Synchronization

Ravi Pudipeddi

Revisiting cmpxchg

- Compare exchange:
-
```
int64 InterlockedCompareExchange(int64 *p, int64 Value, int64 newValue){
       temp = *p;
       if (*p == oldValue) *p = newValue;
     return *p;
  }
```
- Typically used in a loop until exchange succeeds:
-
```
while (InterlockedCompareExchange(&lock, 0, 1) != 0));  // acquire
 InterlockedCompareExchange(&lock, 1, 0); // release
```
- InterlockedIncrement(), InterlockedDecrement()
- Solves the 'ABA' problem where a value is read twice and assumed to be the same between reads.

Cache awareness in locks

- Processors have caches – known as L1 cache which is a cache of the RAM divided into ‘lines’ which are say 64 bytes for core i7.
- When say an integer is accessed by a CPU, the 64-byte aligned line is brought into the CPU cache
- Modifying a variable will cause cache flushes of the lines in other CPUs which may have the line cached.
- Locks tend to cause cache flushes on acquire (succeeding) and release

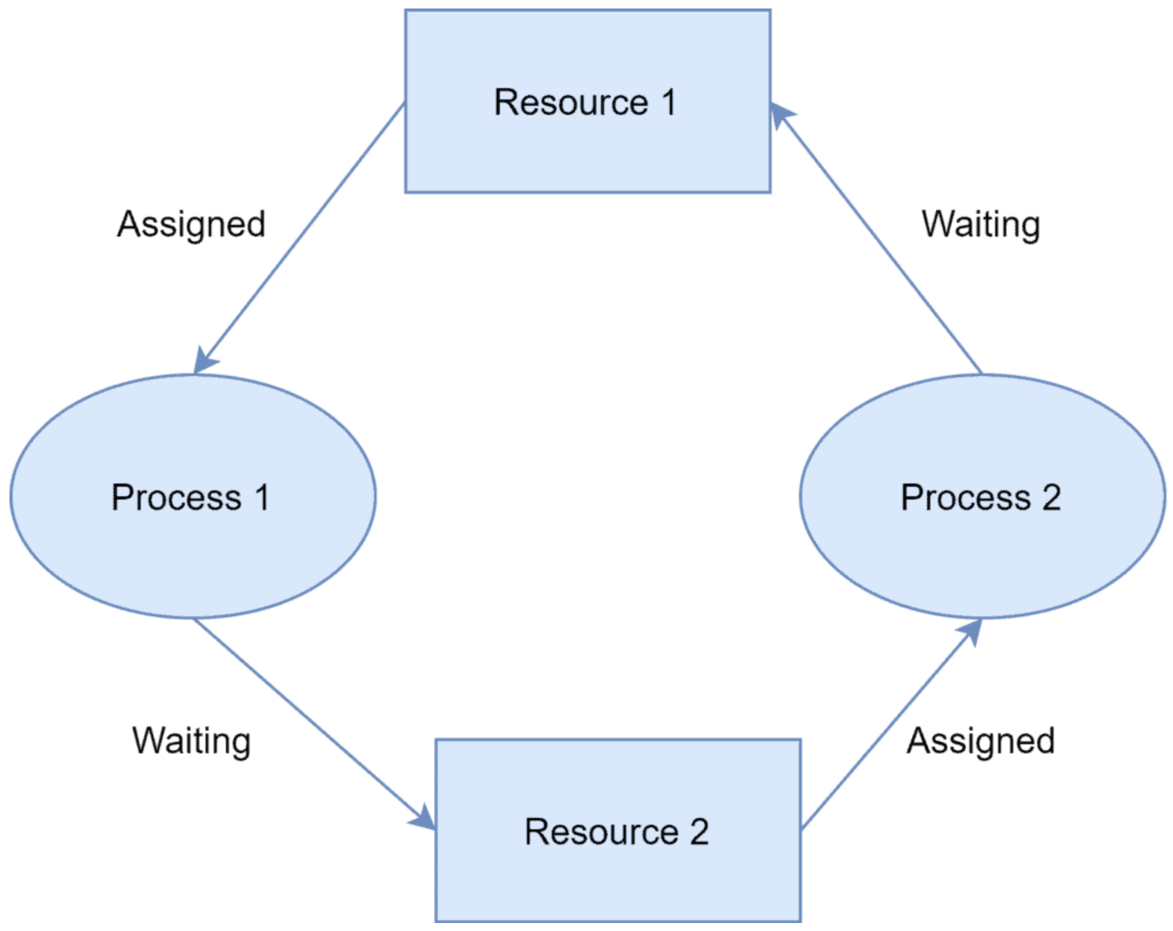
Re-entrant locks

- Threads in kernel – or user – may try to acquire a lock which they hold.
- Simple recursion or mutual recursion can lead to this
- File systems tend to use re-entrant reader/writer locks extensively
    - Fopen() -> CreateFile() -> CreateFile() -> ReadFile() - can have a number of reader locks acquired recursively. Writing can have re-entrant exclusive locks.
- Re-entrant locks need to be used carefully as number of releases need to tally with number of acquires

Deadlocks

- Deadlock example:
  - Thread 1 acquires lock A
  - Thread 2 acquires lock B
  - Thread 1 attempts to acquire B
  - Thread 2 attempts to acquire A
- 3-way dead locks
- Priority inversion: thread A is high priority waiting for lock L, thread B is middle priority ready to run and thread C is low priority holding lock L. B will not let C run thereby preventing A from running
- Boosting C when it holds the lock is a common solution

Deadlock example



Priority inversion example

- Outlook (email client) – has a low background priority scanner to index mail contents for search in the background, but needs access to file and locks it
- Normal priority user process doing computationally intensive operations – say image processing
- Shell (explorer) – foreground priority trying to access file locked by Outlook indexer – leads to UI hang

