Operating Systems

Inter-process Communication (IPC)

Spring 2016

Francesco Fontanella

Inter-process communication

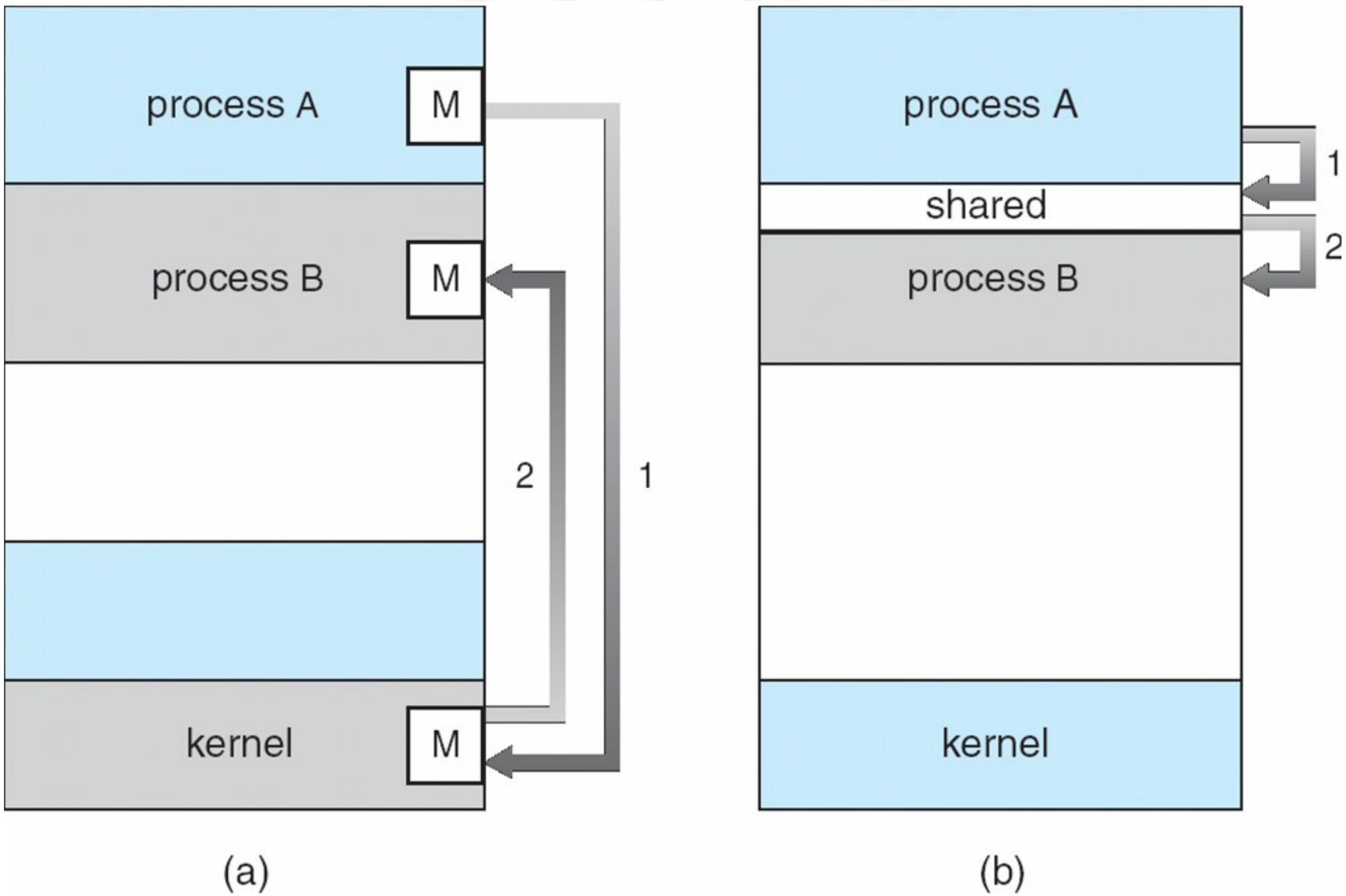
- Processes executing concurrently may be independent or **cooperanting**
- Two (or more) processes are independent if they cannot influence each other

  The OS guarantees for this
- Two (or more) processes cooperate if the execution of a process can influence that of another one

- Inter-process communication (IPC) must be managed by the OS
- Two main models for IPC:
  - **Shared memory**
  - **message passing**
- Most operating systems implement both mechanism for IPC

Message passing

Shared memory



Shared memory

- A process allocates a memory area (usually in its own address space)
- Other processes add this area to their address space
- The OS, **only in this case**, allows other processes to access to the memory of that which allocated the memory area.
- Then communication takes place reading and writing the shared memory
- **data syncronization must be managed by processes**

Producer-consumer problem: solution

- Processes can communicate through a shared-memory buffer
- The producer/consumer process writes/reads to/from the buffer
- The buffer may be:
  - Unlimited (theorically): the producer doesn't worry if the buffer is full
  - Limitated: the buffer has a fixed size. If the buffer is full, the producer must wait

