2023-2024学年春季学期

计算机体系结构安全
Computer Architecture Security

授课团队：史岗，陈李维

计算机体系结构安全

Computer Architecture Security

[第5次课] 计算机内存架构基础（一）

授课教师：陈李维

授课时间：2024. 3. 25

内容概要

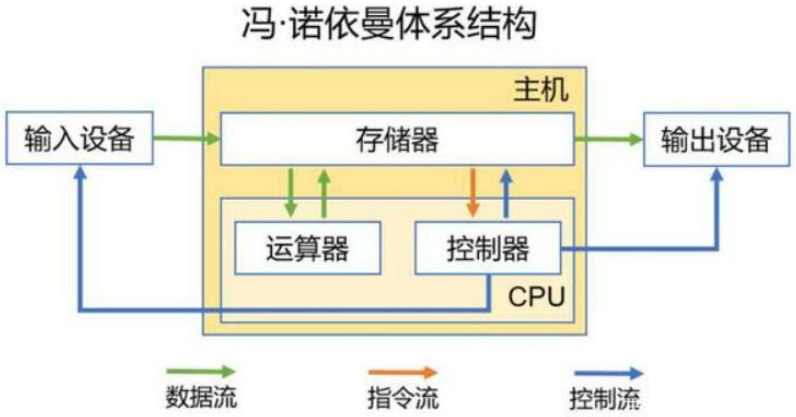
- 计算机存储结构简介
- 计算机内存架构基础知识
- 计算机内存漏洞详细介绍
  - 缓冲区溢出漏洞
  - 堆漏洞
  - 内存信息泄露漏洞
  - 其他内存漏洞
- 总结

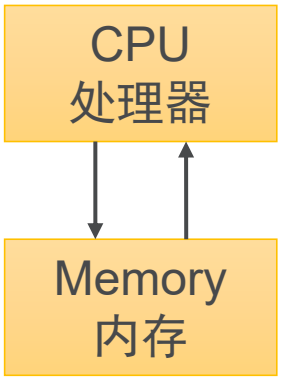
内容概要

- 计算机存储结构简介
- 计算机内存架构基础知识
- 计算机内存漏洞详细介绍
  - 缓冲区溢出漏洞
  - 堆漏洞
  - 内存信息泄露漏洞
  - 其他内存漏洞
- 总结

冯诺依曼结构的基本特点

- 计算机硬件由运算器、控制器、**存储器**、输入设备和输出设备五大部分组成。
  - 哪些部分最重要，是否可以进一步化简?
- 计算机处理的数据和指令一律用二进制数表示；
- 指令和数据不加区别混合存储在**同一个存储器**中；
- 顺序执行程序的每一条指令；



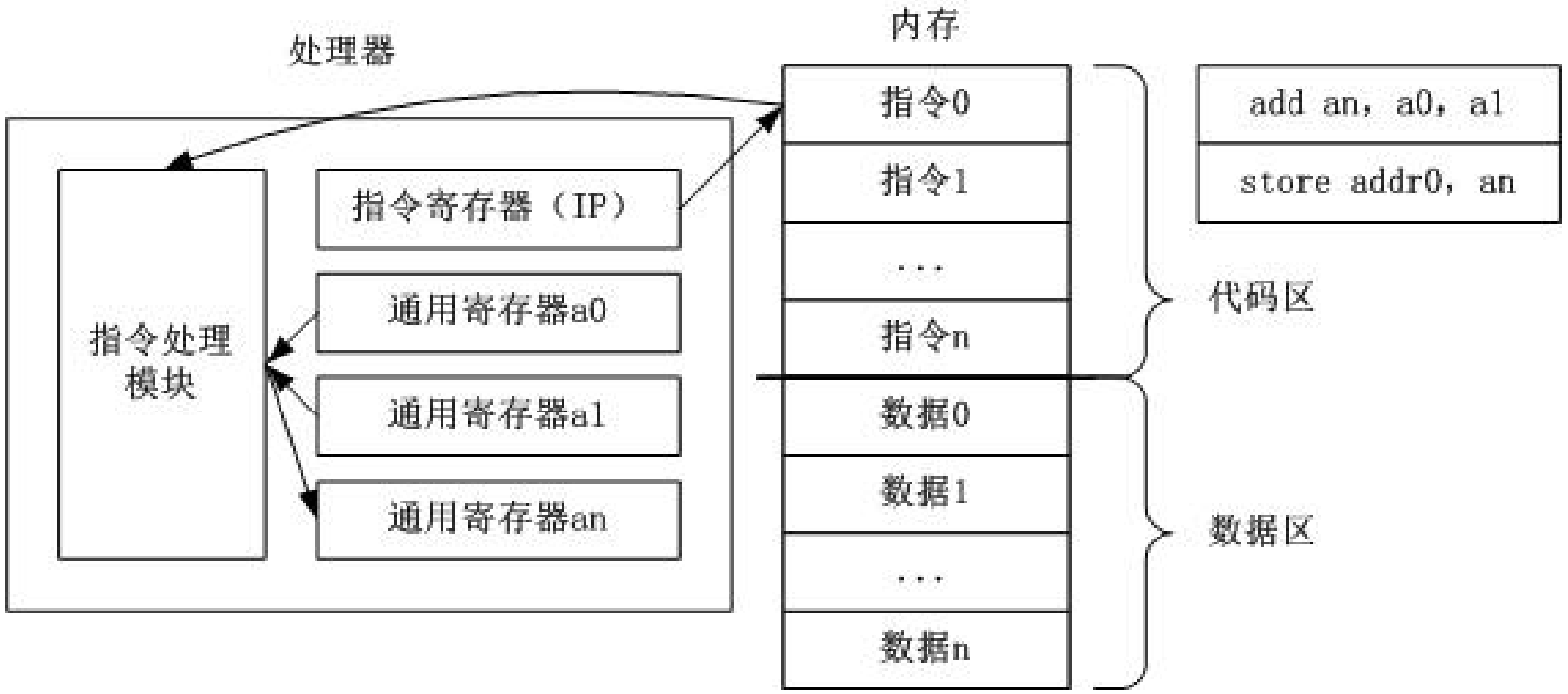


程序运行简单模型

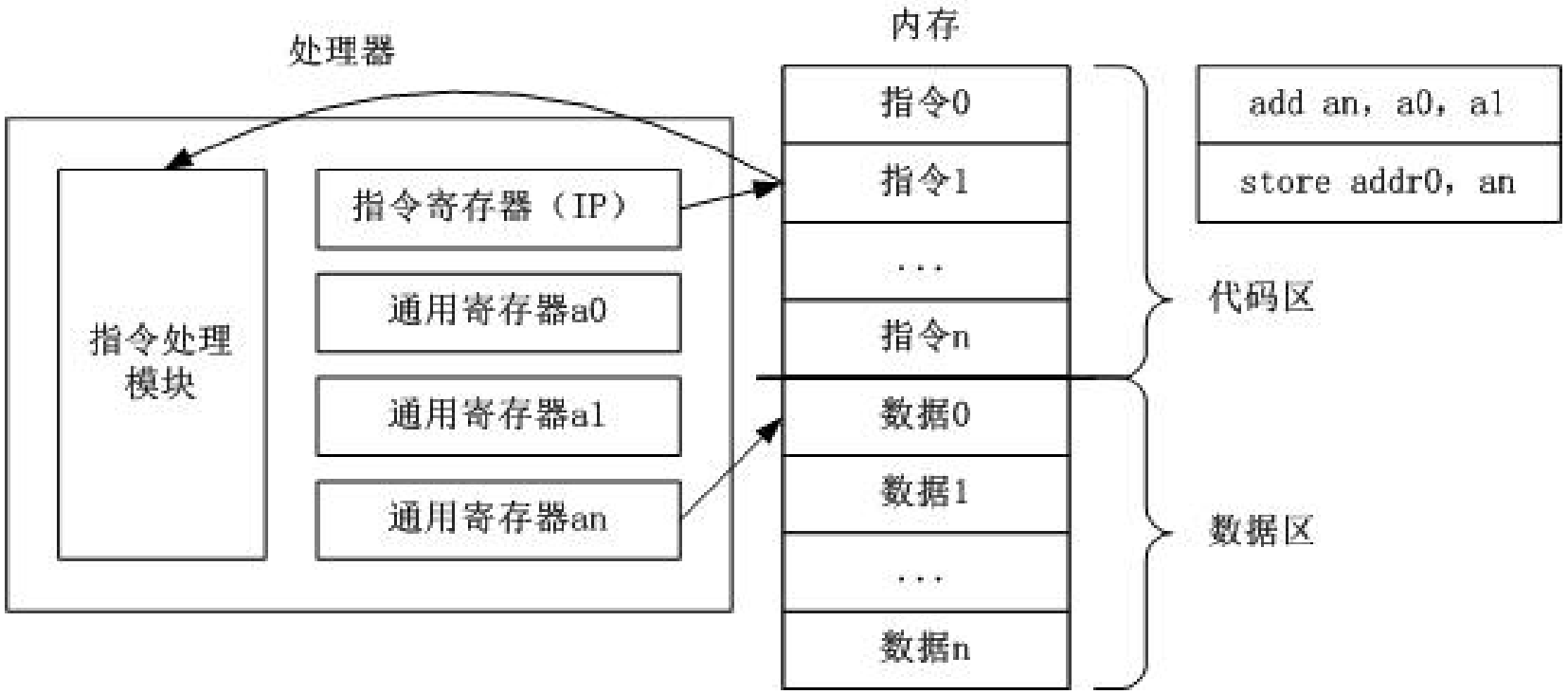
一个程序是如何在计算机系统上运行的?

- 硬件
  - 处理器每次从内存中读取一条指令，按照指令内容执行，然后从内存中取下一条指令执行，一直循环运行。
- 软件
  - 程序经过编译器编译链接，最后变成了由指令和数据组成的可执行二进制文件，保存在内存中。
  - 处理器从内存中读取二进制文件对应的指令和数据，运行程序。
- 计算机系统的运行，就是一条一条的指令在处理器上的执行。

一个程序是如何在计算机系统上运行的?



一个程序是如何在计算机系统上运行的?



硬件组成：**处理+存储**

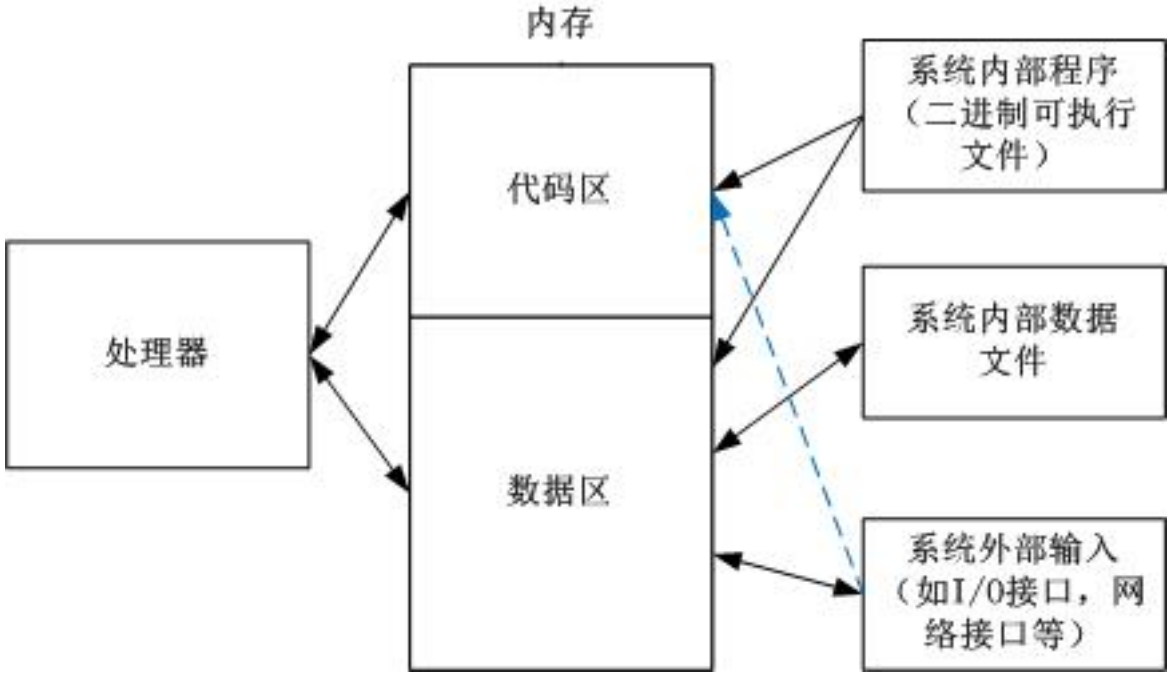
- **处理器**
  - 指令处理模块（取指，译码，执行，提交，写回）
  - 寄存器堆
    - 通用寄存器，用于暂存少量当前待处理的数据。
    - 特殊寄存器，如指令寄存器和栈寄存器等。用于保存一些特殊的数据，如指令指针。
- **存储器**
  - 内存，硬盘
    - 存储所有指令和数据

