Processes

Inter-process communication

Stefano Quer and Pietro Laface
Dipartimento di Automatica e Informatica
Politecnico di Torino

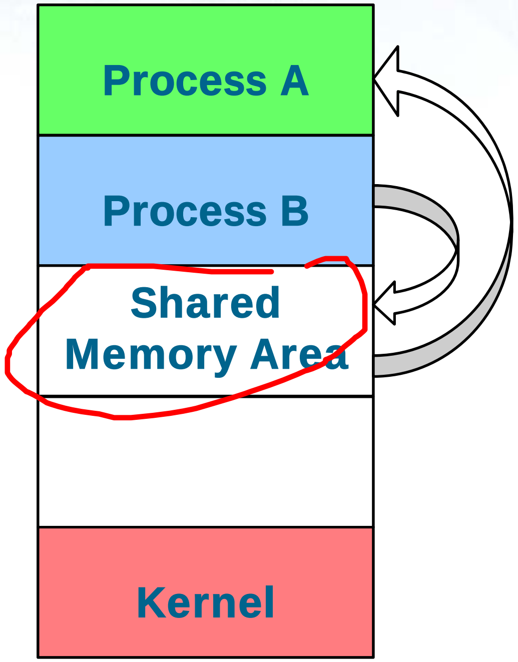
Independent and cooperating processes

- Concurrent processes can be
  - Independent
  - Cooperating
- An **independent** process
  - **Cannot** be influenced by other processes
  - **Cannot** influence other processes
- A set of **cooperating** processes cooperate by sharing data or by exchange of messages
  - Both require appropriate synchronization mechanisms

Inter-Process Communication

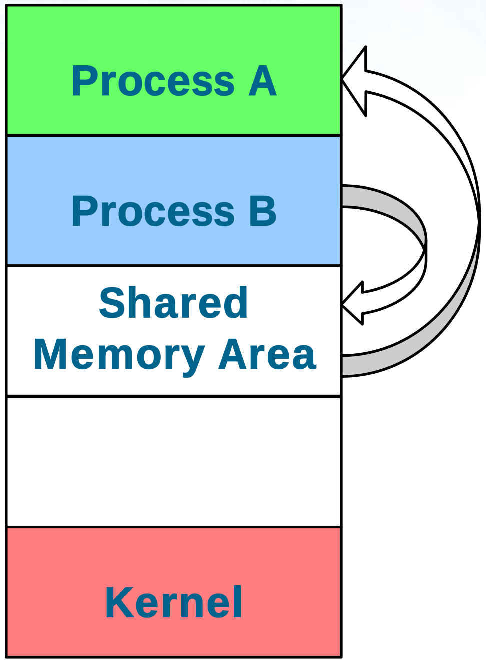
- Information sharing among processes is referred to as **IPC** or **I**nter**P**rocess **C**ommunication
- The main communication models are based on
  - Shared memory
  - Message exchange

Communication models



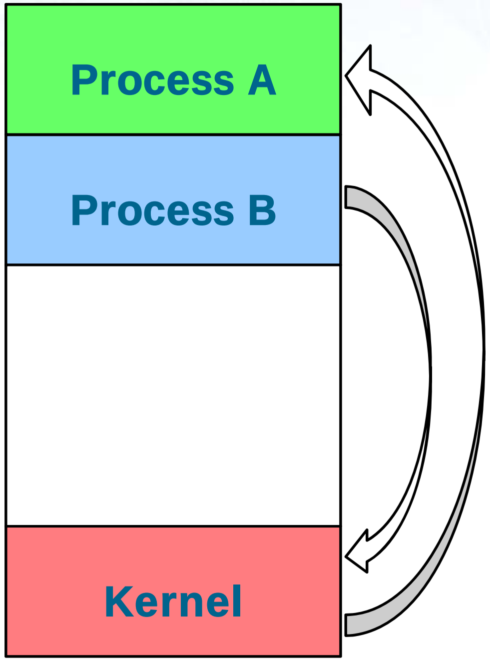
- Shared memory
  - Normally the kernel does not allow a process to access the memory of another process
  - Processes must agree on the access rights and strategies
    - Access rights
    - Access strategies
      - e.g., Producer-consumer with bounded or unbounded buffer

