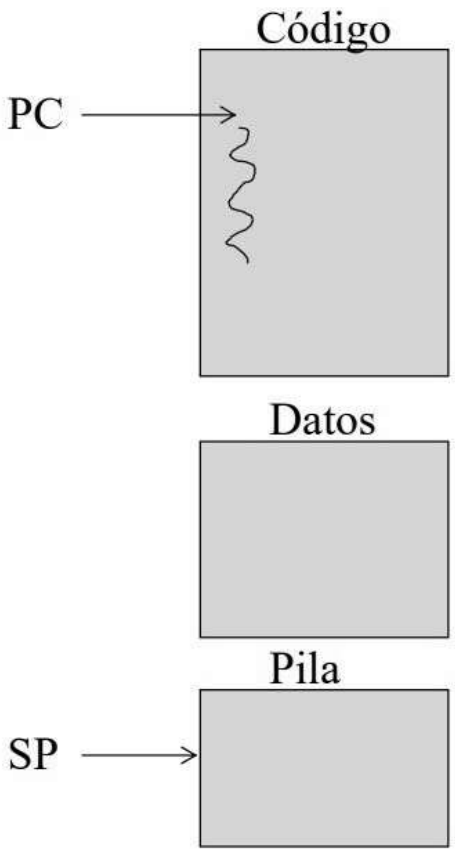
Gestión de threads (hilos)

Sistemas
Operativos
Pablo
Ibáñez

Modelo de proceso



- Programación/ejecución secuencial
  - Simple
  - Válida para resolver muchos problemas
- En algunos casos es un modelo muy limitado. Ejemplos
  - Aplicación con varias actividades concurrentes de E/S y/o cálculo
  - Disponibilidad de varios procesadores
- Alternativa: programación/ejecución concurrente
  - Pensemos en varios procesos

Ejemplo 1: servidor de ficheros en red

- Aplicación con E/S concurrentes
  - Recibe peticiones (read, write, …) de procesos de otras máquinas
  - realiza las operaciones sobre disco y responde las peticiones
- Versión secuencial simple: servicio de peticiones en serie
  - Bucle que lee petición, realiza llamada al sistema bloqueante, y responde
  - Muy bajas prestaciones
- Versión secuencial optimizada: código complejo
  - Usa llamadas al sistema no bloqueantes para recibir peticiones y para realizar las operaciones en disco
  - guarda memoria de las operaciones pendientes de respuesta, que posiblemente llegarán en desorden
- Solución concurrente
  - Cada petición crea un proceso que realiza la operación sobre disco y responde

Ejemplo 2: word

- Aplicación con varias actividades concurrentes: entrada de texto, corrector ortográfico, formato en pantalla
- Cada vez que se introduce nuevo texto desde teclado hay que ejecutar el código del corrector ortográfico. Opciones:
  - Se ejecuta corrector y después se continúa: se desatiende la lectura de teclado durante un tiempo
  - Se ejecuta corrector a la vez que se atiende teclado: se complica mucho la programación de la aplicación
- Además, word tiene que realizar otras tareas como la organización del texto en pantalla
  - espaciado entre caracteres, líneas y párrafos
  - formatos de letra, …
- Solución concurrente: cada actividad un proceso
  - Cada actividad se programa de forma independiente
  - Se despierta solo cuando se requiere, distintas prioridades, ...

