



ΙΟΝΙΟ ΠΑΝΕΠΙΣΤΗΜΙΟ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

ΕΡΓΑΣΙΑ ΕΑΡΙΝΟΥ ΕΞΑΜΗΝΟΥ 2018

ΠΑΡΑΛΛΗΛΟΣ ΠΡΟΓΡΑΜΜΑΤΙΣΜΟΣ

ΕΡΓΑΣΙΑ #2

**–ΥΛΟΠΟΙΗΣΗ QUICKSORT ΜΕ ΧΡΗΣΗ POSIX THREADS, THREAD
POOL ΚΑΙ GLOBAL CIRCULAR BUFFER ΩΣ ΟΥΡΑ ΕΡΓΑΣΙΩΝ–**

ΥΠΕΥΘΥΝΟΣ ΚΑΘΗΓΗΤΗΣ:

ΜΙΧΑΗΛ ΣΤΕΦΑΝΙΔΑΚΗΣ

ΣΤΟΙΧΕΙΑ ΦΟΙΤΗΤΗ:

ΜΙΧΑΗΛ – ΧΡΥΣΟΒΑΛΑΝΤΗΣ ΠΑΓΚΡΑΚΙΩΤΗΣ (Α.Μ.: Π2014035)

1. Περιγραφή κώδικα:

1.1. Circular Buffer ως ουρά εργασιών

Ο **Circular Buffer** δημιουργήθηκε ως ένας πίνακας από structs. (array of structs).

Κάθε task που θα δημιουργείται θα έχει πληροφορίες για έναν πίνακα τον οποίο θέλουμε να ταξινομήσουμε. Αρχικά λοιπόν, ορίστηκε μια struct task_parameters στην οποία θα αποθηκεύονται οι παράμετροι των tasks. Αποτελείται από έναν δείκτη (που δείχνει στο πρώτο κελί του πίνακα), μια μεταβλητή που έχει το μέγεθος του πίνακα και μια μεταβλητή shutdown η οποία γίνεται ίση με 1 όταν έχει ταξινομηθεί ολόκληρος ο αρχικός πίνακας μεγέθους N.

Στη συνέχεια, έγινε η δημιουργία ενός πίνακα circular_buffer[] του οποίου το κάθε ένα κελί αντιπροσωπεύει ένα task και αποτελείται από ένα struct task_parameters με τις παραμέτρους που το περιγράφουν.

Επειδή θέλουμε ο buffer να είναι κυκλικός, χρειαζόμαστε δυο μεταβλητές cbuf_in και cbuf_out οι οποίες κρατάνε τον αριθμό του κελιού του buffer στο οποίο μπορούμε να προσθέσουμε ένα task και τον αριθμό του κελιού από το οποίο μπορούμε να αφαιρέσουμε ένα task αντίστοιχα.

Προκειμένου να προσθέσουμε ένα νέο task στον circular buffer υπάρχει η συνάρτηση putNewTask και για να αφαιρέσουμε ένα task από τον buffer η getNewTask.

1.2. putNewTask()

Η **putNewTask** είναι τύπου void και δέχεται σαν ορίσματα έναν δείκτη προς το πρώτο κελί του πίνακα, το μέγεθός του και μια μεταβλητή shutdown.

Πριν περαστούν τα νέα στοιχεία στον buffer γίνεται έλεγχος για το αν ο buffer είναι ήδη γεμάτος. Αν είναι γεμάτος, με την εντολή pthread_cond_wait(&msg_out,&mutex); περιμένουμε μέχρι κάποιο μήνυμα (task) να αφαιρεθεί ώστε να απελευθερώσει χώρο και να γίνει signal η conditional variable msg_out.

Όταν υπάρξει ελεύθερος χώρος, τα δεδομένα αποθηκεύονται στον buffer, η cbuf_in δείχνει στο επόμενο κενό κελί, ο αριθμός διαθέσιμων μηνυμάτων αυξάνεται κατά 1 και κάνουμε signal την msg_in.

1.3 getNewTask()

Η **getNewTask**() είναι τύπου int, δέχεται σαν ορίσματα έναν διπλό δείκτη (ο οποίος δείχνει σε δείκτη προς το πρώτο κελί πίνακα) και έναν δείκτη ο οποίος δείχνει σε μεταβλητή με το μέγεθος πίνακα.

