Locks: Principles

ICS432

Concurrent and High-Performance Programming

Henri Casanova (henric@hawaii.edu)

Disclaimer

- The first few slides are a review of ICS332 material
- But this material is so important, that going over it again is a good idea
    - Especially if ICS332 was a few semesters ago

Thread Safety

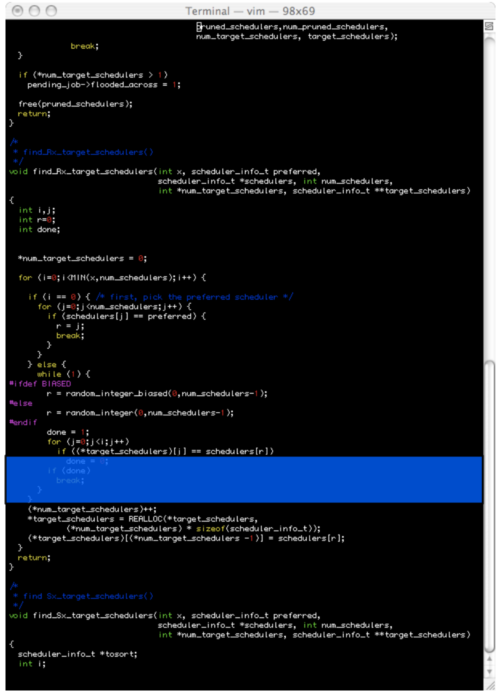
- In the previous module we've talked about the need for thread safety
  - i.e., the need to have threads read/write the same memory locations without race condition
- In this module we talk about how we can make a piece of code thread-safe
- The most basic concept for achieving thread safety is called a **Lock**

Atomicity and mutual exclusion

- What we need is a mechanism that makes a **sequence** of operations atomic
- Atomicity is really mutual exclusion
  - Whenever the sequence is initiated by thread A, we are guaranteed that no other thread can initiate it before thread A completes it
  - If you've taken databases, you can think of this is a kind of transaction
- This is a great idea, but how can we specify this in a program?
- Answer: critical sections
- A critical section is a section of code in which only one thread is allowed at a time
  - Not necessarily a contiguous section of code

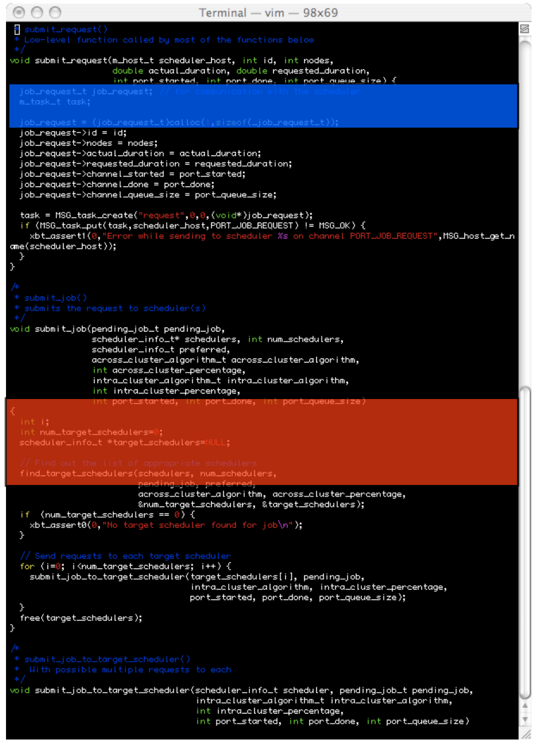
Critical Sections

- Two critical sections in a program
  - No two threads can be in “blue” code at the same time
  - No two threads can be in “red” code at the same time
  - We could have one thread doing “blue”, one thread doing “red”, and many threads doing anything else



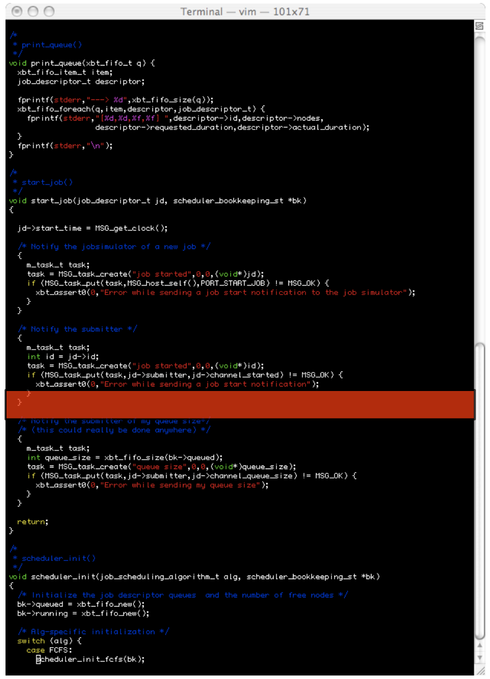
Terminal — vim — 98x69

```
                                              pruned_schedulers,num_pruned_schedulers,
                                              num_target_schedulers, target_schedulers);
            break;
  }

  if (*num_target_schedulers > 1)
    pending_job->flooded_across = 1;

  free(pruned_schedulers);
  return;
}

/*
 * find_Rx_target_schedulers()
 */
void find_Rx_target_schedulers(int x, scheduler_info_t preferred,
                               scheduler_info_t *schedulers, int num_schedulers,
                               int *num_target_schedulers, scheduler_info_t **target_schedulers)
{
  int i,j;
  int r=0;
  int done;


  *num_target_schedulers = 0;

  for (i=0;i<MIN(x,num_schedulers);i++) {

    if (i == 0) { /* first, pick the preferred scheduler */
      for (j=0;j<num_schedulers;j++) {
        if (schedulers[j] == preferred) {
          r = j;
          break;
        }
      }
    } else {
      while (1) {
#ifdef BIASED
        r = random_integer_biased(0,num_schedulers-1);
#else
        r = random_integer(0,num_schedulers-1);
#endif
        done = 1;
        for (j=0;j<i;j++)
          if ((*target_schedulers)[j] == schedulers[r])
            done = 0;
        if (done)
          break;
      }
    }
    (*num_target_schedulers)++;
    *target_schedulers = REALLOC(*target_schedulers,
            (*num_target_schedulers) * sizeof(scheduler_info_t));
    (*target_schedulers)[(*num_target_schedulers -1)] = schedulers[r];
  }
  return;
}

/*
 * find Sx_target_schedulers()
 */
void find_Sx_target_schedulers(int x, scheduler_info_t preferred,
                               scheduler_info_t *schedulers, int num_schedulers,
                               int *num_target_schedulers, scheduler_info_t **target_schedulers)
{
  scheduler_info_t *tosort;
  int i;
```



