DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico II

Pthreads

Sistemas Operativos

Primer Cuatrimestre de 2017

Integrante	LU	Correo electrónico
Alem Santiago	650/14	santialem.trev@gmail.com
Alliani Federico	183/15	fedealliani@gmail.com
Raposeiras Lucas	034/15	lucas.raposeiras@outlook.com

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2160 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (54 11) 4576-3359
http://www.fcen.uba.ar

Resumen

El objetivo de este Trabajo Práctico consiste en comprender el uso y funcionamiento de los *threads* (en particular de la API *pthreads*) y manejar tareas concurrentemente sin tener problemas como *race conditions*, *deadlocks*, etc.

En este informe explicaremos cómo realizamos cada ejercicio del Trabajo Práctico, así como también algunas conclusiones obtenidas.

Índice

1. Ejercicio 1 **3**

2. Ejercicio 2 **4**

3. Ejercicio 3 **5**

4. Ejercicio 4 **5**

5. Ejercicio 5 **6**

6. Ejercicio 6 **6**

1. Ejercicio 1

Función `void push_front(const T& val)`

Para que la lista pueda ser utilizada por distintos *threads* la implementación de la misma debe agregar elementos para manejar la concurrencia. En concreto, cuando se quiere agregar un nuevo elemento a la lista usando la función `void push_front(const T& val)`, es necesario modificar la estructura interna de la misma, lo cual implica hacer uso de la *exclusión mutua*.

Debido a esto, declaramos la variable `mutex_lista` del tipo `pthread_mutex_t` dentro de los *miembros privados* de la clase `Lista`. La misma se inicializa en la función `Lista()` (constructor de `Lista`) mediante la función `int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *attr)`.

Explicación de la implementación

Primero se crea el nuevo nodo a ser agregado, pasándole como parámetro al constructor de `Nodo` el valor indicado en el parámetro `val` de `push_front`. Luego se hace un *lock* en el *mutex* de la lista debido a que se comenzará a modificar la estructura interna de la misma. Una vez *bloqueado* se puede modificar la estructura interna, en este caso haciendo que el nodo siguiente al nuevo nodo sea la actual *cabeza* de la lista, y que la nueva *cabeza* de la lista pase a ser el nuevo nodo. Por último, se *desbloquea* el *mutex*.

Constructor `ConcurrentHashMap()`

La representación del `ConcurrentHashmap` se realiza sobre la variable `Lista<pair<string, unsigned int>>* tabla[26]`, la cual contiene una `Lista` por cada letra del abecedario. Por otro lado, para garantizar la concurrencia entre los elementos de cada lista, se utilizarán 26 *mutex* (uno por cada lista) los cuales se encuentran en el el arreglo `mutexes[26]` de tipo `pthread_mutex_t`.

Explicación de la implementación

El constructor crea espacio para cada *Lista* (una por cada letra del abecedario) y además inicializa los *mutex* correspondientes a cada una de ellas.

Función `void addAndInc(string key)`

Explicación de la implementación

El primer paso consiste en reconocer a qué lista corresponde la palabra pasada por el parámetro `key`. Esto se realiza con la función auxiliar `unsigned int mapCharToInt(char a)`, la cual recibe como parámetro el primer `char` de `key` y devuelve un `unsigned int` entre 0 y 25.

Una vez obtenido el número de lista (entre 0 y 25) a la cual corresponde la palabra del parámetro `key`, se realiza una búsqueda lineal en la misma para comprobar si la palabra ya se encuentra definida. Si la misma ya estaba definida, se incrementa la cantidad de apariciones, y en caso contrario, se crea un nuevo elemento en la lista de tipo `pair<string, unsigned int>` con los valores `(key, 1)`, representando que la palabra del parámetro `key` contiene exactamente una aparición a partir de ahora, y dicho elemento es agregado a la lista usando la función `push_front`.

Para evitar *race conditions*, se decide *bloquear* el *mutex* correspondiente a la lista de la palabra del parámetro `key` antes de realizar la búsqueda y *desbloquearlo* luego de modificar la lista correspondiente a dicha palabra.

Función `bool member(string key)`

Explicación de la implementación

El primer paso consiste en reconocer a qué lista correspondería la palabra pasada por el parámetro `key`. Esto se realiza con la función auxiliar `unsigned int mapCharToInt(char a)`, la cual recibe como parámetro el primer `char` de `key` y devuelve un `unsigned int` entre 0 y 25.

Una vez obtenido el número de lista (entre 0 y 25) a la cual correspondería la palabra del parámetro `key`, se realiza una búsqueda lineal en la misma para comprobar si existe algún `pair<string, unsigned int>` donde la primer componente sea igual a `key`. Si esto ocurre, se retorna `true`, y en caso contrario, se retorna `false`.

Función `pair<string, unsigned int> maximum(unsigned int nt)`

Para que los distintos *threads* compartan información entre sí, creamos la estructura `maximum_struct`, la cual contendrá información sobre el `ConcurrentHashmap` actual, *mutex* a utilizar entre los *threads*, cantidad de filas sin procesar, etc.

