Synchronization Data Structures

```
// Task: Implement a queue with a maximum capacity.

// PART 1

// Basic Queue: no synchronization

typedef struct queueelem {
    queuedata_t data;
    struct queueelem *next;
} queueelem_t;

typedef struct queue {
    int capacity;
    int size;
    queueelem_t *head;
    queueelem_t *tail;
} queue_t;

// A queue is a first-in, first-out (FIFO) list of data items, with length
// at most 'q->capacity'.  We store those items as a singly linked list.
// We add elements at the 'tail' end of the queue, and remove elements from
// the 'head' end.  After calls to 'queue_enq(q, 1)', 'queue_enq(q, 2)',
// and 'queue_enq(q, 3)', the queue 'q' might look like this:
//
//    'q'
// +--------+
// | cap. 8 |
// | size 3 |               queueelem_t's
// |        |       +------+    +------+    +------+
// | head --------->|  1  ----->|  2  ----->|  3 XX|
// |        |       +------+    +------+    +------+
// | tail -----\                               ^
// +--------+   \-----------------------------/
//
// Then, after calling 'queue_deq(q)' (which would return 1):
//
//    'q'
// +--------+
// | cap. 8 |
// | size 2 |
// |        |                   +------+    +------+
// | head ---------------------->|  2  ----->|  3 XX|
// |        |                   +------+    +------+
// | tail -----\                               ^
// +--------+   \-----------------------------/
//
```

```
// Then, after calling 'queue_enq(q, 4)':
//
//    'q'
// +--------+
// | cap. 8 |
// | size 3 |
// |        |                  +------+    +------+    +------+
// | head ---------------------->|  2  ----->|  3  ----->|  4 XX|
// |        |                  +------+    +------+    +------+
// | tail -----\                                           ^
// +--------+   \-----------------------------------------/
//
// Some invariants:
// * q->size >= 0
// * q->size <= q->capacity
// * If q->tail == NULL, then q->head == NULL and q->size == 0
// * If q->head == NULL, then q->tail == NULL and q->size == 0
// * If q->tail != NULL, then q->tail->next == NULL

// Create a new queue
queue_t *
queue_init(int capacity)
{
    queue_t *q = (queue_t *) malloc(sizeof(queue_t));
    q->head = q->tail = NULL;
    q->size = 0;
    q->capacity = capacity;
    return q;
}

// Enqueue 'data'
void
queue_enq(queue_t *q, queuedata_t data)
{
    queueelem_t *qe = (queueelem_t *) malloc(sizeof(queueelem_t));
    assert(q->size < q->capacity);

    qe->data = data;
    qe->next = NULL;
    if (q->tail)
        q->tail->next = qe;
    else
        q->head = qe;
    q->tail = qe;
    q->size++;
}
```

```
// Dequeue and return
queuedata_t
queue_deq(queue_t *q)
{
    queueelem_t *qe;
    queuedata_t data;
    assert(q->size > 0);

    qe = q->head;
    q->head = qe->next;
    if (q->head == NULL)
        q->tail = NULL;
    q->size--;

    data = qe->data;
    free((void *) qe);
    return data;
}
```

```
// PART 2

// Synchronized Queue: add spin locks to prevent race conditions

typedef struct queue {
    // ...
    spinlock_t lock;
} queue_t;

queue_t *
queue_init(int capacity)
{
    // ...
