操作系统内核设计赛设计文档（决赛）

成员组成

- 学校：中国科学院大学
- 队名：C-core
- 队员：
  - 王函
  - 裴晓坤
  - 黄天羿

概述

本次决赛第一阶段中，我们需要在初赛操作系统的基础上分别采用静态链接和动态链接方式运行官方仓库给出的 **libc-test**（testsuits-for-oskernel/libc-test at libc-test · oscomp/testsuits-for-oskernel (github.com)）测试的指定部分功能。该测试使用 riscv64-musl 交叉工具链（Cross-compilation toolchains for Linux - riscv64 toolchains (bootlin.com)）编译，musl库版本为1.2.0。由于本阶段远端评测机的K210板卡突然无法使用，本次采用 qemu (ver 4.2.1) 进行测试与评判，具体方法为制作包含测试文件的FAT32文件系统的SD卡镜像，使用qemu的-initrd参数将其预读取在内存的64-68M位置，并通过对该位置内存读写模拟对SD卡的读写，这样做与在K210板卡上使用SD卡驱动进行读写在效果上是完全相同的，之后不再赘述。

设计实现

需要实现的系统调用

本次使用官方仓库中源码编译完成的libc-test测试用例如下（对测试本身运行的原理，此处不再过多描述，可以参见仓库源码；需要注意的是，此处产生的.exe文件实际上就是elf格式的文件）：

```
// 本次测试需要运行的主程序
runtest.exe
// 本次测试中实际包含测试用例的文件（动态/静态）
entry-dynamic.exe  entry-static.exe
// 测试需要运行的命令行列表（列出需要实现的功能与测试的使用方法；动态/静态/全部）
run-dynamic.sh run-static.sh run-all.sh
// 动态链接中entry-dynamic.exe需要使用的动态库文件
libc.so tls_align_dso.so tls_get_new-dtv_dso.so tls_init_dso.so dlopen_dso.so
```

由于本次官方没有给出与初赛类似的系统调用说明文档，本次我们使用测试用例仓库说明文档中给出的方法，使用如下命令的方式导出runtest.exe和entry-static.exe中的系统调用号（由于entry-dynamic.exe是动态链接方式调用，无法导出调用号；此外由于写a7寄存器时不一定是使用系统调用的情况，以及有的时候a7寄存器由其他寄存器赋值，此处搜不到，所以必须结合objdump的内容，根据实际情况进行分析，必要时临时添加系统调用）：

```
objdump -d objfile | grep -B 9 ecall | grep "li.a7" | tee syscall.txt
```

加以整理汇总后，最终需要实现的系统调用如下：

SYS_fcntl (25) FCNTL - Linux手册页-之路教程 (onitroad.com)

SYS_ioctl (29) IOCTL - Linux手册页-之路教程 (onitroad.com)

SYS_statfs (43) / SYS_fstatfs (44) STATFS - Linux手册页-之路教程 (onitroad.com)

SYS_lseek (62) LSEEK - Linux手册页-之路教程 (onitroad.com)

SYS_readv (65) / SYS_writev (66) READV - Linux手册页-之路教程 (onitroad.com)

SYS_ppoll (73) POLL - Linux手册页-之路教程 (onitroad.com)

SYS_fstatat (79) STAT - Linux手册页-之路教程 (onitroad.com)

SYS_utimensat (88) UTIMENSAT - Linux手册页-之路教程 (onitroad.com)

SYS_exit_group (94) EXIT_GROUP - Linux手册页-之路教程 (onitroad.com)

SYS_set_tid_address (96) SET_TID_ADDRESS - Linux手册页-之路教程 (onitroad.com)

SYS_futex (98) FUTEX - Linux手册页-之路教程 (onitroad.com)

SYS_set_robust_list (99) / SYS_get_robust_list (100) GET_ROBUST_LIST - Linux手册页-之路教程 (onitroad.com)

SYS_clock_gettime (113) CLOCK_GETRES - Linux手册页-之路教程 (onitroad.com)

SYS_sched_setscheduler (119) SCHED_SETSCHEDULER - Linux手册页-之路教程 (onitroad.com)

SYS_sched_getaffinity (123) SCHED_SETAFFINITY - Linux手册页-之路教程 (onitroad.com)

SYS_kill (129) KILL - Linux手册页-之路教程 (onitroad.com)

SYS_tkill (130) TKILL - Linux手册页-之路教程 (onitroad.com)

SYS_rt_sigaction (134) SIGACTION - Linux手册页-之路教程 (onitroad.com)

SYS_rt_sigprocmask (135) SIGPROCMASK - Linux手册页-之路教程 (onitroad.com)

SYS_rt_sigtimedwait (137) SIGWAITINFO - Linux手册页-之路教程 (onitroad.com)

SYS_rt_sigreturn (139) SIGRETURN - Linux手册页-之路教程 (onitroad.com)

SYS_setsid (157) SETSID - Linux手册页-之路教程 (onitroad.com)

SYS_getrlimit (163) / SYS_setrlimit (164) GETRLIMIT - Linux手册页-之路教程 (onitroad.com)

SYS_geteuid (175) GETUID - Linux手册页-之路教程 (onitroad.com)

SYS_getegid (177) GETGID - Linux手册页-之路教程 (onitroad.com)

SYS_gettid (178) GETTID - Linux手册页-之路教程 (onitroad.com)

SYS_sysinfo (179) SYSINFO - Linux手册页-之路教程 (onitroad.com)

SYS_socket (198) SOCKET - Linux手册页-之路教程 (onitroad.com)

SYS_bind (200) BIND - Linux手册页-之路教程 (onitroad.com)

SYS_listen (201) LISTEN - Linux手册页-之路教程 (onitroad.com)

SYS_getsockname (204) GETSOCKNAME - Linux手册页-之路教程 (onitroad.com)

SYS_setsockopt (208) GETSOCKOPT - Linux手册页-之路教程 (onitroad.com)

SYS_mremap (216) MREMAP - Linux手册页-之路教程 (onitroad.com)

SYS_mprotect (226) MPROTECT - Linux手册页-之路教程 (onitroad.com)

