CSCB09 Software Tools and Systems Programming

Inter Processes Communication: Pipes

Marcelo Ponce

Winter 2023

Department of Computer and Mathematical Sciences - UTSC

Today's class

Today we will discuss the following topics:

- Quick recap about processes
- How processes communicate...
- Pipes

Acknowledgement *Prof. Bianca Schroeder*

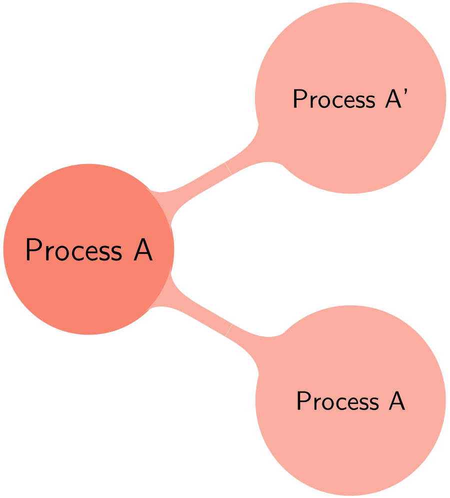
Processes Recap

Processes can create other processes

- The `fork` system call creates a *child process*:

```c
#include <unistd.h>

pid_t fork(void);
```

- `fork` creates a **duplicate** of the currently running program.
- Both processes run concurrently and independently.
- After `fork()` both execute the next instruction after `fork`.



waitpid

- What if a process wants to wait for a particular child (rather than any child)
- What if a process does not want to block when no child has terminated?

```
pid_t waitpid(pid_t pid, int *status, int options);
```

- First parameter specifies PID of child to wait for
- If options is 0, `waitpid` **blocks** (just like `wait`)
- If options is `WNOHANG`, it immediately returns 0 instead of blocking when no terminated child

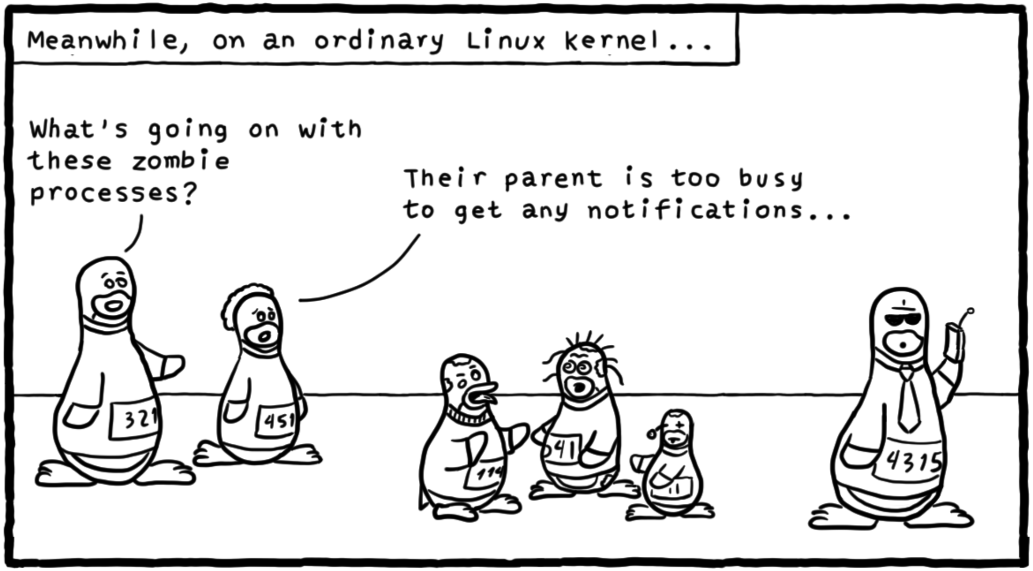
Processes Pathologies

How does a child become a zombie?

