Lab7

编程题

1 \* 分别编写基于UNIX System V IPC的管道、共享内存、信号量和消息队列的Linux应用程序，实现进程间的数据交换。

答：

首先，实现管道，我们先在mod.rs中申明管道系统调用：





接着fs.rs中定义管道创建函数：



之后，我们利用该系统调用来编写管道的程序:



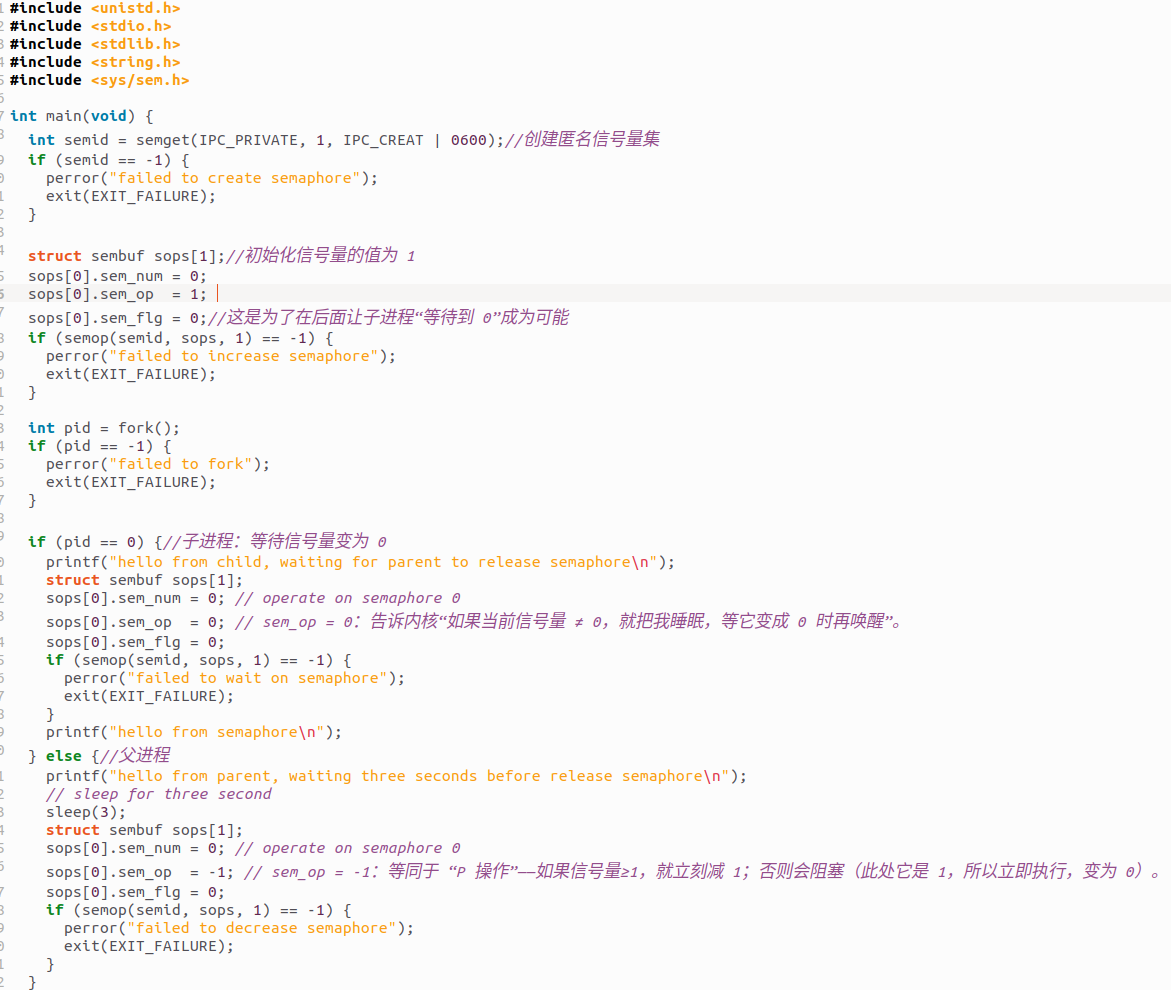
同理，我们可以编写剩下的linux应用程序：  
共享内存：



信号量：



消息队列：



2 \*\* 分别编写基于UNIX的signal机制的Linux应用程序，实现进程间异步通知。

答：可以编写以下程序：



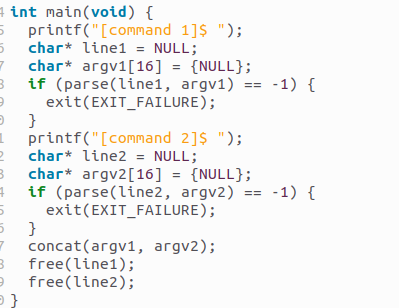
其中sigaction：推荐的 POSIX 接口，用于精细控制信号处理行为，比老的signal() 更安全可靠。而信号处理器则是一旦收到指定信号，内核中断用户代码执行，运行处理函数，然后恢复或退出。kill：不仅用于终止进程，也可发送任何信号给指定 PID，触发信号处理逻辑。典型模式：子进程挂起等待，父进程异步发送信号，子进程捕获后执行回调并退出，用于进程间简单的“事件通知”。

*3 \*\** 参考rCore Tutorial 中的shell应用程序，在Linux环境下，编写一个简单的shell应用程序，通过管道相关的系统调用，能够支持管道功能。

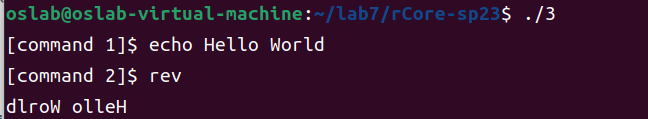
答：

要完成这个系统，我们可以选择从标准输入读取两条命令，然后把它们通过 UNIX 管道（pipe）连接起来，相当于在 shell 中执行。根据这个思路，我们可以编写程序：





我们运行该程序，就能看到command，我们对其进行输入，就相当于是在使用shell了，我们可以尝试使用在command1中输入指令echo Hello World ，然后在command 2 中输入rev,然后我们就可以看到进程倒序打印出了Hello World .



问答题：

