TEC | Tecnológico de Costa Rica

Escuela de Ingeniería en Computación
411 Ingeniería en Computación - 2018
IC6600 - Principios de Sistemas Operativos

# Tarea 2

Biblioteca Sync

Ávila Ramírez Paublo Alexander
pavila@estudiantec.cr
2022035584

Reyes Rodríguez Ricardo Andrés
rireyes@estudiantec.cr
2022101681

Zúñiga Campos Omar Jesús
omzuniga@estudiantec.cr
2022019053

San José, Costa Rica
Setiembre 2024

# Índice general

# Capítulo 1

# Introducción

El Proyecto Gutenberg es una biblioteca virtual la cual tuvo inicio en 1971 que fue creada con el fin de digitalizar y archivar distintas obras culturales con el fin de que estuvieran al alcance de cualquier persona con acceso a Internet. Este movimiento fue fundado por Michael Hart con la idea de distribuir libros electrónicos de dominio público y esa idea al día de hoy se volvió en una biblioteca virtual con una gran cantidad de textos históricos y literarios de diversas categorías como es el caso de filosofía, ciencia, literatura clásica y muchos más géneros de tal forma que ha logrado brindar un mejor acceso al conocimiento alrededor del mundo (Hart, 1992).

El caso de una biblioteca virtual como es el caso del Proyecto Gutenberg resulta en un sitio con una amplía cantidad de archivos, lo que se traduce en una gran cantidad de datos, siendo así que el uso de algoritmos de compresión son bastante precisos para entender el funcionamiento de un sitio como este además de que han jugado un papel fundamental en las áreas de la computación y ciencias de datos. Como tal, hay dos tipos principales de compresión partiendo los que no presentan pérdidas que son capaces de realizar la reconstrucción de los datos originales dando como resultado los originales de forma exacta una vez descomprimidos, de tal forma que no se pierden datos, donde un ejemplo es el algoritmo de Huffman, el cual emplea árboles binarios con el fin de asignar códigos cortos a los caracteres más frecuentes de tal forma que optimiza el espacio requerido para el almacenamiento de la información (Molina, 2018). Por otra parte, los algoritmos de compresión con pérdida como es el caso de imágenes JPEG sacrifican una parte de la precisión de los datos originales con el fin de lograr una compresión más alta debido a que estas son aplicaciones que no requieren una recuperación exacta de los datos originales (Powell, 2024).

Es importante resaltar que en ciertas implementaciones de estos algoritmos de compresión, muchas veces se incluyen conceptos como son el caso del paralelismo y concurrencia pues son bastante útiles para la optimización de este tipo de algoritmos para realizar varias tareas donde la concurrencia utiliza la capaci-

dad de un sistema para manejar varias tareas de forma intercalada mientras que el paralelismo es una ejecución simultánea de varias tareas en distintos núcleos de procesamiento donde los dos enfoques son capaces de mejorar la eficiencia y tiempos de respuesta en sistemas con una gran cantidad de datos o necesitan realizar varios procesos de forma simultánea (Won, 2021).

## 1.1. Antecedentes

El Proyecto Gutenberg fue fundado en 1971 por Michael S. Hart, el cual haciendo uso del Laboratorio de Investigación de Materiales de la Universidad de Illinois para la creación del primer libro digital habilitado al público que se trató ni nada más ni nada menos que la Declaración de Independencia de los Estados Unidos, que fue hecho con el deseo de Hart para que las personas fueran capaces de acceder de manera libre y gratuita al conocimiento a través del internet, lo cual fue una revolución en la forma que se distribuyen y consumen los textos. Conforme avanzó el tiempo, el Proyecto Gutenberg fue creciendo al punto que es una de las bibliotecas digitales más grandes del mundo, contando con más de 70,000 libros de forma gratuita al mismo tiempo que este proyecto fue la inspiración de otras plataformas y archivos de dominio público (Hart, 1992).

Es evidente que el proyecto Gutenberg fue un antes y un después, tal y como fue el caso del algoritmo de Huffman que hizo posible que los textos en forma digital pudieran comprimirse sin pérdida. Este algoritmo fue propuesto por David A. Huffman en 1952 y es parte de los algoritmos de compresión sin pérdida que hace uso de la frecuencia de aparición de los símbolos para construir un árbol binario y asignar códigos cortos a los símbolos con mayor frecuencia y códigos largos a los que aparecen menos, resultando así en una compresión bastante significativa sin que haya pérdida de información. Esto fue fundamental para que surgieran sistemas de compresión que siguen siendo bastante utilizados como es el caso de ZIP en 1989 de la mano de Phil Katz o GZIP en 1992 como parte del proyecto GNU (Molina, 2018).

Como tal, los algoritmos de compresión sin pérdida han estado siendo utilizados desde mediados del siglo XX, siendo uno de los más conocidos y relevantes del Huffman, creado en 1952. Sin embargo, también están los algoritmos con cierta pérdida que su aplicación es debido a medios digitales donde es tolerable cierta pérdida de información, como es el caso del formato MP3 que fue lanzado en 1993 que usa la compresión de audio porcentual, que fue desarrollado principalmente por el Fraunhofer Institute en Alemania. Otro ejemplo es la compresión de vídeo como es el caso del MPEG-1, que fue publicado en 1993 por el grupo Moving Pictures Experts Group (MPEG), que a su vez establecieron estándares como es el caso de MPEG-2 y MPE-G4 en los años siguientes, permitiendo así que haya una optimización de la forma en que se almacenan los medios audiovisuales (Powell, 2024).

Finalmente, muy de la mano con ese concepto de optimizar recursos y sacar el máximo provecho, se encuentran los conceptos de paralelismo y concurrencia, que han sido clave desde aproximadamente los años 1960 y adquirieron mucho más poder con el desarrollo de procesadores multinúcleos capaces de soportar paralelismo, haciendo que el concepto adquiriera más popularidad en los años 2000, sobre todo por los procesadores multinúcleo como es el caso del Intel Core Duo en 2006. En cambio, la concurrencia a la hora de diseñar y programar software tiene sus orígenes con los trabajos de Edsger W. Dijkstra en los años 1960, cuando se encontraba investigando los primeros mecanismos para el control de concurrencia para sistemas operativos y el desarrollo de lenguajes de programación concurrentes como el caso de Communicating Sequential Processes, que fue introducido por Tony Hoare en 1978 (Tanenbaum y García, 2003).

# Capítulo 2

# Desarrollo

## 2.1. Funcionamiento de fork

La función `fork()` consiste en una llamada al sistema en entornos Unix, como es el caso de Linux y es empleada con el propósito de crear nuevos procesos. En el caso de C, se encarga de dividir un proceso en dos: El proceso padre y el proceso hijo, donde el proceso hijo es básicamente una copia del padre pero tiene un espacio de direcciones independiente, por lo que posee sus propios recursos aunque comparten el mismo contenido de memoria en un inicio. ADemás, ambos procesos continúan su ejecución a partir del punto donde se llamó a `fork()` con la diferencia que la función devuelve valores distintos en cada proceso, lo que es muy importante tener en cuenta a la hora de realizar la coordinación entre ellos (Tanenbaum y García, 2003).

En concreto, `fork()` retorna un valor de 0 en el proceso hijo y el Process ID mejor conocido como PID del proceso hijo en el proceso padre. En caso de que llegue a fallar la creación del proceso hijo, `fork()` devuelve -1 para indicar un error. Todo esto brinda una forma de diferenciar entre proceso padre e hijo en base al valor de retorno para que puedan realizar tareas específicas (Hu, 2023).

Un aspecto muy importante de `fork()` es su capacidad para separar la memoria entre un proceso padre y los procesos hijos pues aunque ambos procesos comparten en un inicio las mismas páginas de memoria, que se le conoce como *copy-on-write*, cualquier modificación en la memoria por cualquiera de los procesos produce que se haga una copia de las páginas afectadas, de tal forma que los cambios en un proceso no afectan los del otro. Esto permite que haya una independencia de los procesos en cuanto a la gestión de memoria a la vez que aprovecha la eficiencia al no hacer una duplicación inmediata de toda la memoria (Hu, 2023).

