|  |  |  |
| --- | --- | --- |
| **验收成绩** | **报告成绩** | **总评成绩** |
|  |  |  |

**武汉大学计算机学院**

**本科生课程实验报告**

**操作系统内核实验**

专 业 名 称 ：计算机科学与技术

课 程 名 称 ：操作系统课程设计

指 导 教 师 ：李祖超

学 生 学 号 ：2022302111469

学 生 姓 名 ：王垚

学 年 学 期 ：2023-2024学年第二学期

完 成 时 间 ： 2024.7.15

成 绩 ：

二○二四年七月

**郑 重 声 明**

本人呈交的实验报告，是在指导老师的指导下，独立进行实验工作所取得的成果，所有数据、图片资料真实可靠。尽我所知，除文中已经注明引用的内容外，本实验报告不包含他人享有著作权的内容。对本实验报告做出贡献的其他个人和集体，均已在文中以明确的方式标明。本实验报告的知识产权归属于培养单位。

本人签名： 王垚 日期： 2024.7.10

摘 要

操作系统内核实验的实验目的是通过实践性的实验，加深堆操作系统内核原理和实现的理解。通过对 xv6 操作系统的内核结构进行实验，深入理解操作系统的基本组成部分，包括进程管理、内存管理、文件系统和设备驱动等。在实践过程中学习和掌握在内核空间进行编程的技巧，了解在内核中与在用户空间中编程的区别，掌握对内存管理、进程调度和中断处理等方面的编程技巧。通过实验锻炼在复杂系统中定位和解决问题的能力，熟悉常用的内核调试工具和方法。通过对内核代码的阅读和修改，理解操作系统设计中的关键思想和策略，如同步机制、资源管理和系统调用接口设计等。通过具体实现和测试内存分配算法，加深对内存管理的理解。

实验设计主要遵循xv6系统的内核结构和武汉大学操作系统实验手册中的要求，在完成实验手册中的要求的同时，通过阅读xv6相关资料以及在网络中寻找相关资料来加强自己对操作系统内核的理解。

实验内容主要包括：

·实验前环境搭建以及相关知识准备：在QEMU模拟器中运行xv6，熟悉基本的操作和使用方法；阅读xv6的内核代码，了解内核初始化、进程管理、内存管理和文件系统等模块的基本实现。

·系统调用实验：对xv6代码进行修改，增加一个新的系统调用，实现返回正在使用的进程数的要求。了解系统调用的工作流程以及内核态和用户态之间调用的联系。

·内存管理实验：在 xv6 的内存管理模块中实现伙伴系统内存分配算法。定义伙伴系统的数据结构。实现 buddy\_malloc() 和 buddy\_free() 函数，完成内存块的分配和释放。实现内存块的分割和合并逻辑，确保内存分配的高效性。在编写完成后对伙伴系统得到内存分配和释放编写测试程序进行验证，并通过xv6中consoleread函数实现在命令行中直接进行交互的功能。

实验结论为通过对 xv6 内核的阅读和修改，我深入理解了操作系统内核的基本结构和实现细节，掌握了进程管理、内存管理的核心原理。实验中的调试过程，锻炼了我在虚拟机环境下、xv6操作系统的环境中定位和解决问题的能力，熟悉了内核调试工具和方法。我成功实现了伙伴系统的内存分配算法，并通过实验验证了其正确性和有效性，进一步加深了我对内存管理策略的理解和认识。

**关键词：**xv6系统；系统调用；内存管理；伙伴系统

目 录

[1 实验环境搭建 6](#_Toc560)

[1.1 环境搭建 6](#_Toc29571)

[1.1.1准备Linux环境 6](#_Toc13043)

[1.1.2 下载xv6的源代码 6](#_Toc29148)

[1.1.3 下载所需要的工具 6](#_Toc22307)

[1.1.4 make qemu进入xv6 7](#_Toc13373)

[1.1.5 配置Vscode环境 8](#_Toc23888)

[1.2 问题及解决方案 10](#_Toc11110)

[2 系统调用实验 11](#_Toc1506)

[2.1实验内容 11](#_Toc1570)

[2.2 实验原理 11](#_Toc31862)

[2.3 实验步骤 12](#_Toc428)

[2.4 实验结果 14](#_Toc4389)

[3 内存管理实验 16](#_Toc15966)

[3.1 实验内容 16](#_Toc14583)

[3.2 实验原理 16](#_Toc17102)

[3.2.1 kalloc() 16](#_Toc30488)

[3.2.2 kfree() 16](#_Toc26809)

[3.2.3伙伴系统 17](#_Toc7304)

[3.3 实验步骤 17](#_Toc2676)

[3.4 实验结果 25](#_Toc31316)

[3.5 实验中遇到的问题及解决方案 35](#_Toc15547)

[4 实验总结 37](#_Toc17056)

[5 参考文献 38](#_Toc18)

[教师评语评分 39](#_Toc30478)

# 1 实验环境搭建

在进行实验之前，我们首先要搭建好基于RISC-V的xv6实验环境，通过使用交叉编译技术和模拟器来模拟RISC-V的环境。在xv6操作系统开发的过程中，交叉编译指的是在Linux上编译生成一个可以在xv6架构的模拟环境（qemu）上运行的xv6内核和用户程序。

## 1.1 环境搭建

### 1.1.1准备Linux环境

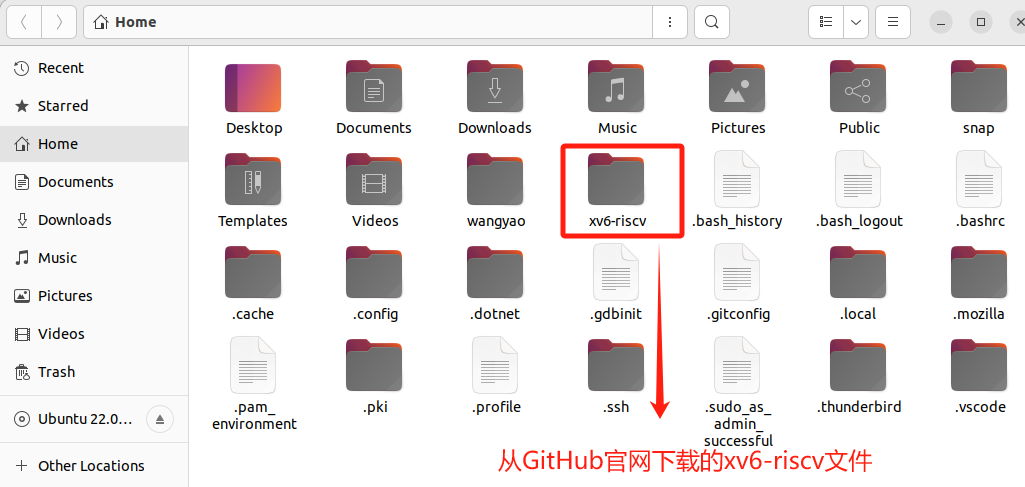
我所用的Linux环境是Ubuntu22.04，环境准备好之后使用以下命令来安装交叉编译器，其余内容不再赘述。

1. sudo apt-get install gcc-riscv64-linux-gnu

### 1.1.2 下载xv6的源代码

从官方的GitHub仓库上将xv6的源代码进行clone，并在终端使用cd xv6-riscv命令进入该文件环境

1. git clone https:*//github.com/mit‐pdos/xv6‐riscv.git*



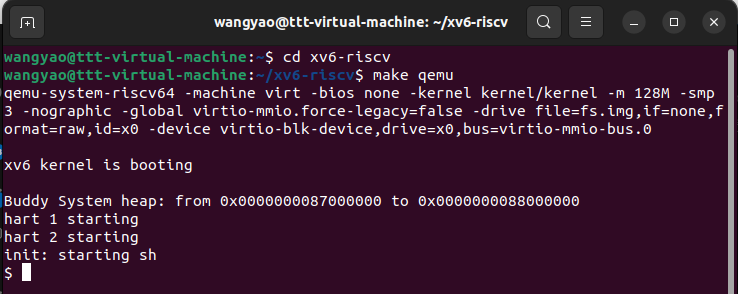
### 1.1.3 下载所需要的工具

在ubuntu系统中，使用以下的命令对所需要的工具进行安装，而事实上只需要根据终端中运行make qemu 所给出的报错，安装所需要的工具包即可。

1. sudo apt install binutils‐riscv64‐linux‐gnu
2. sudo apt install gcc‐riscv64‐linux‐gnu
3. sudo apt install gdb‐multiarch
4. sudo apt install qemu‐system‐misc opensbi u‐boot‐qemu qemu‐utils

