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：

```
pub const SYSCALL_PIPE: usize = 59;
```

```
        SYSCALL_CLOSE => sys_close(args[0]),
        SYSCALL_PIPE => sys_pipe(args[0] as *mut usize),
        SYSCALL_DUP => sys_dup(args[0]),
```

接着 fs.rs 中定义管道创建函数：

```
pub fn sys_pipe(pipe: *mut usize) -> isize {//函数签名
    let task = current_task().unwrap();
    let token = current_user_token();//获得当前执行的任务（进程）
    let mut inner = task.inner_exclusive_access();
    let (pipe_read, pipe_write) = make_pipe();//创建管道对象
    let read_fd = inner.alloc_fd();//为读端分配一个文件描述符
    inner.fd_table[read_fd] = Some(pipe_read);
    let write_fd = inner.alloc_fd();//为写端分配另一个文件描述符
    inner.fd_table[write_fd] = Some(pipe_write);
    *translated_refmut(token, pipe) = read_fd;
    *translated_refmut(token, unsafe { pipe.add(1) }) = write_fd;//把 FD 写回用户空间
    0
}
```

之后，我们利用该系统调用来编写管道的程序：

```c
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(void) {
  int pipefd[2];
  if (pipe(pipefd) == -1) {//调用pipe系统调用
    perror("failed to create pipe");
    exit(EXIT_FAILURE);
  }//创建管道

  int pid = fork();
  if (pid == -1) {
    perror("failed to fork");
    exit(EXIT_FAILURE);
  }

  if (pid == 0) {//子进程逻辑（读端）
    close(pipefd[1]);
    char buf;
    while (read(pipefd[0], &buf, 1) > 0) {
      printf("%s", &buf);
    }
    close(pipefd[0]);
  } else {//父进程逻辑（写端）
    close(pipefd[0]); // close the read end
    // parent writes
    char* msg = "hello from pipe\n";
    write(pipefd[1], msg, strlen(msg));
    close(pipefd[1]);
  }

  return EXIT_SUCCESS;
}
```

同理，我们可以编写剩下的 linux 应用程序：

共享内存：

```c
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/shm.h>

int main(void) {
  int shmid = shmget(IPC_PRIVATE, sysconf(_SC_PAGESIZE), IPC_CREAT | 0600);//创建共享内存段
  if (shmid == -1) {
    perror("failed to create shared memory");
    exit(EXIT_FAILURE);
  }

  int pid = fork();
  if (pid == -1) {
    perror("failed to fork");
    exit(EXIT_FAILURE);
  }

  if (pid == 0) {//子进程逻辑
    char* shm = shmat(shmid, NULL, 0);//把那一页映射到子进程的地址空间，返回基址 shm
    while (!shm[0]) {//不断检查 shm[0]（第 1 个字节）是否变为非 0。这个字节在父进程写完消息后被置为 1
    }
    printf("%s", shm + 1);//读取并打印
  } else {//映射共享内存 同子进程。
    char* shm = shmat(shmid, NULL, 0);
    strcpy(shm + 1, "hello from shared memory\n");//写入消息
    shm[0] = 1;
  }

  return EXIT_SUCCESS;
}
```

信号量：

```c
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/shm.h>

int main(void) {
  int shmid = shmget(IPC_PRIVATE, sysconf(_SC_PAGESIZE), IPC_CREAT | 0600);//创建共享内存段
  if (shmid == -1) {
    perror("failed to create shared memory");
    exit(EXIT_FAILURE);
  }

  int pid = fork();
  if (pid == -1) {
    perror("failed to fork");
    exit(EXIT_FAILURE);
  }

  if (pid == 0) {//子进程逻辑
    char* shm = shmat(shmid, NULL, 0);//把那一页映射到子进程的地址空间，返回基址 shm
    while (!shm[0]) {//不断检查 shm[0]（第 1 个字节）是否变为非 0。这个字节在父进程写完消息后被置为 1
    }
    printf("%s", shm + 1);//读取并打印
  } else {//映射共享内存 同子进程。
    char* shm = shmat(shmid, NULL, 0);
    strcpy(shm + 1, "hello from shared memory\n");//写入消息
    shm[0] = 1;
  }

  return EXIT_SUCCESS;
}
```