Producer/consumer problem

- A process (the producer) generates data (record, chars, objects, etc.) and it wants send them to another process (consumer) which processes the data

- They communicate through a shared variable

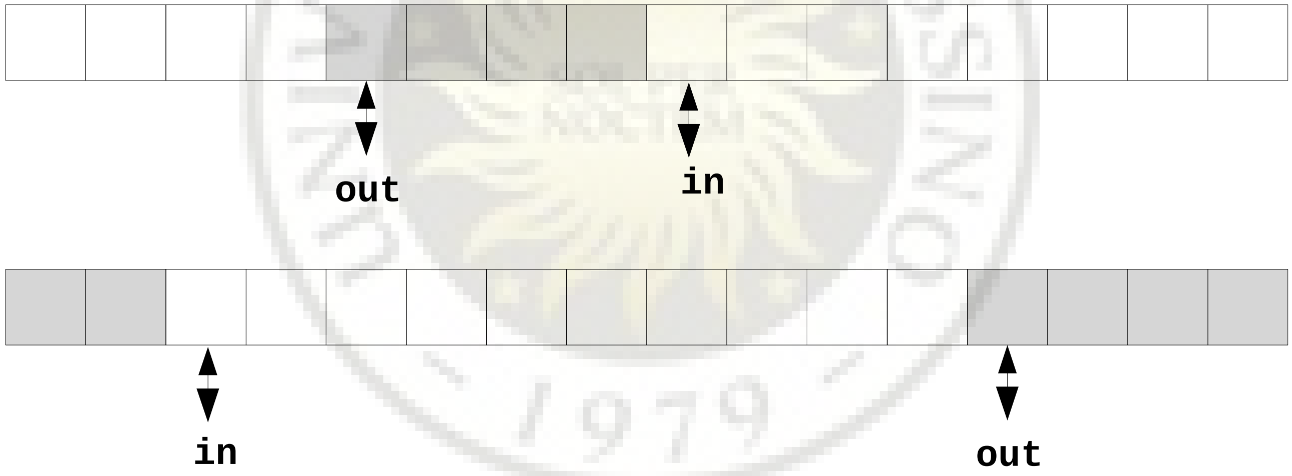
- Data synchronization must be guaranteed:
  - The producer does not overwrite data not yet processed by the consumer
  - The consumer must wait for the generation of new data

Producer-consumer problem: solution

- Processes can communicate through a shared-memory buffer
- The producer/consumer process writes/reads to/from the buffer
- The buffer may be:
  - Unlimited (theorically): the producer doesn't worry if the buffer is full
  - Limitated: the buffer has a fixed size. If the buffer is full, the producer must wait

Circular array

- A limited buffer can be implemented through a circular array



producer/consumer: example

```
#define BUFFER_SIZE 10
#define IN  0
#define OUT 1
typedef struct {
   .
   .
   .
} item;

item buffer[BUFFER_SIZE];
int inout[2];

inout[IN] = inout[OUT] = 0;
```

Producer

```
void producer(item buffer[], int &in, int out)

  item tmp;

  while (true) {
    /* item production */
        .
        .
    while (( (in + 1) % BUFFER_SIZE )  == out)
     ;   /* waits if there is no room in the buffer*/
     buffer[in] = item;
    in = (in + 1) % BUFFER SIZE;
  }
}
```

Busy waiting

Consumer

```
void consumer(item buffer[], int in, int &out)
{
  item tmp

  while (true) {
    while (in == out)
      ; // buffer empty: waiting

    // the item is pop out
    tmp = buffer[out];
    out = (out + 1) % BUFFER SIZE;

    /* si consuma l'item*/
        .
        .
  }
}
```

Busy waiting

POSIX shared memory

- Syscall to allocate shared memory:

```
int shmget(key_t key, size_t size, int shmflg)
seg_id = shmget(IPC_PRIVATE, size, S_IRUSR| S_IWUSR)
```

  `key`: segment (area) identifier. The constant value `IPC_PRIVATE` specifies that a new segment must be allocated.

  `size`: segment size.

  `shmflg`: specifies the access mode. `S_IRUSR| S_IWUSR` specifies both read and write

  – **returns** the id of the segment allocated

- To “attach” a shared memory area (segment) to the the address space of the calling process:

```
sh_mem = (char *) shmat (id, NULL, 0)
```

`id`: segment id

  - The second parameter may specify **where** to attach the segment. It the NULL value is passed, the OS chooses a suitable (unused) address
  - The third parameter specifies the access mode:
    - 0 read only
    - > 0 read and write

- When the segment is no longer needed, it can be “detached” from the address space of the calling process by the function:

  ```
  shmdt(const void *shmaddr)
  ```

