

Contenido

1. Introducción
2. Procesos pesados
 - 2.1 exec()
3. Depuración de programas.
4. Procesos ligeros (hilos)
5. Comunicación
 - 5.1 Paso de mensajes
 - 5.2 Semáforos

1. Introducción (I)

- POSIX (Portable Operating System Interface for Unix): Conjunto de estándares definidos por IEEE para UNIX que definen la interfaz de programación de aplicaciones (API, Application Programming Interface), así como la interfaz para utilidades y el shell.
 - A pesar de que UNIX es ya de por sí estándar, había muchos “sabores” que imposibilitaban la portabilidad de los programas (código fuente).
 - También se puede aplicar a otros sistemas operativos:
 - LINUX
 - Windows
 - Mac OS
- El estándar define, por ejemplo:
 - Tipos de datos.
 - Nombres de funciones y sus valores devueltos.
 - Identificación de recursos.
 - Biblioteca C.
 - **Creación y control de procesos.**
 - Aquí vamos a seguir este estándar en la creación de procesos concurrentes.

1. Introducción (II)

- **Procesos pesados:** Tienen su propia pila, registros y datos.
 - Su coste de creación en tiempos de CPU y memoria son mayores que los ligeros.
 - Se crean vía *fork()*.
- **Procesos ligeros:** Comparten código, memoria y dispositivos. Se conocen también como **hilos**.
 - Se crean y destruyen rápidamente.
 - El cambio de contexto entre hilos de un mismo padre es menos costoso.

2. Procesos Pesados (I)

- Llamada al sistema **fork()**. Permite crear procesos hijos.
 - Tras su ejecución existen en ejecución dos procesos, padre (el que ejecutó `fork()`) e hijo (el nuevo creado), cuyas imágenes de memoria son básicamente iguales.
 - Padre e hijo comparten:
 - Segmento de texto (código).
 - Dispositivos y ficheros que tuviera abiertos el proceso padre.
 - Comparten `stdin`, `stdout` y `stderr`.: Esto quiere decir que la salida estándar y salida de error estándar de padre e hijo aparecerán mezcladas.
 - Padre e hijo no comparten:
 - PIDs: el PID de ambos es diferente (son procesos diferentes).
 - Segmento de datos y pila: la ejecución de `fork()` crea un nuevo espacio en memoria para el segmento de datos y pila del proceso hijo. En el momento de su creación los valores almacenados son exactamente iguales que los del padre (se produce un duplicado). Después, cada uno puede ejecutar partes de código diferentes, que afectarán a su zona de memoria, teniendo, entonces, evoluciones diferentes.
 - En la llamada a `fork()` el sistema devuelve un 0 al proceso hijo y un valor positivo distinto de 0 (PID del hijo) al padre. Si el valor devuelto es negativo significará que ha habido algún error en la creación del hijo y ésta no se ha podido realizar.
 - De esta manera podremos diferenciar en el código a cada proceso y que cada uno ejecute la parte que le corresponda.

2. Procesos Pesados (II)

- Ejemplo de uso de fork (I)

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

main()
{
    int pid;
    /******
    creación de un proceso concurrente con el creador
    *****/
    pid=fork ();
    if (pid == -1) {
        printf ("error en creacion de proceso hijo\n");
        exit(1);
    } else
        if ( pid == 0)    /* proceso hijo */ {
            printf ("Proceso hijo 1\n"); fflush(stdout);
            exit(0); /* terminacion con codigo 0 */
        } else /* proceso padre */ {
            printf ("Proceso padre\n"); fflush(stdout);
            wait(0);
        }
}
```

Memoria

Padre

2. Procesos Pesados (III)

- Ejemplo de uso de fork (II)

Proceso Padre

```
# include <stdio.h>
main()
{
    int pid;
    /*****
    *****
    creación de un proceso concurrente con
    el creador
    *****/
    /*****/
    pid=forck ();
    if ((pid == -1) {
        printf ("error en creacion de proceso
        hijo\n");
        exit(1);
    } else
        if ( pid == 0) /* proceso hijo */ {
            printf ("Proceso hijo 1\n");
            fflush(stdout);
            exit (0); // terminacion. Retorna 0
        } else /* proceso padre */ {
            printf ("Proceso padre\n");
            fflush(stdout);
            wait (0);
        }
}
```

Proceso Hijo

```
# include <stdio.h>
main()
{
    int pid;
    /*****
    *****
    creación de un proceso concurrente con
    el creador
    *****/
    /*****/
    pid=forck ();
    if ((pid == -1) {
        printf ("error en creacion de proceso
        hijo\n");
        exit(1);
    } else
        if ( pid == 0) /* proceso hijo */ {
            printf ("Proceso hijo 1\n");
            fflush(stdout);
            exit (0); // terminacion. Retorna 0
        } else /* proceso padre */ {
            printf ("Proceso padre\n");
            fflush(stdout);
            wait (0);
        }
}
```

Memoria

Padre

Hijo

Ambos procesos continúan ejecutando la siguiente sentencia al fork().

2. Procesos Pesados (IV)

- Ejemplo de uso de fork (III)

Proceso Padre

