



SISTEMAS OPERATIVOS:

Lección 5: Hilos y Procesos

Jesús Carretero Pérez
Alejandro Calderón Mateos
José Daniel García Sánchez
Francisco Javier García Blas
José Manuel Pérez Lobato
María Gregoria Casares Andrés



ADVERTENCIA

- Este material es un simple gui3n de la clase: no son los apuntes de la asignatura.
- El conocimiento exclusivo de este material no garantiza que el alumno pueda alcanzar los objetivos de la asignatura.
- Se recomienda que el alumno utilice los materiales complementarios propuestos.



- **Concepto de hilo.**
- Modelos de hilos.
- Aspectos de diseño.
- Hilos en pthreads.
- Entorno de un proceso.
- Señales.
- Temporizadores.
- Excepciones.

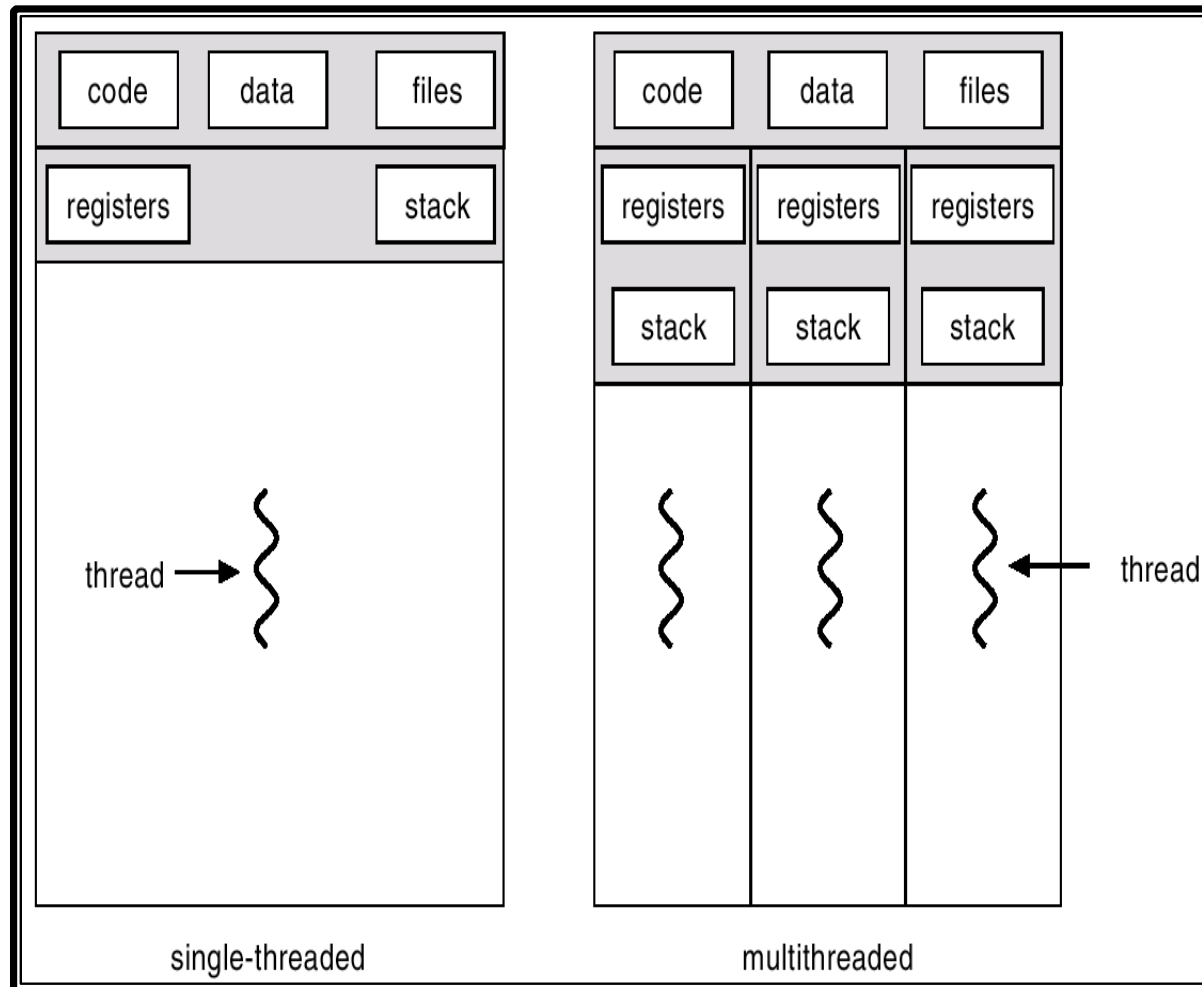


- Un proceso incluye un único hilo de ejecución.
- Diseño de aplicación con varias tareas concurrentes:
 - Un proceso receptor de peticiones y lanzar un proceso por petición.
 - Un proceso receptor y un conjunto fijo de procesos de tratamiento de peticiones.



