

TEMA 3 SSOO: PROCESOS e HILOS

1.- PROGRAMA

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

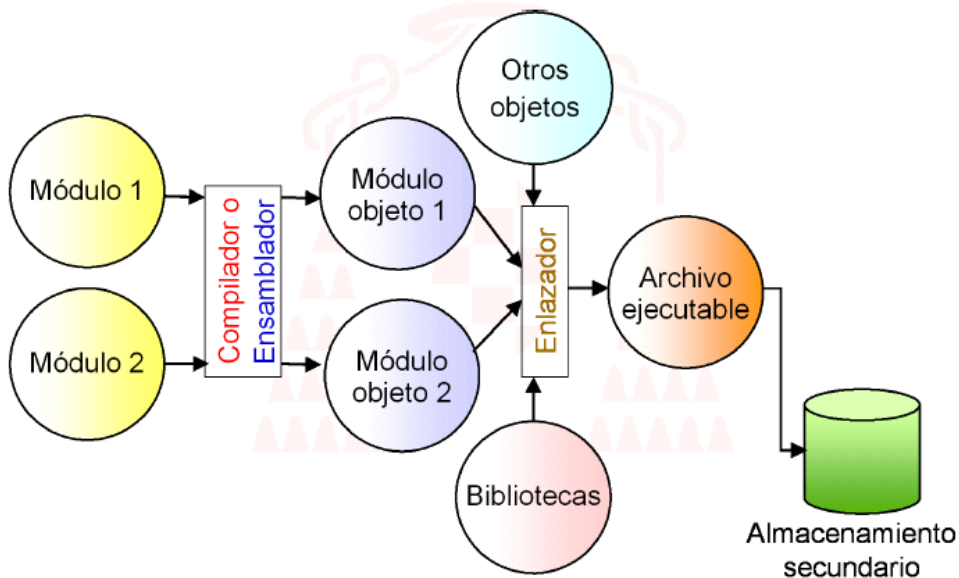
Un programa debe poseer unas características para ser tratado como tal:

- a.- Residen en dispositivos de almacenamiento permanente
- b.- Generalmente, para ser ejecutados deben residir en la memoria principal.
- c.- Deben ser una entidad estática

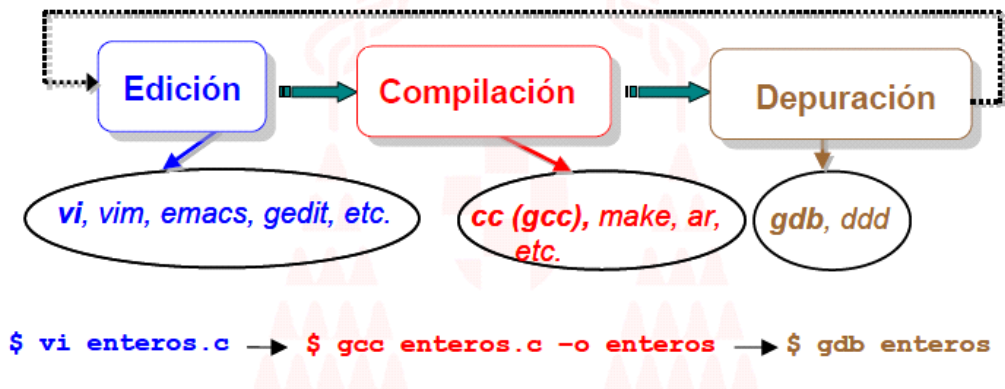
Un programa cuando esta siendo desarrollado pasa por una serie de fases:

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

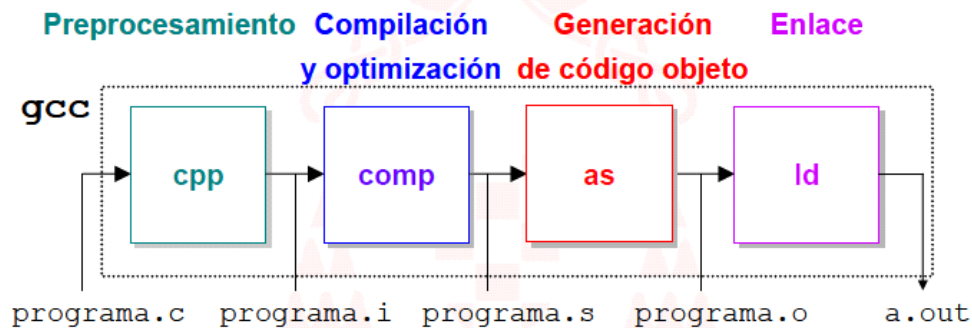
En la siguiente imagen observamos la generación de un ejecutable dentro de las fases de desarrollo de un programa:



A continuación en la siguiente imagen vemos la fases de desarrollo de un programa en Unix:

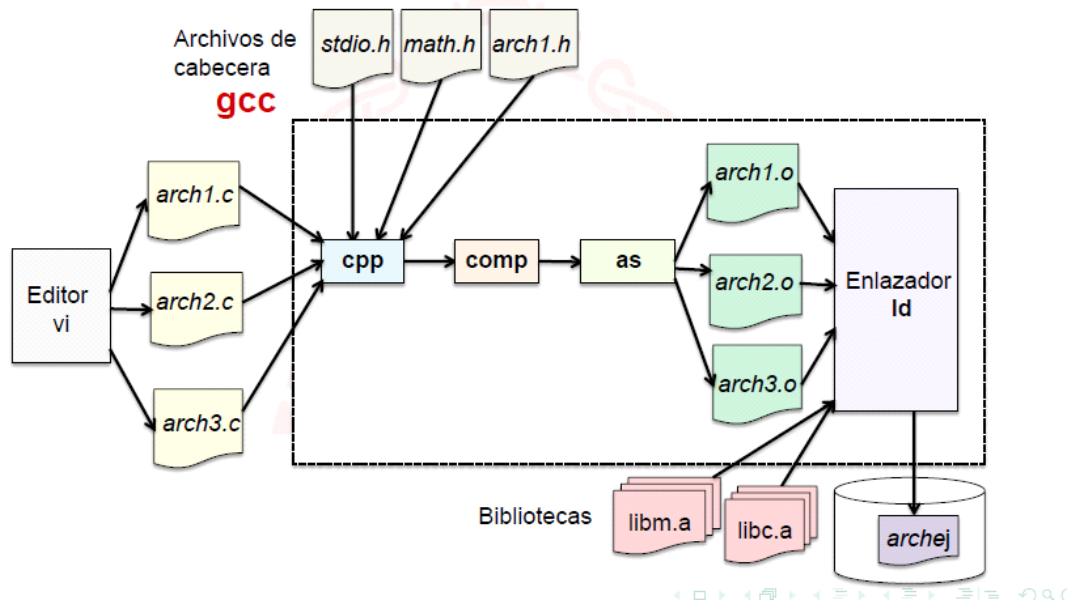


Cuando tenemos desarrollado el programa empezamos la fase de compilación:

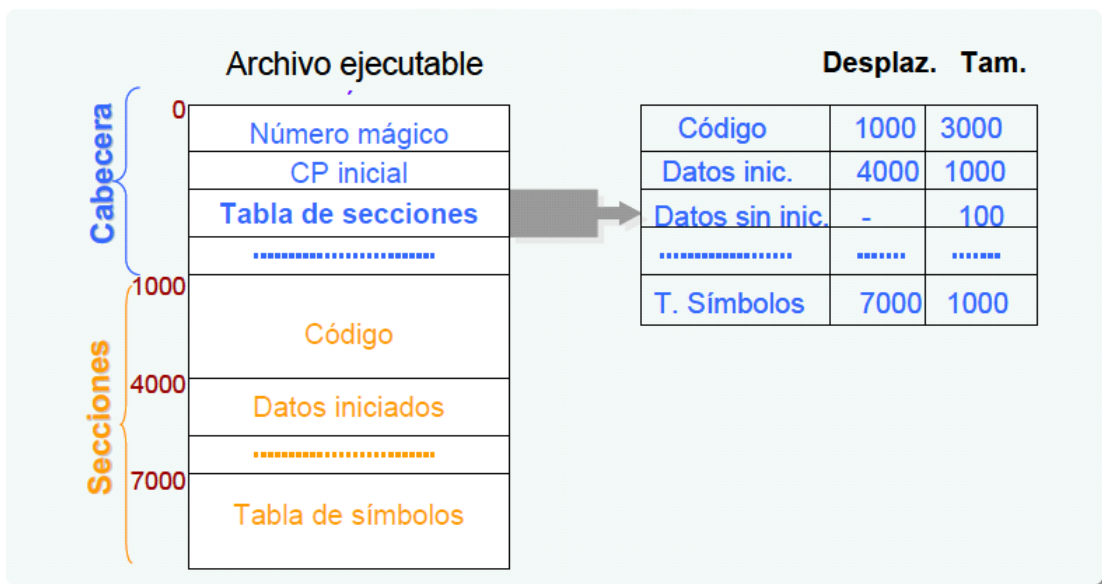


La orden de compilación **gcc** tiene unas opciones para generar los archivos intermedios: -E, -S, -c. A parte de estas opciones **gcc** posee otras como: -o, -Wall, -g, etc...

Con **gcc** podemos compilar varios módulos a la vez, en vez de compilar de uno en uno:



Todos los ejecutables tienen un determinado formato, este formato lo observamos en la siguiente imagen:



Formatos de un ejecutable: ELF, EXE, COM, etc...

2.- PROCESOS

Un proceso es un programa en ejecución. Varios procesos pueden ejecutar el mismo programa y un proceso puede derivar de otro proceso.

Los procesos tienen unas características que los definen:

- 1.- Son entidades dinámicas.
- 2.- Los procesos se componen de: código, datos, pila, heap, etc...
- 3.- El sistema operativo controla su ejecución
- 4.- El sistema operativo también lleva la contabilidad de todos los procesos existentes en el sistema operativo.

2.1.- LOS PROCESOS EN UNIX

En UNIX cada proceso tiene un identificador único denominado PID.

Los procesos forman una jerarquía cuya raíz es el proceso **init**:

- a.- El proceso **init** es el primer proceso que arranca en UNIX
- b.- Su PID es 1
- c.- La función principal de este proceso es arrancar los distintos procesos, por ejemplo, el login para permitir el acceso a los usuarios.

En UNIX existen los procesos padres y éstos se designan con un PPID que es PID de su proceso padre.

2.2.- BLOQUE DE CONTROL DE PROCESO (PCB)

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

El Bloque de Control de Proceso contiene información referente a:

- a.- Estado actual del proceso
- b.- Identificación unívoca del proceso
- c.- Prioridad de un proceso
- d.- Puntero a la zona de memoria que se le asigna
- e.- Los punteros a los recursos asociados al proceso
- f.- Área de salvaguarda de registros
- g.- Un puntero al siguiente PCB, lo que da lugar a la formación de una lista enlazada.

En Linux el PCB se denomina ***task_struct***

2.3.- ESTADOS DE UN PROCESO

En la siguiente imagen podemos observar cada uno de los estados de un proceso:



A continuación definiremos cada uno de estos estados:

- a.- ZOMBIE:
- b.- EJECUCIÓN:
- c.- ESPERA:
- d.- LISTO:
- e.- PARADO:

2.4.- MAPA DE MEMORIA DE UN PROCESO

En el PDF de diapositivas de teoría encontramos imágenes de los mapas de memoria de un proceso desde la diapositiva 15 hasta la 19 (ambas incluidas)

3.- SERVICIOS POSIX PARA GESTIÓN DE PROCESOS

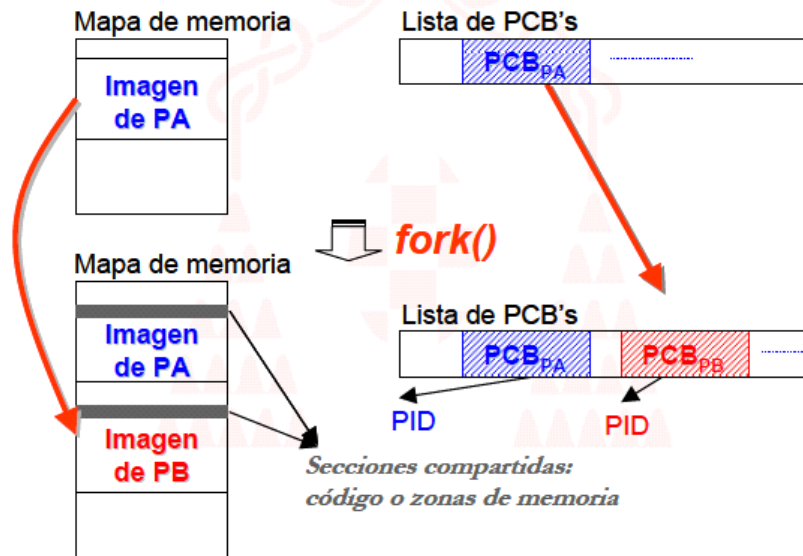
Los servicios POSIX para gestión de procesos se la dan los siguientes usos:

- a.- Creación de procesos
- b.- Ejecución de un programa
- c.- Finalización de procesos

d.- Identificación de procesos

e.- Gestión de entorno de procesos

3.1.- CREACIÓN DE UN PROCESO



Prototipo de la **creación de un proceso** mediante el Servicio POSIX --> `pid_t id`

`id = fork();`

Esto crea una **descripción** { crea una copia (proceso hijo) del proceso que la invoca (proceso padre) } y **devuelve** si la llamada se ejecuta correctamente: al padre el PID del proceso hijo y al hijo un 0, en caso de que ocurra algún error mostrará un -1 y en la variable global `errno` el código del error.

Para la **ejecución de un programa** mediante el Servicio POSIX tenemos:

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 excvce (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: En caso de error devuelve -1 y en la variable `errno` el código del error.

Para la **finalización de un proceso** usaremos: **`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 éste existe.

También podemos encontrar la **espera por la finalización de un proceso**:

`pid_t wait(int *status);`

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

Descripción: Permiten que un proceso espera por la finalización de un proceso hijo y obtenga información sobre su estado de terminación **`status`**.

Devuelve: El identificador del proceso hijo cuya ejecución finalizó

4.- HILOS

El objetivo de los hilos es compartir recursos entre procesos cooperantes de forma cómoda.

El concepto de hilo lo sacamos con:

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

Los hilos poseen una serie de características como son:

- a.- Cada hilo comparte con los otros hilos cooperantes: código, datos y recursos del SO
- b.- Una tarea sin hilos no tiene capacidad de ejecución
- c.- Los *threads* son muy adecuados para sistemas distribuidos y sistemas

multiprocesador

d.- Un proceso tradicional (proceso pesado) se compone de una tarea con un hilo de ejecución

4.1.- IMPLEMENTACIÓN Y PROGRAMACION

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, ya que pueden producirse errores de sincronización

La mayoría de los SSOO modernos soportan *threads*

4.2.- HILOS VS PROCESOS

a.- Los hilos se crean y se destruyen más rápidamente que los procesos

b.- El tiempo de conmutación entre hilos de la misma tarea es más rápida que la conmutación entre procesos

c.- Todos los hilos de una tarea comparten memoria, lo que provoca una menor sobrecarga de comunicaciones

d.- Un proceso tradicional (proceso pesado) se compone de una tarea con un hilo de ejecución

5.- 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.

Existen una serie de razones para sincronizar la ejecución de los procesos:

a.- Es necesario compartir recursos en exclusión mutua

b.- Un proceso debe esperar hasta que otro le envíe la información que necesita para continuar, es decir, detener su ejecución

c.- Varios procesos quieren escribir a la vez en la misma posición de memoria (condiciones de carrera)

Un proceso no puede detener su ejecución por si mismo, así como tampoco alterar la

ejecución de otro. Para estas acciones es necesaria la intervención del núcleo.

5.1.- MECANISMO DE SINCRONIZACIÓN CON SEMÁFOROS

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

Existen varias opciones sobre los semáforos:

a.- Operación P (semáforo):

- Si el semáforo tiene valor positivo, se decrementa en una unidad
- Si el semáforo es cero, proceso que invoca la llamada queda bloqueado en un lista de espera

b.- Operación V (semáforo):

- Si el semáforo no tiene procesos bloqueados a la espera, se incrementa en un unidad.
- Si el semáforo tiene procesos bloqueados, se despierta al primero de ellos

La siguiente imagen muestra la solución a algunos problemas típicos con semáforos:

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);
...

