PROGRAMACIÓN MULTITHREADING

Sergio Nesmachnow (sergion@fing.edu.uy)
Gerardo Ares (gares@fing.edu.uy)

Escuela de Computación de Alto Rendimiento
(ECAR 2012)

TEMA 2:
PROGRAMACIÓN BÁSICA

PROGRAMACIÓN MULTITHREADING

Escuela de Computación de Alto Rendimiento (ECAR 2012)

CONTENIDO

- **Introducción threads.**
- **Modelos de planificación.**
- **API de programación.**

PROGRAMACIÓN PARALELA

INTRODUCCIÓN THREADS

INTRODUCCIÓN

- En los sistemas tradicionales UNIX los procesos tienen un espacio de direccionamiento virtual y un hilo (*thread*) de ejecución.
- Sobre fines de la década de los 80 surgen los primeros sistemas UNIX con más de un procesador (multiprocesadores).
- Los diseñadores de los sistemas operativos se abocaron al desarrollo de sistemas escalables para las nuevas arquitecturas de multiprocesadores.
- Surgen nuevos núcleos que tienen varios hilos de ejecución y, además, se proveen primitivas para permitir que los procesos también tengan varios hilos de ejecución en el contexto de un proceso.
- Todos los hilos de ejecución comparten el espacio de direccionamiento del proceso.
- No existe ningún tipo de protección de memoria entre los hilos de un mismo proceso.
- La programación con threads debe proveer principalmente mecanismos de sincronización entre los hilos.

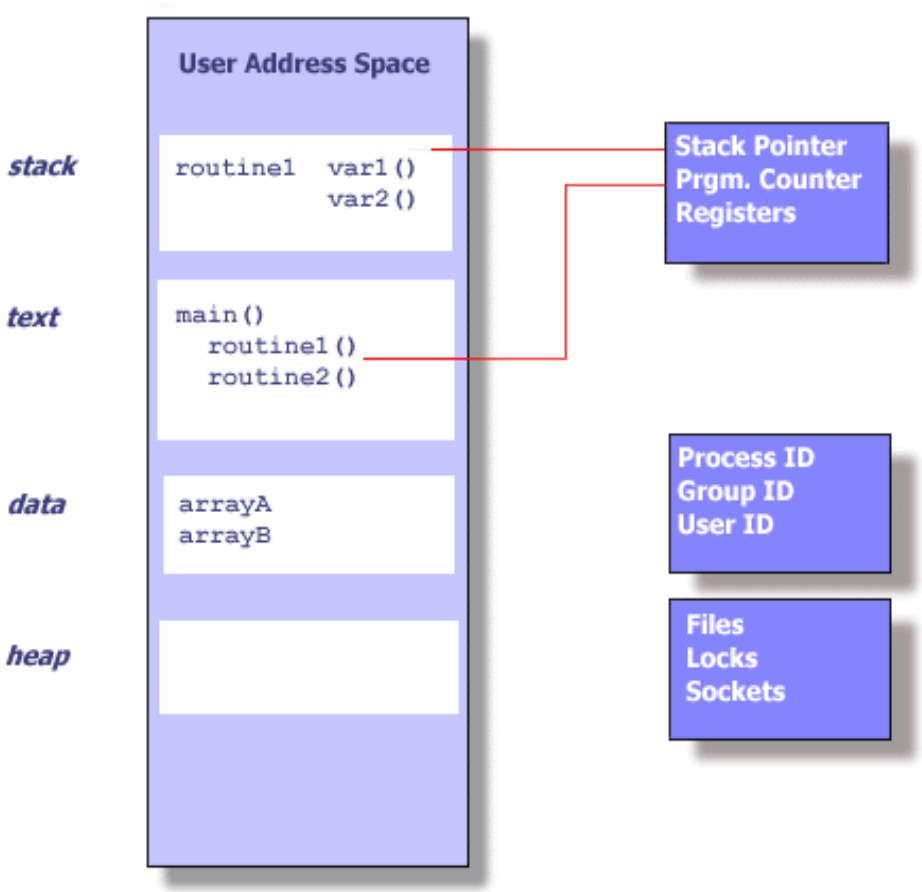
INTRODUCCIÓN

- **Cada hilo contiene información privada:**
  - **Contador de programa (ip : *instruction pointer*).**
  - **Pila (*Stack*).**
  - **Espacio para alojar los registros de la CPU cuando el hilo no está en ejecución.**
  - **Estado del hilo.**
- **Comparten:**
  - **Espacio de direccionamiento.**
  - **Archivos abiertos por el proceso.**
  - **Señales.**
  - **Información de conteo (*accounting*).**

