深 圳 大 学 实 验 报 告

课程名称：操作系统

实验项目名称：实验 3 内存分配与回收

学院：计算机与软件学院

专业：计算机科学与技术

指导教师：谭舜泉

报告人：黎浩然 冯海月 学号：2018112061 2018191116

实验时间：2021 年 5 月 18 日-2021 年 6 月 6 日

实验报告提交时间：2021 年 6 月 6 日

教务部制

实验目的与要求：

实验目的：

(1)、掌握计算机操作系统管理进程、处理机、存储器、文件系统的基本方法。

(2)、了解进程的创建、撤消和运行，进程并发执行；自行设计解决哲学家就餐问题的并发线程，了解线程（进程）调度方法；掌握内存空间的分配与回收的基本原理；通过模拟文件管理的工作过程，了解文件操作命令的实质。

(3)、了解现代计算机操作系统的工作原理，具有初步分析、设计操作系统的能力。

(4)、通过在计算机上编程实现操作系统中的各种管理功能，在系统程序设计能力方面得到提升。

实验要求：

(1)、回答以下问题：

kmem 中的 freelist 指针指向空闲物理块链表。空闲物理块链表中的节点为 run 结构体。但是：

```
struct run {
  struct run *next;
};
```

可以看到这个结构体只有指向下一个节点的指针。请解释这个链表中的空闲物理块保存在哪里呢？

(2)、大作业 partI-4 其中一个问题：

(vm.c L151) kpgdir = setupkvm();

通过 setupkvm 函数，创建了调度器所用的页表。请深入 setupkvm 函数内部，确定在创建页表过程中，总共调用了多少次 kalloc 函数分配 4K 物理块用于存放页表项？

请编程验证大作业 partI-4 的理论推导结果，在终端中输出在 setupkvm 函数中调用 kalloc 函数的次数。

说明：

（1）本次实验课作业满分为 100 分，占总成绩的比例（待定）。

（2）本次实验课作业截至时间 2021 年 6 月 6 日（周日）23:59。

（3）报告正文：请在指定位置填写，本次实验不需要单独提交源程序文件。

（4）个人信息：WORD 文件名中的“姓名”、“学号”，请改为你的姓名和学号；实验报告的首页，请准确填写“学院”、“专业”、“报告人”、“学号”、“班级”、“实验报告提交时间”等信息。