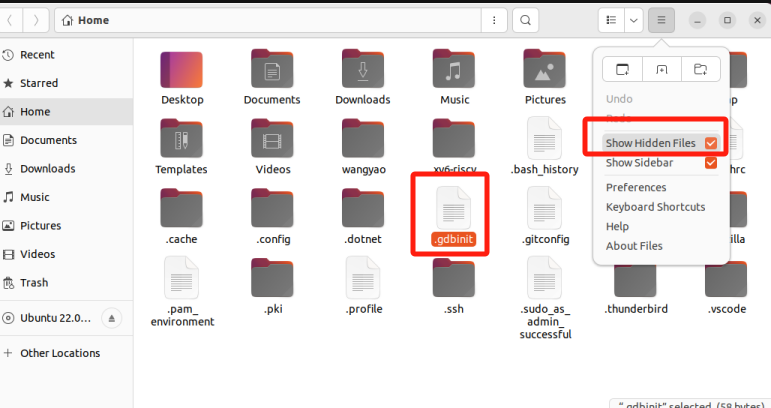
### 1.1.4 make qemu进入xv6

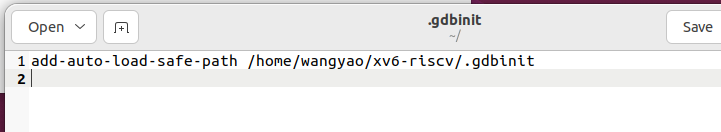
在终端中输入cd xv6-riscv 进入xv6目录，接着输入make qemu，进入xv6操作系统。终端出现以下提示就说明前面的配置成功，可以继续。



接着，按ctrl+a，再点击x，退出qemu。

在终端中新开一个窗口，输入make qemu-gdb，将会在当前目录（xv6-riscv）生成.gdbinit，gdb会默认首先执行当前目录下的.gdbinit文件，但是在第一次执行的时候会被阻止，这时需要在Home目录下将隐藏的文件全部展现出来，这时就会注意到这里也有一个.gdbinit文件，在这个文件中添加一行代码（意思是将该路径添加到gdb 的自动加载安全路径列表中），即可解决上述问题。





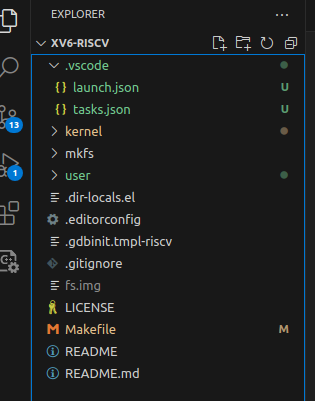
GDB在启动的时候会尝试自动加载以下初始化文件（.gdbinit文件），但是出于安全考虑，GDB默认不会自动加载位于用户主目录意外路径的.gdbinit文件，该命令告诉GDB，/home/wangyao/xv6-riscv/.gdbinit这个路径下的.gdbinit文件是安全的，可以自动加载。

### 1.1.5 配置Vscode环境

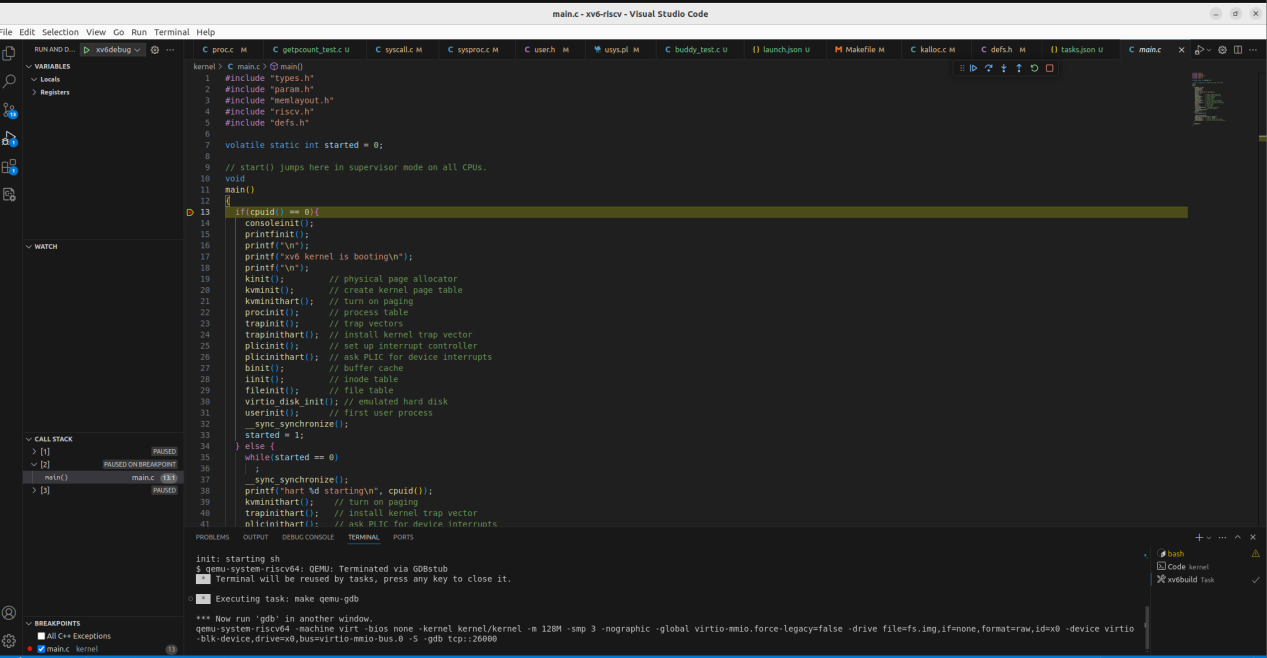
在开始前，先在Vscode中安装C/C++拓展，接着用Vscode打开xv6-riscv文件，创建.vscode文件夹，在该文件夹下创建以下两个文件

1. launch.json
2. {
3. *// Use IntelliSense to learn about possible attributes.*
4. *// Hover to view descriptions of existing attributes.*
5. *// For more information, visit: https://go.microsoft.com/fwlink/?linkid=830387*
6. "version": "0.2.0",
7. "configurations": [
8. {
9. "name": "xv6debug",
10. "preLaunchTask": "xv6build",
11. "type": "cppdbg",
12. "request": "launch",
13. "program": "${workspaceFolder}/kernel/kernel",
14. "args": [],
15. "stopAtEntry": false,
16. "cwd": "${workspaceFolder}",
17. "miDebuggerServerAddress": "127.0.0.1:26000",
18. "environment": [],
19. "externalConsole": false,
20. "MIMode": "gdb",
21. "miDebuggerPath": "/usr/bin/gdb-multiarch",
22. "setupCommands": [
23. {
24. "description": "pretty-printing",
25. "text": "-enable-pretty-printing",
26. "ignoreFailures": true
27. }
28. ]
29. }
30. ]
31. }
32. tasks.json
33. *// xv6-riscv/.vscode/tasks.json*
34. {
35. "version": "2.0.0",
36. "tasks": [
37. {
38. "label": "xv6build",
39. "type": "shell",
40. "isBackground": true,
41. "command": "make qemu-gdb",
42. "problemMatcher": [
43. {
44. "pattern": [
45. {
46. "regexp": ".",
47. "file": 1,
48. "location": 2,
49. "message": 3
50. }
51. ],
52. "background": {
53. "beginsPattern": ".\*Now run 'gdb' in another window.",
54. *// 要对应编译成功后,一句echo的内容. 此处对应 Makefile Line:170*
55. "endsPattern": "."
56. }
57. }
58. ]
59. }
60. ]
61. }

