**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 ：2021300004010

学 生 姓 名 ：刘禹铄

学 年 学 期 ：2023-2024学年第二学期

完 成 时 间 ：\_\_\_\_\_\_\_\_\_\_\_\_\_\_

成 绩 ：\_\_\_\_\_\_\_\_\_\_\_\_\_

二○二四年五月

**摘 要**

XXXX实验的实验目的是XXXX。

实验设计主要遵循XXXX。

实验内容主要包括：

实验结论为XXXX

……

……

……

**关键词：**关键词1；关键词2；关键词3

**目 录**

[1 实验环境搭建 4](#_Toc165314926)

[1.1 实验要求 4](#_Toc165314927)

[1.2 环境搭建过程 4](#_Toc165314928)

[1.3 相关测试结果 4](#_Toc165314929)

[2 系统调用实验 6](#_Toc165314930)

[2.1 实验要求 6](#_Toc165314931)

[2.2 xv6相关原理 7](#_Toc165314932)

[2.3 修改方案 7](#_Toc165314933)

[2.4 测试方案 7](#_Toc165314934)

[2.5 相关测试结果及分析 7](#_Toc165314935)

[3 内存管理实验 7](#_Toc165314936)

[3.1 实验要求 7](#_Toc165314937)

[3.2 xv6相关原理 8](#_Toc165314938)

[3.3 修改方案 8](#_Toc165314939)

[3.4 测试方案 8](#_Toc165314940)

[3.5 相关测试结果及分析 8](#_Toc165314941)

[参考文献 9](#_Toc165314942)

# 实验环境搭建

## 1.1 实验要求

在x86架构的机器上构建基于RISC-V的xv6实验环境。具体方法为使用交叉编译技术和模拟器来模拟RISC-V环境。另外，在搭建实验环境之前，需要了解什么是交叉编译，为什么需要交叉编译工具链，以及如何在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。

## 1.2 环境搭建过程

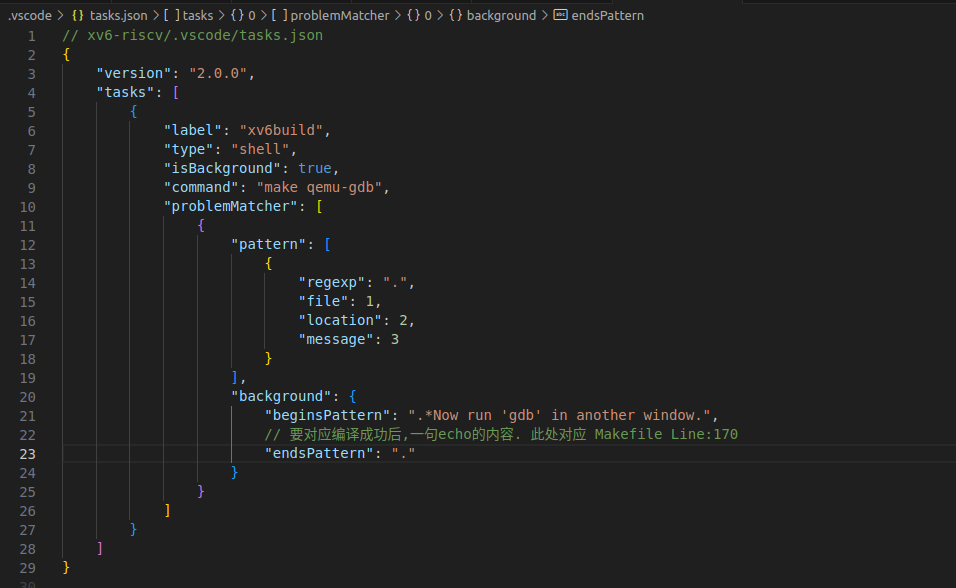
首先，在Linux（Ubuntu 22.04）系统上安装交叉编译工具链：binutils-riscv64-linux-gnu 、gcc-riscv64-linux-gnu 以及 gdb-multiarch。然后使用`git clone https://github.com/mit-pdos/xv6-riscv.git` 获取 xv6 源代码。然后在 `xv6-riscv` 目录下执行 `make qemu`，编译成功后，qemu 模拟器正常启动。

接下来要做的是启动带有GDB调试器的qemu模拟器。首先在 `xv6-riscv` 目录下执行 `make qemu-gdb`，此时进程会阻塞，然后另开一个终端在 `xv6-riscv` 目录下执行 `gdb-multiarch kernel/kernel`。至此，本实验成功启动了带有GDB调试器的qemu模拟器。

