Virtual Memory: Concepts & Virtual Memory: Systems

康子熙 赵廷昊 余文凯 许珈铭

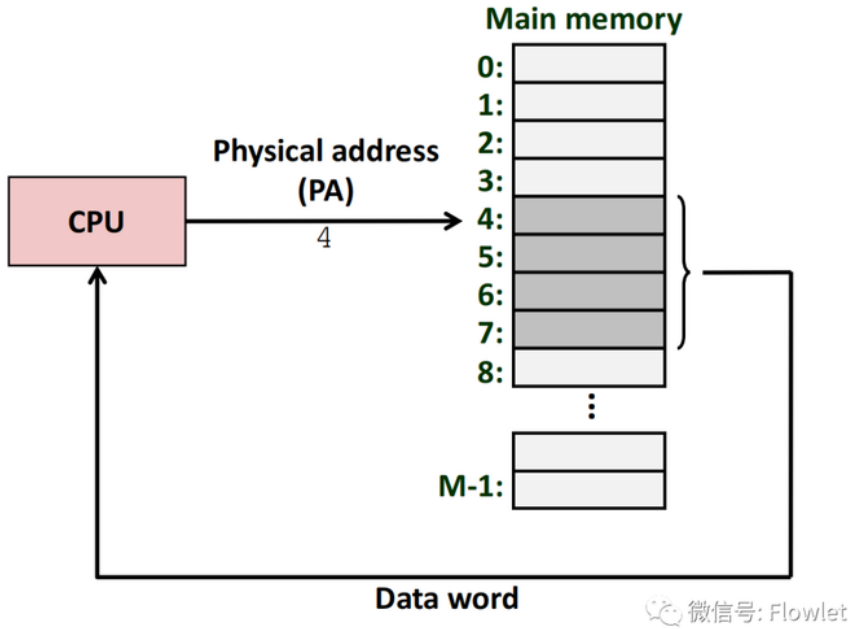
2023.12.3

Virtual Memory: Concepts (CS:APP Ch. 9.1-9.7)

赵廷昊

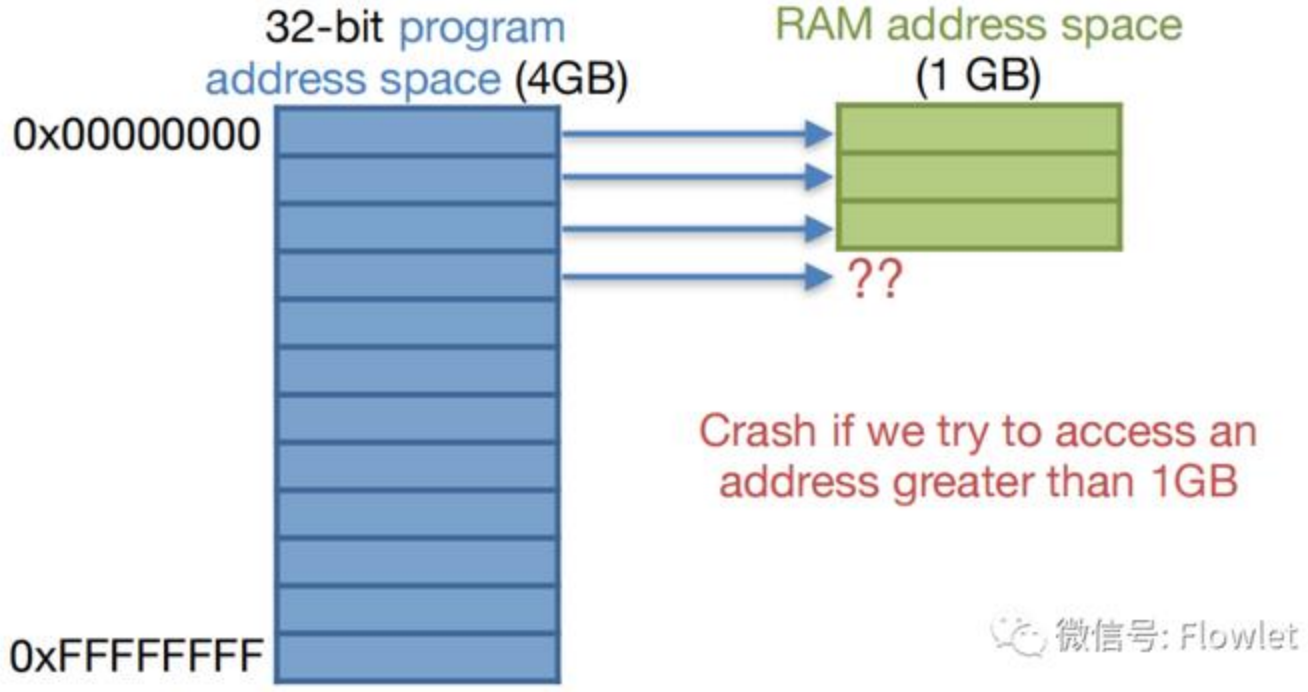
物理内存与物理寻址

- 计算机系统的主存（Primary Storage）被组织成一个由 M 个连续的字节（bytes）大小的单元组成的数组。每字节都有一个唯一的物理地址（Physical Address，PA）。



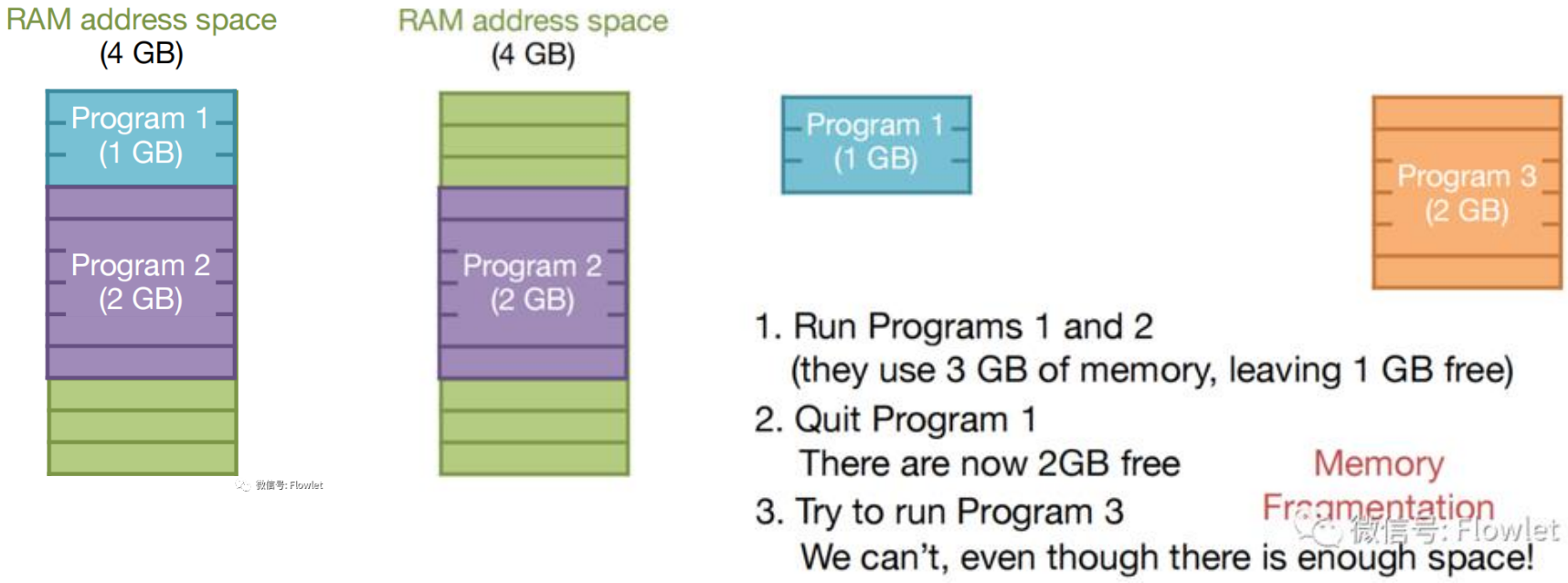
物理寻址的问题

- **物理内存不足**
- 当应用程序的访问地址空间（4GB）超过物理内存的实际地址空间（1GB DRAM）范围时，就会导致程序崩溃。



物理寻址的问题

- **物理内存不足**
- **内存碎片化**
- 程序 1 退出，剩余 2 GB RAM，程序 3 依然无法运行。因为剩余的 2 GB RAM内存空间不是连续的。



物理寻址的问题

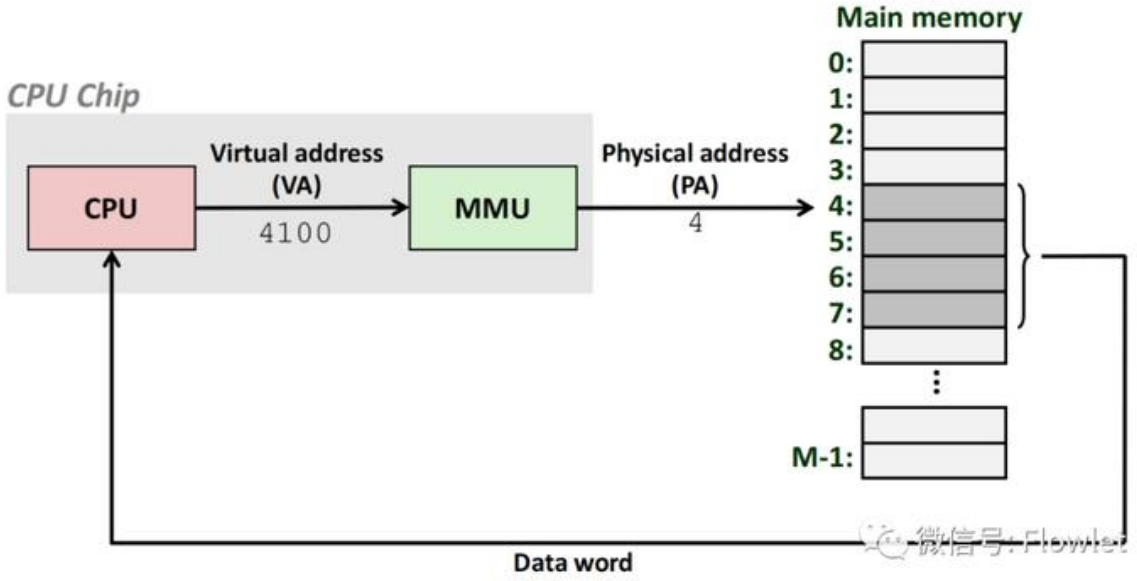
- **物理内存不足**
- **内存碎片化**
- **内存安全问题**
- 每个程序都可以访问 相同的物理内存地址空间，如果多个程序访问相同的物理地址，会导致数据污染或崩溃。

虚拟内存

- All problems in computer science can be solved by another level of indirection.