Communication models



- The most common methods for shared buffer use a
  - **File**
    - Sharing the name or the file pointer or descriptor before `fork/exec`
  - **Mapped file**
    - Associates a shared memory region to a file
- These techniques allow sharing a large amount of data

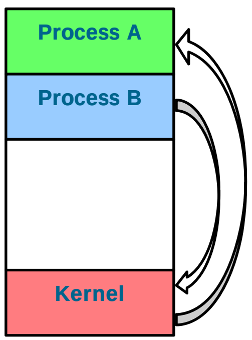
Communication models



- Message exchange
  - Setup of a communication channel
  - Useful for exchanging limited amounts of data
  - Uses system calls
    - which introduce overhead

Communication channels

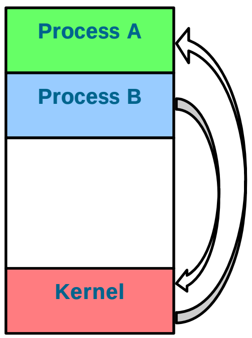
- A communication channel can offer direct or indirect communication
  - Direct
    - Is performed naming the sender or the receiver
      - `send    (to_process,    message)`
      - `receive (from_process, &message)`
  - Indirect
    - Performed through a **mailbox**
      - `send    (mailboxAddress, message)`
      - `receive (mailBoxAddress, &message)`



which model?

Communication channels

- Synchronous or asynchronous synchronization
- Both sending or receiving messages can be
  - Synchronous, i.e., blocking
  - Asynchronous, i.e., non-blocking
- Limited or unlimited capacity queue
  - If the capacity is zero, the channel cannot allow waiting messages (no buffering)
  - If the capacity is limited the sender blocks when the queue is full



Communication channels

- UNIX makes available
  - **Half-duplex pipes**
  - FIFOs
  - Full-duplex pipes
  - Named full-duplex pipes
  - Message queues
  - **Semaphores**
  - Sockets

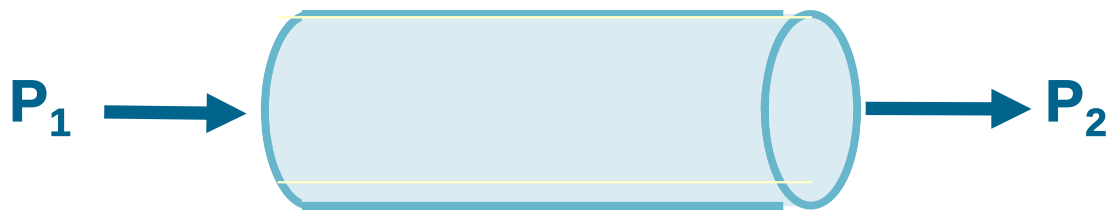
Extensions of the pipes not covered in this course

For process synchronization

Network process communication.

Pipes

- Allow creating a **data stream among** processes
  - The user interface to a pipe is similar to file access
  - A pipe is accessed by means of two descriptors (integers), one for each end of the pipe
  - A process (P_1) writes to an end of the pipe, another process (P_2) reads from the other end



Pipes

Simplex, for synchronization problems

- Historically, they have been
  - **half-duplex**
    - Data can flow in both directions (from P_1 to P_2 or from P_2 to P_1), but **not** at the same time
    - Full-duplex models have been proposed more recently, but they have limited portability
  - A pipe can be used for communication among a parent and its offspring, or among processes with a **common ancestor**