Si bien existe una diferencia entre el proceso padre y el hijo, este último se

encarga de heredar varios aspectos del proceso padre, como es el caso de los descriptores de archivos abiertos la máscara de señales, variables de entorno y el directorio de trabajo actual pero no hereda propiedades como es el caso de identificadores de procesos de grupo, bloqueos de archivos y alarmas pendientes. Además, una vez llamado `fork()`, los procesos son capaces de interactuar de distintas maneras, como es el caso del proceso padre que es capaz de esperar a que el proceso hijo termine por medio de funciones como `wait()` o `waitpid()`, que permiten una sincronización básica (Tanenbaum y García, 2003).

En sí, el uso de `fork()` en sistemas de multiprocesamiento ha adquirido una gran relevancia pues cada proceso hijo se puede programar para que sea ejecutado en un núcleo distinto de la CPU aunque esto está sujeto en gran medida a las capacidades del sistema operativo y de hardware pero en cualquier caso da la posibilidad de que se ejecuten múltiples tareas de forma simultánea, lo que mejora el rendimiento cuando se trabaja en sistemas multitarea (Hu, 2023).

Pese a que esta es una herramienta muy poderosa, puede llegar a presentar errores por temas como e la falta de recursos en el sistema como es insuficiente memoria o superación del número máximo permitido en procesos del sistema, lo cual resulta que haya un valor de retorno -1. En sistemas que tienen una alta carga, el uso excesivo de `fork()` sin que se tenga un buen manejo de los procesos hijo puede ocasionar que haya un problema que se conoce como *fork bomb*, donde la creación de demasiados procesos saturan los recursos del sistema (Tanenbaum y García, 2003).

En el caso de C, se usa `fork()` en progarmas que necesitan de una división de trabajo entre procesos y para que esto pueda ser comprendido, se presenta un ejemplo de esto a continuación:

```
1  #include <stdio.h>
2  #include <unistd.h>
3  #include <sys/types.h>
4
5  int main() {
6      pid_t pid = fork();
7
8      if (pid == -1) {
9          // Error creando del proceso
10         perror("fork failed");
11         return 1;
12     } else if (pid == 0) {
13         // Proceso hijo
14         printf("Proceso hijo con PID: %d\n", getpid());
15     } else {
16         // Proceso padre
17         printf("Proceso padre con PID: %d y proceso hijo tiene PID:
               %d\n", getpid(), pid);
18     }
19
20     return 0;
21 }
```

En el ejemplo anterior, se muestra la forma en que el proceso padre e hijo imprimen sus identificaodres correspondientes que son los PID haciendo uso `getpid()` que deuelve el PID del proceso que lo está llamando, mientras que el valor que devuelve `fork()` es el que marca la diferencia entre el proceso padre y el proceso hijo.

El uso de la función `fork()` en C plantea importantes desafíos en términos de eficiencia y consumo de recursos. Aunque el mecanismo de copia diferida, conocido como *copy-on-write*, reduce significativamente el impacto de duplicar el espacio de memoria, la creación de un nuevo proceso sigue siendo más costosa que la de un hilo (*thread*). Esto se debe a que los hilos comparten el mismo espacio de direcciones y los recursos de su proceso padre, mientras que los procesos creados con `fork()` son completamente independientes, lo que implica una mayor carga para el sistema. No obstante, esta independencia entre procesos puede resultar beneficiosa en cuanto a seguridad y estabilidad (Hu, 2023).

En los sistemas operativos modernos, se suele preferir combinar `fork()` con una llamada a `exec()`, como `execve()` o `execl()`, para el reemplazo de la imagen del proceso hijo con la ejecución de un nuevo programa lo cual es muy común en los *shells* de sistemas operativos, donde se emplea `fork()` para la creación de un nuevo proceso y luego se usa `exec()` para que el proceso hijo ejecute un programa diferente, mientras que el proceso padre permanece sin cambios (Tanenbaum y García, 2003).

En conclusión, `fork()` es una herramienta esencial para la creación de procesos en sistemas Unix. A través de su capacidad para diferenciar entre procesos padre e hijo mediante el valor de retorno y su manejo separado de la memoria, permite que cada proceso pueda ejecutar tareas distintas. Aunque tiene limitaciones en cuanto al uso de recursos, su combinación con otras llamadas al sistema, como `exec()`, le otorga flexibilidad para gestionar el paralelismo y la concurrencia en los sistemas multitarea (Tanenbaum y García, 2003).

## 2.2. Funcionamiento de la biblioteca pthreads

La biblioteca `pthread` ofrece una interfaz estándar para la creación y gestión de hilos en sistemas compatibles con POSIX, como Linux y otros sistemas Unix. Estos hilos, a diferencia de los procesos tradicionales, son unidades de ejecución más livianas que comparten el mismo espacio de memoria y otros recursos del proceso que los generó. Esto facilita una comunicación y sincronización más eficiente. Por lo tanto, es especialmente útil para aplicaciones que necesitan ejecutar múltiples tareas de manera simultánea sin el elevado costo asociado con la creación de procesos completos. (Tanenbaum y García, 2003).

Como tal, un hilo es una secuencia de instrucciones las cuales son ejecutadas

de forma independiente dentro del contexto de un proceso y aunque los hilos comparten recursos como la memoria, los descriptores de archivos y las variables globales, cada hilo posee su propia pila (*stack*), conjunto de registros y contador de programa, lo que hace que pueda funcionar de manera autónoma y para facilitar el uso de esta herramienta, La biblioteca `pthread` proporciona una serie de funciones para crear, controlar y sincronizar hilos dentro de un mismo proceso. En el lenguaje C, se accede a esta funcionalidad mediante la inclusión de la cabecera `pthread.h`, y todas las funciones de la biblioteca empiezan con el prefijo `pthread_` (GeeksforGeeks, 2023b).

La función más básica que ofrece `pthread` es `pthread_create()`, la cual permite generar un nuevo hilo (GeeksforGeeks, 2023b). Su prototipo es el siguiente:

```
1  int pthread_create(pthread_t *tid, const pthread_attr_t *attr, void
       *(*func)(void *), void *arg);
```

En la declaración de esta función, `tid` es un puntero a una variable de tipo `pthread_t` donde se almacenará el identificador del hilo creado la cual tiene el parámetro `attr` que es un puntero a una estructura `pthread_attr_t`, que permite especificar los atributos del hilo, aunque si se pasa `NULL`, se utilizarán los valores predeterminados. Por otro lado, `func` es un puntero a la función que el hilo ejecutará y `arg` es un puntero a los datos que se pasarán a dicha función (GeeksforGeeks, 2023b).

Por ejemplo, en un programa simple que imprime un mensaje desde un hilo, la implementación se vería de la siguiente manera:

```
1  #include <stdio.h>
2  #include <pthread.h>
3
4  void *show_message(void *param) {
5      printf("Mensaje desde el hilo\n");
6      return NULL;
7  }
8
9  int main() {
10     pthread_t hilo;
11
12     pthread_create(&hilo, NULL, show_message, NULL);
13     pthread_join(hilo, NULL);
14
15     return 0;
16 }
```

En este caso, se crea un hilo que ejecuta la función `show_message`. Luego de crearlo, el programa principal espera que el hilo termine con la función `pthread_join()` (GeeksforGeeks, 2023b).

Un reto bastante frecuente a la hoar de utilizar hilos es la sincronización, ya que estos pueden ejecutarse de forma concurrente y acceder a los mismos recursos, lo que puede ocasionar condiciones de carrera (*race conditions*) y otros problemas relacionados con la consistencia de los datos. Para mitigar estos riesgos,

la biblioteca `pthread` incluye varias herramientas como los *mutexes* (bloqueos mutuos) y variables de condición (Basanta, 2010).

