

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 ： 刘华俊

学 生 学 号 ： 2022302031031

学 生 姓 名 ： 张锐

学 年 学 期 ： 2023-2024 学年第三学期

完 成 时 间 ： 2024 年 7 月 4 日

二〇二四年七月

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：____张锐____

日期：____2024. 7. 11____

目录

摘 要	5
1 实验背景与目的	7
1.1 实验背景	7
1.2 实验目的	8
2 xv6 操作系统环境搭建与调试	8
2.1 实验工具	8
2.2 实验要求	8
2.3 环境搭建过程	8
2.3.1 准备 Linux 环境	8
2.3.2 下载 xv6 源代码	8
2.3.3 安装工具链	8
2.4 实验结果	9
2.4.1 通过控制台与之交互	9
2.5 实验总结	10
3 xv6 新增系统调用	10
3.1 实验要求	10
3.2 xv6 系统调用原理分析	10
3.3 为 xv6 新增系统调用	11
3.3.1 实现系统调用函数	12
3.3.2 分配系统调用号	13
3.3.3 更新系统调用调度程序	14
3.3.4 定义用户态中系统调用接口	14
3.3.5 用户空间库	14
3.3.6 测试系统调用	15
3.4 用户函数与内核函数的对应关系	15

3.5 实验总结	16
4 xv6 内核内存管理	17
4.1 利用首次适应算法实现动态内存分配	17
4.1.1 动态内存分配需要实现的功能	17
4.1.2 首次适应算法简介	17
4.2 xv6 内核内存管理机制分析	18
4.2.1 总体概述	18
4.2.2 相关函数分析	18
4.3 实验步骤	18
4.3.1 开辟堆空间	18
4.3.2 宏定义	19
4.3.3 初始化	19
4.3.4 内存的释放与空闲块的合并	20
4.3.5 内存的分配	23
4.3.6 测试结果	24
4.4 伙伴算法实现动态内存分配	30
4.4.1 伙伴算法简介	30
4.4.2 伙伴算法的实现	31
4.5 两种算法的对比总结	34
4.6 实验总结	34
参考资料	35
教师评语评分	36

摘 要

本次实验以 Linux 系统为运行环境，QEMU 为运行框架，xv6 操作系统为核心代码，完成了对 xv6 操作系统的整体学习以及架构的认识，同时实现了返回进程数量的系统调用，和分别用首次适应算法和伙伴算法实现动态内存分配功能。

操作系统内核实验主要内容包括三部分，分别为：xv6 操作系统环境搭建和调试实验、系统调用实验、内存管理实验。

在 xv6 操作系统环境搭建和调试实验中，基于 Vmware Ubuntu 系统搭建开发平台，完成 gcc、gdb、Qemu 的安装，实现 Xv6 的编译调试，并利用 VSCode 搭建集成开发环境。

在系统调用实验中，在 xv6 操作系统中引入一个新的系统调用，该调用遍历 PCB 并打印当前表中进程数。用户态方面，修改了 user.h, usys.pl 等文件以新增 API，创建 getproc.c 文件以测试调用 API；内核态方面，新增 SYS_getprocs 系统调用编号 22，在 syscall.c 文件中将系统调用绑定到 sys_getprocs 函数，并在 proc.c 文件中实现 getprocs () 函数，经测试正确输出进程信息。除此之外，我还深入研究了系统调用中用户函数与内核函数的匹配过程，建立了一个完整的用户——内核系统调用的认知体系

在内存管理实验中，分别用首次适应算法和伙伴系统的算法实现了动态内存分配的功能，用于分配和释放不同大小的内存块，并且保证了代码的可靠性。两

种方法都使用了链表，链表直接指向空闲内存区。配合 `minit`、`mymalloc`、`myfree` 等函数对堆管理，通过新增系统调用提供测试手段，验证堆相关功能。最后，我还对比了两个算法的利弊，并分析了伙伴系统能够在如今的操作系统中得到广泛运用的原因。

最后，我总结了本次实验的历程以及遇到的困难，并在附录附上了本次实验的核心代码。

关键词： xv6；系统调用；首次适应算法；伙伴系统；

1 实验背景与目的

1.1 实验背景

操作系统作为计算机系统的核心软件，在信息技术迅猛发展的今天扮演这至关重要的角色。随着硬件技术的进步和应用需求的多样化，操作系统不断演变，成为现代计算机环境中不可或缺的组成部分。目前，操作系统的生态主要有几大主流的操作系统组成，包括微软的 Windows、苹果的 macOS 和 Linux，以及移动设备上广泛使用多个 Android 和 iOS。这些操作系统各具特色，满足了不同用户群体和应用场景的需求。

深入操作系统的定义，操作系统（Operating System，简称 OS）是管理计算机硬件和软件资源的系统软件，它为计算机程序提供了通用服务。操作系统的主要功能包括：

- 进程管理：创建、调度和终止进程，提供进程间通信和同步机制。
- 内存管理：管理系统内存，提供内存分配和回收，支持虚拟内存。
- 文件系统管理：管理文件和目录，提供文件的存储、检索和更新功能。
- 设备管理：控制和管理输入输出设备，如硬盘、键盘、显示器等。
- 用户界面：提供用户与计算机交互的界面，包括命令行界面和图形用户界面。

为了更深入地理解操作系统的设计和实现，许多计算机科学课程和研究项目都会选择使用教学操作系统来进行实践。这里就引入了本实验所使用到的 xv6 操作系统。xv6 是一个现代化的教学操作系统，基于 UNIX V6，旨在帮助学生和研究人员学习和理解操作系统的基本原理。xv6 采用 C 语言编写，结构简单但功能完整，包含了进程管理、内存管理、文件系统、设备驱动等核心组件。通过对 xv6 的源码进行分析和实验，学习者可以直观地看到操作系统各个模块的实现细节，并在此基础上进行扩展和优化。在当前计算机科学的前沿，操作系统的研究和开发面临着新的挑战 and 机遇。例如，云计算和虚拟化技术的兴起使得资源管理和隔离技术变得尤为重要；多核处理器和并行计算的普及促使操作系统需要更加高效的调度机制；物联网设备的广泛应用也对操作系统的轻量化和实时性提出了更高的要求。