- Consumo de tiempo en la creación y terminación de procesos.
- Consume de tiempo en cambios de contexto.
- Problemas en la compartición de recursos.

Hilo, Proceso ligero o *Thread*





- La mayoría de los modernos SO proporcionan procesos con múltiples secuencias o hilos de control en su interior.
- Se considera una unidad básica de utilización de la CPU.
- Cada uno comprende:
 - Identificador de thread
 - Contador de programa
 - Conjunto de registros
 - Pila
- Comparten con el resto de hilos del proceso:
 - Mapa de memoria (sección de código, sección de datos, shmem)
 - Ficheros abiertos
 - Señales, semáforos y temporizadores.



- Capacidad de respuesta.
 - Mayor interactividad al separar las interacciones con el usuario de las tareas de procesamiento en distintos hilos.
- Compartición de recursos.
 - Los hilos comparten la mayor parte de los recursos de forma automática.
- Economía de recursos.
 - Crear un proceso consume mucho más tiempo que crear un hilo (Ejemplo: en Solaris relación 30 a 1).
- Utilización sobre arquitecturas multiprocesador.
 - Mayor nivel de concurrencia asignando distintos hilos a distintos procesadores.
 - La mayoría de los sistemas operativos modernos usan el hilo como unidad de planificación.



Soporte de hilos

- Espacio de usuario
- ULT – User Level Threads

- Espacio de núcleo
- KLT – Kernel Level Threads

- Implementados en forma de biblioteca de funciones en espacio de usuario.
- El kernel no tiene conocimiento sobre ellos, no ofrece soporte de ningún tipo.
- Es mucho más rápido pero presentan algunos problemas → Llamadas al sistema bloqueantes.

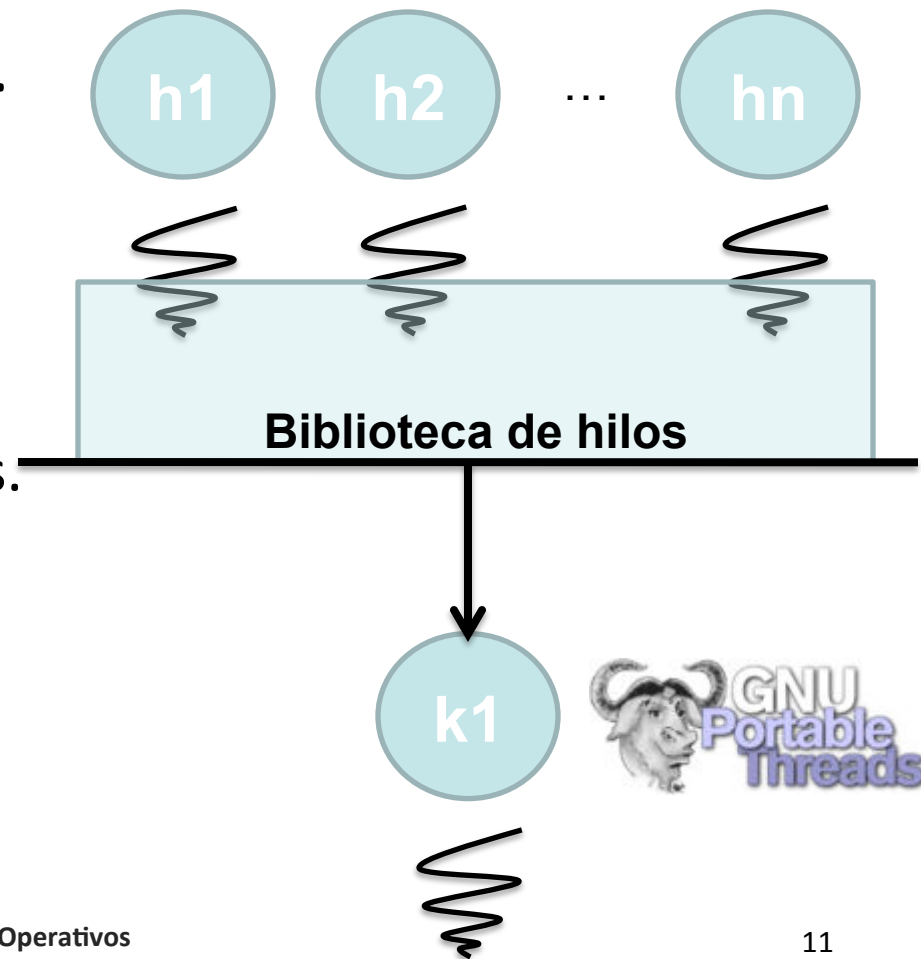
- El kernel se ocupa de crearlos, planificarlos y destruirlos.
- Es un poco más lento ya que hacemos participar al kernel y esto supone un cambio de modo de ejecución.
- En llamadas al sistema bloqueantes sólo se bloquea el thread implicado.
- En sistemas SMP, varios threads pueden ejecutarse a la vez.
- No hay código de soporte para thread en las aplicaciones.
- El kernel también puede usar threads para llevar a cabo sus funciones.



