Project: User-Level Threading

Robbert van Renesse

February 1, 2019

1 Introduction

We would like you to implement user-level threading for EGOS. (You'll find that the same software will run without much trouble on any operating system.) The threading package should be non-preemptive: only one thread can run at a time, and threads voluntarily release control to other threads. In particular, you'll need to implement the following API:

```
void thread_init();
void thread_create(void (*f)(void *arg), void *arg,
            unsigned int stack_size);
void thread_yield();
void thread_exit();
```

For an example of using these calls, refer to Figure 1.

- `thread_init()` initializes the threading package. As a side effect, there will be a single thread running. If at any point in time there are no runnable threads, the process should exit with status 1, that is, it should invoke `sys_exit(1)`.
- `thread_create(void (*f)(void *arg), void *arg, unsigned int stack_size)` creates a new thread. The thread executes function `f` with a single argument `arg`. If function f returns, the thread should be automatically cleaned up as if it invoked `thread_exit()`. `thread_create()` may implicitly yield the CPU to the new thread.

  For example, in Figure 2, `thread_create(consumer, "consumer", 16 * 1024)` creates a new thread with a stack of 16 Kbytes. This thread invokes `consumer("consumer 1")`. 16 KBytes tends to be a large enough stack for small threads.
- `thread_yield()` means that the current thread voluntarily wants to yield the CPU to another runnable thread. If there is no other runnable thread, `thread_yield` is a no-op and returns immediately.
- `thread_exit()` means that the thread is done executing altogether. `thread_exit()` should never "return"; instead, the thread should be cleaned up and another runnable thread, if any, should resume executing.

```
static void test_code(void *arg){
    int i;

    for (i = 0; i < 10; i++) {
        printf("%s here: %d\n", arg, i);
        thread_yield();
    }
    printf("%s done\n", arg);
}

int main(int argc, char **argv){
    thread_init();
    thread_create(test_code, "thread 1", 16 * 1024);
    thread_create(test_code, "thread 2", 16 * 1024);
    test_code("main thread");
    return 0;
}
```

Figure 1: Simple test for threads.

After you're done with this, we would like you to implement a semaphore data structure `struct sema` with the following interface:

```
struct sema;
void sema_init(struct sema *sema, unsigned int count);
void sema_dec(struct sema *sema);
void sema_inc(struct sema *sema);
```

- `void sema_init(struct sema *sema, unsigned int count)` initializes a semaphore with the given `count`. If you use a semaphore as a lock, `count` will typically be 1.
- `void sema_dec(struct sema *sema)` tries to decrement the semaphore. However, the semaphore should never go below 0, and so if it is 0, the invoking thread should block until some other thread invoked `sema_inc()` on the same semaphore.
- `void sema_inc(struct sema *sema)` increments the semaphore. However, if some thread is blocked on the semaphore because it is 0, that thread should be unblocked and the semaphore count should remain 0.

Of these calls, only `sema_dec()` may release the CPU as if the thread invoked `thread_yield()`.

Figure 2 implements the producer/consumer problem with a shared buffer of `NSLOTS` slots. In the example, the main thread becomes the producer and a consumer thread is explicitly spawned. It is easy to spawn additional consumers and producers. Each producer appends its name (in `arg`) to the shared buffer five times. Each consumer, in an infinite loop, takes names of the shared buffer and prints those along with its own name. The output of this particular program should be as follows:

```
#define NSLOTS    3

static struct sema s_empty, s_full, s_lock;
static unsigned int in, out;
static char *slots[NSLOTS];

static void producer(void *arg){
    for (;;) {
        // first make sure there's an empty slot.
        sema_dec(&s_empty);

        // now add an entry to the queue
        sema_dec(&s_lock);
        slots[in++] = arg;
        if (in == NSLOTS) in = 0;
        sema_inc(&s_lock);

        // finally, signal consumers
        sema_inc(&s_full);
    }
}

static void consumer(void *arg){
    unsigned int i;

    for (i = 0; i < 5; i++) {
        // first make sure there's something in the buffer
        sema_dec(&s_full);

        // now grab an entry to the queue
        sema_dec(&s_lock);
        void *x = slots[out++];
        printf("%s: got '%s'\n", arg, x);
        if (out == NSLOTS) out = 0;
        sema_inc(&s_lock);

        // finally, signal producers
        sema_inc(&s_empty);
    }
}

int main(int argc, char **argv){
    thread_init();
    sema_init(&s_lock, 1);
    sema_init(&s_full, 0);
    sema_init(&s_empty, NSLOTS);

    thread_create(consumer, "consumer 1", 16 * 1024);
    producer("producer 1");
    return 0;
}
```

Figure 2: An implementation of Producer/Consumer

```
consumer 1: got 'producer 1'
consumer 1: got 'producer 1'
consumer 1: got 'producer 1'
consumer 1: got 'producer 1'
consumer 1: got 'producer 1'
```

after which the program should exit with status 1 because there are no more runnable threads.

