Universidad de San Andrés

I301 Arquitectura de computadoras y Sistemas Operativos

TP-5 Thread Pool

Fecha de entrega: **Domingo 22/06/2025 11:59 PM**

Importante: La resolución del TP es de manera individual.

Para aprobar el TP, es condición necesaria que el programa funcione correctamente en todos los casos de prueba incluidos en los archivos **tpcustomtest.cc** y **tptest.cc**. Esto implica que:

- La salida del programa debe ser la esperada en cada caso de prueba.
- El programa debe finalizar correctamente en todas las ejecuciones, sin presentar errores de ejecución ni situaciones de deadlocks.

El correcto funcionamiento del programa en los casos de prueba es una **condición necesaria pero no suficiente para la aprobación** del TP. Esto significa que, si el docente lo considera necesario, podrá requerir una defensa oral del trabajo, en la que el estudiante deberá explicar y justificar su implementación.

1. Introducción:

Un ***thread pool*** es un patrón de diseño utilizado para gestionar un grupo de ***workers*** para ejecutar tareas de manera eficiente. En lugar de crear y destruir hilos de forma individual para cada tarea, el thread pool mantiene un grupo de hilos activos que pueden ser reutilizados. Cuando se envía una tarea al thread pool, esta se encola y, si existe un hilo worker disponible, la toma para su ejecución. Una vez que la tarea se completa, el hilo se libera y está disponible para manejar nuevas tareas.
Esto optimiza el rendimiento al evitar el costoso proceso de creación y destrucción de hilos repetidamente.

Los *thre*ad pools son útiles en aplicaciones con múltiples tareas que deben ejecutarse concurrentemente, como servidores web, sistemas de mensajería y aplicaciones de procesamiento por lotes. Proporcionan un equilibrio entre la concurrencia y la utilización eficiente de recursos, ayudando a evitar la sobrecarga del sistema y la competencia excesiva por recursos de CPU. En resumen, un thread pool es una estrategia de gestión de hilos que mejora la eficiencia y el rendimiento al organizar y reutilizar hilos para procesar tareas de manera concurrente.

Esquema general:

- *Inicialización*:
  Se crea un despachador y un número fijo de hilos trabajadores (workers). Este número suele basarse en factores como los recursos de hardware disponibles y las características de la carga de trabajo.

- *Envío de Tareas:*
  Las tareas externas se envían al thread pool para su ejecución. Estas tareas pueden ser cualquier unidad de trabajo que necesite ser procesada de manera concurrente. En nuestro caso son void(*)(void) las funciones.

- *Cola de Tareas*:
  Las tareas enviadas se colocan en una cola de tareas, que actúa como un búfer entre las fases de envío de tareas y ejecución de tareas.
  La cola de tareas se implementa típicamente utilizando una estructura de datos como una cola o un deque.

- *Hilos Trabajadores:*
  El despachador monitorea continuamente la cola de tareas en busca de tareas entrantes. Cuando una tarea está disponible en la cola, un hilo trabajador toma la tarea y comienza a ejecutarla.

- *Ejecución de Tareas:*
  Cada hilo trabajador ejecuta tareas de manera independiente y concurrente con otros hilos en el pool. El thread pool garantiza que varias tareas puedan ejecutarse simultáneamente por diferentes hilos, maximizando el paralelismo y el rendimiento.

- *Finalización de Tareas:*
  Después de completar una tarea, el ***worker*** busca, si existe, más trabajo o entra en un estado de espera si no hay tareas disponibles.

2. Esquema del código

Se les provee los siguientes archivos:

Semaphore.h, Semaphore.cc: que contiene una implementación de un semáforo hecha por la cátedra.

Thread-pool.h: define la clase ThreadPool, diseñada para aceptar una colección de thunks, funciones sin argumentos que **no** retornan ningún valor.

Contiene los siguientes métodos:

- **ThreadPool(size_t numThreads)**: Constructor que configura el Threadpool para crear un número especificado de hilos.

- **void schedule(const std::function<void(void)>& thunk):** Programa el thunk proporcionado para ser ejecutado por alguno de los hilos del ThreadPool tan pronto como todos los thunks previamente programados hayan sido manejados.

- **void wait()**: Bloquea y espera hasta que todos los thunks previamente programados hayan sido ejecutados completamente.

- **~ThreadPool()**: Destructor que espera a que todos los thunks programados se ejecuten y luego cierra correctamente el ThreadPool y cualquier recurso utilizado durante su vida útil.

- La clase también define una política para evitar la copia y asignación, eliminando el constructor de copia y el operador de asignación, asegurando así que no se pueda clonar un **ThreadPool**.

3 - Detalles de la implementación

El constructor de la clase debería hacer lo siguiente:

Iniciar un único hilo ***dispatcher***, que extrae trabajo de la cola, asigna a un **worker** específico y le entrega la función a ejecutar a ese trabajador. Supongamos que **dt** es un miembro privado de ThreadPool de tipo thread:

dt = thread(this { dispatcher(); });

Iniciar un número específico de hilos trabajadores para esto pueden usar **wts** que es un miembro privado de **ThreadPool** de tipo **vector<worker_t>**:

wts[workerId].ts = thread([this, workerId] { worker(workerId); });

Nótese que tanto ***dispatcher*** *como* ***worker*** *deberian ser metodos de la clase ThreadPool.cc*

Les recomendamos fuertemente usar la estructura ***worker_t*** para encapsular toda la información que necesita el método ***worker*** (semáforo o condition variable, id del worker, etc.) para poder ejecutar la función que se está encolando, mediante el método **schedule**.

