TEC | Tecnológico de Costa Rica

Escuela de ingeniería en computación
Ingeniería en Computación
IC-6600 - Principios de sistemas operativos

Proyecto I

Compresión Huffman

Tim Scarlith

Josue Torres

Wilfredo Villegas

San José, Costa Rica
Abril 2025

Índice general

Capítulo 1

Introducción

1.1. Proyecto Gutenberg

El Proyecto Gutenberg es una biblioteca digital en línea que ofrece eBooks gratuitos y de dominio público, accesibles sin coste alguno para cualquier usuario conectado a Internet (Gutenberg, s.f.-a). Fue iniciado en 1971 por Michael S. Hart en la Universidad de Illinois, quien comenzó el proyecto digitalizando la Declaración de Independencia de Estados Unidos (Contributors, 2025c). Su misión es digitalizar y archivar obras culturales, promoviendo la creación y distribución libre de libros electrónicos mediante el trabajo de voluntarios alrededor del mundo (Gutenberg, s.f.-b). A día de hoy, el Proyecto Gutenberg cuenta con más de 70 000 títulos en diversos formatos (HTML, EPUB, MOBI, texto plano e incluso audiolibros), disponibles para su descarga y lectura en casi cualquier dispositivo (Contributors, 2025d).

1.2. Algorimo de Huffman

En 1952, David A. Huffman desarrolló un método de compresión de datos sin pérdida conocido como el algoritmo de Huffman. Este algoritmo asigna códigos de longitud variable a los caracteres, estructurando la información en un árbol binario(Wikipedia contributors, 2024).

Su funcionamiento se basa en calcular la frecuencia de aparición de cada carácter en el texto a comprimir y construir un árbol binario en el que los nodos con menor frecuencia se combinan progresivamente. En cada paso, se unen los dos nodos de menor frecuencia, priorizando que ambos tengan frecuencias iguales si es posible, hasta formar un único nodo raíz(Wikipedia contributors, 2024).

En el árbol resultante, cada rama hacia la izquierda se representa con un 0 y cada rama hacia la derecha con un 1, generando así un código binario único para cada carácter, determinado por el camino desde la raíz hasta su hoja correspondiente(Wikipedia contributors, 2024).

Este método presenta varias ventajas, como su eficiencia al reducir el tamaño de archivos con caracteres que se repiten con frecuencia, y su sencillez en la implementación. Al generar códigos más cortos para los caracteres más comunes, logra una compresión efectiva sin pérdida de información(Google Search, 2024).

Sin embargo, también tiene limitaciones. No es tan eficaz cuando los caracteres tienen frecuencias similares o cuando se aplican sobre archivos muy pequeños, ya que el almacenamiento del árbol puede ocupar más espacio del que se ahorra(Google Search, 2024).

1.3. ¿Qué es un algoritmo de compresión?

Los algoritmos de compresión buscan reducir la redundancia estadística o eliminar datos irrelevantes dentro de un flujo de bits para representar la misma información con menos símbolos (Contributors, 2025a). Dependiendo de su diseño, pueden ser reversibles (sin pérdida) o irreversibles (con pérdida).

1.3.1. Compresión sin pérdida (Lossless)

La compresión sin pérdida permite que, después de descomprimir, se recupere exactamente el mismo archivo original, bit por bit (Contributors, 2025b). Este enfoque elimina únicamente la redundancia estadística, sin descartar ninguna parte de la información esencial (Contributors, 2024)

1.3.2. Compresión con pérdida (Lossy)

La compresión con pérdida es un método de codificación de datos que reduce el tamaño de los archivos eliminando información menos importante o redundante. Este tipo de compresión se utiliza en medios digitales como imágenes, audio y video, donde una ligera pérdida de calidad es aceptable (Fernández, s.f.).

