SISTEMAS OPERATIVOS

3004610 - 1

German Sánchez Torres, I.S., M.Sc., Ph.D.

Profesor, Facultad de Ingeniería - Programa de Sistemas

Universidad del Magdalena, Santa Marta.

Phone: +57 (5) 4214079 Ext 1138 - 301-683 6593

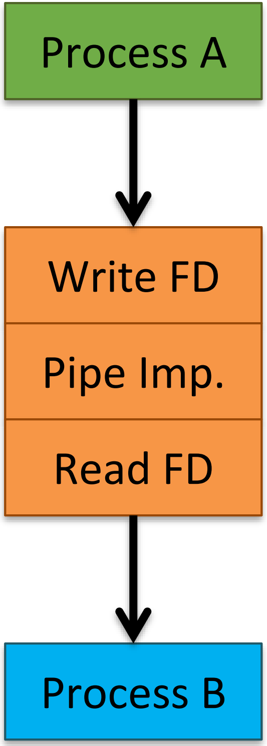
Edificio Docente, Cub 3D401.

Email: sanchez.gt@gmail.com –gsanchez@unimagdalena.edu.co

Ejemplos uso tuberías
int pipe(int [2])

Features of Pipes

- As a general rule, one process will write to the pipe (as if it were a file), while another process will read from the pipe.
- Data is written to one end of the pipe and read from the other end.
- A pipe exists until both file descriptors are closed in all processes
- On many systems, pipes are limited to 10 logical blocks, each block has 512 bytes.
- Data stored in kernel via pipefs filesystem
- Is unidirectional



Process A

Write FD

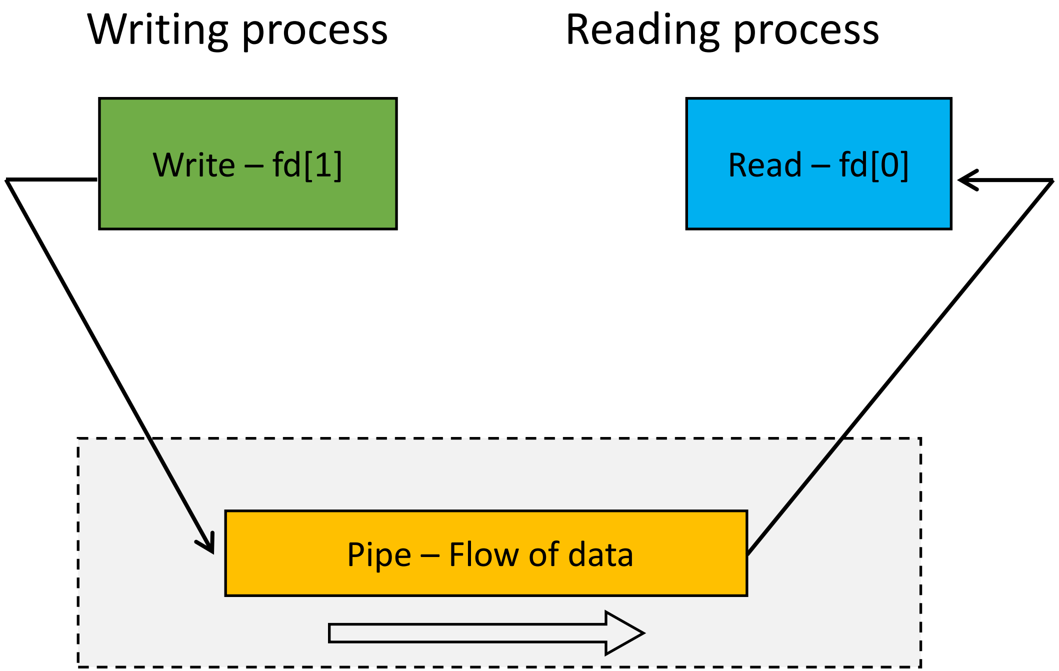
Pipe Imp.