    spinlock_init(&q->lock);
    return q;
}

void
queue_enq(queue_t *q, queuedata_t data)
{
    queueelem_t *qe = (queueelem_t *) malloc(sizeof(queueelem_t));
    qe->data = data;

    // Spin until there's room on the queue
    while (1) {
        spin_lock(&q->lock);
        if (q->size < q->capacity)
            break;
        spin_unlock(&q->lock);
    }

    qe->data = data;
    qe->next = NULL;
    if (q->tail)
        q->tail->next = qe;
    else
        q->head = qe;
    q->tail = qe;
    q->size++;

    spin_unlock(&q->lock);
}
```

```
queuedata_t
queue_deq(queue_t *q)
{
    queueelem_t *qe = NULL;
    queuedata_t data;

    // Spin until there's something on the queue
    while (1) {
        spin_lock(&q->lock);
        if (q->head != NULL)
            break;
        spin_unlock(&q->lock);
    }

    qe = q->head;
    q->head = qe->next;
    if (q->head == NULL)
        q->tail = NULL;
    q->size--;

    // Done with shared variables, so release lock
    spin_unlock(&q->lock);
    data = qe->data;
    free((void *) qe);
    return data;
}
```

```
// PART 3

// Simple implementation of sleep() and wakeup(), to demonstrate the
// sleep/wakeup race condition

void
sleep(process_t *p, waitqueue_t *wq)
{
    p->waiting = wq;
    p->wait_next = wq->next;
    wq->next = p;
    p->blocked = 1;
    yield();    // Will not return until wq is woken up
}

void
wakeup(waitqueue_t *wq)
{
    process_t *p;
    while ((p = wq->next)) {
        wq->next = p->wait_next;
        p->waiting = 0;
        p->wait_next = NULL;
        p->blocked = 0;
    }
}
```

```
// PART 4

// Queue with Condition Variables

// 'q->nonempty' represents the condition 'queue q is not empty'.
// 'q->nonfull' represents the condition 'queue q is not full'.

typedef struct queue {
    // ...
    condition_t nonempty;
    condition_t nonfull;
} queue_t;

void
queue_enq(queue_t *q, queuedata_t data)
{
    queueelem_t *qe = (queueelem_t *) malloc(sizeof(queueelem_t));
    qe->data = data;

    spin_lock(&q->lock);
    while (q->size == q->capacity)
        condition_wait(&q->nonfull, &q->lock);

    qe->data = data;
    qe->next = NULL;
    if (q->tail)
        q->tail->next = qe;
    else
        q->head = qe;
    q->tail = qe;
    q->size++;

    spin_unlock(&q->lock);
    // Q: Why is it OK to signal the condition outside the spin lock?
    condition_signal(&q->nonempty);
}
```

```
queuedata_t
queue_deq(queue_t *q)
{
    queueelem_t *qe = NULL;
    queuedata_t data;

    spin_lock(&q->lock);
    while (q->size == 0)
        condition_wait(&q->nonempty, &q->lock);

    qe = q->head;
    q->head = qe->next;
    if (q->head == NULL)
        q->tail = NULL;
    q->size--;

    spin_unlock(&q->lock);
    data = qe->data;
    free((void *) qe);
    condition_signal(&q->nonfull);
    return data;
}
```

```
// PART 5

// Lock-Free Queue with single writer and single reader

typedef struct queue {
    int capacity;
    queuedata_t *queue;
    int head;
    int tail;
} queue_t;

// Locks and other synchronization objects (such as condition variables)
// are very powerful, but they're not the only way to synchronize!
// A "lock-free" data structure is safe to use concurrently, but totally
// avoids locking.  Some advantages (+) and disadvantages (-):
// (+) Lock-free data structures often perform better, because there's no
//     locking overhead.
// (+) Lock-free data structures avoid concurrency bugs by design.
// (-) Lock-free data structures can be more complex than conventional,
//     locking-based data structures.

// This code implements a lock-free FIFO queue, under the single-reader/
// single/writer assumption.  That is, we assume that at most one process will
// call queue_enq at a time, and similarly for queue_deq; but processes might
// call queue_enq and queue_deq simultaneously, so we still have to be careful!

// The basic idea is simple: we divide up the data structure's state so that
// every member has *exactly one writer*.  Then there's no need for locking,
// because there's no field-level concurrency!

// For example, look at the code for the lock-protected queue.  Notice that
// the 'enq' method mostly references 'q->tail', and the 'deq' method
// mostly references 'q->head'.  This is just like we'd expect: adding data
// to the queue affects the tail, and removing data affects the head.  So
// in the lock-free queue, we just make sure that 'enq' NEVER writes to
// 'q->head', and that 'deq' NEVER writes to 'q->tail'!  But how to do
// this?  We can't use linked lists any longer, since they would require
// 'enq' to write to 'q->head' occasionally.  Instead, we implement the
// queue as an array, and 'q->head' and 'q->tail' as indexes into the
// array.  That gives us everything we want.

// All this is clearer in some examples.  Here's an empty lock-free queue
// with capacity of 8.  Note that there are nine elements in the array;
// this lets us distinguish between an empty queue and a full queue (you'll
// see how).