1.4. Concurrencia y Paralelismo

La concurrencia es la capacidad del sistema para gestionar múltiples procesos o hilos al mismo tiempo. Estos procesos no necesitan estar relacionados entre sí, ya que se ejecutan de manera independiente, cada uno asignado a un núcleo (core) del procesador. Por esta razón, el número máximo de procesos que pueden ejecutarse simultáneamente mediante concurrencia está limitado por la cantidad de núcleos disponibles en el CPU (Blancarte, 2017).

El paralelismo, por otro lado, consiste en tomar un único problema y dividirlo en subproblemas más pequeños que se resuelven de manera concurrente. Cada subproceso se ejecuta simultáneamente en diferentes núcleos, aprovechando al máximo la capacidad del procesador para obtener una solución más rápida(Blancarte, 2017).

La principal diferencia entre paralelismo y concurrencia radica en su objetivo: en la concurrencia, los procesos pueden ser independientes y no necesariamente están trabajando en conjunto. En cambio, en el paralelismo, todos los procesos concurrentes colaboran estrechamente para resolver un mismo problema, y el resultado final depende de la correcta ejecución y combinación de cada subproceso(Blancarte, 2017).

Capítulo 2

Desarrollo

2.1. Libreria pthread

Un hilo (thread) es una unidad de ejecución dentro de un proceso que comparte el mismo espacio de direcciones y recursos. Para el lenguaje de programación C, pthread es un estándar, una API para manejo de hilos en sistemas UNIX.La libreria permiten implementar concurrencia en programas escritos en C, optimizando el uso de los recursos del sistema y permitiendo ejecución paralela de procesos.

Un hilo representa una unidad básica de ejecución dentro de un proceso. Todos los hilos de un mismo proceso comparten el mismo espacio de memoria dinámica y otros recursos como acceso a archivos compartidos. Estas características permite una comunicación rápida entre hilos y agilización de procesos, aunque presentan desafíos como condiciones de carrera, escrituras sucias, sincronización y gestión de recursos compartidos. La biblioteca pthread contorola estos aspectos con herramientas, como mutexes, variables de condición, barreras y semaforos(GeeksforGeeks, 2025).

Funciones de pthread

- Gestión de hilos: Incluye funciones para crear (pthread_create), finalizar (pthread_exit), y esperar la finalización de hilos (pthread_join). También permite configurar atributos como la prioridad y el tamaño de la pila mediante estructuras de atributos (pthread_attr_t)(Staff, 2022).
- Sincronización: Para evitar condiciones de carrera y garantizar la coherencia de los datos compartidos, pthreads proporciona mecanismos como mutexes (pthread_mutex_t) y variables de condición (pthread_cond_t). Los mutexes aseguran que solo un hilo acceda a una sección crítica a la vez, mientras que las variables de condición permiten que los hilos esperen y se notifiquen entre sí sobre ciertos eventos(Staff, 2022).

- Datos específicos de hilo: Permite asociar datos únicos a cada hilo mediante claves (pthread_key_t), lo que es útil para mantener información específica de cada hilo sin interferencia entre ellos(Staff, 2022).

Beneficios y necesidades

El uso de pthreads es mejora las implementaciones de aplicaciones que requieren realizar múltiples tareas simultáneamente, como servidores web, aplicaciones multimedia o programas científicos que realizan cálculos paralelos. Al aprovechar múltiples núcleos de procesador, las aplicaciones multihilo pueden mejorar significativamente su rendimiento y capacidad de respuesta.

Sinembargo, la programación con hilos también cuenta con desafíos, como la necesidad de una sincronización adecuada para evitar errores difíciles de detectar, como condiciones de carrera o deathlock. Por ello, es fundamental comprender correctamente los mecanismos de sincronización.

2.2. Función fork

La función **fork()** , es una llamada al sistema que pertenece a la biblioteca estándar de C (libc) que permite la creación de un nuevo proceso en sistemas operativos de tipo UNIX. Esta función genera un proceso hijo duplicando el proceso que realiza la llamada, conocido como el proceso padre. (fork, s.f.)