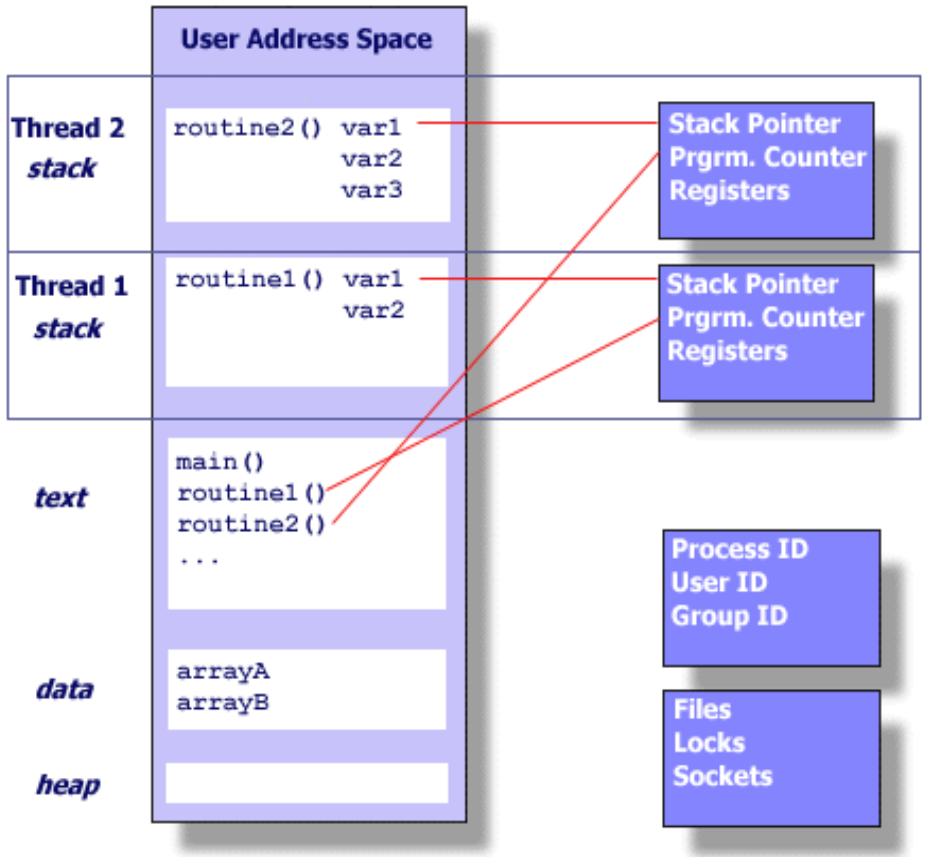
INTRODUCCIÓN

- **Beneficios:**
  - **Mejor aprovechamiento de los recursos de procesadores en ambiente multiprocesadores/multicores.**
  - **Mejor velocidad de comunicación entre hilos de un mismo proceso que entre procesos independientes del sistema.**
  - **Dividir las aplicaciones en módulos funcionales bajo un mismo direccionamiento virtual.**
- **Desventajas:**
  - **Programación más difícil.**
  - **Mayor dificultad de encontrar los errores (debug de la aplicación).**
  - **Limitación en el recurso de memoria.**

INTRODUCCIÓN



UNIX PROCESS



THREADS WITHIN A UNIX PROCESS

PROGRAMACIÓN PARALELA

MODELOS DE PLANIFICACIÓN

THREADS A NIVEL DEL NÚCLEO

- Los hilos pueden ser soportados a nivel del sistema operativo o a nivel de usuario.
- Los dos niveles brindan diferentes formas de ejecución de los procesos multihilados.
- El sistema operativo provee el soporte para la creación, planificación y administración de los hilos.
- El sistema reconoce cada hilo como una unidad de ejecución distinta a ser planificada.
- Beneficios:
  - Mayor nivel de paralelismo en un sistema multiprocesador ya que varios hilos de un proceso pueden estar ejecutando en varios procesadores a la vez.
  - Los hilos son independientes, por lo que al bloquearse un hilo de un proceso (ej.: espera de una entrada/salida o bloqueo por espera de algún evento) los demás hilos del proceso pueden seguir ejecutando.

