Λειτουργικά Συστήματα, 6ο Εξάμηνο, 2022-2023

Δημουλάς Ιωάννης, 03120083,
Ματζόρι Ενρίκα Ηλιάνα, 03120143
oslab20

Άσκηση 3η

Στο τέλος παρατίθεται το Makefile για τις 2 ασκήσεις.

1.1 Συγχρονισμός σε υπάρχοντα κώδικα

Παρατίθεται ο κώδικας της 1ης άσκησης simplesync.c :

```c
/*
* 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 mutex;  // Mutex for synchronization
/* ... */
#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_add_and_fetch(ip, 1);
      /* ... */
    } else {
      /* ... */
      pthread_mutex_lock(&mutex);
      /* You cannot modify the following line */
      ++(*ip);
      pthread_mutex_unlock(&mutex);
      /* ... */
    }
  }
  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 */
      //όρισμα ip είναι η διεύθυνση της μεταβλητής που θέλουμε να αλλάξουμε δλδ το ip είναι η διεύθυνση του val
      //για να μην μπλέκουμε με τα ip,&ip θα μπορούσαμε να βάλουμε το val global μεταβλητή και να γράφαμε πιο
//απλά
      //__sync_sub_and_fetch(&val, 1); και αντίστοιχα για το άλλο __sync_add_and_fetch(&val, 1);
      __sync_sub_and_fetch(ip, 1); //γράφουμε ip και όχι &ip διότι η τιμή του ip είναι η διεύθυνση του val ενώ το &ip
//είναι η
      // διεύθυνση του pointer. Εμείς θέλουμε να αλλάζουμε την τιμή του val και όχι την διεύθυνσή της.
      // Άρα, εμείς θέλουμε να αλλάξουμε την τιμή στην οποία δείχνει ο ip(δλδ το "περιεχόμενο" του ip)(γι'αυτό
//βάζουμε σκέτο ip)
      /* ... */
    } else {
      /* ... */
      pthread_mutex_lock(&mutex);
      /* You cannot modify the following line */
      --(*ip);
      pthread_mutex_unlock(&mutex);
```

```
      /* ... */ //
    }
  }
  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);

  //διαγράφουμε το mutex αφού τελειώσαμε και δεν το χρειαζόμαστε άλλο
  pthread_mutex_destroy(&mutex);

  return ok;
```

```
}
```

ΕΡΩΤΗΣΕΙΣ:

1) Χρησιμοποιήστε την εντολή time(1) για να μετρήσετε το χρόνο εκτέλεσης των εκτελέσιμων. Πώς συγκρίνεται ο χρόνος εκτέλεσης των εκτελέσιμων που εκτελούν συγχρονισμό, σε σχέση με το χρόνο εκτέλεσης του αρχικού προγράμματος χωρίς συγχρονισμό; Γιατί;

Χρησιμοποιώντας την εντολή time μετράμε τον χρόνο εκτέλεσης του προγράμματος simplesync στις διάφορες περιπτώσεις:

Χωρίς συγχρονισμό:

```
oslab20@orion:~/ask3/ask3-without-syncro$ time ./simplesync-atomic
About to decrease variable 10000000 times
About to increase variable 10000000 times
Done increasing variable.
Done decreasing variable.
NOT OK, val = -4898623.

real    0m0.315s
user    0m0.152s
sys     0m0.000s
```

```
oslab20@orion:~/ask3/ask3-without-syncro$ time ./simplesync-mutex
About to increase variable 10000000 times
About to decrease variable 10000000 times
Done decreasing variable.
Done increasing variable.
NOT OK, val = 968995.

real    0m0.309s
user    0m0.156s
sys     0m0.000s
```

Με συγχρονισμό:

Εκτελέσιμο με mutex:

```
oslab20@orion:~/ask3$ time ./simplesync-mutex
About to increase variable 10000000 times
About to decrease variable 10000000 times
Done increasing variable.
Done decreasing variable.
OK, val = 0.

real    0m3.388s
user    0m1.684s
sys     0m0.024s
```

Εκτελέσιμο με atomic operations:

```
oslab20@orion:~/ask3$ time ./simplesync-atomic
About to increase variable 10000000 times
About to decrease variable 10000000 times
Done increasing variable.
Done decreasing variable.
OK, val = 0.

real    0m1.375s
user    0m0.728s
sys     0m0.008s
```

Παρατηρούμε ότι ο χρόνος εκτέλεσης του προγράμματος χωρίς συγχρονισμό είναι αισθητά μικρότερος απ' ό,τι στην περίπτωση των εκτελέσιμων με συγχρονισμό. Αυτό είναι λογικό καθώς στην πρώτη περίπτωση όλα τα νήματα δρουν παράλληλα ενώ στην δεύτερη οι μαθηματικές πράξεις γίνονται σειριακά λόγω συγχρονισμού προκειμένου να επιτευχθεί η ορθότητα των πράξεων.

2) Ποια μέθοδος συγχρονισμού είναι γρηγορότερη, η χρήση ατομικών λειτουργιών ή η χρήση POSIX mutexes; Γιατί;

Αξίζει να σημειωθεί ότι μεταξύ των εκτελέσιμων με συγχρονισμό, ταχύτερο είναι αυτό που χρησιμοποιεί atomic operations. Αυτά εγγυώνται την ατομική εκτέλεση των εντολών που βρίσκονται στο κρίσιμο κομμάτι κώδικα και χρησιμοποιούν κατευθείαν το υλισμικό(hardware) χωρίς καμία εμπλοκή του λειτουργικού συστήματος όπως στην περίπτωση των mutexes, τα οποία εμπλέκουν και το Λειτουργικό Σύστημα ώστε να θέτει σε αναμονή/να ξυπνάει τα νήματα όταν είναι απαραίτητο για τα διάφορα locks/unlocks. Η εμπλοκή αυτή του ΛΣ αυξάνει λοιπόν τον απαιτούμενο χρόνο εκτέλεσης. Αυτό επιβεβαιώνεται και από την μέτρηση του χρόνου εκτέλεσης των προγραμμάτων που έγινε παραπάνω.

