INSTITUTO TECNOLÓGICO DE COSTA RICA

ESCUELA DE COMPUTACIÓN

Principios de Sistemas Operativos

Proyecto 1

Elaborado por:

Mauricio Agüero Márquez - 2020087412

Gustavo Pérez Badilla - 2020084832

I SEMESTRE 2024

Introducción

Este proyecto se centra en el rendimiento de un programa utilizando *n* de procesos, donde se harán comparaciones del rendimiento con distintos valores de *n*. El programa, llamado *copy*, es una herramienta para copia de directorios y archivos, y trabaja utilizando los multiprocesos para optimizar la tarea. Este programa proporciona una solución eficiente para copiar estructuras de directorios completas, incluyendo todos sus archivos y subdirectorios, de un directorio origen a un directorio destino.

La implementación de esta herramienta se basa en un pool de procesos, donde un conjunto predefinido de procesos trabajadores se encargan de copiar los archivos y directorios. Esta estrategia permite distribuir la carga de trabajo entre varios procesos, aprovechando los beneficios del procesamiento paralelo para mejorar el rendimiento y la eficiencia del programa. Además, el programa utiliza la comunicación interprocesos mediante colas de mensajes para coordinar las tareas entre el proceso principal y los procesos trabajadores. Esta comunicación permite asignar archivos específicos para copiar a cada proceso trabajador y recibir notificaciones de finalización de tareas, garantizando así una ejecución ordenada y controlada de las operaciones de copia.

Descripción del problema

El objetivo principal de este proyecto es desarrollar un programa en C que permita copiar de manera eficiente un directorio completo, incluyendo todos sus archivos y subdirectorios, desde una ubicación de origen a una ubicación de destino. El programa debe ser capaz de realizar esta tarea de copia utilizando múltiples procesos trabajadores que operen en paralelo, formando un "pool de procesos".

El programa debe recibir como argumentos el directorio de origen y el directorio de destino. Posteriormente, debe explorar el directorio de origen y sus subdirectorios para identificar todos los archivos y directorios que deben ser copiados. Para cada archivo o directorio encontrado, se debe asignar una tarea a uno de los procesos trabajadores del pool para que realice la copia correspondiente.

Para coordinar las tareas entre el proceso principal y los procesos trabajadores, se utilizará una cola de mensajes. Esta cola permitirá enviar instrucciones desde el proceso principal a los trabajadores, indicándoles qué archivo o directorio deben copiar. Además, se utilizará esta cola para que los trabajadores informen al proceso principal cuando hayan finalizado la copia de un archivo o directorio.

Para evaluar el rendimiento del programa con diferentes configuraciones de "pool de procesos", se debe medir el tiempo total de ejecución del programa en cada configuración. Este análisis permitirá determinar cuál es el tamaño óptimo del pool de procesos que maximiza la eficiencia en la copia de archivos y directorios.

Finalmente, el programa debe generar un archivo de registro en el cual se registre cada operación de copia realizada, incluyendo el nombre del archivo o directorio, el identificador del proceso trabajador que realizó la copia, y el tiempo empleado en la operación.

Definición de estructuras de datos

Aquí se define la estructura para almacenar el texto del mensaje y el tipo (en este caso siempre es 1) :

```
struct msg_buffer {
    long msg_type;
    char msg_text[MSGSZ];
};
```

Esta es útil para guardar información de los archivos y directorios, por ejemplo, nos permite distinguir entre esos dos ya que se debe tener un comportamiento diferente dependiendo de cuál es:

```
struct stat statbuf;
stat(src_path, &statbuf);
```

Esta es usada para el manejo de directorios:

```
struct dirent *dp;
```

Descripción detallada y explicación de los componentes principales del programa

Hay 3 funciones principales además de *main*. La primera se encarga de copiar un archivo desde una ubicación de origen (*src*) a una ubicación de destino (*dst*). Además, registra la operación en un archivo de registro (*logfile*) y utiliza el identificador de hilo (*thread_id*) para identificar el proceso que realizó la copia.

```c
void copy_file(const char *src, const char *dst, FILE *logfile, int thread_id) {
```

La segunda se encarga de copiar un directorio completo desde una ubicación de origen (*src*) a una ubicación de destino (*dst*), incluyendo todos sus archivos y subdirectorios. También registra cada operación en el archivo de registro (*logfile*) y utiliza el identificador de hilo (*thread_id*).

```c
void copy_directory(const char *src, const char *dst, FILE *logfile, int thread_id) {
```