Cuando se llama a la función **fork()** , el sistema crea una copia del espacio de memoria del proceso padre. Inicialmente, ambos procesos contienen la misma información; sin embargo, cada uno opera en su propio espacio de direcciones de memoria, significando que los cambios realizados en uno no afecta al otro. (fork, s.f.)

Algunos aspectos importantes del procesos padre e hijo son:(fork, s.f.)

- El proceso hijo recibe un identificador de proceso (PID) único.
- El proceso hijo no hereda ciertos recursos del sistema, como los bloqueos de memoria, contadores de uso de CPU, señales pendientes, operaciones de entrada/salida asíncronas, ni temporizadores.
- Ambos procesos comparten los descriptores de archivos abiertos, lo que significa que apuntan a las mismas descripciones de archivo a nivel del sistema.
- En sistemas multihilo, solo el hilo que llamó a fork() se replica en el hijo.

Valor de retorno: El valor retorno es decisivo para definir el comportamiento del código posterior a la llamada en la que:

- El proceso padre, retorna el PID (Process ID) del proceso hijo
- El proceso hijo, retorna un cero (0)

- En caso de que exista un error, retorna el valor -1 y no se crea ningún proceso

Esta función es fundamental en la creación de procesos para los sistemas UNIX y es ampliamente utilizada como base para ejecutar nuevos programas mediante su combinación de otras llamadas al sistema.

Capítulo 3

Estrategias de paralelización

3.1. Paralelización con pthread en descompresión

Hilos por archivo

Para el desarrollo de este algoritmo se decidio crear un hilo independiente por cada archivo contenido dentro del archivo comprimido principal. Cada hilo recibe como argumento una estructura que contiene toda la información necesaria para realizar su tarea, incluyendo el árbol de Huffman, la ubicación dentro del archivo comprimido y el nombre del archivo de salida.

```
1 pthread_t *threads = malloc(sizeof(*threads) * fileCount);
2 for (int i = 0; i < fileCount; i++) {
3     pthread_create(&threads[i], NULL, decompressThread, tasks[i]);
4 }
5 for (int i = 0; i < fileCount; i++) {
6     pthread_join(threads[i], NULL);
7 }
```

La sicncirnización fue diseñada de modo que el hilo principal espera a que todos los hilos terminen usando pthread_join, asegurando que todo esté listo antes de finalizar y liberar recursos.

Diseño sin corrupción de datos

El diseño del programa evita conflictos de acceso compartido al usar copias independientes de recursos potencialmente problemáticos. Cada hilo abre su propia instancia del archivo comprimido para lectura, evitando así problemas de acceso concurrente al mismo descriptor de archivo. Además, el árbol de Huffman, que es necesario para la decodificación, se comparte entre hilos, pero solo para lectura.

```
1  void *decompressThread(void *arg) {
2      ThreadArg *t = (ThreadArg *)arg;
3      FILE *fi = fopen(t->compressFileName, "rb");
4      if (!fi) { perror("Error opening compressed file"); free(t);
           return NULL; }
5      unsigned char *outBuf = malloc(t->symbolCount);
6      if (!outBuf) { perror("Buffer allocation error"); fclose(fi);
           free(t); return NULL; }
7      processSegment(fi, t->tree, t->symbolCount, t->offsetBytes,
           outBuf, 1);
8      free(outBuf);
9      fclose(fi);
10     free(t->filePath);
11     free(t);
12     return NULL;
13 }
```

3.2. Paralelización con pthread en compresión

Conteo de frecuancias

El algoritmo aprovecha el paralelismo a nivel de archivo para acelerar el proceso de conteo de frecuencias de símbolos, una etapa esencial en la construcción del árbol de Huffman. Para ello, crea un hilo independiente por cada archivo detectado en el directorio, aprovechando el conteo de frecuencias. Cada hilo lee su archivo y construye una tabla local de frecuencias.

```
1  pthread_t *threads = malloc(file_count * sizeof(pthread_t));
2  for (int i = 0; i < file_count; ++i)
3      pthread_create(&threads[i], NULL, thread_count, &files[i]);
```