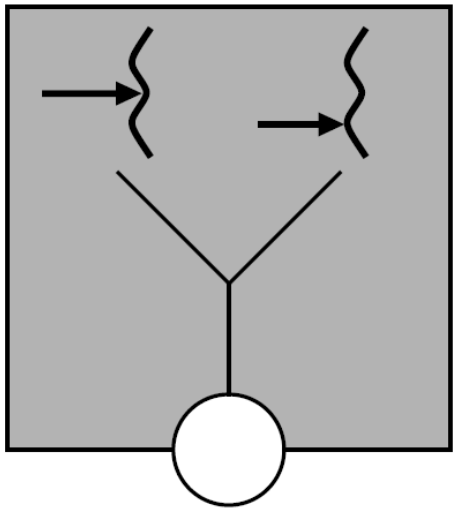
THREADS A NIVEL DE USUARIO

- Son implementados a través de una biblioteca de usuario.
- Es la biblioteca la que provee soporte para la creación, planificación y administración de los hilos.
- El sistema operativo desconoce la existencia de estos hilos, por lo que solamente visualiza una unidad de ejecución.
- Beneficios:
  - El cambio de contexto es menor frente a tener soporte a nivel de núcleo.
  - Permite otro sistema de planificación ya que viene dado en la biblioteca de usuario.
- Los dos niveles brindan diferentes formas de ejecución de los procesos multihilados.

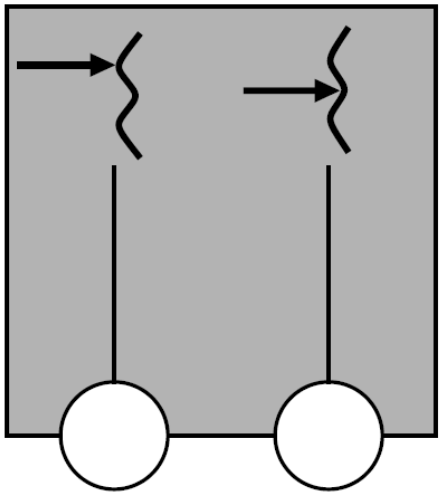
MODELOS DE PLANIFICACIÓN

- **Brindar hilos a nivel de núcleo o de usuario permite definir distintos modelos de planificación:**
  - **Mx1 : Many to One.**
  - **1x1 : One to One.**
  - **MxN : Many to Many.**

