1 Introduction

This is an assignment where you will implement your own thread library. Instead of using the operating systems threads you will create your own scheduler and context handler. Before even starting to read this you should be up and running using regular threads, spin locks, conditional variables, monitors etc. You should also preferably have done a smaller exercise that shows you how we can work with contexts.

Note - the things you will do in this assignment are nothing that you would do in real life. When ever you want to implement a multi-threaded program you would use the pthread library. We will however manage contexts explicitly to implement something that behaves similar to the pthread library. Why? - To better understand how threads work on the inside.

We will call our implementation green since they will be implemented in user space i.e. the operating system will not be involved in the scheduling of threads.

2 Managing contexts

The library functions that we will use are: getcontext(), makecontext(), setcontext() and swapcontext(). Look up the man pages for these functions so that you get a basic understanding of what they do.

The functions that we want to implement are the following (arguments will be described later):

- **green_create()**: initializes a green thread
- **green_yield()**: suspends the current thread and selects a new thread for execution
- **green_join()**: the current thread is suspended waiting for a thread to terminate

Our handler needs to keep track of the currently running thread and a set of suspended threads ready for execution. It must of also keep track of threads that have terminated since the creator of the thread might want to call green_join() and should then continue directly.

We will try to mimic the pthread library so look up the definition of the equivalent functions: pthread_create(), pthread_yield() and pthread_join()
(note - green_join() will take a pointer to a struct where pthread_join() takes a struct).

representing a thread

To follow the structure of the pthread library we will represent threads by a structure. This structure will hold all information that is needed for us to manage the threads; the threads manager will keep a very small state internally.

The structure should of course hold a pointer to the context of the thread. The context is used when we call swapcontext() or setcontext(). When we call the thread for the first time, we will call a function given an argument. When a thread is suspended we will add it to a linked list so we include a next pointer in the structure itself. We also need to keep track of a thread that is waiting to join the thread to terminate so we also keep a pointer for this. The zombie status field will indicate if the thread has terminated or not.

In a file green.h include the following:

```c
#include <ucontext.h>

typedef struct green_t {
    ucontext_t *context;
    void *(*fun)(void *);
    void *arg;
    struct green_t *next;
    struct green_t *join;
    int zombie;
} green_t;

int green_create(green_t *thread, void *(*fun)(void *), void *arg);
int green_yield();
int green_join(green_t *thread);
```

internal state

The internal state of the scheduler is very small. We will only keep track of two things: the running thread and the ready queue. We will also need one global context that will be used to store the main context i.e. the context of the initial running process.

In a file green.c include the following:

```c
#include <stdlib.h>
#include <ucontext.h>
#include <assert.h>
```
#include "green.h"

#define FALSE 0
#define TRUE 1

#define STACK_SIZE 4096

static ucontext_t main_cntx = {0};
static green_t main_green = {&main_cntx, NULL, NULL, NULL, NULL, FALSE};

static green_t *running = &main_green;

We allocate a global green thread main_green that holds a pointer to the main context and otherwise initialized with null pointers and a false zombie flag.

If you read the man pages for makecontext() and swapcontext() you realize that we will have to initiate a contexts before we can use it and this cannot be done at compile time. We could provide a function that the user needs to call to initialize the manager, or a in the first call to green_create() detect that the main context needs to be initialized; we will however use another trick - provide a function that is called when the program is loaded.

In the file green.c add the following definition.

```c
static void init() __attribute__((constructor));

void init() {
    getcontext(&main_cntx);
}
```

The init() function will initialize the main_cntx so when we call the scheduling function for the first time the running thread will be properly initialized.

## 2.1 create a green thread

A new green thread is created in a two stage process. First the user will call green_create() and provide: an uninitialized green_t structure, the function the thread should execute and pointer to its arguments. We will create new context, attach it to the thread structure and add this thread to the ready queue.

When this thread is scheduled it should call the function but in order to do this we set it to call the function green_thread(). This function is responsible for calling the function provided by the user.

```c
int green_create(green_t *new, void *(*)(void *), void *arg) {
```
ucontext_t *cntx = (ucontext_t *)malloc(sizeof(ucontext_t));
getcontext(cntx);

void *stack = malloc(STACK_SIZE);
cntx->uc_stack.ss_sp = stack;
cntx->uc_stack.ss_size = STACK_SIZE;

makecontext(cntx, green_thread, 0);
new->context = cntx;
new->fun = fun;
new->arg = arg;
new->next = NULL;
new->join = NULL;
new->zombie = FALSE;

// add new to the ready queue :

return 0;
}

It is up to you to implement how the ready queue is managed.