SYS_madvise (233) MADVISE - Linux手册页-之路教程 (onitroad.com)

SYS_prlimit (261) GETRLIMIT - Linux手册页-之路教程 (onitroad.com)

SYS_membarrier (283) MEMBARRIER - Linux手册页-之路教程 (onitroad.com)

本次需要实现的即如上所示的一系列系统调用，以及支撑其功能运作的对应系统功能模块。为了实现这些系统调用对应的功能，我们对原本的系统功能模块进行了支持性适配改造与优化，同时也实现了数个全新的功能模块，从而在最后能够成功运行libc-test测试的所有测试点。在下面的部分中，我们将分功能模块支持与完善、系统优化两个方面分别阐述我们在决赛第一阶段中进行的工作。

标准库适配

在初赛阶段，我们不用关心标准库适配的问题，复赛阶段需要支持musl，在实现的过程中我们发现一个程序的运行远没有初赛直接加载一个程序那么简单，在进入main函数之前，需要进行大量的初始化工作，在此期间，也遇到了在标准库代码中出现bug但是又苦于没有良好的调试手段，导致需要根据源码和汇编代码进行调试，但是在一定程度上极大地增强了debug的能力。

我们按照自己的设计将所需要的所有信息都放置到用户栈上，在这里利用了内核管理全局空间的思想，直接拿到需要加载的进程的虚拟地址对应的内核地址进行操作，只需要按照标准库的要求进行适配即可。

功能模块支持与完善

信号(signal)模块

重要数据结构介绍

在UNIX类的操作系统当中，信号是非常重要的一个部分，信号相当于是软件实现的异步中断，因此其具有与中断相似的处理方式，对于信号的注册以及mask的设置内容比较简单，此处简单说一下内核当中注册信号的数据结构，我们在进程的PCB结构当中开辟了一个**sigaction**数组处理对应的64个信号，根据库函数得到其结构如下：

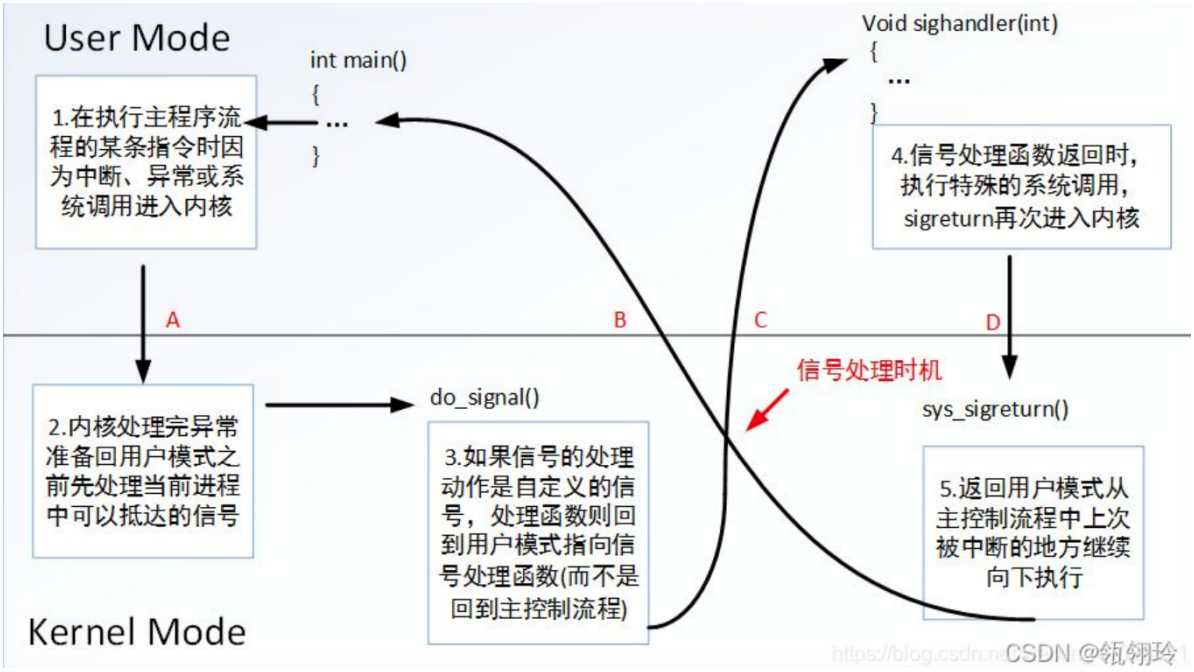
```
// kernel sigaction
typedef struct sigaction{
    void (*handler)(int);
    unsigned long flags;
    void (*restorer)(void);
    sigset_t   sa_mask;
}sigaction_t;
```

其中handler为信号处理函数，restore为退出信号函数，sa_mask为信号处理期间掩码，其代表在信号处理期间需要屏蔽的信号。

信号处理流程

在合适的时机处理信号

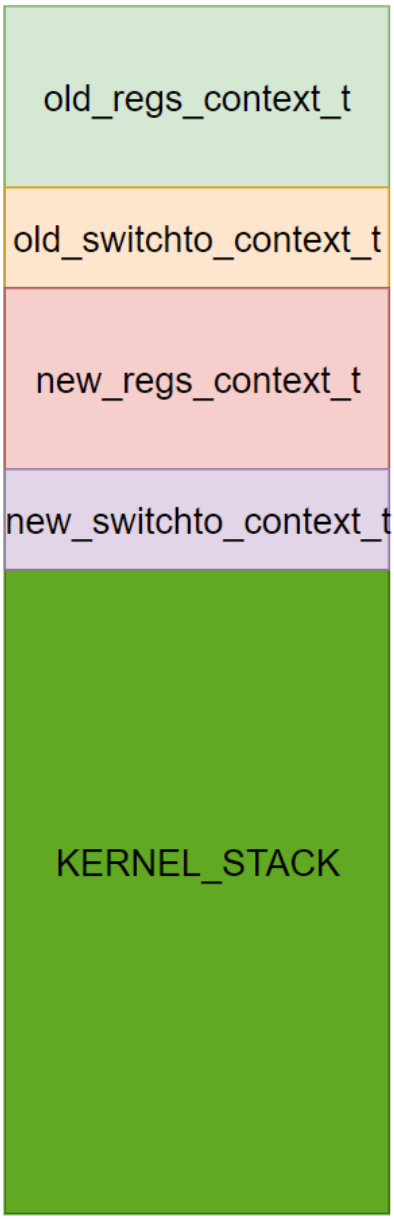
对于信号而言，我们很难做到像芯片中断一样高效，只能尽量去模拟相应的机制，因此我们选择在从内核态进入用户态的阶段调用信号处理函数，大概的流程如下图所示：