Dead lock – circular waiting

- Always acquire resources (locks) in the same order in every process
- In some cases if a resource cannot be obtained process can release all existing resources so that other processes can make progress
- With non blocking locks always finish computation fast

Live locks

- Repeatedly acquiring and release locks if another resource cannot be acquired.
- Loops where one resource is acquired, another is attempted and on failure, the original one is released
- Usually happens processes are trying to backoff acquisition and

Spin counts for blocking locks

- Blocking is an expensive operation: taking off ready queue, and getting rescheduled when lock is available – context switch cost.
- Most blocking locks now allow for a ‘spincount’ to be supplied in terms of number of iterations
- Lock() will spin for lock availability for specified iterations before block.
- For high contention shortly held locks (databases for instance) this avoids unnecessary context overhead trading off with slightly underutilized cpu

Windows pushlocks

- Non re-entrant locks (cannot be recursively acquired)
- Light weight – the lock is pointer-length,
- Uses interlocked compare exchange to implement
- Reader/writer semantics
- Spins instead of blocking – equivalent of the reader/writer blocking lock, excepting it is lighter weight and used for computation only paths
- No guarantee of fairness in acquisition of exclusive lock
- Cache aware versions use lock value per processor
- User mode versions are called ‘SRW locks’

Rundown protection

- Uses a simple 4-byte or 8-byte integer – where lsb indicates wait is active ((refcount & 1) == 1)
- Object reference counting – used for process rundown when all references go to zero
  - Acquire(&refcount) → If low bit is 0, increments ref count by 2 (to keep low bit 0) if no wait in progress, otherwise returns error.
  - Release(&refcount) → decrements ref count by 2
  - Wait(&refcount) → Sets low bit of ref to 1 so no new acquires are granted and blocks until it is zero, so wait is guaranteed to terminate.
- Every time an object (say a process o object or PCB is accessed) – acquire called to protect against process destruction
- Wait called when the process is ‘killed’ or exiting, to ensure it is not destroyed before all references are gone

