Thread synchronization

David Hovemeyer

2 December 2019

```c
const int NUM_INCR=100000000, NTHREADS=2;
typedef struct { volatile int count; } Shared;

void *worker(void *arg) {
  Shared *obj = arg;
  for (int i = 0; i < NUM_INCR/NTHREADS; i++)
    obj->count++;
  return NULL;
}

int main(void) {
  Shared *obj = calloc(1, sizeof(Shared));
  pthread_t threads[NTHREADS];
  for (int i = 0; i < NTHREADS; i++)
    pthread_create(&threads[i], NULL, worker, obj);
  for (int i = 0; i < NTHREADS; i++)
    pthread_join(threads[i], NULL);
  printf("%d\n", obj->count);
  return 0;
}
```

The program uses two threads, which repeatedly increment a shared counter

The counter is incremented a total of 100,000,000 times, starting from 0

So, the final value should be 100,000,000; running the program, we get ▌

```
$ gcc -Wall -Wextra -pedantic -std=gnu11 -O2 -c incr_race.c
$ gcc -o incr_race incr_race.o -lpthread
$ ./incr_race
53015619
```

▌

What happened?

Atomicity

Incrementing the counter (obj->count++) is not *atomic* ▌

In general, we should think of var++ as really meaning

```
reg = var;
reg = reg + 1;
var = reg;
```
▌

When threads are executing concurrently, it's possible for the variable to change between the time its value is loaded and the time the updated value is stored ▌

Example of a *data race* causing a *lost update*

Concurrent access to shared data

Point to ponder: if concurrent access can screw up something as simple as an *integer counter*, imagine the complete mess it will make of your linked list, balanced tree, etc. ▋

Data structures have invariants which must be preserved ▋

Mutations (insertions, removals) often violate these invariants temporarily

- Not a problem in a sequential program because the operation will complete (and restore invariants) before anyone notices

- *Huge* problem in concurrent program where multiple threads could access the data structure at the same time ▋

Synchronization: protect shared data from concurrent access

Full source code for all of today's examples is on web page,
`synch.zip`

Semaphores and mutexes

Critical sections

A *critical section* is a region of code in which *mutual exclusion* must be guaranteed for correct behavior ▮

Mutual exclusion means that at most one concurrent task (thread) may be accessing shared data at any given time ▮

Enforcing mutual exclusion in critical sections guarantees *atomicity*

- I.e., code in critical section executes to completion without interruption ▮

For the shared counter program, the update to the shared counter variable is a critical section

Semaphores and *mutexes* are two types of synchronization constructs available in pthreads

Both can be used to guarantee mutual exclusion

Semaphores can also be used to manage access to a finite resource

Mutexes (a.k.a., ''mutual exclusion locks'') are simpler, so let's discuss them first

`pthread_mutex_t`: data type for a pthreads mutex

`pthread_mutex_init`: initialize a mutex

`pthread_mutex_lock`: locks a mutex for exclusive access

- If another thread has already locked the mutex, calling thread must wait

`pthread_mutex_unlock`: unlocks a mutex

- If any threads are waiting to lock the mutex, one will be woken up and allowed to acquire it

`pthread_mutex_destroy`: destroys a mutex (once it is no longer needed)

Using a mutex

Using a mutex to protected a shared data structure: ▊

- Associate a pthread_mutex_t variable with each instance of the data structure ▊

- Initialize with pthread_mutex_init when the data structure is initialized ▊

- Each critical section is protected with calls to pthread_mutex_lock and pthread_mutex_unlock ▊

- Destroy mutex with pthread_mutex_destroy when data structure is deallocated ▊

It's not too complicated!

Definition of Shared struct type:

```
typedef struct {
    volatile int count;
    pthread_mutex_t lock;
} Shared;
```

Definition of the worker function:

```
void *worker(void *arg) {
    Shared *obj = arg;
    for (int i = 0; i < NUM_INCR/NTHREADS; i++) {
        pthread_mutex_lock(&obj->lock);
        obj->count++;
        pthread_mutex_unlock(&obj->lock);
    }
    return NULL;
}
```

Main function:

```c
int main(void) {
  Shared *obj = calloc(1, sizeof(Shared));
  pthread_mutex_init(&obj->lock, NULL);
  pthread_t threads[NTHREADS];
  for (int i = 0; i < NTHREADS; i++)
    pthread_create(&threads[i], NULL, worker, obj);
  for (int i = 0; i < NTHREADS; i++)
    pthread_join(threads[i], NULL);
  printf("%d\n", obj->count);
  pthread_mutex_destroy(&obj->lock);
  return 0;
}
```

Original version with lost update bug: ▊

```
$ time ./incr_race
52683607


real    0m0.142s
user    0m0.276s
sys     0m0.000s
```

Fixed version using mutex: ▊

```
$ time ./incr_fixed
100000000
```

```
real    0m10.262s
user    0m13.210s
sys     0m7.264s
```

Contention occurs when multiple threads try to access the same shared data structure at the same time

Costs associated with synchronization:

1. Cost of entering and leaving critical section (e.g., locking and unlocking a mutex)

2. Reduced parallelism due to threads having to take turns (when contending for access to shared data)

3. Cost of OS kernel code to suspend and resume threads as they wait to enter critical sections

These costs can be significant! Best performance occurs when threads synchronize relatively infrequently

- Shared counter example is a pathological case

A *semaphore* is a more general synchronization construct, invented by Edsger Dijkstra in the early 1960s

When created, semaphore is initialized with a nonnegative integer count value

Two operations:

- P (''proberen''): waits until the semaphore has a non-zero value, then decrements the count by one

- V (''verhogen''): increments the count by one, waking up a thread waiting to perform a P operation if appropriate

A mutex can be modeled as a semaphore whose initial value is 1

Include the <semaphore.h> header file

Semaphore data type is sem_t

Functions:

- sem_init: initialize a semaphore with specified initial count

- sem_destroy: destroy a semaphore when no longer needed

- sem_wait: wait and decrement (P)

- sem_post: increment and wake up waiting thread (V)

Semaphores are useful for managing access to a limited resource

Example: limiting maximum number of threads in a server application

- Initialize semaphore with desired maximum number of threads

- Use P operation before creating a client thread

- Use V operation when client thread finishes

Semaphore applications

Example: *bounded queue*

- Initially empty, can have up to a fixed maximum number of elements

- When enqueuing an item, thread waits until queue is not full

- When dequeuing an item, thread waits until queue is not empty

Implementation: two semaphores and one mutex

- *slots* semaphore: tracks how many slots are available

- *items* semaphore: tracks how many elements are present

- Mutex is used for critical sections accessing queue data structure

Bounded queue data structure

Bounded queue of generic (void *) pointers

Bounded queue data type:

```
typedef struct {
  void **data;
  unsigned max_items, head, tail;
  sem_t slots, items;
  pthread_mutex_t lock;
} BoundedQueue;
```

Bounded queue operations:

```
BoundedQueue *bqueue_create(unsigned max_items);
void bqueue_destroy(BoundedQueue *bq);
void bqueue_enqueue(BoundedQueue *bq, void *item);
void *bqueue_dequeue(BoundedQueue *bq);
```

Creating bounded queue

The *slots* semaphore initialized with max number of items,
and *items* semaphore initialized to 0

```
BoundedQueue *bqueue_create(unsigned max_items) {
    BoundedQueue *bq = malloc(sizeof(BoundedQueue));
    bq->data = malloc(max_items * sizeof(void *));
    bq->max_items = max_items;
    bq->head = bq->tail = 0;
    sem_init(&bq->slots, 0, max_items);
    sem_init(&bq->items, 0, 0);
    pthread_mutex_init(&bq->lock, NULL);
    return bq;
}
```

Slots decreases (must wait until nonzero before new item can be added), items increases

Queue implemented as a ``circular'' array of pointers: head refers to where next item will be added, tail refers to where next item will be removed

```
void bqueue_enqueue(BoundedQueue *bq, void *item) {
  sem_wait(&bq->slots);           /* wait for empty slot */
  pthread_mutex_lock(&bq->lock);
  bq->data[bq->head] = item;
  bq->head = (bq->head + 1) % bq->max_items;
  pthread_mutex_unlock(&bq->lock);
  sem_post(&bq->items);           /* item is available */
}
```

Items decreases (must wait until nonzero before item can be removed), slots increases

```
void *bqueue_dequeue(BoundedQueue *bq) {
  sem_wait(&bq->items);                /* wait for item */
  pthread_mutex_lock(&bq->lock);
  void *item = bq->data[bq->tail];
  bq->tail = (bq->tail + 1) % bq->max_items;
  pthread_mutex_unlock(&bq->lock);
  sem_post(&bq->slots);                /* empty slot is available */
  return item;
}
```

Queues are useful!

Synchronized queues are *extremely* useful in multithreaded programs! █

In particular they are useful for *producer/consumer* relationships between threads █

- Producer enqueues items █

- Consumer dequeues items █

- Bounded queue: ensures that producer doesn't get too far ahead of consumer █

More generally, a queue can be used to send a *message* to another thread

Creating threads incurs some overhead

Prethreading: program creates a fixed number of threads ahead of time, assigns work to them as it becomes available

Queues are an ideal mechanism to allow the ''master'' thread to send work to the worker threads

A queue can also be used for messages sent from the workers back to the master thread