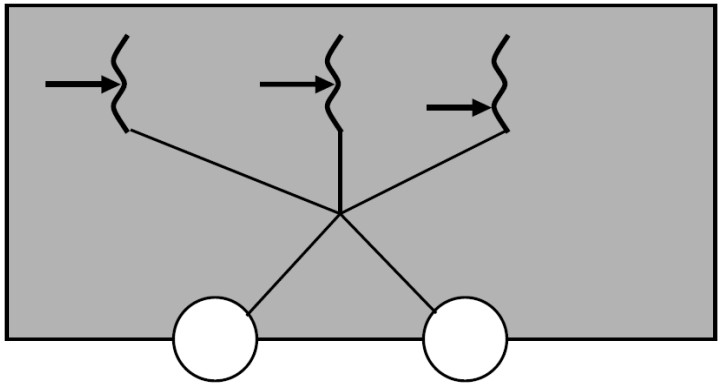
MX1



1x1



MXN

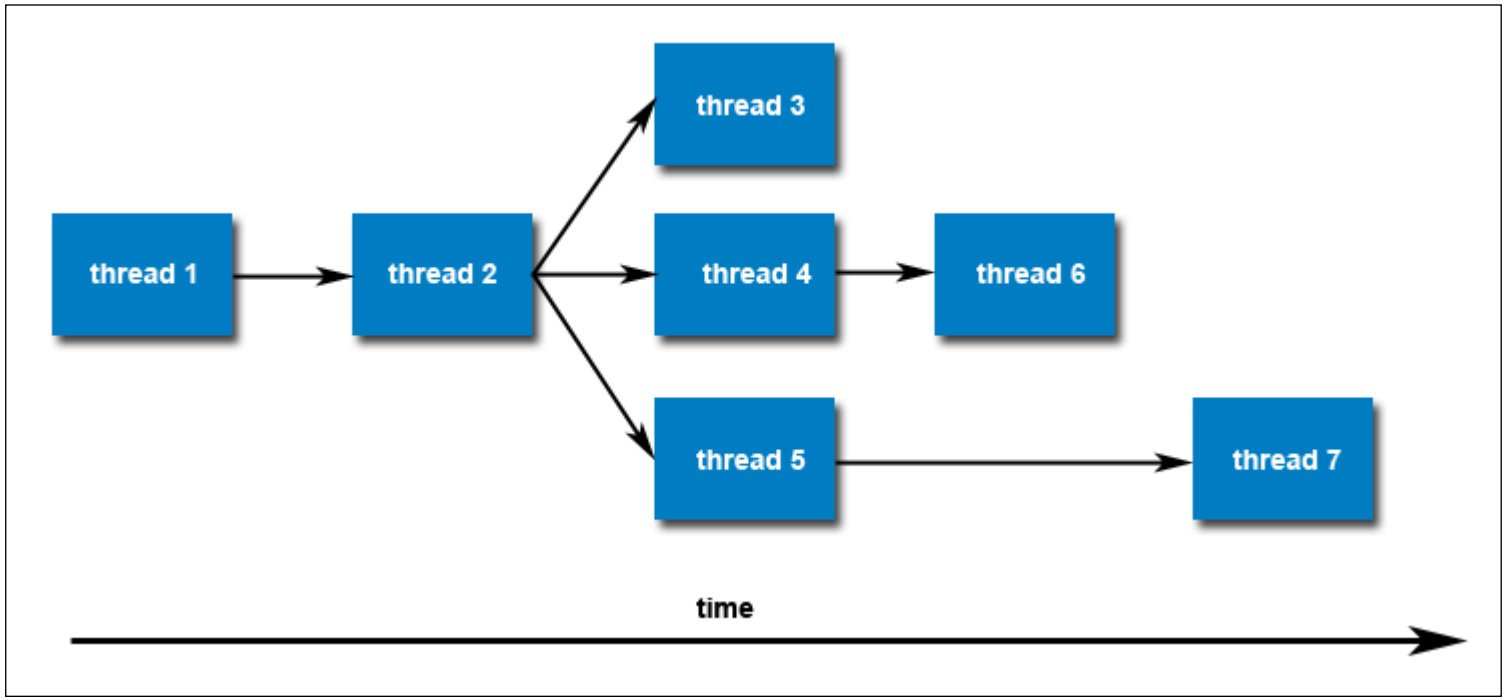


PROGRAMACIÓN PARALELA

API DE PROGRAMACIÓN

CREACIÓN DE THREADS

- Función de creación: `pthread_create`.
- Estas funciones reciben como parámetro un puntero a función que es el código del hilo a ejecutar.
- La función a ejecutar tiene un cabezal predefinido.
- La creación de threads generan, dentro del proceso, una estructura de árbol.



CREACIÓN DE THREADS

- `pthread_create`

- **Entrada:**
  - **pthread_t ***
    Variable que referenciará al thread creado.
  - **const pthread_attr_t ***
    Parámetro de inicialización.
  - **void * (* función_thread) (void *)**
    Puntero a función donde comenzará a ejecutar el thread creado.
  - **void ***
    Parámetro de entrada para el puntero a función.
- **Retorno:**
  - **int**
    Valor de retorno (0 ok, != 0 sino).

CREACIÓN DE THREADS

- El tipo de dato `pthread_t` permite definir threads.
- Generalmente cuando se tienen un conjunto de threads que realizan una misma tarea, se los agrupa en arreglos.

```
pthread_t   thrs[MAX_THREADS];
```

- Esto permite realizar la creación en forma más simple.

```
for (i = 0; i < MAX_THREADS; i++) {
  …
  if (rc = pthread_create(&thrs[i],….))
  …
}
```

CREACIÓN DE THREADS

- Si el parámetro `pthread_attr_t` es nulo, el thread será creado con los valores por defecto:

Atributo	Valor defecto	Resultado
Scope	`PTHREAD_SCOPE_PROCESS`	El thread es configurado como unbounded
Detachstate	`PTHREAD_CREATE_JOINABLE`	El valor de salida del thread es preservado luego que este termina
Stack size	`1 MB`	Tamaño del stack
Priority		El thread hereda la prioridad del padre

CREACIÓN DE THREADS

- **Se puede crear una variable de tipo `pthread_attr_t` para crear un conjunto de thread con alguna característica particular.**

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

pthread_attr_init(&tattr);

pthread_attr_setstacksize((size_t)1024*1024*100);

pthread_create(&thr,&tattr,...);
```

CREACIÓN DE THREADS

- El tamaño del stack es creado al crearse el thread y su comienzo es alineado a un comienzo de página.
- Es importante tener en cuenta que el tamaño del stack es finito y es más reducido que para un proceso con un único hilo.
- Generalmente al final es adicionada una página sin permisos que permite que un proceso no acceda al stack de otro.
- Si el proceso accede genera un señal (SIGSEGV).
- Es importante tener cierto el uso que hará del stack cada thread.

CREACIÓN DE THREADS

- **Prototipo de la función debe ser el siguiente:**

```
void * func_thread (void *)
```

- **El parámetro es de tipo void *, lo que permite pasar cualquier tipo de datos a la función.**