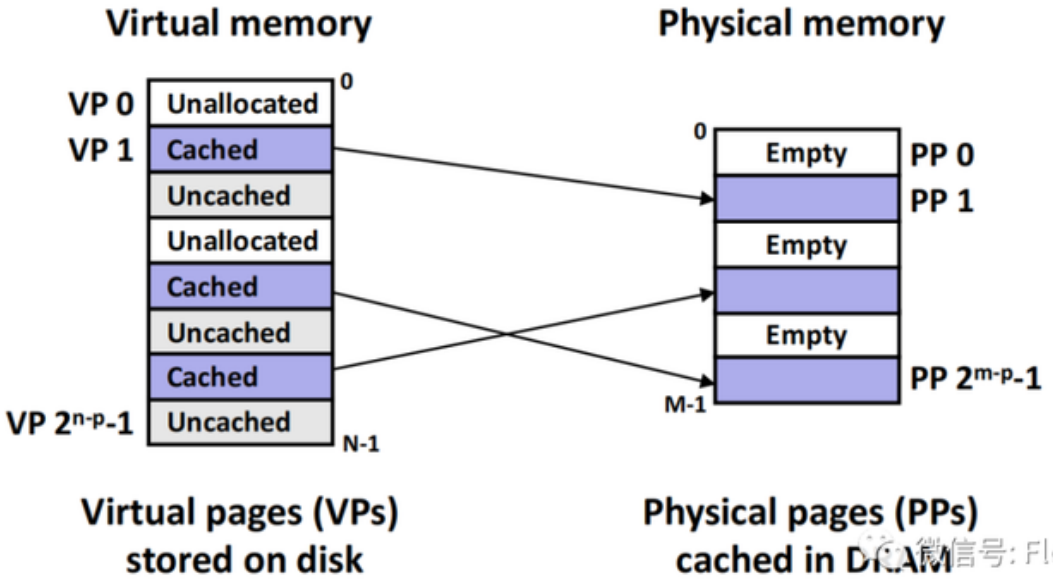
虚拟寻址

- 将一个虚拟地址转换为物理地址的任务叫做地址翻译
- CPU 芯片上叫做内存管理单元（memory Management Unit, **MMU**），利用操作系统管理的存放在主存中的查询表（**Page Table**）来动态的将虚拟地址翻译为物理地址。



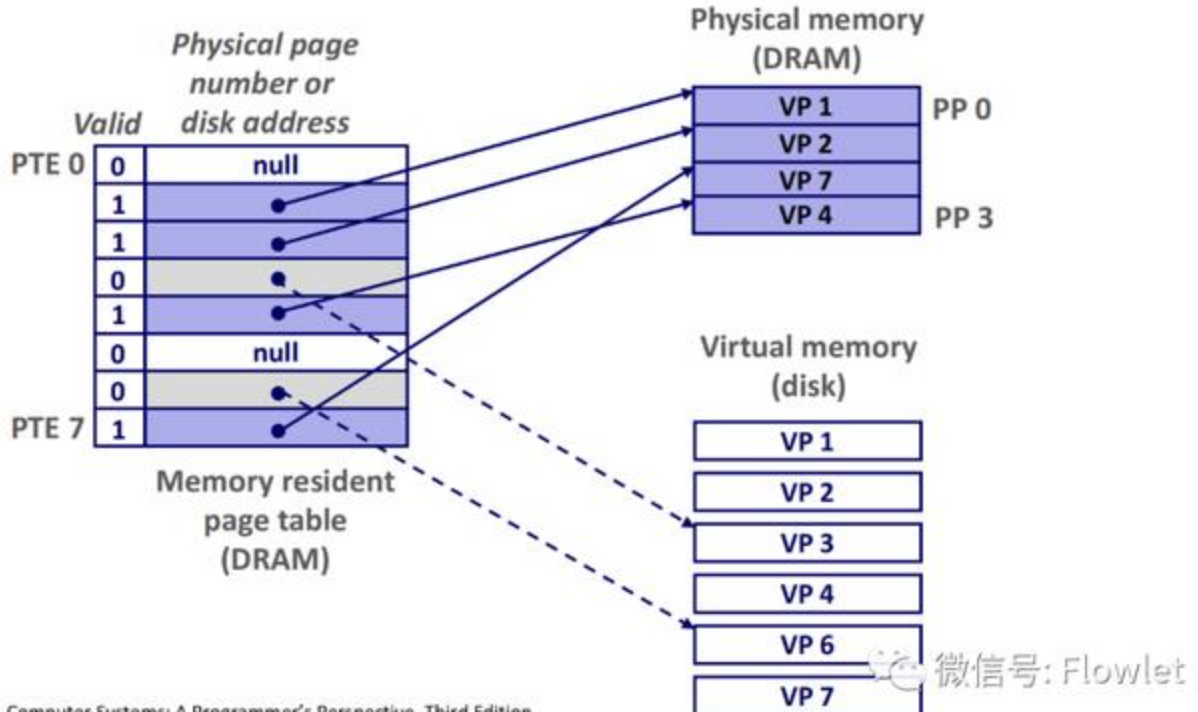
虚拟内存使用主存作为缓存

- 虚拟内存系统将虚拟内存分割为虚拟页（Virtual Page，VP），每个虚拟页的大小为 P=2^p 字节。类似的，物理内存被分割成物理页（Physical Page， PP），大小同样为 P 字节。
- 虚拟页面的集合总是分成 3 个不相交的子集：
- 未分配的： VP0、VP3
- 缓存的： VP1、VP4、VP6
- 未缓存的： VP2、VP5、VP7



