Grupo ARCOS
Universidad Carlos III de Madrid

Lección 3
Procesos e hilos

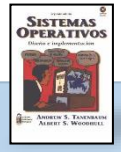
Sistemas Operativos
Ingeniería Informática

Lecturas recomendadas

Base

1. Carretero 2020:
   1. Cap. 5
2. Carretero 2007:
   1. Cap. 3.6 y 3.7
      Cap. 3.9 y 3.13

Recomendada



1. Tanenbaum 2006:
   1. (es) Cap. 2.2
   2. (en) Cap.2.1.7
2. Stallings 2005:
   1. 4.1, 4.4, 4.5 y 4.6
3. Silberschatz 2006:
   1. 4

¡ATENCIÓN!

- Este material es un guión de la clase pero no son los apuntes de la asignatura.
- Los libros dados en la bibliografía junto con lo explicado en clase representa el material de estudio para el temario de la asignatura.

Contenidos

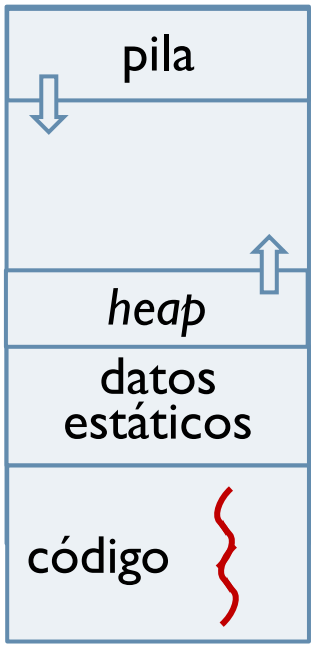
1. Introducción
   - Definición de proceso.
   - Modelo ofrecido: recursos, multiprogramación, multitarea y multiproceso
2. Ciclo de vida del proceso: estado de procesos.
3. Servicios para gestionar procesos que da el sistema operativo.
4. Definición de hilo o *thread*
5. Hilos de biblioteca y núcleo.
6. Servicios para hilos en el sistema operativo.

Contenidos

1. Introducción
   - Definición de proceso.
   - Modelo ofrecido: recursos, multiprogramación, multitarea y multiproceso
2. Ciclo de vida del proceso: estado de procesos.
3. Servicios para gestionar procesos que da el sistema operativo.
4. **Definición de hilo o *thread***
5. Hilos de biblioteca y núcleo.
6. Servicios para hilos en el sistema operativo.

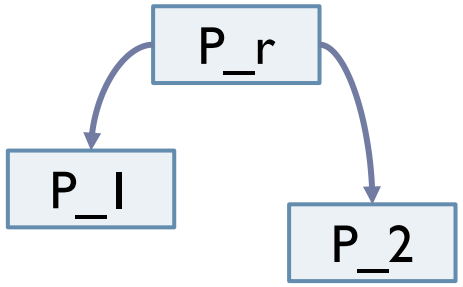
De dónde partir: proceso…

- Definición:
  - Programa en ejecución.
  - Unidad de procesamiento gestionada por el S.O.
- Modelo:
  - Agrupa recursos usados:
    - Ficheros, señales, memoria, …
  - Multiproceso, multitarea y multiprocesador:
    - Cada proceso tiene un único "hilo" de ejecución.
    - Jerarquía de procesos con protección/compartición entre procesos.

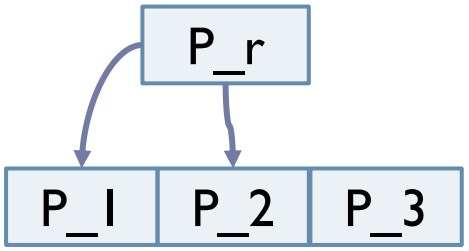


A dónde llegar: Aplicaciones concurrentes

- Diseño basado en cliente/servidor:
  - Bajo demanda:
    - Un proceso espera la llegada de peticiones (receptor de peticiones)
    - Al llegar una petición crea un proceso para atender dicha petición.



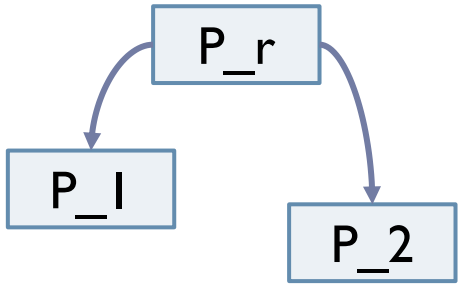
  - Pre-reserva:
    - Un proceso receptor y N procesos de tratamiento de petición bloqueados.
    - Al llegar una petición se desbloquea un proceso, atiende la petición y se vuelve a bloquear.



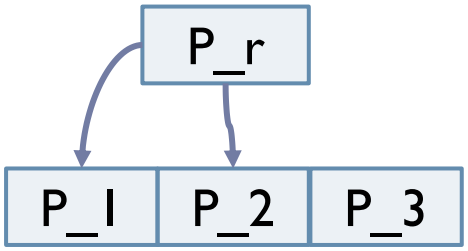
A dónde llegar: Aplicaciones concurrentes

- Consumo de tiempo en la creación y terminación de procesos
- Consumo de tiempo en cambios de contexto
- La compartición de recursos no es fácil (y puede presentar problemas)

- Bajo demanda:
  - Un proceso espera la llegada de peticiones (receptor de peticiones)
  - Al llegar una petición crea un proceso para atender dicha petición.



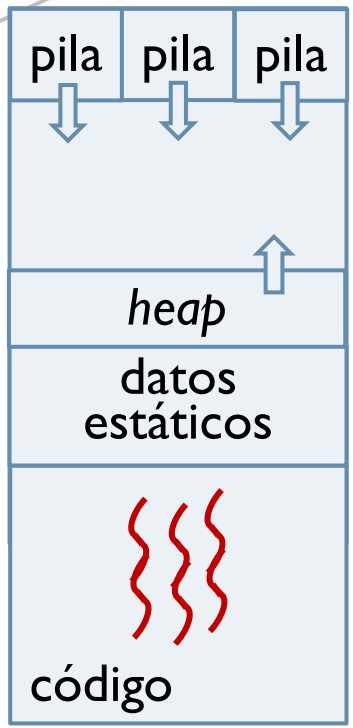
- Pre-reserva:
  - Un proceso receptor y N procesos de tratamiento de petición bloqueados.
  - Al llegar una petición se desbloquea un proceso, atiende la petición y se vuelve a bloquear.



Hilo, proceso ligero o *thread*

- Consumo de tiempo en la creación y terminación de procesos
- Consumo de tiempo en cambios de contexto
- La compartición de recursos no es fácil (y puede presentar problemas)

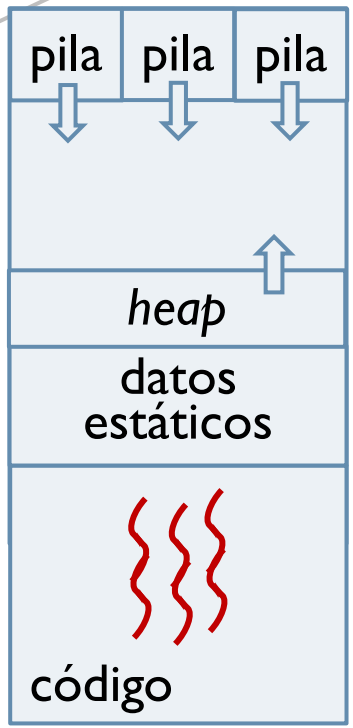
- Modelo de proceso:
  - Agrupa recursos usados:
    - Ficheros, señales, memoria, …
  - Multiproceso, multitarea y multiprocesador:
    - Cada **proceso puede tener varios** "hilos" de ejecución. El hilo es una unidad básica de "utilización" de la CPU (unidad "planificable")
    - Los **hilos de un mismo proceso comparten todos** los recursos del proceso **salvo pila y contexto** (PC, RE, SP, …).
    - Jerarquía de procesos con protección/compartición entre procesos.



Hilo, proceso ligero o *thread*

- Consumo de tiempo en la creación y terminación de procesos
- Consumo de tiempo en cambios de contexto
- La compartición de recursos no es fácil (y puede presentar problemas)

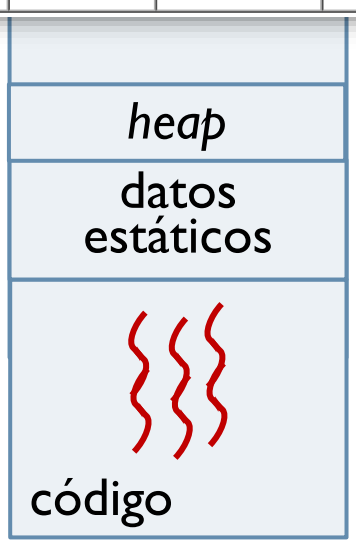
- Beneficios:
  - Compartición de recursos.
    - Facilita comunicación con memoria compartida entre hilos de un mismo proceso.
  - Economía de recursos.
    - Crear un hilo necesita menos tiempo. (Ej. Solaris -> 30x) Cambio de contexto más simple y aprovecha las tablas de traducción de memoria virtual, caché, etc.
  - Capacidad de respuesta.
    - Al separar el tratamiento de las interacciones con el usuario con hilos ofrece mayor interactividad.
  - Mejor aprovechamiento de arquitecturas multiprocesador y multicore.



