



**POLITECNICO
DI TORINO**

Inter Process Communication

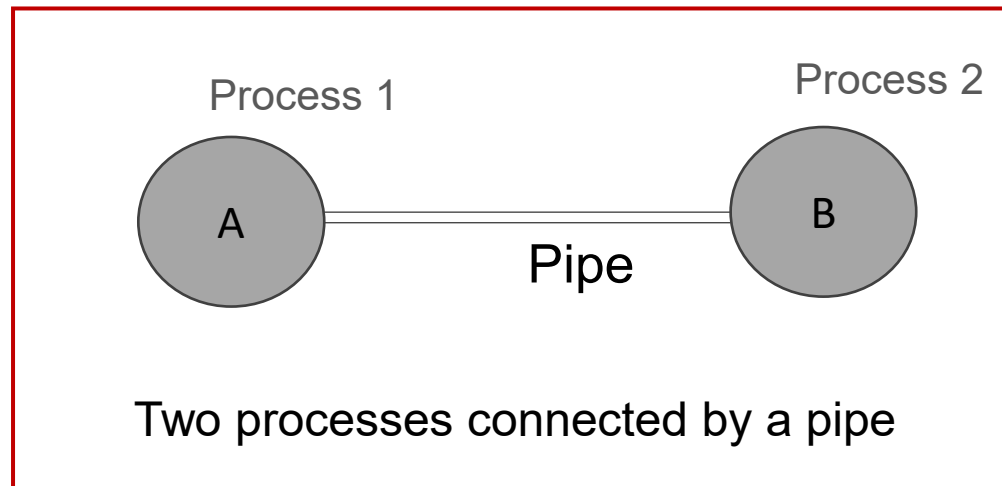
Operating Systems – Sarah Azimi

OUTLINE

- IPC mechanisms:
 - Shared Memory
 - Message Passing
 - **Pipes (named and unnamed)**
 - **Signals**

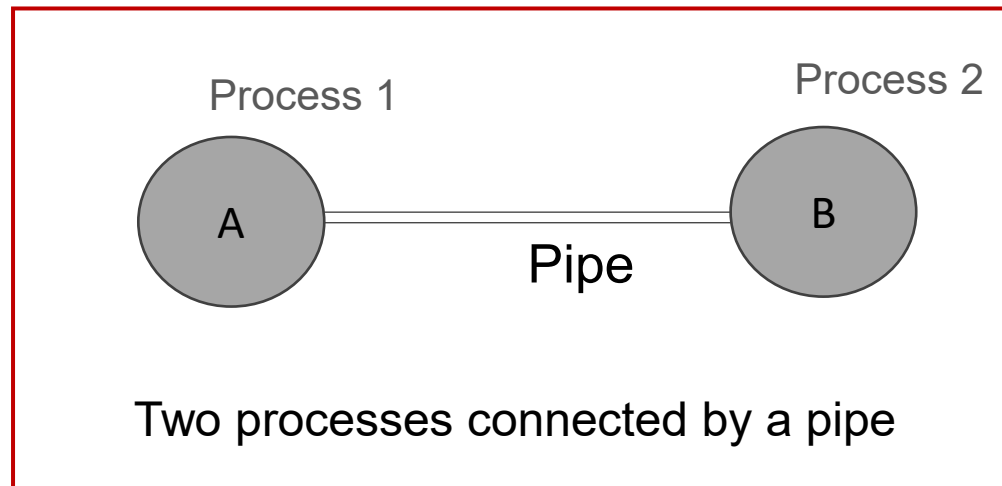
Unix Pipes

- Pipe sets up communication channel between two processes (related or unrelated):
 - Pipes are uni-directional.
 - They can only transfer data in one direction.
 - Both the writer and reader process of a pipeline execute concurrently.
 - A pipe automatically buffer the output of the writer and suspends the writer if the pipe gets too full.
 - Similarly, if a pipe is empty, the reader is suspended until some more output becomes available.



Unix Pipes

- There are two kinds of pipes.
 - **Unnamed piped** for communication between a parent and its child, with one process writing and the other process reading.
 - **Named pipes** for communication between unrelated processes. Any process can communicate with another one using named pipes.