```
# include <stdio.h>
main()
{
    int pid;
    /*****
    creación de un proceso concurrente con
    el creador
    *****/
    /*****/
    pid=forck ();
    if ((pid == -1) {
        printf ("error en creacion de proceso
        hijo\n");
        exit(1);
    } else
        if ( pid == 0) /* proceso hijo */ {
            printf ("Proceso hijo 1\n");
            fflush(stdout);
            exit (0); // terminacion. Retorna 0
        } else /* proceso padre */ {
            printf ("Proceso padre\n");
            fflush(stdout);
            wait (0);
        }
}
```

Proceso Hijo

```
# include <stdio.h>
main()
{
    int pid;
    /*****
    creación de un proceso concurrente con
    el creador
    *****/
    /*****/
    pid=forck ();
    if ((pid == -1) {
        printf ("error en creacion de proceso
        hijo\n");
        exit(1);
    } else
        if ( pid == 0) /* proceso hijo */ {
            printf ("Proceso hijo 1\n");
            fflush(stdout);
            exit (0); // terminacion. Retorna 0
        } else /* proceso padre */ {
            printf ("Proceso padre\n");
            fflush(stdout);
            wait (0);
        }
}
```

Memoria

Padre

Hijo

Ambos procesos pasan a comprobar el valor devuelto por fork().

2. Procesos Pesados (V)

- Ejemplo de uso de fork (IV)

Proceso Padre

```
# include <stdio.h>
main()
{
    int pid;
    /*****
    *****/
    creación de un proceso concurrente con
    el creador
    *****/
    /*****/
    pid=forck ();
    if ((pid == -1) {
        printf ("error en creacion de proceso
        hijo\n");
        exit(1);
    } else
        if ( pid == 0) /* proceso hijo */ {
            printf ("Proceso hijo 1\n");
            fflush(stdout);
            exit (0); // terminacion. Retorna 0
        } else /* proceso padre */ {
            printf ("Proceso padre\n");
            fflush(stdout);
            wait (0);
        }
}
```

El padre ve que fork() devuelve un valor distinto de 0 (PID del hijo) y sigue la rama del *if* correspondiente.

Proceso Hijo

```
# include <stdio.h>
main()
{
    int pid;
    /*****
    *****/
    creación de un proceso concurrente con
    el creador
    *****/
    /*****/
    pid=forck ();
    if ((pid == -1) {
        printf ("error en creacion de proceso
        hijo\n");
        exit(1);
    } else
        if ( pid == 0) /* proceso hijo */ {
            printf ("Proceso hijo 1\n");
            fflush(stdout);
            exit (0); // terminacion. Retorna 0
        } else /* proceso padre */ {
            printf ("Proceso padre\n");
            fflush(stdout);
            wait (0);
        }
}
```

El hijo ve que fork() devuelve un 0 y sigue la otra rama del *if*.

Memoria

Padre

Hijo

2. Procesos Pesados (VI)

- Identidades de padre e hijo
 - El proceso padre sabe de forma explícita la identidad de sus procesos hijo
 - Es el valor devuelto por `fork()`.
 - El proceso hijo observa que `fork()` devuelve un 0
 - 0 no es un PID válido, por lo que se identifica como proceso hijo.
 - Los procesos hijos pueden tener a su vez sus propios hijos.
 - Se establece una jerarquía de procesos debida a la relación padre/hijo.
 - Funciones relacionadas:
 - Para saber su propio PID: `getpid()`. Para saber el PID del padre: `getppid()`;
 - Para no tener problemas, debemos incluir los prototipos que están en :
`#include <sys/types.h> y #include <unistd.h>`
- Cuando en UNIX un proceso termina, todos sus hijos pasan a serlo del *init*.
- Para hacer que el padre espere a que el hijo termine (y no muera antes que su hijo), se debe usar la función **wait()**.
 - Esta función devuelve el PID del hijo terminado.
 - Si tiene varios hijos, *wait()* desbloquea al padre cuando **cualquier** hijo termina.
 - Si tenemos que esperar a que finalicen todos, habrá que poner un `wait()` por cada hijo.
 - Con la función *waitpid()* se puede esperar a la finalización de un hijo particular (el padre conoce explícitamente los PIDs de sus hijos).

2.1 exec()

- Si `fork()` sólo crea clones del padre, ¿cómo crear procesos que realicen tareas totalmente diferentes a las de su progenitor?
 - Ej. Cada vez que ejecutamos un comando el proceso *shell* realiza un `fork()`, es decir, crea un clon de si mismo, sin embargo, sabemos que el proceso hijo ejecutado es diferente al shell y se corresponde al código de cada comando.
 - Vamos a ver cómo se crean procesos independientes del padre en UNIX.
- Hay varias llamadas al S.O. que sustituyen la imagen del código (segmento de texto) que ha heredado el hijo por otra, permitiendo que padre e hijo sean programas completamente diferentes. Estas llamadas son:
 - `execl (camino, arg1, arg2,...)`
 - `execlp(·)`
 - `execle(·)`
 - `execv(·)`
 - `execvp(·)`

2.1 exec()

- Ejemplo.

