2.1 Paradigmas de programación paralela: memoria compartida

Computación de Altas Prestaciones
Grado en Ingeniería de Computadores

Raúl Durán Díaz

Departamento de Automática
Universidad de Alcalá

Curso académico 2018–2019

Universidad de Alcalá

Revisión: 1.14

Contenidos

1 Introducción

2 Paradigmas de uso

3 Programación avanzada

Universidad de Alcalá

Revisión: 1.14

Definición y terminología

- Concepto de hebra.
  - Comprende los elementos necesarios para ejecutar una sucesión de instrucciones máquina:
  - contador de programa, los registros de datos y direcciones, etc.
- Concepto de proceso (en Unix):
  - una o más hebras, espacio de direcciones, descriptores de ficheros, etc.
- Por lo tanto, un proceso puede estar ejecutando varias hebras que compartirán el espacio de direcciones.

Normativa

- Seguimos la norma POSIX 1003.1c – 1995
- El nombre `pthreads` se refiere a POSIX THREADS.

Hebras

- Creación y uso.
  - Creación y gestión de hebras.
- Objeto básico:

```
pthread_t thread;
```

- Funciones relacionadas:

```
int pthread_create(pthread_t *thread,
const pthread_attr_t *attr, void *(*start)(void *),
void *arg);
int pthread_exit(void *value_ptr);
int pthread_detach(pthread_t thread);
int pthread_join(pthread_t thread, void **value_ptr);
int sched_yield(void);
int pthread_equal(pthread_t t1, pthread_t t2);
```

Hebras

- La hebra se representa con el tipo de dato `pthread_t`.
- La ejecución de la hebra comienza llamando a la función que se indique en `start`, a la que se le pasará el argumento `arg`.
- El identificador de la hebra creada se almacena en la variable tipo `pthread_t`.
- El identificador es necesario para actuar sobre la hebra.
- Una hebra puede obtener su propio identificador con `pthread_self()`.
- La función `pthread_equal()` permite comparar dos identificadores.

Hebras

- El main es la hebra inicial.
  - Si se termina, se termina todo el proceso (en el sentido del sistema operativo):
    - obliga a las demás hebras a terminar;
    - devuelve al sistema operativo los recursos de todo el proceso.
  - Por lo demás se comporta como una hebra normal.
- Todas las hebras creadas comparten el espacio de direcciones y los descriptores de ficheros.
- A cada hebra se le asigna una pila propia de tamaño fijo, especificable, mediante el parámetro de atributos, en el momento de creación.

Hebras

- La función `pthread_exit()` termina una hebra.
- Una hebra terminada sólo devuelve sus recursos al sistema operativo si está en estado *detached*.
- Para poner una hebra en estado *detached* se utiliza la función `pthread_detach()`.
- El estado *detached* no afecta a la hebra: sólo informa al sistema operativo de que sus recursos se pueden recuperar cuando ésta termine.

Hebras

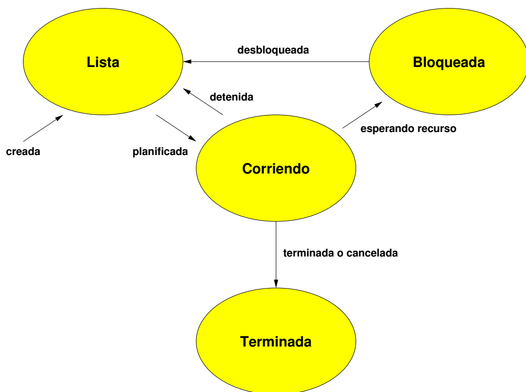

Figura: Ciclo de vida

Universidad de Alcalá
Revisión: 1.14

Hebras

- Una hebra puede devolver un valor al terminar, si otra hebra llama a la función `pthread_join()`.
- Recordemos que la sintaxis es:

```
int pthread_join(pthread_t thread, void **value_ptr);
```

- Si no nos interesa el valor retornado, podemos pasar `NULL` en lugar de `value_ptr`.