- Concepto de hilo.
- **Modelos de hilos.**
- Aspectos de diseño.
- Hilos en pthreads.
- Entorno de un proceso.
- Señales.
- Temporizadores.
- Excepciones.

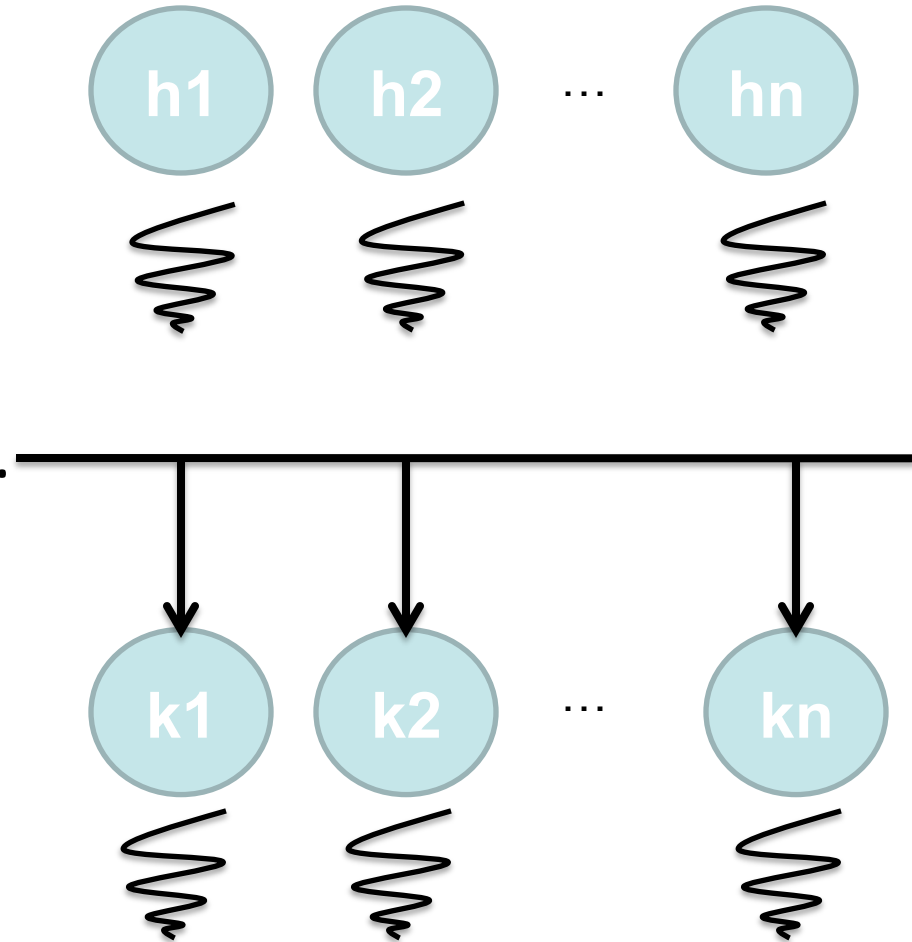
Modelos de múltiples hilos: Muchos a uno

- Hace corresponder múltiples hilos de usuario a un único hilo del núcleo.
- Biblioteca de hilos en espacio de usuario.
- Llamada bloqueante:
 - Se bloquean todos los hilos.
- En multiprocesadores no se pueden ejecutar varios hilos a la vez.



