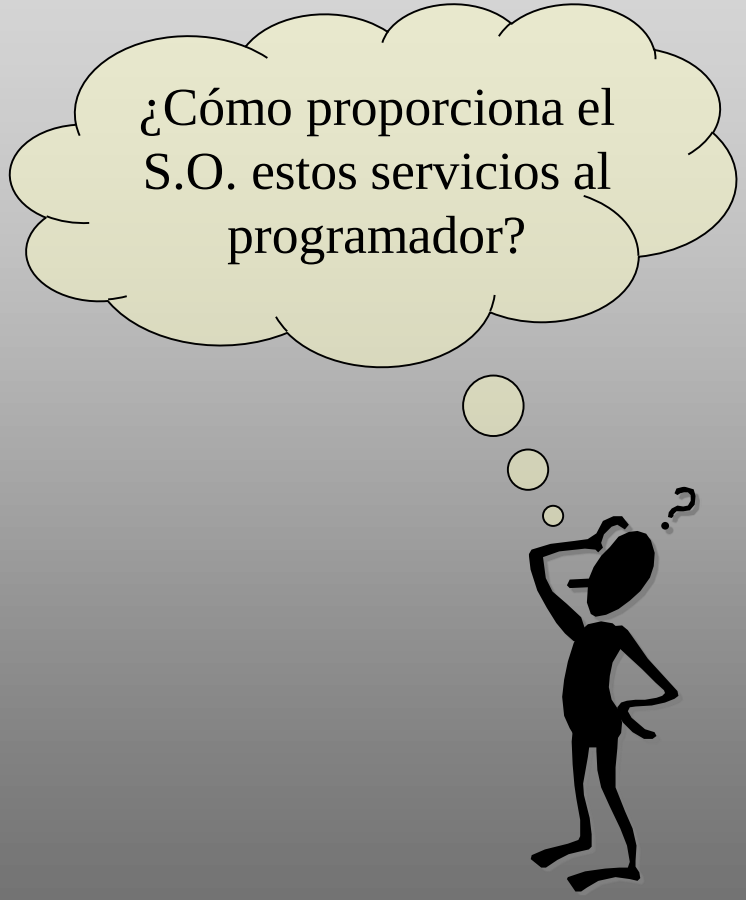
Programación con llamadas al sistema

Interfaz de programación del S.O.

- Conjunto de servicios proporcionados por el S.O. → Interfaz de programación del S.O.
- Los procesos sólo pueden interactuar con el S.O. y el hardware a través de estos servicios → El acceso está controlado.
- El S.O. ejecuta estos servicios en modo privilegiado.



Llamadas al sistema

- Los servicios se proporcionan al programador mediante un conjunto de funciones denominadas **llamadas al sistema**.
- Cada función (servicio) tiene asociado un número entero que lo identifica (código) y una serie de parámetros que deberán ser proporcionados en el momento de su llamada.

¿Cómo pasar los parámetros al S.O.?

¿Cómo llamar a la función (servicio)?

Mecanismo de llamada

- Existen 3 formas básicas de pasar los parámetros:
  a) Cada parámetro a un registro general de la CPU.
  b) Agruparlos en un bloque y pasar su dirección en un registro general.
  c) Los parámetros van a la pila de la máquina (esta es la opción más habitual).
- Se invoca la llamada al sistema mediante una instrucción máquina que habitualmente es un ***trap*** (interrupción software) para que el S.O. tome el control y pueda realizar el servicio solicitado por el proceso de usuario.

Ejemplo de llamada a un servicio

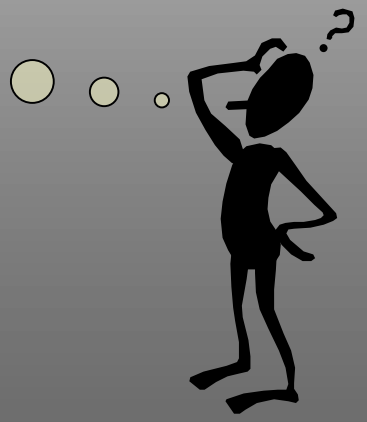

```
MOVE R1,Cod_API
PUSH P1
PUSH P2
PUSH P3
TRAP 3
```

El código de servicio a un registro general de la CPU.

Los parámetros a la pila que comunica el proceso de usuario con el S.O.

Se pasa el control al S.O.

Tiene que haber una forma más sencilla



Librerias de llamadas al sistema (APIs)

Permitir que el usuario pueda utilizar las llamadas al sistema desde los lenguajes de medio/alto nivel como si fueran funciones del propio lenguaje →APIs (*Application Programming Interface)*

