IC6600: Principios de Sistemas Operativos

Proyecto I - II Semestre 2017

Gabriela Garro A - 2014107501

Índice

Introducción

Compilación

```
gcc -pthread -o grep grep.c queue.c
```

Uso del programa

```
./grep [opciones] [expresión regular] [archivos|directorio]
```

- **Opciones:**
  - -l: Solo muestra el nombre del archivo
  - -r: Busca recursivamente en el directorio
- **Expresión regular:**
  - Ver http://pubs.opengroup.org/onlinepubs/7908799/xbd/re.html para más información
- **Archivos / Directorio:**
  - Uno o más archivos/directorios, separado por espacios

Ejemplo de uso

Un ejemplo, en el cual se buscan direcciones de correo electrónico, es el siguiente

```
./grep -r "[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,}" test
```

Descripción del problema

El enunciado del proyecto define los siguientes requerimientos:

1. Realizar en ambiente Unix.
2. Utilizar C; específicamente las librerías y funciones:
    a. “regex.h”:
        i. regcomp
        ii. regexec
    b. “dirent.h”
        i. opendir
        ii. readdir
        iii. closedir
3. Debe de llamarse grep y llamarse de la siguiente manera en la línea de comandos:

```
grep [opciones] [expresión regular] [archivos|directorio]
```

4. Las opciones son:
    a. -l: Mostrar solamente nombre de archivo.
    b. -r: Buscar recursivamente en subdirectorios.
5. La expresión regular corresponde al formato definido por “regex.h”
6. Archivos / Directorio: Se pueden indicar hasta 10 archivos y directorios separados por espacios.
7. Programación con hilos: Mantener una piscina de hilos. Cada uno analizará un archivo, hasta que no queden más archivos.

Estructuras de datos

queue.h

Este es el header de la estructura de datos queue, la cual se utiliza para administrar el trabajo de los hilos.

Para utilizar este archivo, se debe de incluir `"queue.h"` e inicializar la estructura interna llamando a `void queue_init()`.

La estructura de queue mantiene un array interno, al cual se puede acceder con las funciones `void queue_push(char * data)` y `char *  queue_pop()`.

```
#include <stdio.h>
#include <stdbool.h>

#define MAX 256

struct queue
{
    char * queue_array[MAX];
    int front;
    int rear;
     int itemCount;
};
typedef struct queue QUEUE;
QUEUE q;

void queue_init();
void queue_push(char * data);
char *  queue_pop();
char * queue_peek();
void queue_display();
int queue_size();
bool queue_is_void();
bool queue_is_full();
```

Los detalles de su implementación no son relevantes para el resto de la documentación.

Componentes principales

En esta sección, se muestra código del código fuente y se explica su flujo y funcionamiento.

grep.c

Este archivo contiene casi toda la lógica del proyecto.

```
#define MAX_NUM_THREADS 16
#define MAX_NUM_FILES 16
#define MAX_MATCHES 1

// Variables globales
bool arg_l = false;
bool arg_r = false;
char * arg_regular_exp;
char * files[MAX_NUM_FILES];
/* *** */
// Variables de regex
int regex_status;           // Status de la compilación
regex_t regex_comp;   // Expresión compilada
// Declaración de mutex
pthread_mutex_t mutex;
```

Define las siguientes constantes:

- `#define MAX_NUM_THREADS`: La cantidad de threads del pool. Este es el valor que se cambia a la hora de hacer pruebas.
- `#define MAX_NUM_FILES 16`: La cantidad máxima de archivos o directorios que se pueden concatenar en la línea de comandos.
- `#define MAX_MATCHES`: La cantidad máxima de coincidencias en un string. Limitada a 1 por rendimiento.

Las variables globales son analizadas y actualizadas por medio de la función `int get_args(int args, char* argv[]) {}`

main()

La función main se encuentra estructurada de esta manera:

```
int main(int args, char* argv[]) {
	/* *** */
	// Obtener valores de argumentos
	get_args(args, argv);
	// Crear cola de archivos
	queue_init();
	int i;
	for (i = 0; i < num_arg_files; i++) {
		add_files(files[i], false);
	}
	// Inicializar el regex
	regex_status = regcomp(&regex_comp, arg_regular_exp, REG_EXTENDED);
	/* *** */
	// Crear threads
	pthread_t threads[MAX_NUM_THREADS];
	int thread_id;
	for (i = 0; i < MAX_NUM_THREADS; i++) {
		thread_id = pthread_create(&threads[i], NULL, thread_exec,
&i);
		if (thread_id) {
			printf("ERROR: Return code from pthread_create() is
%d.\n", thread_id);
			exit(-1);
		}
	}
	/* *** */
	pthread_exit(NULL);
}
```

add_files()

