## 考试时间：14:00~16:30

考虑到有可能存在**VPN**连接不稳定的情况，建议同学们尽量在本地完成，再**copy**到跳板机相关文件里，避免中途断连影响代码编写。

# lab5-1-exam

## 创建并切换分支

```
git checkout lab5
git add .
git commit --allow-empty -m "save my lab5"
git checkout -b lab5-1-exam
```

## 注意事项

- 你需要先在`fs/fs.h`中加入相关函数的定义，之后在`fs/ide.c`中实现这些函数

## 题目描述

### 第一部分：获取系统时间

#### 获取时间函数**time_read**

- 函数原型：`int time_read()`
- 函数描述：使用已有的`syscall_read_dev`和`syscall_write_dev`系统调用，对时钟设备相关地址进行读写，获取当前的UNIX标准时间并作为返回值返回。
- `Gxemul Real-Time Clock`映射如下表：(Physical Address: **0x15000000**)

| Offset | Effect |
| --- | --- |
| 0x0000 | Read or Write: Trigger a clock update (a gettimeofday() on the host). |
| 0x0010 | Read: Seconds since 1st January 1970 |

> 1. UNIX标准时间是1970年1月1日至今的时间，单位为秒，是一个十位整数，前三位数为165
> 2. 如果获取的时间始终为0，请先通过读写触发时钟更新

# 第二部分：简易磁盘阵列

我们希望你能实现对一个简易磁盘阵列的读写操作。该磁盘阵列类似于Raid 0，包含2个磁盘，数据以扇区为单位按序存储在两块磁盘上。(对于如何挂载多块磁盘镜像，见本地测试部分）

具体需要实现的函数如下：

## 磁盘阵列写入函数**raid0_write**

- 函数原型：`void raid0_write(u_int secno, void *src, u_int nsecs)`
- 参数含义：
  - `secno` 起始扇区号，`*src` 数据来源首地址，`nsecs` 写入的总扇区个数。
  - 即：待写入的扇区范围为`[secno, secno+nsecs)`，扇区范围对应的数据源地址范围为`[src, src+nsecs*0x200)`
- 函数描述：写入数据时，将扇区号为偶数（2k）的数据写入1号磁盘的k号扇区，将扇区号为奇数（2k+1）的数据写入2号磁盘的k号扇区。
- 你可以使用`ide_write`，每次向其中一个磁盘写入一个扇区的数据。

## 磁盘阵列读取函数**raid0_read**

- 函数原型：`void raid0_read(u_int secno, void *dst, u_int nsecs)`
- 参数含义：
  - `secno` 起始扇区号，`*dst` 数据目标首地址，`nsecs` 读取的总扇区个数。
  - 即：待读取的扇区范围为`[secno, secno+nsecs)`，扇区范围对应的数据目标地址范围为`[dst, dst+nsecs*0x200)`
- 函数描述：在读取数据时，如果待读取数据的扇区号为偶数（2k），则读取1号磁盘的k号扇区，如果待读取数据的扇区号为奇数（2k+1），则读取2号磁盘的k号扇区。（写入过程的逆过程）
- 你可以使用`ide_read`，每次从其中一个磁盘读出一个扇区的数据。

# 评测逻辑与样例说明

在测试中，我们仅保留`fs/fs.h`和`fs/ide.c`两个文件，其余文件内容将被忽略。我们将只启用一个用户空间的测试进程来调用相关函数，文件系统进程不会启动。评测具体包含以下测试点：

**测试点1：获取系统时间测试（40分）**

**测试点2：磁盘阵列读写测试（30+30分）**

- 调用`raid0_write`或`ide_write`，写入数据
- 调用`raid0_read`读取磁盘数据，查看读取是否正确
- 调用`ide_read`，检查`raid0_write`写入的数据是否正确

- 调用`ide_write`，检查`raid0_read`能否正确读取数据

# 本地测试

新增测试文件

- 在`fs/`目录下新增`exam.c`文件，将测试程序拷贝进去，测试程序已随题目下发，在`exam.txt`中（`exam.txt`中其中包含三个部分，分别是 编译运行命令 、 测试程序 和 期望输出 ）

生成磁盘镜像

- 修改`fs/Makefile`文件，创建额外两个空的磁盘镜像`fs1.img`、`fs2.img`，具体在构建目标`fs.img: $(FSIMGFILES)`下新增两行（注意新磁盘的`count=64`）

```
dd if=/dev/zero of=../gxemul/fs1.img bs=4096 count=64 2>/dev/null
dd if=/dev/zero of=../gxemul/fs2.img bs=4096 count=64 2>/dev/null
```