- When a child terminates, but its parent process is not waiting for it
- The child (its exit code) is kept around as a zombie until parent collects its exit code through wait – or until parent terminates
- Shows up as "`Z`" in `ps`

How does a child become an orphan?

- If the parent process terminates before the child
- Who is now the new parent?
  - Orphans get adopted by the `init` process
  - `init` is the first process started during booting
  - It's the root of the process hierarchy
  - `init` has a PID of 1
  - The PPID of orphans is 1


Meanwhile, on an ordinary Linux kernel...
What's going on with these zombie processes?
Their parent is too busy to get any notifications...
321
451
114
541
11
4315

Daniel Stori {turnoff.us}

`fork()` **VS** `exec()`

Fork creates a *duplicate* of the current process

How do we actually create a new process that runs a different program?

`fork()` **vs** `exec()`

Fork creates a *duplicate* of the current process

How do we actually create a new process that runs a different program?

- The `fork` system call creates a new process.
- The new process created by `fork()` is a copy of the current process except for the returned value.
- The `exec()` system call replaces the current process with a new program.

exec Functions

- exec() is not one specific function, but a family of functions:

```
// execl: more convenient if there is a fixed number of arguments
execl(char *path, char *arg0, ..., (char *)NULL);
// execv: The last element of this array must be a null pointer
execv(char *path, char *argv[]);

// lp and vp versions use the same functionality that the shell
// would use in locating the program
execlp(char *file, char *arg0, ..., (char *)NULL);
execvp(char *file, char *argv[]);
```

- First parameter: name of executable; then commandline parameters for executable; these are passed as argv[0], argv[1], ..., to the main program of the executable.
- execl and execv differ from each other only in how the arguments for the new program are passed
- execlp and execvp differ from execl and execv only in that you don't have to specify full path to new program

Processes Communication

Processes often need to communicate

- E.g. the different worker processes of a parallel program need to exchange data and/or synchronize
- Options:
    - Exit code – limited solution...
    - Cannot use variables, since after fork each process has separate copy of variables
    - Use files – coordination is difficult

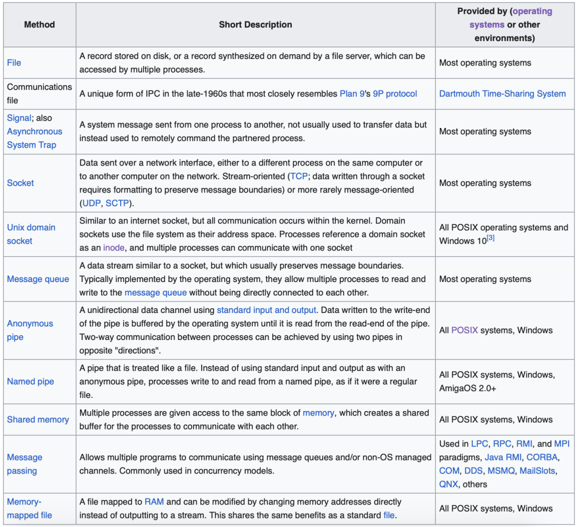
Processes often need to communicate

- E.g. the different worker processes of a parallel program need to exchange data and/or synchronize
- Options:
  - Exit code – limited solution...
  - Cannot use variables, since after fork each process has separate copy of variables
  - Use files – coordination is difficult

Other approaches:

- require different programming *paradigms*
- Shared memory programming, eg. OpenMP
- Message Passing, e.g. MPI
- Semaphores
- ...

