# RThreads

RVR and FBS

September 24, 2019

## 1 `rthread` library

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 gives a simple example of using rthreads. `main()` creates two threads and then runs them to completion. Both threads execute the `incrementer()` code. The first adds 1 to the global `counter` 1,000,000 times, while the second subtracts 1 from `counter` 1,000,000 times. After both threads complete, `main()` prints the value of counter. What do you expect it will print? Run the program a few times and see what happens. Can you explain?

Figure 2 lists the `rthread` API, which supports locks and semaphores. `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.)

Coming back to our example, what is missing is a lock around the shared variable. Figure 3 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`. Alternatively, the shared variable can be protected using a semaphore, as illustrated in Figure 4.

We will finish this section by giving a solution of the classic bounded buffer synchronization problem using semaphores. Figure 5 shows the test code first.

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

void incrementer(void *shared, void *arg) {
    int *counter = shared;
    int *delta = arg;

    for (int i = 0; i < 1000000; i++)
        *counter += *delta;
}

int main() {
    int counter = 0;          // shared among threads
    int up = 1, down = -1;    // arguments to threads

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

Figure 1: Creating two simple threads

```c
typedef rthread_lock_t, rthread_sema_t;

void rthread_create(
    void (*f)(void *shared, void *arg),
    void *shared,
    void *arg);
void rthread_lock_init(rthread_lock_t *lock);
void rthread_sema_init(rthread_sema_t *sema, unsigned init);
void rthread_sema_procure(rthread_sema_t *sema);
void rthread_sema_vacate(rthread_sema_t *sema);
void rthread_delay(unsigned int milliseconds);
void rthread_run(void);
```

Figure 2: `rthread` API

```
#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 3: Using a lock

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

struct shared_data {
    int counter;
    rthread_sema_t mutex;
};

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

    for (int i = 0; i < 1000000; i++) {
        rthread_sema_procure(&sd->mutex);
        sd->counter += *delta;
        rthread_sema_vacate(&sd->mutex);
    }
}

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

    sd.counter = 0;
    rthread_sema_init(&sd.mutex, 1);
    rthread_create(incrementer, &sd, &up);
    rthread_create(incrementer, &sd, &down);
    rthread_run();
    printf("counter = %d\n", sd.counter);
    return 0;
}
```

Figure 4: Using a semaphore

```
#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 5: Producer-Consumer test code

```
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
    rthread_sema_t in_mutex;   // mutex on ->in
    rthread_sema_t out_mutex;  // mutex on ->out
    rthread_sema_t n_empty;    // counts #empty slots
    rthread_sema_t n_full;     // counts #full slots
};

void bb_init(struct bounded_buffer *bb) {
    bb->in = bb->out = 0;
    rthread_sema_init(&bb->in_mutex, 1);
    rthread_sema_init(&bb->out_mutex, 1);
    rthread_sema_init(&bb->n_empty, BB_QSIZE);
    rthread_sema_init(&bb->n_full, 0);
}

void bb_produce(struct bounded_buffer *bb, int item) {
    rthread_sema_procure(&bb->n_empty);
    rthread_sema_procure(&bb->in_mutex);
    bb->queue[bb->in] = item;
    bb->in = (bb->in + 1) % BB_QSIZE;
    rthread_sema_vacate(&bb->in_mutex);
    rthread_sema_vacate(&bb->n_full);
}

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

    rthread_sema_procure(&bb->n_full);
    rthread_sema_procure(&bb->out_mutex);
    item = bb->queue[bb->out];
    bb->out = (bb->out + 1) % BB_QSIZE;
    rthread_sema_vacate(&bb->out_mutex);
    rthread_sema_vacate(&bb->n_empty);

