Locks, Contention and Performance

- 7J Object Level Locking
- 7K Bottlenecks and Contention
- 7M Lock-Free Operations

Object Level Locking

- mutexes protect <u>code</u> critical sections
  - brief durations (e.g. nanoseconds, milliseconds)
  - other threads operating on the same data
  - all operating in a single address space
- persistent objects are more difficult
  - critical sections are likely to last much longer
  - many different programs can operate on them
  - may not even be running on a single computer
- solution: lock objects (rather than code)

Whole File Locking

int flock(*fd*, *operation*)

- supported *operation*s:
  - LOCK_SH … shared lock (multiple allowed)
  - LOCK_EX … exclusive lock (one at a time)
  - LOCK_UN … release a lock
- lock is associated with an open file descriptor
  - lock is released when that file descriptor is closed
- locking is purely advisory
  - does not prevent reads, writes, unlinks

Advisory vs Enforced Locking

- <u>Enforced</u> locking
  - done within the implementation of object methods
  - guaranteed to happen, whether or not user wants it
  - may sometimes be too conservative
- <u>Advisory</u> locking
  - a convention that “good guys” are expected to follow
  - users expected to lock object before calling methods
  - gives users flexibility in what to lock, when
  - gives users more freedom to do it wrong (or not at all)
  - mutexes are advisory locks

Ranged File Locking

int lockf(*fd*, *cmd, offset, len*)

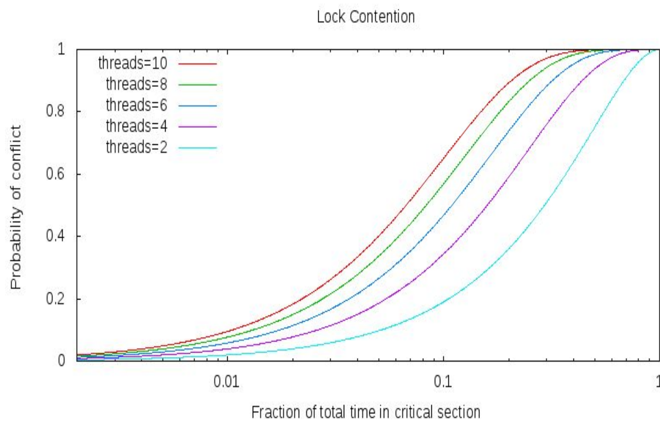
- supported *cmds*:
  - F_LOCK … get/wait for an exclusive lock
  - F_ULOCK … release a lock
  - F_TEST/F_TLOCK … test, or non-blocking request
  - *offset/len* specifies portion of file to be locked
- lock is associated with a file descriptor
  - lock is released when file descriptor is closed
- locking may or may not be enforced
  - depending on the underlying file system

Cost of not getting a Lock

- protect critical sections to ensure correctness
- many critical sections are very brief
  - in and out in a matter of nano-seconds
- blocking is much more (e.g. 1000x) expensive
  - micro-seconds to yield, context switch
  - milliseconds if swapped-out or a queue forms
- performance depends on conflict probability

$C_{expected} = (C_{get} * (1-P_{conflict})) + (C_{block} * P_{conflict})$

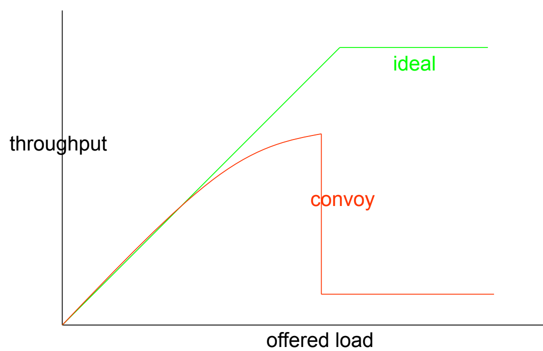
Probability of Conflict



Convoy Formation

- in general
  $P_{conflict} = 1 - (1 - (T_{critical} / T_{total}))^{threads-1}$
  (nobody else in critical section at the same time)
- unless (or until) a FIFO queue forms
  $P_{conflict} = 1 - (1 - ((T_{wait} + T_{critical}) / T_{total}))^{threads}$
  if $T_{wait} >> T_{critical}$, $P_{conflict}$ rises significantly
- if T_{wait} exceeds the mean inter-arrival time
  the line becomes permanent, parallelism ceases,
  (cheap) $T_{critical}$ is replaced by (expensive) T_{wait}

