4g Cache Coherence

There are two critical issues when multithreaded processes share memory resources.

1. Memory **Consistency**
   - Concerns apparent ordering of reads and writes to memory as viewed from different cores (e.g. memory ordering model). We discusses this topic when we talked about sequential consistency and how multithread programs can rely on this property in some cases and fail when it is not provided.
2. **Cache Coherence**
   - Concerns multiple cores accessing the same memory location (via their caches).
   - All cores should see the same value in their caches.
   - The system must ensures that **stale** (out-of-date) copies do not cause problems.

Multicore Caches

- Performance requires multilevel caches (L1, L2, etc.) since access to RAM is relatively slow.
- Caches create an opportunity for cores to disagree about what value is stored at a particular address.
- Bus-based approaches (older)
  - "Snoopy" protocols, each core listens to memory bus.
  - The core writing the value uses write through caching and the other cores invalidate their values for that address when the see a write to the same address from another core.
  - Problem: *Bus-based schemes limit scalability*.

Modern processors use networks such as Infinity Fabric (AMD) or UPI (Intel).

- Caches are divided into slices of bytes called **cache lines**.
- This is the amount of data (64 bytes on x64 and Apple M1 – M3) that is transferred to different caches or main memory to *take advantage of locality of reference.*

3-state Coherence Protocol: Modified, Shared, Invalid (MSI)

- Each cache line is one of three states (other versions have more states):
- **Shared**
  - One or more caches (and memory) have a valid copy.
- **Modified** (sometimes called Exclusive)
  - One cache has a valid copy.
  - That copy is called **dirty** (needs to be written back to main memory).
  - Out-of-date copies in other caches are called **stale**.
  - Must invalidate all other copies before entering this state.
- **Invalid**
  - The data is either not present in cache or is stale and must updated.
- Transitions can take 100 (on smaller machines) to 2000 clock cycles on larger ones.

Core and Bus Actions

- Each core has three possible actions that affect the caches.
- Read (load)
  - *Read without intent to modify*, data can come from memory or another cache.
  - Cache line enters shared state.
- Write (store)
  - *Read with intent to modify* must invalidate all other cache copies if cache line is shared with others.
- Evict
  - *Writeback contents to memory if it is in the modified state*.
  - Discard if it is in the shared state. Save other copy if other copy becomes dirty.
- Performance problem:
  - Every transition requires communicating with other cores.
  - Avoid state transitions whenever possible.

Implications for Multithreaded Design

- Lesson #1: *Avoid false sharing.*
  - Processor shares data in cache line chunks
  - Avoid placing data used by different threads in the same cache line, e.g. make arrays start and stop on a 64B boundry.
- Lesson #2: *Align structures to cache lines.*
  - Place related data you need to access together.
  - Alignment in C11/C++11: `alignas(64) struct foo f;`
- Lesson #3: *Pad data structures*
  - Arrays of structures lead to false sharing.
  - Add unused fields to ensure alignment.
- Lesson #4: *Avoid contending on cache lines*
  - Reduce costly cache coherence traffic.
  - Advanced algorithms spin on a cache line local to a core (e.g., MCS Locks).

4h Deadlock

```
mutex_t m1, m2;

void f1(void *ignored) {
  lock(m1);
  lock(m2);
  /* critical section */
  unlock(m2);
  unlock (m1);
}
```

```
void f2 (void *ignored) {
  lock(m2);
  lock(m1);
  /* critical section */
  unlock(m1);
  unlock(m2);
}
```

- *It is dangerous to acquire locks in different orders:* if T1 executes `f1` exactly when T2 executes `f2` then T1 will wait for T2 to release `m2` and T2 will wait for T1 to release `m1`. This situation is called **deadlock**, i.e. both threads will wait for ever.

More deadlocks

- The same problem can occur with *condition variables*.
  - Suppose resource 1 managed by c_1, resource 2 by c_2
  - T1 has 1, waits on c_2, T2 has 2, waits on c_1.
- Or have combined *mutex/condition variable* deadlock:

```
mutex_t a, b;
cond_t c;

lock(a); lock(b); while (!ready) wait(c, b); // releases b but not a
unlock(b); unlock (a);

lock(a); lock(b); ready = true; signal(c); // needs a to signal c
unlock(b); unlock(a);
```

- Lesson: Dangerous to hold locks when *crossing abstraction barriers*.
  - I.e., top level function holds lock `a` then calls a lower level function that uses condition variables and needs lock `a` and `b` to signal a thread to wake up.

Deadlock conditions

All four of these conditions are necessary for deadlock to occur.