Platform	fork()			pthread_create()		
	real	user	sys	real	user	sys
Intel 2.6 GHz Xeon E5-2670 (16 cores/node)	8.1	0.1	2.9	0.9	0.2	0.3
Intel 2.8 GHz Xeon 5660 (12 cores/node)	4.4	0.4	4.3	0.7	0.2	0.5
AMD 2.3 GHz Opteron (16 cores/node)	12.5	1.0	12.5	1.2	0.2	1.3
AMD 2.4 GHz Opteron (8 cores/node)	17.6	2.2	15.7	1.4	0.3	1.3
IBM 4.0 GHz POWER6 (8 cpus/node)	9.5	0.6	8.8	1.6	0.1	0.4
IBM 1.9 GHz POWER5 p5-575 (8 cpus/node)	64.2	30.7	27.6	1.7	0.6	1.1
IBM 1.5 GHz POWER4 (8 cpus/node)	104.5	48.6	47.2	2.1	1.0	1.5
INTEL 2.4 GHz Xeon (2 cpus/node)	54.9	1.5	20.8	1.6	0.7	0.9
INTEL 1.4 GHz Itanium2 (4 cpus/node)	54.5	1.1	22.2	2.0	1.2	0.6

un mismo proceso.

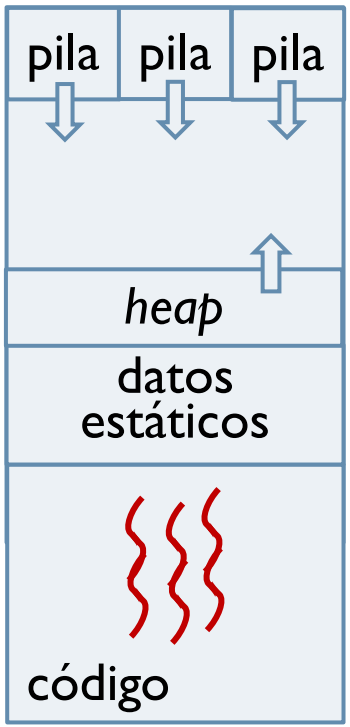
- Economía de recursos.
  - **Crear un hilo necesita menos tiempo**.
    Cambio de contexto más simple y aprovecha las tablas de traducción de memoria virtual, caché, etc.
- Capacidad de respuesta.
  - Al separar el tratamiento de las interacciones con el usuario con hilos ofrece mayor interactividad.
- Mejor aprovechamiento de arquitecturas multiprocesador y multicore.



https://computing.llnl.gov/tutorials/pthreads/

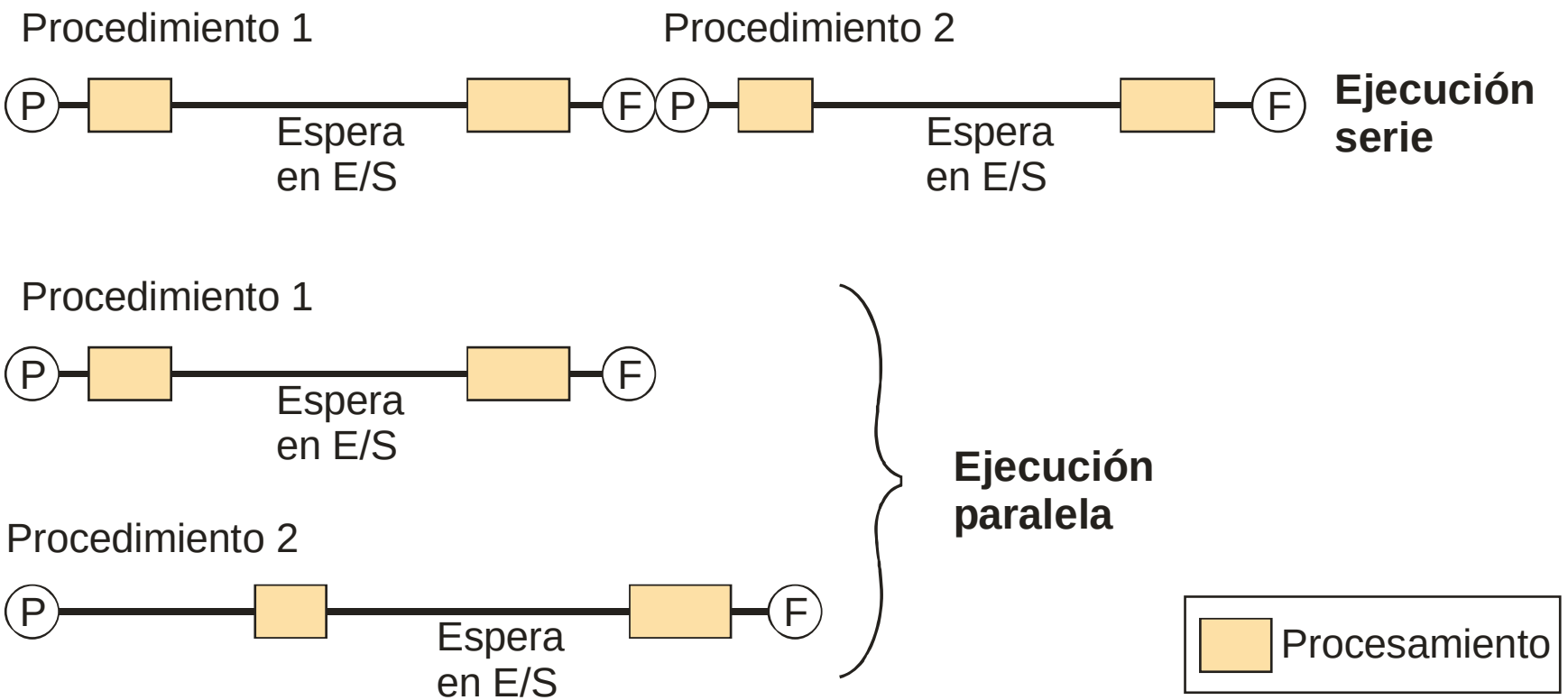
Platform	MPI Shared Memory Bandwidth (GB/sec)	Pthreads Worst Case Memory-to-CPU Bandwidth (GB/sec)
Intel 2.6 GHz Xeon E5-2670	4.5	51.2
Intel 2.8 GHz Xeon 5660	5.6	32
AMD 2.3 GHz Opteron	1.8	5.3
AMD 2.4 GHz Opteron	1.2	5.3
IBM 1.9 GHz POWER5 p5-575	4.1	16
IBM 1.5 GHz POWER4	2.1	4
Intel 2.4 GHz Xeon	0.3	4.3
Intel 1.4 GHz Itanium 2	1.8	6.4

- Compartición de recursos.
  - **Facilita comunicación con memoria compartida entre hilos de un mismo proceso**.
- Economía de recursos.
  - Crear un hilo necesita menos tiempo.
    Cambio de contexto más simple y aprovecha las tablas de traducción de memoria virtual, caché, etc.
- Capacidad de respuesta.
  - Al separar el tratamiento de las interacciones con el usuario con hilos ofrece mayor interactividad.
- Mejor aprovechamiento de arquitecturas multiprocesador y multicore.

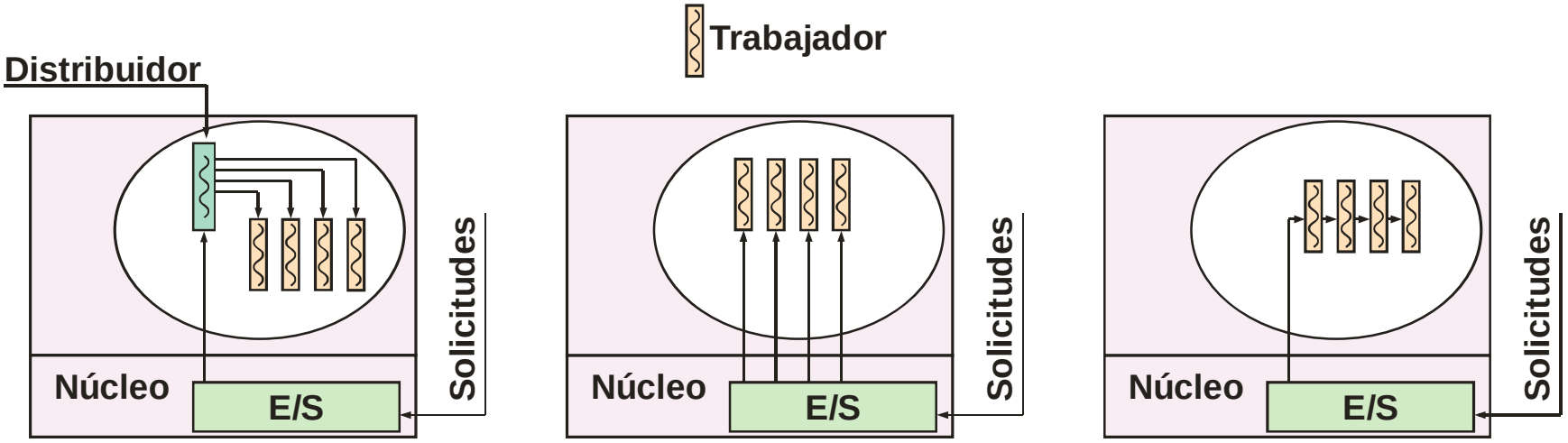


https://computing.llnl.gov/tutorials/pthreads/

Los *threads* permiten paralelizar la ejecución de una aplicación



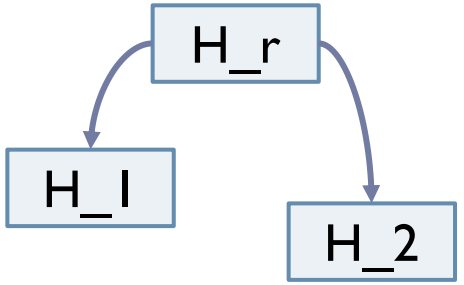
Arquitecturas software basadas en threads