1. 直接通信和间接通信的本质区别是什么？分别举一个例子。

本质区别是地址指定方式的不同，也就是消息是否经过内核，如共享内存就是直接通信，消息队列则是间接通信。

1. \*\* 试说明基于UNIX的signal机制，如果在本章内核中实现，请描述其大致设计思路和运行过程。

首先需要添加两个syscall，其一是注册signal handler，其二是发送signal。其次是添加对应的内核数据结构，对于每个进程需要维护两个表，其一是signal到handler地址的对应，其二是尚未处理的signal。当进程注册signal handler时，将所注册的处理函数的地址填入表一。当进程发送signal时，找到目标进程，将signal写入表二的队列之中。随后修改从内核态返回用户态的入口点的代码，检查是否有待处理的signal。若有，检查是否有对应的signal handler并跳转到该地址，如无则执行默认操作，如杀死进程。需要注意的是，此时需要记住原本的跳转地址，当进程从signal handler返回时将其还原。

1. \*\* 比较在Linux中的无名管道（普通管道）与有名管道（FIFO）的异同。

同：两者都是进程间信息单向传递的通路，可以在进程之间传递一个字节流。异：普通管道不存在文件系统上对应的文件，而是仅由读写两端两个fd表示，而FIFO则是由文件系统上的一个特殊文件表示，进程打开该文件后获得对应的fd。

1. \*\* 请描述Linux中的无名管道机制的特征和适用场景。

无名管道用于创建在进程间传递的一个字节流，适合用于流式传递大量数据，但是进程需要自己处理消息间的分割。

1. \*\* 请描述Linux中的消息队列机制的特征和适用场景。

消息队列用于在进程之间发送一个由type和data两部分组成的短消息，接收消息的进程可以通过type过滤自己感兴趣的消息，适用于大量进程之间传递短小而多种类的消息。

1. \*\* 请描述Linux中的共享内存机制的特征和适用场景。

共享内存用于创建一个多个进程可以同时访问的内存区域，故而消息的传递无需经过内核的处理，适用在需要较高性能的场景，但是进程之间需要额外的同步机制处理读写的顺序与时机。