修改相关文件

- 在`fs/Makefile`的构建目标`all:`后新增`exam.x`和`exam.b`；
- 在`fs/Makefile`的构建目标`fs.img: $(FSIMGFILES)`后新增上述生成磁盘镜像的命令；
- 在`init/init.c`的`mips_init()`中注释掉其余进程，新增`ENV_CREATE(fs_exam)`以运行测试程序；

编译运行

- 通过如下语句可以在启动MOS操作系统时挂载多个磁盘镜像，其中的0、1、2指定了不同磁盘的磁盘ID，0号磁盘为文件系统所用磁盘，1、2号磁盘本题所用磁盘。

```
make clean && make && /OSLAB/gxemul -E testmips -C R3000 -M 64 -d 0:gxemul/fs.img -d 1:gxemul/fs1.img -d 2:gxemul/fs2.img gxemul/vmlinux
```

# 代码提交

```
git add .
git commit -m "xxxxx"
git push origin lab5-1-exam:lab5-1-exam
```

# lab5-1-Extra

# 创建并切换分支

```
git checkout lab5 或 git checkout lab5-1-exam
git add .
git commit --allow-empty -m "save my lab5"
git checkout -b lab5-1-Extra
```

# 题目描述

请你实现一个类RAID4的磁盘阵列，在保留MOS操作系统原有功能的同时，实现对磁盘阵列的读取、写入操作，并在磁盘发生损坏时能够尝试进行数据恢复。

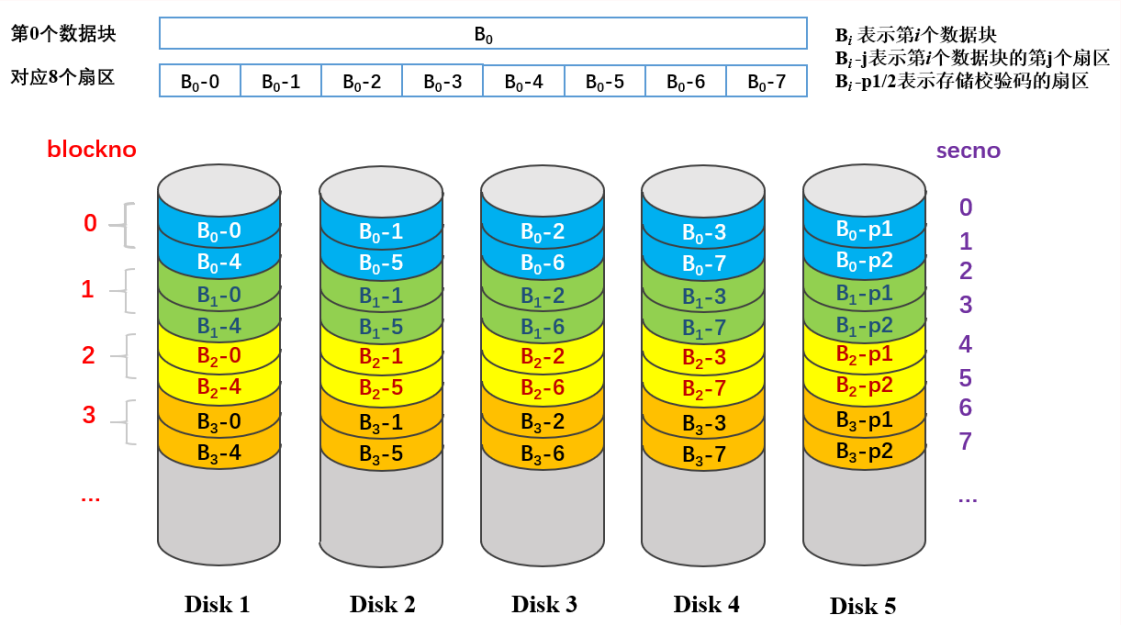
## 类**RAID4**磁盘阵列介绍

设有**n+1**个磁盘。

在RAID4中，数据以块为单位分布在各个磁盘上。其中前n块磁盘存储数据，第n+1块磁盘存储其余n个磁盘同级数据块（同级：不同磁盘的同一柱面、同一扇区）的奇偶校验码。