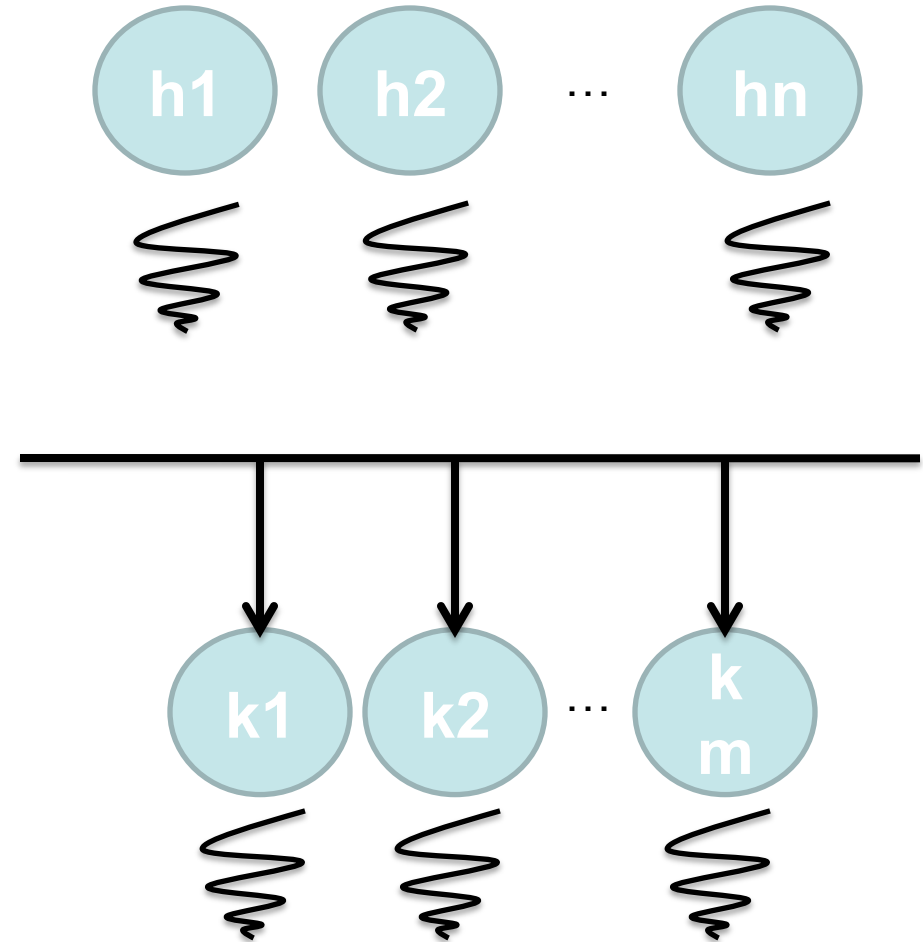
Modelo de múltiples hilos: Uno a uno

- Hace corresponder un hilo del kernel a cada hilo de usuario.
- La mayoría de las implementaciones restringen el número de hilos que se pueden crear.
- Ejemplos:
 - Linux 2.6.
 - Windows.
 - Solaris 9.



Modelos de múltiples hilos: Muchos a muchos

- Este modelo multiplexa los threads de usuario en un número determinado de threads en el kernel.
- El núcleo del sistema operativo se complica mucho.
- Ejemplos:
 - Solaris (versiones anteriores a 9).
 - HP-UX.
 - IRIX.





- Concepto de hilo.
- Modelos de hilos.
- **Aspectos de diseño.**
- Hilos en pthreads.
- Entorno de un proceso.
- Señales.
- Temporizadores.
- Excepciones.



- En los sistemas tipo UNIX ¿Qué se debe hacer si se llama a fork desde un hilo?
 - Duplicar el proceso con todos sus hilos.
 - Apropiado si no se va a llamar luego a exec para sustituir la imagen del proceso.
 - Duplicar el proceso solo con el hilo que llama a fork.
 - Más eficiente si se va a llamar a exec y se van a cancelar todos los hilos.
- **Solución en Linux: Dos versiones de fork.**



- Situación en la que un hilo notifica a otros que deben terminar.
- Opciones:
 - Cancelación asíncrona: Se fuerza la terminación inmediata del hilo.
 - Problemas con los recursos asignados al hilo.
 - Cancelación diferida: El hilo comprueba periódicamente si debe terminar.
 - Preferible.



- Las aplicaciones que reciben peticiones y las procesan pueden usar hilos para el tratamiento.
- Pero:
 - El tiempo de creación/destrucción del hilo supone un retraso (aunque sea menor que el de creación/destrucción de un proceso).
 - No se establece un límite en el número de hilos concurrentes.
 - Si llega una avalancha de peticiones se pueden agotar los recursos del sistema.



- Se crea un conjunto de hilos que quedan en espera a que lleguen peticiones.
- Ventajas:
 - Se minimiza el retardo: El hilo ya existe.
 - Se mantiene un límite sobre el número de hilos concurrentes.



