

# SISTEMAS PARALELOS

---

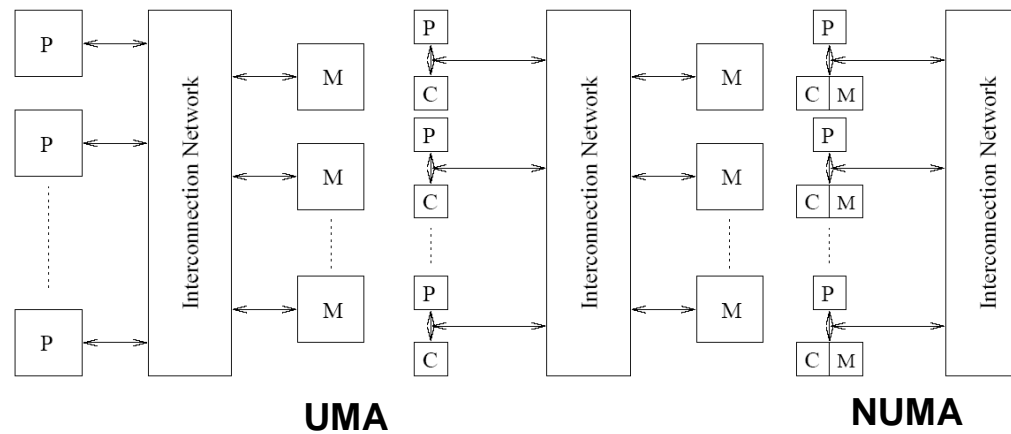
Clase 3 – Programación en memoria compartida // Pthreads

# PROGRAMACIÓN EN MEMORIA COMPARTIDA

---

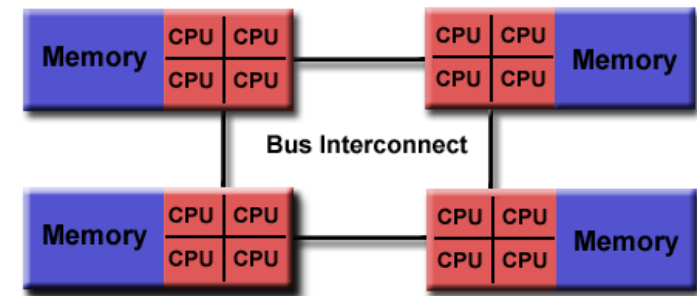
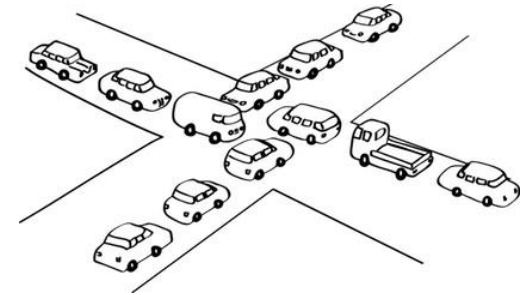
# Plataformas de memoria compartida

- Los procesadores se comunican leyendo y escribiendo variables en un espacio de datos común (memoria compartida)
- Los módulos de memoria pueden ser locales (exclusivos a un procesador) o globales (comunes a todos los procesadores)
- Sub-clasificación por modo de acceso a memoria
  - Acceso uniforme a memoria (UMA)
  - Acceso no uniforme a memoria (NUMA)
- Se necesita un mecanismo de coherencia de caché
- Modelo de programación asociado: memoria compartida. Pasaje de mensajes también es una posibilidad.



# Modelo de memoria compartida

- El problema de la sincronización en memoria compartida (con sus efectos indeseables como los deadlocks) es responsabilidad del programador, utilizando las herramientas que provea el lenguaje.
- Toda sincronización disminuye la eficiencia.
- La localidad de los datos será muy importante en el rendimiento (en particular en arquitecturas NUMA con memoria compartida).
- En algunos lenguajes el programador podrá actuar sobre la localidad de los datos, en otros tendrá que re-estructurar el código.



# Modelo de memoria compartida

- El programador en general no maneja la distribución de los datos ni lo relacionado a la comunicación de los mismos.
- Ventaja: Transparencia para el programador. La ubicación de los datos, su replicación y su migración son transparentes.
- Desventaja: A veces es necesario trabajar sobre esos aspectos para mejorar el rendimiento. Además, resulta difícil la predicción de performance a partir de la lectura del algoritmo.



# Modelo de memoria compartida

Los modelos de programación proveen un soporte para expresar la concurrencia y sincronización:

- Los modelos basados en procesos suponen datos locales (privados) de cada proceso.
- Los modelos basados en threads o procesos “livianos” suponen que toda la memoria es global (*Pthreads*).
- Los modelos basados en directivas extienden el modelo basado en threads para facilitar su creación y sincronización (*OpenMP*).

