14-736: DISTRIBUTED SYSTEMS

LECTURE 4 * FEBRUARY 5, 2018 * SPRING 2018 (KESDEN)

NOT DISTRIBUTED SYSTEMS

- Today is a review of what you may have learned in a prior systems class
- It has nothing to do with distributed systems
- I’m wedging it in here, because it is good for you to know
- It is also important to know how distributed techniques compare
    - Distributed synchronization is our next topic

CONCURRENCY

- It doesn't matter how it arises:
  - True parallelism (multi-core, multi-processor)
  - Scheduler interleaving
  - Processes, Threads
- Sharing is dangerous!
  - Classing "Missed update" problem
  - Read-Update-Write, Read-Update-Write Interleaving
- Vocabulary:
  - Critical resource – shared resource that can't naturally be safely shared (without help)
  - Critical section – code that manipulates the critical resource while being shared

EXAMPLE FROM 213/513/600

C code for counter loop in thread i

```
for (i = 0; i < niters; i++)
    cnt++;
```

Asm code for thread i

```
    movq  (%rdi), %rcx
    testq %rcx,%rcx
    jle   .L2
    movl  $0, %eax
.L3:
    movq  cnt(%rip),%rdx
    addq  $1, %rdx
    movq  %rdx, cnt(%rip)
    addq  $1, %rax
    cmpq  %rcx, %rax
    jne   .L3
.L2:
```

H_i : **Head**

L_i : **Load cnt**

U_i : **Update cnt**

S_i : **Store cnt**

T_i : **Tail**

EXAMPLE FROM 213/513/600

- ***Key idea:*** **In general, any sequentially consistent interleaving is possible, but some give an unexpected result!**

- I_i denotes that thread i executes instruction I
- $\%rdx_i$ is the content of %rdx in thread i's context

i (thread)	$instr_i$	$\%rdx_1$	$\%rdx_2$	cnt
1	H_1	-	-	0
1	L_1	0	-	0
1	U_1	1	-	0
1	S_1	1	-	1
2	H_2	-	-	1
2	L_2	-	1	1
2	U_2	-	2	1
2	S_2	-	2	2
2	T_2	-	2	2
1	T_1	1	-	2

OK

Thread 1
critical section

Thread 2
critical section

EXAMPLE FROM 213/513/600

- **Incorrect ordering: two threads increment the counter, but the result is 1 instead of 2**

