

Lecture 10.1

Inter-process Communication pipes, FIFOs and shared memory

Md Mahbub E Noor

Assistant Professor, Department of CSE

University of Barishal

```
● ● ● ./pipe1 (real)
pipe() gave fd[0] = 3 (read end),
           fd[1] = 4 (write end)
parent 246 wrote 20 bytes
child  247 read 20 bytes:
        "hello through a pipe"
child  247 read again: 0
# 0 = end of file
```

What you will learn today



Say why processes need the kernel to talk, and name the main ways.



Build a pipeline with pipe, fork and dup2; explain blocking, end of file and SIGPIPE.



Describe a pipe inside the kernel: a ring of 16 pages.



Share memory between processes, show a race, and fix it.

Today's plan (60 min)

0-5

Recap of 9

5-10

Why IPC

10-28

Pipes

28-38

Inside the kernel

38-43

FIFOs

43-55

Shared memory

55-60

Checkpoint

Lab 10.1: pipes in mysh.

Quick recap of Lecture 9

Q1

Which register holds the system call number? Which holds the result?

rax both times: the number going in, the result coming out.

Q2

`write(42, "x", 1)` and fd 42 is not open. What does the kernel return? What does your C code see?

The kernel: `-9`. Your code: `-1` with `errno = 9` (`EBADF`).

Q3

Why is one write of 4096 bytes better than 4096 writes of 1 byte?

Each system call costs ≈ 100 ns (≈ 40 function calls). Today: the same rule makes small pipe writes slow.

PART 1

Why processes need help to talk

Each process has its own memory. Every channel between them goes through the kernel.

Separate memories, so the kernel must carry the message

Way	What moves	Who can use it	Lecture
pipe	a stream of bytes, one way	parent and child (related)	today
FIFO (named pipe)	a stream of bytes, one way	any process that can open the name	today
shared memory	nothing moves: both see the same page	related (mmap) or by name (shm_open)	today
Unix domain socket	bytes or messages, both ways	any process, by a file name	CN course
message queue	whole messages, with priorities	any process, by a name	TLPI Ch. 52
signal	only a number: "something happened"	any process you may signal	10.2

Pipe: a tube between two houses: what goes in one end comes out the other, in order

Shared memory: a whiteboard both houses can see: fastest, but two people writing at once make a mess

Signal: a doorbell: it says only that someone rang

SAME AS C

Like two functions

Two C functions talk through arguments and return values: the caller hands data to the callee. A pipe hands bytes from one process to another.

DIFFERENT FROM C

But no shared variables

After fork, x in the child is a copy (Homework 8.1: same &x, different values). A global variable cannot carry a message between processes.

Book: Kerrisk, TLPI Ch. 43 (IPC overview); Bovet & Cesati Ch. 19

PART 2

Pipes

`pipe()`, `fork()`, `dup2()`: the three calls behind every `|` in the shell.

pipe(): two file descriptors and a buffer in the kernel

● ● ● pipe1.c (shortened)

```
int fd[2]; pipe(fd);          /* fd[0] read, fd[1] write */
if (fork() == 0) {            /* child: the reader */
    close(fd[1]);
    n = read(fd[0], buf, sizeof buf - 1);
    ...
    n = read(fd[0], buf, sizeof buf); /* again */
    return 0;
}
close(fd[0]);                  /* parent: the writer */
write(fd[1], msg, strlen(msg));
close(fd[1]);                  /* gives the child EOF */
wait(NULL);
```

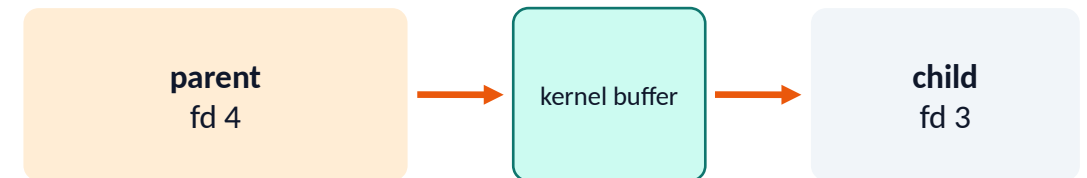
SAME AS C

Like a file

read and write work as on any open file (Lecture 12). fds 0, 1, 2 are taken: the pipe gets 3 and 4.

● ● ● ./pipe1 (real)

```
pipe() gave fd[0] = 3 (read end), fd[1] = 4
parent 246 wrote 20 bytes
child 247 read 20 bytes: "hello through a pipe"
child 247 read again: 0 (0 = end of file)
```



DIFFERENT FROM C

But no name, no seeking

A pipe has no file name and no position: bytes come out in the order they went in, and each byte can be read only once. fork makes the child inherit both ends.