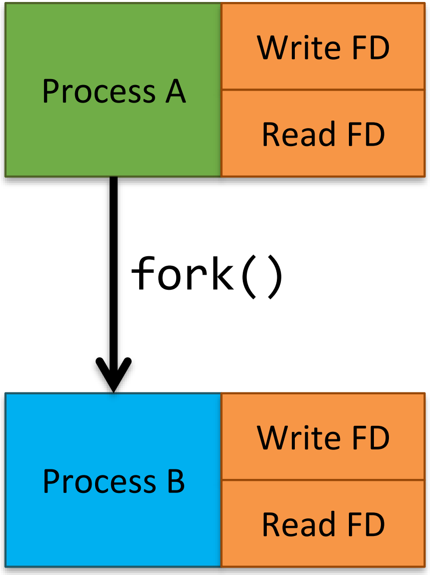
Read FD

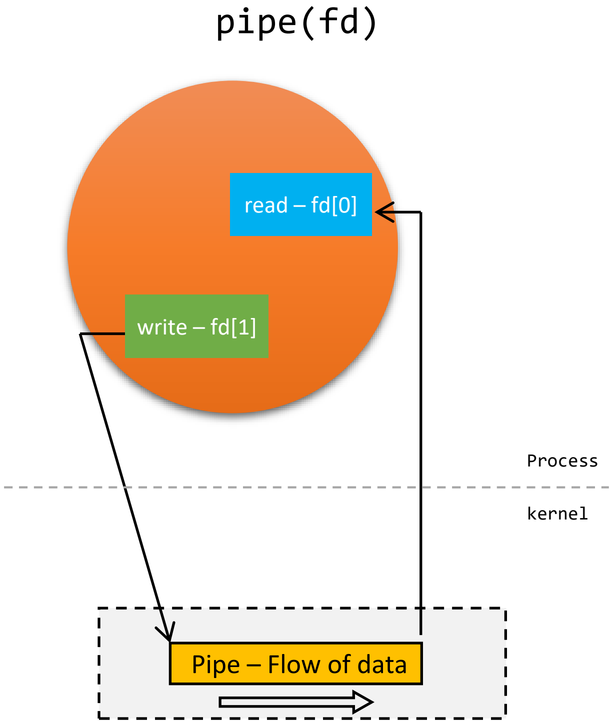
Process B

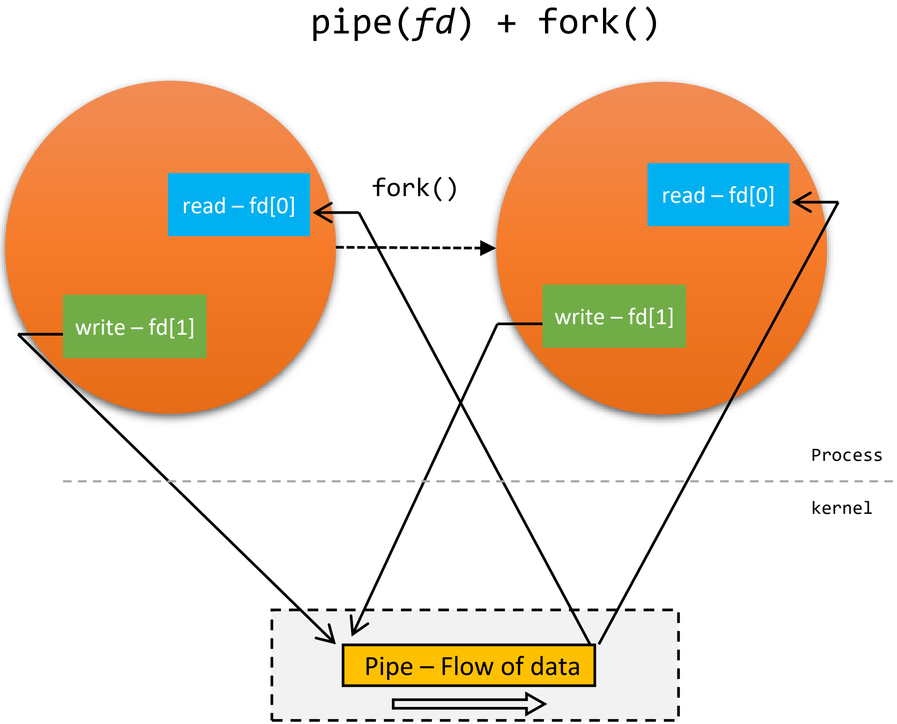
Piping Between Two Processes

- The pipe is represented in an array of 2 file descriptors (int)