Now let's take a look at the green_thread() function. This function will
do two things: start the execution of the real function and, when after re-
turning from the call, terminate the thread.

void green_thread() {
  green_t *this = running;

  (*this->fun)(this->arg);

  // place waiting thread in ready queue :
  // free allocated memory structures :
  // we're a zombie :
  // find the next thread to run :
  running = next;
  setcontext(next->context);
}

The tricky part is what to do when the called function returns. We (that
is you) should check if there is a thread waiting for its termination, and if
so place it in the ready queue. The stack that was allocated, and the space for the context, should be returned to the memory management system; the thread is now a zombie process. There should be a thread in the ready queue so we select the first and schedule it for execution.

2.2 yield the execution

In the initial implementation, scheduling is only done when a thread voluntarily call the \texttt{green\_yield()} function. This function will simply put the running thread last in the ready queue and then select the first thread from the queue as the next thread to run.

\begin{verbatim}
int green_yield() {
    green_t * susp = running;
    // add susp to ready queue :
    // select the next thread for execution :
    running = next;
    swapcontext(susp->context, next->context);
    return 0;
}
\end{verbatim}

The call to \texttt{swapcontext()} will do the context switch for us. It will save the current state in \texttt{susp->context} and continue execution from \texttt{next->context}. Note that when the suspended thread is scheduled, it will continue the execution from exactly this point (read that sentence again, it will be important).

2.3 the join operation

The join operation will wait for a thread to terminate. We therefore add the thread to the \texttt{join} field and select another thread for execution. If the thread has already terminated we can of course continue as if nothing happened.

\begin{verbatim}
int green_join(green_t *thread) {
    if(thread->zombie)
        return 0;

    green_t * susp = running;
    // add to waiting threads :
    // select the next thread for execution :
    running = next;
    swapcontext(susp->context, next->context);
}
\end{verbatim}
We could allow several threads to wait for the same thread. If you read the man pages for `pthread_join()` you will see that they say that the behavior is undefined.

## 2.4 a small test

If you have completed the code above and implemented a ready queue, you should be able to run a small test.

```c
#include <stdio.h>
#include "green.h"

void *test (void *arg) {
    int i (int*) arg;
    int loop = 4;
    while (loop > 0) {
        printf("thread %d: %d\n", i, loop);
        loop--;
        green_yield();
    }
}

int main () {
    green_t g0, g1;
    int a0 = 0;
    int a1 = 1;
    green_create(&g0, test, &a0);
    green_create(&g1, test, &a1);

    green_join(&g0);
    green_join(&g1);
    printf("done\n");
    return 0;
}
```

### 3 Suspending on a condition

Now for the next task: the implementation of conditional variables. These should work as the conditional variables in the pthread library. However, we do not have any mutex structures that can be locked so our implementation is simpler.
You should stop and wonder why we have not implemented any locking functionality. Is it not very dangerous to run multi-threaded programs without locks?

You should implement the following functionality:

- `void green_cond_init(green_cond_t*): initialize a green condition variable`
- `void green_cond_wait(green_cond_t*): suspend the current thread on the condition`
- `void green_cond_signal(green_cond_t*): move the first suspended thread to the ready queue`

You need to define a data structure `green_cond_t` that can hold a number of suspended threads. The implementation of the functions should then be quite simple. Draw some pictures that describes what the operations should do before you start implementing.

When you think you have it you could try something like this (don’t forget to initialize the conditional variable):

```c
int flag = 0;
green_cond_t cond;

void* test(void* arg) {
    int id = *(int*)arg;
    int loop = 4;
    while (loop > 0) {
        if (flag == id) {
            printf("thread %d: %d\n", id, loop);
            loop--;
            flag = (id + 1) % 2;
            green_cond_signal(&cond);
        } else {
            green_cond_wait(&cond);
        }
    }
}
```

4 Adding a timer interrupt

So far we have relied on the threads themselves to either yield the execution or suspend on a conditional variable, before we schedule a new thread for execution. This is fine, and for sure makes things easier, but we might want
to allow several threads to execute concurrently. We therefore introduce a timer-driven scheduling event.

A timer will be set to send the process a signal with regular intervals. When we receive a signal we will suspend the currently running thread and schedule the next one in the run queue. This is exactly what the `green_yield()` function does.

In the beginning of `green.c`:

```c
#define PERIOD 100

static sigset_t block;

void timer_handler(int);
```

Now in the `init()` function we initialize the timer. We initialize the `block` to hold the mask of the `SIGVTALRM`, set the handler to our `timer_handler()` function and associate the signal to the action handler. We then set the timer interval and delay (value) to our `PERIOD` and start the timer.

```c
sigemptyset(&block);
sigaddset(&block, SIGVTALRM);

act.sa_handler = timer_handler;
assert(sigaction(SIGVTALRM, &act, NULL) == 0);

struct sigaction act = {0};
struct timeval interval;
struct itimerval period;

interval.tv_sec = 0;
interval.tv_usec = PERIOD;
period.it_interval = interval;
period.it_value = interval;
setitimer(ITIMER_VIRTUAL, &period, NULL);
```

When the timer expires the handler will be called and its time to schedule the next thread.

```c
void timer_handler(int sig) {
    green_t *susp = running;

    // add the running to the ready queue

    // find the next thread for execution
    running = next;
    swapcontext(susp->context, next->context);
}
```
If you complete the code above it will actually work ... almost. You could test it for a while before you run into a strange segmentation fault and when you do, you will have a very hard time finding the bug.