# Fundamentos del modelo de hilos

- Un thread es un único hilo de control en el flujo de un programa. Por ejemplo (multiplicación de matrices):

```
for (row = 0; row < n; row++)  
    for (col = 0; col < n; col++)  
        c[row][column] = dot_product( get_row(a, row), get_col(b, col));
```

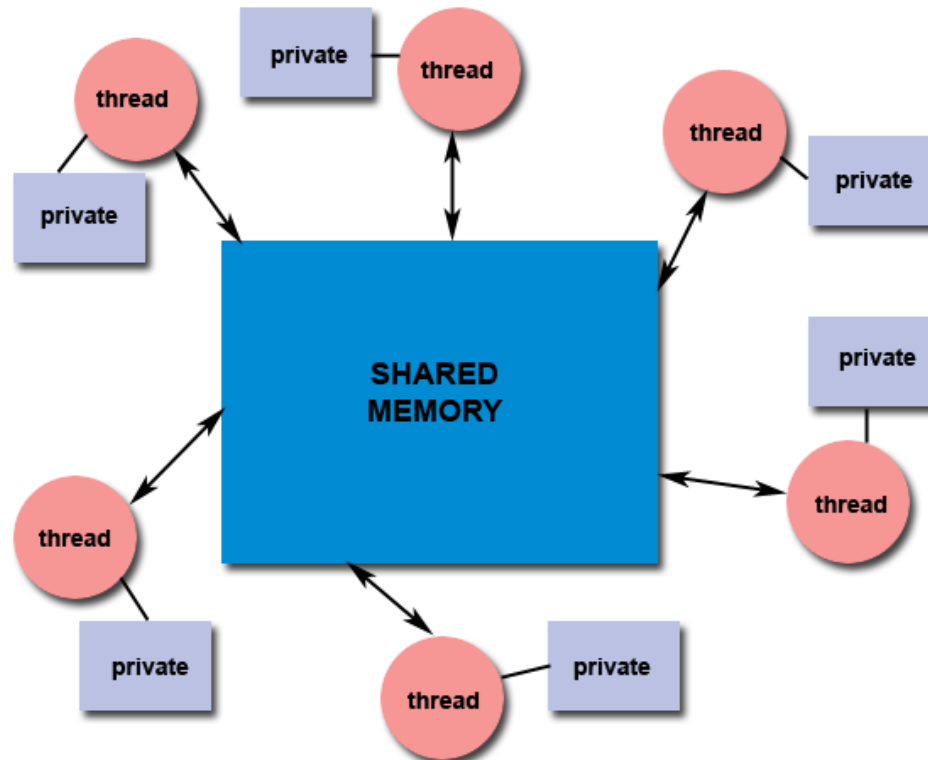
- Puede transformarse en:

```
for (row = 0; row < n; row++)  
    for (col = 0; col < n; col++)  
        c[row][column] = create_thread(dot_product(get_row(a,row),get_col(b,col)));
```

- En este caso el *Thread* funciona como una instancia de una función que retorna antes que la función se haya terminado de ejecutar.

# Fundamentos del modelo de hilos

- Todos los hilos tienen acceso a una memoria compartida global.
- Los hilos a su vez tienen su propio espacio de memoria privada.





# Ventajas del modelo de hilos

- **Portabilidad:** permite migrar aplicaciones entre arquitecturas. Útil para desarrollo.
- **Planificación y balance de carga:** soporta mapeo dinámico a nivel de sistema que minimiza el overhead por ociosidad (lo hacen las APIs de threads). Facilita la distribución de trabajo ante cargas irregulares.
- **Soporte para el manejo de la latencia de memoria:** oculta la latencia por accesos a memoria al permitir multihilado.
- **Facilidad de programación y uso extendido:** más fácil de programar que Pasaje de Mensajes (no requiere el manejo de la comunicación de datos).

# POSIX THREADS

---

# POSIX Threads

- Hasta mediados de los años 90, existían numerosas APIs para el manejo de hilos (incompatibles entre ellas).
- En 1995, IEEE especifica el estándar POSIX Threads (normalmente llamado Pthreads). Básicamente, un conjunto de tipos de datos y funciones para el lenguaje de programación C.
- POSIX se ha establecido como una API estándar para manejo de Threads, provista por la mayoría de los desarrolladores de sistemas operativos.
- Los conceptos que se discutirán son independientes de la API y pueden ser igualmente válidos para utilizar Java Threads, Windows Threads, Solaris Threads, etc.