```
struct persona {
  struct fecha     nacimiento;
  int              sexo;
  ...
};
struct persona   p;
  ...
pthread_create (…., func_thread, (void *) p)
```

TERMINACIÓN DE THREADS

- Función de destrucción: `pthread_exit`.
- Típicamente es ejecutada cuando un thread finaliza su ejecución y no tiene nada más para realizar.
- El programador puede asignar un status a la terminación de forma que algún otro threads (a través de la primitiva `pthread_join`) obtenga un valor de la ejecución.
- La función no elimina ningún recurso asignado al proceso que haya sido pedido por el thread.
- Si el thread `main` finaliza con un `pthread_exit`, todos los threads que hayan sido creados permanecerán activos.

TERMINACIÓN DE THREADS

- `pthread_exit`

- **Salida:**
  - **void ***

    Puntero a un tipo dado por el programador.
- **Retorno:**
  - **int**

    Valor de retorno (0 ok, != 0 sino).

FUNCIÓN DE SINCRONIZACIÓN

- La función `pthread_join` permite a un thread esperar por la finalización de otro thread.
- A su vez obtener el status de finalización.
- La función `pthread_join` es bloqueante sobre el thread que la invoca.
- No es posible esperar por varios threads como la función wait de C.

```
void * getNextPicture(void * arg){
 … // Abrir archivo y devolverlo
}
int main() {
  pthread_t      pic_thread;
  ...
  buff = getNexPicture((void *)argv[1]);
  for (i = 2; i < argc; i++) {
    pthread_create(&pic_thread,NULL,getNextPicture, ...);
    display(buff);
    pthread_join(pic_thread,(void **) &buff);
  }
}
```

FUNCIÓN DE SINCRONIZACIÓN

- `pthread_join`

- **Entreada:**
  - **pthread_t ***

    Identificador del thread por cual se espera.
  - **void ***

    Puntero a un tipo dado por el programador.
- **Retorno:**
  - **int**

    Valor de retorno (0 ok, != 0 sino).

EJEMPLO CREACIÓN-TERMINACIÓN

```
void * thread_cantprimos(void * prm) {
// Calcular cant primos
  cantprimo = 1;
  for (i = 2; i <= numero; i++)
    if (esPrimo(i))
      cantprimo++;

  pthread_exit((void*)cantprimo);
}
```

EJEMPLO CREACIÓN-TERMINACIÓN

```
int main (int argc, char *argv[]) {
  int             rc, cant;
  pthread_t       thr;

  if (rc = pthread_create(&thr, NULL, thread_cantprimos, (void
   *)atol(argv[1]))) {
    printf("Error: pthread_create %d\n", rc);
    exit(-1);
  }
  pthread_join(thr,(void *)&cant);
  printf("Cantidad de primos: %d\n", cant);

  return 0;
}
```

SECCIÓN CRÍTICA

- En la programación concurrente las entidades de ejecución generalmente comparten datos/información.
- El acceso a estos datos compartidos por parte de una entidad de ejecución debe ser sincronizada con las otras entidades de forma de garantizar la integridad de los datos.
- Una sección crítica se define como una porción de código donde se accede a un recurso compartido.
- De esa forma, deben existir herramientas que provean el acceso a secciones críticas.
- En la multiprogramación con pthread se brindan mecanismos de sincronización a través de funciones y de variables de sincronización.
- A su vez, se pueden utilizar los mecanismos de programación en C como semáforos.

SINCRONIZACIÓN

- Debido a que los threads comparten el espacio de direccionamiento (memoria), no es necesario proveer mecanismos de comunicación de datos.
- Por otro lado, si es necesario brindar mecanismos de sincronización entre los threads de un proceso.
- La API de programación brinda dos tipos:
  - Función de sincronización: `pthread_join`.
  - Variables de sincronización:
    - Mutex.
    - Variables de Condición.
    - Reader/Writer Locks.

VARIABLES DE SINCRONIZACIÓN

- **Sincronización MUTEX:**
  - **Logra la mutuo exclusión para el acceso a secciones críticas entre los distintos hilos de un proceso.**
  - **Esto es permite que un único thread pueda ejecutar en la sección crítica.**
  - **Cuando un thread está en una sección crítica, los demás threads que quieran acceder serán bloqueados.**
  - **Una vez que el thread finaliza su ejecución sobre la sección crítica se despertará a otro thread que esté bloqueado para que pueda entrar en la sección crítica.**
  - **Un código para entrar en una sección crítica a través de mutex será el siguiente:**

```
Sincronizar_entrada_sección_crítica(…);
Sección crítica...
Sincronizar_salida_sección_crítica(…);
```

VARIABLES DE SINCRONIZACIÓN

- El tipo de dato `pthread_mutex_t` permite definir variables de sincronización de mutuo exclusión.
- Las variables de inicialización deben ser inicializadas y destruidas a través de las funciones:

```
pthread_mutex_init(pthread_mutex_t *, pthread_mutexattr_t *)
pthread_mutex_destroy(pthread_mutex_t *)
```

- El tipo de dato `pthread_mutexattr_t` permite inicializar el mutex con diferentes atributos.
- Antes que nada debe ser inicializado a través de la función pthread_mutexattr_init().

VARIABLES DE SINCRONIZACIÓN

- **Los atributos para inicializar permiten determinar el comportamiento de la planificación de los thread que operan sobre ellos.**
- **Algunos atributos:**
  - **Scope: Privado o compartido. Permite compartir un mutex con otros procesos a través de memoria compartida.**

    ```
    pthread_mutexattr_setpshared(pthread_mutexattr_t *mattr, int pshared);
    ```

  - **Tipo: Normal, Error Check, Recursive, o default. Comportamiento frente a operaciones de lock/unlock recursivas.**

    ```
    pthread_mutexattr_settype(pthread_mutexattr_t *mattr, int type);
    ```

  - **Protocolo: None, Inherit, protect. Permite heredar la prioridad de otro thread en caso que un thread de mayor prioridad quede bloqueado por este. Permite eliminar inversiín de prioridades.**

    ```
    pthread_mutexattr_setprotocol(pthread_mutexattr_t *mattr, int protocol);
    ```

  - **Robust: Robust, Stalled. Determina el comportamiento de como se comporta un mutex si un thread que lo tenia tomado muere. Los threads quedan bloqueados (robust) o el mutex es liberado no bien el thread muere (stalled, default).**

    ```
    pthread_mutexattr_setrobust_np(pthread_mutexattr_t *mattr, int robust);
    ```

VARIABLES DE SINCRONIZACIÓN

```
pthread_mutexattr_t tmattr;
pthread_mutex_t mutex;

pthread_mutexattr_init(&tmattr);

pthread_mutexattr_setprotocol(&tmattr,PTHREAD_PRIO_INHERIT);

pthread_mutex_init(&mutex,&tmattr);

pthread_mutex_destroy(&mutex);
```

- **El mutex queda inicializado en estado “liberado” para que sea obtenido por un thread.**
- **Un comentario importante es que un thread que obtiene un lock no necesariamente tiene que ser el mismo thread el que lo libera.**

VARIABLES DE SINCRONIZACIÓN

- **Para obtener un lock se tiene una operación bloqueante y otra no bloqueante:**
  - **Funciones bloqueantes:**

```
pthread_mutex_lock(pthread_mutex_t *);
```

  - **Funciones no bloqueantes (spin locks):**

```
pthread_trymutex_lock(pthread_mutex_t *);
```

VARIABLES DE SINCRONIZACIÓN

- **Para obtener un lock se puede utilizar una operación bloqueante u otra no bloqueante:**
  - **Funciones bloqueantes: Los threads que ejecuten esta operación sobre un lock mientras este esta siendo usado por otro serán bloqueados.**

```
pthread_mutex_lock(pthread_mutex_t *);
```

  - **Funciones no bloqueantes (spin locks): El thread que ejecutra esta operación no quedará bloqueado en caso que el lock esté siendo usado por otro thread.**

```
pthread_mutex_trylock(pthread_mutex_t *);
```

VARIABLES DE SINCRONIZACIÓN

```
thr_mutex_t count_mutex;int count;
void increment_count(){
       pthread_mutex_lock(&count_mutex);
       count= count + 1;
       pthread_mutex_unlock(&count_mutex);
}
void increment_count() // Con Spin lock {
  try = 0;
  while(tray < MAX_TRYS && pthreadthr_mutex_trylock(&count_mutex))
     try++;
  if (try == MAX_TRAYS)
    pthread_mutex_lock(&count_mutex);
  count= count + 1;
  pthread_mutex_unlock(&count_mutex);
}
void decrement_count() {
       thr_mutex_lock(&count_mutex);
       count= count - 1;
       thr_mutex_unlock(&count_mutex);
}
```

VARIABLES DE SINCRONIZACIÓN

- Cuando usar operaciones bloqueantes o no bloqueantes?
- Antes de responder debemos recordar que bloquear un thread implica entrar en modo kernel, bloquear el thread en una correspondiente cola, ejecutar el planificador del sistema, cambiar el contexto al nuevo thread (que al poder se de otro proceso se puede invalidar toda la TLB, y, luego, en el momento de volver cargar el contexto del proceso nuevamente.
- La respuesta entonces dependerá de como fue estructurada la programación en cuanto a la sección crítica.
- Para secciones críticas largas es mejor utilizar operaciones bloqueantes.
- Para secciones críticas cortas es mejor utilizar operaciones no bloqueantes.
- A su vez, en un monoprocesador no tiene sentido ejecutar operaciones no bloqueantes… por qué?

VARIABLES DE SINCRONIZACIÓN

- Un ejemplo clásico de la programación concurrente son los lectores y escritores.
- Un conjunto de thread realizan operaciones de lectura en su sección crítica, mientras otro conjunto realiza operaciones de escritura.
- Si la cantidad de escrituras es bastante menor a que las lecturas este problema se resuelve a través de la utilización de dos variables de sincronización.

