Instituto Tecnológico de Costa Rica

Escuela de Ingeniería en Computación

Principios de Sistemas Operativos

Primer Proyecto

Profesor

Armando Arce Orozco

Estudiantes

Daniel Argüello Poma

Alonso Garita Granados

II semestre, 2023

Tabla de Contenidos

1. Introducción

Un proceso es una actividad con espacio de memoria asignado y con un programa o código asociado que debe seguir y ejecutar. Un proceso es una entidad activa, se ejecuta y consume recursos del sistema, del hardware y usa tiempo de procesador. Cada aplicación que abrimos en la computadora conlleva a la ejecución de al menos un proceso, pero en realidad suelen ser más.

En C, y concretamente en los sistemas Unix, se pueden crear procesos hijos a partir de un proceso padre con la función `fork`. Cada hijo es una copia idéntica de su padre hasta el momento en que se ejecutó `fork`; tiene el mismo contenido en memoria (heap, stack, data y text), pero son copias y si el padre altera su contenido en memoria, esto no afecta al contenido del hijo y viceversa.

Por eso existen mecanismos para comunicar procesos en tiempo de ejecución, lo que permite un trabajo colaborativo entre varios procesos para completar una tarea grande y compleja. Los dos mecanismos de comunicación de interprocesos (IPC) son la memoria compartida y la cola de mensajes. Para este proyecto se solicitó usar una cola de mensajes para que varios procesos colaboren en la lectura de un archivo grande para luego buscar en él un patrón de expresión regular.

El resultado esperado es un programa similar al programa *grep* de Unix, pero con multiprocesos. Es probable que la implementación de grep sea mucho más eficiente que la desarrollada en este proyecto, pero el objetivo no es ser mejor que *grep*, sino evidenciar que con varios procesos en colaboración el trabajo de leer y buscar en el archivo es más rápido que con uno solo.

2. Descripción del problema

Objetivo

El objetivo de este proyecto es comparar el rendimiento de realizar una tarea colaborativa utilizando múltiples procesos. Para ello se deberá desarrollar una versión multiprocesos del programa llamado *grep* que permite buscar múltiples palabras sobre un archivo e imprimir el párrafo en dónde aparecen dichas palabras.

En este proyecto se deben realizar múltiples pruebas para determinar la cantidad óptima de procesos para ejecutar este tipo de tarea sobre un único archivo muy grande.

Uso del programa

El programa grep recibe una expresión regular y una lista de archivos:

```
grep 'pattern1|pattern2|pattern3|...' archivo
```

Por omisión el programa debe mostrar la línea del archivo en donde se encuentra la expresión regular.

Manejo de expresiones regulares

El lenguaje C por sí mismo no maneja expresiones regulares, sin embargo se puede usar la librería "regex.h" que cuenta con las funciones regcomp y regexec que permiten realizar esta tarea. La función regcomp se debe ejecutar inicialmente y recibe la expresión regular ingresada por comando, luego la función regexec se debe ejecutar sobre cada una de las líneas del archivo.

Manejo de múltiples procesos

El programa debe contar con múltiples procesos que se encarguen de inspeccionar el archivo. Debido a que las búsquedas se realizan sobre archivos de gran tamaño no sería eficiente crear un único proceso. Es por ello que es necesario crear un conjunto compartido de procesos (pool de procesos) que incluya solo una cantidad limitada de procesos, y en donde cada proceso trabajará sobre un buffer propio en donde se debe leer una porción del archivo. Como parte del proyecto se

deben realizar diferentes pruebas (con diferentes archivos grandes) para determinar un tamaño "óptimo" del pool de procesos. Se utilizará el tiempo de ejecución para determinar dicho tamaño óptimo.

En este caso se utilizará un pool de procesos estático, es decir, el programa creará la totalidad de los procesos al inicio del programa y únicamente utilizará estos para realizar todo el trabajo. Tal como usted debe suponer, a los procesos que terminen de buscar en su porción del archivo deberán leer otra parte del archivo para realizar la búsqueda hasta terminar con todo el archivo. Se debe tener especial cuidado de que dos o más procesos lean la misma porción del archivo, o bien, que traten de escribir al mismo tiempo en la salida.