# Pthreads

- Las rutinas más utilizadas de Pthreads se pueden dividir en 3 grupos:
  - **Manejo de threads:** Creación, terminación, join, asignación y recuperación de atributos, entre otros.
  - **Mutexes:** mecanismos para exclusión mutua.
  - **Variables condición:** mecanismos para sincronización por condición.

# Pthreads – Creación de hilos

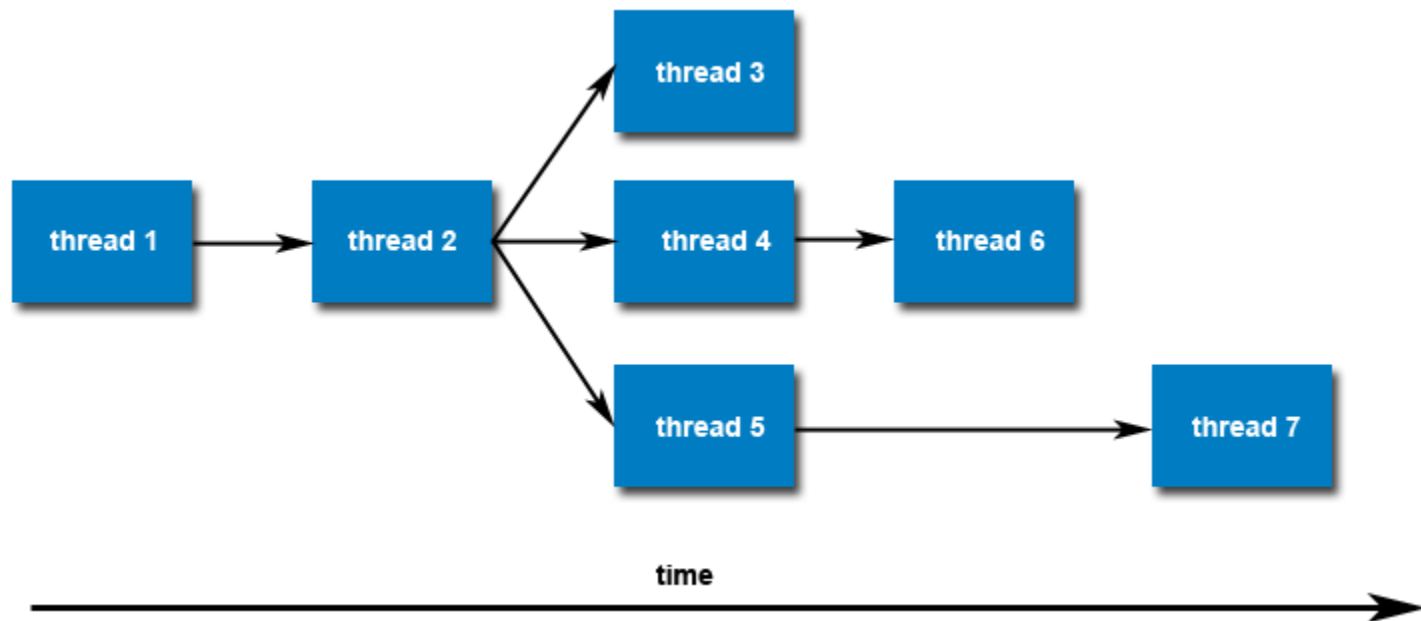
- Inicialmente hay un sólo hilo de ejecución (*hilo main*).
- Todos los demás hilos deben ser creados explícitamente por el programador.
- La función *pthread\_create* crea un hilo y lo pone en ejecución:

```
int pthread_create ( pthread_t    *thread_handle,  
                    const pthread_attr_t *attribute,  
                    void * (*thread_function)(void *),  
                    void *arg);
```

- *thread\_handle* es la dirección de un objeto *pthread\_t*, el cual representa al hilo.
- *attribute* es la dirección de un objeto *pthread\_attr*. NULL para valores por defecto.
- *thread\_function* es la función que contiene el código que ejecutará el hilo creado
- *arg* es el único argumento que se le puede pasar directamente al hilo creado. Debe ser de tipo (void \*).

# Pthreads – Creación de hilos

- Una vez creados, los hilos son pares y pueden crear otros hilos.
- No hay jerarquías o dependencias predefinidas entre los hilos.