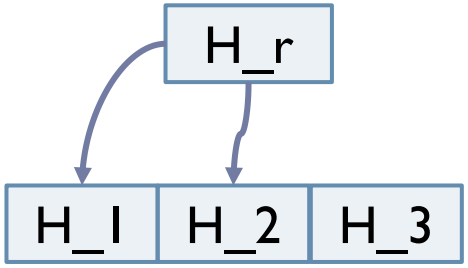


Sistemas operativos: una visión aplicada (© J. Carrete et al.)

Hilos y procesamiento de solicitudes

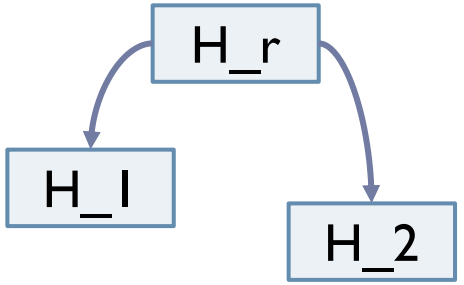
- Diseño basado en cliente/servidor:
  - Bajo demanda:
    - Un hilo espera la llegada de peticiones (receptor de peticiones) y al llegar una petición se crea un hilo para atender dicha petición.
    - Inconvenientes:
      - Tiempo de creación supone un retardo.
      - Si llega una avalancha de peticiones -> DoS
  - Pre-reserva o *Thread Pools*:
    - Se crea un conjunto de hilos que quedan a la espera de que lleguen peticiones.
    - Ventajas:
      - Minimizar el retardo de atención (hilo existe).
      - Se mantiene un límite de # de hilos concurrentes.

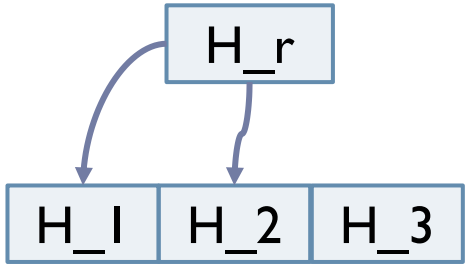




Cancelación/finalización de hilos

- Diseño basado en cliente/servidor:
  - Bajo demanda:
    - Hilo coordinador no crea más hilos y espera a la finalización de los existentes.
  - Pre-reserva o *Thread Pools*:
    - Un hilo notifica al resto de que deben terminar.
    - Opciones:
      - Cancelación asíncrona: se fuerza a la terminación del hilo.
        - Problema: liberación de los recursos en uso.
      - Cancelación diferida: se notifica al hilo de que llega una petición "especial" de finalización de ejecución.
        - Preferible aunque pueda necesitar algo de tiempo.



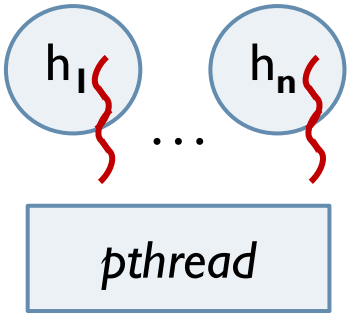


Contenidos

1. Introducción
   - Definición de proceso.
   - Modelo ofrecido: recursos, multiprogramación, multitarea y multiproceso
2. Ciclo de vida del proceso: estado de procesos.
3. Servicios para gestionar procesos que da el sistema operativo.
4. Definición de hilo o *thread*
5. **Hilos de biblioteca y núcleo**
   - **Modelo de hilos.**
6. Servicios para hilos en el sistema operativo.

Hilos de biblioteca vs de núcleo

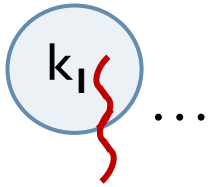
- ULT (*User Level Threads*)
  - **Implementación** de los **servicios de hilo**: **como biblioteca en espacio de usuario**.
    - El *kernel* NO tiene conocimiento sobre la existencia de hilos.
  - Ventaja: más rápidos al no tener que hacer cambios de modo usuario a kernel (y viceversa)
  - Inconveniente: si un hilo hace una llamada bloqueante entonces bloquea todo el proceso.



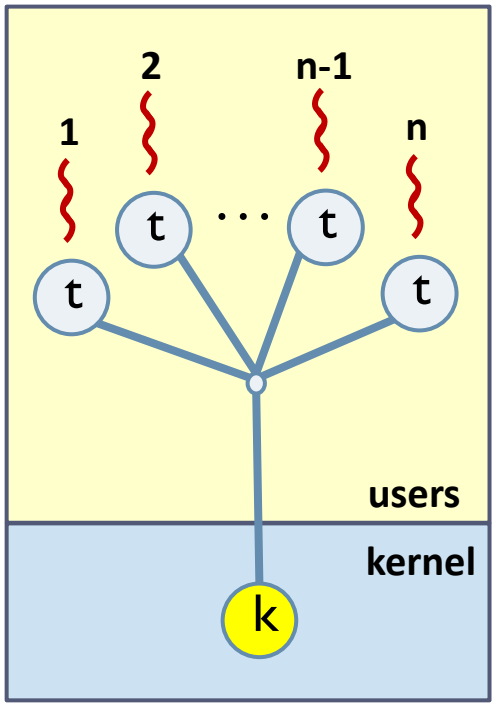
users

kernel

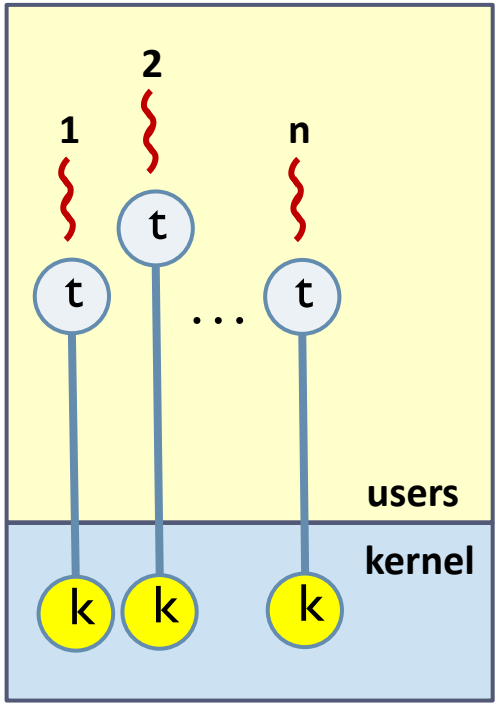
- KLT (*Kernel Level Threads*)
  - **Implementación** de los **servicios de hilo**: **como servicio del kernel**.
    - El *kernel* tiene soporte de hilos (para él y usuarios/as)
  - Inconveniente: más lentos al tener que hacer cambios de modo usuario a kernel (y viceversa)
  - Ventaja: si un hilo hace una llamada bloqueante entonces solo bloquea a dicho hilo.