Relación entre hebras y kernel

- Entre las hebras y el procesador, suele haber un nivel de abstracción intermedio, como, por ejemplo:
  - un proceso Unix tradicional,
  - una hebra de kernel,
  - un *light-weight process*, como en Solaris, etc.
- Estas "entidades de kernel" están gestionadas por el sistema operativo.
- Éste se relaciona con las hebras a través de un subsistema que llamamos planificador.

Universidad de Alcalá

Relación hebras-kernel

El usuario puede ejercer cierto control sobre la *planificación*:

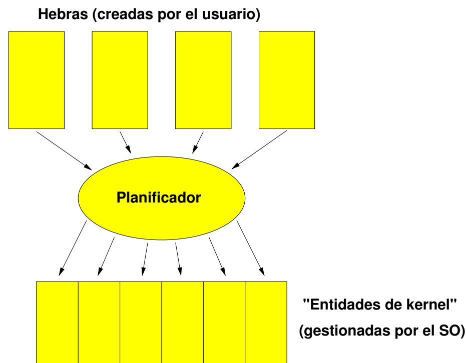

Figura: Relación entre hebras y “entidades” del kernel

Sincronización

- Con frecuencia existen unas relaciones implícitas entre diversas variables de un programa.
- Ejemplo:
    - una cola de elementos con cabecera
        - la cabecera apunta al primer elemento o es NULL;
        - cada elemento apunta al siguiente o a NULL si es el último;
- Si estas relaciones no se respetan, el programa fallará o dará resultados erróneos.

Universidad de Alcalá
Revisión: 1.14

Sincronización

- Secciones críticas:
  - Áreas de código que afectan a datos compartidos.
  - Pueden ser vistas también como invariantes de datos.
- Las relaciones entre variables se pueden "infringir" temporalmente.
- Cuando hablamos de *sincronización* nos referimos a la protección del programa frente a la alteración de las relaciones.
- La sincronización es un proceso cooperativo.
- Los *predicados* son expresiones lógicas que describen el estado de las relaciones.

Sincronización

- Mutexes (semáforos de exclusión mutua)
  - Para sincronizar las hebras hay que asegurar el acceso mutuamente excluyente a los datos compartidos.
  - La sincronización es importante para:
    - modificar datos;
    - para leer datos previamente escritos si el orden es importante.
  - ¡Atención! Con frecuencia, el hardware no nos garantiza un orden predecible en el acceso a memoria.

Creación y destrucción de mutexes

- Objeto básico:

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

- Funciones relacionadas:
  - Creación:

    ```
    int pthread_mutex_init(pthread_mutex_t *mutex,
    pthread_mutexattr_t *attr);
    ```

    - Esta función sirve para crear un mutex dinámicamente.
  - Destrucción:

    ```
    int pthread_mutex_destroy(pthread_mutex_t *mutex);
    ```

    - Un mutex creado dinámicamente se puede destruir una vez se tenga certeza de que no hay hebras que lo retengan bloqueado.

Operaciones con mutexes

- Bloqueo y desbloqueo de mutexes:

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
```

  - Esta función espera hasta conseguir el bloqueo del mutex.
  - ¡Atención! No bloquear un mutex que la misma hebra ya tenía bloqueado previamente.

```
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

  - Una hebra sólo puede desbloquear los mutexes que le pertenecen, es decir, los que ella bloqueó

```
int pthread_mutex_trylock(pthread_mutex_t *mutex);
```

  - Esta función devuelve el error `EBUSY` si el mutex ya estaba bloqueado.
  - Puede servir para evitar los “abrazos mortales”.

