Operating System Design

Processes Scheduling Review
IPC: Pipes

Neda Nasiriani
Fall 2018

Processes

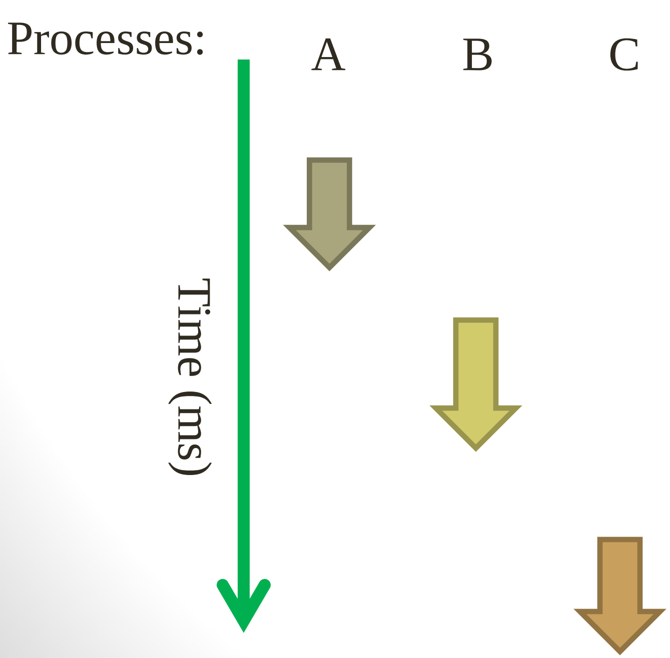
- What is a process?
  - Informally: a program in execution
- Examples of processes in a computer system
  - The Kernel and all its related processes
  - Web browser
  - Word editor
  - JVM
  - Python IDE
  - …
- How can you see the list of processes on your machine?
  - top, htop
  - ps -el

You want to design the OS to allow for multi processes running at the same time…

Assume there is one CPU!

Specs of the multi-process Computer System with one CPU

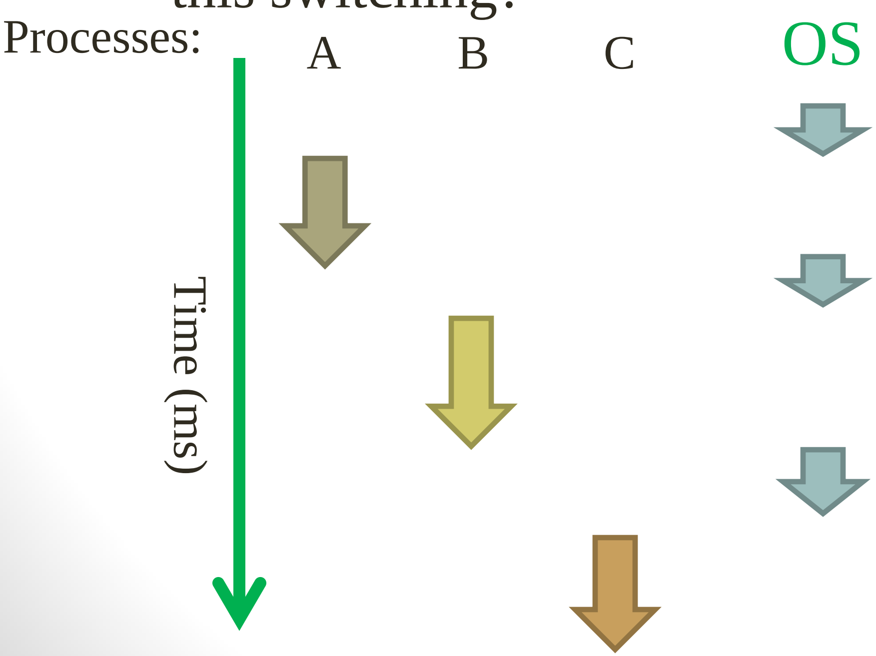
- We want processes to run concurrently, so (i) they can interact with each other, and (ii) maximize CPU utilization
  - Fact: at each time only one process can run on each processor
  - Remedy: So, we should switch processes fast enough so they feel like they are all running simultaneously (illusion)



How can this be implemented in a real computer system?

Specs of the multi-process Computer System with one CPU

- We want processes to run concurrently, so (i) they can interact with each other, and (ii) maximize CPU utilization
  - Fact: at each time only one process can run on each processor
  - Remedy: So, we should switch processes fast enough so they feel like they are all running simultaneously (illusion)
- Can the OS kernel as the main process in the system perform this switching?



