Teoria 3- Pthreads

lunes, 7 de abril de 2025 10:20

Plataformas de memoria compartidda--> usan espacios de datos común.
La memoria puede ser local (exclusiva a un proce) o global
Sub clasificación:
- Acceso uniforme a memoria (UMA) -> cuando a todos los proces les cuensa lo mismo entrar a cualquier lado
- Acceso no uniforme a memoria (NUMA) -> si según donde está el dato es más facil o dificil accederlo

Necesitas coherencia de cache

Modelo de memoria compartida
El programador es quien se ocupa
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 rendimeinto. Además, resulta difícil la predicción de performance a partir de la lectura del algoritmo.

Modelos de memoria compartida:
- Los modelos basados en procesos suponen datos locales (privados) de cada proceso. Se usa cuando cada proceso tiene su propósito específico. No está bueno cuando muchas tareas trabajan colaborativamente.
- Los modelos basados en threads o procesos "livianos" suponen que toda la memoria es global → Pthreads. Bueno cuando hay hilos que trabajan colaborativamente para resolver algo
- Modelos basados en directivas: especialización del anterior, busca facilitar programación.

Thread:
Único hilo de control en el flujo
Cada hilo tiene acceso a una memoria compartida global pero tienen tb un espacio de memoria privado

Portable: no importa en que arquitectura desarrolles, sirve en cualquiera
Oculta latencia: tener muchos hilos se intercalan a medida que tienen los recursos, entonces es como que se disimula la latencia
Planificación y balance de carga. Poco overhead por ociosidad.
Facilidad de programación y uso extendido: más facil de programar que pasaje de memoria

POSIX threads

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.

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

Para ello se emplea la función pthread_join. Esta función bloquea al hilo llamador hasta que el hilo especificado como argumento termine su ejecución.
El join es para esperar hasta que terminen la ehjecuci'pon los hilos. Se sincronizan con exit.

```
/* Espera a que los hilos terminen */
for (i = 0; i < NUM_THREADS; i++)
    pthread_join(threads[i], NULL);
```

Sí o si las funciones que queremos que ejecuten necesitamos que el encabezado sea así

```
void * hello_world (void * ptr) {
```

- 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

- 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 → Condiciones de carrera (*Race conditions)*

Mutex_locks: bloqueos por exclusión mutua

```
int pthread_mutex_lock ( pthread_mutex_t *mutex);

int pthread_mutex_unlock (pthread_mutex_t *mutex);

int pthread_mutex_init (pthread_mutex_t *mutex,
                const pthread_mutexattr_t *lock_attr);
```

- 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.

Trylock la uso para ver si puedo hacer lock. Solo sirve si puedo hacer algo mientras no puedo hacer lock

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

Para invocarla el hilo tiene que tener el control del mutex, sino se cagа
Una vez que dormis se libera el mutex.

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

El llamado a esta función “despierta” a un hilo que esté “dormido” en la variable condición (el hilo a despertar depende de las políticas de planificación)

Para poder invocarla, el hilo debe tener el control del mutex asociado

Usualmente el mutex asociado se libera (permitiendo que otros puedan usarlo)

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)
```

- El hilo se duerme a lo sumo una determinada cantidad de tiempo (*abstime*).

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

- Se despiertan a todos los hilos que están dormidos en la variable condición.