1. \*\* 请描述Linux的bash shell中执行与一个程序时，用户敲击 Ctrl+C 后，会产生什么信号（signal），导致什么情况出现。

会产生SIGINT，如果该程序没有捕获该信号，它将会被杀死，若捕获了，通常会在处理完或是取消当前正在进行的操作后主动退出。

1. \*\* 请描述Linux的bash shell中执行与一个程序时，用户敲击 Ctrl+Zombie 后，会产生什么信号（signal），导致什么情况出现。

会产生SIGTSTP，该进程将会暂停运行，将控制权重新转回shell。

1. \*\* 请描述Linux的bash shell中执行 kill -9 2022 这个命令的含义是什么？导致什么情况出现。

向pid为2022的进程发送SIGKILL，该信号无法被捕获，该进程将会被强制杀死。

1. \*\* 请指出一种跨计算机的主机间的进程间通信机制。

一种常用的、跨越多台主机的进程间通信机制是基于 TCP 套接字（BSD Sockets）的网络通信。

实验练习：

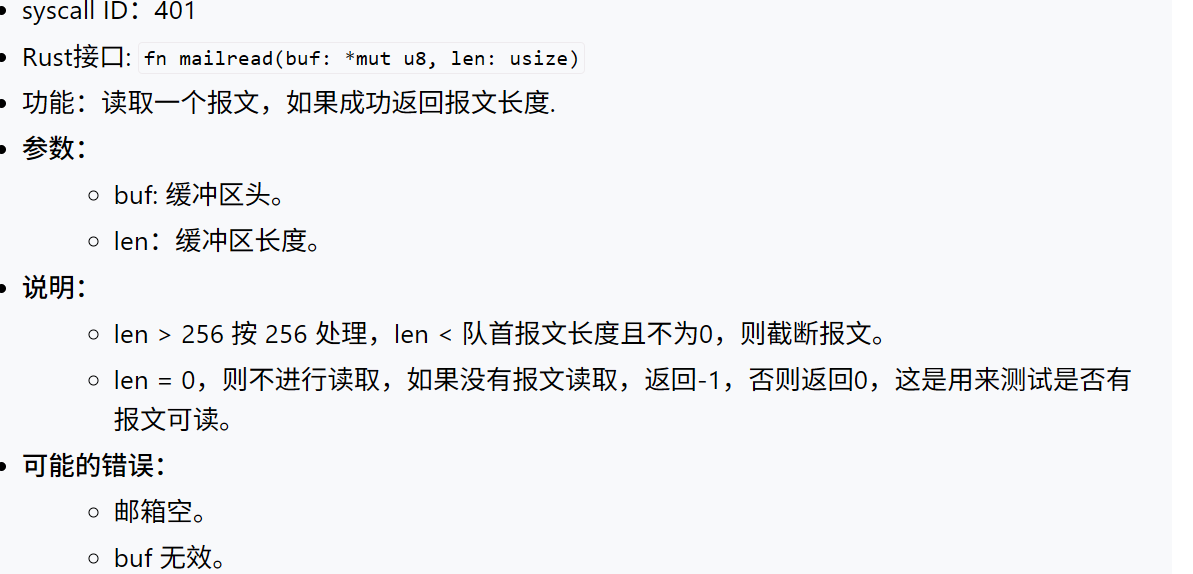
编程作业

进程通信：邮箱

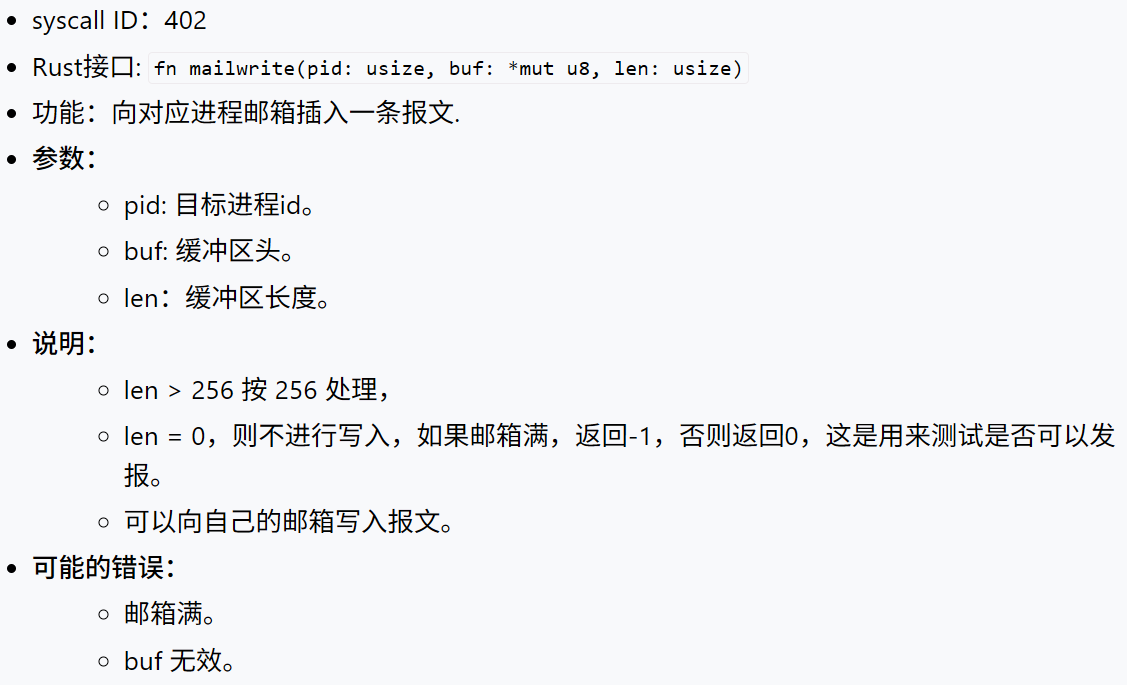