Πριν αποθηκεύσει τα δεδομένα του buffer στις μεταβλητές από τα ορίσματα, γίνεται έλεγχος για το

αν ο buffer είναι κενός. Αν είναι κενός, με την εντολή `pthread_cond_wait(&msg_in,&mutex);` περιμένουμε μέχρι κάποιο μήνυμα (task) να προστεθεί ώστε να γίνει signal η conditional variable `msg_in`.

Όταν προστεθεί ένα task, τα δεδομένα λαμβάνονται από τον buffer και γίνεται ακόμα ένας έλεγχος. Αν το task που επιστράφηκε έχει τιμή `shutdown == 1`, σημαίνει ότι ο αρχικός πίνακας είναι ταξινομημένος οπότε σταματάει η εκτέλεση και γίνεται return η τιμή 0.

Αλλιώς, αποθηκεύουμε τα δεδομένα του task, η `cbuf_out` δείχνει στο επόμενο γεμάτο κελί, ο αριθμός διαθέσιμων μηνυμάτων μειώνεται κατά 1 και κάνουμε signal την `msg_out`.

1.4. inssort(), quicksort()

Η συνάρτηση **inssort()** στο github και στον κώδικα του εργαστηρίου ήταν void, όμως την μετέτρεψα σε int, έτσι ώστε όταν ο αρχικός πίνακας είναι ταξινομημένος, να επιστρέφει τη τιμή 1. Η συνάρτηση **quicksort()** επίσης ήταν void και μετατράπηκε σε int ώστε να επιστρέφει την τιμή 1, όταν η inssort() της επιστρέψει τη τιμή 1.

1.5. thread_func()

Η **thread_func()** είναι η συνάρτηση την οποία θα εκτελέσουν τα threads όταν αυτά δημιουργηθούν στη main. Έχει μια επανάληψη `for(;;)` η οποία εκτελείται “για πάντα”. Μέσα σε αυτή τη for καλεί τη συνάρτηση `getNewTask()` για να πάρει το επόμενο προς εκτέλεση task.

- Αν η συνάρτηση επιστρέψει τη τιμή 1, σημαίνει ότι επιστράφηκε με επιτυχία ένας πίνακας ο οποίος, ανάλογα με το μέγεθός του:
 - Αν είναι μεγαλύτερος από ένα όριο (LIMIT) θα χωριστεί στη μέση με τη βοήθεια της `partition()` και θα προστεθούν δυο νέα tasks στον buffer, ένα για το πρώτο και ένα για το δεύτερο μισό του πίνακα.
 - Αν είναι μικρότερος από ένα όριο (LIMIT) θα κληθεί η `quicksort()`, η οποία θα επιστρέψει μια τιμή. Αν επιστραφεί η τιμή 1, σημαίνει ότι η ταξινόμηση του αρχικού πίνακα έχει ολοκληρωθεί, οπότε γίνεται η προσθήκη στον buffer, ενός task για κάθε ένα thread, με μήνυμα `shutdown`.
(Αριθμός μηνυμάτων `shutdown` που στέλνονται = αριθμός THREADS).
- Αν η `getNewTask()` επιστρέψει τη τιμή 0, σημαίνει ότι το task που επιστράφηκε είχε μήνυμα `shutdown`, οπότε εκτελείται εντολή “break;” για να τερματιστεί η εκτέλεση της `for(;;)`.

Όταν τερματιστεί η for, με την εντολή “`pthread_exit(NULL)`” λέμε στα threads να

απενεργοποιηθούν και να περιμένουν να γίνουν join.

1.6. main()

Αρχικά γίνεται δέσμευση μνήμης N θέσεων τύπου double για τον πίνακα array και ο πίνακας γεμίζει με τυχαίες τιμές.

Ορίζεται το thread pool ως ένας πίνακας τύπου pthread_t.

Με την εντολή “pthread_create(&threadPool[thrnum], NULL, thread_func, NULL)” δημιουργούνται τα threads, των οποίων τα thread IDs αποθηκεύονται στο threadPool[]. Μόλις δημιουργείται ένα thread ξεκινάει την εκτέλεση του στη συνάρτηση thread_func().

Αφού δημιουργηθούν τα threads προστίθεται το πρώτο task στον buffer το οποίο περιέχει δείκτη στο πρώτο κελί του πίνακα array, μέγεθος N και φυσικά shutdown == 0;

Μετά την εκτέλεση της pthread_exit(NULL), κάθε ένα από τα threads του threadPool γίνεται join.

Γίνεται έλεγχος για το αν ο αρχικός πίνακας είναι όντως σωστά ταξινομημένος και εμφανίζεται κατάλληλο μήνυμα.