目前我们希望在一个信号的处理过程中忽略其他的信号，因此采用一个简单的变量进行标记即可。

信号上下文设置

由于信号处理函数是在用户态，因此我们需要支持在信号函数处理期间可能发生的中断以及系统调用，即对于信号处理函数我们也需要完整的上下文结构，当从内核态进入用户态时，其已经保存了此次进入用户态时需要恢复的上下文结构，我们必须对其进行保护而不能将其破坏，因此，我们在进程的内核栈上重新开辟一个新的上下文，其紧挨着原本的上下文，新的内核栈布局如下图所示：



这样便可以实现信号上下文的布置。信号上下文的初始化步骤如下：

- 初始化信号处理函数参数

对于决赛第一阶段的信号处理函数，其主要需要两个参数：信号的值以及部分用户上下文信息。为了实现的方便，我们对信号处理函数设置了新的用户栈，我们从栈顶开辟一定的空间用于存放用户上下文信息并返回用户上下文结构的地址，以及新的用户栈栈顶，具体代码如下：

```
ucontext_t *u_cxt = (ucontext_t *)(trap_sp - sizeof(ucontext_t));
// all reserve
// set value
kmemset(u_cxt, 0, sizeof(ucontext_t));
u_cxt->uc_link = u_cxt;
u_cxt->uc_sigmask.__bits[0] = current_running->sig_mask;
// // copy switch context, reserve temporarily
u_cxt->uc_mcontext.__gregs[0] = current_running->save_context->sepc;
set_u_cxt_sepc = current_running->save_context->sepc;
return ROUNDDOWN(trap_sp - sizeof(ucontext_t), 0x10);
```

主要的三个信息为当前信号掩码以及当前pc值，这样便完成了信号处理函数的参数初始化。

- 初始化上下文寄存器

为了使得进入用户态的信号处理函数能够拿到详细的参数，我们需要对信号上下文寄存器进行初始化工作，具体代码如下：

```
signal_context_buf.regs[10] = (reg_t)(i + 1);
               /* a0 = signum */
signal_context_buf.regs[11] = (reg_t)NULL;
              /* a1 is unuseful */
signal_context_buf.regs[12] = (reg_t)(SIGNAL_HANDLER_ADDR - sizeof(ucontext_t));
                                                    /* a2 is ucontext */
signal_context_buf.regs[1]  = (reg_t)(current_running->sigactions[i].restorer);
               /* should do sigreturn when handler return */

signal_context_buf.regs[2]  = new_sp;
signal_context_buf.sepc = current_running->sigactions[i].handler;
               /* sepc = entry */
```

其中关键点为：

1. 将信号处理函数的参数放置到上下文的a0, a1, a2寄存器
2. 设置sepc寄存器使得能进入信号处理函数
3. 将ra寄存器设置为restorer函数，使得信号处理函数完毕时能够自动调用sigreturn系统调用返回
4. 设置新的信号处理函数用户栈

- 设置新的内核栈顶

根据上面的分析，只需要将内核栈顶减去一个偏移量即可，代码如下：

```
current_running->kernel_sp -= offset;
current_running->save_context = (uint64_t)(current_running->save_context) -
offset;
current_running->switch_context = (uint64_t)(current_running->switch_context) -
offset;
```

- 拷贝上下文信息到内核栈

由于我们需要将上下文信息直接拷贝到内核栈上，因此利用汇编代码进行拷贝，此处不再赘述。

从信号处理函数返回

前面已经提及到我们在信号处理函数的ra寄存器中放置了restorer函数指针，因此信号处理函数处理完毕后将会自动调用sigreturn系统调用。对于目前的信号处理机制，我们只需要将内核栈重新加上一个偏移量并改变一些上下文信息即可，具体的代码如下：

```
uint64_t offset = (sizeof(regs_context_t) + sizeof(switchto_context_t));
// resize kernel_sp
uint64_t core_id = current_running->save_context->core_id;
current_running->kernel_sp += offset;
current_running->save_context = (uint64_t)(current_running->save_context) +
offset;
current_running->switch_context = (uint64_t)(current_running->switch_context) +
offset;
current_running->save_context->core_id = core_id;
// jump to next
current_running->save_context->sepc = u_cxt->uc_mcontext.__gregs[0];
```

其中的关键点在于：

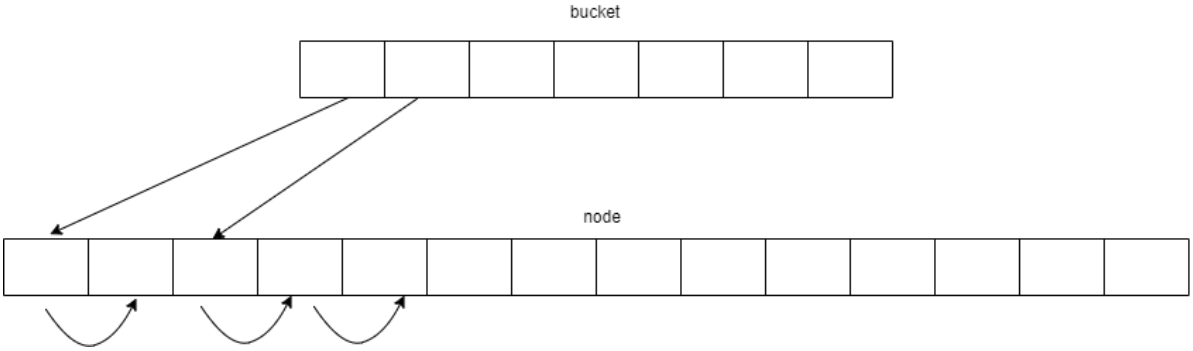
1. 信号处理函数很可能改变程序执行流，即我们可能需要重新设置原本上下文的sepc寄存器。
2. 需要将内核栈顶加上一个偏移量使得其指向进入信号处理函数之前的上下文。
3. 由于我们已经将当前上下文指针重新指向进入信号处理函数之前的上下文，因此只需要执行sret操作重新返回用户态即可。

