

ΛΕΙΤΟΥΡΓΙΚΑ ΣΥΣΤΗΜΑΤΑ ΥΠΟΛΟΓΙΣΤΩΝ

ΑΝΑΦΟΡΑ 2^{ης} ΕΡΓΑΣΤΗΡΙΑΚΗΣ ΑΣΚΗΣΗΣ



Στοιχεία Ομάδας

- Αναγνωριστικό: oslab005
- Μέλος 1^ο: Πέππας Μιχαήλ – Αθανάσιος, Α.Μ: 03121026
- Μέλος 2^ο: Σαουνάτσος Ανδρέας, Α.Μ: 03121197
- Ημερομηνία Παράδοσης Αναφοράς: 15.05.2024

▪ Ενότητα 1 – Συγχρονισμός σε υπάρχοντα κώδικα

Εισαγωγικές ερωτήσεις

Χρησιμοποιούμε το διαθέσιμο Makefile, προκειμένου να μεταγλωττίσουμε και να τρέξουμε το πρόγραμμα `simplesync`. Παρατηρούμε, ότι ενώ έχουμε αρχικοποιήσει μία μεταβλητή (`int val`) στην τιμή 0 και την αυξάνουμε κατά 1 όσες φορές τη μειώνουμε κατά 1 – 10.000.000 φορές – το τελικό αποτέλεσμα που εμφανίζεται στην οθόνη δεν είναι 0, όπως αναμενόταν. Η εξήγηση για αυτό είναι ότι οι διεργασίες που αυξάνουν και μειώνουν την τιμή της μεταβλητής εκτελούνται ταυτόχρονα – και μάλιστα το κρίσιμο τμήμα τους που κάνει (`val++`) ή (`val--`) – δεν είναι συγχρονισμένες, με αποτέλεσμα να μοιράζονται κοινή μνήμη και να αντλούν / αποθηκεύουν την νέα τιμή της σε λάθος στιγμές: πριν ολοκληρωθεί και αποθηκευτεί η αύξηση έχει εκτελεστεί αφαίρεση ή αντιστρόφως.

Ακόμα, παρατηρούμε ότι παράγονται δύο διαφορετικά εκτελέσιμα από το `simplesync.c` : το `simplesync-atomic` και το `simplesync-mutex`, μέσω του Makefile. Ο λόγος που συμβαίνει αυτό είναι αφενός μεν τα `#defined` που έχουμε κάνει μέσα στο `simplesync.c` και φαίνονται παρακάτω:

```
/* Dots indicate lines where you are free to insert code at will */
/* ... */
pthread_mutex_t lock;
#if defined(SYNC_ATOMIC) ^ defined(SYNC_MUTEX) == 0
# error You must #define exactly one of SYNC_ATOMIC or SYNC_MUTEX.
#endif

#if defined(SYNC_ATOMIC)
# define USE_ATOMIC_OPS 1
#else
# define USE_ATOMIC_OPS 0
#endif
```

αφετέρου δε, τα flags κατά τη μεταγλώττιση στο Makefile (`-DSYNC_MUTEX` και `-DSYNC_ATOMIC`), τα οποία δίνουν τιμές 1 στις mutex και atomic παραλλαγές της `simplesync`, αντίστοιχα.

Κατόπιν, επεκτείνουμε τον κώδικα της `simplesync.c` στις δύο παραλλαγές της και το νέο αρχείο `.c` φαίνεται ακολούθως. Οι αλλαγές εντοπίζονται στην `__sync_add_and_fetch` και `__sync_sub_and_fetch` που εξασφαλίζουν το atomicity των προσθέσεων και αφαιρέσεων, αντίστοιχα, και στα mutex lock/unlocks στο κρίσιμο τμήμα της αύξησης/μείωσης της μεταβλητής `val`.

Η αλλαγμένη συνάρτηση που υλοποιεί την αύξηση φαίνεται ακολούθως, ενώ όμοια λογική υπάρχει και στη συνάρτηση που υλοποιεί τη μείωση της μεταβλητής val.

```
41 void *increase_fn(void *arg)
42 {
43     int i;
44     volatile int *ip = arg;
45
46     fprintf(stderr, "About to increase variable %d times\n", N);
47     for (i = 0; i < N; i++) {
48         if (USE_ATOMIC_OPS) {
49             __sync_add_and_fetch(ip,1);
50         } else {
51             pthread_mutex_lock(&lock);
52             /* You cannot modify the following line */
53             ++(*ip);
54             pthread_mutex_unlock(&lock);
55         }
56     }
57     fprintf(stderr, "Done increasing variable.\n");
58
59     return NULL;
60 }
```

Ο συνολικός – αλλαγμένος κώδικας φαίνεται ακολούθως:

simplesync.c

```

1  /*
2  * simplesync.c
3  *
4  * A simple synchronization exercise.
5  *
6  * Vangelis Koukis <vkoukis@cslab.ece.ntua.gr>
7  * Operating Systems course, ECE, NTUA
8  *
9  */
10
11 #include <errno.h>
12 #include <stdio.h>
13 #include <stdlib.h>
14 #include <unistd.h>
15 #include <pthread.h>
16 #include <stdatomic.h>
17
18 /*
19 * POSIX thread functions do not return error numbers in errno,
20 * but in the actual return value of the function call instead.
21 * This macro helps with error reporting in this case.
22 */
23 #define perror_thread(ret, msg) \
24     do { errno = ret; perror(msg); } while (0)
25
26 #define N 10000000
27
28 /* Dots indicate lines where you are free to insert code at will */
29 /* ... */
30 pthread_mutex_t lock;
31 #if defined(SYNC_ATOMIC) ^ defined(SYNC_MUTEX) == 0
32 # error You must #define exactly one of SYNC_ATOMIC or SYNC_MUTEX.
33 #endif
34
35 #if defined(SYNC_ATOMIC)
36 # define USE_ATOMIC_OPS 1
37 #else
38 # define USE_ATOMIC_OPS 0
39 #endif
40
41 void *increase_fn(void *arg)
42 {
43     int i;
44     volatile int *ip = arg;
45
46     fprintf(stderr, "About to increase variable %d times\n", N);
47     for (i = 0; i < N; i++) {
48         if (USE_ATOMIC_OPS) {
49             __sync_add_and_fetch(ip, 1);
50         } else {
51             pthread_mutex_lock(&lock);
52             /* You cannot modify the following line */
53             ++(*ip);
54             pthread_mutex_unlock(&lock);
55         }
56     }
57     fprintf(stderr, "Done increasing variable.\n");