3) Σε ποιες εντολές του επεξεργαστή μεταφράζεται η χρήση ατομικών λειτουργιών του GCC στην αρχιτεκτονική για την οποία μεταγλωττίζετε; Χρησιμοποιήστε την παράμετρο -S του GCC για να παράγετε τον ενδιάμεσο κώδικα Assembly, μαζί με την παράμετρο -g για να συμπεριλάβετε πληροφορίες γραμμών πηγαίου κώδικα (π.χ., “.loc 1 63 0”), οι οποίες μπορεί να σας διευκολύνουν. Δείτε την έξοδο της εντολής make για τον τρόπο μεταγλώττισης του simplesync.c.

Παρακάτω έχουμε το τροποποιημένο Makefile ώστε να δούμε και τον κώδικα assembly για τα 2 εκτελέσιμα.

```
## Simple sync (two versions)
simplesync-mutex: simplesync-mutex.o
	$(CC) $(CFLAGS) -o simplesync-mutex simplesync-mutex.o $(LIBS)

simplesync-atomic: simplesync-atomic.o
	$(CC) $(CFLAGS) -o simplesync-atomic simplesync-atomic.o $(LIBS)

simplesync-mutex.o: simplesync.c
	$(CC) $(CFLAGS) -DSYNC_MUTEX -c -o simplesync-mutex.o simplesync.c

simplesync-atomic.o: simplesync.c
	$(CC) $(CFLAGS) -DSYNC_ATOMIC -c -o simplesync-atomic.o simplesync.c

simplesync-mutex.s: simplesync.c
	$(CC) $(CFLAGS) -DSYNC_MUTEX -S -g -o simplesync-mutex.s simplesync.c

simplesync-atomic: simplesync-atomic.o
	$(CC) $(CFLAGS) -DSYNC_ATOMIC -S -g -o simplesync-atomic.s simplesync.c
```

Για την περίπτωση του εκτελέσιμου αρχείου με atomic operations έχουμε:
Η εντολή `__sync_add_and_fetch(ip, 1);` μεταφράζεται σε assembly ως εξής:

```
.L2:
        .loc 1 50 0
        lock addl       $1, (%rbx)
```

Και η `__sync_sub_and_fetch(ip, 1);` ως εξής:

```
.L7:
        .loc 1 79 0
        lock subl       $1, (%rbx)
```

Παρατηρούμε το lock δίπλα απ'τα addl, subl το οποίο εξασφαλίζει ότι όσο εκτελείται αυτή η εντολή δεν μπορεί κανένα άλλο νήμα να επέμβει σε αυτήν την διεύθυνση, εξασφαλίζοντας έτσι την ατομικότητα.

4) Σε ποιες εντολές μεταφράζεται η χρήση POSIX mutexes στην αρχιτεκτονική για την οποία μεταγλωττίζετε; Παραθέστε παράδειγμα μεταγλώττισης λειτουργίας pthread_mutex_lock() σε Assembly, όπως στο προηγούμενο ερώτημα.

Για την περίπτωση του εκτελέσιμου με POSIX mutexes έχουμε:

Για το thread που εκτελεί το κομμάτι κώδικα που αφορά την increase_fn( ) έχουμε τις παρακάτω εντολές σε assembly:

```
.L2:
        .loc 1 54 0
        movl    $mutex, %edi
        call    pthread_mutex_lock
```

```
.LVL4:
        .loc 1 56 0
        movl    0(%rbp), %eax
        .loc 1 57 0
        movl    $mutex, %edi
        .loc 1 56 0
        addl    $1, %eax
        movl    %eax, 0(%rbp)
        .loc 1 57 0
        call    pthread_mutex_unlock
```

Αντίστοιχα και για το thread που εκτελεί την decrease_fn( ):

```
.L7:
        .loc 1 86 0
        movl    $mutex, %edi
        call    pthread_mutex_lock
.LVL14:
        .loc 1 88 0
        movl    0(%rbp), %eax
        .loc 1 89 0
        movl    $mutex, %edi
        .loc 1 88 0
        subl    $1, %eax
        movl    %eax, 0(%rbp)
        .loc 1 89 0
        call    pthread_mutex_unlock
```

