

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：孔若杉

学生学号：2021300004024

学生姓名：杨乐

学年学期：2023-2024 学年第二学期

完成时间：2023.4.16

成绩：

二〇二四年五月

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： 杨乐

日期： 2024. 5. 1

摘 要

“实验环境搭建”实验的实验目的是：在 x86 架构的机器上构建支持 RISC-V 架构的操作系统 xv6 运行的环境、初步认识 RISC-V 架构。

实验内容主要包括：在 x86 架构的机器上构建基于 RISC-V 的 xv6 实验环境需要使用交叉编译技术和模拟器来模拟 RISC-V 环境

实验结论为：成功构建支持 RISC-V 架构的操作系统 xv6 运行的环境。

“系统调用”实验的实验目的是：熟悉 RISC-V 体系结构、理解 xv6 中系统调用的工作原理、获得修改 xv6 源代码的经验。

实验内容主要包括：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，该调用执行一个简单任务，返回系统中当前进程的数量。

实验结论为可以完成。

“内存管理实验”实验的实验目的是：认识和实践 xv6 操作系统内存管理方法、通过设计和实现类似 malloc 的动态内存分配器，理解操作系统如何处理内存分配和回收，以及如何动态处理不同大小内存块的请求。

实验内容主要包括：设计并实现一个类似 malloc 的动态内存分配器，用于为用户态程序运行时分配和释放不同大小的内存块。

实验结论为伙伴系统在一定程度上可以很好地管理动态内存。

关键词：RISC-V 架构；环境搭建；系统调用；内存管理

目 录

目录

操作系统内核实验	1
摘 要	3
1 xv6 实验环境的搭建	5
1.1 实验内容介绍与前置条件	5
1.1.1 xv6 操作系统介绍	5
1.1.2 配置 linux 环境	5
1.1.3 下载 xv6 源代码	5
1.2 实验环境搭建	5
1.2.1 安装 RISC-V 编译器和工具链	5
1.2.2 QEMU 模拟器	6
1.3 实验 Xv6 环境的编译	6
1.4 工具 Vscode 的 GDB 调试	7
1.5 系统调用断点添加	8
1.6 实验结果及总结	8
2 xv6 系统调用实验	11
2.1 实验综述	11
2.1.1 实验任务	11
2.1.2 实验目的	11
2.2 在 xv6 中实现当前进程数量的系统调用	11
2.3 实验结果	13
3 xv6 内存管理实验	15
3.1 实验综述	15
3.1.1 实验任务	15
3.1.2 实验目的	15
3.2 原 xv6 内存管理	15
3.3 伙伴系统	17
3.3.1 开辟 8MB	17
3.3.2 伙伴系统管理	18
3.3.3 堆的分配补充	23
3.3.4 系统调用	23
3.4 实验结果及结论	24
参考文献	26

1 xv6 实验环境的搭建

1.1 实验内容介绍与前置条件

本实验是在基于 RISC-V 架构的 xv6 操作系统上进行，搭建好所有的实验环境，并且补充一个系统调用的断点。由于之前对于 xv6 操作系统的认识与理解几乎为零，所以先对 xv6 操作系统以及相应的工具方法进行介绍。

1.1.1 xv6 操作系统介绍

Xv6 是由麻省理工学院 (MIT) 为操作系统工程的课程（代号 6.828），开发的一个教学目的的操作系统。Xv6 是在 x86 处理器上 (x 即指 x86) 用 ANSI 标准 C 重新实现的 Unix 第六版 (Unix V6，通常直接被称为 V6)。

xv6 的代码基本用 C 语言编写。整个系统的代码量相对较少，结构清晰，便于学生理解和探索。它支持多进程操作，包括进程的创建、管理、调度以及同步与互斥。此外，xv6 还包含了简单的文件系统，使用户能够进行基本的文件操作，如创建、打开、读写和关闭文件。

1.1.2 配置 linux 环境

本实验的实验环境为 linux，因为之前配置过 VMware 的 ubuntu 镜像，所以仍然选取 VMware 的 Ubuntu 环境。相关配置互联网教程十分详细，不再叙述。

1.1.3 下载 xv6 源代码

通过 git 下载，Git 是一个版本控制系统，用于跟踪 Linux 内核代码的历史更改，方便协作和版本管理。

在 linux 环境下。通过以下 git 命令即可拉取代码：

```
git clone https://github.com/mit-pdos/xv6-riscv.git
```

1.2 实验环境搭建

在 x86 架构的机器上构建基于 RISC-V 的 xv6 实验环境需要使用交叉编译技术和模拟器来模拟 RISC-V 环境。

1.2.1 安装 RISC-V 编译器和工具链

安装 RISC-V 编译器和工具链的目的是通过交叉编译技术来支持在模拟器上的编译。以 ubuntu 为例，通过以下命令来完成工具链的下载：

```
sudo apt install binutils-riscv64-linux-gnu
sudo apt install gcc-riscv64-linux-gnu
sudo apt install gdb-multiarch
sudo apt install qemu-system-misc opensbi u-boot-qemu qemu-utils
```

1.2.2 QEMU 模拟器

QEMU 是一款支持模拟多种硬件平台的开源虚拟机软件，我们通过 QEMU 来模拟 RISC-V 架构虚拟机，然后运行相应的 xv6 操作系统。

在配置工具链时候，可以采取 `make qemu` 编译命令，缺少包装相应包的措施。

在 `xv6-riscv` 目录下，执行 `make qemu`，正常的话就进入了 xv6。如果要退出可以采取以下措施：

1. first press Ctrl + A (A is just key a, not the alt key),
2. then release the keys,
3. afterwards press X.