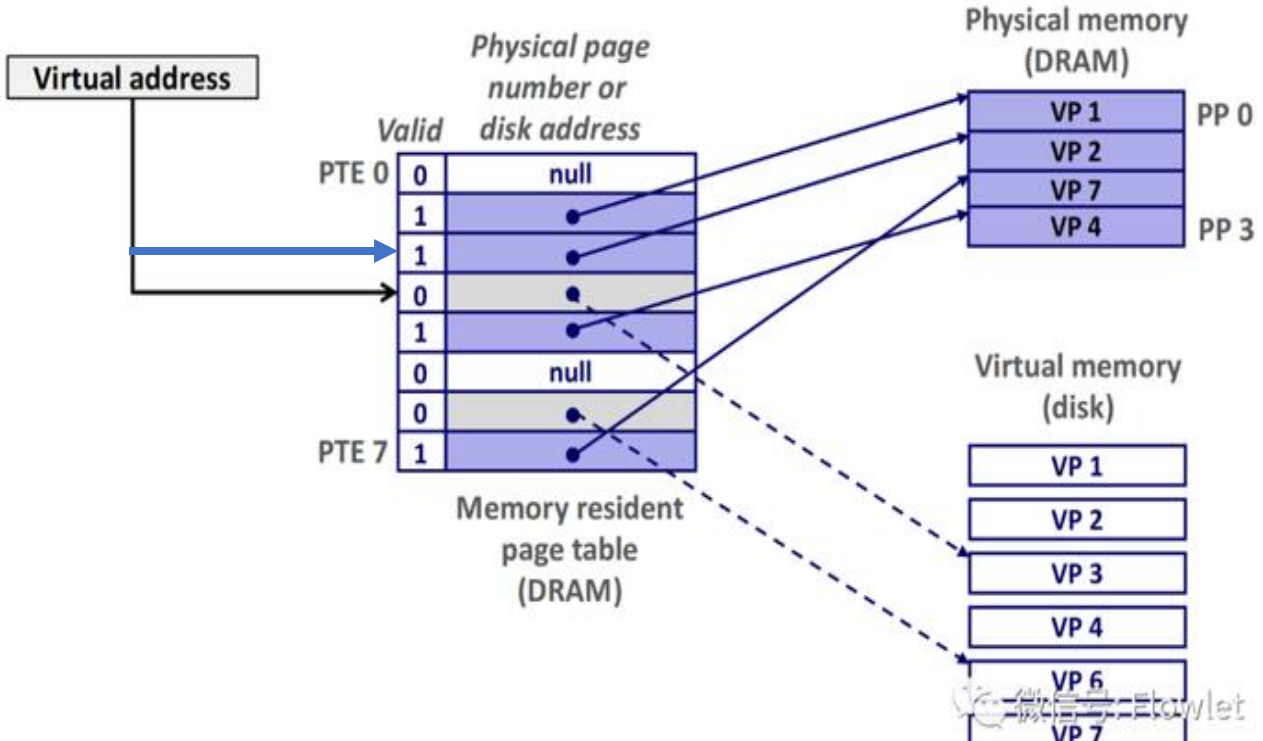
页表

- 虚拟页和物理页的映射关系
- PTE 由一个有效位和一个 n 位地址字段组成。
- MMU 通过页表的有效位来确定一个虚拟页是否缓存在 DRAM 中



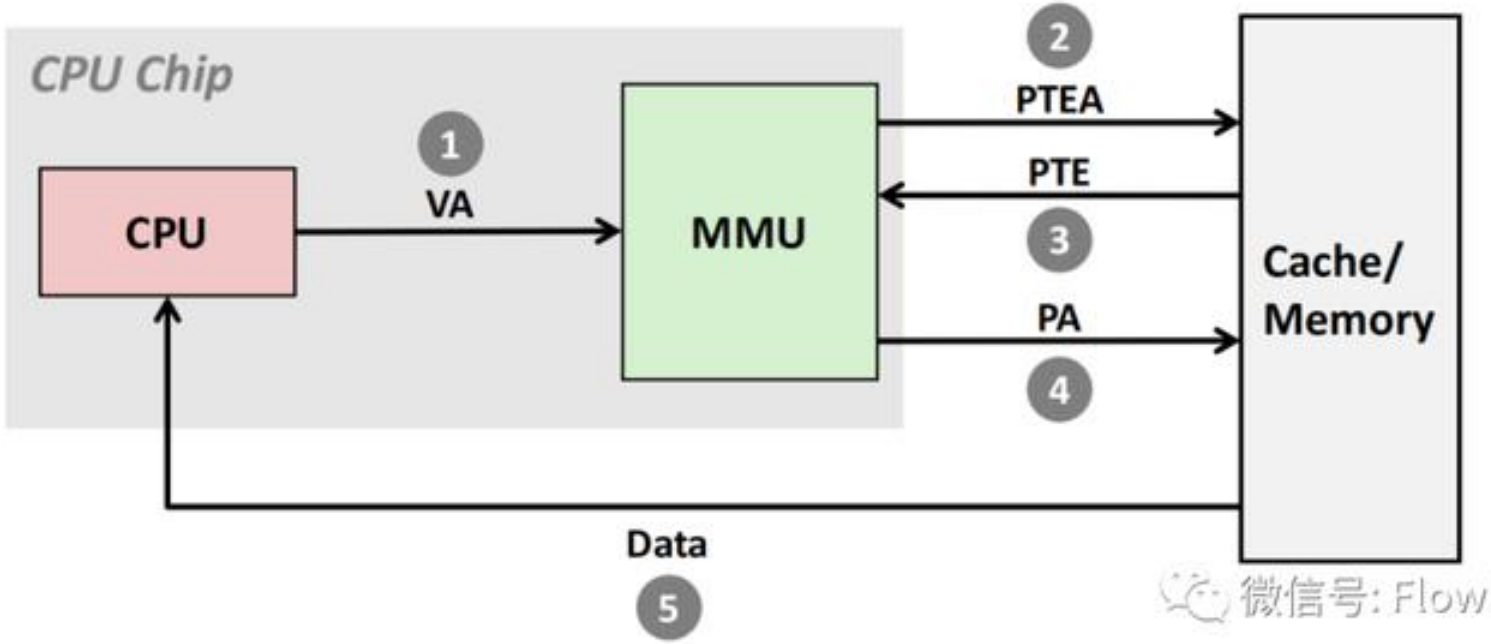
页命中/缺页

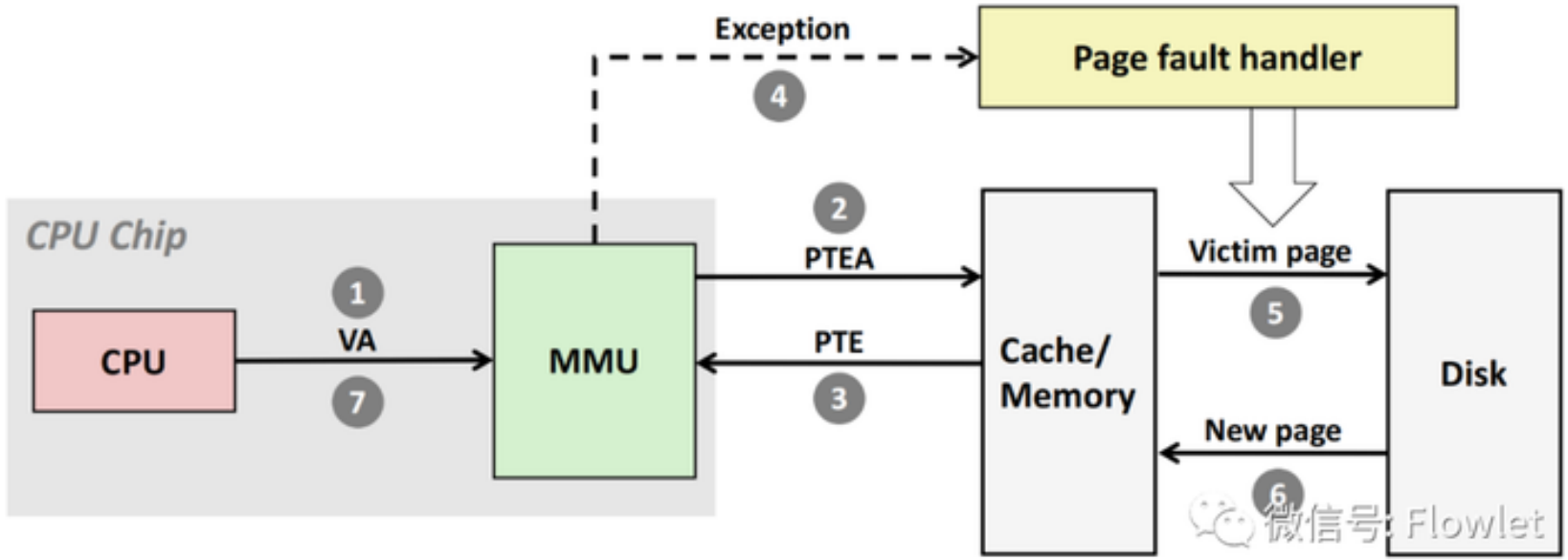
- 地址翻译硬件发现该地址在页表当中有效位为 0，即未被缓存在 DRAM 当中，称为缺页（Page Fault），触发一个**缺页异常**。
- 会选择一个牺牲页并进行替换



地址翻译

- 使用页表的地址翻译



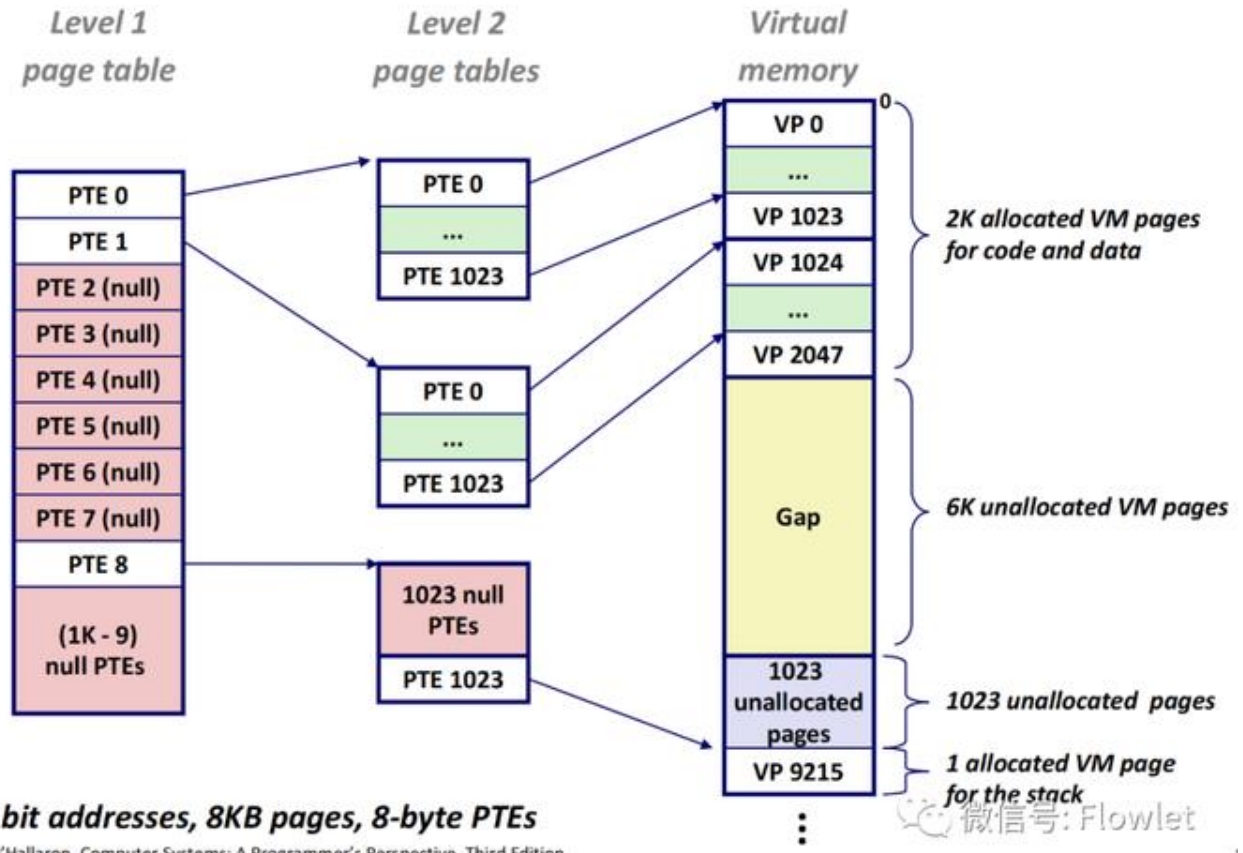


将 **Cache** 与虚拟内存整合在一起

- 将 PTE 存储到一个专门的 L1 Cache——TLB（Translation Lookaside Buffer，后备缓冲器）

多级页表

- 如果一个一级页表是空的，那么二级页表也不会存在。这是一个很大的节约，因为一个典型程序 4G 的虚拟地址空间的大部分都是未分配的。



Virtual Memory: Concepts (CS:APP Ch. 9.8-9.11)

康子熙

问题：怎么实现 memory mapping?