消息队列：

```c
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/sem.h>

int main(void) {
  int semid = semget(IPC_PRIVATE, 1, IPC_CREAT | 0600);//创建匿名信号量集
  if (semid == -1) {
    perror("failed to create semaphore");
    exit(EXIT_FAILURE);
  }

  struct sembuf sops[1];//初始化信号量的值为 1
  sops[0].sem_num = 0;
  sops[0].sem_op  = 1;
  sops[0].sem_flg = 0;//这是为了在后面让子进程“等待到 0”成为可能
  if (semop(semid, sops, 1) == -1) {
    perror("failed to increase semaphore");
    exit(EXIT_FAILURE);
  }

  int pid = fork();
  if (pid == -1) {
    perror("failed to fork");
    exit(EXIT_FAILURE);
  }

  if (pid == 0) {//子进程：等待信号量变为 0
    printf("hello from child, waiting for parent to release semaphore\n");
    struct sembuf sops[1];
    sops[0].sem_num = 0; // operate on semaphore 0
    sops[0].sem_op  = 0; // sem_op = 0：告诉内核“如果当前信号量 ≠ 0，就把我睡眠，等它变成 0 时再唤醒”。
    sops[0].sem_flg = 0;
    if (semop(semid, sops, 1) == -1) {
      perror("failed to wait on semaphore");
      exit(EXIT_FAILURE);
    }
    printf("hello from semaphore\n");
  } else {//父进程
    printf("hello from parent, waiting three seconds before release semaphore\n");
    // sleep for three second
    sleep(3);
    struct sembuf sops[1];
    sops[0].sem_num = 0; // operate on semaphore 0
    sops[0].sem_op  = -1; // sem_op = -1：等同于 “P 操作”—如果信号量≥1，就立刻减 1；否则会阻塞（此处它是 1，所以立即执行，变为 0）。
    sops[0].sem_flg = 0;
    if (semop(semid, sops, 1) == -1) {
      perror("failed to decrease semaphore");
      exit(EXIT_FAILURE);
    }
  }
```

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：

```c
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>

static void sighandler(int sig) {//信号处理函数 sighandler
  printf("received signal %d, exiting\n", sig);//当进程收到信号时，内核会暂停当前执行流，跳到这个函数执行。
  exit(EXIT_SUCCESS);
}

int main(void) {
  struct sigaction sa;// 安装信号处理器
  sa.sa_handler = sighandler;//// 指定处理函数
  sa.sa_flags = 0;//默认行为，不加额外标志
  sigemptyset(&sa.sa_mask);//信号屏蔽集，处理期间不屏蔽任何额外信号
  if (sigaction(SIGUSR1, &sa, NULL) != 0) {
    perror("failed to register signal handler");
    exit(EXIT_FAILURE);
  }

  int pid = fork();
  if (pid == -1) {
    perror("failed to fork");
    exit(EXIT_FAILURE);
  }

  if (pid == 0) {
    while (1) {
      // 子进程：无限循环等待信号
    }
  } else {
    //父进程：发送信号
    kill(pid, SIGUSR1);
  }

  return EXIT_SUCCESS;
}
```

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：

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/wait.h>
#include <unistd.h>

int parse(char* line, char** argv) {
  size_t len;
  if (getline(&line, &len, stdin) == -1)
    return -1;
  line[strlen(line) - 1] = '\0';
  int i = 0;
  char* token = strtok(line, " ");
  while (token != NULL) {
    argv[i] = token;
    token = strtok(NULL, " ");
    i++;
  }
  return 0;
}

int concat(char** argv1, char** argv2) {
    int pipefd[2];
    if (pipe(pipefd) == -1)
      return -1;

    int pid1 = fork();
    if (pid1 == -1)
      return -1;
    if (pid1 == 0) {
      dup2(pipefd[1], STDOUT_FILENO);
      close(pipefd[0]);
      close(pipefd[1]);
      execvp(argv1[0], argv1);
    }

    int pid2 = fork();
    if (pid2 == -1)
      return -1;
    if (pid2 == 0) {
      dup2(pipefd[0], STDIN_FILENO);
      close(pipefd[0]);
      close(pipefd[1]);
      execvp(argv2[0], argv2);
    }

    close(pipefd[0]);
    close(pipefd[1]);
    wait(&pid1);
    wait(&pid2);
    return 0;
}
```

```c
int main(void) {
  printf("[command 1]$ ");
  char* line1 = NULL;
  char* argv1[16] = {NULL};
  if (parse(line1, argv1) == -1) {
    exit(EXIT_FAILURE);
  }
  printf("[command 2]$ ");
  char* line2 = NULL;
  char* argv2[16] = {NULL};
  if (parse(line2, argv2) == -1) {
    exit(EXIT_FAILURE);
  }
  concat(argv1, argv2);
  free(line1);
  free(line2);
}
```

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 .