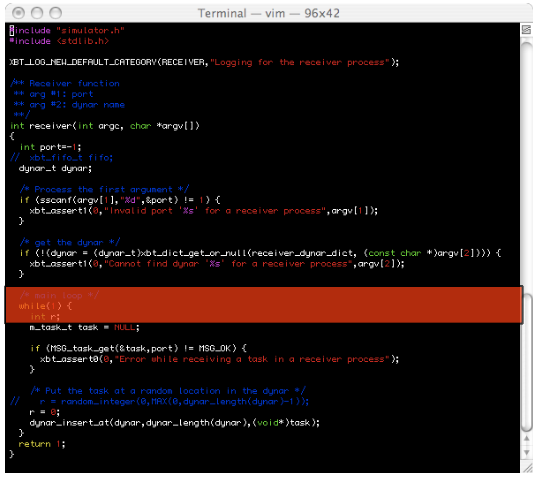
Terminal — vim — 98x69

```
 * submit_request()
 * Low-level function called by most of the functions below
 */
void submit_request(m_host_t scheduler_host, int id, int nodes,
                    double actual_duration, double requested_duration,
                    int port_started, int port_done, int port_queue_size) {
  job_request_t job_request; // for communication with the scheduler
  m_task_t task;

  job_request = (job_request_t)calloc(1,sizeof(_job_request_t));
  job_request->id = id;
  job_request->nodes = nodes;
  job_request->actual_duration = actual_duration;
  job_request->requested_duration = requested_duration;
  job_request->channel_started = port_started;
  job_request->channel_done = port_done;
  job_request->channel_queue_size = port_queue_size;

  task = MSG_task_create("request",0,0,(void*)job_request);
  if (MSG_task_put(task,scheduler_host,PORT_JOB_REQUEST) != MSG_OK) {
    xbt_assert1(0,"Error while sending to scheduler %s on channel PORT_JOB_REQUEST",MSG_host_get_n
ame(scheduler_host));
  }
}

/*
 * submit_job()
 * submits the request to scheduler(s)
 */
void submit_job(pending_job_t pending_job,
                scheduler_info_t* schedulers, int num_schedulers,
                scheduler_info_t preferred,
                across_cluster_algorithm_t across_cluster_algorithm,
                int across_cluster_percentage,
                intra_cluster_algorithm_t intra_cluster_algorithm,
                int intra_cluster_percentage,
                int port_started, int port_done, int port_queue_size)
{
  int i;
  int num_target_schedulers=0;
  scheduler_info_t *target_schedulers=NULL;

  // Find out the list of appropriate schedulers
  find_target_schedulers(schedulers, num_schedulers,
                         pending_job, preferred,
                         across_cluster_algorithm, across_cluster_percentage,
                         &num_target_schedulers, &target_schedulers);
  if  (num_target_schedulers == 0) {
    xbt_assert0(0,"No target scheduler found for job\n");
  }

  // Send requests to each target scheduler
  for (i=0; i<num_target_schedulers; i++) {
    submit_job_to_target_scheduler(target_schedulers[i], pending_job,
                                   intra_cluster_algorithm, intra_cluster_percentage,
                                   port_started, port_done, port_queue_size);
  }
  free(target_schedulers);
}

/*
 * submit_job_to_target_scheduler()
 *  With possible multiple requests to each
 */
void submit_job_to_target_scheduler(scheduler_info_t scheduler, pending_job_t pending_job,
                                    intra_cluster_algorithm_t intra_cluster_algorithm,
                                    int intra_cluster_percentage,
                                    int port_started, int port_done, int port_queue_size)
```



Terminal — vim — 101x71

```
/*
 * print_queue()
 */
void print_queue(xbt_fifo_t q) {
  xbt_fifo_item_t item;
  job_descriptor_t descriptor;

  fprintf(stderr,"---> %d",xbt_fifo_size(q));
  xbt_fifo_foreach(q,item,descriptor,job_descriptor_t) {
    fprintf(stderr,"[%d,%d,%f,%f] ",descriptor->id,descriptor->nodes,
                   descriptor->requested_duration,descriptor->actual_duration);
  }
  fprintf(stderr,"\n");
}

/*
 * start_job()
 */
void start_job(job_descriptor_t jd, scheduler_bookkeeping_st *bk)
{

  jd->start_time = MSG_get_clock();

  /* Notify the jobsimulator of a new job */
  {
    m_task_t task;
    task = MSG_task_create("job started",0,0,(void*)jd);
    if (MSG_task_put(task,MSG_host_self(),PORT_START_JOB) != MSG_OK) {
      xbt_assert0(0,"Error while sending a job start notification to the job simulator");
    }
  }

  /* Notify the submitter */
  {
    m_task_t task;
    int id = jd->id;
    task = MSG_task_create("job started",0,0,(void*)id);
    if (MSG_task_put(task,jd->submitter,jd->channel_started) != MSG_OK) {
      xbt_assert0(0,"Error while sending a job start notification");
    }
  }

  /* Notify the submitter of my queue size*/
  /* (this could really be done anywhere) */
  {
    m_task_t task;
    int queue_size = xbt_fifo_size(bk->queued);
    task = MSG_task_create("queue size",0,0,(void*)queue_size);
    if (MSG_task_put(task,jd->submitter,jd->channel_queue_size) != MSG_OK) {
      xbt_assert0(0,"Error while sending my queue size");
    }
  }

  return;
}

/*
 * scheduler_init()
 */
void scheduler_init(job_scheduling_algorithm_t alg, scheduler_bookkeeping_st *bk)
{
  /* Initialize the job descriptor queues  and the number of free nodes */
  bk->queued = xbt_fifo_new();
  bk->running = xbt_fifo_new();

  /* Alg-specific initialization */
  switch (alg) {
    case FCFS:
      scheduler_init_fcfs(bk);
```



