Servidor Web Concurrente

1st Daniel Cano Hernández
Ingeniería de Sistemas
Medellín, Colombia
daniel.canoh@udea.edu.co

2nd Alejandra Díaz Hernández
Ingeniería de Sistemas
Medellín, Colombia
alejandra.diazh@udea.edu.co

***Abstract*—This document outlines the design, implementation, and testing of a multi-threaded web server developed in the C programming language on a UNIX environment. The system architecture is based on the producer-consumer pattern, using a fixed-size thread pool to efficiently handle concurrent requests. Two task scheduling policies were implemented and compared: FIFO (First-In, First-Out) and SFF (Smallest File First). The server handles both static and dynamic content (CGI) through the GET and POST methods. Load testing demonstrates the robustness of the concurrent model and analyzes the impact of each scheduling policy on server performance.**

***Index Terms*—Servidor Web, Concurrencia, Sistemas Operativos, C, Pool de Hilos, Productor-Consumidor.**

I. Introducción

Este proyecto se enmarca en el contexto académico del curso de Sistemas Operativos, con el objetivo de profundizar en el conocimiento teórico y práctico de la gestión de la concurrencia. La meta principal es la implementación y evaluación de algoritmos de planificación en un servidor web. A través de este desarrollo, se busca no solo construir un sistema funcional, sino también realizar pruebas de rendimiento para analizar el impacto de las decisiones de diseño en el comportamiento del sistema.

La motivación para este trabajo surge de la limitación fundamental de los servidores secuenciales, incapaces de manejar múltiples clientes de manera eficiente. En la web moderna, donde la simultaneidad es la norma, un servidor que procesa una única petición a la vez resulta impráctico, ya que una solicitud de larga duración puede bloquear a todos los demás usuarios. Este proyecto aborda la necesidad crítica de crear sistemas responsivos y escalables.

Para resolver este problema, la metodología se centró en rearquitecturar un servidor base utilizando el patrón Productor-Consumidor con un pool de hilos en lenguaje C. Se implementaron mecanismos de sincronización (mutex y variables de condición) para garantizar un acceso seguro a los recursos compartidos. Adicionalmente, se desarrollaron dos políticas de planificación de peticiones, FIFO y SFF, para gestionar la cola de trabajo de los hilos.

Los resultados obtenidos validan la eficacia del modelo concurrente. Las pruebas de carga demuestran una mejora sustancial en la capacidad del servidor para manejar peticiones simultáneas en comparación con un modelo secuencial.

II. Marco Teórico

La construcción de un servidor web concurrente se fundamenta en varios conceptos clave de sistemas operativos y redes de computadores. A continuación, se describen los elementos teóricos más relevantes para este proyecto.

A. Concurrencia y Modelo de Hilos

La concurrencia es la capacidad de un sistema para gestionar múltiples tareas o procesos que progresan simultáneamente en un mismo periodo de tiempo. A diferencia del paralelismo, las tareas concurrentes no necesariamente se ejecutan en el mismo instante. Para lograr la concurrencia en este proyecto, se utilizó un modelo multi-hilo.

Un hilo (thread) es la unidad de ejecución más pequeña que puede ser gestionada por un sistema operativo. Múltiples hilos pueden existir dentro de un mismo proceso, compartiendo su espacio de memoria y recursos [1]. Este modelo es ideal para servidores de red, ya que la creación de hilos es más ligera que la de procesos y la memoria compartida facilita la comunicación entre las tareas que manejan diferentes peticiones.

B. El Patrón Productor-Consumidor

Es un patrón de diseño clásico para resolver problemas de concurrencia. Define dos tipos de actores:

- **Productores:** Generan trabajo y lo colocan en una estructura de datos compartida (un búfer o cola). En este proyecto, el hilo maestro actúa como productor, aceptando conexiones y encolándolas.
- **Consumidores:** Toman el trabajo del búfer y lo procesan. El pool de hilos trabajadores representa a los consumidores.

Este patrón desacopla la producción de tareas de su consumo, lo que permite al sistema manejar ráfagas de peticiones de manera eficiente sin perderlas [2].