（5）提交方式：请在 BLACKBOARD 平台中按时提交；延迟提交不得分。

（6）发现抄袭（包括复制&粘贴整句话、整张图），该次作业记零分。

（7）期末考试阶段补交无效。

(1)、回答以下问题：

kmem 中的 freelist 指针指向空闲物理块链表。空闲物理块链表中的节点为 run 结构体。但是：

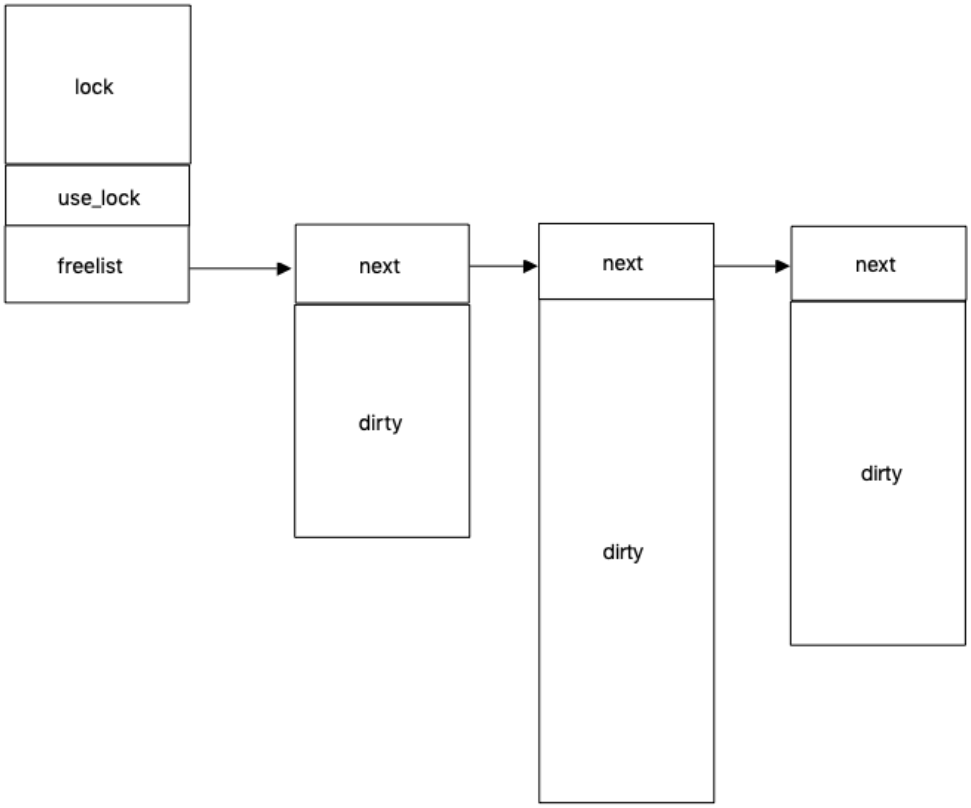
```
struct run {
  struct run *next;
};
```

可以看到这个结构体只有指向下一个节点的指针。请解释这个链表中的空闲物理块保存在哪里呢？

在 kalloc.c 文件中可以看到 kmem 结构体的定义如下：

```
1  // Physical memory allocator, intended to allocate
2  // memory for user processes, kernel stacks, page table pages,
3  // and pipe buffers. Allocates 4096-byte pages.
4
5  #include "types.h"
6  #include "defs.h"
7  #include "param.h"
8  #include "memlayout.h"
9  #include "mmu.h"
10 #include "spinlock.h"
11
12 void freerange(void *vstart, void *vend);
13 extern char end[]; // first address after kernel loaded from ELF file
14
15 struct run {
16   struct run *next;
17 };
18
19 struct {
20   struct spinlock lock;
21   int use_lock;
22   struct run *freelist;
23 } kmem;
```

从 kalloc.c 文件开头的注释可以推测，kmem 结构体中的 freelist 成员就是指向存放空闲物理内存块链表的指针。由于我接触过 GNU libc 中 malloc.c 的实现代码，所以很容易推测每一个物理块的指针被强转为 struct run 的指针。因为在 C 语言中，程序的所有指针类型大小是在编译时由 CPU 的兼容性和编译器的选项共同决定的。因此，在指针间进行强制类型转换是很常见的，一般不会有问题。



上图中的每个块中的 next + dirty 就是空闲的物理内存区域。

如上图所示，假设 freelist 指向的空闲物理内存链表中有三个空闲的物理块；虽然每一个空闲物理块的大小是不一的，但是这不影响我们将指向这些物理块的指针转换为指向 struct run 的指针，我们从代码中证实：

```
59  void
60  kfree(char *v)
61  {
62    struct run *r;
63
64    if((uint)v % PGSIZE || v < end || v2p(v) >= PHYSTOP)
65      panic("kfree");
66
67    // Fill with junk to catch dangling refs.
68    memset(v, 1, PGSIZE);
69
70    if(kmem.use_lock)
71      acquire(&kmem.lock);
72    r = (struct run*)v;
73    r->next = kmem.freelist;
74    kmem.freelist = r;
75    if(kmem.use_lock)
76      release(&kmem.lock);
77  }
```

在 kfree 函数中，由函数名(联想到 GNU libc 中的 free 函数)可以推测参数 v 就是指向要释放的物理内存块的指针。并且该空闲块插入 freelist 的时候是采用头插法的。

并且，在 kfree 函数中首先会检查该物理内存块的首地址是否对齐以及合法（在合法范围内），然后会将空闲块的内容全部字节填充为 \x01。所以上面图中的 dirty 全是 \x01 字节。然后在用头插法插入 freelist 的时候还会获取对应的锁。