Terminal — vim — 96x42

```
#include "simulator.h"
#include <stdlib.h>

XBT_LOG_NEW_DEFAULT_CATEGORY(RECEIVER,"Logging for the receiver process");

/** Receiver function
 ** arg #1: port
 ** arg #2: dynar name
 **/
int receiver(int argc, char *argv[])
{
  int port=-1;
//  xbt_fifo_t fifo;
  dynar_t dynar;

  /* Process the first argument */
  if (sscanf(argv[1],"%d",&port) != 1) {
    xbt_assert1(0,"Invalid port '%s' for a receiver process",argv[1]);
  }

  /* get the dynar */
  if (!(dynar = (dynar_t)xbt_dict_get_or_null(receiver_dynar_dict, (const char *)argv[2]))) {
    xbt_assert1(0,"Cannot find dynar '%s' for a receiver process",argv[2]);
  }

  /* main loop */
  while(1) {
    int r;
    m_task_t task = NULL;

    if (MSG_task_get(&task,port) != MSG_OK) {
      xbt_assert0(0,"Error while receiving a task in a receiver process");
    }

    /* Put the task at a random location in the dynar */
//    r = random_integer(0,MAX(0,dynar_length(dynar)-1));
    r = 0;
    dynar_insert_at(dynar,dynar_length(dynar),(void*)task);
  }
  return 1;
}
```

Critical Sections

- One would like to write code that looks like this:

```
while(true) {
    enter_CS(blue)
    x++
    leave_CS(blue)
}
```

- We would like to have the following properties
  - Mutual exclusion: only one thread can be inside the CS
  - No deadlocks: one of the competing threads will enter the CS
  - No unnecessary delays: a thread enters the CS immediately if no other thread is competing for it
  - Eventual entry: a thread that tries to enter the critical CS will enter it at some point

Critical Sections with Locks

- The concept of a critical section is binary
  - Either 0 threads are in the critical section
  - Or 1 thread is in the critical section
- Therefore, the critical section can be "controlled" with a boolean variable
- This variable is called a **lock**
  - Can take one of two values: **"locked"** or **"unlocked"**
  - Initially set to "**unlocked**"
- Just like going to the toilet (if you've taken ICS332 from me, perhaps you remember this)
  - While the lock is "red" get in the waiting line
  - When the lock becomes "green" if you're first in line go in and set the lock to "red"
  - When you leave, set the lock to "green"

Locks

- Different languages have different ways to declare/use locks
  - We'll see ways to do it in Java and C/C++

- Let's use a C-like syntax for now:
  - Declaration: lock_t *lock = new_lock()
    (initialized to "unlocked")
  - Acquire the lock: lock(lock)
  - Release the lock: unlock(lock)

Lock() and Unlock() pseudo-code

- For now, to understand what these functions do, let’s view them as pseudo-code

```
unlock(lock_t *lock) {
    *lock = UNLOCKED
}
```

```
// “Magically” thread-safe
lock(lock_t *lock) {
    while (*lock == LOCKED) {
        // spin
    }
    *lock = LOCKED
}
```

- We will understand how to implement the above in the next set of lecture notes…
  - Clearly, lock() isn’t thread-safe as written above
  - Anybody sees why?

Lock Typical Use Case: Updates

- The typical (but not the only) use case for locks and creating critical sections is when multiple threads need to **update** the same memory locations
- All lines of code that update a memory location must then be put inside a critical section
- And typically, one uses different locks for different memory location
- Let’s do a straightforward in-class activity here…

In-Class Activity

- Consider the following code fragments, assuming a bunch of threads that call f() and g() over and over
- How many locks do you need to declare (lock1, lock2, lock3, etc.)? Put in the calls to lock/unlock…

```
//global variables
int x=100, y=0;
```

```
void f() {
   x += 2;
   y ++;
}
```

```
void g() {
   x++;
}
```

In-Class Activity

- Consider the following code fragments, assuming a bunch of threads that call f() and g() over and over
- How many locks do you need to declare (lock1, lock2, lock3, etc.)? Put in the calls to lock/unlock…

```
//global variables
int x=100, y=0;
lock_t lock1, lock2;
```

```
void f() {
   lock(lock1);
   x += 2;
   unlock(lock1);
   lock(lock2);
   y ++;
   unlock(lock2);
}
```

```
void g() {
   lock(lock1);
   x++;
   unlock(lock1);
}
```

Two locks
Two critical sections

In-Class Activity

- An implementation with a single lock like this is correct, but not concurrent!!
  - While a thread updates x, no thread can update y
  - May defeat the purpose of using threads

```
//global variables
int x=100, y=0;
lock_t lock1, lock2;
```

```
void f() {
   lock(lock1);
   x += 2;
   y ++;
   unlock(lock1);
}
```

```
void g() {
   lock(lock1);
   x++;
   unlock(lock1);
}
```

In-Class Activity

- An implementation with three locks is incorrect
  - Two threads could be updating variable x at the same time (one going “x += 2” and the other doing “x++”)

```
//global variables
int x=100, y=0;
lock_t lock1, lock2,   lock3;
```

```
void f() {
   lock(lock1);
   x += 2;
   unlock(lock1);
   lock(lock2);
   y ++;
   unlock(lock2);
}
```

```
void g() {
   lock(lock3);
   x++;
   unlock(lock3);
}
```