//
// +--------+     +----+----+----+----+----+----+----+----+----+
// | cap. 8 |     |    |    |    |    |    |    |    |    |    |
// |        |     +----+----+----+----+----+----+----+----+----+
// | head 0 |------^  ^
// | tail 0 |---------+
// +--------+
```

```
//
// Note that 'q->head == q->tail' means the queue is empty.
// After calling 'queue_enq(q, 1)', the queue shifts to this state:
//
// +--------+     +----+----+----+----+----+----+----+----+----+
// | cap. 8 |     |  1 |    |    |    |    |    |    |    |    |
// |        |     +----+----+----+----+----+----+----+----+----+
// | head 0 |-------^    ^
// | tail 1 |------------+
// +--------+
//
// After calling 'queue_enq(q, 2)':
//
// +--------+     +----+----+----+----+----+----+----+----+----+
// | cap. 8 |     |  1 |  2 |    |    |    |    |    |    |    |
// |        |     +----+----+----+----+----+----+----+----+----+
// | head 0 |-------^         ^
// | tail 2 |-----------------+
// +--------+
//
// Then after calling 'queue_deq(q)', which returns 1:
//
// +--------+     +----+----+----+----+----+----+----+----+----+
// | cap. 8 |     |    |  2 |    |    |    |    |    |    |    |
// |        |     +----+----+----+----+----+----+----+----+----+
// | head 1 |------------^    ^
// | tail 2 |-----------------+
// +--------+
//
// Note that the enq operation only writes to (1) the 'tail' member and
// (2) the previously-blank queue entries.  The deq operation, by contrast,
// writes to (1) the 'head' member and (2) the NON-blank queue entries.
// The different operations *never write to the same part of the data
// structure*!  Thus, lock-freedom.
//
// If we add and remove a bunch more items, we might reach a state like
// this:
//
// +--------+     +----+----+----+----+----+----+----+----+----+
// | cap. 8 |     | 10 |    |    |    | 6  | 7  | 8  | 9  |    |
// |        |     +----+----+----+----+----+----+----+----+----+
// | head 5 |--------------------------------^
// | tail 1 |------------^
// +--------+
//
// This queue contains the elements 6, 7, 8, 9, and 10, in that order.
// Notice that the tail pointer has "wrapped around"; this is why the
// queue is called circular.
//
```

```
// If we add 3 more items to the queue, it will become full:
//
// +--------+     +----+----+----+----+----+----+----+----+----+
// | cap. 8 |     | 10 | 11 | 12 | 13 |    | 6  | 7  | 8  | 9  |
// |        |     +----+----+----+----+----+----+----+----+----+
// | head 5 |--------------------------------^
// | tail 4 |---------------------------^
// +--------+
//
// Notice that a "full" queue has one blank slot.  We need that blank slot
// to distinguish the "full" case from the "empty" case.  Specifically:
//    The queue is empty  when  head == tail;
//    The queue is full   when  (head + capacity) % (capacity + 1) == tail
//                              --- that is, when there's one slot left.
//
// Lock-free data structures are a particularly powerful form of advanced
// OS data structure design.  It's worth your time to think about this
// code for a while.  Note that, although the code may appear unfamiliar to
// you, it is *shorter* than any of the locking designs.  Feel free to ask
// questions!
//
// A question for you: How would you make this data structure safe for
// multiple concurrent readers and/or writers?

queue_t *
queue_init(int capacity)
{
    queue_t *q = (queue_t *) malloc(sizeof(queue_t));
    q->queue = (queuedata_t *) malloc(sizeof(queuedata_t) * (capacity + 1));
    q->head = 0;
    q->tail = 0;
    q->capacity = capacity;
    return q;
}

void
queue_enq(queue_t *q, queuedata_t data)
{
    // This is lock-free, but not *wait*-free, since we wait for a slot if the
    // queue is full.  But that can't be avoided unless we implement a
    // non-blocking enq.
    while ((q->tail + 1) % (q->capacity + 1) == q->head)
        yield();

    q->queue[q->tail] = data;
    q->tail = (q->tail + 1) % (q->capacity + 1);
}
```

```
queuedata_t
queue_deq(queue_t *q)
{
    queuedata_t data;

    while (q->head == q->tail)
        yield();

    data = q->queue[q->head];
    q->head = (q->head + 1) % (q->capacity + 1);
    return data;
}
```