Sesión 2: Procesos

Programación para Sistemas

Ángel Herranz

Curso 2023-2024

Objetivo: comunicarse con los procesos

- ¿Cómo ejecutamos nuestros programas (en ejecución = proceso)?

Objetivo: comunicarse con los procesos

- ¿Cómo ejecutamos nuestros programas (en ejecución = proceso)?

línea de comandos

Objetivo: comunicarse con los procesos

- ¿Cómo ejecutamos nuestros programas (en ejecución = proceso)?

  línea de comandos

- ¿Cómo podemos *enviar* información a nuestro programa?
  - Argumentos en la línea de comandos
  - Entrada estándar: `stdin` (ficheros en general)
  - Variables de entorno

Objetivo: comunicarse con los procesos

- ¿Cómo ejecutamos nuestros programas (en ejecución = proceso)?

  línea de comandos

- ¿Cómo podemos *enviar* información a nuestro programa?
  - Argumentos en la línea de comandos
  - Entrada estándar: `stdin` (ficheros en general)
  - Variables de entorno
- ¿Cómo podemos *recibir* información de nuestro programa?
  - Salida estándar: `stdout` (ficheros en general)
  - Y salida de error: `stderr`
  - Estado de terminación: **`exit`** o **`return`** en main

Diagrama de comunicación de un proceso

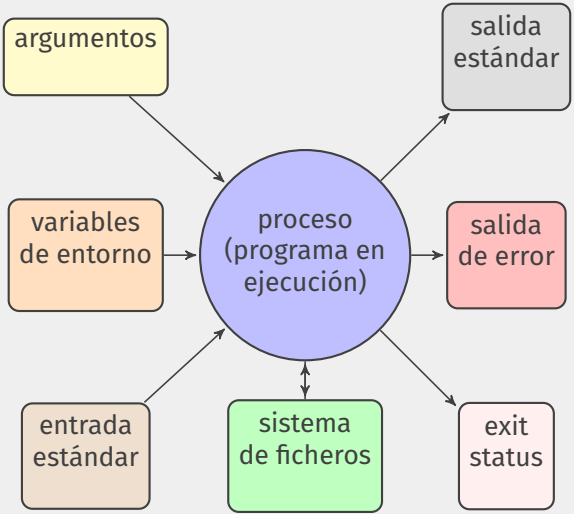
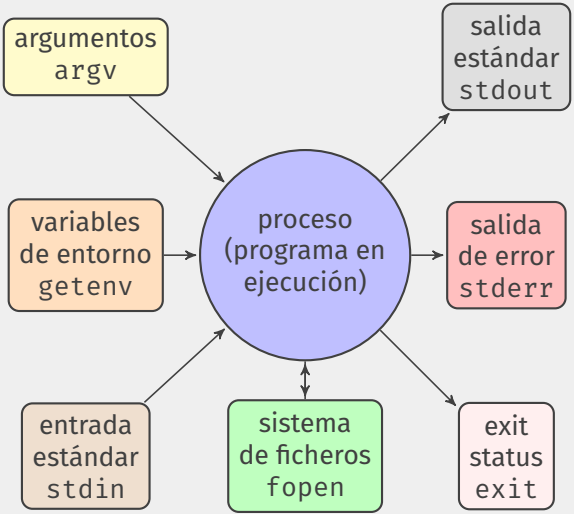


Diagrama de comunicación de un proceso



Otras formas de comunicar con los procesos

- *Pipes* (`pipe`, se manejan como ficheros)
- *Signals*: **`kill`** (ejecutar **`kill`** `-l`)
- *Sockets* (`socket`)
- Memoria compartida (`mmap`, basado en fichero)

Un programa en C

- Aunque no vamos a empezar con C hasta la semana 5 os voy a mostrar un programa en C que hace uso de todo lo anterior:

  Escribir un programa en C que se llame `comm` (`comm.c`) que (1) escriba en la salida de error todos sus argumentos, (2) lea de la entrada estándar una línea y la escriba en la salida estándar, (3) abra un fichero `comm.txt` y escriba la primera línea del fichero en la salidad estándar, (4) lea la variable de entorno `MICOMM` y la imprima en la salida estándar, y (5) termine con "exit status" igual el número de argumentos.