Note que al leer una porción del archivo puede suceder que una línea no sea leída completamente. Por eso cada proceso debe revisar si alguna línea se cortó, y actualizar el puntero de lectura del archivo (fseek) hasta el final de la última línea que se leyó completa.

Sincronización entre procesos

Los procesos se deberán sincronizar entre ellos mediante mensajes (NO sockets). La sincronización incluye determinar la posición de la última lectura al archivo y escribir a la salida. Para ello puede utilizar al padre como un proceso coordinador al cual los otros procesos realizan estas peticiones.

Consideraciones generales

- Todo el desarrollo del proyecto debe realizarse en lenguaje C y sobre ambiente Unix.
- No es permitido el uso de otras librerías adicionales a "regex.h" para programar el proyecto. Tampoco es válida la copia de porciones significativas de código desde Internet.
- El tamaño del buffer de lectura para cada proceso será de 8K (8192 bytes). No es válido leer un tamaño de buffer más grande y luego dividirlo en buffers de 8K. La idea es que cuando algunos procesos están leyendo otro procesen los datos.

- Los procesos deben ir alternando la lectura al archivo. No es válido calcular de antemano las porciones que cada proceso va a leer.
- Se debe realizar el procesamiento sobre un único archivo muy grande (no varios archivos pequeños).
- Se deben utilizar procesos hijos NO hilos de ejecución.
- Se deberá generar una documentación formal, en formato pdf, en donde se describan las diferentes etapas del desarrollo del proyecto, las decisiones de diseño que se tomaron, los mecanismos de programación utilizados, y los resultados de las diferentes pruebas al programa. Dicha documentación deberá incluir al menos las siguientes secciones:
    - Introducción
    - Descripción del problema (este enunciado)
    - Definición de estructuras de datos
    - Descripción detallada y explicación de los componentes principales del programa
    - Mecanismo de creación y comunicación de procesos
    - Pruebas de rendimiento
    - Conclusiones
- El proyecto puede realizarse en grupos de dos estudiantes.
- No se permite la copia de código entre grupos de estudiantes, o entregar código realizado por estudiantes los semestres anteriores, tampoco es permitido utilizar librerías adicionales (desarrolladas por terceros).
- Puede utilizar el sitio del proyecto Gutemberg para descargar archivos de texto muy grandes. En particular puede descargar Don Quijote, Divina Comedia, Los hermanos Karamazov, Les Misérables, Guerra y Paz, y Anna Karenina.

3. Definición de estructuras de datos

Para este proyecto se diseñó la siguiente estructura para el mensaje entre procesos:

```
struct message {
   long type;
   int mensaje[5];
   //0. Posición de lectura | 1. Terminó el archivo | 2. Desuso | 3. Terminó de analizar | 4. i del hijo
   char contenido[MSG_SIZE]; // Agregamos contenido al mensaje
};
```

Los campos de la estructura corresponden a:

`type`: El tipo del mensaje, que es usado por la función msgrcv para que el proceso pueda encontrar el mensaje que necesita en la cola de mensajes.

`mensaje`: Un arreglo de banderas usadas durante todo el programa.

- `mensaje[0]`: Posición del archivo donde un proceso terminó de leer y por donde el siguiente proceso debe continuar.
- `mensaje[1]`: Bandera que indica con 1 si el archivo se terminó de leer completamente.
- `mensaje[2]`: Bandera en desuso que originalmente indicaba al proceso padre cuándo debía imprimir un mensaje de la cola.
- `mensaje[3]`: Bandera mediante la cual un proceso hijo indica con 1 al proceso padre que ya terminó de analizar su buffer y está listo para leer cuando se le ordene.
- `mensaje[4]`: Identificador consecutivo que se le asignó al hijo

`contenido`: Espacio del mensaje donde se coloca el texto que se debe imprimir.