Book: Kerrisk, TLPI §44.2 (creating and using pipes)

Three rules every pipe user must know

● ● ● block.c (QEMU guest, real)

```
# write(100000) into a 64 KiB pipe,  
# nobody reads for 1 s:  
writer 87: state S,  
  waiting in pipe_write  
writer: write returned 100000  
reader got 100000 bytes
```

● ● ● eof.c (real)

```
$ ./eof  
parent read 14 bytes  
parent: read returned 0: end of file  
$ timeout 3 ./eof forget  
parent read 14 bytes  
exit 124: killed after 3 s, still  
waiting
```

● ● ● sigpipe.c ; bash (real)

```
default: child killed by signal 13  
SIGPIPE ignored: write returned -1,  
  errno 32 (Broken pipe)  
$ yes | head -2; echo ${PIPESTATUS[*]}  
141 0      # 141 = 128 + 13
```

1. Full pipe: the writer waits

A pipe holds 64 KiB. A faster writer sleeps (state S) until the reader makes room. Nobody loses data; the writer is just slowed down.

2. End of file needs ALL write ends closed

read() returns 0 only when no process holds a write end any more, including the reader itself and the shell. Forget one close, and the reader waits for ever.

3. Nobody reads: SIGPIPE

Writing into a pipe with no reader kills the writer with SIGPIPE (13), or returns EPIPE if the signal is ignored. That is how `yes | head` stops `yes`.

How the shell runs `ls / | wc -l`

```
● ● ● strace -f bash -c 'ls / | wc -l' (real, the important lines)
```

```
367 pipe2([3, 4], 0)          = 0
367 clone(...SIGCHLD...)      = 368    # child: ls
367 close(4)                  = 0
367 clone(...SIGCHLD...)      = 369    # child: wc
367 close(3)                  = 0
368 close(3)                  = 0
368 dup2(4, 1)                 = 1      # stdout -> pipe
368 close(4)                  = 0
369 dup2(3, 0)                 = 0      # stdin <- pipe
369 close(3)                  = 0
368 execve("/usr/bin/ls", ["ls", "/"], ...) = 0
369 execve("/usr/bin/wc", ["wc", "-l"], ...) = 0
367 wait4(-1, ...) = 368 ; wait4(-1, ...) = 369
```

	fd 0	fd 1	fd 3	fd 4
ls (368)	terminal	pipe (write)	—	—
wc (369)	pipe (read)	terminal	—	—
bash (367)	terminal	terminal	—	—

```
● ● ● sleep 30 | sleep 30 ; ls -l /proc/PID/fd
```

```
pid 349:
l-wx----- ... 1 -> pipe:[2170]
pid 350:
lr-x----- ... 0 -> pipe:[2170]
```

dup2(old, new) makes descriptor new point to the same open file as old. After `dup2(4, 1)`, every `printf` of `ls` goes into the pipe, and `ls` never knows. Programs need no special code to be used in a pipeline.

How big, and how fast? (pipecap.c, pipebw.c)

```
./pipecap (real, same in the guest)
F_GETPIPE_SZ = 65536 bytes
filled with 65536 bytes, then write() said EAGAIN
F_SETPIPE_SZ 1 MiB -> 1048576 bytes
filled with 1048576 bytes
$ cat /proc/sys/fs/pipe-max-size
1048576
```

1 GiB through a pipe, chunk size	Speed	write calls
64 bytes	≈ 85 MB/s	16 777 216
4096 bytes	≈ 2100 MB/s	262 144
65536 bytes	≈ 1600 MB/s	16 384

pipebw.c, two runs on our test machine; varies ±15 %

64 KiB by default

16 pages of 4 KiB. A program can ask for up to 1 MiB with `fcntl(F_SETPIPE_SZ)`.

PIPE_BUF = 4096

A write of at most 4096 bytes is never mixed with another writer's bytes. Bigger writes may be (Homework 2).

Small writes are slow

64-byte chunks: 25 times slower. The cost is the system calls (Lecture 9), not the copying.

C link: `fcntl(fd, F_GETPIPE_SZ)` is the same kind of call as `fcntl(fd, F_SETFL, O_NONBLOCK)`: one system call with a command number, used for many small settings of a file descriptor.