Locks for Data Structures

- A classical use of locks is to protect updates of linked data structures
- Example: Queue and threads
  - Consider a program that maintains a queue (of ints >0)
  - Thread #1 (Producer) adds elements to the queue
  - Thread #2 (Consumer) removes elements from the queue
  - We will see soon why this is very useful

Thread #1

```
int x;
while(1) {
    x = generate();
    insert(list,x);
}
```

Thread #2

```
int x;
while(1) {
    x = remove(list);
}
```

Queue Implementation

```
struct queue_t {
    queue_item *first;
    queue_item *last;
};
```

```
struct queue_item_t {
    int value;
    queue_item *prev;
    queue_item *next;
};
```

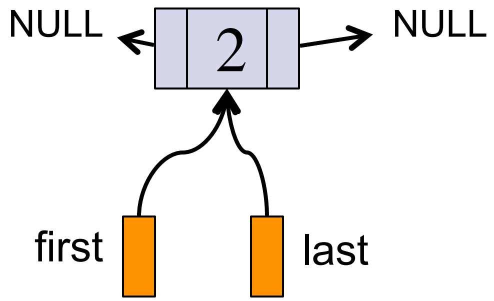
```
void insert (queue_t q, int x) {
    queue_item_t *item = (queue_item_t *) calloc(1, sizeof(queue_item_t));
    item->value = x;
    item->next = q->first;
    if (item->next)
        item->next->prev = item;
    q->first = item;
    if (! q->last)  q->last = item;
}
```

Queue Implementation

```
int remove (queue_t q) {
      queue_item_t *item;
      int x;
      if (! q->last) return -1;  // -1 means “no item”
      x = q->last->value;
      item = q->last->prev;
      free(q->last);
      if (item) {
          item->next = NULL;
      }
      q->last = item;
     if (q->last == NULL) {
          q->first = NULL;
      }
      return x;
}
```

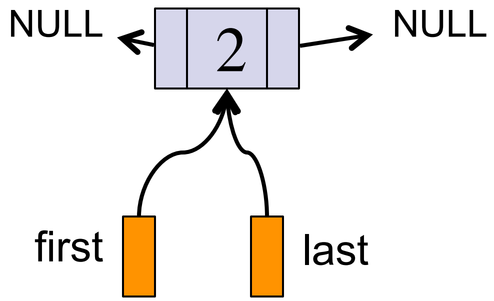
What bad thing could happen?

- Consider the following queue



What bad thing could happen?

- Consider the following queue

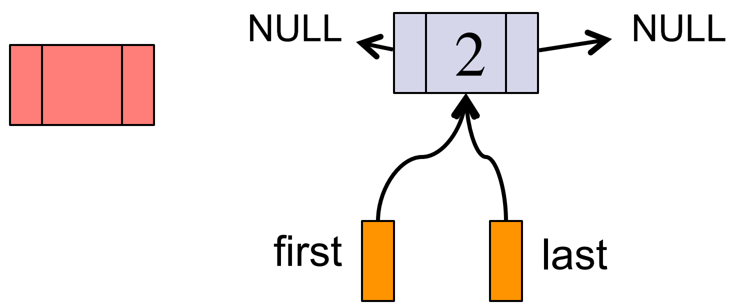


- The Producer calls insert(3)

```
queue_item_t *item = calloc(...)
item->value = x;
item->next = q->first;
if (item->next)
    item->next->prev = item;
q->first = item;
if (! q->last)
    q->last = item;
```

What bad thing could happen?

- Consider the following queue

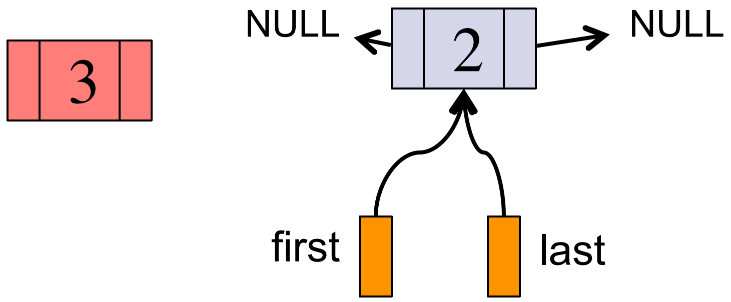


- The Producer calls insert(3)

```
queue_item_t *item = calloc(...)
item->value = x;
item->next = q->first;
if (item->next)
    item->next->prev = item;
q->first = item;
if (! q->last)
    q->last = item;
```

What bad thing could happen?

- Consider the following queue

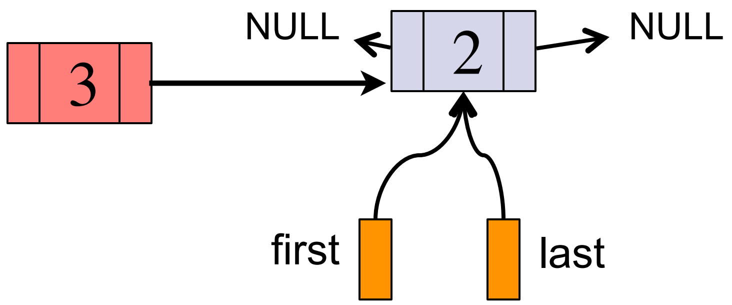


- The Producer calls insert(3)

```
queue_item_t *item = calloc(...)
item->value = x;
item->next = q->first;
if (item->next)
    item->next->prev = item;
q->first = item;
if (! q->last)
    q->last = item;
```

What bad thing could happen?