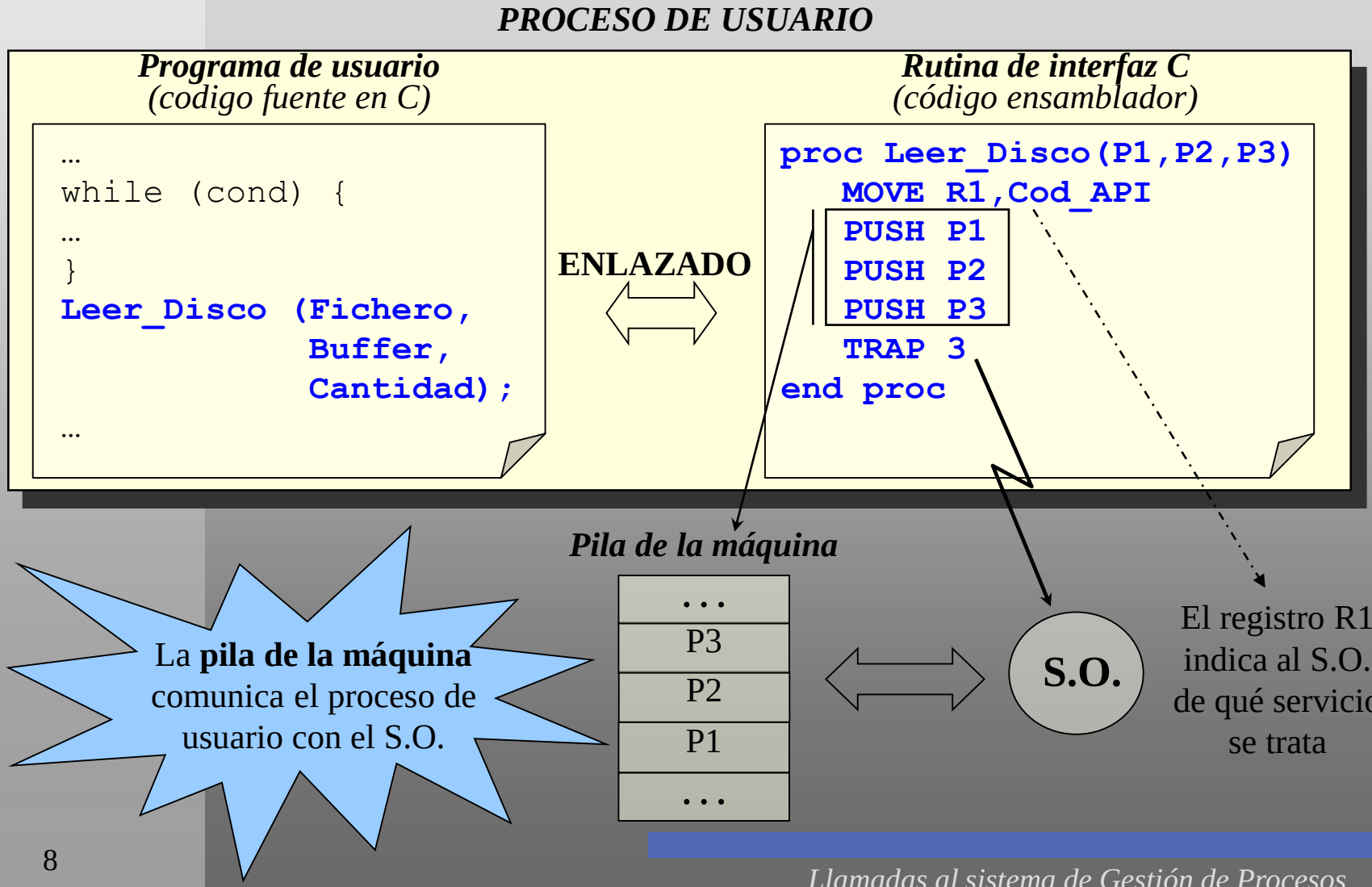
Necesidad de traducir las llamadas del usuario al código ensamblador visto anteriormente.

Las **librerías de llamadas al sistema,** proporcionadas por cada lenguaje, reúnen las denominadas **rutinas de interfaz** (en código objeto) encargadas de dicha traducción.

Rutinas de interfaz

- Hay una **rutina de interfaz** por cada llamada al sistema.
- Recogen la llamada del usuario en el lenguaje de alto nivel y realizan el mecanismo de llamada que espera el S.O.
- El conjunto de todas las rutinas conforman la **librería de llamadas al sistema** ó **APIs** del lenguaje de programación.
- El usuario programador utilizará las APIs que son más sencillas de usar y siguen la sintaxis del lenguaje utilizado (C, Pascal, Ada, etc).

Proceso de usuario y S.O.



Secuencia de una llamada al sistema

Programa de usuario

– Pone parámetros en la pila. – Llama a la rutina de interfaz.	– Recoge el resultado de la pila del proceso. – Siguiente instrucción del programa.

Rutina de interfaz con el S.O.

– Reordena parámetros en la pila. – Pone cod. de servicio en registro general. – *Trap* al sistema operativo.	– Copia el resultado de la operación desde la pila o registro general al parametro de salida del usuario y retorna.

Paso a modo supervisor

Paso a modo usuario

Rutina de tratamiento de interrupción tipo Trap

– Saca parámetros de la pila. – Analiza cuál es el servicio. – Llama a la rutina de S.O. correpondiente.	– Restaura el contexto del proceso y retorna

Rutina del S.O. que realiza el servicio

– Realiza la operación solicitada. – Pone el resultado de la operación en la pila o en un registro y retorna

Portabilidad de las llamadas al sistema

- Las llamadas al sistema son de bajo nivel y dependientes del sistema operativo → **Escasa portabilidad**.
- El **estándar POSIX** (*Portable Operating System Interface*):
  - Especifica un interfaz de S.O. portables.
  - Surgió de la necesidad de unificar distintos S.O. UNIX.
  - Una parte del estándar define los servicios básicos del S.O. (llamadas al sistema).
  - Windows NT ofrece un subsistema para programar aplicaciones POSIX.
- **APIs de Win32** sólo sirven para S.O. Windows.