# Pthreads – Terminación de hilos

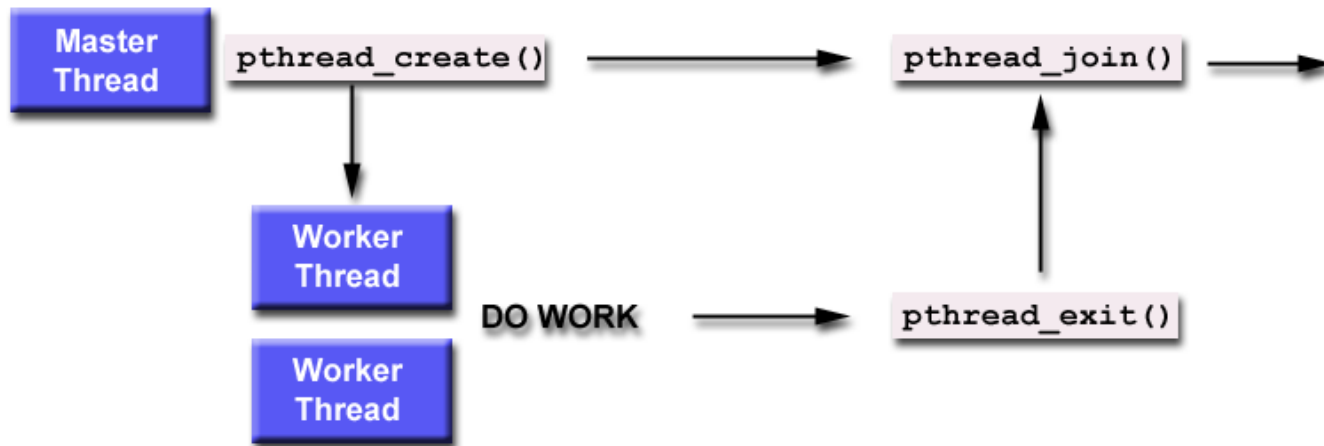
- Para terminar su ejecución los hilos deben invocar a la función *pthread\_exit*:

```
int pthread_exit (void *res);
```

- Esta función finaliza la ejecución del hilo y retorna un valor que puede ser posteriormente leído por otro hilo (en general el hilo que lo creó).

# Pthreads – *Join* de hilos

- El hilo que invoca a la función *pthread\_create* continúa con su ejecución luego del llamado → se requiere sincronización para evitar que el programa termine de forma incorrecta.
- Para ello se emplea la función *pthread\_join*. Esta función bloquea al hilo llamador hasta que el hilo especificado como argumento termine su ejecución.



• — Secuencial — • — Concurrente/Paralela — • — Secuencial — •

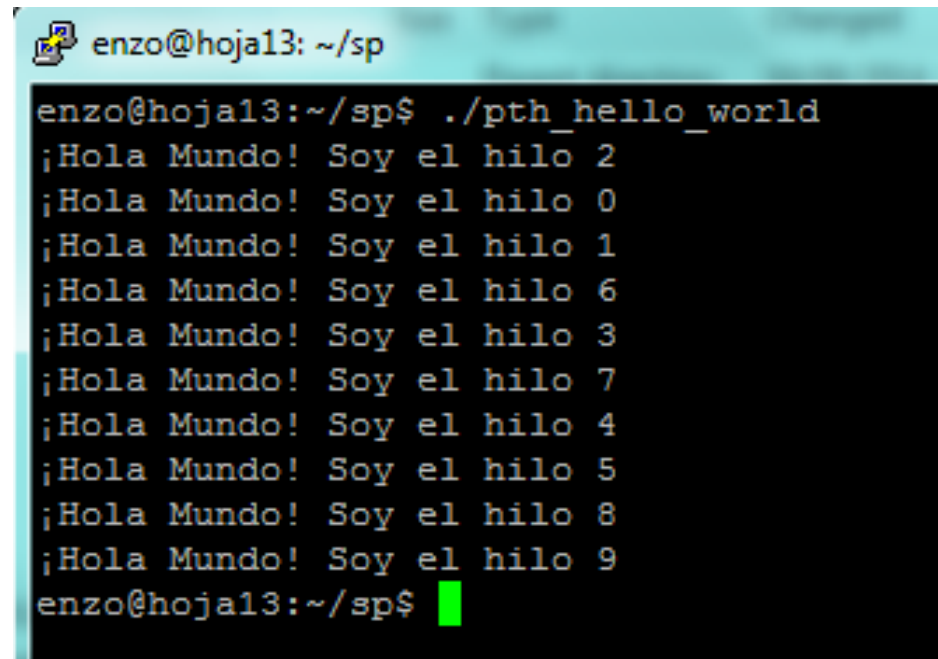


# Pthreads – ¡Hola Mundo!

```
1  #include <stdio.h>
2  #include <pthread.h>
3
4  #define NUM_THREADS 10
5
6  void * hello_world (void * ptr);
7
8  int main() {
9      int i, ids[NUM_THREADS];
10     pthread_attr_t attr;
11     pthread_t threads[NUM_THREADS];
12
13     pthread_attr_init(&attr);
14
15     /* Crea los hilos */
16     for (i = 0; i < NUM_THREADS; i++) {
17         ids[i] = i;
18         pthread_create(&threads[i], &attr, hello_world, &ids[i]);
19     }
20
21     /* Espera a que los hilos terminen */
22     for (i = 0; i < NUM_THREADS; i++)
23         pthread_join(threads[i], NULL);
24
25     return 0;
26 }
```