配置好后，整个文件夹的文件构造如图所示：



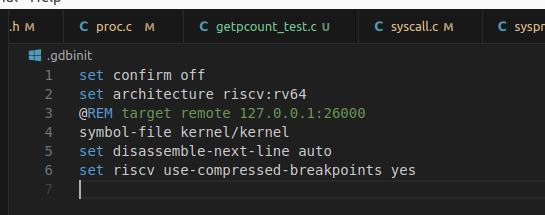
接着就能够通过F5按键来对代码进行调试。点击F5,程序停在断点处，证明环境配置成功。



## 1.2 问题及解决方案

在点击F5对代码进行调试时，Vscode会弹出提示框，显示无法开始调试，这是因为.gdbinit文件中有target remote 127.0.0.1:26000，这个文件会被GDB先执行一遍。而在launch.json中指定"miDebuggerServerAddress": "127.0.0.1:26000",导致同一个remote address被执行了两次，因此报错。

解决方法：将.gdbinit中对应的那行代码注释掉。（@REM是.gdbinit的注释符号）。



# 2 系统调用实验

## 2.1实验内容

在RISC-V体系结构上运行的xv6操作系统中引入一个新的系统调用，执行返回系统中当前进程的数量。

## 2.2 实验原理

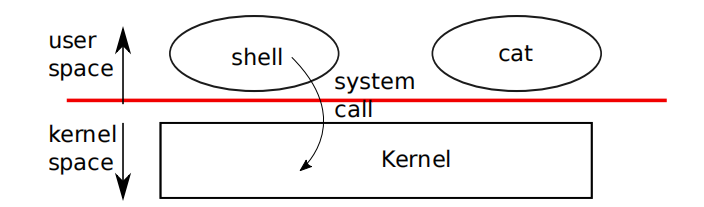
xv6的体系结构主要包含以下几个部分：

·内核（Kernel）：是xv6操作系统的核心，负责管理硬件资源和提供系统服务，包含进程管理、内存管理、文件系统、设备驱动、系统调用接口等关键组件。

·用户空间(User Space):是运行用户进程的地方，包含用户应用程序以及库文件。用户程序通过系统调用接口与内核实现交互，从而使用操作系统提供的服务。

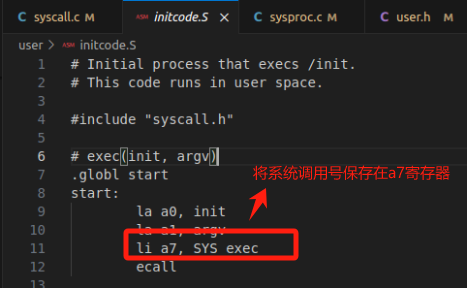
·终端和异常处理：xv6通过中断和异常处理机制处理硬件事件和程序错误。

·系统调用：是用户进程和内核交互的主要方式，用于进程管理、文件操作、设备管理等。



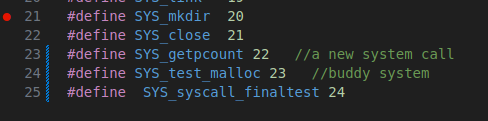
当用户模式下的应用程序试图执行特权指令时，CPU不会直接执行，而是切换到管理模式，以便管理模式的代码可以终止应用程序。

以exec系统调用为例。用户代码将exec所需要的参数放在寄存器a0和a1中，并将系统调用郝放在a7中。系统调用号与syscalls数组中的条目相匹配。ecall指令陷入到内核中，执行uservec、usertrap、syscall，syscall从陷阱帧(trapframe)中保存的a7中检索系统调用号，并用它索引到syscalls中。

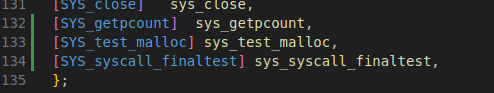
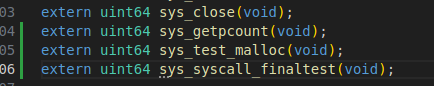


## 2.3 实验步骤

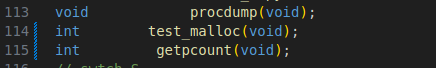
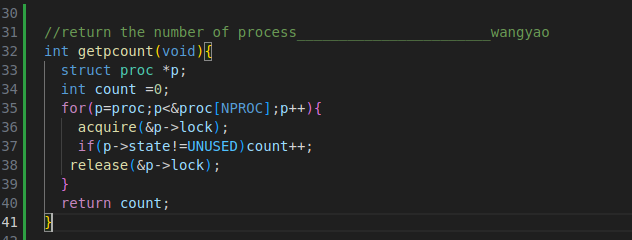
1、在“kernel/syscall.h”中添加SYS\_getpcoount的定义并分配新的系统调用号。



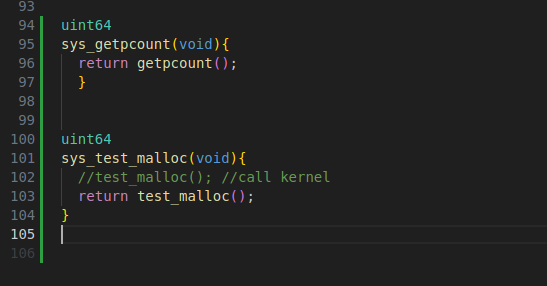
1. 在“kernel/syscall.c”中声明sys\_syscall\_finaltest函数，并在syscalls[]系统调用表中添加新的系统调用，将处理函数绑定到系统调用号。



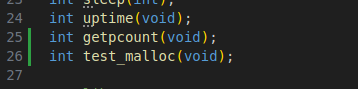
1. 在“kernel/proc.c”文件中添加getpcount()的函数体实现,并在“kernel/def.h”文件中声名函数，方便后面对该函数的调用。

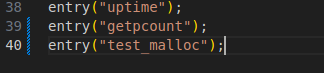


4、在“kernel/sysproc.c”中定义处理函数



5、在“user/user.h”中声明API函数原型，命名为int getpcount(void),编辑“user/usys.pl”脚本添加entry("getpcount");

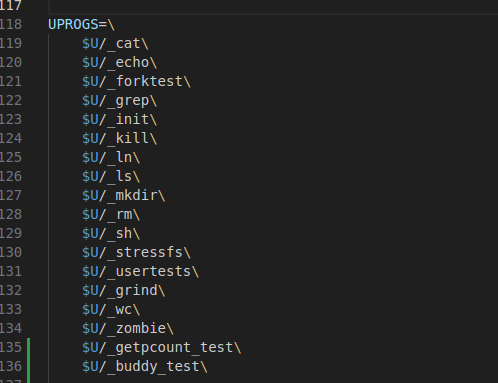




6、在“user”目录下创建一个测试文件，命名为getpcount\_test.c，用于对系统调用进行测试,函数体如下.

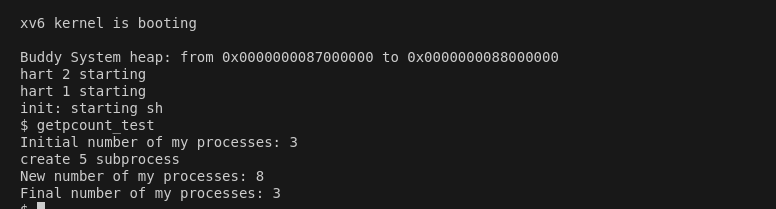
1. #include "kernel/types.h"
2. #include "user/user.h"
3. int main(int argc, char \*argv[])
4. {
5. int pcount = getpcount();
6. printf("Initial number of my processes: %d\n", pcount);
7. *//create process*
8. for(int i=0;i<5;i++){
9. if(fork()==0){
10. *//the subproc sleep for a while and the exit*
11. sleep(10);
12. exit(0);
13. }
14. }
15. printf("create 5 subprocess\n");
16. sleep(5); *//process wait to make sure that all the subprocesses have started*
17. int new\_count=getpcount();
18. printf("New number of my processes: %d\n", new\_count);
19. for (int i=0;i<5;i++){
20. wait(0);
21. }
22. int final\_count=getpcount();
23. printf("Final number of my processes: %d\n", final\_count);
24. exit(0);
25. }