Sincronización

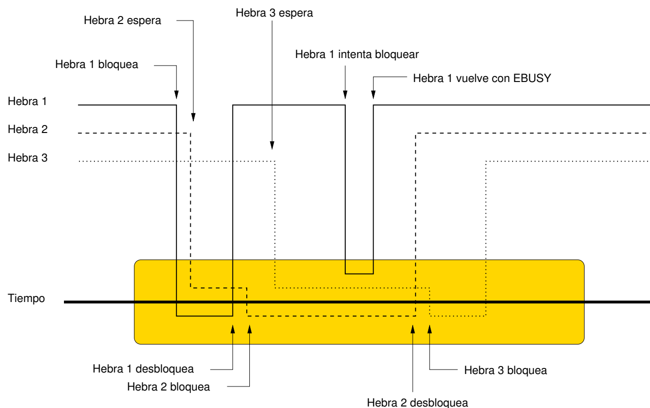

Figura: Esquema de sincronización

Universidad de Alcalá
Revisión: 1.14

Variables de estado

- Se utilizan para comunicar el estado de los datos compartidos.
- Una variable de estado siempre lleva asociado un mutex.
- La hebra bloquea el mutex y después espera cambio de estado.
- El sistema `pthreads` asegura atomicidad en las operaciones *espera cambio de estado* y *desbloqueo de mutex*.
- Las variables de estado sirven para señalizar una condición: no proporcionan exclusión mutua.
- Por eso se utilizan en conjunción con los mutexes.
- Cada variable de estado lleva asociado uno y solo un predicado.

Operaciones sobre las variables de estado

- Creación y destrucción de variables de estado.
- Objeto básico:

```
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

- Funciones relacionadas:
  - Creación:

    ```
    int pthread_cond_init(pthread_cond_t *cond,
    pthread_condattr_t *attr);
    ```

    - Crea una variable de estado dinámicamente.
  - Destrucción:

    ```
    int pthread_cond_destroy(pthread_cond_t *cond);
    ```

    - Una variable de estado creada dinámicamente se puede destruir una vez se tenga certeza de que no hay ni habrá hebras a la espera de cambio de estado, ni ninguna señalizará ningún cambio de estado.

Operaciones de espera sobre variables de condición

- Funciones de espera:
  - Espera ordinaria:

```
int pthread_cond_wait(pthread_cond_t *cond,
pthread_mutex_t *mutex);
```

    - Cada variable de estado debe estar asociada con uno y sólo un mutex.
    - Cuando la hebra invoca esta función, ha de tener bloqueado el mutex.
    - La operación de espera liberará el mutex justo al comenzar la espera.
    - Una vez señalizada la condición, la función se desbloquea y retoma el bloqueo del mutex antes de seguir adelante.

Operaciones de espera sobre variables de condición

- Funciones de espera:
  - Espera con plazo de expiración:

    ```
    int pthread_cond_timedwait(pthread_cond_t *cond,
    pthread_mutex_t *mutex, struct timespec *expiration);
    ```

    - La función puede retornar bien por señalización de la condición bien por expiración del plazo.

Puntos importantes

- Chequear siempre el predicado
  - después de obtener el bloqueo del mutex y antes de llamar a la función de espera;
  - al retornar de la función de espera.
- Lo mejor es esperar dentro de un lazo el cambio de estado.

Señalización de cambio de estado

- Señalizar a una sola hebra:
  ```
  int pthread_cond_signal(pthread_cond_t *cond);
  ```
  - Necesitamos señalizar a sólo una hebra el cambio de estado.
  - Cualquiera de las hebras en espera puede procesar ese cambio.
- Señalizar a todas las hebras simultáneamente:
  ```
  int pthread_cond_broadcast(pthread_cond_t *cond);
  ```
- No es imprescindible bloquear el mutex asociado a una variable de estado antes de señalizarla.
  - Puede ser más eficiente en muchos sistemas.
  - Pero si entre la señalización y el retorno otra hebra adquiere el mutex, la hebra señalizada ha de esperar que ésta lo libere. Puede darse así la inversión de prioridad.

Variables de condición

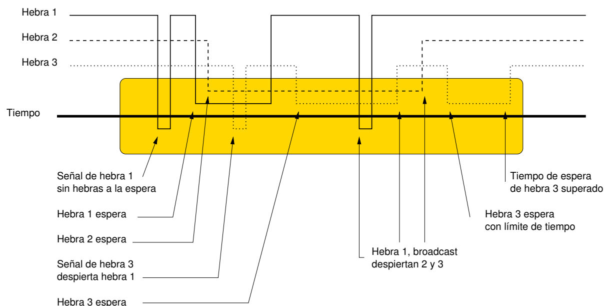

Figura: Ejemplo de operación de variables de condición

Universidad de Alcalá

Revisión: 1.14

Contenidos

1 Introducción

2 Paradigmas de uso

3 Programación avanzada

Paradigmas de uso

- Existen muchas soluciones para estructurar una solución de tipo hebra.
- Los paradigmas básicos son:
  - Modelo trabajo en cadena.
  - Modelo grupo de trabajo (*task pool*).
  - Modelo cliente-servidor.
  - Modelo maestro-esclavo.
  - Modelo productor-consumidor.

  También se pueden considerar combinaciones de esos modelos.

Modelo trabajo en cadena

- El trabajo se descompone en una serie de tareas secuenciales, con una entrada y una salida claramente definidas.
- Cada hebra realiza una o más de estas tareas secuenciales sobre conjuntos de datos sucesivos, pasando el resultado a otra hebra para realice el siguiente paso.

Modelo grupo de trabajo *task pool*

- El trabajo se descompone en tareas paralelas que se almacenan en una estructura de datos compartida (*pool*).
- Cada hebra de un número total de ellas (típicamente fijado) va al *pool*, toma una tarea y la ejecuta.
- A veces, el resultado de la ejecución es una nueva tarea, que se agrega al *pool*.
- El programa se termina cuando no quedan tareas en el *pool* y todas las hebras han terminado la que estuvieran haciendo.

Modelo cliente-servidor

- El servidor (o servidores) realiza una determinada tarea especializada.
- El cliente pide al servidor la realización de ese trabajo y, generalmente, procede a realizar otras tareas, por ejemplo, solicitar otros trabajos a otros servidores.
- El cliente se limita así a coordinar las tareas entre distintos servidores.

Modelo maestro-esclavo

- Existe una hebra principal o maestra, que coordina todo el trabajo.
- La maestra crea, administra y adjudica los trabajos a las hebras esclavas.
- Si una hebra esclava crea más trabajo, la maestra se encarga de coordinar su realización.

Modelo productor-consumidor

- En este modelo las hebras se categorizan en productoras y consumidoras.
- Las hebras productoras generan datos o trabajo que se almacena en una estructura compartida.
- Las consumidoras toman los datos o trabajo y lo van realizando (lo van consumiendo).
- El acceso a la estructura compartida ha de estar, naturalmente, bien sincronizado.

Combinación de modelos

- Los modelos se pueden combinar de cualquier modo imaginable, para adaptarse a las necesidades.
- Ejemplos:
  - Una fase de un modelo trabajo en cadena se puede implementar usando cliente-servidor.
  - Un servidor puede estar implementado usando trabajo en cadena.

Contenidos

1 Introducción

2 Paradigmas de uso

3 Programación avanzada

Universidad de Alcalá

Revisión: 1.14

Programación avanzada: resumen

- Inicialización única.
- Objetos atributo.
  - Atributos de mutex.
  - Atributos de variable de estado.
  - Atributos de hebra.
- Cancelación.
- Datos privados para cada hebra.

Inicialización única

- Se utiliza cuando hay alguna o algunas operaciones que han de ser realizadas *solamente una vez* por no importa qué hebra.
- Ejemplos: inicialización de variables, creación de mutexes, creación de datos específicos de hebra, etc.
- Normalmente usaremos una variable booleana para controlar, protegida por un mutex inicializado estáticamente.
- En un programa estas operaciones se realizan típicamente en el `main`.
- Si esto no es posible, (por ejemplo, en una librería) usamos la función `pthread_once`.

Inicialización única

- Objeto básico:

```
pthread_once_t once_control = PTHREAD_ONCE_INIT;
```

- Funciones relacionadas:

```
int pthread_once(pthread_once_t *once_control, void
(*rutina_inicio)(void));
```

  - La función `rutina_inicio` será ejecutada sólo una vez por no importa qué hebra.
  - La `pthread_once` comprueba primero la variable de control `once_control`, para determinar si se ha completado ya la inicialización.
  - Al terminar, la función llamante tiene garantizada la inicialización.

Universidad de Alcalá

Revisión: 1.14

Objetos atributo

- Un objeto atributo es una lista extendida de argumentos que se añade cuando se van a crear ciertos objetos.
- Permite que la creación de esos objetos sea “sencilla” para el neófito, pero flexible como para que un “experto” pueda sacar todo el partido.
- Se pueden aplicar a:
  - Atributos de mutex.
  - Atributos de variable de estado.
  - Atributos de hebra.
- No todas las implementaciones soportan todos los atributos.

Universidad de Alcalá
Revisión: 1.14

Atributos de mutex

- Objeto básico:

```
pthread_mutexattr_t attr;
```

- Creación:

```
int pthread_mutexattr_init(pthread_mutexattr_t *attr);
```

- Destrucción:

```
int pthread_mutexattr_destroy(pthread_mutexattr_t *attr);
```

- Atributo soportado: compartición del mutex entre procesos.

```
int pthread_mutexattr_getpshared(pthread_mutexattr_t *attr,
int *pshared);
int pthread_mutexattr_setpshared(pthread_mutexattr_t *attr,
int pshared);
```

Atributos de variable de condición

- Objeto básico:

  ```
  pthread_condattr_t attr;
  ```

- Creación:

  ```
  int pthread_condattr_init(pthread_condattr_t *attr);
  ```

- Destrucción:

  ```
  int pthread_condattr_destroy(pthread_condattr_t *attr);
  ```

- Atributo soportado: compartición de la variable de condición entre procesos.

  ```
  int pthread_condattr_getpshared(pthread_condattr_t *attr, int
  *pshared);
  int pthread_condattr_setpshared(pthread_condattr_t *attr, int
  pshared);
  ```

Universidad de Alcalá

Revisión: 1.14

Atributos de hebra

- Objeto básico:

```
pthread_attr_t attr;
```

- Creación:

```
int pthread_attr_init(pthread_attr_t *attr);
```

- Destrucción:

```
int pthread_attr_destroy(pthread_attr_t *attr);
```

Universidad de Alcalá
Revisión: 1.14

Atributos soportados para hebra

- *Detach state*:
  - Si el identificador de hebra puede usarse para hacer *join*, se trata de una hebra JOINABLE.
    - El valor del atributo será `PTHREAD_CREATE_JOINABLE`.
  - Si cuando la hebra termina, puede devolver inmediatamente todos sus recursos al sistema operativo, la hebra es DETACHED.
    - El valor del atributo será `PTHREAD_CREATE_DETACHED`.
  - Por defecto las hebras se crean en modo JOINABLE.

```
int pthread_attr_getdetachstate(pthread_attr_t *attr,int
*detachstate);
int pthread_attr_setdetachstate(pthread_attr_t *attr,int
detachstate);
```

Atributos soportados para hebra

- Tamaño de pila:
  - En algunos casos puede ser interesante modificar la cantidad de memoria reservada para la pila.
    - Ejemplo: si la jerarquía de llamadas a funciones baja demasiados niveles.
  - Atención: cambiar el tamaño de la pila hace el código poco portátil.

```
int pthread_attr_getstacksize(pthread_attr_t *attr,size_t
*stacksize);
int pthread_attr_setstacksize(pthread_attr_t *attr,size_t
stacksize);
```

Cancelación

- Las hebras suelen terminar por sí mismas, pero a veces es interesante cancelarlas.
  - El usuario presiona el botón "Cancelar".
  - Una hebra es parte de un algoritmo redundante en que otra hebra ya ha obtenido resultados válidos.
- Cancelar una hebra no la mata automáticamente: depende de su estado de cancelación.
  - Es, más bien, pedirle de forma "educada" que termine.
- Necesitamos su identificación:

```
int pthread_cancel(pthread_t thread);
```

Universidad de Alcalá

Revisión: 1.14

Cancelación

- Se codifica mediante dos bits:
  - Bit de estado:
    - enable
    - disable
  - Bit de tipo:
    - deferred
    - asynchronous
- Por defecto, la cancelación es deferred. Significa que sólo puede ocurrir en ciertos puntos del programa, en que la propia hebra comprueba si se le ha pedido terminar.
  - Ejemplo: al esperar el cambio de una variable de condición, al leer o escribir un fichero, etc.
  - Se puede pedir explícitamente la comprobación mediante `pthread_testcancel(void);`.

Cancelación

De las funciones que hemos visto, las siguientes siempre son puntos de comprobación de cancelación:

- `pthread_cond_wait`
- `pthread_cond_timedwait`
- `pthread_join`
- `pthread_testcancel`

Generalmente hay más, pero no está garantizado por la norma Posix.

Cancelación

- En el tipo asíncrono indicamos que la hebra puede finalizar inmediatamente, sin ulteriores comprobaciones.
- Funciones de cambio de tipo y estado:

```
int pthread_setcancelstate(int state, int *oldstate);
```

  - Los valores de `state` pueden ser:
    - `PTHREAD_CANCEL_ENABLE`
    - `PTHREAD_CANCEL_DISABLE`

```
int pthread_setcanceltype(int type, int *oldtype);
```

  - Los valores de `type` pueden ser:
    - `PTHREAD_CANCEL_DEFERRED`
    - `PTHREAD_CANCEL_ASYNCHRONOUS`

Cancelación

- Funciones para insertar una rutina de limpieza antes de que se ejecute la cancelación:

```
int pthread_cleanup_push(void (*rutina)(void *), void *arg);
```

  - Envía la rutina `rutina` a la pila de *handlers* de limpieza.
  - Será ejecutada:
    - Si es cancelada la hebra.
    - Si la hebra ejecuta `pthread_exit`.
    - Si la hebra ejecuta `pthread_cleanup_pop` con un valor de argumento distinto de cero.

