Multi-Object Synchronization

Chapter 6 OSPP

Part I

Multi-Object Programs

- What happens when we try to synchronize across multiple objects in a large program?
  - Each object with its own lock, condition variables
- Performance: single object
  - one big lock?
  - worse with multi-object
- Semantics/correctness
- Deadlock
- Eliminating locks

Synchronization Performance

- A program with lots of concurrent threads can still have poor performance on a multiprocessor:
  - Lock contention: only one thread at a time can hold a given lock
  - Shared data protected by a lock may ping back and forth between the cache within each core
  - False sharing: communication between cores even for data that is not shared

Web Server Lock

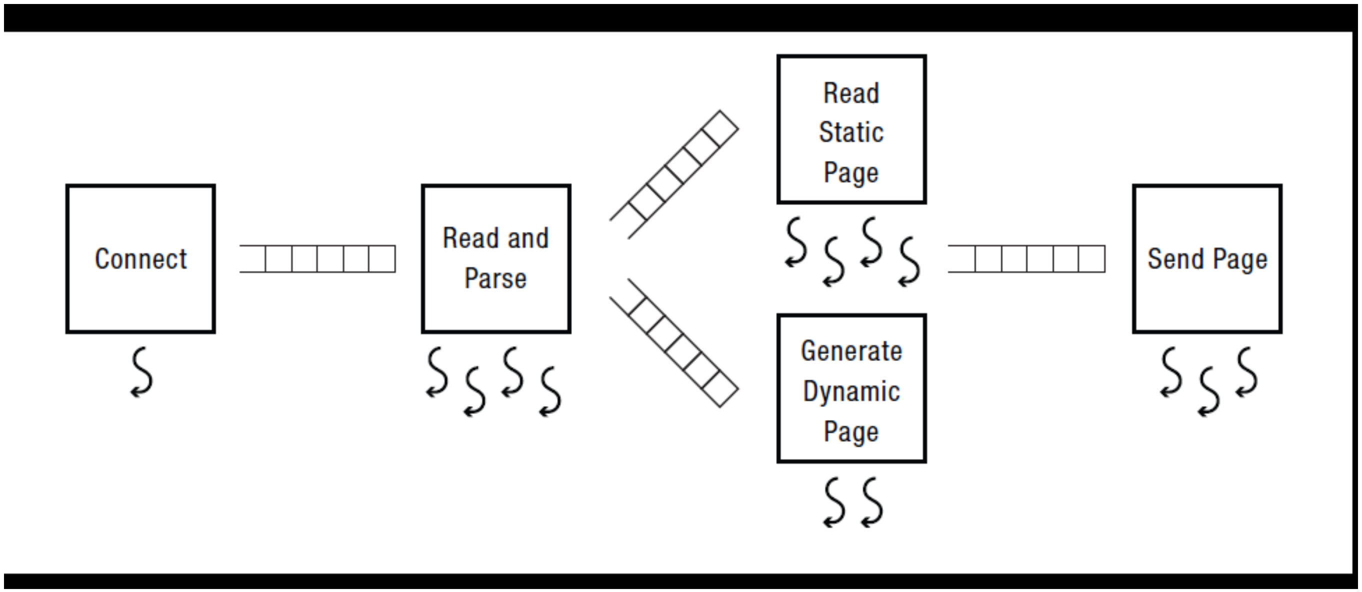
- In a memory cache that is accessed 5% of the time with a single lock
- On a multiprocessor suppose getting the lock is 4 times slower (get lock from another cache)

- Need careful design of shared locking

Reducing Lock Contention

- Fine-grained locking: partition by object
  - Partition object into subsets, each protected by its own lock
  - Example: hash table buckets, hard to resize
- Per-processor data structures: partition by core
  - Partition object so that most/all accesses are made by one processor: reduces false sharing, but cross cache access
  - Example: per-processor heap
- Ownership/Staged architecture: partition by op
  - Only one thread at a time accesses shared data
  - Example: pipeline of threads

Thread Pipelines



- Benefits
  - Modularity
  - Cache locality
  - Problems:

Lock Contention

- Still a major issue on a multiprocessor
- Busy locks can hamper performance
  - Everyone wants to access popular object
- MCS locks (if locks are mostly busy)
- RCU locks (if locks are mostly busy, and data is mostly read-only)
- We’ve seen opts for when lock was mostly FREE (fastpath)

The Problem with Test and Set

```
Counter::Increment() {
   while (test_and_set(&lock))
      ;
   value++;
   lock = FREE;
   memory_barrier();
}
```

What happens if many processors try to acquire the lock at the same time?