Paralelismo por archivo

Una vez generado el árbol de Huffman y la tabla de codificación, el programa genera un hilo por archivo, esta vez para realizar la compresión. Cada hilo accede a su archivo original y lo recorre elemento por elemento, sustituyéndolos por códigos binarios. Esta compresión se realiza en memoria, almacenando el resultado . La tabla de codificación es compartida entre todos los hilos, pero solo se accede en modo lectura lo cual evita cualquier escritura sucia.

```
1  for (int i = 0; i < file_count; ++i)
2      pthread_create(&threads[i], NULL, thread_compress, &files[i]);
```

3.3. Paralelización con fork

La paralelización por medio de procesos (uso del fork) consistió en la creación de 2 programas por separado **(compress.c y decompress.c)**.

La estrategia consistió en que, por cada archivo detectado en el directorio proporcionado en los parámetros del programa, el proceso principal (padre) genera un proceso hijo. En la que cada proceso hijo es responsable exclusivamente

de comprimir un único archivo. De esta manera, se evita que solo un proceso se convierta en un cuello de botella al procesar archivos de forma secuencial.

Cada proceso hijo, procesa la función **compressFile**, en la cual lee el archivo original, obtiene la codificación correspondiente a cada símbolo utilizado en una tabla de codificación previamente construida, y escribe el resultado comprimido en un archivo temporal exclusivo. Como una estrategia adicional, el nombre de cada archivo temporal es generado de forma única utilizando el índice del archivo procesado, y una vez finalizada su tarea, el proceso hijo termina su ejecución mediante **exit(0)**.

Mientras tanto, el proceso padre, continúa iterando sobre los archivos restantes y creando nuevos procesos hijos hasta haber asignado una a cada archivo existente. A medida que se crean los procesos hijos, sus PIDs son almacenados en un arreglo dinámico, en lo cual permite que el proceso padre espera su finalización de forma controlada mediante la función **waitpid(...)**

Una vez que todos los procesos hijos han concluido, el proceso padre procede a agrupar todos los archivos temporales que los hijos crearon. Esto sucede mediante la función **appendTempFiles**, que concatena el contenido de todos los archivos comprimidos temporales en un único archivo de salida y posteriormente elimina los archivos intermedios.

Capítulo 4

Descripción del proyecto

El presente proyecto consiste en una implementación del algoritmo de Huffman para la compresión y descompresión de archivos de texto en el lenguaje de programación C ejecutado a través del sistema operativo Fedora Workstation 41.

4.1. Estructura del proyecto

```
Proyecto-1-Operativos/
├── install.sh ..... Instalación de dependencias
├── Libros/ ..... Todos los libros
├── Fork/ ..... Implementación en fork
│   ├── compress.c
│   ├── decompress.c
│   └── Makefile ..... Compilación automática
├── Hilos/ ..... Implementación en pthreads
│   ├── compress.c
│   ├── decompress.c
│   └── Makefile ..... Compilación automática
└── Serial/ ..... Implementación secuencial
    ├── compress.c
    ├── decompress.c
    └── Makefile ..... Compilación automática
```

4.2. Descarga e instalación de dependencias

Descargue el proyecto que está alojado en github por medio de este enlace: `https://github.com/Willvillegas/Proyecto-1-Operativos` o clone el repositorio por medio del siguiente comando:

```
1    git clone https://github.com/Willvillegas/Proyecto-1-Operativos
```

Luego ejecuta el comando en la terminal **chmod +x install.sh** y seguido de este otro comando **./install.sh** dentro del directorio del proyecto para instalar make y gcc elementos importantes para la compilación del código fuente.