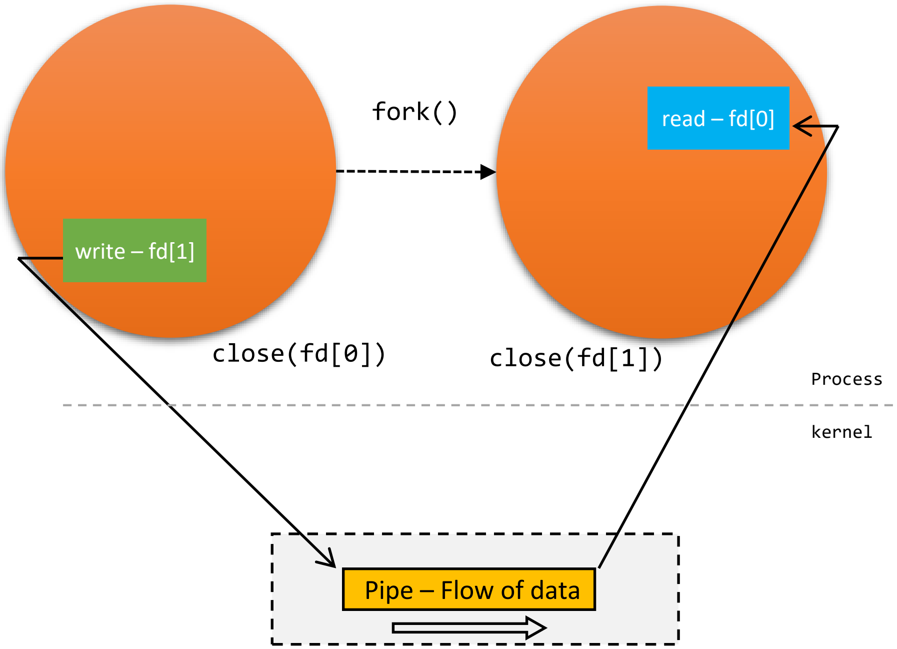


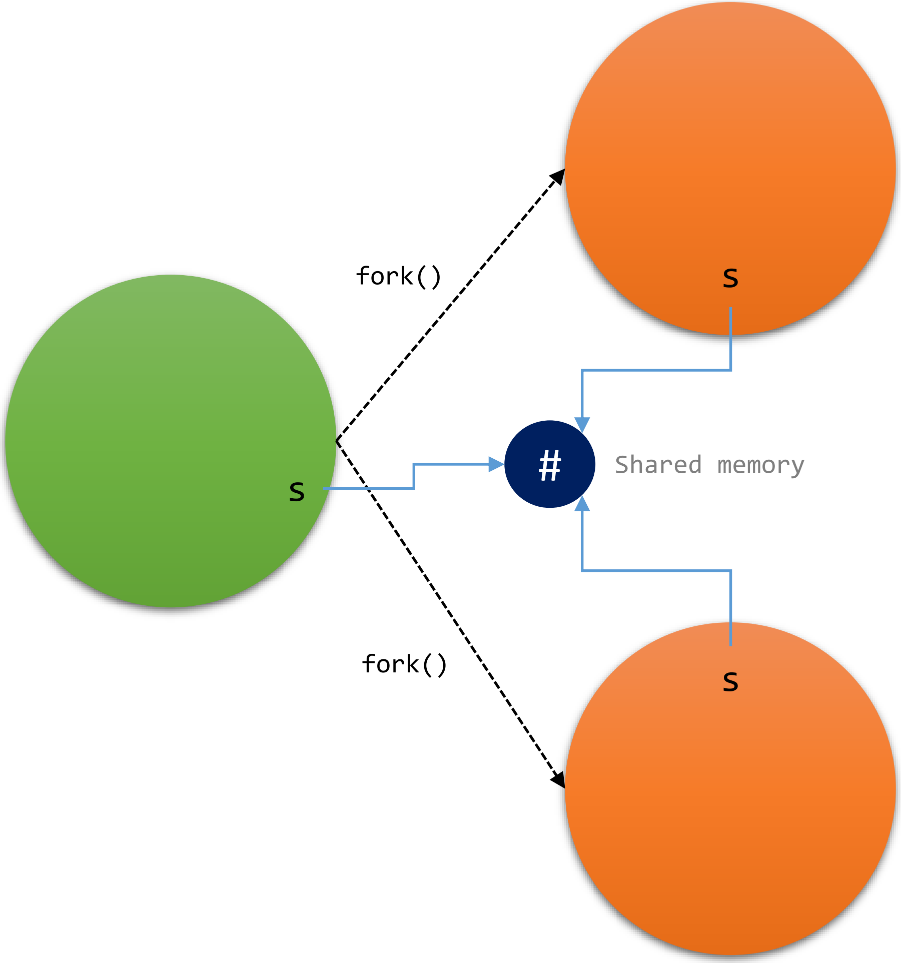


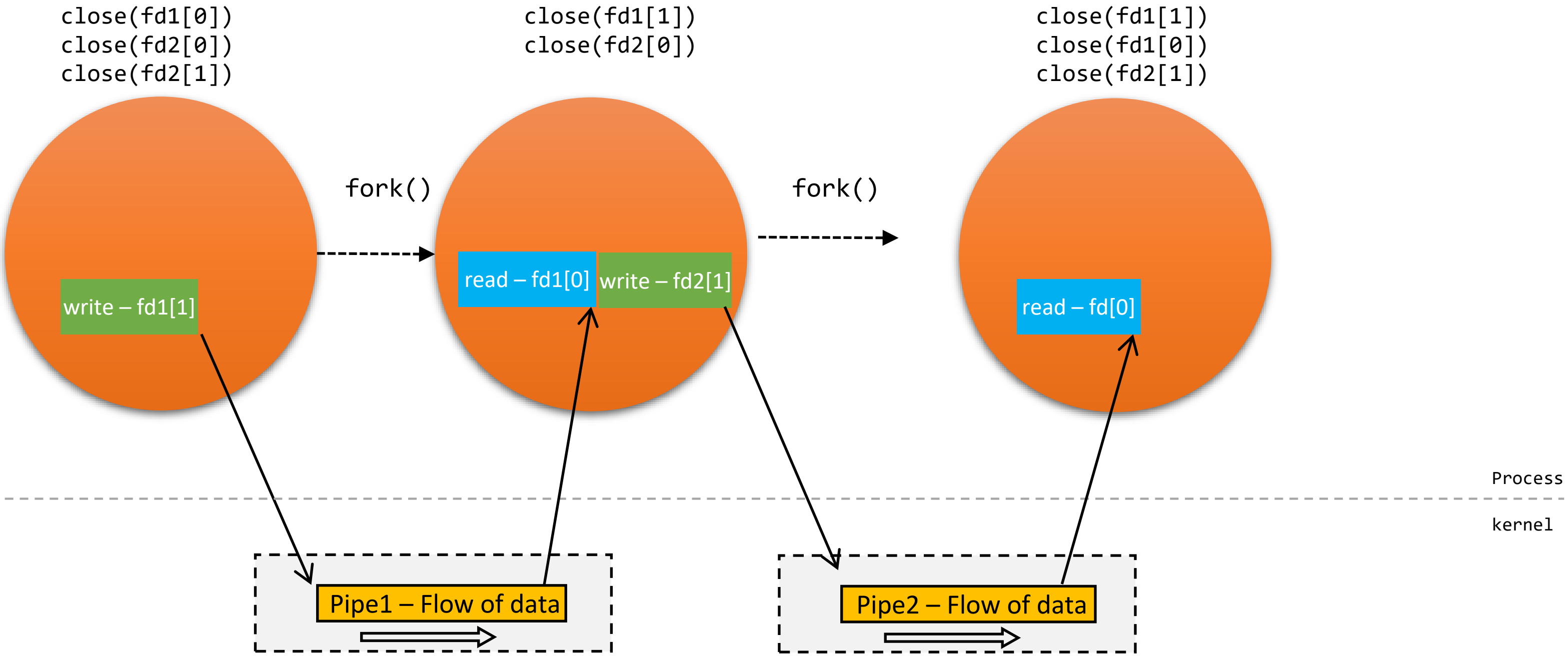

pipe(fd)
read – fd[0]
write – fd[1]
Process
kernel
Pipe – Flow of data


pipe(fd) + fork()
read – fd[0]
write – fd[1]
fork()
read – fd[0]
write – fd[1]
Process
kernel
Pipe – Flow of data

pipe(*fd*) + fork()+ close(fd[1|0])