```
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
main ()
{
    pid_t childpid;
    int status;
    if ((childpid = fork()) == -1) {
        perror ("Error al ejecutar fork");
        exit(-1);
    }
    else if (childpid == 0) { // Hijo
        printf("** Ejecutando ls desde el programa ejemplo de uso de exec() **\n");
        if (execl("/bin/ls", "ls", "-l", NULL) < 0) { // nombre, argv[0], argv[1]...
            perror("Fallo en la ejecución de ls.");
            exit(-1);
        }
        exit(0);
    }
    else if (childpid != wait(&status))
        perror("El hijo ha terminado de forma anormal.");
    exit(0);
}
```

3. Depuración de Programas (I)

- Depurar un programa que crea procesos concurrentes es mucho más complicado que con procesos secuenciales.
- Ahora en la pantalla se mezclarán los mensajes generados por todos los procesos concurrentes, y deberá identificarse adecuadamente quién es el proceso que generó cada uno de ellos.
- Hay que garantizar la causalidad en el orden de los mensajes
 - En un principio es imposible saber el orden en que se van a ejecutar los procesos concurrentes,
 - el orden depende de las velocidades relativas de ejecución de los procesos, de la carga del sistema, del algoritmo de planificación, y de otras circunstancias.
 - La función **fflush(*fichero*)** vacía el búffer asociado a *fichero*.
 - Si a cada sentencia que visualiza un mensaje (ej., usando printf o puts) la sigue una sentencia fflush(stdout), queda asegurado que el mensaje se imprime antes de continuar con la ejecución del resto del proceso.
 - ATENCIÓN: la pantalla (dispositivo **stdout**) es un recurso compartido por todos los procesos.
 - IMPORTANTE: el uso de fflush() penaliza la concurrencia, y sólo debe utilizarse cuando sea estrictamente necesario, como para depuración de código.

3. Depuración de Programas (II)

- Ejemplo (I)

