

Threads

- Los threads son similares a procesos
- En sistemas con un solo procesador se divide el tiempo asignado a cada thread para simular la ejecución simultánea igual que para los procesos
- En sistemas con más de un procesador, pueden ejecutarse simultáneamente.

Ventajas:

- comparten la misma ubicación en memoria
- Threads independientes pueden acceder a las mismas variables en memoria pueden leer o escribir enteros declarados globalmente

Threads

En procesos:

- uso fork() permite crear múltiples procesos
- problema de comunicación: cómo conseguir que múltiples procesos, cada uno con su propio espacio en memoria, se comuniquen.
- No hay una respuesta simple a este problema
- IPC local (comunicación entre procesos), desventajas:
 - Imponen sobrecarga al núcleo, disminuye rendimiento.
 - IPC incrementa la complejidad del código.
- Con una pequeña sincronización, todos los threads podrán leer y modificar las estructuras de datos de nuestros programas

Threads

- Threads son más rápidos
- El kernel no necesita crear una nueva copia independiente del espacio de memoria del proceso, de los descriptores del proceso, etc.
- Ahorra tiempo CPU: creación de diez a cien veces más rápida que creación de nuevo proceso
- Igual que lprocesos, ventaja con múltiples CPUs
- El rendimiento de programas que hacen un uso intensivo de la CPU escalarán linealmente con respecto al número de procesadores
- La llamada UNIX clone() ofrece muchos de los beneficios de los threads, pero no es portable y sólo podrá usarse bajo Linux

Threads

- Pthread es una librería para C de POSIX (Portable Operating System Interface),
- Funciones estandarizadas para el uso de threads en diferentes plataformas.
- Antes: desarrolladores de hardware implementaban versiones privadas para uso de threads, diferían en diferentes plataformas
- En los sistemas UNIX, esta interfaz ha sido especificada por el estándar IEEE POSIX 1003.1c en 1995.
- Pthreads conjunto de tipos y llamadas a funciones en C.
- Los desarrolladores de hardware proveen implementación de Pthreads en forma de archivos que deben ser incluidos en el programa, y una librería a la que debe hacerse referencia al ejecutar el programa.

Threads

- Estándares POSIX son flexibles, diseñados para distintos ambientes
- UNIX no brinda soporte para scheduling de tiempo real
- Que UNIX cumpla con el estándar “IEEE POSIX 1003.1c” no implica que se puedan escribir programas de tiempo real predecibles
- Si el sistema define `_POSIX_THREAD_PRIORITY_SCHEDULING`,
- da soporte a asignación de prioridades para scheduling TR
- Cada thread debe tener definidos dos atributos:
 - Una prioridad
 - Una política de scheduling, que indicará cómo comparten la CPU, los threads que tienen la misma prioridad y están listos para ejecutar (es decir, en estado “ready”).

Operaciones manejo de prioridades

- `int sched_get_priority_max(int policy);` devuelve máximo valor usado en algoritmo de scheduling como prioridad identificado por `policy`
- `int sched_get_priority_min(int policy);` devuelve el mínimo
- **Parámetro `policy`:**
 - `SCHED_FIFO`
 - `SCHED_RR`
 - `SCHED_OTHER`
- **Manejo de la política:**
 - `int pthread_attr_setschedparam(pthread_attr_t *attr, const struct sched_param *param);`
 - `sched_param`, contiene atributo `sched_priority`, usado para setear prioridad mediante la instrucción: `param.sched_priority= priority` (un entero)
 - `pthread_attr_t`, se usa para setear prioridad y política
 - `pthread_attr_setschedparam()`, permite pasarl valores de `param` a `attr` int
 - `pthread_attr_setschedpolicy(pthread_attr_t *attr, int policy);` setea política de atributos de un thread al que hace referencia `attr` con el valor `policy`