方法一：创建 mapping 的同时在物理内存中创建页

Regular file

直接将文件的相应内容以页为单位加载到物理内存，然后更新页表和 vm_area list

Anonymous

直接在物理内存中创建一个全 0 的页，更新页表和 vm_area list

方法二：需要时再创建物理页 ⟹ demand paging

Regular file

先更新 vm_area list, 进程访问相应页时引起 page fault, 然后由 OS handler **将文件内容中相应部分按页加载到物理内存**，再更新该进程的页表

Anonymous

先更新 vm_area list, 进程访问相应页时引起 page fault, 然后由 OS handler **在物理内存中创建一个全 0 页**，再更新该进程的页表

动态vs静态

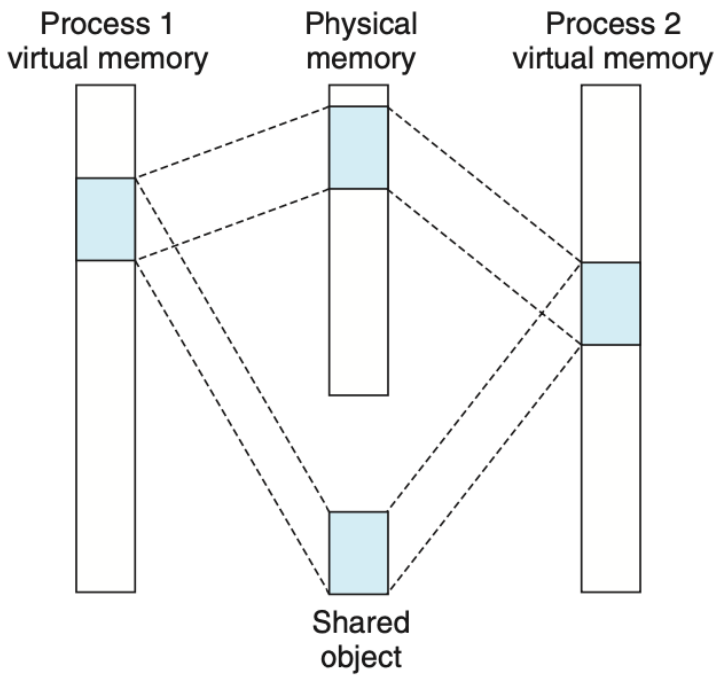
- **动态：**
  - **增加空间效率**
  - **增加了管理复杂性：**可能需要额外的同步机制