Se espera que el usuario llame a esta función de esta manera:

```
queue_init(); int i;
for (i = 0; i < num_arg_files; i++) {
      add_files(files[i], false);
}
```

Toma como parámetros:

- `char * object`: Un puntero a char, con un nombre de archivo o directorio
- `bool check_once`: Una bandera interna para saber si abrir un directorio interno o no. Se recomienda llamar con false.

```
void add_files(char * object, bool check_once) {
      DIR * dir;
      struct dirent * entry;
      if (!(dir = opendir(object)))
            return;
      // Mientras el objeto existe
      while((entry = readdir(dir)) != NULL && !check_once) {
            // Si es un directorio
            if (entry->d_type == DT_DIR) {
                  /* *** */
                  //Actualizar el nombre del path
                  /* *** */
                  if (arg_r) add_files(path, false);
                  else add_files(path, true);
            }
            else {
                  /* *** */
                  queue_push(full_path);
            /* *** */
}
```

El objetivo de esta función es tomar el nombre de un archivo y meterlo en la cola. Si es el nombre de un directorio, mete los archivos internos en la cola.

thread_exec()

```
void * thread_exec(void * _thread_arg) {
      /* *** */
      char * file;
      while (1) {
            pthread_mutex_lock(&mutex);
            if (queue_is_void()) {
                  pthread_mutex_unlock(&mutex);
                  pthread_exit(NULL);
            }
            else {
                  file = queue_pop();
            }
            pthread_mutex_unlock(&mutex);
            // Analizar si el archivo contiene la expresión regular
            // Abrir el archivo
            /* *** */
            /* Leer línea por línea */
            /* *** */
            while((read = getline(&buffer, &len, file_obj)) != -1){
                  // Comparar
                  regmatch_t matches[MAX_MATCHES];
                  if (regexec(&regex_comp, buffer, MAX_MATCHES, matches, 0)
== 0) {
                        if (arg_l)
                              printf("%s\n", file);
                        /* *** */
```

Esta función se mantiene en un ciclo infinito hasta que la cola de archivos se encuentra vacía, con lo cual cierra el hilo. Cada hilo obtiene un solo archivo. Lo abre, lo analiza línea por línea, y compara la expresión regular con cada una de las líneas. Si hay una coincidencia, la escribe a la línea de comandos.

Mecanismo de creación y comunicación de hilos

La función en la que se crean los threads es la siguiente:

```
// Crear threads
pthread_t threads[MAX_NUM_THREADS];
int thread_id;
for (i = 0; i < MAX_NUM_THREADS; i++) {
      thread_id = pthread_create(&threads[i], NULL, thread_exec, &i);
      if (thread_id) {
            printf("ERROR: Return code from pthread_create() is %d.\n",
thread_id);
            exit(-1);
      }
}
```

Se utiliza la función `pthread_create(&threads[i], NULL, thread_exec, &i)` sobre un array de `pthread_t threads[MAX_NUM_THREADS]`.

Los hilos no se comunican entre sí (en el ámbito de mensajes). La ejecución de cada uno es independiente y es solamente cortada hasta que la cola de archivos esté vacía.

Sin embargo, sí comparten un mutex, el cual activan en secciones como esta:

```
pthread_mutex_lock(&mutex);
if (queue_is_void()) {
      pthread_mutex_unlock(&mutex);
      pthread_exit(NULL);
}
else {
      file = queue_pop();
}
pthread_mutex_unlock(&mutex)
```

Pruebas de rendimiento

Las pruebas se hicieron sobre un cuerpo de archivos de texto (1270 archivos) recopilados por esta página, los cuales se encuentran en la carpeta test. Se utilizaron los siguientes comandos:

```
./grep -r e test
```

437,342 coincidencias.

```
./grep -r good test
```

2195 coincidencias.

```
./grep -r the test
```

168,668 coincidencias.

```
./grep -r white test
```

376 coincidencias.