Un *mutex* es un mecanismo que asegura que solo un hilo tenga acceso a un recurso compartido a la vez. El hilo debe bloquear el *mutex* antes de utilizar el recurso y desbloquearlo cuando haya terminado. Mientras el *mutex* esté bloqueado, otros hilos que intenten acceder al recurso quedarán en espera (Tanenbaum y García, 2003). Un ejemplo básico de uso de *mutex* es el siguiente:

```
1  #include <stdio.h>
2  #include <pthread.h>
3
4  pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
5
6  void *critical_section(void *param) {
7      pthread_mutex_lock(&lock);
8      printf("Accediendo a una secci n cr tica\n");
9      pthread_mutex_unlock(&lock);
10     return NULL;
11 }
12
13 int main() {
14     pthread_t t1, t2;
15
16     pthread_create(&t1, NULL, critical_section, NULL);
17     pthread_create(&t2, NULL, critical_section, NULL);
18
19     pthread_join(t1, NULL);
20     pthread_join(t2, NULL);
21
22     return 0;
23 }
```

En este ejemplo, dos hilos intentan acceder a la misma sección crítica de código, pero gracias al *mutex*, solo uno puede hacerlo a la vez (Tanenbaum y García, 2003).

Las variables de condición son otro mecanismo que permite a los hilos esperar hasta que una condición particular se cumpla antes de continuar su ejecución. Su uso generalmente requiere un *mutex* para evitar condiciones de carrera (Tanenbaum y García, 2003). Un ejemplo de uso de una variable de condición es el siguiente:

```
1  #include <stdio.h>
2  #include <pthread.h>
3
4  pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
5  pthread_cond_t condition = PTHREAD_COND_INITIALIZER;
6  int ready = 0;
7
8  void *wait_for_signal(void *param) {
9      pthread_mutex_lock(&lock);
10     while (!ready) {
11         pthread_cond_wait(&condition, &lock);
12     }
```

```
13      printf("Hilo recibi  la se al\n");
14      pthread_mutex_unlock(&lock);
15      return NULL;
16  }
17
18  void *send_signal(void *param) {
19      pthread_mutex_lock(&lock);
20      ready = 1;
21      pthread_cond_signal(&condition);
22      pthread_mutex_unlock(&lock);
23      return NULL;
24  }
25
26  int main() {
27      pthread_t t1, t2;
28
29      pthread_create(&t1, NULL, wait_for_signal, NULL);
30      pthread_create(&t2, NULL, send_signal, NULL);
31
32      pthread_join(t1, NULL);
33      pthread_join(t2, NULL);
34
35      return 0;
36  }
```

En este ejemplo, un hilo espera hasta que otro envía una señal, utilizando la variable de condición para coordinar la ejecución (Tanenbaum y García, 2003).

Además de los *mutexes* y las variables de condición, también existen bloqueos de lectura/escritura (`pthread_rwlock_t`) y semáforos, los cuales ayudan a limitar el número de hilos que pueden acceder simultáneamente a un recurso. Por otra parte, para detener la ejecución de un hilo, se emplea `pthread_exit()`. Es importante asegurarse de que todos los hilos finalicen de la manerra correcta antes de que el proceso principal termine, ya que si este finaliza, todos los hilos que siguen ejecutándose se cerrarán abruptamente (Basanta, 2010).

Por lo tanto, `pthread` es una poderosa herramienta para implementar concurrencia en C, ya que brinda métodos efectivos para la creación y sincronización de hilos que comparten recursos comunes y pese a que la programación multihilo puede ser compleja, la biblioteca proporciona los mecanismos necesarios para manejarla de forma eficiente y sin mucho problema (Tanenbaum y García, 2003).

## 2.3. Estrategia para paralelizar los programas en cada una de las versiones

Este proyecto implementa el algoritmo de compresión y descompresión de archivos de texto utilizando el algoritmo de Huffman. El proyecto incluye tres versiones de la implementación: serial, utilizando `fork()` para crear múltiples procesos y utilizando la biblioteca `pthread` para crear múltiples hilos. A conti-

nuación se describen las estrategias utilizadas para la paralelización y el proceso de descompresión en cada una de las implementaciones.

### 2.3.1. Implementación serial

La implementación serial del algoritmo de Huffman sigue los pasos clásicos del algoritmo sin ningún tipo de paralelización. El proceso sigue el flujo secuencial siguiente:

**Construcción del Árbol de Huffman:** Para la compresión, el árbol de Huffman se construye utilizando las frecuencias de los caracteres en el archivo de entrada. El proceso de construcción sigue estos pasos:

1. Se cuenta la frecuencia de cada carácter en el archivo.
2. Se inserta cada carácter en una cola de prioridad junto con su frecuencia.
3. Los dos nodos de menor frecuencia se extraen de la cola y se crea un nuevo nodo que los une con una suma de sus frecuencias.
4. Este nuevo nodo se inserta nuevamente en la cola de prioridad.
5. El proceso se repite hasta que solo queda un nodo, que será la raíz del árbol de Huffman.

Las funciones clave utilizadas en esta implementación incluyen:

- `build_huffman_tree()`: Esta función construye el árbol de Huffman utilizando un heap para mantener los nodos organizados por frecuencia.
- `generate_huffman_codes()`: Una vez construido el árbol, esta función genera los códigos binarios para cada carácter.
- `compress_file_serial()`: Función que comprime el archivo usando los códigos generados.

**Descompresión en la Versión Serial:** La descompresión comienza leyendo el archivo comprimido y reconstruyendo el árbol de Huffman. Esto se hace utilizando la tabla de códigos de Huffman almacenada en el archivo comprimido. A continuación, el archivo comprimido es recorrido bit a bit, siguiendo el árbol de Huffman para decodificar cada secuencia de bits en el carácter correspondiente.

Las funciones clave en la descompresión incluyen:

- `rebuild_huffman_tree()`: Reconstruye el árbol de Huffman a partir de la información almacenada en el archivo comprimido.
- `decompress_file_serial()`: Realiza la lectura del archivo comprimido y lo descomprime bit por bit utilizando el árbol reconstruido.

En esta implementación, todas las operaciones se realizan en un único hilo de ejecución, lo que limita la capacidad del programa para aprovechar los recursos de procesamiento paralelo disponibles en sistemas multi-núcleo.

### 2.3.2. Implementación usando `fork()`

Para aprovechar el paralelismo, se utilizó la función `fork()` para crear múltiples procesos que trabajen en paralelo. En esta implementación, el archivo de entrada se divide en múltiples secciones, y cada proceso hijo se encarga de comprimir una de esas secciones de forma independiente. La estructura general de la función que implementa este comportamiento es la siguiente:

```
void compress_files_fork(const char* input_dir, const char* output_path) {
    pid_t pids[NUM_PROCESSES];
    for (int i = 0; i < NUM_PROCESSES; i++) {
        pids[i] = fork();
        if (pids[i] == 0) {
            // Proceso hijo: comprime una parte del archivo.
            compress_part_of_file(input_dir, i);
            exit(0);
        }
    }
    for (int i = 0; i < NUM_PROCESSES; i++) {
        waitpid(pids[i], NULL, 0);
    }
}
```

En esta implementación:

- El proceso principal divide el archivo de entrada en `NUM_PROCESSES` partes iguales.
- Cada proceso hijo lee su parte del archivo, calcula las frecuencias de los caracteres para su parte, genera el árbol de Huffman correspondiente, y comprime su sección.
- Al finalizar, el proceso principal utiliza `waitpid()` para esperar que todos los procesos hijos terminen.
- Los resultados se almacenan en archivos temporales, que luego son concatenados por el proceso principal para crear el archivo comprimido final.

**Descompresión en la Versión con `fork()`:** En esta versión, el archivo comprimido se divide en bloques, y cada proceso hijo se encarga de descomprimir uno de esos bloques. El proceso padre crea varios procesos hijos usando `fork()` y cada uno de ellos reconstruye una porción del árbol de Huffman para su sección del archivo. Una vez que todos los procesos han terminado, el proceso padre los sincroniza y reconstruye el archivo descomprimido completo.

El uso de `fork()` implica que cada proceso hijo trabaja en su propio espacio de memoria, por lo que es necesario manejar los archivos de salida de manera que se unan al final. Las funciones clave utilizadas aquí incluyen:

- `decompress_file_fork()`: Cada proceso hijo ejecuta esta función para descomprimir su parte del archivo utilizando su propio árbol de Huffman.

- `wait()`: El proceso padre utiliza esta función para esperar a que todos los procesos hijos terminen su trabajo antes de ensamblar el archivo final.

