7. 进程间通信

授课教师：游伟 副教授

授课时间：周二14:00 – 15:30（立德楼807）

课程主页：https://www.youwei.site/course/kernel

目录

7.1 基本概念

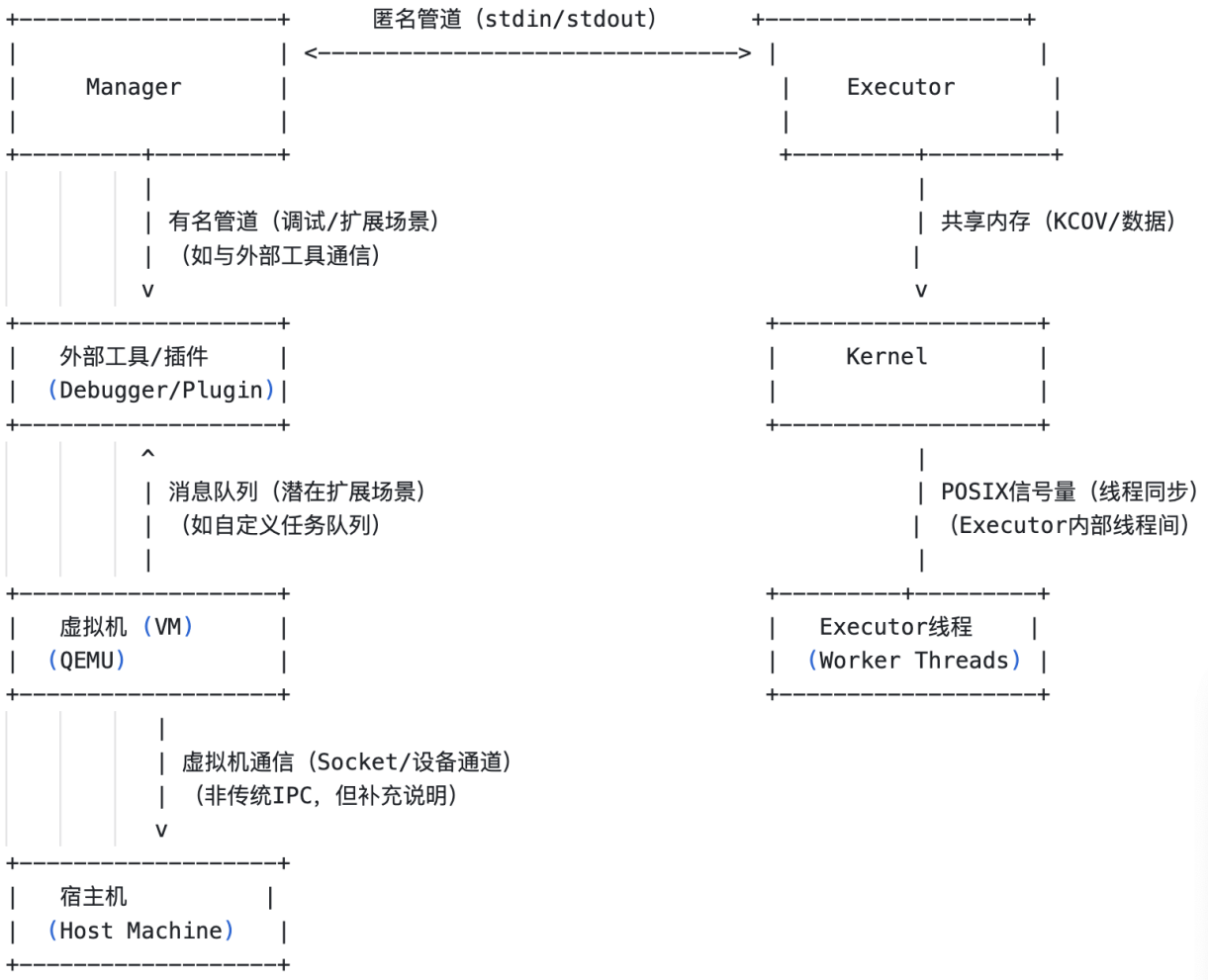
- 应用需求：进程用户空间是相互独立的，一般而言不能相互访问。然而很多情况下进程间需要互相通信，来完成系统的某项功能。
- 应用场景：
  - 数据传输：一个进程要将数据发送给另一个进程，发送的数据量在一个字节到几兆字节之间。
  - 共享数据：多个进程需要操作共享数据，一个进程对共享数据的修改，别的进程应立刻看到。
  - 通知事件：一个进程要向另一个或一组进程发送消息，通知它（它们）发生了某种事件（如进程终止时要通知父进程）。
  - 资源共享：多个进程间共享同样资源。为了做到这一点，需要内核提供锁和同步机制。
  - 进程控制：有些进程希望完全控制另一个进程的执行（如调试进程），此时控制进程希望能够拦截另一个进程的所有陷入和异常，并能够及时知道它的状态改变。

7.1 基本概念

- 进程通信的方式

7.1 基本概念

- 实践框架：一个内核模糊测试框架。



7.1 基本概念

- 实践框架：一个内核模糊测试框架。
  - **匿名管道**：Manager <--> Executor，核心通信通道，传输测试用例和结果（stdin/stdout）。
  - **共享内存**：Executor <--> Kernel，通过 KCOV 共享代码覆盖率数据（mmap 映射 /dev/kcov）。
  - **POSIX 信号量**：Executor 内部线程间，潜在用于多线程同步（如全局状态锁，但 Syzkaller 更依赖原子操作）。
  - **有名管道**：Manager <--> 外部工具/插件，调试或扩展场景（例如通过 mkfifo 创建管道与日志分析工具交互）。
  - **消息队列**：Manager <--> 外部工具/插件，潜在用于任务分发（如将测试任务分发给多个插件进程）。

7.2 管道

- 管道是Linux内核实现的进程间通信机制，其本质特征：
  - 半双工通信：数据单向流动，需明确读写端
  - 内存缓冲区：基于pipe_inode_info结构管理环形缓冲区
  - 生命周期：随进程结束自动销毁（匿名管道）
  - 内核对象：通过虚拟文件系统实现，操作接口为pipefifo_fops
- 管道创建机制pipe系统调用

```
// 传统创建方式
int pipe(int fd[2]);

// 扩展创建方式（支持flags）
int pipe2(int fd[2], int flags);
```

  - flags参数：
    - O_NONBLOCK：非阻塞模式
    - O_CLOEXEC：执行时关闭

7.2 管道

```
// File: chapter8/Pipe/anonymous_pipe.c

1.  #include <unistd.h>
2.  #include <stdio.h>
3.  #include <stdlib.h>
4.  #include <sys/wait.h>
5.  #define LEN 32

6.  int main(int argc,char *argv[])
7.  {
8.    int fd1[2], fd2[2];
9.    int status, pid;
10.   char str[LEN];

11.   pipe(fd1);
12.   pipe(fd2);
13.   pid=fork();

14.   if (pid > 0)       //父进程：往fd1写数据，从fd2读数据
15.   {
16.     close(fd1[0]);  //关闭fd1的读端
17.     close(fd2[1]);  //关闭fd2的写端
18.     write(fd1[1], "hello child!\n", LEN);
19.     read(fd2[0], str, LEN);
20.     printf("%s", str);
21.     wait(&status);
22.     printf("pid: %d\n", WEXITSTATUS(status));
23.     exit(0);
24.   }
25.   else if (pid==0)  //子进程：从fd1读数据，往fd2写数据
26.   {
27.     close(fd1[1]);  //关闭fd1的写端
28.     close(fd2[0]);  //关闭fd2的读端
29.     read(fd1[0], str, LEN);
30.     printf("%s", str);
31.     write(fd2[1], "hello father\n", LEN);
32.     exit(0);
33.   }
34. }
```