基于MOS文件系统的特点和评测的方便，我们在RAID4的基础上进行修改，产生类RAID4磁盘阵列。

在类RAID4磁盘阵列中，数据以扇区为单位分布在各个磁盘上。为了与MOS操作系统“1个块包含8个扇区”对齐，我们将n定义为4（即共5个磁盘）。这样对于每个数据块（大小BY2PG），它的数据以及校验码恰好可以放在5个磁盘的2个同级扇区内。如图所示：




本磁盘阵列中，校验码采用异或校验，由其他磁盘的同级扇区异或得到（$S_5 = S_1 \bigoplus S_2 \bigoplus S_3 \bigoplus S_4$，其中$S_i$表示第$i$个磁盘上的扇区）。得益于异或的性质，当一个磁盘发生损坏时，可以由其他磁盘的数据计算得到该磁盘的数据。而在测试中，我们将通过不挂载特定磁盘来模拟磁盘的损坏。

> 异或的性质：
> $A \oplus B \oplus C = D \Leftrightarrow A \oplus B \oplus D = C \Leftrightarrow A \oplus C \oplus D = B \Leftrightarrow B \oplus C \oplus D = A$
>
> 请你实现相关函数，能够查看磁盘是否损坏，实现对磁盘阵列数据的读取和写入，保证在损坏磁盘个数不超过1个时磁盘的正常读取和写入。

具体需要实现的函数如下：

## 磁盘状态检查函数raid4_valid

- 函数原型：`int raid4_valid(u_int diskno)`
- 函数描述：判断磁盘ID为`diskno`的磁盘是否有效。

> 在`ide_read`和`ide_write`中，对磁盘执行一次读写操作后，会获取读写操作的状态`status`，如果为**0**，表示读写存在异常。在本实验中，理论上只会出现未挂载磁盘导致的异常，因此可以通过该状态判断磁盘是否有效。

- 参考步骤：参考`ide_read`，使用系统调用，选择磁盘ID、指定磁盘读取偏移量为0、执行磁盘读取操作、获取上一次操作的状态，此时得到的状态可以判断磁盘的有效性。
- 返回值：磁盘有效时（磁盘已挂载）返回1，无效时（磁盘未挂载）返回0。
- 如果磁盘ID对应的磁盘不存在，`gxemul`会在屏幕上输出错误信息`[ diskimage_access(): ERROR: trying to access a non-existant IDE disk image (id x)`，这是正常现象，不会导致系统崩溃。

## 磁盘阵列写入函数raid4_write

- 函数原型：`int raid4_write(u_int blockno, void *src)`
- 函数描述：将`src ~ src + BY2PG`的数据写入到5个磁盘组成的磁盘阵列中，其中数据写入1~4号磁盘的相应扇区，计算得到的校验码写入5号磁盘的相应扇区。
- 磁盘损坏情况：
  - 无磁盘损坏：将数据和校验码写入对应磁盘扇区，返回0。
  - 有磁盘损坏：将数据和校验码写入对应磁盘扇区，如果对应磁盘无效，则不写该磁盘，其余磁盘按其该写入的内容照常写入。返回值为磁盘损坏个数。
- 你可以使用`ide_write`和`raid4_valid`来简化部分操作。

## 磁盘阵列读取函数raid4_read

- 函数原型：`int raid4_read(u_int blockno, void *dst)`

- 函数描述：从5个磁盘组成的磁盘阵列中读取一个数据块大小（BY2PG）的数据到目标空间`dst ~ dst + BY2PG`。`blockno`为数据块序号，与各个磁盘的扇区号从0开始建立对应关系，即数据块`blockno`对应的磁盘阵列每个磁盘的扇区号为`[2 * blockno, 2 * blockno + 1]`，如上方介绍中的图。
- 磁盘损坏情况：
  - 无磁盘损坏：将数据读取到目标空间，读取的同时计算校验码。如果校验码正确，返回0；反之，返回-1。
  - 1块磁盘损坏：如果校验码所在磁盘（即5号磁盘）损坏，则拷贝数据到目标空间；如果数据所在磁盘损坏，则通过校验码计算出损坏数据，拷贝完整的数据到目标空间。返回值均为1。
  - 多块磁盘损坏：返回值为磁盘损坏个数，除此之外不需要读取扇区数据。
- 你可以使用`ide_read`和`raid4_valid`来简化部分操作。
- 评测主要有3个测试点，你可以分别实现来获得相应的分数，参考实现结构：

```
invalid = 磁盘损坏个数;
if (invalid == 0) {          // 无磁盘损坏，对应测试点1，40分

} else if (invalid > 1) {    // 多块磁盘损坏，对应测试点2，20分

} else {                     // 一块磁盘损坏，对应测试点3，40分

}
```