1.2 Παράλληλος υπολογισμός του συνόλου Mandelbrot

Παρατίθεται ο κώδικας για την πρώτη εκδοχή του προγράμματος που αφορά τον συγχρονισμό των νημάτων **με σημαφόρους:**

```
/*
* mandel.c
*
* A program to draw the Mandelbrot Set on a 256-color xterm.
*
*/
#include <errno.h>
#include <unistd.h>
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>
#include <assert.h>
#include <string.h>
#include <math.h>
#include <stdlib.h>
#include <semaphore.h>

#include "mandel-lib.h"

#define MANDEL_MAX_ITERATION 100000

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

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

/*
* 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;

//φτιάχνουμε τον σημαφόρο
sem_t *sem;
//FROM phthread-test.c
/*
```

```
* 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)

/*
* A (distinct) instance of this structure
* is passed to each thread
*/
struct thread_info_struct {
  pthread_t tid; /* POSIX thread id, as returned by the library */

  //int* color_val; /* 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;
}

void usage(char *argv0)
{
  fprintf(stderr, "Usage: %s thread_count array_size\n\n"
            "Exactly two argument required:\n"
            "    thread_count: The number of threads to create.\n"
            "    array_size: The size of the array to run with.\n",
        argv0);
  exit(1);
```

```
}



/*
* 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(void *arg)
{
  /*
   * A temporary array, used to hold color values for the line being drawn
   */
  int color_val[x_chars];
  struct thread_info_struct *thr = arg;
  int i;
  //το κάθε thread αναλαμβάνει τις γραμμές  i,i + n,i + 2×n,i+3×n,... όπου n=thrcnt=ο αριθμός των threads
  for(i=thr->thrid; i<y_chars; i+= thr->thrcnt){
    //μπορούν και όλα παράλληλα να κάνουν το compute αλλά το output πρέπει να γίνεται σειριακά,
    //οπότε εντοπίζουμε σαν κρίσιμο κομμάτι κώδικα το output_mandel_line(1, color_val);
    compute_mandel_line(i, color_val);
    sem_wait(&sem[thr->thrid]);
    //λέω στον επόμενο σημαφόρο να αυξηθεί η τιμή του κατά 1 και άρα να περάσει το wait_sem
    //χρησιμοποιούμε το mod διότι όταν φτάσουμε στο τελεταίο
    //όταν φτάσω στο i = thrid του τελευταίου thread αυτό (thr->thrid)+1) δεν υπάρχει
    output_mandel_line(1, color_val);
    sem_post(&sem[((thr->thrid)+1)% thr->thrcnt]);
  }
  return NULL;
}

int main(int argc, char *argv[])
{

  int i, thrcnt , ret;
  struct thread_info_struct *thr;

  /*
   * Parse the command line
   */
  if (argc != 2)
    usage(argv[0]);
  //thrcnt=ο αριθμός των threads που περνάμε σαν input
  if (safe_atoi(argv[1], &thrcnt) < 0 || thrcnt <= 0) {
    fprintf(stderr, "`%s' is not valid for `thread_count'\n", argv[1]);
    exit(1);
  }

  /*
  * Allocate and initialize big array of doubles
  */
```

```
  thr = safe_malloc(thrcnt * sizeof(*thr)); //πίνακας απο structs
  sem = safe_malloc(thrcnt * sizeof(sem_t)); //ένας πίνακας που θα περιέχει όσους σημαφόρους όσα και τα νήματα
  //int line;

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

  /*
   * draw the Mandelbrot Set, one line at a time.
   * Output is sent to file descriptor '1', i.e., standard output.
   */
// for (line = 0; line < y_chars; line++) {
//    compute_and_output_mandel_line(1, line);
// }
  for (i = 0; i < thrcnt; i++) {
    /* Initialize per-thread structure */
    thr[i].thrid = i;
    thr[i].thrcnt = thrcnt;
    //το πρώτο therad είναι το μόνο που θα "μπει" στο sem_wait και
    // γι'αυτό του βάλαμε αρχική τιμή το 1(ώστε το wait να την κάνει decrease κατά 1)
    // ενώ τις υπόλοιπες τις αρχικοποιούμε όλες να είναι σε locked state
    if(i==0){
      sem_init(&sem[i],0,1); //unlocked-state
    }
    else{
      sem_init(&sem[i],0,0); //locked-state
    }
    /* Spawn new thread */
    ret = pthread_create(&thr[i].tid, NULL, compute_and_output_mandel_line, &thr[i]);
    if (ret) {
      perror_pthread(ret, "pthread_create");
      exit(1);
    }
  }

  //περιμένουμε το κάθε thread που δημιουργήθηκε να τελειώσει
  for (i = 0; i < thrcnt; i++) {
    ret=pthread_join(thr[i].tid, NULL);
    if (ret) {
      perror_pthread(ret, "pthread_join");
      exit(1);
    }
  }
  //διαγράφουμε τους σημαφόρους
  sem_destroy(sem);

  reset_xterm_color(1);
  return 0;
}
```