Ejemplo 3: cálculo masivo

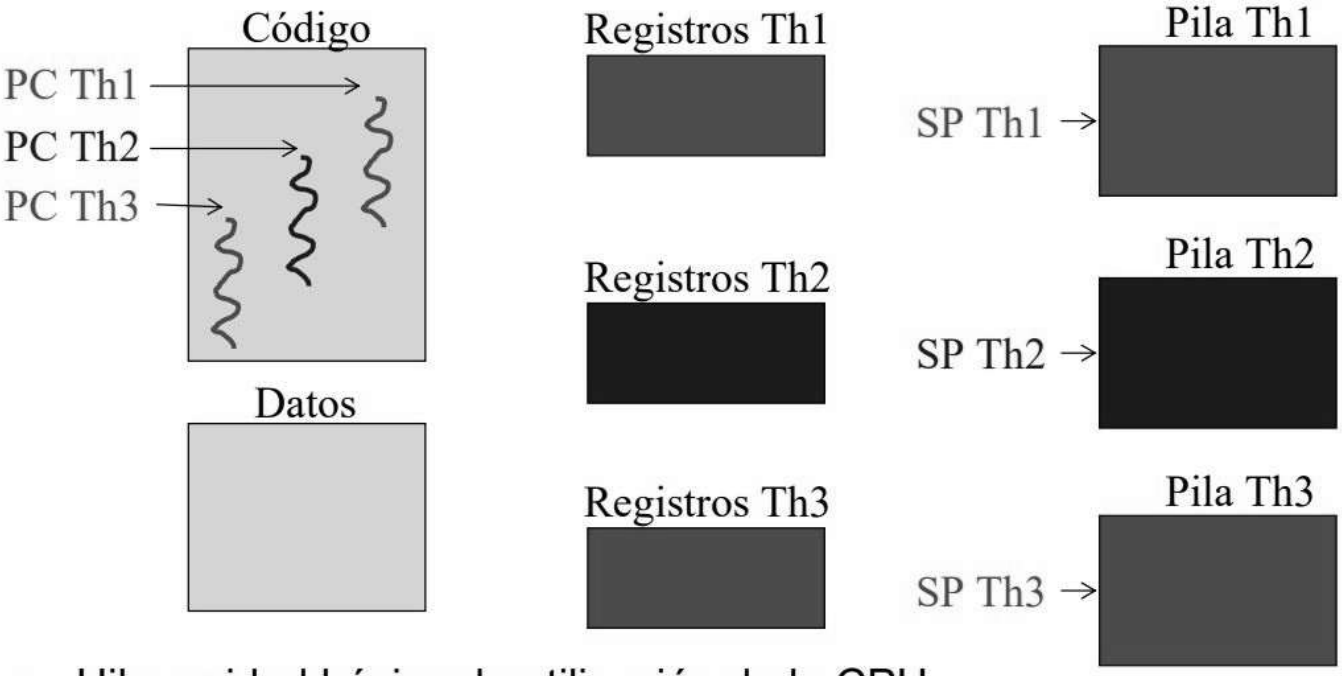
- Ejemplo: sumar los elementos de un vector muy grande
- Disponibilidad de varios procesadores
- Objetivo: aumentar rendimiento
  - Repartir el trabajo entre varios procesos
  - Cada proceso trabaja sobre un trozo del vector
  - Cada proceso ejecuta en un procesador
- Ejemplos 1 y 2
  - independientes de la plataforma, sirven para uno o varios cores
  - Objetivo principal: facilitar la programación
- Ejemplo 3
  - Solo tiene sentido en un sistema multicore,
  - Casi siempre con numero de threads ≤ numero de cores
  - Objetivo: rendimiento
  - Complica la programación

Procesos e hilos

- Los procesos (tal como los hemos visto hasta ahora):
  - **Son propietarios de recursos** – espacio de memoria para almacenar su imagen, ficheros abiertos, tratamiento de señales, …
  - **Son la unidad de ejecución/planificación** – siguen un camino de ejecución, el SO les otorga tiempo en CPU

- En el modelo proceso/hilo, estas dos características se tratan de forma independiente por el sistema operativo
  - Los procesos **son propietarios de recursos**
  - Los hilos **son la unidad de ejecución/planificación**
  - Un proceso puede tener varios hilos