这一章我们实现了基于 pipe 的进程间通信，但是看测例就知道了，管道不太自由，我们来实现一套乍一看更靠谱的通信 syscall吧！本节要求实现邮箱机制，以及对应的 syscall。

邮箱说明：每个进程拥有唯一一个邮箱，基于“数据报”收发字节信息，利用环形buffer存储，读写顺序为 FIFO，不记录来源进程。每次读写单位必须为一个报文，如果用于接收的缓冲区长度不够，舍弃超出的部分（截断报文）。为了简单，邮箱中最多拥有16条报文，每条报文最大长度256字节。当邮箱满时，发送邮件（也就是写邮箱）会失败。不考虑读写邮箱的权限，也就是所有进程都能够随意给其他进程的邮箱发报。

依据题目中的要求，我们来进行**mailread**和**mailwrite**系统调用的编写。



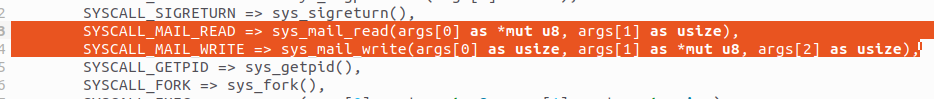
**Mailread介绍**



**Mailwrite介绍**

我们首先在syscall/mod.rs中加入对两个系统调用号的声明：





接着我们根据要求去创建具体的系统调用内容：



**Mailwrite**

该系统调用以进程ID为目标，将用户缓冲区里的数据写入目标进程的 环形邮箱。关键流程包括：参数校验 → 查找进程 → 加锁互斥访问 → 槽位是否已满检查 → 用户空间数据安全翻译 → 拷贝数据 → 更新指针和状态 → 返回结果。