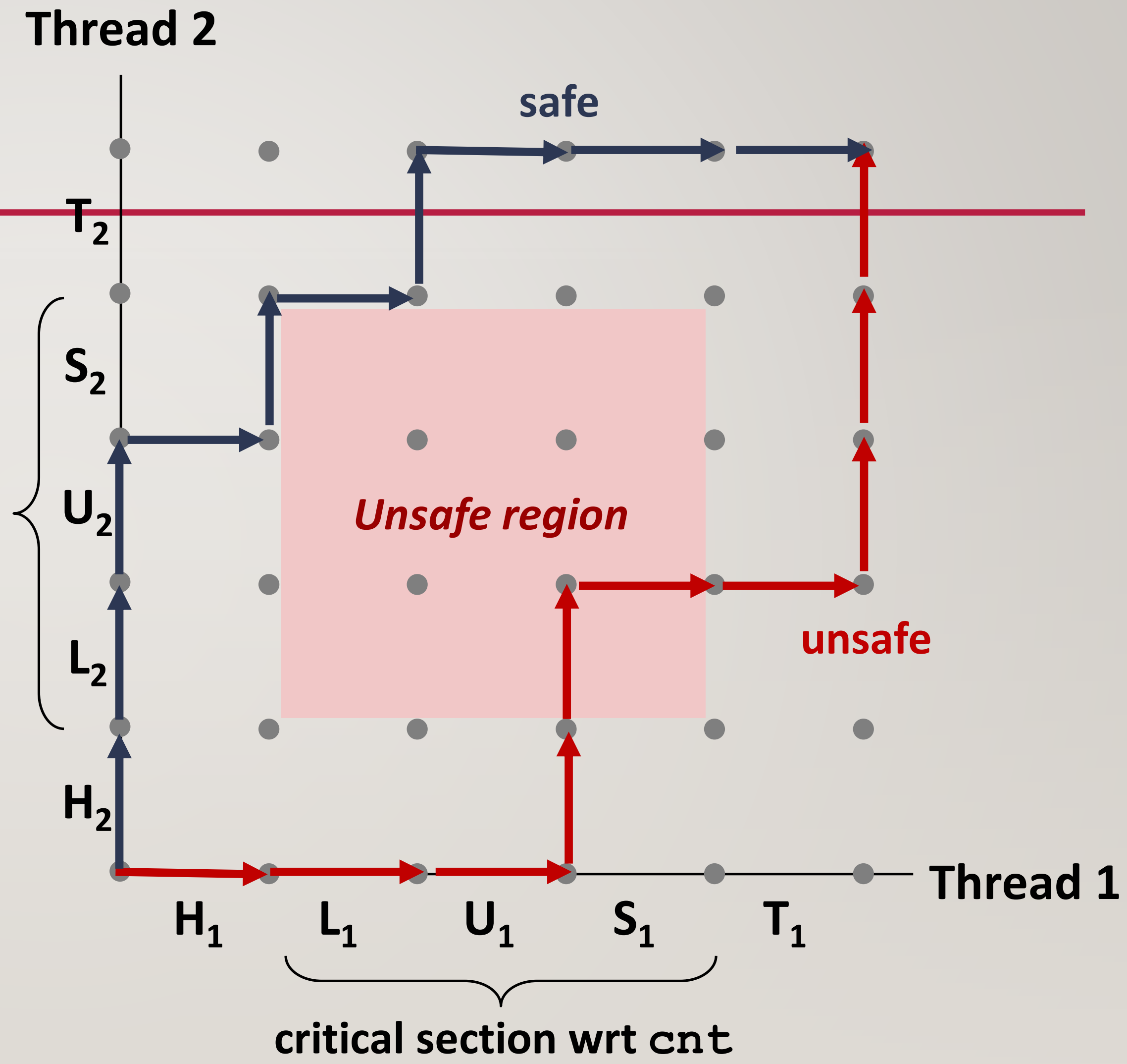
i (thread)	$instr_i$	$\%rdx_1$	$\%rdx_2$	cnt
1	H_1	-	-	0
1	L_1	0	-	0
1	U_1	1	-	0
2	H_2	-	-	0
2	L_2	-	0	0
1	S_1	1	-	1
1	T_1	1	-	1
2	U_2	-	1	1
2	S_2	-	1	1
2	T_2	-	1	1

Oops!

VISUALIZING CONCURRENCY

Def: A trajectory is *safe* iff it does not enter any unsafe region

Claim: A trajectory is correct (wrt `cnt`) iff it is safe



CLASSIC MECHANISM

- Atomic compare-and-swap/test-and-set instructions
  - Provides value and mutates
  - Lock: while (CS<1,x>);
  - Unlock: x = 0
  - Classic spin-lock
- Requires locking memory bus on multi-core/multi-processor
- Busy wait not good for high contention
  - But good for low contention

FUTEX – OPTIMIZATION IN LINUX

- Much like classic spin-lock, but less spinning.
- Shared memory is in user space
- Queues in kernel space if locked
- Decreases spin time
- Tricky to make robust
- Take 15-605 for details

OTHER TECHNIQUES: AVOID CONCURRENCY

- Raise kernel interrupt level (for kernel code)
    - Very costly
- Block processes or threads in schedule
    - Great for threads, user-space things
    - Still takes lower level concurrency control to protect queues, etc.

MUTUAL EXCLUSION

- Policy of one user at a time among many
- A single user of critical section excludes other by mutual agreement
  - What is the mutual agreement? Use of synchronization primitive in code
- Spin locks, futexes classically wrapped in primitives
  - mutex_acquire()/mutex_lock()
  - mutex_release()/ mutex_unlock()

OTHER DISCIPLINES: AT MOST N

- Mutual exclusion isn’t the only policy.
  - At-Most-N is common
  - N buffers available, etc
- Enter “Semaphores”
  - P operation: Wait for resource to be available
    - and decrement internal count when caller gets it
  - V operation: Make resource available
    - and increment internal count of resources
    - also “wake up” anyone waiting
  - No “Peek” (can’t be useful)
- Not necessarily FCFC, unless guaranteed
  - Internal queuing may use a spin lock