这样便完成了整个信号处理流程。

锁(futex)模块

与该部分相关的系统调用有SYS_futex、SYS_set_robust_list和SYS_get_robust_list。

futex锁是一个很重要的点，在第一阶段的pthread中广泛使用，futex锁是锁变量位于用户态的锁，可以避免很多无用的检查锁的系统调用，重点需要实现futex锁的三个功能: WAKE、WAIT和REQUEUE。锁的结构主要是采用一个锁池（futex_node）数组，一个桶数组（用来作为hash索引），每个futex地址计算出的key对应一个桶，相同的key在一个桶的链表上。



主要结构和辅助函数有：

```
typedef struct futex_node
{
    futex_key_t futex_key;//futex的哈希值
    list_node_t list;//node list
    list_head block_queue;//进程阻塞的链表
    struct timespec add_ts;//futex wait时的系统时间
    struct timespec set_ts;//futex wait时传入的timeout
}futex_node_t;
typedef list_head futex_bucket_t;
futex_bucket_t futex_buckets[FUTEX_BUCKETS]; //HASH LIST
futex_node_t futex_node[MAX_FUTEX_NUM];
int futex_node_used[MAX_FUTEX_NUM] = {0};//记录桶的使用情况

void init_system_futex();
//初始化桶的list

static int futex_hash(uint64_t x);
//根据futex的地址计算哈希值

static futex_node_t* get_node(int *val_addr, int create);
//寻找与val_addr匹配的futex_node, create为1，则在没找到的情况下为futex申请一个futex_node

int do_futex_wait(int *val_addr, int val, struct timespec *timeout);
//进程阻塞，timeout不为0时候代表时间

int do_futex_wakeup(int *val_addr, int num_wakeup);
//唤醒futex阻塞的进程，numwakeup代表唤醒的进程个数

int do_futex_requeue(int *uaddr, int* uaddr2, int num);
//将一个锁的阻塞的进程给转移到第二个锁上，num代表转移的最大个数。
```

```
void check_futex_timeout();
//检查futex锁是否超时，如果超时释放进程（仅限于被设置time的锁）

int do_futex(int *uaddr, int futex_op, int val, const struct timespec *timeout,
  /* or: uint32_t val2 */ int *uaddr2, int val3);
//futex系统调用的主函数
```

主要实现思路是，在主函数中，根据输入的futex_op来判断进行的操作，目前测试中需要的操作有三种：WAKE、WAIT和REQUEUE：

- wake操作，直接调用do_futex_wakeup，该函数通过get_node（create为0）获取futex锁对应的node，然后遍历node的链表并unblock进程，注意unblock的个数不能超过num_wakeup。这里解释一下get_node的逻辑：首先根据虚拟地址计算hash值，然后在对应的桶的链表中寻找是否已经存在该futex对应的node，如果存在，直接返回，如果不存在，就寻找一个未使用的node，然后连接到对应的桶中，初始化参数，返回node指针。
- wait操作，首先仍然是通过get_node获取一个node，然后若timeout有效，将timeout记录在node的set_ts段中，并且将此时的时间也记录在node中，然后block进程，如果timeout无效（为0），直接block进程。在futex的超时机制中，有一个check_futex_timeout，每次时钟中断会调用一次，然后获取当前时间，遍历futex_node，寻找设置时间的futex，根据其中记录的add_ts和set_ts来判断是否超时，如果超时则释放阻塞进程，并且将node的设置时间和添加时间清零。
- requeue比较简单，寻找两个锁的node（如果未找到就创建一个），然后将第一个锁的阻塞进程转移到第二个锁上，转移的最大个数是主函数的timeout。

第二个robust锁，主要需要完成两个函数：

```
struct robust_list_head {
    struct robust_list list;
    long futex_offset;
    struct robust_list *list_op_pending;
};
struct robust_list_head *robust_list[NUM_MAX_TASK] = {0};
long get_robust_list(int pid, struct robust_list_head **head_ptr,
                     size_t *len_ptr);
long set_robust_list(struct robust_list_head *head, size_t len);
```

该部分主要是为用户态的robust锁服务。这两个系统调用的作用主要是在内核记录一下用户线程的robust，这种类型的锁主要是解决当一个持有互斥锁的线程退出之后这个锁成为不可用状态的一个问题来的。需要注意的是，get操作的pid为0的时候，得到的是当前线程的robust list链表头。这部分功能比较简单，只需要在内核开个数组记录robust_list_head的地址和对应的pid即可，get操作遍历数组，set操作寻找空数组，然后写入地址。

套接字(socket)模块

值得注意的是，本次我们需要实现的系统调用中包括了socket、bind、connect等与网络有关的系统调用，考虑到我们并没有使用到K210的网络相关功能，开始时我们对这一系列系统调用的实现颇为不解。在参照用例后，我们发现，此处的socket测试用例仅涉及到对本地回环（地址为0x7f000001，即127.0.0.1）的读写测试，所以，此处我们采用建立类似mailbox的缓冲区机制，制作了一套全新的socket系统调用。

我们定义的socket结构体如下所示：

```
struct socket {
    int domain;
    int type;
```

```
    int protocol;
    int backlog;
    uint8_t wait_list[MAX_WAIT_LIST];
    uint8_t used;
    uint8_t status;
    struct sockaddr *addr;
    struct ring_buffer data;
};

struct socket sock[MAX_SOCK_NUM + 1];
mutex_lock_t sock_lock;
```

