# Systems Software/Programming

## Inter-Process Communication

## Inter-Process Communication

- Inter-Process Communication(IPC) is the generic term describing how two processes may exchange information with each other.
- In general, the two processes may be running on the same machine or on different machines
- This communication may be an exchange of data for which two or more processes are cooperatively processing the data or synchronization information to help two independent, but related, processes schedule work so that they do not destructively overlap.

## Inter-Process Communication Methods

- Local → Processes running on the same machine
    - Pipe
    - Signal
    - MultiProcessing (MP) in multi-core/multi-processor architecture
- Distributed → Processes running on different machines
    - Message Passing Interface (MPI)

## Inter-process Communication using Pipe

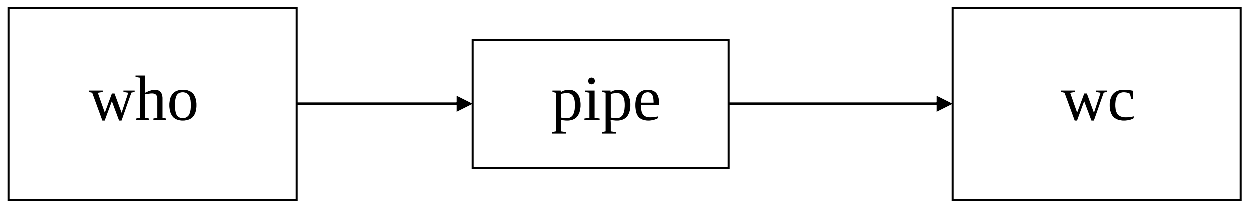
## Pipes

- Pipes are an inter-process communication mechanism that allow two or more processes to send information to each other.
  - commonly used from within shells to connect the standard output of one utility to the standard input of another.
  - For example, here’s a simple shell command that determines how many users there are on the system:

    ```
    $ who | wc -l
    ```

  - The who utility generates one line of output per user.  This output is then “piped” into the wc utility, which, when invoked with the “-l” option, outputs the total number of lines in its input.

## Pipes



*Bytes from “who” flow through the pipe to “wc”*

**A simple pipe**

# PIPEs are more powerful constructs

- Child process exit code send to parent process → limit of only 1 int
- What if child process wants to send larger data of different data types to parent process?
- Also what if parent process wants to send some information to different child processes?
  - if information is static it can be initialized in parent before fork() to share a content of a variable with the respective child.
  - But what if parent generates information along the way after fork()?

## Pipes

- It’s important to realize that both the writer process and the reader process of a pipeline execute concurrently;
  - a pipe automatically buffers the output of the writer and suspends the writer if the pipe gets too full.
  - Similarly, if a pipe empties, the reader is suspended until some more output becomes available.
- All versions of UNIX support unnamed pipes, which are the kind of pipes that shells use.
- System V also supports a more powerful kind of pipe called a named pipe.

## Unnamed Pipes: pipe() System Call

- An unnamed pipe is a unidirectional communications link that automatically buffers its input ( the maximum size of the input varies with different versions of UNIX, but is approximately 5K ) and may be created using the pipe() system call.
- Each end of a pipe has an associated file descriptor:
  - The “write” end of the pipe may be written to using write()
  - The “read” end may be read from using read()
- When a process has finished with a pipe’s file descriptor. it should close it using close()
- Note read(), write and close() are unbuffered I/O System Calls that we have studied earlier

## Unnamed Pipes: pipe() System Call

int pipe( int  fd[2] )

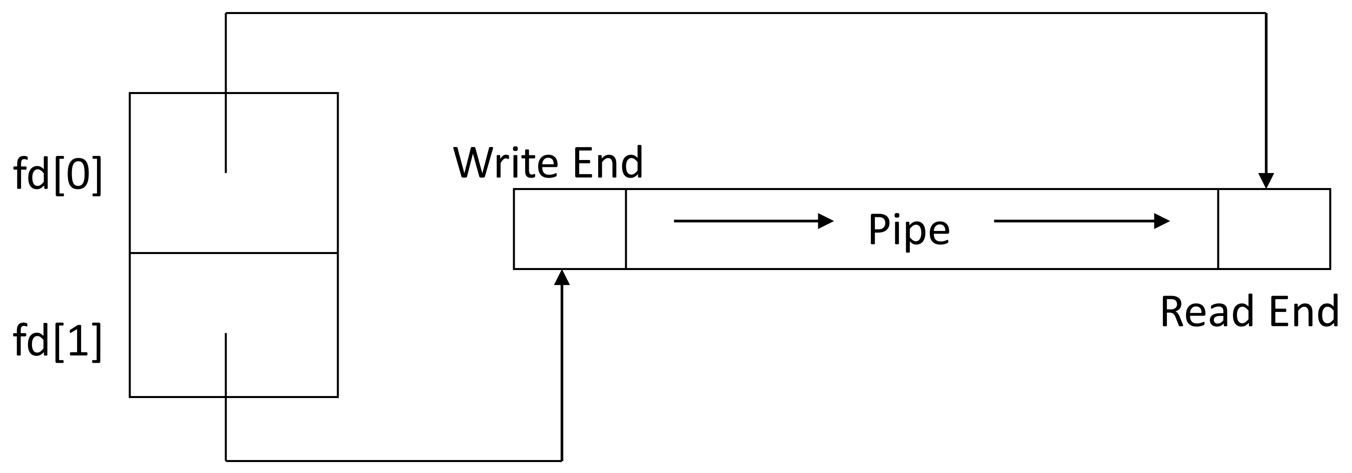
- pipe() creates an unnamed pipe and returns two file descriptors:
  - The descriptor associated with the “read” end of the pipe is stored in fd[0],
  - The descriptor associated with the “write” end of the pipe is stored in fd[1].