Τέλος γίνεται αποδέσμευση του πίνακα και του mutex που χρησιμοποιήθηκε για το “κλείδωμα” ορισμένων κρίσιμων περιοχών στον κώδικα ώστε να έχει πρόσβαση σε αυτές μόνο ένα thread τη φορά.

1.7. Έλεγχος ολοκλήρωσης ταξινόμησης και global_n_counter

Για να ελέγξω αν ο αρχικός πίνακας είναι ταξινομημένος έκανα το εξής:

- Όρισα μια global μεταβλητή τυπου int και την ονόμασα **global_n_counter**.
- Η global_n_counter μετράει το μέγεθος του κομματιού του αρχικού πίνακα το οποίο έχει ήδη ταξινομηθεί από την inssort().
- Κάθε φορά λοιπόν που καλείται η inssort() για ένα πίνακα με μέγεθος n, η τιμή της μεταβλητής global_n_counter αυξάνεται κατά n.
- Όταν η τιμή της global_n_counter γίνει ίση με το μέγεθος του αρχικού πίνακα (N), σημαίνει ότι όλος ο πίνακας είναι ταξινομημένος.
- Όταν γίνει αυτό, η inssort() επιστρέφει τη τιμή 1 στην quicksort() και αυτή με τη σειρά της επιστρέφει τη τιμή 1 στη thread_func().
- Όταν η thread_func() λάβει τη τιμή 1 από τη quicksort() στέλνει μήνυμα shutdown.

2. Παρατήρηση:

Στην εκφώνηση ζητήθηκε:

- το άθροισμα των ταξινομημένων στοιχείων του πίνακα
- ο έλεγχος για το αν ο πίνακας είναι ταξινομημένος
- και η αποστολή μηνύματος shutdown

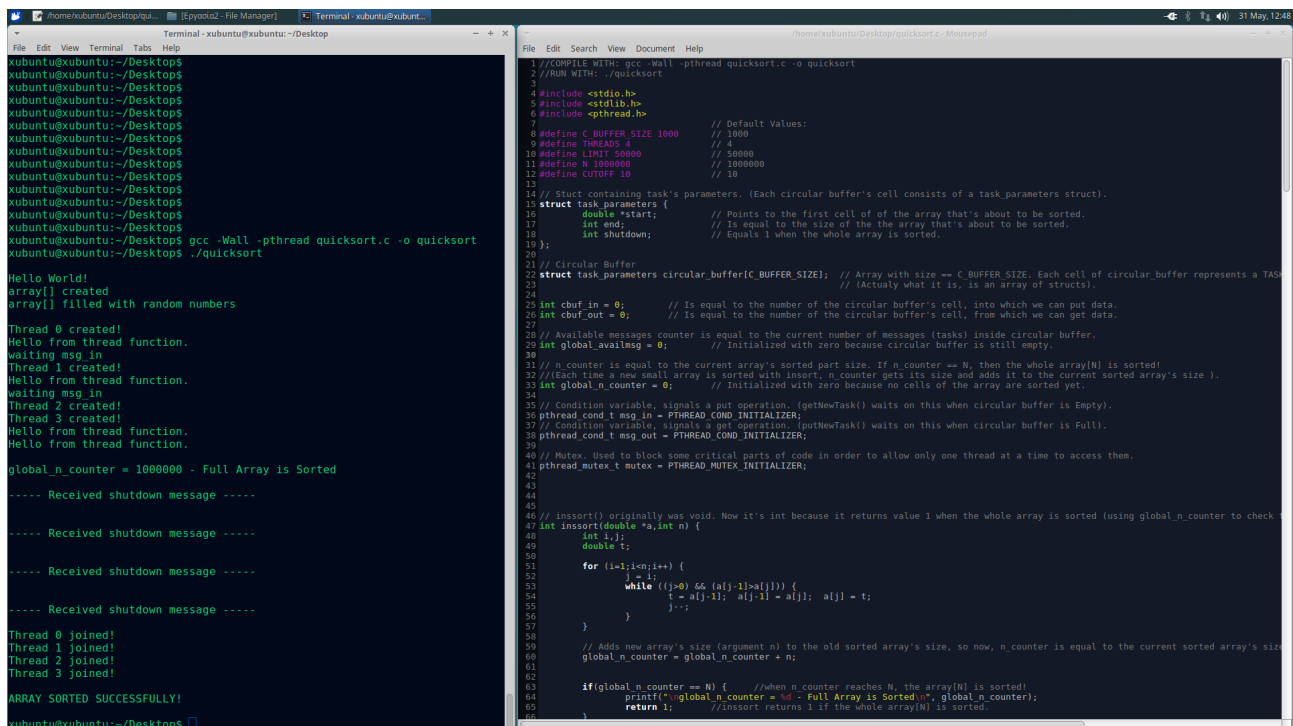
να γίνονται από το κεντρικό thread, στη main().