1.2 实验目的

本次实验我选择了操作系统的内核实验，目的是增加一个系统调用更深刻的了解 xv6 操作系统的框架，以及内核内存管理的任务。

2 xv6 操作系统环境搭建与调试

2.1 实验工具

本实验在 linux 系统上进行，工具有 VSCode、Git、GDB、QEMU 以及 xv6 源码。其中 xv6 介绍如下：xv6 是一款由 MIT 基于 UNIX 第六版开发，用于教学和研究的 UNIX-like 操作系统。其设计遵循 UNIX 操作系统的传统，包括类 UNIX 的系统调用接口、文件系统结构、进程管理和内存管理。同时，xv6 的源代码是公开的，可以在教学和学术研究中自由使用和修改，其简单的设计和精简的实现允许学生和研究者快速了解和上手操作系统的设计和组成。

2.2 实验要求

实验要求在 x86 架构的机器上构建基于 RISC-V 的 xv6 实验环境，安装并使用 RISC-V 编译工具链，使用 RISC-V 版本的 QEMU 对 xv6 源代码进行编译，要求在 QEMU 中成功运行 xv6，能够单步调试，并简单修改源码，能在合适的地方输出字符串。

2.3 环境搭建过程

2.3.1 准备 Linux 环境

在 VMware 虚拟机上安装 Ubuntu iso 文件即可。

2.3.2 下载 xv6 源代码

git clone <https://github.com/mit-pdos/xv6-riscv.git> 即可，本文直接用群中已有压缩包 xv6-riscv-riscv 中的代码

2.3.3 安装工具链

从官网中可以了解输入以下命令：

```
sudo apt-get install git build-essential gdb-multiarch qemu-system-misc gcc-riscv64-linux-gnu binutils-riscv64-linux-gnu 即可分别安装所有的包
```

出现下图说明成功编译了 xv6(忽略“初始化…最小分配单元”)


```
make: *** No rule to make target 'qemu'. Stop.
morri@morri-virtual-machine:~/xv6-labs-2023$ make qemu
qemu-system-riscv64 -machine virt -bios none -kernel kernel/kernel -m 128M -
global virtio-mmio.force-legacy=false -drive file=fs.img,if=none,format=raw,
io-blk-device,drive=x0,bus=virtio-mmio-bus.0

xv6 kernel is booting

-----
>>>>begin first and next fit malloc init
total size is 8388608 B
-----
hart 2 starting
hart 1 starting
init: starting sh
$
exec failed
$
```

图 2.1 make qemu 结果

2.4 实验结果

2.4.1 通过控制台与之交互

输入 `ls` 时其输出系统调用列表，结果如下：

```
-----
>>>>begin first and next fit malloc init
total size is 8388608 B
-----
hart 2 starting
hart 1 starting
init: starting sh
$ ls
.          1 1 1024
..         1 1 1024
README    2 2 2305
xargstest.sh 2 3 93
cat        2 4 32424
echo       2 5 31296
forktest   2 6 15416
grep        2 7 35776
init        2 8 32088
kill        2 9 31296
ln          2 10 31224
ls          2 11 34336
mkdir       2 12 31344
rm          2 13 31328
sh          2 14 53568
stressfs    2 15 32192
usertests   2 16 180776
grind       2 17 47400
wc          2 18 33424
zombie      2 19 30864
getprocs    2 20 31128
memTest     2 21 30976
console     3 22 0
$
```

图 2.2 交互结果展示

2.5 实验总结

在第一个实验中，成功地搭建了基于 RISC-V 架构的 xv6 操作系统运行环境，使用 QEMU 模拟器并配置 VScode 进行了调试，最后实现了通过控制台与之交互，加深了对 xv6 架构的了解，并提升了配置环境、调试代码等实践能力。

3 xv6 新增系统调用

3.1 实验要求

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，该调用返回系统中当前进程的数量。

3.2 xv6 系统调用原理分析

本实验通过分析 xv6 中已有的系统调用来了解 xv6 系统调用执行过程及原理。主要是通过增加系统调用，返回当前系统内所有状态的进程数量。

系统中的进程分为以下六种状态：UNUSED, USED, SLEEPING, RUNNING, RNNABLE, ZOMBIE。其中最重要的三个状态就是 SLEEPING、RUNNABLE 和 RUNNING，分别代表阻塞、就绪和运行状态。这 6 个状态被定义为一个枚举类型 `procstate`，编写在 `proc.h` 文件中。

同时，最重要的内容在 `proc.c` 文件中。`Proc.c` 文件中有着非常重要的一段代码

```
struct proc proc[NPROC];
```

这段代码定义了一个结构体数组，这个数组中给存放了当前系统内的进程。`NPROC=64`，代表了系统中最大进程数量 64。`Proc` 是一个进程结构体，相当于一个进程控制块 PCB。