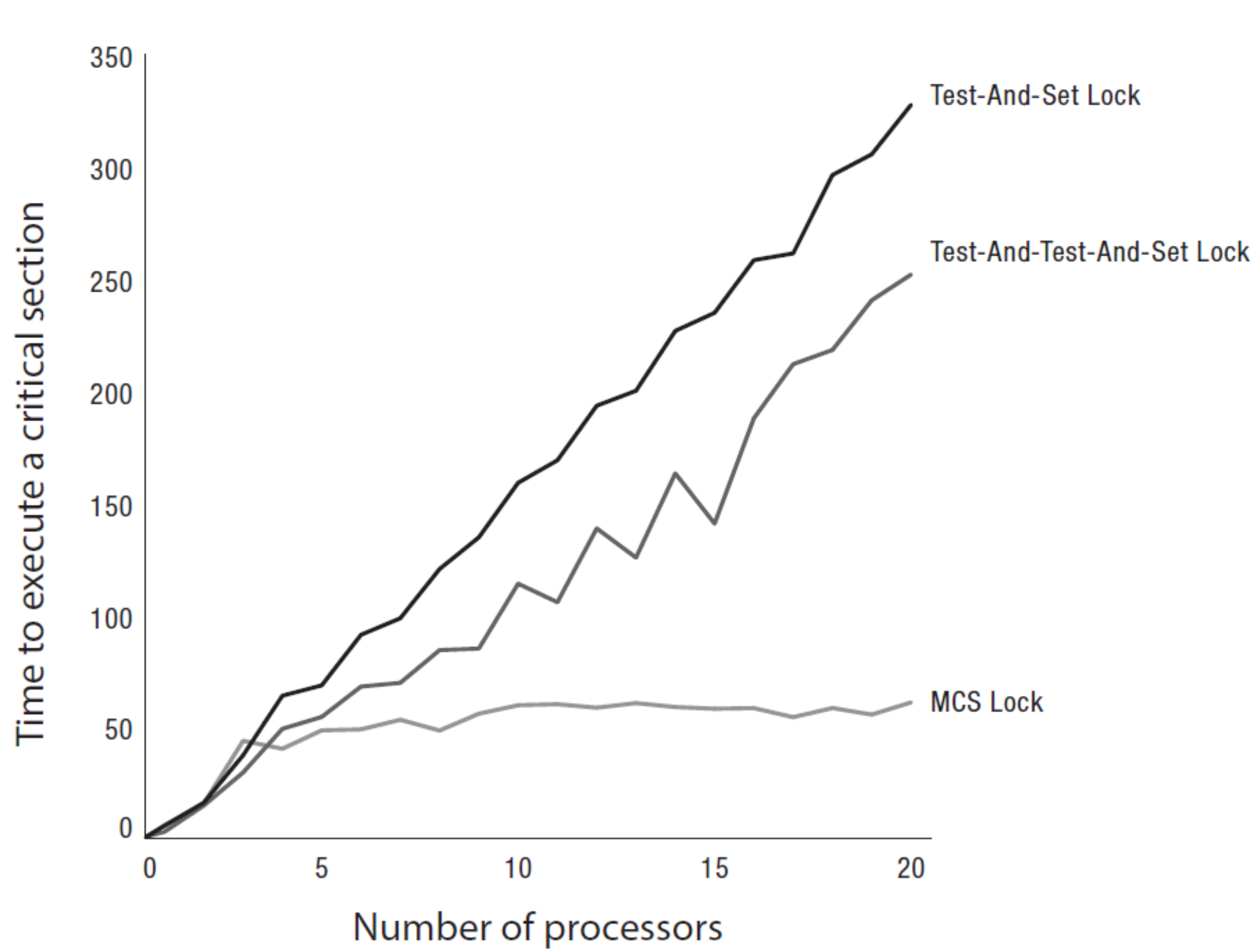
- Hardware doesn't prioritize "FREE"

The Problem with Test and Test and Set

```
Counter::Increment() {
    while (lock == BUSY && test_and_set(&lock))
        ;
    value++;
    lock = FREE;
    memory_barrier();
}
```

What happens if many processors try to acquire the lock?