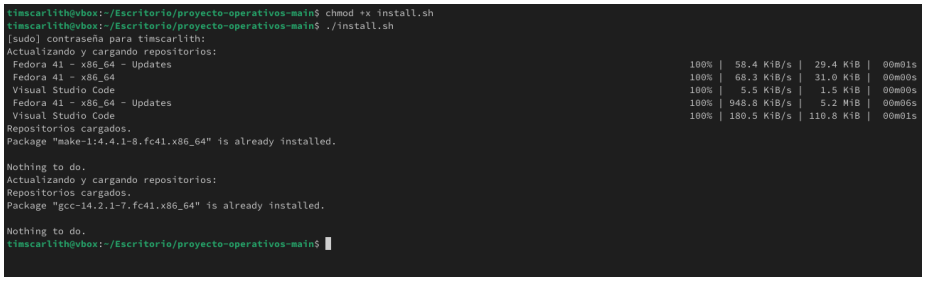

Figura 4.1: Ejecución de dependencias

4.3. Compilación y Ejecución

4.3.1. Compilación

A continuación, puede acceder a cualquiera de las carpetas de la implementación que desea utilizar (Serial, Hilos y Fork) y ejecute el makefile por medio del siguiente comando:

```
1 make
```

después, el código genera un ejecutable con los siguientes nombres: **./compress** y **./decompress**

4.3.2. Ejecución

Para ejecutar cualquiera de los 2 archivos se debe de enviar por parámetros los siguientes comandos:

- **./compress** [**directorio a comprimir**] [**archivo de salida**]
  - Ejemplo: **./compress ../Libros salida.bin**
- **./decompress** [**archivo comprimido**] [**nombre de la carpeta**]
  - Ejemplo: **./decompress salida.bin CarpetaCompresa**

Capítulo 5

Discusión

5.1. Resultado de los multiples experimentos

Los resultados arrojados por los diferentes métodos de compresión son evaluados en segundos y todos los programas se encargaron de evaluar los 99 archivos de extención .txt que fueran optenidos de el enlace del proyecto Gutenberg,(`https://www.gutenberg.org/browse/scores/top#books-last30`).

Implementación Serial

- Tiempo de compresión: 22.862834 segundos
- Tiempo de descompresión: 4.864770 segundos

Implementación con hilos

- Tiempo de compresión:3.298318 segundos
- Tiempo de descompresión: 5.063939 segundos

Implementación con fork

- Tiempo de compresión: 15.082390 segundos
- Tiempo de descompresión: 1.398543 segundos

Podemos analiza que en el procesos de compresión de estos archivos, el algoritmo con la implementación de hilos, es el que realiza la tarea más rápida con una diferencia de 19,564516 segundos más rápido, con respecto al algoritmo con la implementación serial que fue el más lento. Por otra parte, para el algoritmo de descompresión la implementación fork fue la más rápida con una diferencia de 3,665387 segundos más rápido, con respecto al algoritmo que implementa hilos que fue el más lento para este caso particular.

Capítulo 6

Conclusión

Tras evaluar las tres estrategias de paralelización —implementación secuencial, con hilos (pthread) y con procesos (fork)— en las fases de compresión y descompresión de archivos mediante el algoritmo de Huffman, resulta evidente que cada enfoque presenta fortalezas y limitaciones según la naturaleza de la tarea. A partir de los datos obtenidos y del análisis de diseño, se derivan las siguientes conclusiones de nivel universitario:

Las hilos (pthread) ofrecen el mayor beneficio en la etapa de compresión

Al comparar los tiempos de compresión, la implementación con hilos redujo el tiempo desde 22,86 s (serial) hasta 3,30 s, logrando un speedup cercano a 7×. Esto refleja que la sobrecarga de creación y sincronización de hilos es relativamente baja frente al trabajo de procesado de datos, y que compartir estructuras de sólo lectura (árbol de Huffman y tabla de códigos) minimiza los costes de sincronización.

El uso de procesos (fork) es especialmente eficaz para la descompresión

Mientras que la descompresión serial tomó 4,86 s y con hilos 5,06 s, la versión con fork completó la tarea en sólo 1,40 s, consiguiendo un speedup de más de 3× respecto a la serial. Esto sugiere que, para la descompresión, el aislamiento de cada proceso (y posiblemente la menor contención de E/S sobre descriptores de archivo separados) compensa con creces el coste adicional de fork.