OS tasks?

- deciding who should run next,
- Handle interrupts if any happened
- …

Specs of the multi-process Computer System with one CPU

- We want processes to run concurrently, so (i) they can interact with each other, and (ii) maximize CP… utiliz…
  - Fact: at ea… sor
  - Rem… el like
- Ca… t…

Processes:

Time (ms)

What does the OS need to know about the Processes to be able to do this Switching?

…s?

- …ho should run next,
- Handle interrupts if any happened
- …

Processes Components

- What are the main components of a process?
  - Text section
    - The code
  - Stack
    - Local variables
    - Function parameters
    - …
  - Heap
    - Dynamically allocated memory
  - Data Section
    - Global variables
  - What else?

Processes Components

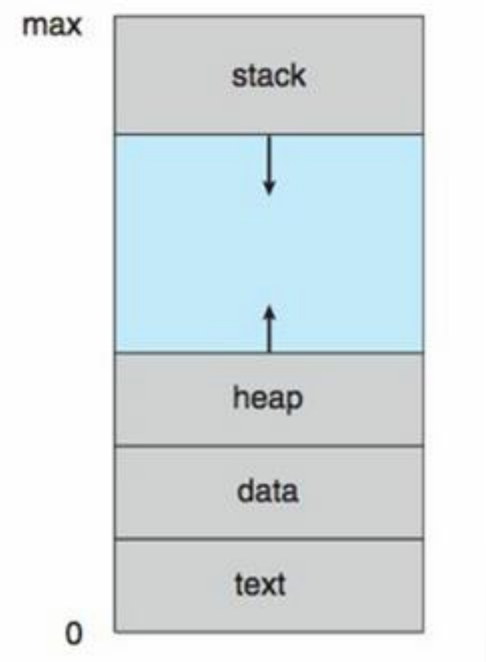
- Assume processes A is running in a system
  - The CPU decides to switch from process A to another process
  - What information will the CPU need to resume process A later?
    - Program Counter
    - Value of registers
- SO, a process is associated with the following components
  - Text section
  - Data section
  - Heap
  - Stack
  - Program Counter
  - Value of Registers

Process A

```
lw $t0, offset($s0)
lw $t1, offset($s1)
add $d, $t0, $t1
.
.
.
```

Processes Components

- A process is associated with the following components
  - Text section
  - Data section
  - Heap
  - Stack
  - Program Counter
  - Value of Registers
- The process in memory looks like this

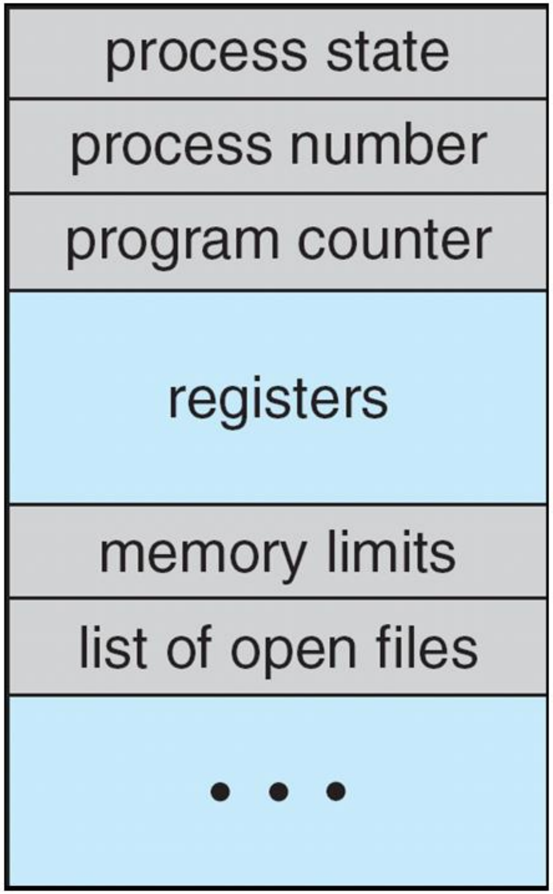


What other information is needed?