Example wait

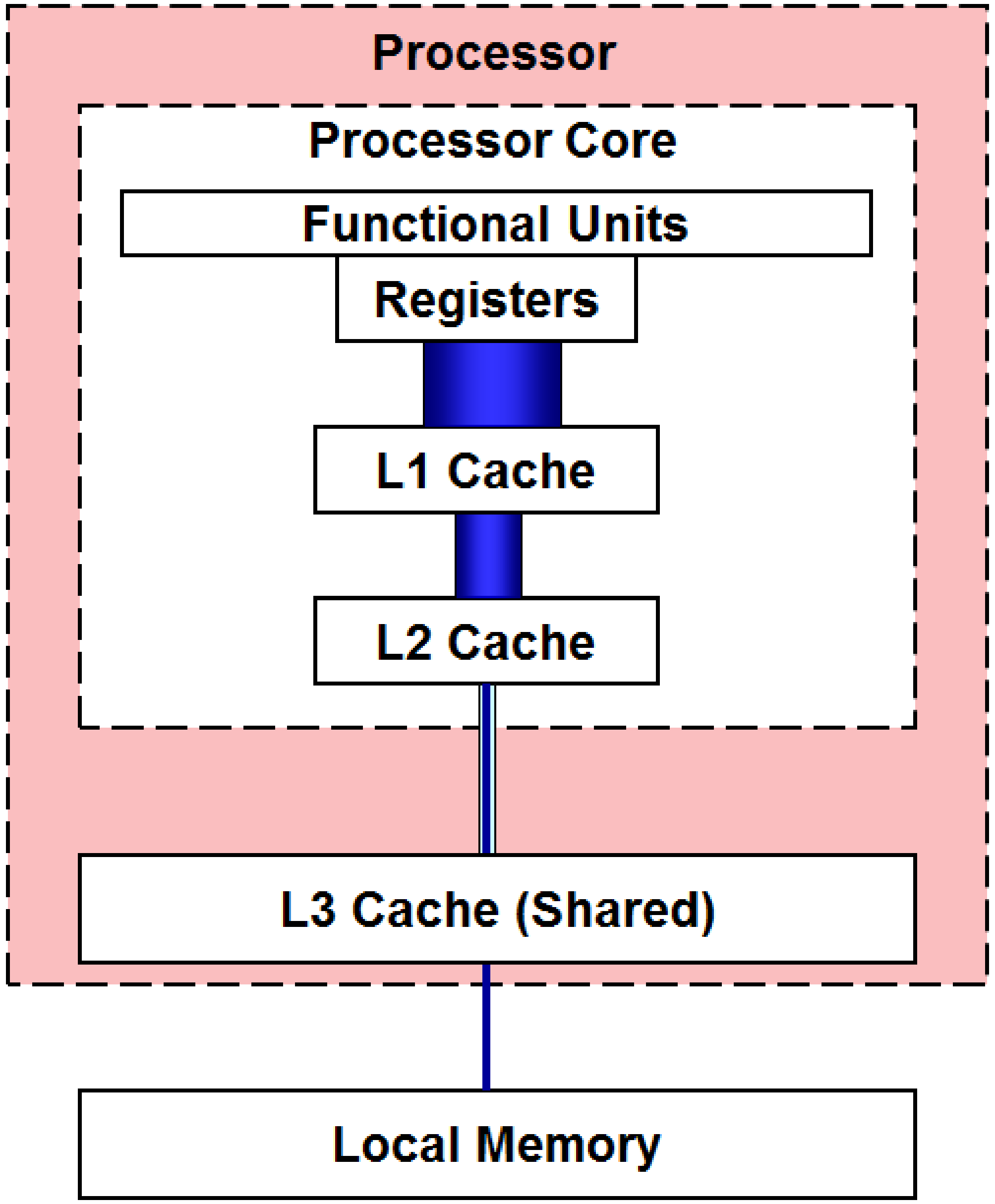
```
Struct wait_block { int globalcount; event zeroevent;}
Struct ref { Union {int count; wait_block wait)};
int acquire(*ref) {
   while (1) {
      oldcount = ref;
      if (oldcount & 1) return -1; // wait in progress
      newcount = oldcount + 2;
      if (CompareExchange(&ref->count, newcount, oldcount) ==
oldcount) return newcount; //succeed
    }
   }
}
```