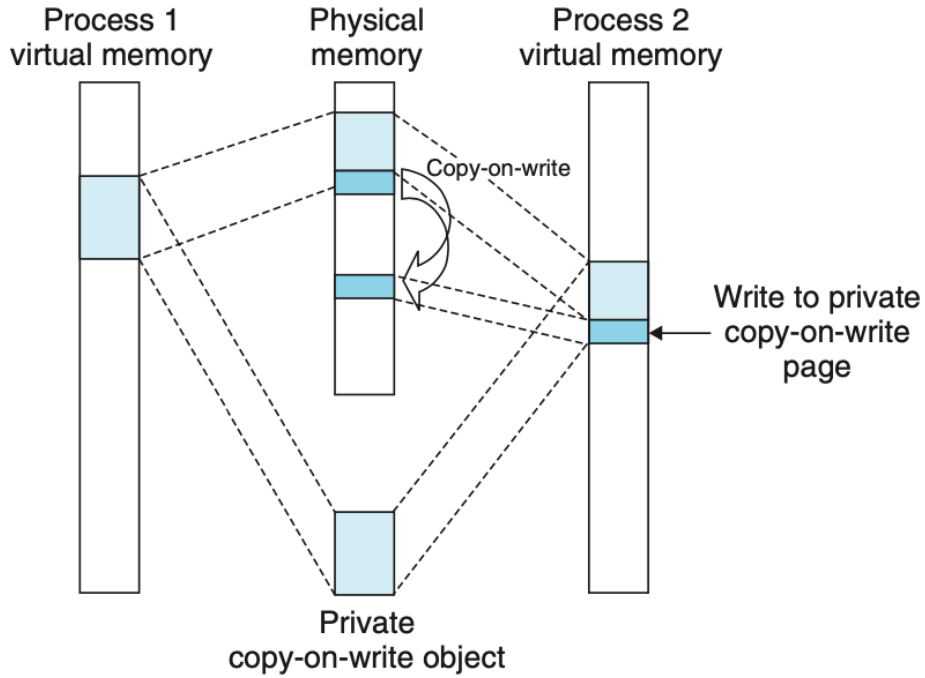
- **静态：**
  - **空间效率低**
  - **简化管理**

- cache的写回机制
- 虚拟内存的写入
- 单例对象实例化（懒汉模式/饿汉模式）

共享对象&私有对象

在读取时相同，再写入时私有对象具有**私有的写时复制**

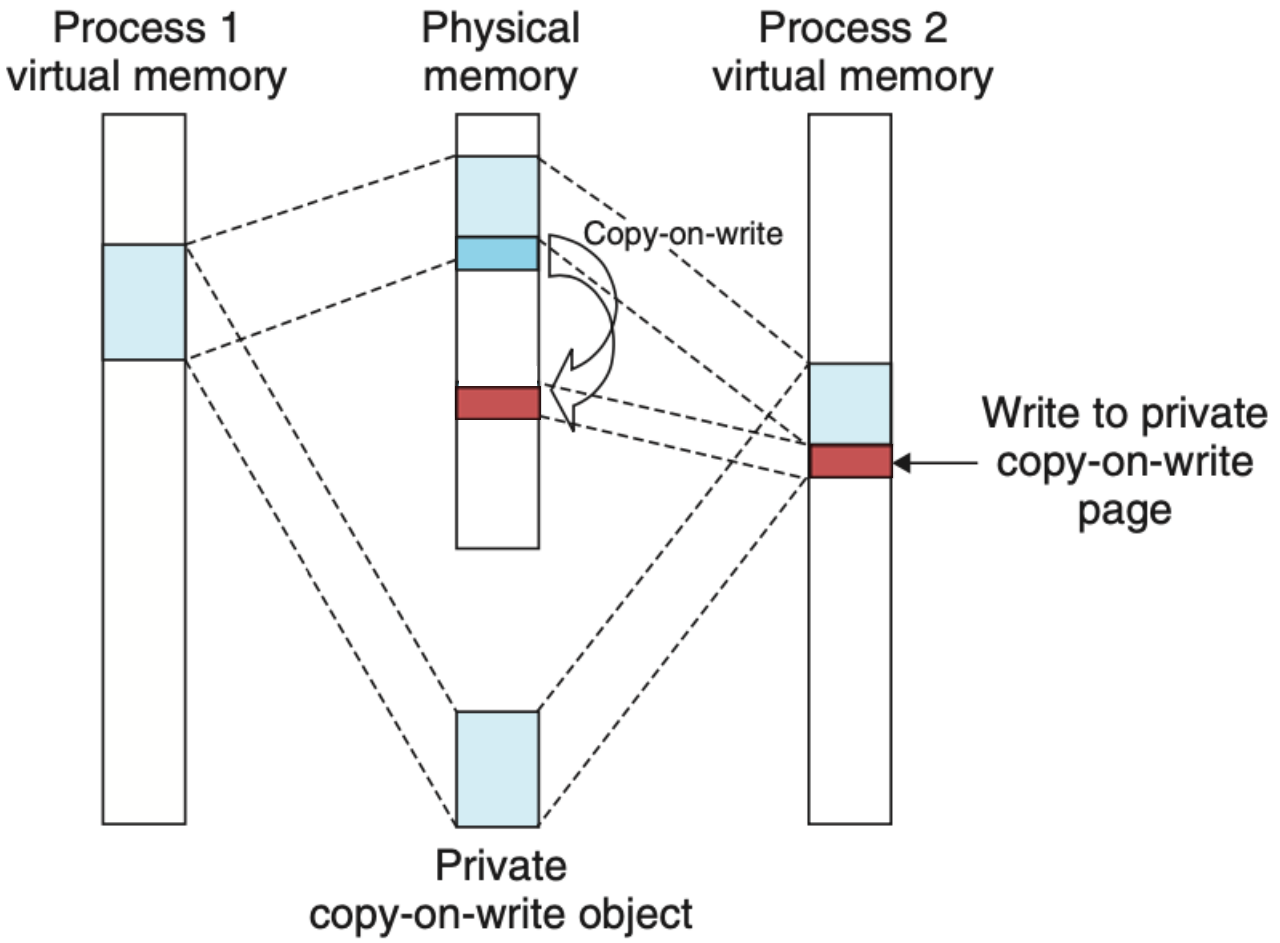




应用：fork函数

- 只创建数据结构，并标记为只读

-对应相同的物理内存



应用：execve函数

- **删除已存在的用户区域**
- **映射私有区域**
- **映射共享区域**
- **设置PC**

- 加载过程中不会创建新页，页表不变

先修改 vm_area list

- 进程访问对应页（如访问栈、从 .text 段取指令）时引起 page fault，再创建物理页并更新页表

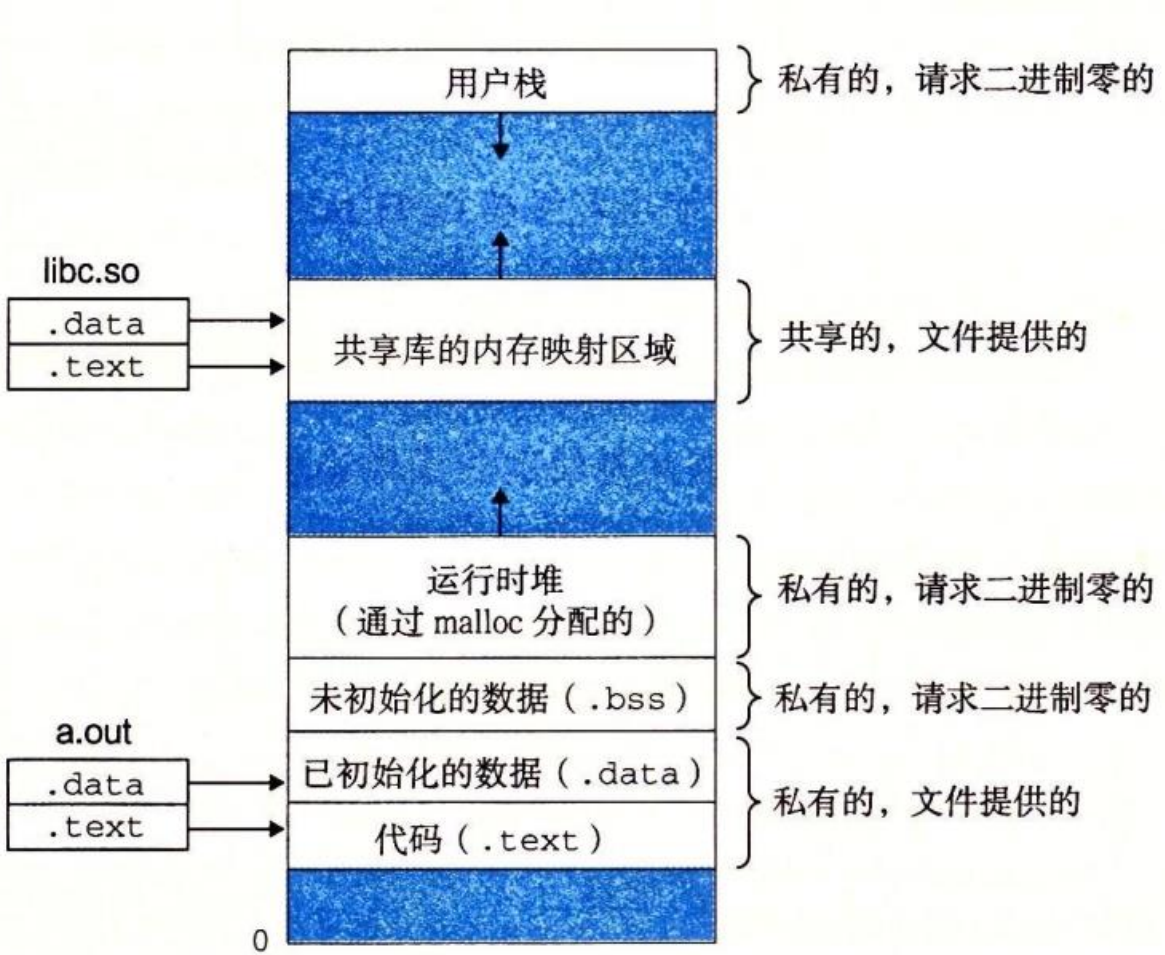


图 9-31　加载器是如何映射用户地址空间的区域的

应用：Swapping

```c
int main() {
    const size_t one_gb = 1024 * 1024 * 1024;
    size_t total_allocated = 0;
    while (1) {
        void* memory = malloc(one_gb); // 分配1GB内存
        if (memory == NULL) {
            break; // 如果内存分配失败，则跳出循环
        }
        memset(memory, 0, one_gb); // 使用0填充内存，确保内存被实际分配
        total_allocated += one_gb;
        printf("已分配 %zu GB内存\n", total_allocated / one_gb);
        sleep(1); // 暂停一段时间，以便观察
    }
```

应用：Swapping

Q: 用户进程在不断地创建物理页，但**物理内存大小有限**

- 在 disk 内开辟一块空间，暂时将不够存的页放在这里

- 希望在物理内存中放置被经常访问的页 ⟹最近最少使用（LRU）算法、时钟算法或其他更复杂的算法

- 如果交换后发生缺页中断，操作系统必须从磁盘的交换空间中检索这个页面，并将其重新加载到物理内存中

- 交换操作涉及磁盘 I/O，比直接从物理内存访问数据要慢得多。因此，频繁的交换会严重影响系统性能

实现：mmap函数

- flags描述被映射对象
共享or私有？ 匿名or普通？

- prot描述新虚拟内存区域的访问权限

```
#include <unistd.h>
#include <sys/mman.h>

void  *mmap(void *start, size_t length, int prot, int flags,
            int fd, off_t offset);
                Returns: pointer to mapped area if OK, MAP_FAILED (−1) on error
```

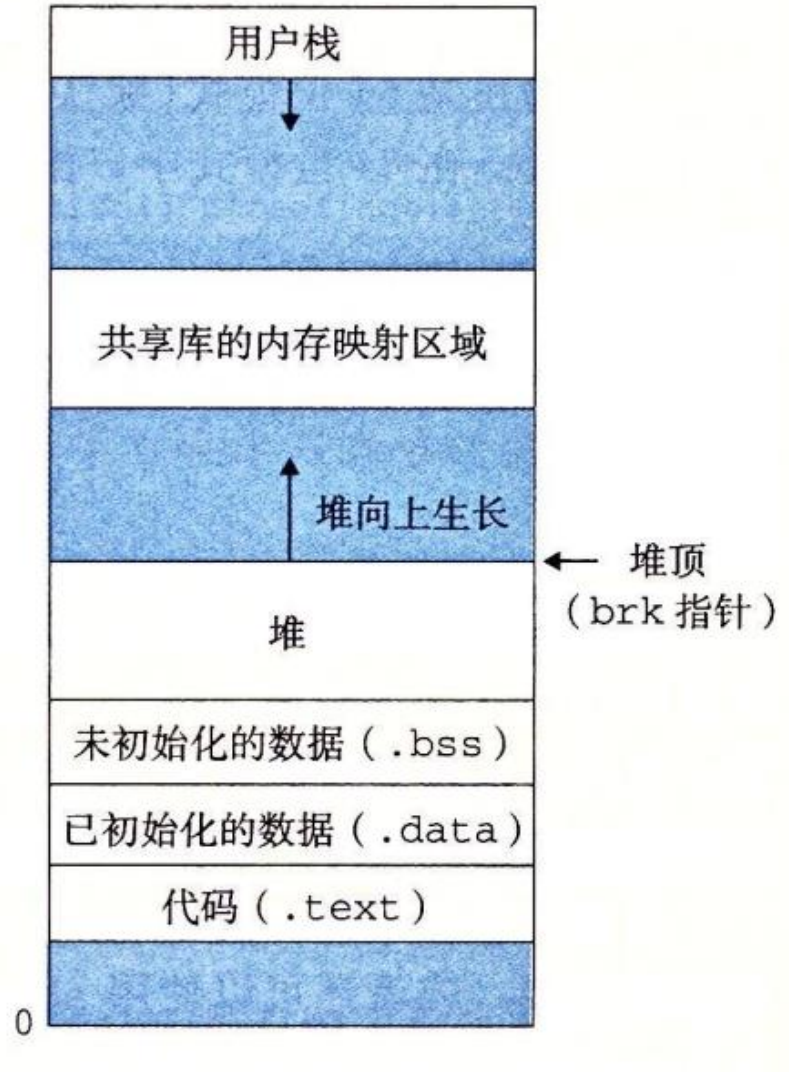
```
#include <unistd.h>
#include <sys/mman.h>

int munmap(void *start, size_t length);
                                        Returns: 0 if OK, −1 on error
```

更方便：动态内存分配器

- 变长数组
- 变长栈帧是可行的，C 语言支持变长栈帧
  - 不安全
  - 性能不够好

- 显式分配：C、C++
- 隐式分配： Lisp、Java

Malloc和new的区别：
malloc 是C语言标准库中的一个函数，它仅分配内存，不调用任何构造函数。malloc 的返回类型是 void*，需要强制类型转换为适当的类型。
new 是C++中的一个操作符，它不仅分配内存，还会调用对象的构造函数来初始化对象。new 返回的是分配类型的指针，无需类型转换



更方便：动态内存分配器

静态生命周期（Static Lifetime）：

在程序开始执行时分配，并在程序终止时释放。只被初始化一次，且其值在函数调用之间保持不变。

例：全局变量和静态变量

自动生命周期（Automatic Lifetime）：

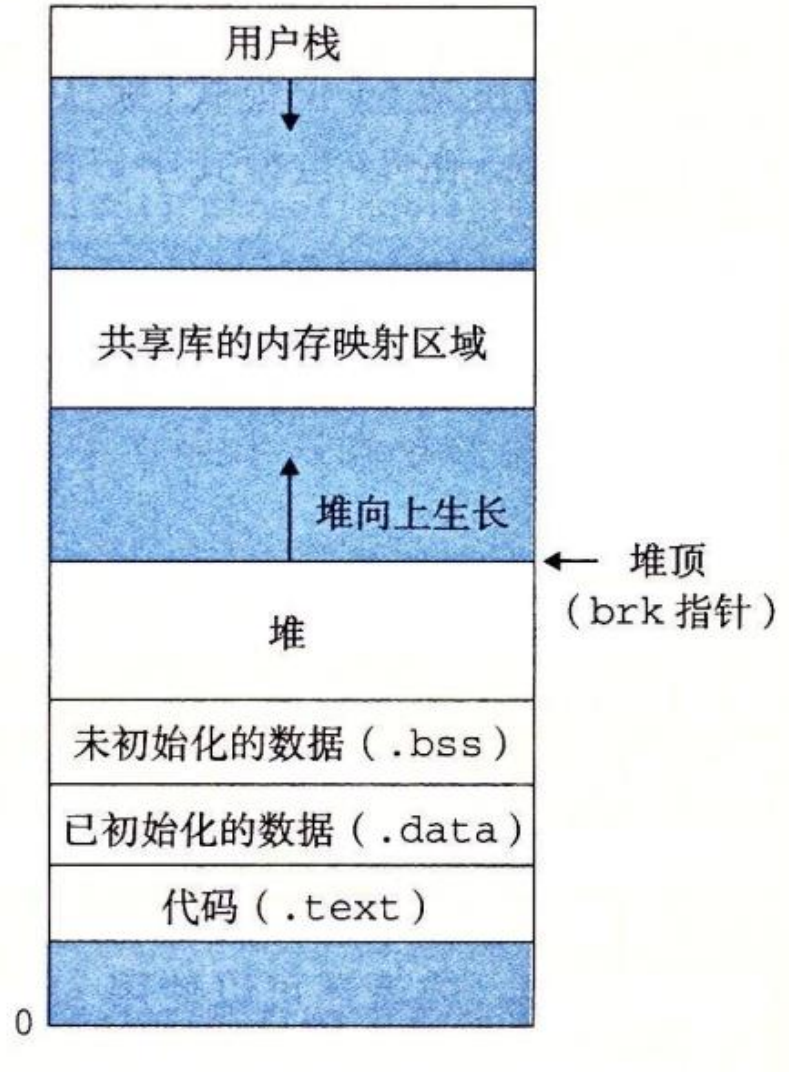
在函数内部声明，这类变量在其定义所在的块（block，通常是一个函数）开始执行时创建，在离开该块时销毁。它们的生命周期仅限于包含它们的块的执行期间。