```
printf ("msj1");
```

```
F(....);
```

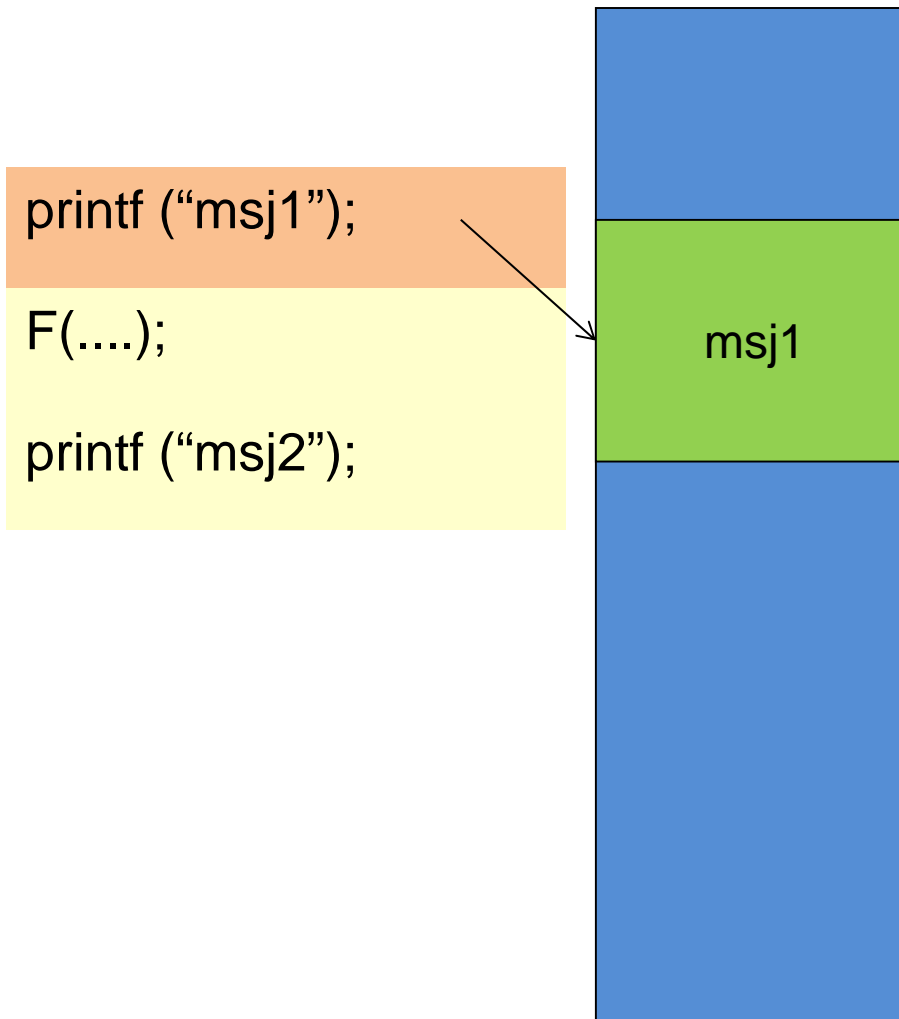
```
printf ("msj2");
```

Buffer de
salida

El uso de buffers de E/S aumenta la eficiencia del sistema, permitiendo un mayor grado de concurrencia. Es decir, el proceso puede seguir ejecutando incluso antes de que la operación de salida se haya completado.

3. Depuración de Programas (III)

- Ejemplo (II)



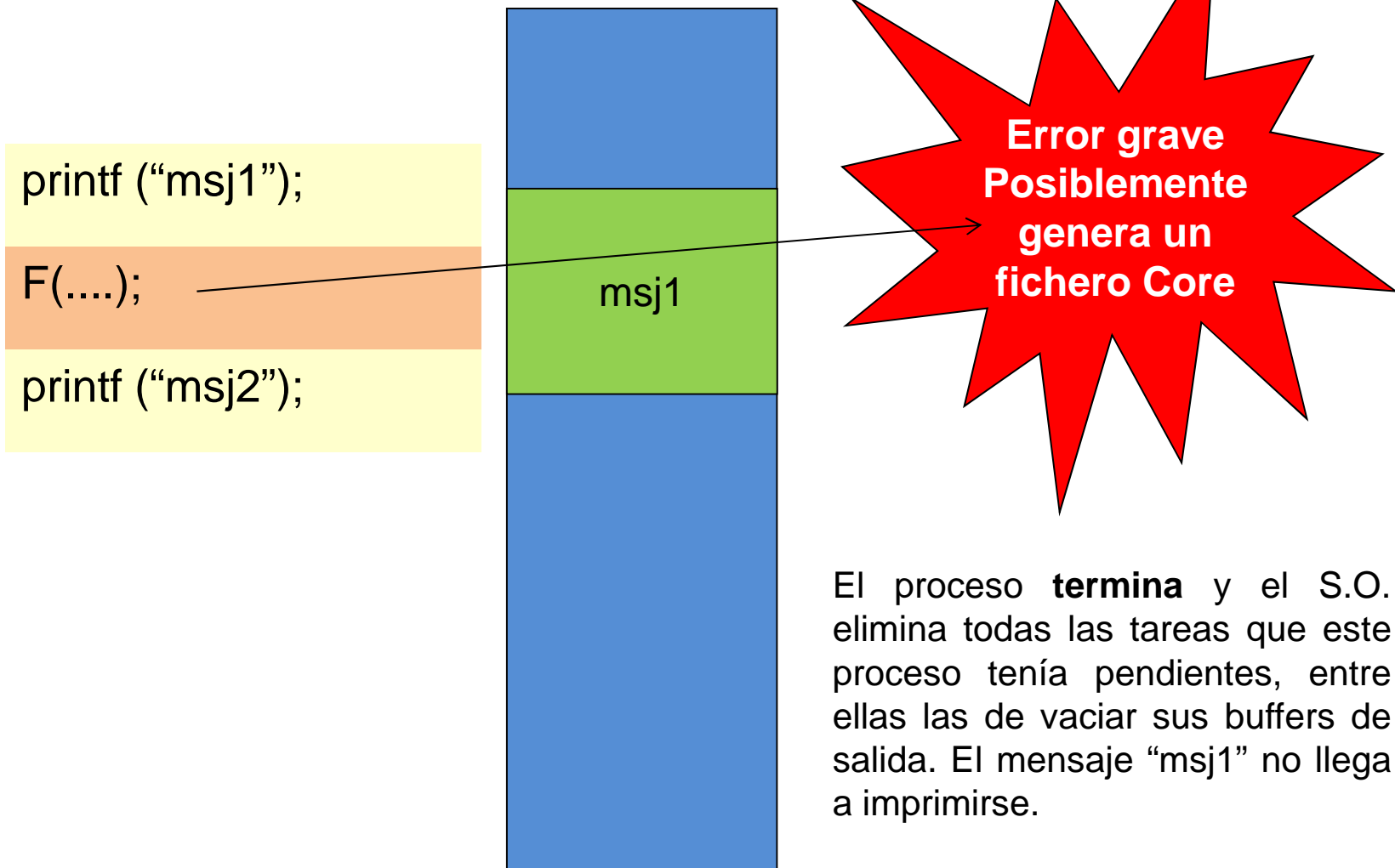
El buffer de salida no se descarga inmediatamente en el dispositivo (ej. pantalla). La escritura se hace

- Cuando está lleno
- Cuando se le indica que así lo haga
- A intervalos regulares de tiempo.
- Cuando el S.O. considera oportuno, ej. cuando no tiene otra tarea más importante que hacer

3. Depuración de Programas (IV)

UNIX (V): Concurrency

- Ejemplo (III)



3. Depuración de Programas (V)

UNIX (V): Concurrency

- Ejemplo (IV)

```
printf ("msj1"); fflush(NULL);  
  
F(.....);  
  
printf ("msj2"); fflush(NULL);
```



3. Depuración de Programas (VI)

UNIX (V): Concurrency

- Ejemplo (V)

```
printf ("msj1"); fflush(NULL);
```

```
F(....);
```

```
printf ("msj2"); fflush(NULL);
```



printf: el mensaje va al buffer

fflush: el proceso pide al S.O. que vacíe el buffer, y no continua su ejecución hasta que el S.O. haya enviado el contenido del buffer al dispositivo.

Se está penalizando la concurrencia.

3. Depuración de Programas (VII)

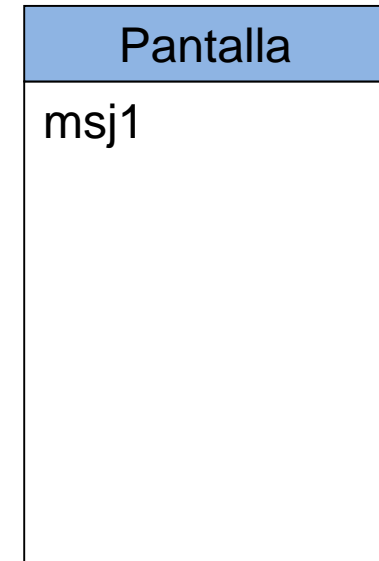
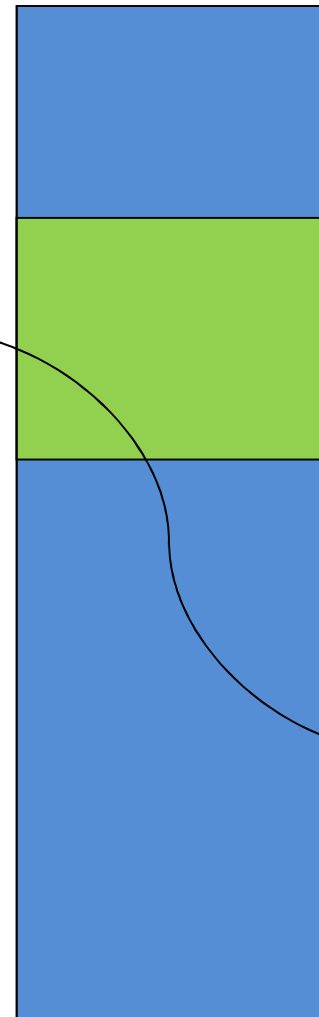
- Ejemplo (VI)