Παρατίθεται ο κώδικας και για την δεύτερη εκδοχή του προγράμματος που αφορά τον συγχρονισμό των νημάτων με μεταβλητές συνθήκης(condition variables):

```
/*
* mandel.c
*
* A program to draw the Mandelbrot Set on a 256-color xterm.
*
*/
                                    //CONDITION VARIABLES
#include <errno.h>
#include <unistd.h>
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>
#include <assert.h>
#include <string.h>
#include <math.h>
#include <stdlib.h>
#include <semaphore.h>

#include "mandel-lib.h"

#define MANDEL_MAX_ITERATION 100000

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

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

/*
* 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;

//φτιάχνουμε το mutex
pthread_mutex_t mutex;
pthread_cond_t condition;
int count=0;
```

```c
//FROM phthread-test.c
/*
* 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)

/*
* A (distinct) instance of this structure
* is passed to each thread
*/
struct thread_info_struct {
   pthread_t tid; /* POSIX thread id, as returned by the library */

   //int* color_val; /* 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;
}

void usage(char *argv0)
{
   fprintf(stderr, "Usage: %s thread_count array_size\n\n"
                   "Exactly two argument required:\n"
                   "    thread_count: The number of threads to create.\n"
```

```
                "    array_size: The size of the array to run with.\n",
        argv0);
  exit(1);
}

/*
* 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(void *arg)
{
    /*
     * A temporary array, used to hold color values for the line being drawn
     */
    int color_val[x_chars]; //(πλήθους όσα ειναι και τα threads)
    struct thread_info_struct *thr = arg;
    int i;

    //το κάθε thread αναλαμβάνει τις γραμμές  i,i + n,i + 2×n,i+3×n,... όπου n=thrcnt=ο αριθμός των threads
    for(i=thr->thrid; i<y_chars; i+= thr->thrcnt){
        //μπορούν και όλα παράλληλα να κάνουν το compute αλλά το output πρέπει να γίνεται σειριακά,
        //οπότε εντοπίζουμε σαν κρίσιμο κομμάτι κώδικα το output_mandel_line(1, color_val);
        compute_mandel_line(i, color_val);
        //φτάνουν όλα τα threads εδώ και μόνο 1 θα μπει
        pthread_mutex_lock(&mutex);
         //χρησιμοποιούμε το count προκειμένου τα threads να μπαίνουν με σειρά και τάξη στο output_mandel_line ώστε να
        //τυπώνονται οι γραμμές με την σωστή σειρά
          //αυτός που πρέπει να προχωρήσει στην εκτύπωση είναι αυτός του οποίου το thrid(0,1,2) ισούται με το εκάστοτε count
        //(προκειμένου να μην τυπωθεί μία γραμμή πριν από κάποια αλλη!)
        while(count != thr->thrid){
                //τα threads που δεν είναι ακόμα η σειρά τους να εκτυπώσουν κάνουν wait(δίνοντας το κλειδί του lock στο επόμενο thread)
            // μέχρι να ικανοποιηθεί το condition, πρακτικά δηλαδή μέχρι αυτός που θα τυπώσει να τους δώσει την εντολή
            // να ξυπνήσουν.
            pthread_cond_wait(&condition, &mutex);
        }
        output_mandel_line(1, color_val);

        //αυξάνουμε το count και έτσι ξέρουμε ποιο θα είναι το επόμενο thread που θα τυπώσει(και εκ τούτου ποιες
        // γραμμές θα τυπωθούν)
            //προφανώς, όπως και στην άσκηση με τους σημαφόρους χρειαζόμαστε το mod καθώς το count θέλουμε να παίρνει μέχρι
        // την τιμή του thrid του τελευταίου thread, και όταν τυπώσει και εκείνο την γραμμή που του αντιστοιχεί
        // κάθε φορά να ξαναγυρίσουμε στο πρώτο thread επαναλαμβάνοντας την διαδικασία
        count  = (count+1) % thr->thrcnt;

          // Επειδή έχουμε cond_broadcast, αυτό ξυπνάει όλα τα threads που κοιμούνται μέσα στο while.Τότε, το count έχει
        // αλλάξει και άρα όλα τα threads κοιτάνε αν ισχύει η όχι η συνθήκη του while.Για όσα ισχύει η συνθήκη
        // ξανακοιμούνται ενώ το 1 που ξύπνησε και δεν ξανακοιμήθηκε προχωράει στην εκτύπωση
        pthread_cond_broadcast(&condition);
```

```
    //κάνουμε unlock γιατί εδώ ολοκληρώνεται η κρίσιμη περιοχή κώδικα
    pthread_mutex_unlock(&mutex);

  }
  return NULL;
}

int main(int argc, char *argv[])
{

  int i, thrcnt , ret;
  struct thread_info_struct *thr;

  /*
   * Parse the command line
   */
  if (argc != 2)
    usage(argv[0]);
  //thrcnt=ο αριθμός των threads που περνάμε σαν input
  if (safe_atoi(argv[1], &thrcnt) < 0 || thrcnt <= 0) {
    fprintf(stderr, "`%s' is not valid for `thread_count'\n", argv[1]);
    exit(1);
  }

  //φτιάχνουμε έναν πίνακα από objects(structs) στον οποίο αποθηκεύουμε ,σε κάθε κελί,
  //τις πληροφορίες του thread στο οποίο αντιστοιχεί
  thr = safe_malloc(thrcnt * sizeof(*thr));


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

  //γεμίζουμε τον πίνακα
  for (i = 0; i < thrcnt; i++) {
    /* Initialize per-thread structure */
    thr[i].thrid = i;
    thr[i].thrcnt = thrcnt;

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

  //περιμένουμε το κάθε thread που δημιουργήθηκε να τελειώσει
  for (i = 0; i < thrcnt; i++) {
    ret=pthread_join(thr[i].tid, NULL);
    if (ret) {
      perror_pthread(ret, "pthread_join");
      exit(1);
    }
```

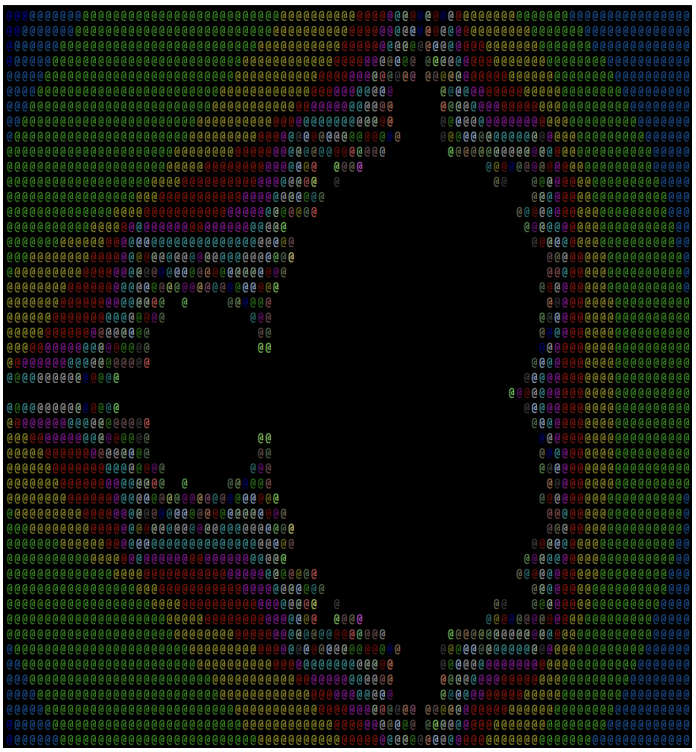
```
  }
  //διαγράφουμε το mutex και το condition αφού τελειώσαμε και δεν τα χρειαζόμαστε άλλο
  pthread_mutex_destroy(&mutex);
  pthread_cond_destroy(&condition);

  reset_xterm_color(1);
  return 0;
}
```