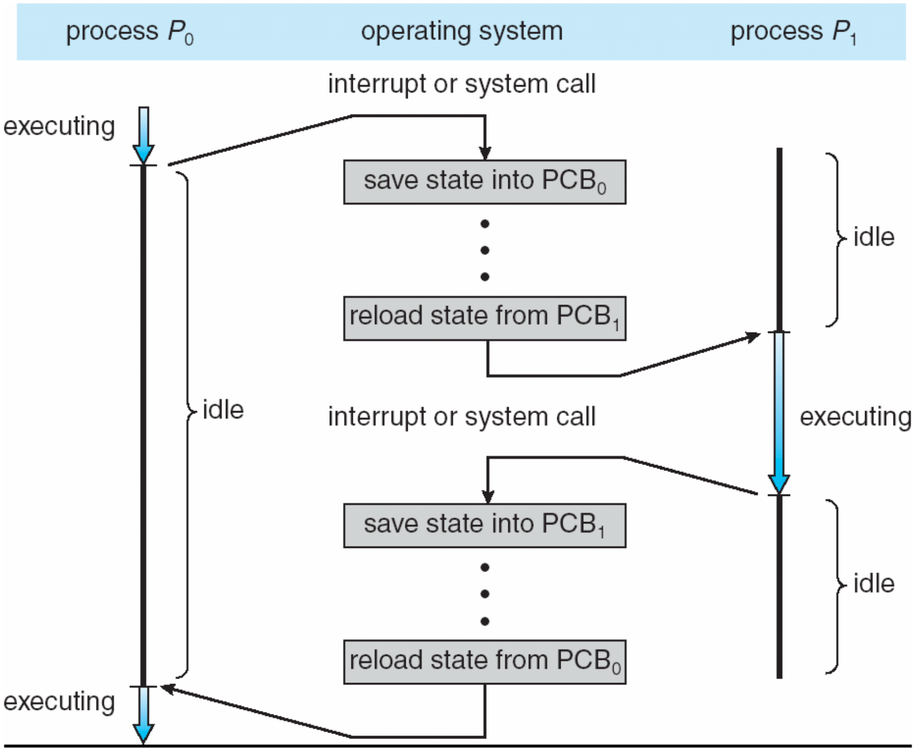
- If you want to design a scheduler to divide your time resource between a bunch of different processes, what info would you need in order to schedule effectively and fairly
  - Process state – running, waiting, etc
  - Program counter – location of instruction to next execute
  - CPU registers – contents of all process-centric registers
  - CPU scheduling information- priorities, scheduling queue pointers
  - Memory-management information – memory allocated to the process
  - Accounting information – CPU used, clock time elapsed since start, time limits
  - I/O status information – I/O devices allocated to process, list of open files

Where to keep that information?

- There is a data type called Process Control Block (PCB) that contains all this information about each process