Modelo de proceso con varios hilos



- Hilo: unidad básica de utilización de la CPU
  - Comprende un ID de hilo, un PC, un conjunto de registros y una pila
  - Comparte con otros hilos: código, datos y recursos de sistema asignados al programa (ficheros abiertos, señales, …)

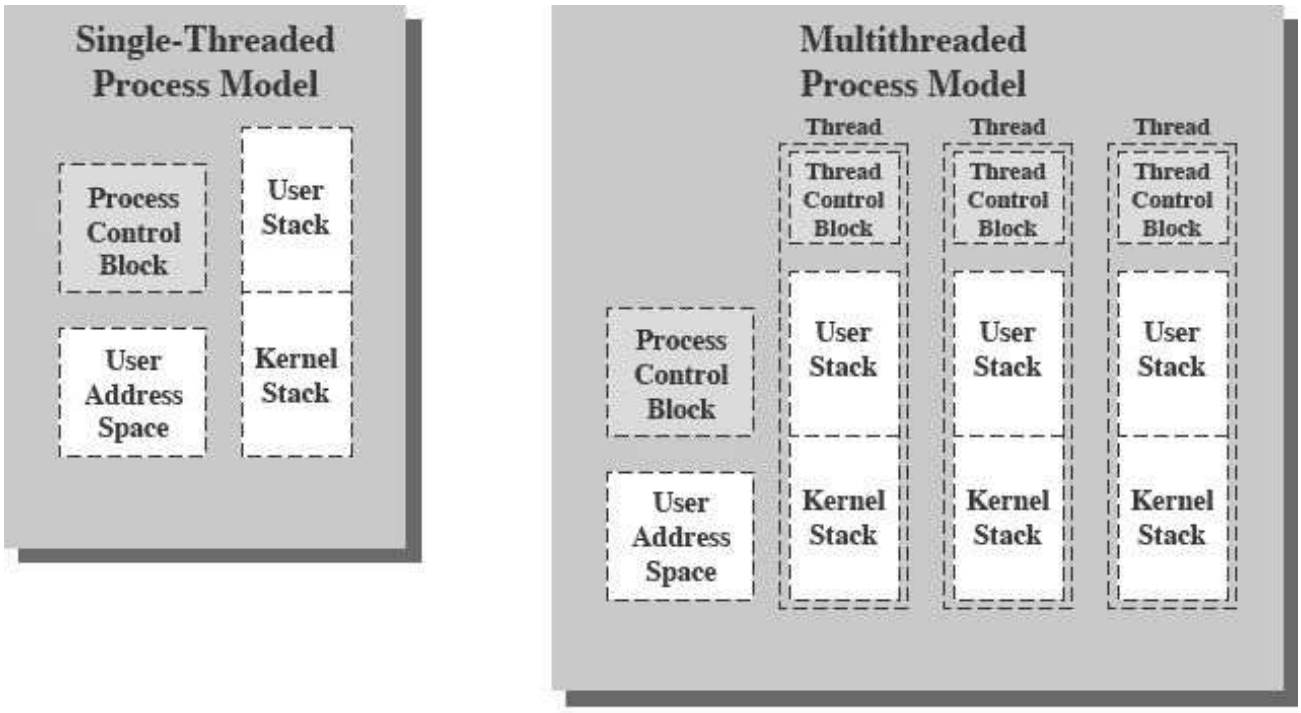
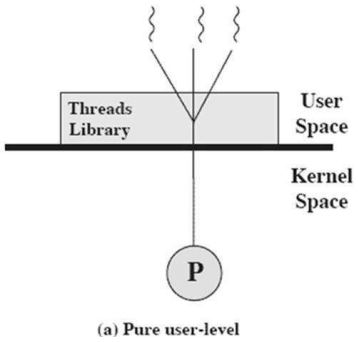