- Concepto de hilo.
- Modelos de hilos.
- Aspectos de diseño.
- **Hilos en pthreads.**
- Entorno de un proceso.
- Señales.
- Temporizadores.
- Excepciones.



- `int pthread_create(pthread_t *thread,
const pthread_attr_t *attr,
void *(*func)(void *),
void *arg)`
 - Crea un hilo e inicia su ejecución.
 - **thread**: Se debe pasar la dirección de una variable del tipo **pthread_t** que se usa como manejador del hilo.
 - **attr**: Se debe pasar la dirección de una estructura con los atributos del hilo. Se puede pasar NULL para usar atributos por defecto.
 - **func**: Función con el código de ejecución del hilo.
 - **arg**: Puntero al parámetro del hilo. Solamente se puede pasar un parámetro.
- `pthread_t pthread_self(void)`
 - Devuelve el identificador del thread que ejecuta la llamada.



- `int pthread_join(pthread_t thread, void **value)`
 - El hilo que invoca la función se espera hasta que el hilo cuyo manejador se especifique haya terminado.
 - **thread**: Manejador de del hilo al que hay que esperar.
 - **value**: Valor de terminación del hilo.
- `int pthread_exit(void *value)`
 - Permite a un proceso ligero finalizar su ejecución, indicando el estado de terminación del mismo.
 - El estado de terminación no puede ser un puntero a una variable local.



Ejemplo

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

struct sumapar {
    int n, m, r;
};
typedef struct sumapar sumapar_t;

void suma(sumapar_t * par) {
    int i;
    int suma=0;
    for (i=par->n; i<=par->m; i++) {
        suma +=i;
    }
    par->r=suma;
}

int main() {
    pthread_t th1, th2;
    sumapar_t s1 = {1,50,0};
    sumapar_t s2 = {51,100,0};

    pthread_create(&th1, NULL,
        (void*)suma, (void*)&s1);
    pthread_create(&th2, NULL,
        (void*)suma, (void*)&s2);

    pthread_join(th1, NULL);
    pthread_join(th2, NULL);

    printf("Suma=%d\n",
        s1.r+s2.r);
}
```



- Cada hilo tiene asociados un conjunto de atributos.
- Atributos representados por una variable de tipo **pthread_attr_t**.
- Los atributos controlan:
 - Un hilo es independiente o dependiente.
 - El tamaño de la pila privada del hilo.
 - La localización de la pila del hilo.
 - La política de planificación del hilo.



- `int pthread_attr_init(pthread_attr_t * attr);`
 - Inicia una estructura de atributos de hilo.
- `int pthread_attr_destroy(pthread_attr_t * attr);`
 - Destruye una estructura de atributos de hilo.
- `int pthread_attr_setstacksize(pthread_attr_t * attr, int stacksize);`
 - Define el tamaño de la pila para un hilo
- `int pthread_attr_getstacksize(pthread_attr_t * attr, int *stacksize);`
 - Permite obtener el tamaño de la pila de un hilo.



- `int pthread_attr_setdetachstate(pthread_attr_t *attr, int detachstate)`
 - Establece el estado de terminación de un proceso ligero.
 - Si "detachstate" = PTHREAD_CREATE_DETACHED el proceso ligero liberara sus recursos cuando finalice su ejecución.
 - Si "detachstate" = PTHREAD_CREATE_JOINABLE no se liberan los recursos, es necesario utilizar pthread_join().
- `int pthread_attr_getdetachstate(pthread_attr_t *attr, int *detachstate)`
 - Permite conocer el estado de terminación



Ejemplo: Hilos independientes

```
#include <stdio.h>
#include <pthread.h>
#define MAX_THREADS 10
void func(void) {
    printf("Thread %d \n", pthread_self());
    pthread_exit(0);
}

int main() {
    int j;
    pthread_attr_t attr;
    pthread_t thid[MAX_THREADS];
    pthread_attr_init(&attr);
    pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
    for(j = 0; j < MAX_THREADS; j++)
        pthread_create(&thid[j], &attr, func, NULL);
    sleep(5);
}
```



- Concepto de hilo.
- Modelos de hilos.
- Aspectos de diseño.
- Hilos en pthreads.
- **Entorno de un proceso.**
- Señales.
- Temporizadores.
- Excepciones.



- Mecanismo de paso de información a un proceso.
- Conjunto e pares `<nombre, valor>`.

- Ejemplo