```
printf ("msj1"); fflush(NULL);
```

```
F(....);
```

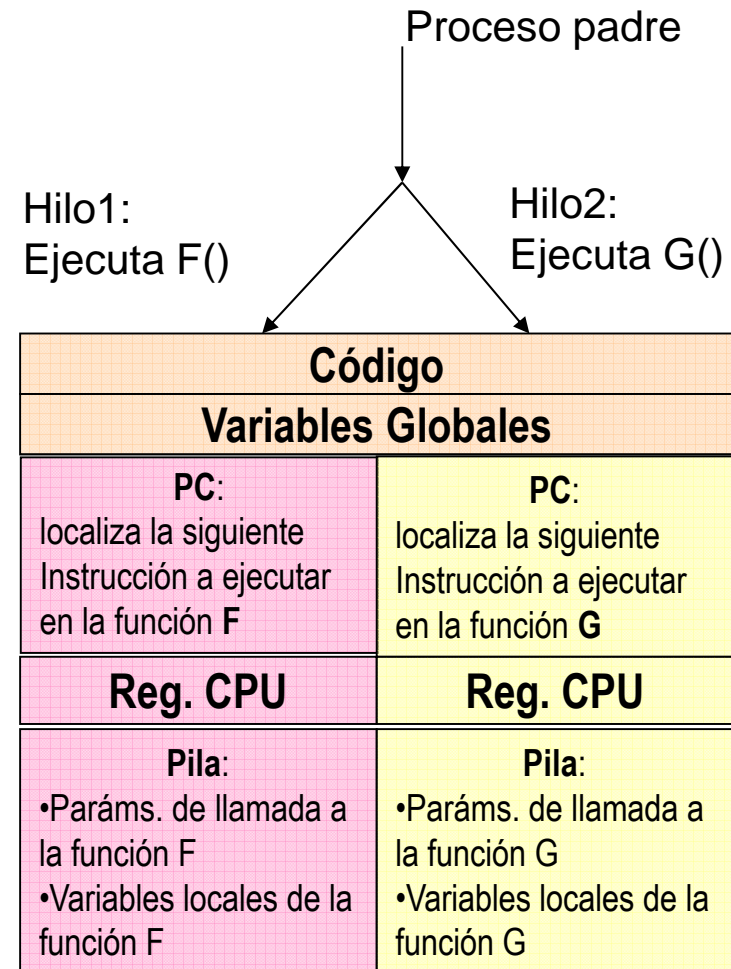
```
printf ("msj2"); fflush(NULL);
```

Al producirse el error, el proceso termina, pero ya sabemos que el error se produjo al entrar en la función F, puesto que ha aparecido "msj1" pero no "msj2".



4. Hilos (I)

- Permiten la ejecución concurrente de varias secuencias de instrucciones asociadas a diferentes funciones dentro de un mismo proceso
- Los hilos hermanos entre sí comparten la misma imagen de memoria, es decir:
 - Código,
 - variables de memoria globales, y
 - los dispositivos y ficheros que tuviera abierto el padre.
- Los hilos hermanos no comparten:
 - El Contador de Programa: cada hilo podrá ejecutar una sección distinta de código.
 - Los registros de CPU.
 - La pila en la que se crean las variables locales de las funciones a las que se va llamando después de la creación del hilo.
 - El estado: puede haber hilos en ejecución, listos, o bloqueados en espera de un evento.

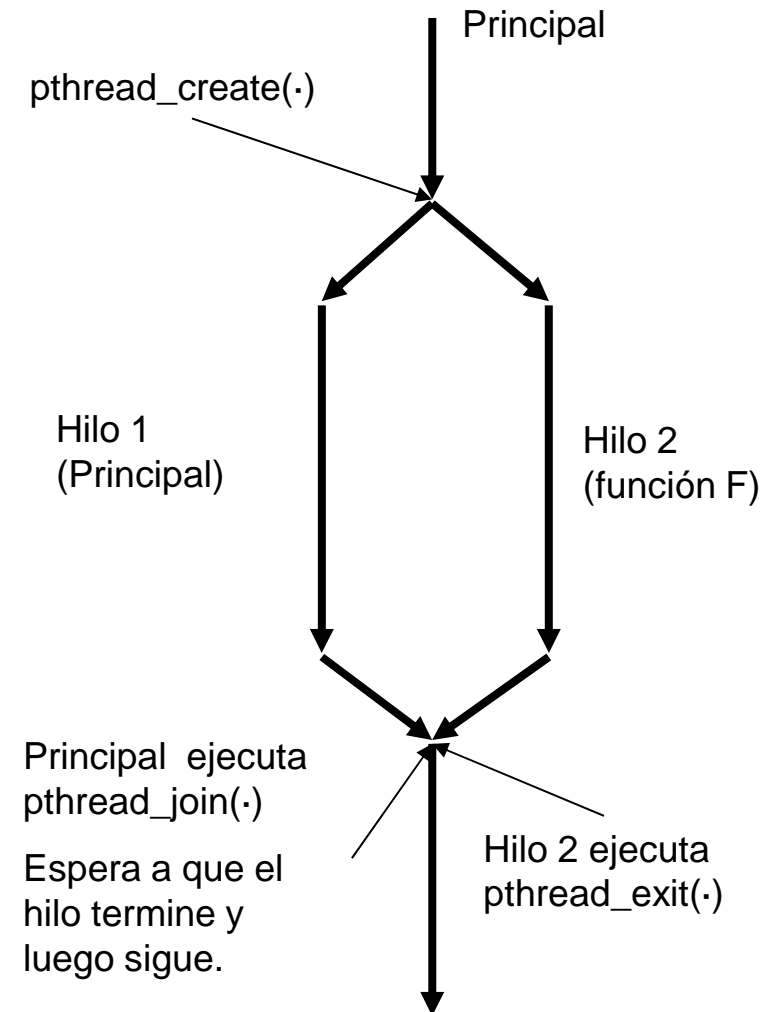


4. Hilos (II)

- Consumen menos memoria.
- Cuesta menos crearlos.
- Ya tienen disponible un mecanismo de comunicación entre ellos: las variables globales que son compartidas.
 - Si bien el uso de variables globales está totalmente desaconsejado en otras circunstancias, en concurrencia a través de hilos es el método idóneo para compartir información.
- Es más sencillo hacer un cambio de contexto entre hilos, lo que los hace ideales para la realización de tareas concurrentes cooperativas.
- La creación de hilos no es una tarea estándar de UNIX
 - Hay que incluir una biblioteca de funciones específica en la línea de compilación:
`gcc fuente.c -lpthread`
- Deben incluirse los ficheros de cabecera: `#include <pthread.h>`

4. Hilos (III)

- Llamadas al sistema para la gestión básica de hilos POSIX:
 - `pthread_create(&id, NULL, función, arg)`: crea un hilo que se asocia a la ejecución de la función que se indica en forma de puntero a función, siempre de tipo void; también se adjunta un puntero al único posible argumento de la función.
 - `pthread_exit(·)`: final de la ejecución de un hilo. Será la última sentencia ejecutada por el hilo.
 - `pthread_join(·)`: espera la finalización del hilo especificado.
 - `pthread_self(·)`: permite que un hilo se identifique a sí mismo
 - `pthread_kill(·)`: detiene la ejecución del hilo especificado



4. Hilos (IV)

- Ejemplo de creación de hilos