La tercera es ejecutada solamente por los procesos hijos, que se encargan de recibir mensajes desde la cola de mensajes y realizar las operaciones de copia correspondientes. Utiliza un identificador de hilo (*thread_id*) y la cola de mensajes (*msqid*) para recibir instrucciones.

```c
void worker(int msqid, FILE *logfile, int thread_id) {
```

Y la última es la función *main*. Obtiene los directorios de origen y destino desde los argumentos de línea de comandos y define la cola de mensajes. Luego inicializa el cronómetro para medir el tiempo total que tardó la tarea de copiar archivos. Crea los procesos hijos y los manda a ejecutar la función *worker*; una vez los hijos terminan de ejecutar esta función, también terminan en su proceso en main. En el padre, se hace el manejo de los directorios y archivos para enviarlos por mensajes a los trabajadores para iniciar las copias. Por último solo espera a que los trabajadores finalicen y calcula el tiempo total de ejecución.

Mecanismo de creación y comunicación de procesos

En esta parte se crean la clave y el identificador de la cola de mensajes:

```c
key_t key = ftok("msgq", 65);
msqid = msgget(key, 0666 | IPC_CREAT);
```

Y en estas imágenes se muestran donde el proceso padre envía los mensajes de los directorios a sus hijo y donde los hijos reciben el mensaje respectivamente:

```c
snprintf(message.msg_text, sizeof(message.msg_text), "%s %s", src_path, dst_path);
message.msg_type = 1;

if (msgsnd(msqid, &message, strlen(message.msg_text) + 1, 0) == -1) {
    perror("msgsnd");
    exit(EXIT_FAILURE);
}
```

```
if (msgrcv(msqid, &message, MSGSZ, 1, 0) == -1) {
    perror("msgrcv");
    exit(EXIT_FAILURE);
}
```

Y por último, donde se envían y reciben los mensajes de finalización respectivamente:

```
for (int i = 0; i < num_workers; i++) {
    snprintf(message.msg_text, sizeof(message.msg_text), "DONE");
    message.msg_type = 1;

    if (msgsnd(msqid, &message, strlen(message.msg_text) + 1, 0) == -1) {
        perror("msgsnd");
        exit(EXIT_FAILURE);
    }
}
```

```
if (strcmp(message.msg_text, "DONE") == 0) {
    break;
}
```

Pruebas de rendimiento

El algoritmo presentó resultados positivos en cuanto a su funcionamiento, logró copiar todos los archivos que se encontraban dentro del directorio que se deseaba copiar, además de esto, también mantiene la jerarquía del directorio, respetando la ubicación de estos tal y como se encuentran en la ubicación original.

