Multi-Process Communication & Coordination

Q: What do you mean by "communication" and "coordination"?

A: A concurrent multi-process program must include *communication* because separate processes have distinct address spaces

Coordination, a trait common to both multi-process and single-process-multi-threaded programs, is the mechanism of deciding which process executes when

I.e., "you go, OK now you go, ..."

UNIX Inter-Process Communication and/or Coordination Mechanisms

1. Pipe
2. FIFO, or *named pipe*
3. Message queue
4. Socket
5. Waiting on multiple descriptors
6. Signal
7. Shared memory & semaphores

1. Pipe, I

`int pipe(fildes[2])`

`fildes[1]` is writable

`fildes[0]` is readable

They are connected – data written into `fildes[1]` can be read from `fildes[0]`

1. Pipe, II

Useful only among processes related as ancestor & descendant

1. Call `pipe` – `fildes[1]` connected to `fildes[0]`

2. `fork` – `fildes[1]` connected to `fildes[0]` INSIDE each process AND BETWEEN both processes

3. In either order:

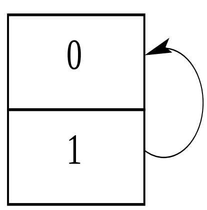
   3a. Child closes `fildes[0]`
   3b. Parent closes `fildes[1]`

Now: child can write to parent

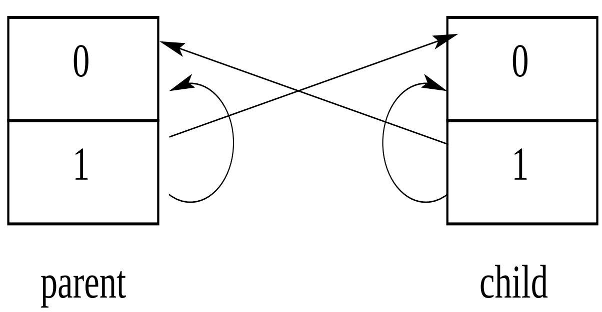
Pipe Example, I

Some funny intermediate states exist

Initally:

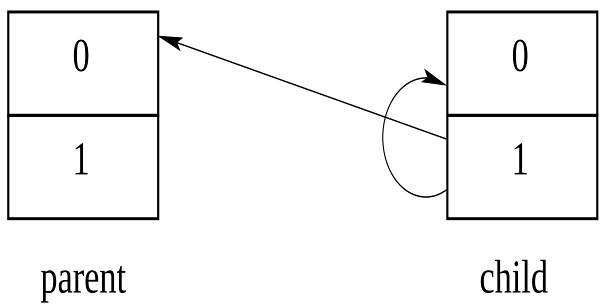


After fork:

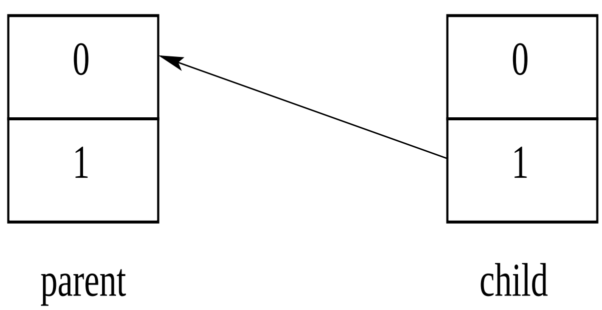


Pipe Example, II

Suppose parent closes first:



Then child closes:



2. Named Pipe

FIFO, aka “named pipe”

```
int mkfifo(char *name, mode_t mode)
```

Unidirectional data flow like with a pipe: one process writes, another reads

But: has a name in file system name space—two *unrelated* processes can use FIFO, unlike pipe

Another unique feature: writes of up to `PIPE_BUF` bytes are *atomic*

3. Message Queue

OS maintains queue of discrete messages

Send:

```
int msgsnd(int msqid, void *ptr, size_t nbytes, int flags)
```

OS considers first `long` of message to be a type field

Receive:

```
int msgrcv(int msqid, void *ptr, size_t nbytes, long type,
                                                  int flags)
```

May receive first msg on queue,
first of its type, or
first of lowest type $\leq$ argument type

Message Queue Limits

1. System-imposed max msg size, `MSGMAX`

2. System-imposed max queue size, `MSGMNB`

3. System-imposed system-wide max number of queues, `MSGMNI`

4. System-imposed system-wide max number of messages, `MSGTQL`

And: message queues are NOT reference counted!

Aside: Reference Counting

Concept of **reference counting**:

- Access to object mediated by trusted software (e.g., access message queues or files via OS)
- Process declares its intention to start/stop using object (e.g., UNIX file system `open` and `close` calls)
- With each start-access, object's reference count +1
  With each stop-access, object's reference count -1
- When object's ref count = 0 trusted software removes it

Value of the concept: trusted software performs automatic garbage collection

Advantage/Disadvantages of Queues

Unrelated processes may use message queues

However, process must invent their own means to share name of queue

Consequences of no reference counting:

- If one process removes queue, it is instantly gone—other processes that later reference queue get errors
- If NO process removes queue, it remains forever (and there are a limited number of queues system-wide)

  The real cost: processes must coordinate to know which is last to use the queue

4. Socket

`int socket(int domain, int type, int protocol)`

Bidirectional data flow

Meant for network communication

"Domain" argument selects network stack; e.g., TCP/IP

"Type" selects type of service; e.g., reliable

"Protocol" selects specific protocol for that type of service

Descriptors

Open pipe, FIFO, socket, file, device — each represented by **descriptor** (aka *file descriptor*)

UNIX philosophy: as much as possible, represent every data source/sink with descriptor

(Messages, not part of original UNIX, violate this philosophy)

5. Waiting on Multiple Descriptors

Common to wait on (typically: read from) multiple descriptors at once

E.g., server has sockets open with N clients, waits for first client input

Problem #1: `read(2)` takes only 1 descriptor argument

Problem #2: `read(2)` blocks when there is no input

Non-blocking I/O

Possible to make *descriptor* non-blocking

I.e., `read(2)` will NOT block if there is no input

(Nor will ANY other normally-blocking system call)

Q: How to make descriptor non-blocking?

A: There are 2 ways: `open(2)` and `fcntl(2)`

Non-blocking Open

Provide O_NONBLOCK flag to open(2):

```
int open(char *name, (O_NONBLOCK | ...), mode)
```

Non-blocking Fcntl

`fcntl(2)` is system call that allows manipulation of properties of descriptor *after it is open*

Provide `O_NONBLOCK` flag to `fcntl(2)`:

```
int fcntl(int fd, F_SETFL, (O_NONBLOCK | ...))
```

`F_GETFL` “command” argument GETS flags,
`F_SETFL` “command” argument SETS flags

Can change state of descriptor during program using `fcntl`

Select

The other way to do non-blocking I/O ...

`select(2)` takes a SET of fds as argument

Actually, 3 sets: "read set," "write set," and "exception set"

`select(2)` also takes timeout argument—can wait for specified time or forever

Select Example

```
int rc, max_fd;
struct timeval timeout;
struct fd_set call_set;

timeout.tv_sec  = 60;
timeout.tv_usec = 0;

FD_ZERO(&call_set);
FD_SET(fd, &call_set);
max_fd = fd;    /* only 1 fd in this example */

num = select(max_fd + 1, &call_set, NULL, NULL, &timeout);

/* this code handles any number of "ready" fds */
for (i=0; i <= max_fd; i++) {
    if (FD_ISSET(i, &call_set)) {
        ...
    }
    if (--num <= 0)
        break;
}
```