最后，我选用了在Visual Studio code 这款 IDE 中编译 xv6 源代码，并且使用QEMU 模拟器启动 xv6 操作系统。具体实现步骤如下：首先我在 `.vscode` 目录下创建了 `launch.json` 和 `tasks.json` 两个文件（文件内容如下图所示），然后在 VScode 中调试整个项目文件，最终成功启动 xv6 操作系统。

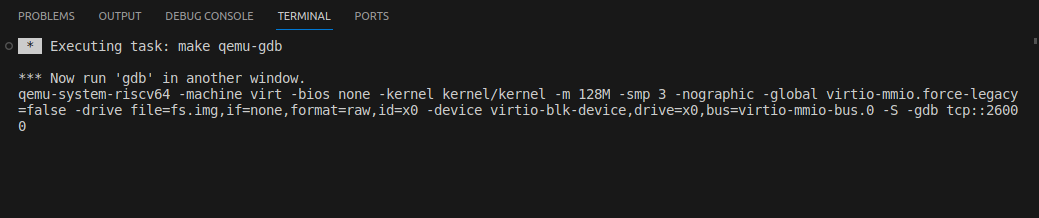


**图1. 1 launch.json 文件**

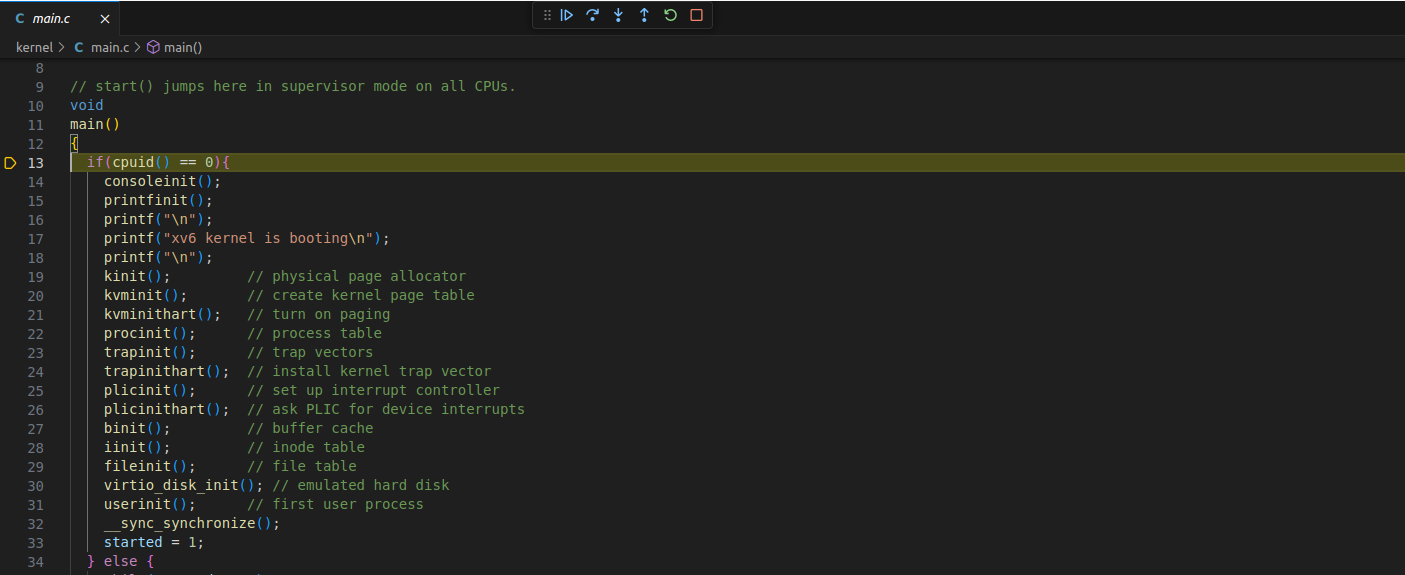


**图1. 2 tasks.json 文件**

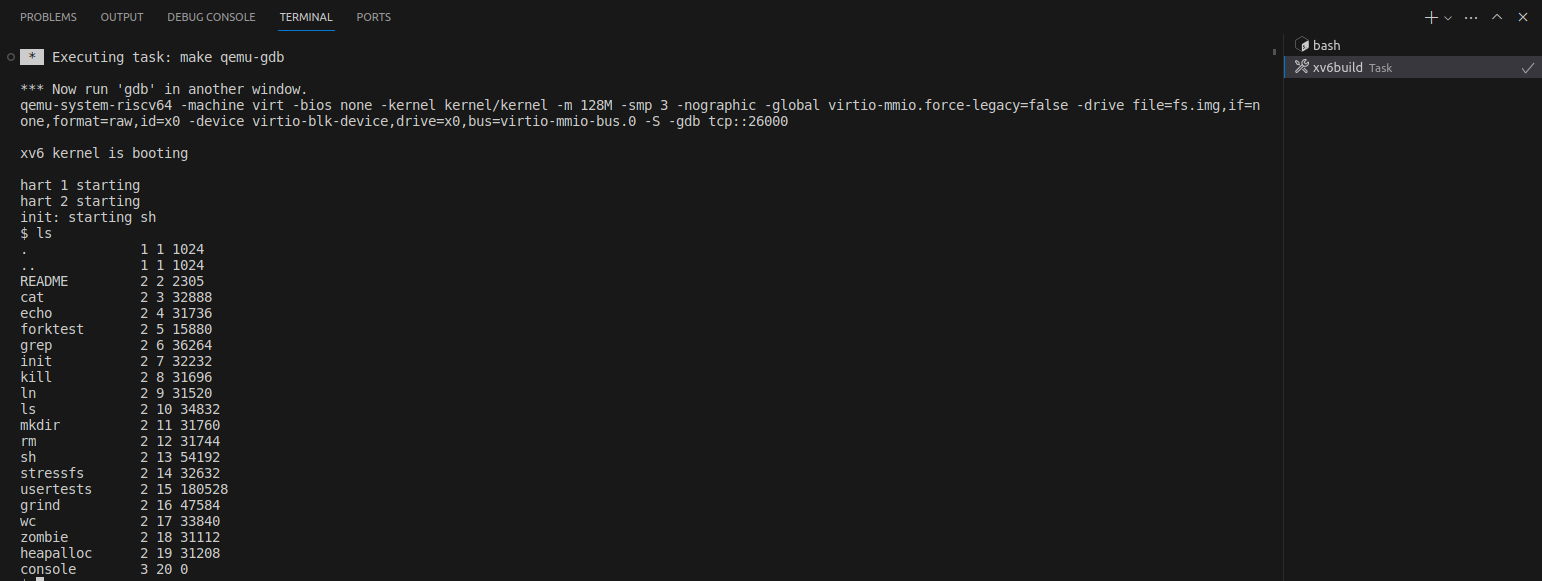
## 1.3 相关测试结果



**图1. 3 成功运行 xv6**



**图1. 4 xv6 main() 函数**

****

**图1. 5 xv6 命令行**

# 系统调用实验

## 2.1 实验要求

**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。具体来说，可以设计一个名为get\_proc\_count()的系统调用，其功能是返回当前系统中运行的进程数量。在引入该系统调用后，需要修改xv6的内核源代码，包括系统调用表、系统调用处理函数以及用户态的接口，以便用户程序可以通过调用get\_proc\_count()来获取进程数量信息。