Políticas de Scheduling

- Pthread define dos políticas: SCHED_FIFO y SCHED_RR
- SCHED_FIFO: planificación “First In First Out” (orden llegada)
 - se ejecute hasta que otro thread con prioridad más alta esté en estado “ready” o hasta que se bloquee voluntariamente. Cuando un thread con la política de scheduling SCHED_FIFO pasa a estado “ready”, se comienza a ejecutar inmediatamente a menos que un thread con igual o mayor prioridad ya se esté ejecutando.
- SCHED_RR: planificación Round Robin
 - tiempo máximo de uso CPU (quantum) a cada thread
 - Pasado: desalojado y retorna a estado listo
 - la lista de procesos se planifica por FIFO
 - si un thread con SCHED_RR se ejecuta más que cuántum sin bloquearse, y otro thread con política de scheduling SCHED_RR o SCHED_FIFO y la misma prioridad se encuentra listo, el thread que se encuentra corriendo (en estado “running”) perderá el procesador así el thread en estado “ready” puede ejecutarse

Políticas de Scheduling

- Pthreads define un nombre adicional para políticas de scheduling, llamado `SCHED_OTHER`
- No define el comportamiento que esta política debería tener
- Cuando se usa, se podría escribir un programa portable que cree threads con la política `SCHED_OTHER`, pero el comportamiento del programa no será portable, porque esta política podrá ser implementada de forma distinta en cada sistema. La explicación oficial de la opción `SCHED_OTHER`, es que provee una forma de declarar que el programa no necesita políticas de scheduling de tiempo real.

Políticas de Scheduling

- A cada thread le puedo asignar política y prioridades diferentes
- Se puede modificar esto de forma dinámica
- Un diseñador de aplicaciones de tiempo real, primero hará una división entre tareas de tiempo real y las menos críticas
- Las tareas de tiempo real, se programarán en un thread cuya política de scheduling sea `SCHED_FIFO`, y tendrán una prioridad alta
- Las tareas restantes, tendrán asignada la política `SCHED_RR`, y una prioridad menor
- Las prioridades de estos threads, deberán ser mayores, que las de cualquier otro thread en el sistema
- Cuando un thread de tiempo real obtiene la CPU, terminará de ejecutar sin interrupciones a menos que se bloquee

Políticas de Scheduling

- Debido a su alta prioridad, los threads pertenecientes a la aplicación de tiempo real, que no sean tan críticos, tendrán tratamiento preferencial, pero compartirán la CPU cuando sus quantums expiren.
- Estos threads harán procesamiento background de la aplicación.
- Ejemplo: procesamiento para equipos químicos
 - Los threads encargados del control de hardware (lecturas de sensores, Computar nuevos valores, enviar señales para ejecutar acción) con política SCHED_FIFO y alta prioridad
- El resto de los threads de tareas menos críticas (actualizar registros de qué empleados manejaron el equipo, en qué horario) se ejecutan con política SCHED_RR y prioridad menor.

Alcance del scheduling

- Qué threads deben competir entre sí para ser elegidos por el scheduler
- Una implementación debería permitir realizar scheduling a nivel proceso o sistema.
- A nivel de proceso:
 - threads seleccionados de un mismo programa
 - El SO elige un proceso para ejecutar, y algún scheduler adicional aplica las reglas de scheduling de POSIX, para determinar qué thread del proceso se ejecutará
- A nivel de sistema:
 - seleccionados entre todos los threads activos del sistema.....
- El estándar requiere un scheduler que pueda trabajar a nivel de proceso para comparar las prioridades de los threads solo con la de los threads del mismo proceso
- Cómo hará esta comparación no está definido
- como resultado, las prioridades seteadas por la librería Pthreads, podrían no tener relación con las prioridades del sistema