ΕΡΩΤΗΣΕΙΣ:

1) Πόσοι σημαφόροι χρειάζονται για το σχήμα συγχρονισμού που υλοποιείτε;

Το κρίσιμο κομμάτι κώδικα (critical section) είναι το σημείο της εκτύπωσης των γραμμών, καθώς αυτή πρέπει να γίνεται σειριακά και να γίνεται απ' το κάθε thread ξεχωριστά, σε αντίθεση με τον υπολογισμό(compute)των γραμμών που θα τυπώσει το κάθε thread που μπορεί να γίνεται παράλληλα απ' όλα τα threads αφού δεν επηρεάζονται κάπως μεταξύ τους. Για το σχήμα συγχρονισμού που υλοποιήσαμε χρησιμοποιούμε τόσους σημαφόρους όσους και τα threads. Επειδή μας ενδιαφέρει η σειρά που τα threads θα μπουν στο κρίσιμο κομμάτι κώδικα και, συγκεκριμένα, θέλουμε το πρώτο thread(με thread id = 0) να είναι και το πρώτο που θα τυπώσει, με την εντολή sem_init(&sem[i],0,1); , αρχικοποιούμε τον "σημαφόρο του" σε unlocked state, δηλαδή με αρχική τιμή 1, ώστε να είναι το μόνο νήμα που θα "μπει" στο κρίσιμο κομμάτι κώδικα, και άρα θα "περάσει" το sem_wait(). Όταν το πρώτο αυτό thread θα εκτυπώσει την πρώτη γραμμή, μέσω της εντολής sem_post() θα δοθεί εντολή στο επόμενο κυκλικά thread(δηλαδή στο συγκεκριμένο παράδειγμα στο 1), να αυξηθεί η τιμή του κατά 1 και άρα να μπορεί πλέον να "περάσει" και εκείνο το sem_wait(). Κατ'αυτόν τον τρόπο επιτυγχάνεται η σειριακή εκτύπωση των γραμμών και επομένως έχουμε το επιθυμητό αποτέλεσμα.

Ενδεικτικά παρατίθεται η έξοδος του προγράμματος για N=3:



2) Πόσος χρόνος απαιτείται για την ολοκλήρωση του σειριακού και του παράλληλου προγράμματος με δύο νήματα υπολογισμού; Χρησιμοποιήστε την εντολή time(1) για να χρονομετρήσετε την εκτέλεση ενός προγράμματος, π.χ., time sleep 2. Για να έχει νόημα η μέτρηση, δοκιμάστε σε ένα μηχάνημα που διαθέτει επεξεργαστή δύο πυρήνων. Χρησιμοποιήστε την εντολή cat /proc/cpuinfo για να δείτε πόσους υπολογιστικούς πυρήνες διαθέτει κάποιο μηχάνημα.

Το μηχάνημα που θα χρησιμοποιήσουμε διαθέτει 8 πυρήνες, το οποίο φαίνεται τρέχοντας την εντολή "lscpu" :

```
oslab20@os-node1:~$ lscpu
Architecture:                    x86_64
CPU op-mode(s):                  32-bit, 64-bit
Byte Order:                      Little Endian
Address sizes:                   40 bits physical, 48 bits virtual
CPU(s):                          8
On-line CPU(s) list:             0-7
Thread(s) per core:              1
Core(s) per socket:              1
Socket(s):                       8
NUMA node(s):                    1
Vendor ID:                       AuthenticAMD
CPU family:                      6
Model:                           6
```

Παρατηρούμε τον χρόνο εκτέλεσης του προγράμματος για διαφορετικό πλήθος νημάτων:
Για 1 νήμα(σειριακός υπολογισμός):

```
real    0m1.318s
user    0m1.269s
sys     0m0.048s
```

Για 2 νήματα:

```
real    0m0.783s
user    0m1.474s
sys     0m0.066s
```

Για 4 νήματα:

```
real    0m0.438s
user    0m1.465s
sys     0m0.069s
```

Για 8 νήματα:

```
real    0m0.274s
user    0m1.476s
sys     0m0.131s
```

Παρατηρούμε ότι ο χρόνος εκτέλεσης του προγράμματος όταν έχουμε παράλληλο υπολογισμό με 2 νήματα είναι σχεδόν υποδιπλάσιος απ' τον χρόνο εκτέλεσης του προγράμματος με σειριακό τρόπο. Κάνοντας διάφορες δοκιμές για πλήθος νημάτων μεγαλύτερο του πλήθους των πυρήνων(δηλ. N>8) μέχρι κάποιο όριο, παρατηρούμε ότι οι διαφορές στον χρόνο εκτέλεσης του προγράμματος είναι αμελητέες. Για μεγάλο αριθμό νημάτων(πχ για N=10.000) παρατηρούμε ότι ο χρόνος εκτέλεσης αυξάνεται σημαντικά. Αυτό συμβαίνει,προφανώς, λόγω του ότι ο χρόνος δημιουργίας των νημάτων δεν είναι πλέον τόσο αμελητέος, δεδομένου του πλήθους τους.