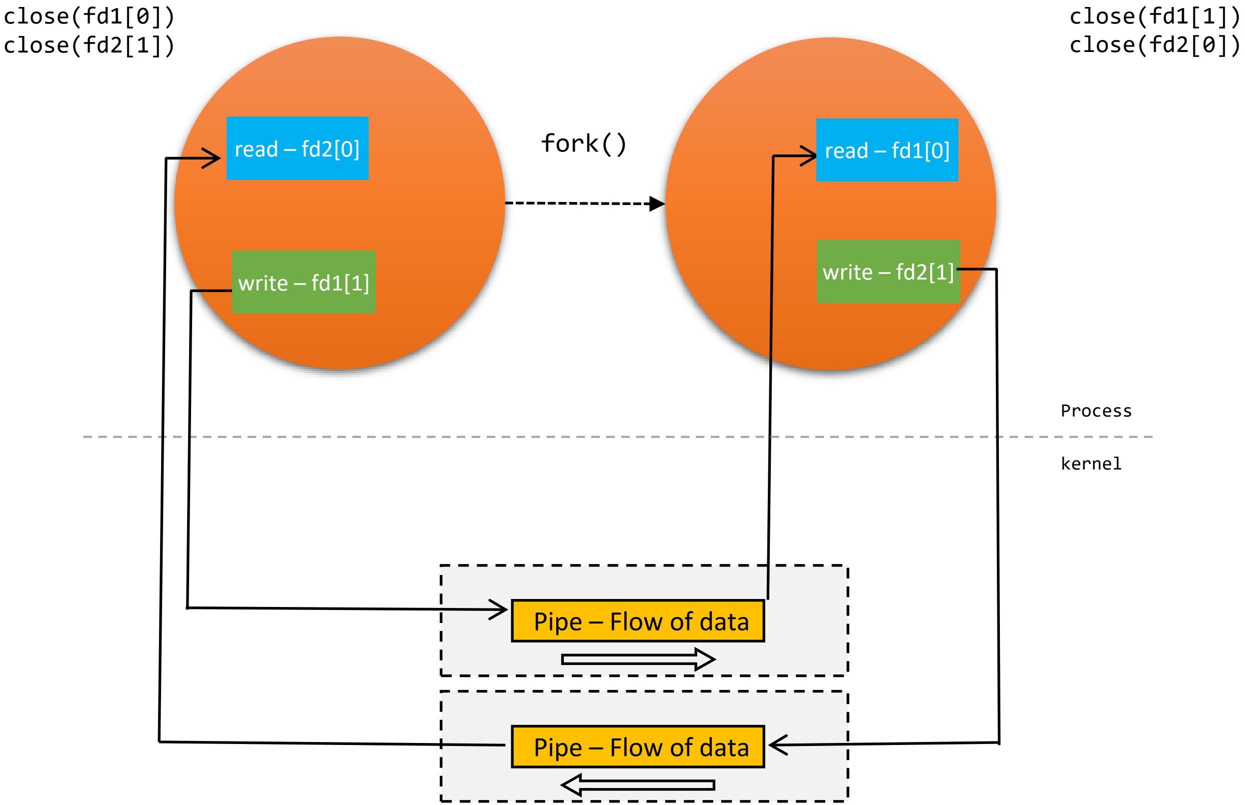

fork()
S
S
#
Shared memory
fork()
S

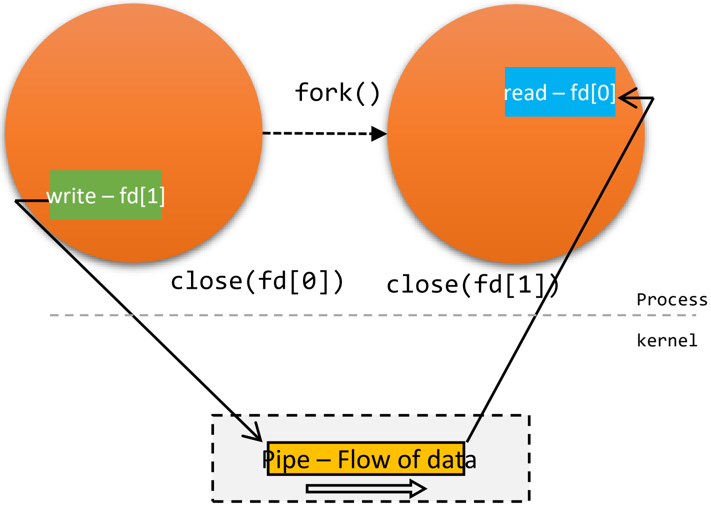


```
close(fd1[0])
close(fd2[0])
close(fd2[1])
```

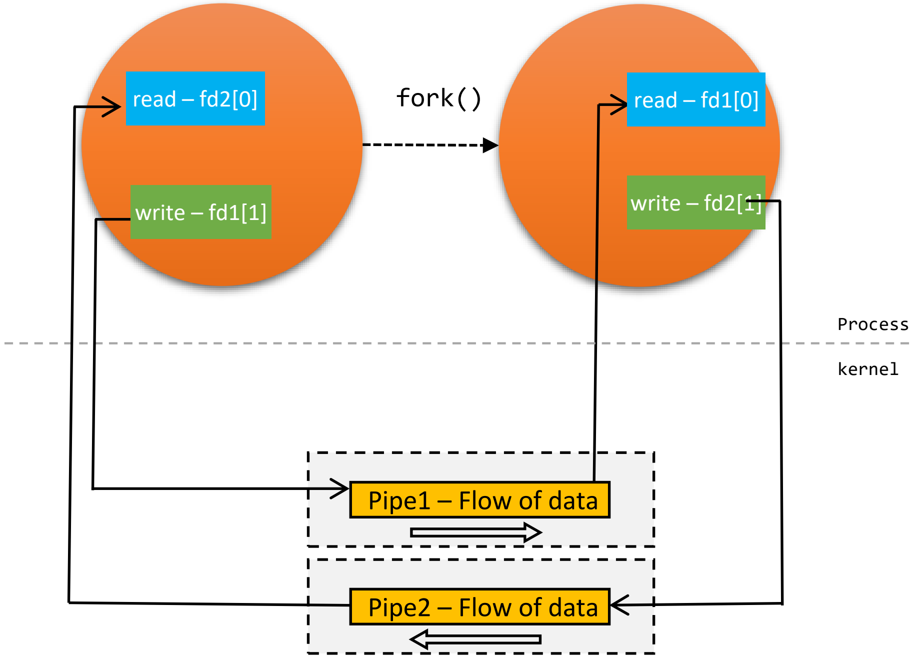
```
close(fd1[1])
close(fd2[0])
```

```
close(fd1[1])
close(fd1[0])
close(fd2[1])
```


close(fd1[0])
close(fd2[1])
close(fd1[1])
close(fd2[0])
read – fd2[0]
write – fd1[1]
fork()
read – fd1[0]
write – fd2[1]
Process
kernel
Pipe – Flow of data
Pipe – Flow of data



```
fd[2]

pipe(fd)

child = fork()
if(!child )
    close(fd[1])
    .
    .
    do_child()
    .
    .
    close(fd[0])
else
    close(fd[0])
    .
    .
    do_root()
    .
    .
    close(fd[1])
    wait()
endif
```



```
fd1[2]
fd2[2]

pipe(fd1)
pipe(fd2)

child = fork()
if(!child )
    close(fd1[1])
    close(fd2[0])
    .
    .
    do_child()
    .
    .
    close(fd1[0])
    close(fd2[1])
else
    close(fd1[0])
    close(fd2[1])
    .
    .
    do_root()
    .
    .
    close(fd1[1])
    close(fd2[0])
    wait()
endif
```

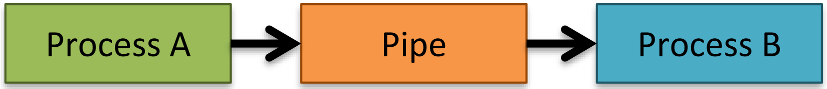
```
fd[3][2]

For(int i=0; i<3; i++)
 write(fd[i][1], &buf, strlen(buf))
```

```
 write(fd1[1], &buf, strlen(buf))
 write(fd2[1], &buf, strlen(buf))
 write(fd3[1], &buf, strlen(buf))
```