Explicación de la implementación

Primero se declara una variable de tipo `struct maximum_struct` la cual es inicializada con los valores necesarios para indicar a los *threads* que se deben procesar todas las filas. Además, se crea un arreglo de tamaño `nt` (parámetro de `maximum`) de tipo `pthread_t`. A continuación se crean y ejecutan los `nt` *threads*, indicando que la función a invocar será la función auxiliar `static void *maximumThread(void* data)`, y que el parámetro a recibir es un puntero a la estructura anteriormente mencionada.

La función auxiliar `maximumThread` toma como parámetro un `void*` que inmediatamente es *casteado* a `maximum_struct*`. Dicha función *bloquea* el *mutex* de la estructura y controla si hay alguna lista sin procesar. Si esto ocurre, el *thread* actual se encargará de procesar la lista que aún no se ha procesado. Esto se hace con la función auxiliar `void *maximumFila(unsigned int ind, pair<string,unsigned int>* arreglo)`. Es importante señalar que, debido al *bloqueo* que se hace en el *mutex* de la estructura, no habrá ningún tipo de inconveniente al consultar las variables internas de la misma, aún sabiendo que la misma es compartida entre todos los `threads`.

La función auxiliar `maximumFila` busca la palabra con mayor cantidad de apariciones dentro de la lista cuyo índice es el indicado en el parámetro `ind`, y almacena el `pair<string, unsigned int>` resultante en la posición correspondiente al índice `ind` del arreglo indicado en el parámetro arreglo. Para evitar concurrencia con `addAndInc`, se utilizan los *mutex* de la clase `ConcurrentHashMap`.

Finalmente, en la función `maximum`, se espera a que terminen todos los *threads* con `pthread_join`. Que todos los *threads* hayan finalizado su ejecución significa que no quedan filas a procesar. Una vez finalizados los *threads* se recorre el arreglo donde se guardó cada `pair` máximo de las 26 filas para encontrar la palabra con mayor cantidad de repeticiones.

Una vez encontrada, la misma es retornada.

2. Ejercicio 2

Función `ConcurrentHashMap count_words(string arch)`

Explicación de la implementación

Esta función recibe como parámetro el nombre del archivo a procesar. Este archivo será procesado sin concurrencia y creará un nuevo `ConcurrentHashMap`.

La implementación consiste en crear una nueva instancia de `ConcurrentHashMap` y luego leer, línea por línea, el archivo del parámetro `arch`, hasta que se alcance el final del mismo. Por la naturaleza de la

composición de los archivos recibidos, cada línea de texto contiene una única palabra, la cual es insertada en el nuevo `ConcurrentHashMap` mediante el uso de la función `addAndInc`.

3. Ejercicio 3

Función `ConcurrentHashMap count_words(list<string> archs)`

Explicación de la implementación

En esta implementación de `count_words` se recibe como parámetro una lista con los nombres de los archivos a procesar. Se utilizará un *thread* por archivo, con lo cual los *threads* deberán compartir cierta información entre ellos. Dicha información se resume en un arreglo de tipo `struct datos_process_file`, el cual contiene el nombre de cada archivo y un puntero al `ConcurrentHashMap` actual.

Cada *thread* agrega las palabras del archivo al `ConcurrentHashmap` pasado en la estructura de manera similar a la función `count_words` del Ejercicio 2.

4. Ejercicio 4

Función `ConcurrentHashMap count_words(unsigned int n, list<string> archs)`

Explicación de la implementación

En esta implementación de `count_words` se recibe como parámetro la cantidad de *threads* a utilizar y la lista de archivos a procesar. El primer paso es crear un `ConcurrentHashMap` vacío para luego agregar todas las palabras de los distintos archivos. Para compartir información entre los distintos *threads*, se define la estructura `count_words_n_threads_struct`, la cual se instancia en la variable `estructura`. Esta estructura contiene la siguiente información:
- La lista de los nombres de los archivos a procesar.
- La cantidad de archivos sin procesar.
- El puntero al `ConcurrentHashMap` creado, para que se agreguen en el mismo `ConcurrentHashMap`.
- Un *mutex* que será utilizado para modificar las variables de esta estructura sin sufrir problemas de concurrencia.

Una vez inicializada esta estructura, se procede a crear y ejecutar los *threads*, indicando que la función a invocar será la función auxiliar `static void *countWordsAuxiliarNThreads(void* estruc)`, y que el parámetro a recibir es un puntero a la estructura anteriormente mencionada

La función `countWordsAuxiliarNThreads` consta de un ciclo que se mantiene iterando mientras haya archivos por procesar. Esta función primero *bloquea* el *mutex* de la estructura para consultar si hay algún archivo disponible para procesar. Aquí se debe *bloquear* el *mutex* ya que la lista de archivos sin procesar es común para todos los threads. Si hay algún archivo sin procesar, se obtiene el primero de ellos y se lo elimina de la lista. Luego se *desbloquea* el *mutex* para realizar el procesado del archivo llamando a la función auxiliar `thread_process_file`, que realiza la misma funcionalidad que la función `count_words` del Ejercicio 2, pero con la salvedad que se le pasa como parámetro el `concurrentHashMap` a ser modificado. Si no hay más archivos para procesar, finaliza el *thread*.