3) Πόσες μεταβλητές συνθήκης χρησιμοποιήσατε στη δεύτερη εκδοχή του προγράμματος σας? Αν χρησιμοποιηθεί μια μεταβλητή πως λειτουργεί ο συγχρονισμός και ποιο πρόβλημα επίδοσης υπάρχει?

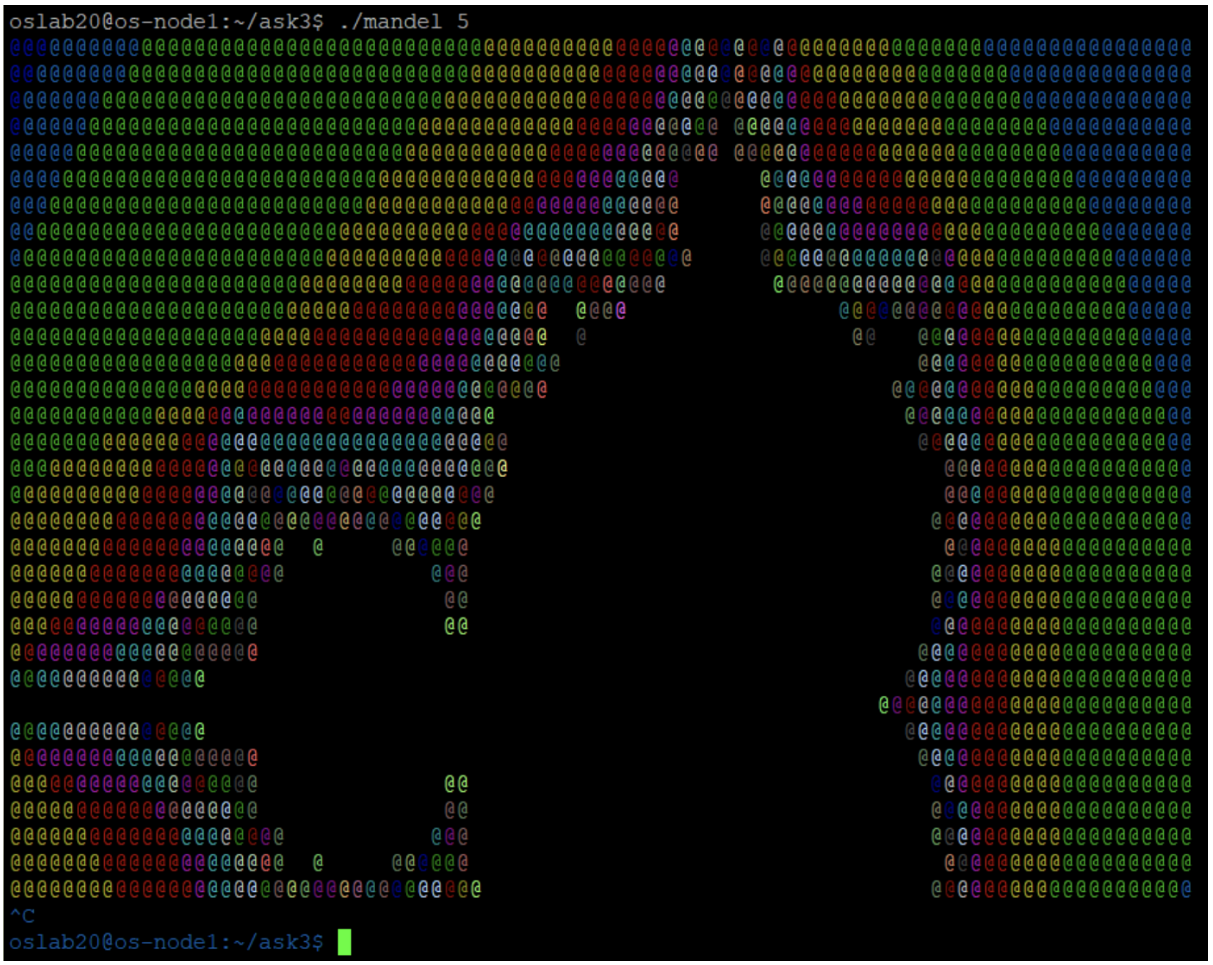
Στην δεύτερη εκδοχή του προγράμματός μας χρησιμοποιήσαμε 1 μεταβλητή συνθήκης. Όταν αυτή ικανοποιείται, το νήμα που μόλις εκτέλεσε το χρήσιμο κομμάτι κώδικα στέλνει εντολή στα υπόλοιπα νήματα που κοιμούνται για να τα "ξυπνήσει" μέσω της εντολής pthread_cond_broadcast(&condition). Ύστερα απ' αυτήν την εντολή, όλα τα νήματα θα ξυπνήσουν αλλά λόγω της συνθήκης μέσα στο while loop, μόνο 1 thread κάθε φορά θα συνεχίσει και θα προχωρήσει στην εκτύπωση(ενώ όλα τα άλλα θα ξανακοιμηθούν με την εντολή pthread_cond_wait(&condition, &mutex);). Το γεγονός αυτό μπορεί να αποτελέσει ένα πρόβλημα στην επίδοση του προγράμματός μας σε σχέση με το αν χρησιμοποιούταν ξεχωριστή μεταβλητή συνθήκης για κάθε thread. Παρόλο που αυτό θα μπορούσε να βελτιωθεί με την χρήση της συνάρτησης pthread_cond_signal(&condition) ώστε κάθε νήμα να αφορά συγκεκριμένη και ξεχωριστή μεταβλητή συνθήκης, η περιπλοκότητα του κώδικα και η μνήμη που θα χρειαζόταν για αυτήν την υλοποίηση δεν θα συνέφερε, δεδομένου ότι ήδη η επίδοση του προγράμματος που παρατέθηκε δεν επιβραδύνεται ουσιαστικά.