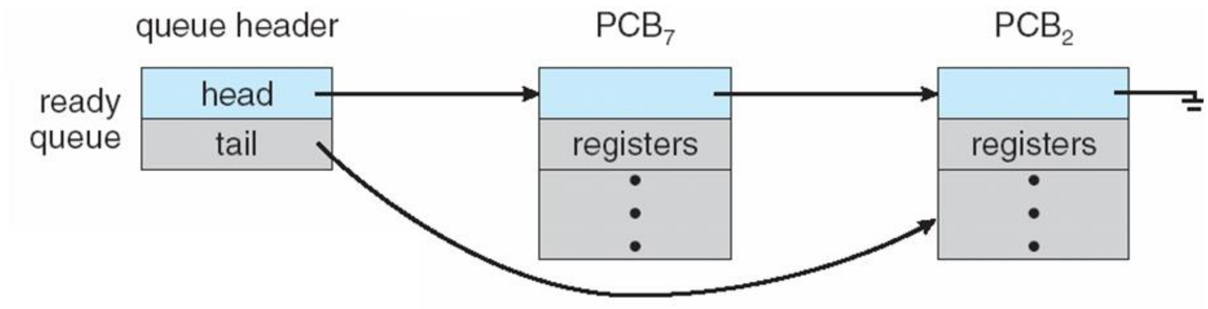
CPU Switch between Processes

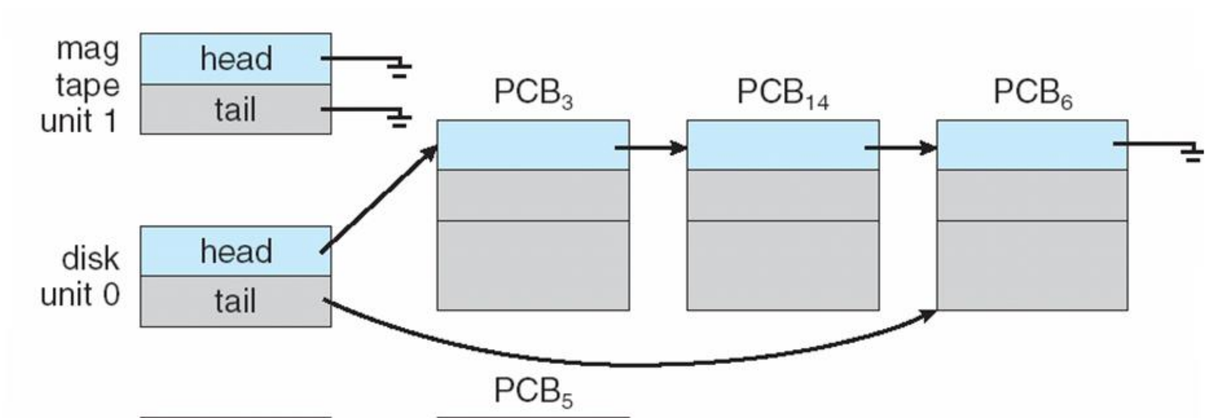


- Context Switch: When CPU switches to another process, the system must **save the state** of the old process and load the **saved state** for the new process via a **context switch**
- This time is pure overhead!

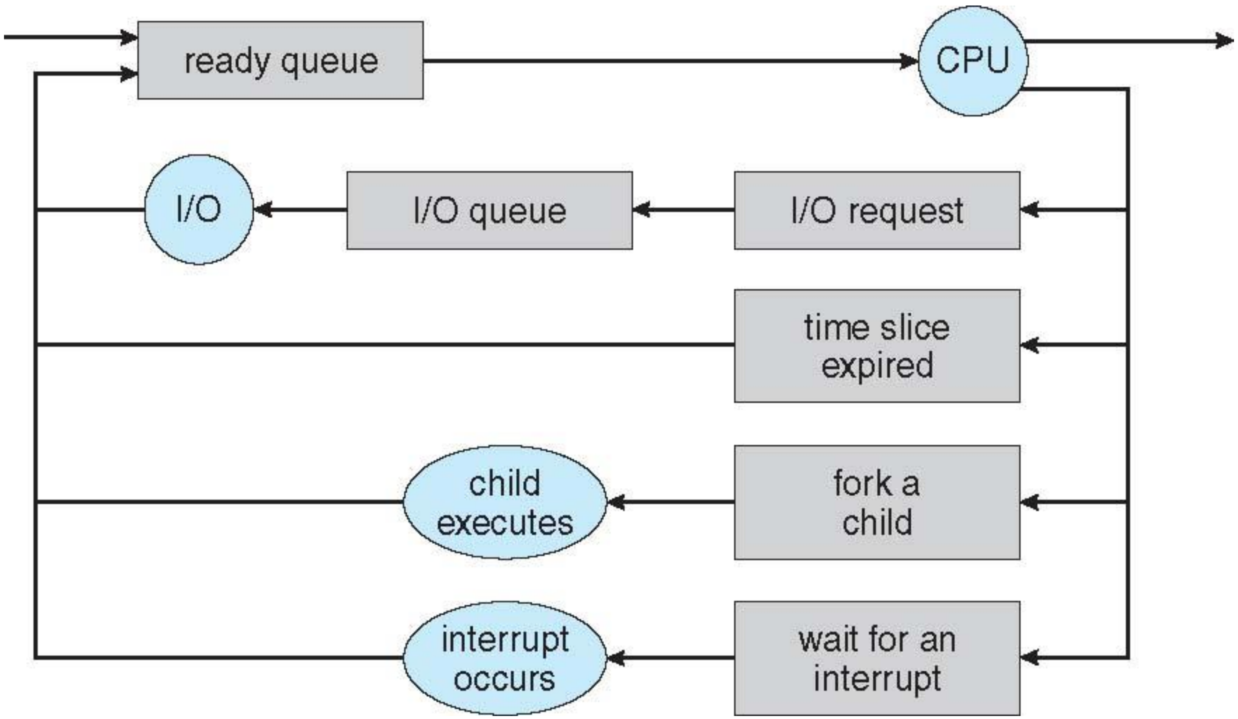
Scheduler

- A list of all processes PCBs is available to OS scheduler



- Ready queue: a list of all processes which are ready and waiting to execute
- Device queue: a list of all processes waiting for an I/O operation on a device, e.g., Disk queue, terminal queue
- 