# Pthreads – ¡Hola Mundo!

```
void * hello_world (void * ptr) {  
  
    int * p, id;  
    p = (int *) ptr;  
    id = *p;  
  
    printf("\n¡Hola Mundo! Soy el hilo %d",id);  
  
    pthread_exit(0);  
}
```

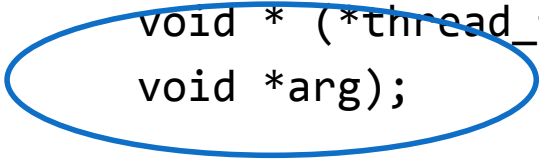


```
enzo@hoja13: ~/sp  
enzo@hoja13:~/sp$ ./pth_hello_world  
¡Hola Mundo! Soy el hilo 2  
¡Hola Mundo! Soy el hilo 0  
¡Hola Mundo! Soy el hilo 1  
¡Hola Mundo! Soy el hilo 6  
¡Hola Mundo! Soy el hilo 3  
¡Hola Mundo! Soy el hilo 7  
¡Hola Mundo! Soy el hilo 4  
¡Hola Mundo! Soy el hilo 5  
¡Hola Mundo! Soy el hilo 8  
¡Hola Mundo! Soy el hilo 9  
enzo@hoja13:~/sp$
```

# Pthreads – Pasaje de parámetros a los hilos

- *pthread\_create* permite pasar un único parámetro a cada hilo:

```
int pthread_create ( pthread_t   *thread_handle,  
                    const pthread_attr_t *attribute,  
                    void * (*thread_function)(void *),  
                    void *arg);
```

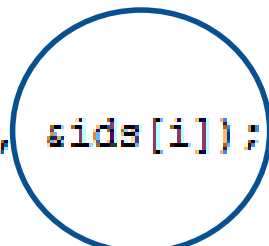


- Analicemos el pasaje de argumentos en el ejemplo anterior...

# Pthreads – Pasaje de parámetros a los hilos

- En el llamado a *pthread\_create*:

```
/* Crea los hilos */  
for (i = 0; i < NUM_THREADS; i++) {  
    ids[i] = i;  
    pthread_create(&threads[i], &attr, hello_world, &ids[i]);  
}
```



- En la función *hello\_world* se debe «castear» al tipo adecuado:

```
void * hello_world (void * ptr) {  
  
    int * p, id;  
    p = (int *) ptr;  
    id = *p;  
}
```

¿Por qué usar un arreglo auxiliar en el llamado a *pthread\_create*?

# Pthreads – Pasaje de parámetros a los hilos

- En el caso en que haya que pasar múltiples parámetros a cada hilo, hay al menos 2 posibilidades:
  - Pasarle un *struct* a cada hilo que contenga todos los argumentos que necesita
  - Mantener uno o más arreglos globales y pasarle el ID a cada hilo para que sepa a qué posición debe acceder

# Pthreads – Primitivas para exclusión mutua

- Comunicación implícita → se pone el esfuerzo en sincronizar tareas concurrentes.
- Cuando múltiples hilos tratan de manejar los mismos datos, el resultado puede ser incoherente si no se sincroniza adecuadamente:

```
if (mi_costo < mejor_costo)
    mejor_costo = mi_costo;
```

- Si tenemos 2 hilos y el valor inicial de *mejor\_costo* (memoria compartida) es 100, y cada hilo tiene su *mi\_costo* en 50 y 75, el valor a guardar en memoria global podría ser cualquiera de los dos.
- Esto depende del scheduling de los hilos → *Race condition*

# Pthreads – Primitivas para exclusión mutua

- El código anterior funcionaría correctamente si fuese una sentencia atómica → corresponde a una sección crítica.
- Las secciones críticas se implementan en Pthreads utilizando *mutex\_locks* (bloqueo por exclusión mutua).
- *mutex\_locks* tienen dos estados: *locked* (bloqueado) y *unlocked* (desbloqueado). En cualquier instante, sólo un hilo puede bloquear un *mutex\_lock* (*lock* es una operación atómica).
- Para entrar en la sección crítica un hilo debe lograr tener control del *mutex\_lock* (bloquearlo).
- Cuando un hilo sale de la sección crítica debe desbloquear el *mutex\_lock*.
- Todos los *mutex\_lock* deben inicializarse como desbloqueados.