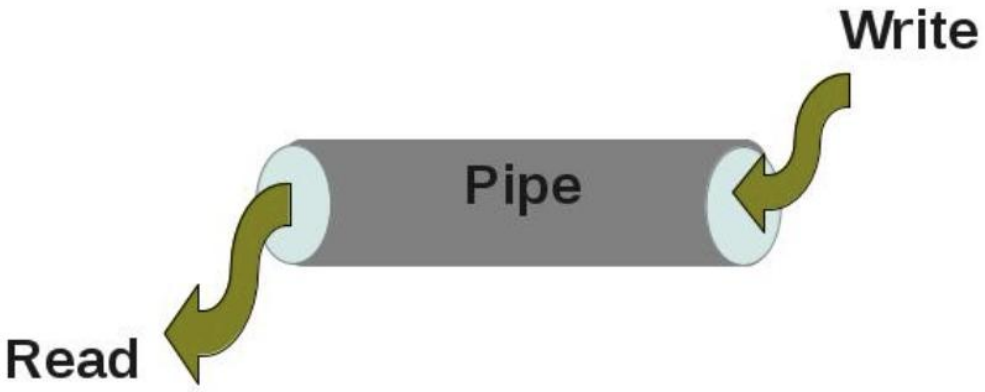
7.2 管道

- do_pipe2：该系统调用实现函数通过__do_pipe_flags创建一对匿名文件描述符，处理flags参数（如O_NONBLOCK），通过copy_to_user将fd返回用户空间，并通过fd_install将文件对象绑定到进程的文件描述符表，完成管道的双端创建。

```
/*
 * sys_pipe() is the normal C calling standard for creating
 * a pipe. It's not the way Unix traditionally does this, though.
 */
static int do_pipe2(int __user *fildes, int flags)
{
        struct file *files[2];
        int fd[2];
        int error;

        error = __do_pipe_flags(fd, files, flags);
        if (!error) {
                if (unlikely(copy_to_user(fildes, fd, sizeof(fd)))) {
                        fput(files[0]);
                        fput(files[1]);
                        put_unused_fd(fd[0]);
                        put_unused_fd(fd[1]);
                        error = -EFAULT;
                } else {
                        fd_install(fd[0], files[0]);
                        fd_install(fd[1], files[1]);
                }
        }
        return error;
}
```

7.2 管道

- Linux管道通过两个核心结构体协同工作：
  - struct pipe_inode_info：管道全局控制结构，管理整个生命周期
  - struct pipe_buffer：数据缓冲区单元，存储实际通信内容
- 二者关系如同机场塔台（控制中心）与停机坪（数据区）的协作模式。塔台（pipe_inode_info）掌握航班动态、调度资源，停机坪（pipe_buffer）提供物理空间存放飞机（数据）。



7.2 管道

```
struct pipe_inode_info {
        struct mutex mutex;
        wait_queue_head_t rd_wait, wr_wait;
        unsigned int head;
        unsigned int tail;
        unsigned int max_usage;
        unsigned int ring_size;
#ifdef CONFIG_WATCH_QUEUE
        bool note_loss;
#endif
        unsigned int nr_accounted;
        unsigned int readers;
        unsigned int writers;
        unsigned int files;
        unsigned int r_counter;
        unsigned int w_counter;
        bool poll_usage;
        struct page *tmp_page;
        struct fasync_struct *fasync_readers;
        struct fasync_struct *fasync_writers;
        struct pipe_buffer *bufs;
        struct user_struct *user;
#ifdef CONFIG_WATCH_QUEUE
        struct watch_queue *watch_queue;
#endif
};
```

7.2 管道

- 该结构体代表一个管道对象，封装了管道的状态、缓冲区数组以及读写双方的同步与管理机制。
  - mutex：管道的互斥锁，用于保护整个管道结构，确保在并发访问（读写）时的数据一致性。
  - rd_wait 与 wr_wait：两个等待队列，分别用于读者和写者。当管道为空时，读者可能需要等待数据；当管道满时，写者则需要等待空间释放。这两个队列用于实现这种阻塞等待机制。
  - head 与 tail：分别表示环形缓冲区中数据的生产位置（head）和消费位置（tail）。内核通过这两个索引在缓冲区数组（bufs）中定位数据的读写位置，实现先进先出的数据传递。
  - max_usage：表示环形缓冲区中允许被占用的最大槽数。该值限制了管道中可以同时存在多少个数据块，从而控制资源使用。
  - ring_size：缓冲区数组的总大小，通常是一个 2 的幂。使用幂次大小便于在计算 head 和 tail 时进行位运算，提高效率。
  - note_loss：（在 CONFIG_WATCH_QUEUE 配置下启用）用于标识下次 read() 时是否应该插入数据丢失（data-lost）的消息。这通常与监控或通知机制有关。
  - nr_accounted：表示该管道在用户空间管道缓冲区计数中所占用的数量，用于跟踪内存或资源使用情况。
  - readers 与 writers：分别记录当前打开该管道的读端和写端数量。它们帮助内核判断管道的使用状态，比如是否有写者还存在，进而决定是否需要发送信号给读者等。

7.2 管道

- 该结构体代表一个管道对象，封装了管道的状态、缓冲区数组以及读写双方的同步与管理机制。
  - files：表示引用该管道的 struct file 数量，即有多少文件描述符指向该管道。这有助于管理管道的生命周期和关闭操作。
  - r_counter 与 w_counter：分别用于记录读者和写者的计数器，这在某些情况下可用于调试、统计或更细粒度的资源管理。
  - poll_usage：一个布尔值，用于指示该管道是否用于 epoll 操作。由于 epoll 机制可能会导致频繁的唤醒，因此内核会通过此标志进行特殊处理。
  - tmp_page：用于缓存释放的页。当管道缓冲区释放后，为了避免频繁分配/释放内存，可以临时保存一个页以供复用。
  - fasync_readers 与 fasync_writers：分别用于管理读端和写端的异步通知（fasync），当管道状态发生变化时，内核可以通过这些结构向等待的进程发送信号。
  - bufs：指向管道缓冲区（pipe_buffer）的环形数组。这个数组保存了实际的数据块，每个元素就是一个 pipe_buffer。head 与 tail 通过索引访问这个数组，实现数据的循环存储。
  - user：指向创建该管道的用户信息（struct user），用于跟踪资源归属以及可能的资源配额管理。
  - watch_queue：（在 CONFIG_WATCH_QUEUE 配置下启用）指向一个监视队列，用于支持特定的监控机制。如果启用，内核可以通过该队列通知监视者管道状态的变化。

7.2 管道

```
struct pipe_buffer {
        struct page *page;
        unsigned int offset, len;
        const struct pipe_buf_operations *ops;
        unsigned int flags;
        unsigned long private;
};
```

- pipe_buffer表示管道中的一个缓冲区，每个缓冲区保存了一段数据。
  - page：指向包含数据的内存页（struct page）。管道的数据实际存放在页中，而此指针记录了对应的内存页。
  - offset：指定数据在该页内的起始偏移量。由于整个页可能被多个用途共享，这里用来标识管道数据在页内的具体位置。
  - len：表示从 offset 开始有效数据的长度。结合 offset 和 len，内核可以确定管道中本次操作所涉及的数据区域。
  - ops：指向与该缓冲区关联的操作集（struct pipe_buf_operations）。该接口提供了一组回调函数，用于处理诸如获取、释放、映射等对缓冲区数据的操作。这种设计使得管道可以支持多种数据缓冲区管理策略。
  - flags：用于记录缓冲区的标志信息，标志位具体意义由管道实现定义，通常用于描述缓冲区的状态或行为特性（例如只读、写时复制等）。
  - private：用于存放与该缓冲区相关的私有数据，由 ops 的回调函数使用。内核通过这个字段传递或保存额外的上下文信息，这样在执行操作时可以访问到特定数据。

7.2 管道

- Linux内核中管道的读写操作是一个复杂但高效的过程，涉及内核数据结构操作、内存管理、进程调度等多个子系统的协同工作。
- 写过程通过pipe_write函数实现。

```
static ssize_t
pipe_write(struct kiocb *iocb, struct iov_iter *from)
{
        struct file *filp = iocb->ki_filp;
        struct pipe_inode_info *pipe = filp->private_data;
        unsigned int head;
        ssize_t ret = 0;
        size_t total_len = iov_iter_count(from);
        ssize_t chars;
        bool was_empty = false;
        bool wake_next_writer = false;

        /* Null write succeeds. */
        if (unlikely(total_len == 0))
                return 0;

        __pipe_lock(pipe);

        if (!pipe->readers) {
                send_sig(SIGPIPE, current, 0);
                ret = -EPIPE;
                goto out;
        }
```