  – **shmaddr**: pointer to the segment to be detached

Example

```c
#include <sys/ipc.h>
#include <sys/shm.h>

int main()
{
   pid_t  pid;
   int seg_id;
   char *sh_mem;

   seg_id = shmget(IPC_PRIVATE, size, S_IRUSR| S_IWUSR);

   sh_mem = (char *) shmat(seg_id, NULL, 1);

    pid = fork();
    if (pid < 0) { // error!
      fprintf(stderr, "Fork Failed");
      exit(-1);
   }
```

```
 if (pid == 0)  // child process
  printf("child process, shared string: %s",  sh_mem);
 else  { // father process
  sprintf (sh_mem, "hello!");
  exit(0);
 }
}
```

Message passing

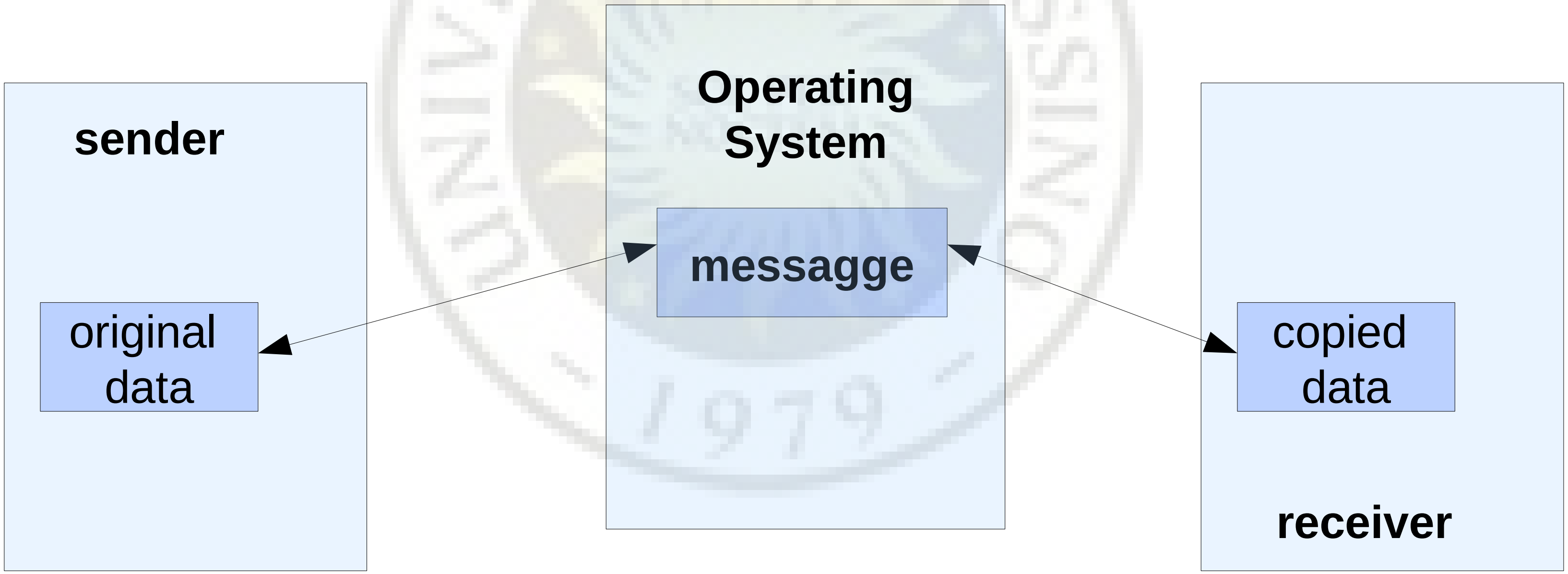
- It allows processes to comunicate and **synchronize** their actions
- A message is a set of information **formatted** by a sender process and **interpretated** by a receiver process
- message passing is implemented by copying the content of the message from the sender space address to the receiver space address
- It allows inter-process communication without memory sharing

send and receive

send(message)
- Used by the sender
- It must specify the receiver process
- Message size can be fixed or variable

receive(message):
- Used by the receiver
- It is not needed to specify the sender

- If two processes P and Q want communicate, must:
  - establish a communication channel
  - Use send and receive for message passing



Direct Communication

- Processes must know the PID of the process they want communicate with
  - send (PID1, m) – sends the message m to PID1 process
  - receive(PID2, m) – receives the message m form the process PID2
- Communication channel properties:
  - It is automatically established
  - It is dedicated for the communication between the two processes
  - It is usually bidirectional

Indirect Communication

- Messages are sent/received to/from **ports** (called also *mailboxes*):
  - Ports are uniquely identified, usually an integer value
  - Two processes may communicate if they share a port
- Communication channel properties:
  - It is established once the port is shared
  - It may be associated to more processes
  - Two processes may share more channels (ports)
  - Channels may be uni/bi-directional

- OS allows processes:
  - To create a port A
  - send/receive messages through the port A
  - Destroy (close) the port A