# Pthreads – Primitivas para exclusión mutua

- Pthreads provee las siguientes funciones para manejar los mutex-locks:

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

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

[illegible]



# Pthreads – Primitivas para exclusión mutua

Ahora se puede escribir el código para calcular el mínimo de una lista de números:

```
pthread_mutex_t    minimum_value_lock;
...
int main(){
    ...
    pthread_mutex_init(&minimum_value_lock, NULL);
    ...
    /* Create y join de threads */
    ...
}

void *find_min(void *list_ptr){
    ...
    pthread_mutex_lock(&minimum_value_lock);
    if (my_min < minimum_value) minimum_value = my_min;
    pthread_mutex_unlock(&minimum_value_lock);
    ...
}
```

# Pthreads – Primitivas para exclusión mutua:

## Problema de productores y consumidores

- Ejemplo: el escenario de productores-consumidores impone las siguientes restricciones:
  - Un hilo productor no debe sobrescribir el buffer compartido cuando el elemento anterior no ha sido consumido por un hilo consumidor.
  - Un hilo consumidor no puede tomar nada de la estructura compartida hasta no estar seguro de que se ha producido algo anteriormente.
  - Los consumidores deben excluirse entre sí.
  - Los productores deben excluirse entre sí.
  - En este ejemplo el buffer es de tamaño 1.

# Pthreads – Primitivas para exclusión mutua:

## Problema de productores y consumidores

*Main* de la solución al problema de productores-consumidores.

```
pthread_mutex_t    task_queue_lock;
int task_available;
...
int main() {
    task_available = 0;
    ...
    pthread_mutex_init(&task_queue_lock, NULL);
    /* Create y join de threads productores y consumidores*/
    ...
}
```

# Pthreads – Primitivas para exclusión mutua:

## Problema de productores y consumidores

Código para los productores

```
void *producer(void *producer_thread_data){
    /* local data structure declarations */
    int inserted;
    struct task my_task;
    while (!done()){
        inserted = 0;
        create_task(&my_task);
        while (inserted == 0) {
            pthread_mutex_lock(&task_queue_lock);
            if (task_available == 0) {
                insert_into_queue(my_task);
                task_available = 1;
                inserted = 1;
            }
            pthread_mutex_unlock(&task_queue_lock);
        }
    }
}
```

# Pthreads – Primitivas para exclusión mutua:

## Problema de productores y consumidores

Código para los consumidores

```
void *consumer(void *consumer_thread_data) {  
    /* local data structure declarations */  
    int extracted;  
    struct task my_task;  
    while (!done()){  
        extracted = 0;  
        while (extracted == 0){  
            pthread_mutex_lock(&task_queue_lock);  
            if (task_available == 1) {  
                extract_from_queue(&my_task);  
                task_available = 0;  
                extracted = 1;  
            }  
            pthread_mutex_unlock(&task_queue_lock);  
        }  
        process_task(my_task);  
    }  
}
```

# Pthreads – Primitivas para exclusión mutua:

## Tipos de exclusión

- Pthreads soporta tres tipos de *locks*: *Normal*, *Recursive* y *Error Check*
  - Un mutex con el atributo *Normal* NO permite que un hilo que lo tienen bloqueado vuelva a hacer un lock sobre él (*deadlock*).
  - Un mutex con el atributo *Recursive* SI permite que un hilo que lo tienen bloqueado vuelva a hacer un lock sobre él (simplemente incrementa una cuenta de control).
  - Un mutex con el atributo *Error Check* responde con un reporte de error al intento de un segundo bloqueo por el mismo hilo.
- El tipo de Mutex puede setearse entre los atributos antes de su inicialización.

# Pthreads – Primitivas para exclusión mutua:

## Overhead por el uso de locks

- Los locks representan puntos de serialización → si dentro de las secciones críticas ponemos segmentos largos de programa tendremos una degradación importante del rendimiento.
- A menudo se puede reducir el overhead por espera ociosa, utilizando la función *pthread\_mutex\_trylock*, la cual retorna el control informando si pudo hacer o no el lock:

```
int pthread_mutex_trylock (pthread_mutex_t *mutex_lock)
```

- Evita tiempos ociosos.
- Menos costoso por no tener que manejar las colas de espera.
- ¿Cuándo usarlo?