实际上，main 函数将内核末尾和 PHYSTOP 之间的内存都作为一个初始的空闲内存池。kinit1 和 kinit2 调用 freerange 将内存加入空闲链表中，freerange 则是通过对每一页调用 kfree 实现该功能。分配器原本一开始没有内存可用，正是对 kfree 的调用将可用内存交给了分配器来管理。

```
void
freerange(void *vstart, void *vend)
{
  char *p;
  p = (char*)PGROUNDUP((uint)vstart);
  for(; p + PGSIZE <= (char*)vend; p += PGSIZE)
    kfree(p);
}
```

保证分配器只会释放对齐的物理地址

我们可以在虚拟内存中看到这样一部分对应的空闲内存空间，如下图所示。

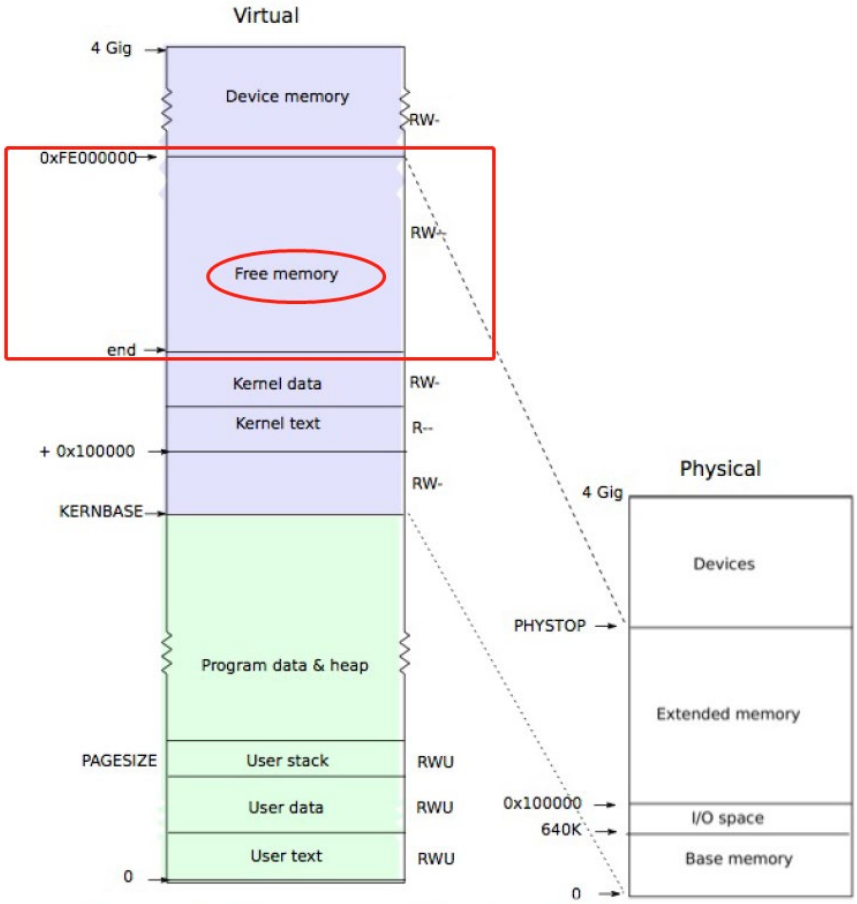


Figure 2-2. Layout of a virtual address space and the physical address space.