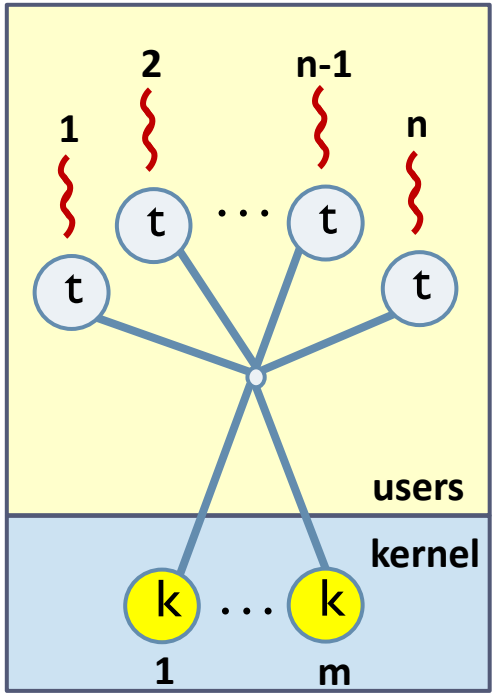
Modelo de múltiples hilos



Muchos a uno



Uno a uno

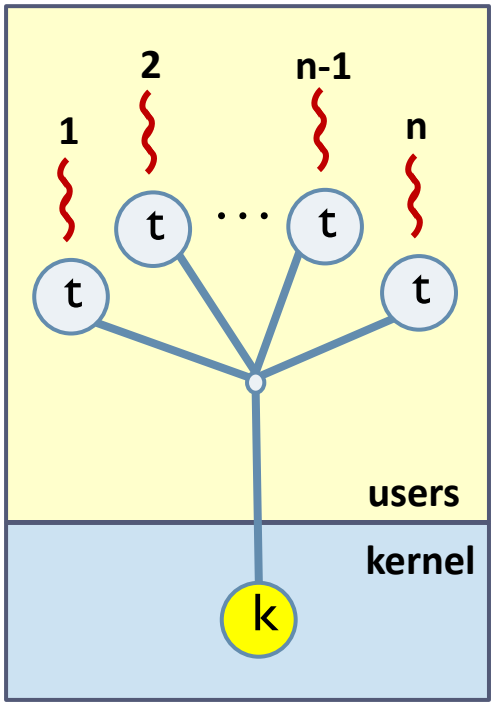


Muchos a muchos

http://www.it.uu.se/education/course/homepage/os/vt18/module-4/implementing-threads/

Modelo de múltiples hilos

- **Muchos a uno**
  - Múltiples hilos de usuario se corresponden con un único hilo de núcleo.
  - Ej.: biblioteca de hilos de usuario.
  - V/I:
    - Llamada bloqueante bloquea todos los hilos.
    - En SMP no se pueden ejecutar varios hilos a la vez.



- Uno a uno
  - Cada hilo de usuario se corresponden con un hilo de núcleo.
  - Ej.: Linux 2.6, Windows, Solaris 9
  - V/I:
    - Llamada bloqueante NO bloquea todos los hilos.
    - En SMP se pueden ejecutar varios hilos a la vez.
- Muchos a muchos
  - Se multiplexa los hilos de usuarios en un número de hilos en el núcleo.
  - Ej.: IRIX, HP-UX, Solaris (antes de 9)
  - V/I:
    - Llamada bloqueante NO bloquea todos los hilos.
    - En SMP se pueden ejecutar varios hilos a la vez.

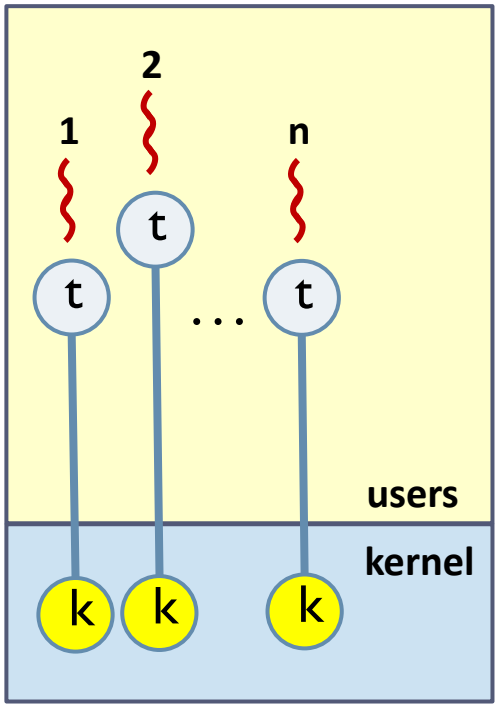
http://www.it.uu.se/education/course/homepage/os/vt18/module-4/implementing-threads/

Modelo de múltiples hilos

- Muchos a uno
  - Múltiples hilos de usuario se corresponden con un único hilo de núcleo.
  - Ej.: biblioteca de hilos de usuario.
  - V/I:
    - Llamada bloqueante bloquea todos los hilos.
    - En SMP no se pueden ejecutar varios hilos a la vez.

Uno a uno

- Cada hilo de usuario se corresponden con un hilo de núcleo.
- Ej.: Linux 2.6, Windows, Solaris 9
- V/I:
  - Llamada bloqueante NO bloquea todos los hilos.
  - En SMP se pueden ejecutar varios hilos a la vez.

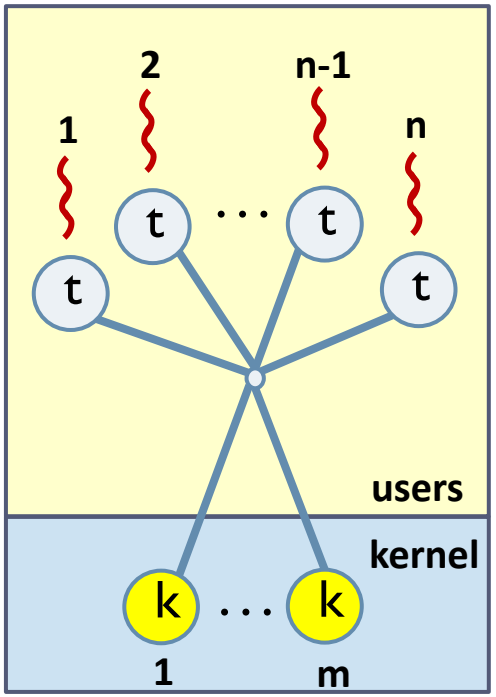


- Muchos a muchos
  - Se multiplexa los hilos de usuarios en un número de hilos en el núcleo.
  - Ej.: IRIX, HP-UX, Solaris (antes de 9)
  - V/I:
    - Llamada bloqueante NO bloquea todos los hilos.
    - En SMP se pueden ejecutar varios hilos a la vez.

http://www.it.uu.se/education/course/homepage/os/vt18/module-4/implementing-threads/

Modelo de múltiples hilos

- Muchos a uno
  - Múltiples hilos de usuario se corresponden con un único hilo de núcleo.
  - Ej.: biblioteca de hilos de usuario.
  - V/I:
    - Llamada bloqueante bloquea todos los hilos.
    - En SMP no se pueden ejecutar varios hilos a la vez.
- Uno a uno
  - Cada hilo de usuario se corresponden con un hilo de núcleo.
  - Ej.: Linux 2.6, Windows, Solaris 9
  - V/I:
    - Llamada bloqueante NO bloquea todos los hilos.
    - En SMP se pueden ejecutar varios hilos a la vez.

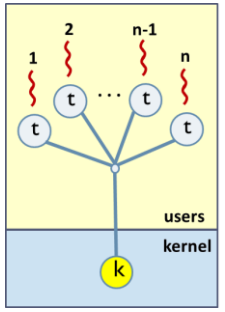


- **Muchos a muchos**
  - Se multiplexa los hilos de usuarios en un número de hilos en el núcleo.
  - Ej.: IRIX, HP-UX, Solaris (antes de 9)
  - V/I:
    - Llamada bloqueante NO bloquea todos los hilos.
    - En SMP se pueden ejecutar varios hilos a la vez.

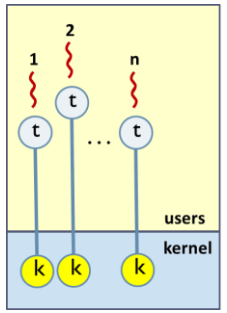
http://www.it.uu.se/education/course/homepage/os/vt18/module-4/implementing-threads/

Modelo de múltiples hilos
resumen

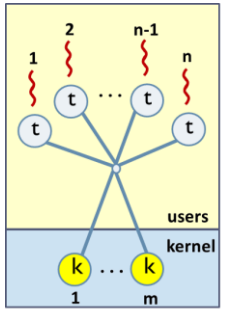
- Muchos a uno
  - Múltiples hilos de usuario se corresponden con un único hilo de núcleo.
  - Ej.: biblioteca de hilos de usuario.
  - V/I:
    - Llamada bloqueante bloquea todos los hilos.
    - En SMP no se pueden ejecutar varios hilos a la vez.



- Uno a uno
  - Cada hilo de usuario se corresponden con un hilo de núcleo.
  - Ej.: Linux 2.6, Windows, Solaris 9
  - V/I:
    - Llamada bloqueante NO bloquea todos los hilos.
    - En SMP se pueden ejecutar varios hilos a la vez.



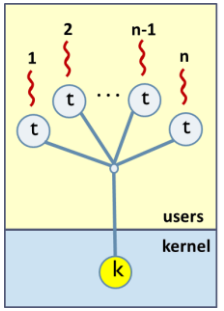
- Muchos a muchos
  - Cada hilo de usuario se corresponden con un hilo de núcleo.
  - Ej.: Linux 2.6, Windows, Solaris 9
  - V/I:
    - Llamada bloqueante NO bloquea todos los hilos.
    - En SMP se pueden ejecutar varios hilos a la vez.