You must also develop a test suite for your threading package. Such a test suite could include some well-known synchronization implementations such as producer/consumer (with multiple producers and consumers), barber shop, dining philosophers, and multi-reader locks.

2 How To Go About This

You can make use of the very same context switch code used by the EGOS kernel to implement processes. The interface is as follows:

```
#include "shared/context.h"

void ctx_switch(address_t *old_sp, address_t new_sp);
void ctx_start(address_t *old_sp, address_t new_sp);
```

Here `address_t` is an unsigned integer type that is large enough to hold a pointer (in this case a stack pointer). `ctx_switch()` pushes the registers (of the current thread) on its stack and then saves the stack pointer in `*old_sp`. It then sets the stack pointer to `new_sp` and pops the registers previously pushed. `ctx_start()` is similar, but instead of restoring registers it invokes `ctx_entry()`, which you can declare as:

```
void ctx_entry(){
    // your code goes here
}
```

You can find the code for `ctx_switch` and `ctx_start` in the assembly files in "shared/asm_*.s", and examples of their use in the Grass kernel in "grass/process.c" among others. You will use `ctx_start` to create a brand new thread with an empty stack, but use `ctx_switch` to switch between existing threads.

You will want to create a thread control block structure `struct thread` for each thread (allocated with `malloc()`), as well as a stack (also allocated with `malloc()`). The thread control block should keep track of the running state of the thread (runnable, blocked on a semaphore, etc.), the stack (which will need to be freed when the threads exits, and the stack pointer).

You will also need queues: one to maintain the list of runnable threads, and one for each semaphore to maintain the threads that are blocked on that semaphore. Fortunately, the GEOS application library contains a simple threading package with the following interface:

```
#include "shared/queue.h"

struct queue;
void queue_init(struct queue *q);
void queue_add(struct queue *q, void *item);
void *queue_get(struct queue *q);
bool_t queue_empty(struct queue *q);
```

- `void queue_init(struct queue *q)` initiazes a queue;
- `void queue_add(struct queue *q, void *item)` adds pointer `item` to the queue;
- `void *queue_get(struct queue *q)` removes an item from the queue and returns it. It returns `NULL` (`(void *) 0`) in case the queue is empty;
- `bool_t queue_empty(struct queue *q)` checks to see if there are any items on the queue without removing any.

For example:

```
#include "shared/queue.h"

struct queue q;

queue_init(&q);
queue_add(&q, "hello world");
assert(!queue_empty(&q));
char *item = queue_get(&q);
assert(strcmp(item, "hello world") == 0);
assert(queue_empty(&q));
```

3 Tips and Tricks

- You will want to keep track of what the current thread is.
- `thread_create()` should allocate a stack and a thread control block. It can then start the new thread immediately using `ctx_start()`. This will invoke `ctx_entry()`, from which you will have to invoke the thread's main function f. However, before you do so, you should put the invoking thread on the run queue (aka ready queue).
- When a thead exits, it cannot clean itself up, because that would destroy its stack and it would no longer be able to switch to the next runnable thread. So it should switch to the next runnable thread and let that thread clean up the exiting thread.

```
#include "egos.h"
#include "../shared/queue.h"
#include "../shared/context.h"

/**** THREADS AND SEMAPHORES ****/

struct thread {
    // your code here
};
struct sema {
    // your code here
};

// declare global variables here

// your thread and semaphore code here

/**** TEST SUITE ****/

// your test code here

int main(int argc, char **argv){
    thread_init();
    // your code here
    return 0;
}
```

Figure 3: Format of Program

- The threading package obtains "control" only when the current thread invokes one of the API functions. There is no "main loop", so all thread scheduling decisions have to be made when the threading package obtains control. Note, however, that `sema_init()` and `sema_inc()` should not change the currently running thread.
- You can develop and test threads first and semaphores later. You'll first want to try and make a test program like Figure 1 work.
- Typically, you don't want the currently running thread to also be on the run queue. The run queue consists of those threads that are runnable (not blocked on a semaphore) but not currently running.

4 Running your code

Create a single C source file in the format of Figure 3. You can add this as an application to EGOS as follows:

- Place your program in the apps directory under, say, `apps/myprogram.c`.
- Edit "Makefile.apps" and add `myprogram.exe` to the `APPS` variable.

- Edit file “grass/main.c”, find where it says “Load the bin directory”, and add a line for “myprogram.exe”.
- Run “make run2”. This will compile your program, erase the existing file system and create a new file system with “myprogram.exe” added to the EGOS “/bin” directory where most executables live.
- Log in to EGOS, and run “myprogram”.

5 Submitting Your Solution

Simply submit your C file as described above. Please follow the format exactly as specified in Figure 3. Your submission should have no more than 200 lines of code in the THREADS AND SEMAPHORES section, no more than 300 lines of code in the TEST SUITE section, no more than 10,000 characters total (excluding comment lines). Exceptions, particularly for the TEST SUITE, can be made with the approval of a TA. You will be graded both on the correctness of the threading and semaphore code, the helpfulness of the comments in the code, as well as the thoroughness of the testing suite.