有了这些基本的信息我对找了 `kill` 的系统调用，从而了解并实现了 `getprocs` 的系统调用

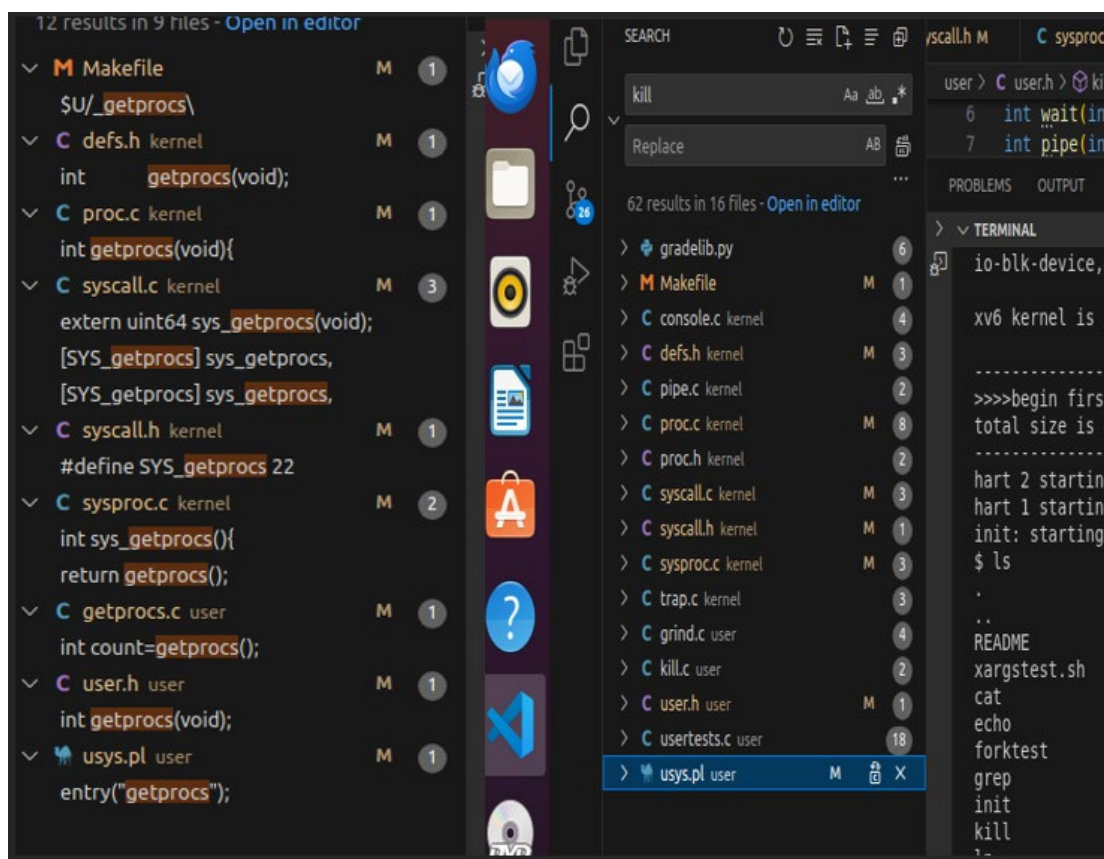


图 3.1 kill 与 getprocs 的对比

a) **用户态**，user/user.h 中声明了用户态中可调用的函数 xxx，允许用户使用该函数。通过 usys.pl 脚本根据该函数名生成 usys.S 中汇编代码 “xxx:li a7, SYS_xxx”，而用户调用时对应汇编为 call fork，就会跳转至上述汇编代码部分执行。该汇编代码实现了将 SYS_xxx 赋给 a7 寄存器，其中 SYS_xxx 是 Fork 的系统调用号，在 syscall.h 文件中定义，并在 syscall.c 文件中将其映射成所对应的系统调用函数。

b) **内核态**，kernel/syscall.h 定义了系统调用号。kernel/syscall.c 中声明了内核态下的所有系统调用，以及一个对应系统调用函数的数组，函数 syscall() 根据系统调用号用查数组的方式实现将系统调用号映射成其对应的系统调用函数并调用，将函数返回值存入用户进程的 a0 寄存器。

c) **整体流程**为用户调用函数 xxx，执行 user/usys.S 中对应汇编代码，控制转移到内核，根据 kernel/syscall.c 中 syscall() 执行 kernel/proc.c 的对应的系统调用函数，返回后将返回值存到 a0 寄存器，然后返回用户代码，系统调用结束。

3.3 为 xv6 新增系统调用

方便起见，实验三中所新增的系统调用过程在此一同展示。

3.3.1 实现系统调用函数

在/kernel/proc.c 中具体实现系统调用函数 getprocs（以及实验三中 memTest）：

在 getprocs 中主要是遍历一遍 proc 数组，如果不是 UNUSED 就定为活跃状态来统计总数量。

```
//added
int getprocs(void){
    struct proc *p;
    int cnt=NPROC;
    for(p=proc;p<&proc[NPROC];p++){
        acquire(&p->lock);
        if(p->state==UNUSED){
            cnt--;
        }
        release(&p->lock);
    }
    return cnt;
}
```

图 3.1 系统调用函数

```

C kalloc.c > memTest()
void memTest()
//printf("%d\n",mmem.freelist->size);
printf("#####\n");
printf(">>>>malloc and single free\n");

m1 = Mymalloc(1024*1024);
showFreeList();
free(m1);
showFreeList();
printf("\n");
printf("\n");
printf("\n");
printf("#####\n");
printf(">>>>incorrect malloc and incorrect free \n");
m1 = Mymalloc(15 * 1024*1024);
free(m1);
printf("\n");
printf("\n");
printf("\n");
printf("#####\n");
printf(">>>>malloc and merge in free\n");
m1 = Mymalloc(2 * 1024*1024);
m2 = Mymalloc(1 * 1024*1024);
free(m1);
showFreeList();
free(m2);
showFreeList();
printf("\n");
printf("\n");
printf("\n");
printf("#####\n");
printf(">>>>first fit malloc \n");
m1 = Mymalloc(1 * 1024);
m2 = Mymalloc(2 * 1024);
m3 = Mymalloc(3 * 1024);
free(m2);
showFreeList();

```

图 3.1 系统调用函数 2

3.3.2 分配系统调用号

在/kernel/syscall.h 中分配系统调用号：

```

#define SYS_getprocs 22
#define SYS_memTest 23

```

图 3.2 系统调用号

3.3.3 更新系统调用调度程序

在/kernel/syscall.c 中新增：

```
//added
[SYS_getprocs] sys_getprocs,
[SYS_memTest] sys_memTest,
```

图 3.3 系统调用映射

```
//added
extern uint64 sys_getprocs(void);
extern uint64 sys_memTest(void);
// extern uint64 sys_addproc(void);
```

图 3.4 系统调用函数声明

3.3.4 定义用户态中系统调用接口

在 /user/user.h 中声明函数 getprocs（以及实验三中 memTest）：

```
int getprocs(void);
void memTest(void);
```

图 3.5 用户态接口声明

这个函数名必须与下文中的 entry 中的名称所匹配。

3.3.5 用户空间库

在/user/usys.pl 中定义新系统调用的用户空间映射，系统将根据该脚本生成 usys.S 汇编代码：

```
#added
entry("getprocs");
entry("memTest");
```

图 3.6 脚本文件修改

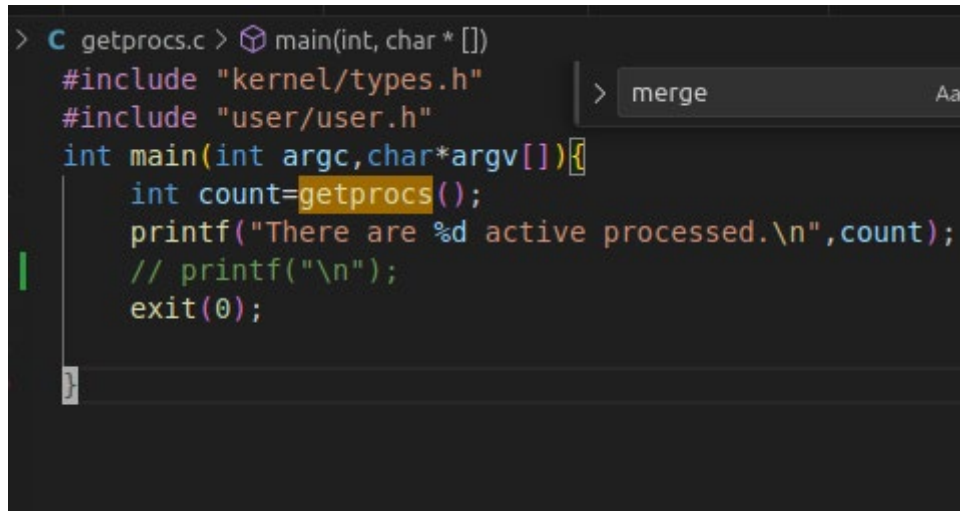
```
getprocs:
    li a7, SYS_getprocs
    ecall
    ret
.global memTest
memTest:
    li a7, SYS_memTest
    ecall
    ret
```