```

```
58
59     return NULL;
60 }
61
62 void *decrease_fn(void *arg)
63 {
64     int i;
65     volatile int *ip = arg;
66
67     fprintf(stderr, "About to decrease variable %d times\n", N);
68     for (i = 0; i < N; i++) {
69         if (USE_ATOMIC_OPS) {
70             __sync_sub_and_fetch(ip, 1);
71         } else {
72             pthread_mutex_lock(&lock);
73             /* You cannot modify the following line */
74             --(*ip);
75             pthread_mutex_unlock(&lock);
76         }
77     }
78     fprintf(stderr, "Done decreasing variable.\n");
79
80     return NULL;
81 }
82
83
84 int main(int argc, char *argv[])
85 {
86     int val, ret, ok;
87     pthread_t t1, t2;
88
89     /*
90      * Initial value
91      */
92     val = 0;
93     if(pthread_mutex_init(&lock, NULL) != 0)
94         {perror("mutex"); exit(1);}
95
96     /*
97      * Create threads
98      */
99     ret = pthread_create(&t1, NULL, increase_fn, &val);
100     if (ret) {
101         perror_thread(ret, "pthread_create");
102         exit(1);
103     }
104     ret = pthread_create(&t2, NULL, decrease_fn, &val);
105     if (ret) {
106         perror_thread(ret, "pthread_create");
107         exit(1);
108     }
109
110     /*
111      * Wait for threads to terminate
112      */
113     ret = pthread_join(t1, NULL);
114     if (ret)
115         perror_thread(ret, "pthread_join");
116     ret = pthread_join(t2, NULL);
117     if (ret)
```

```
118     perror_pthread(ret, "pthread_join");
119
120     /*
121      * Is everything OK?
122      */
123     ok = (val == 0);
124
125     printf("%sOK, val = %d.\n", ok ? "" : "NOT ", val);
126
127     return ok;
128 }
129
```

Το αποτέλεσμα, όπως φαίνεται και παρακάτω, είναι 0, όπως πρέπει:

```
peppasmich@Michael-Peppas:/mnt/d/OneDrive/Μιχαήλ - Αθανάσιος Πέππας -- Βιβλιοθήκη/Ακαδημαϊκά/Σ.Η.Μ.Μ.Υ. Ε.Μ.Π/Μαθήματα - Εξάμηνα/Μεγαλύτερα Εξάμηνα - Ροές/Ροή Υ/Λειτουργι  
κά Συστήματα Υπολογιστών/Εργαστήριο/2η Άσκηση/κώδικες Ασκήσεων/askhsh2$ ./simplesync-atomic  
About to increase variable 10000000 times  
Done increasing variable.  
Done decreasing variable.  
OK, val = 0.  
peppasmich@Michael-Peppas:/mnt/d/OneDrive/Μιχαήλ - Αθανάσιος Πέππας -- Βιβλιοθήκη/Ακαδημαϊκά/Σ.Η.Μ.Μ.Υ. Ε.Μ.Π/Μαθήματα - Εξάμηνα/Μεγαλύτερα Εξάμηνα - Ροές/Ροή Υ/Λειτουργι  
κά Συστήματα Υπολογιστών/Εργαστήριο/2η Άσκηση/κώδικες Ασκήσεων/askhsh2$ ./simplesync-mutex  
About to increase variable 10000000 times  
About to decrease variable 10000000 times  
Done decreasing variable.  
Done increasing variable.  
OK, val = 0.
```

Αξίζει να σημειωθεί πως το πρόγραμμά μας, όπως φαίνεται και από τον κώδικα που παραθέσαμε, ελέγχει την ορθότητα των εισαγόμενων δεδομένων κατά τον χρόνο εκτέλεσης και σε περίπτωση λάθους, τυπώνει το αντίστοιχο μήνυμα στον χρήστη, καθοδηγώντας τον προς την εισαγωγή μιας ορθής εισόδου.

Ερωτήσεις

1. Αρχικά, τρέχουμε με χρήση της εντολής `time` τη `simplesync-atomic`, η οποία έχει παραχθεί από τον αρχικό κώδικα, χωρίς παραλλαγές, δηλαδή χωρίς συγχρονισμό. Το αποτέλεσμα φαίνεται ακολούθως:

```
peppasmich@Michael-Peppas: /mnt/d/OneDrive/Μιχαήλ - Αθανάσιος Πέππας -- Βιβλίο - Ε.Μ.Π/Μαθήματα - Εξάμηνα/Μεγαλύτερα Εξάμηνα - Ραές/Ροή Υ/Λειτουργι  
κά Συστήματα Υπολογιστών/Εργαστήριο/2η Άσκηση/Κώδικες Ασκήσεων/askhsh2$ time ./simplesync-atomic  
About to increase variable 10000000 times  
About to decrease variable 10000000 times  
Done increasing variable.  
Done decreasing variable.  
NOT OK, val = 1229810.  
  
real    0m0.030s  
user    0m0.041s  
sys     0m0.000s
```

Όπως εύκολα γίνεται αντιληπτό, το αποτέλεσμα είναι λανθασμένο, αφού δεν υπάρχει συγχρονισμός. Ακολούθως, τρέχουμε τους σωστούς κώδικες `simplesync-atomic` και `simplesync-mutex`, αντίστοιχα, και το αποτέλεσμα φαίνεται ακολούθως:

```
κά Συστήματα Υπολογιστών/Εργαστήριο/2η Άσκηση/Κώδικες Ασκήσεων/askhsh2$ time ./simplesync-atomic  
About to increase variable 10000000 times  
About to decrease variable 10000000 times  
Done decreasing variable.  
Done increasing variable.  
OK, val = 0.  
  
real    0m0.264s  
user    0m0.501s  
sys     0m0.000s  
peppasmich@Michael-Peppas: /mnt/d/OneDrive/Μιχαήλ - Αθανάσιος Πέππας -- Βιβλιοθήκη/Ακαδημαϊκά/Σ.Η.Μ.Μ.Υ. Ε.Μ.Π/Μαθήματα - Εξάμηνα/Μεγαλύτερα Εξάμηνα - Ραές/Ροή Υ/Λειτουργι  
κά Συστήματα Υπολογιστών/Εργαστήριο/2η Άσκηση/Κώδικες Ασκήσεων/askhsh2$ time ./simplesync-mutex  
About to increase variable 10000000 times  
Done increasing variable.  
Done decreasing variable.  
OK, val = 0.  
  
real    0m1.105s  
user    0m1.504s  
sys     0m0.676s
```

