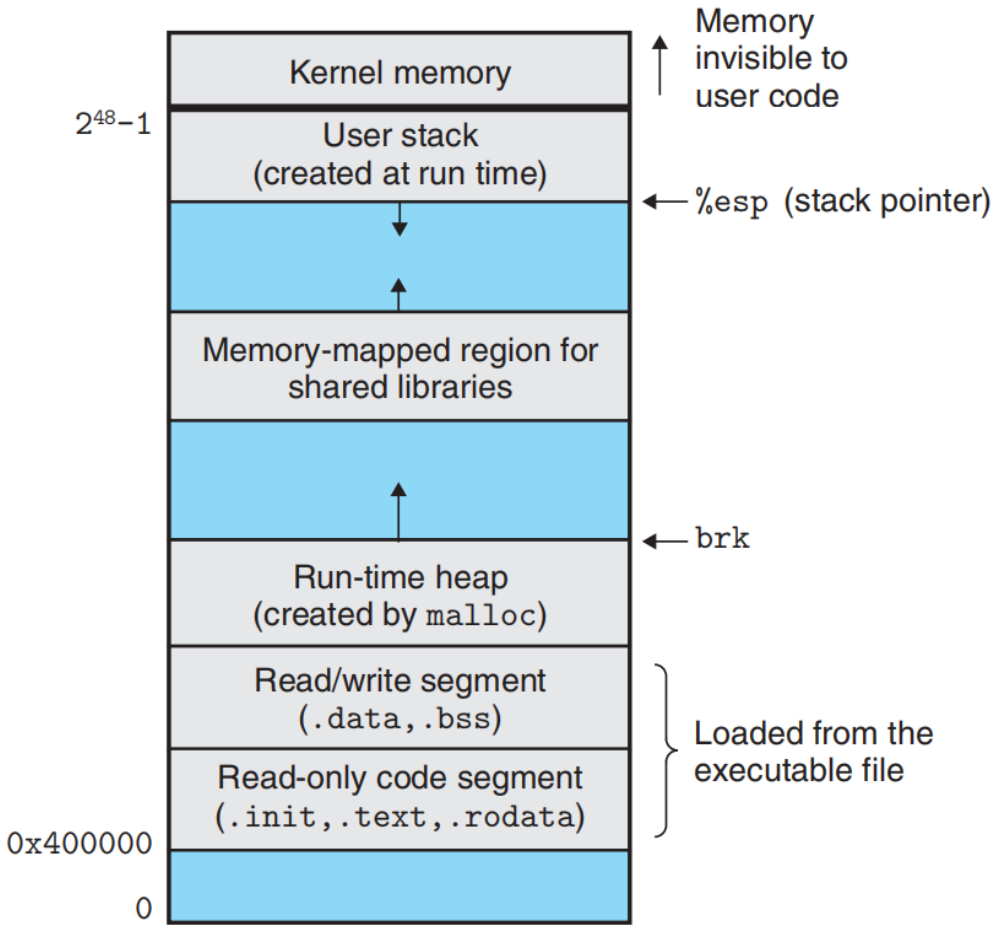
刘昕垚

Content

- 动态链接共享库（DLL）
- 位置无关代码（PIC）
  - PIC数据引用
  - PIC函数调用
- 库打桩

Linux x86-64 运行时内存映像

Figure 7.15
Linux x86-64 run-time memory image. Gaps due to segment alignment requirements and address-space layout randomization (ASLR) are not shown. Not to scale.



静态库

- .a archive文件

- 将相关的可重定位的对象文件连接到具有索引的单个文件中
- 当链接器构造可执行文件时，只复制静态库里被引用的目标模块

静态库

- 缺点：
  - 静态库(.a)需要定期维护与更新
    - 系统库的小错误修复需要每个应用程序显式地重新链接
  - 存储&正在运行的可执行文件中存在重复复制
    - 几乎每一个C程序都使用标准I/O函数，造成内存资源浪费

动态链接共享库

- DLL：Dynamic Linking Library

- **共享库（共享目标）**：一个目标模块

  Linux中：.so

  包含在加载时或运行时动态加载和链接到应用程序中的代码和数据的对象文件

- **动态链接**：

  通过**共享库**加载到任意内存地址，并且和内存中程序链接的过程

- **动态链接器**：

  执行动态链接的程序

动态链接共享库

- 共享方式:
  - 文件系统:

    所有可执行目标文件共享唯一的.so文件里的代码和数据
  - 内存里:

    共享库的.text节的一个副本可以被不同运行的进程共享

    （虚存里细讲）

动态链接共享库

- 第一次加载可执行文件时链接：
  - Linux 的常见情况，由动态链接器自动处理 (ld-linux.so)
  - C标准库一般动态链接 (libc.so)
- 在程序开始后链接
  - 在linux系统中，通过调用 dlopen() 来完成
    - 分发软件
    - 构建高性能Web服务器
- 共享库例程可以由多个进程进行共享

在加载时动态链接共享库

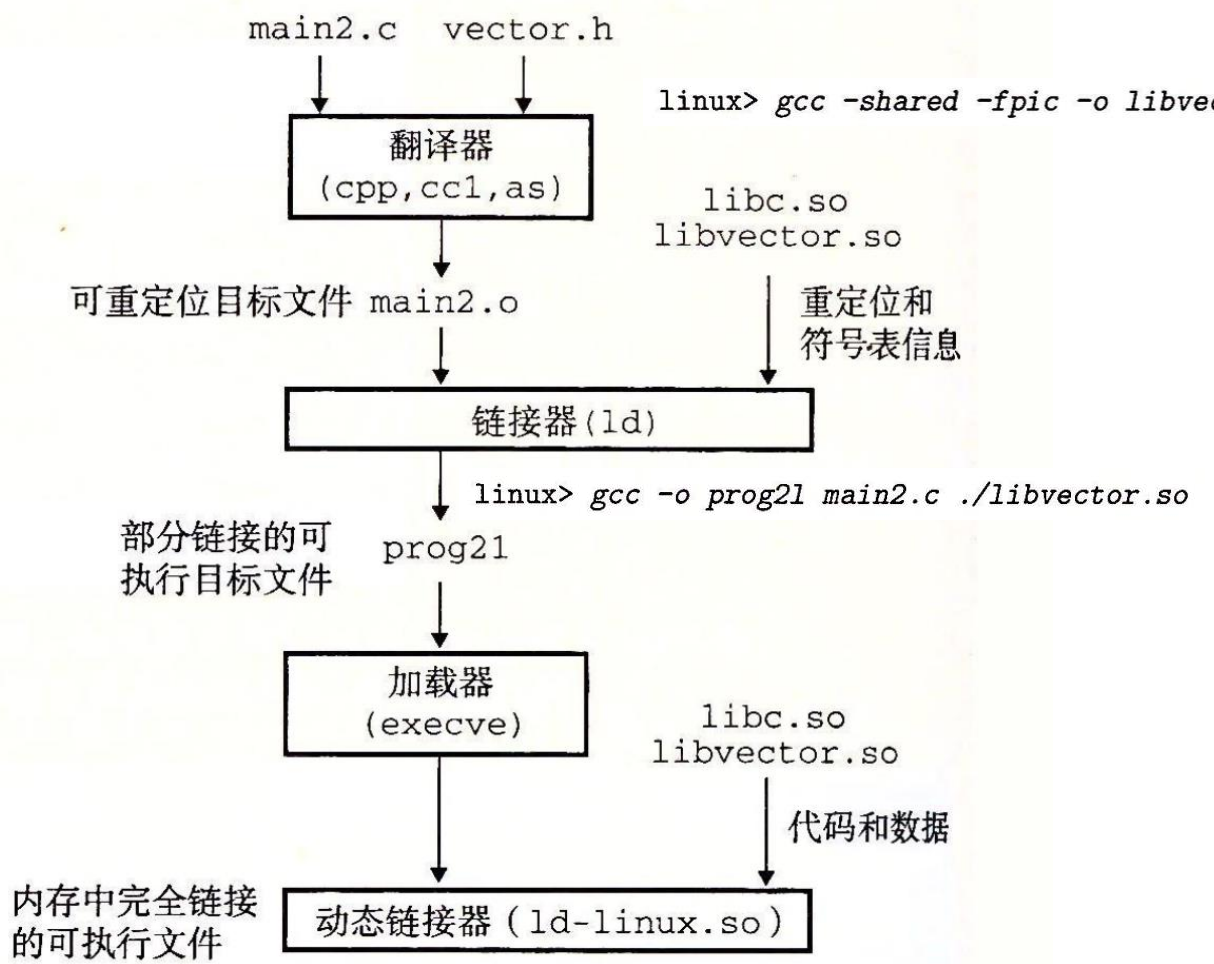


图 7-16　动态链接共享库

注：

- lbvector.so的代码和数据节不复制到可执行文件 中。
- 链接器复制了一些重定位和符号表信息，使得运行时可以解析对 libvector.so 中代码和数据的引用。
- 可执行文件包含一个.interp 节，包含动态链接器的路径名（比如：ld-linux.so）

PIC

- 位置无关代码（Position-Independent Code）
  - 可以加载而无需重新定位的代码

- 共享库的编译必须：

  对GCC使用**-fpic**指使GNU编译系统生成PIC代码

- 对共享库的外部过程&全局变量引用：
  - PIC数据引用
  - PIC函数调用

PIC

- 为什么需要位置无关代码?
  - 链接器一般将程序的加载地址绑定到固定位置
  - 如果一个程序的加载地址是不固定的（例如，共享库）：

    我们使用位置无关代码，生成不取决于加载位置的代码，以动态链接正确执行代码

- 效率略低于绝对寻址
- 如今常用

PIC数据引用

- ELF可执行文件特征：
  - 代码段中任何指令和数据段中任何变量之间的距离都是常量
  - 与代码段和数据段的绝对内存位置无关

- **GOT** 全局偏移量表（Global Offset Table）：
  - 全局数据目标（过程or变量）：8字节条目
  - 每个条目：重定位记录
    - 程序可以使用固定的偏移来访问GOT，并从那里访问数据