Categorías de llamadas al sistema

- Servicios de gestión de procesos.
- Servicios de gestión de procesos ligeros.
- Servicios de gestión de memoria.
- Servicios para la comunicación y sincronización de procesos.
- Servicios de entrada/salida.
- Servicios de archivos y directorios.

Servicios de gestión de procesos

Se pueden clasificar en las siguientes categorías:

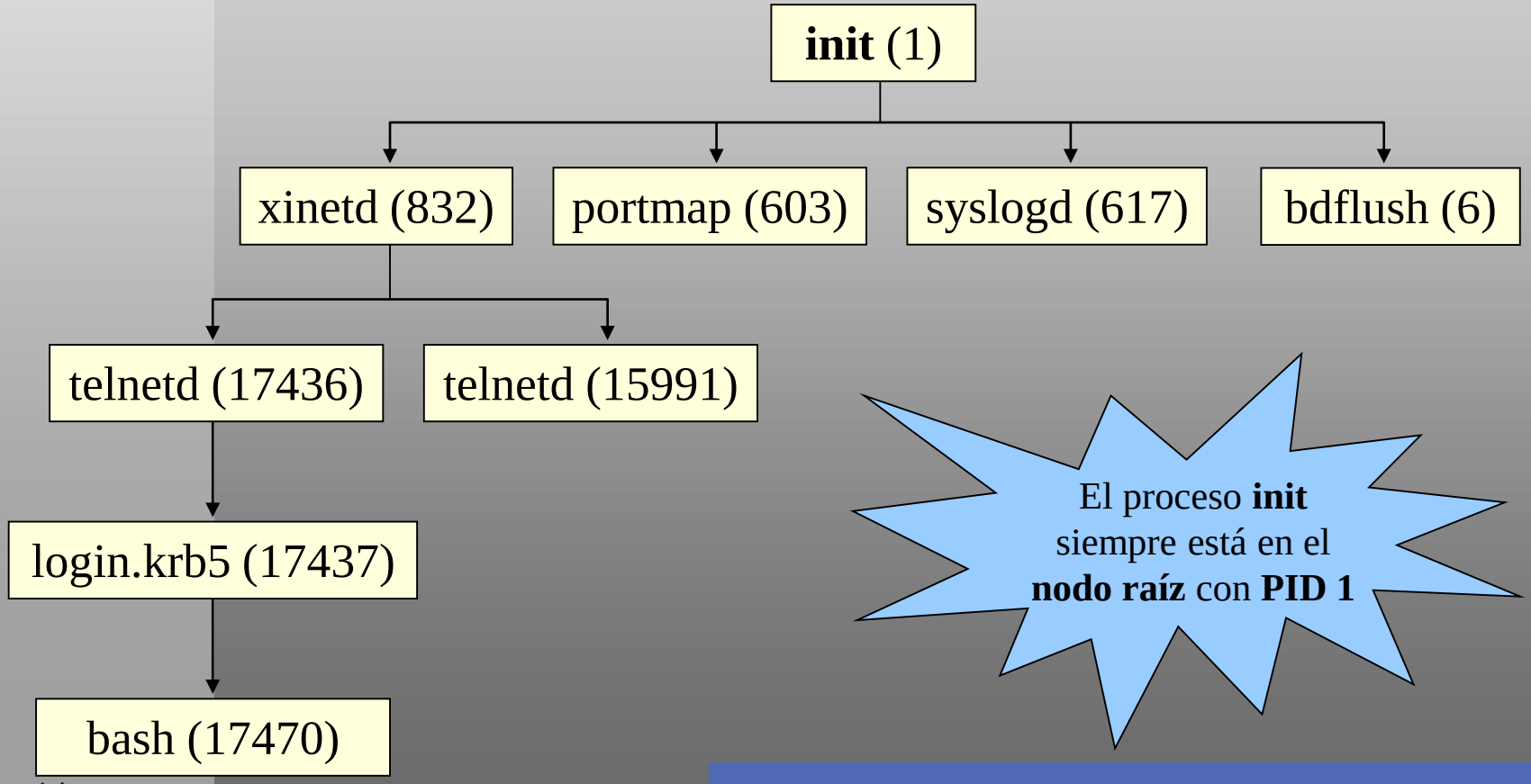
- Identificación de procesos.
- El entorno de un proceso.
- Creación de procesos.
- Terminación de procesos.

POSIX: Identificación de procesos (I)

- POSIX identifica cada proceso mediante un entero positivo único denominado PID (Process IDentification).
- Procesos especiales (nunca terminan):
  - Proceso ***swapper*** que tiene PID 0 y se encarga de volcar páginas a disco.
  - Proceso ***init*** que tiene PID 1 y se encarga de cargar todo el resto de procesos durante el arranque del S.O.
- Los procesos forman una jerarquía en forma de árbol, donde el nodo raíz es el proceso ***init***.
- Los procesos huérfanos pasan a ser hijos del proceso ***init***.