Scheduler

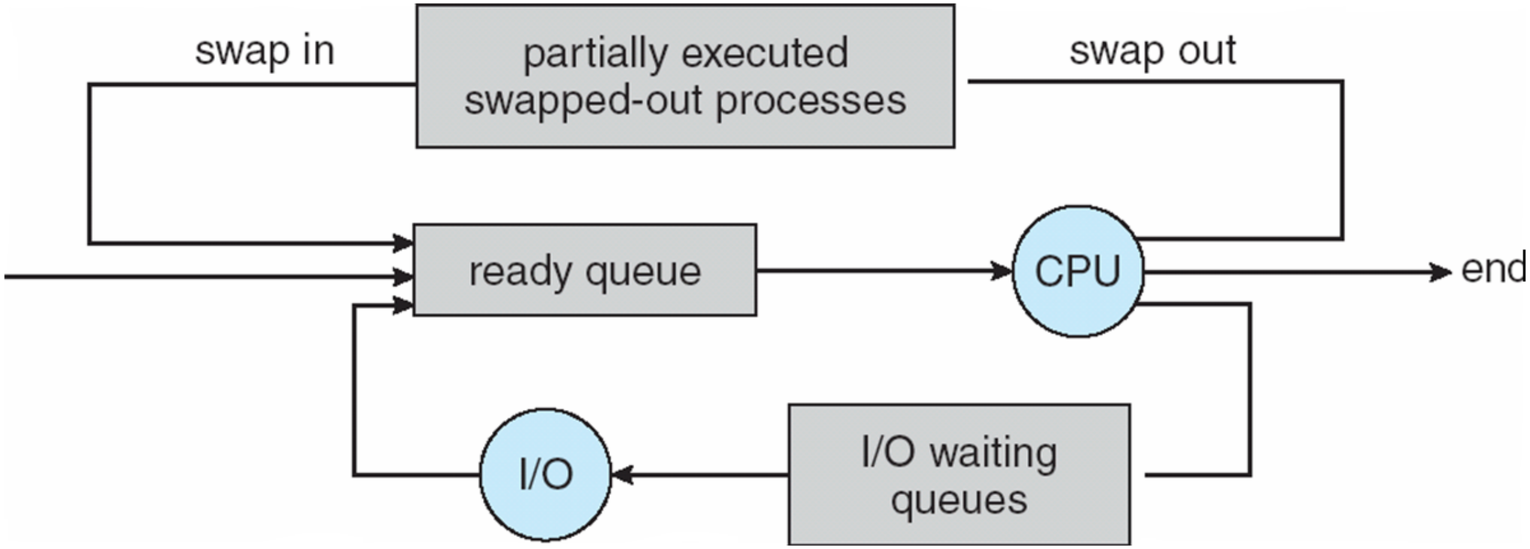

ready queue
CPU
I/O
I/O queue
I/O request
time slice expired
child executes
fork a child
interrupt occurs
wait for an interrupt

Scheduler

- **Short-term scheduler** (or **CPU scheduler**) – selects which process should be executed next and allocates CPU
  - Sometimes the only scheduler in a system
  - Short-term scheduler is invoked frequently (milliseconds) $\Rightarrow$ (must be fast)
- **Long-term scheduler** (or **job scheduler**) – selects which processes should be brought into the ready queue
  - Long-term scheduler is invoked infrequently (seconds, minutes) $\Rightarrow$ (may be slow)
  - The long-term scheduler controls the **degree of multiprogramming**
- Processes can be described as either:
  - **I/O-bound process** – spends more time doing I/O than computations, many short CPU bursts
  - **CPU-bound process** – spends more time doing computations; few very long CPU bursts
- Long-term scheduler strives for good ***process mix***

Medium-Term Scheduler

- **Medium-term scheduler** can be added if degree of multiple programming needs to decrease
  - Remove process from memory, store on disk, bring back in from disk to continue execution: **swapping**