Test (and Test) and Set Performance



Some Approaches

- Insert a delay in the spin loop
  - Helps but acquire is slow when not much contention
- Spin adaptively
  - No delay if few waiting
  - Longer delay if many waiting (give FREE a chance)
- MCS
  - Create a linked list of waiters using compareAndSwap
  - Spin on a per-processor location

What If Locks are Still Mostly Busy?

- MCS Locks
  - Optimize lock implementation for when lock is contended
  - Create a linked list of waiters using atomic compareAndSwap instruction
  - Spin on a per-processor location

- Relies on atomic read-modify-write instructions

MCS Lock

- Maintain a list of threads waiting for the lock
  - Front of list holds the lock
  - MCSLock::tail is last thread in list
  - New thread uses CompareAndSwap to add to the tail

- Lock is passed by setting next->needToWait = FALSE;
  - Next thread spins while its needToWait is TRUE

```
TCB {
    TCB *next;              // next in line
     bool needToWait;
}
MCSLock {
    Queue *tail = NULL; // end of line
}
```

MCS Lock

- Maintain a list of threads waiting for the lock
  - Front of list holds the lock
  - MCSLock::tail is last thread in list
  - New thread uses CompareAndSwap to add to the tail

- Lock is passed by setting next->needToWait = FALSE;
  - Next thread spins while its needToWait is TRUE

```
TCB {
    TCB *next;          // next in line
     bool needToWait;
}
MCSLock {
    Queue *tail = NULL; // end of line
}
```

MCS Lock Implementation: edited

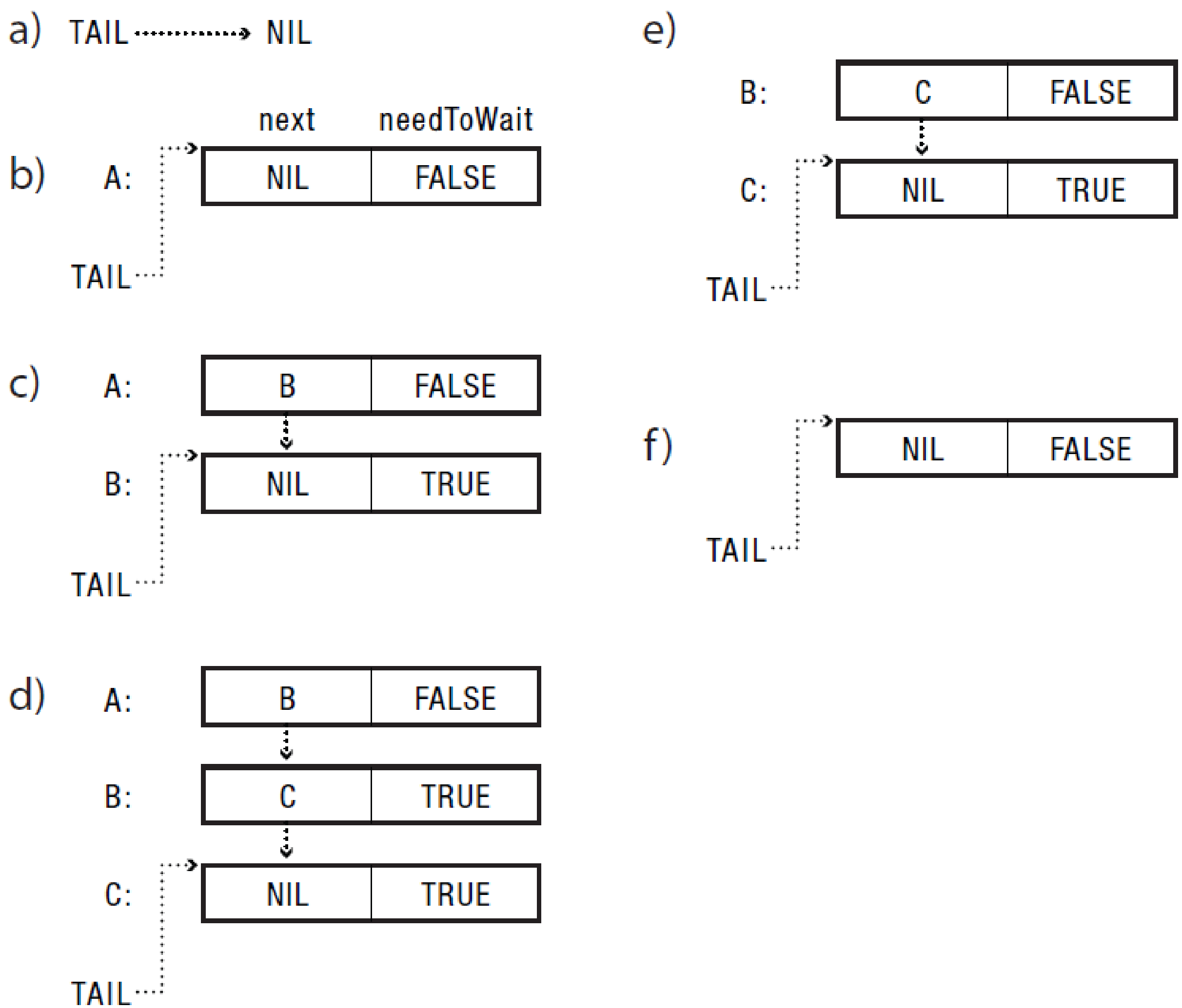
```
MCSLock::acquire() {
   Queue *oldTail = tail;

   myTCB−>next = NULL;
   myTCB−>needToWait = TRUE;
   // keep trying until I can be the tail
   while (!compareAndSwap(&tail,
                 oldTail, &myTCB)) {
      oldTail = tail;
   }
   if (oldTail != NULL) {
      oldTail−>next = myTCB;
      memory_barrier();
      // key: spinning on sep. var!
      while (myTCB−>needToWait)
         ;
   }
}
```

```
MCSLock::release() {
   // if I am the tail, no one is waiting
   if (compareAndSwap(&tail,
                          myTCB, NULL)) ;
   else {
      while (myTCB−>next == NULL)
         ;
      myTCB−>next−>needToWait=FALSE;
   }
}
```

```
bool cas (int *p, int old, new) {
   if (*p ≠ old) {
      return false;
   }
   *p = new;
   return true;
}
```