7.2 管道

- pipe_write 的核心逻辑可以概括为：
  - 首先，函数会检查写入请求的数据长度，如果为 0 则直接返回 0；
  - 接着对管道进行加锁保护，确保后续操作的原子性。
  - 在加锁后，函数首先判断管道中是否有读端存在，如果没有读端，则向当前进程发送 SIGPIPE 信号并返回 -EPIPE 错误。
    - 在一些特殊配置下（如 CONFIG_WATCH_QUEUE 启用时），函数也可能直接返回 -EXDEV，表示此类写入操作不被允许。
- 在正常写入的流程中，pipe_write 会先尝试将新写入的数据与管道中最后一个缓冲区合并，这主要针对小数据写入或部分写入操作。
  - 如果当前管道不是空的，并且新数据的大小（通常小于一页 PAGE_SIZE）满足与现有缓冲区合并的条件（即标志位允许合并且合并后的数据不会超过一个页的大小），则先通过确认缓冲区的状态（调用 pipe_buf_confirm）后，再将数据复制到已存在的页内，并更新该缓冲区中数据的长度。这样既能减少对内存页的额外分配，也提高了小数据写入的效率。

7.2 管道

- 如果合并不成功或数据量较大，则进入一个循环，不断尝试写入剩余数据。
  - 在每次循环中，函数会再次检查管道的读端是否存在，以防止在写入过程中出现读端中断的情况。
  - 如果管道未满（通过对比 head 与 tail 以及最大允许使用槽数判断），则函数尝试使用缓存页（tmp_page），如果缓存页不存在则分配一个新的页，并在自旋锁保护下为即将写入的数据预分配一个缓冲区槽。
  - 随后，将这个新分配的页以及相关的初始状态（如 offset 设置为 0，len 置为 0）插入到缓冲区数组中，并根据管道的类型设置合适的标志。数据则通过 copy_page_from_iter 从用户提供的 iov_iter 中复制到这个页中，如果复制的数据不足一页且仍有数据未复制，可能会因发生错误而返回 -EFAULT。
  - 每成功复制一部分数据，函数会累计返回的字节数，并更新缓冲区中记录的长度。

7.2 管道

- 读过程通过pipe_read实现。

```
static ssize_t
pipe_read(struct kiocb *iocb, struct iov_iter *to)
{
        size_t total_len = iov_iter_count(to);
        struct file *filp = iocb->ki_filp;
        struct pipe_inode_info *pipe = filp->private_data;
        bool was_full, wake_next_reader = false;
        ssize_t ret;

        /* Null read succeeds. */
        if (unlikely(total_len == 0))
                return 0;

        ret = 0;
        __pipe_lock(pipe);

        /*
         * We only wake up writers if the pipe was full when we started
         * reading in order to avoid unnecessary wakeups.
         *
         * But when we do wake up writers, we do so using a sync wakeup
         * (WF_SYNC), because we want them to get going and generate more
         * data for us.
         */
        was_full = pipe_full(pipe->head, pipe->tail, pipe->max_usage);
        for (;;) {
                /* Read ->head with a barrier vs post_one_notification() */
                unsigned int head = smp_load_acquire(&pipe->head);
                unsigned int tail = pipe->tail;
                unsigned int mask = pipe->ring_size - 1;
```

7.2 管道

- pipe_read 的核心逻辑可以概括为：
  - 在对管道加锁并确定读取总量后，函数循环依次确认缓冲区数据、处理丢失通知、复制数据到用户缓冲区、更新缓冲区指针；
  - 并在管道数据不足时适当等待新数据写入，最终释放锁、唤醒等待的写者并更新文件访问状态。
- 首先，函数会计算用户请求读取的数据总量，并对管道数据结构加锁，确保对共享数据操作的原子性。此时，如果读取长度为 0，则立即返回 0，不做其他处理。在加锁后，函数还会记录管道是否“满”——这一信息在后续阶段用于决定是否唤醒等待写入的进程，从而减少不必要的唤醒。

7.2 管道

- 进入主循环后，pipe_read 首先通过内存屏障安全地获取当前生产者（head）和消费者（tail）的索引，从而确定管道中是否有数据可读。
- 接下来，当管道不为空时，函数从缓冲区中取出第一个待消费的数据块。
  - 它首先确定本次可读的数据量，如果缓冲区中的数据量超过用户请求，并且缓冲区标志要求整体读取（如 PIPE_BUF_FLAG_WHOLE），则可能会返回 ENOBUFS 错误；否则，会根据用户请求的长度调整实际读取的数据量。
  - 随后，pipe_read 会调用 pipe_buf_confirm 确认缓冲区数据的有效性，再通过 copy_page_to_iter 将数据从内存页复制到用户提供的缓冲区中。如果数据复制成功，则更新缓冲区的偏移和剩余数据长度；如果缓冲区数据全部读取完毕，则调用 pipe_buf_release 释放该缓冲区，并在自旋锁保护下更新管道的消费指针。

7.2 管道-实例

- 匿名管道写端实现（Manager）：

```go
// pkg/ipc/ipc.go: 启动 Executor 并建立管道
cmd := exec.Command("./executor")
stdin, _ := cmd.StdinPipe()    // 获取 Executor 的 stdin 管道
stdout, _ := cmd.StdoutPipe()  // 获取 Executor 的 stdout 管道
cmd.Start()

// 发送测试用例到 Executor
stdin.Write(serializedProg)

// 读取 Executor 的执行结果
buf := make([]byte, 4096)
n, _ := stdout.Read(buf)
```

- 匿名管道读端实现（Executor）：

```cpp
// executor/executor.cc: 通过 stdin/stdout 与 Manager 通信
char progBuf[4096];
read(STDIN_FILENO, progBuf, sizeof(progBuf)); // 从 stdin 读取测试用例

// 执行测试程序后，将结果写入 stdout
write(STDOUT_FILENO, resultData, resultSize);
```

命名管道

- 命名管道（FIFO）是一种特殊的文件，用于实现不同进程间的通信，其特点是数据按照先入先出的顺序传输。与匿名管道（Pipe）不同，命名管道在文件系统中存在一个对应的文件节点，这使得它们可以通过文件路径进行访问和共享。
- 当我们调用系统调用 mkfifo（或 mknod）来创建 FIFO 时，实际上是向文件系统申请创建一种特殊的文件节点。

```
// 命令行创建
mkfifo /tmp/myfifo

// 系统调用创建
#include <sys/stat.h>
int mkfifo(const char *pathname, mode_t mode);
```

命名管道

```
static int fifo_open(struct inode *inode, struct file *filp)
{
        struct pipe_inode_info *pipe;
        bool is_pipe = inode->i_sb->s_magic == PIPEFS_MAGIC;
        int ret;

        filp->f_version = 0;

        spin_lock(&inode->i_lock);
        if (inode->i_pipe) {
                pipe = inode->i_pipe;
                pipe->files++;
                spin_unlock(&inode->i_lock);
        } else {
                spin_unlock(&inode->i_lock);
                pipe = alloc_pipe_info();
                if (!pipe)
                        return -ENOMEM;
                pipe->files = 1;
                spin_lock(&inode->i_lock);
                if (unlikely(inode->i_pipe)) {
                        inode->i_pipe->files++;
                        spin_unlock(&inode->i_lock);
                        free_pipe_info(pipe);
                        pipe = inode->i_pipe;
                } else {
                        inode->i_pipe = pipe;
                        spin_unlock(&inode->i_lock);
                }
        }
        filp->private_data = pipe;
        /* OK, we have a pipe and it's pinned down */

        __pipe_lock(pipe);

        /* We can only do regular read/write on fifos */
        stream_open(inode, filp);
```

命名管道

- 首先，fifo_open 检查 inode->i_pipe 是否已经存在。如果有其他进程之前打开了这个 FIFO，就直接复用这个 pipe，增加引用计数。否则，就分配一个新的 pipe 结构并初始化，将其绑定到 inode->i_pipe。
- 获取 pipe 后，将其赋值给 filp->private_data，表示当前文件描述符和这个 pipe 关联，同时调用 __pipe_lock 加锁，准备进行后续操作。
- 根据 filp->f_mode 判断打开模式：
  - 只读模式 (O_RDONLY)：增加读计数器，若没有写进程且非 O_NONBLOCK，会阻塞等待写者。
  - 只写模式 (O_WRONLY)：增加写计数器，如果没有读进程且非 O_NONBLOCK，会阻塞等待读者。
  - 读写模式 (O_RDWR)：同时增加读者和写者计数器，不会阻塞，因为读写双方都已存在。
- 一切正常后，解锁 pipe，并返回 0，表示成功打开。如果在过程中发生错误，则减少对应的计数器，并释放 pipe，返回相应的错误码。

