Seminar 3: Green

Sadok Dalin

2018-12-13

1 Problem

In this seminar a thread library green threads was implemented. The API was similar to the API in the pthread library. Among other things, a scheduler and context handler was implemented. Context switching was handled by the OS. The problem was divided into a list of subproblems:

- Implement a thread representation with structures for storing a context, the function to run, function arguments, a zombie flag and pointers to the next and joining threads.
- Implement a ready queue and associated functions for scheduling threads.
- Implement functions for creating, yielding and joining threads.
- Implement functions for handling condition variables.
- Implement a timer interrupt and handler for scheduling threads with interrupts instead of yield.
- Implement mutexes for thread synchronization.
- Change the implementation of condition variables to use mutexes.
- Implement tests for each of the above implementations.
- Implement a benchmark comparing the execution time of pthread and green threads.

2 Solution

Green thread and ready queue implementation
Green threads were implemented as structs. Each thread contained a context, which contained a stack and instruction pointer. The function and arguments for the function to be called from the thread were stored in void pointers. Two pointers to green threads were used to link the thread in to the ready queue and to store threads waiting to be joined. An int was used to indicate whether the thread was dead(had finished running its function).

The ready queue was implemented as a single linked list. A global variable pointed to the first thread in the queue. Subsequent threads were pointed to by the next pointer in each individual thread. The last thread in the queue had the next pointer set to NULL. The join queue was implemented in a similar fashion to allow for multiple threads to join another.

green_create, green_thread, green_yield and green_join
The green_create function was responsible for creating a thread, handling the initial call from the user. In it the memory for the context was allocated, all variables were initialized , and the newly created thread was added to the ready queue. The green_thread function was called when the thread was created, and was responsible for running the function and the following cleanup. After the function had finished running the stack and context was freed. The next thread in the ready queue was then set to run.

The yield function added the running thread to the end of ready queue and swapped context to the first one in the queue. green_join added the calling thread to the join queue of the thread in the single argument. If the queue contained more than one element, the calling thread was pointed to by the next pointer in the thread at the end of the queue.

Implementation of condition variables
The implementation of condition variables was similar to the implementation of the join queue. A green_cond_t struct contained only a single pointer to a green_t thread. Two functions, green_cond_wait and green_cond_signal, were used to control the variables. in green_cond_wait the running thread was added to the waiting queue in the condition variable. The next thread in the ready queue was then scheduled to run.

In green_cond_signal all suspended threads were added to the ready queue. Since the queue was linked using the next pointer in each thread, only the first element in the queue needed to be added to the ready queue.

Implementation of timer interrupts

To avoid the forced usage of yields to switch threads, timer interrupts were implemented. An interval of 100 microseconds was set. The timer handler handled an interrupt by performing the yield operation, scheduling the next thread in the ready queue.

Implementation of mutex locks
Mutexes were implemented to enable thread synchronization. A mutex contained a taken flag and a pointer to the beginning of a waiting queue. Two functions green_mutex_lock and green_mutex_unlock were used take and return the mutex. If the mutex was taken and the currently running thread called green_mutex_lock, it would be added to the back of the mutex waiting queue. The next thread in the ready queue was then scheduled to run. If the lock was not taken a spin lock was used to try and grab the mutex:

```
int try(int *lock) {
    return __sync_val_compare_and_swap(lock, FALSE, TRUE);
}
lock(..) {
    while (try(&mutex->taken) != 0) {}
}
```

The green_mutex_unlock function added the threads waiting for the mutex to the ready queue and released the mutex.

Implementation of atomic operations on conditional variables
To avoid a deadlock situation when using conditional variables in combination with timer interrupts a mutex was added into the implementation. Without the mutex, a waiting thread could be interrupted before it had time to add itself to the waiting queue, thereby letting another thread call green_cond_signal without any effect.

The mutex was acquired before going into the green_cond_wait function and was expected to be held by the thread that initially acquired it. This was done by releasing the mutex just before the context was swapped(to avoid potential deadlocks), and acquiring it after the thread was swapped back.

3 Tests and Benchmarks

Multiple tests were implemented to verify that each incremental addition to the API of the library was working correctly. A similar structure was used for all tests where two threads were incrementing local counters and flipping a flag, requiring a context(thread) switch to keep incrementing the counters. The yield, condition variable and timer interrupt tests were quite simple,

with an if statement to determine which thread should increase its counter. Mutexes were tested by having a shared counter that was incremented by each thread by a set amount. The final value was then checked against (#threads)*(#increments).

The final test consisted of a consumer and producer functions that used mutexes in combination with atomic operation condition variables:

```
void *produce(void *arg)
{
    int val = *(int *)arg;
    green_mutex_lock(&mutex);
    if (count == MAX)
    {
        green_cond_wait(&cond, &mutex);
    }
    put(val);
    green_cond_signal(&cond);
    green_mutex_unlock(&mutex);
}
```

A benchmark was made to compare the execution speed(probably the context switching speed) of green- and pthreads. This was done by creating 100 threads of each type and adding them to an array, followed by a call to join. All of the threads exeucuted a function where a global counter was incremented by a 1000 by each thread. Mutexes were used for synchronization with timer interrupts enabled.

It took $\tilde{1}$47ms for 100 green threads to increment a counter by 1000 and $\tilde{1}$5ms for the 100 pthreads to increment a counter by 1000. This meant that the green thread implementation was on average slower than the pthread library by a factor of 10, a significant difference. One possible reason for the decreased performance of the green thread library was that the pthread library used Linux kernel threads that could be spread out across several CPU cores.

4 Conclusion

Multiple problems and bugs were encountered and solved during the implementation of the green thread library:

- Numerous segmentation violations because of out of bound arrays, going outside of the ready/waiting queue, and trying to access unallocated memory.

- Infinite loops when trying to add items to the ready queue. In some cases, one or more threads in the waiting queue pointed to themselves.
- Premature termination before all threads had increased a counter to a certain value. This was especially prevalent when trying to implement mutexes. This was due to an error in how threads were retrived from the ready queue.
- "double free or corruption" errors. This was probably due to threads being deleted multiple times.