- Consider the following queue

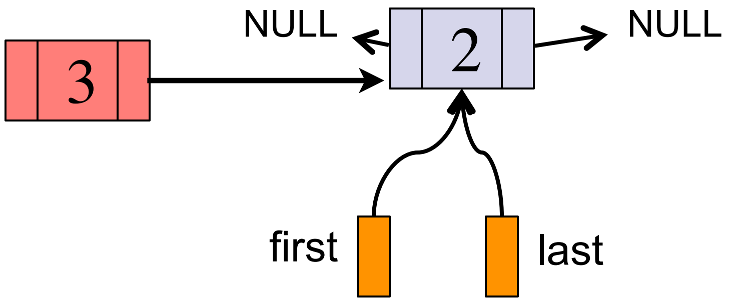


- The Producer calls insert(3)

```
queue_item_t *item = calloc(...)
item->value = x;
item->next = q->first;
if (item->next)
    item->next->prev = item;
q->first = item;
if (! q->last)
    q->last = item;
```

What bad thing could happen?

- Consider the following queue



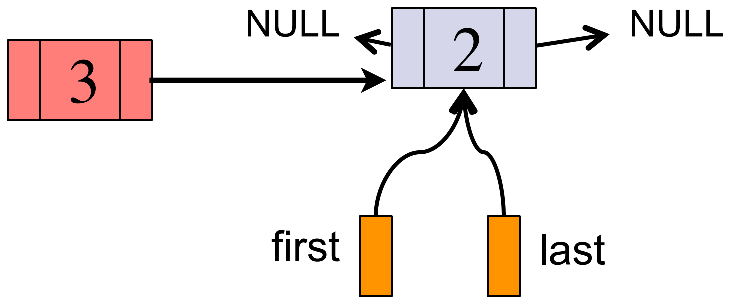
- The Producer calls insert(3)

```
queue_item_t *item = calloc(...)
item->value = x;
item->next = q->first;
if (item->next)
    item->next->prev = item;
q->first = item;
if (! q->last)
    q->last = item;
```

context switch

What bad thing could happen?

- Consider the following queue

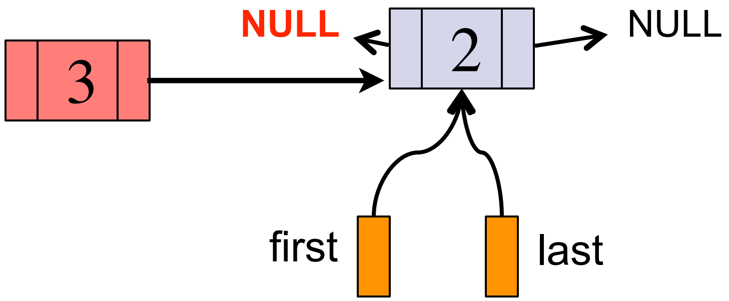


- The Consumer calls remove

```
...
item = q->last->prev;   // returns NULL
free(q->last);
if (item) {
. . .
```

What bad thing could happen?

- Consider the following queue

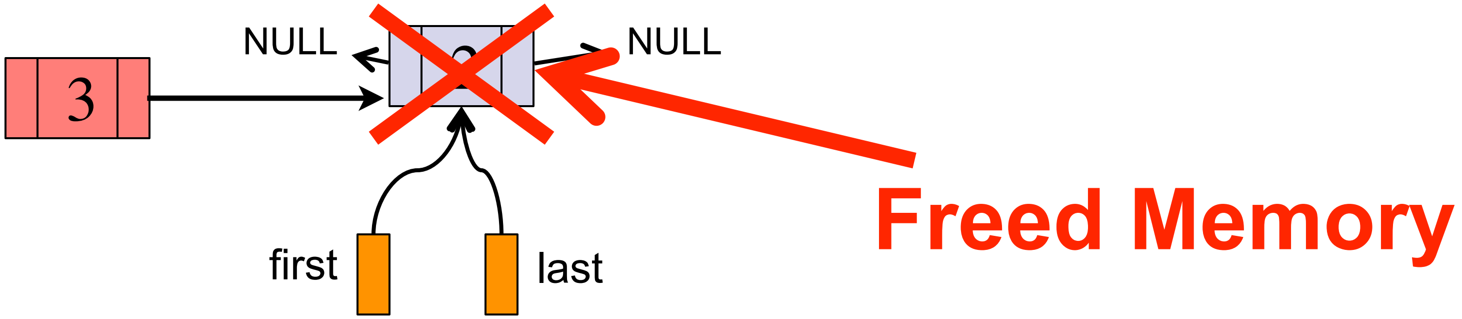


- The Consumer calls remove

```
…
item = q->last->prev;   // returns NULL
free(q->last);
if (item) {
. . .
```

What bad thing could happen?

- Consider the following queue



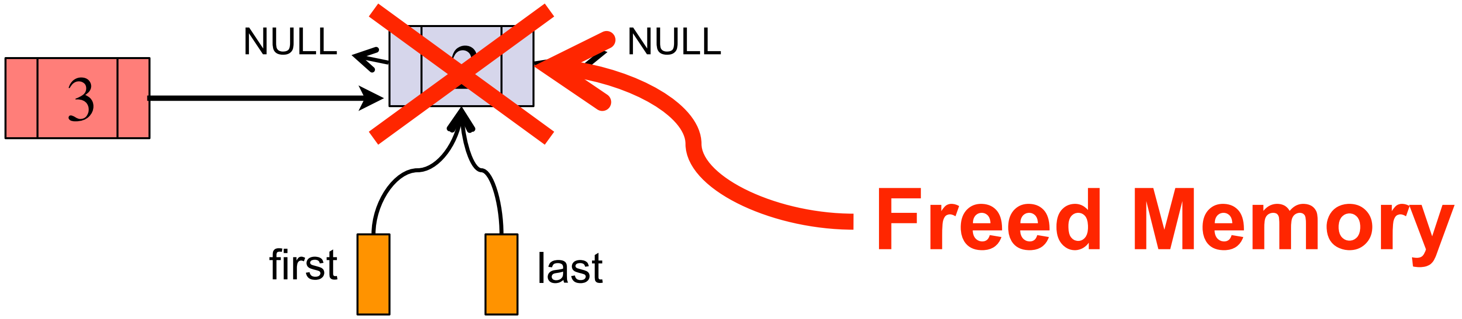
- The Consumer calls remove

```
…
item = q->last->prev;   // returns NULL
free(q->last);
if (item) {
. . .
```

context switch

What bad thing could happen?