```
int pthread_cleanup_pop(int execute);
```

  - Saca de la pila el *handler* introducido por una llamada a la `pthread_cleanup_push`.
  - Además lo ejecuta si el argumento es distinto de cero.

Datos privados para cada hebra

- A veces es necesario que una función conserve datos persistentes de una llamada a otra, es decir, que pueda tener un estado.
- Obviamente necesitamos que ese estado sea particular (= privado) en cada hebra que esté ejecutando en un momento dado (o haya ejecutado) esa función.

La solución es dotar de *datos privados* a cada hebra que ejecuta un código, que irán asociados con una *clave*.

Datos privados para cada hebra

Se utilizan las funciones siguientes:

- `pthread_key_create`
- `pthread_key_delete`
- `pthread_setspecific`
- `pthread_getspecific`

Creación/destrucción de la clave

Tenemos el tipo de dato `pthread_key_t`.

- Para crear una clave:
  - `int pthread_key_create(pthread_key_t *key, void (*destructor)(void *));`
- Para destruir una clave:
  - `int pthread_key_delete(pthread_key_t key);`

¡Importante! La clave se debe crear solo una vez.

Establecimiento y obtención de datos privados

- Para establecer un dato privado:
  - `int pthread_setspecific(pthread_key_t key, const void *dato);`
- Para obtener un dato privado:
  - `void *pthread_getspecific(pthread_key_t key);`

Seguridad y eficiencia

- Código seguro en el modelo de hebra:
  - Llamamos código seguro al que puede ser ejecutado simultáneamente por varias hebras sin resultados destructivos.
- Atención: no es lo mismo “seguro” que “eficiente”.

Protección de datos y código

- Hebra re-entrante:
    - evitar datos estáticos (persistentes de una llamada a otra);
    - independiente de cualquier forma de sincronización entre las hebras;
    - guardar el estado sobre estructuras de datos externas a la función.

Universidad de Alcalá
Revisión: 1.14

Protección de datos y código

- Protección de código frente a protección de datos.
- Conceptualmente es mejor proteger los datos que el código.

Costes de la programación multihebra

- Sobrecarga de computación.
- La sincronización tiene coste computacional: hay que mantenerla al mínimo.
- Deficiencias a niveles más bajos.
- Si se usan hebras para paralelizar la I/O, pueden aparecer cuellos de botella en:
  - Librería ANSI C.
  - Sistema operativo.
  - Sistema de ficheros.
  - Drivers de los dispositivos.
- En procesos de computación intensiva, adecuar el número de hebras al de procesadores (fase de mapeo).

Costes de la programación multihebra

- Autodisciplina del programador.
- La programación con hebras es más difícil.
- Hay que ser cautos con el código desarrollado por terceros.
- Dificultad de depuración.
- Faltan herramientas.
- Los errores son sutiles:
  - La temporización es clave y puede hacer aparecer o desaparecer errores.
  - La corrupción de punteros es muy difícil de rastrear.

Errores frecuentes

- “Inercia” de las hebras.
- Competición entre las hebras.
- Evitar abrazos mortales.
- Inversión de prioridad.
- Compartición de variables de estado en diferentes predicados.
- Compartición de variables privadas.
  - Pila
- Problemas de rendimiento.

"Inercia" de las hebras

- Recordar siempre que las hebras son asíncronas.
- En un sistema monoprocesador, una hebra que crea otra tiene una cierta ventaja sobre ella en cuanto al momento del comienzo de ejecución: hay un "ligero sincronismo".
- También puede pasar lo mismo en un multiprocesador si se ha alcanzado el límite de procesadores disponibles.
- Todo esto crea la ilusión de la "inercia".
- Sin embargo, nunca se debe basar el código en suponer que una hebra recién creada "tarda" en arrancar.

Competición entre las hebras

- La competición se da cuando dos o más hebras quieren llegar o hacer lo mismo a la vez.
- Sólo una de ellas gana la competición.
- Cuál gana viene dado por muchos factores, muchos de ellos fuera de control.
- No olvidar que:
  - una hebra puede ser interrumpida en cualquier punto arbitrario por un plazo indefinido de tiempo;
  - no existe más orden entre ellas que el causado por el programador;
  - planificación no es lo mismo que sincronización.

Evitar abrazos mortales

- Los abrazos mortales surgen por sincronización errónea.
- La hebra A tiene el recurso 1 y no puede seguir hasta tener el recurso 2; la hebra B tiene el recurso 2 y no puede seguir hasta tener el recurso 1.
- Los recursos más conflictivos suelen ser los mutexes.
  - Se debe evitar bloquear más de un mutex simultáneamente.

Inversión de prioridad

- Este problema suele estar asociado a las aplicaciones de tiempo real.
- Ocurre cuando una hebra de alta prioridad es bloqueada en su ejecución por otra hebra de baja prioridad.
- Ejemplo:
  - una hebra de baja prioridad bloquea un mutex;
  - es interrumpida por una hebra de alta prioridad que quiere bloquear el mismo mutex y, por lo tanto, queda bloqueada;
  - una tercera hebra de prioridad media impide la ejecución —y desbloqueo del mutex— de la hebra de baja prioridad;
  - Resultado: la hebra de alta prioridad queda indefinidamente bloqueada.

Errores por compartición

- Compartición de variables de estado en diferentes predicados.
  - Cada variable debe usarse para comprobar una y sólo una condición.
    - Si no se hace así, puede que la señalización de la condición despierte a la hebra que no esperaba por ella... y el programa se para.
- Compartición de variables privadas.
  - Pila.
    - Se puede hacer siempre que se garantice que la función a la que pertenece la pila compartida no hace "return" mientras el resto de hebras usuarias no hayan terminado de hacer uso de ella.
    - ¡Atención a los punteros no inicializados!

Problemas de rendimiento

- Atención al paralelismo “serializado”.
- Usar un sólo “gran” mutex para acceder a una librería es un paralelismo muy grosero. . .
- . . . pero usar demasiados mutexes penaliza fuertemente el rendimiento.
- Para evitar contención con el cache, es bueno alinear y separar datos usados por diferentes hebras y que afecten sustancialmente al rendimiento.