- Primitive
  - **send**(*A*, *m*) – sends the message m to the port A
  - **receive**(*A*, *m*) – receives (copy in) the message m from the port A

Synchronization

- Message passing can be both synchronous (blocking) or asynchronous
- **synchronous sending:** the sender awaits for (blocked) the receiver reception
- **asynchronous sending:** the sender continues to execute normally, checking if the message has been received
- **synchronous reception:** the receiver is blocked while awaiting for the message
- **asynchronous reception:** the receive primitive does not block the receiver, which must check whether the message has been sent or not

Producer-consumer problem

- It can be solved by using synchronous sending and receiving messages
- The producer just sends messages, and then it is blocked until the receiver do not read it
- The consumer instead awaits for producer message, and it is blocked if no messages have been sent

```
#define BUFFER_SIZE 10
#define IN  0
#define OUT 1
typedef struct {
. . .
} item;

item buffer[BUFFER_SIZE];
int inout[2];
```

message passing vs shared memory

Message passing

```
void producer(pid_t c_id)

  item tmp;

  while (true) {
    /*  produce an  item*/
        .
        .
    send(&tmp, sizeof(item),
         c_id);

  }
}
```

Shared memory

```
void producer(item buffer[], int
                   &in, int out)

  item tmp;

  while (true) {
    /* produce an item*/
        .
        .
    while (((in + 1) % BUFFER_SIZE)
              == out)
        ;
     buffer[in] = item;
    in = (in + 1) % BUFFER SIZE;
  }
}
```

Message passing

```
void consumer(pid_t p_id)
{
  item tmp;

  while (true) {
   receive(&tmp, sizeof(item),
           p_id);

    /* consume the item*/
       .
       .
  }
}
```

Shared memory

```
void consumer(item buffer[], int in,
                             int &out)
{
  item tmp;

  while (true) {
    while (in == out)
      ;

    // extract the item
    tmp = buffer[out];
    out = (out + 1) % BUFFER SIZE;

    /* consume the item*/
       .
       .
  }
}
```

producer/consumer: shared memory

```
int main(int argc, char* argv[])
{
/* process id */
 pid_t pid;

  /*  memory segment id */
  int buffer_id, inout_id;;

  item* shared_buffer; /* buffer pointer */
  int*  shared_inout;  /* inout  pointer */

  if ((buffer_id = shmget(IPC_PRIVATE, sizeof(item)*BUFFER_SIZE,
PERMS)) == -1) {
    fprintf(stderr,"ERROR!: impossible to allocate shared memory\n");
    exit(-1);
  }

  if ((inout_id = shmget(IPC_PRIVATE, sizeof(int)*2, PERMS)) == -1) {
    fprintf(stderr,"ERROR!: impossible to allocate shared memory \n");
    exit(-1);
  }
```

```
if ((shared_buffer = (item *) shmat(buffer_id, 0, 0)) == (item  *) -1) {
  fprintf(stderr,"Unable to attach to segment %d\n",buffer_id);
  exit(-1);
}

if ((shared_buffer = (int *) shmat(inout_id, 0, 0)) == (int  *) -1) {
  fprintf(stderr,"Unable to attach to segment %d\n",buffer_id);
  exit(-1);
}

pid = fork();
if (pid < 0) { // error!
  fprintf(stderr, "Fork Failed");
  exit(-1);
}
if (pid == 0)  // child process:  consumer
  consumer(shared_buffer, shared_inout[IN], sharred_inout[OUT])
else // father process: producer
  producer(shared_buffer, shared_inout[IN], shared_inout[OUT])

 exit(0);
}
```