PART 3

A pipe inside the kernel

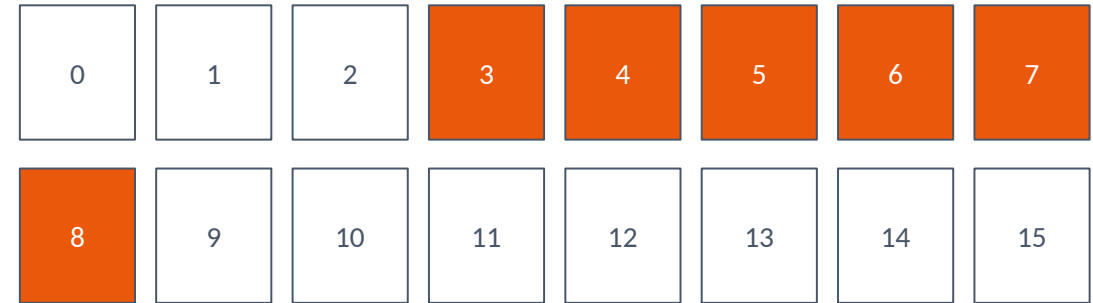
fs/pipe.c: a ring of pages, two wait queues, two counters.

struct pipe_inode_info: a ring buffer of pages

```
include/linux/pipe_fs_i.h, Linux 6.8 (real lines)

5  #define PIPE_DEF_BUFFERS 16
58 struct pipe_inode_info {
60     wait_queue_head_t rd_wait, wr_wait;
61     unsigned int head;
62     unsigned int tail;
63     unsigned int max_usage;
66     unsigned int readers;
67     unsigned int writers;
78     struct pipe_buffer *bufs; /* the ring */
...
148 pipe_empty(): return head == tail;
169 pipe_full(): occupancy(head, tail) >= limit
```

C link: the circular buffer from your data-structures course: an array, a read index and a write index, $i \% N$. With N a power of two, $i \& (N - 1)$ does the same job with no division.



tail = 3 (next to read) head = 9 (next to fill)

each box: one 4 KiB page; slot = counter & 15

head and tail only grow: They are unsigned counters. The slot is counter & (16 - 1); the ring wraps around by itself. Empty: head == tail. Full: head - tail == 16.

Two wait queues: A writer that finds the ring full sleeps on wr_wait; a reader that finds it empty sleeps on rd_wait. Each side wakes the other.

Book: Bovet & Cesati Ch. 19 (pipes); the kernel source

pipe_write and pipe_read: the three rules in the source

fs/pipe.c pipe_write, Linux 6.8 (real lines)

```
467 if (!pipe->readers) {
468     send_sig(SIGPIPE, current, 0);
469     ret = -EPIPE;
    ...
516 if (!pipe_full(head, pipe->tail, ...)) {
523     page = alloc_page(...);
536     pipe->head = head + 1;
550     copied = copy_page_from_iter(page, 0,
        PAGE_SIZE, from);
    ...
567 if (filp->f_flags & O_NONBLOCK) ...
570     ret = -EAGAIN;
589 wait_event_interruptible_exclusive(
    pipe->wr_wait, pipe_writable(pipe));
```

fs/pipe.c pipe_read, Linux 6.8 (real lines)

```
315 if (!pipe_empty(head, tail)) {
337     written = copy_page_to_iter(buf->page,
        buf->offset, chars, to);
354     ... tail = pipe_update_tail(...);
    }
362 if (!pipe->writers)
363     break; /* end of file */
400 wait_event_interruptible_exclusive(
    pipe->rd_wait, pipe_readable(pipe))
```

Two copies: the writer's bytes are copied into a kernel page (`copy_page_from_iter`), and later out to the reader (`copy_page_to_iter`). Safe and simple, but every byte is copied twice. Shared memory (Part 5) copies nothing.

Watch 100000 bytes squeeze through a 64 KiB pipe

A pipe: 16 pages in a ring (fs/pipe.c)

WRITE

FULL

READ

EOF

writer (child)

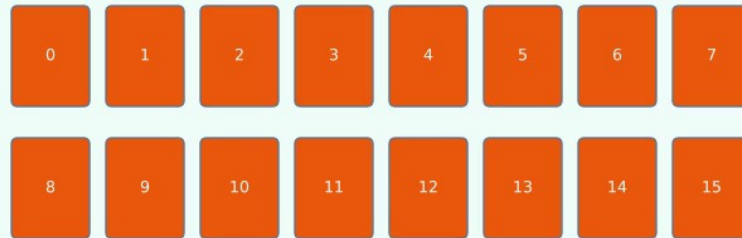
```
write(4, buf, 100000)
```

state S

sent 65,536 B

write() not finished

kernel: struct pipe_inode_info



bufs[16]: one 4 KiB page each = 64 KiB

head = 16 tail = 0 pages in use: 16

writers = 1 readers = 1

wr_wait: writer sleeps here rd_wait: reader sleeps here

reader (parent)

```
read(3, buf, 65536)
```

state S

got 0 B

sleeping 1 s

Watch for

1. **16 pages** fill up: 65536 bytes.
2. The writer **sleeps** inside one write().
3. One read frees **all 16** pages.
4. The ring **wraps**: slots 0 to 8 again.
5. **writers = 0** and empty: read returns 0.