Named Pipes or FIFOs

- Named pipes, often referred to as *FIFOs* (*first in, first out*), are less restricted than unnamed pipes and offer the following advantages:
 - They have a name that exists in the file system.
 - They may be used by unrelated processes.
 - They exist until explicitly deleted.
 - They have a larger buffer capacity, typically about 40K.
- `mkfifo()` to create a named pipe
- `Unlink()` to remove the named pipe from the file system

Named Pipes or FIFOs

- `mkfifo()` allows you to create a FIFO special file (a named pipe).
 - Syntax:

```
int mkfifo (const char *path, mode_t mode);
```
 - `Path` corresponds to the name of the pipe.
 - `Mode` corresponds to the permission mode flags.
 - A named pipe is first opened using `open()`.
 - `Write()` adds data at the start of the FIFO queue.
 - `Read()` removes data from the end of the FIFO queue.

Named Pipes or FIFOs

- `mkfifo()` allows you to create a FIFO special file (a named pipe).
 - Syntax:
`int mkfifo (const char *path, mode_t mode);`
 - When a process has finished using a named pipe, it should close it using `close()`.
 - Writer processes should open a named pipe for writing only
 - Reader processes should open a pipe for reading only.
- when a named pipe is no longer needed, it should be removed from the file system using `unlink()`.
 - Syntax:
`int unlink (const char *path);`

Named Pipes or FIFOs

WRITER

```
# include <stdio.h>
# include <sys/types.h>
# include <sys/stat.h>
# include <fcntl.h>

Int main (){
    const char *pipeNAME = "pipeName";
    int mynumber;
    kmfifo (pipeNAME, 0666);
    int fd = open (pipeName, O_CREATE | O_WRONLY);
    write (fd, mynumber, sizeof(int));
    close (fd);
    unlink (pipeNAME);
}
```

READER

```
# include <stdio.h>
# include <sys/types.h>
# include <sys/stat.h>
# include <fcntl.h>

Int main (){
    const char *pipeNAME = "pipeName";
    int number;
    int fd = open (pipeName, O_CREATE | O_RDONLY);
    read (fd, &number, sizeof(int));
    close (fd);
    unlink (pipeNAME);
}
```


Signal

- A signal is a software notification to a process of an event:
 - Signal is generated by a particular event.
 - Signal is delivered to a process.
 - Signal is handled.

Signal

- A signal is a software notification to a process of an event:
 - Signal is generated by a particular event.
 - Signal is delivered to a process.
 - Signal is handled.
- There are three ways in which a process can respond to a signal:
 - Ignoring the signal
 - `signal (SIG#, SIG_IGN)`
 - Executing the default action associated with the signal
 - `signal (SIG#, SIG_DFL)`
 - Catching the signal by invoking a corresponding signal-handler function
 - `signal(SIG#, myHandler)`
- OS provides a facility for writing your own event handlers in the style of interrupt handlers.

Send a Signal

- To Raise a signal
 - Syntax:
`kill (pid, signal);`
 - `Pid` is an input parameter, id of the process that receives the signal.
 - `Signal` is a signal number.
 - returns 0 to indicate success, error code otherwise.
- Example
 - `pid_t iPid = getpid(); /* Process gets its id.*/`
 - `kill(iPid, SIGINT); /* Process sends itself a SIGINT signal (commits suicide) */`

Signal Handler

- A *signal handler* is used to process signals
- Corresponding to each signal there is a signal handler.
- Called when a process receives a signal.
- When the signal handler returns the process continues, as if it was never interrupted.

User Defined Signal Handlers

```
# include <stdio.h>
# include <signal.h>
Void handle-sigint ()
{
    printf ("Caught signals\n");
}

Int main (){
    signal (SIGINT, handle-sigint);
    while (1) ;
    return 0;
}
```

```
# include <stdio.h>
# include <signal.h>
# include <unistd.h>

Void handle-sigint (){
    printf ("Caught signals\n");
    exit (0);
}

Int main (){
    pid_t pid;
    pid = fork ();
    If (pid ==0){
        signal (SIGINT, handle-sigint);
        while (1);
    }
    else {
        printf ("I am killing my child\n");
        kill (pid, SIGINT)
    }
}
```