La comunicación entre mensajes se realiza mediante una cola, la cual se crea con un valor clave `msqkey`. Se decidió que se usará el PID del proceso padre como el valor clave. Si existe un error al crear la cola se detecta mediante el valor de `msqid` y se aborta el programa; si se crea exitosamente `msqid` contendrá el identificador de la cola para que sea usada por todos los procesos.

```c
//Creación de la cola de msg
key_t msqkey = getpid();
int msqid = msgget(msqkey, IPC_CREAT | 0666);
if (msqid == -1) {
    perror("msgget");
    exit(EXIT_FAILURE);
}
```

El archivo se accede y se lee por medio de un puntero `FILE`. Este tipo forma parte de las librerías estándar de C. Se abre el archivo para leerlo con la función `fopen`. Esta función recibe como parámetro el segundo parámetro que se envía al momento de llamar al programa que contiene el nombre del archivo a leer.

```c
//Abrir el archivo
FILE *fp;
fp = fopen(argv[2], "r");
if (fp == NULL) {
    perror("fopen");
    exit(EXIT_FAILURE);
}
```

El pool de procesos hijos se maneja como un arreglo del tamaño de procesos definido. Cada campo del arreglo representa con un 0 cuando el proceso no está trabajando, y con un 1 cuando el proceso está trabajando, ya sea leyendo o analizando.

```c
//Pool de hijos
int hijos[NUM_PROCESS];
memset(hijos, 0, sizeof(hijos));
```

4. Descripción detallada y explicación de los componentes principales del programa

El código se encuentra estructurado como se describe en esta sección. Todo el programa está escrito en un solo archivo `.c`; al inicio del archivo se incluyen todas las librerías necesarias y se definen las constantes y la estructura `message`, de la cual se declara una instancia `msg`.

La función main recibe los argumentos de la llamada del programa en el parámetro `argv`; en `argv[1]` se recibe el patrón de la expresión regular a buscar, y en `argv[2]` se recibe el nombre del archivo. Al iniciar main se crean la cola de mensajes (identificada por `msqid`), el puntero al archivo (`fp`) y el pool de procesos (`hijos`), y luego se compila la expresión regular en la instancia `regex`.

Tras crear e iniciar todas las estructuras e instancias necesarias para el trabajo, se crean todos los hijos mediante un `for` y la función `fork`. Recordemos que si la función `fork` retorna un 0 entonces estamos ejecutando un proceso hijo y su ejecución queda limitada al código dentro del `if`.

Los procesos hijos se dividen en dos tipos de acuerdo a sus funciones; los procesos lectores se encargan de leer el archivo, guardar su contenido en un buffer y analizar este buffer en busca de coincidencias con el patrón, y por otro lado el proceso pintor cuya única función es imprimir las líneas con coincidencias que envían los lectores. Por ende en el programa siempre habrá al menos dos procesos hijos, un lector y un pintor.

El proceso pintor recibe mensajes de tipo 440 que en su campo contenido tienen el texto que se debe imprimir. El pintor se encicla hasta que un mensaje le indique que se terminó de leer el archivo o que ya no tenga mensajes en la cola.

Cada proceso lector tiene su propio `buffer` de 8K de tamaño y tres punteros para manipularlo. Luego entra en un ciclo infinito donde espera recibir un mensaje con su identificador como tipo. Si recibe la notificación de que ya se terminó de leer todo el archivo, entonces el proceso termina.

Si aún no se ha terminado de leer todo el archivo, entonces el lector procede a leer el archivo y copiar su contenido en su `buffer`. En esta sección también el proceso se asegura de dejar el puntero de lectura en una posición que termine en un cambio de línea y envía un mensaje al padre con esa posición para que otro proceso siga leyendo. Tras enviar el mensaje, el lector tokeniza el buffer con la función `strtok` por saltos de línea y busca en cada token alguna coincidencia con el patrón; y si la encuentra, envía el token por un mensaje al proceso pintor para que lo imprima.

Después del código de los hijos, el código del padre inicia enviando un primer mensaje al primer lector para comenzar con el trabajo. Luego se encicla mientras que el padre tenga que seguir coordinando la lectura del archivo o que algún hijo esté trabajando. En este ciclo el padre espera por un mensaje de algún hijo. Si el hijo le avisa que ya se terminó de leer el archivo, el padre apaga su bandera que indica que aún tiene que coordinar la lectura. Si un proceso le indica que ya terminó de analizar y que está listo para leer, el padre revisa si algún otro hijo sigue trabajando o si ya todos terminaron de analizar. Por último el padre anda a leer a un hijo inactivo, enciende la bandera del hijo en el pool y envía la posición que le envió el hijo anterior.