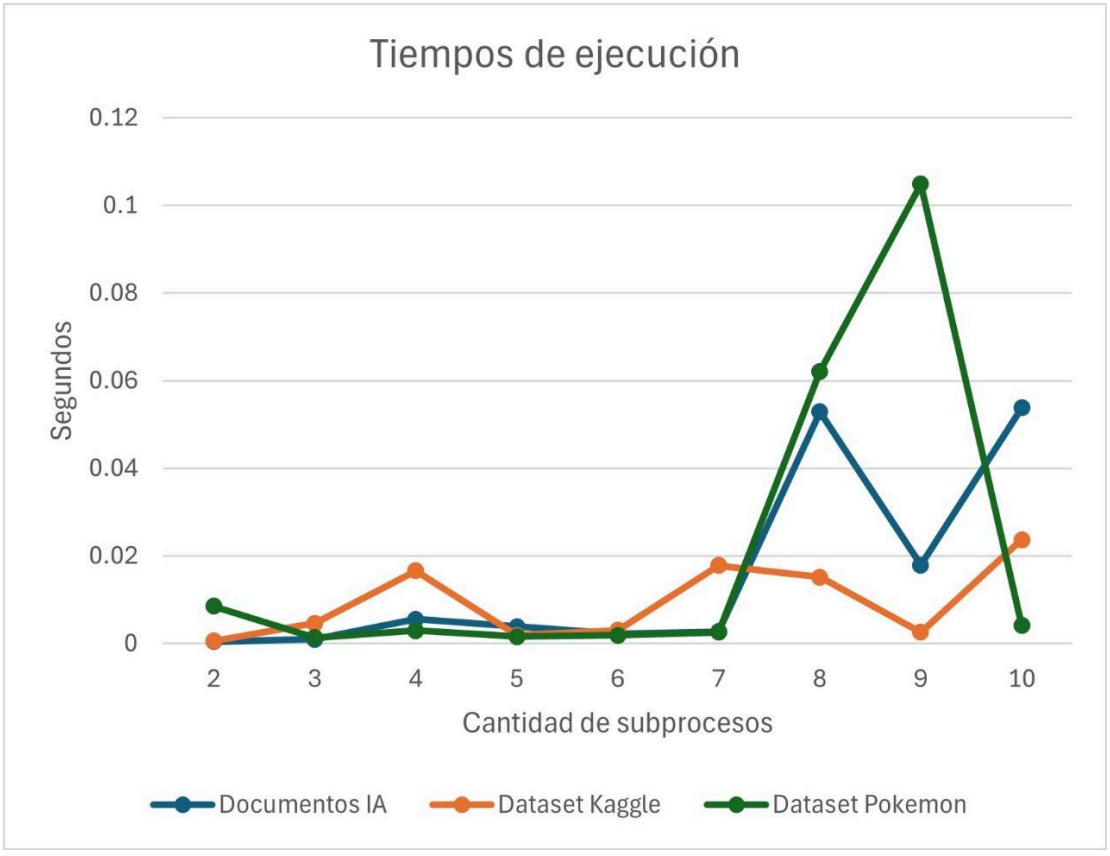
Se realizaron diversas pruebas con múltiples directorios de diferentes tamaños, siendo estos un 24.7 MB, 130 MB y 539 MB, donde el primero corresponde a un repositorio en GitHub, y los restantes son grupos de datos extraídos de la página Kaggle que contienen imágenes para entrenamiento de inteligencia artificial. Gracias a la amplia cantidad de elementos dentro de las carpetas, estas resultan ser excelentes sujetos de prueba para ejecutar el programa.

Las pruebas se hicieron con diferentes cantidades de procesos dentro del Pool, iniciando en 2 y terminando en 10 procesos hijos, obteniendo los siguientes resultados:

# de procesos	Repositorio GitHub (24.7 MB)	Dataset Kaggle (130 MB)	Dataset Kaggle (539 MB)

2	0.000417	0.000624	0.008533
3	0.000936	0.004638	0.001374
4	0.005518	0.016605	0.002975
5	0.003892	0.00191	0.001555
6	0.002174	0.003016	0.001898
7	0.002712	0.01783	0.002572
8	0.052891	0.015116	0.06208
9	0.017887	0.002594	0.104856
10	0.053841	0.023709	0.004132

A raíz de la tabla anterior se obtiene la siguiente gráfica de líneas que refleja de una forma más visual los resultados obtenidos:



Como se puede observar, existen dos segmentos dentro del gráfico donde los tiempos aumentan considerablemente, siendo estos al usar cuatro subprocesos y entre siete a nueve. Por otro lado, al usar seis subprocesos los tiempos de ejecución resultaron ser óptimos para las tres pruebas, donde el mayor tiempo resultó ser de 0.003016 segundos para la carpeta de 130 MB. Sin embargo, cabe destacar que los tiempos de ejecución no reflejan un patrón que indique una mejoría con respecto a la cantidad de subprocesos empleados, es decir, es irregular.

Conclusiones

Gracias a los estudios realizados se puede concluir que el programa cumple con el funcionamiento planteado de forma que logra copiar los archivos de un directorio origen y los pega en la ubicación destino, durante las pruebas realizadas no se encontraron inconsistencias en cuanto a la cantidad de archivos, reflejando que todos estos fueron enviados exitosamente.

Por otro lado, en relación a los tiempos de ejecución se encontraron resultados en común que muestran un atraso al emplear ciertas cantidades de subprocesos, pero al mostrar un patrón irregular, no se puede concluir con certeza que la cantidad de subprocesos afecta directamente en el rendimiento del programa, esto está fuertemente vinculado a la capacidad de las computadoras, ya que estas cuentan con el rendimiento suficiente para realizar este tipo de tareas en tiempos muy breves.