Παρατηρούμε, όπως εξάλλου ήταν και αναμενόμενο, ότι ο χρόνος εκτέλεσης του ασυγχρόνιστου προγράμματος είναι πολύ μικρότερος και από τα δύο προγράμματα που εκτελούν συγχρονισμό και εμφανίζουν – φυσικά – το σωστό αποτέλεσμα. Ο λόγος για αυτό είναι ότι στο αρχικό πρόγραμμα εκτελούνται όλες οι προσθέσεις και οι αφαιρέσεις ταυτόχρονα, χωρίς έτσι να παράγεται σωστό αποτέλεσμα, αφού χάνεται η σωστή αλληλουχία τους και χρησιμοποιούν κοινή μνήμη, ενώ στις άλλες περιπτώσεις εκτελούνται μόνο όταν το κρίσιμο τμήμα της προηγούμενης έχει ολοκληρωθεί, υστερώντας έτσι σε χρόνο, εμφανίζοντας όμως το σωστό αποτέλεσμα.

2. Όπως φαίνεται και από την παραπάνω εικόνα, η μέθοδος των `atomic operations` είναι κατά πολύ (περίπου 5 φορές) γρηγορότερη από τη μέθοδο των `mutexes`. Ο λόγος πίσω από αυτό είναι ότι στην περίπτωση των `atomic operations` οι προσθέσεις και οι αφαιρέσεις μπορούν να τρέχουν παράλληλα – ατομικά – και δεν επικαλύπτεται μόνο το κρίσιμο (σε `assembly`) κομμάτι τους ενώ το μη κρίσιμο εκτελείται ταυτοχρόνως, ενώ στην περίπτωση των `mutexes`, έχουμε κλειδώσει όλο το κομμάτι των προσθέσεων και των αφαιρέσεων, με αποτέλεσμα αυτές να εκτελούνται σειριακά, μία κάθε φορά.

3. Αλλάζουμε το Makefile, προσθέτοντας τα flags -S και -g, ώστε να παραγάγουμε τον αναγνώσιμο κώδικα των αρχείων μας σε assembly, ως εξής:

```
Makefile
1  #
2  # Makefile
3  #
4
5  CC = gcc
6
7  # CAUTION: Always use '-pthread' when compiling POSIX threads-based
8  # applications, instead of linking with "-lpthread" directly.
9  CFLAGS = -Wall -O2 -pthread -S -g
```

Κατόπιν, ανοίγουμε το simplesync-atomic.o και εντοπίζουμε ένα σημείο στο οποίο εκτελούνται τα atomic operations, το οποίο φαίνεται ακολούθως:

```
65  .LBE15:
66      .loc 1 48 3 is_stmt 1 view .LVU16
67      .loc 1 49 4 view .LVU17
68      lock addl    $1, (%rbx)
69      .loc 1 47 22 discriminator 2 view .LVU18
70      .loc 1 47 16 discriminator 1 view .LVU19
71      .loc 1 48 3 view .LVU20
72      .loc 1 49 4 view .LVU21
73      lock addl    $1, (%rbx)
74      .loc 1 47 22 discriminator 2 view .LVU22
75      .loc 1 47 16 discriminator 1 view .LVU23
76      subl    $2, %eax
77      jne .L2
78      .loc 1 57 2 view .LVU24
```

```
164  .LBE21:
165      .loc 1 69 3 is_stmt 1 view .LVU48
166      .loc 1 70 4 view .LVU49
167      lock subl    $1, (%rbx)
168      .loc 1 68 22 discriminator 2 view .LVU50
169      .loc 1 68 16 discriminator 1 view .LVU51
170      .loc 1 69 3 view .LVU52
171      .loc 1 70 4 view .LVU53
172      lock subl    $1, (%rbx)
173      .loc 1 68 22 discriminator 2 view .LVU54
174      .loc 1 68 16 discriminator 1 view .LVU55
175      subl    $2, %eax
176      jne .L8
177      .loc 1 78 2 view .LVU56
```

4. Ομοίως, ανοίγουμε το `simplesync-mutex.o` και εντοπίζουμε ένα σημείο στο οποίο εκτελούνται τα `pthread_mutex_lock` / `pthread_mutex_unlock`, το οποίο φαίνεται ακολούθως:

```
76  .L2:
77      .loc 1 48 3 view .LVU17
78      .loc 1 51 4 view .LVU18
79      movq    %r12, %rdi
80      call    pthread_mutex_lock@PLT
81  .LVL4:
82      .loc 1 53 4 view .LVU19
83      .loc 1 53 7 is_stmt 0 view .LVU20
84      movl    0(%rbp), %eax
85      .loc 1 54 4 view .LVU21
86      movq    %r12, %rdi
87      .loc 1 53 4 view .LVU22
88      addl    $1, %eax
89      movl    %eax, 0(%rbp)
90      .loc 1 54 4 is_stmt 1 view .LVU23
91      call    pthread_mutex_unlock@PLT
```

▪ Ενότητα 2 – Παράλληλος υπολογισμός του συνόλου Mandelbrot

Εισαγωγή

Σημειώνουμε πως αμφότερα τα προγράμματά μας (mandel_sema.c και mandel_cond.c) ελέγχουν την ορθότητα των εισαγόμενων δεδομένων κατά τον χρόνο εκτέλεσης και σε περίπτωση λάθους, τυπώνουν το αντίστοιχο μήνυμα στον χρήστη, καθοδηγώντας τον προς την εισαγωγή μιας ορθής εισόδου.