# Pthreads – Primitivas para sincronización por condición

- Los *locks* representan un mecanismo útil para sincronizar hilos. Sin embargo, un uso indiscriminado de los mismos puede provocar un overhead inaceptable.
  - Por ejemplo, cuando un hilo debe esperar a que ocurra una determinada condición para continuar con su trabajo → el uso de locks para esta situación implica realizar *busy waiting*
- Una solución posible a este problema consiste en emplear *variables condición*.
- Las variables condición permiten que uno o más hilos se autobloqueen hasta que se alcance un estado determinado del programa.
- Cada variable condición estará asociada con un predicado (condición). Cuando el predicado se convierte en verdadero (*True*), la variable condición se utiliza para avisar a el/los hilo/s que están esperando por el cambio de estado de la condición.
- Una única variable condición puede asociarse a varios predicados (aunque dificulta la comprensión y el *debugging*).



# Pthreads – Primitivas para sincronización por condición

- Una variable condición siempre tiene un lock asociada a ella. Cada hilo bloquea este lock y evalúa el predicado definido sobre la variable compartida.
- Si el predicado es falso, el hilo espera en la variable condición (se «duerme» por lo que no usa CPU).
- Al usar variables condición en lugar de locks, estamos reemplazando un mecanismo de sincronización basado en consultas (*polling*) por uno dirigido por interrupciones.

# Pthreads – Primitivas para sincronización por condición

La API Pthreads provee las siguientes funciones para manejar las variables condición:

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

```
int pthread_cond_signal (pthread_cond_t *cond)
```

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

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

# Pthreads – Primitivas para sincronización por condición: Problema de productores y consumidores

*Main* de la solución al problema de productores-consumidores.

```
pthread_cond_t cond_queue_empty, cond_queue_full;
pthread_mutex_t task_queue_cond_lock;
int task_available;
...
int main(){
    ...
    task_available = 0;
    pthread_cond_init(&cond_queue_empty, NULL);
    pthread_cond_init(&cond_queue_full, NULL);
    pthread_mutex_init(&task_queue_cond_lock, NULL);
    /* Create y join de threads productores y consumidores*/
    ...
}
```

# Pthreads – Primitivas para sincronización por condición: Problema de productores y consumidores

Código para los *productores*.

```
void *producer(void *producer_thread_data) {
    int inserted;
    struct task my_task;
    while (!done()) {
        my_task = create_task ();
        pthread_mutex_lock (&task_queue_cond_lock);
        while (task_available == 1)
            pthread_cond_wait (&cond_queue_empty,
                               &task_queue_cond_lock);
        insert_into_queue (my_task);
        task_available = 1;
        pthread_cond_signal (&cond_queue_full);
        pthread_mutex_unlock (&task_queue_cond_lock);
    }
}
```

# Pthreads – Primitivas para sincronización por condición: Problema de productores y consumidores

Código para los *consumidores*.

```
void *consumer(void *consumer_thread_data) {
    struct task my_task;
    while (!done()) {
        pthread_mutex_lock (&task_queue_cond_lock);
        while (task_available == 0)
            pthread_cond_wait (&cond_queue_full,
                               &task_queue_cond_lock);
        my_task = extract_from_queue ();
        task_available = 0;
        pthread_cond_signal (&cond_queue_empty);
        pthread_mutex_unlock (&task_queue_cond_lock);
        process_task (my_task);
    }
}
```

# Pthreads – Primitivas para sincronización por condición

La API Pthreads provee variantes para *wait* y *signal*:

```
int pthread_cond_timedwait ( pthread_cond_t *cond,  
                             pthread_mutex_t *mutex,  
                             const struct timespec *abstime)
```

```
int pthread_cond_broadcast (pthread_cond_t *cond)
```

# Pthreads – Barreras

Pthreads provee las siguientes funciones para implementar puntos de sincronización que involucren a múltiples hilos (barreras):

```
int pthread_barrier_wait(pthread_barrier_t *barrier);
```

```
int pthread_barrier_init(pthread_barrier_t *restrict barrier,  
                        const pthread_barrierattr_t *restrict attr,  
                        unsigned count);
```

```
int pthread_barrier_destroy(pthread_barrier_t *barrier);
```

# Pthreads – Semáforos

- Un semáforo es una estructura de datos que permite sincronizar hilos (tanto para exclusión mutua como para sincronización por condición).
- POSIX definió una API para el uso de semáforos que se puede emplear con Pthreads, aun cuando no es parte del estándar.
- Los tipos de datos y funciones para usar semáforos se encuentran en *semaphore.h*
- Para declarar un semáforo, se usa el tipo *sem\_t*



# Pthreads – Semáforos

- Los semáforos deben inicializarse usando la función *sem\_init*.

```
int sem_init (sem_t *sem, int pshared, unsigned int value);
```