(2)、大作业 partI-4 其中一个问题：

(vm.c L151) kpgdir = setupkvm();

通过 setupkvm 函数，创建了调度器所用的页表。请深入 setupkvm 函数内部，确定在创建页表过程中，总共调用了多少次 kalloc 函数分配 4K 物理块用于存放页表项？

请编程验证大作业 partI-4 的理论推导结果，在终端中输出在 setupkvm 函数中调用 kalloc 函数的次数。

我们先来看看 setupkvm 函数

```
127  // Set up kernel part of a page table.
128  pde_t*
129  setupkvm(void)
130  {
131    pde_t *pgdir;
132    struct kmap *k;
133
134    if((pgdir = (pde_t*)kalloc()) == 0)
135      return 0;
136    memset(pgdir, 0, PGSIZE);
137    if (p2v(PHYSTOP) > (void*)DEVSPACE)
138      panic("PHYSTOP too high");
139    for(k = kmap; k < &kmap[NELEM(kmap)]; k++)
140      if(mappages(pgdir, k->virt, k->phys_end - k->phys_start,
141                  (uint)k->phys_start, k->perm) < 0)
142        return 0;
143    return pgdir;
144  }
```

可以在 setupkvm 函数中看到，函数首先在 134 行调用一次 kalloc 分配一个外层页表所需的空间，并存放该页表地址到 pgdir 变量中。所以只有一个外层页表就是 pgdir 所指向的区域。**而 kalloc 函数的调用次数等于外层页表的个数+内层页表的**个数。

容易知道 mappages 是负责内层页表的分配和映射的，我们来看看这个函数：

```
70  static int
71  mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm)
72  {
73    char *a, *last;
74    pte_t *pte;
75
76    a = (char*)PGROUNDDOWN((uint)va)
77    last = (char*)PGROUNDDOWN(((uint)va) + size - 1);
78    for(;;){
79      if((pte = walkpgdir(pgdir, a, 1)) == 0)
80        return -1;
81      if(*pte & PTE_P)
82        panic("remap");
83      *pte = pa | perm | PTE_P;
84      if(a == last)
85        break;
86      a += PGSIZE;
87      pa += PGSIZE;
88    }
89    return 0;
90  }
```

每一次调用 walkpgdir 会应该就是分配一个内层页表了。因为每一次调用 walkpgdir

如果无错误的话最多只会调用一次 kalloc，并且由其返回的值所在的变量名 pte（page table entry）可以推测。因此只需要考虑调用了多少次 walkpgdir 即可。

值得注意的是 mappages 函数的第 2、3 个参数。这两个参数指示了这次调用分配的内层页表需要映射/管控的虚拟内存范围。由前面的实验我们可以知道：一个内层页表有 4096 个字节，每个条目占 4 个字节，所以共 1024 个条目。而每个条目管控/映射一个内存页面，也就是 4096 字节。

所以一个内层页表负责映射/管控 1024 * 4096 B = 4 MB 的内存。

所以我们只要知道到底有多少内存是需要被映射的，然后将其除以 4MB，就能得到内层页表的数量!!!

```
115 static struct kmap {
116   void *virt;
117   uint phys_start;
118   uint phys_end;
119   int perm;
120 } kmap[] = {
121  { (void*)KERNBASE, 0,              EXTMEM,    PTE_W}, // I/O space
122  { (void*)KERNLINK, V2P(KERNLINK), V2P(data), 0},     // kern text+rodata
123  { (void*)data,     V2P(data),     PHYSTOP,   PTE_W}, // kern data+memory
124  { (void*)DEVSPACE, DEVSPACE,      0,         PTE_W}, // more devices
125 };
```

我们观察 kmap 这个全局结构数组，共有 4 个元素。这 4 个元素就是内层页表需要负责映射的区域。Setupkvm 通过一个 for 循环遍历 kmap 的 4 个元素，然后根据其值调用前面提到的 mappages 函数分配页表进行映射。

```
1  // Memory layout
2
3  #define EXTMEM  0x100000            // Start of extended memory
4  #define PHYSTOP 0xE000000           // Top physical memory
5  #define DEVSPACE 0xFE000000         // Other devices are at high addresses
6
7  // Key addresses for address space layout (see kmap in vm.c for layout)
8  #define KERNBASE 0ex80000000         // First kernel virtual address
9  #define KERNLINK (KERNBASE+EXTMEM)  // Address where kernel is linked
10
11 #ifndef __ASSEMBLER__
12
13 static inline uint v2p(void *a) { return ((uint) (a))  - KERNBASE; }
14 static inline void *p2v(uint a) { return (void *) ((a) + KERNBASE); }
15
16 #endif
17
18 #define V2P(a) (((uint) (a)) - KERNBASE)
19 #define P2V(a) (((void *) (a)) + KERNBASE)
20
21 #define V2P_WO(x) ((x) - KERNBASE)    // same as V2P, but without casts
22 #define P2V_WO(x) ((x) + KERNBASE)    // same as V2P, but without casts
```