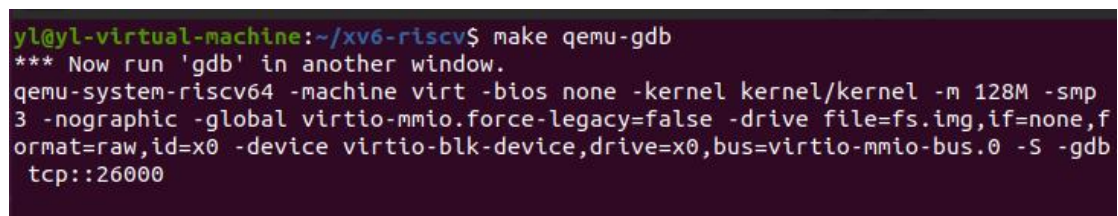
图 1-1 退出 QEMU 操作

1.3 实验 Xv6 环境的编译

在 `xv6-riscv` 目录下，采取 `make qemu-gdb` 的命令，对 xv6 进行编译调试。值得注意的是，在 `xv6` 源代码目录下，通常有一个 `Makefile` 文件定义了如何构建系统。使用 `make` 命令及 RISC-V 工具链构建 xv6。

在执行 `make qemu-gdb` 时，进行会阻塞，这样需要我们在另一个终端下采取以下命令：

```
gdb - multiarch kernel/kernel
```



```
yl@yl-virtual-machine:~/xv6-riscv$ make qemu-gdb
*** Now run 'gdb' in another window.
qemu-system-riscv64 -machine virt -bios none -kernel kernel/kernel -m 128M -smp
3 -nographic -global virtio-mmio.force-legacy=false -drive file=fs.img,if=none,f
ormat=raw,id=x0 -device virtio-blk-device,drive=x0,bus=virtio-mmio-bus.0 -S -gdb
tcp::26000
```

图 1-2 进程阻塞

```

yl@yl-virtual-machine:~/xv6-riscv$ gdb-multiarch kernel/kernel
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04.1) 9.2
Copyright (C) 2020 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from kernel/kernel...
The target architecture is assumed to be riscv:rv64
.gdbinit:3: Error in sourced command file:
Undefined command: ". Try "help".
(gdb)

```

图 1-3 gdb 运行

值得说明的是，在 make qemu-gdb 过程中，可能会出现类似下图问题。

```

warning: File "/home/delta/xv6-riscv/.gdbinit" auto-loading has been declined by your `auto-load safe-path' set
to "$debugdir:$datadir/auto-load".
To enable execution of this file add
  add-auto-load-safe-path /home/delta/xv6-riscv/.gdbinit
line to your configuration file "/home/delta/.gdbinit".
To completely disable this security protection add
  set auto-load safe-path /
line to your configuration file "/home/delta/.gdbinit".

```

图 1-4 可能遇到的问题

这是因为 QEMU-GDB 会默认使用当前目录下的 .gdbinit，但是当前目录的 .gdbinit 可以运行，但是不能 GDB 调试，所以我们要使用上一级目录下的 .gdbinit 来解决这个问题。就可以按照图中的报错和修改提示完成。

1.4 工具 Vscode 的 GDB 调试

我们想用一個更加方便的代码调试改写工具，比如 Vscode 来完成剩下的实验，所以我们还要进行 Vscode 的配置。

Vscode 的一些基本工具如 c++ 等不再叙述，主要时我们需要打开 xv6-riscv 后，运行调试 xv6 代码，需要我们创建两个 json 文件：launch.json 和 tasks.json。

这时我们会遇到类似下图的错误。

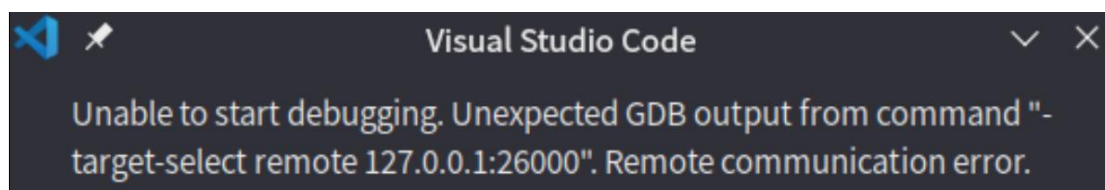


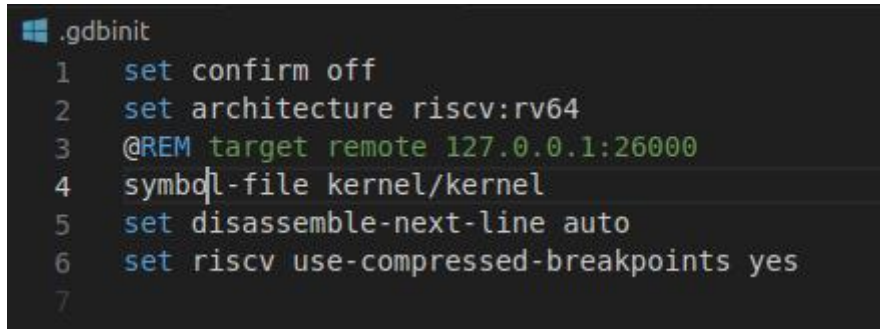
图 1-5 Vscode 报错

这是因为 .gdbinit 中有 target remote 127.0.0.1:26000，这个文件会被

gdb 最先执行一遍.

此外, 我们在 launch.json 中指定了 "miDebuggerServerAddress": "127.0.0.1:26000", 同一个 remote address 被配置了两次.

只需要把 .gdbinit 中的注释掉即可, 如下图

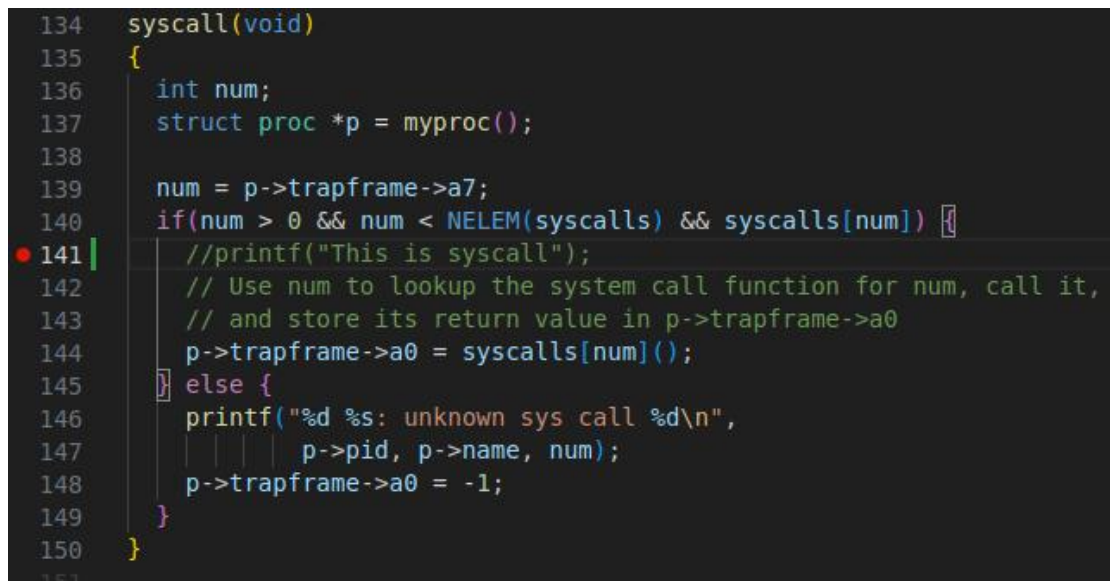
A screenshot of a text editor showing the contents of a .gdbinit file. The file contains seven lines of GDB configuration commands. Line 3, which sets the target remote to 127.0.0.1:26000, is highlighted in green. The other lines are in blue and black text.

```
.gdbinit
1  set confirm off
2  set architecture riscv:rv64
3  @REM target remote 127.0.0.1:26000
4  symbol-file kernel/kernel
5  set disassemble-next-line auto
6  set riscv use-compressed-breakpoints yes
7
```