Inter-Process Communication



Method	Short Description	Provided by (operating systems or other environments)
File	A record stored on disk, or a record synthesized on demand by a file server, which can be accessed by multiple processes.	Most operating systems
Communications file	A unique form of IPC in the late-1960s that most closely resembles Plan 9's 9P protocol	Dartmouth Time-Sharing System
Signal; also Asynchronous System Trap	A system message sent from one process to another, not usually used to transfer data but instead used to remotely command the partnered process.	Most operating systems
Socket	Data sent over a network interface, either to a different process on the same computer or to another computer on the network. Stream-oriented (TCP; data written through a socket requires formatting to preserve message boundaries) or more rarely message-oriented (UDP, SCTP).	Most operating systems
Unix domain socket	Similar to an internet socket, but all communication occurs within the kernel. Domain sockets use the file system as their address space. Processes reference a domain socket as an inode, and multiple processes can communicate with one socket	All POSIX operating systems and Windows 10[3]
Message queue	A data stream similar to a socket, but which usually preserves message boundaries. Typically implemented by the operating system, they allow multiple processes to read and write to the message queue without being directly connected to each other.	Most operating systems
Anonymous pipe	A unidirectional data channel using standard input and output. Data written to the write-end of the pipe is buffered by the operating system until it is read from the read-end of the pipe. Two-way communication between processes can be achieved by using two pipes in opposite "directions".	All POSIX systems, Windows
Named pipe	A pipe that is treated like a file. Instead of using standard input and output as with an anonymous pipe, processes write to and read from a named pipe, as if it were a regular file.	All POSIX systems, Windows, AmigaOS 2.0+
Shared memory	Multiple processes are given access to the same block of memory, which creates a shared buffer for the processes to communicate with each other.	All POSIX systems, Windows
Message passing	Allows multiple programs to communicate using message queues and/or non-OS managed channels. Commonly used in concurrency models.	Used in LPC, RPC, RMI, and MPI paradigms, Java RMI, CORBA, COM, DDS, MSMQ, MailSlots, QNX, others
Memory-mapped file	A file mapped to RAM and can be modified by changing memory addresses directly instead of outputting to a stream. This shares the same benefits as a standard file.	All POSIX systems, Windows

SRC: `https://en.wikipedia.org/wiki/Inter-process_communication`

Pipes

Pipes

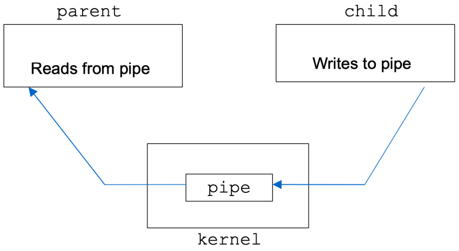
A pipe is a mechanism for interprocess communication
data written to the pipe by one process can be read by another process.

The data is handled in a first-in, first-out (FIFO) order.

Neither pipes nor FIFO special files allow file positioning. Both reading and writing operations happen sequentially; reading from the beginning of the file and writing at the end.

Pipes

- Pipes are a one-way (half-duplex) communication channel
- Pipes are buffers managed by the OS
- Processes use low-level file descriptors for pipe operations



Recall: I/O mechanisms in C

- So far, we've used "File Pointers" (regular files):
  - You use a pointer to a file structure (`FILE *`) as handle to a file.
  - The file struct contains a file descriptor and a buffer.
  - Use for regular files

```c
FILE *fp = fopen("my_file.txt", "w");
fprintf(fp, "Hello!\n");
fclose(fp);
```

- Now we need: File descriptors (low-level):
  - Each open file is identified by a small integer
  - Operations: `open`, `close`, `read`, `write`

File I/O with file descriptors

- `int open(const char *pathname, int flags);`
    - Returns a file descriptor
    - Flags: `O_RDONLY, O_WRONLY, or O_RDWR` to open the file read-only, write-only, or read/write (and a few more, see man pages)
- `ssize_t read(int fd, void *buf, size_t count);`
    - Returns # bytes read, 0 for EOF, -1 for error
- `ssize_t write(int fd, const void* buf, size_t count);`
    - Returns # bytes actually written, -1 for error
- `int close(int fd);`
    - Returns 0 on success, -1 on error