POSIX: Identificación de procesos (II)

Jerarquía de procesos POSIX



El proceso **init** siempre está en el **nodo raíz** con **PID 1**

POSIX: Identificación de procesos (III)

Servicios de *Identificación de procesos* (I)

```
pid_t getpid (void);
// Produce: El identificador (PID) del proceso que realiza la llamada.

pid_t getppid (void);
// Produce: El identificador (PID) del proceso padre que realiza la llamada.
```

POSIX: Identificación de procesos (IV)


```
#include <iostream>
#include <sys/types.h>
#include <unistd.h>

using namespace std;

main()
{
  pid_t id_proceso;
  pid_t id_padre;

  id_proceso = getpid();
  id_padre = getppid();

  cout << "PID proceso: " << id_proceso << endl;
  cout << "PID padre: " << id_padre << endl;
}
```

POSIX: Identificación de procesos (V)

- Los procesos llevan asociados otro tipo de identificadores:
  - Usuario propietario.
  - Usuario efectivo.
  - Grupo propietario.
  - Grupo efectivo.

El **usuario y grupo propietario** indican el ***usuario*** que lanzó el proceso y su ***grupo principal*** respectivamente.

El **usuario y grupo efectivo** indican que el proceso se ejecuta con los privilegios de dicho ***usuario*** y ***grupo***. Por defecto coinciden con el usuario y grupo propietario pero **pueden no coincidir**.

POSIX: Identificación de procesos (VI)

Servicios de *Identificación de procesos* (II)

```
uid_t getuid (void);
// Produce: El identificador (UID) del usuario propietario del proceso.

uid_t geteuid (void);
// Produce: El identificador (UID) del usuario efectivo del proceso.

gid_t getgid (void);
// Produce: El identificador (GID) del grupo propietario del proceso.

gid_t getegid (void);
// Produce: El identificador (GID) del grupo efectivo del proceso.
```

POSIX: Identificación de procesos (VII)


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

using namespace std;

int main()
{
  cout  << "Usuario propietario del proceso: "
        << getuid() << endl;
  cout  << "Usuario efectivo del proceso: "
        << geteuid() << endl;
  cout  << "Grupo propietario del proceso: "
        << getgid() << endl;
  cout  << "Grupo efectivo del proceso: "
        << getegid() << endl;
}
```

POSIX: Identificación de procesos (VIII)

- El usuario con UID 0 es conocido como el **superusuario** y goza de todos los privilegios → Se puede saltar todas las protecciones.
- **Un proceso tiene máximos privilegios (superusuario) cuando su usuario efectivo es 0.**
- La característica _POSIX_ SAVED_IDS (**ID-Guardado**) permite guardar la copia anterior del ID efectivo cuando éste cambia.
- La característica anterior permite a un proceso cambiar temporalmente su nivel de privilegio y luego retornar al que tenía.
- Los IDs *efectivo* y *guardado* cambian con el **permiso S** de los ficheros ejecutables en el momento de su ejecución.

POSIX: Identificación de procesos (IX)

Servicios de *Identificación de procesos* (III)

```
int setuid (uid_t uid);
// Necesita: El UID de un usuario.
// Modifica: El usuario efectivo del proceso a uid.
//           El UID-Guardado pasa a tener el anterior UID efectivo.
// Error: Si uid no coincide con el usuario real o el UID-Guardado.
// Excepción: Si el proceso tiene privilegios de superusuario cambia los
//            tres UIDs a uid (real, efectivo y guardado).
// Produce: 0 en caso de éxito ó -1 en caso de error.

int setgid (gid_t gid);
// Funcionamiento análogo a la función anterior pero trabajando con GIDs.
```

POSIX: Identificación de procesos (XII)

- Cuando un proceso con privilegios de superusuario invoca a **setuid()** o **setgid()** cambia los tres IDs.
- Si este proceso intenta perder temporalmente los privilegios no podrá volver a retomarlos.

Crear una llamada al sistema que sólo cambie el *usuario y grupo efectivos*.

POSIX: Identificación de procesos (XIII)

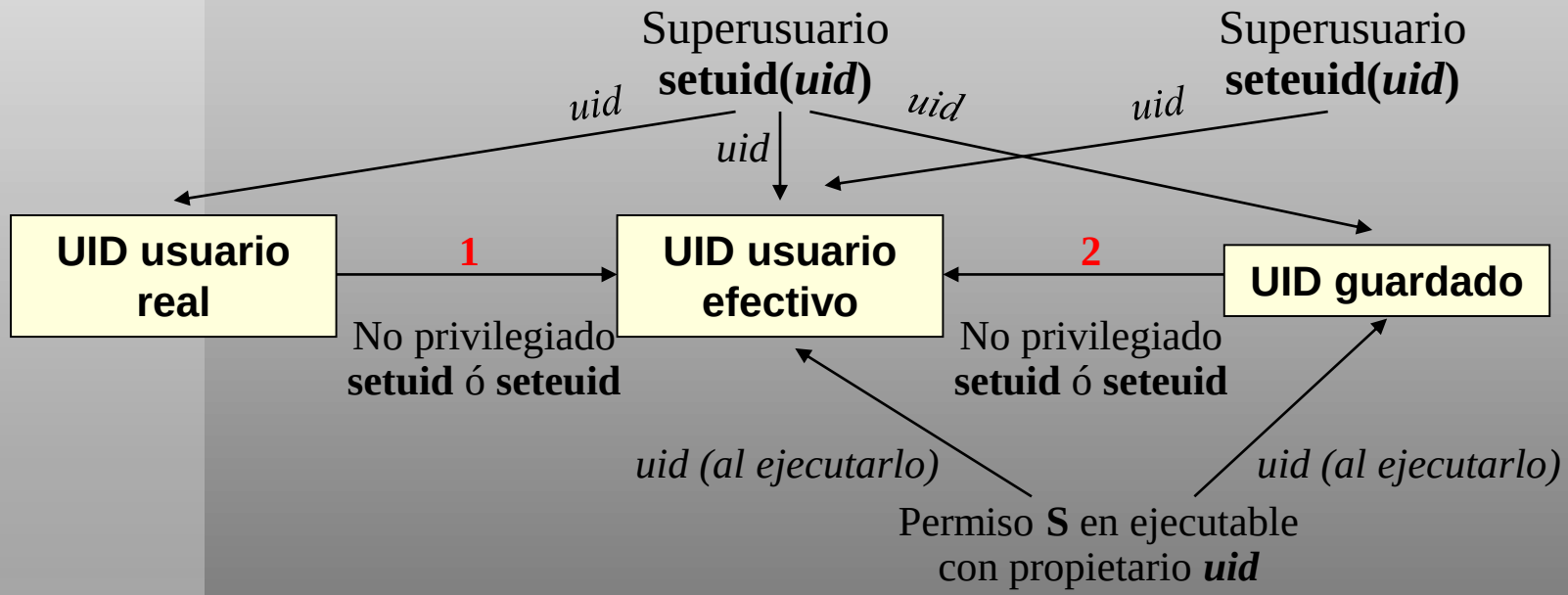
Servicios de *Identificación de procesos* (IV) (NO POSIX)