```
Lectores
Mutex_lock(mtxLec)
 lectores++
 if (lectores == 1)
    Mutex_lock(mtxEscr)
Mutex_unlock(mtxLec)
Sección crítica
Mutes_lock(mtxLec);
  lectores--
  if (lectores == 0)
    Mutex_unlock(mtxEscr)
Mutex_unlock(mtxLec)
```

```
Escritores

Mutex_lock(mtxEscr)
Sección crítica
Mutex_unlock(mtxEscr)
```

VARIABLES DE CONDICIÓN

- Las variables de condición permiten bloquear a un proceso hasta que cierta condición sea verdadera.
- Las variables de condición deben ser usadas siempre con un mutex.
- La condición es controlada bajo la protección del mutex asociado.
- Cuando la condición es falsa el thread se bloquea y libera el lock obtenido para controlar la condición.
- Cuando otro thread cambia algún componente de la condición, este puede despertar a uno o varios threads que estén aguardando por esa condición.
- Los threads serán despertados , obtendrán el lock y evaluarán nuevamente la condición para determinar sus próximas acciones.
- La politica de scheduling determinará como los thread serán despertados.
- Si la política es SCHED_OTHER los thread serán despertados según el orden de prioridad.

VARIABLES DE CONDICIÓN

- **El tipo de dato `pthread_cond_t` permite definir variables de condición.**
- **Las variables de inicialización deben ser inicializadas y destruidas a través de las funciones:**

```
pthread_cond_init(pthread_cond_t *, pthread_condattr_t *)
pthread_cond_destroy(pthread_cond_t *)
```

- **El tipo de dato `pthread_condattr_t` permite inicializar la variable con el atributo de Scope:**
  - **Scope: Privado o compartido. Permite compartir un mutex con otros procesos a través de memoria compartida.**

```
pthread_condattr_setpshared(pthread_condattr_t *mattr, int pshared);
```

- **Es necesario primero inicializar la variable a través de la función `pthread_condattr_init()`.**

VARIABLES DE CONDICIÓN

- La función que bloquea al proceso para espera es:

```
pthread_cond_wait(pthread_cond_t * cv, pthread_mutex_t * mtx);
```

- El thread queda bloqueado a la espera que otro thread ejecute una operación de signal sobre la variable de condición.
- El mutex asociado es liberado al invocar esta función de forma de permitir a otros threads obtener el mutex.

VARIABLES DE CONDICIÓN

- **Para despertar a un thread que espera por un cambio en la condición se utiliza la función:**

```
pthread_cond_signal(pthread_cond_t * cv);
```

- **Al invocarse un thread es depertado y queda en espera para la obtención del mutex asociado.**
- **Esta operación no libera el mutex, eso se debe hacer explicitamente.**
- **Para despertar a todos los procesos que esperan por el mutex se utiliza la función:**

```
pthread_cond_broadcast(pthread_cond_t * cv);
```

- **Todos los threads son despertados y puestos en espera para la obtención del mutex asociado.**

VARIABLES DE CONDICIÓN

- **Sincronización: Variables de condición.**
  - **Cierta condición viene dada por un dato compartido.**
  - **Según el valor del dato compartido se desea continuar o esperar a que cambie.**
- **Ejemplo:**

  Sea `x` un dato compartido.

  Si `x > 10` entonces continuo, sino espero a que `x` sea mayor que `10`.
  - **Solución: Variables de condición.**
  - **Las variables de condición están necesariamente asociadas a una variable de mutuo exclusión.**
  - **Evita el busy waiting.**

VARIABLES DE CONDICIÓN

- **Sincronización: Variables de condición.**

```
Thread 1

pthread_mutex_t   m;
pthread_cond_t    c;
…
pthread_mutex_lock(&m);
while (!condición(dato_compartido))
   pthread_cond_wait(&c,&m);
trabajar();
pthread_mutex_unlock();
```