PIC数据引用

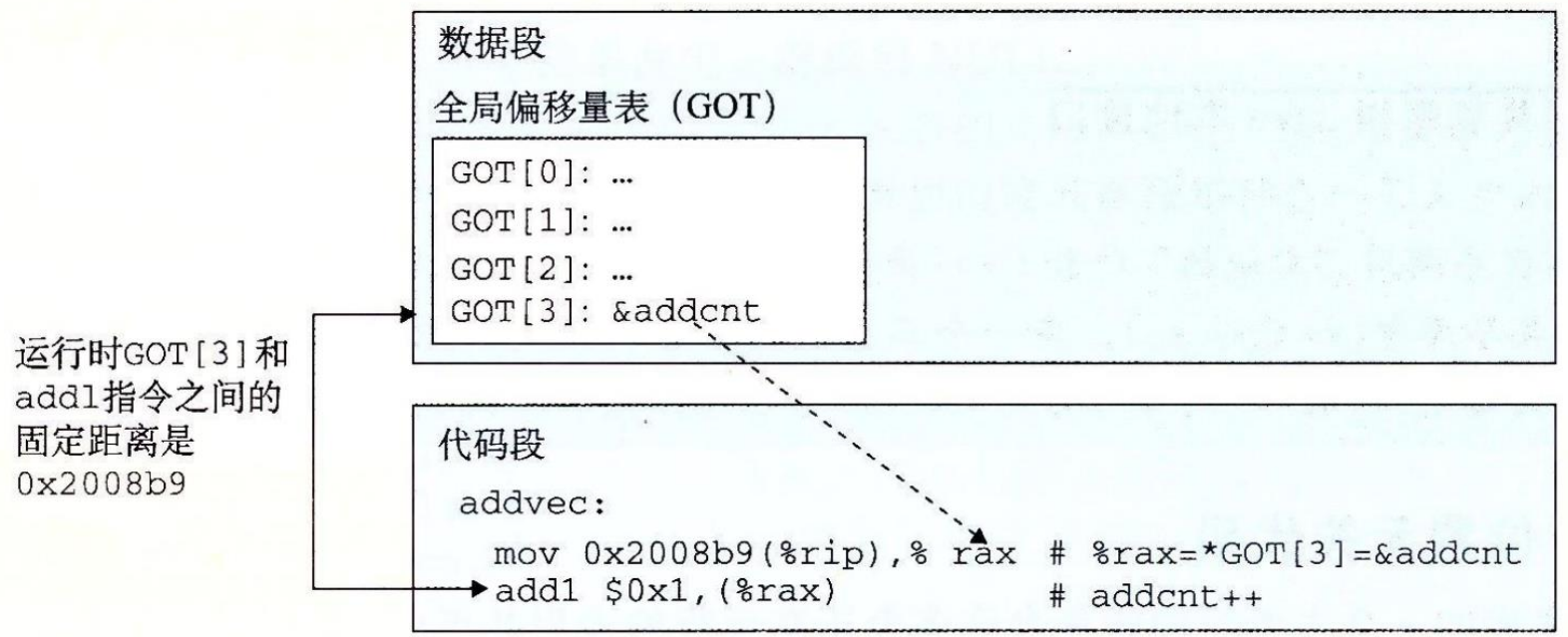


图 7-18 用 GOT 引用全局变量。libvector.so 中的 addvec 例程通过 libvector.so 的 GOT 间接引用了 addcnt

- p.s. 若addcnt由另一共享模块定义，要间接访问

PIC数据引用

- 优势：
  - 减少重定位
  - 不同进程共享内存

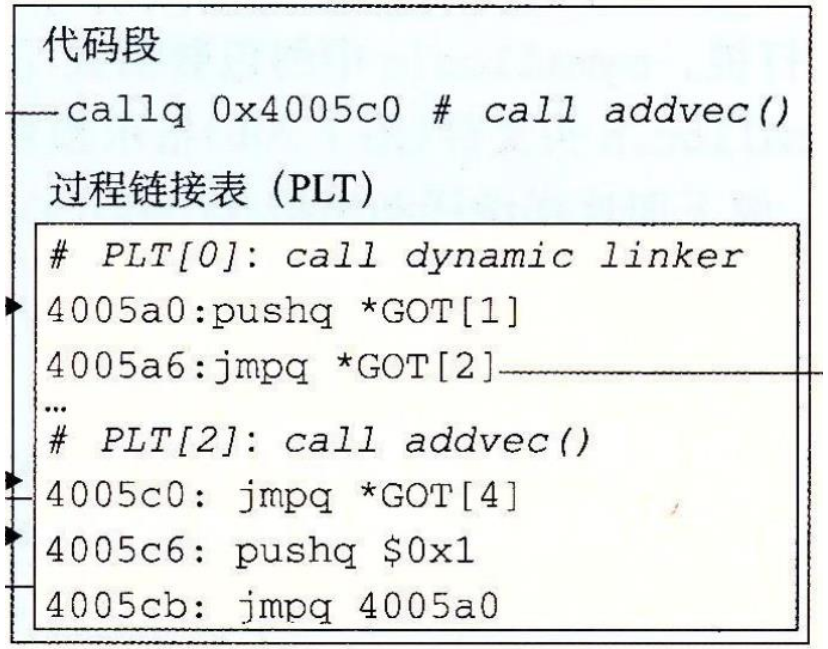
- 劣势：
  - GOT在每次加载时需要重分配
  - GOT可能会很大，会使有PIC的程序更慢

PIC函数调用

- 不能修改调用模块的代码
- 延迟绑定（lazy binding):
  - 将过程地址的绑定推迟到第一次调用该过程时
  - 实现途径:
    - GOT
    - PLT（过程链接表，Procedure Linkage Table）

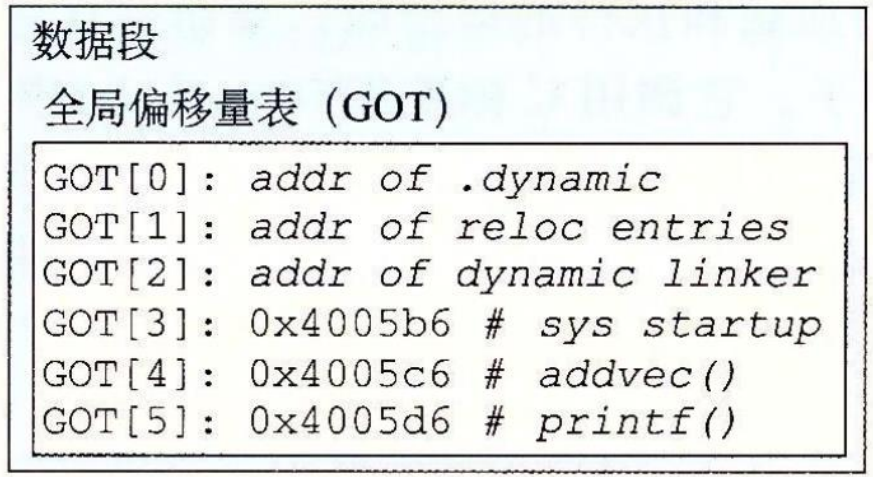
PIC函数调用

- **PLT**：
  - 一个数组，每个条目是**16**字节代码
  - 每个被可执行程序调用的库函数都有自己的PLT条目
  - 每个条目都负责调用一个具体的函数
  - **PLT[0]**：特殊，跳转到动态链接器
  - **PLT[1]**：调用系统启动函数（__libc_start_main），初始化执行环境，调用main函数并处理其返回值
  - **PLT[2]开始的条目**：调用用户代码调用的函数

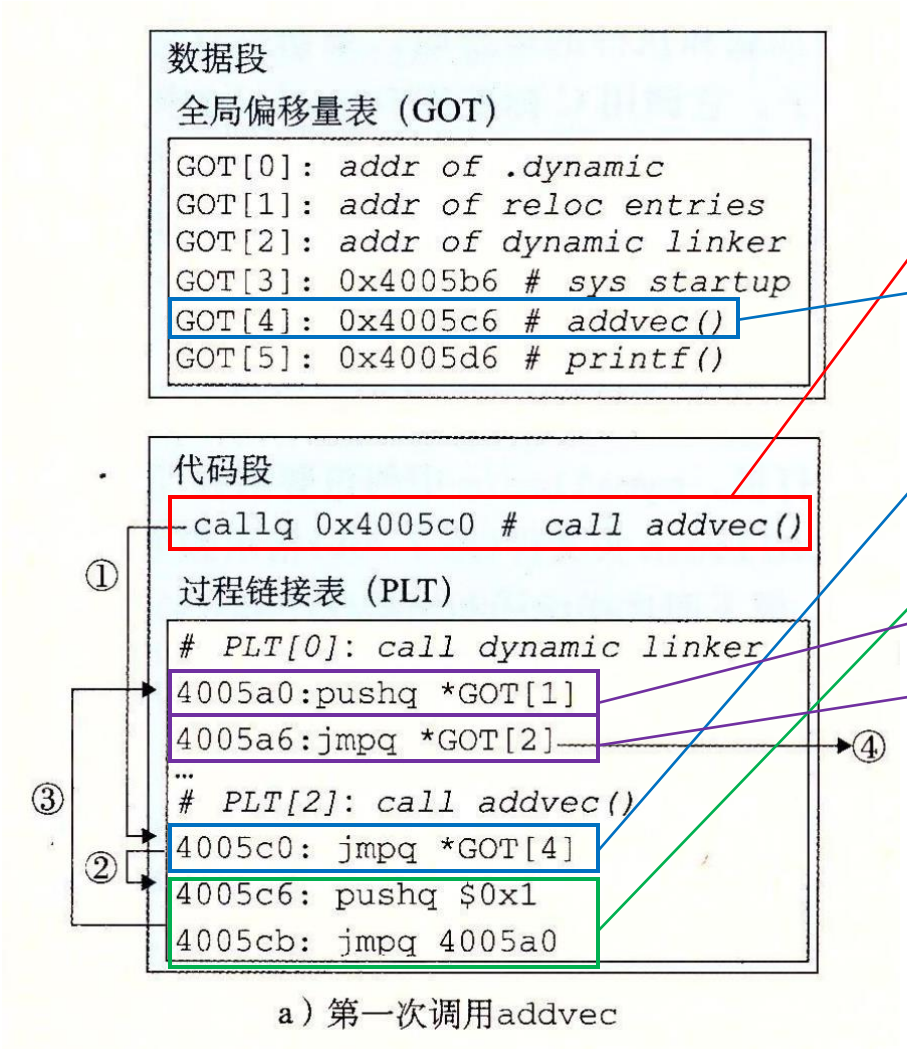


PIC函数调用

- **GOT**：
  - 一个数组，每个条目是**8**字节地址
  - 和PLT联合使用：
    - **GOT[0]、GOT[1]**：动态链接器在解析函数地址时使用的信息
    - **GOT[2]**：动态链接器在ld-linux.so模块中的入口点
    - **其余每个条目**：一个被调用的函数，地址在运行时被解析
    - 每个条目都有一个**相匹配的PLT条目**