```
PATH=/usr/bin:/home/pepe/bin
```

```
TERM=vt100
```

```
HOME=/home/pepe
```

```
PWD=/home/pepe/libros/primer
```

```
TIMEZONE=MET
```



- El entorno de un proceso se coloca en la pila del proceso al iniciarlo.
- Acceso:
 - El sistema operativo coloca algunos valores por defecto (p. ej. PATH).
 - Acceso mediante mandatos (set, export).
 - Acceso mediante API de SO (putenv, getenv).



- Un proceso recibe como tercer parámetro de main la dirección de la tabla de variables de entorno.

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

```
int main(int argc, char** argv, char** envp) {
    for (int i=0;envp[i]!=NULL;i++) {
        printf("%s\n",envp[i]);
    }
    return 0;
}
```



- `char * getenv(const char * var);`
 - Obtiene el valor de una variable de entorno.
- `int setenv(const char * var, const char * val, int overwrite);`
 - Modifica o añade una variable de entorno.
- `int putenv(const char * par);`
 - Modifica o añade una asignación `var=valor`



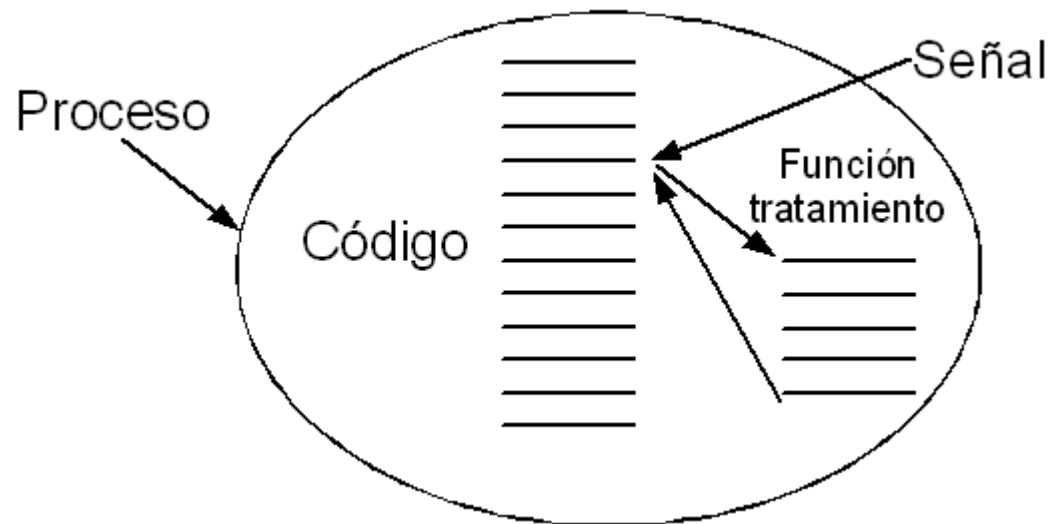
- Concepto de hilo.
- Modelos de hilos.
- Aspectos de diseño.
- Hilos en pthreads.
- Entorno de un proceso.
- **Señales.**
- Temporizadores.
- Excepciones.



- Son un mecanismo que permite avisar a un proceso de la ocurrencia de un evento.
- Ejemplos:
 - Un proceso padre recibe la señal SIGCHLD cuando termina un proceso hijo.
 - Un proceso recibe una señal SIGILL cuando intenta ejecutar una instrucción máquina ilegal.

Son un mecanismo propio de los sistemas UNIX

- Las señales son interrupciones al proceso
- Envío o generación
 - Proceso- Proceso (dentro del grupo) con el kill
 - SO - Proceso





- Otras señales
 - SIGILL instrucción ilegal
 - SIGALRM vence el temporizador
 - SIGKILL mata al proceso
- Cuando un proceso recibe una señal:
 - Si está en ejecución: Detiene su ejecución en la instrucción máquina actual.
 - Si existe una rutina de tratamiento de la señal: Bifurcación para ejecutar la rutina de tratamiento.
 - Si la rutina de tratamiento no termina el proceso: Retorno al punto en que se recibió la señal.



- Otras señales
 - SIGALRM vence el temporizador
 - SIGKILL mata al proceso
- El SO las transmite al proceso
 - El proceso debe estar preparado para recibirla
 - Especificando un procedimiento de señal con sigaction.
 - Enmascarando la señal con sigprocmask.
 - Si no está preparado → acción por defecto
 - El proceso, en general, muere.
 - Hay algunas señales que se ignoran o tienen otro efecto.
- El servicio pause para el proceso hasta que recibe una señal



- `int kill(pid_t pid, int sig)`
 - Envía al proceso "pid" la señal "sig".
 - Casos especiales:
 - `pid=0` → Señal a todos los procesos con gid igual al gid del proceso.
 - `pid < -1` → Señal a todos los proceso con gid igual al valor absolute de pid.
- `int sigaction(int sig, struct sigaction *act, struct sigaction *oact)`
 - Permite especificar la acción a realizar como tratamiento de la señal "sig"
 - La configuración anterior se puede guardar en "oact".