7、将新建的C文件纳入本项目的编译过程，通过编辑Makefile文件，在UPROGS段落中添加一行：$U/\_getpcount\_test\,自此，所有工作结束。



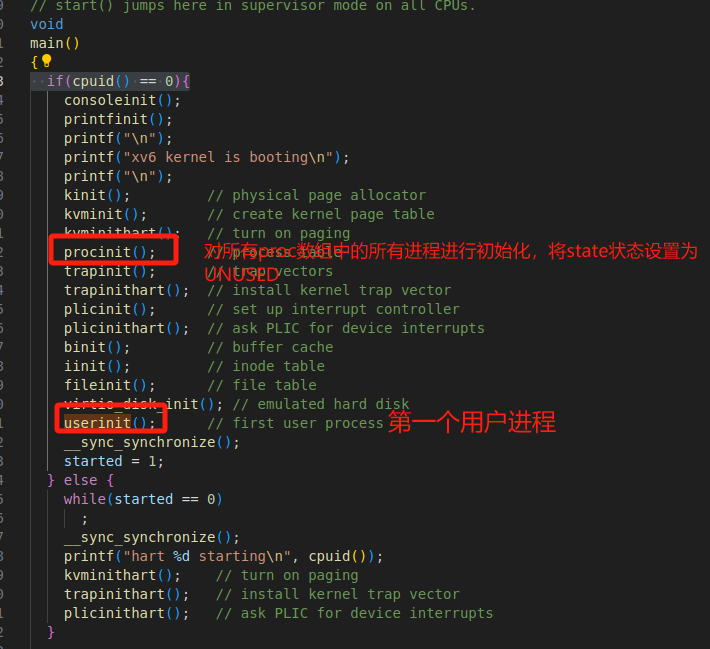
## 2.4 实验结果

重新编译并启动xv6，执行该测试文件，终端返回结果如下：



从运行的结果中看出，在没有添加子进程之前，运行的进程个数为3，原因是：

在程序启动时，先运行kernel/main.c程序，其中的procinit()函数对进程的数组进行初始化，并将所有进程的状态都设置为UNUSED,接下来调用userinit()函数创建第一个进程，并将该进程的状态修改为RUNABLE，此时count=1;userinit()进程启动完毕之后，将启动其他必要的进程，如shell进程，为用户提供一个交互的界面，因此系统会继续执行user/init.c程序，用于启动shell，此时count=2;调用自己的getpcount()程序时,count=3,因此最终返回的进程数是3.



而在创建5个子进程之后，返回的值是8；接着父进程释放所有子进程，最终的结果还是3。

# 3 内存管理实验

## 3.1 实验内容

设计并实现一个类似malloc的动态内存分配器，用于为内核态程序运行时分配和释放不同大小的内存块。借鉴经典的动态内存分配算法，如伙伴系统,设计合适的支持内存分配与合并的数据结构，并设计内存分配和释放的接口函数，比如malloc(size)和free(pointer)。

修改kernel/kalloc.c文件，在xv6内核中实现这种内存分配策略：按照设计的接口，实现分配器。分配器应能够管理多种大小的内存块，并在内核中高效运行。

编写测试用例来验证分配器在不同情况下的行为，包括但不限于内存申请、释放和边界情况处理，验证内存分配器的正确性、以及处理不同大小需求的能力。

## 3.2 实验原理

在xv6内核空间中采用了kalloc()和kfree()函数对内存进行管理，而现在，我在内核空间中Free memory段落的顶部扣减16MB的内存，搭建一个新的堆，以字节为单位重新构造一个分配和释放机制。

### 3.2.1 kalloc()

kalloc()函数用于分配一个页（4KB）的物理内存，并返回该内存的指针。从kmem.freelist中取出一个空闲页，弱空闲列表不为空，则将第一个空闲页从列表中移除，并返回该页的地址。分配后，使用memset将分配的页填充为特定的值帮助调试，在这个过程中使用锁来确保对空闲列表的操作是线程安全的。

### 3.2.2 kfree()

kfree() 函数用于释放一个之前分配的页，将其返回到空闲页列表中。接收一个物理地址pa，检查其对齐和范围以确保是一个有效的页。使用 memset 将释放的页填充为特定的值（例如 1），以帮助调试，防止悬挂引用（dangling references）。

将该页添加回 kmem.freelist，使其可以再次分配。使用锁确保对空闲列表的操作是线程安全的。

### 3.2.3伙伴系统

伙伴系统（Buddy System）是一种内存分配算法，用于动态分配内存并且能够高效地进行内存合并。

内存块 (Block)： 内存被划分成多个大小不同的块，每个块的大小是2的幂。例如，块的大小可能是1KB, 2KB, 4KB, 8KB等。每个块用一个结构体来表示，包含块的大小（阶数）和指向下一个空闲块的指针。

伙伴 (Buddy)： 每个块都有一个“伙伴”，这个伙伴块与其大小相同并且在内存中是相邻的。例如，一个4KB的块从地址0开始，它的伙伴块是从地址4KB开始的另一个4KB的块。

有内存管理器维护一个自由链表数组，每个数组的元素指向对应阶数的空闲块链表。内存分配的主要步骤如下:

· 确定阶数：根据请求的内存大小确定需要分配的最小阶数的块。

· 查找空闲块：从该阶数对应的自由链表中寻找一个空闲块。如果找到，直接分配该块。

·分割块：如果该阶数的自由链表中没有空闲块，则从更大的阶数的自由链表中寻找一个空闲块，将其分割成两个较小的块，直到得到所需大小的块。

通过对以上原理的了解，我将借助伙伴系统的方式实现对堆内存的管理。

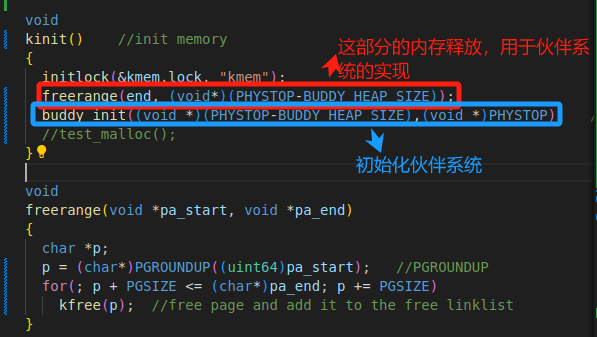
## 3.3 实验步骤

在内存管理实验中，我借鉴伙伴系统的内存管理方式，实现实验的要求。

1. 分配内存

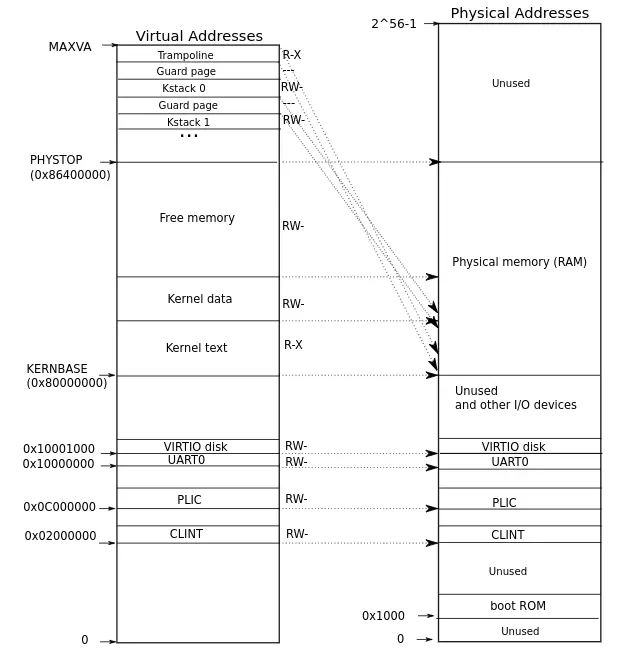
“kernel/kalloc.c”中的freerange(void \*pa\_start, void \*pa\_end)函数用于释放从

pa\_start到pa\_end之间的所有物理页，并将他们添加到空闲页链表。修改kinit()函数，用于初始化系统内存，在这个函数中调用freerange()函数，对end到PHYSTOP-BUDDY\_HEAP\_SIZE之间的空间进行释放。（end指的是内核代码段结束的位置，PHYSTOP是物理内存的终止地址）



qemu模拟器将大小为128MB的物理地址[0x80000000,0x88000000]作为可用的物理内存，而我将[0x87000000,0x88000000]的高16MB分配给了buddy system用作新的内存管理地址。创建buddy\_init(void \*heap\_start,void \*heap\_end)函数对堆地址进行初始化，并在内存初始化之时调用（即在kinit函数中调用）。