图 1- 6 注释相同代码后的 .gdbinit

1.5 系统调用断点添加

在进行以上配置后, 我们可以在 Vscode 中运行 xv6 的代码, 我们想在系统调用例如输入 ls 时可以显示一些内容或者断点停留在系统调用函数处, 所以我们在 syscall.c 文件中加入断点并且输出特定文字。

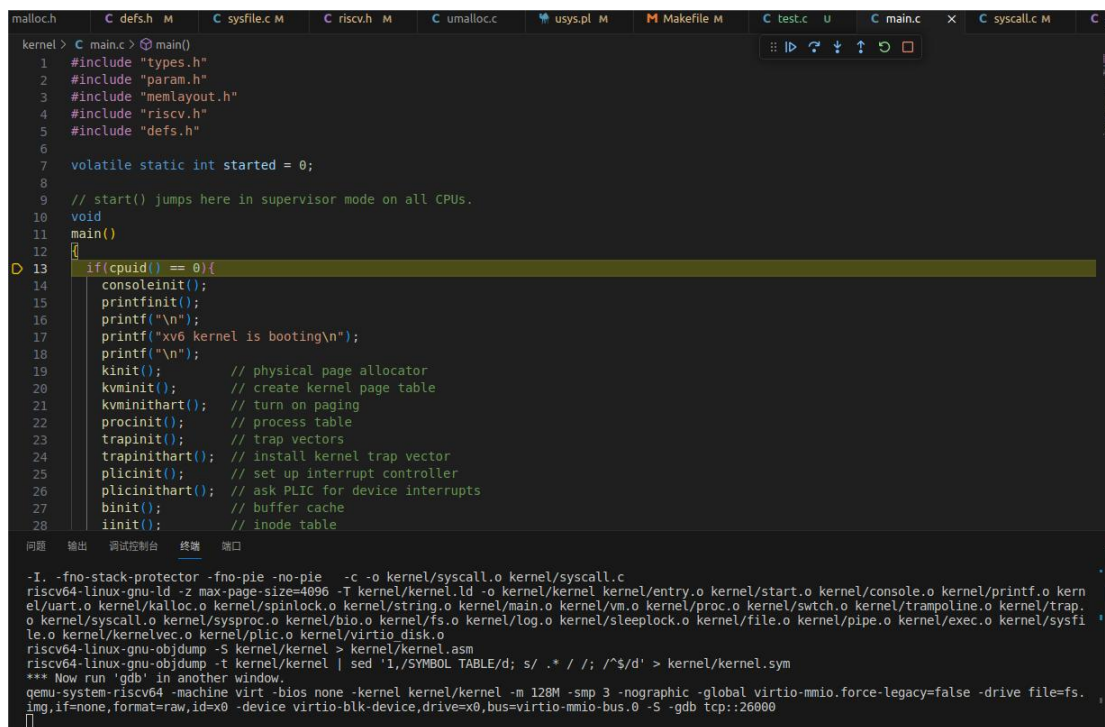
A screenshot of the syscall.c file in a code editor. The code is for the syscall function. A red dot breakpoint is placed on line 141, which contains the comment //printf("This is syscall");. The code includes variable declarations, a struct pointer, and logic to handle system calls by looking up the function in the syscalls array and calling it. Line numbers 134 to 150 are visible on the left margin.

```
134  syscall(void)
135  {
136      int num;
137      struct proc *p = myproc();
138
139      num = p->trapframe->a7;
140      if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
141          //printf("This is syscall");
142          // Use num to lookup the system call function for num, call it,
143          // and store its return value in p->trapframe->a0
144          p->trapframe->a0 = syscalls[num]();
145      } else {
146          printf("%d %s: unknown sys call %d\n",
147              p->pid, p->name, num);
148          p->trapframe->a0 = -1;
149      }
150  }
```

图 1- 7 加入 printf 和断点的 syscall

1.6 实验结果及总结

经过上述的实验步骤, 我们可以在 Vscode 中成功运行 xv6 的代码, 运行结果如下:

