



**Facultad de Ingeniería en  
Electricidad y Computación**

# Programación de Sistemas

## CCPG1051

---

Federico Domínguez, PhD.

Unidad 5 – Sesión 2: Llamadas I/O en Linux

# Agenda

---

Operaciones bitwise

Llamadas I/O

# Manipulación de bits

---

En C es posible manipular directamente los bits en una variable usando operaciones a nivel de bits conocidas como *bitwise operations*.

Estas operaciones trabajan directamente sobre la representación binaria de la variable.

Símbolo	Operador
&	bitwise AND
	bitwise inclusive OR
^	bitwise XOR (eXclusive OR)
<<	left shift
>>	right shift
~	bitwise NOT (one's complement) (unary)

# Manipulación de bits

---

Las operaciones booleanas a nivel de bits se aplican sobre cada bit del valor binario de una variable.

Ejemplo: Si  $A = 0xC8$  (200) y  $B = 0xB8$  (184) entonces  $A \& B = 0x88$  (136)

```
11001000
& 10111000
-----
= 10001000
```

# Manipulación de bits

---

Las operaciones booleanas a nivel de bits se aplican sobre cada bit del valor binario de una variable.

Ejemplo: Si  $A = 0xC8$  (200) y  $B = 0xB8$  (184) entonces  $A | B = 0xF8$  (248)

```
  11001000
| 10111000
-----
= 11111000
```

# Manipulación de bits

---

Las operaciones booleanas a nivel de bits se aplican sobre cada bit del valor binario de una variable.

Ejemplo: Si  $A = 0xC8$  (200) y  $B = 0xB8$  (184) entonces  $A \wedge B = 0x70$  (112)

```
  11001000
^ 10111000
-----
= 01110000
```

# Manipulación de bits

---

Las operaciones booleanas a nivel de bits se aplican sobre cada bit del valor binario de una variable.

Ejemplo: Si  $A = 0xC8$  (200) entonces  $\sim A = 0x37$  (55)

$\sim 11001000 = 00110111$

# Manipulación de bits

---

## Shifts o desplazamiento de bits

Los operadores `<<`, `>>` desplazan los bits en una variable. El operador `<<` o *left shift* simplemente desplaza todos los bits a la izquierda, reemplazando los menos significativos con cero. Los bits que se desbordan son eliminados.

### Ejemplo:

Si  $A = 0xC8$  (200) entonces  $A \ll 2$  es DEPENDE!

$A \ll 2$  11001000  $\rightarrow$  00100000 (32) si A es tipo *char*

$A \ll 2$  0000000011001000  $\rightarrow$  0000001100100000 (800) si A es tipo *short*

$A \ll 3$  0000000011001000  $\rightarrow$  0000011001000000 (1600) si A es tipo *short*



# Manipulación de bits

---

## Shifts o desplazamiento de bits

El operador `>>` o *right shift* es un poco más complicado, el operador desplaza todos los bits a la derecha, eliminando los menos significativos y reemplazando los más significativos con 1 o 0 dependiendo del tipo de datos. Si el tipo de datos es *unsigned*, entonces llena los bits más significativos con el valor de 0, de lo contrario los llena con el valor del bit más significativo.

### Ejemplo:

Si  $A = 0xC8$  (200) entonces  $A \gg 2$  es DEPENDE!

$A \gg 2$  11001000  $\rightarrow$  00110010 (50) si A es tipo *unsigned char* (logical shift)

$A \gg 2$  11001000  $\rightarrow$  11110010 (-14) si A es tipo *char* (arithmetic shift)

$A \gg 3$  0000000011001000  $\rightarrow$  0000000000011001 (25) si A es tipo *short*

# Ejercicio

---

```
/*
```

```
* Retorna verdadero si el bit de "valor" en la posición "pos" es 1. La  
función asume que el primer bit es posición 0.
```

```
*/
```

```
bool isBitSet(unsigned char valor, int pos);
```

# Ejercicio

---

```
/*  
 * Retorna verdadero si el bit de "valor" en la posición "pos" es 1. La  
 * función asume que el primer bit es posición 0.  
 */  
bool isBitSet(unsigned char valor, int pos);
```

<https://repl.it/join/qttquntw-fexadom>

# Llamadas I/O en sistemas UNIX/LINUX

---

Input/Output es el proceso de copiar datos entre la memoria y dispositivos externos como discos duros, terminales y redes.

En C, la librería estándar I/O provee funciones *fprintf* y *fscanf* que permiten I/O entre archivos y buffers.

C++ provee los operadores “<<” y “>>” para operaciones similares.

Estas funciones son implementadas usando funciones I/O que llaman directamente al kernel.

En la gran mayoría de los casos es preferible usar las funciones de alto nivel como *fprintf* o “>>”.

En ciertos casos, es necesario usar directamente las funciones del sistema: drivers, comunicación con sockets y otros.

# En sistemas UNIX, un archivo es una secuencia de bytes y todos los dispositivos son modelados como archivos.

---

La filosofía de UNIX “todo es un archivo” permite el uso de una única interface de bajo nivel para acceder a todo tipo de dispositivos, conocida como UNIX I/O.