Στον κώδικά μου όμως, η main() το μόνο που κάνει είναι να προσθέτει το αρχικό task με τον αρχικό πίνακα μεγέθους N στον buffer για να αρχίσει η διαδικασία της ταξινόμησης.

Το άθροισμα των ταξινομημένων στοιχείων του πίνακα και ο έλεγχος για το αν ο πίνακας είναι ταξινομημένος γίνεται μέσα στην insort().

Η αποστολή shutdown μηνύματος γίνεται από τη thread_func().

3. Ενδεικτικές Οθόνες:



```
Terminal - xubuntu@xubuntu:~/Desktop
xubuntu@xubuntu:~/Desktop$
xubuntu@xubuntu:~/Desktop$
xubuntu@xubuntu:~/Desktop$
xubuntu@xubuntu:~/Desktop$
xubuntu@xubuntu:~/Desktop$
xubuntu@xubuntu:~/Desktop$
xubuntu@xubuntu:~/Desktop$
xubuntu@xubuntu:~/Desktop$
xubuntu@xubuntu:~/Desktop$
xubuntu@xubuntu:~/Desktop$ gcc -Wall -pthread quicksort.c -o quicksort
xubuntu@xubuntu:~/Desktop$ ./quicksort

Hello World!
array[] created
array[] filled with random numbers

Thread 0 created!
Hello from thread function.
Waiting msg in
Thread 1 created!
Hello from thread function.
Waiting msg in
Thread 2 created!
Hello from thread function.
Waiting msg in
Thread 3 created!
Hello from thread function.
Waiting msg in
Thread 4 created!
Hello from thread function.
Waiting msg in

global_n_counter = 1000000 - Full Array is Sorted

----- Received shutdown message -----

----- Received shutdown message -----

----- Received shutdown message -----

----- Received shutdown message -----

Thread 0 joined!
Thread 1 joined!
Thread 2 joined!
Thread 3 joined!
Thread 4 joined!

ARRAY SORTED SUCCESSFULLY!
xubuntu@xubuntu:~/Desktop$
```

```
quicksort.c
1 // COMPILER WITH: gcc -Wall -pthread quicksort.c -o quicksort
2 // RUN WITH: ./quicksort
3
4 #include <stdio.h>
5 #include <stdlib.h>
6 #include <pthread.h>
7
8 #define C_BUFFER_SIZE 1000 // Default Values:
9 #define THREADS 4 // 4
10 #define LIMIT 50000 // 50000
11 #define N 1000000 // 1000000
12 #define CUTOFF 10 // 10
13
14 // Struct containing task's parameters. (Each circular buffer's cell consists of a task parameters struct).
15 struct task_parameters {
16     double *start; // Points to the first cell of of the array that's about to be sorted.
17     int end; // Is equal to the size of the array that's about to be sorted.
18     int shutdown; // Equals 1 when the whole array is sorted.
19 };
20
21 // Circular Buffer
22 struct task_parameters circular_buffer[C_BUFFER_SIZE]; // Array with size == C_BUFFER_SIZE. Each cell of circular_buffer represents a TASK
23 // (Actually what it is, is an array of structs).
24
25 int cbuf_in = 0; // Is equal to the number of the circular buffer's cell, into which we can put data.
26 int cbuf_out = 0; // Is equal to the number of the circular buffer's cell, from which we can get data.
27
28 // Available messages counter is equal to the current number of messages (tasks) inside circular buffer.
29 int global_availmsg = 0; // Initialized with zero because circular buffer is still empty.
30
31 // n_counter is equal to the current array's sorted part size. If n_counter == N, then the whole array[N] is sorted!
32 // Each time a new small array is sorted with insort, n_counter gets its size and adds it to the current sorted array's size.
33 int global_n_counter = 0; // Initialized with zero because no cells of the array are sorted yet.
34
35 // Condition variable, signals a put operation. (getNewTask() waits on this when circular buffer is Empty).
36 pthread_cond_t msg_in = PTHREAD_COND_INITIALIZER;
37 // Condition variable, signals a get operation. (putNewTask() waits on this when circular buffer is Full).
38 pthread_cond_t msg_out = PTHREAD_COND_INITIALIZER;
39
40 // Mutex. Used to block some critical parts of code in order to allow only one thread at a time to access them.
41 pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
42
43
44
45
46 // insort() originally was void. Now it's int because it returns value 1 when the whole array is sorted (using global_n_counter to check)
47 int insort(double *a, int n) {
48     int i, j;
49     double t;
50
51     for (i=1; i<n; i++) {
52         j = i;
53         while ((j>0) && (a[j-1]>a[j])) {
54             t = a[j-1]; a[j-1] = a[j]; a[j] = t;
55             j--;
56         }
57     }
58
59     // Adds new array's size (argument n) to the old sorted array's size, so now, n_counter is equal to the current sorted array's size
60     global_n_counter = global_n_counter + n;
61
62     if (global_n_counter == N) { // When n_counter reaches N, the array[N] is sorted!
63         printf("global_n_counter = %d - Full Array is Sorted!\n", global_n_counter);
64         return 1; // insort returns 1 if the whole array[N] is sorted.
65     }
66 }
```

```
File Edit View Terminal Tabs Help
Terminal - xubuntu@xubuntu: ~/Desktop
xubuntu@xubuntu:~/Desktop$ gcc -Wall -pthread quicksort.c -o quicksort
xubuntu@xubuntu:~/Desktop$ ./quicksort

Hello World!
array[] created
array[] filled with random numbers

Thread 0 created!
Hello from thread function.
waiting msg in
Thread 1 created!
Hello from thread function.
waiting msg in
Thread 2 created!
Thread 3 created!
Hello from thread function.
Hello from thread function.

global_n_counter = 1000000 - Full Array is Sorted

----- Received shutdown message -----

----- Received shutdown message -----

----- Received shutdown message -----

----- Received shutdown message -----

Thread 0 joined!
Thread 1 joined!
Thread 2 joined!
Thread 3 joined!

ARRAY SORTED SUCCESSFULLY!
xubuntu@xubuntu:~/Desktop$
```

```
File Edit View Terminal Tabs Help
Terminal - xubuntu@xubuntu: ~/Desktop
xubuntu@xubuntu:~/Desktop$ gcc -Wall -pthread quicksort.c -o quicksort
xubuntu@xubuntu:~/Desktop$ ./quicksort
```

```
----- Received shutdown message -----  
  
----- Received shutdown message -----  
  
----- Received shutdown message -----  
  
----- Received shutdown message -----  
  
Thread 0 joined!  
Thread 1 joined!  
Thread 2 joined!  
Thread 3 joined!  
  
ARRAY SORTED SUCCESSFULLY!  
xubuntu@xubuntu:~/Desktop$
```