**实验目的：**

1. 熟悉RISC-V体系结构：通过在RISC-V体系结构上操作xv6操作系统，深入了解RISC-V指令集架构及其运行环境，包括内存管理、进程调度等基本原理。
2. 理解xv6中系统调用的工作原理：通过引入新的系统调用，学习系统调用在xv6中的注册、处理和执行过程，包括用户态与内核态的切换、参数传递等关键细节。
3. 获得修改xv6源代码的经验：通过修改xv6内核源代码，包括系统调用表的更新、系统调用处理函数的编写以及用户态接口的设计，培养对操作系统内部机制的理解和修改能力，提高操作系统开发的实践能力。

## 2.2 xv6相关原理

xv6操作系统维护了一个系统调用表，其中包含了所有可用系统调用的函数指针。这个表是由kernel/syscall.c 文件中的静态数组syscalls[] 所实现的，每个系统调用都有一个对应的条目。当用户程序需要执行系统调用时，它会使用特定的指令触发一个软中断，这个软中断将控制权交给操作系统内核。

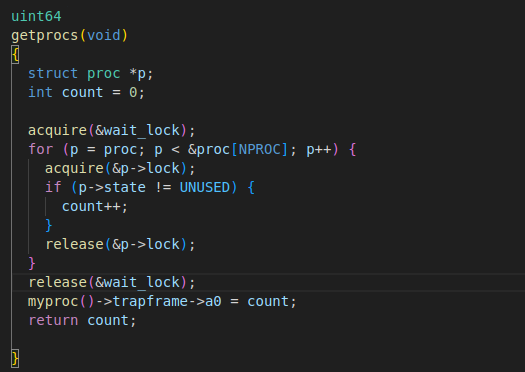
在捕获到软中断后，中断处理程序会从用户程序提供的参数中识别出要执行的系统调用，并在系统调用表中查找相应的函数指针。中断处理程序将控制权转移给系统调用函数，这样操作系统就开始执行用户请求的操作。在系统调用执行完成后，它会将结果返回给用户程序。