Alcance del scheduling

- A nivel de proceso de cada proceso se elegiría el thread con mayor prioridad, y cuando el proceso al que pertenece adquiera la CPU
- será el thread que se ejecutará
 - En un sistema multiprocesador y un proceso que tiene 3 threads, uno de alta prioridad, y dos de prioridad media
 - Se asigna al thread de alta prioridad una CPU
 - Si hubiese otras CPUs corriendo threads con menor prioridad de otros procesos, que los dos threads restantes del proceso, los dejado esperando a que el thread de alta prioridad finalice o se bloquee
 - Puede negarle a una aplicación multithread la posibilidad de ejecutarse en múltiples CPUs al mismo tiempo.
- Si los threads de un proceso con alcance a nivel sistema, tienen prioridad lo suficientemente alta, serían asignados a múltiples CPUs al mismo tiempo.
- Nivel sistema, es más útil que a nivel proceso para aplicaciones de tiempo real, o procesamiento paralelo

Afinidad de procesador

- Entorno de memoria compartida multiprocesador, si un proceso o thread fue asignado a un core, tendrá parte de sus datos copiados en la cache de dicha unidad
- Si el dispatcher decidiera que el proceso tiene que ser ejecutado en otra core, datos se invalidan y vueltos a cargar en la cache de la nueva core
- Procesadores actuales: varios niveles de cache, algunas son compartidas por varios cores
- El SO conoce estructura de las CPU, buscará hacer migraciones más baratas
- No se puede enunciar una regla general sobre qué ocurriría durante una migración de un thread o proceso a otro core
- Si un proceso ejecutado en un core, vuelva a ser ejecutado, se elija la misma para evitar context switch
- este concepto se lo conoce como "affinity": un proceso o thread "tiene afinidad" por un core

Afinidad de procesador

- 2 tipos: suave y dura
 - Suave: un proceso con afinidad a un core, puede ser ejecutado en otro por decisión del dispatcher por determinados patrones de carga, por ejemplo, una cantidad mucho mayor de procesos afines a cierta unidad de procesamiento que a otra, que se encuentren saturando la cola de procesos listos de esa unidad de procesamiento mientras otra tiene tiempo disponible
 - Dura (ofrecida por algunos SO): se garantiza que un proceso siempre será ejecutado en un core o un conjunto de cores determinados
- Es deseable tener control de esto en un programa:
- Si un thread realiza tarea crítica que no debe ser interrumpido por otros
- En este caso el thread será asignado a un core que ningún otro proceso o thread tenga permisos de usar
- El acceso a ciertos recursos (RAM, I/O) tiene diferentes costos desde diferentes cores (arquitectura NUMA ,Non-Uniform Memory Architecture)
 - La memoria debería ser accedida localmente pero este requerimiento no es usualmente visible al scheduler
 - Forzar a un thread a cores que tienen acceso local a la memoria que se usa mayormente mejora la performance.

Afinidad de procesador

- En ambientes de ejecución controlada la asignación de recursos y el trabajo de "book-keeping" o mantenimiento del ambiente (Ej: garbage collection) la performance es local a los cores
- Ayudar a reducir costo de bloqueos si los recursos no tienen que ser protegidos del acceso concurrente desde varios procesos diferentes.
- También aplicaciones que usan un número determinado de cores por licencia

Taskset

- Entorno multiprocesador a veces es deseable dejar sin efecto el scheduler del kernel y darle afinidad a cierto proceso a un core
- El scheduler tratará de mantener a los procesos en la misma unidad de procesamiento por cuestiones de performance
- Forzar una afinidad con un core es útil en ciertas aplicaciones.
- "taskset" permite obtener la afinidad de un proceso en ejecución dado su process ID o ejecutar un nuevo comando (proceso) dada una cierta afinidad
- En las distribuciones linux más antiguas taskset no se encontraba instalado por defecto, había que instalar el paquete "schedutils" ("Linux scheduler utilities")
- Últimas versiones de Linux, instalada por defecto en el paquete "util-linux"