Una de las ventajas de esta implementación es que cada proceso tiene su propio espacio de memoria, por lo que no es necesario preocuparse por la sincronización de recursos compartidos. Sin embargo, esto también implica un mayor consumo de memoria y un costo en la creación de procesos.

### 2.3.3. Implementación usando `pthread`

La tercera versión utiliza la biblioteca `pthread` para crear hilos de ejecución que trabajan de manera concurrente dentro del mismo espacio de memoria. El uso de hilos permite una mayor eficiencia en comparación con los procesos, ya que todos los hilos comparten el mismo espacio de direcciones, lo que facilita la comunicación y el acceso a datos compartidos.

El código principal para la compresión utilizando hilos es el siguiente:

```
void* thread_process(void* arg) {
    thread_data* data = (thread_data*)arg;
    // Comprime la parte del archivo asignada a este hilo.
    compress_file_segment(data->input_file, data->start, data->end);
    return NULL;
}

void compress_files_threads(const char* input_dir, const char* output_path) {
    pthread_t threads[NUM_THREADS];
    thread_data thread_args[NUM_THREADS];

    for (int i = 0; i < NUM_THREADS; i++) {
        thread_args[i].input_file = fopen(input_dir, "rb");
        thread_args[i].start = ...;  // Asignar segmento del archivo
        thread_args[i].end = ...;    // Asignar segmento del archivo
        pthread_create(&threads[i], NULL, thread_process, (void*)&thread_args[i]);
    }

    for (int i = 0; i < NUM_THREADS; i++) {
        pthread_join(threads[i], NULL);
    }
}
```

En esta implementación:

- Primero a cada archivo de entrada se le asigna un hilo que se va a encargar de su descrompesión
- A partir del hilo que se le da a cada archivo se crean múltiples hilos utilizando `pthread_create()`, donde cada hilo recibe una sección del archivo de entrada para comprimir.

- Cada hilo comprime su parte de manera concurrente utilizando la misma función de compresión que en las otras versiones.
- Una vez que todos los hilos han terminado su trabajo, el programa utiliza `pthread_join()` para asegurar que todos los hilos hayan completado antes de proceder a la siguiente etapa.

Al compartir el mismo espacio de direcciones, esta implementación requiere el uso de mecanismos de sincronización como mutexes para evitar condiciones de carrera cuando varios hilos intentan escribir al archivo comprimido simultáneamente. El siguiente fragmento de código muestra cómo se utiliza un mutex para sincronizar el acceso al archivo de salida:

```
pthread_mutex_t file_mutex = PTHREAD_MUTEX_INITIALIZER;

void* thread_process(void* arg) {
    thread_data* data = (thread_data*)arg;
    compress_file_segment(data->input_file, data->start, data->end);

    pthread_mutex_lock(&file_mutex);
    // Escribir al archivo de salida
    pthread_mutex_unlock(&file_mutex);

    return NULL;
}
```

**Descompresión en la Versión con `pthread`:** Para la descompresión, se dividen las tareas entre varios hilos, donde cada hilo es responsable de descomprimir una parte del archivo. Los hilos reconstruyen su porción del archivo utilizando un árbol de Huffman compartido, asegurándose de que cada uno trabaje en una sección distinta del archivo comprimido.

Se utilizan mutexes para asegurar que no haya condiciones de carrera al escribir en el archivo de salida. Las funciones clave en esta versión incluyen:

- `decompress_file_threads()`: Cada hilo ejecuta esta función para descomprimir su parte del archivo.
- `pthread_mutex_lock() y pthread_mutex_unlock()`: Estas funciones se utilizan para sincronizar la escritura al archivo de salida, asegurando que solo un hilo a la vez pueda escribir.
- `pthread_create()`: Esta función es utilizada para crear los hilos y asignarles la tarea de descomprimir partes específicas del archivo.
- `pthread_join()`: Se usa para esperar a que todos los hilos terminen antes de continuar con la ejecución del programa.

### 2.3.4. Comparación de las estrategias

Las tres implementaciones presentan diferentes características en cuanto a la paralelización:

- La versión serial es la más sencilla de implementar pero no explota la capacidad de procesamiento paralelo.
- La versión con `fork()` permite paralelizar la compresión, pero con un costo adicional en la creación de procesos y el uso de memoria separada para cada proceso.
- La versión con `pthread` es más eficiente en el uso de recursos, ya que los hilos comparten memoria y la comunicación entre ellos es más rápida. Sin embargo, requiere mayor cuidado en la sincronización de accesos a recursos compartidos.

Cada estrategia tiene sus propias ventajas y desventajas, y su elección depende de la arquitectura del sistema y los recursos disponibles.

### 2.3.5. Implementación del web crawler

En este proyecto, se utiliza un *web crawler* para descargar archivos de texto que serán procesados por el algoritmo de Huffman. El web crawler está diseñado específicamente para extraer libros del *Proyecto Gutenberg*, un repositorio de textos de dominio público. A continuación, se describe la implementación del *web crawler* y las funciones clave que lo componen.

#### Tecnologías Utilizadas

El web crawler hace uso de las siguientes tecnologías:

- **CURL**: Utilizado para realizar solicitudes HTTP y descargar contenido HTML desde el sitio web objetivo (ZenRows, 2024).
- **TidyLib**: Librería utilizada para limpiar y parsear el contenido HTML (TidyLib, 2017).
- **Regex**: Expresiones regulares para identificar enlaces relevantes y extraer contenido específico del HTML (ZenRows, 2024).

#### Funciones Principales

El web crawler está compuesto por varias funciones que permiten la descarga, procesamiento y almacenamiento de los libros de texto. A continuación, se explican las funciones más importantes:

**Función** `write_memory_callback`

Esta función se encarga de almacenar los datos descargados en memoria durante una transferencia con CURL. La función se define de la siguiente manera:

```
1 size_t write_memory_callback(void *contents, size_t size, size_t
    number_of_members, void *user_pointer);
```

Los parámetros son:

- `contents`: Puntero a los datos descargados.
- `size`: Tamaño de cada elemento en los datos.
- `number_of_members`: Número de elementos de datos.
- `user_pointer`: Puntero a una estructura de datos en memoria que se actualizará con los nuevos datos.

La función ajusta dinámicamente la memoria necesaria para almacenar el contenido descargado y lo copia en la estructura proporcionada.

**Función** `extract_links_from_html`

Esta función recibe una cadena de texto que contiene el HTML descargado y extrae todos los enlaces relevantes para libros de texto. Utiliza expresiones regulares para encontrar enlaces específicos en el HTML y guardarlos en un archivo o en memoria para su posterior procesamiento.

```
void extract_links_from_html(const char *html);
```

Este método es fundamental para filtrar los enlaces útiles de los que no lo son. En el contexto del *Proyecto Gutenberg*, se buscan enlaces que apunten a archivos de texto (.txt) o ePub, que contienen los libros.

**Función** `process_link`

La función `process_link` utiliza CURL para descargar el contenido del enlace identificado. El enlace puede corresponder a un archivo de texto o a una página que contiene múltiples enlaces a archivos.

```
void process_link(CURL *curl, const char *url);
```

Esta función:

- Realiza la descarga del contenido utilizando CURL.
- Pasa el contenido a la función de extracción de enlaces, en caso de que la página descargada contenga varios libros.
- Procesa el contenido descargado, lo limpia y lo almacena en un archivo local.

### Función `download_text_files`

Esta es la función principal que ejecuta el *web crawler* para descargar archivos de texto desde la URL base del *Proyecto Gutenberg*.

```
void download_text_files(const char *base_url);
```

Los pasos seguidos en esta función incluyen:

1. Inicialización de una sesión de CURL para manejar las solicitudes HTTP.
2. Descarga del contenido HTML de la URL base proporcionada.
3. Procesamiento del HTML descargado para identificar y extraer los enlaces de libros.
4. Llamado a `process_link` para descargar los archivos de texto desde los enlaces identificados.
5. Almacenamiento de los archivos de texto en el directorio de salida.