结合 def.h 文件中 EXTMEM，KERNLINK 以及 V2P 的宏定义可以知道：kmap[0]、kmap[1]和 kmap[2]所指示的内存区域是连续的，其大小之和为 PHYSTOP 字节，也就是 0xE000000 B = 224 MB；而 kmap[4] 从 0xFE000000 到 0x0 共有 0x2000000 B = 32MB

于是需要映射的内存区域的大小为 224 MB + 32 MB = 256 MB；需要分配的内层页表的数量为 256MB / 4MB = 64。加上外层页表调用的一次 kalloc，所以**总共调用 65 次 kalloc 函数分配 4K 物理块用于存放页表项。**

下面通过代码验证上面的结论：

C vm.c

```c
#include "param.h"
#include "types.h"
#include "defs.h"
#include "x86.h"
#include "memlayout.h"
#include "mmu.h"
#include "proc.h"
#include "elf.h"

extern char data[];  // defined by kernel.ld
pde_t *kpgdir;  // for use in scheduler()
struct segdesc gdt[NSEGS];

int schedkallocount = 0;  // stats for sched proc kalloc's times
```

事实上，我们只需要测试一次调用 setupkvm 的过程中累计调用了多少次 kalloc 函数就可以了。在 setupkvm 所在的编译单元 vm.c 中定义一个全局变量 schedkallocount 用来计算一次 setupkvm 调用所调用的 kalloc 函数的次数。

```c
pde_t*
setupkvm(void)
{
  pde_t *pgdir;
  struct kmap *k;

  schedkallocount = 0;

  if((pgdir = (pde_t*)kalloc()) == 0)
    return 0;
  memset(pgdir, 0, PGSIZE);
  if (p2v(PHYSTOP) > (void*)DEVSPACE)
    panic("PHYSTOP too high");
  for(k = kmap; k < &kmap[NELEM(kmap)]; k++)
    if(mappages(pgdir, k->virt, k->phys_end - k->phys_start,
                (uint)k->phys_start, k->perm) < 0)
      return 0;

  cprintf("call kalloc times: %d\n", schedkallocount);
  return pgdir;
}
```

在 setupkvm 函数中的开始将这个变量置零，然后在要离开 setupkvm 之前打印出该变量的值。在 kalloc 函数内将 schedkallocount 递增。

C kalloc.c

```c
// Physical memory allocator, intended to allocate
// memory for user processes, kernel stacks, page table pages,
// and pipe buffers. Allocates 4096-byte pages.

#include "types.h"
#include "defs.h"
#include "param.h"
#include "memlayout.h"
#include "mmu.h"
#include "spinlock.h"

void freerange(void *vstart, void *vend);
extern char end[]; // first address after kernel loaded from ELF file
```

Kalloc 函数和 setupkvm 函数不在同一个编译单元内，所以利用 extern 关键字进行外部引用全局变量，并在 kalloc 函数加上一行代码即可。

```c
// Allocate one 4096-byte page of physical memory.
// Returns a pointer that the kernel can use.
// Returns 0 if the memory cannot be allocated.
char*
kalloc(void)
{
  struct run *r;

  if(kmem.use_lock)
    acquire(&kmem.lock);
  r = kmem.freelist;
  if(r)
    kmem.freelist = r->next;
  if(kmem.use_lock)
    release(&kmem.lock);
  schedkallocount++;
  return (char*)r;
```