例：局部变量

动态生命周期（Dynamic Lifetime）：

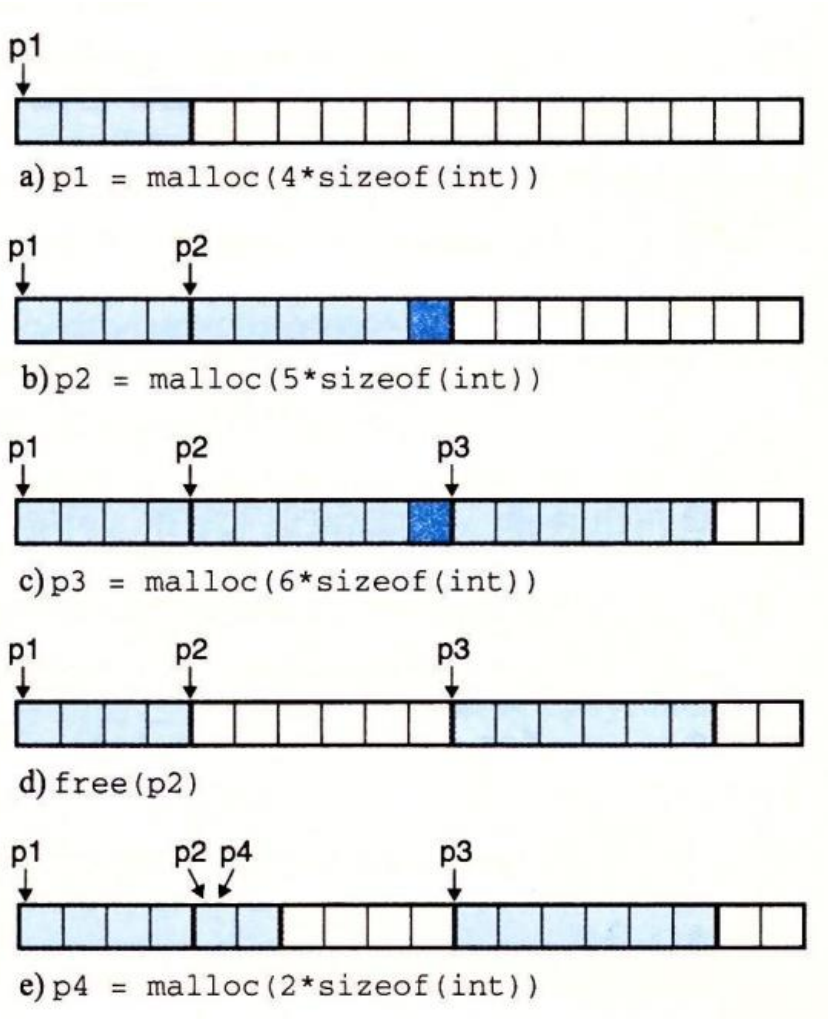
通过动态内存分配函数在堆（heap）上显式创建。不受其定义位置的限制，它们的生命周期从分配内存的那一刻开始，到使用对应的释放函数（如free）释放它们时结束

例：通过malloc分配的内存



分配器的目标

- **时间：最大化吞吐率**
  - 吞吐率：每个单位时间完成的请求数
- **空间：改善峰值利用率**
  - 内部碎片
    - 32位模式中，块地址总是8的倍数
    - 64位模式中，块地址总是16的倍数
  - 外部碎片



Q: 实现动态内存分配器

- **数据结构**
- 放置
- 分割
- 合并

隐式空闲链表（Implicit Free List）

工作原理:

- 在隐式空闲链表中，所有的内存块（无论是已分配的还是空闲的）都按顺序排列。
- 每个块都包含一个头部信息，通常包括块的大小和分配状态（例如，是否被分配）。
- 空闲块和已分配块在物理上相邻，没有明显的区分。

查找空闲块:

当分配内存时，系统遍历整个链表，通过检查块头部的信息来判断块是否空闲，并检查其大小是否满足分配需求。

显式空闲链表（Explicit Free List）

工作原理:

- 在显式空闲链表中，所有的空闲块通过指针直接相连，形成一个链表。
- 每个空闲块包含指向链表中前一个和后一个空闲块的指针。

查找空闲块:

当分配内存时，系统只需要遍历空闲链表，而不是整个内存块列表。

Q: 实现动态内存分配器

- **数据结构**
- 放置
- 分割
- 合并

隐式空闲链表（Implicit Free List）

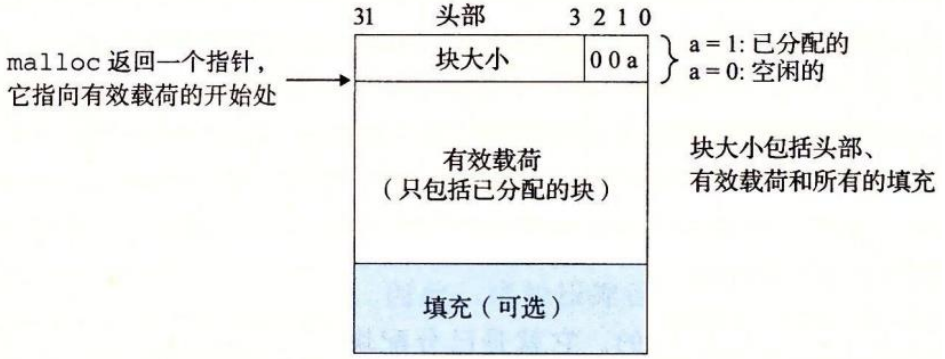
优缺点:

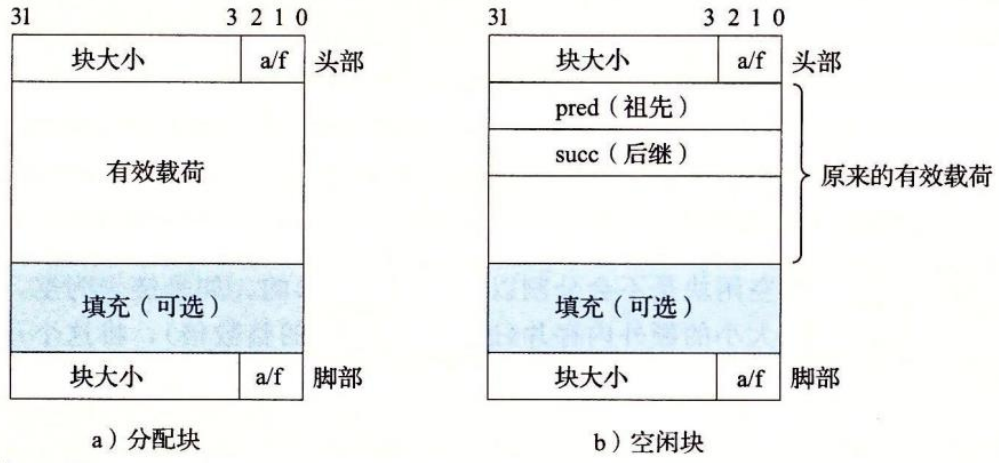
- 优点：实现简单，不需要额外的数据结构来维护空闲块。
- 缺点：可能导致较高的内存分配和释放时间，因为需要遍历整个列表来查找合适的空闲块。

显式空闲链表（Explicit Free List）

优缺点:

- 优点：提高了查找空闲块的效率，减少了内存分配的时间。
- 缺点：实现相对复杂，需要额外的空间来存储指向其他空闲块的指针。





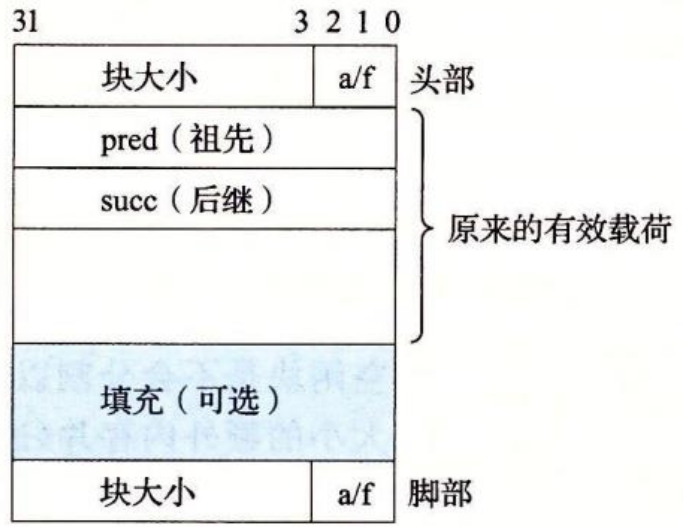
Q: 实现动态内存分配器

- 数据结构
- 放置
- 分割
- 合并

显式空闲链表的维护：

- 后进先出LIFO：利于释放
- 按照地址顺序排序：释放需要线性时间检索，但有更高的内存利用率

缺点：
- 空闲块必须足够大、最小块大小更大
- 提高了内部碎片的程度



b）空闲块

Q: 实现动态内存分配器

- 数据结构
- **放置**
- 分割
- 合并

首次适应（First Fit）
工作原理：从内存块列表的开始处开始搜索，选择第一个足够大的空闲块来满足内存请求。
优点：比较快速，特别是如果有大量小的内存分配请求。
缺点：随着时间推移，可能会在内存的前端区域产生较多小的空闲块，导致内存碎片化。

下一次适应（Next Fit）
工作原理：类似于首次适应，但它从上一次查找结束的地方开始，而不是每次都从头开始。
优点：
- 减少了搜索时间，因为不需要每次都从头开始搜索。
- 有助于分散内存分配，从而可能减少碎片化。

缺点： 内存利用率低

最佳适应（Best Fit）
工作原理：搜索整个内存块列表，选择能够最紧密匹配请求大小的最小空闲块。
优点：最大化每个空闲块的使用，理论上减少了内存碎片化。
缺点：
- 搜索整个列表可能导致较高的搜索时间。
- 随着时间推移，可能导致许多非常小的空闲块，难以再被有效利用。

Q: 实现动态内存分配器

- 数据结构
- 放置
- 分割
- 合并

- 内存利用率：最佳适配 > 首次适配 > 下一次适配
- 搜索时间（理想情况下）：下一次适配 > 首次适配 > 最佳适配

*** 另一种尝试：最差匹配 ***

工作原理

- 在最差匹配策略中，系统会遍历空闲内存块列表，寻找并选择**最大**的空闲块来满足内存请求。
- 如果选中的空闲块大于所需内存，剩余部分将继续作为空闲内存保留。

