RThreads

RVR and FBS

October 29, 2019

1 `rthread` library with condition variables.

The programming homework for CS4410 includes a set of synchronization problems to be implemented in C. To solve these problems you are to use the `rthread` library—a multithreading library that is similar to the POSIX `pthread` library but that, for educational purposes, has been simplified and optimized for easier grading.

Figure 1 lists the `rthread` API, which supports locks and condition variables using Mesa semantics. `rthread_create()` takes three arguments: a pointer to a function that takes two `void *` arguments, and two `void *` arguments. This creates (but not yet runs) a thread that will execute the given function as its main body, with the two specified arguments. By convention, the first argument points to data that is shared among threads, while the second argument points to arguments passed to the specific thread. `rthread_run()` will run any threads that have been created and terminate only when each thread has terminated. (A running thread can create additional threads, but those will run immediately and `thread_run` will continue waiting until those threads have terminated as well.)

Figure 3 shows a bounded buffer implementation using a lock and condition variables in a *monitor* style (without the language support). In it, the `bb_produce()` and `bb_consume()` functions simulate monitor method behavior by grabbing a global lock on entry and releasing it on exit. The `rthread` library provides condition variables with Mesa semantics.

Coming back to our example, what is missing is a lock around the shared variable. Figure 2 illustrates locks with rthreads. The shared data has now been encapsulated in a C struct and associated with a lock. The lock is initialized using `rthread_lock_init()`. Note the `rthread_with` construction in the `incrementer()` function: `rthread_with(lock) S` executes statement `S` (which may be a block of statements) while holding the specified `lock`.

```
typedef rthread_lock_t, rthread_cv_t;

void rthread_create(
    void (*f)(void *shared, void *arg),
    void *shared,
    void *arg);
void rthread_lock_init(rthread_lock_t *lock);
void rthread_cv_init(rthread_cv_t *cv, rthread_lock_t *lock);
void rthread_cv_wait(rthread_cv_t *cv);
void rthread_cv_notify(rthread_cv_t *cv);
void rthread_cv_notifyAll(rthread_cv_t *cv);
void rthread_delay(unsigned int milliseconds);
void rthread_run(void);
```

Figure 1: `rthread` API

2 Programming Problems

2.1 The Smalltown USA Swimming Pool

The Smalltown USA School System has a swimming pool; the pool has 7 swim lanes. During a practice, each lane may be occupied by at most one swimmer.

The pool must be shared by the middle-school swim team and the high-school swim team. The number of students on each team is unspecified and could be larger than 7.

As a matter of policy and safety:

- at most 7 students may be using the pool at any time;
- at any time, either the middle-school swim team or the high-school swim team may be using the pool, BUT NOT members from both teams.

Suppose there is a thread for each student who is a member of the middle-school team or the high-school team.

a) Implement the monitor in Figure 5. Invoking `pool_enter(pool, level)` is required to enter the pool. The objective of `pool_enter` is to keep the pool as full as possible, and therefore the `pool_enter` function should block its invoker only if the pool already contains 7 swimmers or if the pool contains members of the other team.

   A thread invokes `pool_exit(pool, level)` when the associated team member is done swimming and leaves the pool.

   To receive full credit, a solution must satisfy: *A middle-school swimmer may not wake-up a high-school swimmer unless that middle-school swimmer is exiting the pool and the pool will be empty. And similarly for high-school swimmers waking up middle-school swimmers.*

```
#include <stdio.h>
#include <stdlib.h>
#include "rthread.h"

struct shared_data {
    int counter;
    rthread_lock_t lock;
};

void incrementer(void *shared, void *arg) {
    struct shared_data *sd = shared;
    int *delta = arg;

    for (int i = 0; i < 1000000; i++)
        rthread_with(&sd->lock)
            sd->counter += *delta;
}

int main() {
    struct shared_data sd;
    int up = 1, down = -1;

    sd.counter = 0;
    rthread_lock_init(&sd.lock);
    rthread_create(incrementer, &sd, &up);
    rthread_create(incrementer, &sd, &down);
    rthread_run();
    printf("counter = %d\n", sd.counter);
    return 0;
}
```

