OS-lab6 report

刘泓尊 2018011446 计84

Chapter6 实现的内容

1. 实现了文件抽象，提供stdin, stdout, pipe, mail_box 4种标准文件，对应修改了文件相关的syscall
2. PCB中加入文件描述符表，以及基于最先匹配的文件描述符分配。在fork, new的时候复制或加入文件描述符
3. 配合文件，实现了`close`, `read`, `write`, `pipe`, `mail_read`, `mail_write`系统调用
4. 用户库封装了上述syscall，方便应用程序调用

编程作业

1. **`MailPacket`数据结构，表示邮箱里的一个报文，最大256Byte。**

```
const PACKET_BUFFER_SIZE: usize = 256;
pub struct MailPacket {
    arr: [u8; PACKET_BUFFER_SIZE], // 报文缓冲区
    len: usize // 报文长度
}
```

为MailPacket实现了从`UserBuffer`读写缓冲区`arr`的方法，便于上层调用。

`MailPacket::from_buffer(user_buf: UserBuffer)-> MailPacket`: 从UserBuffer构造新的报文`MailPacket`, 截断报文到256字节。

`MailPacket::write_buf(user_buf: UserBuffer)-> usize`: 将`MailPacket`转换成`UserBuffer`, 方便输出。

上面两个方法都借助了`UserBuffer`, 因为它支持迭代，方便对内存的序列读写操作。

2. **`MailBox`数据结构。一个邮箱的抽象。**

```
const MAX_PACKET_NUM: usize = 16;
pub struct MailBox {
    pub size: usize, // 邮箱报文数量
    pub packets: VecDeque<MailPacket>, // 报文队列，最多16个
}
impl MailBox {
    pub fn read(&mut self, user_buf: UserBuffer) -> isize;
    pub fn write(&mut self, user_buf: UserBuffer) -> isize;
}
```

封装了read/write方法，支持从 `UserBuffer` 读写邮箱。每次读写都会创建一个新的报文。对于write, 判断邮箱是否已满，没有满就构造 `MailPacket` , 压入报文队列，返回报文长度；对于 `read` , 邮箱非空，并且user_buf的长度大于0，就弹出队列一个报文，利用 `MailPacket::write_buf` 向user_buf里写入数据，user_buf长度小于0就返回0.

3. **PCB对邮箱的管理**

   因为每个进程都有一个邮箱，所以不需要通过open打开邮箱加入文件描述符表，所以每个PCB直接有一个成员变量 `MailBox` 比较方便。new进程的时候创建空邮箱，fork进程的时候复制父进程所有的报文到子进程邮箱，保证继承。

4. **mail_read/mail_write syscall**

   ```
   pub fn sys_mail_read(buffer: *mut u8, len: usize) -> isize;
   pub fn sys_mail_write(pid: usize, buffer: *mut u8, len: usize) -> isize;
   ```

   首先要把 `buffer` 从虚拟地址转换到物理地址。判断地址的读写权限，权限不足就返回-1.

   对于mail_read，地址要可读。之后从buffer*构建 `UserBuffer` , 之后调用当前进程PCB的 `inner.mail_box.read(UserBuffer)` 即可。

   对于mail_write, 地址要可写。之后从buffer*构建 `UserBuffer` 。之后向pid的进程邮箱写入，如果是当前进程，调用当前进程PCB 的 `inner.mail_box.write(UserBuffer)` 即可。如果不是本进程，在 调度器的就绪队列 中查找pid的进程，找到之后调用该进程的 `inner.mail_box.write(UserBuffer)` 写入邮箱。找不到返回-1.

测试结果截图

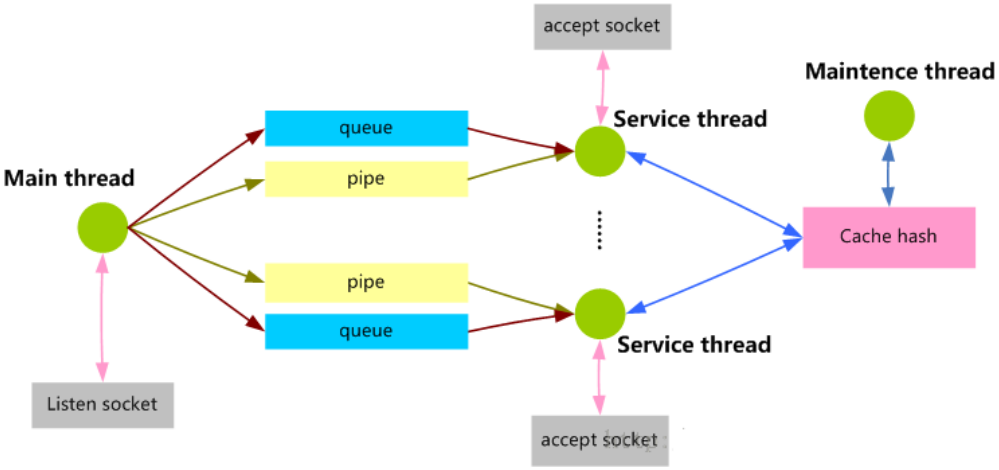
```
mail0 test OK!
Usertests: Test ch6_mail0 in Process 1 exited with code 0
Usertests: Running ch6_mail1
mail1 test OK!
Usertests: Test ch6_mail1 in Process 1 exited with code 0
Usertests: Running ch6_mail2
I am father
father sleep 1s
I am child
child read 1 mail fail
child sleep 2s
father wirte 16 mails succeed
father wirte 1 mail fail
father sleep 1.5s
child read 16 mails succeed
child read 1 mail fail
child sleep 1s
father wirte 1 mail succeed
child read 1 mail succeed
child exit
mail2 test OK!
Usertests: Test ch6_mail2 in Process 1 exited with code 0
Usertests: Running ch6_mail3
mail3 test OK!
Usertests: Test ch6_mail3 in Process 1 exited with code 0
ch6 Usertests passed!
```