Terminology:
Simplex: Mono-directional
Half-Duplex: One-way, or bidirectional, but alternate (walkie-talkie)
Full-Duplex: Bidirectional (telephone)

pipe system call

```
#include <unistd.h>

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

- System call **pipe** creates a pipe
- It returns **two** file descriptors in vector **fileDescr**
  - fileDescr[0]: Typically used for reading
    fileDesrc[1]: Typically used for writing
  - The input stream written on fileDescr[1] corresponds to the output stream read on fileDescr[0]

pipe system call

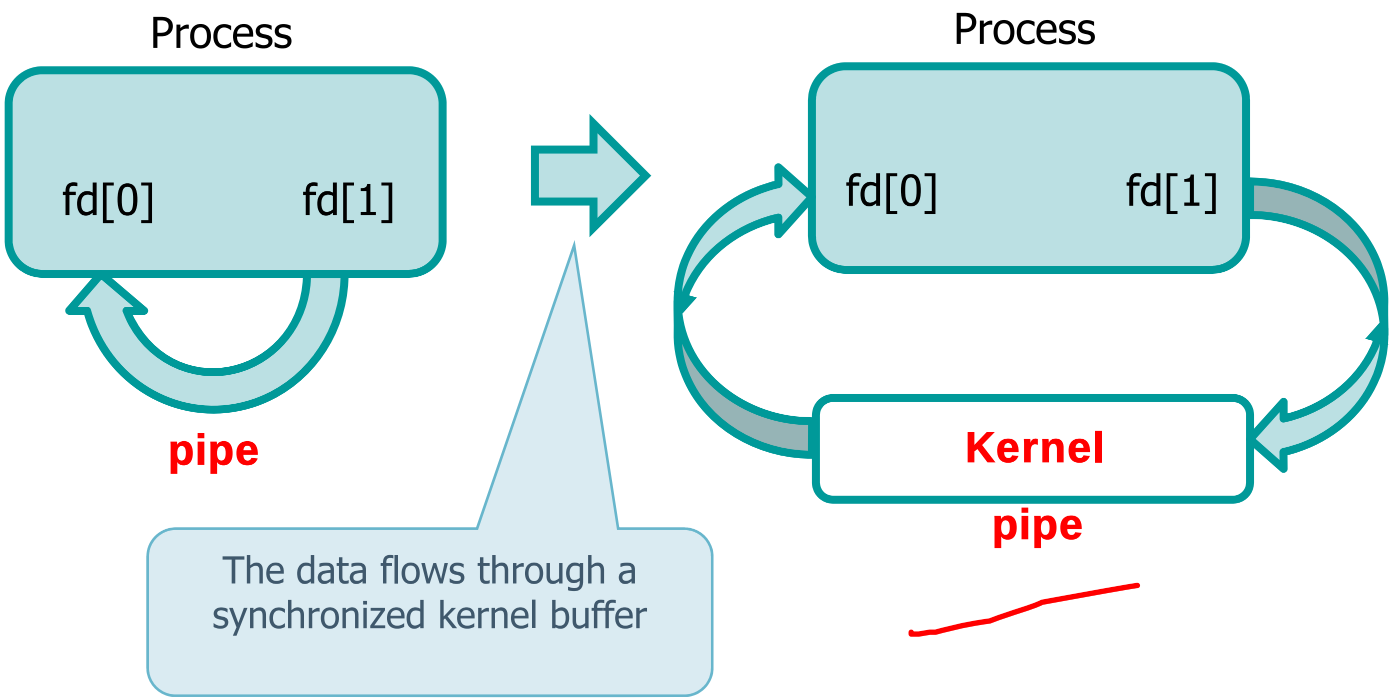
```
#include <unistd.h>