优点

减少小碎片：这种方法倾向于保留较小的空闲块，以便用于较小的内存请求。

Q: 实现动态内存分配器

- 数据结构
- 放置
- 分割
- 合并

简单分离存储（Simple Segregated Storage）

工作原理:

- 内存被分割成几个固定大小的池。每个池只包含特定大小的块。
- 当请求内存时，系统选择合适大小的池，并从中分配一个块。
- **空闲块不会分割以满足分配请求**

分离适配（Segregated Fit）

工作原理:

- 类似于简单分离存储
- **空闲块可选的分割，并将剩余的部分插入到适当的空闲链表中**

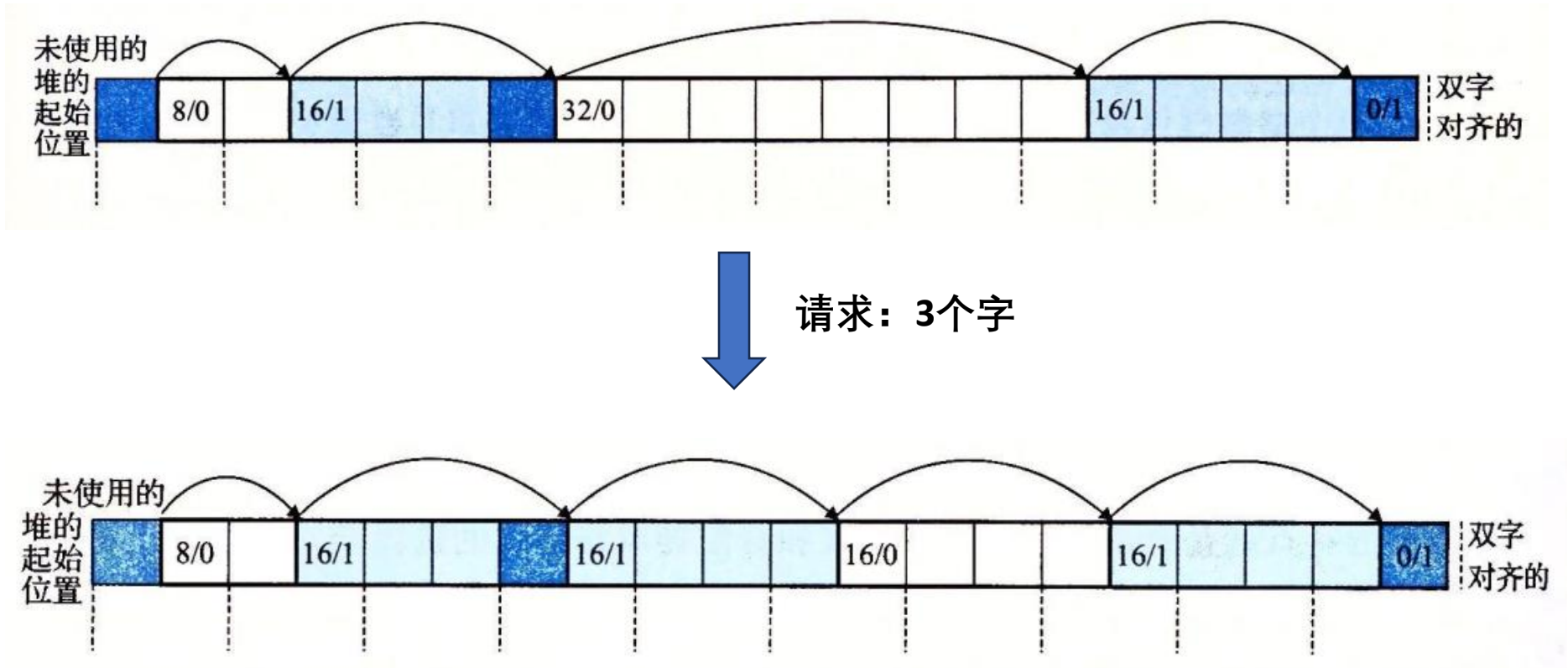
伙伴系统（Buddy System）——分离适配的特例

工作原理:

- 内存被分割成大小为2的幂的块。
- 当请求内存时，选择最小的满足需求的块。如果块太大，就将其分割成两个“伙伴”块。
- 一个块的地址和伙伴的地址只有一位不同

Q: 实现动态内存分配器

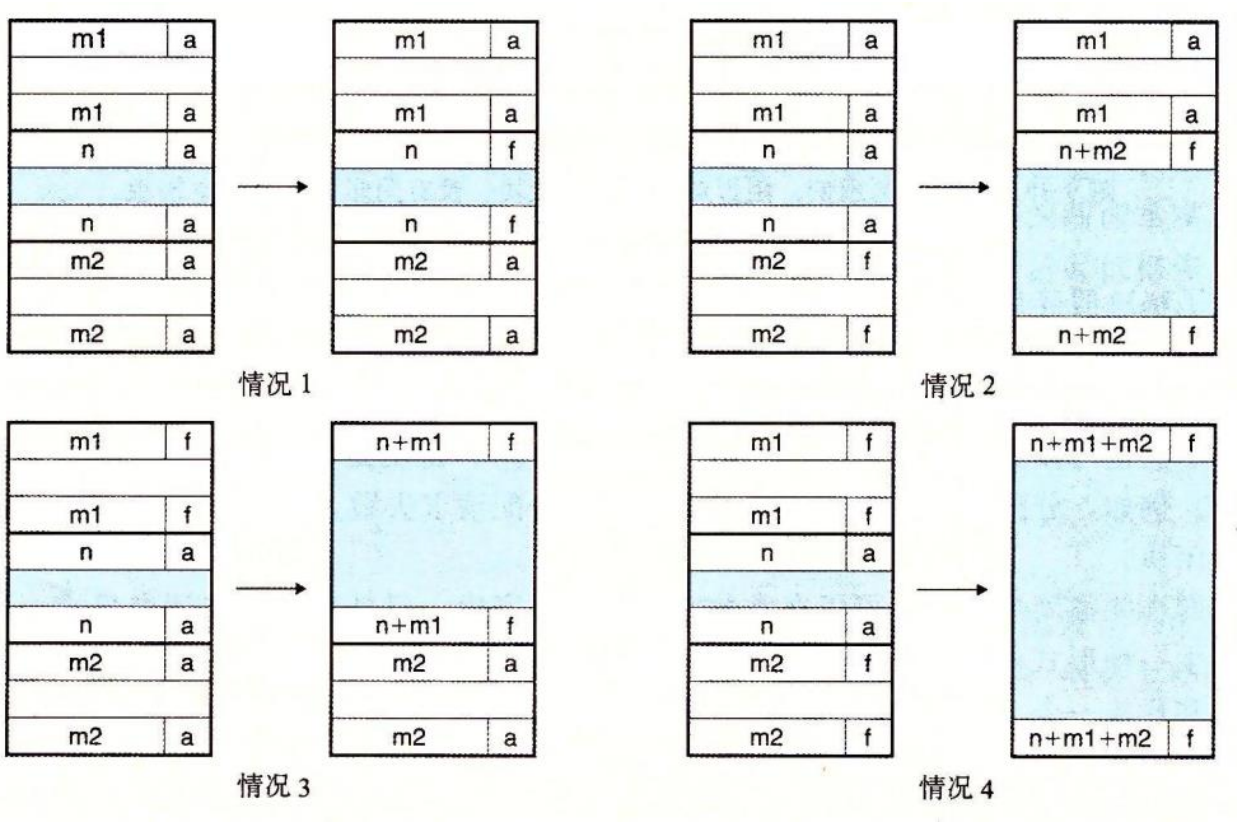
- 数据结构
- 放置
- **分割**
- 合并



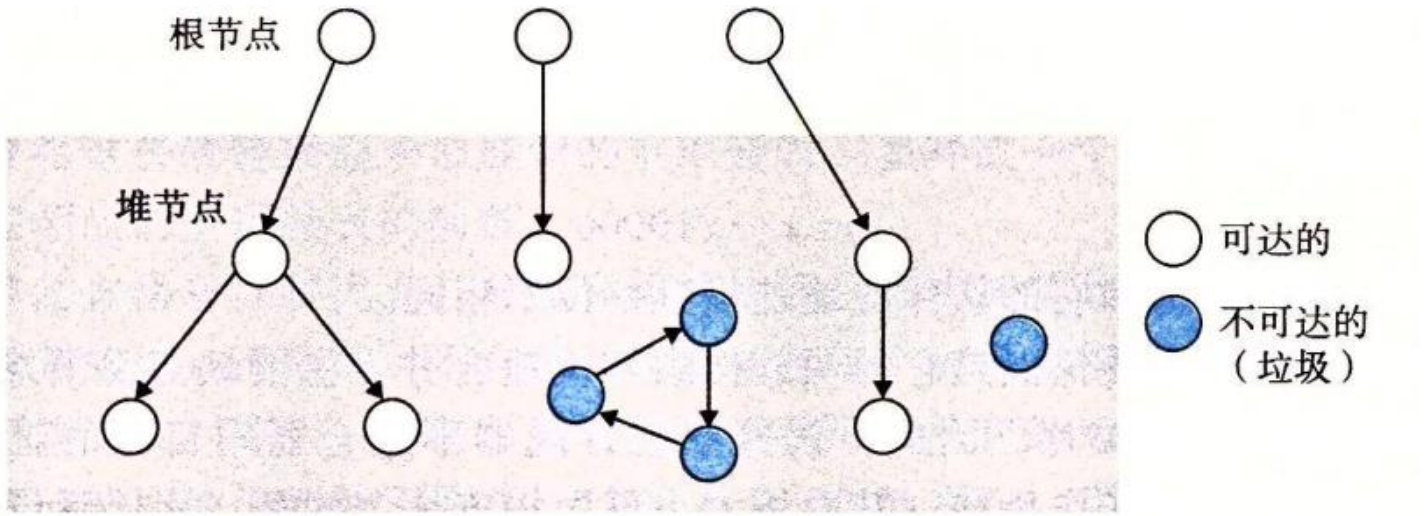
Q: 实现动态内存分配器

- 数据结构
- 放置
- 分割
- 合并

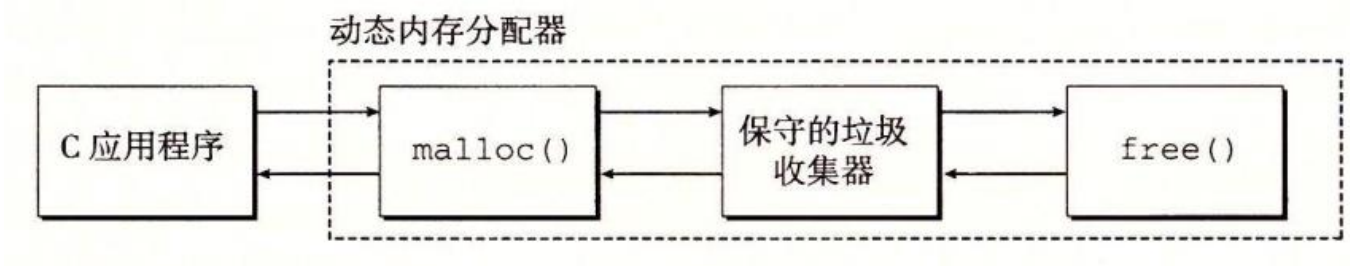
- 优化：已分配块不需要脚部



垃圾收集



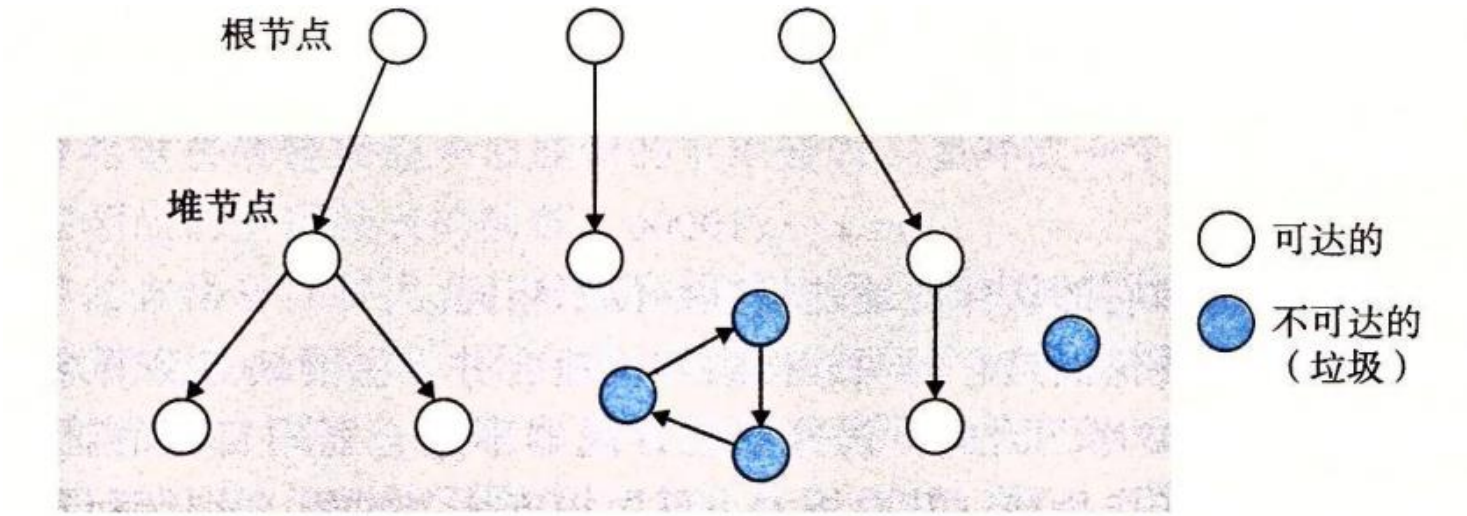
- 保守的垃圾收集器：不可维持可达图的精准表示



垃圾收集

- 根节点：指针（栈中、寄存器中、全局/静态存储区域中）

寄存器扫描：检查所有寄存器的内容，查找指向堆的指针。
栈扫描：从当前堆栈帧（函数调用）开始向下遍历调用栈，查找包含堆指针的局部变量。
全局区域扫描：遍历程序的全局/静态存储区域，查找包含堆指针的全局变量。



垃圾收集

Q: C/C++为什么没有内置的垃圾收集器?

- **性能和控制**：C++被设计为一种高效的语言，它提供了对底层资源的直接控制。显式内存管理使程序员能够精确控制对象的生命周期和内存使用，这对于性能敏感的应用是至关重要的。

- **确定性和可预测性**：垃圾收集的执行时间通常是不确定的，这可能不适合需要精确资源管理的场景。

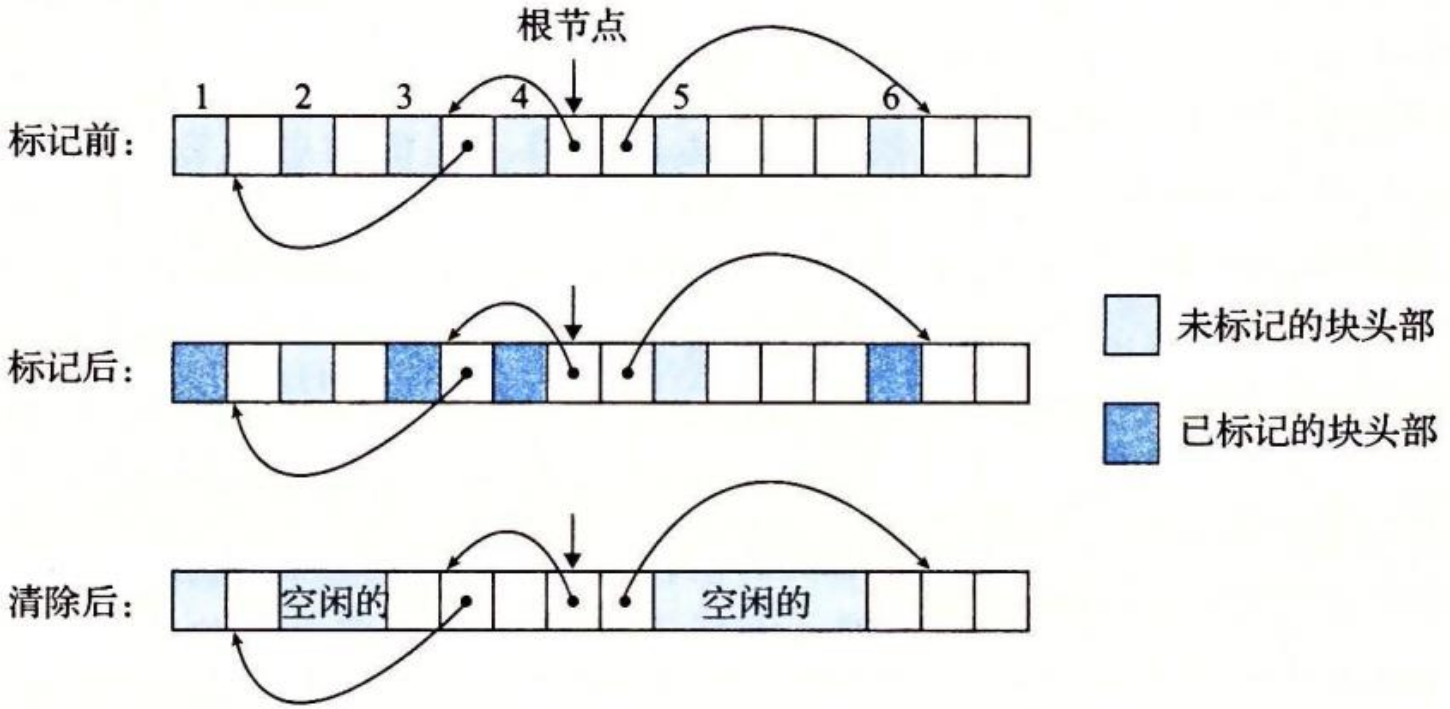
- **兼容性**：C++保持了与C语言的向后兼容性。C语言也是一种不包含内置垃圾收集的语言，它依赖于手动内存管理。这样的设计使得C++能够无缝地与大量现有的C代码和库一起工作。

A: Mark&Sweep垃圾收集器

垃圾收集

A: Mark&Sweep垃圾收集器

建立在malloc包的基础上



垃圾收集

A: Mark&Sweep垃圾收集器

```
void mark(ptr p) {
   if ((b = isPtr(p)) == NULL)
     return;
   if (blockMarked(b))
     return;
   markBlock(b);
   len = length(b);
   for (i=0; i < len; i++)
     mark(b[i]);
   return;
}
```

a）mark 函数

```
void sweep(ptr b, ptr end) {
   while (b < end) {
      if (blockMarked(b))
         unmarkBlock(b);
      else if (blockAllocated(b))
         free(b);
      b = nextBlock(b);
   }
   return;
}
```

b）sweep函数

内存相关错误大赏

- 我们已经接触过的：
  - 间接引用坏指针：避免裸指针的使用
  - 引用不存在的变量：返回局部变量
  - 弄混指针和指向对象
    - 弄混指针和指向对象的大小：在malloc时可能出现问题！
    - 引用指针而非对象：程序不会显式报错，会很危险！
  - 指针运算：ptr++
  - 引起内存泄漏：要记得delete/free
  - 栈缓冲区溢出：gets？使用更安全的函数！（fgets、scanf等）

内存相关错误大赏

- 错位错误

```
1   /* Create an nxm array */
2   int **makeArray2(int n, int m)
3   {
4       int i;
5       int **A = (int **)Malloc(n * sizeof(int *));
6
7       for (i = 0; i <= n; i++)
8           A[i] = (int *)Malloc(m * sizeof(int));
9       return A;
10  }
```

内存相关错误大赏

- 引用已经释放的变量

```
1   int *heapref(int n, int m)
2   {
3       int i;
4       int *x, *y;
5
6       x = (int *)Malloc(n * sizeof(int));
7
8       . . . // Other calls to malloc and free go here
9
10      free(x);
11
12      y = (int *)Malloc(m * sizeof(int));
13      for (i = 0; i < m; i++)
14          y[i] = x[i]++; /* Oops! x[i] is a word in a free block */
15
16      return y;
17  }
```

Practice

余文凯

The End