La función **schedule** debería aceptar un thunk (una función que no toma parámetros y no devuelve ningún valor, expresada como tipo **function<void(void)>**) y añadirla al final de una cola de dichas funciones. Cada vez que se añade una función, se debe notificar al hilo despachador. Una vez que el despachador sea notificado, **schedule** debería retornar de inmediato para que se puedan programar más funciones. schedule debería ser segura en el ámbito de los hilos (es decir, si tu programa tiene más hilos que se ejecutan fuera del ThreadPool, debería ser posible llamar a schedule desde múltiples hilos diferentes sin correr riesgo de encontrar condiciones de carrera).

El hilo ***dispatcher*** debería ejecutar en un loop. Es decir, en cada iteración, debería dormir hasta que schedule le indique que se ha añadido algo a la cola. Luego, esperaría a que un ***worker*** esté disponible, lo seleccionaría, lo marcaría como no

disponible, extraería una función de la cola, pondría una copia de esa función en un lugar donde el ***worker*** pueda acceder a ella y luego señalaría al trabajador para que la ejecutara.

Los hilos ***workers*** también deberían ejecutar en un loop, bloqueándose en cada iteración hasta que el hilo despachador les señale para ejecutar una función asignada. Una vez señalado, el trabajador debería invocar la función, esperar a que se ejecute, y luego marcar su disponibilidad para que pueda volver a ser seleccionado nuevamente por el ***dispatcher***.

La función ***wait*** debería esperar hasta que el ThreadPool esté completamente inactivo. Debería ser posible llamar a ***wait*** múltiples veces y ***wait*** debería ser segura en el ámbito de los hilos.

El destructor del **ThreadPool** debería esperar hasta que todas las funciones se hayan ejecutado (está bien llamar a wait), y luego liberar cualquier recurso del **ThreadPool**. Nuestra solución no utiliza asignación de memoria dinámica pero en caso de que quieran utilizarla recuerden liberar todos los recursos.

¿Se puede implementar ***ThreadPool*** sin un hilo ***dispatcher***?

Sí, es posible implementar un esquema donde los trabajadores son notificados de trabajo entrante, y luego extraen trabajo de la cola sin que el despachador específicamente les entregue el trabajo. Sin embargo, ***queremos que implementen ThreadPool con un hilo dispatcher***. Es una mejor práctica con comunicación/sincronización de hilos, y el hilo ***dispatcher*** es esencial para implementar Thread Pools más capaces (como un Thread Pool con inicialización ***lazy***, donde los hilos trabajadores no se crean a menos que realmente sean necesarios).

4 - Testing

Pueden probar su ***ThreadPool*** usando ***tptest.cc*** y ***tpcustomtest.cc*** (que se debería compilar a ***tptest*** y ***tpcustomtest***). Si revisan los archivos cada uno tiene su propio main por lo que van a tener que modificar el Makefile cada vez que quieran compilarlos. Nuestro consejo es que comiencen con **main.cc**, vean que hay escrito ahí, lo modifiquen, si lo desean, y generen sus propias pruebas.

En **main.cc** les proveemos código que demuestra cómo utilizar la clase **ThreadPool** para dividir una tarea en múltiples workers y coordinar su ejecución para sumar segmentos de un vector de enteros.

A continuación, se describe paso a paso cómo interactúa con la clase **ThreadPool**:

- Define la función **computeSum** que calcula la suma de una subsección de un vector de enteros.

- Crea una instancia de ThreadPool con un número especificado de hilos (numThreads).

- Prepara un vector de enteros data y un vector results para almacenar los resultados de cada segmento procesado por los hilos.

- Calcula el tamaño de cada segmento de datos que cada hilo debería procesar (chunkSize).

- Utiliza un loop para asignar cada segmento de datos a un hilo diferente en el ThreadPool usando el método schedule. Las tareas son encapsuladas en funciones lambda que capturan por referencia las variables necesarias.

- Llama al método wait para bloquear la ejecución hasta que todos los hilos hayan completado sus tareas, asegurando que todos los cálculos hayan terminado antes de proceder.

4 - ¿Qué esperamos?

En este trabajo práctico esperamos que desarrollen el código de la clase ***ThreadPool*** y la nota dependerá exclusivamente que esta implementación siga el patrón de diseño de un Thread Pool, para esto la recomendación es consultar todos los recursos que hay disponibles. Antes de que escriban una línea de código entiendan bien cuál es el problema que están queriendo resolver.

La evaluación también se centrará en el uso eficiente de la memoria, la claridad y legibilidad del código. Si bien **no vamos** a evaluar los test que ustedes generen es fundamental que los casos de prueba cubran una amplia gama de escenarios, incluyendo condiciones límite y casos atípicos, para asegurar que el software es robusto y confiable.

Les recomendamos fuertemente que sigan lo descrito en la sección 3. Seguir este diseño no solo simplificará significativamente la comprensión del concepto, sino que también les ayudará a evitar errores comunes facilitando así un desarrollo más fluido y coherente del proyecto.