```
int seteuid (uid_t euid);
// Necesita: El UID de un usuario.
// Modifica: El usuario efectivo del proceso a euid.
//           El UID-Guardado pasa a tener el anterior UID efectivo.
// Error: Si euid no coincide con el usuario real o el UID-Guardado.
// Excepción: Si el proceso tiene privilegios de superusuario se permite
//            cualquier euid.
// Produce: 0 en caso de éxito ó -1 en caso de error.

int setegid (gid_t egid);
// Funcionamiento análogo a la función anterior pero trabajando con GIDs.
```

POSIX: Identificación de procesos (XIV)

Funcionamiento de setuid() y seteuid()



NOTA: Se intenta primero hacer **1** y si falla se intenta hacer **2**. El *uid* indicado debe coincidir con el real (caso 1) o el guardado (caso 2).

POSIX: El Entorno de un proceso (I)

- Lista de variables que se pasan en el momento de comenzar su ejecución.
- La lista es de la forma ***nombre=valor.*** Las más comunes son:
  - **HOME**: Directorio de trabajo inicial del usuario.
  - **LOGNAME**: Nombre del usuario (login).
  - **PATH**: Directorios para encontrar ejecutables.
  - **SHELL**: Intérprete de comandos por defecto.

POSIX: El entorno de un proceso (II)

Servicios de *Entorno de un Proceso*

```
char *getenv (const char *nombre);
// Necesita: El nombre de una variable de entorno.
// Produce: El valor asociado con dicha variable ó
//                NULL si no existe la variable.

int putenv (const char *cadena);
// Necesita: Una cadena con la forma nombre=valor
// Modifica: El entorno del proceso añadiendo o modificando la
//                variable con el nombre y valor indicado.
// Produce: 0 en caso de éxito ó -1 en caso de fallo.
```

POSIX: El entorno de un proceso (III)

Ejemplo en C para obtener el entorno de un proceso

```
#include <iostream>
#include <stdlib.h>

using namespace std;

main(int argc, char *argv[], char *envp[])
{
  int i;

  for (i=0; envp[i] != NULL; i++)
    cout << envp[i] << endl;
}
```

POSIX: El entorno de un proceso (IV)

Ejemplo de getenv()

```
#include <iostream>
#include <stdlib.h>

using namespace std;

main()
{
  cout << "Soy el usuario " << getenv("LOGNAME")
       << " y estoy trabajando con el Shell "
       << getenv("SHELL") << endl;
}
```

POSIX: Creación de procesos (I)

- Todos los procesos nuevos del sistema se crean con la llamada al sistema **fork()**.
- El proceso que realiza la llamada será el **proceso padre** del nuevo proceso. El proceso nuevo se conoce como **proceso hijo.**
- La relación entre procesos padre e hijo establece la jerarquía de procesos.
- La llamada al sistema **fork()** retorna dos veces: una para el padre y otra para el hijo.

POSIX: Creación de procesos (II)

Semejanzas entre padre e hijo

- El proceso hijo es una copia casi exacta del padre. Se copian los siguientes elementos del padre:
  1. La memoria (área de datos, código y pila).
  2. Casi todo el PCB.
- Al duplicarse los descriptores de fichero del padre, ambos (padre e hijo) comparten los ficheros que aquel tuviera abiertos en el momento de la llamada.

Cuando el fork() retorna, padre e hijo continúan la ejecución en la instrucción siguiente → El proceso hijo no comienza su ejecución desde el principio. (Intentar razonar por qué).

POSIX: Creación de procesos (III)

Diferencias entre padre e hijo

- El valor retornado por **fork().**
- Los PIDs de los procesos.
- El PID del padre de los procesos.
- Las estadísticas del proceso hijo se ponen a 0.
- Las alarmas pendientes del proceso hijo se desactivan.
- El conjunto de señales para el hijo se pone a vacío.

¡El valor retornado por la llamada al sistema fork() será crucial para distinguir quién es el proceso padre y quién el proceso hijo!

POSIX: Creación de procesos (IV)

Servicios de *Creación de Procesos*