    return item;
}
```

Figure 6: Bounded buffer using semaphores

Here `main()` creates `NPRODUCERS` producer threads and `NCONSUMERS` consumer threads, sharing an as yet unspecified instance of a `struct bounded_buffer`. Note that the number of items produced by the producer threads is the same as the number of items consumed by the consumer threads, and so all threads should finish eventually.

Figure 6 shows a bounded buffer implementation using semaphores.

## 2 Programming Problems

### 2.1 Experimenting with Threads and Race Conditions

Look over and run the code in Figure 1 and consider the multiple-choice questions below. Then look for A2 Multiple Choice on CMS and record your responses there.

1.1) Run this concurrent program. Which option best describes the possible outputs?

a) The output of the program is always exactly the same

b) Two outputs of the program could never be identical

c) The outputs vary in an unpredictable way

1.3) How many times would you have to run this program in order to observe a specific interleaving?

a) 10 times

b) 100 times

c) 10,000 times

d) There's no guarantee for observing any specific interleaving of threads no matter how many times

1.4) What does this imply about the effectiveness of testing to find synchronization errors?

a) Running the code repeatedly is a good way to rule out bugs.

b) To make sure your code is correct, you must reason about the code instead of rely on observed outputs

1.5) When both threads terminate, what is the largest possible value that could be printed?

a) 1,000,000
b) 2,000,000
c) Could be ANY non-negative number
d) Could be ANY number
e) 0
f) 1,000,001

1.6) What best describes the range of values that may be printed:

a) -1000,000 or 0 or 1,000,000
b) -2000,000 or 0 or 2,000,000
c) Any integer between -1,000,000 and 1,000,000
d) Any integer between -2,000,000 and 2,000,000
e) Just 0
f) Any integer

1.7) Now consider Figure 3. What best describes the range of values that may be printed:

a) -1000,000 or 0 or 1,000,000
b) -2000,000 or 0 or 2,000,000
c) Any integer between -1,000,000 and 1,000,000
d) Any integer between -2,000,000 and 2,000,000
e) Just 0
f) Any integer

```
#define WHISTLER        0
#define LISTENER        1

struct device {
    // your code here
};

void dev_init(struct device *dev);
void dev_enter(struct device *dev, int which);
void dev_exit(struct device *dev, int which);
```

Figure 7: Prototype synchronization interface for the Lab of Ornithology IoT device.

## 2.2 Bird Songs

The Cornell Lab of Ornithology has an IoT device that is meant to be installed in the woods. It has a speaker that can double as a microphone, but not at the same time. The Lab of O would like to run a variety of applications on the IoT device, some of which want to lure birds by playing bird calls, while others want to record bird songs. Because of the restrictions of the device, multiple applications may play sounds (so-called "whistlers"), or multiple applications may record (so-called "listeners"), but not both at the same time. Your job is to write the necessary synchronization code for those applications. In particular, you are to complete the code in Figure 7.

First you have to initialize the device synchronization code with `dev_init()`. Next, an application can "enter" the device using `dev_enter()`. The parameter `which` specifies whether it is a `WHISTLER` or a `LISTENER`. Similarly, using `dev_exit()` the application testifies that it is no longer using the device. Figure 8 shows some test code.

You are to write the synchronization code using semaphores. Please make sure you write your code clearly and use comments where needed for clarification. Insert `assert()` statements liberally to document your correctness arguments and to check runs of your code. (Assert statements can be automatically disabled during production runs of your code.)

Your code does not have to be free of the possibility of starvation: a steady stream of whistlers can make it impossible for listeners to run and vice versa. Of course, if you write code that prevents that from happening, that would not be a bad thing.

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

#define NWHISTLERS        3
#define NLISTENERS        3
#define NEXPERIMENTS      2

char *whistlers[NWHISTLERS] = { "w1", "w2", "w3" };
char *listeners[NLISTENERS] = { "l1", "l2", "l3" };

void worker(void *shared, void *arg){
    struct device *dev = shared;
    char *name = arg;

    for (int i = 0; i < NEXPERIMENTS; i++) {
        printf("worker %s waiting for device\n", name);
        dev_enter(dev, name[0] == 'w');
        printf("worker %s has device\n", name);
        rthread_delay(random() % 3000);
        printf("worker %s releases device\n", name);
        dev_exit(dev, name[0] == 'w');
        rthread_delay(random() % 3000);
    }
    printf("worker %s is done\n", name);
}

int main(){
    struct device dev;

    dev_init(&dev);
    for (int i = 0; i < NWHISTLERS; i++) {
        rthread_create(worker, &dev, whistlers[i]);
    }
    for (int i = 0; i < NLISTENERS; i++) {
        rthread_create(worker, &dev, listeners[i]);
    }
    rthread_run();
    return 0;
}
```

Figure 8: Test code for Lab of Ornithology IoT device.

```
#define NGPUS    10

struct gpu_info {
    ...
};

void gi_init(struct gpu_info *gi);
void gi_alloc(struct gpu_info *gi,
    unsigned int ngpus,
    /* OUT */ unsigned int gpus[]);
void gi_release(struct gpu_info *gi,
    unsigned int ngpus,
    /* IN */ unsigned int gpus[]);
void gi_free(struct gpu_info *gi);
```

Figure 9: Interface for the GPU allocator.

