前言

1. 说明本项目需要用到x86elf 编译工具链，qemu (其他虚拟机没有添加支持，可以自行完善）目前仅支持在 Macos 上运行（下一轮迭代计划支持 Linux、Windows ）
2. 本项目使用了 git 进行版本管理，每轮迭代都有清楚标注
3. 工具链安装教程

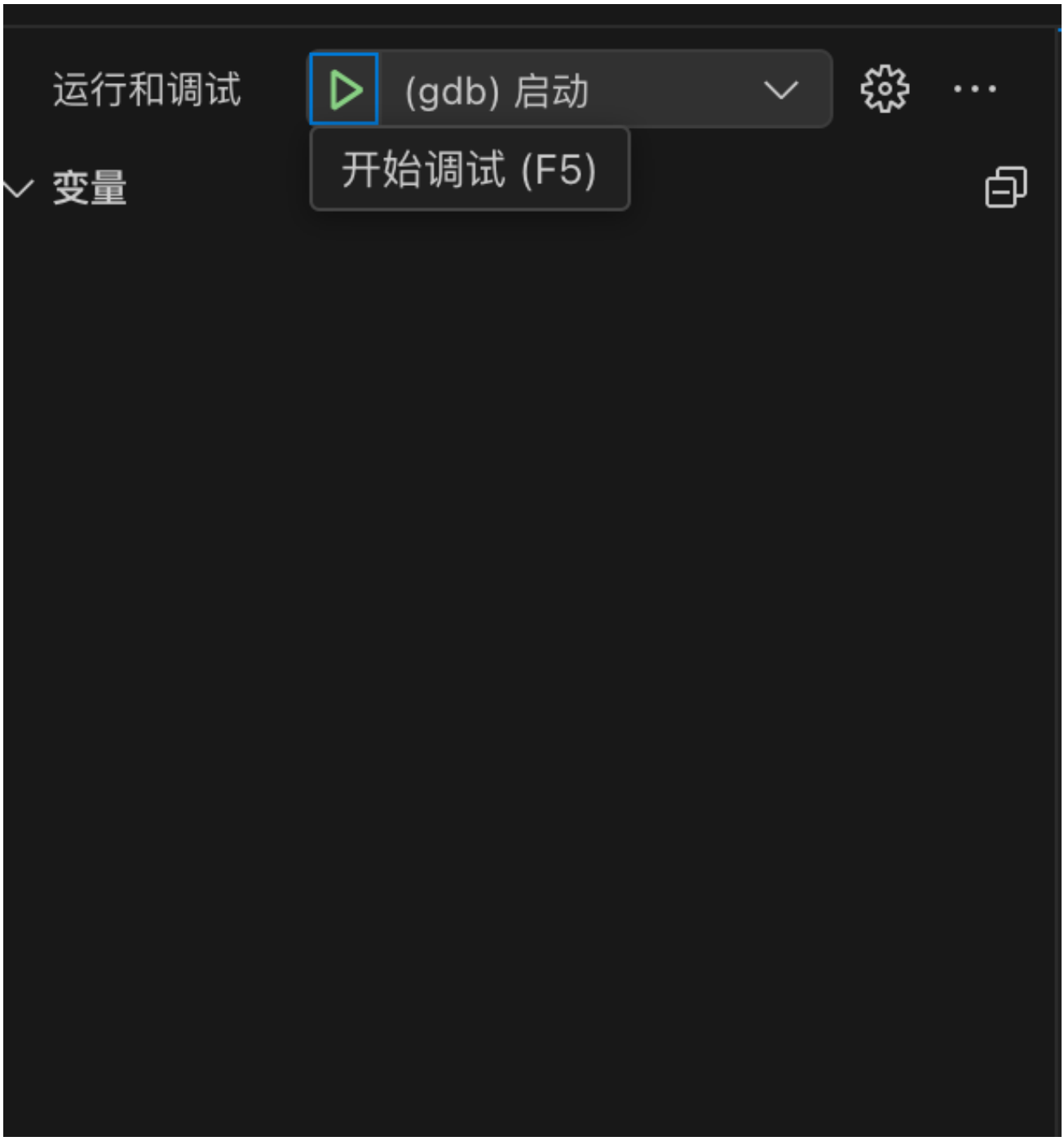
```
brew install x86_64-elf-gcc
brew install x86_64-elf-gdb
brew install qemu-system-i386
```

3. 关于调试：调试建议使用 vscode，相应的配置文件已经写好（下一轮迭代预计支持gdb 一键调试）
4. 关于真机运行：由于目前的较新的电脑都采用了 UEFI 而非 BIOS，故只有能使用 bios 的电脑可以直接裸机运行
5. 关于使用方法具体步骤：

```
make
make start
```

然后在 vscode 里面按 F5

或者点击 gdb 启动即可



更新内容

内存管理系统文档

1. 系统概述

内存管理系统是操作系统的核心组件之一，负责管理和分配物理内存资源。本系统实现了基本的物理内存管理和虚拟内存管理功能，包括页面分配、内存映射、以及动态内存分配等功能。