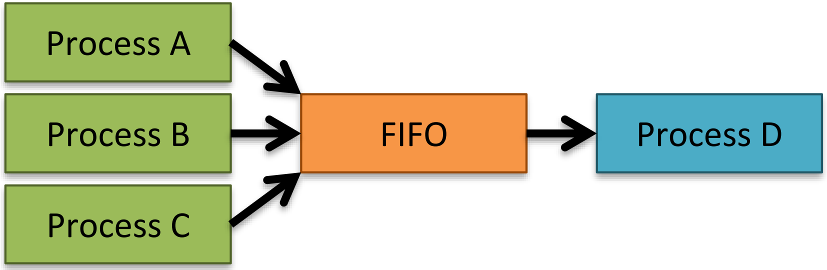
Pipe Paradigms

Pipes are useful for implementing many design patterns and idioms:

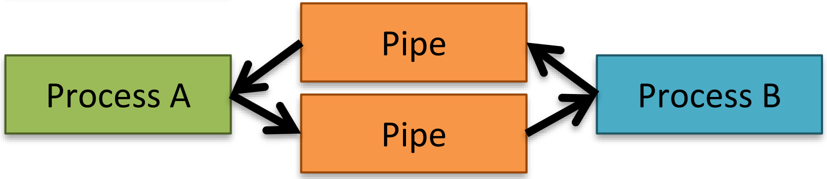
Producer / Consumer



Client / Server



Active Object

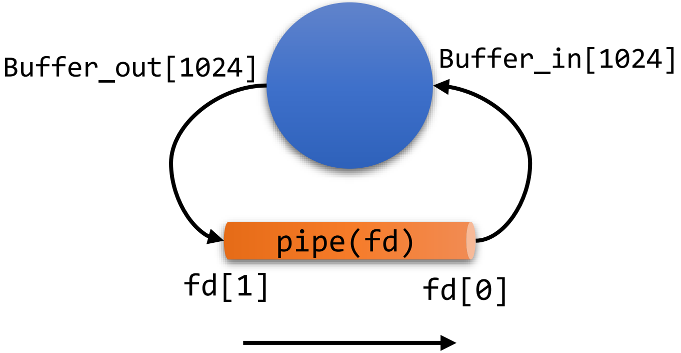


pipe System Call (unnamed)

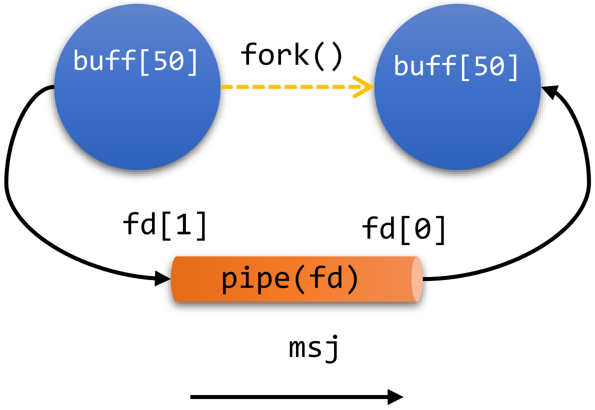
Creates a half-duplex pipe.

- Include(s): **< unistd.h>**
- Syntax: ***int pipe (int pipefd[2]);***
- Return: Success: 0 - Failure: -1 - Sets errno: Yes
- Arguments: None
- If successful, the ***pipe*** system call will return two integer file descriptors, pipefd[0] and pipefd[1].
  - pipefd[1] is the write end to the pipe.
  - pipefd[0] is the read end from the pipe.