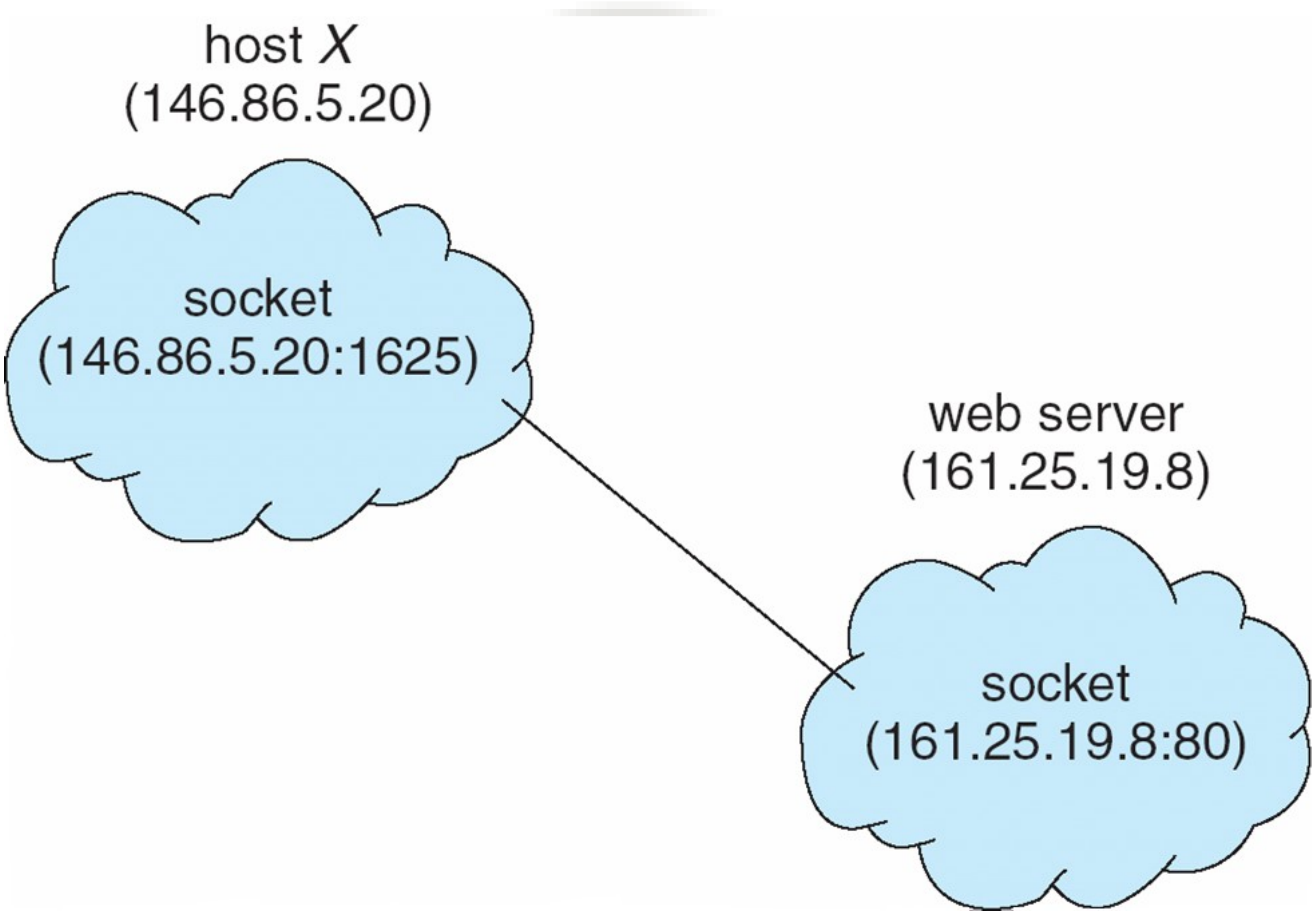
producer/consumer: message passing

```
int main(int argc, char* argv[])
{
  /* process id */
  pid_t pid1, pid2;



  pid1 = getpid();

  pid2 = fork();

 if (pid2 < 0) { // error!
    fprintf(stderr, "Fork Failed");
    exit(-1);
  }

  if (pid2 == 0)  // child process:  consumer
    consumer(pid1);
  else // father process: producer
    producer(pid2);

   exit(0);
}
```

Socket

- A socket is defined as an endpoint of a communication channel
- Two processes across a network can communicate via sockets
- A socket is identified by:
  - IP address
  - port number
  - Example: 143.225.2.121:1625
- The ports in the range 0-1024 are used for standard services.

Example


host X
(146.86.5.20)
socket
(146.86.5.20:1625)
web server
(161.25.19.8)
socket
(161.25.19.8:80)

Pipes

- A pipe is an Inter-process communication channel
- Producer-consumer paradigm
- Unidirectional
- The producer writes on the *write-end* endpoint
- The consumer reads from the read-*end* endpoint
- The OS copies the data from the write-end to the read-end
- In UNIX-based OS, a pipe is a special type of file

