Universidad Nacional Autónoma de México

“Por mi raza hablará el espíritu”

Facultad de Ingeniería

Materia: Sistemas operativos.

Profesor: Gunnar Wolf.

Alumna: González Martínez Michelle Paola.

Fecha de entrega: 16 de octubre de 2025.

Implementación del Servidor Web

Patrón Jefe-Trabajador con Sincronización

1. Problema Seleccionado

He elegido resolver el problema del **"Servidor Web"** que implementa el patrón Jefe-Trabajador. Este problema simula cómo funcionan los servidores web reales como Apache, donde un proceso principal (jefe) recibe las conexiones de red y las distribuye entre múltiples hilos trabajadores que las procesan.

El problema requiere que:

- El jefe lance k hilos trabajadores al iniciar
- Los trabajadores sin trabajo se vayan a dormir
- El jefe reciba conexiones y las asigne a cualquier trabajador disponible
- El jefe despierte al trabajador asignado
- Se mantengan siempre k trabajadores listos para atender solicitudes

2. Lenguaje y Entorno de Desarrollo

- Lenguaje: Python 3 (versión 3.6 o superior)
- Requisitos para Ejecutar:
  Para ejecutar el programa se necesita:
  1. Tener Python 3 instalado: python3 –version
  2. Instalar colorama: pip install colorama
  3. Ejecutar el programa: python servidor_web.py
  4. Para detener: Presionar Ctrl+C

2. Estrategia de Sincronización

Patrón Implementado: Jefe-Trabajador

Este patrón es una variante del modelo Productor-Consumidor donde el jefe actúa como productor de tareas y los trabajadores como consumidores. La diferencia principal es que aquí mantenemos un número fijo de trabajadores que se reutilizan continuamente.

Mecanismos de Sincronización Utilizados

Para resolver este problema utilicé **semáforos**, que son el mecanismo de sincronización que hemos estado estudiando en clase. Los semáforos permiten coordinar el acceso a recursos compartidos y sincronizar la ejecución entre hilos.

Semáforos implementados:

1. **mutex_peticiones** (Semáforo binario/Mutex)
    - Valor inicial: 1
    - Protege el acceso a la lista compartida de peticiones pendientes
    - Garantiza exclusión mutua cuando se agregan o quitan peticiones
2. **sem_hay_peticiones** (Semáforo contador)
    - Valor inicial: 0
    - Cuenta cuántas peticiones hay pendientes de procesar
    - Los trabajadores esperan (duermen) en este semáforo cuando no hay trabajo
    - El jefe incrementa este semáforo al asignar una nueva petición
3. **sem_trabajador_disponible** (Semáforo contador)
    - Valor inicial: NUM_TRABAJADORES (5)
    - Indica cuántos trabajadores están disponibles
    - El jefe espera en este semáforo si todos los trabajadores están ocupados
    - Los trabajadores lo incrementan cuando terminan de procesar una petición

Flujo de Sincronización

El funcionamiento del servidor sigue este flujo:

Inicialización: Al arrancar el servidor, se crean 5 hilos trabajadores. Cada trabajador inmediatamente intenta hacer acquire() en sem_hay_peticiones, pero como su valor inicial es 0, todos los trabajadores se bloquean y "duermen" esperando trabajo.

Llegada de una conexión: Cuando el jefe recibe una nueva conexión, primero hace acquire() en sem_trabajador_disponible. Si hay trabajadores disponibles, continúa. Si todos están ocupados, el jefe se bloquea hasta que uno se libere.

Asignación de trabajo: El jefe crea la petición con sus datos (tipo de petición HTTP, recurso solicitado, tiempo de procesamiento). Luego usa el mutex para agregar la petición a la lista compartida de forma segura. Finalmente, hace release() en sem_hay_peticiones, lo que despierta a uno de los trabajadores dormidos.

Procesamiento: El trabajador despertado toma la petición de la lista (usando el mutex para acceso seguro), la procesa durante el tiempo simulado, y cuando termina hace release() en sem_trabajador_disponible para indicar que está libre nuevamente. Luego vuelve a esperar en sem_hay_peticiones.

Este ciclo garantiza que:

- Los trabajadores realmente duermen cuando no hay trabajo (no hacen busy-waiting)
- No hay condiciones de carrera al acceder a la lista de peticiones
- El jefe no asigna trabajo si no hay trabajadores disponibles
- Siempre hay k trabajadores listos (o esperando estar listos)

3. Características de la Implementación

Simulación de Peticiones HTTP

El programa simula peticiones HTTP reales con:

- Diferentes métodos: GET, POST, PUT, DELETE
- Varios recursos: páginas HTML, APIs, imágenes, estilos CSS
- Tiempos de procesamiento variables (simulando complejidad diferente)

Interfaz Visual

Siguiendo el estilo de los ejemplos de clase, utilicé colores para distinguir los diferentes actores:

- **Magenta** para el jefe
- **Verde** para los trabajadores procesando
- **Cyan** para inicialización de trabajadores
- **Amarillo** para el estado del sistema

- **Blanco** para mensajes generales

Información en Tiempo Real

Cada 5 segundos se muestra el número de peticiones pendientes en la cola, permitiendo observar cómo el sistema maneja la carga de trabajo.

4. Posibles Mejoras

Aunque el programa funciona correctamente, se podrían agregar algunas mejoras:

1. **Manejo de prioridades**: Implementar una cola de prioridad para que ciertas peticiones (como las de API) se procesen antes.
2. **Límite de cola**: Agregar un límite máximo de peticiones en cola para evitar que el servidor se sature.
3. **Balanceo de carga**: En lugar de que cualquier trabajador tome la siguiente petición, se podría implementar un sistema que distribuya la carga de manera más equitativa.
4. **Persistencia**: Guardar un log de las peticiones procesadas para análisis posterior.

Conclusión

Esta implementación demuestra cómo los patrones de sincronización que estudiamos en clase se aplican a problemas reales.

El uso de semáforos permite coordinar múltiples hilos de manera eficiente y segura. Los trabajadores duermen cuando no hay trabajo (usando sem_hay_peticiones.acquire()), lo que es mucho más eficiente que hacer polling constante. El jefe puede manejar la distribución de trabajo sin preocuparse por los detalles de qué trabajador específico tomará cada petición.

Este ejercicio refuerza la importancia de usar mecanismos de sincronización apropiados en lugar de soluciones ad-hoc con condicionales y verificación de estado, que pueden llevar a condiciones de carrera y uso ineficiente de recursos.