No existe "la mejor" estrategia de paralelización universal, sino que depende de la fase del algoritmo

Para compresión, donde el cuello de botella es el recorrido y codificación en memoria de grandes volúmenes de símbolos, el paralelismo ligero de los hilos es más adecuado.

Y por otro lado la descompresión, donde cada subárbol se recorre de forma independiente y la E/S domina, el aislamiento de procesos reduce la contención y mejora el rendimiento. Esta diferenciación fundamenta la recomendación de elegir la técnica de paralelización en función de las características específicas de cada etapa del pipeline de Huffman.

La arquitectura compartida de memoria vs. la independencia de procesos influye en la escalabilidad

El enfoque con hilos aprovecha eficientemente la memoria compartida para estructuras de sólo lectura, logrando escalar casi linealmente con el número de archivos. En cambio, la independencia de procesos evita puntos calientes de acceso concurrente pero incurre en mayor coste de creación; aun así, es más escalable para operaciones I/O-intensivas como descompresión.

Potencial para diseñar soluciones híbridas

Dado que cada método brilla en una fase distinta, un diseño híbrido que use hilos para compresión y procesos para descompresión podría combinar lo mejor de ambos mundos, maximizando el rendimiento en aplicaciones reales donde ambas etapas se encadenan.

Referencias

Blancarte, O. (2017). *Concurrencia vs paralelismo.* Descargado de https://www.oscarblancarteblog.com/2017/03/29/concurrencia-vs-paralelismo/

Contributors, W. (2024, 28 de Noviembre). *Compresión de imagen.* Wikipedia. Descargado de https://es.wikipedia.org/wiki/Compresin_de_imagen?

Contributors, W. (2025a, 5 de Abril). *Data compression.* Wikipedia. Descargado de en.wikipedia.org/wiki/Data_compression?

Contributors, W. (2025b, 2 de Marzo). *Lossless compression.* Wikipedia. Descargado de https://en.wikipedia.org/wiki/Lossless_compression?

Contributors, W. (2025c, 6 de Marzo). *Project gutenberg.* Wikipedia. Descargado de https://en.wikipedia.org/wiki/Project_Gutenberg?

Contributors, W. (2025d, 6 de Enero). *Proyecto gutenberg.* Wikipedia. Descargado de https://es.wikipedia.org/wiki/Proyecto_Gutenberg?

Fernández, G. F. (s.f.). *Compresión con pérdidas (lossy).* Publicación de un blog. Descargado de https://1library.co/article/compresi%C3%B3n-p%C3%A9rdidas-lossy-elementos-sistemas-operativos-representaci%C3%B3n-inf.zgl53e7q

fork(2) - linux manual page. (s.f.). https://man7.org/linux/man-pages/man2/fork.2.html. (Accessed: 2025-4-24)

GeeksforGeeks. (2025). *Multithreading in c.* https://www.geeksforgeeks.org/multithreading-in-c/.

Google Search. (2024). *Búsqueda: algoritmo de huffman reseña.* https://www.google.com/search?q=algoritmo+de+Huffman+rese%C3%B1a.

Gutenberg, P. (s.f.-a). *About project gutenberg.* Publicación de un blog. Descargado de https://www.gutenberg.org/about/

Gutenberg, P. (s.f.-b). *Project gutenberg.* Publicación de un blog. Descargado de https://www.gutenberg.org/about/background/history_and_philosophy.html?

Staff, C. . C. (2022). *Pthread library - cs 162 project 2.* https://cs162.org/static/proj/proj-threads/docs/tasks/pthread/.

Wikipedia contributors. (2024). *Algoritmo de huffman — wikipedia, la enciclopedia libre.* Descargado de https://es.wikipedia.org/wiki/Algoritmo_de_Huffman