## 2.3 修改方案

在`user/user.h`头文件中创建一个新的系统调用函数原型。



在xv6内核中，创建一个新的C文件来实现系统调用。



在`kernel / syscall.h`文件中为自定义的系统调用分配一个唯一的系统调用号。



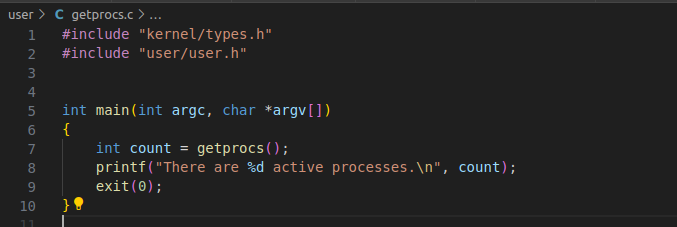
在`kernel / syscall.c`文件中修改系统调用调度器函数，以处理新的系统调用编号。

编辑`user/usys.pl`脚本以定义新系统调用的用户空间映射。

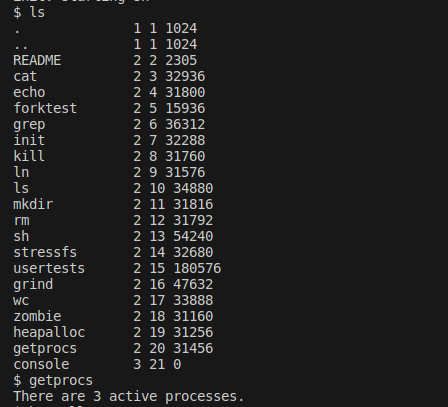


## 2.4 测试方案

在用户态编写一个程序，调用getprocs()并将计数显示到控制台。



## 2.5 相关测试结果及分析



# 3 内存管理实验

## 3.1 实验要求

**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。具体来说，首先在内核空间中的 Free memory 段落的最高地址区域开辟16MB的堆。将这16MB的空间另起炉灶，搭建一个堆，实现以字节为单位的分配释放机制。堆和 kmalloc() 并行工作，各管理内核空间中的一片区域。另外要求提供malloc()、free()函数，还需要在内核中增加一个演示函数，连续执行一系列的堆分配、堆释放，并在屏幕上输出必要堆状态信息，此外还需增加一个系统调用，应用程序通过系统调用，启动演示。

**实验目的：**

1. 认识和实践xv6操作系统内存管理方法。
2. 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 xv6相关原理

xv6 操作系统通过`kernel / kalloc.c`文件实现了一个物理内存分配器，用于分配和释放物理内存页面。物理内存页面的大小为4096字节（4KB），这是xv6中的页面大小。在该文件代码中定义了一个struct run结构体，用于表示空闲的物理内存块。每个struct run结构体包含一个指向下一个空闲块的指针。另外，在kinit()函数中，系统初始化了内存分配器。它通过计算得到要分配的空间大小，并从物理内存的末尾开始分配。然后调用freerange()函数将物理内存空间划分成页面，并加入到空闲列表中。

而kalloc() 函数用于从空闲列表中分配一个物理内存页面。它首先获取锁以确保线程安全，然后从空闲列表中取出一个空闲块，并将其从空闲列表中移除。最后释放锁并返回该页面的地址。