Performance: resource convoys



Bottlenecks vs. Convoys

- Both involve reduced parallelism
  - many threads waiting for a particular resource
- Bottleneck … problem is the resource
  - the resource is saturated
  - resource throughput limits system throughput
- Convoys … problem is the queue
  - the resource may not be highly utilized
  - precipitated by preemption in critical section
  - line persists due to mandatory FIFO queuing

Contention Reduction

- eliminate the critical section entirely
  - eliminate shared resource, use atomic instructions
- eliminate preemption during critical section
  - by disabling interrupts … not always an option
  - avoid resource allocation within critical section
- reduce time spent in critical section
  - reduce amount of code in critical section
- reduce frequency of critical section entry
  - reduce use of the serialized resource
  - reduce exclusive use of the serialized resource
  - spread requests out over more resources

Reducing Time in Critical Section

- eliminate potentially blocking operations
  - allocate required memory before taking lock
  - do I/O before taking or after releasing lock
- minimize code inside the critical section
  - only code that is subject to destructive races
  - move all other code out of the critical section
  - especially calls to other routines
- cost: this may complicate the code
  - unnaturally separating parts of a single operation

Reduce Time or Preemption

```
int List_Insert(list_t *l, int key) {
        pthread_mutex_lock(&l->lock);
        node_t new = (node_t*)
      malloc(sizeof(node_t));
        if (new == NULL) {
                perror("malloc");
                pthread_mutex_unlock(&l->lock);
                return(-1);
        }
        new->key = key;
        new->next = l->head;
        l->head = new;
        pthread_mutex_unlock(&l->lock);
        return 0;
}
```

```
int List_Insert(list_t *l, int key) {
        node_t new = (node_t*)
      malloc(sizeof(node_t));
        if (new == NULL) {
                perror("malloc");
                return(-1);
        }
        new->key = key;
        pthread_mutex_lock(&l->lock);
        new->next = l->head;
        l->head = new;
        pthread_mutex_unlock(&l->lock);
        return 0;
}
```


Reduced Use of Critical Section

- can we use critical section less often
  - less use of high-contention resource/operations
  - batch operations
- consider "sloppy counters"
  - move most updates to a private resource
  - costs:
    - global counter is not always up-to-date
    - thread failure could lose many updates
  - alternative:
    - sum single-writer private counters when needed

Non-Exclusivity: read/write locks

- reads and writes are not equally common
  - file read/write: reads/writes > 50
  - directory search/create: reads/writes > 1000
- only writers require exclusive access
- read/write locks
  - allow many readers to share a resource
  - only enforce exclusivity when a writer is active
  - policy: when are writers allowed in?
    - potential starvation if writers must wait for readers

Spreading requests: lock granularity

- coarse grained - one lock for many objects
  - simpler, and more idiot-proof
  - greater resource contention (threads/resource)
- fine grained - one lock per object (or sub-pool)
  - spreading activity over many locks reduces contention
  - dividing resources into pools shortens searches
  - a few operations may lock multiple objects/pools
- TANSTAAFL
  - time/space overhead, more locks, more gets/releases
  - error-prone: harder to decide what to lock when

Partitioned Hash Table

```
int Hash_Insert(hash_t *h, int key) {
        int bucket = key % h->num_buckets;
        list_t *l = &h->lists[bucket];
        return List_Insert(l, key);
}
```

- Each list_t is still protected by a lock
  - but contention has been greatly reduced
- Partitioning function must be race-free
  - no critical-section to protect
  - per partition load depends on request randomness

Non-Blocking Single Reader/Writer

```
int SPSC_put(SPSC *fifo, unsigned char c) {
        if (SPSC_bytesIn(fifo) == fifo->full)
                return(-1);
        *(fifo->write) = c;
        if (fifo->write == fifo->wrap)
                fifo->write = fifo->start;
        else
                fifo->write++;
        return( c );
}
```

```
int SPSC_get(SPSC *fifo) {
        if (SPSC_bytesIn(fifo) == 0)
                return(-1);
        int ret = *(fifo->read);
        if (fifo->read == fifo->wrap)
                fifo->read = fifo->start;
        else
                fifo->read++;
        return(ret);
}
```

```
int SPSC_bytesIn(SPSC *fifo) {
        return(fifo->write >= fifo->read ?
                fifo->write – fifo->read :
                fifo->full – (fifo->read – fifo->write));
}
```