其中，domain、type和protocol都是socket系统调用的参数，此处用于保存；backlog在listen时设置，代表最大等待个数；wait_list为一个静态数组，其中存放等待于此的socket的数组下标（0号socket不使用）；used代表该socket是否被使用；status代表socket的状态，在我们的简单实现中有CLOSED、LISTEN、ACCEPTED和ESTABLISHED四种类型；addr为指向用户绑定的socket地址结构体的指针；data为模拟传输数据使用的环形缓冲区（mailbox）。

如下是全套socket操作的函数：

```
// 分配并初始化一个socket，将对应的信息填入其中
int do_socket(int domain, int type, int protocol);
// 将socket的addr绑定在用户态指定的addr上
int do_bind(int sockfd, struct sockaddr *addr, socklen_t addrlen);
//将socket的状态置为LISTEN
int do_listen(int sockfd, int backlog);
// 连接至sockaddr对应的socket，即在所有socket中搜索sockaddr对应且状态为LISTENED的
socket，
// 并将自己放入其的wait_list中。如果该socket是非阻塞的则直接返回，否则循环等待。
int do_connect(int sockfd, struct sockaddr *addr, socklen_t addrlen);
// 用于接受socket等待列表中的连接，并创建一个新的socket指代该连接
int do_accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
// 向sockaddr对应的socket发送数据（实际上是向其缓冲区写入数据）
ssize_t do_sendto(int sockfd, void *buf, size_t len, int flags,
                  struct sockaddr *dest_addr, socklen_t addrlen);
// 从src_addr地址读取数据（实际上是读取自己的缓冲区）
ssize_t do_recvfrom(int sockfd, void *buf, size_t len, int flags,
                    struct sockaddr *src_addr, socklen_t *addrlen);
// 关闭socket连接,即重置socket状态
int32_t close_socket(uint32_t sock_num);
// 此外，还有用于获取socket对应地址的getsockname和setsockopt，由于不涉及连接，此处不表
```

此外，由于实际上对于用户来说socket与fd采用相同的方式管理，所以我们采用分配与socket一一映射的fd，并同时映射操作（如close）的方法进行适配。具体的适配方法在fd的设计中体现，改动并不大，与dup和pipe的原理类似，在此不再赘述。

文件系统模块（实现/完善）

决赛阶段，根据测试文件的需要，我们增加了一些新的功能。

- `int fcntl(int fd, int cmd, ... /* arg */ )`，主要根据输入的操作cmd，对fd进行操作，操作类型有F_GETFD，返回文件描述符的flag； F_SETFD，将文件描述符标志设置为arg指定的值。F_GETFL，返回文件访问模式和文件状态标志；F_SETFL，将文件状态标志设置为arg指定的值等等，根据cmd，这些只需要在进程pcb中的fd表中寻找对应的fd进行操作即可。
- `ssize_t readv(int fd, const struct iovec *iov, int iovcnt)`，readv()系统调用从与文件描述符fd相关联的文件中读取iovcnt缓冲区到iov描述的缓冲区中("分散输入")。可以直接在内核

复用read操作。iov是一个结构体指针，结构体存有缓冲区地址，以及缓冲区长度，iovcnt代表iov这个结构体的个数。

- `ssize_t writev(int fd, const struct iovec *iov, int iovcnt)`，writev()系统调用将iov描述的数据的iovcnt缓冲区写入与文件描述符fd相关联的文件("聚集输出")。可以直接在内核复用write操作。
- `int fstatat(int dirfd, const char *pathname, struct stat *statbuf, int flags)`，根据获取文件状态，可以借助之前初赛实现的openat函数，获取fd，然后使用fstat函数获取文件状态，之后close文件。
- `int statfs(const char * path，struct statfs * buf)`和`int fstatfs(int fd，struct statfs * buf)`，用来获取文件系统的状态，path: 位于需要查询信息的文件系统的文件路径名。fd： 位于需要查询信息的文件系统的文件描述词。buf：以下结构体的指针变量，用于储存文件系统相关的信息。直接根据buf结构体成员赋值即可，不再赘述。
- `int utimensat(int dirfd, const char *pathname, const struct timespec times[2], int flags)`，utimensat()以纳秒级精度更新文件的时间戳。使用utimensat()可以通过路径名中指定的路径名指定文件。更新的文件时间戳记设置为文件系统支持的最大值，该最大值不大于指定的时间。如果时间规范结构之一的tv_nsec字段具有特殊值UTIME_NOW，则将相应的文件时间戳记设置为当前时间。如果时间规范结构之一的tv_nsec字段具有特殊值UTIME_OMIT，则相应的文件时间戳保持不变。在这两种情况下，都会忽略相应的tv_sec字段的值。如果times为NULL，则两个时间戳都设置为当前时间。
- `ssize_t pread(int fd, void *buf, size_t count, off_t offset)`和`ssize_t pwrite(int fd, const void *buf, size_t count, off_t offset)`主要用于多线程并行时候保证原子读写，读写不冲突，但考虑到我们做了大内核锁，以及决赛第一阶段使用单核，不会产生多线程冲突，所以直接套用了原本文件读写函数。
- `off_t lseek(int fd, off_t offset, int whence)`，该函数只需要根据whence修改对应的文件描述符的文件读写偏移量即可，比较简单，不再赘述。

动态链接

动态链接是本次比赛新增的赛题，对于动态链接的程序，我们需要适配musl的动态链接器，下面将对其进行简要的介绍。

musl-c动态链接其简介

musl-c的动态链接器代码主要位于`ldso/dlstart.c`，`ldso/dynlink.c`。musl-c的动态链接器重定位主要分为三个阶段，第一阶段是`_dlstart_c`函数，第二阶段是`__dls2`和`__dls2b`函数，第三个阶段是`__dls3`函数。其余关于动态链接器的连接原理将不再进行赘述。