命名管道-实例

- 命名管道写端实现（Manager）：

```
// pkg/log/log.go: 将日志写入有名管道
pipe, _ := os.OpenFile("/tmp/syzkaller.log.pipe", os.O_WRONLY, 0666)
pipe.WriteString("Crash detected: " + crashInfo)
```

- 命名管道读端实现（外部工具）：

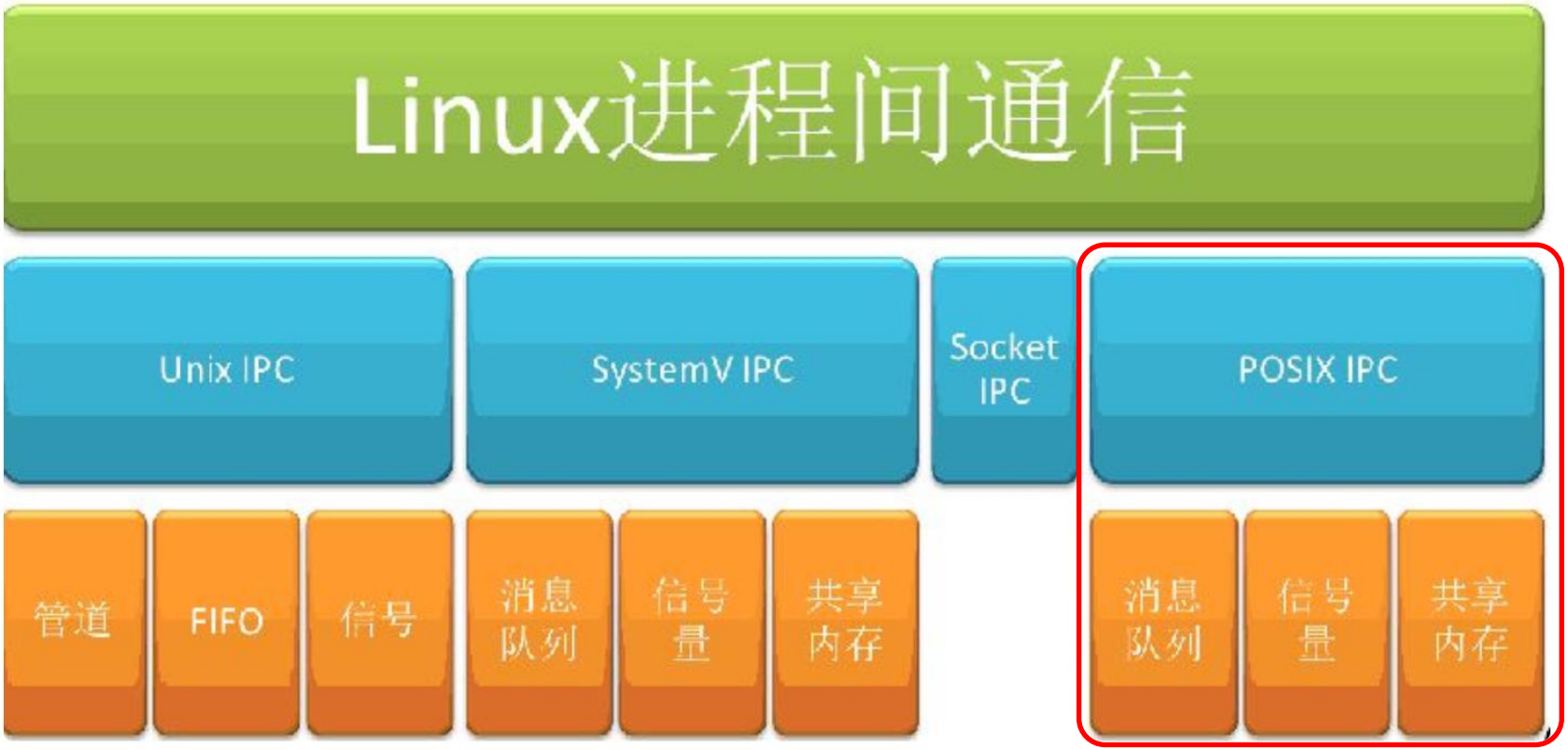
```
# 外部工具通过 cat 或自定义程序读取管道
mkfifo /tmp/syzkaller.log.pipe
cat /tmp/syzkaller.log.pipe
```

FIFO 与匿名管道的区别

- 虽然 FIFO 和匿名管道在读写操作上使用相同的 pipefifo_fops，但它们在创建和使用上的区别主要体现在下面几个方面：
  - 创建时机与数据结构初始化：
    - 匿名管道： 内核在创建 pipe 时会同时建立好读写两端以及关联的 pipe_inode_info 对象和缓冲区。
    - 命名管道： 通过 mkfifo 创建 FIFO 文件后，仅仅在文件系统中留下了一个特殊文件节点；实际的 pipe_inode_info 对象和数据缓冲区会在进程打开 FIFO（调用 fifo_open）时才创建。
  - 打开 FIFO 时的阻塞行为：
    - 如果一个进程以只读方式打开 FIFO 文件，而此时没有写进程打开，它会阻塞等待（除非设置了 O_NONBLOCK 标志），避免出现读不到数据的情况。
    - 同理，以只写方式打开 FIFO 文件时也会阻塞，直到有进程以读方式打开；在非阻塞模式下，如果没有对应的对端，则会返回错误。
    - 如果以读写方式打开 FIFO，则不会出现阻塞现象，因为该进程既能读又能写。

7.3 POSIX通信

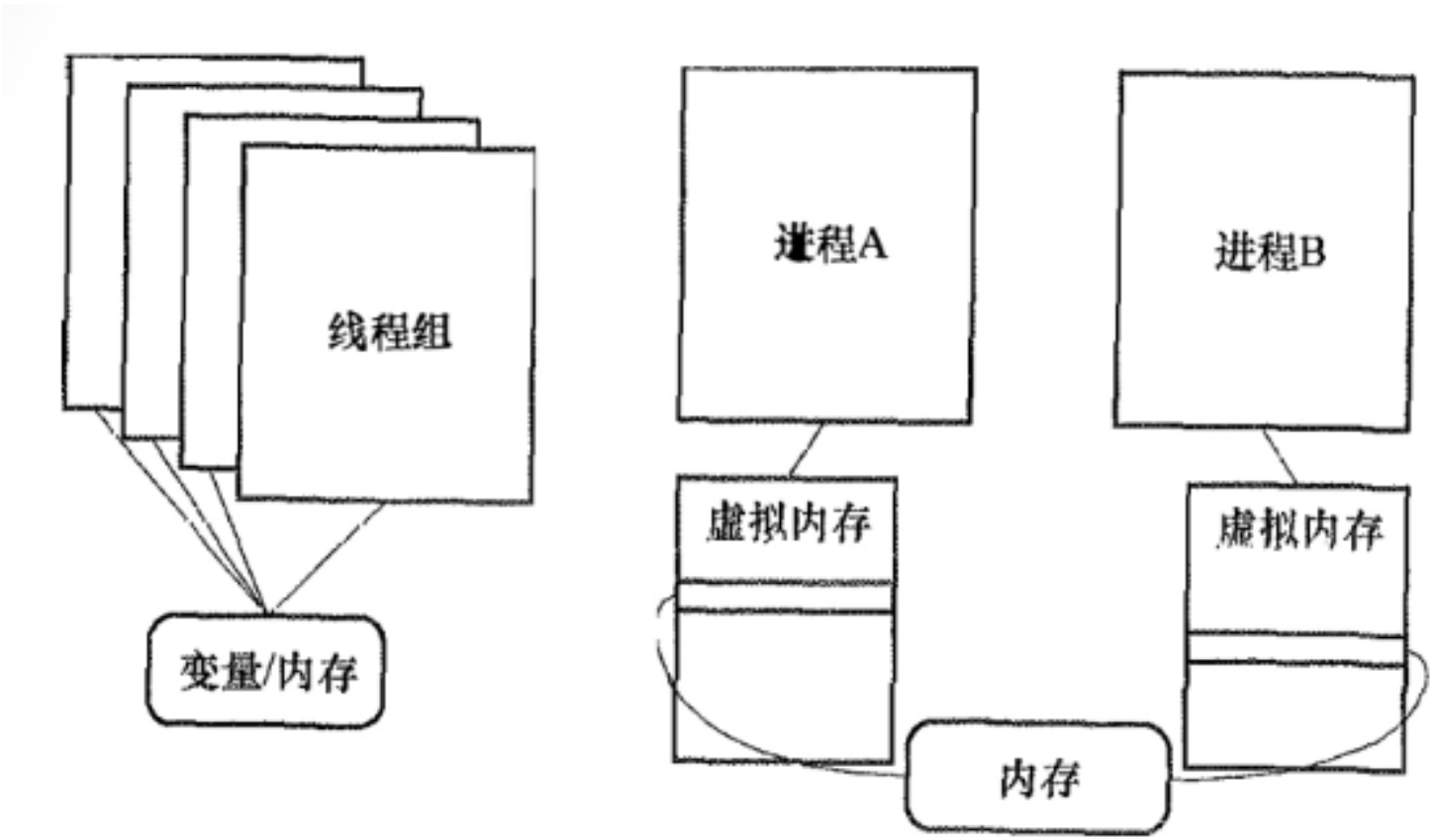
- POSIX通信包括semaphore（信号量）、shared memory（共享内存）和message queue（消息队列）3种，它们适用的场景不同。