软件组成：程序+数据，**指令+数据**

- 程序，即可执行二进制文件，保存在内存中
  - 指令
  - 程序内含数据
- 中间数据，程序运行过程中生成的中间数据，保存在内存中
  - 静态数据
  - 动态数据
- 输入数据，先存入内存，再进行处理
  - 系统内部文件
  - 系统外部输入数据（用户输入，网络数据等）

计算机指令和数据的来源

- 指令来源：系统内部程序，**系统外部输入（如安装程序、脚本等）**
- 数据来源：系统内部程序和文件，系统外部输入，程序动态生成



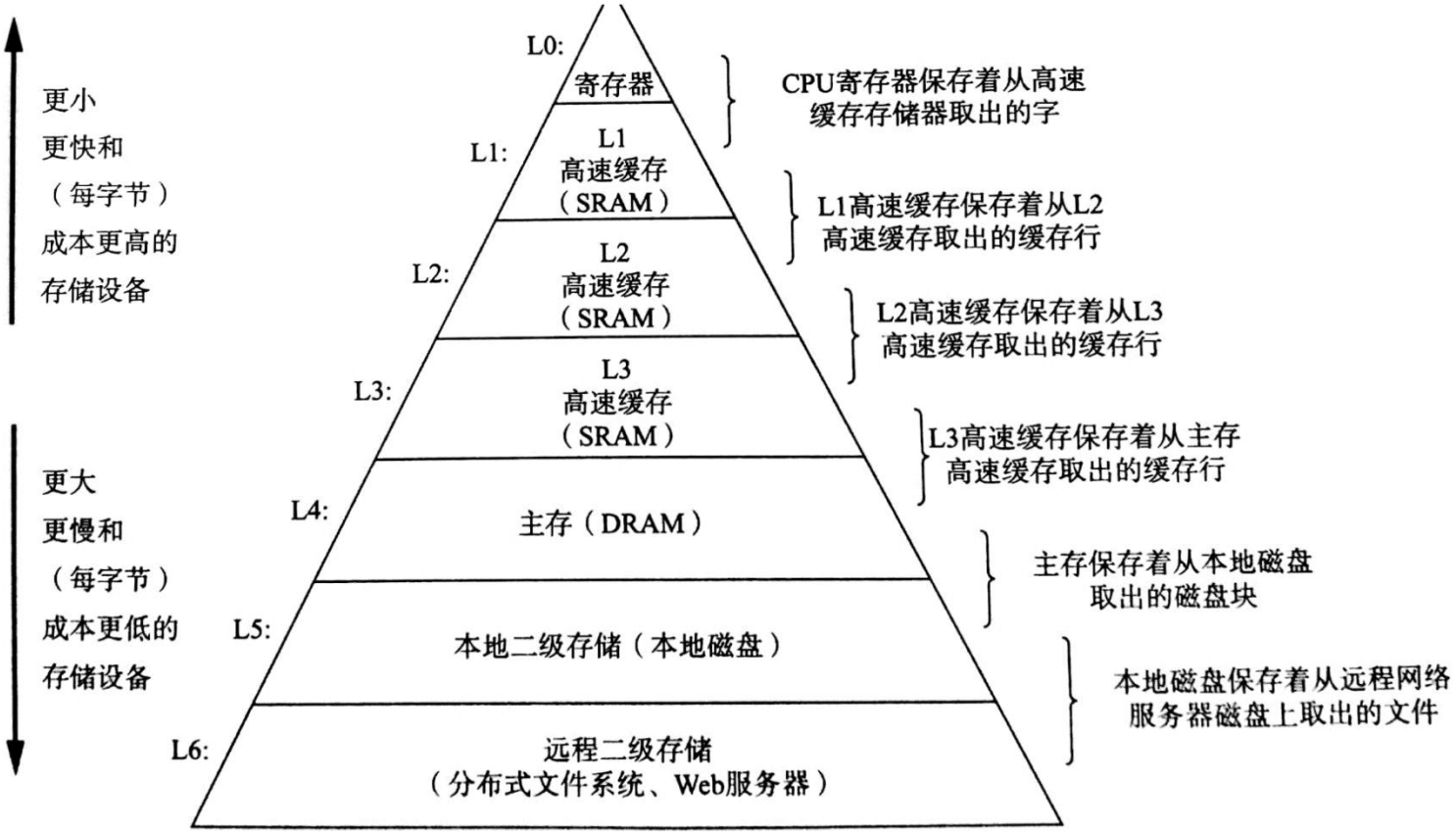
计算机存储层次结构图

寄存器堆

高速缓存
cache

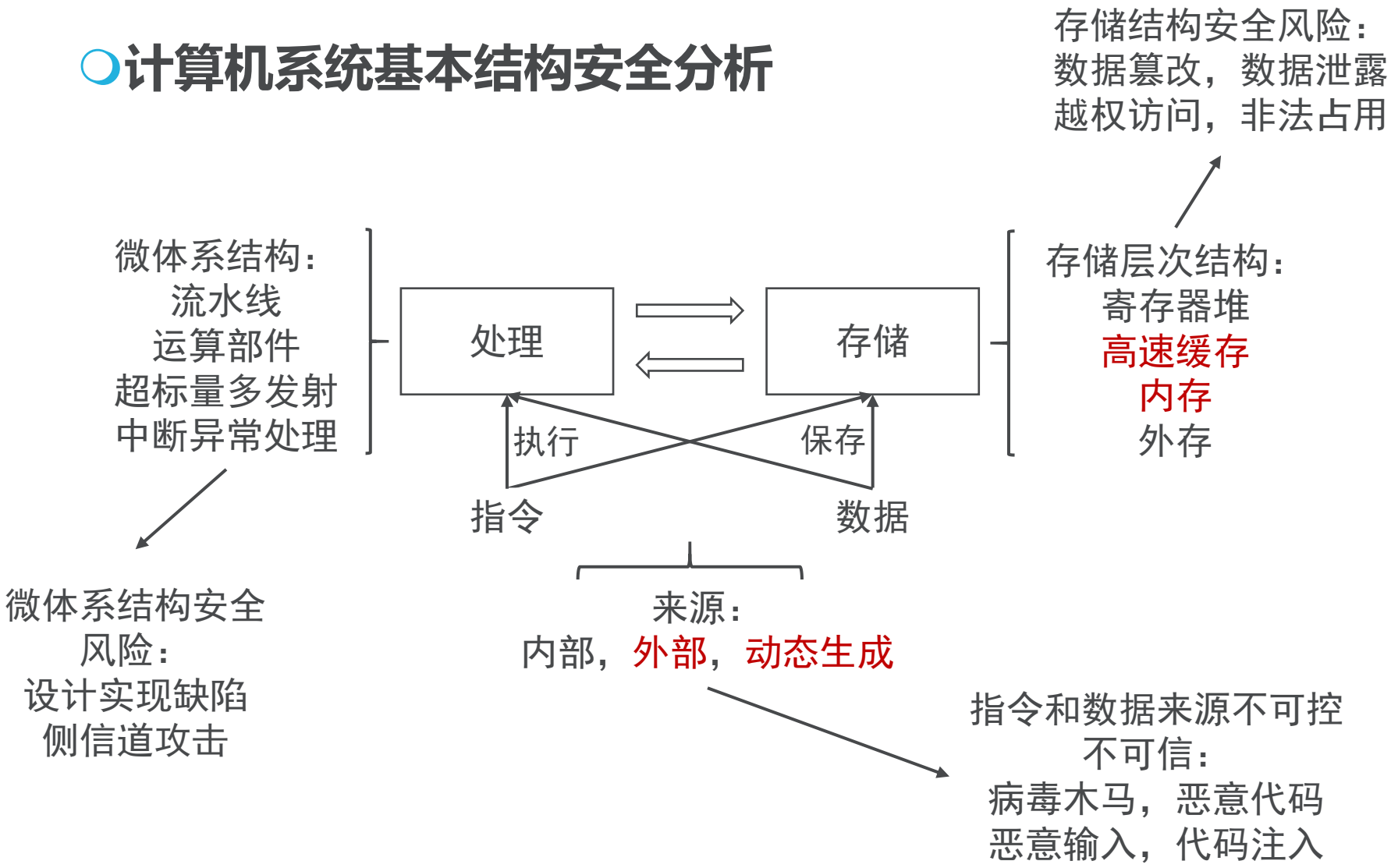
内存（主存）

外存（硬盘，
光盘，磁盘）



提升效率，降低成本

计算机系统基本结构安全分析



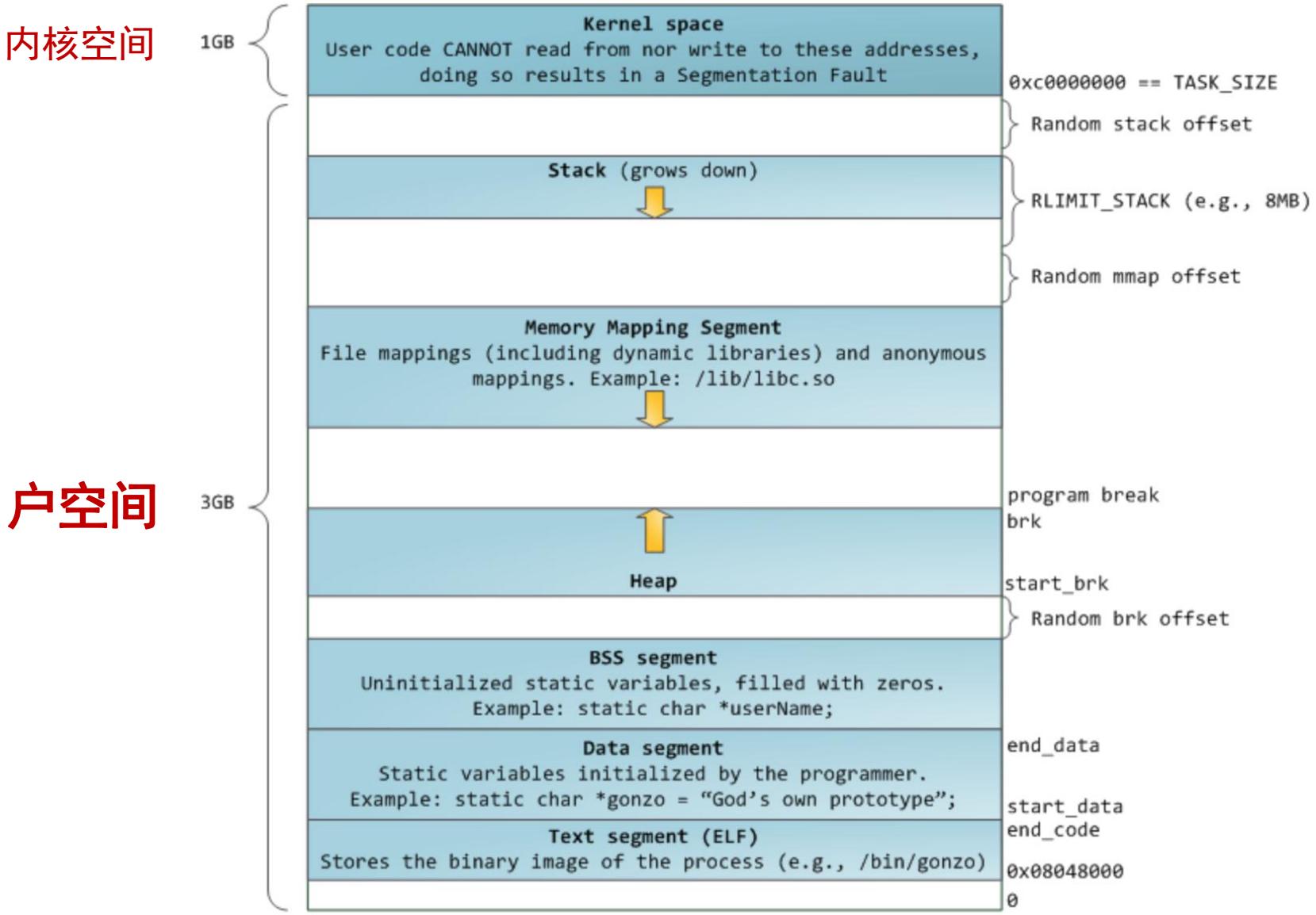
计算机存储层次的安全分析

- 高速缓存cache
  - 对软件不可见，用户无法直接操控，属于CPU的一部分（微体系结构）
  - 主要攻击思路：侧信道攻击
  - 主要安全影响：信息泄露
- 主存（内存）
  - 所有的指令和数据都保存在内存中，软件可见可控，用户可以直接控制和管理，是攻击的重灾区
  - 也就是说，只要控制了内存，就能实现任何操作，也就控制了整个计算机系统。
  - 攻击思路：非法读，非法写，非法执行，非法占用，侧信道攻击。。。
  - 安全影响：系统被控制，系统崩溃，信息泄露。。。

