# Seminar #3. Green threads

Albert Asratyan, TCOMK3

December 2018

## 1 Introduction

In this assignment we will implement our own thread library. For that, a scheduler and context handler will be created. However, operating system's threads are still going to be used for benchmarking and comparing to the created implementation. Regarding context, the following functions are going to be used: getcontext(), makecontext(), setcontext() and swapcontext(). green_create(), green_yield() and green_join() are going to be created and used instead of the corresponding pthread_create(), pthread_yield() and pthread_join().

## 2 Handling threads using queues

First, we need to set up a queue structure for the threads. The implementation is rather straight forward:

```
typedef struct queue {
  struct green_t *head;
  struct green_t *tail;
} queue;
```

Then, two functions, enqueue() and dequeue() are required to work with the queue that we have established:

```
static void enqueue(queue *queue, green_t *thread) {
  if (queue->head == NULL || queue->tail == NULL) {
    queue->head = thread;
    queue->tail = thread;
  } else {
    queue->tail->next = thread;
    queue->tail = thread;
  }
}

static green_t *dequeue(queue *queue) {
  green_t *thread;
```

```
  thread = queue->head;
  queue->head = queue->head->next;
  if (thread->next == NULL) {
    queue->tail == NULL;
  }
  thread->next = NULL;

  return thread;
}
```

## 2.1 green_yield()

This function suspends the currently running thread and selects a new thread for execution. Our version for pthread_yield:

```
int green_yield() {
  blockI();
  green_t * susp = running;
  enqueue(readyqueue, susp);
  green_t *next = dequeue(readyqueue);
  running = next;
  swapcontext(susp->context, next->context);
  unblockI();
  return 0;
}
```

## 2.2 green_join()

This function waits for the specified thread to terminate. If that thread has been terminated already, returns immediately.

```
int green_join(green_t *thread) {
  if(thread->zombie)
    return 0;
  blockI();
  green_t *susp = running;
  thread->join = susp;
  green_t *next = dequeue(readyqueue);
  running = next;
  swapcontext(susp->context, next->context);
  unblockI();
  return 0;
}
```

### 2.3 green_cond_init()

This function initializes the condition variable referenced by cond.

```
void green_cond_init(green_cond_t *cond) {
  cond->queue = malloc(sizeof(queue));
}
```

### 2.4 green_con_wait()

This function is used to block on a condition variable. It is called with mutex locked by the calling thread or undefined behavior will result.

```
int green_cond_wait(green_cond_t *cond, green_mutex_t *mutex) {
  blockI();
  green_t *susp = running;
  enqueue(cond->queue, susp);
  if(mutex != NULL) {
    mutex->taken = FALSE;
    green_t *thread = mutex->susp;
    enqueue(readyqueue, thread);
  }
  green_t *next = dequeue(readyqueue);
  running = next;
  swapcontext(susp->context, next->context);
  if(mutex != NULL) {
    while(mutex->taken) {
      mutex->susp = susp;
      green_t *next = dequeue(readyqueue);
      running = next;
      swapcontext(susp->context, next->context);
    }
    mutex->taken = TRUE;
  }
  unblockI();
  return 0;
}
```

### 2.5 green_cond_signal()

This function is used to unblock threads on a condition variable.

```
void green_cond_signal(green_cond_t *cond){
  if (cond->queue->head == NULL) {
    return;
  }
  blockI();
```

```
  green_t *thread = dequeue(cond->queue);
  enqueue(readyqueue, thread);
  unblockI();
}
```

### 2.6 green_mutex_init()

This function initializes a mutex referenced by mutex variable.

```
int green_mutex_init(green_mutex_t *mutex) {
  mutex->taken = FALSE;
  mutex->susp = NULL;
}
```

### 2.7 green_mutex_lock()

This function locks the mutex object referenced by mutex.

```
int green_mutex_lock(green_mutex_t *mutex) {
  blockI();
  green_t *susp = running;
  while(mutex->taken) {
    mutex->susp = susp;
    green_t *next = dequeue(readyqueue);
    running = next;
    swapcontext(susp->context, next->context);
  }
  mutex->taken = TRUE;
  unblockI();
  return 0;
}
```