7.3.1 POSIX信号量

- semaphore（信号量）可用于进程间同步，可以简单把它当成一个不会小于0的整数。值为0时，获取信号量的进程会阻塞直到它大于0，释放信号量它的值加一，获取信号量成功它的值减一。
- 既然是进程间共享，那么它必须对所有参与进程都可见，有两种策略可以实现：**有名信号量**和**内存信号量**。
- Linux实现有名信号量的方式是创建一个同名文件，进程间通信的载体就是该文件。内存信号量没有名字。这种情况下，通信的进程需要共享存放信号量的内存。
  - 多线程使用全局变量，多进程使用内存映射。

7.3.1 POSIX信号量



7.3.1 POSIX信号量

- 有名信号量通过sem_open创建，sem_unlink销毁。
  - name是信号量的唯一标识符，命名规则：
    - 必须以斜杠/开头（Linux要求），例如/my_semaphore。
    - 不能包含其他斜杠字符。
    - 最大长度通常为NAME_MAX（255字节）
  - oflag可以控制信号量的打开方式。

```
sem_t *sem_open(const char *name, int oflag, .../* mode_t mode */ );

int sem_unlink(const char *name);
```

7.3.1 POSIX信号量

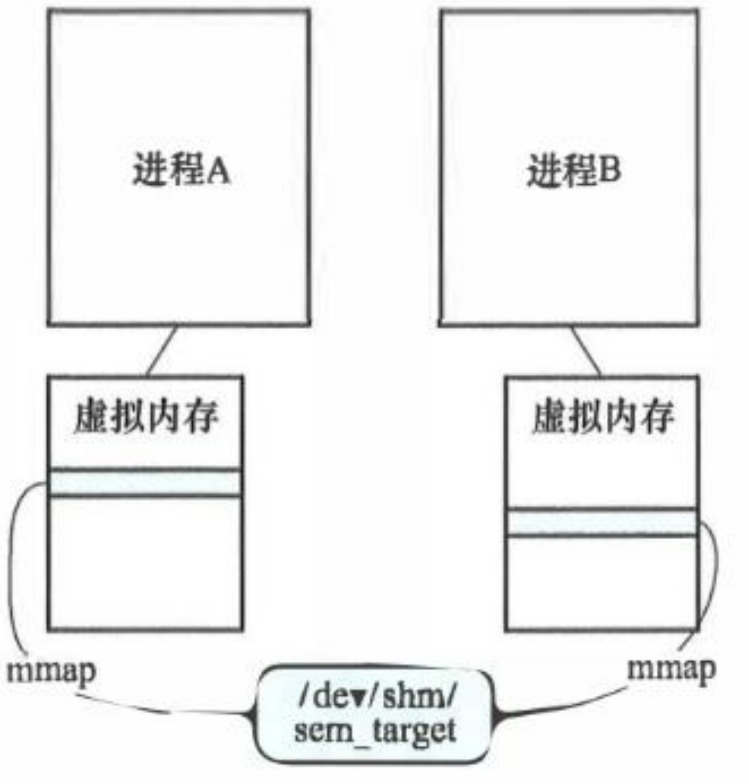
- 内存信号量本质上就是一块足够存放sem_t类型的变量的共享内存，不需要open。使用sem_init初始化，sem_destroy销毁。
- 在对信号量的使用上，两种方式并无差别。使用sem_post释放信号量，使用sem_wait获取信号量。
- sem是指向信号量对象的指针，pshared是进程共享标志位（0：线程间共享，1：进程间共享），value是信号量初始值。

```
int sem_init(sem_t *sem, int pshared, unsigned int value);

int sem_destroy(sem_t *sem);

int sem_post(sem_t *sem);

int sem_wait(sem_t *sem);
```

7.3.1 POSIX信号量

- POSIX信号量效率很高，主要得益于sem_post和sem_wait的实现方式：它们不涉及系统调用，而是通过原子操作直接操作内存实现的。
- 内核中没有sem_xxx类的系统调用，上述函数都是glibc封装的。有名信号量所需的sem_open的实现分两步：
  1. 根据传递的name得到目标文件路径。目标文件的目录不是随意的，glibc会选取tmpfs和shm类型的文件系统的挂载点，优先选择/dev/shm。
  2. 如果/dev/shm/name不存在且标志位置位了O_CREAT，则创建文件。然后调用mmap映射文件到内存，映射后的虚拟地址就是信号量的地址。
- 由此可见，POSIX信号量也是通过共享内存实现的，不过这段内存的载体是文件。

7.3.1 POSIX信号量



7.3.1 POSIX信号量-实例

- 信号量实现（Executor）：

```
// executor/executor.cc: 线程同步（假设场景）
#include <semaphore.h>
sem_t sem;
sem_init(&sem, 0, 1);  // 初始化信号量

// 线程1锁定资源
sem_wait(&sem);
// 操作共享资源...
sem_post(&sem);

// 线程2锁定资源
sem_wait(&sem);
// 操作共享资源...
sem_post(&sem);
```

7.3.2 POSIX共享内存

- POSIX的共享内存和信号量相似，没有专门的系统调用，由glibc封装（shm_open和shm_unlink）。
- name是共享内存对象的名称，oflag是打开方式标志组合，mode是权限模式（仅在创建时有效）。

```
int shm_open(const char *name, int oflag, mode_t mode);

int shm_unlink(const char *name);
```

- shm_open和sem_open很类似，不同的是shm_open只打开文件，没有调用mmap映射文件，而是返回一个文件描述符，让用户手动mmap。

7.3.2 POSIX共享内存-实例

- 共享内存实现（Executor）：

```
// executor/executor.cc: 映射 KCOV 共享内存
int kcovFd = open("/dev/kcov", O_RDWR);
ioctl(kcovFd, KCOV_INIT_TRACE, TRACE_PAGE_SIZE);
void *kcovMem = mmap(
    NULL,
    TRACE_PAGE_SIZE,
    PROT_READ | PROT_WRITE,
    MAP_SHARED,
    kcovFd,
    0
);
// 读取内核写入的覆盖率数据
uint64_t *cover = (uint64_t *)kcovMem;
```

7.3.3 POSIX消息队列

- 消息队列和前两者不同，它有一套专属的系统调用。

```
mqd_t mq_open(const char *name, int oflag);
mqd_t mq_open(const char *name, int oflag, mode_t mode,
                    struct mq_attr *attr);    //可变参数
int mq_close(mqd_t mqdes);
int mq_unlink(const char *name);
int mq_getattr(mqd_t mqdes, struct mq_attr *attr);
int mq_setattr(mqd_t mqdes, const struct mq_attr *newattr,
                    struct mq_attr *oldattr);
int mq_notify(mqd_t mqdes, const struct sigevent *sevp);
int mq_send(mqd_t mqdes, const char *msg_ptr,
                    size_t msg_len, unsigned int msg_prio);
int mq_timedsend(mqd_t mqdes, const char *msg_ptr,
                    size_t msg_len, unsigned int msg_prio,
                    const struct timespec *abs_timeout);
ssize_t mq_receive(mqd_t mqdes, char *msg_ptr,
                        size_t msg_len, unsigned int *msg_prio);
ssize_t mq_timedreceive(mqd_t mqdes, char *msg_ptr,
                        size_t msg_len, unsigned int *msg_prio,
                        const struct timespec *abs_timeout);
```