图 3.7 生成汇编结果

3.3.6 测试系统调用

为了测试系统调用编写对应的测试代码。

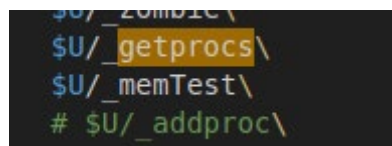
a) 创建文件 user/getprocs.c



```
> C getprocs.c > main(int, char * [])
#include "kernel/types.h"
#include "user/user.h"
int main(int argc, char*argv[]) {
    int count=getprocs();
    printf("There are %d active processed.\n", count);
    // printf("\n");
    exit(0);
}
```

图 3.8 测试代码

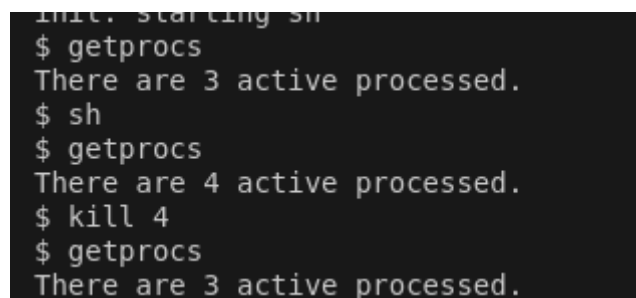
b) 修改 xv6 目录下的 Makefile 文件



```
$U/_zombie\
$U/_getprocs\
$U/_memTest\
# $U/_addproc\
```

图 3.9 Makefile 文件修改

c) 运行 xv6，在命令行中输入 getprocs 即可测试系统调用，结果如下：



```
init: starting sh
$ getprocs
There are 3 active processed.
$ sh
$ getprocs
There are 4 active processed.
$ kill 4
$ getprocs
There are 3 active processed.
```

图 3.10 系统调用结果

3.4 用户函数与内核函数的对应关系

其实这整个过程中最重要的是弄清楚用户态的函数是如何和内核台的函数

实现一一对应的。接下来我将会讲解其中的对应规则。

首先，在 MakeFile 中输入的 “\$U/_getprocs\” 的作用是给用户态增加一个可以执行的指令。添加后，getprocs 指令将会在输入 ls 指令后显示在可用的指令中。那么输入这个指令会执行哪里的代码？答案是 getprocs.c 这个文件的代码。可以发现，这个文件里有一个 main 函数，即代表了这个文件编译出来结果是一个可执行程序。因此该指令执行的就是其对应的.c 文件里面的代码。

执行这个程序，程序中就调用了那个声明的用户函数。这个时候又有疑问，user.h 中的声明函数来自哪里，其实就是 user.pl 里面的代码了。

```
sub entry {  
    my $name = shift;  
    print ".global $name\n";  
    print "${name}:\n";  
    print " li a7, SYS_${name}\n";  
    print " ecall\n";  
    print " ret\n";  
}
```

而 use.pl 里这段代码正式产生的汇编代码这也是区别系统调用与普通的调用最关键的部分，而 SYS_\${name}\n 以及其他 name 正和系统调用号对应，这样用户态的函数就对应到相应的系统调用中了，就会完成相应内核态的功能。

3.5 实验总结

在本次实验中，深入理解了 xv6 系统调用的工作流程和实现方法。通过阅读和修改 xv6 操作系统的源代码，了解了系统调用的整体框架，将操作系统理论运用于实际，亲身体会系统调用号的功用：程序为系统调用分配唯一的系统调用号，这是内核态与用户态进行交流的重要方式。

4 xv6 内核内存管理

4.1 利用首次适应算法实现动态内存分配

4.1.1 动态内存分配需要实现的功能

在 xv6 系统中，内核态的内存主要有 `kalloc.c` 和 `vm.c` 两个文件，其中 `kalloc.c` 实现了以页（4KB）为单位的动态内存分配和释放，`vm.c` 则实现了物理地址和虚拟地址之间的转换。

因为 `kalloc.c` 只能以单页为单位分配和释放内存，并不能实现变长的动态内存分配，我们的实验任务便是根据课上所学的操作系统理论知识，写一个能够实现动态内存分配和释放内存的内存管理机制，类似于 c 语言上的 `malloc` 和 `free` 的功能。

4.1.2 首次适应算法简介

首次适应算法 (First-Fit Algorithm) 是操作系统内存分配的一中经典算法，主要用于动态内存分配。他的基本思想是从内存的起始位置开始查找，找到第一个足够大的空闲块分配给请求的进程。以下是对首次适应算法的详细介绍。

首次适应算法：

1. 初始化：内存初始化，系统会维护一个空闲块链表，所有空闲内存块都在这个链表中。
2. 内存请求：当有进程请求分配内存时，系统从空闲链表的头开始查找，知道找到第一个足够大的空闲块。
3. 分配内存：如果找到的空闲块的大小恰好等于请求的大小，则将整个空闲块分配给进程，并从空闲链表中移除该块。如果找到的空闲块的大小大于请求的大小，则将请求的大小部分分配给进程，剩余部分作为新的空闲块，保存在空闲链表中。
4. 内存释放：当进程释放内存时，系统将释放的内存块插入空闲链表，并与相邻的空闲块合并，减少内存碎片。