La función `thread_process_file` trabaja concurrentemente con los otros *threads*, pues utiliza la función `addAndInc`, la cual está implementada para trabajar concurrentemente.

Por último, en la función `count_words` se espera a que terminen todos los *threads*, y se retorna el `ConcurrentHashMap` creado con las palabras de todos los archivos.

5. Ejercicio 5

Función `pair<string, unsigned int> maximum(unsigned int p_archivos, unsigned int p_maximos, list<string> archs)`

Explicación de la implementación

En esta función se recibe como parámetro la cantidad de *threads* a utilizar para leer los archivos (`p_archivos`), la cantidad de *threads* a utilizar para calcular máximos (`p_maximos`) y la lista de archivos a procesar (`archs`). Cada *thread* creará un `ConcurrentHashMap` por cada archivo utilizando la versión no concurrente de `count_words`.

Al principio de la función se utiliza una estructura llamada `datos_multiple_hashmap` que será compartida por todos los *threads*. Esta estructura contiene la lista de archivos a procesar, un `vector` con los `ConcurrentHashMap` creados por cada *thread* y un *mutex* que se utilizará para la modificación de esta estructura.

Cada *thread* saca un elemento de la lista de archivos a procesar (utilizando apropiadamente el *bloqueo* del *mutex*), si es que todavía quedan archivos sin procesar. Si aún quedan archivos por procesar, se procesa utilizando la función `count_words` no concurrente. Luego de procesar el archivo, guarda el `ConcurrentHashMap` resultante en el vector de `ConcurrentHashMap` procesados. Cada *thread* se ejecutará hasta que se hayan procesado todos los archivos.

Una vez que todos los *threads* hayan procesado todos los archivos, se realiza un *merge* de los `ConcurrentHashMap` creados utilizando la función `void merge(ConcurrentHashMap* source)`, que agrega los elementos del `ConcurrentHashMap` pasado por parámetro al `ConcurrentHashMap` pasado implícitamente (`this`). Una vez finalizados todos los *merges*, se tiene como resultado un `ConcurrentHashMap` con todas las palabras. Luego se llama a la función *maximum* de dicho `ConcurrentHashMaps`, pasándole como parametro la cantidad de *threads* `p_maximos`.

Por ultimo, se retorna el par máximo devuelto por la función `maximum`.

6. Ejercicio 6

Explicación de la implementación

En este ejercicio se utiliza la misma implementación del ejercicio 5, pero con la versión concurrente de `count_words`.

Comparación de resultados

Para comparar los resultados realizamos un *test*, el cual compara la cantidad de tiempo que tarda la función `maximum` (Ejercicio 5) y la función `maximum2` (Ejercicio 6) con los mismos parámetros de entrada.

Los resultados obtenidos son los siguientes:

	p_archivos=1	**p_archivos=2**	**p_archivos=4**	**p_archivos=8**
p_maximos=1	14 593 609 343	11 752 632 666	8 804 726 736	9 819 379 066
p_maximos=2	14 713 620 479	12 331 453 337	9 159 324 633	11 399 697 736
p_maximos=4	14 835 799 742	11 759 645 475	9 262 119 470	12 203 685 806
p_maximos=8	15 324 590 217	11 613 811 748	8 981 544 344	8 583 646 426

Cuadro 1: Tabla de valores de tiempo en nanosegundos (ns) utilizando la función `maximum` (Ejercicio 5).

	p_archivos=1	**p_archivos=2**	**p_archivos=4**	**p_archivos=8**
p_maximos=1	9 809 264 980	6 871 078 097	4 175 067 807	5 416 394 546
p_maximos=2	10 034 566 387	7 265 990 847	4 087 496 479	4 537 711 316
p_maximos=4	9 613 755 799	6 773 415 116	4 210 014 678	3 875 106 174
p_maximos=8	9 691 172 420	6 672 397 190	4 011 369 259	4 135 598 422

Cuadro 2: Tabla de valores de tiempo en nanosegundos (ns) utilizando la función `maximum2` (Ejercicio 6).

	p_archivos=1	**p_archivos=2**	**p_archivos=4**	**p_archivos=8**
p_maximos=1	4 784 344 363	4 881 554 569	4 629 658 929	4 402 984 520
p_maximos=2	4 679 054 092	5 065 462 490	5 071 828 154	6 861 986 420
p_maximos=4	5 222 043 943	4 986 230 359	5 052 104 792	8 328 579 632
p_maximos=8	5 633 417 797	4 941 414 558	4 970 175 085	4 448 048 004

Cuadro 3: Tabla de valores en nanosegundos (ns) de la diferencia de tiempo entre la función `maximum` (Ejercicio 5) y la función `maximum2` (Ejercicio 6).