Example continued..

```
void wait(*ref) {
   wait_block waitblock;
   while (1) {
     oldcount = ref;
     waitblock.event = onstackevent;
     waitblock.globalcount = oldcount-2;
     new = &waitblock | 1; //set low bit to zero to indicate this
is a waitblock;
      if (CompareExchange(ref, new, oldcount) break;
   }
   wait_for_event(&ref->waitblock.event);
}
```

Example release

```
void release(*ref) {
 while (1) {
     if (ref & 1) {
          …  mask low bit to zero for ref  as ref is a pointer now.
          val = InterlockedDecrement(&(ref->globalcount), 2)); // decrement by 
2;
          if ((val == 0) {
              signal(&ref->event); // waiter if any is woken
          }
          return;
    } else {
        if (CompareExchange(&ref, oldCount-2, oldCount)== oldCount)return;  // 
successfully decremented ref
  }
}
```



Cache L1:	**64 KB (per core)**
Cache L2:	256 KB (per core)
Cache L3:	**8 MB** (shared)

Cache aware rundown protection

- Ref count is per-processor, cache line sized and aligned.
- Acquire() only atomically increments the ref count of the current processor, if the lsb is 0
- Release() only decrements the ref count of the current processor – this means release can make that count negative
    - Wait() – atomically marks each per-processor ref count of all processors to be not granted (lsb set to 1)
    - Migrates the *sum* of all per-processor counts to a global count
    - Release() decrements the global count if lsb is 1, otherwise decrements per-processor count
    - Once the global count reaches 0, Release() signals waiter to wakeup

Queued spinlocks

- Traditional spinlock causes cache contention and doesn’t guaranteed ordering:

```
int {
    lock(lock) { while test_and_set(lock) == 1);
    return 0;
}
```

- Cache lines flushed when lock is set above which is a huge penalty on multi-proc systems.
- Queued spin locks benefits:
  - Lock is granted in the order they are requested to processors
  - Only the next waiter in line actually spins reducing contention

```
void lock(struct queued_spinlock *lock)
{
    lock->next = NULL;
    parent = add_and_return_parent(queue, lock);

    // if parent is not null the lock is already acquired and we
    // need to wait until it is released
    if (parent) {
        lock->is_locked = 1;
        parent->next = lock;

        while (lock->is_locked == true)
            ;
    }

    // we own the lock
}
```

```
void unlock(struct queued_spinlock *lock)
{
    // Is there a waiter?
    if (lock->next != NULL) {
       // wake up the next waiter
       lock->next->is_locked = false;
    }

    // No other waiters.
}
```

Memory barriers/fences

- Data Synchronization Barrier (DSB)
  - Forces the processor to wait for all pending explicit data accesses to complete before any additional instruction stages can be executed. There is no effect on pre-fetching of instructions. Expensive – processing is held up
- Data Memory Barrier (DMB)
  - Ensures that all memory accesses in program order before the barrier are observed in the system before any explicit memory accesses that appear in Instruction Synchronization Barrier (ISB). Cheaper as only load/stores are blocked
- Instruction synchronization barrier (ISB)
  - Flushes the pipeline and prefetch buffer(s) in the processor, so that all instructions following the ISB are fetched from cache or memory, after the instruction has completed.

DMB example

- `LOAD X0, [X1] // Must be seen by the memory system before the // STR below.`
- `DMB …`
- `ADD X2, #1 // May be executed before or after the memory //system sees LOAD`
- `STORE X3, [X4] // Must be seen by the memory system after the LOAD above.`

DSB example

- `STORE X0, [X1] // Access must have completed before DSB can complete`
- `DSB ISH`
- `ADD X2, X2, #3 //Cannot be executed until DSB completes`

Some conclusions

- Learn to debug deadlocks
- Do not rely on memory ordering – by default a memory location changed on current processor may not immediately reflect in another processor’s cache
- Interlocked operations have penalties (cache line flushes)
- Correctness over performance – though performance is a feature no in operating systems.