- Usaremos el editor de texto `nano` para crear el fichero `comm.c`
- Usaremos el editor de texto `nano` para crear el fichero `comm.txt`
- El compilador de C `gcc` para compilar el programa `comm`
- Y finalmente ejecutaremos el programa `comm` con varios argumentos

Un programa en C: paso a paso

- Crear un directorio `02-bash_basics`: “`mkdir 02-bash_basics`”
- entrar en dicho directorio: “**`cd`** `02-bash_basics`”
- Crear el fichero `comm.txt` con varias líneas: “`nano comm.txt`”
- Crear el fichero `comm.c` con el programa: “`nano comm.c`”
- Compilar el programa: “`gcc -o comm comm.c`”
- Crear la variable de entorno: “**`export`** `MICOMM=prueba`”
- Ejecutar el programa, por ejemplo así: “`./comm 1 DOS III 100`”
- Observar y entender todo lo que está pasando
- En otro terminal listar procesos: “`ps -a`”
- Comprobar el *exit status* al terminar: “`echo $?`”

“Ya se programar en C”

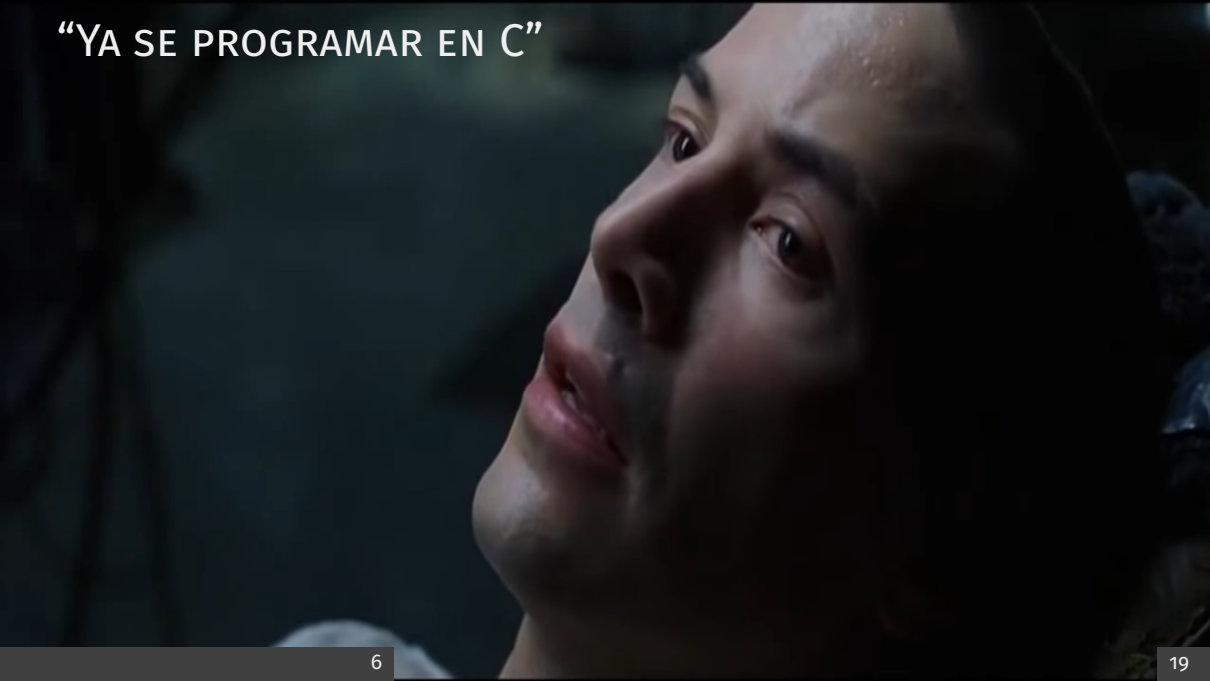
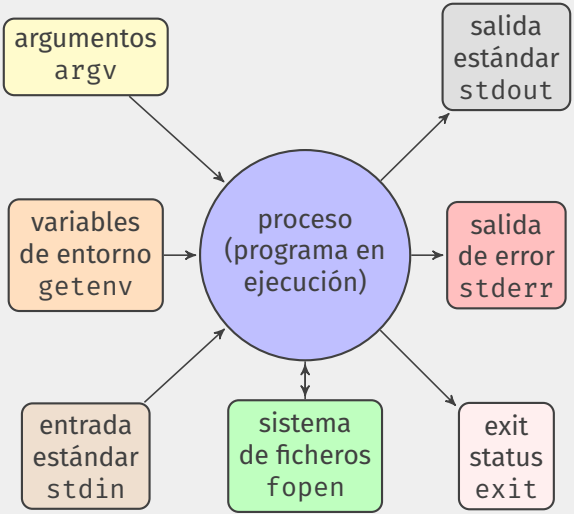


Diagrama de comunicación de un proceso



Comandos y argumentos

```
$ comando [arg1 [arg2 [arg3 [arg4 [...]]]]]
```

Comandos y argumentos

```
$ comando [arg1 [arg2 [arg3 [arg4 [...]]]]]

$ ls        -al    /      ~
$ comm      1      DOS    III    100
```