4) Το παράλληλο πρόγραμμα που φτιάξατε, εμφανίζει επιτάχυνση; Αν όχι, γιατί; Τι πρόβλημα υπάρχει στο σχήμα συγχρονισμού που έχετε υλοποιήσει; Υπόδειξη: Πόσο μεγάλο είναι το κρίσιμο τμήμα; Χρειάζεται να περιέχει και τη φάση υπολογισμού και τη φάση εξόδου κάθε γραμμής που παράγεται;

Το παράλληλο πρόγραμμα που υλοποιήσαμε εμφανίζει επιτάχυνση σε σχέση με το πρόγραμμα χωρίς τον συγχρονισμό για 1 μόνο νήμα. Αυτό συμβαίνει καθώς επιτυγχάνουμε να έχουμε παράλληλο υπολογισμό(compute) από πολλά νήματα αλλά σειριακή εκτύπωση(output). Αυτό επιτυγχάνεται μέσω του στρατηγικού ορισμού ως κρίσιμου τμήματος κώδικα μόνο ό,τι αφορά την εκτύπωση. Αντιθέτως, δεν θα παρατηρούσαμε καμία επιτάχυνση αν στο κρίσιμο κομμάτι κώδικα συμπεριλαμβάναμε και το τμήμα του υπολογισμού. Οπότε πρακτικά, όλα θα γινόντουσαν σειριακά.

5) Τι συμβαίνει στο τερματικό αν πατήσετε Ctrl-C ενώ το πρόγραμμα εκτελείται; Σε τι κατάσταση αφήνεται, όσον αφορά το χρώμα των γραμμάτων; Πώς θα μπορούσατε να επεκτείνετε το mandel.c σας ώστε να εξασφαλίσετε ότι ακόμη κι αν ο χρήστης πατήσει Ctrl-C, το τερματικό θα επαναφέρεται στην προηγούμενη κατάστασή του;

Αν πατήσουμε Ctrl-C ενώ τρέχει το πρόγραμμα θα έχουμε το εξής αποτέλεσμα:



Πρακτικά δηλαδή το πρόγραμμα τερματίζει την στιγμή που πατάμε το Ctrl-C, αλλά το χρώμα των γραμμάτων του τερματικού αλλάζει από λευκό στο τελευταίο χρώμα που εκτυπώθηκε(εδώ μπλε).Προκειμένου να αποφευχθεί αυτό θα χρησιμοποιήσουμε ένα signal handler. Ενδεικτικά το κάνουμε στο πρώτο εκτελέσιμο του προγράμματος, όπου ο συγχρονισμός γίνεται με σημαφόρους.
Οι διαφοροποιήσεις λοιπόν στον κώδικα, κάνοντας #include <signal.h>, θα είναι οι εξής:

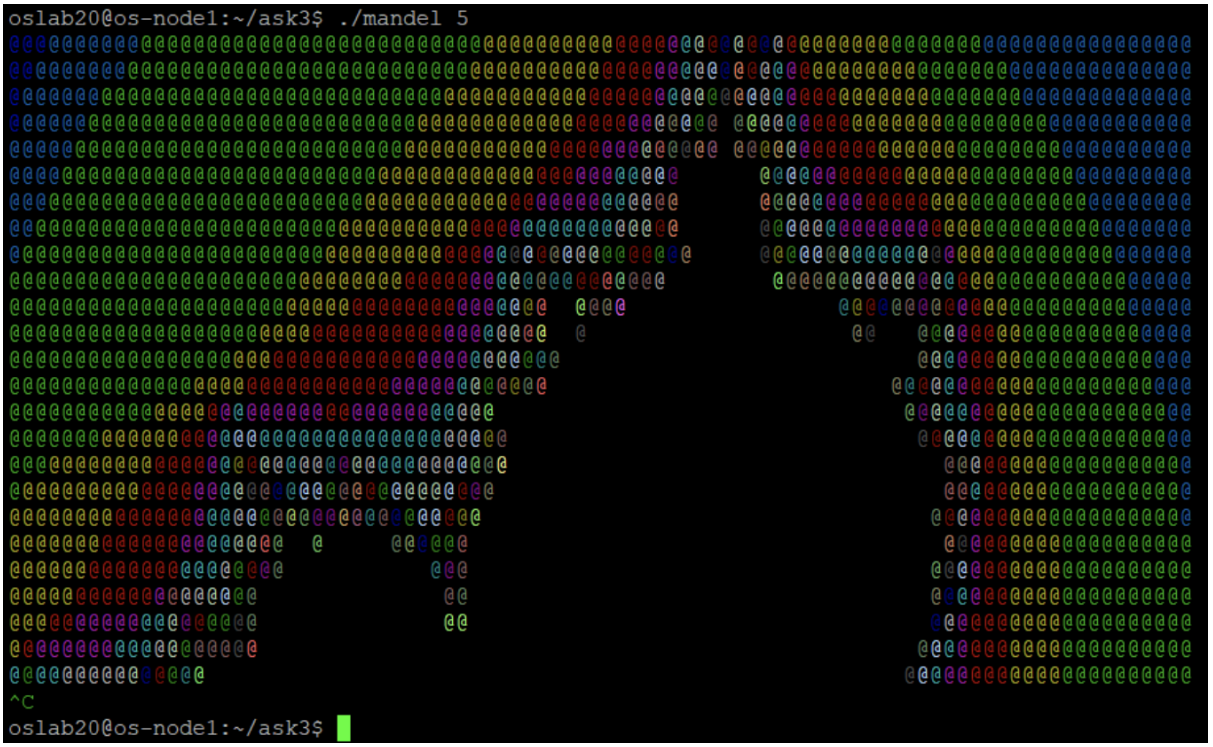
```
//αν πατηθεί Ctrl-C και το τερματικό παραμείνει χρωματισμένο
void signal_handler(int x){
   reset_xterm_color(1); //επαναφέρει το χρώμα του τερματικού σε λευκό
   printf("\n");
   exit(x);
}
```