```
struct sigaction {  
    void (*sa_handler) (); /* Manejador */  
    sigset_t sa_mask; /* Señales bloqueadas */  
    int sa_flags; /* Opciones */  
};
```

- **Manejador:**
 - SIG_DFL: Acción por defecto (normalmente termina el proceso).
 - SIG_IGN: Ignora la señal.
 - Dirección de una función de tratamiento.
- Máscara de señales a bloquear durante el manejador.
- Opciones normalmente a cero.



- `int sigemptyset(sigset_t * set);`
 - Crea un conjunto vacío de señales.
- `int sigfillset(sigset_t * set);`
 - Crea un conjunto lleno con todas la señales posibles.
- `int sigaddset(sigset_t * set, int signo);`
 - Añade una señal a un conjunto de señales.
- `int sigdelset(sigset_t * set, int signo);`
 - Borra una señal de un conjunto de señales.
- `int sigismember(sigset_t * set, int signo);`
 - Comprueba si una señal pertenece a un conjunto.



- Ignorar la señal SIGINT

- Se produce cuando se pulsa la combinación de teclas Ctrl +C

```
struct sigaction act;  
act.sa_handler = SIG_IGN;  
act.flags = 0;  
sigemptyset(&act.sa_mask);  
Sigaction(SIGINT, &act, NULL);
```




- `int pause(void)`
 - Bloquea al proceso hasta la recepción de una señal.
 - No se puede especificar un plazo para desbloqueo.
 - No permite indicar el tipo de señal que se espera.
 - No desbloquea el proceso ante señales ignoradas.
- `int sleep(unsigned int sec)`
 - Suspende un proceso hasta que vence un plazo o se recibe una señal.



- Concepto de hilo.
- Modelos de hilos.
- Aspectos de diseño.
- Hilos en pthreads.
- Entorno de un proceso.
- Señales.
- **Temporizadores.**
- Excepciones.



- El sistema operativo mantiene un temporizador por proceso (caso UNIX).
 - Se mantiene en el BCP del proceso un contador del tiempo que falta para que venza el temporizador.
 - La rutina del sistema operativo actualiza todos los temporizadores.
 - Si un temporizador llega a cero se ejecuta la función de tratamiento.
- En UNIX el sistema operativo envía una señal SIGALRM al proceso cuando vence su temporizador.



- `int alarm(unsigned int sec)`
 - Establece un temporizador.
 - Si el parámetro es cero, desactiva el temporizador.



Ejemplo: Imprimir un mensaje cada 10 segundos

```
#include <signal.h>
#include <stdio.h>
void tratar_alarma(void) {
    printf("Activada \n");
}

int main() {
    struct sigaction act;

    /* establece el manejador para SIGALRM */
    act.sa_handler = tratar_alarma;
    act.sa_flags = 0;      /* ninguna acción específica */
    sigaction(SIGALRM, &act, NULL);

    act.sa_handler = SIG_IGN;      /* ignora SIGINT */
    sigaction(SIGINT, &act, NULL);
    for(;;){      /* recibe SIGALRM cada 10 segundos */
        alarm(10);
        pause();
    }
}
```



Finalización temporizada

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

pid_t pid;

void tratar_alarma(void) {
    kill(pid, SIGKILL);
}

main(int argc, char **argv) {
    int status;
    char **argumentos;
    struct sigaction act;
    argumentos = &argv[1];
    pid = fork();
```

```
switch(pid) {
    case -1: /* error del fork() */
        perror("fork");
        exit(-1);
    case 0: /* proceso hijo */
        execvp(argumentos[0], argumentos);
        perror("exec");
        exit(-1);
    default: /* padre */
        /* establece el manejador */
        act.sa_handler = tratar_alarma;
        act.sa_flags = 0;
        sigaction(SIGALRM, &act, NULL);
        alarm(5);
        wait(&status);
}
exit(0);
}
```



- Concepto de hilo.
- Modelos de hilos.
- Aspectos de diseño.
- Hilos en pthreads.
- Entorno de un proceso.
- Señales.
- Temporizadores.
- **Excepciones.**



- Win32 ofrece una gestión estructurada de excepciones.
- Extensión al lenguaje C para gestión estructurada de excepciones.
 - No es parte de ANSI/ISO C.

```
__try {  
    /* Código principal */  
}  
  
__except (expr) {  
    /* Tratamiento de  
    excepción */  
}
```

- La expresión de `__except` debe evaluarse a:
 - **EXCEPTION_EXECUTE_HANDLER**
 - Entra en el bloque.
 - **EXCEPTION_CONTINUE_SEARCH**
 - Propaga la excepción.
 - **EXCEPTION_CONTINUE_EXECUTION**
 - Ignora la excepción.