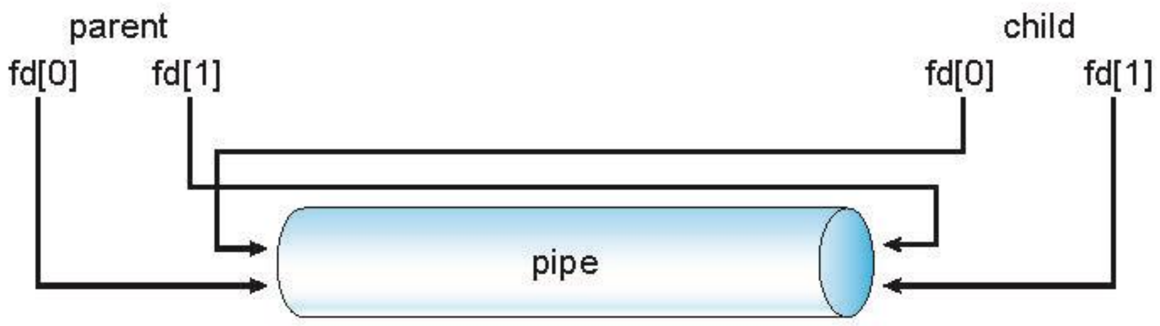
IPC: Pipes

Pipes

- Acts as a channel allowing two processes to communicate
- Issues:
  - Is communication unidirectional or bidirectional?
  - In the case of two-way communication, is it half or full-duplex?
  - Must there exist a relationship (i.e., ***parent-child***) between the communicating processes?
  - Can the pipes be used over a network?
- Ordinary pipes – cannot be accessed from outside the process that created it. Typically, a parent process creates a pipe and uses it to communicate with a child process that it created.
- Named pipes – can be accessed without a parent-child relationship.

Ordinary Pipes

- Ordinary Pipes allow communication in standard producer-consumer style
- Producer writes to one end (the **write-end** of the pipe)
- Consumer reads from the other end (the **read-end** of the pipe)
- Ordinary pipes are therefore unidirectional
- Require parent-child relationship between communicating processes

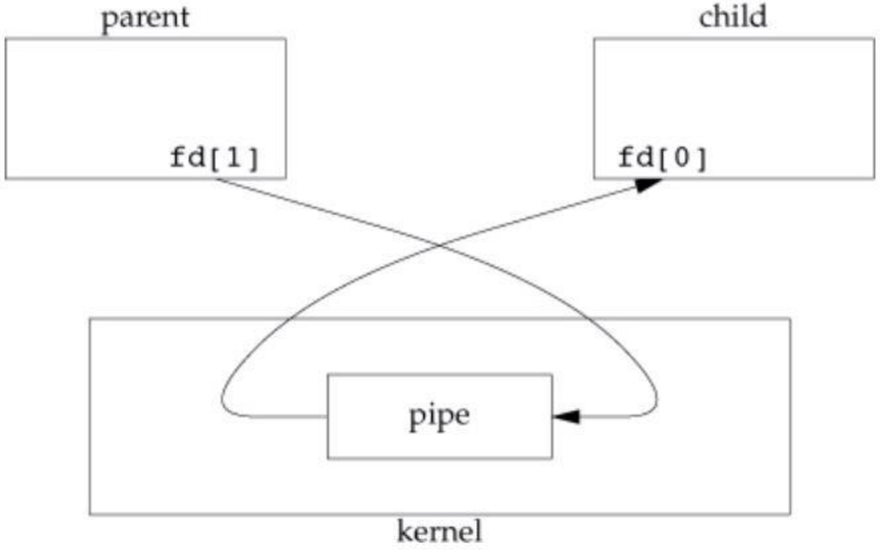


- Windows calls these **anonymous pipes**

Pipes: creation and setup

```
#include <unistd.h>

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



- The data in the pipe flows through the kernel.
- Normally, the process that calls pipe then calls fork, creating an IPC channel from the parent to the child, or vice versa.

Pipes: creation and setup

```
#include "apue.h"

int
main(void)
{
    int     n;
    int     fd[2];
    pid_t   pid;
    char    line[MAXLINE];

    if (pipe(fd) < 0)
        err_sys("pipe error");
    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid > 0) {        /* parent */
        close(fd[0]);
        write(fd[1], "hello world\n", 12);
    } else {                     /* child */
        close(fd[1]);
        n = read(fd[0], line, MAXLINE);
        write(STDOUT_FILENO, line, n);
    }
    exit(0);
}
```

Named Pipes

- Named Pipes are more powerful than ordinary pipes
- Communication is bidirectional
- No parent-child relationship is necessary between the communicating processes
- Several processes can use the named pipe for communication
- Provided on both UNIX and Windows systems

Activity!