### 2.8 green_mutex_unlock()

This function releases the mutex object referenced by mutex.

```
int green_mutex_unlock(green_mutex_t *mutex) {
  blockI();
  if (mutex->susp) {
    green_t *thread = mutex->susp;
    enqueue(readyqueue, thread);
  }
  mutex->taken = FALSE;
  unblockI();
  return 0;
}
```

| # | green | | | pthread | | |
|---|---|---|---|---|---|---|
| | 100 | 1000 | 10000 | 100 | 1000 | 10000 |
| 1 | 1282 | 12741 | 109009 | 1572 | 13244 | 121973 |
| 2 | 1039 | 12003 | 114427 | 1755 | 12432 | 123072 |
| 3 | 708 | 14130 | 110567 | 1561 | 12745 | 121044 |
| 4 | 720 | 10860 | 110472 | 1529 | 12582 | 123248 |
| 5 | 1214 | 9839 | 109079 | 1693 | 13729 | 122550 |
| Average | 993 | 11915 | 110711 | 1622 | 12946 | 122377 |

Table 1: Clock ticks taken for each of the runs

# 3 Testing

To see the difference between our green version and the pthread version of threads, some tests of time required to complete a certain workload have been conducted. Tests have been run for 100, 1000, and 10000 iterations of 2 threads 5 times each. CPU clock ticks were used as a measuring unit. The test function for green threads is the following:

```
clock_t start = clock();
green_t g0, g1;
int a0 = 0;
int a1 = 1;
green_cond_init(&condition);
green_mutex_init(&mutex);
green_create(&g0, test, &a0);
green_create(&g1, test, &a1);
green_join(&g0);
green_join(&g1);
printf("done\n");
clock_t end = clock();
long int time_spent = (long int)(end - start);
printf("Took %lds\n", time_spent);
```

It must be noted, that this test function has printf() functions in it for each of the iterations, which slows down the execution time. However, this does not change the results, because the pthread equivalent of this test has also had printf() functions in it for fairness. The results are displayed in the table above.

It can be noticed that green implementation takes less clock ticks. However, to see the results better, let's graph the data:

# 4 Analysis and Conclusion

The graph shows that our implementation requires less ticks and, therefore, we can make a conclusion that our green implementation was faster than pthread. One possible explanation to why this is the case is because our implementation

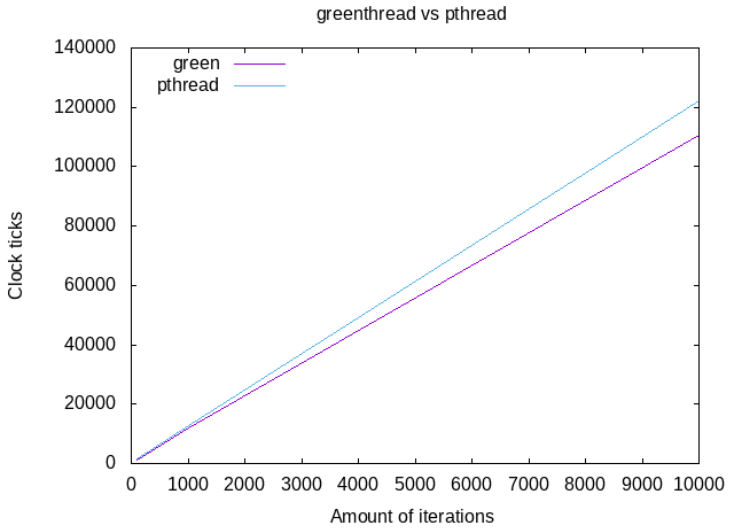


Figure 1: Plotted data

is extremely bare-bones and does not require as much overhead as pthread. However, pthread works in cases, where our implementation does not, which makes 10-15% slower performance more than justifiable.

To sum up, we have created threads, similar to pthread, that performs a bit faster at the expense of reliability. However, multithreading should be flawless and reliable, and because of this, pthread's slightly slower performance means nothing and it is the superior choice.