Λειτουργικά Συστήματα – Άσκηση 3η

ΑΓΓΕΛΟΣ ΣΤΑΗΣ 03117435, ΣΩΚΡΑΤΗΣ ΠΟΥΤΑΣ 03117054 (Ομάδα Oslabb08)

Άσκηση 1.1 – Συγχρονισμός σε Υπάρχοντα Κώδικα

Κώδικας

```
/*
 * simplesync.c
 *
 * A simple synchronization exercise.
 *
 * Vangelis Koukis <vkoukis@cslab.ece.ntua.gr>
 * Operating Systems course, ECE, NTUA
 *
 */

#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>

/*
 * POSIX thread functions do not return error numbers in errno,
 * but in the actual return value of the function call instead.
 * This macro helps with error reporting in this case.
 */
#define perror_pthread(ret, msg) \
      do { errno = ret; perror(msg); } while (0)

#define N 10000000

/* 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

void *increase_fn(void *arg)
{
      int i;
      volatile int *ip = arg;

      fprintf(stderr, "About to increase variable %d times\n", N);
      for (i = 0; i < N; i++) {
            if (USE_ATOMIC_OPS) {
```

```
                /* ... */
                /* You can modify the following line */
                __sync_fetch_and_add(&ip,1);
                /* ... */
            } else {
                pthread_mutex_lock(&lock);
                /* You cannot modify the following line */
                ++(*ip);
                pthread_mutex_unlock(&lock);
            }
        }
        fprintf(stderr, "Done increasing variable.\n");

        return NULL;
}

void *decrease_fn(void *arg)
{
        int i;
        volatile int *ip = arg;

        fprintf(stderr, "About to decrease variable %d times\n", N);
        for (i = 0; i < N; i++) {
            if (USE_ATOMIC_OPS) {
                /* ... */
                /* You can modify the following line */
                __sync_fetch_and_sub(&ip,1);
                /* ... */
            } else {
                pthread_mutex_lock(&lock);
                /* You cannot modify the following line */
                --(*ip);
                pthread_mutex_unlock(&lock);
            }
        }
        fprintf(stderr, "Done decreasing variable.\n");

        return NULL;
}


int main(int argc, char *argv[])
{
        int val, ret, ok;
        pthread_t t1, t2;

        /*
         * Initial value
         */
        val = 0;

        /*
         * Create threads
         */
        ret = pthread_create(&t1, NULL, increase_fn, &val);
        if (ret) {
            perror_pthread(ret, "pthread_create");
```

```
            exit(1);
      }
      ret = pthread_create(&t2, NULL, decrease_fn, &val);
      if (ret) {
            perror_pthread(ret, "pthread_create");
            exit(1);
      }

      /*
       * Wait for threads to terminate
       */
      ret = pthread_join(t1, NULL);
      if (ret)
            perror_pthread(ret, "pthread_join");
      ret = pthread_join(t2, NULL);
      if (ret)
            perror_pthread(ret, "pthread_join");

      /*
       * Is everything OK?
       */
      ok = (val == 0);

      printf("%sOK, val = %d.\n", ok ? "" : "NOT ", val);

      return ok;
}
```

Έξοδος Εκτέλεσης για simplesync-mutex

```
About to increase variable 10000000 times
About to decrease variable 10000000 times
Done decreasing variable.
Done increasing variable.
OK, val = 0.
```

Έξοδος Εκτέλεσης για simplesync-atomic

```
About to increase variable 10000000 times
About to decrease variable 10000000 times
Done decreasing variable.
Done increasing variable.
OK, val = 0.
```

Ερωτήσεις

1) Σε μια περίπτωση εκτέλεσης του αρχικού προγράμματος που δεν εκτελεί συγχρονισμό προκύπτει:

```
real     0m0.038s
user     0m0.072s
sys      0m0.000s
```

Αντίστοιχα, για το πρόγραμμα που εκτελεί συγχρονισμό με mutexes:

```
real  0m3.364s
user  0m3.388s
sys   0m2.544s
```

Ενώ το εκτελέσιμο που χρησιμοποιεί ατομικές λειτουργίες

```
real  0m0.412s
user  0m0.812s
sys   0m0.004s
```

Είναι εμφανές ότι το πρόγραμμα που δεν συγχρονίζει τα νήματά του είναι πιο γρήγορο σε σχέση με τα άλλα δύο. Αυτό συμβαίνει καθώς κατά την περίπτωση μη συγχρονισμού τα νήματα εκτελούν κώδικα (στο κρίσιμο τμήμα) παράλληλα χωρίς καμία αναμονή ενώ στη περίπτωση συγχρονισμού το ένα από τα δύο νήματα για να προχωρήσει τον δικό του υπολογισμό απαιτείται να περιμένει το άλλο που εκτελεί το κρίσιμο τμήμα του κώδικα του.