首次适应算法的优点：

1. 首次适应算法实现简单，分配速度快，尤其是在内存空闲块较少的情况下效率高。

2. 只需要从头到尾扫描空闲链表，不需要复杂的数据结构。

首次适应算法的缺点：

1. 首次适应算法容易产生外部碎片。
2. 如果空闲链表比较长，且大量小块空闲内存，则可能需要较长的时间才能找到合适的块。

4.2 xv6 内核内存管理机制分析

4.2.1 总体概述

xv6 中的内存管理主要通过 `kernel/kalloc.c` 和 `kernel/vm.c` 实现，`memlayout.h` 中定义内存管理中相关的参数，内核在 `main.c` 文件中完成初始化，会调用 `vm.c` 中的 `kvminit` 和 `kalloc.h` 中的 `kinit` 来对内核的内存申请和页面分配初始化。通过 `vm.c` 中的页面表机制实现虚拟内存的管理，包括将虚拟地址映射到物理地址、处理页面错误、页面的分配和回收等。`kalloc.c` 是 xv6 内核的 `malloc` 和 `free` 函数的具体实现，用于获取单个页或释放页，其中页的大小为 4KB，地址范围为 `0x8000_0000` 到 `0x8800_0000`，相关定义在 `memlayout.h` 和 `riscv.h` 中。

4.2.2 相关函数分析

`kalloc.c` 主要组成部分为初始化函数 `kinit` 链表初始化函数 `freerange`，页申请函数 `kalloc` 与页释放函数 `kfree`，同时定义了结构体 `run` 和 `kmem` 用于管理页。`run` 作为链表节点结构体，`kmem` 定义了 `spinlock` 作为自旋锁，保证内存单元读写的独立性，定义了 `run` 指针 `freelist` 作为页表头用于管理空闲单元。`freerange` 函数遍历待初始化的内存地址，调用 `kfree` 函数对内存进行释放，`freerange` 的地址每次递增页大小来构造链表。`kinit` 函数调用 `initlock` 对 `spinlock` 进行初始化，再使用 `freerange` 进行页链表的构造和初始化。`kfree` 是释放页内存的函数，由于 `kalloc` 函数申请单位为页，因此 `kfree` 函数释放单位也为页，用头插法将空闲块插入到空闲链表。`kalloc` 函数每次从空闲链表头取出一页，进行分配。另外，由于 xv6 系统支持多线程，为了保证数据一致性，每次改变内存前需要申请锁，完成操作后释放。

4.3 实验步骤

4.3.1 开辟堆空间

通过修改 `kernel/memlayout.h` 中 `PHYSTOP` 值并新增 `HEAPSTART` 和 `HEAPSTOP` 表示开辟堆空间的起始和终止地址，实现从原有的 `kalloc` 机制中扣减 8MB

```

48
49 //added
50 #define PHYSTOP (KERNBASE + 120*1024*1024)
51 #define NEWSTOP (PHYSTOP + 8*1024*1024)//8MB
52
53

```

图 4.1 声明堆起始终止地址

修改/kernel/vm.c，在内核虚拟内存空间中映射刚开辟的内存空间，将物理地址和权限标志写入页表项中，完成映射。

```

//added
kvmmmap(kpgtbl,PHYSTOP,PHYSTOP,NEWSTOP-PHYSTOP,PTE_R | PTE_W);

```

图 4.2 修改映射

由于此时 PHYSTOP 仍表示需 kalloc 管理的内存空间的终止地址，因此 kalloc.c 中需用到该地址的 kfree 和 kinit 部分无需更改。以此实现了从原有的 kalloc 机制中扣减 8MB，并实现 kalloc 与我们的堆的隔离。

4.3.2 宏定义

首先在我对照着 kalloc 的形式定义了我的单链表宏定义，其中 mrun 表示某一空闲块的信息而 mmem 表示分配的空闲链。

```

struct mrun{
    struct mrun *next;
    uint size;
};

struct{
    struct spinlock lock;
    struct mrun *freelist;
}mmem;

```

图 4.3 空闲块和空闲链信息的宏定义

4.3.3 初始化

将所有内存初始化，并打印空闲链表的地址以及大小。和 kalloc 不同由于这个是空闲分区链，首次初始化时，只有一整块空闲链表，不需要分别对每一页都进行初始化。

```

void minit() { //only 1 block just need init 1 time
    printf("-----\n");
    printf(">>>>begin first and next fit malloc init\n");

    initlock(&mmem.lock, "mmem");
    uint size=NEWSTOP-PHYSTOP;
    memset((void*)PHYSTOP, 1, size);

    struct mrun *b;
    b=(struct mrun*)PHYSTOP;
    b->size=size;
    b->next=NULL;
    // printf("b size: %d\n", b->size);
    acquire(&mmem.lock);
    mmem.freelist=b;
    release(&mmem.lock);

    printf("total size is %d B\n", mmem.freelist->size);
    printf("-----\n");
}

```

图 4.4 内存的初始化

不仅如此我还写了一个 showFreeList 函数来更好地展示当前所有空闲块的首地址以及大小

```

void showFreeList(){
    printf("-----\n");
    printf(">>>>free list\n");
    struct mrun*current=mmem.freelist;
    while(current->size>0){
        printf("address:%p\n", current);
        printf("size:%dB\n", current->size);
        if(current->next==NULL) {
            break;
        }
        current=current->next;
    }
    printf("-----\n");
}

```

图 4.5 显示空闲链表

4.3.4 内存的释放与空闲块的合并

a) 合并内存块，由于所有空闲块都是单链表存储所以如果这内存的首地址加上他的大小等于下一个空闲内存块的首地址就合并，但是由于下下块也可能是空闲的所以需要 continue

```

void mergeFreeList(){
    struct mrun *b=mmem.freelist;
    printf("-----\n");
    while(b){
        if(((uint)b+b->size)==(uint)(b->next)){ //fang fa
            printf(">>>>merge\n");
            printf("address1:%p\n",b);
            printf("size1:%dB\n",b->size);
            printf("address2:%p\n",b->next);
            printf("size2:%dB\n",b->next->size);

            printf("sumsize:%dB\n", (b->size+b->next->size));
            b->size=b->size+b->next->size;
            b->next=b->next->next;
            continue; //he bing wan yi hou ke neng hai he xia yi kuai he bing
        }
        b=b->next;
    }
    printf("-----\n");
    printf("\n");
}

