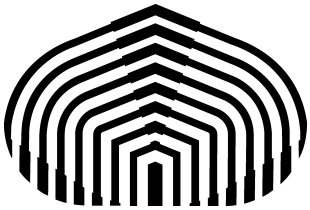

UNIVERSIDAD SIMÓN BOLÍVAR.
INGENIERÍA DE LA COMPUTACIÓN.
DEPT.COMPUTACIÓN Y TECNOLOGÍA DE LA INFORMACIÓN.
CI3825: SISTEMAS DE OPERACIÓN I.

Proyecto 1: Palíndromos

PROFESOR:

Carlos Gomez

.

ALUMNOS:

José D. Bracuto D. 13-10173

Miguel C. Canedo R. 13-10214

Índice

Introducción

En este informe se explicará como nuestro programa determina si existen palíndromos en un árbol de directorios. Un palíndromo es una palabra de tres o más letras que se lee igual al derecho y al revés. El objetivo de este proyecto es la formación de palabras mediante la concatenación de los nombres de una ruta de directorios, que podrían empezar y terminar en cualquier directorio del árbol formado a partir del la ubicación donde el programa es ejecutado. Ademas se explicarán tanto las formas con las que logramos realizar dicho proyecto como las cualidades que tiene el mismo.

Compilación y ejecución

Para compilar el programa primero se debe estar en la ubicacion actual del mismo y correr el script "make".

```
mike@mikeVirtual:~/Escritorio/LabOperativos/Proyecto1$ make
```

Igualmente para correr el programa programa se deberá correr con "./main" seguido de los flags según sea el caso.

```
mike@mikeVirtual:.../Proyecto1$ ./main -d <carpeta> -m <altura> -f
```

Los flags son opcionales y sus funciones son:

- **-d <carpeta> :** Establece <carpeta> como el directorio desde el cual inicia el árbol. Si no se usa éste flag, por defecto el directorio desde el que iniciará sera el directorio en el que se esta ejecutando el programa.
- **-m <altura> :** Establece <altura> como la altura máxima de árbol que se explorará, es decir se restringirá la profundidad del árbol a crear. Si no se usa éste flag, por defecto el programa recorrerá todas las rutas posibles en las que tenga permiso.
- **-f :** Establece que se debe incluir los nombres de los archivos en la lectura de los directorios. Si no se usa éste flag el árbol se realizará sin tomar en cuenta los archivos, solo los directorios.

Estrategia de creación de procesos

Para la realización de este programa creamos dos procesos. El primer proceso, el "proceso padre", se va a encargar de detectar los palíndromos en los path que le indique el segundo proceso, el "proceso hijo". Para esto se creó un proceso hijo con la función fork() y seguidamente se continuó la ejecución alternando según sea necesario entre los procesos padre e hijo.

```
...
pid_t child = fork(); //Se crea el proceso hijo

if (childpid == 0){ // Se ejecuta el proceso hijo
        // proceso de recorrido de directorio
}
else{ // Se ejecuta el proceso padre
        // proceso de busqueda de palindromos
}
```

Proceso hijo

El proceso hijo se encargará del recorrido de los directorios recursivamente mediante la función recorrer(**char* path, int profundidad**), la cual realiza un recorrido de los directorios con permisos validos desde el directorio especificado (path) hasta la profundidad deseada (profundidad), para luego escribir dicha ruta como un **string** en el pipe para mandárselo al proceso padre. **(Para mayor información acerca de la comunicación entre el proceso padre e hijo revise la sección Estrategia de comunicación y sincronización de procesos)**

Proceso padre

El proceso padre se encargará de la búsqueda de palíndromos en el **string** mandado por el hijo. Realizando primero a la eliminación de los separadores ('/') del string para luego proceder con la búsqueda de los palíndromos llamando a la función subPalindromos(**char* str, char* palindromos[], int nPalindromos[]**) que detecta y almacena los substrings que sean palindromos del **string** "str" y retorna la cantidad de palíndromos encontrados.