Atomic Instructions – Compare & Swap

```
/*
 * Concept: Atomic Compare and Swap
  * this is implemented in hardware, not code
 */
int CompareAndSwap( int *ptr, int expected, int new) {
     int actual = *ptr;
     if (actual == expected)
          *ptr = new;
     return( actual );
}
```


Solving the checkbook problem

```
int current_balance;
Writecheck( int amount ) {
  int oldbal, newbal;
  do {
    oldbal = current_balance;
    newbal = oldbal - amount;
    if (newbal <0) return (ERROR);
  } while (!compare_and_swap( &current_balance, oldbal, newbal))
...
}
```

Lock-Free Multi-Writer

```
// push an element on to a singly linked LIFO list
void SLL_push(SLL *head, SLL *element) {
     do {
          SLL *prev = head->next;
          element->next = prev;
     } while ( CompareAndSwap(&head->next, prev, element) != prev);
}
```


Spin Locks vs Atomic Updates

```
void SLL_push(SLL *head, SLL *element) {
     do {
          SLL *prev = head->next;
          element->next = prev;
     } while ( CompareAndSwap(&head->next, prev,  element) != prev);
}
```

```
DLL_insert(DLL *head, DLL*element) {
     while(TestAndSet(lock,1) == 1);
          DLL *last = head->prev;
          element->prev = last;
          element->next = head;
          last->next = element;
          head->prev = element;
     lock = 0;
}
```


(Spin Locks vs Atomic Update Loops)

- both involve spinning on an atomic update
  - but they are not the same
- a spin-lock
  - spins until the lock is released
  - which could take a very long time
- an atomic update loop
  - spins until there is no conflict during the update
  - impossible to be preempted holding lock
  - conflicting updates are actually very rare

Evaluating Lock-Free Operations

- Effectiveness/Correctness
  - effective against all conflicting updates
  - cannot be used for complex critical sections
- Progress
  - no possibility of deadlock or convoy
- Fairness
  - small possibility of brief spins
- Performance
  - expensive instructions, but cheaper than syscalls

Reading and Assignments

Reading:

- Reiher: Measuring Operating System Performance
- Kampe: Load and Stress Testing

Projects:

- start looking at project 2B (lock granularity & performance)

Supplementary Slides

Performance: lock contention

- The riddle of parallelism:
  - parallelism: if one task is blocked, CPU runs another
  - concurrent use of shared resources is difficult
  - critical sections serialize tasks, eliminating parallelism
- What if everyone needs to use one resource?
  - one process gets the resource
  - other processes get in line behind him (convoy)
  - parallelism is eliminated; B runs after A finishes
  - that resource becomes a *bottle-neck*

Performance of Locking

- Locking typically performed as an OS system call
  - Particularly for enforced locking
- Typical system call overheads for lock operations
- If they are called frequently, high overheads
- Even if not in OS, extra instructions run to lock and unlock

Eliminating Critical Sections

- Eliminate shared resource
  - Give everyone their own copy
  - Find a way to do your work without it
- Use atomic instructions
  - Only possible for simple operations
- Great when you can do it
- But often you can't

Locking Costs

- Locking called when you need to protect critical sections to ensure correctness
- Many critical sections are very brief
  - In and out in a matter of nano-seconds
- Overhead of the locking operation may be much higher than time spent in critical section

Solutions that do work

- avoid shared data whenever possible
- eliminate critical sections w/atomic instructions
  - atomic (uninteruptable) read/modify/write operations
  - can be applied to 1-8 contiguous bytes
  - simple: increment/decrement, and/or/xor
  - complex: test-and-set, exchange, compare-and-swap
- use atomic instructions to implement locks
  - use the lock operations to protect critical sections

Limitations of atomic instructions

- only update a small number of contiguous bytes
  - cannot be used to atomically change multiple locations (e.g. insertions in a doubly-linked list)
- they operate on a single memory bus
  - cannot be used to update records on disk
  - cannot be used across a network
  - <u>lock-out</u> and <u>synchronized write are very expensive</u>
- they are not higher level locking operations
  - they cannot "wait" until a resource becomes available