```
pid_t fork ();
// Modifica: Crea un nuevo proceso que es casi una copia del proceso
//                que realiza la llamada.
// Produce: En el proceso padre el PID del nuevo proceso ó -1 si error.
//               En el proceso hijo siempre un 0
```

POSIX: Creación de procesos (V)


```
#include <iostream>
#include <sys/types.h>
#include <unistd.h>

using namespace std;
main()
{
  pid_t pid;
  pid = fork();
  switch (pid) {
    case -1: // Error del fork
      perror("fork");
      break;
    case 0: // Proceso hijo
      cout << "Proceso hijo " << getpid() << "; padre = "
           << getppid() << endl;
      break;
    default: // Proceso padre
      cout << "Proceso padre " << getpid() << "; padre = "
           << getppid() << endl;
  }
}
```

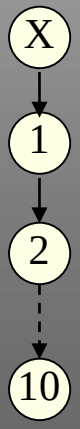
POSIX: Creación de procesos (VI)

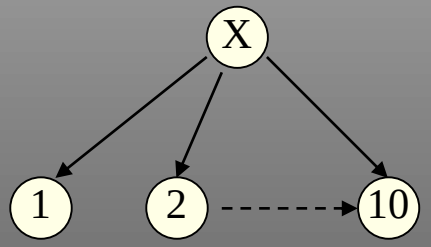

```
#include <iostream>
#include <sys/types.h>
#include <unistd.h>

using namespace std;
main()
{
  for (int i=0; i < 10; i++)
    if (fork()) break;
  cout << "Proceso " << getpid()
       << "; padre = "
       << getppid() << endl;
}
```

```
#include <iostream>
#include <sys/types.h>
#include <unistd.h>

using namespace std;
main()
{
  for (int i=0; i < 10; i++)
    if (!fork()) break;
  cout << "Proceso " << getpid()
       << "; padre = "
       << getppid() << endl;
}
```





POSIX: Creación de procesos (VII)


```
#include <iostream>
#include <sys/types.h>
#include <unistd.h>

using namespace std;
main()
{
  for (int i=0; i < 3; i++)
    fork();
  cout << "Proceso " << getpid()
       << "; padre = "
       << getppid() << endl;
}
```

POSIX: Creación de procesos (VIII)


```
#include <iostream>
#include <sys/types.h>
#include <unistd.h>

using namespace std;

int glob = 6;
main()
{
  int var;
  pid_t pid;

  cout << "Comienzo la ejecución" << endl;
  var = 88;
  if ((pid = fork()) == EXIT_FAILURE) exit(EXIT_FAILURE);
  if (!pid) { glob++; var++; }
  else // Se asegura que padre espera a que termine hijo
    sleep(2);
  cout << "pid = " << getpid() << ", glob = " << glob
       << ", var = " << var << endl;
  exit(EXIT_SUCCESS);
}
```

POSIX: Ejecución de procesos (I)

- La llamada al sistema genérica **exec()** cambia el programa que está ejecutando el proceso que lo llama.
- Esta llamada al sistema **NO CREA** un nuevo proceso.
- La llamada al sistema consta de dos fases:
  1. Fase de vaciado.
  2. Fase de carga.

La llamada al sistema **exec()** **NUNCA retorna** si tiene éxito.

POSIX: Ejecución de procesos (II)

Fase de vaciado

- Se libera de la memoria el área de texto, datos y pila del proceso.
- **Se conserva** la siguiente información:
  - Entorno del proceso.
  - Identificadores del proceso (salvo quizás los efectivos).
  - Descriptores de ficheros abiertos (si así viene indicado).
  - Terminal asociada al proceso.
  - Tiempo hasta siguiente señal de alarma.
  - Directorios raíz y por defecto.
  - Máscara de señales y señales pendientes.

POSIX: Ejecución de procesos (III)

Fase de carga

- Se asigna al proceso nuevo espacio de memoria.
- Se carga el área de datos y texto del nuevo programa.
- Se crea una pila inicial del proceso con su entorno y los parámetros del programa.
- Modificar campos del PCB relativos a la ubicación del proceso en memoria principal.

POSIX: Ejecución de procesos (IV)

Servicio de *Ejecución de Procesos (I)*

int **execl** (const char *path, const char *arg1, const char *arg2, …);
// Necesita: La ruta del programa ejecutable y la lista de sus parámetros.
// Modifica: El programa que ejecuta el proceso.
// Produce: -1 en caso de error.

int **execv** (const char *path, char *const argv[]);
// Necesita: La ruta del programa ejecutable y vector de parámetros.

int **execle** (const char *path, const char *arg1, …, char *const envp[]);
// Necesita: La ruta del programa ejecutable, la lista de parámetros y
// el vector con el entorno del proceso.

int **execve** (const char *path, char *const argv[], char *const envp[]);
// Necesita: La ruta del programa ejecutable, el vector de parámetros
// y el vector con el entorno del proceso.

POSIX: Ejecución de procesos (V)

Servicio de *Ejecución de Procesos (II)*