- Consider the following queue



- The Producer resumes

```
queue_item_t *item = calloc(...)
item->value = x;
item->next = q->first;
if (item->next)
    item->next->prev = item;
q->first = item;
if (! q->last)
    q->last = item;
```

Freed Memory Access

So what?

- In this example, the producer updates memory that has been de-allocated by another thread!
- In Java we would get an exception once in a while
- C doesn’t zero out or track freed memory and we would get a segmentation fault once in a while
    - A third thread could have done a malloc and be given the memory that has been de-allocated
    - Then the producer could modify the memory used by that third thread for whatever purpose!
    - This could cause a bug in that third thread that could be very difficult to track (because that thread may have nothing to do with the queue!)
    - Basically, if you have threads and you get unexplained segmentation faults, you may have a race condition
        - Even if the segmentation fault occurs in a part of the code that has nothing to do, time- or space-wise, with the relevant part of the code!
- Let’s use locks and fix it!

Simple Solution

```
lock_t lock;  // global variable
```

```
void producer() {
  int x;
  while(1) {
    x = generate();
    lock(lock);
    insert(list,x);
    unlock(lock);
  }
}
```

```
void consumer() {
  int x;
  while(1) {
    lock(lock);
    x = remove(list);
    unlock(lock);
    process(x);
  }
}
```

Simple Solution

```
                lock_t lock;  // global variable

void producer() {                               er() {
  int x;
  while(1) {
    x =
                                                  ove(list);
                                                 nlock(lock);
    u                                             process(x);
  }                                             }
}                                             }
```

While one thread is modifying the queue (inserting or removing), no other thread can insert or remove

Simple Solution

- Important: we use a single lock that is referenced and used by both threads
  - All threads have to wait for the same “toilet”
- The solution is simple: place lock()/unlock() calls around all calls that manipulate the queue
  - Sometimes determining what calls and code segments modify a data structure requires some thought
- The critical section is then the whole queue implementation
- This is the typical strategy when using a non-thread-safe implementation of the queue abstract data type
- To produce a thread-safe implementation of the queue, one needs to create critical sections **within** the queue methods

Thread Safe Queue

```
 struct queue_t {
     queue_item *first;
     queue_item *last;
     lock_t *lock; // each queue has its lock
 };
```

```
struct queue_item_t {
    int value;
    queue_item *prev;
    queue_item *next;
};
```

```
void insert (queue_t q, int x) {
   lock(q.lock);
     queue_item_t *item = (queue_item_t) calloc(1,sizeof(queue_item_t));
     item->value = x;
     item->next = q->first;
     if (item->next)
       item->next->prev = item;
     q->first = item;
     if (! q->last)  q->last = item;
   unlock(q.lock);
}
```

Thread-Safe Queue

```
int remove (queue_t q) {
     queue_item_t *item;
     int x;
     lock(q.lock);
     if (! q->last) return -1;
     x = q->last->value;
     item = q->last->prev;
     free(q->last);
     if (item) {  item->next = NULL;  }
     q->last = item;
     if (q->last == NULL) {
          q->first = NULL;
     }
     unlock(q.lock);
     return x;
}
```

Critical Sections and Performance

- An easy way to make code thread safe is to put a lot of things in critical sections
  - “I don’t really know what these functions do, I’ll use a single lock and put all calls in a critical section”
- **Problem: Critical sections reduce concurrency**
  - Because in a critical section there can be only one thread
- At the extreme, the code becomes purely sequential
  - Great for correctness, but not desired for multi-core performance and/or interactivity
- For better concurrency: make **short** critical sections
  - Use many locks whenever possible to generate many shorter independent critical sections rather than a few longer ones
    - Threads can be in **different** critical sections at the same time
- Goal: one should put only what’s necessary between lock() and unlock()

Better Thread Safe Queue

```
void insert (queue_t q, int x) {
    // lock(q.lock);
    queue_item_t *item = (queue_item_t) calloc(1,sizeof(queue_item_t));
    item->value = x;
    lock(q.lock);
    item->next = q->first;
    if (item->next)
      item->next->prev = item;
    q->first = item;
    if (! q->last)  q->last = item;
   unlock(q.lock);
}
```

taken outside of the CS

A consumer can operate on the queue while a producer is allocating memory for a new element

⇒ **more concurrency**

Good General Principles

- Try to make critical sections as short as possible
- Try to avoid critical sections by replicating or splitting shared data whenever possible
  - It may be that data structures can be reorganized so that threads don't step on each others' toes
  - Example: use two separate counters to avoid the "lost update" problem in our first simple example, and sum them up when both threads have completed
- Let's look at the two versions of code for computing the sum of an array…

Sum Computation

```
// Global variables
lock_t lock;
int sum = 0;
int Array[1000];
```

```
// Thread #1
for (int i = 0; i < 500; i++) {
  lock(lock);
  sum += Array[i];
  unlock(lock);
}
```

```
// Thread #2
for (int i = 500; i < 1000; i++) {
  lock(lock);
  sum += Array[i];
  unlock(lock);
}
```

Version #1

- This code is very sequential
- Only the loop index updates can be done concurrently
- All sum computations are done sequentially
- lock() and unlock() are each called 1000 times, which is bad for performance due to locking overhead

Sum Computation

```
// Global variables
lock_t lock;
int sum = 0;
int Array[1000];
```

