

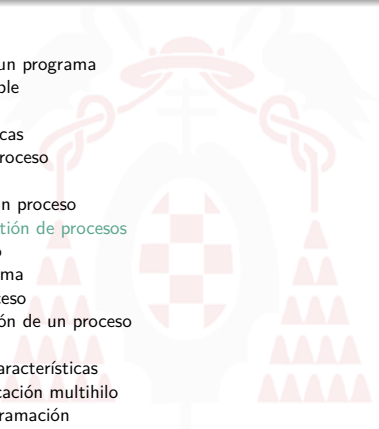
# Procesos e Hilos

*Departamento de Automática*  
Universidad de Alcalá



/gso>

# Índice

- 
- 1 Programa
    - Concepto de programa
    - Fases de desarrollo de un programa
    - Formato de un ejecutable
  - 2 Procesos
    - Concepto y características
    - Bloque de control de proceso
    - Estados de un proceso
    - Mapa de memoria de un proceso
  - 3 Servicios POSIX para gestión de procesos
    - Creación de un proceso
    - Ejecución de un programa
    - Finalización de un proceso
    - Espera por la finalización de un proceso
  - 4 Hilos
    - Objetivo, concepto y características
    - Estructura de una aplicación multihilo
    - Implementación y programación
    - Hilos vs. procesos
    - Ejemplo de programación con hilos
  - 5 Sincronización de Procesos/Hilos
    - Motivos
    - Sincronización con semáforos

# Concepto y características de un programa

## Programa

Un programa es una colección de instrucciones y de datos almacenados en un archivo ordinario.

## Características de un programa

- Residen en dispositivos de almacenamiento permanente.
- Generalmente, para ser ejecutados deben residir en MP.
- Entidad estática.

# Fases de desarrollo de un programa

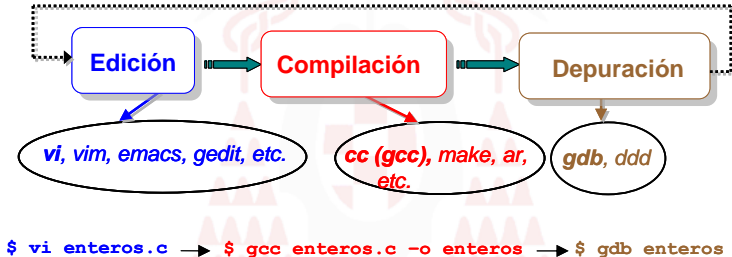
## Ciclo de desarrollo

- 1 Edición.
- 2 Compilación.
- 3 Ejecución.
- 4 **Depuración.**

## Generación de un ejecutable

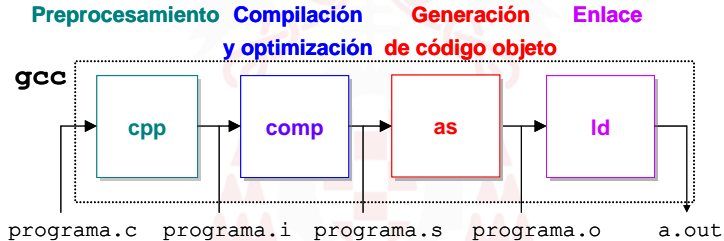


# Fases de desarrollo de un programa en UNIX



# Fases de desarrollo de un programa en UNIX

## Compilación



- La orden `gcc` tiene opciones para generar los archivos intermedios (`-E`, `-S`, `-c`).
- Otras opciones de `gcc`: `-o`, `-Wall`, `-g`, etc.