Taskset

- La afinidad con un core se hace mediante máscara de bits
- Una máscara podría especificar más CPUs de las que existen en el sistema, la máscara sólo tendrá efecto sobre los bits que corresponden a los cores que se encuentran físicamente en el sistema
- Si se especifica una máscara inválida se devuelve un error
- Si se quiere indicar que para una unidad de procesamiento el proceso no tendrá afinidad se usa el bit 0, caso contrario el bit 1
- Las máscaras se especifican típicamente en hexadecimal
 - Ej: 0x00000001 especifica procesador = 0, 0x00000003 a 0 y 1, 0xFFFFFFFF a todos (desde el 0 al 31)
- Demás parámetros especifican:
 - -a, --all: Asigna o devuelve afinidad de todos los threads de un ID de proceso dado
 - -p, --pid: Opera sobre un ID de proceso existente sin ejecutar una nueva tarea
 - -c, --cpu-list: Especifica lista numérica de procesadores en lugar de una máscara de bit. Los números deben estar separados por comas y pueden incluir rangos. Por ejemplo: 0,5,7,9-11.

Taskset

- si ejecutamos:
- `taskset -c 0 ./ejemplo`
 - Asignando al programa “ejemplo” afinidad con el procesador 0, lo que obligadamente fuerza a ese programa a ejecutarse en ese único core, como todos los threads que el programa “ejemplo” pueda tener, tendrían el mismo pid, todos se ejecutarán en ese mismo procesador

Manejo de afinidad

- La librería Pthread brinda dos operaciones para manejo de afinidad:
- `pthread_setaffinity_np()`: especifica para un determinado thread con cuál o cuáles unidades de procesamiento tendrá afinidad
- `pthread_getaffinity_np()`: devuelve un conjunto de unidades de procesamiento con las cuales tiene afinidad
 - Ambas operaciones tienen tres parámetros, el primero de tipo `pthread_t`, que indica el thread al que se hace referencia, el segundo del tipo `size_t` que indica el tamaño del tercer parámetro, y el tercero es un conjunto del tipo `cpu_set_t`, que contiene las CPU con las cuales el thread tiene o tendrá afinidad.
- El tipo `cpu_set_t`, es sencillo de usar:
 - `cpu_set_t cpuset;` -----> se declara un conjunto del tipo `cpuset`
 - `CPU_ZERO(&cpuset)`-----> la función `CPU_ZERO` recibe como parámetro el puntero a un conjunto y lo inicializa vacío
 - `CPU_SET(cpu, &cpuset)`----> la función `CPU_SET` agrega `cpu` (representado por un número entero, al conjunto `cpuset`.
 - `CPU_ISSET(cpu, &cpuset)`--> se usa para saber si `cpu` está incluido en el conjunto `cpuset` en ese mismo procesador

Threads

- Mejor alternativa: threads POSIX (pthreads)
- Portable: código que funcione bajo Solaris, FreeBSD, Linux y otros

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

void *thread_function(void *arg) {
    int i;
    for ( i=0; i<20; i++ ) {
        printf("Thread says hi!\n");
        sleep(1);
    }
    return NULL;
}

int main(void) {
    pthread_t mythread;

    if ( pthread_create( &mythread, NULL,
        thread_function, NULL ) ) {
        printf("error creating thread.");
        abort();
    }

    if ( pthread_join ( mythread, NULL ) ) {
        printf("error joining thread.");
        abort();
    }

    exit(0);
}
```

- Compilar y ejecutar:
 - se guarda como thread1.c
 - \$ gcc thread1.c -o thread1 -lpthread
 - \$./thread1

Threads

- `main()`: tipo `pthread_t`, definido en `pthread.h`, (thread id)
- `pthread_create()`: devuelve cero si todo va bien. `if()` detecta falla
- Argumentos:
 - puntero hacia `mythread`, `&mythread`
 - `NULL`, puede ser usado para definir ciertos atributos (argumentos por defecto)
 - nombre de la función que el nuevo thread ejecutará cuando comience.
- muestra en pantalla: "Thread says hi!" 20 veces y termina
- Pasar argumento: usamos el cuarto argumento de la llamada a `pthread_create()`
- Aca definido `NULL` (no necesita pasar ningún dato)
- El programa en dos hilos: programa principal también se considera un hilo
- el thread principal ejecuta secuencialmente "if (pthread_join(...))"
- Se detiene y espera a combinarse con otro thread (proceso de limpieza)
- `thread_function()` tarda 20 segundos en completarse
- Antes de que `thread_function()` concluya, `main()` llama a `pthread_join()`, se duerme) y espera que `thread_function()` concluya
- Cuando `thread_function()` termine, `pthread_join()` retornará
- Si no se une a thread, seguirá contando para el límite total de hilos del sistema
- Si no se hace una limpieza adecuada de hilos, podría causar que las nuevas llamadas a `pthread_create()` fallen.

Threads