内容概要

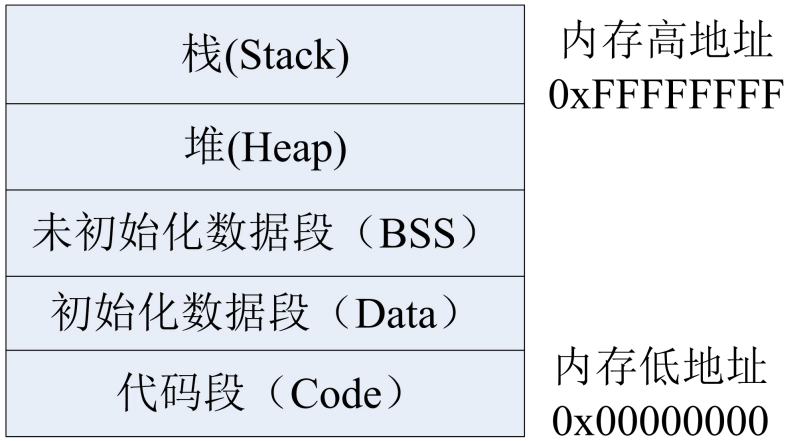
- 计算机存储结构简介
- **计算机内存架构基础知识**
- 计算机内存漏洞详细介绍
  - 缓冲区溢出漏洞
  - 堆漏洞
  - 内存信息泄露漏洞
  - 其他内存漏洞
- 总结

32位环境下典型的内存空间布局



简化的用户内存空间布局

- 代码段：存储指令（Text，Code）
- 数据段：存储程序中内含的数据（静态）
  - 初始化数据段（Data）
  - 未初始化数据段（BSS）
- 堆栈段：存储程序运行过程中产生的中间数据（动态）
  - 栈（Stack）
  - 堆（Heap）



栈（Stack）

- 是一块**连续的**内存空间
  - 先入后出
  - 生长方向与内存的生长方向正好相反，从高地址向低地址生长
- 栈用于保存程序运行的**中间数据**
  - 函数的参数
  - 函数返回地址
  - 一些通用寄存器(EDI,ESI...)的值
  - 当前正在执行的函数的局部变量

堆（Heap）

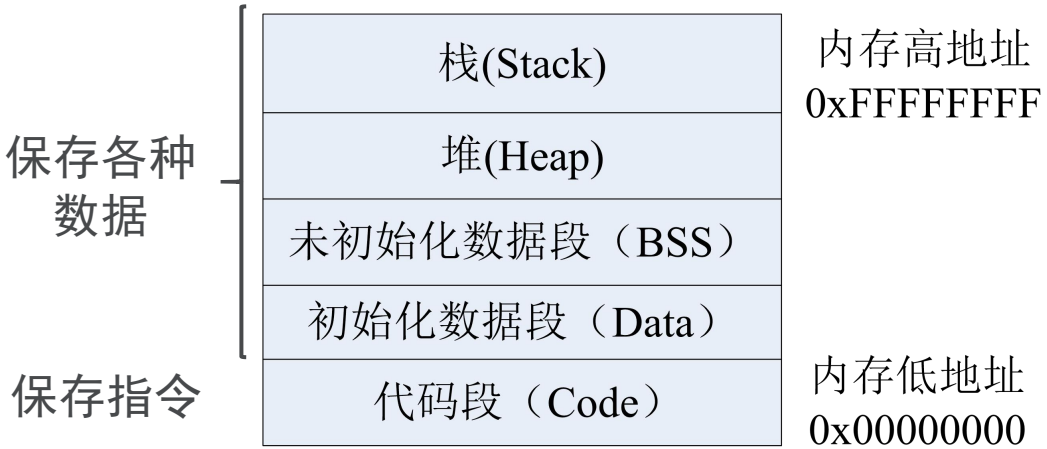
- 是位于数据段之上的一段内存区域。
- 堆允许程序在运行时**动态**的申请一块内存空间，用于存放**用户自定义的数据**。
- 堆的使用比栈更加灵活。

栈和堆的比较

- 栈：
  - 由系统自动分配，先进后出
  - 存放函数的参数、局部变量的值等
  - 向低地址扩展，是一段连续的内存空间
  - 方便快捷，自由度低
- 堆：
  - 需要程序员自己管理，链表结构，顺序随意
  - 存放程序员自定义的数据
  - 向高地址扩展，存放区域可能不连续
  - 灵活可控，自由度高

内存的安全分析

- **所有的指令和数据都保存在内存中**
  - **攻击目标：**
    - 指令+数据：病毒木马，恶意程序
    - 指令：恶意代码，恶意脚本，代码注入（把数据作为指令）
    - 数据：恶意输入，数据篡改，数据泄露
    - 空间：资源耗尽，空间占用
  - **攻击手段：**
    - 非法读，非法写，非法执行，非法占用，侧信道攻击。。。



内存的主要安全问题

- 内存损坏漏洞（Memory corruption bug，简称**内存漏洞**）
  - 用户或程序员**自行管理**内存导致的安全漏洞
  - 举例：缓冲区溢出漏洞，use after free漏洞
- 内存管理问题
  - 操作系统、编译器等对内存的**管理机制**存在问题，缺少足够的检查和管控
  - 举例：内存资源耗尽，占用内存未释放
- 内存设计缺陷
  - 内存的**设计和实现**存在安全缺陷
  - 举例：Rowhammer攻击

内存损坏漏洞（内存漏洞）

- 是目前最古老最经典也最严重的安全问题之一
  - 古老：70-80年代就已经出现，是最早出现的安全漏洞之一
  - 经典：原理简单，难以根治，长期存在，讲课最爱
  - 严重：一直占据着最危险漏洞排行榜的前三位，漏洞数量极多，漏洞分布广泛，漏洞危害性极大
- 原理分析
  - 根源：一些常用低级编程语言，如C、C++等，**允许用户直接管理内存，而且缺少对边界、指针的检查。**
  - 定义：由于软件设计或实现的错误，导致攻击者能够构造特定的**输入数据，非法修改或读取**内存中的数据，从而实现攻击者预期的攻击操作。

内存漏洞分类

- 按照漏洞所在位置，内存漏洞可以分为：
  - 栈漏洞
  - 堆漏洞
  - 数据段漏洞，如BSS漏洞
- 按照漏洞利用方式，内存漏洞可以分为：
  - 空间漏洞
    - 本质：**指针越界**
    - 缓冲区溢出漏洞（栈溢出，堆溢出，BSS溢出）
  - 时间漏洞
    - 本质：**悬空指针**
    - Use after free漏洞

类型安全语言：一种解决思路

- JAVA，Python等类型安全的语言，可以**从根源上避免出现内存漏洞**，如缓冲区溢出、悬空指针等。
- 类型安全语言的基本特征：
  - 没有指针（避免悬空指针、指针越界等）
  - 检查数组对象边界（避免缓冲区溢出）
  - 自动的垃圾回收（避免堆漏洞）

类型安全语言的问题

- JAVA，Python等类型安全的语言，虽然可以避免出现内存漏洞，但是存在其他安全漏洞，如即时编译的问题、**反序列化漏洞**等。
- 由于**兼容性**难以实现，现实世界中依然需要使用C，C++语言，很多常见软件都是C、C++语言编写的。
  - 操作系统内核
  - 类型安全高级语言的解释器（JAVA、Python等）
  - 驱动，库文件等
- C语言的**效率**要高于JAVA等类型安全语言，适合用于编写效率要求高的应用程序。

小结

- 介绍程序运行基本模型、计算机存储层次结构和计算机内存结构，让大家对计算机内存架构有一定的了解
- 简单介绍了内存安全相关的基本概念，如内存布局、栈和堆、内存漏洞的定义和分类等，理解内存安全的基本原理

内容概要

- 计算机存储结构简介
- 计算机内存架构基础知识
- 计算机内存漏洞详细介绍
  - **缓冲区溢出漏洞**
  - **堆漏洞**
  - **内存信息泄露漏洞**
  - **其他内存漏洞**
- 总结

缓冲区溢出漏洞

- 缓冲区（buffer），是程序运行期间在内存中分配的一个**连续的存储空间**，用于存放程序运行所需的各种数据。
- 缓冲区溢出（buffer overflow），是指向固定长度的缓冲区写入超出预先分配长度的内容，造成缓冲区数据溢出，而覆盖了缓冲区相邻的内存空间。

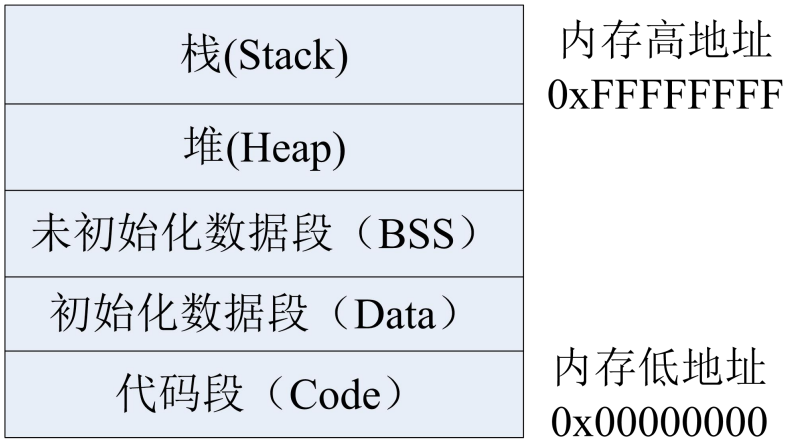
缓冲区溢出漏洞示例：

```
void func(char *input)
{
    char buffer[16];
    strcpy(buffer， input);
}
```

- 在函数func中，strcpy()将直接把input中的内容复制到buffer中。这样只要input的长度大于16，就会造成buffer的溢出，使程序运行出错。
- 存在像strcpy这样问题的标准函数还有strcat()，sprintf()，vsprintf()，gets()，scanf()以及在循环内的getc()，fgetc()，getchar()等。

缓冲区溢出漏洞

- 缓冲区是内存的具体化，是一段连续的内存空间。
  - 缓冲区和内存的关系，相当于进程和程序的关系
- 运行一个程序时，计算机会在内存中开辟一段连续的内存空间，即缓冲区。
- 我们所说的内存布局，实际上就是一个程序的缓冲区在内存中的布局。



缓冲区溢出漏洞

缓冲区溢出漏洞分类：

- 按照在内存中的不同溢出位置，缓冲区溢出漏洞分类：
  - 栈溢出漏洞
  - 堆溢出漏洞
  - BSS溢出漏洞
- 按照溢出数据类型，缓冲区溢出漏洞分类：
  - 整数溢出漏洞
  - 字符串溢出漏洞
  - 数组溢出漏洞
  - 内存空间溢出漏洞