```
#include <stdio.h>
#include <pthread.h>

int argext[2]={2,3}; // variables Globales--> variables compartidas

void *mifuncion(void *arg) {
    printf("Soy mifuncion...\n");
    printf("Arg. ext. 1: %d. Arg. ext. 2: %d\n", argext[0], argext[1]);
    printf("Trato de modificar la variable global argext...\n");
    argext[0] = 7;
    argext[1] = 8;
    printf("Arg. ext. 1: %d. Arg. ext. 2: %d\n", argext[0], argext[1]);
    printf("Saliendo de mifuncion...\n");
    fflush(stdout); // me aseguro que el mensaje se escribe
    pthread_exit(NULL);
}

main (){
    pthread_t tid;

    printf("Creando hilo...\n"); fflush(stdout);
    pthread_create(&tid, NULL, mifuncion, (void *) NULL);
    printf("Hilo creado. Esperando su finalizacion...\n"); fflush(stdout);
    pthread_join(tid, NULL);
    printf("Hilo finalizado...\n"); fflush(stdout);
    printf("Valor de argext en hilo principal: {%d,%d}\n", argext[0],argext[1]);
}
```

4. Hilos (V)

- Atributos de los hilos.
 - Se especifican en el segundo argumento de *pthread_create(.)* .
 - Si vale *NULL* se crean los hilos con los atributos por defecto (recomendado).
 - Son características modificables de los hilos, mediante *pthread_attr_init(.)*.
 - Los atributos y sus valores por defecto son (ver manual de *pthread_attr_init*):

Attribute	Default Value	Meaning of Default
scope	PTHREAD_SCOPE_SYSTEM	resource competition within process
detach state	PTHREAD_CREATE_JOINABLE	joinable by other threads
stack address	0x40196000	stack allocated by system
stack size	0x201000 bytes	1 or 2 megabyte
sched priority	0	priority of the thread
sched policy	SCHED_OTHER	determined by system
Inherit sched	PTHREAD_EXPLICIT_SCHED	scheduling policy and parameters not inherited
guardsize	4096 bytes	size of guard area for a thread's created stack

- Atributo *scope*: permite crear hilos de usuario o hilos de sistema.
 - **Hilos de usuario** (PTHREAD_SCOPE_PROCESS) o hilo no vinculado: los diferentes hilos se asociarán a un único proceso ligero (LWP). Si uno se bloquea todos se bloquean. No aprovechan las ventajas de los multiprocesadores.
 - **Hilos de sistema** (PTHREAD_SCOPE_SYSTEM) o hilo vinculado: se creará un nuevo LWP y el hilo se asociará a él. Son gestionados, por lo tanto, por el sistema operativo.

4. Hilos (V)

- Ejemplo gestión del atributos de ámbito (scope).