1. Αρχικά, συντάσσουμε τον ακόλουθο κώδικα, ο οποίος φέρει το όνομα mandel_sema.c και υλοποιεί το ζητούμενο με τη χρήση σημαφόρων. Το struct του καθενός thread φαίνεται ακολούθως και περιέχει το id του, τον συνολικό αριθμό των νημάτων και έναν πίνακα στον οποίο αποθηκεύονται τα δεδομένα του compute του mandel_line και αργότερα απεικονίζονται. Το ίδιο struct χρησιμοποιείται και για το mandel_cond.c στο επόμενο ερώτημα και για τον λόγο αυτό δεν θα παρατεθεί ξανά.

```
65  struct thread_info_struct {
66      pthread_t tid; /* POSIX thread id, as returned by the library */
67
68      int *color_val; /* Pointer to array to manipulate */
69      int thrid; /* Application-defined thread id */
70      int thrcnt;
71  };
```

Η λογική πίσω από τον κώδικά μας είναι η εξής: εάν έχουμε n threads να τρέχουν το mandel, καθένα από αυτά έχει τον δικό του σημαφόρο (έχουμε δηλαδή έναν πίνακα από σημαφόρους, τον sem) αρχικοποιημένο σε 0, εκτός από το 1^ο thread, του οποίου είναι αρχικοποιημένος σε 1. Έτσι, αφού το 1^ο thread εκτελέσει το κρίσιμο τμήμα του, αυτό θα ξυπνήσει το 2^ο, το 2^ο το 3^ο κ.ο.κ., μέσω της sem_post του αντίστοιχου σημαφόρου. Επίσης, έχουμε χειριστεί το πρόγραμμά μας έτσι ώστε το κρίσιμο τμήμα να περιλαμβάνει μόνο τη φάση του output και όχι και αυτή του compute, για λόγους επίδοσης που θα σχολιαστούν αργότερα, στα αντίστοιχα ζητούμενα ερωτήματα, και άρα τα computes γίνονται ταυτόχρονα και αποθηκεύονται στον ξεχωριστό πίνακα του κάθε thread. Ο κώδικας αυτός φαίνεται ακολούθως:

mandel_sema.c

```
1  /*
2   * mandel_sema.c
3   *
4   * A program to draw the Mandelbrot Set on a 256-color xterm, with semaphores.
5   *
6   */
7
8  #include <signal.h>
9  #include <errno.h>
10 #include <pthread.h>
11 #include <stdio.h>
12 #include <unistd.h>
13 #include <assert.h>
14 #include <string.h>
15 #include <math.h>
16 #include <stdlib.h>
17 #include <semaphore.h>
18
19 #include "mandel-lib.h"
20
21 #define MANDEL_MAX_ITERATION 100000
22
23 /*****
24  * Compile-time parameters *
25  *****/
26
27 /*
28  * POSIX thread functions do not return error numbers in errno,
29  * but in the actual return value of the function call instead.
30  * This macro helps with error reporting in this case.
31  */
32 #define perror_pthread(ret, msg) \
33     do { errno = ret; perror(msg); } while (0)
34
35 /*
36  * Output at the terminal is is x_chars wide by y_chars long
37  */
38 int y_chars = 50;
39 int x_chars = 90;
40
41 /*
42  * The part of the complex plane to be drawn:
43  * upper left corner is (xmin, ymax), lower right corner is (xmax, ymin)
44  */
45 double xmin = -1.8, xmax = 1.0;
46 double ymin = -1.0, ymax = 1.0;
47
48 /*
49  * Every character in the final output is
50  * xstep x ystep units wide on the complex plane.
51  */
52 double xstep;
53 double ystep;
54
55 void sig_handler(int signum)
56 {
57     reset_xterm_color(1);
```

```
58     exit(1);
59 }
60
61 /*
62  * A (distinct) instance of this structure
63  * is passed to each thread
64  */
65 struct thread_info_struct {
66     pthread_t tid; /* POSIX thread id, as returned by the library */
67
68     int *color_val; /* Pointer to array to manipulate */
69     int thrid; /* Application-defined thread id */
70     int thrcnt;
71 };
72
73 int safe_atoi(char *s, int *val)
74 {
75     long l;
76     char *endp;
77
78     l = strtol(s, &endp, 10);
79     if (s != endp && *endp == '\0') {
80         *val = l;
81         return 0;
82     } else
83         return -1;
84 }
85
86 void usage(char *argv0)
87 {
88     fprintf(stderr, "Usage: %s thread_count array_size\n\n"
89             "Exactly one argument required:\n"
90             "    thread_count: The number of threads to create.\n",
91             argv0);
92     exit(1);
93 }
94
95 void *safe_malloc(size_t size)
96 {
97     void *p;
98
99     if ((p = malloc(size)) == NULL) {
100         fprintf(stderr, "Out of memory, failed to allocate %zd bytes\n",
101                 size);
102         exit(1);
103     }
104
105     return p;
106 }
107
108 /*
109  * This function computes a line of output
110  * as an array of x_char color values.
111  */
112 void compute_mandel_line(int line, int color_val[])
113 {
114     /*
115      * x and y traverse the complex plane.
116      */
117     double x, y;
```

```
118
119     int n;
120     int val;
121
122     /* Find out the y value corresponding to this line */
123     y = ymax - ystep * line;
124
125     /* and iterate for all points on this line */
126     for (x = xmin, n = 0; n < x_chars; x+= xstep, n++) {
127
128         /* Compute the point's color value */
129         val = mandel_iterations_at_point(x, y, MANDEL_MAX_ITERATION);
130         if (val > 255)
131             val = 255;
132
133         /* And store it in the color_val[] array */
134         val = xterm_color(val);
135         color_val[n] = val;
136     }
137 }
138
139 /*
140  * This function outputs an array of x_char color values
141  * to a 256-color xterm.
142  */
143 void output_mandel_line(int fd, int color_val[]) {
144     char point = '@';
145     char newline = '\n';
146
147     for (int i = 0; i < x_chars; i++) {
148         /* Set the current color, then output the point */
149         set_xterm_color(fd, color_val[i]);
150         if (write(fd, &point, 1) != 1) {
151             perror("compute_and_output_mandel_line: write point");
152             exit(1);
153         }
154     }
155
156     /* Now that the line is done, output a newline character */
157     if (write(fd, &newline, 1) != 1) {
158         perror("compute_and_output_mandel_line: write newline");
159         exit(1);
160     }
161 }
162
163 /* Start function for each thread */
164 sem_t *sem;
165
166 void *compute_and_output_mandel_line(void *arg)
167 {
168     struct thread_info_struct *thr = arg;
169
170     for (int i = thr->thrid; i < y_chars; i += thr->thrcnt) {
171         compute_mandel_line(i, thr->color_val);
172         sem_wait(&sem[i % thr->thrcnt]);
173         output_mandel_line(1, thr->color_val);
174         sem_post(&sem[(i+1) % thr->thrcnt]);
175     }
176     return NULL;
177 }
```

```
178
179 int main(int argc, char *argv[])
180 {
181     int thrcnt, ret;
182     struct thread_info_struct *thr;
183
184     // Signal Handler
185     struct sigaction sa;
186     sa.sa_flags = SA_RESTART;
187     sa.sa_handler = sig_handler;
188     if (sigaction(SIGINT, &sa, NULL) < 0)
189     {
190         perror("sigaction");
191         exit(1);
192     }
193
194     /*
195     * Parse the command line
196     */
197     if (argc != 2)
198         usage(argv[0]);
199     if (safe_atoi(argv[1], &thrcnt) < 0 || thrcnt <= 0) {
200         fprintf(stderr, "'%s' is not valid for `thread_count'\n", argv[1]);
201         exit(1);
202     }
203
204     xstep = (xmax - xmin) / x_chars;
205     ystep = (ymax - ymin) / y_chars;
206
207     thr = safe_malloc(thrcnt * sizeof(*thr));
208     sem = safe_malloc(thrcnt * sizeof(sem_t));
209
210     for(int i=0; i<thrcnt; i++)
211     {
212         thr[i].thrcnt = thrcnt;
213         thr[i].thrid = i;
214         thr[i].color_val = safe_malloc(x_chars * sizeof(int));
215         if(i == 0) sem_init(&sem[i], 0, 1);
216         else sem_init(&sem[i], 0, 0);
217
218         /* Spawn new thread */
219         ret = pthread_create(&thr[i].tid, NULL, compute_and_output_mandel_line, &thr[i]);
220         if (ret) {
221             perror_pthread(ret, "pthread_create");
222             exit(1);
223         }
224     }
225
226     /*
227     * Wait for all threads to terminate
228     */
229     for (int i=0; i<thrcnt; i++) {
230         ret = pthread_join(thr[i].tid, NULL);
231         if (ret) {
232             perror_pthread(ret, "pthread_join");
233             exit(1);
234         }
235     }
236
237     for(int i=0; i<thrcnt; i++)
```

```
238     sem_destroy(&sem[i]);  
239  
240     reset_xterm_color(1);  
241     return 0;  
242 }  
243
```