1. *//initial the buddy system*
2. void buddy\_init(void \*heap\_start,void \*heap\_end){
3. buddy\_system.base=(char \*)heap\_start;  *//the start area of the heap*
4. memset(buddy\_system.free\_list,0,sizeof(buddy\_system.free\_list));
5. struct block \*initial\_block=(struct block \*)buddy\_system.base;
6. initial\_block->next=0;
7. buddy\_system.free\_list[MAX\_ORDER]=initial\_block;
8. printf("Buddy System heap: from %p to %p\n",buddy\_system.base,buddy\_system.base+BUDDY\_HEAP\_SIZE);
9. }



内核的内存布局

1. 创建所需的数据结构

我创建了两个结构体block:表示一个空闲内存块，其中包含指向下一个空闲内存块的next指针以及该内存块对应的阶数:order;接着创建了一个buddy系统，包含了一个空闲指针数组free\_list[]和堆的其实地址：base(堆内存管理的区域从base开始)；如free\_list[3]存储的是大小为8个单位的空闲内存块链表的头指针。

接着，我声明了一个buddy 结构体类型的编量buddy\_system，用于管理整个buddy系统的状态。

1. struct block{   *//memory block*
2. struct block \*next;
3. int order; *//its order num*
4. };
5. struct buddy{
6. struct block \*free\_list[MAX\_ORDER+1];  *//store the linklist*
7. char \*base;   *//the start area of heap*
8. };
9. struct buddy buddy\_system;  *//create buddy system*

在该伙伴系统中所关注的重要参数如下：

在内核空间的最高地址区域，我开辟了16MB的堆。

最小块的大小是16B(在程序中，地址的计算是以字节为单位)；

堆的大小是16MB

最大的阶数是20：由于堆的大小是16MB=16\*2^20,因此最大的阶数是20

1. *//buddy system parameter*
2. #define MIN\_BLOCK\_SIZE 16  *//16B*
3. #define MAX\_ORDER 20  *//16\*2^20=16MB=BUDDY\_HEAP\_SIZE*
4. #define BUDDY\_HEAP\_SIZE (16\*1024\*1024)  *//16MB*

在我创建的伙伴系统中，我将通过以下的机制进行内存的管理：

1. 内存块的划分

· 内存区域被划分成大小为 2 的幂次方的块。每个块有一个 order，表示块的大小。

· 例如，order 为 0 的块大小为 1 个单位，order 为 1 的块大小为 2 个单位。

1. 分配和释放内存

· 当请求内存时，系统会找到适当大小的块进行分配。如果没有找到合适的块，系统会分裂更大的块来满足需求。

· 当释放内存时，系统会尝试合并相邻的块，以减少碎片并优化内存使用。

1. 空闲块的管理

· 系统使用一个链表数组 free\_list[]来管理不同大小的空闲内存块。free\_list[i] 存储大小为 2^i 的空闲内存块链表头。

· 每个空闲块通过 struct block 结构体的 next 指针链接在一起。

1. 合并空闲块

· 当两个相邻的同阶空闲块被释放时，它们可以合并成一个更大的块，其阶数增加 1。

· 这种合并过程持续进行，直到不能再合并为止。

1. 实现内存分配

我创建了一个buddy\_malloc(uint64 size)函数对内存进行分配：

首先我构建一个get\_order函数，便于在每次分配内存块时，对内存块大小相应的order的值进行计算：当所求的内存大小小于最小的块的大小时，向上取整，分配一个最小的块给它，否则，通过一个循环来寻找正确的order。

1. *//calculate the order of the block*
2. int get\_order(int size){
3. int order =0;
4. if(size<MIN\_BLOCK\_SIZE){
5. size=MIN\_BLOCK\_SIZE;
6. }
7. while((1<<(order+4))<size){  *//move left 4bit (not suit )*
8. order++;
9. }
10. return order;
11. }
12. void \*
13. buddy\_malloc(uint64 size){
14. printf("try to malloc the area");
15. int order=get\_order(size+sizeof(struct block));  *//the block is a space of pointer*
16. for (int current\_order =order;current\_order<=MAX\_ORDER;current\_order++){
17. if(buddy\_system.free\_list[current\_order]){
18. *//find the block and allocate it*
19. struct block \*block\_to\_alloc=buddy\_system.free\_list[current\_order];
20. buddy\_system.free\_list[current\_order]=block\_to\_alloc->next;
21. block\_to\_alloc->order=current\_order;
22. *//if the current\_order>order,we should cut the block to a smaller one*
23. while (current\_order>order){
24. current\_order--;
25. unsigned long block\_address=(unsigned long) block\_to\_alloc;
26. struct block \*buddy\_block =(struct block \*)(block\_address+(1<<(current\_order+4)));
27. //add the buddy\_block to the free\_list
28. buddy\_block->next =buddy\_system.free\_list[current\_order];
29. buddy\_block->order=current\_order;  *//set the order*
30. block\_to\_alloc->order=current\_order;
31. buddy\_system.free\_list[current\_order]=buddy\_block;
32. }
33. void \*my\_ptr=(void \*)((char\*)block\_to\_alloc+sizeof(struct block ));
34. return my\_ptr;
35. }
36. printf("fail to alloctae the area."); //fail
37. return 0;
38. }

在计算阶数时，要在分配的内存大小size的基础上加上struct block结构体的大小，因为我们需要在分配的块中存储struct block的信息（包含next指针和order）；接着我通过for循环来寻找合适的内存块：

从order开始遍历整个free\_list数组来寻找合适大小的空闲块，当寻找到current\_order阶数对应的空闲块时，将其从空闲链表的首部进行移除，并设置块的order。接着，要对块的大小进行判断，如果current\_order=order（get\_order所计算的合适的大小），那么则表示这个块就是大小最合适的块，不需要进行分割，于是直接分配，如果current\_order>order,则证明这个块的大小比它所需要的内存大小更大，如果直接分配则会造成浪费，因此要对该内存块进行分割，我采用while循环进行实现，只有在current\_order=order的条件下才会跳出循环，保证了内存分配的合理性。

对于while 循环的逻辑：如果找到的块的current\_order依旧大于order，那么则需要将块分割成更小的块，直到等于order；而buddy\_block是分割出来的块，需要通过头插法将其添加到对应的current\_order的空闲链表中。

 struct block \*buddy\_block =(struct block \*)(block\_address+(1<<(current\_order+4)));

用于计算buddy\_block的地址（在block\_address的基础上加上块的大小，4对应的是16B=2^4B）

当无法找到对应的块进行分配时，在终端输出失败信息并return 0。

1. 实现内存释放并合并

我创建一个buddy\_free(void \*ptr)函数来对之前分配的内存块进行释放，并调用merge\_block( struct block\* free\_block,int order)函数对空闲块进行合并。

由于在分配地址时，加上了指针的大小，因此在释放空间时，首先要减去指针的大小，再利用头插法将该空闲块利用头插法插入空闲链表中。接着对空闲链表中的伙伴块进行合并。在合并时考虑两种不同的情况：伙伴块的地址在该空闲块之前或之后。

以伙伴块的地址在空闲块之前为例：首先设定一个merge变量，标志着在本次函数调用的过程中是否有对块进行合并，接着计算buddy\_block的地址。当寻找到对应的伙伴块时，将该伙伴块从空闲链表中移出，并合并，更新新的合并后的空闲块的地址（哪个在前就以哪个块的地址作为新的空闲块的地址）和对应的阶数，并将merge的值设为1，表示成功合并；另一种情况同理，当整个函数体执行过后，merge的值仍为0时，表示空闲链表中没有需要合并的块，因此不进行合并，退出函数，否则，只要有发生合并，那么则需要进行递归，继续进行合并，直到所有需要合并的块都被合并为止。