- Los argumentos del programa son los argumentos de la función `main`
- `argc`: número de argumentos incluyendo el comando
- `argv`: array con los argumentos
- Esto pasa en todos los lenguajes de programación

Variables de entorno

- Las variables de entorno se establecen antes de ejecutar el comando:

```
$ variable=valor mandato [arg1 [arg2 [...]]]
```

- Y pueden ser exploradas en el programa (función `getenv`)

```
$ MICOMM=prueba ./comm 1 DOS III 100
```

⚠ Sin espacios alrededor de `=`

Equivalente a

```
$ export MICOMM=prueba
$ ./comm 1 DOS III 100
$ unset MICOMM
```

Variables de entorno

- Las variables de entorno se establecen antes de ejecutar el comando:

```
$ variable=valor mandato [arg1 [arg2 [...]]]
```

- Y pueden ser exploradas en el programa (función `getenv`)

```
$ MICOMM=prueba ./comm 1 DOS III 100
```

⚠ Sin espacios alrededor de `=`

Equivalente a

```
$ export MICOMM=prueba
$ ./comm 1 DOS III 100
$ unset MICOMM
```

Y luego en C...

```
#include <stdlib.c>
...
char *s = getenv("MICOMM");
```

El interés del *exit status* (código de terminación)

- Ejecutar y comprobar cómo de mal ha terminado

El interés del *exit status* (código de terminación)

- Ejecutar y comprobar cómo de mal ha terminado

```
$ ./comm 1 DOS III 1000
...
$ echo $?
4
$ ./comm
...
$ echo $?
0
```

- Y esto, ¿Para qué sirve?

El interés del *exit status* (código de terminación)

- Ejecutar y comprobar cómo de mal ha terminado

```
$ ./comm 1 DOS III 1000
...
$ echo $?
4
$ ./comm
...
$ echo $?
0
```

- Y esto, ¿Para qué sirve? (interesante la sintaxis del **if**)

```
$ if ./comm; then echo BIEN; else echo MAL; fi
BIEN
```

Comunicación por Ficheros

- En Linux todos los dispositivos son ficheros
- Entrada estándar `stdin`: fichero especial de entrada (por defecto *conectado* al teclado)
- No confundir la entrada estándar con los argumentos
- Salida estándar `stdout`: fichero especial de salida (por defecto *conectado* a la *pantalla*, con confundir con)
- Salida de error `stderr`: fichero especial de salida en el que el programa escribir mensajes de error (por defecto *conectado* a la *pantalla*)
- Esos ficheros se manejan como cualquier otro fichero previamente abierto
- No confundir la salida estándar o la de error con el *exit status*

¿Qué hay detrás de los mandatos conocidos?

- El comando `ls` es realmente un programa: `/usr/bin/ls`
- Para saber qué programa es:

```
which ls
```

- Para saber qué hace:

```
man ls
```

- ¿Y **cd**?

¿`which` **`cd`**? ¿`man` **`cd`**?

¿Qué hay detrás de los mandatos conocidos?

- El comando `ls` es realmente un programa: `/usr/bin/ls`
- Para saber qué programa es:

  `which ls`
- Para saber qué hace:

  `man ls`
- ¿Y **cd**?

  ¿`which` **cd**? ¿`man` **cd**?
- ¿Por qué no hay programa ni manual de **cd**?

¿Qué hay detrás de los mandatos conocidos?

- El comando `ls` es realmente un programa: `/usr/bin/ls`
- Para saber qué programa es:

```
which ls
```

- Para saber qué hace:

```
man ls
```

- ¿Y **cd**?

¿`which` **cd**? ¿`man` **cd**?

- ¿Por qué no hay programa ni manual de **cd**?
- Los mandatos de Bash pueden ser programas o *built in commands*:

```
man bash
```

y buscar “SHELL BUILTIN COMMANDS”

¿Dónde están los programas?

- Los programas están en el sistema de ficheros (“`which nano`”)
- ¿Cómo los busca Bash?

Variable de entorno PATH

[1]Separadas por el caracter “`:`” en Unix

¿Dónde están los programas?

- Los programas están en el sistema de ficheros (“`which nano`”)
- ¿Cómo los busca Bash?

Variable de entorno PATH

- Un path es una lista de directorios[1]
- Mira y cambia el PATH

```
$ echo $PATH
$ ls
$ which ls
```

```
$ PATH=
$ echo $PATH
$ ls
$ which ls
```

[1] Separadas por el caracter “:” en Unix

Más sobre las variables de entorno

- Los programas usan variables de entorno, algunas son comunes:

  `PATH`, `PS1`, `USER`, `SHELL`, `PWD`, `HOSTNAME`, `LANG`, `EDITOR`, etc.