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

- Return value
  - 0 on success
  - -1 on error

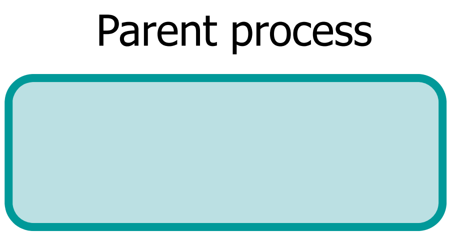
pipe system call

- Using a pipe inside a process is possible but not much useful



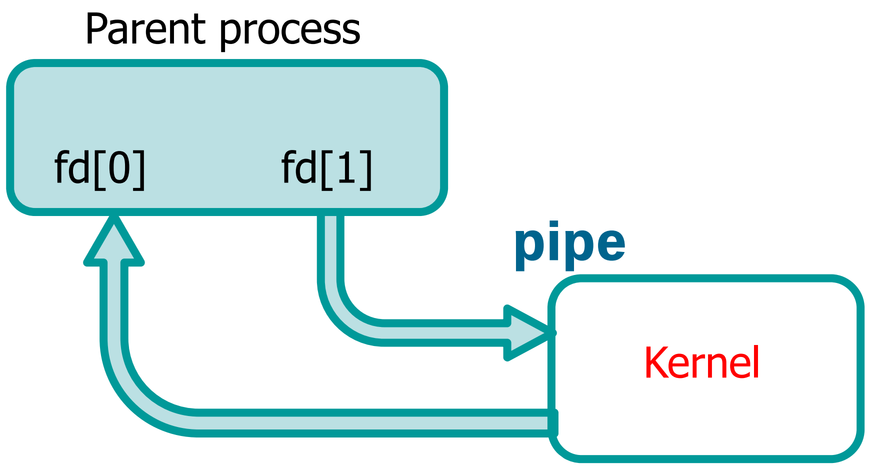
pipe system call

- A pipe typically allows a parent and a child to communicate
- Parent must fork **after** creating the pipe



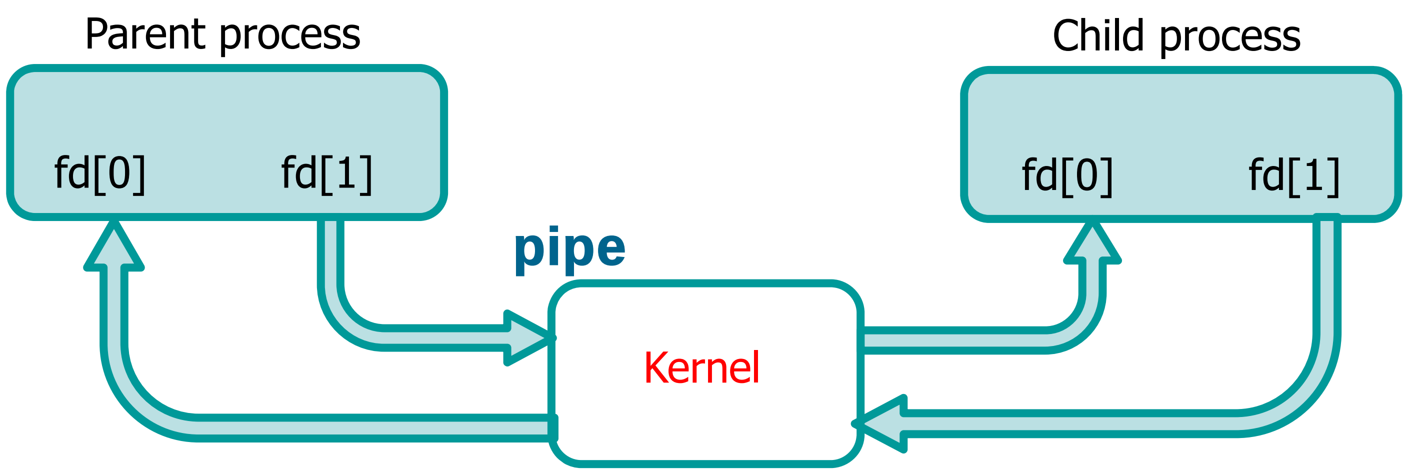
pipe system call

- A pipe typically allows communicating a parent and a child
- Parent must fork **after** creating the pipe



pipe system call

- The parent process creates a pipe
- Performs a **`fork`**
- The child process **inherits** the file descriptors