1. void merge\_block( struct block\* free\_block,int order){
2. *//merge the block of the free\_list*
3. unsigned long free\_addr=(unsigned long)free\_block;
4. int merge=0; *//flag*
5. *//calculate the buddy\_block of the free block*
6. *//case `1:before*
7. unsigned long buddy\_addr=free\_addr-(1<<(order+4));
8. if((char\*)buddy\_addr>=buddy\_system.base){
9. struct block \*buddy\_block =(struct block\*)buddy\_addr;
10. *//search for the buddy block in the free list*
11. struct block \*\*current=&buddy\_system.free\_list[order];  *//current here means address*
12. struct block \*prev=0;
13. while (\*current){
14. if(\*current==buddy\_block&&buddy\_block->order==order){
15. *//find the buddy\_block and remove it*
16. if(prev){
17. prev->next=buddy\_block->next;
18. }else {
19. buddy\_system.free\_list[order]=buddy\_block->next;
20. }
21. buddy\_system.free\_list[order]=free\_block->next;
22. *//update*
23. free\_block =(struct block\*)(free\_addr<buddy\_addr?free\_addr:buddy\_addr);
24. order++;
25. merge=1;
26. break;
27. }
28. prev=\*current;
29. current=&(\*current)->next;
30. }
31. }
32. *//case 2: buddy\_block is behind the free\_block*
33. if(!merge){
34. buddy\_addr=free\_addr+(1<<(order+4));
35. if((char\*)buddy\_addr<=buddy\_system.base+BUDDY\_HEAP\_SIZE){
36. struct block \*buddy\_block =(struct block\*)buddy\_addr;
37. *//search for the buddy block in the free list*
38. struct block \*\*current=&buddy\_system.free\_list[order];
39. struct block \*prev=0;
40. while (\*current){
41. if(\*current==buddy\_block&&buddy\_block->order==order){
42. *//find the buddy\_block and remove it*
43. if(prev){
44. prev->next=buddy\_block->next;
45. }else {
46. buddy\_system.free\_list[order]=buddy\_block->next;
47. }
48. buddy\_system.free\_list[order]=free\_block->next;
49. *//update*
50. free\_block =(struct block\*)(free\_addr<buddy\_addr?free\_addr:buddy\_addr);
51. order++;
52. merge=1;
53. break;
54. }
55. prev=\*current;
56. current=&(\*current)->next;
57. }
58. }
59. }
60. if(merge){ *//add the bigger free block to the free list*
61. free\_block->next=buddy\_system.free\_list[order];
62. free\_block->order=order;  *//renew the order*
63. buddy\_system.free\_list[order]=free\_block;
64. }
65. if(merge){
66. merge\_block(free\_block,order);  *//continue check\_recursion*
67. }
68. }
69. void
70. buddy\_free(void \*ptr){
71. struct block \*free\_block =(struct block\*)((char\*)ptr-sizeof(struct block));
72. int order =free\_block->order;
73. free\_block->next=buddy\_system.free\_list[order];  *//put into the free\_list according to the order*
74. buddy\_system.free\_list[order]=free\_block;
75. merge\_block(free\_block,order);
76. }
77. 利用show()函数输出空闲链表的状态
78. void show(){
79. printf("the status of the heap:\n");
80. for (int i=0;i<=MAX\_ORDER;i++){
81. struct block \*current =buddy\_system.free\_list[i];
82. printf("Order %d :\n",i);
83. while (current){
84. printf("block address:%p\n",current);
85. current=current->next;
86. }
87. }}

通过show()函数在终端中打印空闲链表的信息，以此来验证内存的分配和合并结果的正确性。

6、编写buddy\_test系统调用，对buddy system的实现进行测试

1. int
2. test\_malloc(void){
3. printf("The initial state of heap:\n");
4. show();
5. printf("try to allocate a 16MB+2B area \n");
6. char \*pointer0=(char \*)buddy\_malloc(16\*1024\*1024+2);
7. if(pointer0){
8. buddy\_free(pointer0);
9. printf("free pointer0\n");
10. show();
11. }
12. char \*pointer1=(char \*)buddy\_malloc(100);
13. printf("allocate 100B at %p\n",pointer1);
14. show();
15. char \*pointer2=(char \*)buddy\_malloc(100000);
16. printf("allocate 100000B at %p\n",pointer2);
17. show();
18. char \*pointer3=(char \*)buddy\_malloc(2000000);
19. printf("allocate 2000000B at %p\n",pointer3);
20. show();
21. if(pointer1)
22. {buddy\_free(pointer1);
23. printf("free pointer1\n");
24. show();
25. }
26. if(pointer2){
27. buddy\_free(pointer2);
28. printf("free pointer2\n");
29. show();
30. }else{
31. printf("p2 not exist");
32. }
33. if(pointer3){
34. buddy\_free(pointer3);
35. printf("free pointer3\n");
36. show();
37. }else{
38. printf("p3 not exist");
39. }
40. show();
41. return 23;
42. }

首先，我尝试分配一个16MB+2B大小的内存，这个size大于堆的最大容量，用于测试系统能否对特殊的不满足要求的情况给出正确的反应。接着，我依次对内存进行100B、10000B、200000B的分配，并查看分配后空闲链表的状态，最后对分配的内存块进行回收以及合并。

## 3.4 实验结果

终端输出结果如下：(省去不必要的堆的打印信息)

1. $ buddy\_test
2. The initial state of heap:
3. the status of the heap:
4. //......
5. Order 20 :
6. block address:0x0000000087000000
7. try to allocate a 16MB+2B area
8. try to malloc the area
9. fail to alloctae the area.
10. try to malloc the area
11. allocate 100B at 0x0000000087000010
12. the status of the heap:
13. Order 0 :
14. Order 1 :
15. Order 2 :
16. Order 3 :
17. block address:0x0000000087000080
18. Order 4 :
19. block address:0x0000000087000100
20. Order 5 :
21. block address:0x0000000087000200
22. Order 6 :
23. block address:0x0000000087000400
24. Order 7 :
25. block address:0x0000000087000800
26. Order 8 :
27. block address:0x0000000087001000
28. Order 9 :
29. block address:0x0000000087002000
30. Order 10 :
31. block address:0x0000000087004000
32. Order 11 :
33. block address:0x0000000087008000
34. Order 12 :
35. block address:0x0000000087010000
36. Order 13 :
37. block address:0x0000000087020000
38. Order 14 :
39. block address:0x0000000087040000
40. Order 15 :
41. block address:0x0000000087080000
42. Order 16 :
43. block address:0x0000000087100000
44. Order 17 :
45. block address:0x0000000087200000
46. Order 18 :
47. block address:0x0000000087400000
48. Order 19 :
49. block address:0x0000000087800000
50. Order 20 :
51. try to malloc the area
52. allocate 100000B at 0x0000000087020010
53. the status of the heap:
54. Order 0 :
55. Order 1 :
56. Order 2 :
57. Order 3 :
58. block address:0x0000000087000080
59. Order 4 :
60. block address:0x0000000087000100
61. Order 5 :
62. block address:0x0000000087000200
63. Order 6 :
64. block address:0x0000000087000400
65. Order 7 :
66. block address:0x0000000087000800
67. Order 8 :
68. block address:0x0000000087001000
69. Order 16 :
70. block address:0x0000000087100000
71. Order 17 :
72. block address:0x0000000087200000
73. Order 18 :
74. block address:0x0000000087400000
75. Order 19 :
76. block address:0x0000000087800000
77. Order 20 :
78. try to malloc the area
79. allocate 2000000B at 0x0000000087200010
80. the status of the heap:
81. Order 0 :
82. Order 1 :
83. Order 2 :
84. Order 3 :
85. block address:0x0000000087000080
86. Order 4 :
87. block address:0x0000000087000100
88. Order 5 :
89. block address:0x0000000087000200
90. Order 6 :
91. block address:0x0000000087000400
92. Order 7 :
93. block address:0x0000000087000800
94. Order 8 :
95. block address:0x0000000087001000
96. Order 9 :
97. block address:0x0000000087002000
98. Order 10 :
99. block address:0x0000000087004000
100. Order 11 :
101. block address:0x0000000087008000
102. Order 12 :
103. block address:0x0000000087010000
104. Order 13 :
105. Order 14 :
106. block address:0x0000000087040000
107. Order 15 :
108. block address:0x0000000087080000
109. Order 16 :
110. block address:0x0000000087100000
111. Order 17 :
112. Order 18 :
113. block address:0x0000000087400000
114. Order 19 :
115. block address:0x0000000087800000
116. Order 20 :
117. free pointer1
118. the status of the heap:
119. //......
120. block address:0x0000000087000000
121. Order 14 :
122. block address:0x0000000087040000
123. Order 15 :
124. block address:0x0000000087080000
125. Order 16 :
126. block address:0x0000000087100000
127. Order 17 :
128. Order 18 :
129. block address:0x0000000087400000
130. Order 19 :
131. block address:0x0000000087800000
132. Order 20 :
133. free pointer2
134. the status of the heap:
135. //......
136. Order 17 :
137. block address:0x0000000087000000
138. Order 18 :
139. block address:0x0000000087400000
140. Order 19 :
141. block address:0x0000000087800000
142. Order 20 :
143. free pointer3
144. the status of the heap:
145. //......
146. block address:0x0000000087000000