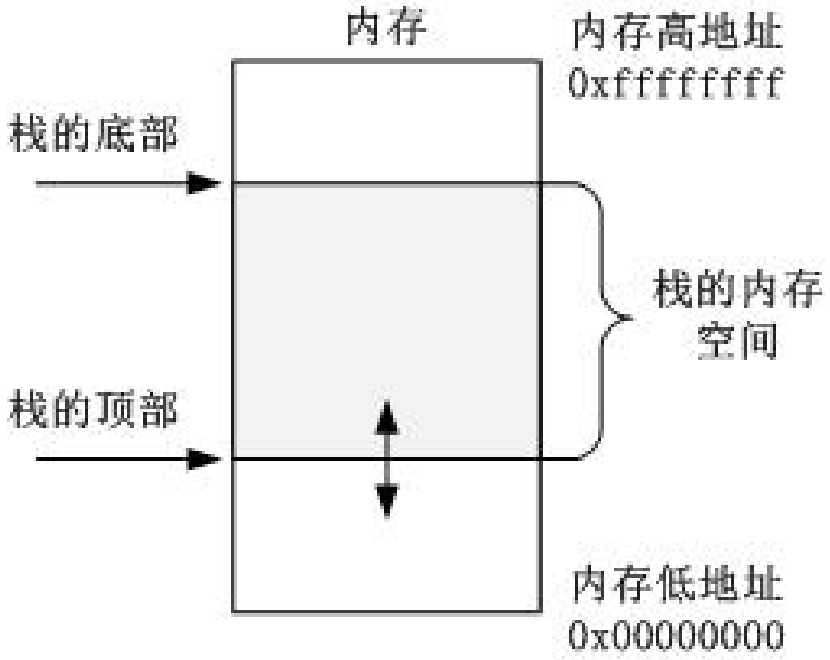
栈的基本知识

栈（Stack）

- 是一块**连续的**内存空间
  - 先入后出
  - 生长方向与内存的生长方向正好相反，从高地址向低地址生长
- 栈用于保存程序运行的**中间数据**
  - 函数的参数
  - **函数返回地址**
  - 一些通用寄存器(EDI,ESI...)的值
  - 当前正在执行的函数的局部变量

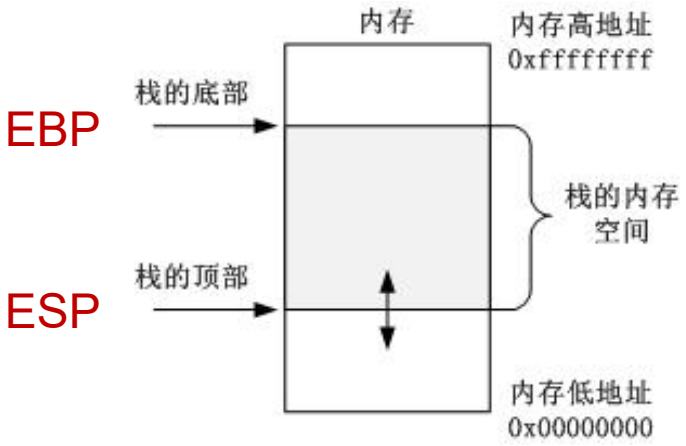
栈的正常使用

- 程序员角度：
  - int a = 0;
  - char buf[10];
  - int func(int a, int b, char *p);
- 汇编代码：
  - push %eax
  - pop %ebx
  - call func （调用func）
  - ret （函数返回）



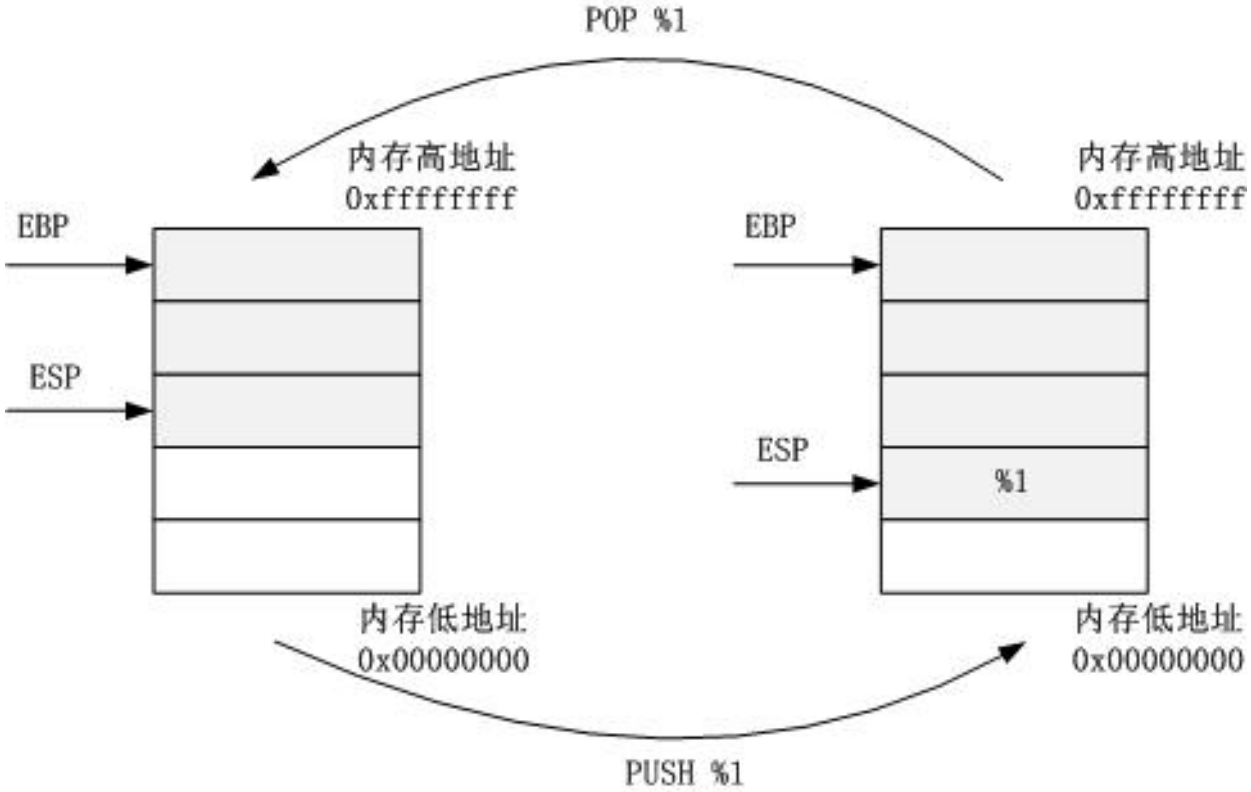
和栈相关的三个重要寄存器

- 1）SP(ESP)：extended stack pointer
  - 即栈顶指针，随着数据入栈出栈而发生变化。
- 2）BP(EBP)：extended base pointer
  - 即基地址指针，用于标识栈中一个相对稳定的位置。通过BP，可以很方便地引用函数参数以及局部变量。
- 3）IP(EIP)：extended instruction pointer
  - 指令指针，即指令寄存器，用于标示处理器当前执行的指令。



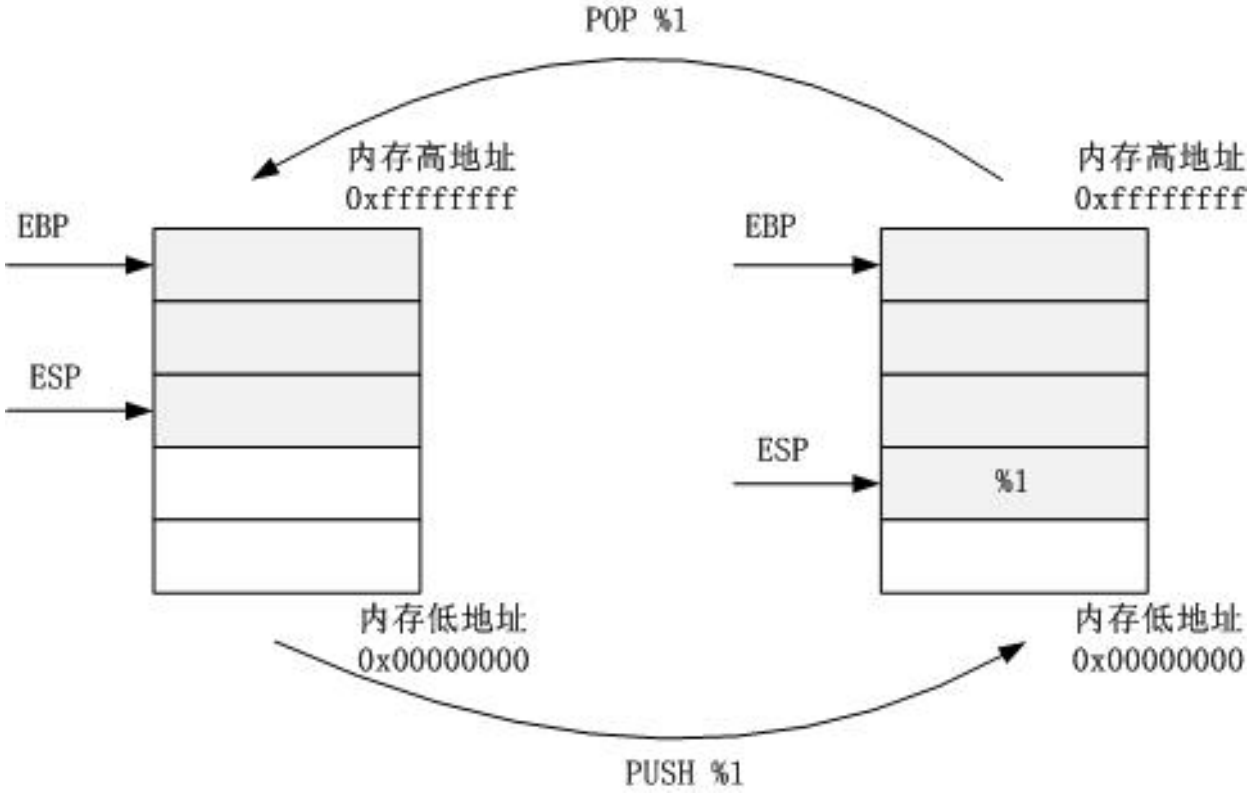
操作栈的几条基本指令

- 1) PUSH %1
  - 压栈，栈中增加一个数据，栈指针esp减4，将%1中数据存入栈中
  - sub $0x4, %esp
  - mov %1, (%esp)



操作栈的几条基本指令

- 2）POP %1
  - 出栈，栈中减少一个数据，将栈中数据存入%1中，栈指针esp加4
  - mov (%esp), %1
  - add $0x4, %esp



操作栈的几条基本指令

- 3）CALL addr
  - 函数调用，首先将返回地址压入栈顶，然后将程序跳转到当前调用函数的起始地址
  - push %eip
  - jump addr
- 4）RET
  - 函数返回，首先将栈顶地址弹出到指令寄存器EIP，然后按照该EIP继续执行程序。
  - pop %eip

函数调用具体过程

- 1）函数调用前：
  - 将参数压栈（push xxx）
  - 将当前指令寄存器压栈，作为返回地址，然后跳转到子函数执行（call sub_func）
- 2）进入子函数：
  - 将当前基地址指针压栈（push %ebp，即主函数的基地址指针）
  - 将当前栈指针拷贝给基地址指针，作为新的基地址指针（mov %esp, %ebp，即子函数的基地址指针）
  - 将栈指针减去适当的数值，为本地变量留出一定空间（sub $0x8, %esp）

函数调用具体过程

- 3）子函数返回：
  - 将当前基地址指针拷贝到栈指针，让栈指针重新指向前一个函数备份的基地址指针值（mov %ebp, %esp）
  - 将栈顶的前一个函数备份的基址寄存器值弹出，存入基址寄存器（pop %ebp，获取主函数的基地址指针）
  - 将压栈的返回地址弹出，存入指令寄存器，子函数返回，返回主函数继续执行（ret）

函数调用具体过程

func.c:

```
int func(int a, int b) {
    int retVal = a + b;
    return retVal;
}
int main() {
    int result = func(1, 2);
    return 0;
}
```