通过这种机制，不同进程间可基于 消息邮箱（mailbox）的方式进行松耦合的、可靠的点对点通信。



**Mailread**

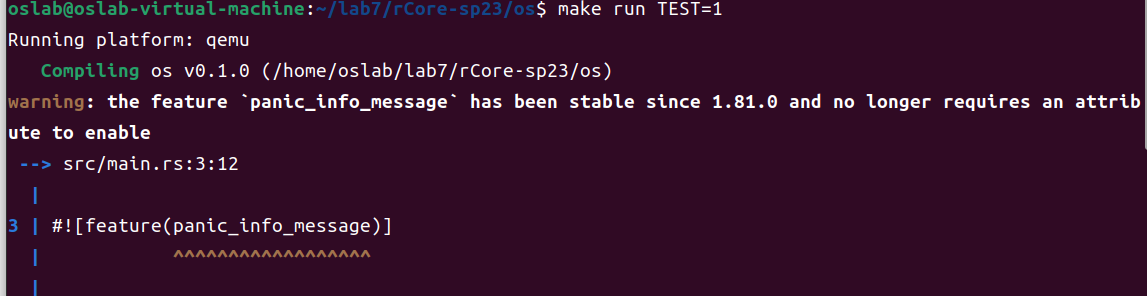
这里我们首先进行了参数校验，判断 len 是否为 0，若是直接返回 0，无需进一步操作。接着，获取当前进程上下文，调用 current\_task() 拿到当前任务对象，并通过 inner\_exclusive\_access() 锁住它，以保证对任务数据的安全访问。取出用户页表 Token (get\_user\_token())，为后续的用户缓冲区映射做准备。

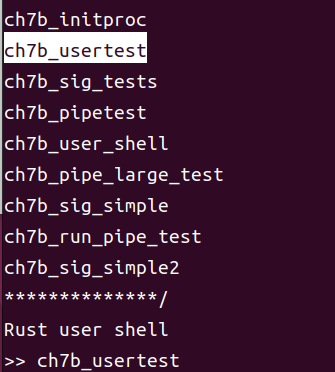
再定位并锁定邮箱缓冲区，通过 inner.mailbox.buffer.exclusive\_access() 获得对当前进程邮箱缓冲区的独占访问权限。

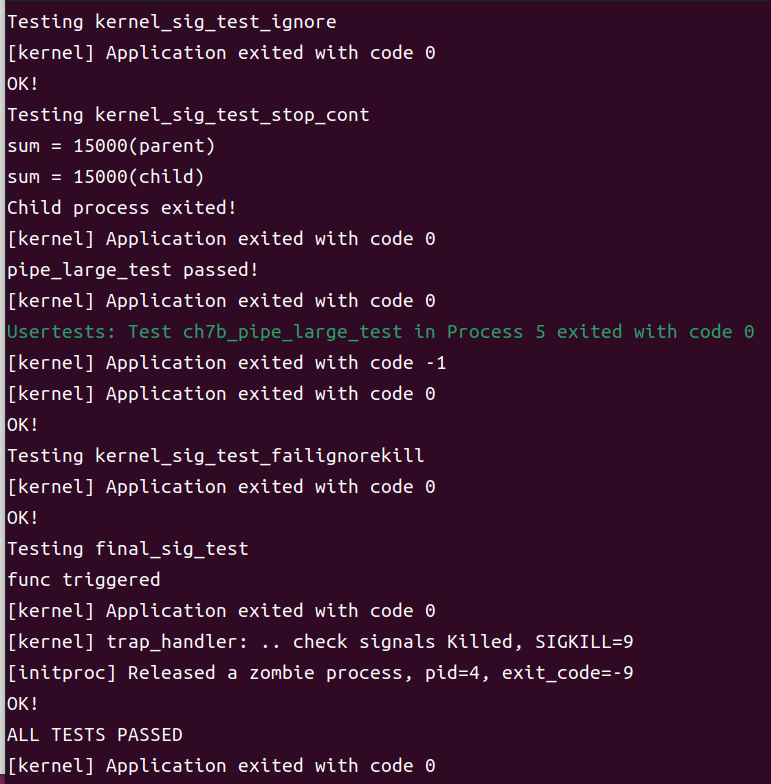
之后邮箱状态检查，调用 mailbox\_inner.is\_empty() 判断邮箱中是否有消息，若空则返回 -1，提示“无可读消息”。然后，计算读取长度，读取指针 head 指向待读的消息槽位。取请求长度 len 与该槽位实际消息长度 arr[head].len 的最小值，避免读取越界。之后用户空间映射与数据拷贝使用 translated\_byte\_buffer(token, buf, mlen) 将用户缓冲区映射到内核可写的指针集合。通过循环，将内核邮箱缓存区中对应的每个字节，逐一拷贝到用户缓冲区。再更新环形队列指针与状态，将 head 向后移动一位（并取模环绕）。如果移动后 head == tail，说明已读完所有消息，将状态置为 Empty；否则保持 Normal。