2. Για το ερώτημα αυτό, συντάσσουμε τον ακόλουθο κώδικα, ο οποίος φέρει το όνομα `mandel_cond.c` και υλοποιεί το ζητούμενο με τη χρήση `condition variables`. Επισημαίνουμε πως το κάθε `thread` χρησιμοποιεί το `struct` που παρατέθηκε προηγουμένως. Η λογική αυτού του προγράμματος είναι η εξής: έχουμε n `threads`, τα οποία πάλι (για λόγους επίδοσης) τρέχουν παράλληλα το `compute` κάθε γραμμής που έχουν αναλάβει και έχουν ως κρίσιμο τμήμα μόνο αυτό της απεικόνισης μιας γραμμής, πριν το οποίο περιμένουν σε μια κοινή κελιδαριά. Έτσι, όντας συνδεδεμένα σε μια κοινή κλειδαριά (`lock`), έχουμε έναν πίνακα από `conditions` (`cond`), ένα για κάθε νήμα. Με τη βοήθεια ενός μετρητή, αρχικά τρέχει μόνο το υπ' αριθμόν 0 νήμα και ακολούθως, αφού εκτελέσει το κρίσιμο τμήμα του, κάνει `broadcast` στο επόμενο (στο αντίστοιχο `condition`) για να ξεκινήσει. Έτσι, το `mandel` απεικονίζεται επιτυχώς στην έξοδο, με τη βέλτιστη επίδοση.

Αξίζει να επισημάνουμε ότι τα `threads` μοιράζονται τις `global` μεταβλητές τους. Για τον λόγο αυτό, προτιμήσαμε κάθε `thread` να κάνει `malloc` έναν δικό του πίνακα, ώστε να μπορούν όλα μαζί – παράλληλα – να υπολογίζουν την έξοδο μίας γραμμής, περιορίζοντας έτσι το κρίσιμο τμήμα των `threads` μόνο στη διαδικασία του `output`, επιτυγχάνοντας βέλτιστη επίδοση. Αυτό, γίνεται φανερό στον τρόπο που αρχικοποιούμε το `struct` κάθε `thread` και στο σώμα της συνάρτησης που αυτό καλεί.

Ο κώδικας του `mandel_cond.c` φαίνεται ακολούθως:

mandel_cond.c

```
1  /*
2   * mandel_cond.c
3   *
4   * A program to draw the Mandelbrot Set on a 256-color xterm, with condition variables.
5   *
6   */
7
8  #include <signal.h>
9  #include <errno.h>
10 #include <pthread.h>
11 #include <stdio.h>
12 #include <unistd.h>
13 #include <assert.h>
14 #include <string.h>
15 #include <math.h>
16 #include <stdlib.h>
17 #include <semaphore.h>
18
19 #include "mandel-lib.h"
20
21 #define MANDEL_MAX_ITERATION 100000
22
23 /*****
24  * Compile-time parameters *
25  *****/
26
27 /*
28  * POSIX thread functions do not return error numbers in errno,
29  * but in the actual return value of the function call instead.
30  * This macro helps with error reporting in this case.
31  */
32 #define perror_pthread(ret, msg) \
33     do { errno = ret; perror(msg); } while (0)
34
35 /*
36  * Output at the terminal is is x_chars wide by y_chars long
37  */
38 int y_chars = 50;
39 int x_chars = 90;
40
41 /*
42  * The part of the complex plane to be drawn:
43  * upper left corner is (xmin, ymax), lower right corner is (xmax, ymin)
44  */
45 double xmin = -1.8, xmax = 1.0;
46 double ymin = -1.0, ymax = 1.0;
47
48 /*
49  * Every character in the final output is
50  * xstep x ystep units wide on the complex plane.
51  */
52 double xstep;
53 double ystep;
54
55 void sig_handler(int signum)
56 {
57     reset_xterm_color(1);
```

```
58     exit(1);
59 }
60
61 /*
62  * A (distinct) instance of this structure
63  * is passed to each thread
64  */
65 struct thread_info_struct {
66     pthread_t tid; /* POSIX thread id, as returned by the library */
67
68     int *color_val; /* Pointer to array to manipulate */
69     int thrid; /* Application-defined thread id */
70     int thrcnt;
71 };
72
73 int safe_atoi(char *s, int *val)
74 {
75     long l;
76     char *endp;
77
78     l = strtol(s, &endp, 10);
79     if (s != endp && *endp == '\0') {
80         *val = l;
81         return 0;
82     } else
83         return -1;
84 }
85
86 void usage(char *argv0)
87 {
88     fprintf(stderr, "Usage: %s thread_count array_size\n\n"
89             "Exactly one argument required:\n"
90             "    thread_count: The number of threads to create.\n",
91             argv0);
92     exit(1);
93 }
94
95 void *safe_malloc(size_t size)
96 {
97     void *p;
98
99     if ((p = malloc(size)) == NULL) {
100         fprintf(stderr, "Out of memory, failed to allocate %zd bytes\n",
101                 size);
102         exit(1);
103     }
104
105     return p;
106 }
107
108 /*
109  * This function computes a line of output
110  * as an array of x_char color values.
111  */
112 void compute_mandel_line(int line, int color_val[])
113 {
114     /*
115      * x and y traverse the complex plane.
116      */
117     double x, y;
```

```
118
119     int n;
120     int val;
121
122     /* Find out the y value corresponding to this line */
123     y = ymax - ystep * line;
124
125     /* and iterate for all points on this line */
126     for (x = xmin, n = 0; n < x_chars; x+= xstep, n++) {
127
128         /* Compute the point's color value */
129         val = mandel_iterations_at_point(x, y, MANDEL_MAX_ITERATION);
130         if (val > 255)
131             val = 255;
132
133         /* And store it in the color_val[] array */
134         val = xterm_color(val);
135         color_val[n] = val;
136     }
137 }
138
139 /*
140  * This function outputs an array of x_char color values
141  * to a 256-color xterm.
142  */
143 void output_mandel_line(int fd, int color_val[]) {
144     char point = '@';
145     char newline = '\n';
146
147     for (int i = 0; i < x_chars; i++) {
148         /* Set the current color, then output the point */
149         set_xterm_color(fd, color_val[i]);
150         if (write(fd, &point, 1) != 1) {
151             perror("compute_and_output_mandel_line: write point");
152             exit(1);
153         }
154     }
155
156     /* Now that the line is done, output a newline character */
157     if (write(fd, &newline, 1) != 1) {
158         perror("compute_and_output_mandel_line: write newline");
159         exit(1);
160     }
161 }
162
163 /* Start function for each thread */
164 pthread_mutex_t lock;
165 pthread_cond_t *cond;
166 int counter = 0;
167
168 void *compute_and_output_mandel_line(void *arg)
169 {
170     struct thread_info_struct *thr = arg;
171
172     for(int line = thr->thrid; line < y_chars; line += thr->thrcnt) {
173         compute_mandel_line(line, thr->color_val);
174         pthread_mutex_lock(&lock);
175         if(line != counter) pthread_cond_wait(&cond[thr->thrid], &lock);
176         counter++;
177         output_mandel_line(1, thr->color_val);
```

```
178     pthread_cond_broadcast(&cond[(line+1) % thr->thrcnt]);
179     pthread_mutex_unlock(&lock);
180 }
181     return NULL;
182 }
183
184 int main(int argc, char *argv[])
185 {
186     int thrcnt, ret;
187     struct thread_info_struct *thr;
188
189     // Signal Handler
190     struct sigaction sa;
191     sa.sa_flags = SA_RESTART;
192     sa.sa_handler = sig_handler;
193     if (sigaction(SIGINT, &sa, NULL) < 0)
194     {
195         perror("sigaction");
196         exit(1);
197     }
198
199     /*
200      * Parse the command line
201      */
202     if (argc != 2)
203         usage(argv[0]);
204     if (safe_atoi(argv[1], &thrcnt) < 0 || thrcnt <= 0 || argv[1][0] == '1') {
205         fprintf(stderr, "'%s' is not valid for `thread_count'\n", argv[1]);
206         exit(1);
207     }
208
209     xstep = (xmax - xmin) / x_chars;
210     ystep = (ymax - ymin) / y_chars;
211
212     thr = safe_malloc(thrcnt * sizeof(*thr));
213     cond = safe_malloc(thrcnt * sizeof(pthread_cond_t));
214
215     for(int i=0; i<thrcnt; i++)
216     {
217         thr[i].thrcnt = thrcnt;
218         thr[i].thrid = i;
219         thr[i].color_val = safe_malloc(x_chars * sizeof(int));
220
221         /* Spawn new thread */
222         ret = pthread_create(&thr[i].tid, NULL, compute_and_output_mandel_line, &thr[i]);
223         if (ret) {
224             perror_pthread(ret, "pthread_create");
225             exit(1);
226         }
227     }
228
229     /*
230      * Wait for all threads to terminate
231      */
232     for (int i=0; i<thrcnt; i++) {
233         ret = pthread_join(thr[i].tid, NULL);
234         if (ret) {
235             perror_pthread(ret, "pthread_join");
236             exit(1);
237         }
238     }
239 }
```

```
238     }  
239  
240     reset_xterm_color(1);  
241     return 0;  
242 }  
243
```

Ερωτήσεις

1. Για το σύστημα συγχρονισμού που υλοποιήσαμε, χρειαστήκαμε αριθμό σημαφόρων ίσο με τον αριθμό των threads που τρέχουν συγχρόνως και που εισάγονται από τον χρήστη κατά τον χρόνο εκτέλεσης του προγράμματός μας. Ειδικότερα, χρησιμοποιήσαμε έναν πίνακα από σημαφόρους (τον sem), ο οποίος είναι αρχικοποιημένος στο 0, εκτός από την 1^η θέση του, που είναι αρχικοποιημένη στο 1. Έτσι, θα εκτελεστεί μόνο το 1^ο thread (αφού τα άλλα θα περιμένουν) και τα επόμενα θα καλούνται διαδοχικά, καθώς το προηγούμενο θα κάνει sem_post το επόμενο. Αυτό γίνεται φανερό στον κώδικα mandel_sema.c που παραθέσαμε παραπάνω.
2. Αρχικά, χρησιμοποιούμε την εντολή cat /proc/cpuinfo και διαπιστώνουμε ότι το μηχανήμα μας διαθέτει 4 πυρήνες (σε αντίθεση με τον orion που διαπιστώσαμε ότι έχει μόνο 1), καθιστώντας τη δοκιμή πολλών threads εφικτή. Αυτό, φαίνεται ακολούθως:

```
processor       : 7
vendor_id      : GenuineIntel
cpu family     : 6
model          : 158
model name     : Intel(R) Core(TM) i7-7700HQ CPU @ 2.80GHz
stepping       : 9
microcode      : 0xffffffff
cpu MHz        : 2807.998
cache size     : 6144 KB
physical id    : 0
siblings       : 8
core id        : 3
cpu cores      : 4
apicid         : 7
initial apicid : 7
fpu            : yes
fpu_exception  : yes
cpuid level    : 21
wp             : yes
```

Επομένως, τρέχουμε το mandel πρώτα και το mandel_sema μετά, με 2 threads και με κατάλληλη χρήση της time, και το αποτέλεσμα φαίνεται ακολούθως:

real	0m0.549s	real	0m0.307s
user	0m0.477s	user	0m0.547s
sys	0m0.070s	sys	0m0.051s

Όπως ήταν αναμενόμενο, η εκτέλεση του mandel διήρκησε πολύ περισσότερο (σχεδόν τον διπλάσιο πραγματικό χρόνο) από αυτή του mandel_sema με 2 threads, αφού η διαδικασία των computes γίνεται πλέον παράλληλα από τα 2 threads και έχουμε απομονώσει μόνο το πραγματικό κρίσιμο τμήμα τους. Παρατηρούμε, έτσι, μια μεγάλη βελτίωση στην επίδοση του προγράμματός μας.