问答作业

1. **举出使用 pipe 的一个实际应用的例子。**

   pipe是一种用于父子间进程半双工通讯的机制。可以用来主线程给工作线程分发任务。

   Memcached是一个著名的分布式缓存系统，和redis类似。它基于libevent网络事件库开发，分主线程和工作线程，主线程监听客户端请求，收到请求后创建clientfd, 选择一个工作线程，将clientfd放入工作线程的消息队列，之后通过和工作线程之间的pipe通知工作线程，工作线程发现pipe有可读内容之后，开始读取消息队列，开始处理自己的事务。



2. **假设我们的邮箱现在有了更加强大的功能，容量大幅增加而且记录邮件来源，可以实现“回信”。考虑一个多核场景，有 m 个核为消费者，n 个为生产者，消费者通过邮箱向生产者提出订单，生产者通过邮箱回信给出产品。**

   - **假设你的邮箱实现没有使用锁等机制进行保护，在多核情景下可能会发生哪些问题？单核一定不会发生问题吗？为什么？**

     多核下，两个消费者可能同时写入一个邮箱，虽然报文不同，但是可能同时进行入队操作，在队尾指针来不及移动的情况下，两个报文就写入到了同一个地方，会造成报文丢失或混淆。如果邮箱只能盛下一个报文的时候，同时来了2个报文，那么2个报文也都会认为自己可以加入邮箱，但实际只加入了一个报文。

     单核下涉及到OS的任务切换。对我们实现的mail_read/mail_write系统调用而言，trap到内核之后执行完毕就会返回用户态，中间不涉及任务切换，所以对邮箱的读写是原子的。只有在用户态读写缓冲区的时候可能会被切换。所以我们的实现下单核不会出现共享数据的冲突问题。

     单核下，如果对同一报文的读写是分多次系统调用进行的，就会发生读写没有完成就切换的问题，这时就需要加锁了。

   - **请结合你在课堂上学到的内容，描述读者写者问题的经典解决方案，必要时提供伪代码。**

     读者写者问题需要保证：1.读者、写者不同时存在；2.写者、写者不同时存在。对于本问题而言，因为读操作也涉及报文队列的出队，所以 读者、读者也不能同时存在。这个问题可以通过**加锁**解决。

     在我的实现中，会发生共享数据冲突的只有`MailBox::read`和`MailBox::write`，所以对这`MailBox`操作的时候加锁即可（**也就是`mail_box: Mutex<MailBox>`，比较简单，就不演示代码了**）。但实际上进程控制块`TaskControlBlockInner`已经加锁了，所以我的代码在多核下也能正常运行。

     对于一般的读者-写者问题，我简单构造一个共享缓冲区，有若干读者（消费者）和若干写者（生产者），一个简单的基于pthread的用户态C++程序如下：

```
1   const int NUM_THREADS = 10;
2   const int MAX_LENGTH = 100;
```

```
pthread_mutex_t mu;
char msg[MAX_LENGTH];
bool ok;

void* run(void* rank){
    long my_rank = (long)rank;
    while(1){
        pthread_mutex_lock(&mu); // 加锁，进入临界区
        if(my_rank % 2 == 1){ // 奇数消费者
            if(ok){
                printf("message received by %ld: %s\n", my_rank, msg);
                ok = 0;
                pthread_mutex_unlock(&mu); // 共享数据修改完成，离开临界区
                break;
            }
        }else{ // 偶数生产者
            if(!ok){
                sprintf(msg, "Hello from thread %ld", my_rank);
                ok = 1;
                pthread_mutex_unlock(&mu); // 共享数据修改完成，离开临界区
                break;
            }
        }
        pthread_mutex_unlock(&mu);
    }
    return nullptr;
}

int main() {
    ok = 0;
    pthread_mutex_init(&mu, nullptr); // 初始化锁
    pthread_t* threads = new pthread_t[NUM_THREADS];
    for(size_t i = 0; i < NUM_THREADS; i++){ // fork
        pthread_create(&threads[i], nullptr, ::run, (void*)i);
    }
    for(size_t i = 0; i < NUM_THREADS; i++){ // join
        pthread_join(threads[i], nullptr);
    }
    pthread_mutex_destroy(&mu);
    delete[] threads;
    return 0;
}
```

一般对于多进程的消息队列，是需要加互斥锁的，同时读（读也会弹出队列）、同时写、同时读写都会有共享数据竞争；

对于多进程的共享缓冲区，同时读是可以的，因为读操作不会修改缓冲区，所以可以使用读写锁(RWLock)，支持同时读，只有同时写或同时读写时互斥。读写锁在`pthread`里是`pthread_rwlock_t`.

- **由于读写是基于报文的，不是随机读写，你有什么点子来优化邮箱的实现吗?**

  如果外层的PCB没有加锁，就需要对mail_box加锁。因为mail_box是个队列，只有队头可读，队尾可写，所以可以细粒度地**只对队头（以及队头指针）、队尾（以及队尾指针）加不同的锁。**这样就可以在队列长度大于1时，**读者读队头的同时，写者可以写队尾。**当然，细粒度的管理会增加实现复杂度，但是在高并发场景下会带来潜在的性能提升。