```
Thread 2

...
pthread_mutex_lock(&m);
Operar(dato_compartido);
pthread_cond_signal(&c);
pthread_mutex_unlock();
...
```

- **Al invocar a `pthread_cond_wait` se libera la variable de mutuo exclusión asociada.**
- **Cuando un thread es despertado por un signal cuando se vuelve a ejecutar se obtiene la variable de mutuo exclusión.**
- **`pthread_cond_broadcast` despierta a todos los threads que están en una variable de condición.**

VARIABLES DE CONDICIÓN

- **Ejemplo de signal broadcast:**

```
get_resources(int amount) {
  pthread_mutex_lock(&rsrc_lock);
  while (resources < amount) {
    pthread_cond_wait(&rsrc_add, &rsrc_lock);
  }
  resources -= amount;
  pthread_mutex_unlock(&rsrc_lock);
}
add_resources(int amount) {
  pthread_mutex_lock(&rsrc_lock);
  resources += amount;
  pthread_cond_broadcast(&rsrc_add);
  pthread_mutex_unlock(&rsrc_lock);
}
```

VARIABLES DE SINCRONIZACIÓN

- **Volviendo al caso de lectores-escritores se quiere dar prioridad a los escritores.**
- **La estrategia es que cuando llegue un escritor "avise" y, de esa forma, los próximos lectores se bloquearan antes de intentar acceder.**

```
Lectores
Mutex_lock(mtxEscrEspera)
While (Escritores != 0)
   Esperar(cond,mtxEscrEspera)
Mutex_lock(mtxLec)
 lectores++
 if (lectores == 1)
    Mutex_lock(mtxEscr)
Mutex_unlock(mtxLec)
Mutex_unlock(mtxEscrEspera)
Sección crítica
Mutes_lock(mtxLec);
  lectores--
  if (lectores == 0)
    Mutex_unlock(mtxEscr)
Mutex_unlock(mtxLec)
```

```
Escritores
Mutex_lock(mtxEscrEspera)
Escritores++
Mutex_unlock(mtxEscrEspera)

Mutex_lock(mtxEscr)
Sección crítica
Mutex_unlock(mtxEscr)

Mutex_lock(mtxEscrEspera)
Escritores--
Signal_broadcast(cond)
Mutex_unlock(mtxEscrEspera)
```

READ-WRITE LOCKS

- El tipo de dato `pthread_rwlock_t` permite definir variables de sincronización para lectores/escritores.
- Las variables de inicialización deben ser inicializadas y destruidas a través de las funciones:

```
pthread_rwlock_init(pthread_rwlock_t *, pthread_rwlockattr_t *)
pthread_rwlock_destroy(pthread_rwlock_t *)
```

- El tipo de dato `pthread_rwlockattr_t` permite inicializar la variable con el atributo de Scope:
  - Scope: Privado o compartido. Permite compartir un mutex con otros procesos a través de memoria compartida.

```
pthread_rwlockattr_setpshared(pthread_rwlockattr_t *mattr, int pshared);
```

- Es necesario primero inicializar la variable a través de la función `pthread_rwlockattr_init()`.

READ-WRITE LOCKS

- El acceso a la sección crítica por parte de los lectores puede ser realizada con la operación bloqueante:

```
pthread_rwlock_rdlock (pthread_rwlock_t * rw)
```

- O con la operación no bloqueante:

```
pthread_rwlock_tryrdlock (pthread_rwlock_t * rw)
```

- El acceso por parte de los escritores es a través de la siguiente operación bloqueante:

```
pthread_rwlock_wrlock (pthread_rwlock_t * rw)
```

- O con la operación no bloqueante:

```
pthread_rwlock_trywrlock (pthread_rwlock_t * rw)
```

READ-WRITE LOCKS

- La liberación del locks para los escritores y lectores es a través de la misma función:

```
pthread_rwlock_unlock(pthread_rwlock_t *)
```

- POSIX no define claramente si los escritores o lectores tienen prioridad por lo que se tendrá que ver en la implementación utilizada.

VARIABLES DE SINCRONIZACIÓN

- **Ejercicio:**

Implementar los lectores-escritores con rwlocks.

FUNCIÓN DE SINCRONIZACIÓN

- Pthread brinda la función de sincronización barrier que permite sincronizar varios threads en un punto.
- Las barreras se representan a través del tipo de datos:

```
pthread_barrier_t
```

- En la inicialización se debe especificar la cantidad de threads que se necesitarán para lograr la sincronización:

```
pthread_barrier_init(pthread_barrier_t *, pthread_barrierattr_t,
unsigned count)
```

- Para sincronizarse los threads deben ejecutar la primitiva:

```
pthread_barrier_wait(pthread_barrier_t *)
```

- Una vez que todos los threads llegan a ese punto se continua la ejecución.