```

图 4.6 合并函数

b) 释放内存块首次适应算法需要按照地址大小对空闲内存块进行排序，所以需要比较内存首地址的大小。

然后就是分几种情况讨论，首先若要释放的内存为空则 free fail；若释放的内存超过了大小则 free fail；若释放的内存块的地址比所有空闲块的地址都大则直接插在后面；若释放的内存块的地址在空闲内存块地址的中间，则遍历空闲块插入这个空闲内存块。

```

void free(void *ptr){
    printf("-----\n");
    printf(">>>>begin free\n");

    if(ptr==NULL){
        printf("no address,free fail\n");
        return;
    }
    if((uint)ptr>NEWSTOP||(uint)ptr<PHYSTOP)
    {
        printf("address out of range,free fail\n");
        return;
    }

    struct mrun *b=(struct mrun *)ptr;
    printf("free address:%p\n",b);
    printf("free size:%dB\n",b->size);

    acquire(&mmem.lock);
    struct mrun *i=mmem.freelist;
    // first fit
    if((uint)i>(uint)b)
    {
        b->next=i;
        mmem.freelist=b;
        release(&mmem.lock);
        mergeFreeList();
        printf("free success!!!\n");
        printf("-----\n");
        return;
    }
}

```

图 4.7 释放函数 1

```

}
// int flag=0;
for(;i->next!=NULL;i=i->next){
    if((uint)i<(uint)b&&(uint)(i->next)>(uint)b)
    {
        b->next=i->next;
        i->next=b;

        // flag=1;

        mergeFreeList();
        release(&mmem.lock);
        printf("free success!!!\n");
        printf("-----\n");
        return;
    }
}
// if(!flag){
    i->next=b;
    b->next=NULL;
// }
mergeFreeList();
release(&mmem.lock);
printf("free success\n");
printf("-----\n");
printf("\n");
}

```

图 4.8 释放函数 2

4.3.5 内存的分配

遍历所有的空闲块，找到第一个符合大小的内存块即分配。需要注意的是当 size 的大小刚刚好等于需要分配的大小的时候是不能分配的，因为要给存储节点的指针留位置。而且在整个内存分配的过程中是不能创建变量的，只能用指针，因为没有人给他分配。

```

void *Mymalloc(uint size){
    struct mrun *b = mmem.freelist;
    struct mrun *pre;
    // printf("%d \n", mmem.freelist->size);
    printf("-----\n");
    printf(">>>begin malloc %dB\n",size);
    // printf("%d",b->size);
    // printf("111111");
    acquire(&mmem.lock);

    for(;b;b=b->next){
        // printf("1111\n");
        // printf("%p\n",&b);
        // printf("%d\n",b->size);
        if((b->size) >= size){
            // printf("2222222\n");
            if((b->next==NULL)&&( b->size == size)) //bu neng wan quan fen yao gei zhi zhen liu wei zhi
                break;
            printf("malloc address:%p\n",b);
            printf("free size is %dB\n",b->size);
            struct mrun *new=(struct mrun*)((uint)b+size);
            new->size=b->size-size;
            new->next=b->next;
            b->size=size;
            if(b==mmem.freelist){
                mmem.freelist=new;
            }else{
                pre->next=new; //
            }
            release(&mmem.lock);
            printf("malloc %dB success!!!\n",size);
            printf("-----\n");
            printf("%d\n",size);
            return b;
        }
    }
}

```

图 4.9 分配函数 1

```

58     printf("\n");
59     return b;
60 }
61 pre=b;
62
63 }
64 printf("malloc %dB failed\n",size);
65 release(&mmem.lock);
66 printf("-----\n");
67 return NULL;
68 }
69

```

图 4.10 分配函数 2

4.3.6 测试结果

a) 为方便调用所写函数，在 defs.h 文件中声明所写函数。


```

62 // kalloc.c
63 void*      kalloc(void);
64 void      kfree(void *);
65 void      kinit(void);
66
67 // log.c

```

图 4.11 声明所写函数

b) 在 main.c 中用所写 minit 函数初始化堆。注意这个 minit 只能在 kvminit 之前，因为内存的初始化要在内存映射之前。

```

kinit();      // physical page allocator

minit();      //added //zhi neng fang zai zhe li

kvminit();    // create kernel page table

```

图 4.12 在 main 函数中调用初始化函数

c) 按要求在内核中增加演示函数，连续执行一系列的堆分配，堆释放，为该函数分配系统调用号，在用户态定义系统调用接口，实现系统调用等步骤在实验二中已一并讲解过程。此处仅展示演示函数 memTest()

```

void memTest()
{
    struct mrun *m1, *m2, *m3, *m4;
    printf("\n");
    printf("\n");
    printf(">>>>begin memTest\n");
    //printf("%d\n",mmem.freelist->size);
    printf("#####\n");
    printf(">>>>malloc and single free\n");

    m1 = Mymalloc(1024*1024);
    showFreeList();
    free(m1);
    showFreeList();
    printf("\n");
    printf("\n");
    printf("\n");
    printf("#####\n");
    printf(">>>>incorrect malloc and incorrect free \n");
    m1 = Mymalloc(15 * 1024*1024);
    free(m1);
    printf("\n");
    printf("\n");
    printf("\n");
    printf("#####\n");
    printf(">>>>incorrect malloc and incorrect free\n");
}

```

图 4.13 内核中的演示函数 1

```

printf(">>>>malloc and merge in free\n");
m1 = Mymalloc(2 * 1024*1024);
m2 = Mymalloc(1 * 1024*1024);
free(m1);
showFreeList();
free(m2);
showFreeList();
printf("\n");
printf("\n");
printf("\n");
printf("#####\n");
printf(">>>>first fit malloc \n");
m1 = Mymalloc(1 * 1024);
m2 = Mymalloc(2 * 1024);
m3 = Mymalloc(3 * 1024);
free(m2);
showFreeList();
m4 = Mymalloc(1 * 1024);
showFreeList();
free(m1);
free(m4);
showFreeList();
printf("#####\n");
printf("\n");
exit(0);

```

图 4.14 内核中的演示函数 2

该演示函数先是执行了单个内存块的分配和释放，然后执行了不正确的内存分配分配了过大的内存块，然后进行了内存的分配和合并，然后进行了首次适应算法的展示。

```

>>>>begin memTest
#####
>>>>malloc and single free
-----
>>>>begin malloc 1048576B
malloc address:0x0000000087800000
free size is 8388608B
malloc 1048576B success!!!
-----

-----
>>>>free list
address:0x0000000087900000
size:7340032B
-----
-----
>>>>begin free
free address:0x0000000087800000
free size:1048576B
-----
>>>>merge
address1:0x0000000087800000
size1:1048576B
address2:0x0000000087900000
size2:7340032B
sumsize:8388608B
-----

free success!!!
-----
-----
>>>>free list
address:0x0000000087800000
size:8388608B
-----