The thing that will eventually happen is that we will have a timer interrupt when we’re in one of the functions that manipulate the state of the green threads. Imaging what could happen if we are in the middle of a yield operation and change the run queue. We need to prevent these interrupts when we change the state.

Fortunately for us, we are not the only ones with this problem so we have a simple way to block and unblock these interrupts:

```c
    sigprocmask(SIG_BLOCK, &block, NULL);
```

Now for the tricky part - if you for example want to block interrupts in the `green_yield()` function, it could end like this:

```c
    swapcontext(susp->context, next->context);
    sigprocmask(SIG_UNBLOCK, &block, NULL);
}
```

What will happen? When we enter the yield function we block the timer interrupts; then we call the `swapcontext()` function. This means that we leave the current execution and continue from wherever the next thread was suspended. This means that it will continue in a state where the interrupt signal is blocked.

Changing the order does not help us since we would then have small window where we allow interrupts but have not yet scheduled the next thread.

```c
    sigprocmask(SIG_UNBLOCK, &block, NULL);
    swapcontext(susp->context, next->context);
}
```

Before you get this to work you need to think it through more than once - in what positions could a green thread start executing? If they start with the signals blocked, could it unblock them?

## 5 A mutex lock

When you have your timer interrupts working, write a small test program that shows that it works. Also write a program that shows a situation where our threads library falls short.
Since a thread now can be interrupted at any point in the execution we will have a problem when we update shared data structures. A simple example where two threads read and increment a shared counter will lead to very unpredictable behavior. We need a way to synchronize our threads and a mutex construct would do the trick.

We will need a structure to represent a mutex and since we will have threads suspended on the lock we let it hold a list of suspended threads. In the \texttt{green.h} file we add the following:

\begin{verbatim}
typedef struct green_mutex_t {
volatile int taken;
struct green_t *susp;
} green_mutex_t;

int green_mutex_init( green_mutex_t *mutex );
int green_mutex_lock( green_mutex_t *mutex );
int green_mutex_unlock( green_mutex_t *mutex );
\end{verbatim}

The function \texttt{green_mutex_init()} is trivial since all we have to do is initialize the fields:

\begin{verbatim}
int green_mutex_init( green_mutex_t *mutex ) {
mutex->taken = FALSE;
mutex->susp = NULL;
}
\end{verbatim}

The function that tries to take the lock will, if the lock is taken, look very similar to the yield procedure:

\begin{verbatim}
int green_mutex_lock( green_mutex_t *mutex ) {
    // block timer interrupt

green_t *susp = running;
while(mutex->taken) {
    // suspend the running thread
    // find the next thread
    running = next;
    swapcontext(susp->context, next->context);
}
    // take the lock
    // unblock
return 0;
}
\end{verbatim}

The unlock function is very similar to the signal operation:
Complete the code and write a small program that shows that it works.

6 The final touch

We now only have one thing left, an atomic operation that will suspend on a conditional variable and release a lock. If we do not have this operation we will not be able to coordinate operations properly. Describe what could happen in the following program that wants to modify a flag that is synchronized using a conditional variable.

```c
int green_mutex_unlock(green_mutex_t *mutex) {
    // block timer interrupt
    : // move suspended threads to ready queue
    : // release lock
    : // unblock
    return 0;
}
```

We need, as in the pthread library, a function that suspends on a conditional variable and releases a lock in one atomic operation. When the function returns, the lock should be held. We therefore change the function `green_cond_wait()` to also take a mutex as an argument.

```c
int green_cond_wait(green_cond_t *cond, green_mutex_t *mutex) {
    // block timer interrupt
    : // suspend the running thread on condition
    : :
    return 0;
}
```
if (mutex != NULL) {
    // release the lock if we have a mutex
    :
    // schedule suspended threads
    :
}
// schedule the next thread :
running = next;
swapcontext(susp->context, next->context);

if (mutex != NULL) {
    // try to take the lock
    while (mutex->taken) {
        // bad luck, suspend :
    }
    // take the lock
    mutex->taken = TRUE;
} // unblock :
return 0;
}

Rewrite your test program and use the atomic conditional wait function. You should now be able to have a producer and consumer synchronize their actions using conditional variables and mutex locks.

7 Summary

A threads library is not rocket science but it takes some time before you get it right. The problem is that when it fails it is very hard to figure out what went wrong. Does your implementation work correctly? A single threaded program is easier to debug since it follows the same execution path every time you run it. In our implementation things were predictable up until we introduced the timer interrupts. How do we know that we have covered all corner situations? Is extensive testing the only tool we have? If your threads library was controlling the threads of a web browser it might not matter very much, what if it was used in an air plane control system - would you sleep well tonight?