# Fases de desarrollo de un programa en UNIX

## Ejemplo 1

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

int imprimir = 1;
char *nomprog;

int main(int argc,
        char *argv[])
{
    int i;
    int *ptr;
    nomprog = argv[0];
```

```
    printf("Núm. de args = %d", argc);
    printf("Nombre del prog.: %s\n",
           nomprog);

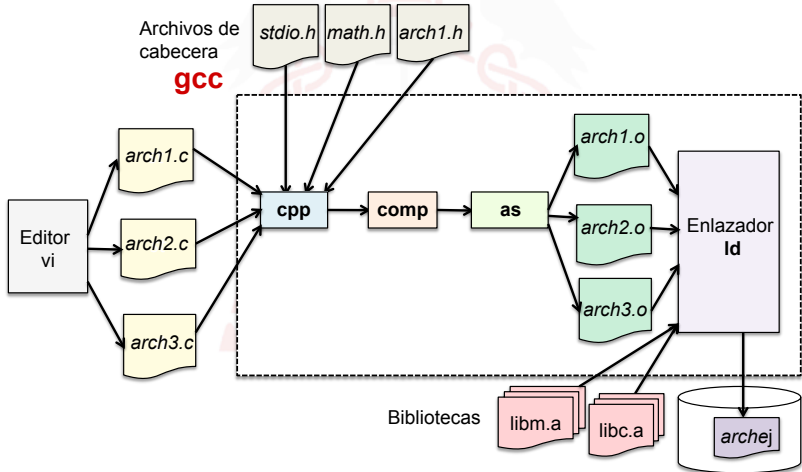
    for (i=1; i <argc; i++)
    {
        ptr = malloc(strlen(argv[i])+1);
        strcpy(ptr, argv[i]);
        if (imprimir) printf("%s\n", ptr);
        free(ptr);
    }