动态链接器加载内存布局

由于动态链接器是内存无关的代码，通过readelf命令可以查看到其虚地址起始为0，如下图所示：

```
Elf 文件类型为 DYN (共享目标文件)
Entry point 0x65b08
There are 5 program headers, starting at offset 64

程序头:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  LOAD           0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x00000000000a22a4 0x00000000000a22a4  R E    0x1000
  LOAD           0x00000000000a2cf0 0x00000000000a3cf0 0x00000000000a3cf0
                 0x00000000000007f8 0x0000000000003528  RW     0x1000
  DYNAMIC        0x00000000000a2eb0 0x00000000000a3eb0 0x00000000000a3eb0
                 0x0000000000000150 0x0000000000000150  RW     0x8
  GNU_STACK      0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000000 0x0000000000000000  RW     0x10
  GNU_RELRO      0x00000000000a2cf0 0x00000000000a3cf0 0x00000000000a3cf0
                 0x0000000000000310 0x0000000000000310  R      0x1

 Section to Segment mapping:
  段节...
   00     .hash .gnu.hash .dynsym .dynstr .rela.dyn .rela.plt .plt .text .rodata
   01     .data.rel.ro .dynamic .data .got .bss
   02     .dynamic
   03
   04     .data.rel.ro .dynamic
```

在我们的设计当中，我们将其固定加载到到**0x8000000**起始处的地址，其实这个地址应当是动态的，但是为了调试的简单，我们对其进行了特殊处理。

动态链接器加载流程

在我们的设计当中，动态链接器的加载分为如下几个步骤：

- 首先按照正常流程，将可执行程序的LOAD段加载进入虚拟内存，此处不再赘述。
- 为动态链接器设置好各项参数，包括命令行参数，环境变量，辅助数组，这些信息被我们一致地放置到用户栈上，动态链接器将会根据需要取用，这里需要进行简单的介绍。
    - 环境变量设置

      环境变量需要设置加载动态链接库的默认位置，由**“LD_LIBRARY_PATH”**指定，我们将其简单地设置为当前目录。
    - 辅助数组
        - AT_PHDR ：链接的程序的程序段的起始地址
        - AT_PHENT：链接的程序的段头大小
        - AT_PHNUM：链接程序的段头数目
        - AT_BASE：动态连接器的实际加载虚拟地址，此处为0x80000000
        - AT_ENTRY：链接程序的实际入口
- 完成这些工作之后，我们便可以将动态链接器像加载一个正常的elf一样将其加载进入内存。
- 跳转到动态连接器的入口，注意此处需要将实际的入口加上内核实际加载的虚地址。

这样，当动态链接器完成所有的工作之后，将会将控制权主动交付给需要链接的程序。

其他

除了上面的主要功能修正之外，我们的系统还实现了一些比较小的调整：

1. 对于测试样例中要求的不同的ERRNO的传递，我们直接在内核的系统调用中返回对应的负值即可实现，其他返回（如函数调用失败的-1）由库函数实现；错误与负值的对应关系体现在库函数的errno.h中，将其移植入内核后，可以通过查询系统调用的文档，确定对应情况下的返回负值。（具有代表性的如ENOENT,EMFILE和EINVAL）
2. 我们实现了对单个进程进行资源限制的rlimit系列系统调用（prlimit），目前能够用于限制进程使用文件描述符的最大数量，之后还可以进一步扩充其功能以寻求改进。

3. 为了实现pthread库以及signal、futex等模块的要求，对于clone函数，我们补充实现了flag对应的CLONE_VM、CLONE_FILES、CLONE_CHILD_SETTID等功能的处理，同时也补充了其他一系列函数的flag处理，使系统调用功能更为完备。

系统优化

share fd机制

该机制是在fd_table(后续会讲)设计之前，做的一个临时共享fd的方法，在测试fflush_exit时候，fork一个进程，然后子进程对fd写，另一个子进程对fd读，按照我们当时的设计，每个进程会有一个fd表，但这个表是每个pcb内部的私有成员，这就导致父进程和子进程的fd指向同一个文件，但却不能同步读写的pos，所以我们设计了这个share_fd，每当打开一个文件，就会申请一个share_fd，对该文件读写的时候会更新对应的share_fd，share_fd结构体里面有个版本号，来判断是否需要更新，从而保证fork出去的进程，能同步fork时候的fd。

FD_table分配机制

此机制是完善clone的时候引入的，clone的flag有个CLONE_FILES，该flag指明子进程要和父进程共享相同的文件描述符表，所以我们更改了之前私有fd表的方法，改成pcb中只记录fd表的指针，然后改成初始化pcb申请fd表的方式，这样，当clone需要共享fd表时，只需要子进程指向fd表即可，而且fd表还需要一个num记录使用的进程数，根据此来确定是否随着进程回收fd表。

bcache机制

在决赛第一阶段中，考虑到读写sd卡代价较高，我们不再使用直接的sd_read_sector和sd_write_sector，而转为移植使用xv6-k210作品中的sd cache，即bcache。其设计原理如下：

1. 使用如下的结构体作为扇区单元：

```
struct buf {
    uint32              dev;        // 块设备号
    uint32              sectorno;   // 扇区号
    uint32              refcnt;
    uint16              valid;      // 内容是否有效
    uint8               dirty;      // 是否脏
    uint8               disk;       // 是否正在进行 I/O
    list_node_t         list;       // 双向链表节点
    list_node_t         hash;       // 哈希链表节点
    sleeplock_t         lock;
    char               data[BSIZE]; // 数据
};
```

2. 维护一个buf静态数组（用于充当cache）以及一个hash链表（用于快速查找对应的buf结构体），使用一套较为复杂的机制进行bcache链表的分配、查找与管理。
3. 维护如下的对外接口：

```
// 用于初始化bcache的所有buf结构体
void            binit(void);
// 用于获取（分配）dev号设备的sectorno号扇区的对应结构体，置buf锁，返回指向该结构体的指针
struct buf*     bget(uint dev, uint sectorno);
// 用于将dev号设备的sectorno号扇区读取到一个buf中，返回指向该结构体的指针
// 函数中调用bget进行获取，如果不存在对应的buf，则直接分配一个buf
struct buf*     bread(uint dev, uint sectorno);
// 将bwrite缓冲区内的内容写回磁盘，调用brelse
void            bwrite(struct buf *b);
// 释放buf的锁，将其重新放回lru队列内（cache满的情况下可供释放与重新分配）
void            brelse(struct buf *b);
```