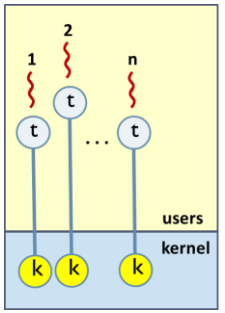
http://www.it.uu.se/education/course/homepage/os/vt18/module-4/implementing-threads/

Modelo de múltiples hilos
resumen

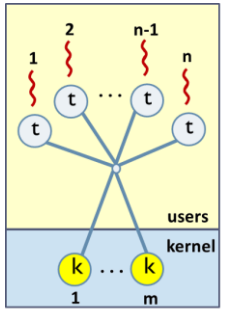
- Muchos a uno
  - Múltiples hilos de usuario se corresponden con un único hilo de núcleo.
  - Ej.: biblioteca de hilos de usuario.
  - V/I:
    - Llamada bloqueante bloquea todos los hilos.
    - En SMP no se pueden ejecutar varios hilos a la vez.



- Uno a uno
  - Cada hilo de usuario se corresponden con un hilo de núcleo.
  - Ej.: Linux 2.6, Windows, Solaris 9
  - V/I:
    - Llamada bloqueante NO bloquea todos los hilos.
    - En SMP se pueden ejecutar varios hilos a la vez.

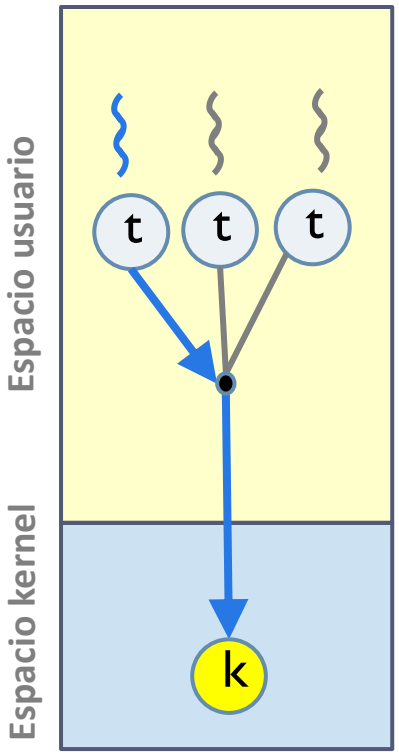


- Muchos a muchos
  - Cada hilo de usuario se corresponden con un hilo de núcleo.
  - Ej.: Linux 2.6, Windows, Solaris 9
  - V/I:
    - Llamada bloqueante NO bloquea todos los hilos.
    - En SMP se pueden ejecutar varios hilos a la vez.

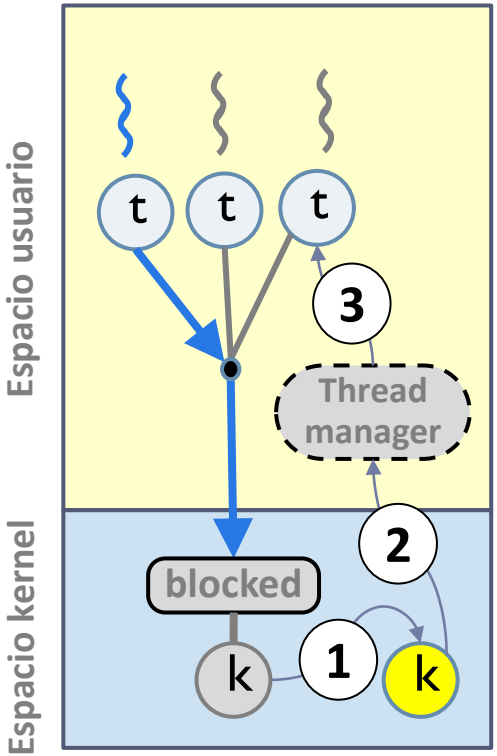


http://www.it.uu.se/education/course/homepage/os/vt18/module-4/implementing-threads/

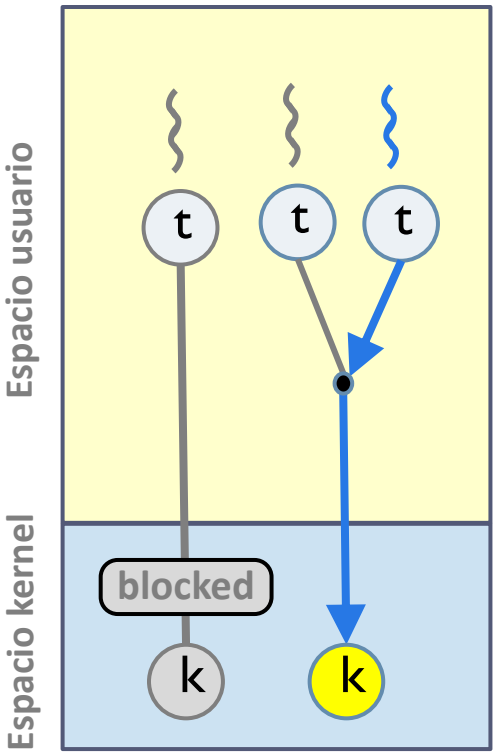
Activación por planificador



Situación inicial



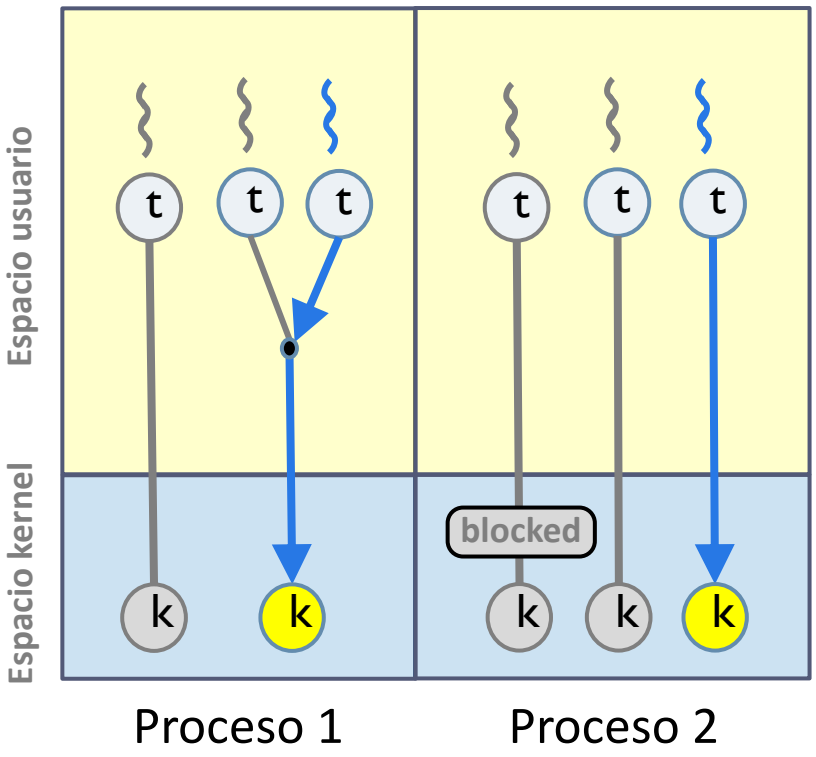
Activación



Situación final

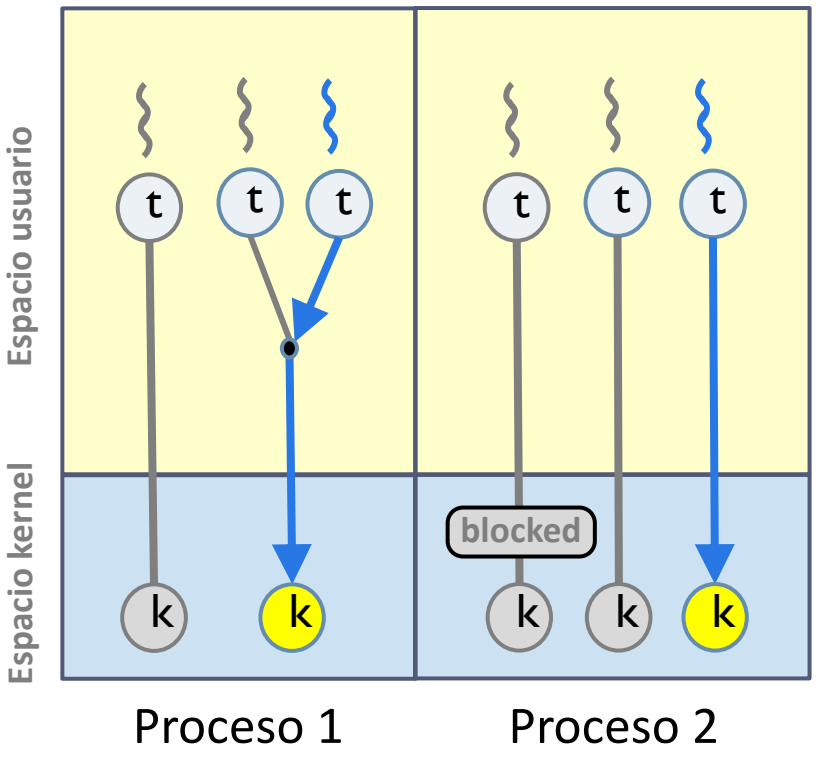
http://www.it.uu.se/education/course/homepage/os/vt18/module-4/implementing-threads/