```
int execlp (const char *file, const char *arg1, const char *arg2, …);
// Necesita: El nombre del programa ejecutable y la lista de parámetros.
//           (Busca el ejecutable en los directorios indicados en PATH)

int execvp (const char *file, char *const argv[]);
// Necesita: El nombre del programa ejecutable y el vector de parámetros.
//           (Busca el ejecutable en los directorios indicados en PATH)
```

POSIX: Ejecución de procesos (VI)

Servicio de *Ejecución de Procesos (III)*

El nombre de la función contiene caracteres con distinto significado:

l Los argumentos del programa se dan como una lista indefinida de parámetros, terminados con el parámetro NULL.

v Los argumentos del programa se dan mediante un vector de cadenas, siendo la última posición del vector el puntero nulo (valor NULL)

e Se va a indicar el entorno inicial del proceso mediante un vector de cadenas, siendo la última posición del vector el puntero nulo.

p El ejecutable debe buscarse no solo en el directorio actual sino también en los directorios indicados por la variable de entorno PATH.

POSIX: Ejecución de procesos (VII)


```
#include <iostream>
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

using namespace std;
main()
{
  pid_t pid = fork();
  switch (pid) {
    case -1: // Error del fork
      perror("fork");
      break;
    case 0: // Proceso hijo
      execlp("ls", "ls", "-l", NULL);
      perror("exec");
      break;
    default: // Proceso padre
      cout << "Proceso padre" << endl;
  }
}
```

POSIX: Terminación de procesos (I)

- Un proceso puede terminar su ejecución de forma normal o de forma anómala.
- Casos en que termina de forma normal:
  A. Ejecutando la función **return()** en la función **main()**
  B. Ejecutando la función de librería **exit()**
  C. Ejecutando la llamada al sistema **_exit()**

Las tres alternativas anteriores son de diferente nivel de abstracción. La función **return()** llama en realidad a la función **exit()** y ésta a su vez a la llamada al sistema **_exit()**

POSIX: Terminación de procesos (II)

Servicios de *Terminación de procesos* (I)

```
void _exit (int status);
// Necesita: El código de terminación (indica estado en que termina)
// Modifica: Termina normalmente el proceso que lo llama y realiza las
//           gestiones que se indican a continuación:
//
//    1 Se cierran todos los descriptores de ficheros.
//    2 Si el proceso padre está ejecutando un wait() o waitpid() se le
//      notifica la terminación del hijo.
//    3 Si el proceso padre no está ejecutando las llamadas al sistema
//      del punto 2 el código de terminación se salva hasta que lo haga.
//    4 El sistema operativo libera todos los recursos del proceso.
```

POSIX: Terminación de procesos (III)

Esperar por los procesos hijos

- En numerosas ocasiones es necesario que un proceso no haga nada hasta que finalice uno o varios de sus hijos → Necesaria sincronización entre padre e hijos.
- La llamada al sistema **wait()** suspende la ejecución del proceso hasta que finaliza uno de sus hijos.
- Se puede obtener información adicional de cómo y por qué finalizaron dichos procesos.

POSIX: Terminación de procesos (IV)

Servicios de *Terminación de procesos* (II)

```
pid_t wait (int *status);
// Necesita: Una variable entera status.
// Modifica: Suspende la ejecución del proceso hasta que finalice uno de
//           sus hijos. La variable status almacenará información sobre
//           el hijo que haya terminado.
// Produce:  El PID del proceso hijo que haya terminado ó -1 si error
pid_t waitpid (pid_t pid, int *status, int options);
// Necesita: El PID de un proceso hijo, una variable status y opciones
//           de llamada (la más interesante es WNOHANG).
// Modifica: Suspende la ejecución del proceso hasta que finalice el
//           proceso hijo PID. La variable status almacenará información
//           sobre dicho hijo.
// Produce:  El PID del proceso hijo que haya terminado ó -1 si error
```

POSIX: Terminación de procesos (V)

Combinaciones de Terminación

- Si un **proceso hijo acaba antes de que el padre haga un wait()** el proceso hijo queda en estado ***zombie*** (se libera todo menos su PCB).
- Si un **proceso hace un wait() con hijos en estado *zombie*** la llamada retorna inmediatamente y libera a uno de estos hijos.
- Si un **proceso acaba antes que sus hijos** éstos pasan a tener como proceso padre al proceso **init** (PID=1). Este proceso está realizando constantemente la llamada al sistema **wait()**

POSIX: Terminación de procesos (VI)

Macros para capturar el estado del hijo

WIFEXITED(status): Devuelve **cierto** si el hijo terminó normalmente.

WEXITSTATUS(status): Devuelve el valor indicado en la llamada _exit() del hijo

WIFSIGNALED(status): Devuelve **cierto** si el hijo terminó por una señal que no tenía manejador

WTERMSIG(status): Devuelve el número de señal que finalizó el proceso hijo.

POSIX: Gestión de señales (I)

- Los procesos pueden recibir señales →**Interrupciones *Software***
- Estas señales pueden proceder del sistema operativo o de otros procesos.
- Pueden aplicarse como mecanismo para sincronizar varios procesos entre sí.
- Ciertas señales especiales permiten terminar, abortar o matar procesos bloqueados.

POSIX: Gestión de señales (II)

Categorías de servicios relativos a señales

- **Envío de señales**, es decir enviar una señal a un proceso.
- **Captura de una señal** para realizar una acción determinada cuando llega dicha señal.
- **Operaciones sobre conjuntos de señales** que permitirán hacer referencia a varias señales a la vez.
- **Bloqueo de señales** para posponer su captura.
- **Espera de señales** que bloquea al proceso hasta recibir una señal
- **Servicios de temporización**.

POSIX: Gestión de Señales (III)

Servicio de *Envío de señal*