## 2.3 A GPU Allocator

Your computer has 10 GPUs, but a GPU can only be used by one thread at a time. However, sometimes a thread needs more than one GPU. You are to complete the code in Figure 9.

Here `struct gpu_info` holds the state of the GPU allocator, which is initialized with `gi_init()`. Any allocated memory, if any, can be released with `gi_free()` when done. `gi_alloc(ngpus, gpus)` allocates `ngpus` GPUs and returns the identifiers of the allocated GPUs in the provided `gpus` array, which has `ngpus` (or more) entries. If there are not enough gpus available, `gi_alloc` should wait. (Note that if a thread tries to allocate more than `NGPUS` gpus, it will end up waiting for ever.) When the thread is done with the allocated GPUs, it can release them by invoking `gi_release()`.

Figure 10 shows some testing code in which two threads, "Jane" and "Joe," each try to allocate a random number of GPUs for five times. Note that it uses a `print_lock`. Why is that?

Figure 11 shows an implementation of the GPU allocator that is intended for sequential code, but is not thread-safe. Can you see why? You can base your thread-safe allocator on this one if you like.

You may be tempted to do something like shown in Figure 12. However, doing so could lead to *deadlock*. Do you understand why? So you will have to come up with another solution that is not prone to deadlock.

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

#define NGPUS    10

rthread_lock_t print_lock;

// YOUR CODE GOES HERE

void gpu_user(void *shared, void *arg) {
    struct gpu_info *gi = shared;
    unsigned int gpus[NGPUS];

    for (int i = 0; i < 5; i++) {
        rthread_delay(random() % 3000);
        unsigned int n = 1 + (random() % NGPUS);
        rthread_with(&print_lock)
            printf("%s %d wants %u gpus\n", arg, i, n);
        gi_alloc(gi, n, gpus);
        rthread_with(&print_lock) {
            printf("%s %d allocated:", arg, i);
            for (int i = 0; i < n; i++)
                printf(" %d", gpus[i]);
            printf("\n");
        }
        rthread_delay(random() % 3000);
        rthread_with(&print_lock)
            printf("%s %d releases gpus\n", arg, i);
        gi_release(gi, n, gpus);
    }
}

int main() {
    rthread_lock_init(&print_lock);
    struct gpu_info gi;
    gi_init(&gi);
    rthread_create(gpu_user, &gi, "Jane");
    rthread_create(gpu_user, &gi, "Joe");
    rthread_run();
    gi_free(&gi);
    return 0;
}
```

Figure 10: Test code for GPU allocator

```c
#include <string.h>
#include <assert.h>

struct gpu_info {
    int allocated[NGPUS];
    unsigned int nfree;
};

void gi_init(struct gpu_info *gi) {
    memset(gi->allocated, 0, sizeof(gi->allocated));
    gi->nfree = NGPUS;
}

void gi_alloc(struct gpu_info *gi,
        unsigned int ngpus,
        /* OUT */ unsigned int gpus[]) {
    assert(ngpus <= gi->nfree);
    gi->nfree -= ngpus;

    unsigned int g = 0;
    for (unsigned int i = 0; g < ngpus; i++) {
        assert(i < NGPUS);
        if (!gi->allocated[i]) {
            gi->allocated[i] = 1;
            gpus[g++] = i;
        }
    }
}

void gi_release(struct gpu_info *gi,
        unsigned int ngpus,
        /* IN */ unsigned int gpus[]) {
    for (unsigned int g = 0; g < ngpus; g++) {
        assert(gpus[g] < NGPUS);
        assert(gi->allocated[gpus[g]]);
        gi->allocated[gpus[g]] = 0;
    }
    gi->nfree += ngpus;
    assert(gi->nfree <= NGPUS);
}

void gi_free(struct gpu_info *gi) {
}
```

Figure 11: GPU allocator that is not thread-safe

```
void gi_alloc(struct gpu_info *gi,
        unsigned int ngpus,
        /* OUT */ unsigned int gpus[]) {
    for (int i = 0; i < ngpus; i++)
        rthread_sema_procure(&gi->sema);
    ...
}
```

Figure 12: Not good code for `gi_alloc()`.