Και στην compute_and_output_mandel_line() προσθέτουμε την κόκκινη εντολή:

```
for(i=thr->thrid; i<y_chars; i+= thr->thrcnt){
  //μπορούν και όλα παράλληλα να κάνουν το compute αλλά το output πρέπει να
γίνεται σειριακά,
```

```
   //οπότε εντοπίζουμε σαν κρίσιμο κομμάτι κώδικα το output_mandel_line(1,
color_val);
  compute_mandel_line(i, color_val);
  sem_wait(&sem[thr->thrid]);
  //λέω στον επόμενο σημαφόρο να αυξηθεί η τιμή του κατά 1 και άρα να περάσει
το wait_sem
  //χρησιμοποιούμε το mod διότι όταν φτάσουμε στο τελεγταίο
   //όταν φτάσω στο i = thrid του τελευταίου thread αυτό (thr->thrid)+1) δεν
υπάρχει
  output_mandel_line(1, color_val);
  sem_post(&sem[((thr->thrid)+1)% thr->thrcnt]);
  signal(SIGINT, signal_handler);
}
```

Πλέον, αν πατήσουμε Ctrl-C θα έχουμε το εξής αποτέλεσμα:



Τέλος, παρατίθεται το Makefile για τις 2 ασκήσεις:

```
#
# Makefile
#

CC = gcc

# CAUTION: Always use '-pthread' when compiling POSIX threads-based
# applications, instead of linking with "-lpthread" directly.
CFLAGS = -Wall -O2 -pthread
LIBS =

all:   pthread-test   simplesync-mutex   simplesync-atomic   kgarten   mandel
mandel-test mandel-cond-var
```

```
## Pthread test
pthread-test: pthread-test.o
	$(CC) $(CFLAGS) -o pthread-test pthread-test.o $(LIBS)

pthread-test.o: pthread-test.c
	$(CC) $(CFLAGS) -c -o pthread-test.o pthread-test.c

## Simple sync (two versions)
simplesync-mutex: simplesync-mutex.o
	$(CC) $(CFLAGS) -o simplesync-mutex simplesync-mutex.o $(LIBS)

simplesync-atomic: simplesync-atomic.o
	$(CC) $(CFLAGS) -o simplesync-atomic simplesync-atomic.o $(LIBS)

simplesync-mutex.o: simplesync.c
	$(CC) $(CFLAGS) -DSYNC_MUTEX -c -o simplesync-mutex.o simplesync.c

simplesync-atomic.o: simplesync.c
	$(CC) $(CFLAGS) -DSYNC_ATOMIC -c -o simplesync-atomic.o simplesync.c

simplesync-mutex.s: simplesync.c
	$(CC) $(CFLAGS) -DSYNC_MUTEX -S -g -o simplesync-mutex.s simplesync.c

simplesync-atomic.s: simplesync.c
	$(CC) $(CFLAGS) -DSYNC_ATOMIC -S -g -o simplesync-atomic.s simplesync.c


## Kindergarten
kgarten: kgarten.o
	$(CC) $(CFLAGS) -o kgarten kgarten.o $(LIBS)

kgarten.o: kgarten.c
	$(CC) $(CFLAGS) -c -o kgarten.o kgarten.c


## Mandel
mandel: mandel-lib.o mandel.o
	$(CC) $(CFLAGS) -o mandel mandel-lib.o mandel.o $(LIBS)

mandel-lib.o: mandel-lib.h mandel-lib.c
	$(CC) $(CFLAGS) -c -o mandel-lib.o mandel-lib.c $(LIBS)

mandel.o: mandel.c
	$(CC) $(CFLAGS) -c -o mandel.o mandel.c $(LIBS)

mandel-test: mandel-lib.o mandel-test.o
	$(CC) $(CFLAGS) -o mandel-test mandel-lib.o mandel-test.o $(LIBS)

mandel-test.o: mandel-test.c
	$(CC) $(CFLAGS) -c -o mandel-test.o mandel-test.c $(LIBS)

mandel-cond-var: mandel-lib.o mandel-cond-var.o
	$(CC) $(CFLAGS) -o mandel-cond-var mandel-lib.o mandel-cond-var.o $(LIBS)
```

```
mandel-cond-var.o: mandel-cond-var.c
	$(CC) $(CFLAGS) -c -o mandel-cond-var.o mandel-cond-var.c $(LIBS)

clean:
	rm -f *.s *.o pthread-test simplesync-{atomic,mutex} kgarten mandel
```