```
/* simple.c -- multithreaded "hello world" */
#ifdef __linux__
# define _REENTRANT
# define _POSIX_SOURCE
#endif
#ifdef __linux__
# define _P __P
#endif

#include <pthread.h>
#include <string.h> /* for strerror() */
#include <stdio.h>

#define NTHREADS 4
#define errexit(code,str) \
    fprintf(stderr,"%s: %s\n",(str),strerror(code)); \
    exit(1);

/****** this is the thread code */
void *hola(void * arg)
{
    int myid=*(int *) arg;

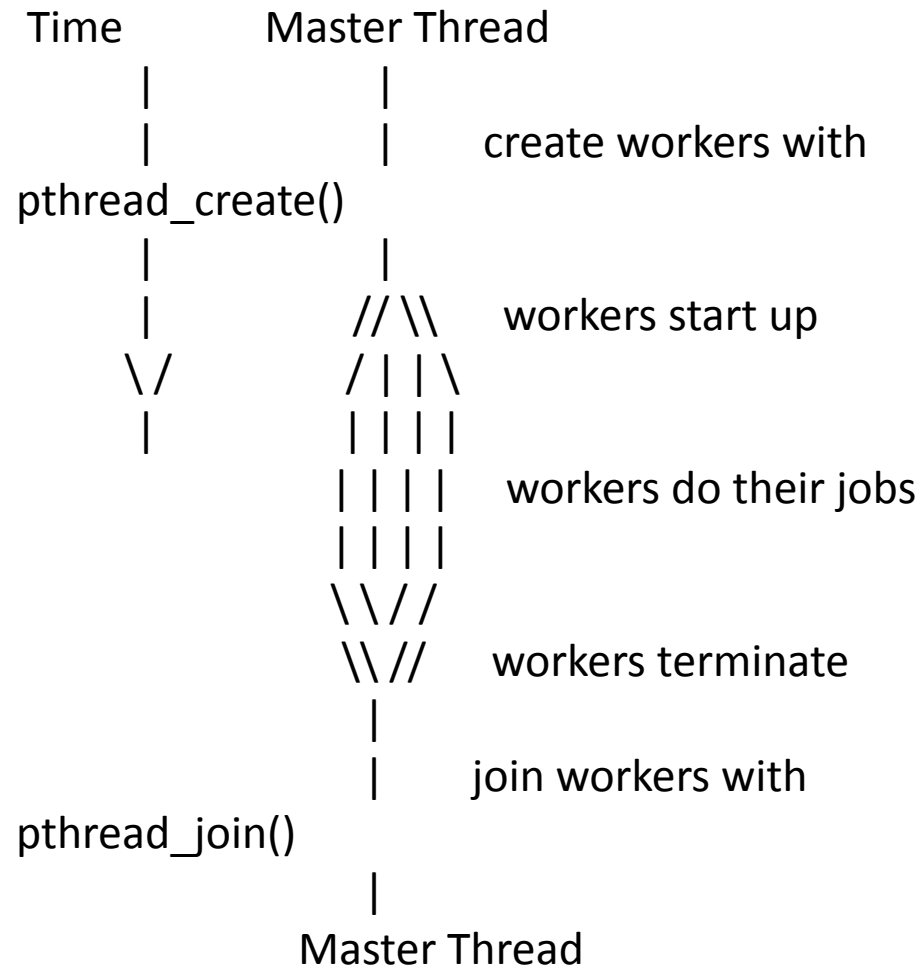
    printf("Hello, world, I'm %d\n",myid);
    return arg;
}
```