1. *Limited access* (mutual exclusion):
   Resource can only be shared with a finite number of threads.
2. *No preemption of resources:*
   Once resource granted, the resouces cannot be taken away.
3. Multiple independent requests (*hold and wait*):
   Don't ask all at once, i.e. do not wait for next resource while holding current one.
4. *Circularity in graph of requests*

There are two approaches to dealing with deadlock:

- Pro-active: prevention
- Reactive: detection + corrective action

Prevent by eliminating one condition

1. Limited access (mutual exclusion):
   - Buy more resources, split into pieces, or virtualize to make "infinite" copies.
2. No preemption:
   - Physical memory: virtualized with VM, can take physical page away and give to another process!
3. Multiple independent requests (hold and wait):
   - Wait on all resources at once (must know in advance), i.e. do not hold and wait.
4. *Circularity in graph of requests*
   - Single lock for entire system: (problems?)
   - Partial ordering of resources (next)...

Cycles and deadlock

- *View the system as graph.*
  - Processes, threads, and resources are nodes.
  - Resource requests and assignments are edges.
- *If the graph has no cycles → no deadlock.*
- If graph contains a cycle,
  - it is definitely deadlock if only one instance per resource type, e.g. one mutex `m1`.
  - Otherwise, maybe deadlock, maybe not, e.g. multiple pages in RAM.
- Prevent deadlock with *partial order on resources.*
  - E.g., always acquire mutexes in a specific order, e.g. `m1` before `m2`
  - Statically assert (decide on a) lock ordering for the whole code base (e.g., VMware ESX)
  - There are libraries routines that dynamically find potential deadlocks, e.g. [Witness]

Wait Channels

4i Sychronization Implementation

Mutexes, semaphores, and condition variables (but not spinlocks) use wait channels.

- They *manage a list of sleeping threads*. Each mutex, etc, gets its own wait channel.
- Works with thread scheduler, i.e. threads waiting for mutex `m1` wait on `m1`'s wait channel.
- `void WaitChannel_Lock(WaitChannel *wc);`
  Lock wait channel for sleep and wake operations.
  Prevents a race between sleep and wake operations.
- `void WaitChannel_Sleep(WaitChannel *wc);`
  Blocks calling thread on wait channnel `wc`, i.e. puts it to sleep.
  Causes a context switch (e.g., `thread_yield`).
- `void WaitChannel_WakeAll(WaitChannel *wc);`
  Unblocks all threads sleeping on the wait channel
- `void WaitChannel_Wake(WaitChannel *wc);`
  Unblocks one thread sleeping on the wait channel

Hand-over-Hand Locking

- **Hand-over-hand locking** allows for fine-grained locking, i.e. many locks.
- Useful for concurrent access to a single data structure.
  - Hold at most two locks: the previous lock and the next one: working your way through a sequence of steps.
  - Locks must be ordered in a sequence.
- Example: we have locks A, B, C

```
lock(A)
// Do step 1.
lock(B)
unlock(A)
// Do step 2.
lock(C)
unlock(B)
// Do step 3.
unlock(C)
```

Example Semaphores

- Mutexes, CVs and Semaphores use wait channels and hand-over-hand locking.
- Lock order:
  - first acquire `Spinlock sem_lock`,
  - then `WaitChannel_Lock`,
  - then release `Spinlock sem_lock`.
- While the spinlock `sem_lock` is being held, no other thread can access this struct.
- After `WaitChannel_Lock` is called, no other thread can access this wait channel until the calling thread is put to sleep and the kernel releases the WaitChannel's lock.

```
typedef struct Semaphore {
    int         sem_count;
    Spinlock    *sem_lock;    // exclusive access to this struct's fields
    WaitChannel *sem_wchan; // queue where blocked threads wait (i.e. sleep)
} Semaphore;
```

Example: Semaphores Implementation

```
Semaphore_Wait(Semaphore *sem) {
  Spinlock_Lock(&sem->sem_lock);
  while (sem->sem_count == 0) {
    /* Locking the wchan prevents a race on sleep */
    WaitChannel_Lock(sem->sem_wchan);
    /* Release spinlock before sleeping */
    Spinlock_Unlock(&sem->sem_lock);
    /* Wait channel protected by it's own lock */
    WaitChannel_Sleep(sem->sem_wchan);
    /* Recheck condition, no locks held */
    Spinlock_Lock(&sem->sem_lock);
  }
  sem->sem_count--;
  Spinlock_Unlock(&sem->sem_lock);
}
```

Decrement the semaphore count xor block if it is 0.

Example: Semaphores Implementation

```
Semaphore_Post(Semaphore *sem) {
  Spinlock_Lock(&sem->sem_lock);
  sem->sem_count++;
  WaitChannel_Wake(sem->sem_wchan);
  Spinlock_Unlock(&sem->sem_lock);
}
```

- *Increment the semaphore count and wake one thread if it is blocked.*
- The actual spinlock the wait channel uses is abstracted (hidden away) in the calls to `WaitChannel_Lock`, `WaitChannel_Sleep`, `WaitChannel_Wake`, and `WaitChannel_WakeAll`.