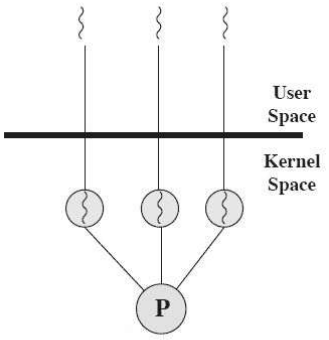
Figure 4.2 Single Threaded and Multithreaded Process Models

Formas de implementar hilos

- Hilos de usuario
  (User Level Thread, ULT)
  - La gestión de hilos la lleva la propia aplicación (biblioteca)
  - El SO no da ningún soporte, no conoce los hilos
- Hilos de sistema
  (Kernel level Thread, KLT)
  - también llamados: kernel-supported threads, lightweight processes
  - El SO conoce y gestiona hilos, planifica a nivel de hilo



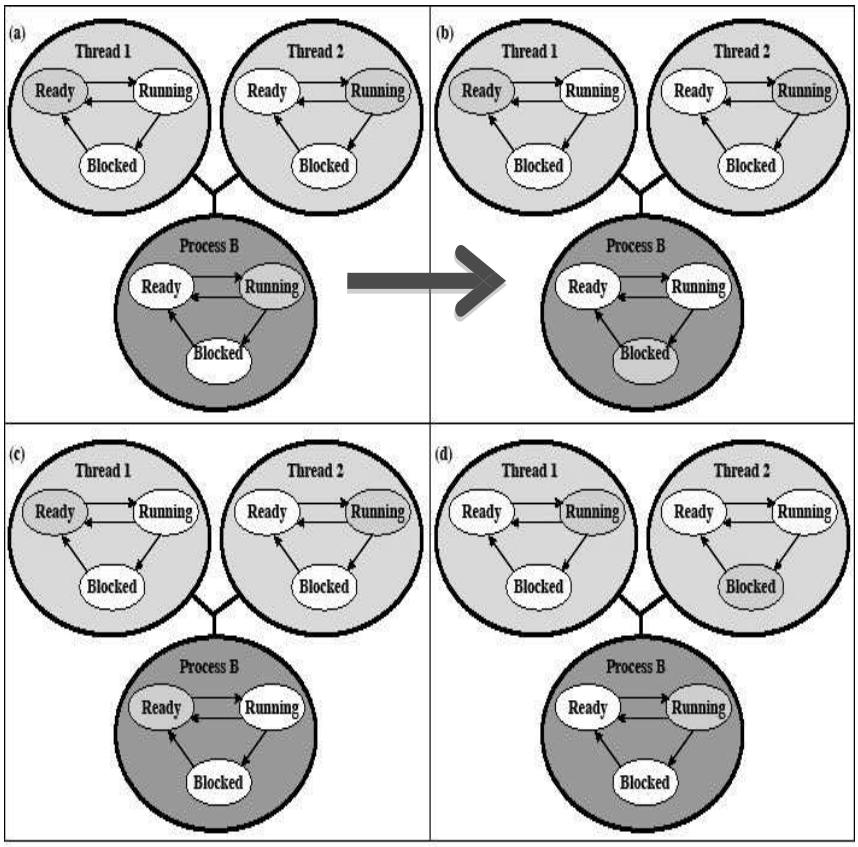
(a) Pure user-level



(b) Pure kernel-level

Hilos de usuario: planificación en dos niveles

Situación inicial



Colored state
is current state

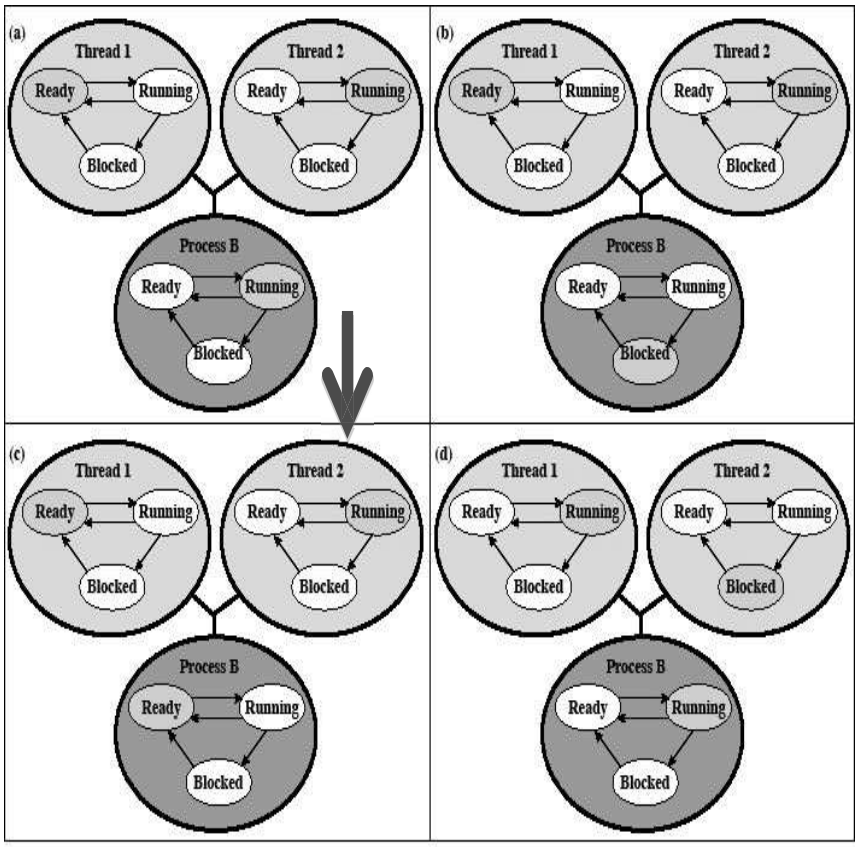
Figure 4.7 Examples of the Relationships Between User-Level Thread States and Process States