# PIPEs are generally Unidirectional

- Assume that the following code was executed:

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

data structure as shown below will be created



# READ and WRITE conditions for PIPE for OS Implementation

- READ Conditions:
  - If a process reads from a pipe whose “write” end has been closed, the “read()” call returns a value of zero, indicating the end of input.
  - If a process reads from an empty pipe whose “write” end is still open, it sleeps until some input becomes available.
  - If a process tries to read more bytes from a pipe that are present, all of the current contents are returned and read() returns the number of bytes actually read.

## READ and WRITE conditions for PIPE for OS Implementation

- WRITE Conditions:
  - if a process writes to a pipe whose “read” end has been closed, the write fails and the writer process is sent a SIGPIPE signal. the default action of this signal is to terminate the process.
  - If a process writes fewer bytes to a pipe than the pipe can hold, the write() is guaranteed to be atomic; that is, the writer process will complete its system call without being preempted by another process.
  - If the kernel cannot allocate enough space for a new pipe, pipe() returns a value of -1; otherwise, it returns a value of 0.

## How OS would implement READ and WRITE for PIPE?

**Variable to be used in read and write algos.**

```
READ_END=0          // pipefd read index i.e. pipefd[READ_END]
WRITE_END=1         // pipefd write index i.e. pipefd[WRITE_END]
read_offset=0       // offset for the next read from pipe
write_offset=0      // offset for the next write to pipe
pipebufsize=64*1024       // maximum size of pipebuf
String pipebuf      // pipe wrap-around buffer to store pipe data
pipefilled          // number (count) of bytes already in pipe
```

# How OS would implement READ for PIPE?

```
int READ(int pipefd[2], String appbuf, int appbufsz):
  IF pipefd[WRITE_END] IS CLOSED THEN
    return 0
  ELSE
    IF pipefilled IS 0 THEN READER PROCESS SLEEPS
    FOR DATA TO WRITTEN TO PIPE (BLOCKING READ)
    IF pipefilled <= appbufsize THEN
      COPY ALL BYTES FROM pipebuf TO appbuf
      pipefiled_ret = pipefilled
      pipefilled = 0
      read_offset =  (read_offset + appbufsize) %
    pipebufsize;
      return pipefilled_ret
    ELSE // pipefilled > appbufsize
      COPY appbufsize BYTES FROM pipebuf TO appbuf
      pipefilled_ret = appbufsize
      pipefilled = pipefilled – appbufsize
      read_offset =  (read_offset + appbufsize) %
    pipebufsize
      return pipefilled_ret
```

## How OS would implement WRITE for PIPE?

```
int WRITE(int pipefd[2], String appbuf, int appbufsz):
  IF pipefd[READ_END] IS CLOSED THEN
    SEND SIGPIPE SIGNAL TO WRITER PROCESS
  ELSE
    IF pipebufsize == pipefilled return 0
    IF (pipebufsize – pipefilled) >= appbufsize THEN
      COPY ALL BYTES FROM appbuf TO pipebuf
      pipefilled = pipefilled + appbufsize
      write_offset =  (write_offset + appbufsize) % pipebufsize
      pipefiled_ret = appbufsize
      return pipefilled_ret
    ELSE // (pipebufsize - pipefilled < appbufsize)
      COPY (pipebufsize – pipefilled) BYTES FROM appbuf TO pipebuf
      pipefilled_ret = pipebufsize – pipefilled
      pipefilled = pipebufsize
      read_offset =  (read_offset + appbufsize) % pipebufsize
      return pipefilled_ret
```

# Application Flow to Use PIPE(s)

- Unnamed pipes are usually used for communication between a parent process and its child, with one process writing and the other process  reading.
- The typical sequence of events for such a communication is as follows:
    1. The main process creates an unnamed pipe using pipe().
    2. The main process forks creating parent and child processes (sharing pipfd).
    3. The writer (parent/child) closes its “read” end of the pipe, and the designated reader (child/parent) closes its “write” end of the pipe.
    4. The processes communicate by using write() and read() system calls.
    5. Each process closes its active pipe descriptor when it is finished using them.

## Example of Unidirectional Communication

- Here’s a small program that uses a pipe to allow the child to send message to parent which is read  by parent and display it:
- ProcessManagement\talk_unidirectional.c

## Each message must have “End of Message” indication

- The child included the phrase’s NULL terminator as part of the message so that the parent could easily display it.