Syscall pipe

```
#include <unistd.h>
int pipe(int *fd);
```

– Returns: 0 in case of success, -1 otherwise

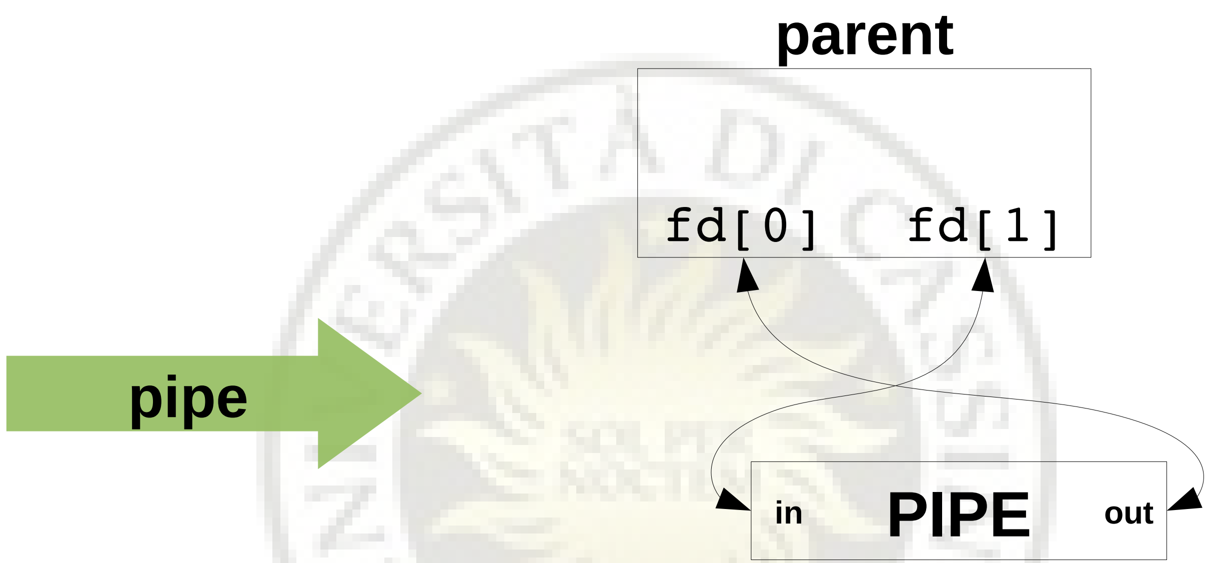
– Stores in the fd parameter the values of two file descriptors

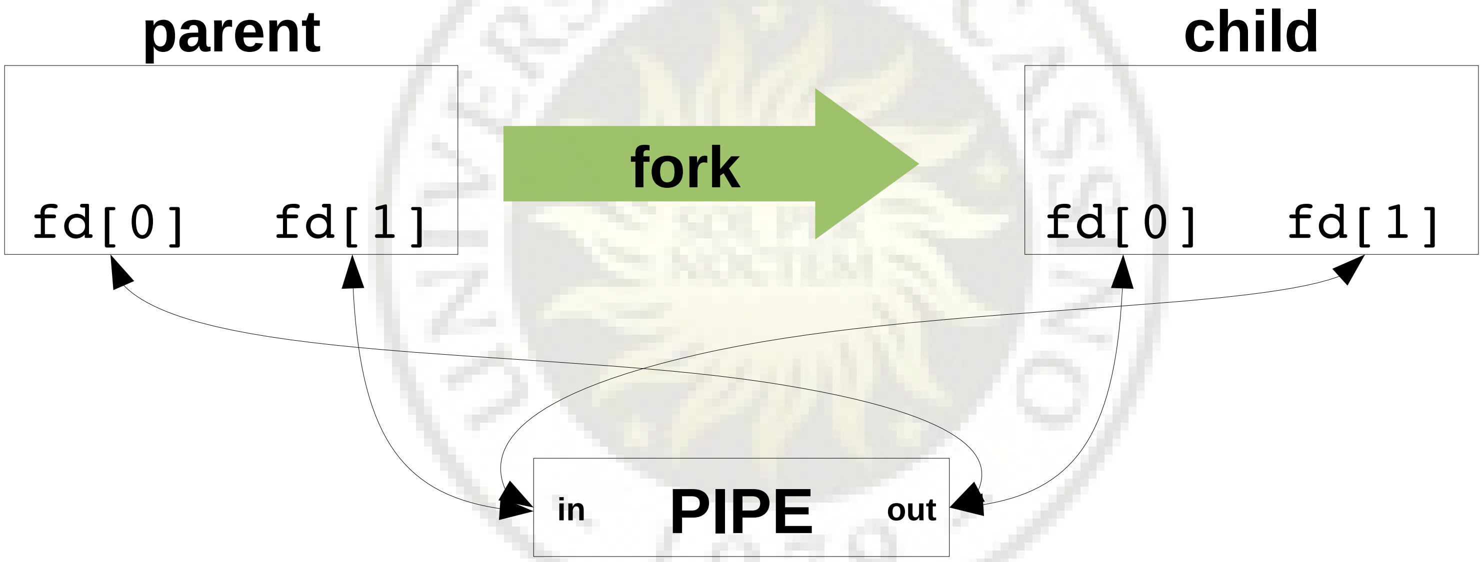
- `filedes[0]` contains the descriptor of a file opened in <u>read mode</u>
- `filedes[1]` contains the descriptor of a file opened in <u>write mode</u>
- Data flow from the write descriptor to the read descriptor:
  - The OS transforms the output of `filedes[1]` into the input of `filedes[0]`