```

图 4.15 测试结果 1

先进行了单个内存的分配和释放，并进行了释放后的两个内存块的合并

```

#####
>>>>incorrect malloc and incorrect free
-----
>>>>begin malloc 15728640B
malloc 15728640B failed
-----
-----
>>>>begin free
no address,free fail

```

图 4.16 测试结果 2

这个是不正确的内存分配与释放，由于没有足够的大空间则无法进行分配，也无法进行内存的释放。

```
#####
>>>>first fit malloc
-----
>>>>begin malloc 1024B
malloc address:0x0000000087800000
free size is 8388608B
malloc 1024B success!!!
-----

-----begin malloc 2048B
malloc address:0x0000000087800400
free size is 8387584B
malloc 2048B success!!!
-----

-----begin malloc 3072B
malloc address:0x0000000087800c00
free size is 8385536B
malloc 3072B success!!!
-----

-----begin free
free address:0x0000000087800400
free size:2048B
-----

free success!!!
-----

>>>>free list
address:0x0000000087800400
$ hex ailed
$ dross:0x0000000087801800
size:8382464B
-----
```

图 4.17 测试结果 3.1

```

>>>>begin malloc 1024B
malloc address:0x0000000087800400
free size is 2048B
malloc 1024B success!!!
-----

-----
>>>>free list
address:0x0000000087800800
size:1024B
address:0x0000000087801800
size:382464B
-----

>>>>begin free
free address:0x0000000087800000
free size:1024B
-----

free success!!!
-----

>>>>begin free
free address:0x0000000087800400
free size:1024B
-----

>>>>merge
address1:0x0000000087800000
size1:1024B
address2:0x0000000087800400
size2:1024B
sumsize:2048B
>>>>merge
address1:0x0000000087800000
size1:2048B
address2:0x0000000087800800
size2:1024B
sumsize:3072B
-----

free success!!!
-----

>>>>free list
address:0x0000000087800000
size:3072B
address:0x0000000087801800
size:8382464B
-----
#####

```

图 4.18 测试结果 3.2

这次测试分别分配了 1 MB 2MB 3MB 然后又释放了 2MB 再次分配了 1MB 而最终这个 1MB 会在第一个 1MB 与 3MB 之间的内存而不是后面的空闲区。

对于这个内存分配的实验的说明：

怎样从原有的 kalloc 机制中扣减 8MB 或 16MB，并实现 kalloc 与我们的堆的隔离？通过修改 memlayout.h、vm.c 来实现隔离。而最重要的是需要进行物理内存和虚拟内存的映射，我最开始因为没有映射导致 panic 的错误。从中我也查资料可以知道 xv6 是三级页表的结构，并看了其映射过程。

然后就是分配内存块的时候，注意要给指针留位置，不能把所有的内存都分出去。

在 kalloc.c 里面的代码，不能直接定义变量，必须定义指针类型，因为作为系统内核，并没有其他部分给这部分分配内存。

4.4 伙伴算法实现动态内存分配

4.4.1 伙伴算法简介

伙伴系统是一种用于内存管理的算法，广泛应用于操作系统和一些内存分配库中。它通过将内存块划分成大小为 2 的幂次方的块，并使用一种特殊的合并管理机制来管理空闲内存。

其工作原理如下：

1. 内存块的划分：内存被划分为大小为 2 的幂次方的块，这些块被称为“伙伴”。例如，内存的大小可以分为 1KB, 2KB, 4KB 等。
2. 空闲块的管理：系统维护一个空闲链表组，每个链表对应一个大小的空闲块。空闲块按大小分类，并存储在相应的链表中。
3. 内存分配：当一个进程请求分配内存时，系统在对应大小的链表中查找。如果找到一个足够大的空闲块，则直接分配。如果没有找到足够大的块，则在系统中从更大的块中分裂出一个适合大小的块。
4. 内存释放：当进程释放内存时，系统会将释放的块，与其“伙伴”合并。如果两个伙伴都空闲，则合并成一个更大的块。

伙伴算法的优点：

1. 减少外部碎片：通过合并伙伴块，减少外部碎片，提高内存利用率。
2. 分配和释放速度快：分配和释放操作时间的复杂度均为 $O(\log n)$ ，其中 n 是内存的大小。

伙伴算法的缺点：

1. 内存碎片：由于内存块的大小为 2 的幂次方，可能会导致内部碎片。
2. 复杂性：相较于简单的首次适应算法，伙伴系统的复杂度较高。

4.4.2 伙伴算法的实现

4.4.2.1 宏定义

用 24 表示最大的内存数为 8MB, block 表示当前内存块信息, freelist 为内存链信息。

```
#define MAX 24
struct block{
    struct block*next;
};
struct freelist{
    struct block*head;
};
```

图 4.19 宏定义

4.4.2.2 初始化

初始化时，先对空闲链表组进行初始化，此时只有最大的空闲块数量是 1，其余均为 0。之后所有的块分配会从这个最大的块开始分配。

```
void buddy_init(){
    for(int i=0;i<MAX;i++){
        freelist[i].head=NULL;
    }
    // the initial block is a large block
    struct block *initblock=(struct block *)PHYSTOP;
    initblock->next=NULL;
    freelist[MAX-1].head=initblock;
}
```

图 4.20 初始化

4.2.2.3 分配函数

分配函数首先需要根据输入的大小向上取整，然后从对应的链表开始找空闲块，如果找到了空闲块直接分配，如果没找到就从更大的块中找空闲块。注

意，每次分裂内存块时为了方便，都从地址低的那一端分。

```
void *buddy_alloc(uint size){
    int order;
    for(order=0;order<MAX;order++){
        if((1<<order)>=size){
            break;
        }
    }
    if(order==MAX){
        return;
    }
    for(int i=order;i<MAX; i++){
        if.freelist[i].head!=NULL){
            struct block*block=free[i].head;

            free[i].head=block->next;
            while(i>order){
                i--;
                struct block*buddy=(struct block*)((uint64)block+(1<<i));
                buddy->next=free[i].head;
                free[i].head=buddy;
            }
            return (void*)block;
        }
    }
    return NULL;
}
```