2) Στη περίπτωση των POSIX mutexes το νήμα που δε μπορεί να εισέλθει στο κρίσιμο τμήμα του εκτελεί αναμονή με έργο σπαταλώντας υπολογιστικό χρόνο μέχρι το κλείδωμα να γίνει διαθέσιμο. Συνεπώς η χρήση ατομικών λειτουργιών είναι ταχύτερη σε σχέση με την περίπτωση των POSIX mutexes.

3) Χρησιμοποιώντας την εντολή

```
gcc -Wall -O2 -pthread -DSYNC_ATOMIC -g -S simplesync.c
```

ώστε να παραχθεί το αντίστοιχο αρχείο assembly βλέπουμε ότι η εντολή

```
__sync_add_and_fetch(ip, 1);
```

μεταφράζεται στην εντολή assembly `lock addl $1,(%rbx)`

όπου το lock αποτελεί prefix που εξασφαλίζει την ατομικότητα της εντολης που ακολουθεί (addl) και η οποία αυξάνει τον καταχωρητή %rbx που είναι αποθηκευμένη η μεταβλητή counter κατά 1.

4) Χρησιμοποιώντας αντίστοιχα την εντολή

```
gcc -Wall -O2 -pthread -DSYNC_MUTEX -g -S simplesync.c
```

ώστε να παραχθεί το αντίστοιχο αρχείο assembly βλέπουμε ότι οι γραμμές

```
pthread_mutex_lock(&lock);
++(*ip);
pthread_mutex_unlock(&lock);
```

μεταγλωτίζονται στις

```
.L2:
    .loc 1 57 0
    movl    $lock, %edi
    call    pthread_mutex_lock
.LVL4:
    .loc 1 58 0
    movl    0(%rbp), %eax
```

```
    .loc 1 59 0
    movl    $lock, %edi
    .loc 1 58 0
    addl    $1, %eax
    movl    %eax, 0(%rbp)
    .loc 1 59 0
call    pthread_mutex_unlock
```

Άσκηση 1.2 – Παράλληλος Υπολογισμός του Συνόλου Mandelbrot

Κώδικας

```
/*
 * mandel.c
 *
 * A program to draw the Mandelbrot Set on a 256-color xterm.
 *
 */

#include <stdio.h>
#include <unistd.h>
#include <assert.h>
#include <string.h>
#include <math.h>
#include <stdlib.h>
#include <semaphore.h>
#include <pthread.h>
#include <errno.h>

#include "mandel-lib.h"

#define MANDEL_MAX_ITERATION 100000

#define perror_pthread(ret, msg) \
      do { errno = ret; perror(msg); } while (0)

/***************************
 * Compile-time parameters *
 ***************************/

/*
 * Output at the terminal is is x_chars wide by y_chars long
*/
int y_chars = 50;
int x_chars = 90;

/*
 * The part of the complex plane to be drawn:
```

```
 * upper left corner is (xmin, ymax), lower right corner is (xmax, ymin)
*/
double xmin = -1.8, xmax = 1.0;
double ymin = -1.0, ymax = 1.0;

/*
 * Every character in the final output is
 * xstep x ystep units wide on the complex plane.
 */
double xstep;
double ystep;

struct thread_info_struct {
      pthread_t tid; /* POSIX thread id, as returned by the library */

      int ** arr; /* Pointer to array to manipulate */

      int thrid; /* Application-defined thread id */
      int thrcnt;
};

int safe_atoi(char *s, int *val)
{
      long l;
      char *endp;

      l = strtol(s, &endp, 10);
      if (s != endp && *endp == '\0') {
                  *val = l;
                  return 0;
      } else
                  return -1;
}

void *safe_malloc(size_t size)
{
      void *p;

      if ((p = malloc(size)) == NULL) {
                  fprintf(stderr, "Out of memory, failed to allocate %zd bytes\n",
                        size);
                  exit(1);
      }

      return p;
}

/*
 * This function computes a line of output
```

```
 * as an array of x_char color values.
 */
void compute_mandel_line(int line, int color_val[])
{
    /*
     * x and y traverse the complex plane.
     */
    double x, y;

    int n;
    int val;

    /* Find out the y value corresponding to this line */
    y = ymax - ystep * line;

    /* and iterate for all points on this line */
    for (x = xmin, n = 0; n < x_chars; x+= xstep, n++) {

       /* Compute the point's color value */
       val = mandel_iterations_at_point(x, y, MANDEL_MAX_ITERATION);
       if (val > 255)
             val = 255;

       /* And store it in the color_val[] array */
       val = xterm_color(val);
       color_val[n] = val;
    }
}

/*
 * This function outputs an array of x_char color values
 * to a 256-color xterm.
 */
void output_mandel_line(int fd, int color_val[])
{
    int i;

    char point ='@';
    char newline='\n';

    for (i = 0; i < x_chars; i++) {
       /* Set the current color, then output the point */
       set_xterm_color(fd, color_val[i]);
       if (write(fd, &point, 1) != 1) {
             perror("compute_and_output_mandel_line: write point");
             exit(1);
       }
    }
```