Estados de un proceso con hilos



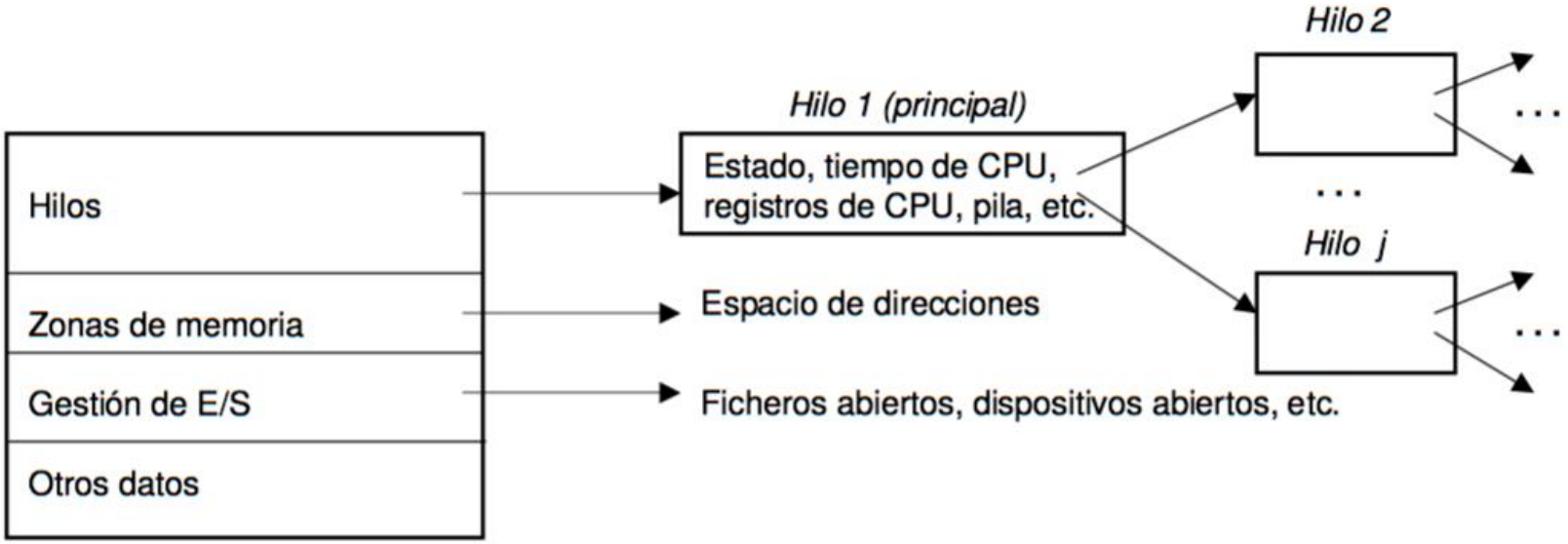
- Estado de un proceso con hilos es la combinación de los estados de sus hilos:
  - Si hay un hilo en ejecución -> Proceso en ejecución
  - Si no hay hilos en ejecución pero sí preparados (listos) -> Proceso preparado
  - Si todos sus hilos bloqueados -> Proceso bloqueado

Planificación con hilos (por defecto)



- La planificación de ejecución de *threads* se basa en el modelo de prioridades y no utiliza el modelo de segmentación por segmentos de tiempo.
  - Por ejemplo FIFO con prioridades (no *Round-Robin*)
- Un *thread* continuará ejecutándose en la CPU hasta pasar a un estado que no le permita seguir en ejecución.
  - Alternancia explícita mediante sleep() o yield()

BCP proceso con hilos



Sistemas operativos: una visión aplicada (© J. Carrete et al.)

Problemas/Peculiaridades usando hilos

- Gestión de señales.
  - *Inicio monohilo*: Las señales en UNIX se usan para notificar a un proceso de que ha ocurrido un evento.
  - *Entorno multihilo*: ¿Qué hilo le llega la señal?
    - A) Enviar la señal al hilo implicado. B) Enviar la señal a todos los hilos. C) Enviar la señal a ciertos hilos del proceso. D) Asignar a un hilo específico la recepción de todas las señales del proceso.
  - ¿Se pueden mandar señales entre hilos de un mismo proceso?
- Llamada al sistema fork().
  - *Inicio monohilo*: Llamada para crear una copia del proceso que llama.
  - *Entorno multihilo*: si un hilo llama a fork() ¿Se hace una copia del proceso con todos los hilos o una copia con solo el hilo que llamó a fork()?
    - A) Depende del sistema. B) En la copia solo está el hilo implicado. C) En la copia están todos los hilos implicados. D) Hay distintas llamadas al sistema que permite seleccionar el comportamiento. Ej. en Linux: clone()

Ejemplo (1/3): thread_create() + fork()

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

#define NTHREADS 2
#define NSECONDS 3

void * do_something1 ( void *arg )
{
        int tid = pthread_self() ;
        for (int i=0; i<NSECONDS; i++)
        {
             printf("(%d) <pid=%d, tid=%d>\n", i, getpid(), tid) ;
             sleep(1) ;
        }

        pthread_exit(NULL);
}

int main ( int argc, char *argv[] )
{
        int ret, status ;
        pthread_t tid[NTHREADS+1] ;
        pthread_attr_t attr ;

        pthread_attr_init(&attr) ;
        pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE) ;

        // Two process making threads...
        // int pid = fork() ;

        // creat threads
        for (int j=0; j<NTHREADS; j++)
        {
             ret = pthread_create(&(tid[j]), &attr, do_something1, NULL) ;
             if (ret) {
                 printf("ERROR on pthread_create(): %d\n", ret) ;
                 exit(-1) ;
             }
        }

        // ONE process making threads...
        int pid = fork() ;

        if (pid != 0)
        {
                // father wait for threads
                for (int j=0; j<NTHREADS; j++)
                {
                     ret = pthread_join(tid[j], NULL) ;
                     if (ret) {
                         printf("ERROR on pthread_join(): %d\n", ret) ;
                         exit(-1) ;
                     }
                }

                // resources back...
                pthread_attr_destroy(&attr) ;
        }
        if (pid != 0)
        {
                // father wait for children
                while (pid != wait(&status)) ;

        }

        return 0;
}
```

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

#define NTHREADS 2
#define NSECONDS 3

void * do_something1 ( void *arg )
{
    int tid = pthread_self() ;
    for (int i=0; i<NSECONDS; i++)
    {
        printf("(%d) <pid=%d, tid=%d>\n",
                i, getpid(), tid) ;
        sleep(1) ;
    }

    pthread_exit(NULL);
}
```

Ejemplo (2/3): thread_create() + fork()

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

#define NTHREADS 2
#define NSECONDS 3

void * do_something1 ( void *arg )
{
        int tid = pthread_self() ;
        for (int i=0; i<NSECONDS; i++)
        {
             printf("(%d) <pid=%d, tid=%d>\n", i, getpid(), tid) ;
             sleep(1) ;
        }

        pthread_exit(NULL);
}

int main ( int argc, char *argv[] )
{
        int ret, status ;
        pthread_t tid[NTHREADS+1] ;
        pthread_attr_t attr ;

        pthread_attr_init(&attr) ;
        pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE) ;

        // Two process making threads...
        // int pid = fork() ;

        // creat threads
        for (int j=0; j<NTHREADS; j++)
        {
             ret = pthread_create(&(tid[j]), &attr, do_something1, NULL) ;
             if (ret) {
                 printf("ERROR on pthread_create(): %d\n", ret) ;
                 exit(-1) ;
             }
        }

        // ONE process making threads...
        int pid = fork() ;

        if (pid != 0)
        {
                // father wait for threads
                for (int j=0; j<NTHREADS; j++)
                {
                     ret = pthread_join(tid[j], NULL) ;
                     if (ret) {
                         printf("ERROR on pthread_join(): %d\n", ret) ;
                         exit(-1) ;
                     }
                }

                // resources back...
                pthread_attr_destroy(&attr) ;
        }
        if (pid != 0)
        {
                // father wait for children
                while (pid != wait(&status)) ;

        }

        return 0;
}
```

```
int main ( int argc, char *argv[] )
{
    int ret, status ;
    pthread_t tid[NTHREADS+1] ;
    pthread_attr_t attr ;

    pthread_attr_init(&attr) ;
    pthread_attr_setdetachstate(&attr,
                    PTHREAD_CREATE_JOINABLE);