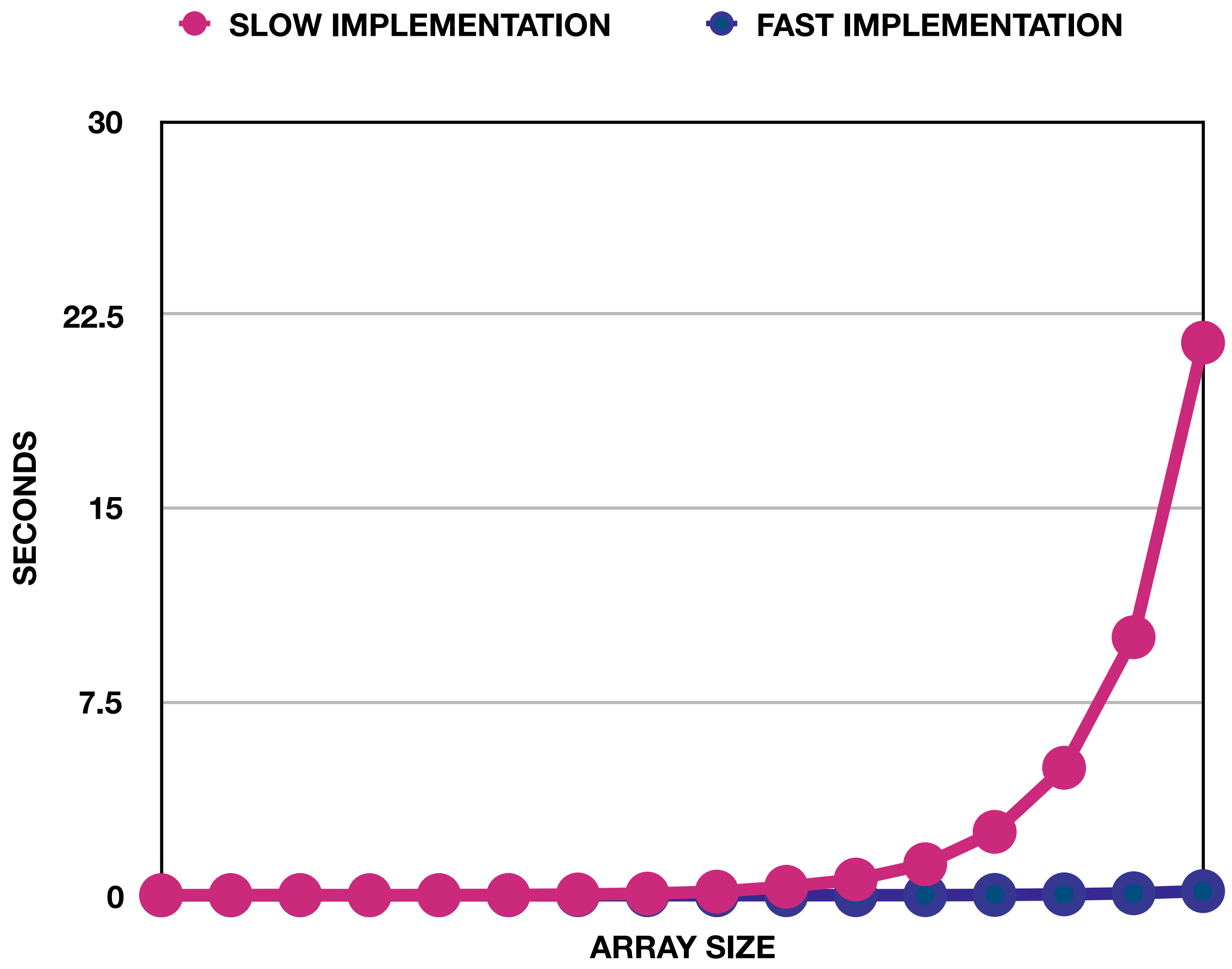
```
// Thread #1
int sum1 = 0;
for (int i = 0; i < 500; i++) {
  sum1 += Array[i];
}
lock(lock)
sum += sum1;
unlock(lock)
```

```
// Thread #2
int sum2 = 0;
for (int i = 500; i < 1000; i++) {
  sum2 += Array[i];
}
lock(lock)
sum += sum2;
unlock(lock)
```

- Almost perfectly concurrent
- Only two additions are done sequentially
- lock() and unlock() are each called only twice

Version #2

Performance Comparison on my Laptop using Java



Good General Principles

- It is often not a good idea to take the sequential code and merely add lock()/unlock() calls around all race-condition-prone statements
- One should rethink/reorganize the code so that making it concurrent is easier and more efficient
    - Or even better, design code with concurrency in mind from the get go
- Mutual exclusion via locks is something you should try to avoid if you can
    - Much later in the semester we'll talk about lock-free concurrency
- If using locks is necessary, then you have to use them sparingly (short critical sections, few critical sections, few locks) and correctly

How Many Locks to Use?

- Try to use different locks for different data items
  - See our queue example, in which we attached a different lock to each queue
- One concern is that with many locks there is more memory consumption and there may be more overhead
- Say you have a 1GB array of 1-byte elements, and 30 threads updating the elements in whatever sequence
  - One lock for the whole array: no concurrency, tiny memory consumption
  - One lock per element: great concurrency, memory more than doubled!
  - One lock per 1K elements: perhaps a good compromise?

Common Misunderstanding

- Note that in the previous slide I wrote: “1 lock per element”
- This may give the wrong impression that one associates a lock to a zone of memory
  - e.g., “you should lock that variable”
  - e.g., “you don’t need any locks for that array”
- When we say “lock variable x” what we mean is: place lock/unlock calls around each statement in the whole code that updates variable x
- If the code is well-designed, then we shouldn’t have update statements for a variable over the whole code
  - e.g., if you code has tons of x++ all over the place, better to have an increment() method inside of which you can place the calls to lock / unlock once and for all

DeadLocks

- **Deadlock**: a common problem when synchronizing threads with *multiple* locks
    - You write your program with many threads and locks, you run it, and at some point, it's stuck
- Deadlock can happen with nested critical sections
- Classic example (which can lead to a deadlock):

```
. . .
lock(lock1)
. . .
lock(lock2)
. . .
unlock(lock2)
. . .
unlock(lock1)
. . .
```

```
. . .
lock(lock2)
. . .
lock(lock1)
. . .
unlock(lock1)
. . .
unlock(lock2)
. . .
```