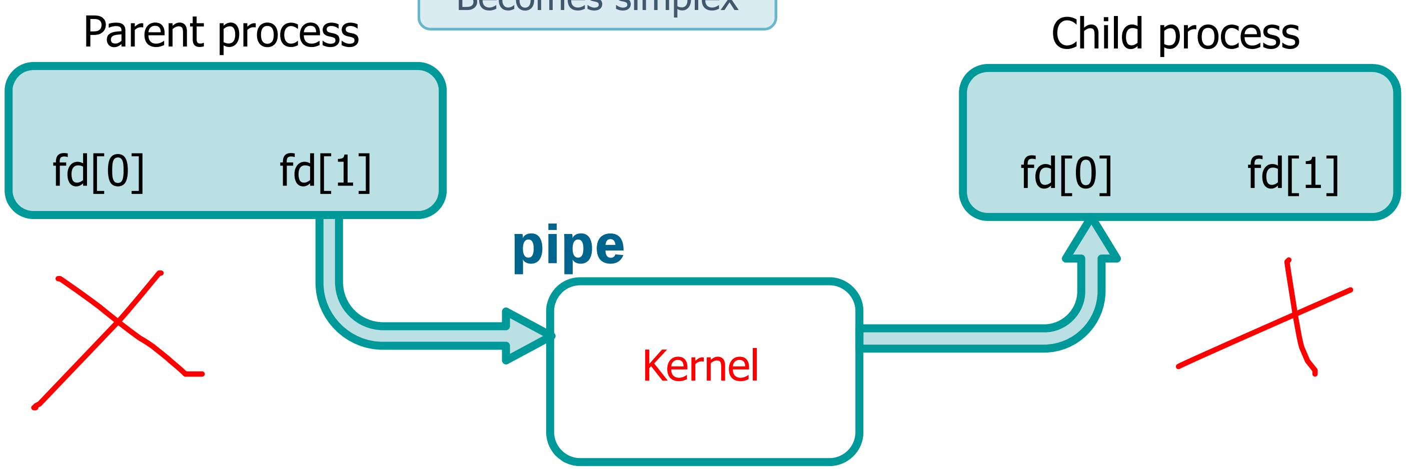
pipe system call

- One of the two processes (e.g., the parent) writes on pipe, the other (e.g., the child) reads from pipe

Half-duplex mode

- The descriptor that is not used by a process should be closed

Becomes simplex



Pipe I/O

- R/W on pipes do not differ to R/W on files
  - Use `read` and `write` system calls
  - It is possible to have more than one reader and writer on a pipe, but
    - Data can be interlaced using more than one writer
    - Using more readers, it is undetermined which reader will read the next data from the pipe

Pipe I/O

System call read

- Blocks the process if the pipe is empty (**it is blocking**)
- If the pipe contains less bytes than the ones specified as argument of the read, it **returns only the bytes available on the pipe**
- If all file descriptors referring to the write-end of a pipe have been closed, then an attempt to read from the pipe will see end-of-file (read returns 0)

Pipe I/O

- System call write
  - Blocks the process if the pipe is full (**it is blocking**)
  - The dimension of the pipe depends on the architecture and implementation
    - Constant **`PIPE_BUF`** defines the number of bytes that can be written atomically on a pipe
    - Standard value of **`PIPE_BUF`** is **`4096`** on Linux
  - If all file descriptors referring to the read-end of a pipe have been closed, then a write to the pipe will cause a SIGPIPE signal to be generated for the calling process

Example

- Create a pipe shared between parent and child
- Transfer a single character from parent to child
- Logical flow
  - Pipe create
  - Process fork
  - Close the unused-ends of the pipe
  - **`read`** and **`write`** operations at the two pipe ends

Example

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

int main () {
  int n;
  int file[2];
  char cW = ‘x’;
  char cr;
  pid_t pid;
  if (pipe(file) == 0) {
    pid = fork ();
    if (pid == -1) {
      fprintf(stderr, "Fork failure");
      exit(EXIT_FAILURE);
    }
```

Example