从运行结果来看，当分配的块过大或者在空闲链表中没有找到足够大的空闲块时，终端会输出无法分配的信息。在正常的分配和合并中，buddy system能够满足需求。

**补充部分：**

在验收的过程中，为了方便进行实时的操作，我采用在命令行直接进行交互的方式。通过调用xv6系统中consoleread(int user\_dst, uint64 dst, int n)函数，从控制台中读取输入并将其写入用户空间缓冲区。以下是修改后的test\_malloc函数，定义一个buf缓冲区用于接收控制台的输入，根据控制台的输入选择对应的功能并调用相应的函数。

1. int
2. test\_malloc(void){
3. char buf[100];
4. int flag=1;
5. while(flag){
6. printf("Choose an option:\n");
7. printf("1 Allocate Memory\n");
8. printf("2 Free Memory\n");
9. printf("3 Quit\n");
10. printf("4 show the heap\n");
11. printf("please enter your choice:\n");
12. consoleread(0,(uint64)buf,sizeof(buf));
13. int choice=atoi(buf);
14. switch(choice){
15. case 1:
16. allocate\_memory(); //调用分配内存的函数
17. break;
18. case 2:
19. free\_memory(); //释放内存
20. break;
21. case 3:
22. flag=0; //退出程序
23. break;
24. case 4:
25. show(); //显示现在堆的状态
26. break;
27. default:
28. printf("Invalid choice.\n"); //非法输入
29. break;
30. }}

allocate\_memory(void)函数的定义如下：

1. void
2. allocate\_memory(void){
3. char buf[100]; //创建一个缓冲区终于接收用户在控制台的输入字符
4. printf("please enter the size you want to allocate:\n");
5. consoleread(0,(uint64)buf,sizeof(buf)); //从控制台读取用户输入并存储在buf缓冲区中
6. int size=atoi(buf);
7. printf("please enter the id for the block\n");
8. char buf0[100];
9. consoleread(0,(uint64)buf0,sizeof(buf0));
10. int id=atoi(buf0);
11. printf("try to allocate a %p area,id is %d \n",size,id);
12. char \*pointer=buddy\_malloc(size);
13. if(pointer){
14. int i;
15. for( i=0;i<100;i++){
16. if(used\_list[i].used\_block==0){
17. used\_list[i].used\_block = (struct block \*)pointer; *// Assuming struct block \*used\_block points to the allocated block*
18. used\_list[i].id = id;
19. break;
20. }
21. }
22. if(i==100){
23. printf("ERROR:used\_list is full\n");
24. }
25. }else{
26. printf("ERROR:allocate failed.");
27. }
28. show();
29. }

free\_memory(void)函数的定义如下：

1. void
2. free\_memory(void){
3. char buf[100];
4. printf("please enter the num of block you want to free:\n");
5. consoleread(0,(uint64)buf,sizeof(buf));
6. int id=atoi(buf);
7. *// Find the block in used\_list and free it*
8. int i;
9. for (i = 0; i < 100; i++) {
10. if (used\_list[i].used\_block != 0 && used\_list[i].id == id) {
11. struct block \*ptr = used\_list[i].used\_block; *// Assuming used\_block is a pointer to the allocated block*
12. buddy\_free(ptr); *// Free the memory block*
13. used\_list[i].used\_block = 0; *// Clear the entry in used\_list*
14. used\_list[i].id = 0; *// Reset the id*
15. printf("Block with id %d freed successfully.\n", id);
16. break;
17. }
18. }
19. if (i == 100) {
20. printf("Error: Block with id %d not found.\n", id);
21. }
22. show();
23. }