File I/O with file descriptors

- `int open(const char *pathname, int flags);`
    - Returns a file descriptor
    - Flags: `O_RDONLY, O_WRONLY, or O_RDWR` to open the file read-only, write-only, or read/write (and a few more, see man pages)
- `ssize_t read(int fd, void *buf, size_t count);`
    - Returns # bytes read, 0 for EOF, -1 for error
- `ssize_t write(int fd, const void* buf, size_t count);`
    - Returns # bytes actually written, -1 for error
- `int close(int fd);`
    - Returns 0 on success, -1 on error
- Example (error checking ommitted...):

```
char buf[6];
int fd = open ("filename.txt", O_RDONLY);
read (fd, buf, 6);
close(fd);
```

Handling Pipes with File Descriptors

- How to open/create a pipe?
  ```
  int pipe(int pipefd[2]);
  ```
- pass a pointer to two integers (i.e. an array or a `malloc` of two ints) and pipe fills it with two newly opened FDs.
- Returns 0 on success, -1 on error

Handling Pipes with File Descriptors

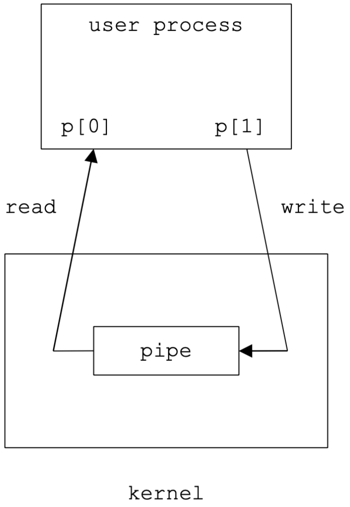
- How to open/create a pipe?

  `int pipe(int pipefd[2]);`

- pass a pointer to two integers (i.e. an array or a `malloc` of two ints) and pipe fills it with two newly opened FDs.
- Returns 0 on success, -1 on error
- Example:

```c
int p[2];
if ( pipe(p) == -1 ) {
  perror("pipe");
  exit(1);
}
```

  - `p[0]` is now open for reading
  - `p[1]` is now open for writing



Example: Process talking to itself

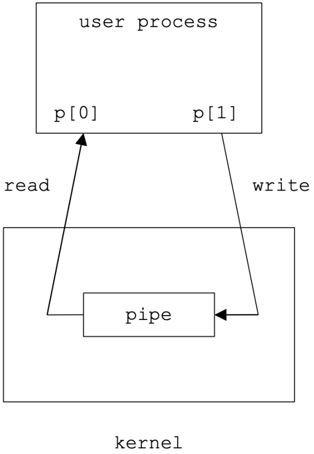
```c
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

#define MSG_SIZE 13

int main(void) {
    char *msg = "hello, world\n";
    char buf[MSG_SIZE];
    int p[2];

    if (pipe(p) == -1) {
  perror("pipe");
  exit(1);
    }

    write(p[1], msg, MSG_SIZE);
    read(p[0], buf, MSG_SIZE);

    write (STDOUT_FILENO, buf, MSG_SIZE);

    return 0;
}
```

```c
ssize_t read(int fd, void *buf, size_t
    count);

ssize_t write(int fd, const void *buf,
    size_t count);
```

First, write `msg` to the pipe

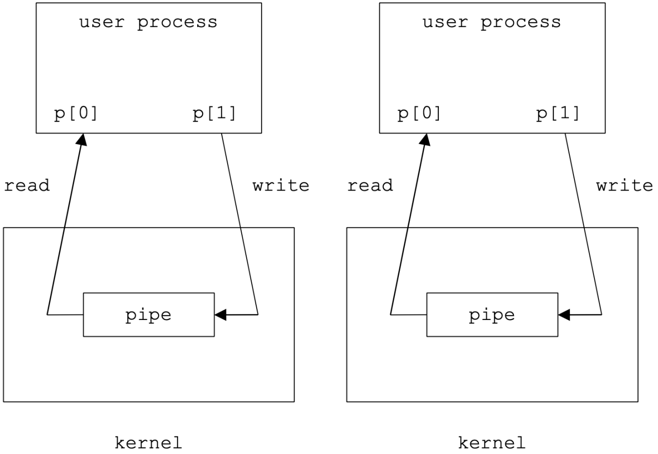
Then, read from the pipe

A process talking to itself is not very useful
How do we get two processes talking to each other using pipes?