```
if (pid == 0) {
    // Child reads
    close (file[1]);
    n = read (file[0], &cR, 1);
    printf("Read %d bytes: %c\n", n, cR);
    exit(EXIT_SUCCESS);
  } else {
    // Parent writes
    close (file[0]);
    n = write (file[1], &cW, 1);
    printf ("Wrote %d bytes: %c\n", n, cW);
  }
}
exit(EXIT_SUCCESS);
}
```

Close unused end (good practice)

close write terminal

Child reads

close read terminal

Parent writes

The two process synchronize because read and write are possibly blocking.

More complex data communication requires a communication protocol

Example

- Which is the dimension of a pipe?
- Since `write` is a blocking system call, we can continue to write a byte to the pipe until the process is blocked because the pipe is full

Example

```
...
#define SIZE  512*1024

int fd[2];

static void signalHandler (int signo) { ... }     ???

int main () {
...
int i, n, nR, nW;
char c = '1';
setbuf (stdout, 0);

...

// Install Signal Handler
... signal (SIGUSR1, signalHandler) ...
```

Example

```
pipe(fd);
n = 0;
if (fork()) {
  fprintf (stdout, "\nParent PID=%d\n", getpid());
  sleep (1);
  for (i=0; i<SIZE; i++) {
    nW = write (fd[1], &c, 1);
    n = n + nW;
    fprintf (stdout, "W %d\r", n);
  }
} else {
  fprintf (stdout, "Child  PID=%d\n", getpid());
  sleep (10);
  for (i=0; i<SIZE; i++) {
    nR = read (fd[0], &c, 1);
    n = n + nR;
    fprintf (stdout, "\t\t\t\tR %d\r", n);
  }
}
```

Parent writes a byte at a time

\r = CR = Carriage Return (not Line Feed)

The child reads **after 10 seconds**

Example

```
> ./pgrm
Parent PID=2272
Child  PID=2273
W 0

...

W 65536

...

W 65536 R 0

...

W 524288 R 524288
```

The number of written bytes increases up to the dimension of the pipe

When the pipe is full, write blocks the parent

After 10 seconds the child begins to read the pipe, consuming its data

R & W are concurrent, the processes terminate after SIZE writes and reads

Example

- What happens if a pipe is not used according to the half-duplex protocol?
  - It is possible to change pipe the ends for the read and write operations?
  - It is possible to have multiple readers and writers?
- The result is undefined, but it is possible to obtain corrected results for the first case

Example

Program receives a string in argv[1]

```
int fd[2];
setbuf (stdout, 0);
pipe (fd);
if (fork()!=0) {
  while (1) {
    if (strcmp(argv[1],"P")==0||strcmp(argv[1],"PC")==0) {
      c = ‘P';
      fprintf (stdout, “Parent writes %c\n", c);
      write (fd[1], &c, 1);
    }
    sleep (2);
    if (strcmp(argv[1],"C")==0||strcmp(argv[1],"PC")==0) {
      read (fd[0], &c, 1);
      fprintf (stdout, “Parent reads %c\n", c);
    }
    sleep (2);
  }
  wait ((int *) 0);
}
```

If argv[1] is “P"
the parent writes only
and the child reads only

If argv[1] is “C"
the parent reads only
and the child writes only

Example

```
} else {
  while (1) {
    if (strcmp(argv[1],“P")==0||strcmp(argv[1],“PC")==0) {
      read (fd[0], &c, 1);
      fprintf (stdout, "Child reads %c\n", c);
    }
    sleep (2);
    if (strcmp(argv[1],“P")==0||strcmp(argv[1],“PC")==0) {
      c = 'C';
      fprintf (stdout, "Child writes %c\n", c);
      write (fd[1], &c, 1);
    }
    sleep (2);
  }
  exit (0);
}
```

If argv[1] is “PC"
parent and child
alternate write operations

Example

```
> ./pgrm P
Parent writes P
Child reads P
...
^C
> ./pgrm C
Child Write C
Father Read C
...
^C
> ./pgrm PC
Parent writes P
Child reads P
Child writes C
Parent reads C
...
^C
```

Only parent writes

Only child writes

Parent and child alternate writing Every 2 secs

How they would alternate without sleep?