Finalmente, cuando el padre ya no tenga que coordinar la lectura y todos sus hijos hayan terminado de analizar, el padre notificará a todos los lectores que deben terminar, y luego le notifica al pintor que debe terminar. El padre espera por que cada uno de sus hijos terminen y luego cierra el archivo, elimina la cola y termina el programa.

5. Mecanismo de creación y comunicación de procesos

Creación de procesos

Los procesos hijos se crean en un ciclo `for`, y por tanto siendo `NUM_PROCESS` el número total de procesos hijos; hay un proceso pintor y `NUM_PROCESS-1` lectores. Luego de la función `fork`, si el valor de retorno es 0 se sabe que es un proceso hijo que se identifica con el valor `i` del for al momento de

crearlo. Si el valor de `i` es igual a `NUM_PROCESS`, entonces es el último proceso hijo y será el proceso pintor.

```
//Crear hijos
for (int i = 1; i <= NUM_PROCESS; i++) {
    if (fork() == 0) {
        //Hijo pintor
        if(i == NUM_PROCESS){
            ...
        }

        //Hijos lectores
        char buffer[BUFFER_SIZE];
        char *inicioLinea = buffer;
        char *finalLinea = buffer;
        char *inicioArchivo = buffer;
        while (1) {
            ...
        }
    }
}
```

Tipos de mensajes

Se eligieron los siguientes tipos de mensajes enviados por `msg.type` para que cada proceso busque y envíe los mensajes de las funciones que le corresponden.

Tipo	Descripción
`i`	Mensaje de padre a lector para que inicie la lectura. El valor de este tipo corresponde con el valor de `i` del proceso.
`100`	Mensaje del lector al padre con la última posición que leyó del archivo a su buffer. El siguiente hijo lee a partir de esta posición.
`440`	Mensajes hacia el pintor; los lectores se comunican con él para imprimir los resultados y el padre para terminar su ejecución.

Comunicación entre procesos

A continuación se describe el proceso de comunicación entre procesos siguiendo la secuencia de emisor-receptor en su secuencia esperada.

El padre envía un primer mensaje al primer lector (`i = 1`) con la posición inicial del archivo (`actualPosition = 0`) mediante `msgsnd`.

```
  //El padre
   int i = 1;
   msg.type = i;
   msg.mensaje[0] = actualPosition;
   msgsnd(msqid, (void *)&msg, sizeof(msg) - sizeof(long),
IPC_NOWAIT);
```

El lector recibe el mensaje de la cola con el tipo (`msg.type`) igual a su valor `i`. De este mensaje importa `msg.mensaje[0]` que contiene la posición desde la cual empezar a leer, y `msg.mensaje[1]` que le indica que ya se terminó de leer el archivo y como el lector estaba esperando, entonces debe terminar.

```
   //Hijos lectores
   char buffer[BUFFER_SIZE];
   char *inicioLinea = buffer;
   char *finalLinea = buffer;
   char *inicioArchivo = buffer;
   while (1) {
      msgrcv(msqid, &msg, sizeof(msg), i, 0);
      if (msg.mensaje[1] == 1) {
         fclose(fp);
         exit(0);
      }
      //Lectura del archivo
      fseek(fp, msg.mensaje[0], SEEK_SET);
      size_t bytesRead = fread(buffer, sizeof(char), sizeof(buffer), fp);
      if (bytesRead != sizeof(buffer)) {
         msg.mensaje[1] = 1;
      }
```

Cuando un lector lee mediante `fread`, la función retorna el número de bytes que leyó, si el número que leyó es menor al tamaño del `buffer` es porque llegó al

final del archivo y enciende la bandera `msg.mensaje[1]` para indicar que ya concluyó todo el proceso de lectura.