    return 0;
} /* fin main */
```

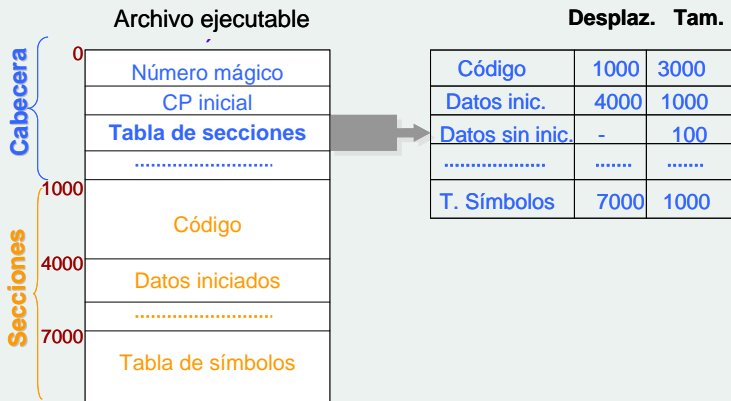


# Fases de desarrollo de un programa en UNIX

## Compilación de varios módulos



# Formato de un ejecutable



⇒ Formatos: ELF, EXE, COM, etc.

# Proceso

## Concepto de proceso

Un proceso es un programa en ejecución.

## Características de un proceso

- Entidad dinámica.
- Los procesos se componen de: código, datos, pila, heap, etc.
- El sistema operativo le asigna recursos: CPU, memoria, archivos, etc.
- El sistema operativo controla su ejecución.
- El sistema operativo lleva la contabilidad de todos los procesos existentes en el sistema.

⇒ Varios procesos pueden ejecutar el mismo programa.

⇒ Un proceso puede derivar de otro proceso.

# Procesos en UNIX

- Cada proceso tiene un identificador único denominado PID.
- Los procesos forman una jerarquía de procesos cuya raíz es el proceso **init**.
  - **init** es el primer proceso que arranca en UNIX.
  - Su PID es 1.
  - La función principal de **init** es arrancar distintos procesos, por ejemplo, login.
- El PPID de un proceso es el PID de su proceso padre.

# Bloque de control de proceso (PCB)

## Concepto

El PCB es una estructura de datos con información de un proceso.

El PCB contiene información referente a:

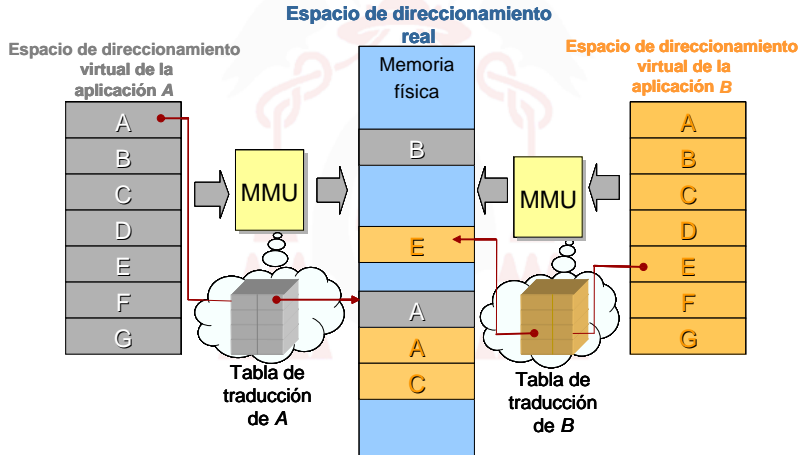
- Estado actual del proceso.
- Identificación unívoca del proceso.
- Prioridad del proceso.
- Puntero a la zona de memoria asignada.
- Punteros a los recursos asociados.
- Área de salvaguarda de registros.
- Un puntero al siguiente PCB, formando una lista enlazada.

⇒ En Linux, el PCB se denomina `task_struct`.

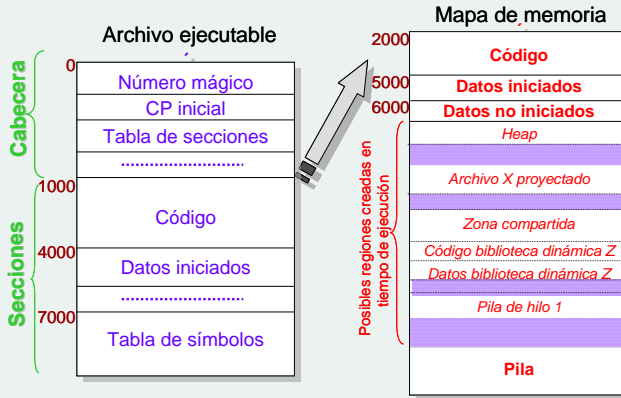
**Zombie**



# Espacio de direccionamiento virtual de un proceso



# Ejecutable y Mapa de memoria de un proceso



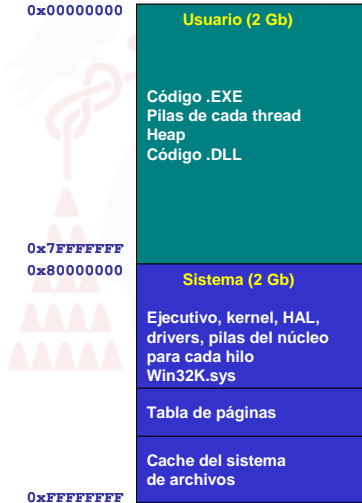
⇒ **Propiedades de la región:** Soporte (archivo/anónimo), compartición, protección, y tamaño.







# Mapa de memoria de un proceso en Windows



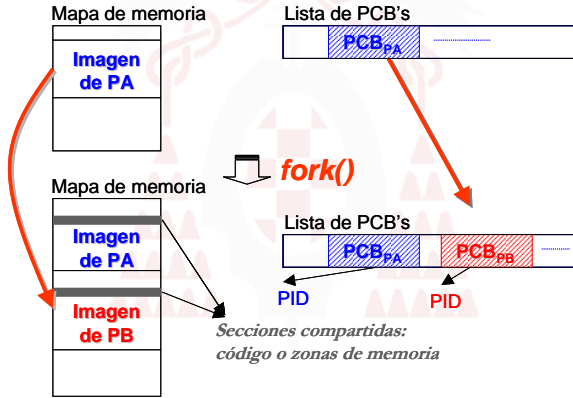
# Servicios POSIX para gestión de procesos

## Uso de los servicios POSIX para gestión de procesos

- Creación de procesos
- Ejecución de un programa
- Finalización de procesos
- **Identificación de procesos**
- **Gestión del entorno de procesos**

# Creación de un proceso

## Esquema



# Creación de un proceso

## Servicio POSIX

### Prototipo