1.1 主要功能

- 物理页面的分配与回收
- 基于位图的物理内存管理

- 动态内存分配（堆管理）
- 虚拟内存映射

1.2 系统结构

```
内存管理系统
├── 物理内存管理 (PMM)
│   ├── 位图分配器
│   ├── 页面分配
│   └── 页面回收
├── 虚拟内存管理 (VMM)
│   ├── 页表管理
│   ├── 地址映射
│   └── TLB管理
└── 堆内存管理
    ├── 块分配
    ├── 块合并
    └── 内存对齐
```

2. 核心数据结构

2.1 内存管理器 (memory_manager_t)

```c
typedef struct memory_manager {
    uint32_t total_pages;      // 总的物理页面数
    uint32_t free_pages;       // 空闲页面数
    uint8_t *bitmap;           // 位图，用于记录页面使用情况
    uint32_t bitmap_length;    // 位图长度（字节）
    uint32_t kernel_page_dir;  // 内核页目录物理地址
} memory_manager_t;
```

2.2 内存块头部 (block_header_t)

```c
typedef struct block_header {
    uint32_t size;           // 块大小（不包括头部）
    uint8_t is_free;         // 是否空闲
    struct block_header* next; // 下一个块
} block_header_t;
```

3. API 详细说明

3.1 页面管理 API

alloc_page

```c
uint32_t alloc_page(void);
```

功能描述： 分配一个物理页面
参数： 无
返回值：

- 成功：返回分配的页面的物理地址
- 失败：返回 0
  实现细节：

1. 检查是否还有空闲页面
2. 在位图中查找第一个空闲页面
3. 标记该页面为已使用
4. 更新空闲页面计数
5. 返回页面的物理地址

free_page

```
void free_page(uint32_t page_addr);
```

功能描述： 释放一个物理页面
参数：

- page_addr: 要释放的页面的物理地址
  返回值： 无
  实现细节：

1. 检查地址是否页对齐
2. 将页面在位图中标记为空问
3. 更新空闲页面计数

3.2 堆内存管理 API

kmalloc

```
void* kmalloc(uint32_t size);
```

功能描述： 分配指定大小的内存
参数：

- size: 请求分配的内存大小（字节）
  返回值：
- 成功：返回分配的内存地址
- 失败：返回 NULL
  实现细节：

1. 对齐请求的大小到4字节边界
2. 使用最佳适配算法查找合适的空闲块

3. 如果需要，分割大块以减少内存碎片
4. 必要时扩展堆空间
5. 返回分配的内存地址

kfree

```
void kfree(void* ptr);
```

功能描述： 释放之前分配的内存
参数：

- ptr: 要释放的内存地址
  返回值： 无
  实现细节：

1. 获取并验证块头部信息
2. 标记块为空闲
3. 合并相邻的空闲块
4. 更新相关数据结构

4. 内存布局

4.1 物理内存布局

```
+------------------+ 0x00000000
|      BIOS区域      |
+------------------+ 0x00007C00
|    引导扇区代码     |
+------------------+ 0x00007E00
|    加载器代码       |
+------------------+ 0x00010000
|     内核代码       |
+------------------+ 0x00100000
|     内存位图       |
+------------------+ 位图结束
|     动态分配区域    |
+------------------+ 最大物理地址
```

4.2 虚拟内存布局

```
+------------------+ 0x00000000
|     内核空间       |
+------------------+ 0x40000000
|     用户空间       |
+------------------+ 0xFFFFFFFF
```

5. 错误处理

5.1 常见错误情况

1. 内存耗尽
2. 地址未对齐
3. 无效的释放操作
4. 页表项不存在

5.2 错误处理策略

- 分配失败时返回NULL或0
- 对无效操作进行忽略处理
- 保持系统稳定性为首要目标

6. 优化建议

6.1 当前限制

1. 简单的最佳适配算法可能导致碎片
2. 未实现内存压缩
3. 缺乏内存使用统计
4. 未实现缓存机制

6.2 改进方向

1. 实现内存池，提高小块内存分配效率
2. 添加内存压缩机制
3. 实现页面置换算法
4. 添加内存使用统计和监控功能
5. 实现缓存机制
6. 优化内存对齐策略
7. 添加内存保护机制

7. 使用示例

7.1 基本内存分配

```
// 分配一个页面
uint32_t page = alloc_page();
if (page == 0) {
    // 处理分配失败
}
```

```

// 释放页面
free_page(page);

// 动态分配内存
void* buffer = kmalloc(1024);
if (buffer == NULL) {
    // 处理分配失败
}

// 使用内存
memset(buffer, 0, 1024);

// 释放内存
kfree(buffer);
```

7.2 地址映射

```
// 映射虚拟地址到物理地址
int result = map_page(virtual_addr, physical_addr, PTE_P | PTE_W);
if (result != 0) {
    // 处理映射失败
}
```

8. 调试指南

8.1 调试技巧

1. 使用内存检查点验证分配状态
2. 检查位图一致性
3. 验证页表映射正确性
4. 监控内存使用模式