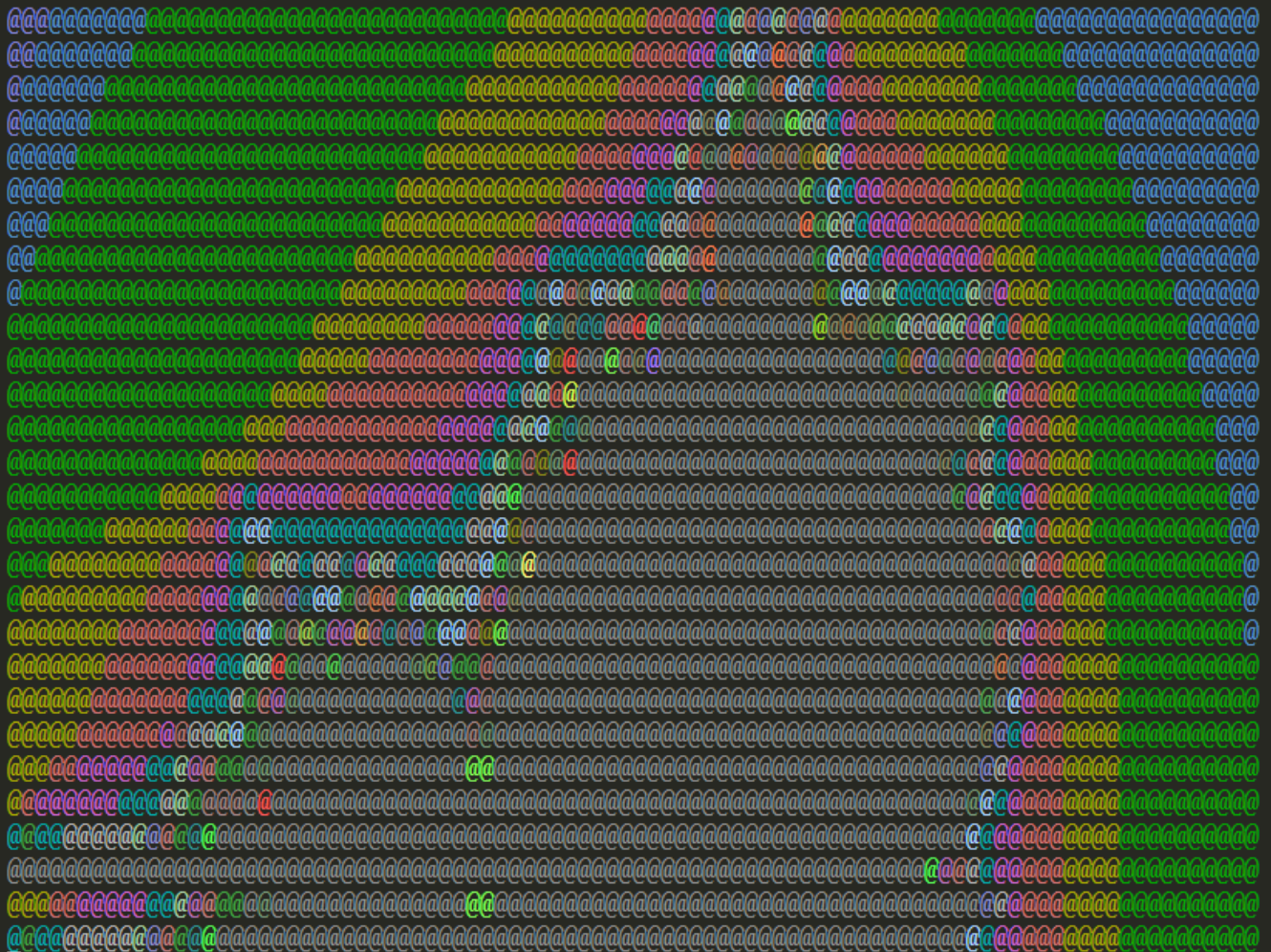
3. Όπως φαίνεται και στον κώδικα `mandel_cond.c` που παραθέσαμε παραπάνω, έχουμε χρησιμοποιήσει αριθμό `condition variables` ίσο με τον αριθμό των `threads` που χρησιμοποιούμε και που εισάγονται από τον χρήστη κατά τον χρόνο εκτέλεσης του προγράμματός μας. Ο λόγος για αυτό είναι ο εξής: τα `threads` είναι συνδεδεμένα σε μια κοινή κλειδαριά (`lock`) και έχουμε έναν πίνακα από `conditions` (`cond`), ένα για κάθε νήμα. Με τη βοήθεια ενός μετρητή (`counter`), αρχικά τρέχει μόνο το υπ' αριθμόν 0 νήμα και ακολούθως, αφού εκτελέσει το κρίσιμο τμήμα του, κάνει `broadcast` στο επόμενο (στο αντίστοιχο δηλαδή `condition`) για να ξεκινήσει. Έτσι, τα `computes` γίνονται παράλληλα και τα `outputs` με τη σειρά που πρέπει, επιτυγχάνοντας βέλτιστη επίδοση.

Έστω, τώρα, ότι χρησιμοποιούμε μόνο ένα `condition variable`, έστω `cond` και ότι – χωρίς βλάβη της γενικότητας – έχουμε 2 `threads`. Στην περίπτωση αυτή, θα δημιουργηθεί το εξής πρόβλημα συγχρονισμού: αρχικά, θα τρέξει το 1^ο `thread` τη διαδικασία του `compute` του, όπως και το 2^ο, παράλληλα, ενώ στην συνέχεια, το 1^ο θα τρέξει και το κρίσιμο τμήμα του (`output`), ενώ το 2^ο θα κάνει `wait`, περιμένοντας το 1^ο να τελειώσει και να κάνει `broadcast` στο `cond`, ώστε να ξεκινήσει κι αυτό. Όμως, μπορεί το 1^ο `thread` να τελειώσει το κρίσιμο τμήμα του και να κάνει `broadcast` πριν το 2^ο κάνει `wait` και αυτό θα έχει ως αποτέλεσμα το 2^ο να κάνει `wait` για πάντα, αφού το 1^ο δεν θα ξανακάνει `broadcast` ποτέ. Άρα, το 2^ο `thread` δεν θα κάνει ποτέ `output` και το πρόγραμμά μας θα κολλήσει στο σημείο αυτό. Αντίστοιχο πρόβλημα θα παρατηρηθεί και εάν έχουμε παραπάνω από 2 `threads` σε κοινό `cond`.