- Pero cada programador puede *definir* las suyas:

  `JAVA_HOME`, `CLASSPATH`, `MICOMM`

- Todo lo que se haga con ellas se pierde entre sesiones

Ficheros de inicialización de Bash

- Las variables de entorno se establecen al arrancar Bash aprovechando los ficheros de inicialización
- `/etc/profile` *The systemwide initialization file, executed for login shells*
- `/etc/bash.bashrc` *The systemwide per-interactive-shell startup file*
- `~/.bash_profile` *The personal initialization file, executed for login shells*
- `~/.bashrc` *The individual per-interactive-shell startup file*
- `~/.bash_logout` *The individual login shell cleanup file, executed when a login shell exits*
- `/etc/bash.logout` *The systemwide login shell cleanup file, executed when a login shell exits*

Salida estándar y redirección I

- ¿Qué hace `echo`? (`man echo`)

Salida estándar y redirección I

- ¿Qué hace `echo`? (`man echo`)
- ¿Te atreves a programar `echo` en C?

Salida estándar y redirección I

- ¿Qué hace `echo`? (`man echo`)
- ¿Te atreves a programar `echo` en C?
- Por defecto la salida estándar está dirigida a “la pantalla” (terminal)
- Pero la salida estándar se puede redirigir a un fichero:

```
$ echo Fíjate en los    espacios
$ echo "En un lugar de la mancha..." > miquijote.txt
```

Salida estándar y redirección I

- ¿Qué hace `echo`? (`man echo`)
- ¿Te atreves a programar `echo` en C?
- Por defecto la salida estándar está dirigida a “la pantalla” (terminal)
- Pero la salida estándar se puede redirigir a un fichero:

```
$ echo Fíjate en los    espacios
$ echo "En un lugar de la mancha..." > miquijote.txt
```

- ¿Qué es `echo`? ¿Dónde está? ¿Por qué aparece en negrita en estas transparencias? ¿Has probado `which`? ¿Has mirado en `man bash`?

Salida estándar y redirección II

- Descargar El Quijote en texto plano

  `https://babel.upm.es/~angel/teaching/pps/quijote.txt`

Salida estándar y redirección II

- Descargar El Quijote en texto plano

  `https://babel.upm.es/~angel/teaching/pps/quijote.txt`

- ¿Qué hace `cat`? (`man cat`)

Salida estándar y redirección II

- Descargar El Quijote en texto plano

  `https://babel.upm.es/~angel/teaching/pps/quijote.txt`

  - ¿Qué hace `cat`? (`man cat`)
- Ejecutar y *diseccionar*

```
$ cat quijote.txt
$ cat
```

  - ¿Qué ocurre?

Salida estándar y redirección II

- Descargar El Quijote en texto plano

  `https://babel.upm.es/~angel/teaching/pps/quijote.txt`

- ¿Qué hace `cat`? (`man cat`)
- Ejecutar y *diseccionar*

```
$ cat quijote.txt
$ cat
```

- ¿Qué ocurre? Prueba a escribir

```
Los animales son felices mientras
tengan salud y suficiente comida.
Ctrl-d
```

Entrada estándar y salida estándar y redirección

Ejecutar y *diseccionar*

```
$ cat < quijote.txt
$ cat > la_conquista.txt
Los animales son felices mientras
tengan salud y suficiente comida.
Ctrl-d
$ cat quijote.txt la_conquista.txt
$ cat quijote.txt la_conquista.txt > dos_libros.txt
```

Conectando salida estándar y entrada estándar

`man grep`, `man wc`

Conectando salida estándar y entrada estándar

- `man grep`, `man wc`
- Jugamos con El Quijote:
  - ¿Cuántas líneas tiene el fichero descargado?
  - Buscar líneas con “Sancho”
  - Contar número de líneas con “Sáncho”:

    `https://app.wooclap.com/IDIYRB`

Conectando salida estándar y entrada estándar

- `man grep`, `man wc`
- Jugamos con El Quijote:
  - ¿Cuántas líneas tiene el fichero descargado?
  - Buscar líneas con “Sancho”
  - Contar número de líneas con “Sáncho”:

    `https://app.wooclap.com/IDIYRB`
- ¿Cuántas palabras diferentes hay en el fichero `quijote.txt`?

  `https://app.wooclap.com/IDIYRB`
  - Pista 1: puedes usar los comandos `sed`, `sort`, `uniq`
  - Pista 2: `sed 's/ /\n/g' quijote.txt`
  - Pista 3: `grep -v '^$'`