Its write() has not returned yet: it is still busy with the same call. A writer that is faster than its reader is slowed down automatically. This is called back-pressure.

Click the video to play or pause. About 1 minute, no sound.

PART 4

FIFOs: pipes with a name

Unrelated processes can find each other through the file system.

A FIFO is a pipe that has a file name

```
● ● ● fifo_demo.sh (real; the guest says the same)
$ mkfifo /tmp/myfifo
$ ls -l /tmp/myfifo
prw-r--r-- 1 root root 0 Oct 14 15:31 /tmp/myfifo
$ cat /tmp/myfifo &                # the reader
$ cat /proc/$!/wchan
wait_for_partner
$ echo "hello from another process" > /tmp/myfifo
hello from another process
```

p: the file type in `ls -l`: a FIFO. Its size is always 0: the data never touches the disk.

open() waits: Opening for reading waits until someone opens for writing, and the other way round: `wait_for_partner` (`fs/pipe.c 1092`).

Same code: After `open`, reads and writes go to `pipe_read` and `pipe_write`: the same ring, the same three rules.

SAME AS C

Like `fopen` on a file

Any program that can open a file can use a FIFO: `cat`, `echo >`, your C code with `open()`. No fork needed: the name is the meeting point.

DIFFERENT FROM C

But not stored

Nothing is written to the disk. When both sides close, the data is gone; only the name stays until `rm`.

Book: Kerrisk, TLPI §44.7 (FIFOs)

PART 5

Shared memory, and the race it brings

No copies at all. The price: you must coordinate yourself.

One page, two processes: MAP_SHARED

● ● ● shmrace.c (core lines)

```
struct shared { volatile long counter; sem_t lock; };
struct shared *s = mmap(NULL, sizeof *s,
    PROT_READ | PROT_WRITE,
    MAP_SHARED | MAP_ANONYMOUS, -1, 0);
for (k = 0; k < 2; k++)
    if (fork() == 0) { /* both children see the same page */
        for (i = 0; i < N; i++) s->counter++;
```

	fork, normal memory	fork, MAP_SHARED
after fork	copy-on-write: each gets its own copy when it writes	the same physical page in both
child writes x = 2	parent still sees its old x	parent sees 2 at once
copies per message	—	zero: just write to memory

Related processes

mmap with MAP_SHARED | MAP_ANONYMOUS before fork, as here.

Unrelated processes

shm_open("/name", ...) gives a file in /dev/shm; each process mmmaps it.

Fastest IPC

No system call per message, no copy. But: who writes when? Next slide.

Book: Kerrisk, TLPI §49.7 (anonymous mappings) and Ch. 54

Two processes, one counter: a race condition

```
./shmrace (2 CPUs, real) 2 x 10 000 000 increments
```

```
plain      counter = 11754548 (expected 20000000) 0.06 s
atomic     counter = 20000000 (expected 20000000) 0.54 s
semaphore  counter = 20000000 (expected 20000000) 2.25 s
taskset -c 0 (1 CPU), and the 1-CPU guest: plain = 20000000
```

```
objdump -d shmrace (real)
```

```
1480: mov 0x0(%rbp),%rax # plain: load
1484: add $0x1,%rax      #      add
1488: mov %rax,0x0(%rbp) #      store
14b0: lock addq $0x1,0x0(%rbp) # atomic
```

time	process A (CPU 0)	process B (CPU 1)	counter
1	load counter → 5		5
2		load counter → 5	5
3	add 1 → 6, store		6
4		add 1 → 6, store	6 (lost one)

Atomic instruction: `lock add`: the CPU makes load-add-store one step for all CPUs. 9× slower than plain, but correct.

Semaphore: `sem_wait` / `sem_post` around the critical section: only one process inside. Correct for any amount of work; slowest here.

The worst kind of bug: on one CPU it did not show at all. It appears only sometimes, only on some machines. Shared data needs a rule, every time.

Book: Bryant & O'Hallaron, CS:APP §12.5 (*badcnt.c*, *semaphores*)

Which one should I use?