Estrategia de comunicación y sincronización de procesos

Para lograr una comunicación y sincronización efectiva se decidió utilizar un **pipe** para ir almacenando en forma de string los distintos path, el cual cumple la función de comunicar ambos procesos. Ademas estarán operando junto al **pipe** dos **semáforos** (leer y escribir) que servirán para llevar el control del momento cuando el padre debe **leer** el **pipe** o el momento cuando el hijo debe **escribir** en dicho **pipe**. El proceso para esto se puede descomponer en tres casos:

1. En el comienzo del programa se nombran los **semáforos** (leer y escribir), se crea el **pipe** y se abren los **semáforos** antes nombrados, iniciando el **semáforo** leer apagado mientras que escribir inicia encendido. Ademas de des-linkearlos del programa principal para que en el momento en el que los procesos cierren dichos **semáforos**, estos sean destruidos.

```
...
sem_t *leer, *escribir; // Se nombran los semaforos para
   leer y escribir
...
pipe(p)  //Se crea el pipe "p"

// Se abren los semaforos antes nombrados
leer = sem_open("semL", (O_CREAT|O_EXCL), 1, 0);
escribir = sem_open("semE", (O_CREAT|O_EXCL), 1, 1);

sem_unlink("semL");  //Des-linkeo de semaforos
sem_unlink("semE");
...
```

2. Para el caso cuando que se esté ejecutando el proceso hijo, se deberá esperar hasta se pueda escribir en el pipe mediante el **semáforo** escribir, seguidamente se escribe el path en el **pipe** y por último se cambia el estado del **semáforo** leer para que el proceso padre pueda leer lo que contenga el **pipe**.

```
...
sem_wait(escribir); // Se espera hasta el momento en el
   que se pueda escribir en el pipe

write(p[1], path, 100000); // Se escribe en el pipe

sem_post(leer); // Se cambia el semaforo para que el
   proceso padre pueda leer el pipe
...
```

3. Para el caso en el que se esté ejecutando el proceso padre, se cierra el lado de escritura del **pipe** y se espera hasta el momento en el que se pueda leer dicho **pipe** mediante el **semáforo** leer y finalmente luego de leer lo que contenga el **pipe**, se vuelve a cambiar el **semáforo** escribir para que el proceso hijo pueda escribir en nuevamente en él.

```
...
close(p[1]); // se cierra el pipe

while(read(p[0], path, 100000) > 0){ // Se lee el pipe

        sem_wait(leer); // Se espera hasta el momento en
           el que se pueda leer el pipe

        // proceso de busqueda de palindromos

    sem_post(escribir); // Se modifica el semaforo para
       que se pueda escribir en el pipe

}
...
```

Manejo de strings

Tanto la búsqueda de palíndromos como el manejo de strings se ubico en un .c aparte llamado **manejoStrings.c** con su respectivo .h incluido. En este se importaron las librerías **stdlib.h , stdio.h y string.h** para poder utilizar ciertas

funciones que facilitarían el manejo de strings. En este archivo se puede encontrar las funciones:

concat

Esta función recibe dos strings y los concatena agregando un separador '/' entre ellos mediante la asignación de memoria y asignaciones directas para luego devolver un string con dicho resultado. Siendo esta función de suma utilidad al momento de construir los path.

quitarSeparador

Es un funcion simple a la que se le pasa como argumento un string y este lo recorre eliminando el separador '/'.

printArray

Función que recibe un arreglo de strings y un entero que representa el tamaño de dicho arreglo con la finalidad de imprimir cada elemento separándolos por ',' y liberando el espacio utilizado.

esLetra

Esta función se encarga de revisar si un carácter pasado como parámetro es o no una letra ('a', ... , 'z' ó 'A', ... , 'Z').

subPalindromos

Finalmente llegamos a la función principal con la que encontramos los palíndromos en el string suministrado. Esta función trabaja con una matriz "tabla[i][j]" de dimensiones $n * n$, donde n representa el tamaño del string. Esta función se divide en tres fases:

1. En la primera fase se detectan los substrings de tamaño 1, es decir, todas las letras por si solas. Por lo que se modifica las posiciones de la Diagonal Principal de la tabla y en el caso de los caracteres del castellano, como estos ocupan el doble de espacio entonces se modificarían dichos espacios en la tabla.

```
...
tabla[i][i] = TRUE; // substrings de tamanio 1 (i=j)

if (str[i] == -61) { // caracteres del castellano
        tabla[i][i+1] = TRUE;  // ocupa dos espacios
        tabla[i+1][i] = TRUE;
    }
```