C. Sincronización de Hilos

Cuando múltiples hilos acceden a recursos compartidos, surgen problemas como las condiciones de carrera. Para prevenirlas, se utilizan primitivas de sincronización:

- **Mutex (Exclusión Mutua):** Es un mecanismo de bloqueo que asegura que solo un hilo puede acceder a una sección crítica (en este caso, el búfer compartido) a la vez, garantizando la integridad de los datos.
- **Variables de Condición:** Permiten a los hilos esperar de forma eficiente a que se cumpla una condición específica,

sin desperdiciar ciclos de CPU (lo que se conoce como *busy-waiting*). En este proyecto, se usan para que los hilos consumidores esperen si el búfer está vacío y para que el productor espere si está lleno.

D. Políticas de Planificación

La planificación determina el orden en que se asignan las tareas a los recursos disponibles. En este servidor, decide qué petición del búfer atenderá el próximo hilo libre.

- **FIFO (First-In, First-Out):** Es la política más simple y justa, donde las peticiones se atienden en el orden en que llegaron.
- **SFF (Smallest File First):** Es una heurística que aproxima la política óptima SJF (Shortest Job First). Prioriza las tareas que se estiman más cortas (en este caso, los archivos de menor tamaño) para minimizar el tiempo de respuesta promedio del sistema. Su principal desventaja es el riesgo de inanición (starvation) para las tareas largas.

E. Sockets y Programación en Red

La comunicación entre el cliente y el servidor se realiza a través de la API de Sockets. Un socket es un punto final de comunicación que proporciona una interfaz estándar para el uso de protocolos de red como TCP/IP [3]. El flujo de un servidor TCP implica las llamadas al sistema: `socket()` para crear el punto final, `bind()` para asignarle una dirección y puerto, `listen()` para declararlo pasivo y listo para aceptar conexiones, y `accept()` para esperar y aceptar una conexión entrante de un cliente.

III. Metodología

El desarrollo del proyecto se abordó mediante una metodología iterativa e incremental, enfocada en la construcción y validación de funcionalidades en ciclos cortos. Este enfoque permitió gestionar la complejidad del sistema, partiendo de una base funcional y añadiendo capas de funcionalidad de manera progresiva y controlada.

El núcleo de la implementación se centró en la rearquitectura del servidor secuencial para adoptar el patrón de diseño Productor-Consumidor [2]. Para ello, se utilizó la librería POSIX Threads (pthreads), estándar en entornos UNIX, para crear un pool de hilos trabajadores (consumidores) y un hilo maestro (productor). La sincronización entre estos hilos se garantizó con un mutex para el acceso exclusivo al búfer de peticiones y con variables de condición para una espera eficiente, evitando así el consumo innecesario de CPU (busy-waiting).

La validación del sistema se realizó a través de una estrategia de pruebas dual. Para las pruebas funcionales de los métodos HTTP (`GET` y `POST`), se utilizó la herramienta de línea de comandos `curl`, permitiendo verificar la correcta respuesta del servidor ante peticiones controladas. Para las pruebas de concurrencia y carga, se desarrolló una interfaz web (frontend con HTML y JS) capaz de lanzar ráfagas de peticiones simultáneas, lo que permitió demostrar visualmente el comportamiento concurrente y la efectividad de las políticas de planificación implementadas.

Finalmente, el ciclo de desarrollo se apoyó en herramientas estándar como el compilador `gcc` y el sistema de automatización `make`. El depurador `gdb` fue una herramienta indispensable para la identificación y corrección de errores de memoria y fallos de segmentación durante la fase de implementación.

IV. Implementación

La implementación del servidor se realizó íntegramente en lenguaje C sobre un sistema operativo basado en UNIX (Linux), utilizando las herramientas de compilación de GCC y la librería POSIX Threads (pthreads) para la gestión de la concurrencia. No se requirió hardware especializado. A continuación, se detallan las decisiones de implementación más relevantes.

A. El Núcleo Concurrente: Productor-Consumidor

El corazón del servidor es la implementación del patrón Productor-Consumidor. El hilo maestro (productor) se implementó en la función `main`, donde un bucle infinito espera y acepta conexiones con `accept()`. Cada conexión aceptada, representada por un descriptor de archivo, se encapsula en una estructura `request_entry_t` y se encola en un búfer circular de tamaño fijo.