- **DWORD GetExceptionCode()**
 - No es una llamada al sistema: Macro.
 - Solamente se puede usar dentro de tratamiento de excepciones.
 - Existe una lista larga de valores que puede devolver:
 - **EXCEPTION_ACCESS_VIOLATION**
 - **EXCEPTION_ILLEGAL_INSTRUCTION**
 - **EXCEPTION_PRIV_INSTRUCTION**
 - . . .



Copia de cadenas segura

```
LPTSTR CopiaSegura(LPTSTR s1, LPTSTR s2) {  
    __try {  
        return strcpy(s1, s2);  
    }  
    __except (GetExceptionCode() == EXCEPTION_ACCESS_VIOLATION?  
        EXCEPTION_EXECUTE_HANDLER : EXCEPTION_CONTINUE_SEARCH) {  
        return NULL;  
    }  
}
```



- Un proceso puede tener varios hilos de ejecución.
- Una aplicación multihilo consume menos recursos que una aplicación multiproceso.
- Cada sistema operativo tiene un modo de soporte de hilos entre ULT y KLT.
- PTHREADS es una biblioteca de hilos de usuario.
- Win32 ofrece hilos en el núcleo con soporte para conjuntos de hilos (*Thread Pools*).



- Las variables de entorno permiten pasar información a los procesos.
- Las señales POSIX se pueden ignorar o tratar.
- Los temporizadores tienen distinta resolución de POSIX in Win32.
- El tratamiento estructurado de excepciones permiten tratar situaciones anómalas mediante una extensión del lenguaje C.



- **Básica**

- **Carretero 2007:**

- 3.6. Señales y excepciones.
- 3.7. Temporizadores.
- 3.13. Servicios.
- 3.9 Threads

- Complementaria

- **Stallings 2005:**

- 4.1 Procesos e hilos.
- 4.4 Gestión de hilos y SMP en Windows.
- 4.5 Gestión de hilos y SMP en Solaris.
- 4.6 Gestión de procesos e hilos en Linux.

- **Silberschatz 2006:**

- 4 Hebras.



SISTEMAS OPERATIVOS:

Lección 5: Hilos y Procesos

Material Complementario



- **DWORD GetEnvironmentVariable(LPCTSTR lpszName, LPTSTR lpszValue, DWORD valueLenght);**
 - Devuelve el valor de una variable de entorno.
- **BOOL SetEnvironmentVariable(LPCTSTR lpszName, LPTSTR lpszValue);**
 - Modifica o crea una variable de entorno.
- **LPVOID GetEnvironmentStrings();**
 - Obtiene un puntero a la tabla de variables de entorno.



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

int main() {
    char * lpszVar;
    void * lpvEnv;

    lpvEnv = GetEnvironmentStrings();
    if (lpvEnv == NULL) {
        exit(-1);
    }

    char * p = lpszVar;
    while (p!=NULL) {
        printf("%s\n",p);
        while (p!=NULL) p++;
        p++;
    }

    printf("\n");
    FreeEnvironmentStrings(lpszVar);

    return 0;
}
```




- **UINT SetTimer(HWND hWnd, UINT nIDEvent, UINT uElapse, TIMERPROC lpTimerFunc);**
 - Activa un temporizador y ejecuta la función lpTimerFunc cuando venza el tiempo.
 - La función debe cumplir con:
 - **VOID TimerFunc(HWND hWnd, UINT uMsg, UINT idEvent, DWORD dwTime);**
- **BOOL KillTimer(HWND hWnd, UINT uIdEvent);**
 - Desactiva un temporizador.
- **VOID Sleep(DWORD dwMilliseconds);**
 - Hace que el hilo actual se suspenda durante un cierto tiempo.



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

```
VOID Mensaje (HWND, UINT, UINT, DWORD) {
    printf("Tiempo finalizado");
}
```

```
int main() {
    tid = SetTimer(NULL, 2, 10, Mensaje); /* 2 msec */
    realizar_tarea();
    KillTimer(NULL, tid);
    return 0;
}
```