Ubuntu 16.04 64位系统:

```
gcc func.c -o func
objdump –d func > func.s
vim func.s
```

```
00000000004004d6 <func>:
  4004d6:      push   %rbp
  4004d7:      mov    %rsp,%rbp
  4004da:      mov    %edi,-0x14(%rbp)
  4004dd:      mov    %esi,-0x18(%rbp)
。。。
  4004e8:      mov   %eax, -0x4(%rbp)
  4004eb:      mov    -0x4(%rbp),%eax
  4004ee:      pop    %rbp
  4004ef:      retq
00000000004004f0 <main>:
  4004f0:      push   %rbp
  4004f1:      mov    %rsp,%rbp
  4004f4:      sub    $0x10,%rsp
  4004f8:      mov    $0x2,%esi
  4004fd:      mov    $0x1,%edi
  400502:      callq  4004d6 <func>
  400507:      mov    %eax,-0x4(%rbp)
```

栈的基本知识

```
00000000004004d6 <func>:
  4004d6:      push   %rbp    //保存main函数的rbp
  4004d7:      mov    %rsp,%rbp    //设置子函数func的rbp
  4004da:      mov    %edi,-0x14(%rbp)    //使用子函数func的栈空间（sub $0x18, %rsp）
  4004dd:      mov    %esi,-0x18(%rbp)
。。。
  4004e8:      mov   %eax, -0x4(%rbp)    //将运算结果保存在栈中
  4004eb:      mov    -0x4(%rbp),%eax    //保存子函数func的返回参数
  4004ee:      pop    %rbp    //恢复main函数的rbp（mov %rbp, %rsp）
  4004ef:      retq    //子函数func返回
00000000004004f0 <main>:
  4004f0:      push   %rbp    //保存前一个函数的rbp
  4004f1:      mov    %rsp,%rbp    //设置main函数的rbp
  4004f4:      sub    $0x10,%rsp    //为main函数分配可用的栈空间
  4004f8:      mov    $0x2,%esi    //保存子函数func的参数（64位保存在寄存器中）
  4004fd:      mov    $0x1,%edi
  400502:      callq  4004d6 <func>    //调用子函数func
  400507:      mov    %eax,-0x4(%rbp)    //将子函数func的返回结果保存在栈中
```

栈溢出漏洞利用的基本思路

- 在栈中，**函数的局部变量是一个挨着一个连续排列的**。
- 如果这些局部变量中有数组之类的缓冲区，并且程序中存在数组**越界的漏洞**，那么**越界的数组元素**就有可能破坏栈中相邻变量的值，**甚至破坏栈帧中所保存的EBP值、返回地址等重要数据**。
  - 注意：大多数情况下，局部变量在栈中的分布是相邻的，但也有可能出于编译优化等需要而有所例外。这里出于讲述基本原理的目的，可以暂时认为局部变量在栈中是紧挨在一起的。

栈溢出漏洞

- 栈溢出漏洞：就是指向栈中固定长度的数据写入**超出预先分配长度**的内容，造成栈数据溢出，而覆盖了栈数据相邻的内存空间。
- 特点
  - 缓冲区在**栈**中分配
  - 拷贝的数据过长，**超过**预先分配的长度
  - **覆盖**了栈中的函数返回地址或其它一些重要数据结构、函数指针

栈溢出漏洞示例

```
void  function(char *large_string)
{
    char buffer[4];
    strcpy(buffer, large_string);
}
 void main (int argc, char **argv)
{
    char large_string[8];
    function(large_string);
}
```

- large_string的长度为8，buffer的长度为4，将large_string拷贝到buffer，造成栈溢出。

栈溢出漏洞利用方式

- 当调用函数时
  - call指令会将返回地址（call指令下一条指令地址）压入栈
  - ret指令会把压栈的返回地址弹给EIP
- 栈溢出漏洞的利用
  - 通过缓冲区溢出漏洞修改保存在栈中的返回地址
  - 当函数调用返回时，EIP获得被修改后的返回地址，并执行shellcode
- 栈溢出漏洞本身不难理解，**困难的是对栈溢出漏洞的利用**
  - **挑战1**: 将修改后的返回地址填到正确的位置
  - **挑战2**: 返回地址能正确地指向shellcode

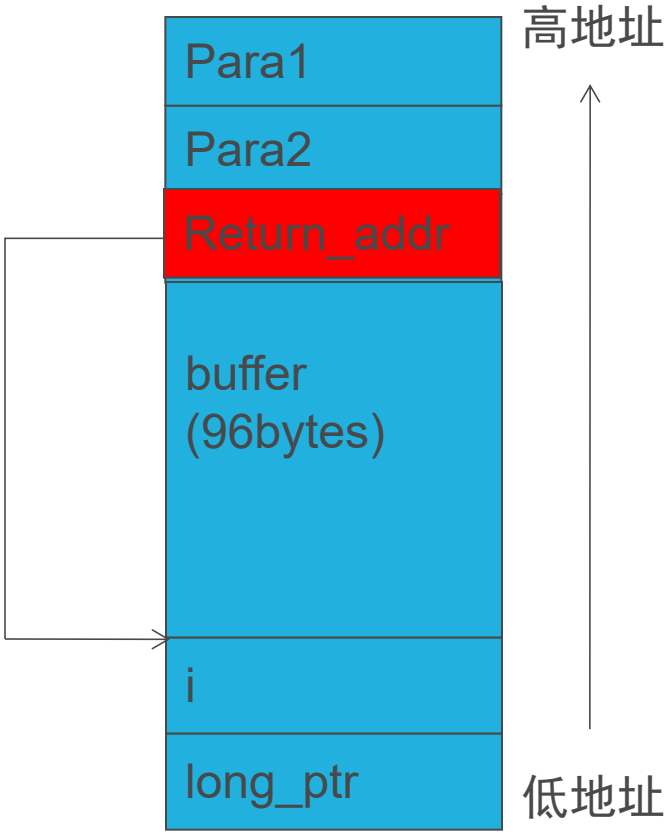
栈溢出漏洞

栈溢出漏洞示例程序：

```
char shellcode[] = "\xeb\x1f\x……";
char large_string[128];

int main(int argc, char **argv) {
    char buffer[96];
    int i;
    long *long_ptr = (long *) large_string;

    for (i = 0; i < 32; i++)
        *(long_ptr + i) = (int) buffer;
    for (i = 0; i < (int) strlen(shellcode); i++)
        large_string[i] = shellcode[i];
    strcpy(buffer, large_string);
    return 0; }
```

栈溢出漏洞利用示例

```
for ()     *(long_ptr + i) = (int) buffer;
for ()    large_string[i] = shellcode[i];
strcpy(buffer, large_string);
```



小结

- 介绍了缓冲区溢出漏洞基本概念。
- 然后，以栈为例子，详细介绍了栈的正常运行过程，栈溢出漏洞及其利用过程。
- 栈溢出漏洞原理和实现都很简单，希望大家能够动动手，自己实现一下，实际观察程序的运行过程。

讨论

- 攻击是一种艺术
  - 原理简单
  - 全面深刻的认识
  - 对细节的把握
  - 创造性思维，独到眼光
  - 动手能力

内容概要

- 计算机存储结构简介
- 计算机内存架构基础知识
- 计算机内存漏洞详细介绍
  - 缓冲区溢出漏洞
  - 堆漏洞
  - 内存信息泄露漏洞
  - 其他内存漏洞
- 总结

堆的基本知识

堆（Heap）

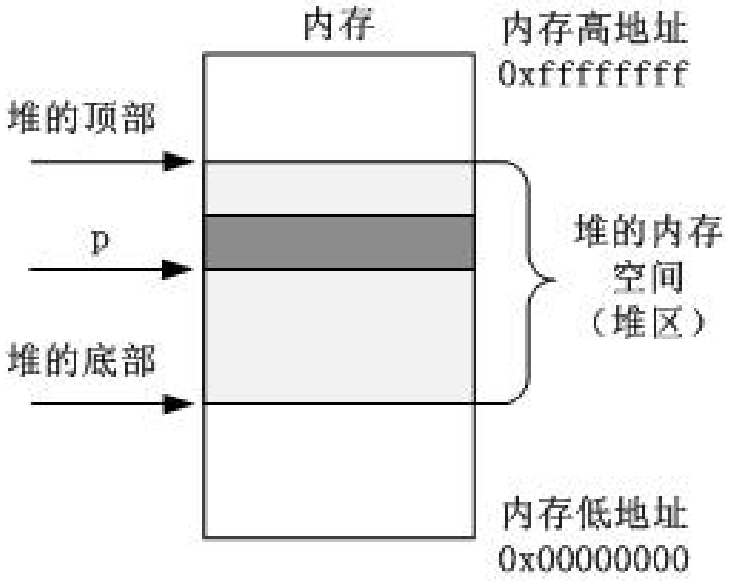
- 是位于数据段之上的一段内存区域。
- 堆允许程序在运行时**动态**的申请一块内存空间，用于存放**用户自定义的数据**。
- 堆的使用比栈更加灵活。

栈和堆的比较

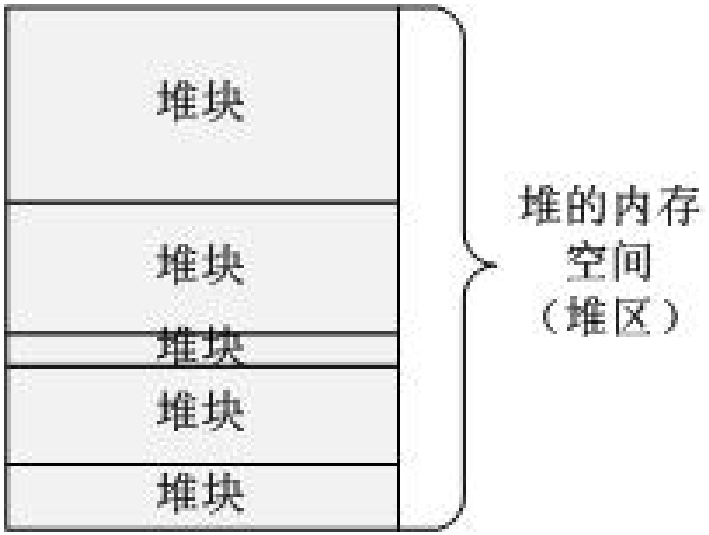
- 栈：
  - 由系统自动分配，先进后出
  - 存放函数的参数、局部变量的值等
  - 向低地址扩展，是一段连续的内存空间
  - 方便快捷，自由度低
- 堆：
  - 需要程序员自己管理，链表结构，顺序随意
  - 存放程序员自定义的数据
  - 向高地址扩展，存放区域可能不连续
  - 灵活可控，自由度高

堆的正常使用

- 程序员角度：
  - int *p = malloc(100);
  - *(p+1) = 0;
  - free(p);
- 程序内存空间：
  - 从堆内存中取出一块空闲的空间，将该空闲空间的地址赋给p。
  - 将p指向的堆内存释放。

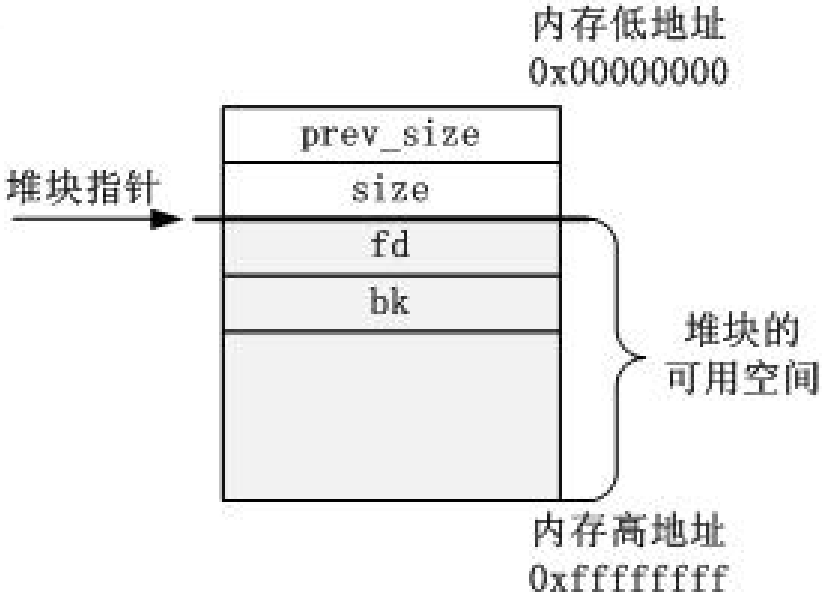


- **堆块**（chunk），是堆内存管理的**最小**操作单位，分为**空闲态和占用态**。
  - 操作系统将整个堆内存空间分为**许多个连续的大小不一的堆块**，具体的划分和释放过程是在程序运行过程中逐步进行的。
  - 从本质上来说，堆块就是内存中一块连续的区域，通过堆块中特定位置的某些标识符加以区分。