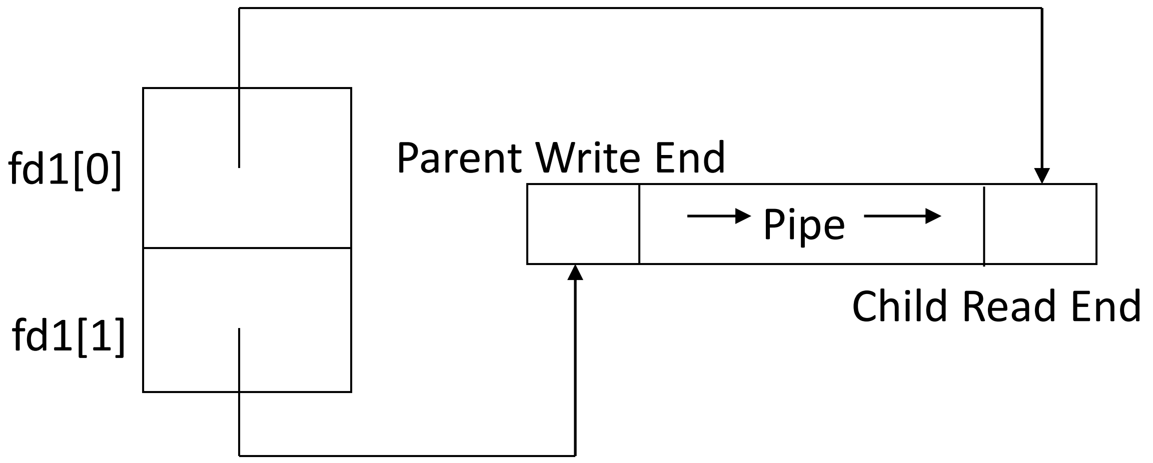
write(fd[WRITE], phrase, strlen(phrase)+1); → length+1

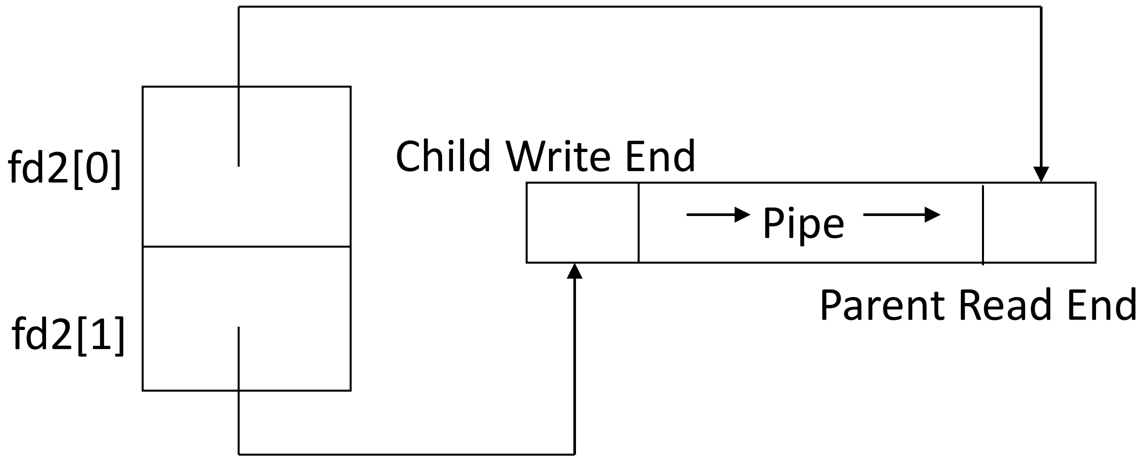
- When a writer process sends more than one variable-length message into a pipe, it must use a protocol to indicate to the reader the location for the end of the message.
- Methods for such indication include :
  - sending the length of a message(in bytes) before sending the message itself
  - ending a message with a special character such as a new line or a NULL

# Bidirectional PIPE with 2 PIPEs

- Pipe are 1-way communication between parent and child that means for bidirectional (2-way) communication you need 2 pipes

```
int fd1[2], fd2[2];
pipe(fd1); pipe(fd2);
```





## Example of Bidirectional Communication

- Please note : Bidirectional communication is only possible by using two pipes.
- ProcessManagement\talk_bidirectional.c

## Our own implementation of Shell pipe operator (|)

- UNIX shells use unnamed pipes to build pipelines, connecting the standard output of the first to the standard input of the second.
- For example we can pass two arguments to ProcessManagement\connect.c first argument generating output on STDOUT and the other taking input which will be redirected from STDOUT.

- Code uses dup2 system call: duplicate a file descriptor and we can use the file descriptors interchangeably to refer to the file

```
int dup2(int oldfd, int newfd);
```

## Unnamed Pipes: pipe() System Call

```
$ who          ---> execute "who" by itself.
ubuntu18 :0         2019-02-13 10:11 (:0)

$ who | wc      ---> pipe "who" through "wc".
      1       5      44     …1 line,  5 words, 44 chars.

$ connect  who  wc   ---> pipe "who" through "wc".
      1       5      44     …1 line,  5 words, 44 chars.
```