



**POLITECNICO
DI TORINO**

Inter Process Communication

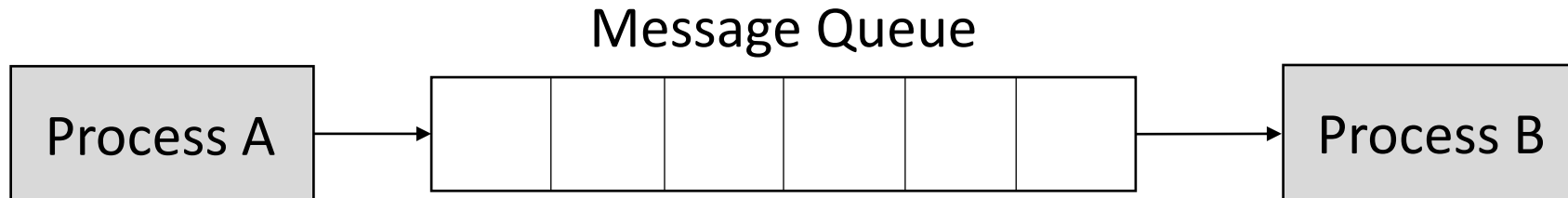
Operating Systems – Sarah Azimi

OUTLINE

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

Message Passing

- Message Passing provides a mechanism for processes to communicate and synchronize their actions without sharing the same address space.
- IPC facility provides two operations:
 - **Send** (message)
 - **Receive** (message)
- If processes P and Q wish to communicate, they need to:
 - Establish a communication link between them.
 - Exchange messages via send/receive functions.



Message Queue

- To perform communication using message queues, following are the steps:
 - Step 1 – Create a message queue or connect to an already existing message queue
 - `msgget()`
 - Step 2- Write into message queue
 - `msgsnd()`
 - Step 3 – Read from the message queue
 - `msgrcv()`
 - Step 4 – Perform control operations on the message queue
 - `msgctl()`

Create a message queue

- To create or allocate a message queue
 - Syntax:

```
int msgget (key_t key, int flag);
```
 - Key is an integer that specifies the queue key, that may be one of:
 - `IPC_PRIVATE`: to create a private message queue
 - Positive integer: To create a publicly accessible message queue.
 - `flag` is used to indicate creation conditions and access permissions. It is bitwise or of flag values. The flag values include these:
 - `IPC_CREAT`: A new queue should be created
 - `IPC_EXCL`: It causes `msgget` to fail if a queue key that is specified already exists.
 - Mode flags: This value is made of 9 bits indicating permissions granted to owner, group and world to control access to segment. Execution bits are ignored.
- This function returns a message queue identifier (`msgid`) on success and -1 in case of failure.

Send a message queue

- To send or append a message into the message queue
 - Syntax:

```
int msgsnd (int msgid, const void *msgp, size_t msgsz, int msgflg);
```
 - `msgid` is the message queue identifier. It is the return value of `msgget` in case of success.
 - `msgp` is the pointer to the message. It is defined in the structure of the following form:

```
Struct msgbuf {  
    long mtype;  
    char mtext [1];  
};
```
 - `msgflag` indicates certain flags such as:
 - `IPC_NOWAIT` which returns immediately when no message is found in queue.
 - `MSG_NOERROR` truncates message text if it is more than `msgsz` byte.
- It return 0 in success and -1 in case of failure.

Receive a message on a queue

- Receiving a message from message queue by calling `msgrcv()`.
 - Syntax:
`int msgrcv(int msqid, void *msgp, size_t msgsz, long msgtype, int msgflg);`
 - `msqid` corresponds to the message queue identifier (the value returned by `msgget()`)
 - `msgp` points to a message received from the sender.
 - `msgsz` specifies the actual size of the message text.

Receive a message on a queue

- Receiving a message from message queue by calling `msgrcv()`.
 - Syntax:
`int msgrcv(int msqid, void *msgp, size_t msgsz, long msgtype, int msgflg);`
- `msgtype` can be used by the receiver for message selection.
 - If `msgtype` is 0 ---> Reads the first message available in a FIFO queue.
 - If `msgtype` is +ve ---> Reads first message on queue whose type equals `msgtype`.
 - If `msgtype` is -ve ---> Reads first message on queue whose type is the lowest value less than or equal to the absolute value of `msgtype`.

Receive a message on a queue

- Receiving a message from message queue by calling `msgrcv()`.
 - Syntax:
`int msgrcv(int msqid, void *msgp, size_t msgsz, long msgtype, int msgflg);`
- The `msgflag` is a bit mask constructed by *ORing* together zero or more of the possible flags (see the man pages):
 - `IPC_NOWAIT`: it returns immediately if no message of the requested type is in the queue. The system call fails with `errno` set to `ENOMSG`.