kfree() 函数则是用于释放一个物理内存页面。它接收一个指向要释放页面的指针，然后将该页面重新加入到空闲列表中。

在分配和释放内存时，xv6 使用memset()函数将分配的内存页面填充为特定的垃圾数据，以便在程序出错时能够更容易地发现问题。

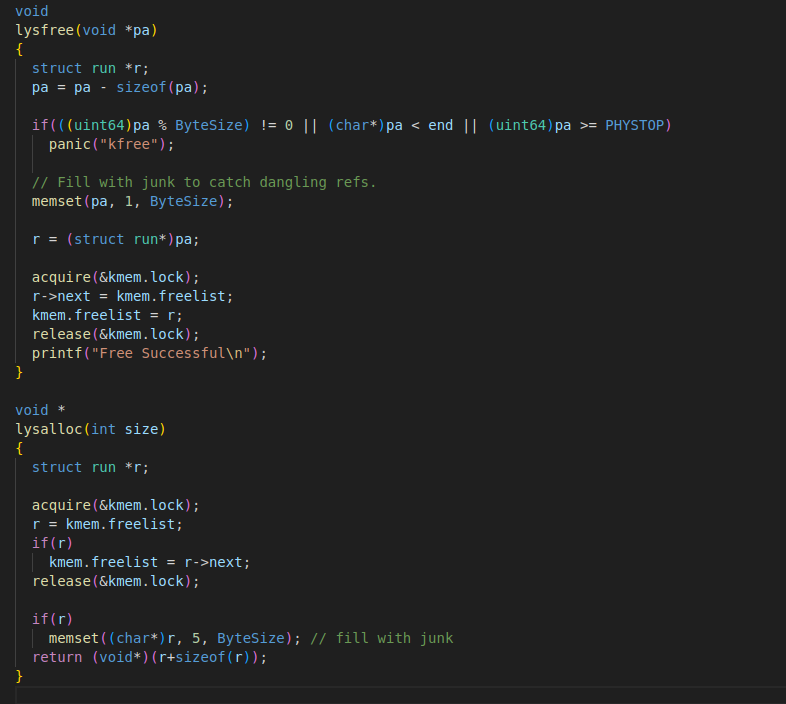
另外，为了确保对空闲列表的访问是线程安全的，xv6 操作系统使用了自旋锁来保护对空闲列表的访问。

## 3.3 修改方案

在`kernel / kalloc.c` 文件中添加了用于堆分配和释放的alloc()函数和free()函数，其中alloc() 函数的实现步骤如下：

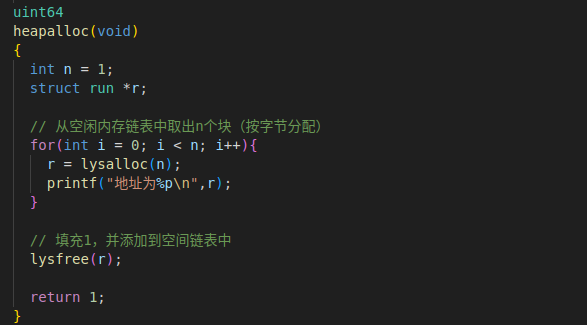
首先获取全局内存管理器 kmem 中的锁，以确保在进行内存分配时不会被其他线程中断或修改数据。然后将全局内存管理器中的自由内存链表头指针 kmem.freelist 赋值给一个指向 struct run 结构体的指针r。如果 r 不为空，说明存在可用的空闲内存块，则将全局内存管理器的自由内存链表头指针指向 r 指向的下一个内存块，即将 r 从空闲链表中移除。接下来释放全局内存管理器的锁，允许其他线程对内存管理器进行操作。如果成功获取了一个空闲内存块，则执行以下操作：`memset((char\*)r, 5, ByteSize);` 使用 memset 函数将内存块 r 的内容填充为 5，即用 5 来填充内存块的每个字节。最后返回指向分配的内存块的指针。由于 r 是指向 struct run 结构体的指针，所以 sizeof(r) 得到的是 struct run 结构体的大小。r + sizeof(r) 计算的是指向分配的内存块末尾后面一个位置的指针，然后将其强制类型转换为 void\* 类型，并返回。

free()函数的实现步骤如下：定义一个指向 struct run 结构体的指针 r，用于操作释放的内存块。对传入的指针 pa 进行调整，将其向前移动了 sizeof(pa) 个字节的距离。这个操作过减去指针大小，将 pa 指向了该内存块的头部。然后检查内存块的起始地址是否对齐于 ByteSize，如果不对齐，则触发 panic。另外还要检查内存块的起始地址是否在有效范围内，如果不在有效范围内，则触发 panic。接下来使用 memset 函数将内存块 pa 的内容填充为 1，即用 1 来填充内存块的每个字节，这样可以捕捉到悬空引用。并将调整后的 pa 转换为 struct run 结构体指针，以便后续操作。获取全局内存管理器 kmem 中的锁，以确保在进行内存释放时不会被其他线程中断或修改数据。将要释放的内存块 r 的下一个指针指向当前自由内存链表的头指针，实现将该内存块加入到自由内存链表中。更新全局内存管理器的自由内存链表头指针，将其指向释放的内存块 r。释放全局内存管理器的锁，允许其他线程对内存管理器进行操作。最后打印释放成功的消息。



## 3.4 测试方案

在`kernel / proc.c` 文件中定义一个测试函数，用于测试对堆的内存分配和释放，代码实现见图。



另外添加了一个名为 `heapalloc`的系统调用，用于调用测试函数并将相关结果显示到控制台。

## 3.5 相关测试结果及分析



# 参考文献

1. xv6: A Simple, Unix-Like Teaching Operating System. Russ Cox , Frans Kaashoek , Robert Morris. September 5, 2022.