- See ICS 332 (Operating Systems)

Deadlocks

- The previous example is trivial
- But in practice code can become complex and deadlocks do happen and require careful debugging
    - The calls to lock() and unlock() are not always close to each other in the code
- We will discuss a case-study in a couple of lectures that will highlight many thread synchronization problems, including deadlocks

Re-entrant (Recursive) Locks

- A lock is said to be re-entrant or recursive if a single thread can acquire it multiple times
- Example: lock(A); lock(A); unlock(A); unlock(A)
- If the lock is not re-entrant the above code deadlocks
- Can be convenient for the following idiom (a thread-safe method that calls another thread-safe method):

```
void f() {
  lock(A);
  ...
  g();
  unlock(A);
}
```

```
void g() {
  lock(A);
  ...
  unlock(A);
}
```

- By default, in Java, locks are re-entrant
- In C with PThread locks can be made re-entrant
- Some argue that re-entrant locks are a bad idea

Always Lock()-Unlock() on the same thread!

- The typical usage of locks in a thread is to have a call to lock() followed by a call to unlock()
- In most languages, a thread cannot call unlock() on a lock that thread hasn't acquired first
  - And even if you could, it's considered a HORRIBLE practice
- This is annoyingly "implementation-dependent"
  - Java: "A `Lock` implementation will usually impose restrictions on which thread can release a lock (typically only the holder of the lock can release it) and may throw an (unchecked) exception if the restriction is violated. Any restrictions and the exception type must be documented by that `Lock` implementation."
  - C with Pthreads: "If a thread attempts to unlock a mutex that it has not locked or a mutex which is unlocked, undefined behavior results."
  - But a "good" system should throw an error, and you shouldn't do it

Lock for non-Updates

- We have said that the typical use-case for creating critical sections is to “protect” updates to memory locations
- But there are many times when one needs to put locks around things that seem atomic!
- This is pretty counter-intuitive, and it’s complicated because it depends on what the program does and what its intent is
- Let just see two simple examples….

Critical Section For Non-Update

- Consider the following code fragments assuming that:
  - Thread #1 will call f() once
  - Thread #2 will call g() once
  - These are the only lines of code that reference variable x

```
//global variable
int x=100;
```

```
void f() {
   x = 2;
}
```

```
void g() {
   x++;
}
```

- The only two acceptable outcomes of this program are x=2 or x=3
- The “x++” statement is not atomic and we should protect it with a lock
- But what about the “x=2” statement? It’s atomic so we are fine????
- NO: if we don’t put a lock around “x=2” we could have:
  - Thread #2 reads value 100 from RAM, and gets context-switched out
  - Thread #1 sets x to 2 in RAM
  - Thread #2 is context-switched back in, computes 101, and writes it to RAM
  - We end up with a wrong execution!!

Critical Section For Non-Update

- Consider the following code fragments assuming that:
  - Thread #1 will call f() once
  - Thread #2 will call g() once
  - These are the only lines of code that reference variable x

```
//global variable
int x=100;
lock_t lock1;
```

```
void f() {
   lock(lock1);
   x = 2;
   lock(lock1);
}
```

```
void g() {
   lock(lock1);
   x++;
   lock(lock1);
}
```

- This is one of the reasons concurrency is deemed difficult… it's often a bit counter-intuitive and requires careful thinking
  - A lot of it comes from experience
- Let's look at another example….

Is this Java Stack Thread-Safe?

```
classStack<E> {

  private E[] array =
    (E[]) new Object[SIZE];
  int index= -1;

  threadsafe void push(E val) {
    array[++index] = val;
  }

  threadsafe E pop() {
    return array[index--];
  }

  E peek() {
    return array[index];
  }

}
```

- Here we don’t have explicit lock but assume the language provides a threadsafe keyword
- The idea was to not make the peek() method thread safe, because it just does a memory access, which is atomic
- Is this ok or not?

Is this Java Stack Thread-Safe?

```
classStack<E> {

  private E[] array =
     (E[]) new Object[SIZE];
  int index= -1;

  threadsafe void push(E val) {
    array[++index] = val;
  }

  threadsafe E pop() {
    return array[index--];
  }

  E peek() {
    return array[index];
  }

}
```

- Think about the code for push() in "assembly":

```
mov R1, [index]
inc R1
mov [index], R1
mul R1, 4
mov R2, array
add R2, R1
mov [R2], val
return
```

- The code for peek() is:

```
mov R3, [index]
mul R3, 4
mov R4, array
add R4, R3
return [R4]
```

- The blue code could be interleaved anywhere in the red code! (because peek() is not thread safe)

A Bad Interleaving

```
classStack<E> {

  private E[] array =
    (E[]) new Object[SIZE];
  int index= -1;

  threadsafe void push(E val) {
    array[++index] = val;
  }

  theadsafe E pop() {
    return array[index--];
  }

  E peek() {
    return array[index];
  }

}
```

```
mov R1, [index]
inc R1
mov [index], R1
mov R3, [index]
mul R4, 8
mov R4, array
add R4, R3
return [R4]
mul R1, 8
mov R2, array
add R2, R1
mov [R2], val
return
```

Push() increments index, and is about to put data at array[index]

Peek() returns the value array[index], which is uninitialized!

- We MUST make the peek() method thread safe, even though the code of that method is only "reading data"

Conclusion

- To prevent race conditions one can use locks to create critical sections
- Using locks requires care:
  - Long critical sections reduce concurrency
  - Calling lock()/unlock() has overhead
  - Using too few locks reduces concurrency
  - Using many locks requires memory
  - Using locks can lead to deadlocks
  - Sometimes one need to “lock” a section of code that looks atomic

- Next up: How are locks implemented?
- But before that: Homework Assignment #3
  - A “pencil and paper” assignment