	Pipe	FIFO	Shared memory
Data copies	2 (in and out of the kernel)	2	0
System calls per message	1 write + 1 read	same	none (after mmap)
Who can use it	related processes	anyone who can open the name	related, or by name (shm_open)
Order and waiting	built in: FIFO order, blocking	built in	your job: atomics, semaphores
Ends by itself	EOF when writers close	same	no: you need your own signal
Typical use	shell pipelines, parent ↔ child	simple local servers, logs	big data: video frames, databases

Rule of thumb: start with a pipe. It is simple and safe, and the kernel does the waiting for you. Move to shared memory only when you have measured that copying is the problem, and then plan the synchronisation before you write the first line.

LAB 10.1

Pipes in mysh

Your shell from Lab 8.1 learns `cmd1 | cmd2`.

Lab 10.1 tasks, step by step

- 1 **TODO 1** in mysh_pipe_starter.c: find the word `|` in `argv`
- 2 **TODO 2**: `pipe(fd)`
- 3 **TODO 3**: child 1: `dup2(fd[1], 1)`, close both ends, `run(left)`
- 4 **TODO 4**: child 2: `dup2(fd[0], 0)`, close both ends, `run(right)`
- 5 **TODO 5**: the shell closes both ends, then waits for both and reports

Before TODO 1: `ls / | wc -l` runs `ls` with the arguments `"/", "|", "wc", "-l"`: `ls` complains that `"|"` and `"wc"` do not exist. The `|` is the shell's job, not the kernel's.

● ● ● the finished shell (real session)

```
mysh> ls / | wc -l
26
mysh> cat /etc/passwd | grep root
root:x:0:0:root:/root:/bin/bash
mysh> yes | head -3
y
y
y
[908 killed by signal 13]
mysh> nosuch | wc -l
mysh: nosuch: No such file or directory
[910 exited with status 127]
0
mysh> echo hello | tr a-z A-Z
HELLO
```

Checkpoint: test yourself

- Q1** pipe(fd): which index do you read from? → fd[0] (read); fd[1] is the write end
- Q2** A reader waits for ever on read(). Most likely cause? → some process still holds a write end (often the parent or the reader itself)
- Q3** What happens when you write into a pipe nobody can read? → SIGPIPE kills the writer (status 141 in the shell), or EPIPE if ignored
- Q4** How does ls send its output into the pipe without knowing it? → the shell did dup2(pipe write end, 1) before exec
- Q5** Two processes do counter++ on shared memory. Why is the result too small? → load, add, store can interleave: updates are lost; use an atomic or a semaphore

Key ideas from today



Processes share no memory by default; every channel between them is made by the kernel.



A pipe is a one-way byte stream: `pipe()` gives a read end and a write end; `fork` shares them.



Full: the writer waits. EOF: all write ends closed. No reader: `SIGPIPE`.



The shell's `|` is `pipe` + `fork` + `dup2` + `exec`; programs never know.



Inside: a ring of 16 pages (64 KiB) with head, tail and two wait queues; every byte copied twice.



Shared memory copies nothing, but `counter++` from two processes loses updates without atomics or a semaphore.

Homework 10.1 (due before Lecture 10.2)

1. Long pipes

Extend your mysh to any number of commands: `ls / | sort -r | head -3`. You need $n - 1$ pipes; close every end you do not use.

2. Mixed?

`interleave.c`: two writers send blocks of A and B into one pipe. Run it with 4096, 65536 and 100000 bytes per write. Which sizes get mixed? What does POSIX promise?

3. Forgotten close

In your Lab 10.1 shell, comment out the shell's `close(fd[1])`. What happens with `ls / | wc -l`, and why? Which kernel line decides?

4. One CPU

Run `taskset -c 0 ./shmrace`. Is the plain counter still wrong? Explain why one CPU hides the bug, and why that is not a fix.

5. Small chunks

Run `pipebw`. Using Lecture 9's cost of a system call, estimate the 64-byte speed before you run it, then compare.

Next lecture and reading

NEXT: LECTURE 10.2

Signals

The kernel's doorbell: Ctrl-C, SIGCHLD, SIGPIPE and friends. How a signal is sent, how a handler runs on your own stack, and why read() sometimes returns EINTR.

Main text

M. Kerrisk, **TLPI**, Ch. 44 (pipes, FIFOs), Ch. 49 and 54 (shared memory) today; Ch. 20–22 (signals) for 10.2.

Kernel side

D. P. Bovet & M. Cesati, **Understanding the Linux Kernel**, 3rd ed., Ch. 19 (pipes, FIFOs) and Ch. 11 (signals).

C side

R. E. Bryant & D. R. O'Hallaron, **CS:APP**, 3rd ed., §12.5 (semaphores) and §8.5 (signals).

Lab files

lab10_files.zip: mysh_pipe_starter.c, the demo programs, and mytimeout_starter.c for Lab 10.2.