PIC函数调用过程-第一次



a）第一次调用addvec

① 调用进入PLT[2]（addvec的PLT条目）

② 通过GOT[4]进行间接跳转

- 每个GOT条目初始时都指向它对应的PLT条目的第二条指令

③ 把addvec的ID(0x1)压入栈中，PLT[2]跳转到PLT[0]

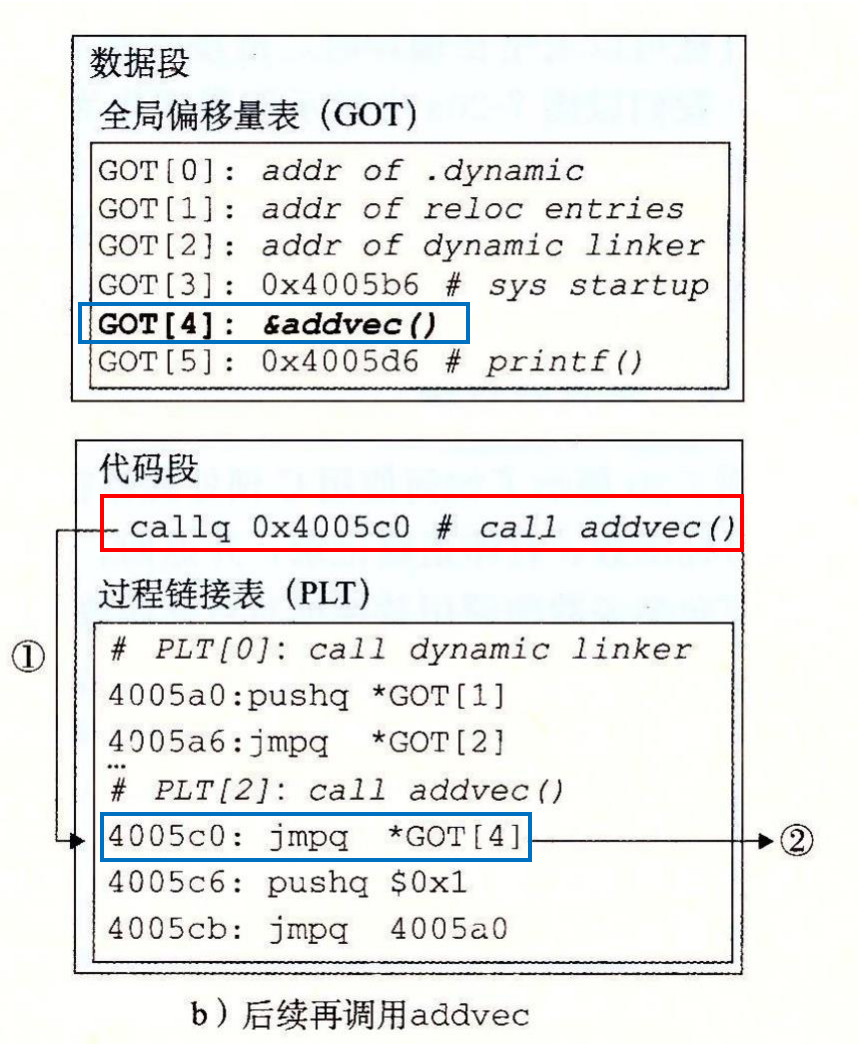
④ 通过GOT[1]把动态链接器的一个参数压栈

⑤ 通过GOT[2]间接跳转进动态链接器

⑥ 链接器使用两个栈条目来确定addvec的运行时位置，重写GOT[4]