MCS In Operation



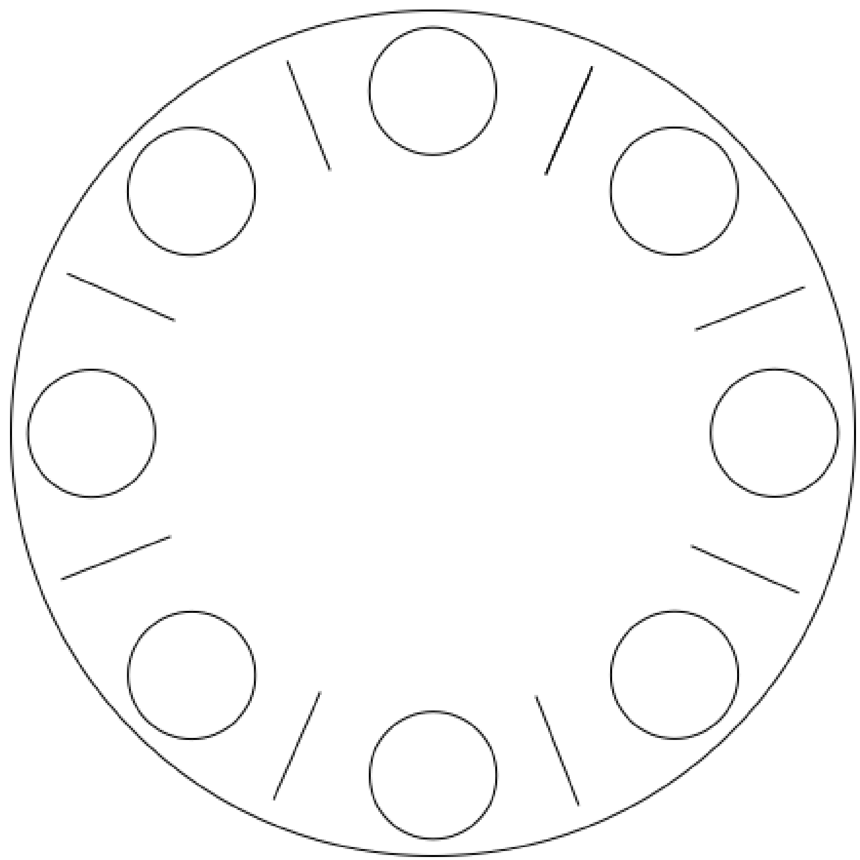
Deadlock Definition

- Resource: any (passive) entity needed by a thread to do its job (CPU, disk space, memory, lock)
  - Preemptable: can be taken away by OS
  - Non-preemptable: must leave with thread
- Starvation: thread waits indefinitely
- Deadlock: circular waiting for resources
  - Deadlock => starvation, but not vice versa

Example: two locks (recursive waiting)

Thread A	Thread B
lock1.acquire();	lock2.acquire();
lock2.acquire();	lock1.acquire();
lock2.release();	lock1.release();
lock1.release();	lock2.release();

Dining Lawyers



Each lawyer needs two chopsticks to eat.
Each grabs chopstick on the right first.

Necessary Conditions for Deadlock

- Limited access to resources
  - If infinite resources, no deadlock!
- No preemption
  - If resources are virtual, can break deadlock
- Multiple independent requests
  - “wait while holding”
- Circular chain of requests

Question

- How does Dining Lawyers meet the necessary conditions for deadlock?
  - Limited access to resources
  - No preemption
  - Multiple independent requests (wait while holding)
  - Circular chain of requests

- How can we modify Dining Lawyers to prevent deadlock?

Preventing Deadlock

- Exploit or limit program behavior
  - Limit program from doing anything that might lead to deadlock
- Predict the future
  - If we know what program will do, we can tell if granting a resource might lead to deadlock
- Detect and recover
  - If we can rollback a thread, we can fix a deadlock once it occurs

Exploit or Limit Behavior

- Provide enough resources
  - How many chopsticks are enough?
- Eliminate wait while holding
  - Release lock when calling out of module
  - Telephone circuit setup: p. 303
  - Internet router: p. 303 (conservative: drop pkts)
- Eliminate circular waiting
  - Lock ordering: always acquire locks in a fixed order
  - Example: move file from one directory to another