```
#include <stdio.h>
#include <pthread.h>
#include <errno.h>
main () {
    pthread_t tid;
    int misargs[2];
    int tipohilo;
    pthread_attr_t atributos;
    void *mifuncion(void *arg);

    //Creo mi propia estructura de atributos para luego modificarla
    if (pthread_attr_init(&atributos) != 0) {
        perror("En creación de estructura de atributos.");
        exit(-1);
    }
    // Compruebo el campo contentionscope con pthread_attr_getscope()
    if (pthread_attr_getscope(&atributos, &tipohilo) != 0) {
        perror("En la obtención de atributos.");
        exit(-1);
    }
    printf("Atributo de ambito por defecto: %s\n", (tipohilo==PTHREAD_SCOPE_SYSTEM) ? "de
núcleo" : "de usuario");
    // Cambio el ámbito en mis atributos
    if (pthread_attr_setscope(&atributos, PTHREAD_SCOPE_SYSTEM) != 0) {
        perror("En el cambio de atributos.");
        exit(-1);
    }
    if (pthread_attr_getscope(&atributos, &tipohilo) != 0) {
        perror("En la obtención de atributos.");
        exit(-1);
    }
    printf("Nuevo atributo de ambito: %s\n", (tipohilo==PTHREAD_SCOPE_SYSTEM) ? "de núcleo" :
"de usuario");
    // Ahora ya se pueden crear los hilos con el nuevo atributo
    printf("Crear hilo...\n");
```


4. Hilos (VI)

- Paso de argumentos.
 - Cada hilo lleva asociada una función que ejecuta. Cuando finaliza esta función, el hilo termina.
 - Por definición la función es de tipo void.
 - Por definición la función siempre recibe un único argumento de tipo puntero a void
 - puntero a void: dirección de memoria que localiza un dato de tipo no definido.
 - Debemos hacer una adaptación de tipo (cast) tanto en la llamada a `pthread_create`, como en el cuerpo de la función asociada al hilo.

4. Hilos (VII)

- Ejemplo de paso de argumentos desde el padre al hilo

```
#include <stdio.h>
#include <pthread.h>

int argext[2]={2,3}; // Variables Globales--> Variables compartidas

void *mifuncion(void *arg) {
    printf("%s\n", (char*) arg); // Modifico tipo de argumento
    printf("Arg. ext. 1: %d. Arg. ext. 2: %d\n", argext[0], argext[1]);
    printf("Trato de modificar los argumentos...\n");
    argext[0] = 7;
    argext[1] = 8;
    printf("Arg. ext. 1: %d. Arg. ext. 2: %d\n", argext[0], argext[1]);
    printf("*** Saliendo de mifuncion ***\n\n");
    fflush(stdout); // me aseguro que el mensaje se escribe
    pthread_exit(NULL);
}

main (){
    pthread_t tid;
    char mensaje[] = "\n*** Empezando el hilo ***";

    printf("Creando hilo...\n"); fflush(stdout);
    pthread_create(&tid, NULL, mifuncion, (void *) mensaje); // Paso argumento
    printf("Hilo creado. Esperando su finalizacion...\n"); fflush(stdout);
    pthread_join(tid, NULL);
    printf("Hilo finalizado...\n"); fflush(stdout);
}
```

4. Hilos (VIII)

- Ejemplo paso argumentos (II) (Aviso: el código no está completo)

No seguro

```
#include <stdio.h>
#include <pthread.h>

void *hilo(void *arg) {
    int *id;
    id = (int *) arg;
    printf ("Valor pasado: %d\n", *id);
    pthread_exit (0);
}

main (){
    pthread_t tid[NUM_HILOS];

    ...

    // Creación de los hilos.
    for (i = 0; i < MAX_HILOS; i++)
        pthread_create (&tid[i], &atributos, hilo,
            (void *) &i);

    // Hilo base espera a fin de todos hilos hijo
    for (i = 0; i < MAX_HILOS; i++)
        pthread_join (tid[i], NULL);

    return 0;
}
```

Seguro

```
#include <stdio.h>
#include <pthread.h>

void *hilo(void *arg) {
    int *id =(int *) arg;
    printf ("Valor pasado: %d\n", *id);
    pthread_exit (0);
}

main (){
    pthread_t tid[NUM_HILOS];

    ...