图 4.21 内存块的分配

4. 2. 2. 4 内存块的释放

根据输入的地址和大小确定要释放的内存块，然后通过二进制地址只有一位不同，两者在对应位置的异或关系，找到其伙伴。每次合并时搜索一遍同级的空闲块链表，看看是否能找到对应的伙伴，如果找到就合并。


```

void buddy_free(void *ptr, uint size){
    int order;
    for(order=0; order<MAX; order++){
        if((1<<order)>=size){
            break;
        }
        struct block *block=(struct block *)ptr;
        while(order<MAX-1){
            struct block *buddy=(struct block *)((uint64) block^(1<<order));
            struct block **current=&freelist[order].head;
            while(*current!=NULL&&*current!=buddy){
                current=&(*current)->next;
            }
            if(*current==NULL){
                break;
            }
            *current=(*current)->next;
            block=(struct block *)((uint64)block&(uint64)buddy);
            order++;
        }
        block->next=freelist[order].head;
        freelist[order].head=block;
    }
}

```

图 4.22 内存的释放

4.2.2.5 伙伴算法的演示

这里简单演示一下仅仅分配了 3MB 的内存并释放掉。

```

$ memtest
the size of memory is 1
the size of memory is 2
the size of memory is 4
the size of memory is 8
the size of memory is 16
the size of memory is 32
the size of memory is 64
the size of memory is 128
the size of memory is 256
the size of memory is 512
the size of memory is 1024
the size of memory is 2048
the size of memory is 4096 0x0000000087801000
the size of memory is 8192 0x0000000087802000
the size of memory is 16384 0x0000000087804000
the size of memory is 32768 0x0000000087808000
the size of memory is 65536 0x0000000087810000
the size of memory is 131072 0x0000000087820000
the size of memory is 262144 0x0000000087840000
the size of memory is 524288 0x0000000087880000
the size of memory is 1048576 0x0000000087900000
the size of memory is 2097152 0x0000000087a00000
the size of memory is 4194304 0x0000000087c00000
the size of memory is 8388608
the size of memory is 16777216 0x0000000087e00000

```

图 4.23 内存分配以后

```
the size of memory is 1 0x0000000087800000
the size of memory is 2 0x0000000087800000
the size of memory is 4 0x0000000087800000
the size of memory is 8 0x0000000087800000
the size of memory is 16 0x0000000087800000
the size of memory is 32 0x0000000087800000
the size of memory is 64 0x0000000087800000
the size of memory is 128 0x0000000087800000
the size of memory is 256 0x0000000087800000
the size of memory is 512 0x0000000087800000
the size of memory is 1024 0x0000000087800000
the size of memory is 2048 0x0000000087800000
the size of memory is 4096 0x0000000087801000
the size of memory is 8192 0x0000000087802000
the size of memory is 16384 0x0000000087804000
the size of memory is 32768 0x0000000087808000
the size of memory is 65536 0x0000000087810000
the size of memory is 131072 0x0000000087820000
the size of memory is 262144 0x0000000087840000
the size of memory is 524288 0x0000000087880000
the size of memory is 1048576 0x0000000087900000
the size of memory is 2097152 0x0000000087a00000
the size of memory is 4194304 0x0000000087c00000
the size of memory is 8388608
```

图 4.24 内存释放

4.5 两种算法的对比总结

在操作系统内存管理中，首次适应算法和伙伴算法时常见的内存分配的算法。首次适应算法比较简单，但是外部碎片化验收，平均时间复杂度为 $O(n)$ ，内存利用率较高，但容易产生大量小碎片；而伙伴算法，较为复杂，内存碎片可能要多，平均时间复杂度为 $O(\log n)$ 最坏为 $O(n)$ ，内存利用率较低，只能以 2 的幂次方为单位分配。

4.6 实验总结

在本次实验中，我首先在 Linux 中用 QEMU 搭建了一个 xv6 操作系统运行的环境，然后分别实现了系统调用和内存管理两大任务。

在任务 1 中，我实现了系统调用返回进程数量的功能。我认为其最关键的一点主要是用户态函数和内核函数到底是如何映射的。

而在任务 2 中，我分别用两个算法实现了内核内存分配的功能，我认为其中有几个注意的点，第一就是不要忘记物理内存和虚拟内存的映射，从系统顶部扣去内存之后，需要增加其映射。第二就是注意 `kalloc.c` 中尽量不要创建变量，因为没有其作为系统内核，不会再有人给它分配了，创建的都是指针结构体，而在伙伴系统中为了方便，创建了几个有待改进。

而在伙伴系统中我觉得这数据结构用树似乎比链表更合适，因为时间有限，而且 C 语言的模板库不多，实现起来过于复杂，未能做改进。

总之，这次试验算是结束了，我真实的参与了小型的操作系统调用，和内存分配，虽然遇到了很多困难但是还是学习到了很多东西。我觉得意义匪浅，希望在未来，有更多的机会进阶自己的技术，更加努力学习计算机知识。

最后对本次实验做的一些小总结如下图：

系统调用

1. 修改makefile文件 添加u/_getprocs u_memTest 作为用户的目标文件，即编译后的可执行文件
2. 在defs.h 文件里 proc添加getprocs 在kalloc里添加memTest
3. 在 proc里添加 getprocs函数 在kalloc里添加对应的memTest相关函数
4. 改变syscall 文件里的内容 在.h里面添加宏定义 和在.c文件里添加系统调用号和处理函数的映射关系 trapframe中获取系统调用号存到 a0寄存器中
5. 在sysproc.c里添加对应的系统调用的接口 两个
6. 在user.h里添加对应的函数名，并在user文件里写出对应的函数实现
7. usys.pl 文件通常用于生成系统调用的汇编代码，它的作用是将系统调用的声明转换为汇编语言代码。 在usys.pl里添加系统调用的接口

内核动态内存分配

1. 先从memlayout中最顶端的地址扣除8MB 需要在kvmmap里完成虚拟地址与物理地址的直接映射 xv6是三级页表
2. 初始化内存写minit 与kinit不同 128MB可以分成很多4KB而 动态分配内存只有一大块所以只用初始化一次 注意minit在主函数的位置
3. 单链表循环遍历freelist的个数
4. 合并空闲内存块，注意合并方法，如果这个内存的首地址加上它的大小等于下一块空闲内存的首地址就合并，可能下下块也是空闲的所以要用continue
5. free 首次适应算法要按照地址递增顺序 排列空闲块
6. 分配内存块，不能完全分配，要给指针留位置
7. 注意所有的kalloc 里面的都不能直接创建变量，因为没人给他分配

图 4.25 总结

参考资料

[1]MIT6.1810<https://pdos.csail.mit.edu/6.828/2023/tools.html>

教师评语评分

评语： _____

评分：_____

评阅人：

年 月 日