⑦ 把控制传递给addvec。

PIC函数调用过程-后续



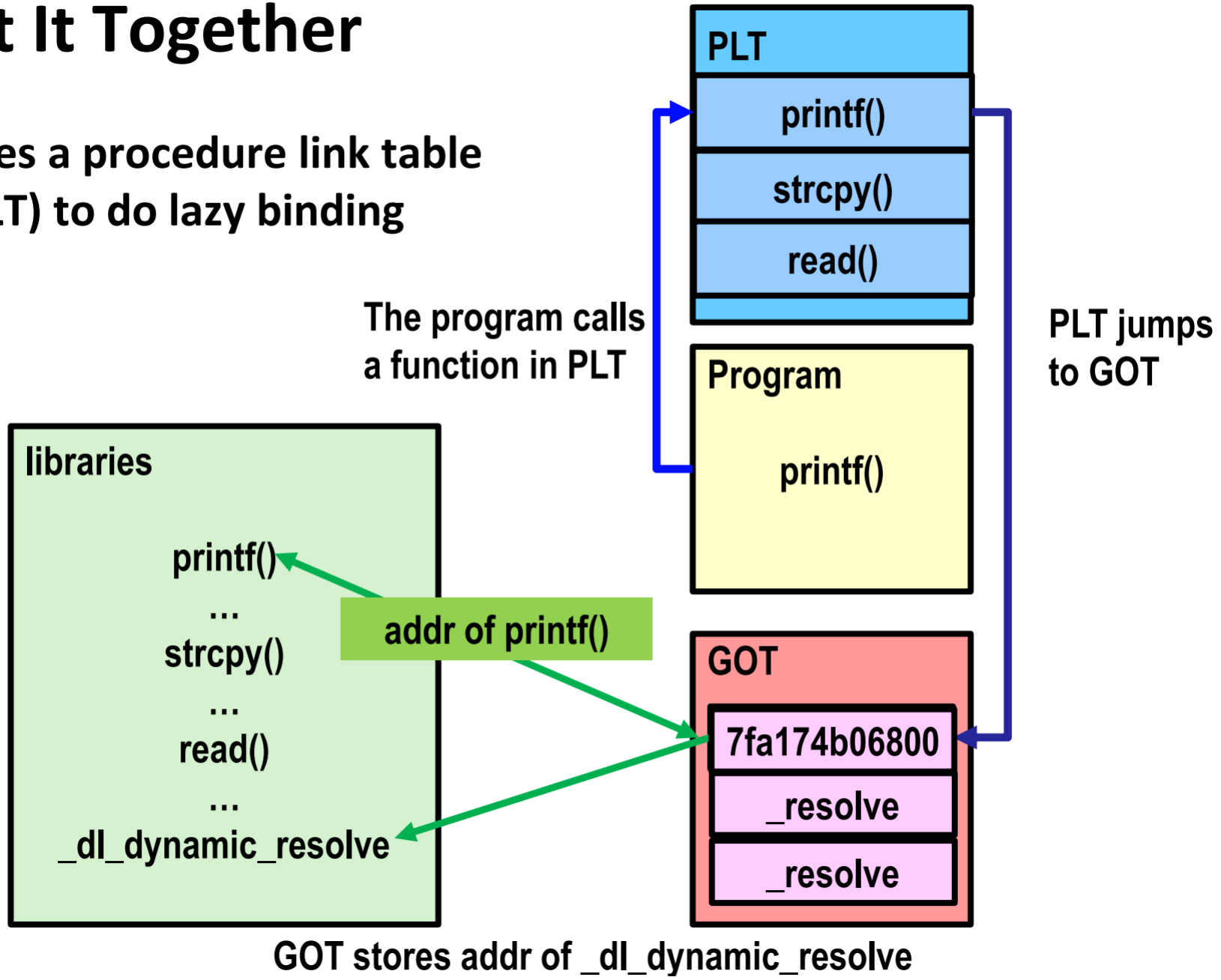
b）后续再调用addvec

① 调用进入PLT[2]（addvec的PLT条目）

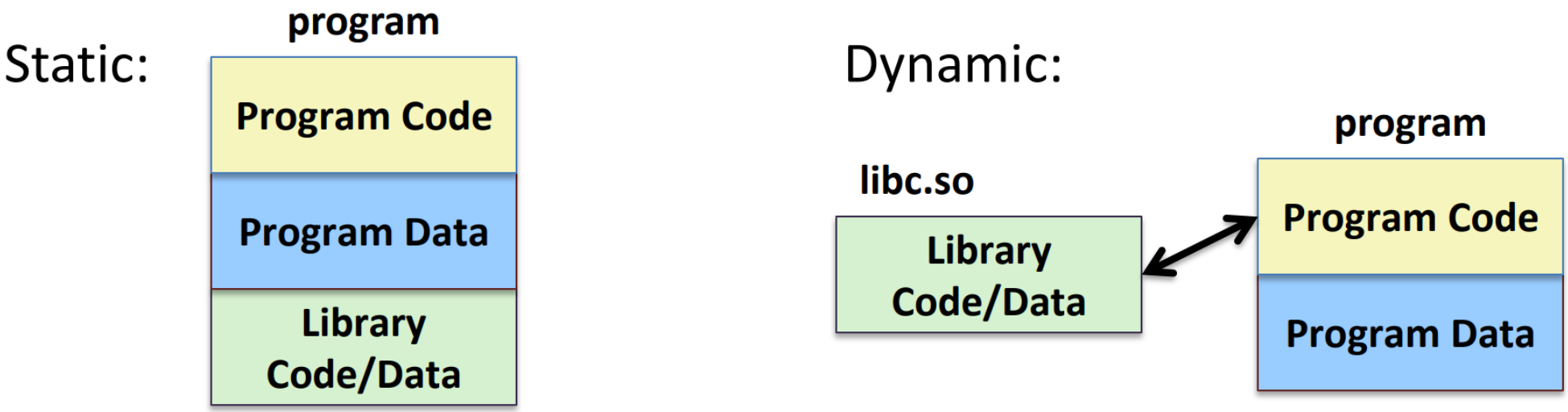
② 通过GOT[4]间接跳转将控制直接转移到addvec

Put It Together

- Uses a procedure link table (PLT) to do lazy binding

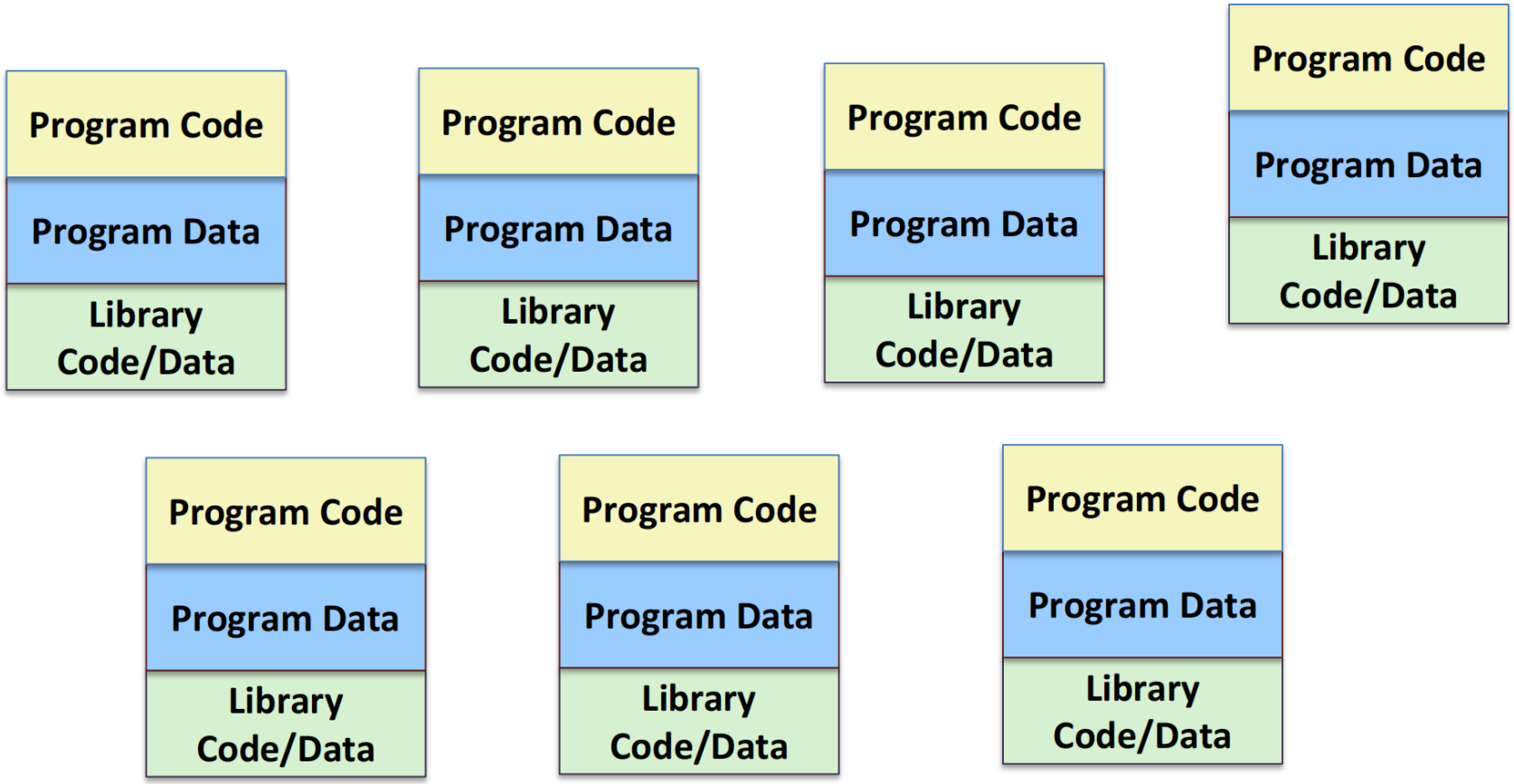


静态库vs动态链接共享库



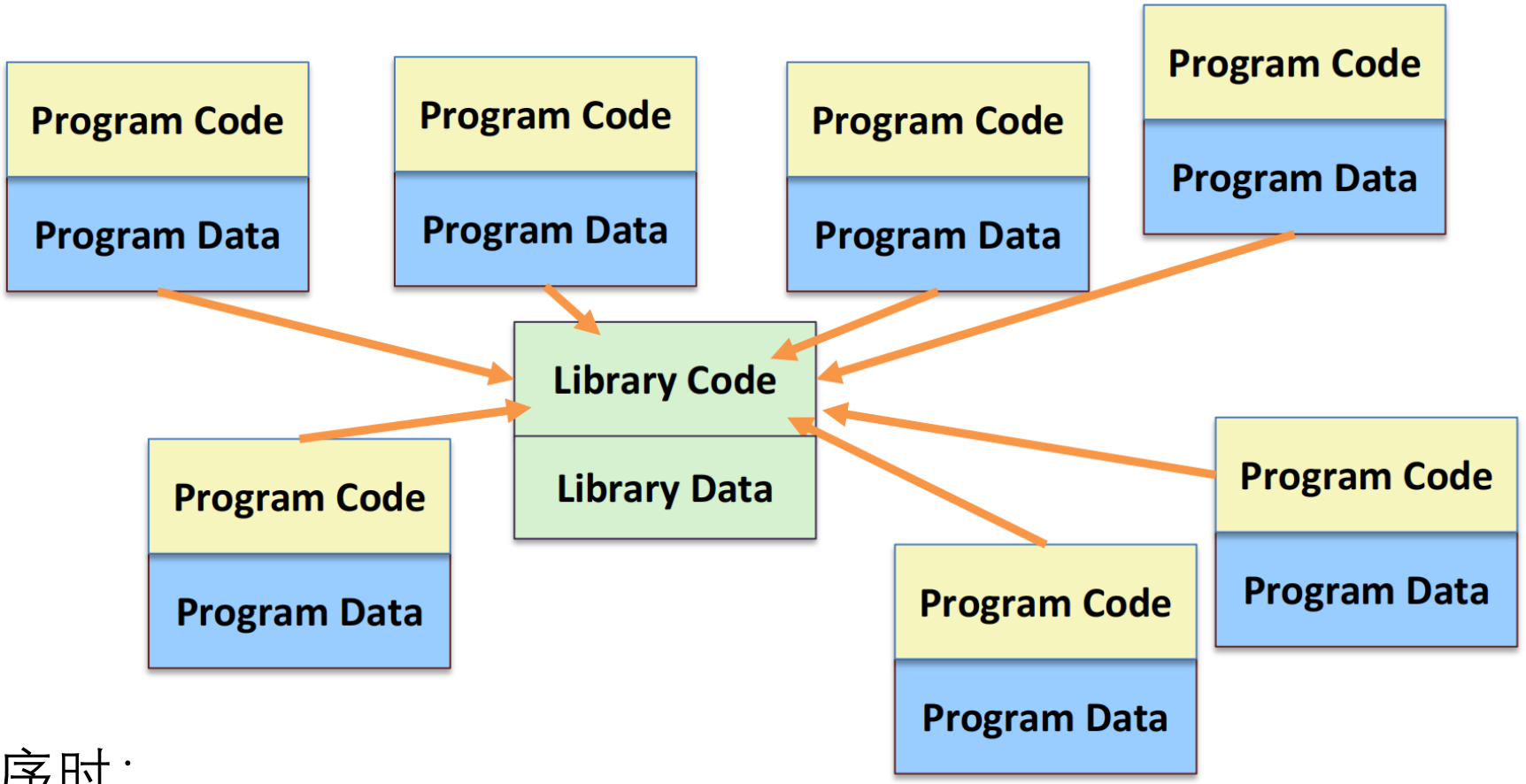
静态库vs动态链接共享库

- 对于多个程序:
- Static:



静态库vs动态链接共享库

- Dynamic:



- 在有多个程序时:

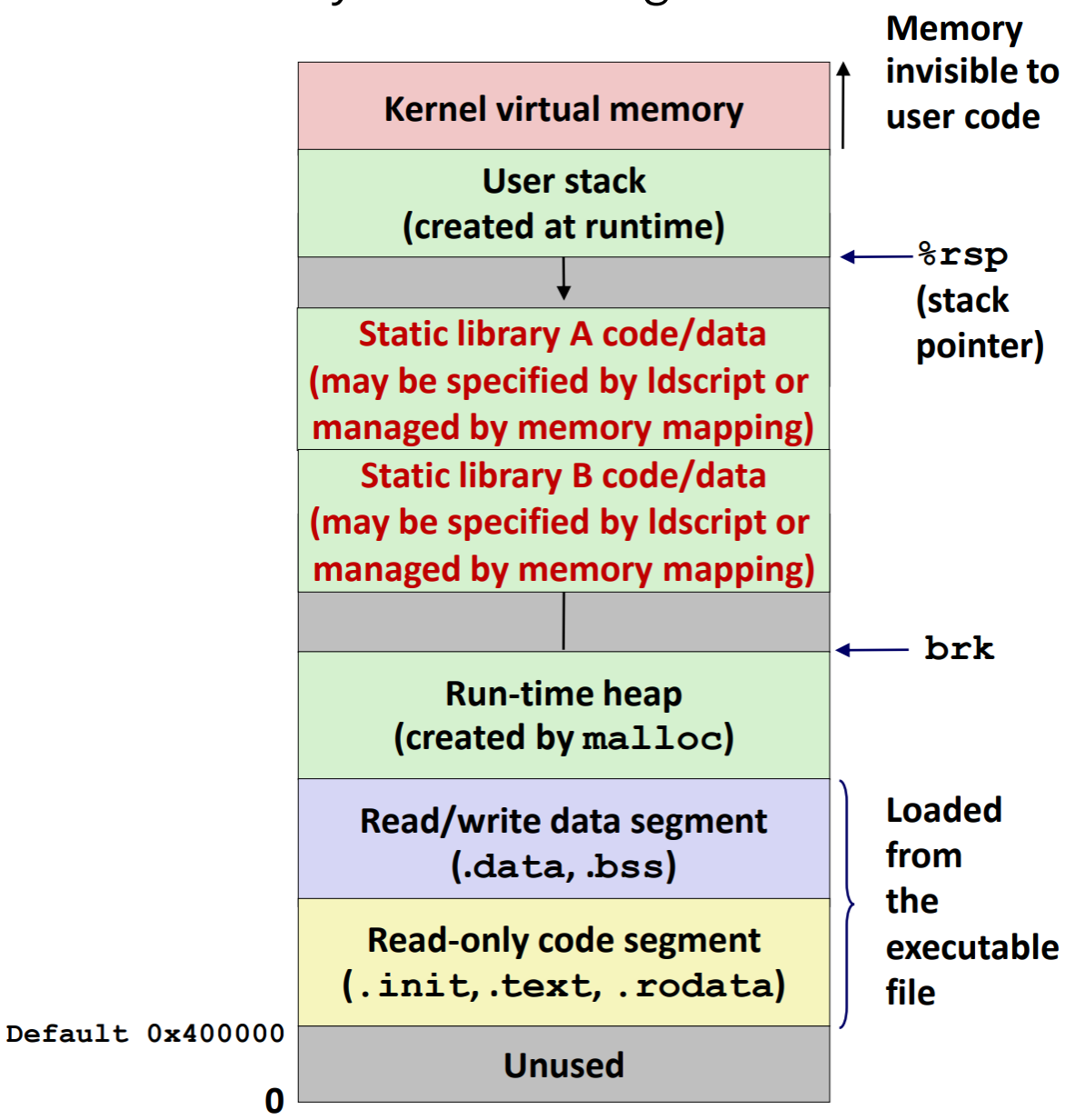
  动态库比静态库更节省空间

静态库vs动态链接共享库

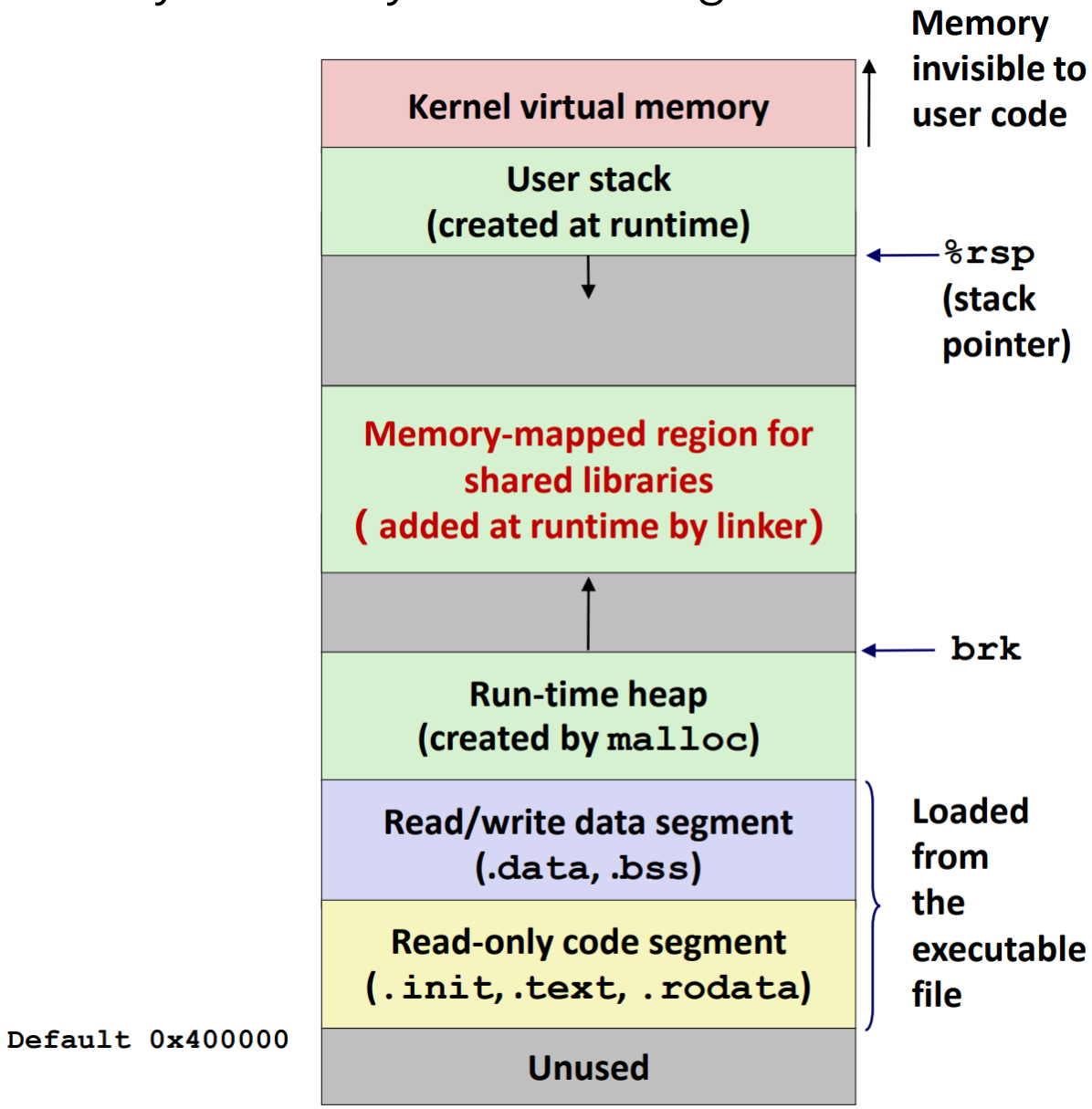
- Static:
  - 不需要在运行时查找库
  - 不需要额外的PLT连接
  - 使用每个程序中的每个库的副本消耗更多的内存

- Dynamic:
  - 更少的磁盘空间/内存
  - 共享库已经在内存和热缓存
  - 发生查找和连接开销

Statically Linked Program



Dynamically Linked Program



静态库vs动态链接共享库

	Compile-time	Link-time	Load-time	Run-time
Static linking	Textual source inclusion	OBJ/LIB inclusion	-	-
Dynamic linking	-	-	Reference to an external DLL	Loading an external DLL

库打桩

- 截获对共享库函数的调用，取而代之执行自己的代码

- 用处：
  - 追踪对某个特殊库函数的调用次数
  - 验证和追踪它的输入和输出值
  - 把它替换成一个完全不同的实现

- 实现：使用包装函数

库打桩

- 打桩发生时间:
  - **编译时**
    - 需要能够访问程序的**源代码**
  - **链接时**
    - 需要能够访问程序的**可重定位对象文件**
  - **程序被加载和执行的运行时**
    - 需要能够访问**可执行目标文件**

处理目标文件的工具

- AR：创建静态库，插入、删除、列出和提取成员。
- STRINGS：列出一个目标文件中所有可打印的字符串。
- STRIP：从目标文件中删除符号表信息。
- NM：列出一个目标文件的符号表中定义的符号。
- ·SIZE：列出目标文件中节的名字和大小。
- READELF：显示一个目标文件的完整结构，包括ELF头中编码的所有信息。包含SIZE和NM的功能。
- OBJDUMP：所有二进制工具之母。能够显示一个目标文件中所有的信息。它最大的作用是反汇编.text节中的二进制指令。

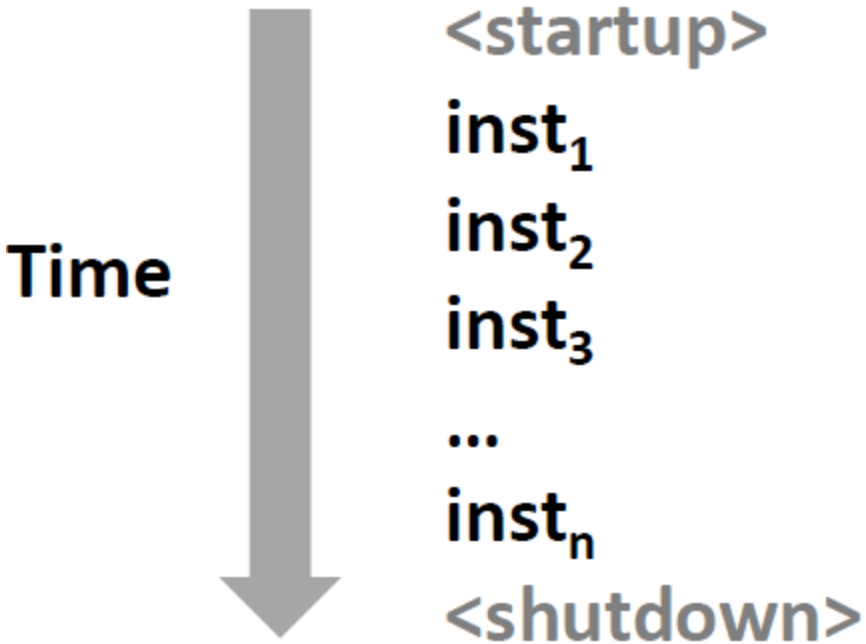
- LDD：列出一个可执行文件在运行时所需要的共享库（Linux系统为操作共享库）

ECF: Exceptions & Processes (CS:APP Ch. 8.1-8.4)