Th2 SC
Bloquea
Al proceso B

Hilos de usuario: planificación en dos niveles

Situación inicial

Quantum proceso B

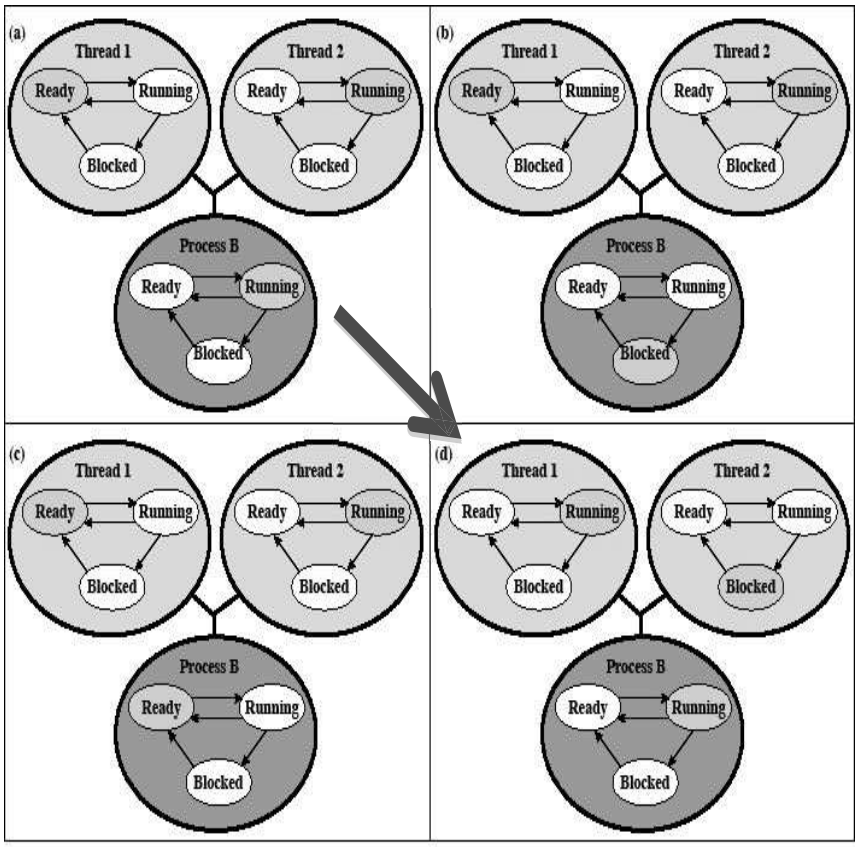


Colored state
is current state

Figure 4.7 Examples of the Relationships Between User-Level Thread States and Process States

Hilos de usuario: planificación en dos niveles

Situación inicial



Colored state
is current state

Figure 4.7 Examples of the Relationships Between User-Level Thread States and Process States

Th2 sincroniza con Th1 y se Bloquea

Hilos de sistema: Ventajas/Desventajas

- Si un hilo se bloquea, no bloquea a todo el proceso. El kernel puede poner en ejecución otros hilos del mismo proceso
- En un sistema multiprocesador, el kernel puede poner simultáneamente en ejecución varios hilos del mismo proceso
- Tienen mayor coste en cambio de contexto de hilo
- Tienen mayor coste para crear, señalizar/esperar, …

Latencia en uSg. VAX/UNIX	ULT	KLT	proceso
Crear	34	948	11.300
Señalizar-esperar	37	441	1.840

Bibliotecas de hilos

- Contienen la colección de funciones ofrecidas al usuario para trabajar con hilos
  - Crear, terminar, funciones de sincronización,…
- Ejemplos:
  - POSIX Pthreads (existe como ULT y KLT)
  - Windows Threads (KLT)
  - Java Threads (se implementa sobre la librería del host)
- Veremos Pthreads en el siguiente tema

Hilos en Linux

- Además de fork(), linux ofrece llamada clone()
- Clone() permite determinar el grado de compartición entre padre e hijo

flag	meaning
CLONE_FS	File-system information is shared.
CLONE_VM	The same memory space is shared.
CLONE_SIGHAND	Signal handlers are shared.
CLONE_FILES	The set of open files is shared.

Sistemas
Operativos

SC's
Pthreads

[Ste05]: cap.

SC's
Pthreads

- Identificación de threads
- Creación de threads
- Terminación de threads
- Threads + fork(), exec()
- Ejemplo

Identificación de threads

- Tipo de dato `pthread_t` para el identificador de thread (unsigned int en Solaris)

```
#include <pthread.h>
pthread_t pthread_equal(pthread_t
  tid1, pthread_t      tid2);
```

  - Devuelve (≠0,TRUE) si `tid1=tid2`
  - Devuelve (0, FALSE) si `tid1≠tid2`

```
pthread_t pthread_self(void);
```

  - Devuelve el identificador del thread que llama (similar a `getpid()` en procesos)

Creación de threads

```
#include <pthread.h>
int pthread_create(pthread_t
  *tidp, const  pthread_attr_t
  *attr,
  void*(*start_rtn)(void *),
  void*arg);
```

- Devuelve: 0 si OK, número de error si falla
  - `tidp`: Variable donde pone el identif. de thread creado
  - `attr`: Atributos de creación del thread (NULL para atributos por defecto).
  - `start_rtn`: función donde comienza la ejecución del thread
  - `arg`: argumento(s) de la función `start_rtn`

Terminación de threads

- Si un thread ejecuta `exit`, termina el proceso entero (todos los threads).
- Para que un thread termine sin acabar el proceso entero debe hacer:
  - Retornar de su rutina de comienzo. El valor devuelto es el código de terminación del thread.
  - Thread cancelado por otro thread en el mismo proceso (`pthread_cancel`)
  - Ejecutar `pthread_exit`

```
#include  <pthread.h>
int   pthread_exit(void    *rval_ptr);
```