这些接口对应的具体实现在./drivers/sdcard/bio.c中，此处不再介绍。

4. 将对文件系统开放的disk_read与disk_write修改如下，从而与bcache模块进行交互（此处的写操作采用write through策略）：

```
uint8_t disk_write(uint8_t *data_buff, uint32_t sector, uint32_t count)
{
        for (int i = 0; i < count; i++)
        {
            struct buf *b = bget(0,sector + i);
            kmemcpy(b->data,data_buff + 512 * i,512);
            bwrite(b);
        }
        return 0;
}
uint8_t disk_read(uint8_t *data_buff, uint32_t sector, uint32_t count)
{
        for (int i = 0; i < count; i++)
        {
            struct buf *b = bread(0,sector + i);
            kmemcpy(data_buff + 512 * i,b->data,512);
            brelse(b);
        }
        return 0;
}
```

除此之外，还仿照sd_read_sector与sd_write_sector编写了在bcache实现中专门用于填充buf结构体的sd_read_sector_bio与sd_write_sector_bio，其与原本函数唯一的区别是相当于直接使用传入的buf结构体内存储的参数与缓冲区作为函数的参数。

以上即为bcache机制的介绍。虽然对于分散的随机读写而言，bcache的性能较直接使用sd卡驱动函数弱，但是对于大量集中读写的情况，经过测试，bcache的性能可以达到直接使用sd卡驱动函数的10倍以上（读写时间缩短到1/10），这是非常巨大的提升。

Fast_load_elf机制 / 回收机制

fast_load_elf机制是我们综合对可执行文件在内存中的镜像布局进行分析而得出的设计。对于fat32文件系统，如果没有对其进行一定的优化，其复杂度将会是O(n^2)，而在此前我们在k210上进行操作时，就发现从SD卡中读取并加载可执行文件的效率极其低下，甚至在测评机上直接被判定超时。考虑到本次测试主要是反复地运行几个测试程序，我们设计了新的fast_load_elf机制，下面对其设计和思想进行介绍。

Fast_load_elf主要思想——测试程序共享只读内存页面

在linux的elf文件格式当中，进行了一个分程序段的设计，一般其将只读和可执行代码放置到一个数据段，这样做有几个显著的好处：

- 独立各个数据域，方便进行权限的管理。
- 增强局部性。
- 多个进程可以共享数据段。

在此处我们主要利用多个进程共享数据段的特点。这里我们通过对entry-static.exe的数据段进行分析来展示这种设计的优点，利用readelf命令查看结果如下图所示：

```
程序头：
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  LOAD           0x0000000000000000 0x0000000000010000 0x0000000000010000
                 0x000000000009579c 0x000000000009579c  R E    0x1000
  LOAD           0x0000000000095fd0 0x00000000000a6fd0 0x00000000000a6fd0
                 0x0000000000001a58 0x0000000000006ee8  RW     0x1000
  TLS            0x0000000000095fd0 0x00000000000a6fd0 0x00000000000a6fd0
                 0x0000000000000017 0x0000000000000017  R      0x8
  GNU_STACK      0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000000 0x0000000000000000  RW     0x10
  GNU_RELRO      0x0000000000095fd0 0x00000000000a6fd0 0x00000000000a6fd0
                 0x0000000000000030 0x0000000000000030  R      0x1
```

可以发现第一个LOAD程序段占据了内存中大量的空间，相比之下，数据段仅仅只是一小部分。因此，如果能共享第一个段，每次新加载entry-static.exe时只需要新分配6个页面，可以省下数百个页面的内存读写，这对于减少IO操作和提高效率贡献巨大。

Fast_load_elf数据结构设计

由于在比赛当中多个进程共存将会导致屏幕输出杂乱，从而影响评判结果，我们新增加了一个自定义的系统调用来预加载测试程序，每个测试程序由这样一个数据结构保存。

```
typedef struct excellent_load
{
    // for debug
    char name[50];
    uint32_t used;
    // base pgdir for the exec and read_only
    uintptr_t base_pgdir;
    // save the data page page
    list_head list;
    // phdr head
    Elf64_Phdr phdr[LOAD_PHDR_NUM];
    int phdr_num;
    // elf head
    ELF_info_t elf;
    int dynamic;
} excellent_load_t;
```

需要保存详细的程序段头的信息，以及其是否为动态链接，其中list为保存这些内容的物理页面链表，在此处我们只保存file_size的大小以节约内存，在测试开始之前我们需要将其预加载到内存当中。

缓冲区预加载流程

只需要加载file_size的大小即可，代码逻辑如下：

```
while (ph_entry_count--) {
    fat32_lseek(fd, ptr_ph_table, SEEK_SET);
    fat32_read(fd, &phdr, sizeof(Elf64_Phdr));
    if (phdr.p_type == PT_INTERP)
    {
        exe_load->dynamic = 1;
    }

    if (phdr.p_type == PT_LOAD) {
        // copy phdr
        kmemcpy(&exe_load->phdr[index++], &phdr, sizeof(Elf64_Phdr));
        uint64_t length = phdr.p_filesz;
        uint64_t al_length = 0;
        uint64_t fd_base = phdr.p_offset;
        // the ptr
        fat32_lseek(fd, phdr.p_offset, SEEK_SET);
        while (length)
        {
            uint64_t offset_remain = ROUND(fd_base, 0x1000) - fd_base;

            char * buffer = kmalloc(PAGE_SIZE);
            page_node_t* alloc =  &pageRecyc[GET_MEM_NODE((uintptr_t)buffer)];
            // add list
            list_del(&alloc->list);
            list_add_tail(&alloc->list, &exe_load->list);
            uint32_t read_length;
            if (offset_remain)
                read_length = MIN(offset_remain, phdr.p_filesz - al_length);
            else
                read_length = ((phdr.p_filesz - al_length) >= PAGE_SIZE) ?
PAGE_SIZE :
            phdr.p_filesz - al_length;

            fat32_read(fd, buffer, read_length);
            fd_base += read_length;
            al_length += read_length;
            length -= read_length;
        }
        exe_load->elf.edata = phdr.p_vaddr + phdr.p_memsz;
        exe_load->phdr_num++;
        //
        if (is_first) {
            exe_load->elf.text_begin = phdr.p_vaddr;
            is_first = 0;
        }
    }
    ptr_ph_table += ph_entry_size;
}
```