```
pid_t fork();
```

- **Descripción**

Crea una copia (proceso hijo) del proceso que la invoca (proceso padre)

- **Devuelve**

Si la llamada se ejecuta correctamente:

- ⇒ Al padre: el pid del proceso hijo
- ⇒ Al hijo: 0

En caso contrario: -1 y en la variable global `errno` el código del error

# Creación de un proceso

## Ejemplo

```
#include <sys/types.h>
#include <stdio.h>
int main(void)
{
    pid_t id;
    id = fork();
    if(id == -1) {
        perror("Error en el fork");
        exit(1);
    }
    if (id == 0) {
        while (1) printf("Hola: soy el hijo\n");
    }
    else {
        while (1) printf("Hola: soy el padre\n");
    }
}
```





# Ejecución de un programa

## Servicio POSIX

### Prototipo

```
int execl (const char *path, const char *arg, ...);  
int execlp (const char *file, const char *arg, ...);  
int execv (const char *path, char const *argv[]);  
int execvp (const char *file, char const *argv[]);  
int exccve (const char *path, char *argv[], char *  
const envp[]);
```

- **Descripción**

Reemplazan la imagen del proceso actual por la imagen de un proceso a partir de un archivo ejecutable

- **Devuelve**

Si hay error: -1 y en la variable global `errno` el código del error

# Ejecución de un programa

## Ejemplo

```
#include <unistd.h>
#include <stdio.h>
int main(void)
{
    pid_t id;
    char *args[3];
    args[0] = "ps";
    args[1] = "-l";
    args[2] = NULL;

    if (execvp(args[0], args) < 0) {
        perror("Error en execvp");
        exit(-1);
    }
}
```

# Finalización de un proceso

## Servicio POSIX

### Prototipo

```
void exit(int status);
```

- **Descripción**

Termina la ejecución de un proceso. El proceso da una indicación de cómo finalizó mediante `status`

- Cuando un proceso termina, sus hijos no mueren y suelen ser adoptados por el proceso `init`
- El valor de `status` es retornado al proceso padre (**¡si existe!**)

# Espera por la finalización de un proceso

## Servicio POSIX

### Prototipo

```
pid_t wait(int *status);  
pid_t waitpid(pid_t pid, int *status, int options);
```

- **Descripción**

o Permiten que un proceso espere por la finalización de un proceso hijo y obtenga información sobre su estado de terminación en `status`

- **Devuelve**

El identificador del proceso hijo cuya ejecución finalizó

⇒ ¿Cómo ejecuta una orden la *shell*?

# Espera por la finalización de un proceso

## Ejemplo 1/2

```
#include <sys/types.h>
#include <sys/wait.h>
int main(void)
{
    pid_t id;
    int estado;

    id = fork();
    if (id == -1) {
        perror("Error en el fork");
        exit(-1);
    }
```

# Espera por la finalización de un proceso hijo

## Ejemplo 2/2