杨斯淇

概念引入

控制流：从加电到断电，一串**指令地址**



“平滑”：**跳转、调用、返回**，由程序内部引起

响应另外的系统状态变化，如硬件的信号、软硬件的交互等。

控制流加入**“突变”**也就是异常，这样的控制流称为异常控制流。

此处的异常和我们一般说的略有不同，诸如上下文切换、信号等是操作系统的正常机制，但它们**影响了控制流平滑运行**，因而被称为异常。

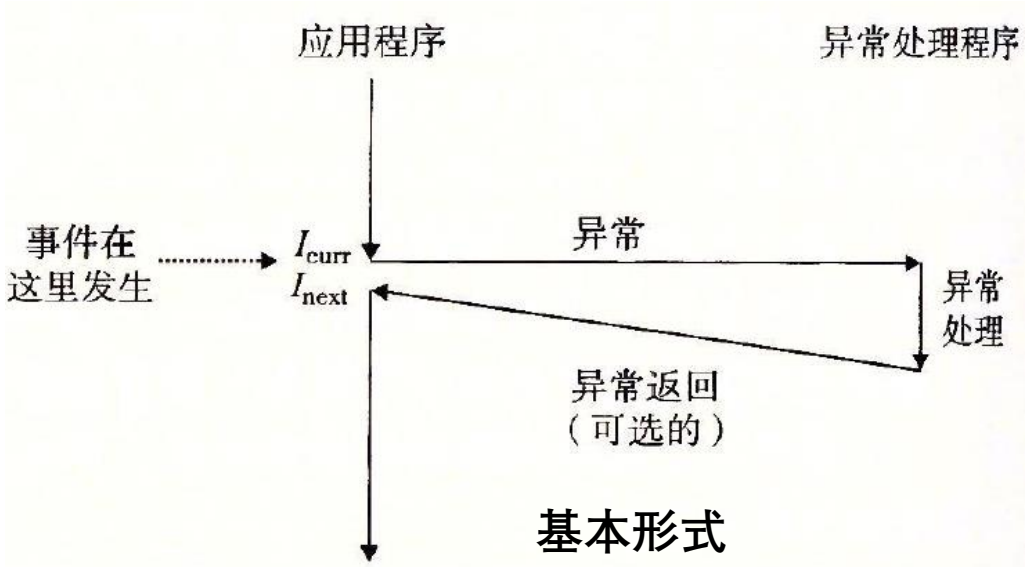
低层次机制：异常

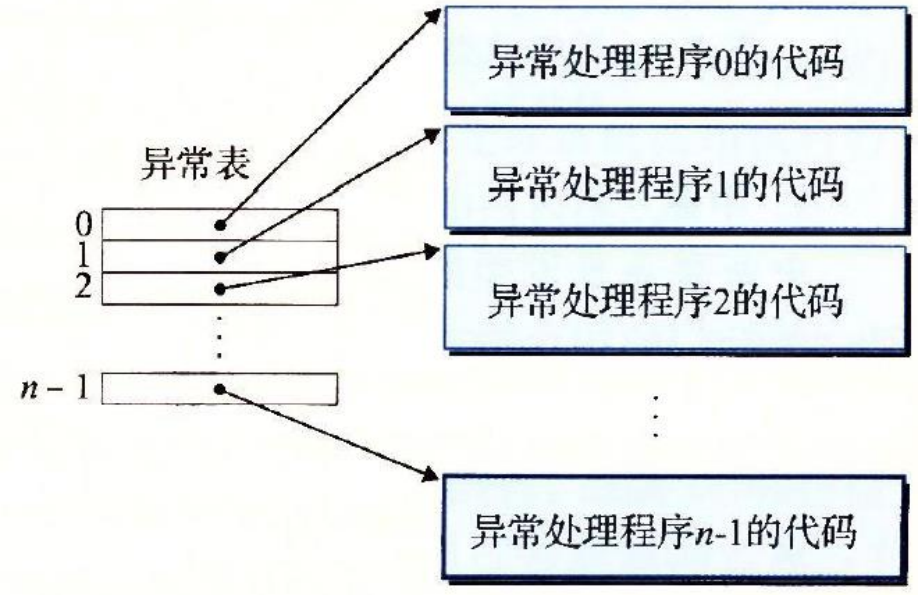
硬件与软件结合

- 异常----异常号----异常处理程序

- 异常表、跳转到异常表由**硬件**实现
- 异常处理程序由**软件（操作系统）**实现
- 启动时，异常表由**操作系统**填写

调用	异常
返回地址一定是下一条指令	返回地址可能是当前指令，可能是下一条，也可能不返回
只把参数和返回地址压入用户栈中	会把恢复中断状态所需的额外的处理器状态（如EFLAGS）压入内核栈中
运行在用户模式	运行在内核模式（超级用户模式）





低层次机制：异常

****每种类型的样例可以记忆一些，遇到了也可以简单分析一下

异常的类别：中断，陷阱，故障，终止。
异步异常是**由处理器外部事件引起的**，同步异常是**执行指令的结果**。

- 中断：由I/O设备信号引起，**异步**，总是返回到下一条指令。
- 陷阱：“有意的”通过执行syscall实现，用于实现系统调用，同步，总是返回到下一条指令。**使得用户程序可以受控的使用内核服务**。
- 故障：可能被修正，同步，若成功修正则返回到当前指令，否则不返回并终止。
  eg.**缺页异常**
- 终止：不可恢复，同步，不返回。

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

- 下列行为分别触发什么类型的异常？

1. 执行指令mov $57, %eax; syscall
2. 程序执行过程中，发现它所使用的物理内存损坏了
3. 程序执行过程中，试图往main函数的内存中写入数据
4. 按下键盘
5. 磁盘读出了一块数据
6. 用read函数发起磁盘读
7. 用户程序执行了指令lgdt，但是这个指令只能在内核模式下执行

1. （陷阱）
2. （中止）
3. （故障）
4. （中断）
5. （中断）
6. （陷阱）
7. （故障）

触发异常的流程：

- 检测事件，确定异常号，切换内核模式，保存上下文
- 用异常表基址寄存器和异常号得到处理例程的入口地址
- 触发异常，间接跳转
- 进行异常处理
- （如果需要返回）恢复上下文，回到用户模式

低层次机制：异常

异常示例：0-31号对应Intel架构师定义的异常，32-255号对应操作系统定义的**中断或陷阱**

Linux中的系统调用：C程序使用syscall可以调用任何系统调用，但是实际上**一般使用标准C库提供的包装函数**，它们打包好参数并进行系统调用，传递返回值。

系统调用的汇编级实现：系统调用通过汇编指令syscall实现，参数通过寄存器传递，**%rax包含系统调用号**，至多6个参数通过%rdi、%rsi、%rdx、%r10、%r8和%r9传递，返回值存储在%rax中，**负的返回值表明发生了错误**

异常号	描述	异常类别
0	除法错误	故障
13	一般保护故障	故障
14	缺页	故障
18	机器检查	终止
32~255	操作系统定义的异常	中断或陷阱

进程：逻辑流与私有地址空间

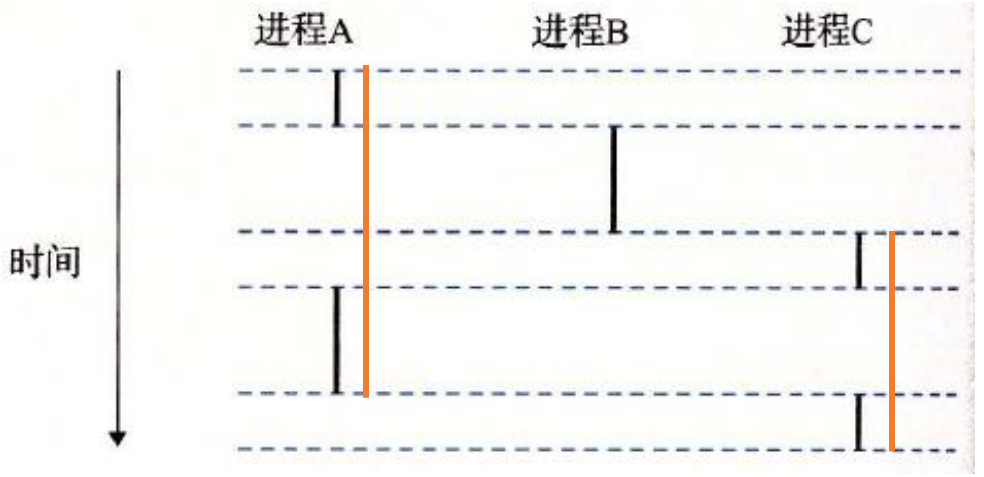
进程：一个**执行中**程序的**实例**
context：存放程序运行所需要的**状态**