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

int main(){
   int fd[2], n;
   char Buffer_out[1024];
   char Buffer_in [1024];
   char data[] = {"Hola mundo pipes"};
   pipe(fd);

   strcpy(Buffer_out, data);
   printf("[%d]write:--> %s\n", getpid(), Buffer_out);
   write(fd[1], Buffer_out, strlen(Buffer_out));

   n = read(fd[0], Buffer_in, 1024);
   Buffer_in[n] = '\0';
   printf("[%d]read:<-- %s\n",getpid(), Buffer_in );
return EXIT_SUCCESS;
}
```

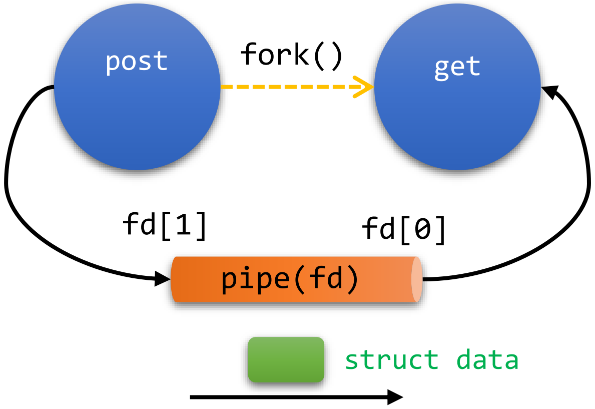


```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <wait.h>

int main(){
   int fd[2],n;
   char buff[50];
   pipe(fd);

   if(fork()){
      char msg[] = {"Mensaje de texto"};
      close(fd[0]);
      write(fd[1], msg, strlen(msg));
      printf("[%d]write:--> %s\n", getpid(), msg);
      wait(NULL);
   }
   else{
      close(fd[1]);
      n = read(fd[0], buff, 50);
      buff[n] = '\0';
      printf("[%d] read:<-- %s\n",getpid(), buff );
   }
return EXIT_SUCCESS;
}
```



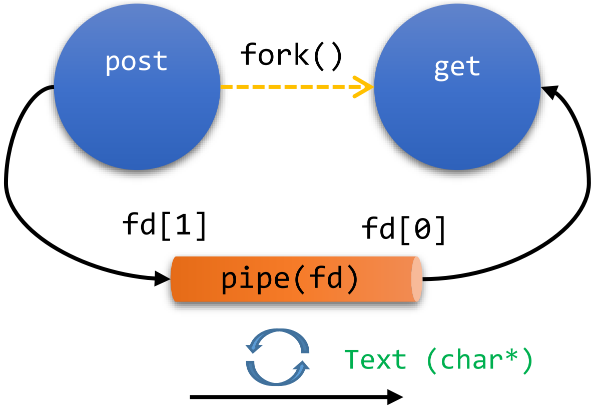
```
#include <stdio.h>
#include <wait.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#define MAX_BUFF 1024

struct data{
   int a;
   float b;
};

int main(){
   int fd[2], n;
   pipe(fd);

   if(fork()==0){
      struct data get;
      close(fd[1]);
      n = read(fd[0], &get, sizeof(struct data));
      printf("[%d]read:<--\t[%d|%.2f]\n", getpid(), get.a, get.b);
   }
```

```
 else{
      struct data post;
      close(fd[0]);
      post.a = 10;
      post.b = 2.3;
      printf("[%d]write:-->\t[%d|%.2f]\n", getpid(), post.a,
post.b);
      write(fd[1], &post, sizeof(struct data));
      wait(NULL);
   }
return EXIT_SUCCESS;
}
```



```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/wait.h>

#define MAX_READ 256
#define EOL '\0'
char buff[MAX_READ];


int main(){
   int fd[2], n;
   pipe(fd);
   printf("write quit to exit\n\n");
   if(fork()!=0){
      close(fd[0]);
      do{
         fgets(buff, MAX_READ, stdin);
         if(strlen(buff)>1){
            buff[strlen(buff)-1] = '\0';
            printf("[%d]write-->:%s\n",getpid(),buff);
            write(fd[1], buff, strlen(buff));
         }
       }while(strcmp(buff,"quit") !=0);
      close(fd[1]);
      wait(NULL);
    }
```

```
   else{
         close(fd[1]);
         while( (n=read(fd[0],buff, MAX_READ)) >0 ){
            buff[n] = EOL;
            printf("[%d]read<--:%s\n",getpid(),buff);
          }
      }
return EXIT_SUCCESS;
  }
```


post
fork()
get
fd[1]
fd[0]
pipe(fd)
Text (char*)