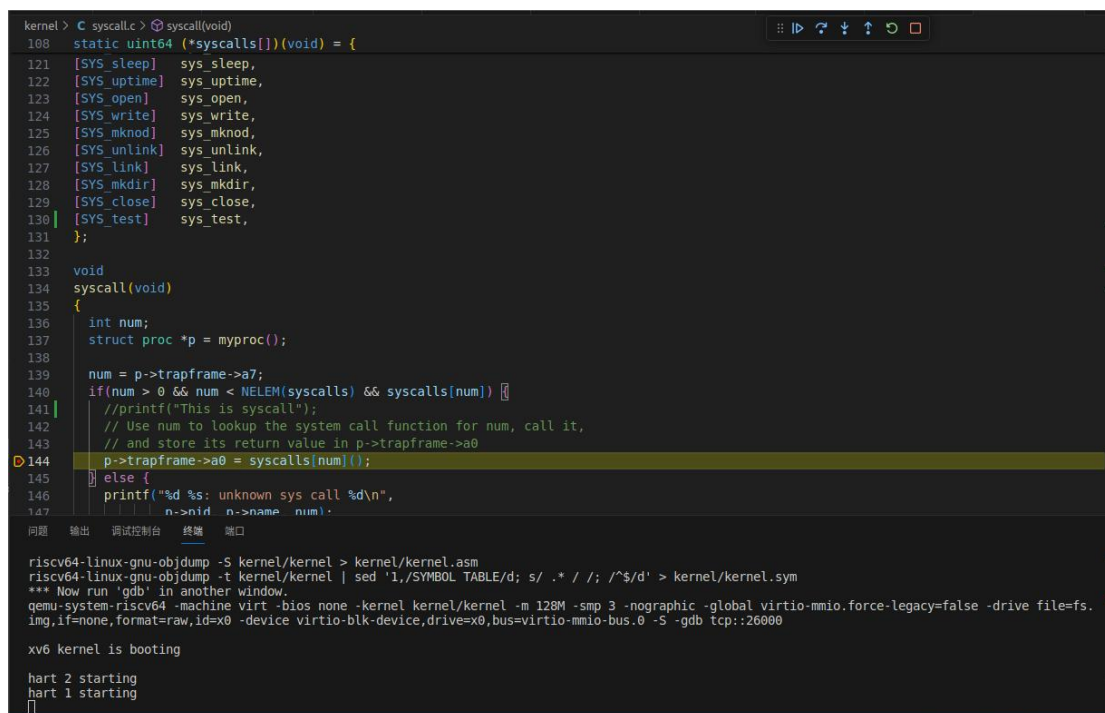


```
kernel > C main.c > main()
1 #include "types.h"
2 #include "param.h"
3 #include "memlayout.h"
4 #include "riscv.h"
5 #include "defs.h"
6
7 volatile static int started = 0;
8
9 // start() jumps here in supervisor mode on all CPUs.
10 void
11 main()
12 {
13     if(cpuid() == 0){
14         consoleinit();
15         printfinit();
16         printf("\n");
17         printf("xv6 kernel is booting\n");
18         printf("\n");
19         kinit(); // physical page allocator
20         kvminit(); // create kernel page table
21         kvminithart(); // turn on paging
22         procinit(); // process table
23         trapinit(); // trap vectors
24         trapinithart(); // install kernel trap vector
25         plicinit(); // set up interrupt controller
26         plicinithart(); // ask PLIC for device interrupts
27         binit(); // buffer cache
28         iinit(); // inode table
29     }
30 }
```

```
-I. -fno-stack-protector -fno-pie -no-pie -c -o kernel/syscall.o kernel/syscall.c
riscv64-linux-gnu-ld -z max-page-size=4096 -T kernel/kernel.ld -o kernel/kernel kernel/entry.o kernel/start.o kernel/console.o kernel/printf.o kernel/uart.o kernel/kalloc.o kernel/spinlock.o kernel/string.o kernel/main.o kernel/vm.o kernel/proc.o kernel/swtch.o kernel/trampoline.o kernel/trap.o kernel/syscall.o kernel/sysproc.o kernel/bio.o kernel/fs.o kernel/log.o kernel/sleeplock.o kernel/file.o kernel/pipe.o kernel/exec.o kernel/sysfile.o kernel/kernelvec.o kernel/plic.o kernel/virtio disk.o
riscv64-linux-gnu-objdump -S kernel/kernel > kernel/kernel.asm
riscv64-linux-gnu-objdump -t kernel/kernel | sed '1,/SYMBOL TABLE/d; s/ ./ /; /$/d' > kernel/kernel.sym
*** Now run 'gdb' in another window.
qemu-system-riscv64 -machine virt -bios none -kernel kernel/kernel -m 128M -smp 3 -nographic -global virtio-mmio.force-legacy=false -drive file=fs.img,if=none,format=raw,id=x0 -device virtio-blk-device,drive=x0,bus=virtio-mmio-bus.0 -S -gdb tcp::26000
```

图 1- 8 Vscode 调式初始化

我们如果加入断点，会发现在调试过程中成功在断点处停留。



```
kernel > C syscall.c > syscall(void)
108 static uint64 (*syscalls[])(void) = {
109     [SYS_sleep] sys_sleep,
110     [SYS_uptime] sys_uptime,
111     [SYS_open] sys_open,
112     [SYS_write] sys_write,
113     [SYS_mknod] sys_mknod,
114     [SYS_unlink] sys_unlink,
115     [SYS_link] sys_link,
116     [SYS_mkdir] sys_mkdir,
117     [SYS_close] sys_close,
118     [SYS_test] sys_test,
119 };
120
121 void
122 syscall(void)
123 {
124     int num;
125     struct proc *p = myproc();
126
127     num = p->trapframe->a7;
128     if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
129         //printf("This is syscall");
130         // Use num to lookup the system call function for num, call it,
131         // and store its return value in p->trapframe->a0
132         p->trapframe->a0 = syscalls[num]();
133     } else {
134         printf("%d %s: unknown sys call %d\n",
135             num, p->name, num);
136     }
137 }
```

```
riscv64-linux-gnu-objdump -S kernel/kernel > kernel/kernel.asm
riscv64-linux-gnu-objdump -t kernel/kernel | sed '1,/SYMBOL TABLE/d; s/ ./ /; /$/d' > kernel/kernel.sym
*** Now run 'gdb' in another window.
qemu-system-riscv64 -machine virt -bios none -kernel kernel/kernel -m 128M -smp 3 -nographic -global virtio-mmio.force-legacy=false -drive file=fs.img,if=none,format=raw,id=x0 -device virtio-blk-device,drive=x0,bus=virtio-mmio-bus.0 -S -gdb tcp::26000

xv6 kernel is booting

hart 2 starting
hart 1 starting
```

图 1- 9 Vscode 断点调试

当我们输入 ls 命令时，会发现系统会多次在 syscall 系统调用处停留，这样证明我们的实验的成功，输入如下：

```
console 3 20 0
$ ls
.          1 1 1024
..         1 1 1024
README    2 2 2305
cat        2 3 32360
echo       2 4 31232
forktest  2 5 15336
grep       2 6 35720
init       2 7 32032
kill       2 8 31232
ln         2 9 31160
ls         2 10 34280
mkdir      2 11 31272
rm         2 12 31264
sh         2 13 53512
stressfs   2 14 32136
usertests  2 15 180616
grind      2 16 47344
wc         2 17 33360
zombie     2 18 30808
test       2 19 30696
console    3 20 0
$
```