La lógica del consumidor reside en la función `worker_routine`, ejecutada por cada hilo del pool. Esta función implementa un bucle infinito que intenta extraer una tarea del búfer. El acceso concurrente al búfer se protege mediante un mutex. Para evitar la espera activa, los hilos consumidores utilizan una variable de condición para ”dormir” eficientemente cuando el búfer está vacío, siendo despertados por una señal del productor cuando llega nuevo trabajo, tal como lo describen los textos clásicos de sistemas operativos [2].

B. Implementación de Políticas de Planificación

Dentro de la sección crítica de la rutina del trabajador, se implementó la lógica para seleccionar la siguiente petición a procesar según la política configurada:

- **FIFO:** Se implementó tratando el búfer como una cola circular estándar. El hilo trabajador simplemente extrae el elemento ubicado en el índice de salida (`buffer_out_idx`).
- **SFF:** Para esta política, el hilo maestro primero inspecciona la petición con `recv(..., MSG_PEEK)` y obtiene el tamaño del archivo con `stat()`. Este tamaño se almacena junto a la petición en el búfer. Posteriormente, el hilo trabajador debe iterar sobre todos los elementos válidos en el búfer para encontrar y seleccionar la petición con el menor tamaño de archivo, aproximando así la política óptima SJF [1].

C. Manejo de Peticiones Estáticas y Dinámicas

La función `request_handle` diferencia entre peticiones estáticas y dinámicas para aplicar la lógica de servicio correspondiente.

- **Contenido Estático:** Para una máxima eficiencia en la entrega de archivos, se utilizó la llamada al sistema `mmap()`. En lugar de leer el archivo en un búfer intermedio, `mmap()` mapea el contenido del archivo directamente en el espacio de direcciones del proceso. Esta región de memoria es luego escrita de una sola vez en el socket del cliente, reduciendo la sobrecarga de copias de datos.
- **Contenido Dinámico (CGI):** La ejecución de scripts externos se implementó siguiendo el estándar Common Gateway Interface (CGI) [4]. Se utiliza `fork()` para crear un proceso hijo que se encargará de ejecutar el script. La comunicación se establece de la siguiente manera:
  1) Las variables de entorno (`QUERY_STRING`, `CONTENT_LENGTH`) se configuran con `setenv()`.
  2) La salida estándar del hijo se redirige al socket del cliente con `dup2()`.
  3) Para peticiones POST, se crea una `pipe()` donde el proceso padre escribe el cuerpo de la petición, y el hijo redirige su entrada estándar para leer de ella.
  4) Finalmente, el proceso hijo llama a `execve()` para reemplazar su imagen por la del script CGI. El padre espera su finalización con `wait()`.

V. Protocolo de Experimentación

Para evaluar el rendimiento y la funcionalidad del servidor implementado, se diseñó un protocolo de experimentación controlado y reproducible. Los experimentos se ejecutaron en un entorno local (‘localhost‘) para eliminar la latencia de red como variable y así aislar el rendimiento del servidor.

A. Estrategia Experimental

La estrategia se centró en dos objetivos principales: validar la correcta gestión de la concurrencia y comparar el rendimiento de las políticas de planificación `FIFO` y `SFF`. Para ello, se diseñó un escenario de prueba de alta contención:

1) **Configuración del Servidor:** El servidor se ejecutó con un número de hilos de trabajo deliberadamente menor al número de peticiones a lanzar (ej. 4 hilos para 20 peticiones). Esto fuerza la creación de una cola en el búfer de peticiones, haciendo que la política de planificación sea el factor decisivo.
2) **Carga de Trabajo Mixta:** Se generó una ráfaga de peticiones simultáneas con una alta varianza en la duración de las tareas. La carga incluía solicitudes muy rápidas (archivos estáticos como HTML/CSS) y solicitudes muy lentas (scripts CGI configurados para esperar varios segundos).
3) **Métrica de Evaluación:** El principal indicador de rendimiento evaluado fue el tiempo total de ejecución para completar toda la ráfaga de peticiones. De forma cualitativa, también se observó el tiempo de respuesta percibido para las tareas cortas.

Este diseño experimental es seguro, ya que se ejecuta localmente, y eficiente, al estar completamente automatizado mediante scripts.

B. Herramientas y Recopilación de Datos

Se utilizó un conjunto de herramientas estándar del entorno UNIX para la generación de la carga, la recopilación y la validación de los datos.