    // creat threads
    for (int j=0; j<NTHREADS; j++)
    {
        ret = pthread_create(&(tid[j]),
                             &attr,
                             do_something1,
                             NULL);
        if (ret) {
            printf("ERROR p_create(): %d\n",
                    ret) ;
                   exit(-1) ;
        }
    }
```

Ejemplo (3/3): thread_create() + fork()

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

#define NTHREADS 2
#define NSECONDS 3

void * do_something1 ( void *arg )
{
        int tid = pthread_self() ;
        for (int i=0; i<NSECONDS; i++)
        {
             printf("(%d) <pid=%d, tid=%d>\n", i, getpid(), tid) ;
             sleep(1) ;
        }

        pthread_exit(NULL);
}

int main ( int argc, char *argv[] )
{
        int ret, status ;
        pthread_t tid[NTHREADS+1] ;
        pthread_attr_t attr ;

        pthread_attr_init(&attr) ;
        pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE) ;

        // Two process making threads...
        // int pid = fork() ;

        // creat threads
        for (int j=0; j<NTHREADS; j++)
        {
             ret = pthread_create(&(tid[j]), &attr, do_something1, NULL) ;
             if (ret) {
                 printf("ERROR on pthread_create(): %d\n", ret) ;
                 exit(-1) ;
             }
        }

        // ONE process making threads...
        int pid = fork() ;

        if (pid != 0)
        {
                // father wait for threads
                for (int j=0; j<NTHREADS; j++)
                {
                     ret = pthread_join(tid[j], NULL) ;
                     if (ret) {
                         printf("ERROR on pthread_join(): %d\n", ret) ;
                         exit(-1) ;
                     }
                }

                // resources back...
                pthread_attr_destroy(&attr) ;
        }
        if (pid != 0)
        {
                // father wait for children
                while (pid != wait(&status)) ;

        }

        return 0;
}
```

```
        // new process with fork()
        int pid = fork() ;

        if (pid != 0)
        {    // father wait for threads
             for (int j=0; j<NTHREADS; j++)
             {
                 ret = pthread_join(tid[j], NULL);
                 if (ret) {
                    printf("ERROR p_join(): %d\n",
                           ret);
                    exit(-1) ;
                 }
             }

             // resources back...
             pthread_attr_destroy(&attr) ;
        }
        if (pid != 0) {
             // father wait for children
             while (pid != wait(&status)) ;
        }

        return 0;
}
```

Ejemplo: thread_create() + fork()

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

#define NTHREADS 2
#define NSECONDS 3

void * do_something1 ( void *arg )
{
        int tid = pthread_self() ;
        for (int i=0; i<NSECONDS; i++)
        {
             printf("(%d) <pid=%d, tid=%d>\n", i, getpid(), tid) ;
             sleep(1) ;
        }

        pthread_exit(NULL);
}

int main ( int argc, char *argv[] )
{
        int ret, status ;
        pthread_t tid[NTHREADS+1] ;
        pthread_attr_t attr ;

        pthread_attr_init(&attr) ;
        pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE) ;

        // Two process making threads...
        // int pid = fork() ;

        // creat threads
        for (int j=0; j<NTHREADS; j++)
        {
             ret = pthread_create(&(tid[j]), &attr, do_something1, NULL) ;
             if (ret) {
                 printf("ERROR on pthread_create(): %d\n", ret) ;
                 exit(-1) ;
             }
        }

        // ONE process making threads...
        int pid = fork() ;

        if (pid != 0)
        {
                // father wait for threads
                for (int j=0; j<NTHREADS; j++)
                {
                     ret = pthread_join(tid[j], NULL) ;
                     if (ret) {
                         printf("ERROR on pthread_join(): %d\n", ret) ;
                         exit(-1) ;
                     }
                }

                // resources back...
                pthread_attr_destroy(&attr) ;
        }
        if (pid != 0)
        {
                // father wait for children
                while (pid != wait(&status)) ;

        }

        return 0;
}
```

```
alex@patata:$ gcc -Wall -g \
                  -o ths_and_fork \
                  ths_and_fork.c -lpthread


alex@patata:$ ./ths_and_fork
(0) <pid=426, tid=-1144539392>
(0) <pid=426, tid=-1136146688>
(1) <pid=426, tid=-1144539392>
(1) <pid=426, tid=-1136146688>
(2) <pid=426, tid=-1144539392>
(2) <pid=426, tid=-1136146688>
```

Contenidos

1. Introducción
   - Definición de proceso.
   - Modelo ofrecido: recursos, multiprogramación, multitarea y multiproceso
2. Ciclo de vida del proceso: estado de procesos.
3. Servicios para gestionar procesos que da el sistema operativo.
4. Definición de hilo o *thread*
5. Hilos de biblioteca y núcleo.
6. **Servicios para hilos en el sistema operativo.**

Proceso “pesado” vs “ligero”

llamadas similares pero no iguales

	Proceso “pesado”	Proceso “ligero”
Crear	fork ()	pthread_create (…)
Esperar	wait (…)	pthread_join (…)
Terminar	exit (…)	pthread_exit (…)
Identificar	getpid()	pthread_self (…)

Creación de hilo

Servicio	`int pthread_create(pthread_t *thread,`
	`const pthread_attr_t *attr,`
	`void *(*func)(void *),`
	`void *arg);`

Argumentos	▫ **thread**: dirección de una variable tipo pthread_t donde se almacenará el identificador del hilo. ▫ **attr**: dirección de una estructura con los atributos del hilo. Se puede pasar NULL para usar atributos por defecto. ▫ **func**: puntero a función con el código que ejecutará el hilo. ▫ **arg**: puntero al parámetro del hilo. Solamente se puede pasar un parámetro (puede ser un puntero a una estructura).
Devuelve	▫ 0 si todo va bien. ▫ Código de error en caso de error.
Descripción	▫ Solicita la creación de un hilo.

Espera por hilo

Servicio	`int pthread_join (pthread_t thread, void **value) ;`
Argumentos	▫ **thread**: identificador del hilo por el que hay que esperar. ▫ **value**: si el valor es distinto de NULL, será la dirección de memoria donde almacenar el valor de terminación del hilo.
Devuelve	▫ 0 si todo va bien. ▫ Código de error en caso de error.
Descripción	▫ El hilo que invoca la función se espera hasta la terminación del hilo cuyo identificador se especifica por parámetro. Si el hilo ya había terminado entonces pthread_join termina inmediatamente. Dos hilos que ejecuten pthread_join por un mismo hilo supone comportamiento indefinido.

Terminación de hilo

Servicio	`void pthread_exit(void *retval) ;`
Argumentos	▫ **retval**: estado de terminación del hilo. NO puede ser un puntero a una variable local.
Devuelve	▫ No devuelve nada.
Descripción	▫ Permite a un proceso ligero finalizar su ejecución, indicando el estado de terminación del mismo al hilo padre (si es "joinable"). ▫ Cuando un hilo termina los recursos compartidos a nivel de proceso (mutex, semáforos, ficheros, etc.) no son liberados.

Identificación de hilo

Servicio	`pthread_t pthread_self(void) ;`
Argumentos	▫ No tiene argumentos.
Devuelve	▫ Devuelve el identificador del hilo que ejecuta la llamada.
Descripción	▫ Permite a un hilo conocer su identificador.

Ejemplo: crear y esperar

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

#define NUM_THREADS     5
pthread_t threads[NUM_THREADS];

void *th_function ( void *arg )
{
    printf("Hello world from thread #%ld!\n", (long)arg);
    pthread_exit(NULL);
}

int main ( int argc, char *argv[] )
{
    for (int t=0; t<NUM_THREADS; t++) {
        int rc = pthread_create(&(threads[t]), NULL, th_function, (void *)(long)t);
        if (rc) { printf("ERROR from pthread_create(): %d\n", rc); exit(-1); }
    }
    for (int t=0; t<NUM_THREADS; t++) {
        int rc = pthread_join(threads[t], NULL) ;
        if (rc) { printf("ERROR from pthread_join(): %d\n", rc); exit(-1); }
    }
    pthread_exit(NULL);
}
```

Ejemplo: crear

```
# gcc -Wall -g -o h h.c –lpthread
# ./h
Hello world from thread #0!
Hello world from thread #2!
Hello world from thread #1!
Hello world from thread #3!
Hello world from thread #4!
```

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

#define NUM_THREADS     5
pthread_t threads[NUM_THREA

void *th_function ( void *arg )
{
    printf("Hello world from thread #%ld!\n", (long)arg);
    pthread_exit(NULL);
}

int main ( int argc, char *argv[] )
{
    for (int t=0; t<NUM_THREADS; t++) {
        int rc = pthread_create(&(threads[t]), NULL, th_function, (void *)(long)t);
        if (rc) { printf("ERROR from pthread_create(): %d\n", rc); exit(-1); }
    }
    for (int t=0; t<NUM_THREADS; t++) {
        int rc = pthread_join(threads[t], NULL) ;
        if (rc) { printf("ERROR from pthread_join(): %d\n", rc); exit(-1); }
    }
    pthread_exit(NULL);
}
```

Atributos de un hilo

Servicio	`int pthread_create(pthread_t *thread,`
	`const pthread_attr_t *attr,`
	`void *(*func)(void *),`
	`void *arg);`

Argumentos	▫ **thread**: dirección de una variable tipo pthread_t donde se almacenará el identificador del hilo. ▫ **attr**: dirección de una estructura con los atributos del hilo. Se puede pasar NULL para usar atributos por defecto. ▫ **f** untero a función con el código que ejecutará el hilo. al parámetro del hilo. Solamente se puede pasar un

- 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.

Atributos de un hilo

	Obtener	Establecer
Separado o no	pthread_attr_getdetachstate (…)	pthread_attr_setdetachstate (…)
Tamaño de pila	pthread_attr_getstacksize (…)	pthread_attr_setstacksize (…)
Localización pila	pthread_attr_getstackaddr (…)	pthread_attr_setstackaddr (…)
Política de planificación	pthread_attr_getscope (…)	pthread_attr_setscope (…)

Ejemplo: atributos para los hilos