### Generación de Nombres de Archivos

La función `generate_unique_filename` es utilizada para generar un nombre de archivo único basado en un nombre base. Esta función es especialmente útil para evitar sobrescribir archivos cuando se descargan múltiples libros de texto.

```
char *generate_unique_filename(const char *base_path);
```

### Limpieza de Directorios

Para garantizar que el directorio donde se almacenan los libros esté siempre limpio antes de ejecutar una nueva descarga, se utiliza la función `remove_directory`, que elimina recursivamente todos los archivos y subdirectorios de un directorio dado.

```
int remove_directory(const char *dir_path);
```

Esta función se ejecuta antes de cada nueva sesión de descarga para asegurar que no existan archivos previos en conflicto.

### Conclusión

El *web crawler* implementado en este proyecto es fundamental para obtener los archivos de texto necesarios para las pruebas del algoritmo de Huffman. Utiliza herramientas robustas como CURL para descargar contenido web y expresiones regulares para identificar y extraer enlaces de libros. El uso de TidyLib permite limpiar el HTML descargado para obtener enlaces precisos y sin errores (ZenRows, 2024). Este web crawler es capaz de descargar automáticamente los archivos de texto de interés desde el *Proyecto Gutenberg* y almacenarlos en el sistema de archivos local para su posterior procesamiento.

## 2.4. Descripción del proyecto con instrucciones de cómo compilarlo y correrlo en el entorno base del curso

Este proyecto implementa el algoritmo de compresión y descompresión de archivos de texto utilizando Huffman. Se han desarrollado tres versiones del algoritmo para comprimir los archivos: serial, utilizando `fork()` para crear múltiples procesos y utilizando la biblioteca `pthread` para crear múltiples hilos. Adicionalmente, se ha desarrollado un web crawler para obtener los archivos de texto plano utilizados en las pruebas.

El objetivo del proyecto es comparar el rendimiento de las diferentes versiones del algoritmo y demostrar la eficiencia de la paralelización en comparación con la implementación serial. Las versiones desarrolladas incluyen:

- **Fork**: Implementación que utiliza la función `fork()` para crear múltiples procesos que comprimen partes del archivo en paralelo.
- **Thread**: Implementación multi-hilo utilizando la biblioteca `pthread`, que permite comprimir diferentes partes del archivo de manera concurrente dentro del mismo proceso.
- **Serial**: Implementación tradicional del algoritmo de Huffman que realiza la compresión de forma secuencial.

### 2.4.1. Código fuente

El código fuente del proyecto puede descargarse desde el siguiente enlace de GitHub:

```
https://github.com/ch0so/proyecto_1_IC6600.git
```

### 2.4.2. Instrucciones para la instalación

Este proyecto debe ejecutarse en un entorno Linux, en concreto, fue hecho en **Fedora Workstation 40**. A continuación, se detallan los pasos para configurar el entorno y resolver todas las dependencias necesarias para que el proyecto funcione correctamente.

### 2.4.3. Instalación de Fedora Workstation

Si no cuentas con Fedora Workstation 40, se puede descargar e instalarlo desde el siguiente enlace:

```
https://fedoraproject.org/es/workstation/download/
```

En esa dirección, solamente se debe seguir la documentación oficial proporcionada en el enlace para realizar la instalación.

### 2.4.4. Clonación del repositorio y dependencias

Una vez que se tenga el sistema operativo configurado, se deben seguir estos pasos:

1. Primero, se debe clonar el repositorio del proyecto desde GitHub:

   ```
   git clone https://github.com/ch0so/proyecto_1_IC6600.git
   ```

2. Luego, es necesario acceder al directorio del proyecto:

   ```
   cd proyecto_1_IC6600
   ```

3. Finalmente, se debe ejecutar el script de instalación para resolver todas las dependencias necesarias y compilar el proyecto:

   ```
   sudo bash dependencies.sh
   ```

   Este script se encargará de instalar las bibliotecas necesarias (como `pthread` y otras dependencias de compilación) y compilar el proyecto. Además, se encarga de compilar el programa.

### 2.4.5. Compilación

El proyecto utiliza un archivo `Makefile` para automatizar la compilación. Una vez que se hayan instalado todas las dependencias, se puede compilar el proyecto ejecutando el siguiente comando desde el directorio principal del proyecto:

```
make
```

Este comando compilará todas las versiones del algoritmo de Huffman (serial, `fork`, y `pthread`) y dejará los ejecutables listos para su uso.

### 2.4.6. Ejecución

El proyecto incluye varias opciones para ejecutar las diferentes versiones del algoritmo de compresión y descompresión. A continuación se describen las opciones disponibles y los comandos correspondientes:

1. **Ver lista de opciones**:

   ```
   ./project_1
   ```

Este comando muestra todas las opciones disponibles para ejecutar el programa.

2. **Descargar libros (Web Crawler)**:

```
./project_1 download_books
```

   Esta opción activa el web crawler para descargar los archivos de texto plano necesarios para las pruebas del algoritmo.

3. **Eliminar archivos comprimidos**:

```
./project_1 clear_directory_compressed
```

   Esta opción elimina los archivos binarios generados después de la compresión.

4. **Eliminar archivos descomprimidos**:

```
./project_1 clear_directory_decompressed
```

   Esta opción elimina los archivos generados durante la descompresión.

5. **Comprimir utilizando `fork()`**:

```
./project_1 compressed_fork
```

   Esta opción comprime los archivos utilizando la versión del algoritmo que emplea `fork()` para crear múltiples procesos.

6. **Descomprimir utilizando `fork()`**:

```
./project_1 decompressed_fork
```

   Esta opción descomprime los archivos utilizando múltiples procesos creados con `fork()`.

7. **Comprimir en modo serial**:

```
./project_1 compressed_serial
```

   Esta opción ejecuta la versión serial del algoritmo de compresión de Huffman.

8. **Descomprimir en modo serial**:

```
./project_1 decompressed_serial
```

   Esta opción ejecuta la versión serial para descomprimir los archivos.

9. **Comprimir utilizando** `pthread`:

```
./project_1 compressed_threads
```

   Esta opción comprime los archivos utilizando hilos (*threads*) para dividir el trabajo entre varias tareas concurrentes.

10. **Descomprimir utilizando** `pthread`:

```
./project_1 decompressed_threads
```

    Esta opción descomprime los archivos utilizando hilos.

## 2.5. Discusión

Para la recopilación de datos, se llevaron a cabo experimentos utilizando diversas configuraciones de núcleos de procesamiento, con el objetivo de obtener mediciones precisas del tiempo de ejecución del algoritmo. Se emplearon configuraciones de 2 y 4 núcleos, realizando varias repeticiones para asegurar que los datos sean precisos. Aunque en un escenario ideal se habrían utilizado un mayor número de núcleos para optimizar el rendimiento, las limitaciones de hardware de las máquinas lo impidieron.

### 2.5.1. Implementación serial

Para el caso del algoritmo serial, no se realizaron pruebas con diferentes cantidades de núcleos, dado que su implementación no utiliza paralelización. Como resultado, el rendimiento no variaría al modificar el número de núcleos disponibles. Los resultados obtenidos para este algoritmo son los siguientes:

Estos datos serán utilizados como referencia para el análisis de la implementación con threads y fork.