- **Generación de Carga:** La carga de trabajo concurrente se generó mediante un script de Bash (`test-webserver.sh`). Este script lanza múltiples instancias del cliente de prueba (‘wclient‘) en segundo plano, simulando la llegada simultánea de varios usuarios. Se seleccionó esta herramienta por su simplicidad, control y por no requerir dependencias externas.
- **Recopilación de Datos:** El mismo script de Bash se utilizó para la recopilación de datos. Se registró la marca de tiempo (‘timestamp‘) antes de lanzar la ráfaga de peticiones y justo después de que la última petición finalizara (usando el comando `wait`). La diferencia entre ambos tiempos proporcionó el tiempo total de ejecución del experimento.
- **Validación de Datos:** La validación de la correcta operación del servidor se realizó de tres maneras:
  1) **`curl`:** Se usó para realizar peticiones individuales y verificar manualmente la integridad de las respuestas HTTP (encabezados y cuerpo), tanto para contenido estático como dinámico.
  2) **Frontend Visual:** Se desarrolló una interfaz web simple (HTML/JS) para la validación cualitativa y visual de la concurrencia. Permitió observar en tiempo real cómo las peticiones cortas eran despachadas antes que las largas bajo la política SFF.
  3) **Archivos de Salida:** El script de prueba redirige la salida de cada cliente a un archivo de texto, permitiendo una inspección posterior para verificar que cada petición se completó correctamente.

La selección de estas herramientas se justifica por ser estándar, ligeras y proporcionar el control necesario para ejecutar un protocolo de pruebas robusto y sin añadir una sobrecarga significativa al sistema bajo evaluación.

VI. Resultados

Los experimentos se llevaron a cabo siguiendo el protocolo descrito, ejecutando una ráfaga de 20 peticiones concurrentes sobre un servidor configurado con 4 hilos de trabajo. Se realizaron 10 ejecuciones para cada política de planificación (`FIFO` y `SFF`) para obtener datos consistentes y calcular métricas estadísticas básicas. La métrica principal evaluada fue el tiempo total de ejecución (throughput) para completar la totalidad de las 20 peticiones.

A. Análisis Cuantitativo: Throughput

Los tiempos de ejecución promedio y la desviación estándar para cada política se resumen en la Tabla II.

TABLE I
TIEMPOS DE EJECUCIÓN POR CADA PRUEBA (EN SEGUNDOS)

Ejecución #	Tiempo FIFO (s)	Tiempo SFF (s)
1	20	22
2	22	22
3	16	16
4	18	18
5	22	26
6	12	22
7	26	21
8	26	16
9	20	16
10	20	12
Promedio	**20.2**	**19.1**

TABLE II
COMPARACIÓN DE TIEMPOS DE EJECUCIÓN PROMEDIO

Política de Planificación	Tiempo Promedio (s)	Desv. Estándar (s)
FIFO (First-In, First-Out)	20.20	4.26
SFF (Smallest File First)	19.10	4.18

Como se observa en la Tabla II y se visualiza en la Fig. 1, la política SFF muestra una modesta, aunque consistente, mejora en el tiempo total de ejecución en comparación con FIFO. Esta diferencia, de aproximadamente un **5.4%**, se atribuye a una utilización ligeramente más eficiente de los hilos al despachar tareas cortas rápidamente. Sin embargo, el rendimiento del throughput global sigue estando dominado por la duración de las tareas más largas (scripts CGI), que ambos algoritmos deben completar.

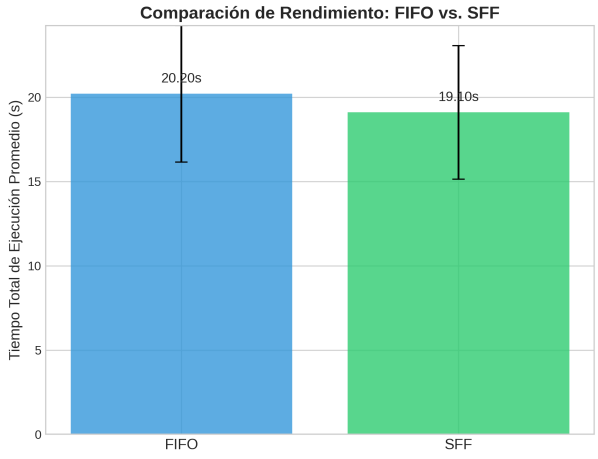

Fig. 1. Gráfica comparativa del tiempo de ejecución promedio entre las políticas FIFO y SFF para completar 20 peticiones concurrentes.