Tras mover la posición del puntero hasta la última línea completa acabada en un salto de línea, el lector envía un mensaje al padre (`msg.type = 100`) con la posición en que terminó mediante `msg.mensaje[0]`.

```
   fseek(fp, j, SEEK_CUR);
   actualPosition += j+1;
   msg.mensaje[0] = actualPosition;
   msg.mensaje[2] = 0;
   msg.mensaje[3] = 0;
   msg.type = 100;
   msgsnd(msqid, (void *)&msg, sizeof(msg) - sizeof(long),
IPC_NOWAIT);
```

El padre recibirá este mensaje y evaluará las banderas de msg.mensaje. Si `msg.mensaje[1]` está encendida, entonces se apaga la bandera trabajando y ya no se debe coordinar la lectura pues ya terminó. Si `msg.mensaje[3]` está encendida es porque el proceso emisor terminó de analizar y no está trabajando; el padre revisa si alguno otro está trabajando y si sí, sigue esperando a que todos terminen para parar.

```
 while (trabajando == 1 || hijosTrabajando == 1) {
   msgrcv(msqid, &msg, sizeof(msg), 100, 0);
   if (msg.mensaje[1] == 1) { //El archivo se terminó de leer
     trabajando = 0;
   } else if (msg.mensaje[3] == 1) { //Un hijo terminó de analizar
     hijos[msg.mensaje[4] - 1] = 0;
     hijosTrabajando = 0;
     for (int j = 0; j < NUM_PROCESS-1; j++) { //Revisa si otro aún está
trabajando
       if (hijos[j] == 1) {
         hijosTrabajando = 1;
       }
      }
   } else { //Coordina la lectura del archivo
     hijos[i - 1] = 1;
     i++;
     if (i == NUM_PROCESS) {
       i = 1;
```

```
    }
    msg.type = i;
    actualPosition = msg.mensaje[0];
    msg.mensaje[0] = actualPosition;
    msgsnd(msqid, (void *)&msg, sizeof(msg) - sizeof(long), IPC_NOWAIT);
  }
}
```

Si ambas banderas están apagadas, el padre manda un mensaje al siguiente proceso del pool `hijos` con la posición desde la cual debe empezar a leer. El tipo del mensaje es el `i` identificador del proceso.

Volviendo a los lectores, cuando un lector encuentra una coincidencia, envía un mensaje al pintor (`msg.type=440`) y copia en msg.contenido el texto que debe imprimir el pintor.

```
  msg.mensaje[2] = 0;
  msg.mensaje[1] = 0;
  msg.mensaje[3] = 0;
  msg.mensaje[4] = i;
  msg.type = 440;
  // Copiamos el contenido al mensaje para que el padre lo
imprima
  memset(msg.contenido, 0, sizeof(msg.contenido));
  strncpy(msg.contenido, inicioLinea + pmatch[0].rm_so, len);
  msgsnd(msqid, (void *)&msg, sizeof(msg) - sizeof(long),
IPC_NOWAIT);
```

El pintor va a estar esperando a recibir mensajes de tipo 440 hasta que el padre le indique que termine.

```
while (bandera == 1){
  if (msgrcv(msqid, &msg, sizeof(msg), 440, IPC_NOWAIT) != -1){
    if (msg.mensaje[1] == 1) {
      bandera = 0;
    }
    printf("%s\n", msg.contenido);
    msgctl(msqid,IPC_STAT,&info);
   }
}
```

```
fclose(fp);
exit(0);
```

El lector, tras evaluar todo su buffer y haber encontrado todas las coincidencias, envía un mensaje al padre (`msg.type=100`) indicando que está listo para volver a leer con la bandera `msg.mensaje[3]=1`.

```
//Hijo listo para volver a leer
msg.mensaje[2] = 0;
msg.mensaje[1] = 0;
msg.mensaje[3] = 1;
msg.mensaje[4] = i;
msg.type = 100;
msgsnd(msqid, (void *)&msg, sizeof(msg) - sizeof(long), 0);
```