**Abrir archivos:** Una aplicación notifica al *kernel* que tiene la intención de acceder un archivo con la llamada de sistema *open*.

**Cerrar archivos:** Una aplicación debe notificar al *kernel* que libere los recursos de memoria relacionados a la gestión de un archivo con la llamada *close*.

**Descriptor de archivo:** Una llamada a *open* retorna un descriptor de archivo, un entero no negativo.

**Posición de archivo:** El *kernel* mantiene un puntero de posición de archivo para cada archivo abierto. Es un offset del principio del archivo.

**Lectura:** Una llamada a *read* copia *n* bytes de un archivo, empezando desde la posición de archivo, hacia un buffer en memoria.

**Escribir:** Una llamada a *write* copia *n* bytes desde un buffer en memoria hacia un archivo, empezando desde la posición de archivo.

Una aplicación notifica al **kernel** que tiene la intención de acceder a un archivo con la llamada de sistema *open*.

---

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

```
int open(char *filename, int flags, mode_t mode);
```

Returns: new file descriptor if OK, -1 on error

# La llamada *open* usa *bit masks* para configurar los modos de acceso.

---

- `O_RDONLY`: Reading only
- `O_WRONLY`: Writing only
- `O_RDWR`: Reading and writing
- `O_CREAT`: If the file doesn't exist, then create a *truncated* (empty) version of it.
- `O_TRUNC`: If the file already exists, then truncate it.
- `O_APPEND`: Before each write operation, set the file position to the end of the file.

```
fd = Open("foo.txt", O_WRONLY|O_APPEND, 0);
```

Cuando una aplicación no va a acceder más a un archivo notifica al **kernel** con la llamada de sistema *close*.

---

```
#include <unistd.h>
```

```
int close(int fd);
```

Returns: zero if OK, -1 on error



Las llamadas *read* y *write* copian *n* bytes entre memoria y el archivo (o viceversa) retornando el número de bytes copiado.

---

```
#include <unistd.h>
```

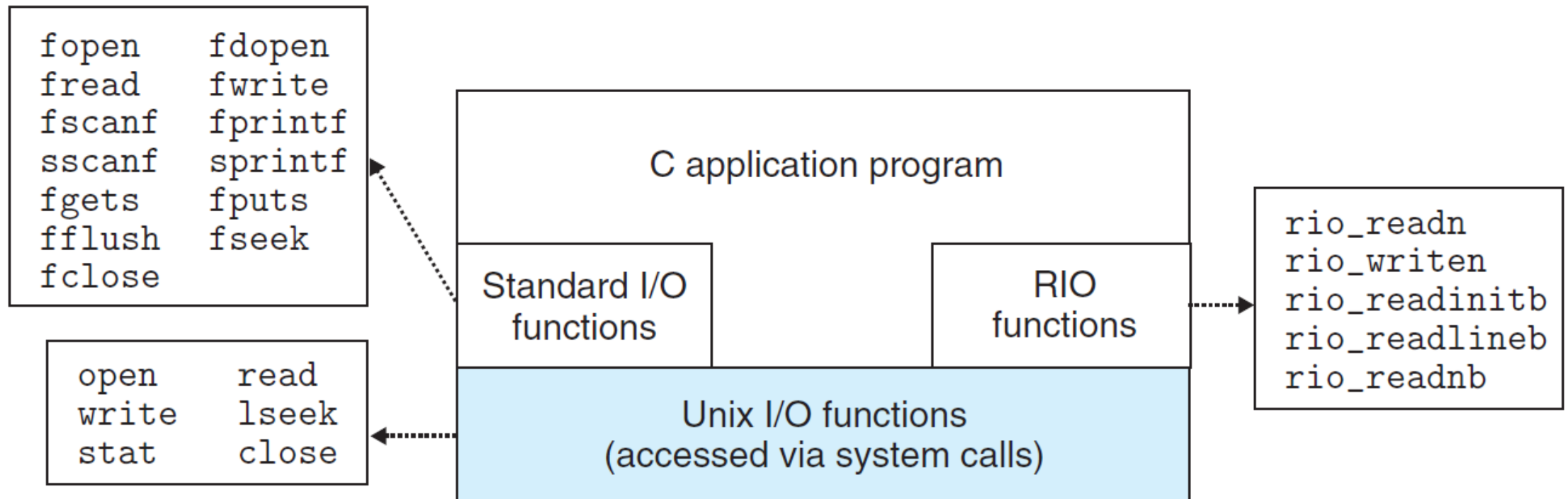
```
ssize_t read(int fd, void *buf, size_t n);
```

Returns: number of bytes read if OK, 0 on EOF, -1 on error

```
ssize_t write(int fd, const void *buf, size_t n);
```

Returns: number of bytes written if OK, -1 on error

# Las funciones I/O estándar y UNIX I/O



# Demostración

---

<https://repl.it/join/unzizuex-fexadom>

# Referencias

---

Libro guía Computer Systems: A programmers perspective. Secciones 10.0 – 10.4, 10.10 – 10.12

**Importante:** En la segunda edición equivale a 10.0 – 10.3, 10.8 – 10.10