Example

Thread 1	Thread 2
1. Acquire A	1.
2.	2. Acquire B
3. Acquire C	3.
4.	4. Wait for A
5. If (maybe) Wait for B	

How can we make sure to avoid deadlock?

Deadlock Dynamics

- Safe state:
  - For any possible sequence of future resource requests, it is possible to eventually grant all requests
  - May require waiting even when resources are available!
- Unsafe state:
  - Some sequence of resource requests can result in deadlock
- Doomed state:
  - All possible computations lead to deadlock

Banker's Algorithm

- Grant request iff result is a safe state
- Sum of maximum resource needs of current threads can be greater than the total resources
  - Provided there is some way for all the threads to finish without getting into deadlock
- Example: proceed iff
  - total available resources - # allocated >= max remaining that might be needed by this thread in order to finish
  - Guarantees this thread can finish

Banker's Algorithm: insights

- Only allows safe states
- All resource needs are declared upfront, may wait
- Paging: 8 total, A wants 4, B wants 5, C wants 5

Process	Allocation											
A	0	1	1	1	2	2	2	3	3	3	wait	wait
B	0	0	1	1	1	2	2	2	3	3	3	wait
C	0	0	0	1	1	1	2	2	2	wait	wait	wait
Total	0	1	2	3	4	5	6	7	8	8	8	8

On the other hand, if the system follows the Banker's Algorithm, then it can delay some processes and guarantee that all processes eventually complete:

Process	Allocation																		
A	0	1	1	1	2	2	2	3	3	3	4	0	0	0	0	0	0	0	0
B	0	0	1	1	1	2	2	2	wait	wait	wait	wait	3	4	4	5	0	0	0
C	0	0	0	1	1	1	2	2	2	wait	wait	wait	3	3	wait	wait	4	5	0
Total	0	1	2	3	4	5	6	7	7	7	8	4	6	7	7	8	4	5	0

Optimistic Approach

- Optimize case with limited contention
- Proceed without the resource
  - Requires robust exception handling code
  - Amazon example p. 300
- Transactions: Roll back and retry
  - Transaction: all operations are provisional until have all required resources to complete operation