Al final, cuando todo el archivo haya sido leído y todos los lectores hayan terminado de analizar sus buffers, el padre envía un mensaje vacío a todos los lectores y al pintor con la bandera `msg.mensaje[1]` encendida, indicando que deben terminar para luego terminar el programa.

```
  //Notifica a los hijos lectores que ya pueden terminar (exit/morir)
  for (int j = 1; j < NUM_PROCESS; j++) {
      msg.type = j;
      msg.mensaje[1] = 1;
      msgsnd(msqid, (void *)&msg, sizeof(msg) - sizeof(long), 0);
  }
  //Notifica al hijo pintor que ya puede terminar (exit/morir)
  msg.type = 440;
  msg.mensaje[1] = 1;
  msgsnd(msqid, (void *)&msg, sizeof(msg) - sizeof(long), 0);
```

6. Pruebas de rendimiento

El comando `time` en los sistemas Linux muestra el tiempo de ejecución de un programa en segundos tras terminar. Se obtienen tres resultados de este comando:

- `real`: el tiempo que pasó desde que se inicia el programa hasta que termina, es el tiempo que percibe la persona usuaria.
- `user`: el tiempo de CPU que se ejecutó en modo usuario.
- `sys`: el tiempo de CPU que se ejecutó en modo kernel.

Para efectos de comparación se usará el tiempo `sys`, debido a que el tiempo `real` y `user` se pueden ver muy afectados por procesos ajenos al programa. Además, es importante destacar que en cada ejecución el tiempo puede aumentar o disminuir levemente, pero siempre arroja un resultado similar.

Propiedades de la máquina de prueba

- Tipo de máquina: Virtual
- Sistema operativo: Linux Mint 21.2
- Procesador: 2 núcleos del procesador físico
- Procesador físico: AMD Ryzen 5 5600G
- Memoria principal: 8 GB

Prueba #1

Texto: Don Quijote de la Mancha (2130394 caracteres). Tiempos en segundos.

Patrón	Procesos				
	2	10	20	30	50
"Dulcinea"	0.029	0.012	0.016	0.013	0.014
"de l.s otr.s"	0.022	0.015	0.013	0.015	0.020
"Ham*e"	0.026	0.015	0.013	0.017	0.014

A partir de estos resultados se concluye que el número óptimo de procesos está entre 2 y 10. La reducción de tiempo con más de 10 procesos es despreciable. Recordemos que con 2 procesos tenemos un lector y un pintor, el mínimo que necesita el programa para funcionar, pero con un único lector el proceso es más lento que con varios.

Prueba #2

Texto: La Divina Comedia (725979 caracteres). Tiempos en segundos.

Es importante aclarar que el tamaño del archivo de La Divina Comedia utilizado es casi tres veces más pequeño que el de Don Quijote de la Mancha.

Patrón	Procesos			
	4	6	8	10
"espíritu"	0.011	0.009	0.008	0.008
"est.s"	0.014	0.009	0.007	0.009
"para.*s"	0.009	0.004	0.005	0.005

A partir de estos resultados se concluye que el número óptimo de procesos es 6. La diferencia entre 4 y 6 procesos es considerable, pero entre 6, 8 y 10 procesos la diferencia de tiempo es despreciable y las variaciones se deben probablemente a factores externos al programa. Como son 6 procesos hijos, entonces tenemos 5 lectores y el pintor.

7. Conclusiones

- C ofrece muchas más librerías y funciones en los ambientes Unix que en un sistema Windows, por ejemplo regex.h y las funciones fork y las relacionadas a la comunicación por cola (msgsnd, msgrcv, msgctl).
- C es un lenguaje que tiene algunas carencias que hacen que sea incómodo de usar como no tener un tipo booleano o strings de forma nativa.
- A pesar de no tener strings de forma nativa, C cuenta con varias funciones para manejar los punteros a caracteres como `strtok` y `strcpy`.
- La programación con multiprocesos en definitiva reduce el tiempo de ejecución de un programa. Es importante estructurar cuidadosamente la comunicación entre procesos para lograr mejores resultados.
- Trabajar con una máquina virtual limita bastante el rendimiento de la ejecución, pero es una opción para usar un sistema operativo distinto sin tener que fragmentar el disco duro.
- La comunicación por una cola de procesos soluciona problemas de interbloqueo y tras realizar este proyecto consideramos que sí era una mejor opción que usar memoria compartida.
- A diferencia de los hilos, los procesos no comparten memoria y por ello el proceso de comunicación es más complicado; pero la coordinación entre procesos es más sencilla que coordinar hilos.