```
oslab@oslab-virtual-machine:~/lab7/rCore-sp23$ ./3
[command 1]$ echo Hello World
[command 2]$ rev
dlroW olleH
```

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。

2. ** 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。

3. ** 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。

4. ** 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。

5. ** 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。

6. ** 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。

7. ** 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。

8. ** 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。

9. ** 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。

10. ** 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。

- syscall ID：401
- Rust接口: `fn mailread(buf: *mut u8, len: usize)`
- 功能：读取一个报文，如果成功返回报文长度.
- **参数：**
  - buf: 缓冲区头。
  - len：缓冲区长度。
- **说明：**
  - len > 256 按 256 处理，len < 队首报文长度且不为0，则截断报文。
  - len = 0，则不进行读取，如果没有报文读取，返回-1，否则返回0，这是用来测试是否有报文可读。
- **可能的错误：**
  - 邮箱空。
  - buf 无效。

Mailread 介绍

- syscall ID：402
- Rust接口: `fn mailwrite(pid: usize, buf: *mut u8, len: usize)`
- 功能：向对应进程邮箱插入一条报文.
- **参数：**
    - pid: 目标进程id。
    - buf: 缓冲区头。
    - len：缓冲区长度。
- **说明：**
    - len > 256 按 256 处理，
    - len = 0，则不进行写入，如果邮箱满，返回-1，否则返回0，这是用来测试是否可以发报。
    - 可以向自己的邮箱写入报文。
- **可能的错误：**
    - 邮箱满。
    - buf 无效。

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：

```rust
pub const SYSCALL_MAIL_READ: usize = 401;
pub const SYSCALL_MAIL_WRITE: usize = 402;
pub const SYSCALL_DUP: usize = 24;
```

```rust
        SYSCALL_SIGRETURN => sys_sigreturn(),
        SYSCALL_MAIL_READ => sys_mail_read(args[0] as *mut u8, args[1] as usize),
        SYSCALL_MAIL_WRITE => sys_mail_write(args[0] as usize, args[1] as *mut u8, args[2] as usize),
        SYSCALL_GETPID => sys_getpid(),
        SYSCALL_FORK => sys_fork(),
```

接着我们根据要求去创建具体的系统调用内容：

```rust
pub fn sys_mail_write(pid: usize, buf: *mut u8, len: usize) -> isize {
    if core::ptr::null() == buf {
        return -1;
    }
    if len == 0 {
        return 0;
    }
    if let Some(target_task) = pid2task(pid) {
        let target_task_ref = target_task.inner_exclusive_access();
        let token = target_task_ref.get_user_token();
        let mut mailbox_inner = target_task_ref.mailbox.buffer.exclusive_access();
        if mailbox_inner.is_full() {
            return -1;
        }
        let mailbox_tail = mailbox_inner.tail;
        mailbox_inner.status = MailBoxStatus::Normal;
        // the truncated mail length
        let mlen = len.min(MAX_MAIL_LENGTH);
        // prepare source data
        let src_vec = translated_byte_buffer(token, buf, mlen);
        // copy from source to dst
        for (idx, src) in src_vec.into_iter().enumerate() {
            unsafe {
                mailbox_inner.arr[mailbox_tail].data[idx..=idx].copy_from_slice(
                    core::slice::from_raw_parts(
                            src.as_ptr(),
                            core::mem::size_of::<u8>()
                            )
                    );
            }
        }
        // store the mail length
        mailbox_inner.arr[mailbox_tail].len = mlen;

        mailbox_inner.tail = (mailbox_tail + 1) % MAX_MESSAGE_NUM;
        if mailbox_inner.tail == mailbox_inner.head {
            mailbox_inner.status = MailBoxStatus::Full;
        }
        return 0;
    }
    -1
}
```

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。