### 2.5.2. Implementación con fork

#### Algoritmo Fork (2 Núcleos)

Para las primeras pruebas, se emplearon dos núcleos con el objetivo de medir el tiempo de ejecución en un entorno limitado. A continuación, se presentan los resultados obtenidos:

| **Cantidad de iteraciones** | **Tiempo promedio al comprimir (ns)** | **Tiempo promedio al comprimir (s)** | **Tiempo promedio al descomprimir (ns)** | **Tiempo promedio al descomprimir (s)** |
|---|---|---|---|---|
| 10 | 29,987,991,000 | 29.987991 | 11,136,590,000 | 11.13659 |
| 25 | 32,323,056,000 | 32.323056 | 12,757,031,000 | 12.757031 |
| 50 | 31,972,297,000 | 31.972297 | 12,284,387,000 | 12.284387 |
| 100 | 31,511,964,000 | 31.511964 | 12,972,826,000 | 12.972826 |

Cuadro 2.1: Tiempo de Ejecución utilizando algoritmo serial

| **Cantidad de iteraciones** | **Tiempo promedio al comprimir (ns)** | **Tiempo promedio al comprimir (s)** | **Tiempo promedio al descomprimir (ns)** | **Tiempo promedio al descomprimir (s)** |
|---|---|---|---|---|
| 10 | 20,764,095,000 | 20.764095 | 11,780,475,000 | 11.78047 |
| 25 | 19,931,348,000 | 19.931348 | 11,404,381,000 | 11.404381 |
| 50 | 21,047,556,000 | 21.047556 | 11,191,059,000 | 11.191059 |
| 100 | 21,986,878,000 | 21.986878 | 11,480,328,000 | 11.480328 |

Cuadro 2.2: Tiempo de Ejecución utilizando algoritmo de Fork con 2 núcleos

**Comparación entre algoritmo serial y fork (2 Núcleos)** Se comparan a continuación los tiempos de ejecución del algoritmo Fork frente al Serial, utilizando dos núcleos. Según (College Transitions, 2024), la aceleración se calcula con la siguiente fórmula:

$$\text{Aceleración} = \left( \frac{T_{\text{serial}} - T_{\text{fork}}}{T_{\text{serial}}} \right) \times 100$$

En donde:

- $T_{\text{serial}}$: tiempo de ejecución promedio del algoritmo serial.
- $T_{\text{fork}}$: tiempo de ejecución promedio del algoritmo fork.

| **Cantidad de iteraciones** | **Tiempo promedio al comprimir Fork (s)** | **Tiempo promedio al comprimir Serial (s)** | **Diferencia de tiempo Serial - Fork (s)** | **Aceleración del algoritmo aproximada ( %)** |
|---|---|---|---|---|
| 10 | 20.764095 | 29.987991 | 9.223896 | 30.7 % |
| 25 | 19.931348 | 32.323056 | 12.391708 | 38.3 % |
| 50 | 21.047556 | 31.972297 | 10.924741 | 34.1 % |
| 100 | 21.986878 | 31.511964 | 9.525086 | 30.2 % |

Cuadro 2.3: Comparación entre algoritmo serial y fork con 2 núcleos (compresión)

En este caso, se observa una mejora significativa de aproximadamente un 30.2 % en la compresión, lo que indica una optimización del rendimiento del algoritmo al utilizar dos núcleos. A continuación, se compararon los datos del proceso de descompresión.

| **Cantidad de iteraciones** | **Tiempo promedio al descomprimir Fork (s)** | **Tiempo promedio al descomprimir Serial (s)** | **Diferencia de tiempo Serial - Fork (s)** | **Aceleración del algoritmo aproximada ( %)** |
|---|---|---|---|---|
| 10 | 11.78047 | 11.13659 | -0.643879 | -5.78 % |
| 25 | 11.404381 | 12.757031 | 1.352649 | 10.6 % |
| 50 | 11.191059 | 12.284387 | 1.093328 | 8.9 % |
| 100 | 11.480328 | 12.972826 | 2.492497 | 12.3 % |

Cuadro 2.4: Comparación entre algoritmo serial y fork con 2 núcleos (descompresión)

Aquí se observa una mejora de aproximadamente un 12.3 % en la descompresión, lo que sugiere que la implementación con Fork ofrece mejoras tanto en la compresión como en la descompresión. Para un análisis más exhaustivo, realizaremos pruebas con cuatro núcleos.

### Algoritmo Fork (4 Núcleos)

A continuación, se presentan los resultados obtenidos al emplear cuatro núcleos:

| Cantidad de iteraciones | Tiempo promedio al comprimir (ns) | Tiempo promedio al comprimir (s) | Tiempo promedio al descomprimir (ns) | Tiempo promedio al descomprimir (s) |
|---|---|---|---|---|
| 10 | 10,105,182,000 | 10.105182 | 8,499,972,000 | 8.499972 |
| 25 | 10,835,036,000 | 10.835036 | 8,462,184,000 | 8.462184 |
| 50 | 10,804,299,000 | 10.804299 | 8,357,108,000 | 8.357108 |
| 100 | 10,521,905,000 | 10.521905 | 8,985,330,000 | 8.985330 |

Cuadro 2.5: Tiempo de Ejecución utilizando algoritmo de Fork con 4 núcleos

**Comparación entre algoritmo serial y fork (4 Núcleos)**

Se comparan ahora los tiempos de ejecución del algoritmo Fork frente al Serial, utilizando cuatro núcleos:

| Cantidad de iteraciones | Tiempo promedio al comprimir Fork (s) | Tiempo promedio al comprimir Serial (s) | Diferencia de tiempo Serial - Fork (s) | Aceleración del algoritmo aproximada ( %) |
|---|---|---|---|---|
| 10 | 10.105182 | 29.987991 | 19.882809 | 66.3 % |
| 25 | 10.835036 | 32.323056 | 21.488020 | 66.4 % |
| 50 | 10.804299 | 31.972297 | 21.167998 | 66.2 % |
| 100 | 10.521905 | 31.511964 | 20.990059 | 66.6 % |

Cuadro 2.6: Comparación entre algoritmo serial y fork con 4 núcleos (compresión)

En este caso, se observa una mejora considerable de aproximadamente un 66.6 % en la compresión, lo que demuestra una optimización significativa en el uso de cuatro núcleos. A continuación, se analizan los resultados de la descompresión.

| Cantidad de iteraciones | Tiempo promedio al descomprimir Fork (s) | Tiempo promedio al descomprimir Serial (s) | Diferencia de tiempo Serial - Fork (s) | Aceleración del algoritmo aproximada ( %) |
|---|---|---|---|---|
| 10 | 8.499972 | 11.13659 | 2.636618 | 23.6 % |
| 25 | 8.462184 | 12.757031 | 4.294847 | 33.6 % |
| 50 | 8.357108 | 12.284387 | 3.927279 | 31.9 % |
| 100 | 8.985330 | 12.972826 | 3.987496 | 30.7 % |

Cuadro 2.7: Comparación entre algoritmo serial y fork con 4 núcleos (descompresión)

En este caso, se observa una mejora significativa de aproximadamente un 30.7 % en la descompresión, lo cual demuestra que la implementación con Fork mejora ambos procesos, tanto compresión como descompresión.

#### Análisis de resultados

**Fork con 2 núcleos** Al comparar los resultados del algoritmo Fork con 2 núcleos frente al serial, se observa una mejora considerable en la compresión, con una aceleración de 30.2 % con 100 iteraciones. Esto indica que el algoritmo implementado con fork logra una redistribución eficiente de las tareas en la compresión, reduciendo el tiempo de ejecución.

En cuanto a la descompresión, la mejora es más moderada, con una aceleración de 12.3 % con 100 iteraciones. Esto demuestra que la implementación con fork mejora ambos procesos, tanto compresión como descompresión.

**Fork con 4 núcleos** Al utilizar 4 núcleos, la aceleración fue de 66.6 %, mostrando una reducción significativa en el tiempo de compresión. Este resultado confirma que la compresión es una operación altamente paralelizable, y el uso de más núcleos permite procesar las tareas de manera casi simultánea.

En la descompresión, también se logró una mejora significativa, con aceleraciones que alcanzaron hasta un 30.7 %. A pesar de que el proceso de descompresión es menos eficiente en cuanto a paralelización que la compresión, el uso de 4 núcleos demostró un impacto positivo.