7.3.3 POSIX消息队列

- Linux以文件作为消息队列的载体，mq_open会创建文件，返回的消息队列描述符实际上就是文件描述符。

```
mqd_t mq_open(const char *name, int oflag, ... /* mode_t mode, struct mq_attr *attr */);
```

  - name：消息队列的唯一标识符。
  - oflag：通过位指定打开方式。
  - mode：设置队列的访问权限（类似文件权限）。

7.3.3 POSIX消息队列

- mq_open创建/打开的文件属于特殊的文件系统mqueue，它定义了一个内嵌inode，叫做mqueue_inode_info，由mqueue_alloc_inode分配。

```
struct mqueue_inode_info {
        spinlock_t lock;
        struct inode vfs_inode;
        wait_queue_head_t wait_q;

        struct rb_root msg_tree;
        struct rb_node *msg_tree_rightmost;
        struct posix_msg_tree_node *node_cache;
        struct mq_attr attr;

        struct sigevent notify;
        struct pid *notify_owner;
        u32 notify_self_exec_id;
        struct user_namespace *notify_user_ns;
        struct ucounts *ucounts;        /* user who created, for accounting */
        struct sock *notify_sock;
        struct sk_buff *notify_cookie;

        /* for tasks waiting for free space and messages, respectively */
        struct ext_wait_queue e_wait_q[2];

        unsigned long qsize; /* size of queue in memory (sum of all msgs) */
};
```

7.3.3 POSIX消息队列

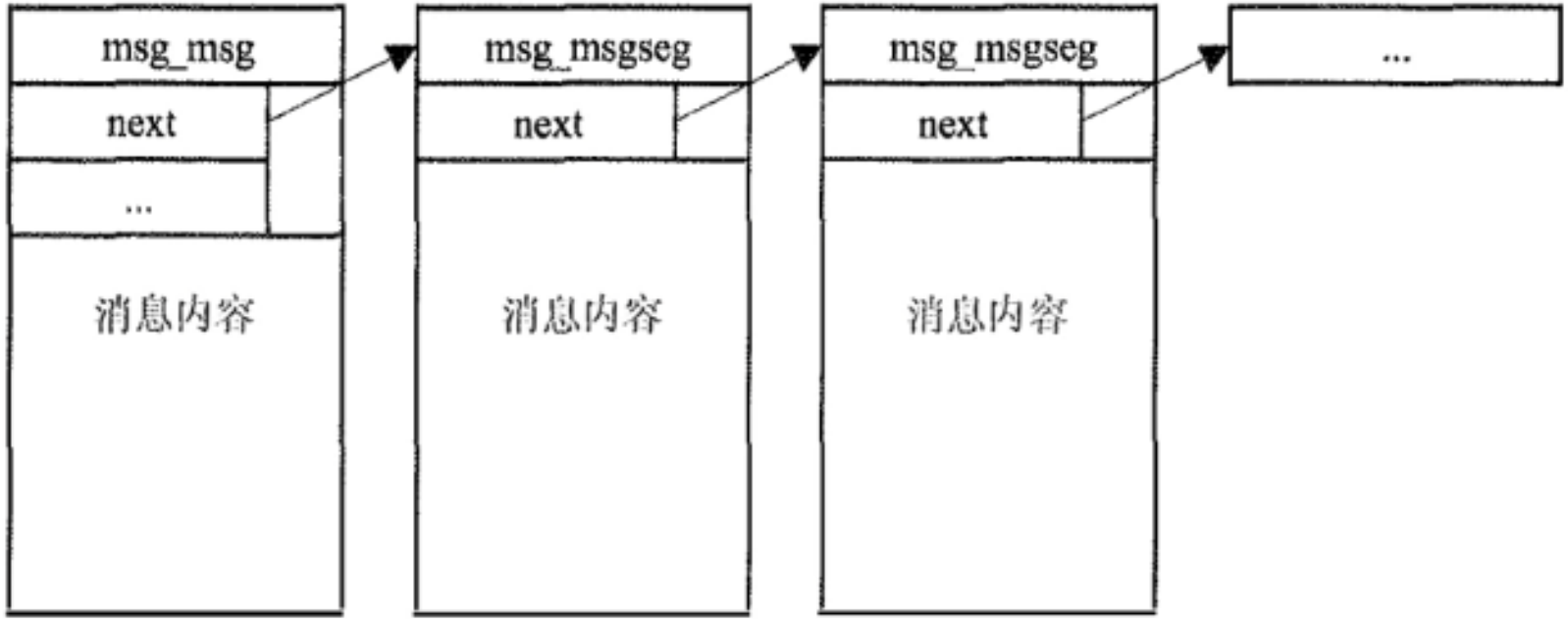
- spinlock_t lock：自旋锁保护消息队列的并发访问，确保原子操作；
- struct inode vfs_inode：嵌入VFS inode结构实现消息队列的文件系统抽象；
- wait_queue_head_t wait_q：管理等待消息到达/队列空间释放的阻塞进程队列；
- struct rb_root msg_tree：红黑树结构按优先级排序存储消息节点，实现高效消息检索；
- struct rb_node *msg_tree_rightmost：指向红黑树最右节点，快速定位最高优先级消息；
- struct posix_msg_tree_node *node_cache：缓存已释放的消息节点内存，提升消息分配性能；
- struct mq_attr attr：存储消息队列容量属性（mq_maxmsg/mq_msgsize）及运行时状态（mq_curmsgs）；
- struct sigevent notify：配置异步事件通知参数，定义消息到达时的信号或线程唤醒方式；
- struct pid *notify_owner：记录注册异步通知的进程PID，确保事件投递的目标准确性；
- u32 notify_self_exec_id：跟踪进程exec事件，防止通知发送到已替换镜像的进程；
- struct user_namespace *notify_user_ns：记录通知发起者的用户命名空间，用于权限校验；
- struct ucounts *ucounts：关联用户资源计数器，实施RLIMIT_MSGQUEUE配额限制；
- struct sock *notify_sock/netlink_cookie：支持通过Netlink socket发送异步事件通知的通信对象；
- struct ext_wait_queue e_wait_q[2]：分别管理等待队列非空（读等待）和队列未满（写等待）的进程；
- unsigned long qsize：实时统计队列内存占用量（所有消息长度总和），用于资源监控。

7.3.3 POSIX消息队列

- 由msg_tree可见，消息是由红黑树管理的。红黑树直接管理posix_msg_tree_node对象，每个对象都管理一组消息。它们的顺序由priority字段决定，越大优先级越高。优先级在mq_send时指定。同一个优先级的多个消息会在同一个posix_msg_tree_node的msg_list中。
- 消息由msg_msg结构体表示。

```
/* one msg_msg structure for each message */
struct msg_msg {
        struct list_head m_list;
        long m_type;
        size_t m_ts;            /* message text size */
        struct msg_msgseg *next;
        void *security;
        /* the actual message follows immediately */
};
```

7.3.3 POSIX消息队列



7.3.3 POSIX消息队列

- mq_send和mq_timesend用于发送消息，前者调用后者实现，最终都通过mq_timesend系统调用实现。主要实现以下步骤

1. 参数检查，消息的优先级不能大于MQ_PRIO_MAX（32768），消息的长度不能大于info->attr.mq_msgsize等。
2. 调用load_msg复制消息，它先调用alloc_msg申请足够内存存储msg_msg和msg_msgseg对象和消息，然后将消息从用户空间拷贝到内核空间（copy_from_user）。
3. 如果队列上消息已满，且文件置位了O_NONBLOCK时返回错误，没有置位的话则睡眠。在info->e_wait_q[SEND].list链表上的消息会被读取。
4. 如果消息未满，有进程等待接受消息（info->e_wait_q[RECV].list不为空）的情况下，调用pipeline_send唤醒链表上的最后一个进程（优先级最高）接受消息。没有进程接受消息的情况下，将消息按照优先级插入info->msg_tree红黑树中

7.3.3 POSIX消息队列

- mq_receive和mq_timereceive用来接受消息。任务如下：
  1. 参数检查，要读的消息长度msg_len不能小于info->attr.mq_msgsize。msg_len是用户提供的，过短可能无法读完一整条信息。
  2. 如果队列上没有消息，且文件置位了O_NONBLOCK时返回错误，没有置位的话则睡眠，在info->e_wait_q[RECV].list链表上等待消息。如果成功等到消息，则复制消息到用户态。
  3. 如果队列上有消息，则调用msg_get获取消息，然后将消息复制到用户空间。如果info->e_wait_q[SEND].list链表上有进程在等待，则找到最后一个进程（优先级最高），将它的消息插入队列并唤醒。