```
....

int main ( int argc, char *argv[] )
{
    pthread_attr_t attr;
    int ret;  size_t stacksize;

    ret = pthread_attr_init(&attr) ;
    ret = pthread_attr_getstacksize(&attr, &stacksize);
    ret = pthread_attr_setstacksize(&attr,   stacksize);
    ret = pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED) ;

    for (int t=0; t<NUM_THREADS; t++) {
          ret = pthread_create(&(threads[t]), &attr, th_function, (void *)(long)t);
          if (ret) { printf("ERROR from pthread_create(): %d\n", ret); exit(-1);  }
    }
    sleep(10) ;

    pthread_exit(NULL);
    ret = pthread_attr_destroy(&attr) ;
}
```

- `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 ret;  size_t st

    ret = pthread_attr_init(&attr) ;
    ret = pthread_attr_getstacksize(&attr, &stacksize);
    ret = pthread_attr_setstacksize(&attr,   stacksize);
    ret = pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED) ;

    for (int t=0; t<NUM_THREADS; t++) {
          ret = pthread_create(&(threads[t]), &attr, th_function, (void *)(long)t);
          if (ret) { printf("ERROR from pthread_create(): %d\n", ret); exit(-1);  }
    }
    sleep(10) ;

    pthread_exit(NULL);
    ret = pthread_attr_destroy(&attr) ;
}
```

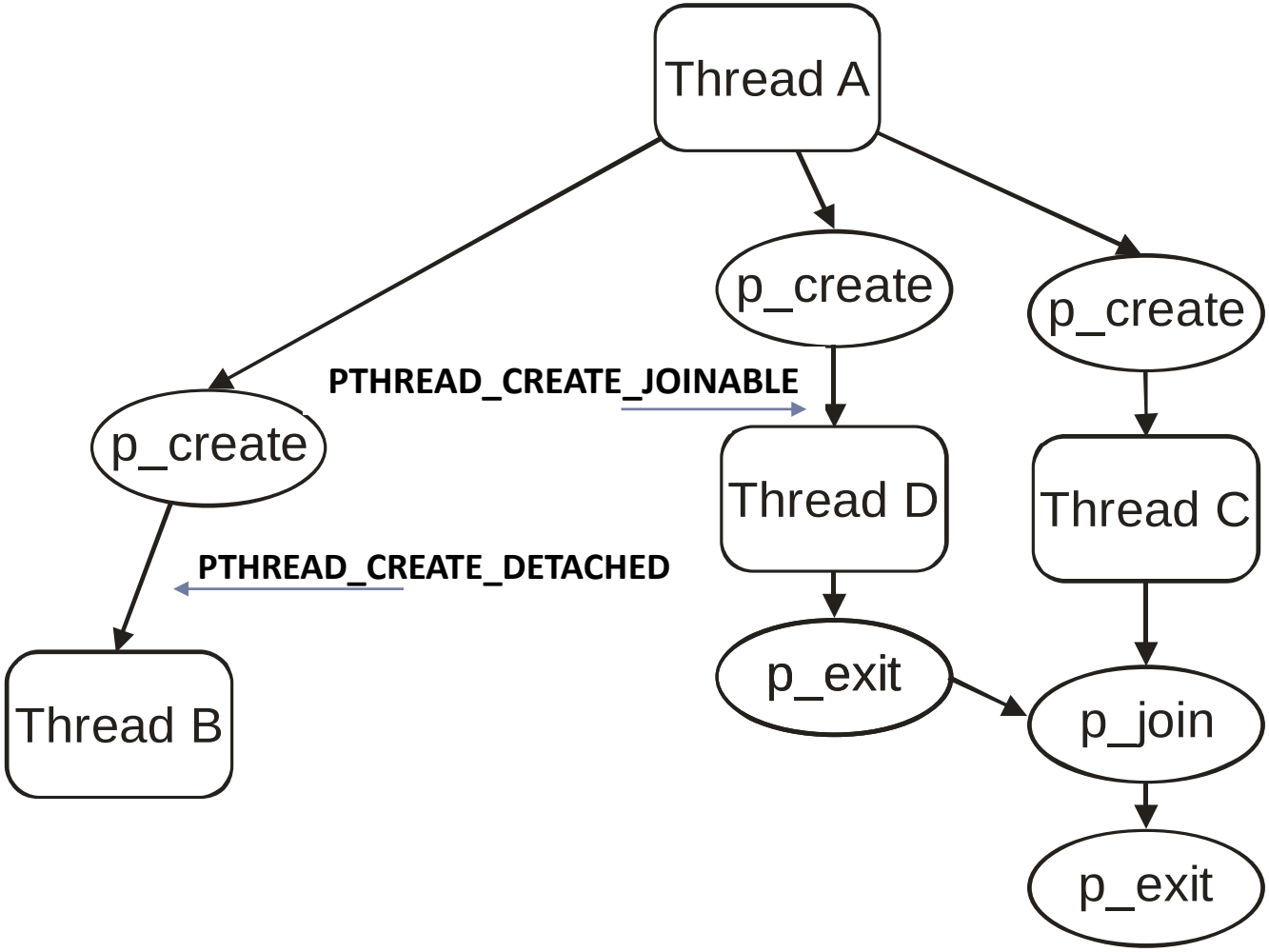
- `int pthread_attr_setdetachstate ( pthread_attr_t *attr, int detachstate);`
  - Establece el comportamiento al terminar con detachstate:
    - PTHREAD_CREATE_JOINABLE: unirse con pthread_join
    - PTHREAD_CREATE_DETACHED: liberar recursos y desacoplarse.
- `int pthread_attr_getdetachstate ( pthread_attr_t *attr, int *detachstate);`
  - Permite obtener el tipo de comportamiento al terminar.

```
    ret = pthread_a
    ret = pthread_attr_ge                (&attr, &stacksize);
    ret = pthread_attr_setstacks.e(&attr,   stacksize);
    ret = pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED) ;

    for (int t=0; t<NUM_THREADS; t++) {
          ret = pthread_create(&(threads[t]), &attr, th_function, (void *)(long)t);
          if (ret) { printf("ERROR from pthread_create(): %d\n", ret); exit(-1);  }
    }
    sleep(10) ;

    pthread_exit(NULL);
    ret = pthread_attr_destroy(&attr) ;
}
```

Ejemplo de jerarquía de *threads*



- `int pthread_attr_setscope ( pthread_attr_t *attr, int scope);`
  - Permite indicar el planificador deseado:
    - PTHREAD_SCOPE_PROCESS: hilos del mismo proceso -> PCS.
    - PTHREAD_SCOPE_SYSTEM: hilos de cualquier proceso -> SCS.
  - Es posible que el SO pueda limitar, Ej.: Linux y MacOS a usuarios -> SCS.
- `int pthread_attr_getscope ( pthread_attr_t *attr, int *scope);`
  - Permite obtener el tipo de planificador.

```
    ret = pthread_attr_...
    ret = pthread_attr_getscope(&attr, &scope);
    ret = pthread_attr_setscope(&attr,   scope);
    ret = pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED) ;

    for (int t=0; t<NUM_THREADS; t++) {
          ret = pthread_create(&(threads[t]), &attr, th_function, (void *)(long)t);
          if (ret) { printf("ERROR from pthread_create(): %d\n", ret); exit(-1);  }
    }
    sleep(10) ;

    pthread_exit(NULL);
    ret = pthread_attr_destroy(&attr) ;
}
```

- `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 ret;  size_t stacksize;

    ret = pthread_attr_init(&attr) ;
    ret = pthread_attr_getstacksize(&attr, &stacksize);
    ret = pthread_attr_setstacksize(&attr,   stacksize);
    ret = pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED) ;

    for (int t=0; t<NUM_THREADS; t++) {
          ret = pthread_create(&(threads[t]), &attr, th_function, (void *)(long)t);
          if (ret) { printf("ERROR from pthread_create(): %d\n", ret); exit(-1);  }
    }
    sleep(10) ;

    pthread_exit(NULL);
    ret = pthread_attr_destroy(&attr) ;
}
```

Grupo ARCOS
Universidad Carlos III de Madrid

Lección 3
Procesos e hilos

Sistemas Operativos
Ingeniería Informática