图 1- 10 命令 ls 的输出

本次实验所遇到的问题基本上都是老师提供的文档^[1]中所提，所以没有太大的问题，实验过程比较顺利。

2 xv6 系统调用实验

2.1 实验综述

2.1.1 实验任务

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，该调用执行一个简单任务，比如返回系统中当前进程的数量。

2.1.2 实验目的

熟悉 RISC-V 体系结构。

理解 xv6 中系统调用的工作原理。

获得修改 xv6 源代码的经验。

2.2 在 xv6 中实现当前进程数量的系统调用

在实现我们的系统调用前，我们要明确系统调用的步骤，我们想在用户态调用内核态定义的代码，首先我们要在 Makefile 文件中定义相关的命令，以便于用户可以在用户态通过命令来调用系统调用。

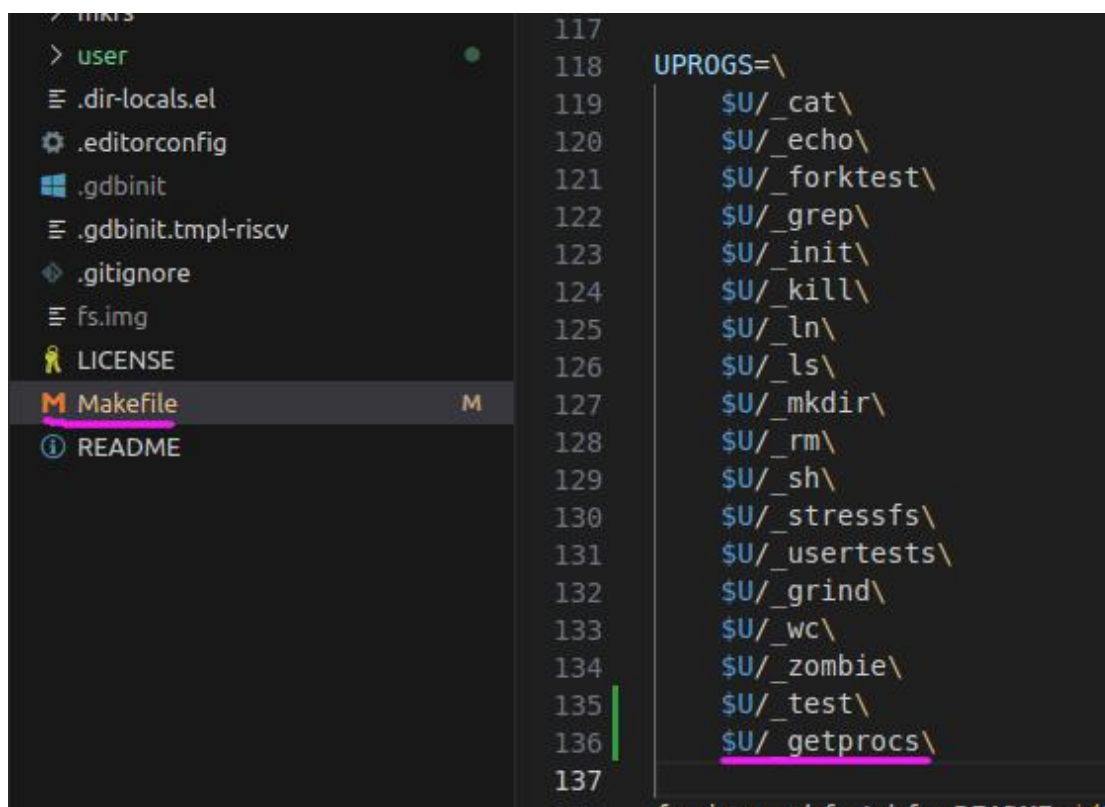


图 2- 1 Makefile 文件添加命令

然后我们需要在用户态定义一个有关 `getprocs` 调用的函数，这样在命令 `getprocs` 输入后，我们会调用相应的用户态的 `getprocs.c` 文件，然后其中的 `main` 将获取到的命令行中的参数传给内核态中的 `getprocs` 系统调用中。

```
user > C getprocs.c > main()
1  #include "kernel/types.h"
2  #include "user/user.h"
3
4  int main()
5  {
6      int count = getprocs();
7      printf("There are %d active processes\n",count);
8      exit(0);
9  }
```

图 2- 2 用户态 `getprocs.c` 实现

别忘记在 `user/user.h` 头文件中声明一个新的系统调用函数原型。在本实验中，命名为 `int getprocs(void)`；

编辑 `user/usys.pl` 脚本以定义新系统调用的用户空间映射，它将生成 `usys.S` 汇编代码。添加如下行：`entry("getprocs")`；

```
38  entry("uptime");
39  entry("test");
40  entry("getprocs");
```

图 2- 3 `usr.pl` 脚本添加用户空间映射

然后我们需要在内核态给分配系统调用号。

```
21  #define SYS_mkdir  20
22  #define SYS_close  21
23  #define SYS_test   22
24  #define SYS_getprocs 23
```

图 2- 4 系统调用号 23

然后在 `syscall.c` 将系统调用与内核内函数实现关联起来。

```
128  [SYS_link]      sys_link,
129  [SYS_mkdir]     sys_mkdir,
130  [SYS_close]     sys_close,
131  [SYS_test]      sys_test,
132  [SYS_getprocs]  sys_getprocs,
133  };
134
```

图 2-5 系统调用与函数关联

```
102 extern uint64 sys_mkdir(void);
103 extern uint64 sys_close(void);
104 extern uint64 sys_test(void);
105 extern uint64 sys_getprocs(void);
106
```

图 2-6 函数声明

然后在 proc.c 文件中实现相应的 getprocs() 函数即可。

```
uint64
sys_getprocs(void)
{
    struct proc *p;
    int count = 0;

    acquire(&wait_lock);
    for(p = proc; p < &proc[NPROC]; p++){
        acquire(&p->lock);
        if(p->state!=UNUSED)
        {
            count++;
        }
        release(&p->lock);
    }
    release(&wait_lock);
    myproc()->trapframe->a0 = count;
    printf("%d",count);
    return count;
}
```

图 2-7 内核实现 getprocs

2.3 实验结果

最后我们通输入命令 getprocs，最后输出活跃的进程为 3 个。

```
xv6 kernel is booting

hart 1 starting
hart 2 starting
init: starting sh
$ getprocs
3There are 3 active processes
$
```