7.3.3 POSIX消息队列-实例

- 消息队列写端实现（Manager）：

```
/*
#include <mqueue.h>
mqd_t mq_open(const char *name, int oflag, ...);
*/
import "C"

// 打开消息队列
qName := C.CString("/syz-tasks")
mq := C.mq_open(qName, C.O_WRONLY)
C.mq_send(mq, dataPtr, C.size_t(dataSize), 0)
```

- 消息队列读端实现（外部插件）：

```
// 插件接收消息队列任务
mqd_t mq = mq_open("/syz-tasks", O_RDONLY);
mq_receive(mq, buf, sizeof(buf), NULL);
```

7.4 横向对比

- 这里通过一个实验测试不同进程间通信方式（匿名/有名管道、POSIX信号量/共享内存/消息队列）的性能差异。
- 数据量：传输总数据量1MB，分块大小为1KB（共1024次传输）。
- 信号量测试：执行2048次同步操作（每次 post + wait）。

7.4 横向对比

- 匿名管道测试

```
[测试匿名管道]
>> 第 1 次运行
[匿名管道] 耗时: 1.944 毫秒
>> 第 2 次运行
[匿名管道] 耗时: 1.432 毫秒
>> 第 3 次运行
[匿名管道] 耗时: 2.176 毫秒
>> 第 4 次运行
[匿名管道] 耗时: 1.588 毫秒
>> 第 5 次运行
[匿名管道] 耗时: 1.389 毫秒
>> 第 6 次运行
[匿名管道] 耗时: 1.412 毫秒
>> 第 7 次运行
[匿名管道] 耗时: 1.523 毫秒
>> 第 8 次运行
[匿名管道] 耗时: 2.007 毫秒
>> 第 9 次运行
[匿名管道] 耗时: 1.336 毫秒
>> 第 10 次运行
[匿名管道] 耗时: 1.576 毫秒
```

7.4 横向对比

- 有名管道测试

```
[测试有名管道]
>> 第 1 次运行
[有名管道] 耗时: 2.223 毫秒
>> 第 2 次运行
[有名管道] 耗时: 2.455 毫秒
>> 第 3 次运行
[有名管道] 耗时: 2.230 毫秒
>> 第 4 次运行
[有名管道] 耗时: 2.144 毫秒
>> 第 5 次运行
[有名管道] 耗时: 2.438 毫秒
>> 第 6 次运行
[有名管道] 耗时: 2.322 毫秒
>> 第 7 次运行
[有名管道] 耗时: 2.455 毫秒
>> 第 8 次运行
[有名管道] 耗时: 2.305 毫秒
>> 第 9 次运行
[有名管道] 耗时: 2.181 毫秒
>> 第 10 次运行
[有名管道] 耗时: 2.365 毫秒
```

7.4 横向对比

- 共享内存测试

```
[测试共享内存]
>> 第 1 次运行
[共享内存] 同步耗时: 12.727 毫秒
>> 第 2 次运行
[共享内存] 同步耗时: 10.371 毫秒
>> 第 3 次运行
[共享内存] 同步耗时: 8.351 毫秒
>> 第 4 次运行
[共享内存] 同步耗时: 5.284 毫秒
>> 第 5 次运行
[共享内存] 同步耗时: 9.251 毫秒
>> 第 6 次运行
[共享内存] 同步耗时: 8.463 毫秒
>> 第 7 次运行
[共享内存] 同步耗时: 10.115 毫秒
>> 第 8 次运行
[共享内存] 同步耗时: 6.697 毫秒
>> 第 9 次运行
[共享内存] 同步耗时: 4.751 毫秒
>> 第 10 次运行
[共享内存] 同步耗时: 4.626 毫秒
```

7.4 横向对比

- 消息队列测试

```
[测试消息队列]
>> 第 1 次运行
[消息队列] 吞吐量: 784313.73 KB/s
>> 第 2 次运行
[消息队列] 吞吐量: 963855.42 KB/s
>> 第 3 次运行
[消息队列] 吞吐量: 1212121.21 KB/s
>> 第 4 次运行
[消息队列] 吞吐量: 1111111.11 KB/s
>> 第 5 次运行
[消息队列] 吞吐量: 975609.76 KB/s
>> 第 6 次运行
[消息队列] 吞吐量: 349344.98 KB/s
>> 第 7 次运行
[消息队列] 吞吐量: 1066666.67 KB/s
>> 第 8 次运行
[消息队列] 吞吐量: 1066666.67 KB/s
>> 第 9 次运行
[消息队列] 吞吐量: 1095890.41 KB/s
>> 第 10 次运行
[消息队列] 吞吐量: 1126760.56 KB/s
```

7.4 横向对比

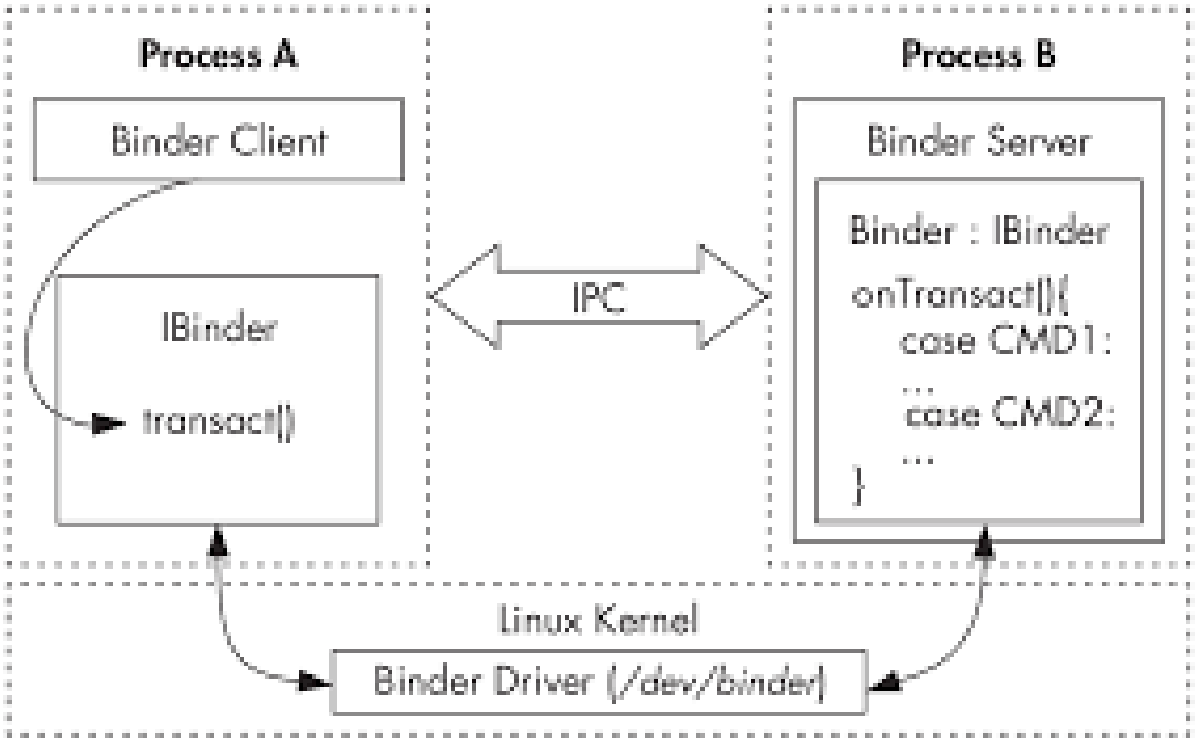
- 信号量测试

```
[测试信号量]
>> 第 1 次运行
[信号量] 平均延迟: 0.516 微秒/次
>> 第 2 次运行
[信号量] 平均延迟: 0.480 微秒/次
>> 第 3 次运行
[信号量] 平均延迟: 0.456 微秒/次
>> 第 4 次运行
[信号量] 平均延迟: 0.216 微秒/次
>> 第 5 次运行
[信号量] 平均延迟: 0.252 微秒/次
>> 第 6 次运行
[信号量] 平均延迟: 0.398 微秒/次
>> 第 7 次运行
[信号量] 平均延迟: 0.599 微秒/次
>> 第 8 次运行
[信号量] 平均延迟: 0.446 微秒/次
>> 第 9 次运行
[信号量] 平均延迟: 0.662 微秒/次
>> 第 10 次运行
[信号量] 平均延迟: 0.581 微秒/次
```