Conway's game of life
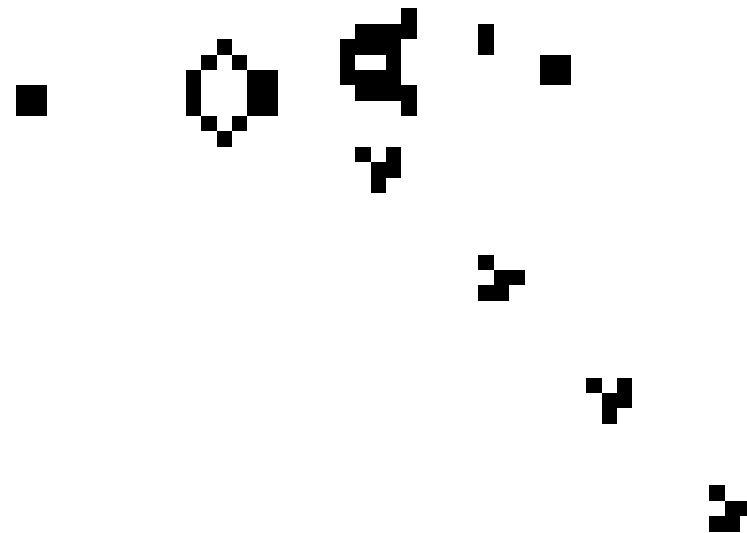
- Grid-based cellular automaton, cells are alive (1) or dead (0)

- Live cells with 2 or 3 live neighbors survive

- Dead cells with 3 live neighbors become alive

- Otherwise, cell dies (or stays dead)

- Over many generations, complex patterns can emerge

Grid data type:

```
typedef struct {
  unsigned nrows, ncols;
  char *cur_buf, *next_buf;
} Grid;
```

Two buffers, one for current generation, one for next generation
(swap after each generation is simulated)

Sequential computation function:

```
void life_compute_next(Grid *grid, unsigned start_row,
                       unsigned end_row);
```

Updates cells in next generation for specified range of grid rows

Sequential computation

Simulating specified number of generations:

```
for (unsigned i = 0; i < num_gens; i++) {
  life_compute_next(grid, 1, grid->nrows - 1);
  grid_flip(grid);
}
```

Note that border cells are never updated (and are always 0)

Conway's game of life is not quite an *embarrassingly parallel* computation

- Computation of generation n must finish before computation of generation $n+1$ can start ▮

Could start a new batch of worker threads each generation

- But we'll repeatedly pay the thread startup and teardown costs ▮

Prethreading approach: ▮

- Create fixed set of worker threads ▮

- ''Command queue'' allows master thread to send tasks to the workers ▮

- ''Done queue'' allows workers to notify master thread when tasks are finished

Work data type, has queues and main Grid data structure

```
typedef struct {
    BoundedQueue *cmd_queue;
    BoundedQueue *done_queue;
    Grid *grid;
} Work;
```

Task data type, represents a range of grid rows for a worker to update

```
typedef struct {
    unsigned start_row, end_row;
} Task;
```

worker function, executed by each worker thread:

```
void *worker(void *arg) {
  Work *w = arg;

  while (1) {
    Task *t = bqueue_dequeue(w->cmd_queue);
    if (t->end_row == 0) { break; }

    /* do sequential computation */
    life_compute_next(w->grid, t->start_row, t->end_row);

    /* inform main thread that task is done */
    bqueue_enqueue(w->done_queue, t);
  }

  return NULL;
}
```

Master thread:

```
Work w = { bqueue_create(NUM_THREADS), bqueue_create(NUM_THREADS), grid };
pthread_t threads[NUM_THREADS];
for (unsigned i = 0; i < NUM_THREADS; i++) {
  pthread_create(&threads[i], NULL, worker, &w);
}


for (unsigned i = 0; i < num_gens; i++) {     /* simulation loop */
  distribute_work(&w, 0);
  wait_until_done(&w);
  grid_flip(grid);
}


distribute_work(&w, 1);                            /* send shutdown message */


for (unsigned i = 0; i < NUM_THREADS; i++) { /* wait for workers to finish */
  pthread_join(threads[i], NULL);
}
```

Distributing work:

```
void distribute_work(Work *w, int done) {
  unsigned rows_per_thread = (w->grid->nrows - 2) / NUM_THREADS;
  for (unsigned i = 0; i < NUM_THREADS; i++) {
    Task *task = malloc(sizeof(Task));
    if (done) {
      task->end_row = 0;
    } else {
      task->start_row = 1 + (i*rows_per_thread);
      if (i == NUM_THREADS-1) { task->end_row = w->grid->nrows - 1; }
                        else { task->end_row = task->start_row + rows_per_thread; }
    }

    bqueue_enqueue(w->cmd_queue, task);
  }
}
```

Waiting for workers to finish their tasks:

```
void wait_until_done(Work *w) {
  for (unsigned i = 0; i < NUM_THREADS; i++) {
    Task *t = bqueue_dequeue(w->done_queue);
    free(t);
  }
}
```

Sequential vs. parallel

Using a 1000x1000 cell input, 10,000 generations, sequential vs. parallel with 4 worker threads, on a Core i5-3320M (dual core, hyperthreaded):

```
$ ./life_seq board.txt 10000 out10000.txt
Computation finished in 59007 ms
$ ./life_par board.txt 10000 out10000par.txt
Computation finished in 32208 ms
$ diff out10000.txt out10000par.txt
```
no output

We got about a 2x speedup using four threads

Relatively large chunks of work were assigned

- Costs of synchronization amortized over relatively large amounts of sequential computation done by worker threads

Queues are an effective mechanism for communication between threads