```
/****** this is the main thread's code */
int main(int argc,char *argv[])
{
    int worker;
    pthread_t threads[NTHREADS];
    /* holds thread info */
    int ids[NTHREADS]; /*
    holds thread args */
    int errcode; /* holds
    pthread error code */
    int *status; /* holds
    return code */

    /* create the threads */
    for (worker=0; worker<NTHREADS;
    worker++) {
        ids[worker]=worker;
        if
        (errcode=pthread_create(&threads[worker]
        ,/* thread struct */
        NULL, /*
        default thread attributes */
        hola, /* start
        routine */
        &ids[worker])) { /*
        arg to routine */
            errexit(errcode,"pthread_create");
        }
    }
}
```

```
/* reap the threads as they exit */
for (worker=0; worker<NTHREADS;
worker++) {
    /* wait for thread to terminate */
    if
    (errcode=pthread_join(threads[worker],(voi
    d *) &status)) {
        errexit(errcode,"pthread_join");
    }
    /* check thread's exit status and release
    its resources */
    if (*status != worker) {
        fprintf(stderr,"thread %d terminated
        abnormally\n",worker);
        exit(1);
    }
    return(0);
}

/* EOF simple.c */
```



Threads

- `fork()` implica proceso padre y proceso hijo (nuevo proceso)
- Esto crea una relación jerárquica cuando se espera a que procesos hijo concluyan
- La función `waitpid()`, por ejemplo, indica al proceso actual que espere a que los procesos hijo concluyan
- `waitpid()` se usa para implementar una sencilla rutina de limpieza en nuestro proceso padre.
- Con threads POSIX esta relación jerárquica no existe
- Un threads principal puede crear otro threads, y este crear otro nuevo
- Pero POSIX ve todos los hilos como conjunto de elementos idénticos
- No espera a que un proceso hijo concluya: se debe especificar el hilo al que estamos esperando indicando la tid adecuada a `pthread_join()`
- Si thread1 crea thread 2, no es necesario para thread1 llamar a `pthread_join()` para thread2
- Cualquier otro thread en el programa puede hacerlo: se puede crear, por ejemplo, una "lista muerta" global que contenga todos los hilos detenidos y tener otro hilo de limpieza especial, que sencillamente espera a que algún elemento se añada a esta lista
- El hilo de limpieza llama a `pthread_join()` para enhebrarlo consigo mismo. Ahora, todo el proceso de limpieza será manejado de forma cómoda y eficiente con un simple hilo.

Threads

```
/*thread2.c*/
#include <pthread.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>

int myglobal;

void *thread_function(void *arg) {
    int i,j;
    for ( i=0; i<20; i++ ) {
        j=myglobal;
        j=j+1;
        printf(".");
        fflush(stdout);
        sleep(1);
        myglobal=j;
    }
    return NULL;
}
```

```
int main(void) {

    pthread_t mythread;
    int i;

    if ( pthread_create( &mythread, NULL,
        thread_function, NULL ) ) {
        printf("error creating thread.");
        abort();
    }

    for ( i=0; i<20; i++ ) {
        myglobal=myglobal+1;
        printf("o");
        fflush(stdout);
        sleep(1);
    }

    if ( pthread_join ( mythread, NULL ) ) {
        printf("error joining thread.");
        abort();
    }

    printf("\nmyglobal equals %d\n",myglobal);