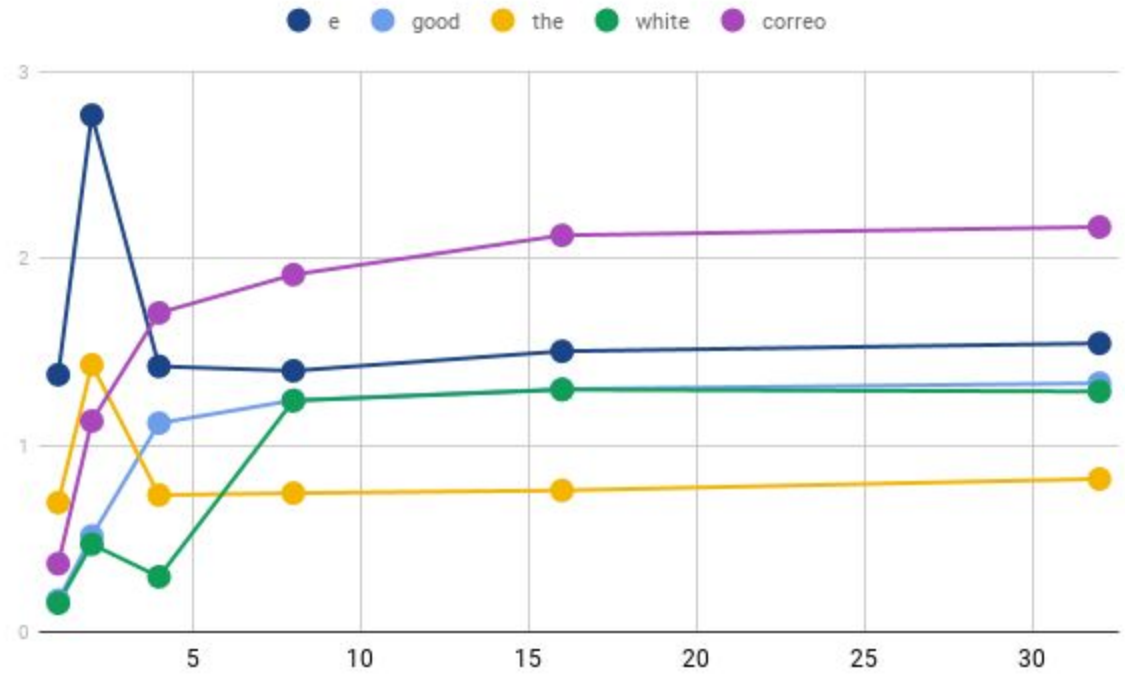
```
./grep -r “[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,}” test
```

96 coincidencias. Esta expresión regular indica direcciones de correo.

A continuación, se indica el tiempo en microsegundos que tarda el programa en analizar la carpeta, variando la cantidad de hilos.

Cantidad de hilos	**Tiempo en segundos**				
	e	**good**	**the**	**white**	**correo**
1	1.378319	0.170072	0.693289	0.154566	0.365777
2	2.768497	0.514412	1.431610	0.470345	1.130200
4	1.422975	1.118721	0.733536	0.295674	1.709287
8	1.399099	1.240975	0.743831	1.239143	1.914162
16	1.503741	1.300827	0.758316	1.298035	2.124867
32	1.545989	1.332717	0.819646	1.288218	2.169314

Por la gráfica, se puede notar que cuando se utiliza un solo hilo, la cantidad de tiempo que tarda es significativamente menor. Sin embargo, esto se le puede atribuir a que el mecanismo de medición utilizado, mide la cantidad de ciclos de reloj del programa: si el programa tiene muchos hilos, cuenta más ciclos de reloj.



Si se analiza la tendencia de duración de las búsquedas, se puede ver que no existe mucha diferencia entre utilizar 16 y 32 hilos: aproximadamente la misma cantidad de hilos se terminan utilizando en ambos casos, independientemente de cuántos se hayan creado.

Las búsquedas de “good”, “white” y correos se comportan de manera similar: su mínimo de tiempo se logra con un solo hilo, y después de 8 hilos, dura casi la misma cantidad de tiempo.

Se debe de investigar la causa por la cual buscar con 2 hilos la letra “e” y “the” aumenta de manera tan significativa la búsqueda, en comparación con 1 y 4 hilos.

Conclusiones

1. Se debe de valorar la relevancia de medir la cantidad de ciclos de reloj. Se intentó medir la cantidad de tiempo transcurrido, sin embargo, esta medición no es constante ni buen indicador porque puede variar dependiendo de atrasos de E/S u otros procesos en el sistema que afectan al programa.
2. El mecanismo de medición utilizado, mide la cantidad de ciclos de reloj del programa: si el programa tiene muchos hilos, cuenta más ciclos de reloj.
3. Según los mecanismos de medición de ciclos de reloj utilizados, se concluye de manera consistente que **al utilizar un hilo, el programa se ejecuta en la menor cantidad de ciclos de reloj.**
4. En las búsquedas de “good”, “white” y correos, el mínimo de tiempo se logra con un solo hilo. Después de 8 hilos, no hay variaciones: dura casi la misma cantidad de tiempo.
5. Buscar con 2 hilos la letra “e” y “the” aumenta de manera significativa la búsqueda, en comparación con 1 y 4 hilos.
6. No existe mucha diferencia entre utilizar 16 y 32 hilos: aproximadamente la misma cantidad de hilos se terminan utilizando en ambos casos, independientemente de cuántos se hayan creado.