最后，读取结果，成功读取后返回 0；任何提前失败情况（如空邮箱或其他错误）均已通过 return -1 或 return 0 处理。

做完之后，我们使用“make run TEST=1”来运行测试文件“ch7b\_usertest”







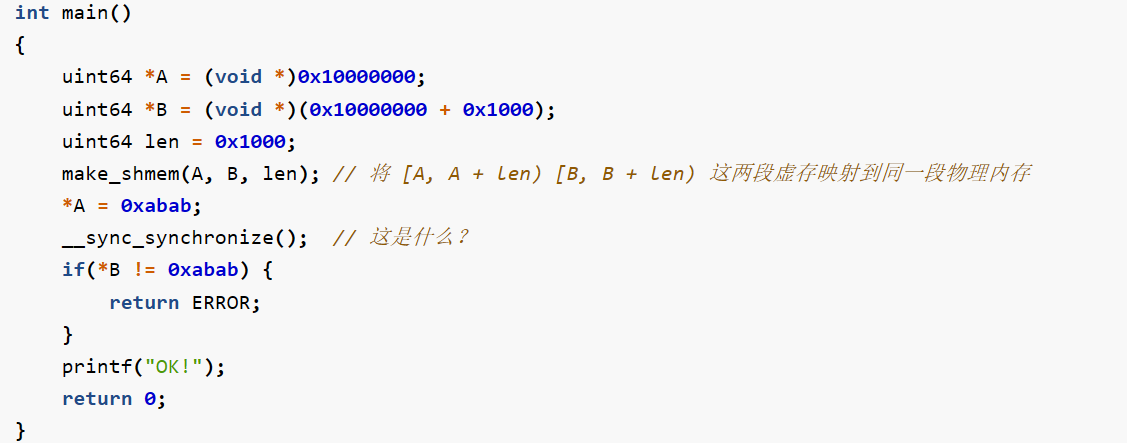
可以看到输出了“ALL TESTS PASSED”表明我们成功通过了所有测试。

问答作业：

1. 举出使用 pipe 的一个实际应用的例子。

答：一个非常典型且你几乎每天都会用到的“真实应用”就是 tar+gzip（或 bzip2、xz）组合——在打包归档时，tar 会把数据流 “管道” 给压缩程序，而不在磁盘上先写出一个完整的未压缩归档文件。

1. 共享内存的测例中有如下片段(伪代码)：



请查阅相关资料，回答 \_\_sync\_synchronize 这行代码的作用，如果去掉它可能会导致什么错误？为什么？

\_\_sync\_synchronize() 在 GCC 的 legacy 原子内置函数中，作用就是产生一个 全内存屏障（full memory barrier），它同时对编译器和硬件生效：

编译器屏障：禁止编译器把屏障前后的任何内存读写指令重排序到屏障之外；

CPU（硬件）屏障：在多数架构上，会生成诸如 x86 的 mfence、ARM 的 dmb ish 等指令，确保屏障之前的所有写操作在屏障之后对所有核可见，且屏障之后的读操作不会在屏障之前就执行.

去掉它可能导致的错误有：

一是写入不可见，也就是在弱内存模型（如 ARM、POWER 等）或高优化级别下，\*A = 0xabab; 的写操作可能被暂存在处理器的 Store Buffer 中，还没真正写回物理内存。

二是乱序读取，即使是在同一核上，CPU 也可能为了性能把后面的读操作(\*B)和前面的写操作(\*A)调换顺序执行。

这样就会出现：写往 A 的数据并未“刷”到共享物理页，或读 B 时恰好过早执行，于是 \*B 读到的仍是旧值（比如 0），程序就会误判走到 ERROR 分支。就类似于锁的用法。