Receive a message on a queue

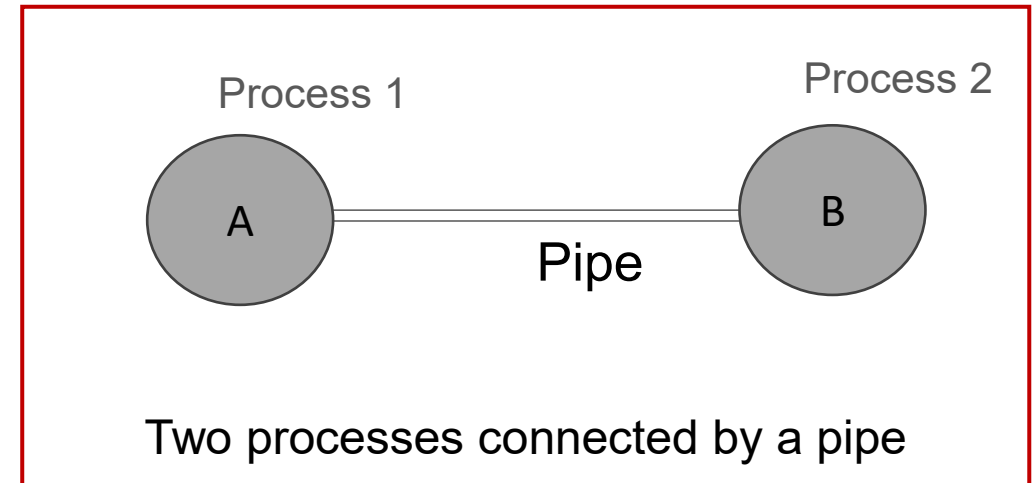
- Receiving a message from message queue by calling `msgrcv()`.
 - Syntax:
`int msgrcv(int msqid, void *msgp, size_t msgsz, long msgtype, int msgflg);`
 - Return value:
 - If successful, `msgrcv` returns the number of bytes in the text of the message.
 - If unsuccessful, it returns -1.

Remove a message on a queue

- To perform control operations on a message queue.
 - Syntax:
`int msgctl (int msgid, int *cmd, struct msqid_ds *buf);`
 - `msgid` corresponds to the message queue identifier (the value returned by `msgget()`)
 - `cmd` is the command to perform the required control operation on the message queue.
 - `IPC_RMID` to removed the message queue immediately. The removal is immediate and any other process still using the message queue will get an error on its next attempted operation on the queue.
 - `buf` is the pointer to the message queue structure named `struct msqid_ds`.

Pipes

- Pipe sets up communication channel between two processes (related or unrelated):
 - Pipes are uni-directional.
 - They can only transfer data in one direction.
 - 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.
 - 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.



Unnamed pipe

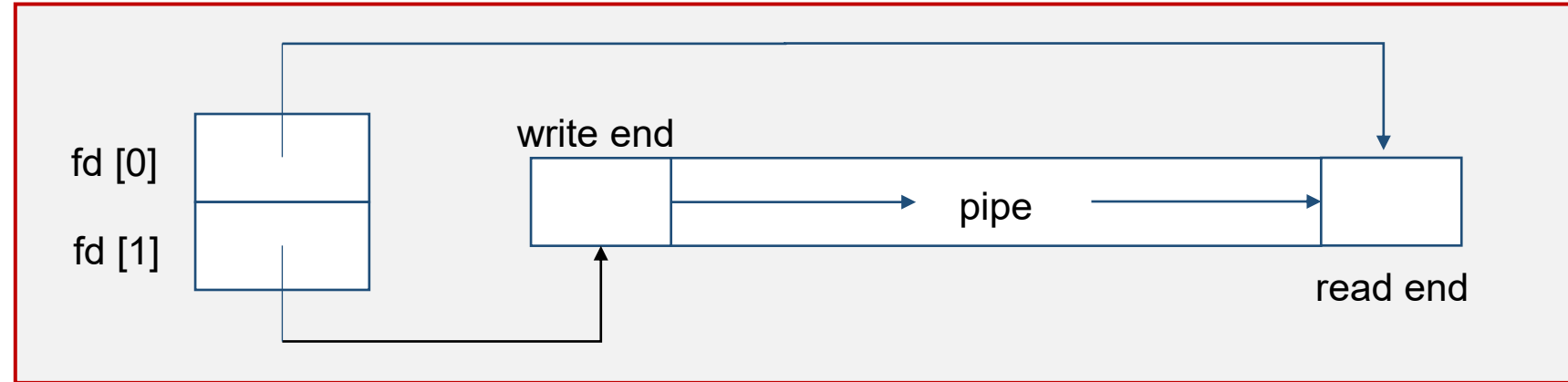
- An unnamed pipe is a unidirectional communications link that automatically buffers its data.

- Syntax:

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

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

```
main()
{
    int fd[2];
    pipe(fd);
    ...
}
```



“fd[2]”:

- 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 at fd [1]

“pipe()” creates an un named pipe and returns two file descriptions: fd [0], fd [1]

- If the kernel cannot allocate enough space for a new pipe, “pipe()” returns a value of -1, otherwise, it returns a value of 0.

Unnamed pipe sequence of events

- The parent process creates an unnamed pipe using “`pipe()`”
- The parent process `forks`
- The processes communicate by using “`write()`” and “`read()`” calls
- Each process `closes` its active pipe descriptor when it is finished.

```
int read (int fd, char *buf, int size)
```

For reading “size” bytes from the files specified by “fd” into the memory location pointed by “buf”

```
int write (int fd, char *buf, int size)
```

For writing the bytes stored in “buf” to the file specified by “fd”

Unnamed pipe Scheme

- The typical sequence of events for a communication is as follows:

“pipeFD[2]”:

- The descriptor associated with the “read” end of the pipe is stored in pipeFD [0].
- The descriptor associated with the “write” end of the pipe is stored at pipeFD [1].

“pipe()” creates an un named pipe and returns two file descriptions: pipeFD [0], pipeFD[1].

```
int write (int fd, char * buf, int size)
```

```
int read (int fd, char * buf, int size)
```

```
# include <sys/wait.h>    /*wait*/
# include <stdio.h>
# include <stdlib.h>       /*exit functions*/
# include <unistd.h>       /*read, write, pipe*/

Int main (){
    int pipeFDs [2];       /*two file descriptors*/

    pipe(pipeFDs);
    pid = fork ();
    If ( pid == 0){ // in the child
        write (pipeFD [1], yourMessage, strlen (yourMessage));
        close (pipeFD [1]);
    }
    else { // in the father
        byteRead = read (pipeFD [0], msg, 100);
        close (pipeFD [0]);
    }
}
```