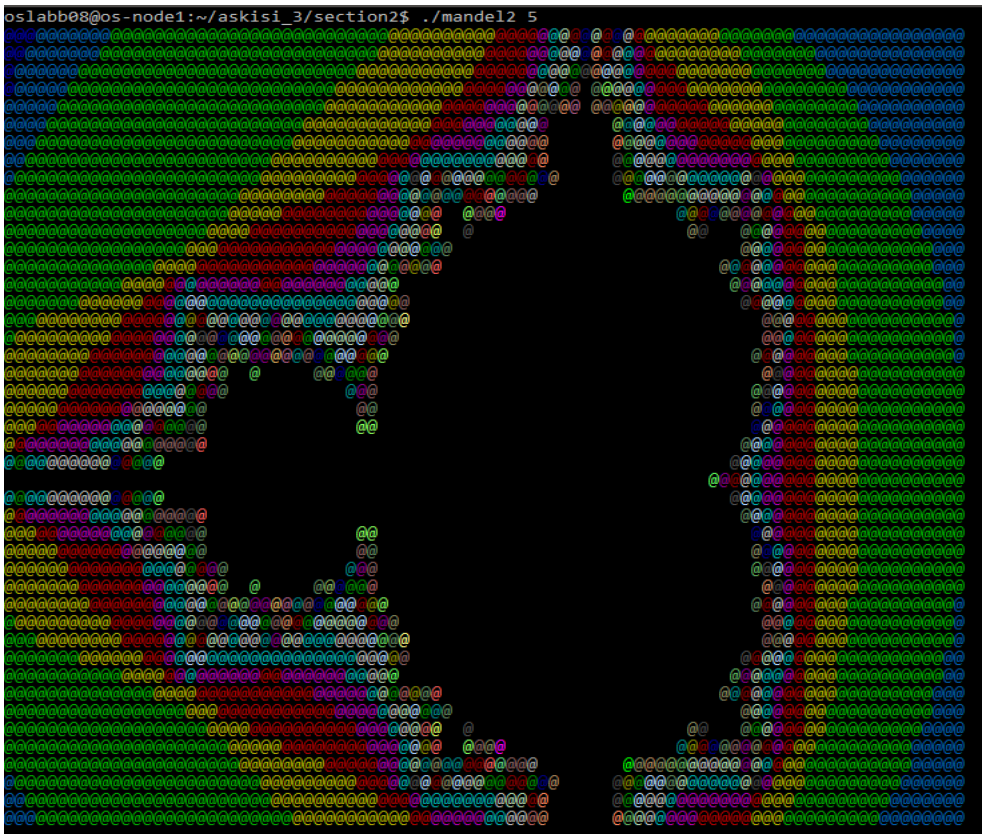
```
    /* Now that the line is done, output a newline character */
    if (write(fd, &newline, 1) != 1) {
       perror("compute_and_output_mandel_line: write newline");
       exit(1);
    }
}

void compute_and_output_mandel_line(int fd, int line)
{
    /*
     * A temporary array, used to hold color values for the line being drawn
     */
    int color_val[x_chars];

    compute_mandel_line(line, color_val);
    output_mandel_line(fd, color_val);
}

sem_t sema[50];

void* thread_start_fn(void* arg){
    struct thread_info_struct *thr = arg;

    int row_index;
    for (row_index = thr->thrid; row_index < y_chars; row_index += thr->thrcnt)
       compute_mandel_line(row_index, thr->arr[row_index]);

      for (row_index = thr->thrid; row_index < y_chars; row_index += thr->thrcnt){
      sem_wait(&sema[row_index]);
      output_mandel_line(1, thr->arr[row_index]);
      sem_post(&sema[row_index+1]);
      sem_post(&sema[row_index]);

      }

      return NULL;
}

int main(int argc, char ** argv)
{
    if(argc !=  2){
       printf("wrong usage: needs one int arguement:: the number of threads\n");
       return 1;
    }

    int nthreads;
    if (safe_atoi(argv[1], &nthreads) < 0 || nthreads <= 0) {
      fprintf(stderr, "`%s' is not valid for `thread_count'\n", argv[1]);
      exit(1);
```

```
  }

  // int color_table[y_chars][x_chars];

  int ** color_table = safe_malloc(y_chars * sizeof(int *));
  int i;
  for(i = 0; i < y_chars; i++)
   color_table[i] = safe_malloc(x_chars * sizeof(int));

  xstep = (xmax - xmin) / x_chars;
ystep = (ymax - ymin) / y_chars;

  sem_init(&sema[0], 0 ,1);
  for(i = 1; i < y_chars; i++){
  sem_init(&sema[i], 0 ,0);
  }

  struct thread_info_struct *thr;
  thr = safe_malloc(nthreads * sizeof(*thr));
int ret;
  for (i = 0; i < nthreads; i++) {
        /* Initialize per-thread structure */
        thr[i].arr = color_table;
        thr[i].thrid = i;
        thr[i].thrcnt = nthreads;

        /* Spawn new thread */
        ret = pthread_create(&(thr[i].tid), NULL, thread_start_fn, &thr[i]);
        if (ret) {
              perror_pthread(ret, "pthread_create");
              exit(1);
        }
  }

  /*
  * Wait for all threads to terminate
  */
  for (i = 0; i < nthreads; i++) {
        ret = pthread_join(thr[i].tid, NULL);
        if (ret) {
                    printf("ret on pthread_join kicks\n");
              perror_pthread(ret, "pthread_join");
              exit(1);
        }
  }

  for(i = 0; i < y_chars; i++){
  sem_destroy(&sema[i]);
  }
```

```
    reset_xterm_color(1);

    return 0;
}
```