```
int kill (pid_t pid, int sig);
// Necesita: El PID del proceso que recibirá la señal.
//           El número de señal a enviar.
// Modifica: Envía la señal sig al proceso pid
// Produce:  0 si tuvo éxito o -1 en caso de error.
```

Un proceso P1 puede enviar una señal a otro P2 si:

1. **El UID efectivo de P1 es cero.**
2. **El UID real o efectivo de P1 es igual al UID real o guardado de P2.**

POSIX: Gestión de Señales (IV)

Listado de Señales

SIGABRT: Terminación anormal.
SIGALRM: Fin de temporización.
SIGFPE: Operación aritmética errónea.
SIGHUP: Desconexión del terminal de control.
SIGILL: Instrucción hardware inválida.
SIGINT: Atención interactiva.
SIGKILL: Matar al proceso **(no se puede ignorar ni capturar)**
SIGPIPE: Escritura en tubería sin lectores.
SIGQUIT: Terminación interactiva.
SIGSEV: Referencia a memoria inválida.
SIGTERM: Terminar al proceso.
SIGCHLD: Terminó uno de los procesos hijos **(por defecto se ignora).**
SIGSTOP: Bloquear al proceso **(no se puede ignorar ni capturar)**.
SIGCONT: Continuar el proceso si está parado.
SIGUSR1: Señal definida por la aplicación.
SIGUSR2: Señal definida por la aplicación.

POSIX: Gestión de Señales (V)

Proceso P1

```
#include <sys/types.h>
#include <unistd.h>
#include <signal.h>

main()
{
  pid_t pid=fork();
  if (!pid)
    execlp("xemacs", NULL);
  else {
    ...
    kill(pid, SIGUSR1);
    ...
  }
}
```

Ejecuta el xemacs
(Proceso hijo es Proceso P2)

Proceso P2

Imagen del xemacs

Envía señal SIGUSR1

POSIX: Gestión de Señales (VI)

Captura de una señal (I)

- Permite realizar una determinada acción cuando llega una señal al proceso.
- A cada señal se le puede asociar su propia acción, de entre las tres siguientes:
  1. Realizar la acción por defecto (generalmente finalizar el proceso).
  2. Invocar a una función del programa.
  3. Ignorar la señal.

POSIX: Gestión de Señales (VII)

Captura de una señal (II)

- La acción que se quiere realizar sobre una señal se especifica mediante la siguiente estructura:

```
struct sigaction {
    void (*sa_handler)(int);
    sigset_t sa_mask;
    int sa_flags;
};
```

1 **SIG_DFL**
2 **Función manejadora.**
3 **SIG_IGN**

Conjunto de señales bloqueadas mientras se trata la señal

POSIX: Gestión de Señales (VIII)

Servicio de *captura de señal*

```
int sigaction (int sig, struct sigaction *nuevo, struct sigaction *actual);
// Necesita: El número de señal a capturar.
//           Un puntero a la estructura que indica la nueva acción a
//           realizar ó NULL si no se desea cambiar la captura.
//           Un puntero a una variable donde dejar la información de la
//           captura actual ó NULL si no se desea obtener esa información.
// Modifica: El comportamiento del proceso cuando recibe la señal sig
// Produce:  0 si tuvo éxito o -1 en caso de error.
```

POSIX: Gestión de Señales (IX)

Proceso P1

```
#include <sys/types.h>
#include <unistd.h>
#include <signal.h>

main()
{
  pid_t pid=fork();
  if (!pid)
    execlp("a.out", NULL);
  else {
    ...
    kill(pid, SIGUSR1);
    ...
  }
}
```

Proceso P2 (a.out)

```
#include <sys/types.h>
#include <signal.h>

void Manejador(int sig) {
  ...
}
main()
{
  struct sigaction act;

  act.sa_handler=Manejador;
  act.sa_flags=0;
  sigemptyset(&act.sa_mask);
  sigaction(SIGUSR1, &act, NULL);
  ...
  ...
}
```

2

3

1

Envía señal SIGUSR1

POSIX: Gestión de Señales (XV)

Servicio de *espera de señales*

```
int pause (void);
// Modifica: Bloquea al proceso hasta la recepción de cualquier señal
//                no ignorada.
```

POSIX: Gestión de Señales (XVI)

Servicios de *temporización*

```
unsigned int alarm (unsigned int seconds);
// Necesita: Un número de segundos
// Modifica: Envía la señal SIGALRM al propio proceso cuando transcurre
//           el número de segundos indicado.

int sleep (unsigned int seconds);
// Necesita: Un número de segundos
// Modifica: Bloquea al proceso hasta que transcurra el número de
//           segundos indicado o hasta la recepción de una señal no
//           ignorada.
```