    - El puntero `rval_ptr` está disponible para el resto de threads por medio de `pthread_join`

Terminación de threads

```
#include <pthread.h>
int pthread_join(pthread_t
  tid, void  **rval_ptr);
```

- Devuelve: 0 si OK, número de error si falla
- El thread que ejecuta `pthread_join` se bloquea hasta que el thread `tid` termina (similar a `waitpid` en procesos). `rval_ptr` vale:
  - Si el thread `tid` simplemente retorna de su función start, `rval_ptr` toma el valor devuelto
  - `rval_ptr=PTHREAD_CANCELED`si el thread `tid` es cancelado
  - Si no interesa el valor devuelto por el thread `tid`, poner `NULL` en `rval_ptr`

Threads + fork() exec()

- Si un thread hace fork(), dos posibilidades
  - Nuevo proceso con solo un thread (el que hace fork)
    - Sería lo lógico si luego se llama a exec()
  - Nuevo proceso con todos los threads
    - Sería lo lógico si no hay exec() posterior
- En hendrix (solaris 10)
  - fork() duplica unicamente el thread que llama a fork()
  - forkall() nueva SC para replicar todos los threads
- Si un thread hace exec(), funciona como siempre, es decir, desaparecen todos los threads y se sustituye por el ejecutable indicado en exec()

Thr_n.c

```
#include <stdio.h> <stdlib.h> <pthread.h> "error.h"
struct arg {         /* estructura para pasar arg/res al thread */
    int ini;         /* indice inicial*/
    int fin;         /* indice final */
    int res;         /* resultado devuelto */
};
int n,n_thr,*v;      /* var global compartidas por todos los threads */

void main(int argc,char *argv[]) {
    int i,tam,error,suma;
    pthread_t *tid;
    struct arg *param;

    if (argc != 3){printf("Uso: thr_n <int> <n_thr>\n");exit(1);}
    n=atoi(argv[1]);n_thr=atoi(argv[2]);

    v=calloc(n,sizeof(int));  /* reserva vector a sumar */
    tid=calloc(n_thr,sizeof(pthread_t));          /* reserva tid_threads */
    param=calloc(n_thr,sizeof(struct arg));       /* reserva parametros */

    for (i=0;i<n;i++) v[i]=1; /* inicializacion vector a sumar */
    tam=n/n_thr;    /* tamaño trozo para cada thread */
    printf("Calculando S(%d) en %d threads -> tam=%d\n",n,n_thr,tam);
```

Thr_n.c

```
/* creando threads para suma parciales */
   for (i=0;i<n_thr;i=i+1) {
       param[i].ini=tam*i;                  /* param1 del thread */
       param[i].fin=tam*(i+1);    /* param2 del thread */
       error=pthread_create(&tid[i],NULL,start,&param[i]);
       if (error!=0) syserr(pthread_create);
   }

   suma=0;
   if (n_thr*tam<n)                /* falta sumar el resto */
       for (i=n_thr*tam;i<n;i=i+1) suma=suma+v[i];

  for (i=0;i<n_thr;i=i+1) {        /* espera terminacion threads */
       pthread_join(tid[i],NULL);
       suma=suma+param[i].res;    /* extrae resultado del thread */
   }
   printf("Terminado. S(%d)=%d\n",n,suma);
}
```

Thr_n.c

```
void *start(void *p) {
   pthread_t tid;
   int i,ini,fin,tmp;

   ini=((struct arg *)p)->ini;     /* ini=arg1 */
   fin=((struct arg *)p)->fin;      /* fin=arg2 */
   tmp=0;
   for (i=ini;i<fin;i=i+1) tmp=tmp+v[i];/* calcula la suma parcial */
    ((struct arg *)p)->res=tmp;    /* almacena resultado */
    pthread_exit(NULL);            /* acaba "sin devolver" resultado */
}
```