# 注意事项

- 你需要先在`fs/fs.h`中加入以上函数的定义，之后在`fs/ide.c`中实现这些函数，写在其余文件的内容在评测时将被忽略。
- 建议在`raid4_read/write`里先使用`raid4_valid`函数来判断磁盘是否被挂载，而不是修改`ide_read/write`函数来获得错误反馈
- 对于局部数组，如有必要，可以使用`user_bzero`，避免未初始化带来的错误。

# 评测逻辑与样例说明

在测试中，我们仅保留`fs/fs.h`和`fs/ide.c`两个文件，其余文件内容将被忽略。我们将只启用一个用户空间的测试进程来调用相关函数，文件系统进程不会启动。评测具体包含以下三个测试点：

## 测试点**1**：无磁盘损坏的磁盘交互测试（**40**分）

- 调用`raid4_valid`验证有效性；调用`raid4_write`检验返回值；调用`ide_read`检验写入值是否正确；调用`raid4_read`检验数据和返回值；调用`ide_write`函数，修改磁盘阵列中部分磁盘扇区的值，测试校验码是否生效。

### 测试点2：多块磁盘损坏测试（20分）

- 调用`raid4_valid`验证有效性；调用`raid4_write`检验返回值；调用`ide_read`检验写入值是否正确；调用`raid4_read`检验返回值。

### 测试点3：一块磁盘损坏恢复测试（40分）

- 调用`raid4_valid`验证有效性；调用`raid4_write`检验返回值；调用`ide_read`检验写入值是否正确；调用`raid4_read`检验数据和返回值，检验数据是否正常恢复。

# 本地测试

新增测试文件

- 在`fs/`目录下新增`extra.c`文件，将测试程序拷贝进去，测试程序已随题目下发，在`extra.txt`中（`extra.txt`中其中包含三个部分，分别是 编译运行命令 、 测试程序 和 期望输出 ）

修改相关文件

- 在`fs/Makefile`的构建目标`all:`后新增`extra.x`和`extra.b`；
- 参考exam中的做法，在`fs/Makefile`的构建目标`fs.img: $(FSIMGFILES)`后新增生成磁盘镜像的5条命令，创建额外五个空的磁盘镜像`fs1.img ~ fs5.img`；
- 在`init/init.c`的`mips_init()`中注释掉其余进程，新增`ENV_CREATE(fs_extra)`以运行测试程序；

编译运行

- 通过如下语句可以在启动MOS操作系统时挂载多个磁盘镜像，其中0号磁盘为文件系统所用磁盘，1~5号磁盘本题所用磁盘。
- 在执行运行命令前，请务必重新编译：`make clean && make`。
- 运行指令：（你需要在以下几种情况下均能成功结束）

```
# 情况一：无磁盘损坏
/OSLAB/gxemul -E testmips -C R3000 -M 64 -d 0:gxemul/fs.img -d 1:gxemul/fs1.img -d 2:gxemul/fs2.img -d 3:gxemul/fs3.img -d 4:gxemul/fs4.img -d 5:gxemul/fs5.img gxemul/vmlinux
# 情况二：损坏多块磁盘(如3、4号)
/OSLAB/gxemul -E testmips -C R3000 -M 64 -d 0:gxemul/fs.img -d 1:gxemul/fs1.img -d 2:gxemul/fs2.img -d 5:gxemul/fs5.img gxemul/vmlinux
# 情况三：损坏1块数据盘(如1号)
/OSLAB/gxemul -E testmips -C R3000 -M 64 -d 0:gxemul/fs.img -d 2:gxemul/fs2.img -d 3:gxemul/fs3.img -d 4:gxemul/fs4.img -d 5:gxemul/fs5.img gxemul/vmlinux
# 情况四：损坏校验盘(5号)
/OSLAB/gxemul -E testmips -C R3000 -M 64 -d 0:gxemul/fs.img -d 1:gxemul/fs1.img -d 2:gxemul/fs2.img -d 3:gxemul/fs3.img -d 4:gxemul/fs4.img gxemul/vmlinux
```

# 代码提交

```
git add .
git commit -m "xxxxx"
git push origin lab5-1-Extra:lab5-1-Extra
```