Παράδειγμα εξόδου εκτέλεσης για υπολογισμό με 5 νήματα

Εντολή Εκτέλεσης: ./mandel 5



Ερωτήσεις

1) Για τον παραπάνω συγχρονισμό χρειάζονται 50 σημαφόροι (όσες και οι γραμμές που πρόκειται να τυπωθούν). Ο τρόπος που χρησιμοποιούνται είναι ο εξής: αρχικοποιούνται όλοι στην τιμή 0 εκτός από τον πρώτο sema[0]. Κάθε νήμα για να τυπώσει την γραμμή row_index πρέπει να “περάσει” την sem_wait(&sema[row_index]) και να μπει στο critical section. Αυτό αρχικά μπορεί να το κάνει μόνο το πρώτο νήμα αφού μόνο αυτός ο σημαφόρος έχει τιμή 1. Αφού, λοιπόν, τυπώσει την πρώτη γραμμή, το πρώτο νήμα ελευθερώνει με sem_post τον επόμενο σημαφόρο (sema[1]) ώστε να μπορέσει να τυπωθεί η επόμενη γραμμή. Έτσι, κάθε νήμα τυπώνει την γραμμή για την οποία έχει έρθει η σειρά και ελευθερώνει τον επόμενο σημαφόρο, εξασφαλίζοντας τη σωστή σειρά

εκτύπωσης των γραμμών. Σε σχέση με τον υπολογισμό τους, αυτός μπορεί να γίνει παράλληλα και δεν χρειάζεται συγχρονισμό.

2) Με την εκτέλεση της εντολής
`time ./mandel`
βλέπουμε ότι ο πραγματικός χρόνος εκτέλεσης (real) του προγράμματος με σειριακό υπολογισμό είναι για μια περίπτωση εκτέλεσης 1.017s. Ο αντίστοιχος χρόνος του προγράμματος με παράλληλο υπολογισμό και 2 νήματα είναι 0.538s.

3) Το παράλληλο πρόγραμμα εμφανίζει σημαντική επιτάχυνση καθώς ο υπολογισμός των γραμμών γίνεται παράλληλα και μόνο το τύπωμα αυτών γίνεται σειριακά με τον τρόπο που περιγράφηκε στο πρώτο ερώτημα .

4) Αν το πρόγραμμα τερματιστεί πρόωρα με interrupt από το πληκτρολόγιο, το τερματικό αφήνεται με διαφορετικό χρώμα χαρακτήρων. Αυτό συμβαίνει επειδή δεν καλείται η `reset_xterm_color(1)` για να επαναφέρει το χρώμα των χαρακτήρων στο τερματικό. Προκειμένου να αποφύγουμε αυτή την κατάσταση θα μπορούσαμε να υλοποιήσουμε έναν SIGINT handler για τη περίπτωση τερματισμού κατ' αυτόν τον τρόπο ο οποίος θα καλούσε τη `reset_xterm_color(1)` και θα τερμάτιζε το πρόγραμμα.
Παρατίθεται και το αντίστοιχο στιγμιότυπο εξόδου εκτέλεσης.