### 2.5.3. Implementación con pthread

#### Algoritmo pthread (2 Núcleos)

Para las primeras pruebas, se emplearon dos núcleos con el objetivo de medir el tiempo de ejecución en un entorno limitado. A continuación, se presentan los resultados obtenidos:

| **Cantidad de iteraciones** | **Tiempo promedio al comprimir (ns)** | **Tiempo promedio al comprimir (s)** | **Tiempo promedio al descomprimir (ns)** | **Tiempo promedio al descomprimir (s)** |
|---|---|---|---|---|
| 10 | 21,250,703,000 | 21.250703 | 11,154,577,000 | 11.154577 |
| 25 | 20,177,750,000 | 20.177750 | 12,174,877,000 | 12.174877 |
| 50 | 20,229,151,000 | 20.229151 | 10,509,841,000 | 10.509841 |
| 100 | 20,479,915,000 | 20.479915 | 10,842,795,000 | 10.842795 |

Cuadro 2.8: Tiempo de Ejecución utilizando algoritmo de Threads con 2 núcleos

#### Comparación entre algoritmo serial y threads (2 Núcleos)

Se comparan a continuación los tiempos de ejecución del algoritmo Threads frente al Serial, utilizando dos núcleos. Según (College Transitions, 2024), la

aceleración se calcula con la siguiente fórmula:

$$\text{Aceleración} = \left( \frac{T_{\text{serial}} - T_{\text{thread}}}{T_{\text{serial}}} \right) \times 100$$

En donde:

- $T_{\text{serial}}$: tiempo de ejecución promedio del algoritmo serial.
- $T_{\text{thread}}$: tiempo de ejecución promedio del algoritmo thread.

| **Cantidad de iteraciones** | **Tiempo promedio al comprimir Thread (s)** | **Tiempo promedio al comprimir Serial (s)** | **Diferencia de tiempo Serial - Thread (s)** | **Aceleración del algoritmo aproximada ( %)** |
|---|---|---|---|---|
| 10 | 21.250703 | 29.987991 | 8.737288 | 29.1 % |
| 25 | 20.177750 | 32.323056 | 12.145306 | 37.5 % |
| 50 | 20.229151 | 31.972297 | 11.743146 | 36.7 % |
| 100 | 20.479915 | 31.511964 | 11.032049 | 35 % |

Cuadro 2.9: Comparación entre algoritmo serial y threads con 2 núcleos (compresión)

En este caso, se observa una mejora de aproximadamente un 35 % en la compresión, lo que indica una optimización del rendimiento del algoritmo al utilizar dos núcleos. A continuación, se compararon los datos del proceso de descompresión.

| **Cantidad de iteraciones** | **Tiempo promedio al descomprimir Thread (s)** | **Tiempo promedio al descomprimir Serial (s)** | **Diferencia de tiempo Serial - Thread (s)** | **Aceleración del algoritmo aproximada ( %)** |
|---|---|---|---|---|
| 10 | 11.154577 | 11.13659 | -0.017986 | -0.16 % |
| 25 | 12.174877 | 12.757031 | 0.582154 | 4.56 % |
| 50 | 10.509841 | 12.284387 | 1.774546 | 14.4 % |
| 100 | 10.842795 | 12.972826 | 2.130031 | 16.4 % |

Cuadro 2.10: Comparación entre algoritmo serial y threads con 2 núcleos (descompresión)

Aquí se observa una mejora de aproximadamente un 16.4 % en la descompresión, lo que sugiere que la implementación con Threads ofrece mejoras tanto en la compresión como en la descompresión. Para un análisis más exhaustivo, realizaremos pruebas con cuatro núcleos.

**Algoritmo thread (4 Núcleos)**

A continuación, se presentan los resultados obtenidos al emplear cuatro núcleos:

| **Cantidad de iteraciones** | **Tiempo promedio al comprimir (ns)** | **Tiempo promedio al comprimir (s)** | **Tiempo promedio al descomprimir (ns)** | **Tiempo promedio al descomprimir (s)** |
|---|---|---|---|---|
| 10 | 13,506,957,000 | 13.506957 | 8,364,552,000 | 8.364552 |
| 25 | 11,897,433,000 | 11.897433 | 7,798,895,000 | 7.798895 |
| 50 | 12,039,212,000 | 12.039212 | 8,837,559,000 | 8.837559 |
| 100 | 12,362,055,000 | 12.362055 | 8,026,399,000 | 8.026399 |

Cuadro 2.11: Tiempo de Ejecución utilizando algoritmo de Threads con 4 núcleos

**Comparación entre algoritmo serial y threads (4 Núcleos)**

Se comparan ahora los tiempos de ejecución del algoritmo Threads frente al Serial, utilizando cuatro núcleos:

| **Cantidad de iteraciones** | **Tiempo promedio al comprimir Threads (s)** | **Tiempo promedio al comprimir Serial (s)** | **Diferencia de tiempo Serial - Fork (s)** | **Aceleración del algoritmo aproximada ( %)** |
|---|---|---|---|---|
| 10 | 13.506957 | 29.987991 | 16.481034 | 54.9 % |
| 25 | 11.897433 | 32.323056 | 20.425623 | 65.1 % |
| 50 | 12.039212 | 31.972297 | 19.933085 | 62.3 % |
| 100 | 12.362055 | 31.511964 | 19.149909 | 60.7 % |

Cuadro 2.12: Comparación entre algoritmo serial y threads con 4 núcleos (compresión)

En este caso, se observa una mejora considerable de aproximadamente un 60.7 % en la compresión, lo que demuestra una optimización significativa en el uso de cuatro núcleos. A continuación, se analizan los resultados de la descompresión.

| Cantidad de iteraciones | Tiempo promedio al descomprimir Thread (s) | Tiempo promedio al descomprimir Serial (s) | Diferencia de tiempo Serial - Fork (s) | Aceleración del algoritmo aproximada (%) |
|---|---|---|---|---|
| 10 | 8.364552 | 11.13659 | 2.772038 | 24.8 % |
| 25 | 7.798895 | 12.757031 | 4.958136 | 38.8 % |
| 50 | 8.837559 | 12.284387 | 3.446828 | 24.8 % |
| 100 | 8.026399 | 12.972826 | 4.946427 | 38.1 % |

Cuadro 2.13: Comparación entre algoritmo serial y threads con 4 núcleos (descompresión)

En este caso, se observa una mejora significativa de aproximadamente un 38.1 % en la descompresión, lo cual demuestra que la implementación con threads mejora tanto la compresión como la descompresión.

#### Análisis de resultados

**Threads con 2 núcleos** Al comparar los resultados del algoritmo Threads con 2 núcleos frente al serial, se observa una mejora considerable en la compresión, con una aceleración de 35 % con 100 iteraciones. Esto indica que el algoritmo implementado con threads logra una redistribución eficiente de las tareas en la compresión, reduciendo el tiempo de ejecución. En cuanto a la descompresión, la mejora es más moderada, con una aceleración de 16.4 % con 100 iteraciones. Esto demuestra que la implementación con threads mejora ambos procesos, tanto compresión como descompresión.

**Threads con 4 núcleos** Al utilizar 4 núcleos, la aceleración fue de 60.7 %, mostrando una reducción significativa en el tiempo de compresión. Este resultado confirma que la compresión es una operación altamente paralelizable, y el uso de más núcleos permite procesar las tareas de manera casi simultánea.

En la descompresión, también se logró una mejora significativa, con aceleraciones que alcanzaron hasta un 38.1 %. A pesar de que el proceso de descompresión es menos eficiente en cuanto a paralelización que la compresión, el uso de 4 núcleos demostró un impacto positivo.

### 2.5.4. Análisis de resultados

En esta sección se analizan y comparan los resultados obtenidos de las implementaciones del algoritmo de Huffman utilizando dos enfoques de paralelización: Fork y Threads. Las comparaciones se basan en la aceleración tanto en el proceso de compresión como en el de descompresión, utilizando configuraciones de 2 y 4 núcleos. A continuación se detallan los hallazgos clave.

**Comparación en configuración de 2 núcleos**

| Implementación | Aceleración Compresión ( %) | Aceleración Descompresión ( %) |
|---|---|---|
| Fork | 30.2 % | 12.3 % |
| Threads | 35 % | 16.4 % |

Cuadro 2.14: Comparación de aceleraciones con dos núcleos

Con 2 núcleos, el algoritmo Thread logró una aceleración del 35 %, ligeramente superior a la aceleración del 30.2 % obtenida con Fork. Esto sugiere que, en un entorno de bajos núcleos, Threads distribuye las tareas de compresión de manera más eficiente que Fork, posiblemente debido a la menor sobrecarga de la creación de hilos frente a los procesos.