图 2-8 系统调用实验结果

实验结果表明此时有 3 个活跃的进程，其中第一个 3 为内核时输出的 3 个活跃进程，后面 “There are 3 active processes” 为用户态接收到的进程数。我

们通过系统调用实验，对 xv6 系统整体有一个比较粗略的认识，同时对于操作系统内核与用户态分离的思想也有了进一步的认识。

3 xv6 内存管理实验

3.1 实验综述

3.1.1 实验任务

设计并实现一个类似 malloc 的动态内存分配器，用于为用户态程序运行时分配和释放不同大小的内存块。

3.1.2 实验目的

认识和实践 xv6 操作系统内存管理方法。

通过设计和实现类似 malloc 的动态内存分配器，理解操作系统如何处理内存分配和回收，以及如何动态处理不同大小内存块的请求。

3.2 原 xv6 内存管理

xv6 系统中内存管理的机制，主要位于 kalloc.c 和 vm.c 中，我们可以通过理解内核使用 kalloc() 和 kfree() 来分配和释放内存的机制。kalloc() 仅用于 xv6 的内核空间，并且它总是分配固定大小的内存块(4KB)。这一点我们可以在 memlayout.h 和 risc.h 中看出。下图中可以看出 kalloc.c 管理的内存从 0x80000000 到 0x88000000。

```
#define KERNBASE 0x80000000L
#define PHYSTOP (KERNBASE + 128*1024*1024)
```

图 3-1 管理的内存

并且每次分配固定大小内存块（4KB）。

```
#define BYTES 8 //bytes
#define PGSIZE 4096 // bytes per page
#define PGSHIFT 12 // bits of offset within a page
```

图 3-2 固定页大小 4KB

并且发现 vm.c 中的内容不修改就可以完成相应任务，vm 中的主要工作是采取直接映射的方式。所以 vm.c 中的内容在这里也不再分析，所以我们主要分析 kalloc.c 中的内容。

首先 extern 定义了一个 end[]，表示内核管理的末地址。

```
extern char end[]; // first address after kernel.
                // defined by kernel.ld.
```

图 3-3 内存代码的末地址

然后是一个结构体 struct，本质上是一个结点，用来作为链表结点。

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

图 3-4 run 结点

然后定义一个 kmem，用来保存自旋锁和空闲链表的首结点。

```
struct {
    struct spinlock lock;
    struct run *freelist;
} kmem;
```

图 3-5 空闲链表

然后是一个初始化内存管理，它调用了 freerange 这个函数，然后初始化了 kmem 中的自旋锁以便后续使用。

```
void
kinit()
{
    initlock(&kmem.lock, "kmem");
    freerange(end, (void*)MYPHYSTOP);
}
```

图 3-6 内存管理初始化

然后说明一下调用的 freerange，它通过传入首地址和末地址，在这个地址范围中，以块大小调用 kfree 函数，来释放所管理的内存。

```
void
freerange(void *pa_start, void *pa_end)
{
    char *p;
    p = (char*)PGROUNDUP((uint64)pa_start);
    for(; p + PGSIZE <= (char*)pa_end; p += PGSIZE)
        kfree(p);
}
```


图 3- 7 freerange 函数

对于 kfree 函数，他是以块大小（4KB）来管理，先释放这个块，然后使用头插法把块插入到空闲链表。

```
void
kfree(void *pa)
{
    struct run *r;

    if(((uint64)pa % PGSIZE) != 0 || (char*)pa < end || (uint64)pa >= PHYSTOP)
        panic("kfree");

    // Fill with junk to catch dangling refs.
    memset(pa, 1, PGSIZE);

    r = (struct run*)pa;

    acquire(&kmem.lock);
    r->next = kmem.freelist;
    kmem.freelist = r;
    release(&kmem.lock);
}
```

图 3- 8 kfree 函数

对于分配内存，是和 kfree 函数正好相反，通过空闲链表，用头插法取出结点，然后将该结点管理的内存分配出去。

```
void *
kalloc(void)
{
    struct run *r;

    acquire(&kmem.lock);
    r = kmem.freelist;
    if(r)
        kmem.freelist = r->next;
    release(&kmem.lock);

    if(r)
        memset((char*)r, 5, PGSIZE); // fill with junk
    return (void*)r;
}
```

图 3- 9 kalloc 函数

3. 3 伙伴系统

3. 3. 1 开辟 8MB

通过 kinit 时候 freerange 的范围下调到自定义的位置来开辟出 8MB. 原来

freerange 的范围是 0x80000000-0x88000000, 现在改为 0x80000000-0x87800000, 这样可以开辟出 8MB 大小。

```
void
kinit()
{
    initlock(&kmem.lock, "kmem");
    freerange(end, (void*)MYPHYSTOP);
}
```

图 3- 10 kinit 新定义

```
#define KERNBASE 0x80000000L
#define PHYSTOP (KERNBASE + 128*1024*1024)
#define MYPHYSTOP (KERNBASE + 120*1024*1024)
```

图 3- 11 MYPHYSTOP 位置

然后在 vm.c 中, 因为还是采取直接映射的方式, 并且就算拿出来的 8MB 还要映射, 所以 vm.c 中并没有修改。

所以这样空闲链表中的地址只有 0x80000000-0x87800000, 无论是 kfree 还是 kalloc 都不会超过这个范围。

3. 3. 2 伙伴系统管理

我采用一种比较简单的伙伴系统来管理, 因为我的伙伴系统采取的满二叉树的结构来管理, 如果管理最小单位为 8B 所占内存会很大, 所以仍然采取最小单位为块大小来管理。