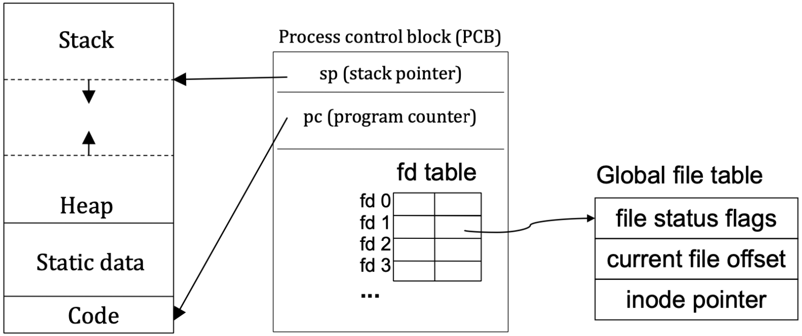


A process talking to itself is not very useful

How do we get two processes talking to each other using pipes?



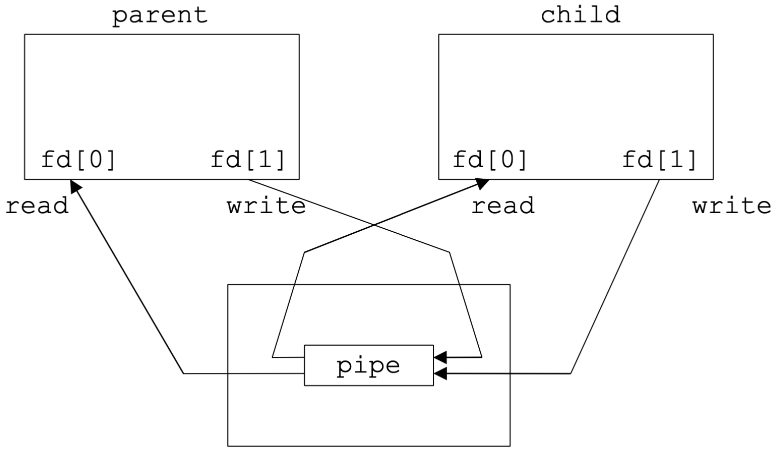
The OS manages file descriptors



- Per-process FD table has all open files of a process
- Global file table has all files open system-wide
- On `fork`, a child gets a copy of parent's FD table, so it will have same files open
- The FD table is preserved on `exec`

On fork, child gets copy of parent's file descriptors, including pipes

Qn: can we make the child send a message to the parent?



Example 1: Child talking to parent

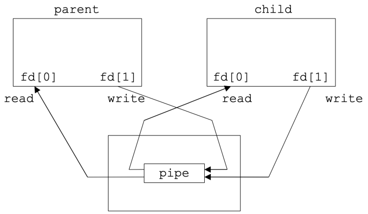
```c
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

#define MSG_SIZE 13

int main(void) {
    char *msg = "hello,␣world\n";
    char buf[MSG_SIZE];
    int p[2];

    if (pipe(p) == -1) {
  perror("pipe");
  exit(1);
    }

    if (fork() == 0) {
  // child writes
  write(p[1], msg, MSG_SIZE);
    } else {
  // parent read from pipe and prints
  read(p[0], buf, MSG_SIZE);
  write(STDOUT_FILENO, buf, MSG_SIZE);
    }

    return 0;
}
```