编译 xv6：

```
root@cd691b04d3cd:/home/xv6-public# make qemu-nox
gcc -fno-pic -static -fno-builtin -fno-strict-aliasing -Wall -MD -ggdb -m32 -Werror -fno-omit-frame-pointer -fno-stack-protector   -c -o vm.o vm.c
ld -m    elf_i386 -T kernel.ld -o kernel entry.o bio.o console.o exec.o file.o fs.o ide.o ioapic.o kalloc.o kbd.o lapic.o log.o main.o mp.o picirq.o pipe.o proc.o sp
inlock.o string.o swtch.o syscall.o sysfile.o sysproc.o timer.o trapasm.o trap.o uart.o vectors.o vm.o  -b binary initcode entryother
objdump -S kernel > kernel.asm
objdump -t kernel | sed '1,/SYMBOL TABLE/d; s/ .* / /; /^$/d' > kernel.sym
dd if=/dev/zero of=xv6.img count=10000
10000+0 records in
10000+0 records out
5120000 bytes (5.1 MB, 4.9 MiB) copied, 0.0341814 s, 150 MB/s
dd if=bootblock of=xv6.img conv=notrunc
1+0 records in
1+0 records out
512 bytes copied, 6.598e-05 s, 7.8 MB/s
dd if=kernel of=xv6.img seek=1 conv=notrunc
265+1 records in
265+1 records out
136048 bytes (136 kB, 133 KiB) copied, 0.000930101 s, 146 MB/s
qemu-system-x86_64  -nographic -hdb fs.img xv6.img -smp 2 -m 512
WARNING: Image format was not specified for 'fs.img' and probing guessed raw.
         Automatically detecting the format is dangerous for raw images, write operations on block 0 will be restricted.
         Specify the 'raw' format explicitly to remove the restrictions.
WARNING: Image format was not specified for 'xv6.img' and probing guessed raw.
         Automatically detecting the format is dangerous for raw images, write operations on block 0 will be restricted.
         Specify the 'raw' format explicitly to remove the restrictions.
qemu-system-x86_64: warning: TCG doesn't support requested feature: CPUID.01H:ECX.vmx [bit 5]
qemu-system-x86_64: warning: TCG doesn't support requested feature: CPUID.01H:ECX.vmx [bit 5]
xv6...
cpu1: starting
call kalloc times: 65
cpu0: starting
call kalloc times: 65
init: starting sh
call kalloc times: 65
call kalloc times: 65
```

```
qemu-system-x86_64: warning: TCG doesn't support requested feature: CPUID.01H:ECX.vmx [bit 5]
qemu-system-x86_64: warning: TCG doesn't support requested feature: CPUID.01H:ECX.vmx [bit 5]
xv6...
cpu1: starting
call kalloc times: 65
cpu0: starting
call kalloc times: 65
init: starting sh
call kalloc times: 65
call kalloc times: 65
$
```

可以看到，每一次调用 setupkvm **一共调用了 65 次 kalloc**，所以分配了一个外层页表和 64 个内层页表。

++

其他（例如感想、建议等等）。

（1） 通过阅读源代码，深入理解了 xv6 物理内存分配器部分实现原理。xv6 会通过维护一个物理页组成的链表来寻找空闲页。结构体 kmem 中包含空闲页的链表 freelist，还有一个保护空闲链表的锁 lock。分配器将每个空闲页的 run 结构体保存在该空闲页本身中。函数 kfree 将这部分内存释放，用脏数据填充。

（2） 理论推导 setupkvm 函数通过 65 次对 kalloc 函数的调用，分配 4K 物理块用于存放调度器所需的页表项。由于 kmap 前三个段在虚拟空间中连续，其空间尺寸为 PHYSTOP=0xE000000，加上 kmap 第四段的大小，即为内层页表需要映射的内存区域。kalloc 的函数的调用次数实际上等于外层页表（1）和内层页表（64）的总个数。同时，我们编写代码来验证理论分析的正确性。

指导教师批阅意见：   成绩评定：   指导教师签字：谭舜泉 2021 年 6 月 13 日
备注：

注：1、报告内的项目或内容设置，可根据实际情况加以调整和补充。

2、教师批改学生实验报告时间应在学生提交实验报告时间后 10 日内。