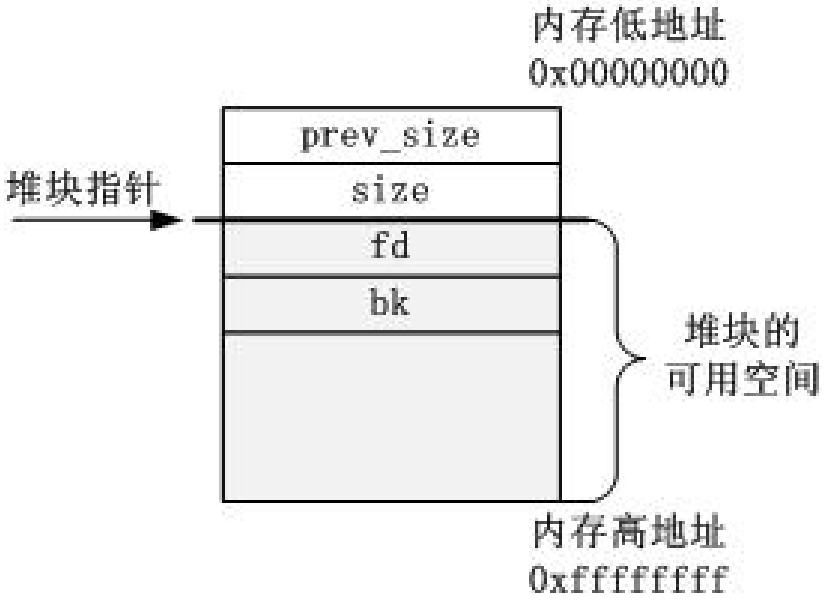


堆块的数据结构

```
struct malloc_chunk {

  INTERNAL_SIZE_T      prev_size;  /* Size of previous chunk (if free).  */
  INTERNAL_SIZE_T      size;       /* Size in bytes, including overhead. */

  struct malloc_chunk* fd;         /* double links -- used only if free. */
  struct malloc_chunk* bk;

  /* Only used for large blocks: pointer to next larger size.  */
  struct malloc_chunk* fd_nextsize; /* double links -- used only if free. */
  struct malloc_chunk* bk_nextsize;
};
```



- Linux系统中堆块的数据结构（32位系统）：
  - prev_size：前一个堆块的长度。只在前一个堆块为空闲时才会被赋值，否则会被置0。
  - size：当前堆块的长度（当前堆块的可用长度+8）
  - fd（forward）：下一个空闲堆块的地址。当堆块为空闲时，才有意义。
  - bk（backward）：上一个空闲堆块的地址。当堆块为空闲时，才有意义。



堆块的数据结构

- 每个分配的堆块总是有8字节（或16字节）的**元数据**，在这之后才是程序可以正常使用的缓冲区。
  - 32位系统中prev_size和size长度分别为32bit，4个字节，所以元数据为8个字节。
  - 64位系统中prev_size和size长度分别为64bit，8个字节，所以元数据为16个字节。
- 为了简化内存的管理，堆块的大小总是**8字节的倍数**，所以32位系统中最小堆块的大小为16字节（8+8，元数据+可用空间）。

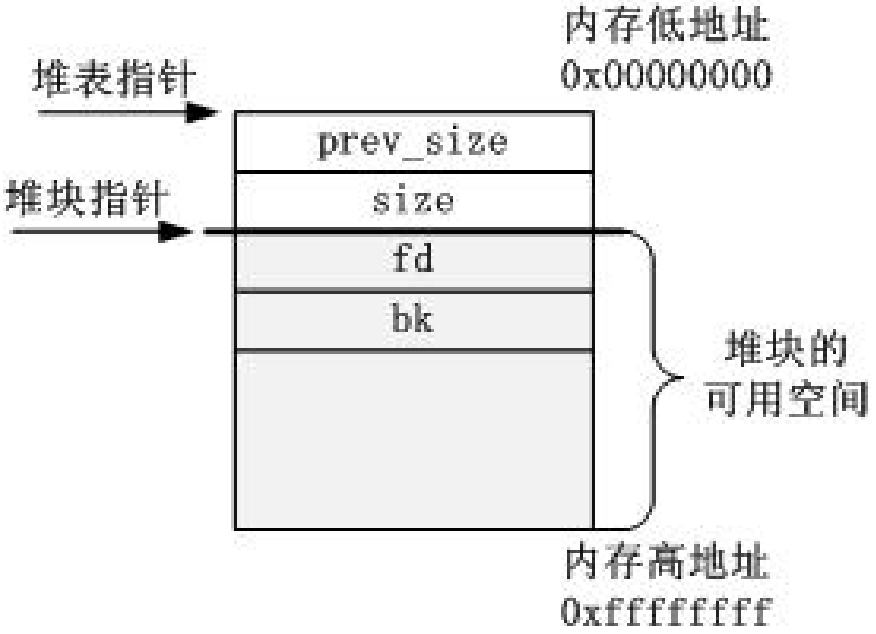
堆块的数据结构

- 因为堆块的大小总是8字节的倍数，所以size的最后3位在 正常情况下总是置0。
- 为了充分利用内存，堆管理器将size的最后3个比特位用作标志位。
  - size的第0位（最低位）用于标记前一个堆块是否已经被分配。0表示未分配，1表示已经分配。
  - size的第1位用于标记当前堆块是否为mmap分配。一般mmap用于分配空间较大的堆块。
  - size的第2位用于标记当前堆块的进程相关信息。

堆的基本知识

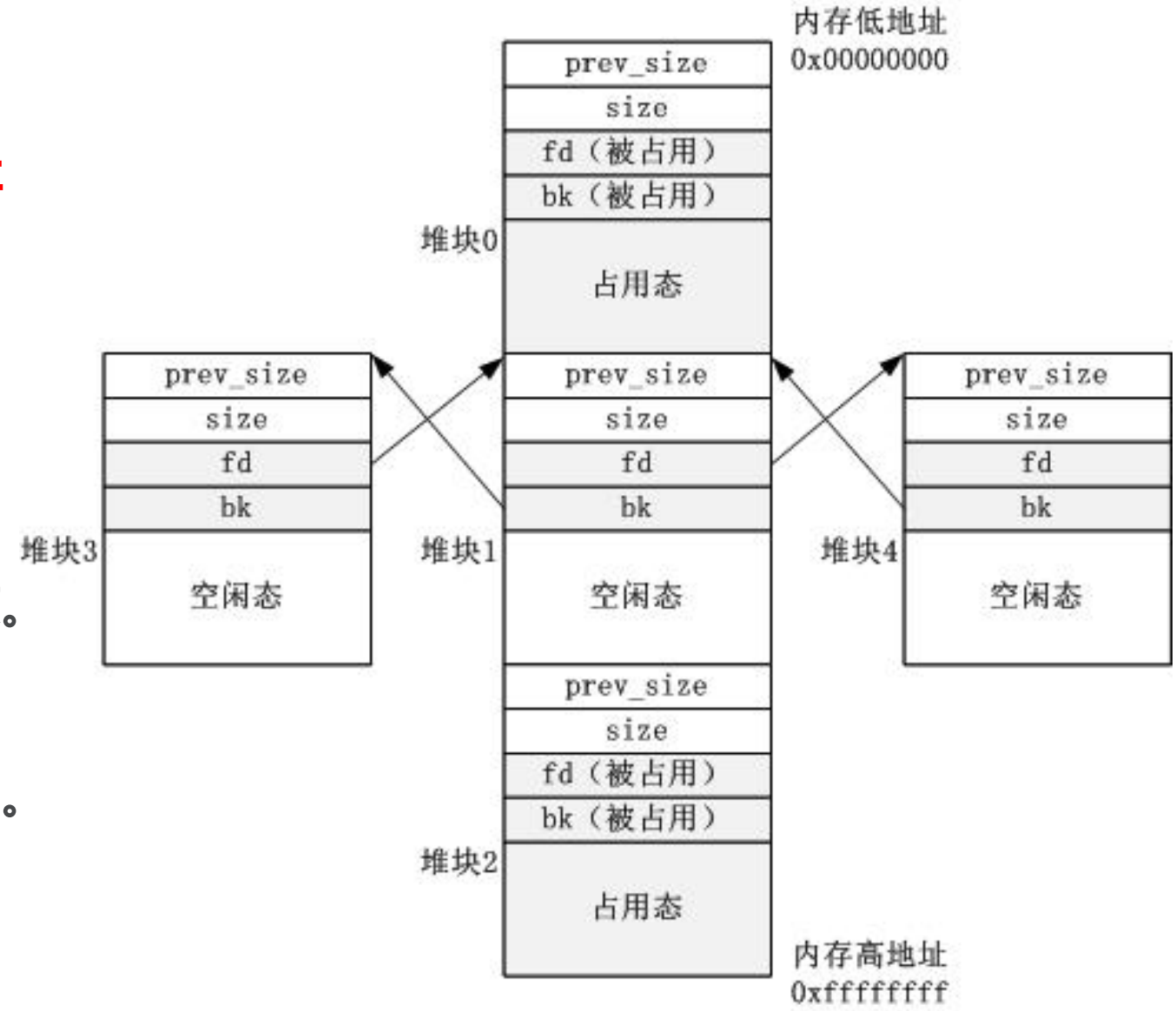
- 堆顶块（Top chunk），是一个特殊的空闲堆块，处于堆内存的**最顶部**。
  - 最初始，整个堆区就是一个堆顶块。
  - 然后，根据程序的申请，逐渐从堆顶块中割取更小的堆块，分配给程序使用。
- 堆管理器使用break指针来管理堆顶块，**break指针始终指向堆顶块的头部**。

- **堆表**，即**空闲**堆块列表，用于索引所有的**空闲堆块**。堆表中的索引（fd和bk）指向空闲堆块的真正的头部（即prev_size的地址）。
- **占用态的堆块由使用它的程序索引**，指向堆块数据区的头部（即fd的地址）。