- Para decrementar un semáforo (P) se debe usar *sem\_wait*.

```
int sem_wait(sem_t *sem);
```

- Para incrementar un semáforo (V) se debe usar *sem\_post*.

```
int sem_post(sem_t *sem);
```

- Para destruir un semáforo se debe usar *sem\_destroy*.

```
void sem_destroy(sem_t *sem);
```

# Pthreads – Semáforos: cálculo del mínimo de una lista de números

Ahora se puede escribir el código para calcular el mínimo de una lista de números pero usando semáforos:

```
sem_t    sem;
...
int main() {
    ...
    sem_init(&sem, NULL, 1);
    ...
    /* Create y join de threads */
    ...
}

void *find_min(void *list_ptr){
    ...
    sem_wait(&sem);
    if (my_min < minimum_value) minimum_value = my_min;
    sem_post(&sem);
    ...
}
```

# Pthreads – Semáforos: Problema de productores y consumidores

*Main* de la solución al problema de productores-consumidores.

```
sem_t sem_empty, sem_full;
...
int main(){
    ...
    sem_init(&sem_empty, 1);
    sem_init(&sem_full, 0);
    /* Create y join de threads productores y consumidores*/
    ...
}
```

# Pthreads – Semáforos: Problema de productores y consumidores

Código para los *productores*.

```
void *producer(void *producer_thread_data) {  
    struct task my_task;  
    while (!done()) {  
        my_task = create_task ();  
        sem_wait(sem_empty);  
        insert_into_queue (my_task);  
        sem_post(sem_full);  
    }  
}
```

# Pthreads – Semáforos: Problema de productores y consumidores

Código para los *consumidores*.

```
void *consumer(void *consumer_thread_data) {  
    struct task my_task;  
    while (!done()) {  
        sem_wait(sem_full);  
        my_task = extract_from_queue ();  
        sem_post(sem_empty);  
        process_task (my_task);  
    }  
}
```

Costo de programación y rendimiento?

# Pthreads – Planificación de hilos

- El sistema operativo es responsable de planificar la ejecución de los hilos. Sin embargo, el programador puede influenciarlo usando los *atributos de planificación*.
- La prioridad de planificación de un hilo determina qué nivel de privilegio tendrá el mismo sobre los demás en la planificación.
- El planificador mantiene una cola separada de hilos por cada prioridad definida.
- Al momento de seleccionar un hilo para ejecutar, se elige alguno que esté listo de la cola que tenga mayor prioridad.
- Si hay varios hilos posibles en la cola seleccionada, se elige uno de ellos de acuerdo a la política de planificación.

# Pthreads – Planificación de hilos

- Para asignar y recuperar los atributos de planificación, la API de Pthreads ofrece las siguientes funciones:

```
int pthread_attr_getschedparam (const pthread_attr_t *attr,  
                                struct sched_param *param)
```

```
int pthread_attr_setschedparam (pthread_attr_t *attr,  
                                const struct sched_param *param)
```

- Para asignar y recuperar la prioridad mínima y máxima de una de terminada política de planificación, se pueden usar las siguientes funciones:

```
int sched_get_priority_min (int policy)  
int sched_get_priority_max (int policy)
```

# Pthreads – Planificación de hilos

- La política de planificación determina cómo se ejecutan y comparten recursos los hilos de una misma prioridad (en especial, el tiempo que cada uno se ejecuta).
- Pthreads soporta tres políticas de planificación diferentes:
  - SCHED\_FIFO (*first-in, first-out*): una vez en ejecución, el hilo se ejecuta hasta que termina, se bloquea o hasta que un hilo de mayor prioridad pueda ejecutarse. Los hilos de la misma prioridad son ejecutados *en orden*.
  - SCHED\_RR (*round-robin*): Similar a SCHED\_FIFO pero los hilos se ejecutan a lo sumo una determinada cantidad de tiempo (configurable).
  - SCHED\_OTHER: política adicional, no definida en el estándar. Su comportamiento depende completamente de la implementación.



# Bibliografía usada para esta clase

- POSIX Threads tutorial. Blaise Barney, Lawrence Livermore National Laboratory.  
<https://computing.llnl.gov/tutorials/pthreads/>
- Capítulo 7, An Introduction to Parallel Computing. Design and Analysis of Algorithms (2da Edition). Grama A., Gupta A., Karypis G. & Kumar V. (2003) Inglaterra: Pearson Addison Wesley.
- Capítulo 6, Parallel Programming for Multicore and Cluster Systems. Rauber, T. & Rüniger, G. (2010). EEUU: Springer-Verlag Berlin Heidelberg.