Pipe+fork

- A typical pipe usage is the following:

```
int fd[2];
 .
 .
 .
pipe(fd);
pid=fork();
.
.
.
```


parent
fd[0]
fd[1]
pipe
in
PIPE
out

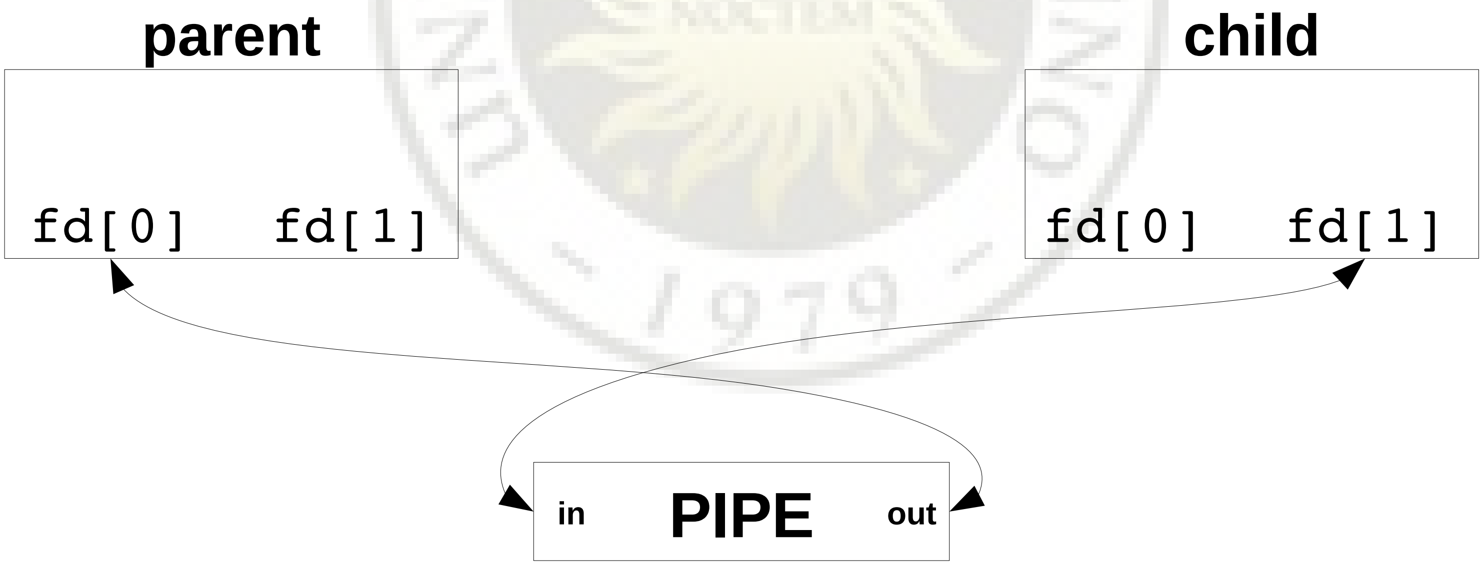

parent
child
fork
fd[0]
fd[1]
fd[0]
fd[1]
in
PIPE
out

From father to child

```
if (pid > 0) // father
  close(fd[0]);
else close(fd[1]); // child
```

from the child to the father

```
if (pid > 0) // father
  close(fd[1]);
else close(fd[0]); // child
```



Pipe usage

- Once a pipe has been created, and the direction chosen, the user can read/write from/to by using the standard (binary) I/O functions
- data from the pipe are read in the <u>same order</u> in which they were written to
- Pipes have a <u>limited capacity</u> (PIPE_BUF constant)

write

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

- If the pipe is full, the write syscall **blocks** the caller until there is no space in the pipe for all the data (no partial writes)
- It returns the number of bytes actually written
- If the fd reading end is closed, this function generates an error (SIGPIPE signal)

Read

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

- Read data from the pipe. Data can't be read again or sent back.
- If the pipe is empty, the calling process is blocked until data are available (blocking syscall)
- If the fd writing end is closed, the function empty the pipe and then returns EOF

Example

```
int main(void)
{
 char write_msg[BUFFER_SIZE] = "Greetings";
 char read_msg[BUFFER_SIZE];
 pid_t pid;
 int fd[2];

 /** create the pipe */
 if (pipe(fd) == -1) {
     fprintf(stderr,"Pipe failed");
     return 1;
 }

 /** now fork a child process */
 pid = fork();

 if (pid < 0) {
     fprintf(stderr, "Fork failed");
     return 1;
 }
```

```
if (pid > 0) {  /* parent process */
   /* close the unused end of the pipe */
   close(fd[READ_END]);

  /* write to the pipe */
  write(fd[WRITE_END], write_msg, strlen(write_msg)+1);

   /* close the write end of the pipe */
   close(fd[WRITE_END]);
}
else { /* child process */
  /* close the unused end of the pipe */
  close(fd[WRITE_END]);

 /* read from the pipe */
 read(fd[READ_END], read_msg, BUFFER_SIZE);
 printf("child read %s\n",read_msg);

 /* close the write end of the pipe */
 close(fd[READ_END]);
}
```

```
}
```

FIFO (Named pipe)

- father-child relationship no longer needed
- **They continue to exist indepently from the processes which created/used them**
- It can be opened by multiple processes for reading or writing
- The following function creates a new fifo:

```
int mkfifo(const char *pathname, mode_t mode)
```

- They can be managed by the function:

```
open(), read(), write() e close()
```