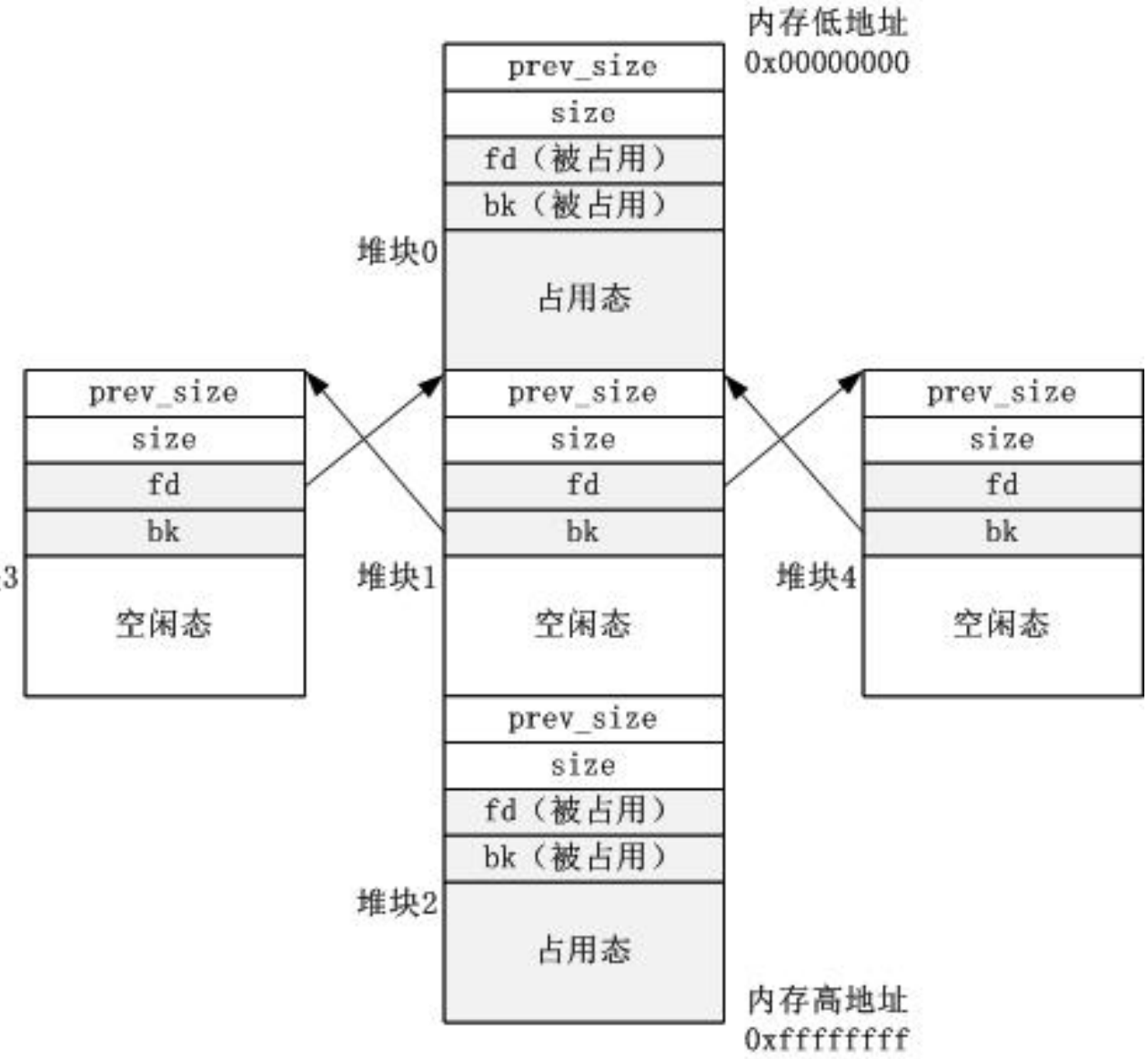
堆块的两种相邻关系

- 1）堆块在**内存地址空间的相邻关系**，如堆块0、1、2。
- **prev_size和size**用于表示内存地址空间相邻堆块的信息。
- 堆块1的prev_size用于表示堆块0的长度。
- 堆块1的size的最低位用于表示堆块0的状态，此时应该为1。



堆块的两种相邻关系

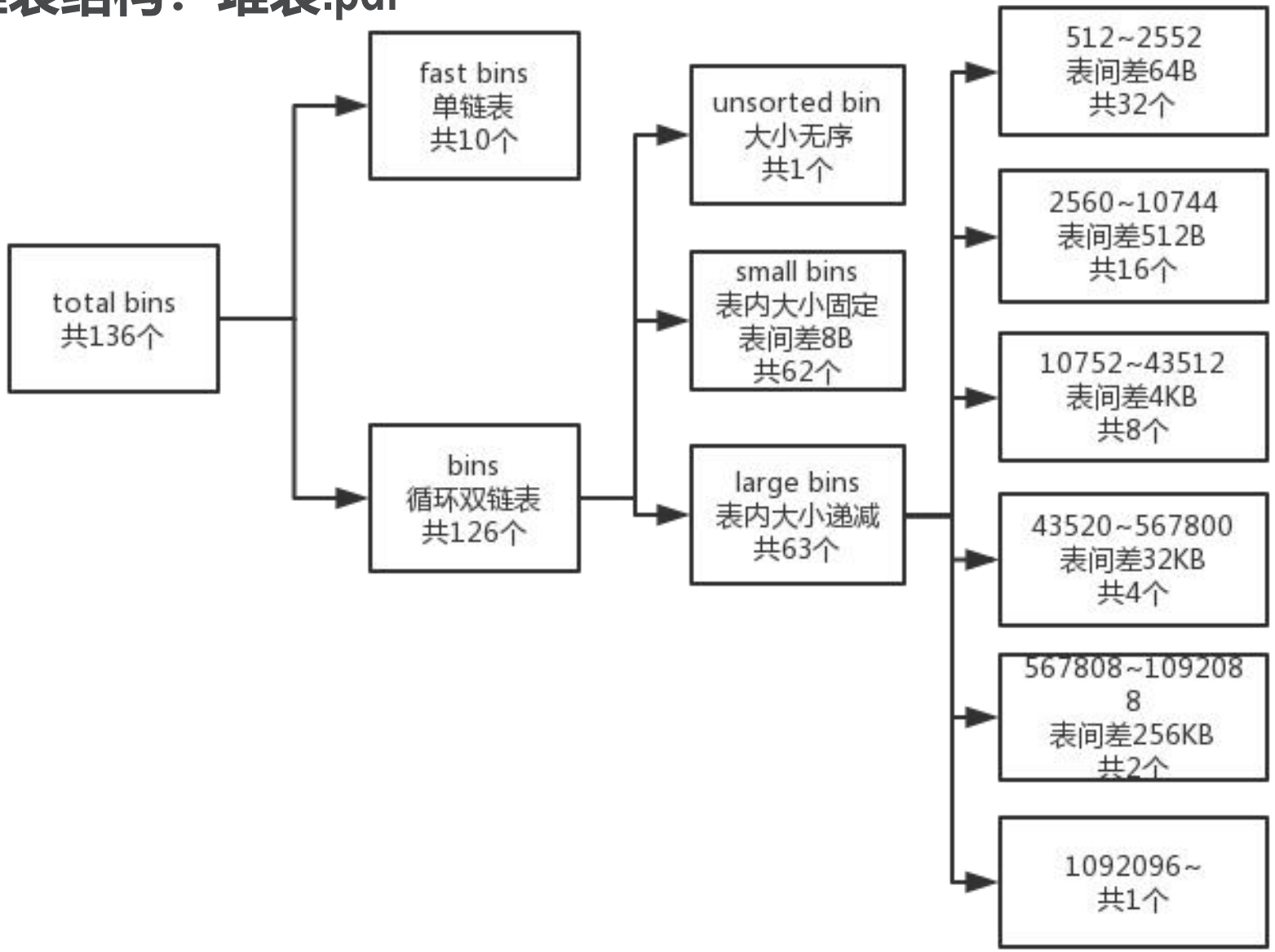
- 2）空闲堆块在**堆表中的相邻关系**，如堆块3、1、4。
- **fd和bk**用于表示堆表相邻空闲堆块的地址。
- 堆块1的fd指向堆块4的头部。
- 堆块1的bk指向堆块3的头部。



堆的基本知识

- 堆表有4种不同类型，一共分为136个箱子：
  - Fast bins（快表），一共有10个箱子
  - Small bins，一共有62个箱子
  - Large bins，一共有63个箱子
  - Unsorted bins（无序表），只有一个箱子
- bin：箱子或容器，就是一个**链表**，用于索引空闲堆块。不同的箱子用于索引不同大小的空闲堆块。

堆表结构：堆表.pdf


total bins
共136个
fast bins
单链表
共10个
bins
循环双链表
共126个
unsorted bin
大小无序
共1个
small bins
表内大小固定
表间差8B
共62个
large bins
表内大小递减
共63个
512~2552
表间差64B
共32个
2560~10744
表间差512B
共16个
10752~43512
表间差4KB
共8个
43520~567800
表间差32KB
共4个
567808~1092088
表间差256KB
共2个
1092096~
共1个

堆表类型

- Fast bins：定长箱子，单链表，最优先分配，精确匹配，不可分割，块大小在64B以下（32位），某些情况下可以合并相邻块后放入Unsorted bins。
- Small bins：定长箱子，双链表，前62个列，每一列bin中的chunk大小都相同，但不同列的bin中的chunk大小不同，相差8字节。
- Large bins：不定长箱子，双链表，后63个列，从512B开始，每列空闲块从大到小排序。
- Unsorted bins（无序表）：不定长箱子，双链表，堆区中的堆块被free后并不立即被清除出堆区，而是先链入无序表中，大小无序。要分配新的堆块且不满足快表时，会优先使用无序表中的堆块，依序找到不小于指定大小的堆块后分割使用。

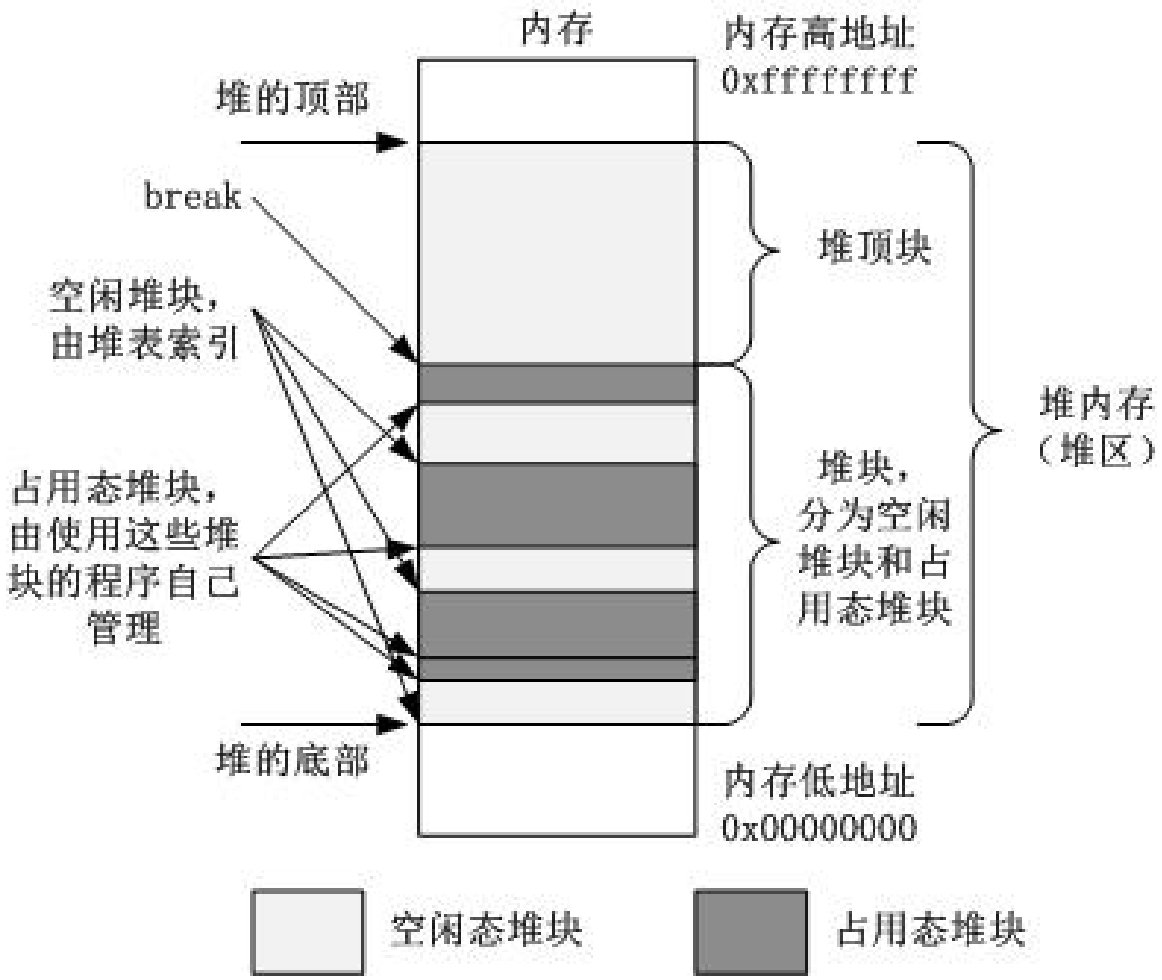
堆表的优先级

- Fast bins的构造和其他堆表不同，Fast bins中堆块的分配和释放速度更快。
- 空闲堆块**分配的优先级**：
  - Fast bins > Unsorted bins > Small bins > Large bins > Top chunk（堆顶块）
- 占用态堆块**释放的优先级**：
  - fast bins优先级最高。如果有大小合适的堆块（小于88字节），则直接被释放回fast bins。
  - 否则，占用态堆块被优先放回到Unsorted bins，然后再根据实际情况，逐步进行调整。

堆的基本知识

堆的几个基本概念

- 堆内存，也就是堆区，即整个堆内存空间。
- 堆块，堆内存管理的最小单元，分为空闲态和占用态。
- 堆顶块，处于堆顶的一个特殊空闲堆块，由break指针索引。
- 堆表，索引所有空闲堆块的链表（除了堆顶块）。