Pipes and EOF

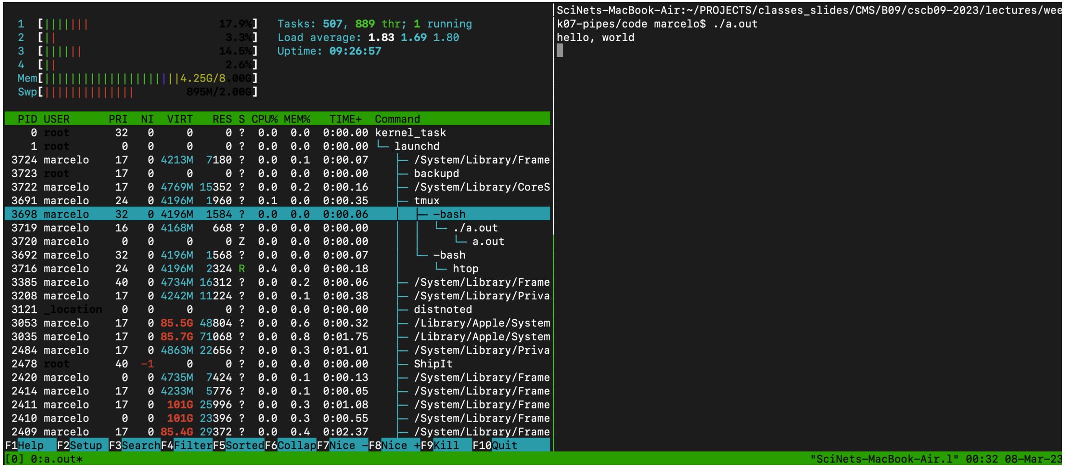
- What if the parent does not know beforehand when and how much data a child will write?
- If no writing end is open, `read` detects EOF and returns 0.
- `read` blocks until data is available in the pipe.
- All open pipes (and other FDs) are closed when a process exits.

Example 2: Child talking to parent

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

#define BUF_SIZE 4  // small
#define MSG_SIZE 13

int main(void) {
    char *msg = "hello,␣world\n";
    char buf[BUF_SIZE];
    int p[2], nbytes;

    if (pipe(p) == -1) {
  perror("pipe");
  exit(1);
    }

    if (fork() == 0) {
  // child writes
  write(p[1], msg, MSG_SIZE);
    } else {
  // parent read from pipe and prints
  while ( (nbytes=read(p[0], buf, MSG_SIZE)) > 0 )
    write(STDOUT_FILENO, buf, MSG_SIZE);
    }

    return 0;
}
```



Parent hangs, even after child exits. Why is EOF not detected?



```
  1  [|||||||                17.9%]   Tasks: 507, 889 thr; 1 running
  2  [||                      3.3%]   Load average: 1.83 1.69 1.80
  3  [||||||                 14.5%]   Uptime: 09:26:57
  4  [||                      2.6%]
  Mem[||||||||||||||||||||||4.25G/8.00G]
  Swp[|||||||||||||         895M/2.00G]