    // Se inicializa vector.
    // Se asegura que al crear los hilos, los
    // argumentos tengan valor fijo.
    for (i = 0; i < NUM_HILOS; i++)
        I[i] = i;

    // Creación de los hilos.
    for (i = 0; i < NUM_HILOS; i++)
        pthread_create (&tid[i], &atributos, hilo,
            (void *) &I[i]);

    // Hilo base espera a fin de todos hilos hijo
    for (i = 0; i < NUM_HILOS; i++)
        pthread_join (tid[i], NULL);

    return 0;
}
```

5. Comunicación

- La comunicación entre procesos (Inter-process communication, IPC), es un conjunto de métodos que permiten el intercambio de datos y/o la coordinación entre procesos.
 - Los procesos pueden estar en la misma máquina o en distintas.
 - Aquí sólo nos centramos en el primer caso.
- Algunas soluciones UNIX:
 - Usando ficheros (No es específico de UNIX. Es la solución más simple. Poco eficiente.)
 - Mediante señales.
 - Mediante memoria compartida.
 - **Semáforos.**
 - Tuberías (pipe).
 - **Paso/Cola de mensajes.**

5.1 Paso de Mensajes

- Llamadas al sistema para la creación y gestión de la cola: `msgget()` y `msgctl()`
 - `msgctl()` permite, entre otras cosas, gestionar el número de colas, el número de mensajes en estas y el tamaño de los mensajes.
- Llamadas al sistema para el envío y recepción de mensajes: `msgsnd()` y `msgrcv()`
 - El envío (`msgsnd()`) no es bloqueante, salvo que la cola de mensajes esté llena.
 - Si la cola no está llena se añade el mensaje y se sigue con la siguiente instrucción.
 - Si la cola está llena el envío se bloquea hasta que haya espacio, salvo que en el último parámetro a `msgsnd()` se especifique lo contrario: `msgflg` vale `IPC_NOWAIT` (ver manual de `msgsnd()`)
 - La recepción puede ser bloqueante o no, dependiendo del valor de último parámetro.
 - Por defecto es bloqueante.
 - **NOTA:** ésta es una de la implementaciones en UNIX, en otras implementaciones y/o en otros sistemas operativos el envío de mensajes puede ser bloqueante, es decir, el emisor se bloquea en envío hasta que el destinatario haya recibido el mensaje.
- Estructura que define el mensaje que se intercambia


```
struct msgbuf {
    long mtype;    /* message type, must be > 0 */
    char mtext[n]; /* message data */
};
```

 - Mediante `mtype` podemos definir distintos “canales” en una misma cola.
 - El mensaje es una cadena de texto (`mtext`). El valor máximo de `n` depende de cada sistema.

5.2 Semáforos (I)

- Semáforo: variable entera compartida entre los procesos/hilos que se quiere sincronizar.
- Operaciones:
 - Inicialización (sem, valor)
 - $P(sem) \rightarrow wait(sem)$
 - $V(sem) \rightarrow signal(sem)$
- Importante: P y V son operaciones atómicas sobre la variable *sem*.
- Operación P: el proceso que la ejecuta pasa al estado bloqueado.
 - Se evita la espera activa
 - Definición (pseudocódigo):

```
typedef sem{
    int cont; // valor del semáforo
    Lista L;  // cola asociada de procesos bloqueados esperando a pasar el sem.
};
```

```
P(sem){
    sem.cont--;
    if (sem.cont < 0) {
        añadir el proceso a sem.L;
        Pedir al S.O. Que bloquee el proceso.
    }
}
```

```
V(sem) {
    sem.cont++;
    if (sem.cont <= 0) {
        // Había al menos 1 proceso esperando.
        El S.O. extrae un proceso de sem.L, lo desbloquea y lo pasa a listos.
    }
}
```

5.2 Semáforos (II)

- Semáforos POSIX.
- En el estándar POSIX hay dos tipos de semáforos:
 - **Binarios:** sólo pueden tomar valor 1 ó 0; usados generalmente para garantizar la exclusión mutua.
 - **Genéricos:** pueden tomar cualquier valor. Su uso es general (incluye el de garantizar la exclusión mutua).

5.2 Semáforos (III)

- Semáforos POSIX Genéricos
 - Pueden utilizarse para sincronizar hilos y/o procesos pesados.
 - Algunas versiones SO sólo permiten su uso con hilos.
 - Se declaran como variables de tipo `sem_t`.
 - Se manejan con las siguientes funciones:
 - Inicialización:
 - `sem_init(sem_t *sem, int pshared, unsigned int value);`
 - » `*sem`: puntero a variable semáforo (tipo `sem_t`).
 - » `pshared`: valor indicativo de si el semáforo sincronizará hilos del mismo o diferente proceso. Por defecto se pone un valor 0 ya que sólo serán usados dentro del mismo proceso.
 - » `value`: valor inicial del semáforo.
 - Sincronización:
 - `sem_wait(sem_t * sem);` → Función P
 - `sem_post(sem_t * sem);` → Función V
 - Destrucción:
 - `sem_destroy(sem_t * sem);`
 - Ficheros de cabecera:
 - `#include <semaphore.h>`

5.2 Semáforos (IV)

- Ejemplo de uso: exclusión mutua

main

```
// Listado parcial

typedef struct {
    int campo1;
    double campo2;
} _TDato;
_Tdato d ; // variable global compartida por todos los hilos
sem_t mutex ; // semáforo (también variable compartida)

main () {
    // Inicialización semáforo para exclusión mutua (valor=1)
    sem_init(&mutex, 0, 1);

    // creación de hilos
    ...

    sem_destroy(&mutex);
}
```

hilo

```
// Listado parcial

void hilo () {

    // Tareas iniciales

    sem_wait (&mutex); // Acceso excluyente

    // sección crítica
    // uso del recurso compartido d

    sem_post (&mutex); // Libera semáforo

    ...
}
```