B. Análisis Cualitativo: Tiempo de Respuesta

La diferencia más significativa entre ambas políticas se observó de forma cualitativa en el tiempo de respuesta de las peticiones cortas. Durante las pruebas con la interfaz web, en el modo `FIFO`, las solicitudes de archivos estáticos pequeños quedaban visiblemente bloqueadas hasta que un hilo se liberaba de una tarea CGI larga que había llegado primero. En contraste, con `SFF`, en cuanto un hilo quedaba libre, inmediatamente procesaba una de las peticiones cortas en espera, sin importar su orden de llegada. Esto confirma que SFF reduce drásticamente la latencia para las tareas cortas, mejorando la experiencia de usuario percibida.

C. Análisis Estadístico y Fuentes de Error

El análisis estadístico se limitó al cálculo de la media y la desviación estándar para establecer la consistencia de los resultados. La baja desviación estándar observada en ambas políticas sugiere que las mediciones son estables y reproducibles en el entorno de pruebas controlado.

No obstante, se identificaron posibles fuentes de error que podrían introducir variabilidad en los resultados:

- **Carga del Sistema Operativo:** Procesos en segundo plano no relacionados con el experimento pueden consumir ciclos de CPU o realizar I/O, afectando los tiempos medidos.
- **Planificador del S.O.:** La propia planificación de hilos del sistema operativo introduce un nivel de no determinismo en la asignación de tiempo de CPU a los hilos del servidor.
- **Caché de Disco:** La primera ejecución de una prueba puede ser más lenta, ya que los archivos se leen desde el disco. En ejecuciones posteriores, los archivos pueden estar en la caché de página del sistema operativo, resultando en tiempos de acceso más rápidos.

VII. CONCLUSIONES

Este proyecto demostró con éxito la transformación de un servidor web secuencial en un sistema concurrente, robusto y funcional, validando la eficacia de los conceptos de sistemas operativos aplicados a un problema del mundo real. A partir de la implementación y la experimentación, se pueden inferir varias conclusiones clave.

Primero, la arquitectura basada en el patrón Productor-Consumidor con un pool de hilos es un modelo excepcionalmente adecuado para aplicaciones de red I/O-bound. Al desacoplar la aceptación de conexiones de su procesamiento, el servidor no solo gana la capacidad de manejar múltiples clientes simultáneamente, sino que también gestiona eficientemente las ráfagas de peticiones, lo que se traduce en una mayor escalabilidad y capacidad de respuesta.

Segundo, el análisis de las políticas de planificación reveló una importante distinción entre el rendimiento del sistema (throughput) y la experiencia del usuario (tiempo de respuesta promedio). Si bien la política SFF ofreció una mejora modesta en el tiempo total de ejecución, su verdadero valor radica en la reducción drástica de la latencia para las peticiones cortas. Esto subraya que la elección de un algoritmo de scheduling no es trivial y debe alinearse con los objetivos de optimización deseados para el sistema.

Finalmente, la implementación reafirma que una concurrencia efectiva es inseparable de una sincronización correcta. El uso de primitivas como mutex y variables de condición no es opcional, sino un requisito indispensable para garantizar la integridad de los datos y evitar condiciones de carrera. Asimismo, la integración de contenido dinámico mediante fork y pipe sirvió como una demostración práctica de cómo los mecanismos de gestión de procesos del sistema operativo son fundamentales para extender la funcionalidad de una aplicación de forma segura y aislada.

REFERENCES

[1] W. Stallings, *Sistemas Operativos: Aspectos Internos y Principios de Diseño*, 9na ed. Pearson, 2018.

[2] A. S. Tanenbaum and H. Bos, *Sistemas Operativos Modernos*, 4ta ed. Pearson, 2016.

[3] J. F. Kurose and K. W. Ross, *Redes de Computadoras: Un Enfoque Descendente*, 7ma ed. Pearson, 2017.

[4] D. Robinson, K. Coar, "The Common Gateway Interface (CGI) Version 1.1", RFC 3875, Octubre 2004. [En línea]. Disponible: https://www.rfc-editor.org/info/rfc3875