  PID USER      PRI  NI  VIRT   RES S CPU% MEM%   TIME+  Command
    0 root       32   0     0     0 ?  0.0  0.0  0:00.00 kernel_task
    1 root        0   0     0     0 ?  0.0  0.0  0:00.00 └─ launchd
 3724 marcelo    17   0 4213M  7180 ?  0.0  0.1  0:00.07    ├─ /System/Library/Frame
 3723 root       17   0     0     0 ?  0.0  0.0  0:00.00    ├─ backupd
 3722 marcelo    17   0 4769M 15352 ?  0.0  0.2  0:00.16    ├─ /System/Library/CoreS
 3691 marcelo    24   0 4196M  1960 ?  0.1  0.0  0:00.35    ├─ tmux
 3698 marcelo    32   0 4196M  1584 ?  0.0  0.0  0:00.06    │  ├─ -bash
 3719 marcelo    16   0 4168M   668 ?  0.0  0.0  0:00.00    │  │  └─ ./a.out
 3720 marcelo     0   0     0     0 Z  0.0  0.0  0:00.00    │  │     └─ a.out
 3692 marcelo    32   0 4196M  1568 ?  0.0  0.0  0:00.07    │  └─ -bash
 3716 marcelo    24   0 4196M  2324 R  0.4  0.0  0:00.18    │     └─ htop
 3385 marcelo    40   0 4734M 16312 ?  0.0  0.2  0:00.06    ├─ /System/Library/Frame
 3208 marcelo    17   0 4242M 11224 ?  0.0  0.1  0:00.38    ├─ /System/Library/Priva
 3121 _location   0   0     0     0 ?  0.0  0.0  0:00.00    ├─ distnoted
 3053 marcelo    17   0 85.5G 48804 ?  0.0  0.6  0:00.32    ├─ /Library/Apple/System
 3035 marcelo    17   0 85.7G 71068 ?  0.0  0.8  0:01.75    ├─ /Library/Apple/System
 2484 marcelo    17   0 4863M 22656 ?  0.0  0.3  0:01.01    ├─ /System/Library/Priva
 2478 root       40  -1     0     0 ?  0.0  0.0  0:00.00    ├─ ShipIt
 2420 marcelo     0   0 4735M  7424 ?  0.0  0.1  0:00.13    ├─ /System/Library/Frame
 2414 marcelo    17   0 4233M  5776 ?  0.0  0.1  0:00.05    ├─ /System/Library/Frame
 2411 marcelo    17   0  101G 25996 ?  0.0  0.3  0:01.08    ├─ /System/Library/Frame
 2410 marcelo     0   0  101G 23396 ?  0.0  0.3  0:00.55    ├─ /System/Library/Frame
 2409 marcelo    17   0 85.4G 29372 ?  0.0  0.4  0:02.37    ├─ /System/Library/Frame
F1Help  F2Setup F3SearchF4FilterF5SortedF6CollapF7Nice -F8Nice +F9Kill  F10Quit
```

```
SciNets-MacBook-Air:~/PROJECTS/classes_slides/CMS/B09/cscb09-2023/lectures/wee
k07-pipes/code marcelo$ ./a.out
hello, world
```

```
[0] 0:a.out*                                    "SciNets-MacBook-Air.l" 00:32 08-Mar-23
```

Example 3: Child talking to parent

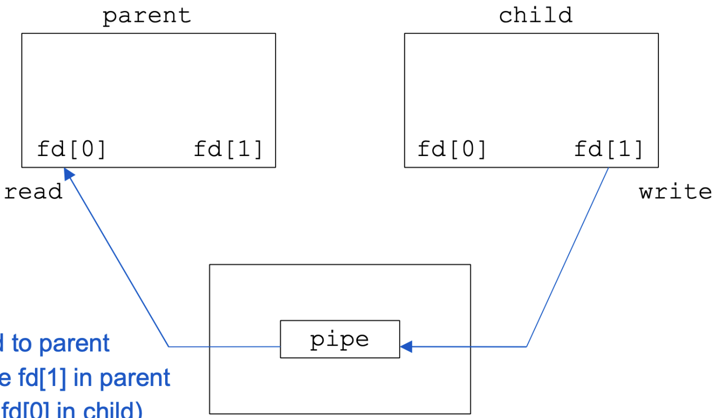
```c
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

#define BUF_SIZE 4  // small
#define MSG_SIZE 13

int main(void) {
    char *msg = "hello, world\n";
    char buf[BUF_SIZE];
    int p[2], nbytes;

    if (pipe(p) == -1) {
  perror("pipe");
  exit(1);
    }
```

```c

    if (fork() == 0) {
  // child writes
  write(p[1], msg, MSG_SIZE);
    } else {
  // parent needs to close its writing
      end
  close(p[1]);