Το πρόβλημα αυτό, αποτυπώνεται κάτωθι, όπου παραθέτουμε τον αλλαγμένο κώδικα στο σημείο της κλήσης της συνάρτησης από το κάθε `thread` και το αντίστοιχο `output`, το οποίο παραμένει παγωμένο για πάντα:

```
168 void *compute_and_output_mandel_line(void *arg)
169 {
170     struct thread_info_struct *thr = arg;
171
172     for(int line = thr->thrid; line < y_chars; line += thr->thrcnt) {
173         compute_mandel_line(line, thr->color_val);
174         pthread_mutex_lock(&lock);
175         if(line != counter) pthread_cond_wait(&cond[0], &lock);
176         counter++;
177         output_mandel_line(1, thr->color_val);
178         pthread_cond_broadcast(&cond[0]);
179         pthread_mutex_unlock(&lock);
180     }
181     return NULL;
182 }
```


κά Συστήματα Υπολογιστών/Εργαστήριο/2η Άσκηση/Κώδικες Ασκήσεων/askhsh2\$./mandel_cond 6

A colorful fractal image generated by the Mandelbrot set, showing a complex, self-similar pattern with various colors like red, green, blue, and yellow.

4. Για τους λόγους που ήδη έχουμε αναφέρει, αμφότερα τα προγράμματα που υλοποιήσαμε για το mandel με πολλαπλά threads, εμφανίζουν σημαντική επιτάχυνση, η οποία (έπειτα από δοκιμές) είναι ολοένα και μεγαλύτερη αυξανομένου του αριθμού των νημάτων, αν και όχι με τον ίδιο ρυθμό, δηλαδή από ένα σημείο threads και πάνω (που φυσικά εξαρτάται από τον αριθμό των πυρήνων του μηχανήματος) η μείωση στον χρόνο είναι μικρή, έως και αμελητέα. Ο λόγος πίσω από αυτό είναι ο εξής: στην περίπτωση που ορίσουμε ως κρίσιμο τμήμα σε κάθε thread τις φάσεις τόσο του υπολογισμού όσο και της εμφάνισης μια γραμμής, αυτό θα έχει ως αποτέλεσμα να τρέχει μόνο ένα νήμα κάθε φορά και η όλη διαδικασία να εκτελείται σειριακά, όπως στο αρχικό mandel! Ένα όμως φροντίσουμε να έχουμε ξεχωριστές δομές δεδομένων για κάθε νήμα (όπως και κάναμε), τότε η φάση του compute – που όπως τελικά διαπιστώνουμε είναι αρκετά χρονοβόρα – δύναται να βγει εκτός του κρίσιμου τμήματος και σε αυτό να παραμείνει μόνο η φάση της αποτύπωσης. Έτσι, ο χρόνος εκτέλεσης μειώνεται αισθητά και γίνεται πραγματική αξιοποίηση του μεγάλου αριθμού νημάτων.

Για να γίνει αυτό φανερό, αλλάζουμε το κρίσιμο τμήμα (έστω του `mandel_sema.c`), ώστε αυτό να περιλαμβάνει αμφότερες τις φάσεις που περιγράψαμε, ως εξής:

```
166 void *compute_and_output_mandel_line(void *arg)
167 {
168     struct thread_info_struct *thr = arg;
169
170     for (int i = thr->thrid; i < y_chars; i += thr->thrcnt) {
171         sem_wait(&sem[i % thr->thrcnt]);
172         compute_mandel_line(i, thr->color_val);
173         output_mandel_line(1, thr->color_val);
174         sem_post(&sem[(i+1) % thr->thrcnt]);
175     }
176     return NULL;
177 }
```

Ο χρόνος εκτέλεσης (έστω για 4 νήματα) φαίνεται ακολούθως και παρατηρούμε ότι είναι ο ίδιος με αυτόν του mandel:

```
real    0m0.554s
user    0m0.509s
sys     0m0.040s
```

5. Παρατηρούμε ότι στην περίπτωση που πατήσουμε Ctrl+C πριν το πρόγραμμά μας τερματίσει, το χρώμα των γραμμών του τερματικού αλλάζει: από άσπρο γίνεται το χρώμα που είχε ο χαρακτήρας του mandel που ζωγραφιζόταν όταν το πρόγραμμα σταμάτησε. Αυτό, φαίνεται ακολούθως:

```
oslab005@os-node2:~/askhsh2$ This is before Ctrl+C  
+bash: This: command not found  
oslab005@os-node2:~/askhsh2$ ./mandel  
  
oslab005@os-node2:~/askhsh2$ This is after Ctrl+C  
+bash: This: command not found  
oslab005@os-node2:~/askhsh2$
```

Στο σημείο αυτό, επισημαίνουμε ότι τα δύο προγράμματα που παραθέσαμε στην αρχή έχουν την τελική τους μορφή και για τον λόγο αυτό περιέχουν από την αρχή τον κατάλληλο χειρισμό του σήματος Ctrl+C που θα εξηγήσουμε τώρα.

Προκειμένου, επομένως, το χρώμα του τερματικού να επαναφέρεται στο λευκό (default) πριν το πρόγραμμά μας τερματίσει, προσθέτουμε κατάλληλο χειρισμό του σήματος Ctrl+C, όπως φαίνεται ακολούθως:

```
55 void sig_handler(int signum)
56 {
57     reset_xterm_color(1);
58     exit(1);
59 }
```

```
184 // Signal Handler
185 struct sigaction sa;
186 sa.sa_flags = SA_RESTART;
187 sa.sa_handler = sig_handler;
188 if (sigaction(SIGINT, &sa, NULL) < 0)
189 {
190     perror("sigaction");
191     exit(1);
192 }
```

Σ.Η.Μ.Μ.Υ. Ε.Μ.Π.
Μάϊος 2024