*区分程序与进程

程序是一串代码和数据，可以存在磁盘上或在地址空间中
进程是程序的具体实例，**程序运行在进程的上下文中**

两个关键抽象：
- **独立的逻辑控制流**
- **私有的地址空间**

逻辑控制流：**多个程序的PC组成的序列**



- 竖直的条：一个进程逻辑控制流的一部分
- 进程**轮流使用**处理器，使得系统可以同时存在多个进程，但某时刻只执行一个，其他**挂起**

并发流	**开始到结束的时间段有重叠**
并行流	并发并且在**运行在不同处理器/计算机**

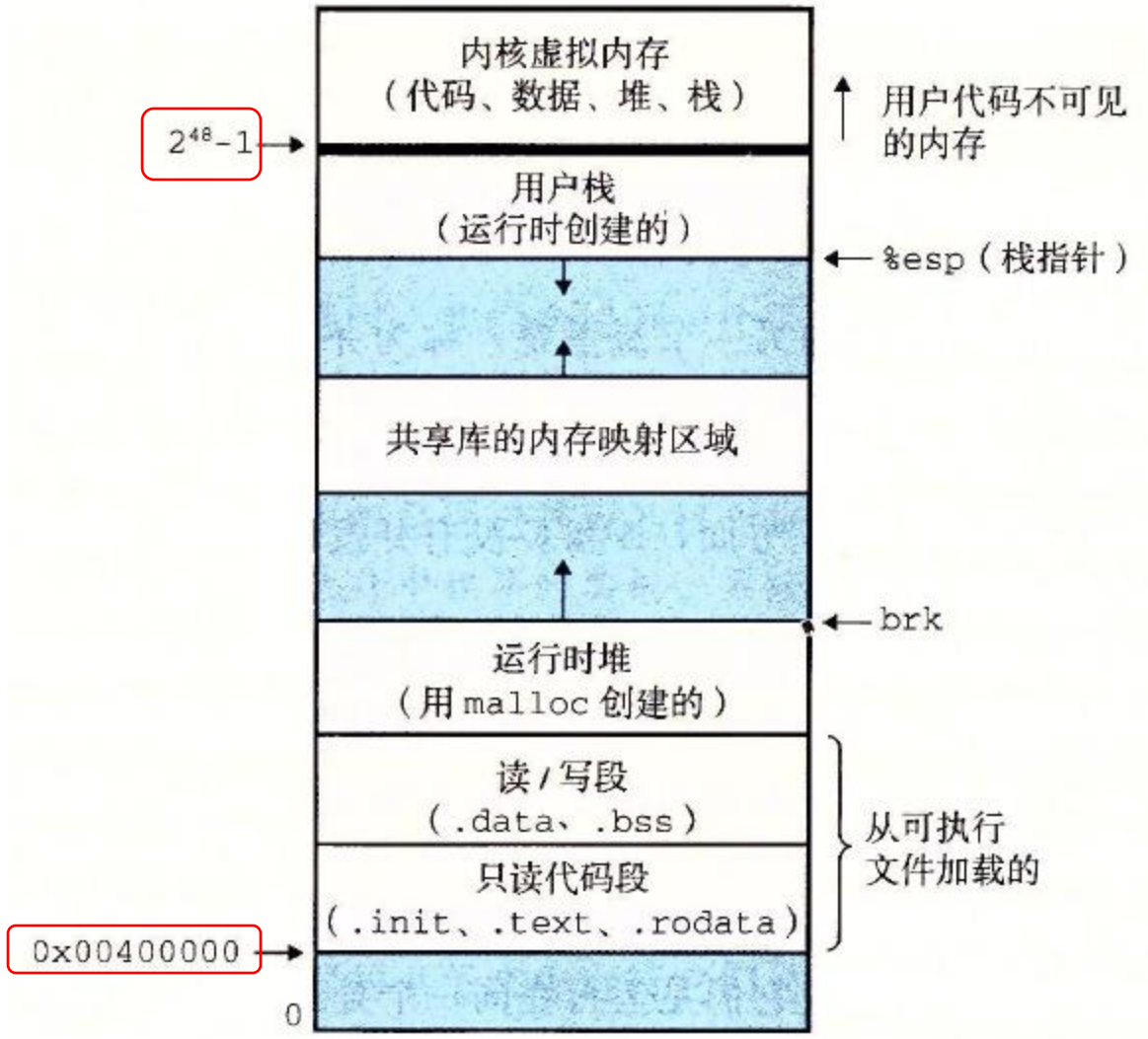
并发流的判断标准：开始结束时间

并行流⊆并发流

进程：逻辑流与私有地址空间

地址空间：内存中所有地址的集合，
0~2^n-1
私有地址空间：不能被其他进程读写

通用结构如右图



进程：用户、内核模式与上下文切换

内核模式与用户模式
- 用模式位区分
- **用户模式：**
  - 指令受限
  - 访问**内核区**的代码和数据
  - only by系统调用接口---直接引用×
- **内核模式：**超级用户模式，操作不受限
- **只有通过异常**，进程可以从用户模式陷入内核模式

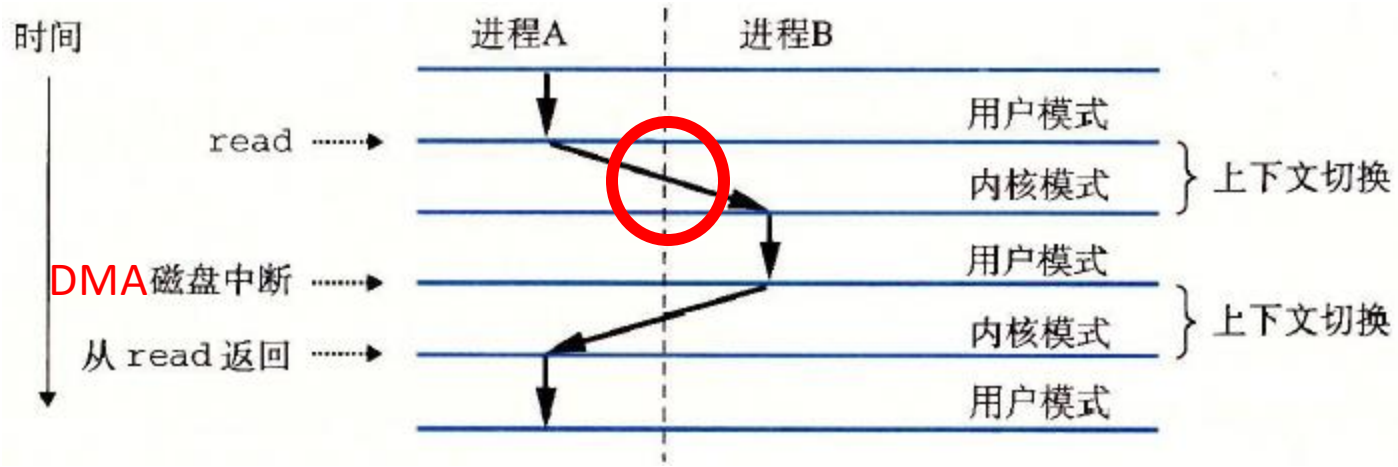
调度：控制流从一个进程传递到另一个进程，通过内核中的调度器代码处理
- 调度通过上下文切换完成

没有单独的内核进程，而是作为现有进程的一部分运行。

上下文切换：较高层形式的异常控制流

上下文的含义：
- 页表
- 进程表
- 文件表
- .....

发生的情况：
- 内核代表用户系统调用 read sleep
- 中断



进程控制：C中的系统调用函数

一系列系统函数或者系统调用接口。后者例如fork(), waitpid(), execve()，它们封装了更底层的操作系统提供给用户的系统调用

获取进程ID（PID）：

```
pid_t getpid(void);
pid_t getppid(void);
```

getpid：返回调用者PID

getppid：返回调用者父进程PID

```
1  void unix_error(char *msg) /* Unix-style error */
2  {
3      fprintf(stderr, "%s: %s\n", msg, strerror(errno));
4      exit(0);
5  }
```

创建和终止进程：

- 进程总是处在**运行、停止、终止**三种状态之一
- 停止的进程被挂起，不会被调度，收到对应信号时重新运行
- 终止的原因有三种：**信号，从主程序返回，调用exit**
- exit:以status退出状态终止进程

```
void exit(int status);
```

```
pid_t fork(void);
```

Returns: 0 to child, PID of child to parent, −1 on error

错误处理：

- 一般用-1表示发生错误
- 同时设置全局变量errno表示错误类型
- 可以通过错误包装函数简化代码

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

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

进程控制：C中的创建子进程

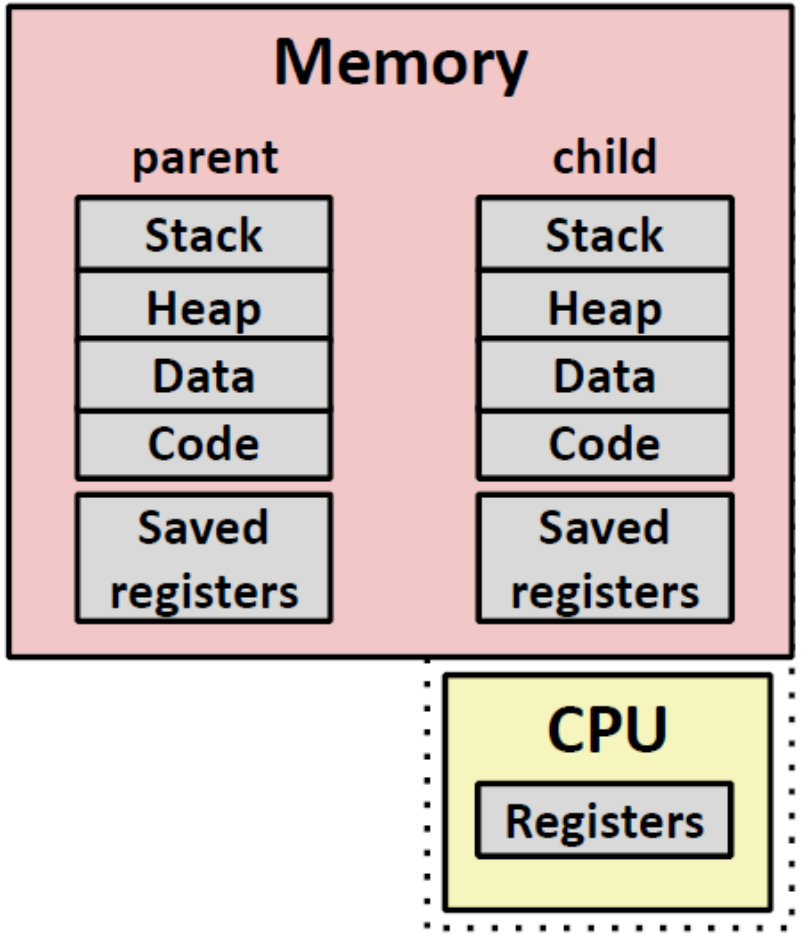
创建子进程：fork()

- 新创立的子进程除了PID，与父进程几乎是一致的

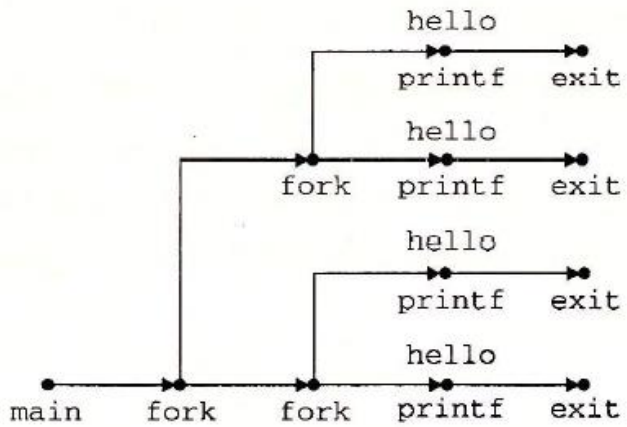
```
pid_t fork(void);
```

特点：

- **调用一次，返回两次**，子进程中返回0，父进程中返回子进程PID，错误返回-1
- **并发执行**
- **相同但是独立的地址空间**
  - 两个上下文在内存中相互独立
- **共享文件**，子进程也会继承父进程打开的所有文件（如stdout）



```
1   int main()
2   {
3       Fork();
4       Fork();
5       printf("hello\n");
6       exit(0);
7   }
```



- 下列程序可能的输出是：
  - A. 1 2 -1 0
  - B. 0 0 -1 1
  - C. 1 -1 0 0
  - D. 0 -1 1 2

A

```
int count = 0;
int pid = fork();
if (pid == 0) {
    printf("count = %d\n", --count);
} else {
    printf("count = %d\n", ++count);
}
printf("count = %d\n", ++count);
```