```
if (id == 0) {  
    printf("Soy el hijo\n");  
    sleep(5);  
    printf("Hijo: despierta y finaliza\n");  
    exit(0);  
} else {  
    printf("Soy el padre y espero ... \n");  
    wait(&estado);  
    printf("Padre: el hijo terminó con estado =%\n", estado);  
    exit(0);  
}  
}  
/* Fin de main */
```

# Hilos

## Objetivo

Compartir recursos entre procesos cooperantes de forma cómoda

## Concepto

- Hilos = proceso ligero = threads = *lightweight process* (LWP) = Unidad fundamental de uso del procesador.
- Básicamente se compone de un CP, una serie de registros y un área de pila
- Un hilo pertenece a otra entidad conocida como tarea (*task*), y sólo a una.
- El código, los datos y los recursos son propiedad de la tarea a la que pertenece el hilo.

# Hilos

## Características

- Cada hilo comparte con los otros hilos cooperantes: código, datos y recursos del SO
- Una tarea sin hilos no tiene capacidad de ejecución
- Los threads son muy adecuados para sistemas distribuidos y sistemas multiprocesador
- Un proceso tradicional (proceso pesado) se compone de una tarea con un hilo de ejecución



**Proceso clásico = proceso pesado**

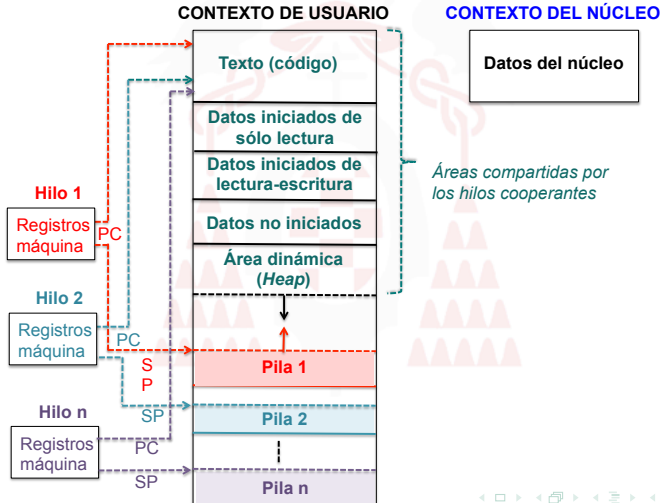


**Procesos ligeros**



# Ejemplo de mapa de memoria de una aplicación con n hilos

## Estructura de una aplicación multihilo



# Hilos

## Implementación y programación

- Los hilos pueden ser implementados en espacio de usuario o soportados por el núcleo como llamadas al sistema
- La programación con hilos debe hacerse cuidadosamente; pueden producirse errores de sincronización
- La mayoría de los S.O. modernos soportan threads (OS/2, Mach, W2K, Chorus, Linux, etc.)
- ¿Cómo programar con hilos?: Bibliotecas estándar *DCE Threads*, *POSIX Threads*, *Sun threads*, etc.

# Hilos vs. procesos

- Los hilos se crean y se destruyen más rápidamente que los procesos
- El tiempo de conmutación entre hilos de la misma tarea es más rápida que la conmutación entre procesos
- Todos los hilos de una tarea comparten memoria  $\Rightarrow$  Menor sobrecarga de comunicaciones
- Un proceso tradicional (proceso pesado) se compone de una tarea con un hilo de ejecución

## Ejemplo de programación con hilos

```
#include <pthread.h>
void * Hilo (void *arg) {
    printf("%s", "Soy el hilo ->");
    printf("%d\n", arg);
    pthread_exit(0);
} /* Fin de Hilo */