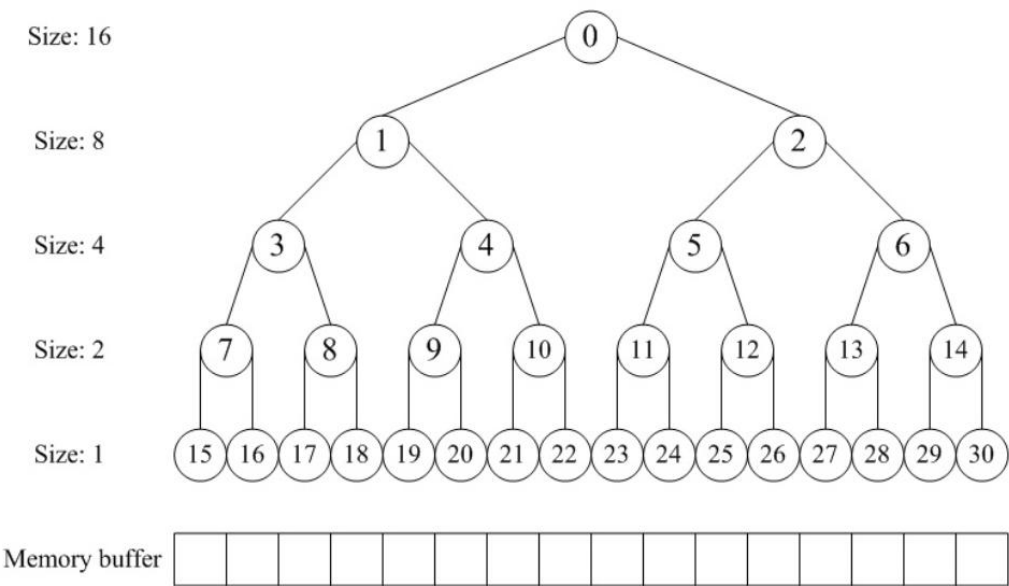


图 3- 12 伙伴系统简易数据结构

大致的数据结构如上图，首先定义这个数据结构。

```
struct buddy {
    struct spinlock lock;
    unsigned size;
    unsigned longest[2*2048-1]; //[2*2048-1];
}self;
```

图 3- 13 伙伴系统数据结构实现

定义了一个自旋锁，要管理的块数 size（注意我们开辟了 8MB，一个块大小是 4KB，所以有 8MB/4KB=2048 块），然后是要管理 size 所需的二叉树结点，因为这是满二叉树的结点数，在这里采取顺序表的形式，所以总共需要 2*2048-1 个结点。longest 数组每个元素都对应一个二叉树的结点，元素的值代表对应内存块的空闲容量。

然后我们对伙伴系统初始化，初始化包括两方面，一个是对管理内存的伙伴系统初始化，一个是对 kinit 没有初始化的内存进行初始化。

```
void buddy_new(int size) {
    unsigned node_size;
    int i;

    if (size < 1 || !is_power_of_2(size)) {
        panic("buddy_new");
    }
    initlock(&self.lock, "self");
    freerange((void*)MYPHYSTOP, (void*)PHYSTOP);
    printf("free了从%p到%p\n", PHYSTOP, MYPHYSTOP);

    self.size = size;
    node_size = size * 2;

    acquire(&self.lock);
    // 遍历每个二叉树结点，为其赋值
    for (i = 0; i < 2 * size - 1; ++i) {
        // 注意：1也是2的幂
        if (is_power_of_2(i + 1)) {
            node_size /= 2;
        }
        // 第一层的结点的值是size，第二层是size/2，第三层是size/4，以此类推
        // 每个值代表的是空闲内存块的数量
        self.longest[i] = node_size;
    }
    release(&self.lock);
}
```

图 3- 14 伙伴系统初始化

其中，我们初始化了定义的 self 这个伙伴系统，然后对 kinit 未初始化的内存仍然采用 kreerange 来初始化，但是初始化并不会让原先的内存管理访问到，因为 kinit 没有初始化这一部分，所以 kfree 和 kalloc 也不能访问这一内存。

因为我们用伙伴系统，所以要判断初始化的内存是不是 2 的幂次方，如果不是不能构成一个满二叉树，所以要用到判断 2 的幂次的函数。

```
inline bool is_power_of_2(unsigned x) {
    return !(x & (x - 1));
}
```

图 3- 15 判断 2 的幂次的函数

然后是内存分配。其中要用到 5 个函数。

```
inline unsigned max(unsigned __a, unsigned __b)
{
    if (__a < __b) return __b;
    return __a;
}
```

图 3- 16 max 函数

```
void buddy_alloc(unsigned size) {
    unsigned index = 0;
    unsigned node_size;
    unsigned offset = 0;

    if(size <= 0) {
        size = 1;
    } else if(!is_power_of_2(size)) {
        size = fixsize(size); // 向上调整到2的幂次
    }

    // 如果根节点下挂的size都不够分配，就返回失败
    if(self.longest[index] < size) {
        panic("buddy_alloc");
    }

    acquire(&self.lock);
    // 由大到小搜索最符合size的结点
    // 并在搜索的过程中，更新index，优先使用左孩子
    for(node_size = self.size; node_size != size; node_size /= 2) {
        if(self.longest[left_leaf(index)] >= size) {
            index = left_leaf(index);
        } else {
            index = right_leaf(index);
        }
    }
}
```

图 3- 17 分配函数 1

```

        index = right_leaf(index);
    }
}

// 找到对应的结点了，就将其管理的空闲内存块数量标记为0
self.longest[index] = 0;
// 这里的node_size是对应层的结点所管理的内存的大小，而index是结点的编
// 根据这个算法，offset恰好是分配内存的起始/索引，从offset往后数size个块的内存都是可用的
offset = (index + 1) * node_size - self.size;

// 因为更新了longest[index]的标记，所以需要更新它上层所有父节点的标记
// 其中的原理是，如果小块内存被占用，那么大块内存就不满足原来的可用状态了
while(index) {
    index = parent(index);
    self.longest[index] = max(self.longest[left_leaf(index)], self.longest[right_leaf(index)]);
}
void *pa = (void*)(MYPHYSTOP + 4096*offset);
memset(pa, 5, size*PGSIZE);
printf("分配了从%p开始到%p的地址\n", pa, pa+size*PGSIZE);
release(&self.lock);
}

```

图 3- 18 分配函数 2

```

inline unsigned left_leaf(unsigned index) {
    return index * 2 + 1;
}

inline unsigned right_leaf(unsigned index) {
    return index * 2 + 2;
}

inline unsigned parent(unsigned index) {
    return (index + 1) / 2 - 1;
}

```

图 3- 19 顺序表返回左右孩子节点或者父节点

```

inline unsigned fixsize(unsigned size) {
    size |= size >> 1;
    size |= size >> 2;
    size |= size >> 4;
    size |= size >> 8;
    size |= size >> 16;
    return size + 1;
}