Figure 2: Using a lock

```
struct bounded_buffer {
    int queue[BB_QSIZE];    // the item storage
    int in;                 // where to insert a new item
    int out;                // where to retrieve an item
    int cnt;                // the number of filled slots
    rthread_lock_t lock;    // a "monitor" lock
    rthread_cv_t prod_cv;   // cond. var. to wait for room
    rthread_cv_t cons_cv;   // cond. var. to wait for items
};

void bb_init(struct bounded_buffer *bb) {
    rthread_lock_init(&bb->lock);
    rthread_cv_init(&bb->prod_cv, &bb->lock);
    rthread_cv_init(&bb->cons_cv, &bb->lock);
    bb->in = bb->out = bb->cnt = 0;
}

void bb_produce(struct bounded_buffer *bb, int item) {
    rthread_with(&bb->lock) {
        while (bb->cnt == BB_QSIZE)
            rthread_cv_wait(&bb->prod_cv);
        bb->queue[bb->in] = item;
        bb->in = (bb->in + 1) % BB_QSIZE;
        bb->cnt++;
        rthread_cv_notify(&bb->cons_cv);
    }
}

int bb_consume(struct bounded_buffer *bb) {
    int item;

    rthread_with(&bb->lock) {
        while (bb->cnt == 0)
            rthread_cv_wait(&bb->cons_cv);
        item = bb->queue[bb->out];
        bb->out = (bb->out + 1) % BB_QSIZE;
        bb->cnt--;
        rthread_cv_notify(&bb->prod_cv);
    }
    return item;
}
```

Figure 3: Bounded buffer using a lock and condition variables

```c
#include <stdio.h>
#include <stdlib.h>
#include "rthread.h"

#define BB_QSIZE      10      // size of bounded buffer

// Bounded buffer code goes here
struct bounded_buffer;
void bb_init(struct bounded_buffer *bb);
void bb_produce(struct bounded_buffer *bb, int item);
int bb_consume(struct bounded_buffer *bb);

#define NPRODUCERS    4
#define NCONSUMERS    2
#define SCALE         500000

void producer(void *shared, void *arg) {
    for (int i = 0; i < SCALE * NCONSUMERS; i++)
        bb_produce(shared, i);
}

void consumer(void *shared, void *arg) {
    for (int i = 0; i < SCALE * NPRODUCERS; i++)
        (void) bb_consume(shared);
}

int main() {
    struct bounded_buffer bb;

    bb_init(&bb);
    for (int i = 0; i < NPRODUCERS; i++)
        rthread_create(producer, &bb, 0);
    for (int i = 0; i < NCONSUMERS; i++)
        rthread_create(consumer, &bb, 0);
    rthread_run();

    return 0;
}
```

Figure 4: Producer-Consumer test code

```
#define MIDDLE     0
#define HIGH       1
#define NLANES     7

struct pool {
    rthread_lock_t lock;
    // you can add more monitor variables here
};

void pool_init(struct pool *pool){
    memset(pool, 0, sizeof(*pool));
    rthread_lock_init(&pool->lock);
    // initialize your monitor variables here
}

void pool_enter(struct pool *pool, int level){
    rthread_with(&pool->lock) {
        // write the code here to enter the pool
    }
}

void pool_exit(struct pool *pool, int level){
    rthread_with(&pool->lock) {
        // write the code here to exit the pool
    }
}
```

Figure 5: Smalltown USA Swimming Pool Interface.

```
#include <stdio.h>
#include <stdlib.h>
#include "rthread.h"

#define NMIDDLE         10
#define NHIGH           10
#define NEXPERIMENTS     5

char *middle[] = {
   "m0", "m1", "m2", "m3", "m4", "m5", "m6", "m7", "m8", "m9"
};
char *high[]   = {
   "h0", "h1", "h2", "h3", "h4", "h5", "h6", "h7", "h8", "h9"
};

void swimmer(void *shared, void *arg){
    struct pool *pool = (struct pool *) shared;
    char *name = (char *) arg;
    for (int i = 0; i < NEXPERIMENTS; i++) {
        rthread_delay(random() % 1000);
        printf("swimmer %s entering pool\n", name);
        pool_enter(pool, *name == 'h');
        printf("swimmer %s entered pool\n", name);
        rthread_delay(random() % 1000);
        printf("swimmer %s leaving pool\n", name);
        pool_exit(pool, *name == 'h');
        printf("swimmer %s left pool\n", name);
    }
}

int main(){
    struct pool pool;
    pool_init(&pool);
    for (int i = 0; i < NMIDDLE; i++) {
        rthread_create(swimmer, &pool, middle[i]);
    }
    for (int i = 0; i < NHIGH; i++) {
        rthread_create(swimmer, &pool, high[i]);
    }
    rthread_run();
    return 0;
}
```

Figure 6: Smalltown USA Swimming Pool Example Program.

See Figure 6 for an example program that creates threads for the various swimmers. Only submit your solution, not the test program. You can submit your solution in a file called "A3a.c".

b) Same as above, but now you should assume

   - the middle school swim team has 7 or fewer members;
   - the high school swim team has 7 or fewer members.

   As before, the members from both teams may not be in the pool at the same time.

   To ensure that each team does get a chance to practice, the following policy is enforced:

   - If a team member A (say) wants to enter the pool that is already occupied by kids from the same team as A, but kids from the other team are waiting to enter the pool, then A must wait until the waiting kids from the other team get their chance to enter the pool.

   You can again use the program in Figure 6, but set `NMIDDLE` and `NHIGH` to at most 7 each.

For both (a) and (b), only submit the monitor code, not the test code. You can submit your solution in a file called "A3b.c".