main()
{
    pthread_t hilo1, hilo2;
    int i = 1;
    pthread_create(&hilo1, NULL, Hilo, &i);
    i++;
    pthread_create(&hilo2, NULL, Hilo, &i);
    sleep(30);
    printf("Finaliza el hilo principal\n");
}
```

# Sincronización de la ejecución de un proceso o hilo

Con frecuencia los procesos deben coordinarse entre si porque cooperan para lograr un fin o compiten por algún recurso.

## Razones para sincronizar la ejecución de los procesos

- Es necesario compartir recursos en exclusión mutua.
- Un proceso debe esperar hasta que otro le envíe la información que necesita para continuar (detener su ejecución).
- Varios procesos quieren escribir a la vez en la misma posición de memoria (**condiciones de carrera**).

Un proceso no puede detener por sí mismo su ejecución. Tampoco puede alterar la ejecución de otro. Es necesaria la intervención del núcleo.

# Mecanismo de sincronización con semáforos

Los semáforos son objetos proporcionados por el núcleo y se comparten por todos los procesos participantes en la sincronización.

## Operaciones sobre los semáforos.

- Operación P(semáforo):
  - Si el semáforo tiene valor positivo, se decrementa en una unidad.
  - Si el semáforo es cero, el proceso que invoca la llamada queda bloqueado en una lista de espera.
- Operación V(semáforo):
  - Si el semáforo no tiene procesos bloqueados a la espera, se incrementa en una unidad.
  - Si el semáforo tiene procesos bloqueados, se despierta al primero de ellos.

## Solución a algunos problemas típicos con semáforos (I)

### Sección crítica (condiciones de carrera)

$P_1$

...

P(S1);

a++;

V(S1);

...

$P_2$

...

P(S1);

a++;

V(S1);

...

### Productor-Consumidor Básico

$P_{productor}$

...

datos=crear();

V(S1);

...

$P_{consumidor}$

...

P(S1);

consumir(datos);

...

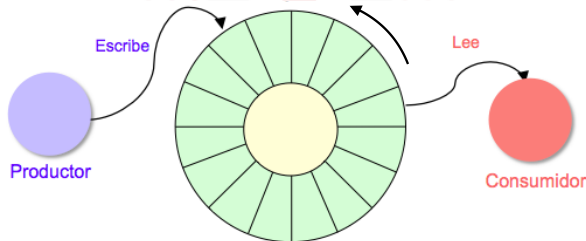
- ¿Afecta el valor al que estén inicializados los semáforos?

# Solución a algunos problemas típicos con semáforos (II)

## Productor-Consumidor genérico

### Descripción del problema

- Un proceso produce datos que, posteriormente, son procesados por otro proceso.
- Uso de un *buffer* (generalmente, circular) como recurso compartido.
- Ejemplo: manejador de teclado y programa que recoge los caracteres de un *buffer*.





# Solución a algunos problemas típicos con semáforos (III)

## Productor-Consumidor genérico

### Restricciones

- 1 El productor espera a que haya elementos del *buffer* vacíos.
- 2 El consumidor espera a que haya datos en el *buffer*.
- 3 Un único proceso puede manipular el *buffer* a la vez.

### Semáforos compartidos

- `smf_llenos = 0`
- `smf_vacios = num_elementos_buffer`
- Semáforo para garantizar exclusión mutua: `exmut = 1`

# Solución a algunos problemas típicos con semáforos (III)

## Productor-Consumidor genérico

### Restricciones

- 1 El productor espera a que haya elementos del *buffer* vacíos.
- 2 El consumidor espera a que haya datos en el *buffer*.
- 3 Un único proceso puede manipular el *buffer* a la vez.

### Semáforos compartidos

- `smf_llenos = 0`
- `smf_vacios = num_elementos_buffer`
- Semáforo para garantizar exclusión mutua: `exmut = 1`

## Solución a algunos problemas típicos con semáforos (IV)

### Productor-Consumidor genérico

#### Productor




```
P(smf_vacios);  
P(exmut);  
    /* Produce un dato */  
    /* en buffer      */  
V(exmut);  
V(smf_llenos);
```

#### Consumidor

```
P(smf_llenos);  
P(exmut);  
    /* Consume un dato */  
    /* del buffer      */  
V(exmut);  
V(smf_vacios);
```

- ¿Es importante el orden en que se ejecutan las primitivas P() y V()?
- ¿Por qué, por ejemplo, el productor hace P(smf\_vacios) y V(smf\_llenos)?
- ¿Sería necesario extender el código si hay varios consumidores?  
¿Cómo?

# Referencias bibliográficas I

-  [Sánchez, 2005] S. Sánchez Prieto.  
*Sistemas Operativos.*  
Servicio de Publicaciones de la UA, 2005.
-  [Márquez, 2004] Francisco M. Márquez.  
*UNIX. Programación Avanzada.*  
Ed. Ra-Ma, 2004.
-  [Tanenbaum, 2009] A. Tanenbaum.  
*Sistemas Operativos Modernos.*  
Ed. Pearson Education, 2009.
-  [Stallings, 1999] W. Stallings.  
*Organización y arquitectura de Computadores.*  
Ed. Prentice Hall, 1999.