  // parent read from pipe and prints
  while ( (nbytes=read(p[0], buf,
      MSG_SIZE)) > 0 )
    write(STDOUT_FILENO, buf, MSG_SIZE)
        ;
    }

    return 0;
}
```

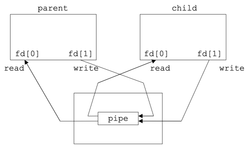
Closing ends of pipes

- In general, each process will read or write a pipe (not both)
- Close the end you are not using
  - Before reading: close `p[1]`
  - Before writing: close `p[0]`

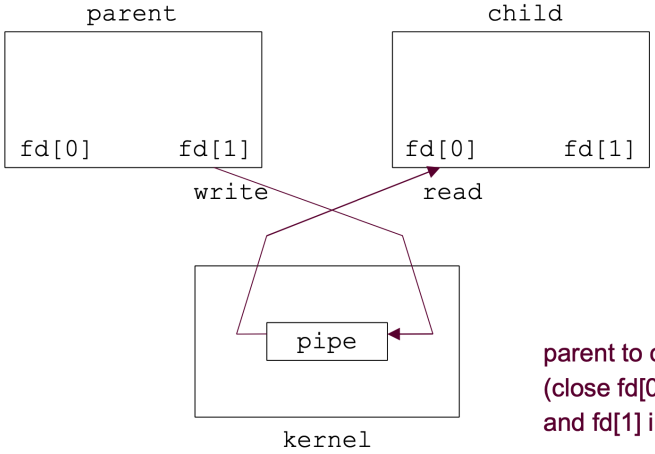
Direction of data flow?



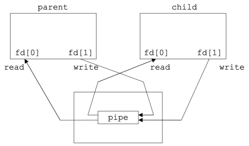
child to parent
close fd[1] in parent
and fd[0] in child)



Direction of data flow?



parent to child
(close fd[0] in parent
and fd[1] in child)



Pipes Communication-Types i

/* Self-comm. process */

Step 1 - Create a pipe.

Step 2 - Send a message to the pipe.

Step 3 - Retrieve the message from the pipe and write it to the standard output.

Step 4 - Send another message to the pipe.

Step 5 - Retrieve the message from the pipe and write it to the standard output.

Note - Retrieving messages can also be done after sending all messages.

/* Parent-Child comm. */

Step 1 - Create a pipe.

Step 2 - Create a child process.

Step 3 - Parent process writes to the pipe.

Step 4 - Child process retrieves the message from the pipe and writes it to the standard output.

Step 5 - Repeat step 3 and step 4 once again.

/* Two-way communication w/Pipes */

Step 1 - Create two pipes. First one is for the parent to write and child to read, say as pipe1. Second one is for the child to write and parent to read, say as pipe2.

Step 2 - Create a child process.

Step 3 - Close unwanted ends as only one end is needed for each communication.

Step 4 - Close unwanted ends in the parent process, read end of pipe1 and write end of pipe2.

Step 5 - Close the unwanted ends in the child process, write end of pipe1 and read end of pipe2.

Step 6 - Perform the communication as required.

Summary: Pipes and File Descriptors

- A forked child inherits file descriptors from its parent
- `pipe()` creates an OS internal system buffer and two file descriptors, one for reading and one for writing.
- After the pipe call, the parent and child should close the file descriptors for the opposite direction.

Summary: Pipes and File Descriptors

- A forked child inherits file descriptors from its parent
- `pipe()` creates an OS internal system buffer and two file descriptors, one for reading and one for writing.
- After the pipe call, the parent and child should close the file descriptors for the opposite direction.

```
ssize_t read(int fd, void *buf, size_t count);
  // Returns #bytes read, 0 for EOF, -1 for error

ssize_t write(int fd, const void *buf, size_t count);
  // Returns #bytes actually written, -1 for error

int close(int fd);
  // Returns 0 on success, -1 on error

int pipe(int pipefd[2]);
  // Returns 0 on success, -1 on error
```

References

References

- https://www.gnu.org/software/libc/manual/html_node/Pipes-and-FIFOs.html