    exit(0);
}
```

Threads

- Tanto main() como el nuevo incrementan myglobal, 20 veces
- Compilamos y ejecutamos:
\$ gcc thread2.c -o thread2 -lpthread
\$./thread2
..0.0.0.0.00.0.0.0.0.0.0.0.0.0.0..0.0.0.0.0
myglobal equals 21
- myglobal comienza = 0, 2 threads lo incrementan en 20, myglobal debía ser = 40
- thread_function(): copia myglobal a v. local j, dormimos 1 segundo, y copia j a myglobal
- Main() incrementa myglobal después de que el thread copie myglobal en j
- Cuando thread_function() vuelve a escribir j en myglobal, sobrescribe lo de main()
- Evitar uso de variable local e incrementar myglobal acá funciona pero no es correcta:
- Los threads ejecutan simultáneamente, incluso aún con un solo procesador (núcleo usa la división de tiempo para simular multitarea real) podemos, desde el punto de vista de un programador, imaginar que ambos hilos se ejecutan simultáneamente
- thread2.c tiene problemas porque el código en thread_function() asume que myglobal no será modificada durante ~1 segundo antes de que sea incrementada. Tenemos que encontrar una forma de que un hilo le indique al otro que "espere" mientras se están haciendo los cambios a myglobal

Threads

```
/*Codigo modificado*/

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

int myglobal;
pthread_mutex_t
mymutex=PTHREAD_MUTEX_INITIALIZER;

void *thread_function(void *arg) {
    int i,j;
    for ( i=0; i<20; i++ ) {
        pthread_mutex_lock(&mymutex);
        j=myglobal;
        j=j+1;
        printf(".");
        fflush(stdout);
        sleep(1);
        myglobal=j;
        pthread_mutex_unlock(&mymutex);
    }
    return NULL;
}
```

```
int main(void)
{
    pthread_t mythread;
    int i;
    if ( pthread_create( &mythread, NULL,
        thread_function, NULL) )
    {
        printf("error creating thread.");
        abort();
    }

    for ( i=0; i<20; i++)
    {
        pthread_mutex_lock(&mymutex);
        myglobal=myglobal+1;
        pthread_mutex_unlock(&mymutex);
        printf("o");
        fflush(stdout); sleep(1);
    }
    if ( pthread_join ( mythread, NULL ) )
    {
        printf("error joining thread.");
        abort();
    }
    printf("\nmyglobal equals %d\n",myglobal);
    exit(0);
}
```

Threads

- Las llamadas `pthread_mutex_lock()` y `pthread_mutex_unlock()` proporcionan exclusión mutua
- Dos hilos no pueden mantener el mismo mutex bloqueado al mismo tiempo.
- Si "a" intenta bloquear un mutex mientras que "b" tiene el mismo mutex bloqueado, "a" se va a dormir
- Cuando "b" realice el mutex (`pthread_mutex_unlock()`), "a" será capaz de bloquear el mutex (retorna desde `pthread_mutex_lock()` con el mutex bloqueado)
- Si "c" trata de bloquear el mutex mientras que "a" lo mantiene bloqueado, "c" se queda dormido
- Todos los threads dormidos por `pthread_mutex_lock()` en un mutex previamente bloqueado, son puestos en cola para acceder a dicho mutex.
- `pthread_mutex_lock()` y `pthread_mutex_unlock()` se usan para proteger estructuras de datos. Un solo thread a la vez puede acceder a una cierta estructura de datos bloqueándola y desbloqueándola
- POSIX garantiza bloqueo sin dormir al thread si trata de bloquear mutex no bloqueado

Threads

Bibliografía:

- <http://metalab.unc.edu/pub/Linux/docs/faqs/Threads-FAQ/html/>
- <http://www.math.arizona.edu/swig/pthreads/threads.html>
- http://dis.cs.umass.edu/~wagner/threads_html/tutorial.html
- manual pthread Linux (man -k pthread)
- <http://pauillac.inria.fr/~xleroy/linuxthreads/>
- http://www.amazon.com/exec/obidos/ASIN/0201633922/o/qid=961544788/sr=8-1/ref=aps_sr_b_1_1/002-2882413-1227240
- W. Richard Stevens UNIX Network Programming: Network APIs: Sockets and XTI, Volume 1