```rust
pub fn sys_mail_read(buf: *mut u8, len: usize) -> isize {
    if len == 0 {
        return 0;
    }//获取当前任务和锁
    let task = current_task().unwrap();//获取当前执行的内核任务（进程/线程）对象。
    let inner = task.inner_exclusive_access();//对该任务内部数据加锁，获得排他访问。
    let token = inner.get_user_token();//取出当前进程的用户地址空间标识，用于地址转换。
    let mut mailbox_inner = inner.mailbox.buffer.exclusive_access();//对该进程的邮箱缓冲区加锁，获得可变引用。
    if mailbox_inner.is_empty() {//检查邮箱是否为空
        return -1;
    }
    let mailbox_head = mailbox_inner.head;//计算实际读取长度
    let mlen = len.min(mailbox_inner.arr[mailbox_head].len);
    let dst_vec = translated_byte_buffer(token, buf, mlen);// 准备用户缓冲区映射
    let src_ptr = mailbox_inner.arr[mailbox_head].data.as_ptr();
    for (idx, dst) in dst_vec.into_iter().enumerate() {//数据拷贝循环
        unsafe {
            dst.copy_from_slice(
                core::slice::from_raw_parts(
                    src_ptr.wrapping_add(idx) as *const u8,
                    core::mem::size_of::<u8>()
                    )
            );
        }
    }//更新邮箱状态及指针
    mailbox_inner.status = MailBoxStatus::Normal;//重置状态：先将状态标为 Normal。
    mailbox_inner.head = (mailbox_head + 1) % MAX_MAIL_LENGTH;//读指针后移：head 增一并环绕。
    if mailbox_inner.head == mailbox_inner.tail {
        mailbox_inner.status = MailBoxStatus::Empty;//检查空：如果移动后 head 追上了 tail，说明所有消息都已读完，标记 Empty。
    }
    0
}
```

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。

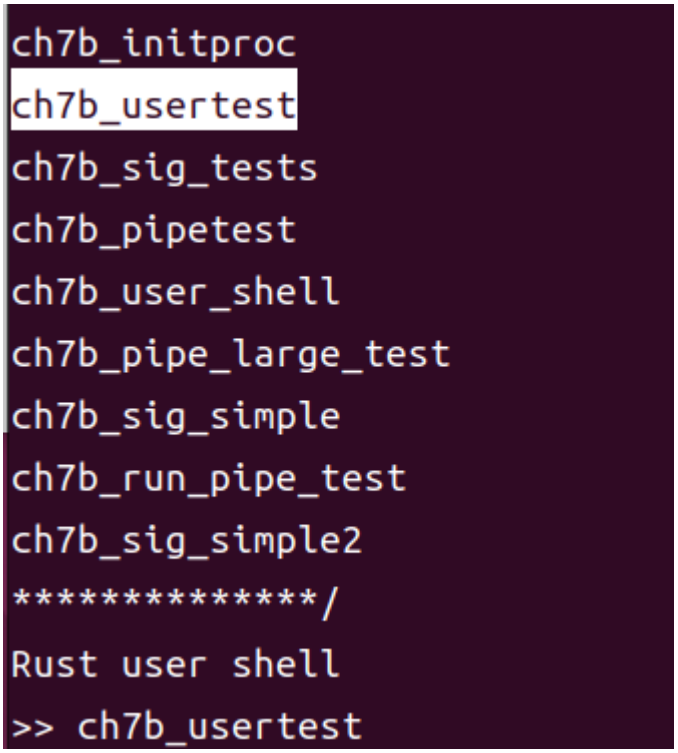
再定位并锁定邮箱缓冲区，通过 inner.mailbox.buffer.exclusive_access() 获得对当前进程邮箱缓冲区的独占访问权限。

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”

```
oslab@oslab-virtual-machine:~/lab7/rCore-sp23/os$ make run TEST=1
Running platform: qemu
   Compiling os v0.1.0 (/home/oslab/lab7/rCore-sp23/os)
warning: the feature `panic_info_message` has been stable since 1.81.0 and no longer requires an attrib
ute to enable
 --> src/main.rs:3:12
  |
3 | #![feature(panic_info_message)]
  |            ^^^^^^^^^^^^^^^^^^
  |
```


ch7b_initproc
ch7b_usertest
ch7b_sig_tests
ch7b_pipetest
ch7b_user_shell
ch7b_pipe_large_test
ch7b_sig_simple
ch7b_run_pipe_test
ch7b_sig_simple2
**********/
Rust user shell
>> ch7b_usertest

```
Testing kernel_sig_test_ignore
[kernel] Application exited with code 0
OK!
Testing kernel_sig_test_stop_cont
sum = 15000(parent)
sum = 15000(child)
Child process exited!
[kernel] Application exited with code 0
pipe_large_test passed!
[kernel] Application exited with code 0
Usertests: Test ch7b_pipe_large_test in Process 5 exited with code 0
[kernel] Application exited with code -1
[kernel] Application exited with code 0
OK!
Testing kernel_sig_test_failignorekill
[kernel] Application exited with code 0
OK!
Testing final_sig_test
func triggered
[kernel] Application exited with code 0
[kernel] trap_handler: .. check signals Killed, SIGKILL=9
[initproc] Released a zombie process, pid=4, exit_code=-9
OK!
ALL TESTS PASSED
[kernel] Application exited with code 0
```

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。

2. 共享内存的测例中有如下片段(伪代码)：

```c
int main()
{
    uint64 *A = (void *)0x10000000;
    uint64 *B = (void *)(0x10000000 + 0x1000);
    uint64 len = 0x1000;
    make_shmem(A, B, len); // 将 [A, A + len) [B, B + len) 这两段虚存映射到同一段物理内存
    *A = 0xabab;
    __sync_synchronize();  // 这是什么？
    if(*B != 0xabab) {
        return ERROR;
    }
    printf("OK!");
    return 0;
}
```

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。