对有向图的处理方法

1. 拓扑排序

对一个有向无环图(Directed Acyclic Graph简称DAG)G进行拓扑排序，是将G中所有顶点排成一个线性序列，使得图中任意一对顶点u和v，若边<u,v>∈E(G)，则u在线性序列中出现在v之前。通常，这样的线性序列称为满足拓扑次序(Topological Order)的序列，简称拓扑序列。简单的说，由某个集合上的一个偏序得到该集合上的一个全序，这个操作称之为拓扑排序。

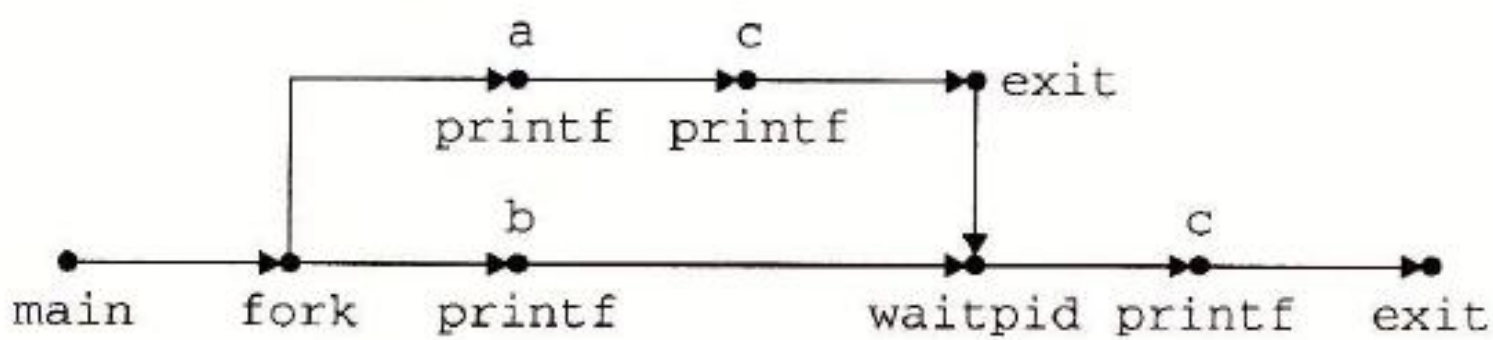
练习题 8.3 列出下面程序所有可能的输出序列：

code/ecf/waitprob0.c

```
1   int main()
2   {
3       if (Fork() == 0) {
4           printf("a"); fflush(stdout);
5       }
6       else {
7           printf("b"); fflush(stdout);
8           waitpid(-1,
9       }
10      printf("c"); ff
11      exit(0);
12  }
```

子父进程可以任意交替

bacc、acbc、**abcc**



进程控制：回收子进程

```
pid_t waitpid(pid_t pid, int *statusp, int options);
```

waitpid()是另一个重点

- 僵尸子进程原则上要回收
- 如果子进程尚未结束而父进程结束了，则它们都变成孤儿，由init代管（PID为1)
- Init进程是系统启动时由内核创建的，不会终止。

成功回收一个子进程就会返回

- 返回值为**成功回收的子进程的PID**
- 任何子进程都未终止：0
- 若错误则返回-1（错误类型保存在？）
    - errno

pid参与回收的子进程，

- pid>0则只有PID=pid的子进程参与回收，
- Pid=-1则全部子进程参与回收
- 还支持其他类型，但是本书没有讨论

options决定默认行为

- 0（默认情况）：等待直至一个子进程终止，回收并返回
- WNOHANG：若无子进程终止，直接返回
- WUNTRACED：子进程停止也会返回PID
- WCONTINUED：子进程从停止重新执行也会返回PID
- 组合：使用“|”结合后缀记忆

statusp若非空，则在其指向的status中保存一些状态信息
可以通过一些宏来解析信息（见书）

青春版：wait(statusp) 默认pid=-1，options=0

进程控制：休眠和运行

- **sleep**指定**时间**，若时间到了则返回0，否则返回剩余时间
- **pause**使得调用函数休眠直到接收信号，总返回-1

```
unsigned int sleep(unsigned int secs);

int pause(void);
```

- **execve**加载并运行程序
- execve在**当前进程的上下文**中加载并运行一个程序
- 除非出现错误否则**不返回**
- 不创建新进程，而是覆盖当前进程的地址空间

```
int execve(const char *filename, const char *argv[],
           const char *envp[]);
```

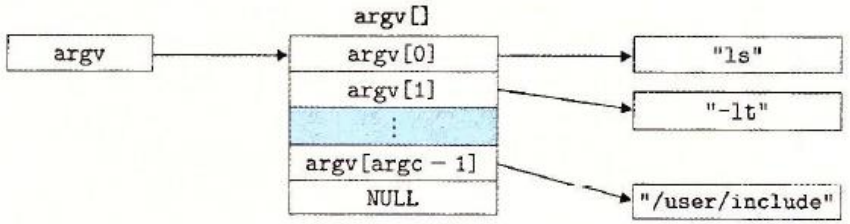


图 8-20　参数列表的组织结构

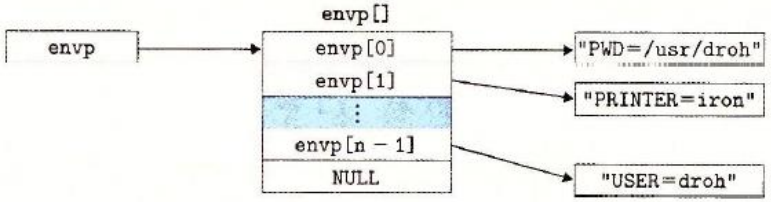
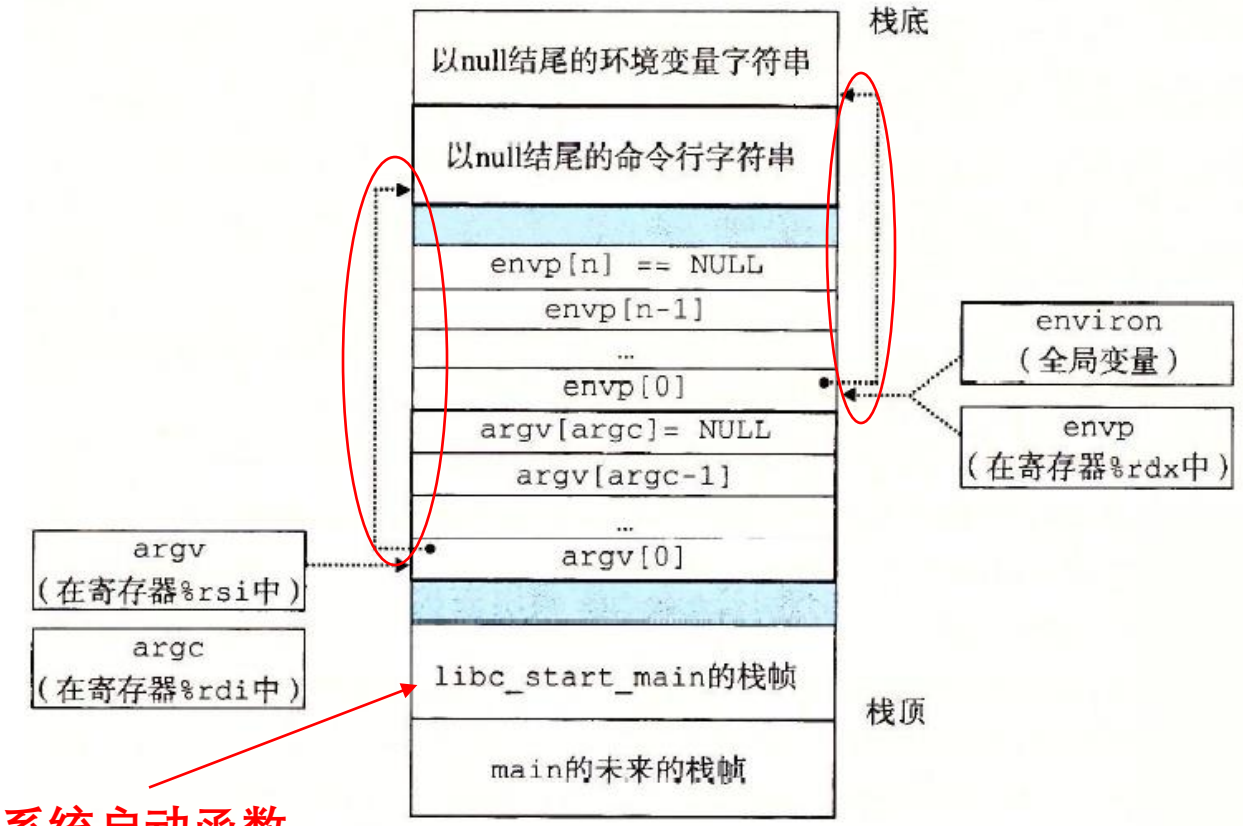


图 8-21　环境变量列表的组织结构

*****getenv、setenv、unsetenv：操作环境数组，查阅书本**



进程控制：shell

shell的部分实现

- execve不创建新进程，而是在当前进程的上下文中运行指定程序
- 因此，**fork与execve一起使用**可以在新进程中加载并运行一个程序
- 同时，利用**waitpid可以回收前台子进程**，避免僵死进程的内存占用。利用其他系统调用函数，可以完成对一些shell命令的实现

- 以上三者结合使用，可以完成对shell的部分实现，详见书本图8-23、8-24

code/ecf/shellex.c

```
1   #include "csapp.h"
2   #define MAXARGS   128
3
4   /* Function prototypes */
5   void eval(char *cmdline);
6   int parseline(char *buf, char **argv);
7   int builtin_command(char **argv);
8
9   int main()
10  {
11      char cmdline[MAXLINE]; /* Command line */
12
13      while (1) {
14          /* Read */
15          printf("> ");
16          Fgets(cmdline, MAXLINE, stdin);
17          if (feof(stdin))
18              exit(0);
19
20          /* Evaluate */
21          eval(cmdline);
22      }
23  }
```

code/ecf/shellex.c

- ECF上半节概念、记忆性内容较多，为本章之后的内容作铺垫，考试也会在选择中考察一些概念理解以及知识点记忆
- fork的返回两次是比较容易搞混的，善用进程图

祝大家shelllab顺利！

Processes & threads & coroutines

许珈铭

Rings & modes

许珈铭

Practice

刘昕垚 杨斯琪

The End