4. Συμπέρασμα:

Παρά τη διαφορά που αναφέρθηκε παραπάνω, ο κώδικας κάνει compile χωρίς errors και ταξινομεί με επιτυχία τον πίνακα.

5. Βιβλιογραφία:

- Θεωρία και παραδείγματα σχετικά με το πως λειτουργεί ένας Circular Buffer:
 - https://en.wikipedia.org/wiki/Circular_buffer
 - <https://embeddedartistry.com/blog/2017/4/6/circular-buffers-in-c>
- Θεωρία σχετικά με C Structures:
 - <http://www.zentut.com/c-tutorial/c-structure/>
- How do you make an array of structs in C? (Τρόπος υλοποίησης Circular Buffer):
 - <https://stackoverflow.com/questions/10468128/how-do-you-make-an-array-of-structs-in-c>

- Από τα 3 αρχεία της εκφώνησης (barrier-example.c , cv-example.c , mutex-example.c) χρησιμοποίησα τα πάντα από το πως να κάνω create και join τα threads, χρήση conditional variables, δημιουργία thread pool, mutex lock/unlock κλπ, μεχρι και την ιδέα να υλοποιήσω τον circular buffer ως array of structs όπως γίνεται εδώ για τις παραμέτρους των threads: “struct thread_params tparm[THREADS];”.
 - <https://gist.github.com/mixstef/966be631a5d2601c4264>
- Από αρχείο σειριακής αναζήτησης του quicksort και τον κώδικα υλοποίησης quicksort με threads (του εργαστηρίου) πήρα τις 3 συναρτήσεις (αν και στη συνέχεια άλλαξα τη quicksort() και την inssort()), καθώς και το βασικό κομμάτι της συνάρτησης work() που εκτελούν τα threads.
 - <https://gist.github.com/mixstef/322145437c092783f70f243e47769ac6>