Esto es probablemente debido a que los hilos tienen una sobrecarga menor en comparación con los procesos. La creación de threads consume menos recursos del sistema, ya que comparten el mismo espacio de direcciones y tienen menor costo de creación y gestión en comparación con los procesos que requieren su propio espacio de memoria. Además, los threads permiten una comunicación más eficiente, lo que reduce los tiempos de espera y aumenta la utilización efectiva de los núcleos. (GeeksforGeeks, 2023a)

En la descompresión, la aceleración con Threads alcanzó un 16.4 %, mientras que Fork solo obtuvo una aceleración de 12.3 %. Aunque ambos enfoques muestran una mejora moderada en la descompresión, el enfoque de Threads vuelve a ser más eficiente en esta configuración.

Esto es dado que la descompresión suele ser un proceso más ligero en términos de computación, la eficiencia en la gestión de recursos y la comunicación interna cobra mayor importancia (GeeksforGeeks, 2023a). Esto explica por qué los hilos siguen mostrando una ventaja en la descompresión. Este análisis indica que con 2 núcleos, Threads ofrece una ligera ventaja en términos de rendimiento en ambos procesos.

**Comparación en configuración de 4 4úcleos**

| Implementación | Aceleración Compresión ( %) | Aceleración Descompresión ( %) |
|---|---|---|
| Fork | 66.6 % | 30.7 % |
| Threads | 60.7 % | 38.1 % |

Cuadro 2.15: Comparación de aceleraciones con cuatro núcleos

Al aumentar el número de núcleos a 4, Fork mostró una aceleración del 66.6 %, superando ligeramente al 60.7 % alcanzado con Threads. En este caso,

Fork aprovecha de manera más efectiva los 4 núcleos para la compresión, beneficiándose de la paralelización del proceso de manera más eficiente que con 2 núcleos.

Los procesos pueden aprovechar de manera más efectiva múltiples núcleos para realizar tareas en paralelo de manera independiente (University, 2021). A medida que aumentan los núcleos, el aislamiento de los procesos y la capacidad de trabajar de manera autónoma en diferentes núcleos puede ser más beneficioso, ya que cada proceso forked tiene su propio espacio de direcciones y recursos dedicados, lo que reduce la interferencia entre tareas.

Por otro lado, aunque Threads también mejora su rendimiento con más núcleos, no alcanza el nivel de Forks. Esto podría deberse a que los hilos, aunque eficientes, comparten el mismo espacio de direcciones y pueden generar mayor competencia por los recursos del núcleo cuando se aumenta la carga de trabajo y los núcleos disponibles.

En la descompresión, Threads muestra una ventaja, con una aceleración del 38.1 %, mientras que Fork alcanza un 30.7 %. Aunque ambos enfoques mejoran significativamente la descompresión con 4 núcleos, Threads sigue mostrando un mejor rendimiento en la descompresión en comparación con Fork, incluso con más recursos disponibles.

La naturaleza menos intensiva de la descompresión no parece beneficiar tanto a Fork, mientras que la baja sobrecarga de los threads sigue siendo una ventaja en este proceso, permitiendo una mejor paralelización. (Won, 2021)

### Conclusiones de datos

El análisis de los resultados obtenidos de las implementaciones del algoritmo de Huffman utilizando dos enfoques de paralelización, Fork y Threads, bajo configuraciones de 2 y 4 núcleos, ha permitido extraer las siguientes conclusiones clave:

Analizando los datos obtenidos, se puede concluir lo siguiente:

- **Threads** tiende a ser más eficiente en configuraciones con menor cantidad de núcleos (2 núcleos), ya que reduce la sobrecarga del sistema y mejora la utilización de recursos en tareas paralelas.
- **Fork**, por otro lado, parece beneficiarse más de configuraciones con mayor número de núcleos (4 núcleos), especialmente en tareas intensivas como la compresión, donde su capacidad de aislar los procesos en diferentes núcleos se traduce en una mejor paralelización.
- En procesos menos intensivos como la descompresión, **Threads** continúa mostrando un rendimiento superior independientemente del número de núcleos, lo que sugiere que este enfoque es más adecuado cuando se busca minimizar la sobrecarga y maximizar la eficiencia en tareas livianas.

En conclusión, para la compresión en sistemas con más de 2 núcleos, **Fork** puede ser una opción más eficiente, mientras que **Threads** es más adecuado

para configuraciones con menos núcleos y para la descompresión, donde la comunicación eficiente y la baja sobrecarga juegan un papel crucial.

## 2.6. Conclusiones

- La concurrencia por medio de `pthread` permite que se puedan compartir de forma eficiente recursos entre hilos, lo que permite que haya un menor consumo tanto de memoria así como en la creación de tareas concurrentes, lo que hace que `pthread` sea una herramienta considerable a utilizar pa aplicaciones que necesitan que haya una alta eficiencia en el manejo de recursos y que se vean beneficiadas por la implementación de hilos para llevar a cabo múltiples tareas.

- El algoritmo de Huffman, al emplear códigos binarios de longitud variable según la frecuencia de los caracteres, es sumamente eficiente para archivos cuyos caracteres tengan distribuciones no uniformes debido a la naturaleza del algoritmo que a su vez favorece a que se realice una compresión sin pérdida para que así se pueda reducir el tamaño de los archivos sin que se vean afectados los datos.

- La compresión y descompresión de archivos en paralelo, utilizando técnicas como `pthread` o procesos separados mediante `fork()`, permite optimizar el uso de sistemas multicore puesto que se encargan de que las tareas sean distribuidas entre varias unidades de procesamiento, lo cual permite aprovechar al máximo la capacidad de procesamiento con la que cuenta el equipo al mismo tiempo que se consigue una aceleración que varía de acuerdo a la naturaleza de las tareas y qué tanto puedan sacar provecho de estas herramientas.

# Referencias


Basanta, P. (2010). *Programación con hilos.* Universidad Carlos III de Madrid. Descargado de https://www.it.uc3m.es/pbasanta/asng/CES/M2/concurrent_1_es.pdf

College Transitions. (2024, 30 de March). *How to calculate acceleration – 3 formulas you must know.* Blog Post. Descargado de https://www.collegetransitions.com/blog/how-to-calculate-acceleration-formula/

GeeksforGeeks. (2023a). *Difference between process and thread.* Publicación de un blog. Descargado de https://www.geeksforgeeks.org/difference-between-process-and-thread/

GeeksforGeeks. (2023b, 9 de May). *Thread functions in c/c++ [funciones de hilos en c/c++].* GeeksforGeeks. Descargado de https://www.geeksforgeeks.org/thread-functions-in-c-c/

Hart, M. (1992). *The history and philosophy of project gutenberg.* Project Gutenberg. Descargado de https://www.gutenberg.org/about/background/history_and_philosophy.html

Hu, J. (2023, 12 de October). *Usa la función fork en c.* Delft Stack. Descargado de https://www.delftstack.com/es/howto/c/fork-in-c/

Molina, R. (2018). *Codificación huffman.* Depto. Ciencias de la Computación e Inteligencia Artificial, Universidad de Granada. Descargado de http://www.kramirez.net/RI/Material/Internet/T3_CODIGO_DE_HUFFMAN.pdf

Powell, Z. (2024, 16 de January). *Lossy vs lossless compression: Comprehensive analysis.* ShortPixel. Descargado de https://shortpixel.com/blog/lossy-vs-lossless/

Tanenbaum, A., y García, R. (2003). *Sistemas operativos modernos.* Pearson Educación. Descargado de https://books.google.co.cr/books?id=g88A4rxPH3wC

TidyLib. (2017, 1 de March). *tidybuffio.h [documentación de la api].* TidyLib. Descargado de https://api.html-tidy.org/tidy/tidylib_api_5.4.0/tidybuffio_8h.html

University, C. (2021). *Concurrency lecture (cs 2112).* Material de curso. Descargado de https://www.cs.cornell.edu/courses/cs2112/2021fa/lectures/lecture.html?id=concurrency

Won, D. (2021). *Process vs thread: What's the difference?* Publicación en Medium. Descargado de https://medium.com/@denniswon/process-vs-thread-whats-the-difference-23cb30a772c4

ZenRows. (2024, 1 de June). *Web scraping with c in 2024 [raspado web con c en 2024].* ZenRows. Descargado de https://www.zenrows.com/blog/web-scraping-c#get-the-html