2. En esta fase se deberá detectar los substrings de tamaño 2 hay dos letras consecutivas iguales (‘aa’, ‘bb’, ‘Aa’, ‘bB’, ‘áá’, ‘áÁ’, etc.) considerando los mismos como palíndromos. Durante esta etapa se consideraron también los caracteres del castellano como la mayúsculas y minúsculas.

```
// Si hay dos caracteres del castellano entrara en este
   condicional.
if (str[i] == -61 && str[i] == str[i+2] && (str[i+1] ==
   str[i+3] || (str[i+1] - str[i+3] == 32 || str[i+3] -
   str[i+1] == 32) ) ) {
        tabla[i][i+3] = TRUE;
        continue;
}

// En caso de que los dos caracteres no sean del
   castellano y sean iguales, entra en este condicional.
if ((str[i] != -61 && str[i] == str[i+1]) || (esLetra(str
   [i]) && (str[i] - str[i+1] == 32 || str[i+1] - str[i]
   == 32)))
        tabla[i][i+1] = TRUE;
```

3. Finalmente en esta fase, se trabaja con cada substring de tamaño 3 o mayor. En cada iteración se va verificando cada substring que comience en la posición **i** y termine en la posición **j**. Para ello se revisa si tabla$[i+1][j-1]$ está marcada como **True**, indicando que le substring que va desde la posición **i+1** hasta la **j-1** es un palíndromo, y que ademas se cumpla que el carácter en la posición **i** del string original es igual al carácter en la posición **j**. Si ambas condiciones se cumplen, significa que el substring que va desde **i** hasta **j** es un Palíndromo. Finalmente se procede a marcar como **True** la posición tabla$[i][j]$ y a verificar

si dicho palíndromo ya fue reportado e incluido en el arreglo palindromos[] que, en caso de no estar presente, el mismo se agrega al arreglo.

Cabe destacar que a lo largo de cada fase explicada previamente se tuvieron en cuenta la aparición de caracteres del castellano como la letra **'ñ'** o las **vocales acentuadas** (las cuales se consideraban diferentes a las vocales sin acento). Además de considerar durante toda la corrida la indiferencia de si una letra era mayúscula o minúscula, ambas era tratadas como iguales (es decir, 'e' == 'E'), y a su vez de consideraban al **espacio en blanco** como un carácter más.

Explicaciones extras

Se podría añadir como una explicación extra, que a pesar de ciertas herramientas que facilitaban el trabajo del recorrido de directorios, como la función **ftw()**, el equipo decidió implementar de manera propia una función recursiva que se encargara de realizar dicho trabajo. Esta función es **recorrer(char* path, int profundidad)**, la cual intenta abrir el directorio indicado por el path, si posee los permisos necesarios, y revisar los archivos y directorios que contenga, para luego entrar en los directorios, con la excepción de los casos especiales " **.** " y " **..** " los cuales hacen referencia al directorio mismo y al directorio previo, respectivamente. La decisión de implementar dicha función fue con la intención de tener un mejor acercamiento a como deben funcionar los procesos en el entorno de UNIX.

Conclusiones y lecciones aprendidas

Después de haber realizado este proyecto podemos concluir que C es un lenguaje muy completo en el cual se pueden acceder a varios tipos de información de manera relativamente simple, siempre y cuando se tenga una previa investigación y proceso de comprensión adecuado. Pero también es cierto que se debe de estar pendiente y tener constantemente en cuenta como se trabajará y manejaran todos los procesos y apuntadores de memoria puesto que en el peor de los casos un mal manejo de procesos conllevaría a que la computadora se quede pegada y un mal manejo de la memoria conllevaría a eliminar información crucial sin poder luego recuperarla, por eso es que se aconseja entender bien la teoría antes de llevarla a la práctica.

Referencias

- http://cs.indstate.edu/~cbasavaraj/cs559/the_c_programming_language_2.pdf
- http://devdocs.io/c/
- http://www.um.es/earlyadopters/actividades/a3/PCD_Activity3_Session1.pdf