```

图 3- 20 将想要的大小变为 2 的幂次大小

因为代码中写的有注释，仔细看能看懂，先不做详细介绍了。主要是先把要分配的 size 先上调到最小 2 的幂次，比如要分配 31 个块，那么会把 size 调为 32，给他分配 32 个块。

具体分配措施是在伙伴系统中找到最适合的最偏左的结点，然后把该结点中的数置为 0，表示下面管理的所有块都被分配，然后更新所有父节点的能管理块

的大小。每个树结点维护的数是自己孩子结点的数的最大值，如果两个孩子都是他们相应的最大值，那么自己管理的数是两个孩子结点的数的和。

最后用偏移量 and 对应结点控制的 size 来分配空间。

而对于 free 的方式，如下。

```
void buddy_free(unsigned offset) {
    unsigned node_size, index = 0;
    unsigned left_longest, right_longest;

    if (offset < 0 && offset >= self.size)
        panic("buddy_free");

    node_size = 1;
    index = offset + self.size - 1;
    acquire(&self.lock);
    // 找到被占用的那个内存块，并更新node_size，即那层的结点所管理的内存块的大小
    for(; self.longest[index] != 0; index = parent(index)) {
        node_size *= 2;

        // 如果根节点的都被占用了，则直接返回
        if(index == 0) {
            memset((void*)MYPHYSTOP, 1, 2048*PGSIZE);
            self.longest[index] = node_size;
            printf("释放了从%p开始到%p的地址\n", MYPHYSTOP, MYPHYSTOP+2048*PGSIZE);
            release(&self.lock);
            return;
        }
    }
}
```

图 3- 21free1

```
void *pa = (void*)(MYPHYSTOP + 4096*offset);
memset(pa, 1, node_size*PGSIZE);
printf("释放了从%p开始到%p的地址\n", pa, pa+node_size*PGSIZE);

// 归还内存，将longest恢复到原来结点的值
self.longest[index] = node_size;

// 接下来恢复被占用结点的所有父节点
while(index) {
    index = parent(index);
    node_size *= 2;

    // 在向上回溯的过程中，如果发现左右孩子的元素加起来等于自己，则说明需要合并
    // 否则取他们中大的那个
    left_longest = self.longest[left_leaf(index)];
    right_longest = self.longest[right_leaf(index)];

    if(left_longest + right_longest == node_size) {
        self.longest[index] = node_size;
    } else {
        self.longest[index] = max(left_longest, right_longest);
    }
}
release(&self.lock);
}
```

图 3- 22 free2

具体操作可以参考代码中的注释，他的基本思路是先根据输入的块的 offset 偏移量，做回溯找到分配出去的块，也就是从叶子节点往根节点找，找到第一个值为 0 的结点，然后恢复结点值，同样检查父节点，需要不需要合并块。

3.3.3 堆的分配补充

因为伙伴系统的数据结构是全局变量，它的定义并不在分配的内存中，所以我们把分配出去的内存看作为连续的 2048 个块，主要通过伙伴系统的满二叉树的管理来决定每个块是否分配出去，并且是否有合适的块大小。如图 3-17 和图 3-18，对于找到的要分配出去的块，将对应的块填入随机数，然后分配。如果我们想感知内存使用情况，可以通过伙伴系统的二叉树情况来查看。

如何找到足够大的连续空间，我们通过从根节点遍历（根节点管理的块最多），直到找到最适合的树结点。

对于空闲区域的合并通过每次释放时，看自己的伙伴结点是否也是空闲，如果空闲，则合并同一由父节点管理，同样向上遍历，父节点也会查看自己伙伴是否空闲，一直遍历到某父节点伙伴不空闲，或者到根节点。

3.3.4 系统调用

本实验的系统调用和实验 2 几乎一样（具体可查看实验 2，在这里不在叙述），为数不多的区别就是调用的函数主体。

并且对于图中所有的分配和释放，可以随意的组合或者注释，通过预测结果与实际结果的对比来判断实验是否成功。

同时在代码中定义了初始化错误，分配错误，释放错误三种错误，如果出现相应问题则调用 panic，输出相应错误，以便调试的正确性。

我们通过开放系统调用接口，让使用者用自己享用的方式来调用分配和释放函数。

```

uint64
sys_test(void)
{
    printf("管理内存%p到%p的内存\n", PHYSTOP, MYPHYSTOP);
    printf("初始化\n");
    buddy_new(2048);
    printf("分配1块\n");
    buddy_alloc(1);
    printf("分配1024块\n");
    buddy_alloc(1024);
    // printf("分配2048块\n");
    // buddy_alloc(2048);
    printf("free第1块\n");
    buddy_free(0);
    printf("分配1024块\n");
    buddy_alloc(1024);
    printf("分配1023块\n");
    buddy_alloc(1023);
    //printf("分配1块\n");
    //buddy_alloc(1);
    printf("free1023块\n");
    buddy_free(1024);
    printf("分配1块\n");
    buddy_alloc(1);
    return 0;
}

```

图 3- 23 test 实际测试代码

3. 4 实验结果及结论

如上图测试代码，结果如下

```

hart 2 starting
hart 1 starting
init: starting sh
$ test
管理内存 0x0000000088000000到 0x0000000087800000的内存
初始化
free了从 0x0000000088000000到 0x0000000087800000
分配 1块
分配了从 0x0000000087800000开始到 0x0000000087801000的地址
分配 1024块
分配了从 0x0000000087c00000开始到 0x0000000088000000的地址
free第 1块
释放了从 0x0000000087800000开始到 0x0000000087801000的地址
分配 1024块
分配了从 0x0000000087800000开始到 0x0000000087c00000的地址
分配 1023块
panic: buddy_alloc

```


图 3-24 实验结果、

可以发现与预期结果相同，在分配 1024 块，再分配 1024 块之后，如果要再分配 1023 块，会出现问题。

老师提出的问题在上述实验报告中已经全部解答：

问题一：参考 3.3.1

问题二：参考 3.3.2

问题三：参考 3.3.3

问题四：参考 3.3.2 和 3.3.3

问题五：参考 3.3.2 和 3.3.3

问题六：参考 3.3.4 和 3.4

问题七：参考 3.3.4

参考文献

- [1] Atled. 从零开始使用 Vscode 调试 XV6. [Online] <https://zhuanlan.zhihu.com/p/501901665>