Example

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


int mkfifo(const char *pathname, mode_t mode);
```

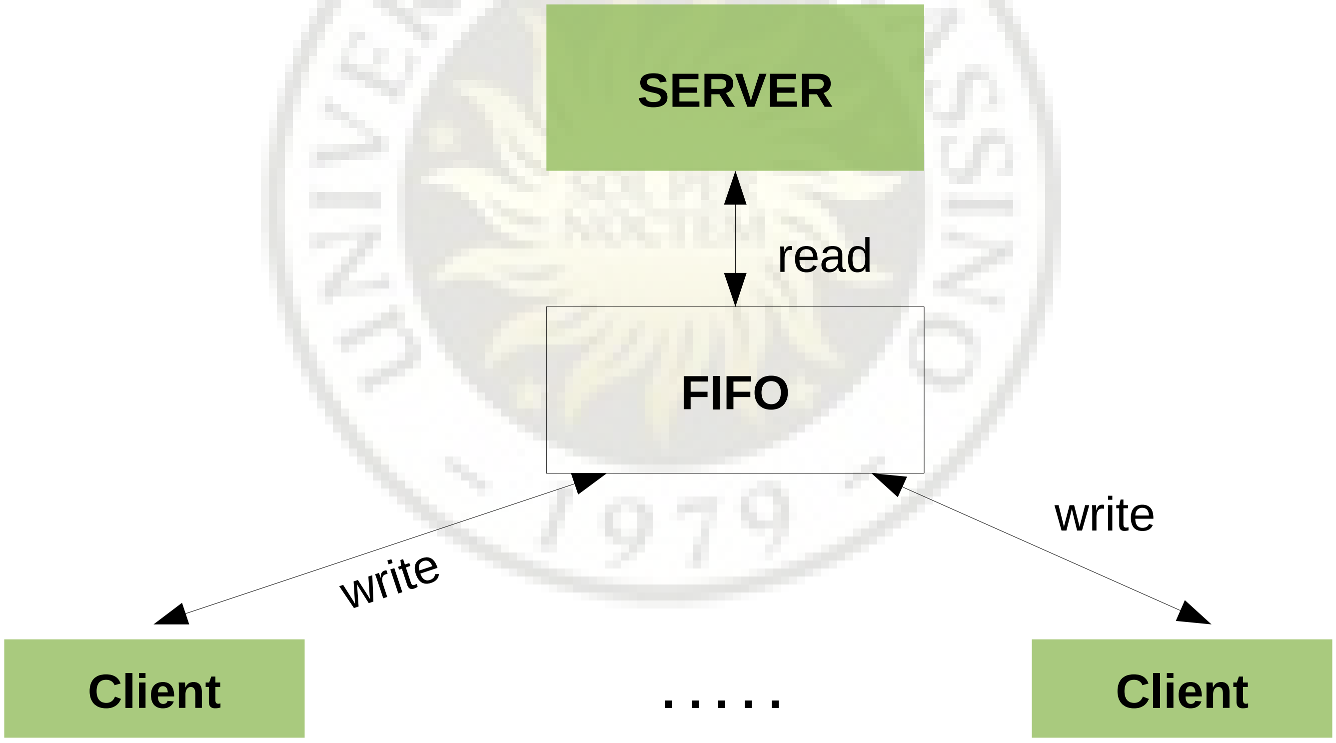
Returns 0 if successfull, -1 otherwise

Remarks

- FIFO is a (special) type of file:
  - Use the functions, open(), read(), write(), etc
  - it is on file system
- as concerns the data access:
  - data can be read only in first-in-first-out order (no random access, no lseek)
  - Data can't be read again or sent back

Example

- Client processes can send request to a server process



FIFO: client

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

int main(){
  int fds, pid;
  char c[5],f[9];
  if((fds=open("FIFO",O_WRONLY))<0){ //Opens the FIFO
    printf("Error: impossible to open the FIFO\n");
    exit(0);
  }
   pid=getpid();

   write(fds,&pid,sizeof(pid)); //send the message to the server
   close(fds); //closes the well-known FIFO
   while(1) //waits to be killed
    ;
}
```

FIFO: server

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

int main(){
  int fd;
  int pidc;

  //CREATES THE FIFO
  if(mkfifo("FIFO",S_IRWXU|S_IRGRP|S_IROTH)<0){
    printf("\nImpossible to create the FIFO\n");
    exit(0);
  }
```

CONTINUE...

FIFO: server

```
  while(1){
    if((fds=open("FIFO",O_RDONLY)) < 0){ //Open the FIFO
      printf("\nError: it is impossible to open the
FIFO\n");
      exit(0);
      }
    If (read(fds,&pidc,sizeof(pidc))<0){ //read from the
FIFO
        printf("Error: message not valid\n");
        return;
      }
      printf("the server read from the FIFO %d\n",pidc);
      kill(pidc,SIGKILL); //kills the client process

      close(fds); //closes the well-known FIFO
  }

    return;
}
```