SEMPAPHORE IMPLEMENTATION: WITH MUTEXES

```
P (csem) {
      while (1)  {
         acquire_mutex (csem.mutex);
         if (csem.value <= 0) {
            release_mutex (csem.mutex);
            continue;
         }
         else {
             csem.value = csem.value – 1;
             release_mutex (csem.mutex);
             break;
         }
      }
   }
```

```
V (csem)
   {
       acquire_mutex (csem.mutex);
       csem.value = csem.value + 1;
       release_mutex (csem.mutex);
   }
```

SEMAPHORE IMPLEMENTATION: HELP FROM OS/THREAD SCHEDULER

- Deschedule (move to blocked queue) blocked thread or process
  - Then there is no spinning
- Reschedule (move to ready queue) when resource available
  - Will need to recheck availability, of course
- Much more efficient
- But, doesn't eliminate lower level blocking
  - Just moves it to the queue
  - Queue operations are fast, so lower contention, so less blocking

PRIMITIVES SO FAR: MUTEXES, SEMPAHORES

- Mutexes/Futexes
  - Enforce mutual exclusion
  - Can be used to develop more complex policies based upon mutually exclusive access to state variables
    - Consider semaphore implementation
  - Conceptually "lock and unlock" one or more associated resources.
- Semaphores
  - At-Most-N policies
    - At-Most-1 is a special case usage, "Binary Semaphore", and is like a mutex.
  - Conceptually, manages a pool of resources.

CONDITION VARIABLES

- Used to block while waiting for an event
  - Don’t count resources like semaphores
  - Don’t lock like mutexes
  - No predicate
  - Wait for event
- Operations:
  - Wait – always waits. No predicate
  - Signal – Wake up a waiter
  - Broadcast – Wake up all waiters.
- Uses:
  - Other synchronization primitives
  - Waiting for things like buffers or pages of memory

CONDITION VARIABLES: ADDING A PREDICATE

- A wait that always waits is known as a sleep.
  - Not very useful for concurrency control
  - CV waits work with mutexes to enable a predicate

```
void wait (condition *cv, mutex *mx)
 {
   mutex_acquire(&c->listLock);  /* protect the queue */
   enqueue (&c->next, &c->prev, thr_self()); /* enqueue */
   mutex_release (&c->listLock); /* we're done with the list */

   /* The suspend and release_mutex() operation should be atomic */
   release_mutex (mx));
   thr_suspend (self);  /* Sleep 'til someone wakes us */

   mutex_acquire (mx); /* Woke up -- our turn, get resource lock */

   return;
 }
```

CONDITION VARIABLES: SIGNAL IMPLEMENTATION

```
void signal (condition *c)
  {
    thread_id tid;

    mutex_acquire (c->listlock); /* protect the queue */
    tid = dequeue(&c->next, &c->prev);
    mutex_release (listLock);

    if (tid>0)
      thr_continue (tid);

    return;
  }
```

CONDITION VARIABLES: EXAMPLE LOCK FROM CVS

```
spin_lock s;

GetLock (condition cv, mutex mx)
{
  mutex_acquire (mx);
  while (LOCKED)
    wait (c, mx);

  lock=LOCKED;
  mutex_release (mx);
```

```
ReleaseLock (condition cv, mutex mx)
{
  mutex_acquire (mx); /* Prevent lost wake-up */
    lock = UNLOCKED;
    signal (cv);
  mutex_release (mx);
}
```

SEMAPHORES FROM CONDITION VARIABLES

```
void semP (sem *s)
{
  mutex_acquire (s->mutex);
  while (s->count < 1)
      cond_wait(s->cv, s->mutex);
  s->count--;
  mutex_release (s->mutex);
}

void semV(sem *s) {
        mutex_acquire (s->mutex); /* Prevent lost wake-up */
        cond_signal(s->cv);
        mutex_release (s->mutex);
}
```

CONDITION VARIABLES

- Very common use is in implementation of "Monitors"
  - A high-level protected box for critical
  - Only one of associated methods can run at a time
  - Scheduling has to account for blocking within methods, etc.
  - 15-605 for details
- Common example of a Monitor paradigm
  - Java synchronized methods

PRIMITIVES

- Mutexes – Mutual Exclusion
- Semaphores – Pools of equivalent resources
- Condition Variables -- Events

COMMON REQUIREMENT

- Shared memory with atomic instructions
- Global scheduler managing concurrency for efficiency
  - Parallelism and interleaving
- Do we have these in distributed systems?
  - Can we get them?