Avance Proyecto final Sistemas Operativos

1. Integrantes:
    - David Alejandro Gutierrez Leal
    - Laura Andrea Castrillón Fajardo
    - Laura Indabur
    - Samuel Martínez Arteaga

2. Descripción de avances.

Para el avance del proyecto hemos trabajado en las funcionalidades de forma individual: comandos, compresión y encriptación, priorizando que cada módulo sea consistente y validable por separado antes de integrarlos.

- **Comandos y validación de parámetros (comandos.h / comandos.cpp):** Se definió la estructura Parámetros y funciones para parsear y validar argumentos desde una futura interfaz. A partir de la entrada, se convierten argumentos en *flags* y valores (-c, -d, -e, -u, -ce, -ud; opciones: --comp-alg, --enc-alg, -i, -o, -k). Se incluyó ayuda con -h/--help y reglas como: exigir al menos una operación, requerir -i/-o, y solicitar algoritmos y clave según la operación. En caso de error, se imprime un mensaje claro y se finaliza con exit(1).

- **Compresión LZ77 (formato y lógica):** Implementación con formato binario: [HEADER: 8 bytes][TOKEN_1: 5 bytes]...[TOKEN_N: 5 bytes]. Cada token es LITERAL (carácter sin comprimir) o REFERENCE (coincidencia con texto previo). Todo en *little-endian* (tipo, valor, distancia).
  Compresión con ventana deslizante de 4 KB y *lookahead* de 18 bytes, búsqueda completa de coincidencias y soporte de solapamientos circulares (p. ej., "aaaaa" → [LITERAL 'a'][REF(4,1)]).
  La descompresión lee los tokens en orden y copia byte a byte para manejar correctamente los solapamientos. Se usó <algorithm> (solo std::min), funciones inline para conversiones *little-endian*, std::vector para eficiencia y validaciones de integridad. Sin concurrencia.

- **Herramientas base para Huffman (bits y codificación):** Clases BitWriter y BitReader para operar a nivel de bit (códigos de longitud variable). BitWriter usa máscaras con OR (|) y *flush* automático por byte; BitReader usa AND (&) y recorre índices de bit de 7 a 0. Se unificó la estructura de Token con LZ77: LITERAL (0–255) o REFERENCE (distancia, longitud). El formato binario mantiene 5 bytes por token: 1 para tipo, 2 para valor y 2 para distancia, en *little-endian*. Árbol de Huffman clásico: conteo de frecuencias, priority queue con

comparador a medida y combinación de los dos nodos menos frecuentes hasta la raíz. Generación de códigos por recorrido recursivo (izquierda "0", derecha "1"). Solo contenedores estándar (std::vector, std::map, std::priority_queue).

- **Cifrado en flujo ChaCha20 – (encriptación):** Implementación del stream cipher ChaCha20 con helpers little-endian (load32_le, store32_le) y constantes "expand 32-byte k". Núcleo con rotl32 y quarter_round (rotaciones 16/12/8/7).
  Estado por bloque: 16 palabras = 4 constantes + 8 de clave (256 bits) + contador (32 bits) + nonce (96 bits).
  chacha20_init() carga key/nonce/counter y deja un estado base para depuración.
  chacha20_block() aplica 20 rondas (10 columnas + 10 diagonales), suma con el estado original y serializa 64 bytes de keystream; incrementa el contador.
  chacha20_xor() procesa buffers arbitrarios en trozos de 64 B haciendo XOR keystream⊕datos (mismo camino para cifrar/descifrar), manejando bloques parciales.