加载一个新的程序

对于只读数据区，我们直接将指定的物理地址映射到对应的虚拟内存而不重新分配页面，通过**alloc_page_point_phyc**函数实现该功能，代码如下：

```
if (!(exe_load->phdr[index].p_flags & PF_W) && !exe_load->dynamic)
{
    alloc_page_point_phyc(v_base, pgdir, buffer_begin, MAP_USER);
    bytes_of_page = (uchar *)buffer_begin;
} else
{
    // this will never happen for the first phdr
    // else alloc and copy
    bytes_of_page = alloc_page_helper(v_base, pgdir, MAP_USER);
    kmemcpy(
        bytes_of_page,
        buffer_begin,
        MIN(exe_load->phdr[index].p_filesz - i, NORMAL_PAGE_SIZE));
}
```

对于可写数据区，我们直接用内存复制命令进行操作，此处不再赘述。按照我们在k210上测试的经验，速度提升达到数十倍以上，内存占用上也更加节省。

回收一个进程

对于进程的回收，我们只需要遍历三级页表，读取相应数据结构中保存的程序头信息，如果回收的虚拟地址属于程序的只读数据区则将其保留，否则将其回收.

Lazy_mmap机制

为了尽可能减少对磁盘的读写与内存使用，我们还设计了Lazy mmap机制，即在使用mmap系统调用时，不直接分配页面，而是对其进行标记，等到实际读取/写入该页时，再在缺页处理程序中进行拷贝，这种思路与fork的copy_on_write类似。在mmap系统调用中，当map类型为MAP_ANONYMOUS时，进行lazy_brk，即只修改结构体数据而不实际调用alloc_page_helper分配页面（此时相当于分配内存，可以缺页时再分）；而如果使用文件映射，则将该fd的mmap做上标记而不直接拷贝（start=0时同样使用lazy_brk），在缺页处理时分配完新页后遍历fd搜索当前页是不是某个fd的映射范围，如果是，则此时再从fd的对应位置拷贝内容。同样地，在munmap时，只需要对分配了的页进行写回即可。这样做可以在很大程度上节约内存空间，也能够在频繁使用mmap时显著提升处理性能。

问题与解决方案

上板调试较为困难

本次测试的前一阶段仍然使用K210进行上板测试，存在的问题和初赛类似，即没有类似于GDB的调试工具，只能够通过printk等方法来寻找bug，这样的难度随着系统规模的上升在逐渐提高，最后就算看到了出错信息，需要找很久也不一定能够找到问题的根源。除此之外，K210作为硬件设备，不可避免地会出现各种预期外的情况，如内存部分损坏、sd卡接触不良等一系列问题均可能发生，如笔者的SD卡在调试过程中就发生了损坏，又如我们在评测机上遇到的一系列问题（如本地能跑过但是评测机超时、以及后期出现评测机板卡故障的情况）。总体上来说，板上现象与预期不符是一个较为严重的问题，主要在于难以找到出错原因；后期使用qemu，能够使用GDB，一定程度上减小了debug的难度。

需要自己查找资料，翻阅库函数与样例进行特征性的适配

由于在决赛第一阶段中官方没有给出初赛那样详细的信息，所以对用例的适配过程，包括对系统调用的编写、模块的设计与用例错误信息的分析等环节，都非常考验我们的信息搜集和代码阅读分析能力。有时如果对系统调用的理解出现了一定偏差，或者忽略了flag中要求的某个功能，或是忽视了某些运行标准与宏/结构体定义，就会造成很大的问题，如果没有搜集信息与综合分析理解的能力，就容易连报错信息都看不懂，更不用说找到问题所在了。在系统调用搜集与功能模块设计上，除了从测试代码中提取信息之外，更需要上网查找与搜集大量有关系统调用的设计标准与功能特性，才能实现预期的功能；在代码阅读与错误分析上，除了参考libc-test测试的源码之外，我们还需要同时参考musl库源码与编译链源码，综合阅读与分析，才能找到我们需要的函数定义与报错原因。在这个过程中，我们学到了不少有关系统调用和系统设计的知识，也极大地提升了工程能力。

系统规模较大与协同开发引发的各种问题

本次决赛第一阶段中，由于引入了如signal、futex与动态链接等较为复杂与困难的功能实现，系统的复杂度相较初赛有了很大的提升，对于系统整体的理解要求也更高。由于我们队的三名队员使用git进行线上的协作开发，并使用qq线上进行互相通信，在协同开发的过程中也遇到了不少问题，如：代码风格不统一造成的阅读困难；对其他队员负责编写模块的代码与相应框架代码理解上的困难与错误；在编写某个模块时没有理解（忘记了）其他模块的调用接口规范，从而传入了错误的指针而造成整体功能实现错误；对内存分布与回收的理解不足，导致出现一系列莫名其妙的问题（如栈地址太低导致的覆盖、inst page fault、页表PTE被篡改、PCB未回收导致占用爆炸等）；对git的使用不够娴熟，导致错误地覆盖分支 / merge他人分支，或出现开发中代码版本难以统一的问题；自动merge本身功能运用不好，导致出现部分merge的现象，需要手动进行修改。以上都是分布式开发过程中容易遇到的问题，通过加强自身工程能力与培养与队友的默契，及时沟通，能够部分解决。

总结

本次的设计任务相较初赛难度提升了一个档次，而对我们的系统思维与设计能力也是一次考验。一开始认为很多功能的实现无从入手，而在充分收集资料后终于实现了相关的功能，感到非常有成就感；同样地，有的功能开始时我们认为很简单，但是实现上就会遇到各种各样的问题，和其他模块的协作也需要特别注意，甚至merge他人的代码都需要十分小心。在官方宣布改为采用qemu时，由于我们开始时没有实现virtio功能，一度非常沮丧与焦虑，好在后来采用ramdisk解决了这个问题，经过三人持续且艰苦的努力奋斗，也终于实现了所有功能点的通过，可以说努力最终有了回报。