7.4 Binder机制-为什么需要 Binder?

- 传统 IPC 机制如管道、Socket 和共享内存在 Android 系统中存在显著缺陷。这些机制通常需要多次数据拷贝和上下文切换，导致性能低下，难以满足移动设备对高效通信的需求。此外，传统 IPC 缺乏细粒度的权限控制机制，无法保障多应用场景下的安全性，容易引发恶意访问或数据泄露。
- Binder 的诞生正是为了解决这些问题。它的设计目标聚焦于三个核心方向：高效性、安全性和易用性。通过内存映射（mmap）技术，Binder 实现了单次数据拷贝，大幅降低通信延迟。同时，Binder 基于 Linux 的 UID/PID 机制进行身份校验，确保只有授权进程可以访问敏感服务。对于开发者，Binder 通过高层抽象（如 AIDL）隐藏了底层细节，简化了跨进程调用的实现。

7.4 Binder机制-为什么需要 Binder?



7.4 Binder机制-Binder 的应用场景

- Binder 在 Android 系统中无处不在。例如，系统服务（如 ActivityManagerService）运行在独立进程中，应用需通过 Binder 调用其接口以执行启动 Activity、绑定服务等操作。

- 跨进程数据共享场景中，ContentProvider 和 BroadcastReceiver 也依赖 Binder 实现数据传递。例如，应用通过 ContentResolver 访问其他应用的数据时，Binder 负责在进程间封装和传输查询请求及结果。

- 此外，应用间通信（如支付 SDK 与宿主应用的交互）同样基于 Binder。开发者通过 bindService 绑定到远程服务，Binder 在此过程中代理方法调用并管理通信生命周期。

7.4 Binder机制-技术解析

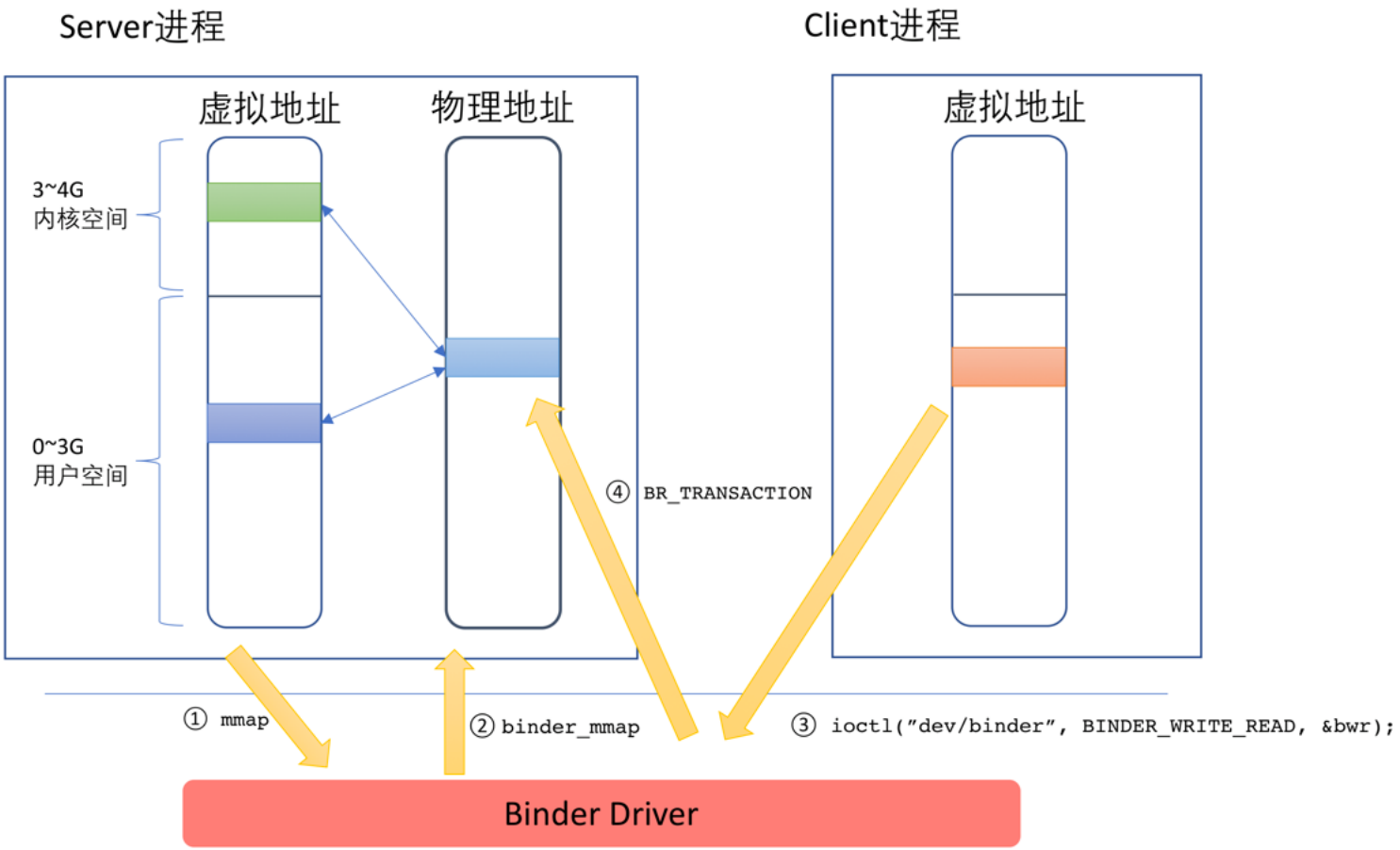
- /dev/binder 是 Linux 内核中 Binder 驱动的设备文件接口，扮演用户空间与内核通信的桥梁。用户程序通过 open 和 ioctl 等系统调用操作该设备文件，从而触发 Binder 驱动的逻辑。

- 权限管理是 /dev/binder 的重要特性。其默认权限为 rw-rw----，仅允许 system 组进程直接访问。普通应用无法绕过 Android 框架层直接操作该设备，这有效防止了恶意进程滥用 Binder 驱动。

7.4 Binder机制-技术解析

- Binder 驱动通过内存映射（mmap）建立共享内存区域，实现高效数据传输。当进程调用 mmap 时，驱动为其分配一块内核与用户空间共享的内存。数据从发送方用户空间拷贝至共享区域后，接收方可直接读取，避免了传统 IPC 的多次拷贝开销。
- 事务处理是 Binder 驱动的另一核心功能。客户端通过 BINDER_WRITE_READ 命令提交事务，驱动解析请求数据（如方法名、参数）后，将其路由至目标进程的线程池。服务端线程处理请求并返回结果，驱动再将响应数据写回客户端共享内存。
- 引用计数机制则用于管理 Binder 对象的生命周期。每个 Binder 对象（如服务接口）维护一个引用计数器，跨进程传递时计数器递增。当引用归零时，驱动自动释放对象占用的资源，避免内存泄漏。

7.4 Binder机制-技术解析



7.4 Binder机制-Binder 驱动的内核实现

- Binder 驱动的源码位于 Linux 内核的 drivers/android/binder.c，涵盖事务路由、线程调度和内存管理等核心逻辑。例如，binder_transaction 函数负责解析请求数据并投递至目标进程的待处理队列。

- Android 内核补丁对标准 Linux 内核进行了扩展，以支持 Binder 的独特需求。例如，新增的 BINDER_TYPE_PTR 数据类型用于在共享内存中传递对象引用，减少数据拷贝开销。

实验

- 实验一：根据提供的测试代码，测试不同IPC方式的效率。参考：http://10.10.21.30/linux-kernel/practice_kern/-/tree/main/IPCtest。
- 实验二：实现用户态代码，通过binder进行通信。
  1. 理解 Binder 进程间通信（IPC）机制的核心原理。
  2. 掌握 Binder 驱动与用户空间程序的交互方式。
  3. 熟悉 Binder 事务（Transaction）的构造与处理流程。
  - 参考：http://10.10.21.30/linux-kernel/practice_kern/-/tree/main/BinderIPC