通过这两个函数，我实现了能够直接在控制台随时分配固定一定大小的块并给分配后的块赋予id号，在释放内存块时，通过给定id号，就能够释放指定块的内存。

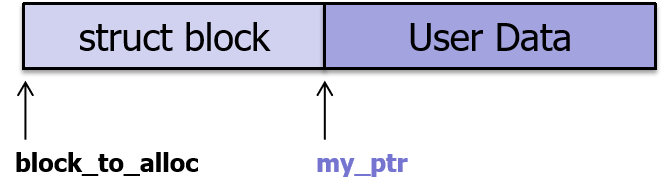
控制台的效果如下：（省去不必要的堆输出信息）

1. init: starting sh
2. $ buddy\_test
3. Choose an option:
4. 1 Allocate Memory
5. 2 Free Memory
6. 3 Quit
7. 4 show the heap
8. please enter your choice:
9. 1
10. please enter the size you want to allocate:
11. 10000
12. please enter the id for the block
13. 1
14. try to allocate a 0x0000000000002710 area,id is 1
15. try to malloc the area
16. the status of the heap:
17. //。。。。。。
18. Order 10 :
19. block address:0x0000000087004000
20. Order 11 :
21. block address:0x0000000087008000
22. Order 12 :
23. block address:0x0000000087010000
24. Order 13 :
25. block address:0x0000000087020000
26. Order 14 :
27. block address:0x0000000087040000
28. Order 15 :
29. block address:0x0000000087080000
30. Order 16 :
31. block address:0x0000000087100000
32. Order 17 :
33. block address:0x0000000087200000
34. Order 18 :
35. block address:0x0000000087400000
36. Order 19 :
37. block address:0x0000000087800000
38. Order 20 :
39. Choose an option:
40. 1 Allocate Memory
41. 2 Free Memory
42. 3 Quit
43. 4 show the heap
44. please enter your choice:
45. 2
46. please enter the num of block you want to free:
47. 0
48. Error: Block with id 0 not found.
49. the status of the heap:
50. //......
51. Order 10 :
52. block address:0x0000000087004000
53. Order 11 :
54. block address:0x0000000087008000
55. Order 12 :
56. block address:0x0000000087010000
57. Order 13 :
58. block address:0x0000000087020000
59. Order 14 :
60. block address:0x0000000087040000
61. Order 15 :
62. block address:0x0000000087080000
63. Order 16 :
64. block address:0x0000000087100000
65. Order 17 :
66. block address:0x0000000087200000
67. Order 18 :
68. block address:0x0000000087400000
69. Order 19 :
70. block address:0x0000000087800000
71. Order 20 :
72. Choose an option:
73. 1 Allocate Memory
74. 2 Free Memory
75. 3 Quit
76. 4 show the heap
77. please enter your choice:
78. 1
79. please enter the size you want to allocate:
80. 3
81. please enter the id for the block
82. 2
83. try to allocate a 0x0000000000000003 area,id is 2
84. try to malloc the area
85. the status of the heap:
86. Order 0 :
87. Order 1 :
88. block address:0x0000000087004020
89. Order 2 :
90. block address:0x0000000087004040
91. Order 3 :
92. block address:0x0000000087004080
93. Order 4 :
94. block address:0x0000000087004100
95. Order 5 :
96. block address:0x0000000087004200
97. Order 6 :
98. block address:0x0000000087004400
99. Order 7 :
100. block address:0x0000000087004800
101. Order 8 :
102. block address:0x0000000087005000
103. Order 9 :
104. block address:0x0000000087006000
105. Order 10 :
106. Order 11 :
107. block address:0x0000000087008000
108. Order 12 :
109. block address:0x0000000087010000
110. Order 13 :
111. block address:0x0000000087020000
112. Order 14 :
113. block address:0x0000000087040000
114. Order 15 :
115. block address:0x0000000087080000
116. Order 16 :
117. block address:0x0000000087100000
118. Order 17 :
119. block address:0x0000000087200000
120. Order 18 :
121. block address:0x0000000087400000
122. Order 19 :
123. block address:0x0000000087800000
124. Order 20 :
125. Choose an option:
126. 1 Allocate Memory
127. 2 Free Memory
128. 3 Quit
129. 4 show the heap
130. please enter your choice:
131. 2
132. please enter the num of block you want to free:
133. 1
134. Block with id 1 freed successfully.
135. the status of the heap:
136. Order 0 :
137. Order 1 :
138. block address:0x0000000087004020
139. Order 2 :
140. block address:0x0000000087004040
141. Order 3 :
142. block address:0x0000000087004080
143. Order 4 :
144. block address:0x0000000087004100
145. Order 5 :
146. block address:0x0000000087004200
147. Order 6 :
148. block address:0x0000000087004400
149. Order 7 :
150. block address:0x0000000087004800
151. Order 8 :
152. block address:0x0000000087005000
153. Order 9 :
154. block address:0x0000000087006000
155. Order 10 :
156. block address:0x0000000087000000
157. Order 11 :
158. block address:0x0000000087008000
159. Order 12 :
160. block address:0x0000000087010000
161. Order 13 :
162. block address:0x0000000087020000
163. Order 14 :
164. block address:0x0000000087040000
165. Order 15 :
166. block address:0x0000000087080000
167. Order 16 :
168. block address:0x0000000087100000
169. Order 17 :
170. block address:0x0000000087200000
171. Order 18 :
172. block address:0x0000000087400000
173. Order 19 :
174. block address:0x0000000087800000
175. Order 20 :
176. Choose an option:
177. 1 Allocate Memory
178. 2 Free Memory
179. 3 Quit
180. 4 show the heap
181. please enter your choice:
182. 2
183. please enter the num of block you want to free:
184. 2
185. Block with id 2 freed successfully.
186. the status of the heap:
187. //......
188. Order 20 :
189. block address:0x0000000087000000
190. Choose an option:
191. 1 Allocate Memory
192. 2 Free Memory
193. 3 Quit
194. 4 show the heap
195. please enter your choice:
196. 3//退出程序

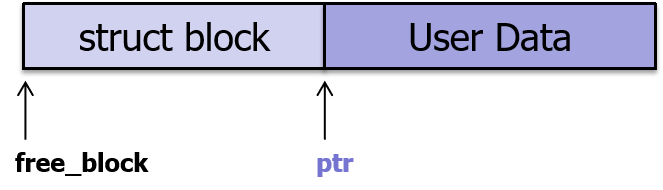
## 3.5 实验中遇到的问题及解决方案

1、在分配内存和回收内存时计算大小需要注意block结构体的大小。

在分配内存时，我们所要分配的内存大小不仅仅是参数传入的size,还必须包括结构体block的大小。在buddy system中，每个内存块通常会包含一些元数据（metadata），如块的大小、状态等。这些元数据通常存储在一个结构体 struct block中。在内存分配时，需要为这部分元数据预留空间。因此，计算实际需要的内存大小时要加上struct block 的大小。



分配的内存大小时，需要加上 struct block 的大小。最终返回的指针指向的是my\_ptr指针的位置，也就是直接指向用户数据部分。



当释放内存时，传入的指针时ptr，指向的是用户的数据部分，但是我们要释放的完整的内存还包括了全面的struct block部分，因此在计算free\_block对应的起始地址时，我们应该在指针所指的地址基础上减去struct block的大小。

2、在释放内存时，对块进行合并要采用递归的方式。

在每一次释放内存之后，我们都要检查空闲链表free\_list中是否存在能够被合并的空闲块，如果有符合的块则进行合并。如果在本次查找伙伴块的过程中发生了合并，那么这时就要采用递归的方式，继续检查在合并之后，空闲链表中是否还存在能够合并的伙伴块，直到没有能够合并的块为止。这样才能够保证堆的内存得到正确的管理。

3、在通过命令行进行直接交互的实现中，不能直接用到c语言库中的atoi函数。

在xv6系统中无法直接使用c语言库中的函数，因此要将键盘输入的字符串转换成整数，需要自己定义实现atoi函数，从而实现基本的字符串解析和整数转换功能。

# 4 实验总结

本次操作系统内核实验旨在通过实际动手操作，深入理解操作系统内核的基本原理和实现。通过对 xv6 操作系统内核结构的实验，实验过程中我认真了解了操作系统的基本组成部分，包括进程管理、内存管理、文件系统和设备驱动等。同时，通过实践学习在内核空间编程的技巧，了解内核与用户空间编程的差异，并掌握内存管理、进程调度和中断处理等方面的编程技巧。

在内核实验的过程中，我主要完成了两项实验任务：系统调用和内存管理。在系统调用实验过程中，我在xv6操作系统引入了一个新的系统调用，用于返回系统中当前进程的数量。在测试函数中，结果能够准确地返回当前系统中的进程数量；在内存管理实验中，通过学习系统默认的kalloc()和kfree()函数，我了解到了xv6中基本的内存管理方法，并借助伙伴系统的思想编写了buddy\_alloc()和buddy\_free()接口函数并对16MB大小的内存进行管理，在测试用例中，我验证了内存分配器在不同情况下的行为，并验证函数的有效性和正确性。通过实现伙伴系统的内存分配算法，我成功在xv6中创建了新的内存管理机制，并且在终端实现了实时交互。

通过本次实验，我深入理解了操作系统内核的基本结构和实现细节，掌握了进程管理和内存管理的核心原理。在实验过程中，我学会了在虚拟机环境下定位和解决问题，熟悉了常用的内核调试工具和方法。此外，通过实现和测试内存分配算法，加深了我对内存管理策略的理解和认识。本次实验不仅锻炼了我的编程能力，还提升了我在复杂系统中分析和解决问题的能力，为未来深入研究操作系统打下了坚实的基础。在查找各种xv6系统相关资料的过程中，我学习了操作系统内核的各个模块及其交互方式。这不仅加深了我对操作系统多任务处理的理解，也使我更加熟悉了相关的数据结构和算法。

在实验过程中，我还学会了使用多种内核调试工具，如GDB、QEMU等。这些工具不仅帮助我定位和解决了实验中的各种问题，还提高了我在实际开发中进行内核调试的能力。例如：通过QEMU，我可以在虚拟环境中测试不同的内核配置和功能，实现了实验结果的快速迭代和验证。

此外，实验还让我体验了操作系统开发中的一些实际挑战和复杂性。例如，在实现内存管理模块时，我遇到了内存泄漏和碎片化的问题。通过查阅文献和反复调试，我最终找到了合适的解决方案。这一过程不仅提升了我的问题解决能力，也让我深刻认识到细致和耐心在操作系统开发中的重要性。

总之，通过这次实验，我不仅加深了对操作系统原理的理解，掌握了关键的技术技能，还提升了在复杂系统中分析和解决问题的能力。为未来深入研究操作系统及相关领域打下了坚实的基础。这次实验经历将成为我职业生涯中的宝贵财富，激励我不断探索和进步。

# 5 参考文献

1. xv6 book 2022-riscv
2. [xv6实验课程--系统调用\_xv6添加系统调用-CSDN博客](https://blog.csdn.net/lhwhit/article/details/116482806)
3. [手把手教你使用 VS Code 编译和调试 C/C++ 工程，这下都明白啦！ - 知乎 (zhihu.com)](https://zhuanlan.zhihu.com/p/678594027)

教师评语评分

评语：

评分：

评阅人：

年 月 日

（备注：对该实验报告给予优点和不足的评价，并给出百分之评分。）