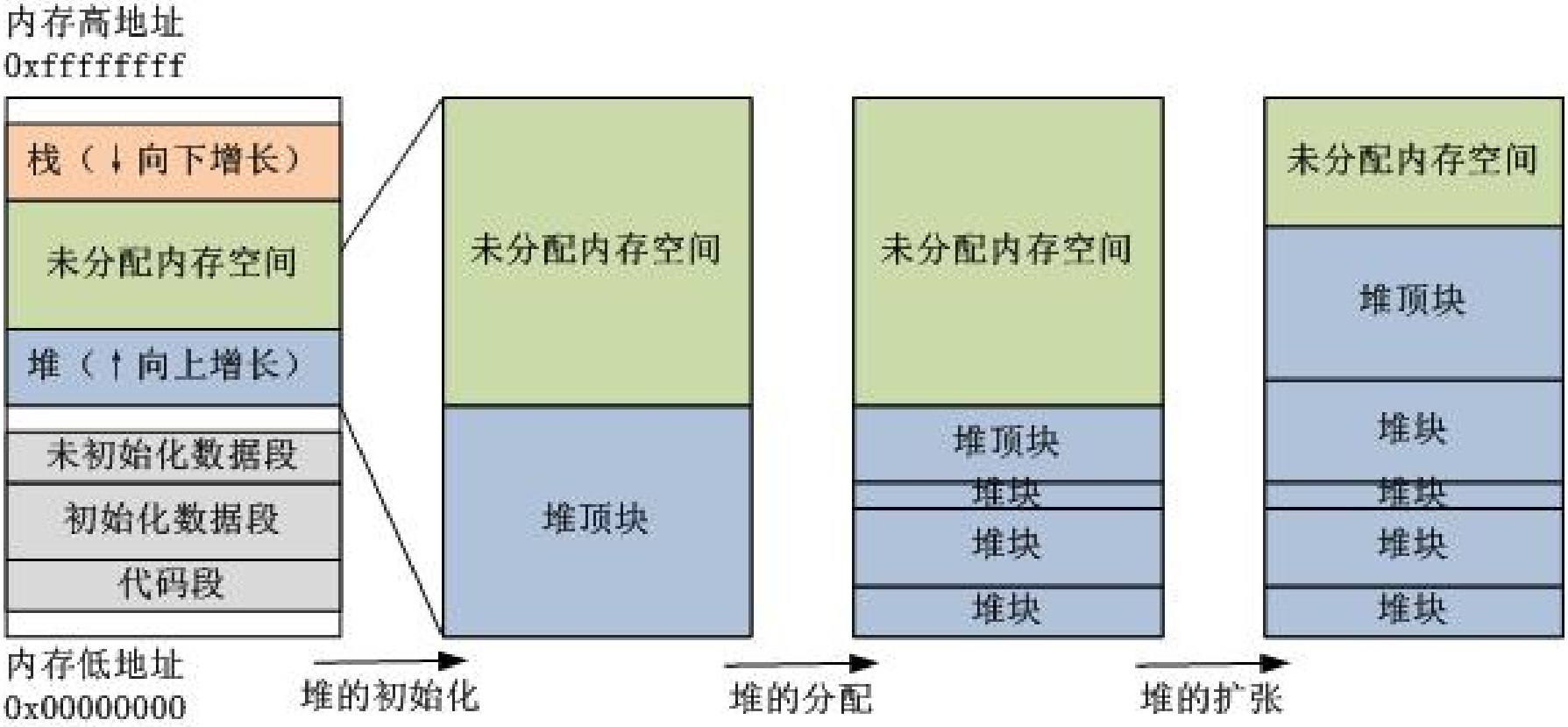


堆的管理

- 对堆内存的管理，实际上就是对**堆块的分配和释放过程**。
- 程序员通过malloc和free来操作堆内存中的堆块。
- **malloc**用于申请一个空闲的堆块。
- **free**用于释放当前被使用的堆块。

堆块分配的基本过程

- 堆内存，最初始就是一个堆顶块。
- 随着堆块的分配和释放，堆内存变成了一个堆顶块和多个堆块。



堆的初始化

- 在调用第一个malloc之前，程序进程中是没有堆内存的。
- 当调用第一个malloc时，系统才会给程序进程分配堆内存。
- 初始分配的堆内存（堆顶块）**远大于**申请的内存，这样后续的内存申请可以通过直接切割剩余堆内存（堆顶块）来实现。
- 当堆内存的大小不足或者有过多空闲时，操作系统会自动调整堆内存大小。

堆块的分配

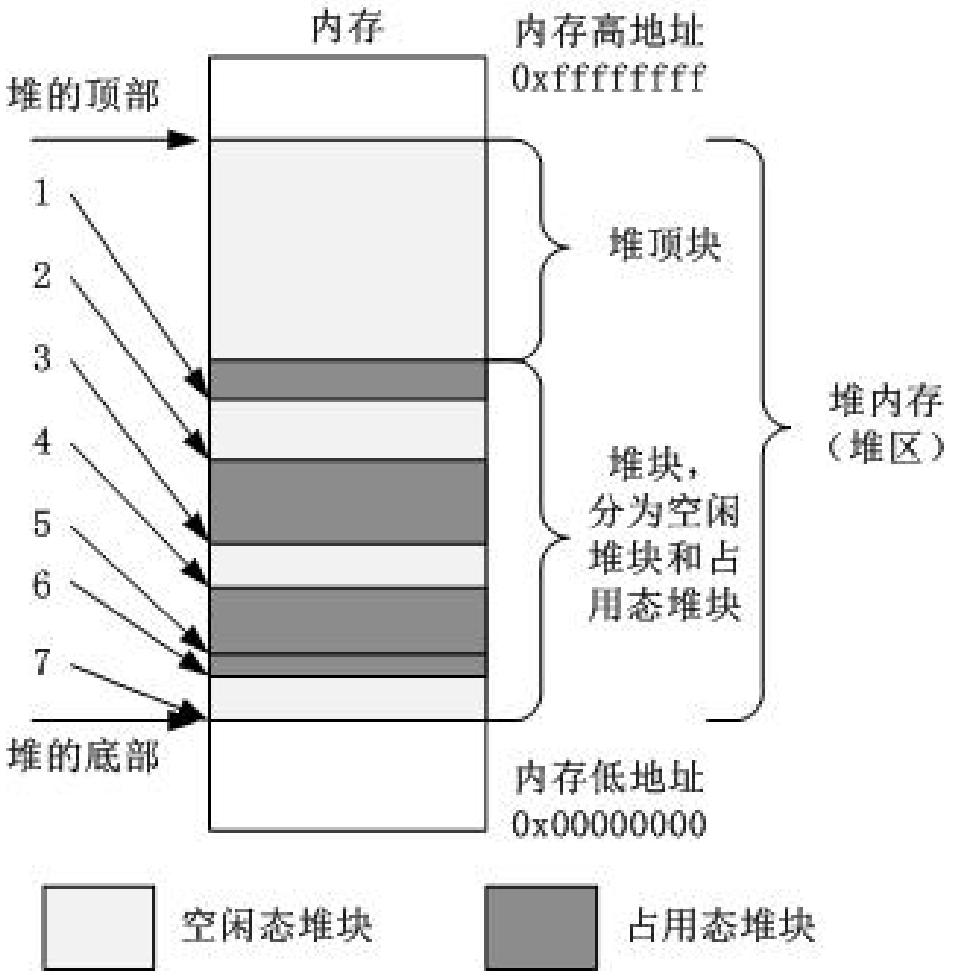
- 程序使用malloc申请某个大小的内存区域。
- 堆管理器在**堆表**中搜索对应大小的空闲堆块。
  - 如果能在堆表中找到，则从堆表中取出该空闲堆块，并将其修改为占用态。
  - 如果不能找到合适大小的，则从堆表中找一个略大的空闲堆块，切割成合适大小。
  - 如果还是不能找到，则直接从堆顶块中切割出一个对应大小的堆块。
- 最后，堆管理器返回一个指向该堆块**数据区头部**的指针。

堆块的释放

- 程序使用free释放指针p指向的堆块。
- 堆管理器将被释放的堆块改为空闲态。
  - 如果该堆块与空闲堆块相邻，则将该空闲堆块依次与相邻的空闲堆块进行**空闲堆块合并**。
  - 如果该空闲堆块和堆顶块相邻，则将该空闲堆块和堆顶块合并。
- 如果没有和堆顶块合并，则堆管理器将空闲堆块放回到堆表中。
- 注意：**程序自身需要将p指针置空**，以防止因为悬空指针而造成堆漏洞。

相邻空闲堆块合并

- 这里的相邻堆块是指在堆内存空间中地址相邻的空闲堆块。
- **优先**与前一个空闲堆块（**低地址方向**）进行合并。
- 堆顶块处于最高地址，所以最后和堆顶块合并。
  - 例如，当堆块1被释放时，堆块1先和空闲堆块2合并，然后和堆顶块合并。



空闲堆块合并过程

- 当堆块Q被释放，需要检查与堆块Q相邻堆块P1和P2的状态。
  - 检查Q的前一个堆块P1是否为空闲态：即Q的size字段第0位的值是否为0。
  - 如果P1为空闲堆块，则通过拆链（unlink）过程将P1从堆表中取下，将P1和Q合并成新的空闲堆块。
  - 同样，检查Q的后一个堆块P2 是否为空闲态（即Q的后后个堆块P3的size字段第0位是否为0）。如果P2也是空闲堆块，则再次进行空闲堆块合并。

拆链函数unlink

- 空闲堆块合并时，需要将与Q相邻的空闲堆块P从堆表中取出，即堆块P的unlink过程。
- 堆表是一个**双向链表**，所以unlink函数具有以下操作，即双向链表的拆链操作。

```
#define unlink(P) {
    FD = P->fd;
    BK = P->bk;
    …
   FD->bk = BK; //P->fd->bk = P->bk
    BK->fd = FD; //P->bk->fd = P->fd
    … }
```

空闲堆块合并的作用

- 确保堆内存中不会存在两个在地址空间相邻的**连续空闲堆块**。
- 也就是说，从内存地址空间上看，一个空闲堆块的上一个堆块和下一个堆块肯定都是占用态堆块。
- 因为，一旦有两个相邻的空闲堆块，就会触发空闲堆块合并操作，使得这两个空闲堆块合并成为一个空闲堆块。

堆块操作示例程序

```
char *p0 = malloc(248);
char * p1 = malloc(504);
char * p2 = malloc(760);
char * p3 = malloc(1016);
free(p0);
free(p2);
free(p1);  //空闲堆块合并，p1和p0合并，然后和p2合并
free(p3);  //空闲堆块合并，最后和堆顶块合并
```

堆块操作示例

- 见“堆分配与堆块合并.pdf”
- 示例中地址增长方向：从上向下
- 示例中堆区起始地址：0x0804c000
- 示例中只使用了一个堆表，即Unsorted bins（无序表），起始地址：0xb7fc4470
- 堆块释放是将被释放的空闲堆块插入到unsorted bins的头部。所以，后释放的空闲堆块处于先释放的空闲堆块的前面。

小结

- 详细介绍了堆的相关知识，以及堆的正常管理过程（即堆块的分配和释放过程，空闲堆块合并过程等）。
- 下一节课将详细介绍堆漏洞的原理和具体实现过程。
  - 堆溢出漏洞（heap overflow）
  - 重复释放漏洞（double free）
  - 释放后使用漏洞（use after free）

作业

- 实验：**栈溢出漏洞的构建和利用**
  - 自行构建一个包含**栈溢出漏洞**的程序
  - 利用调试工具，详细观察漏洞程序的运行过程，重点观察**函数调用及返回过程**和**栈溢出过程**中栈的变化
  - 利用以上观察结果，构建一个栈溢出漏洞的**攻击实例**，能够控制函数返回跳转到任意指定位置
  - 注意：当前现实系统中有不少针对栈溢出的**防御措施**，在构建攻击时请自行考虑如何绕过这些防御（实在绕不开，可**直接关闭**相关安全机制。。。）
- 报告：
  - 将栈溢出漏洞的构建及利用整个过程进行描述和分析，形成实验报告
  - 附上最终的漏洞代码及攻击代码，并在报告中对相关代码进行注释和分析
- 扩展：
  - 感兴趣的同学可以尝试对堆进行攻击和利用
  - 可以将相关调研过程和实验过程补充到实验报告中，可适当加分（不强制）


Q&A