CSCI 350
Ch. 6 – Multi-Object Synchronization

Mark Redekopp
Michael Shindler & Ramesh Govindan

Overview

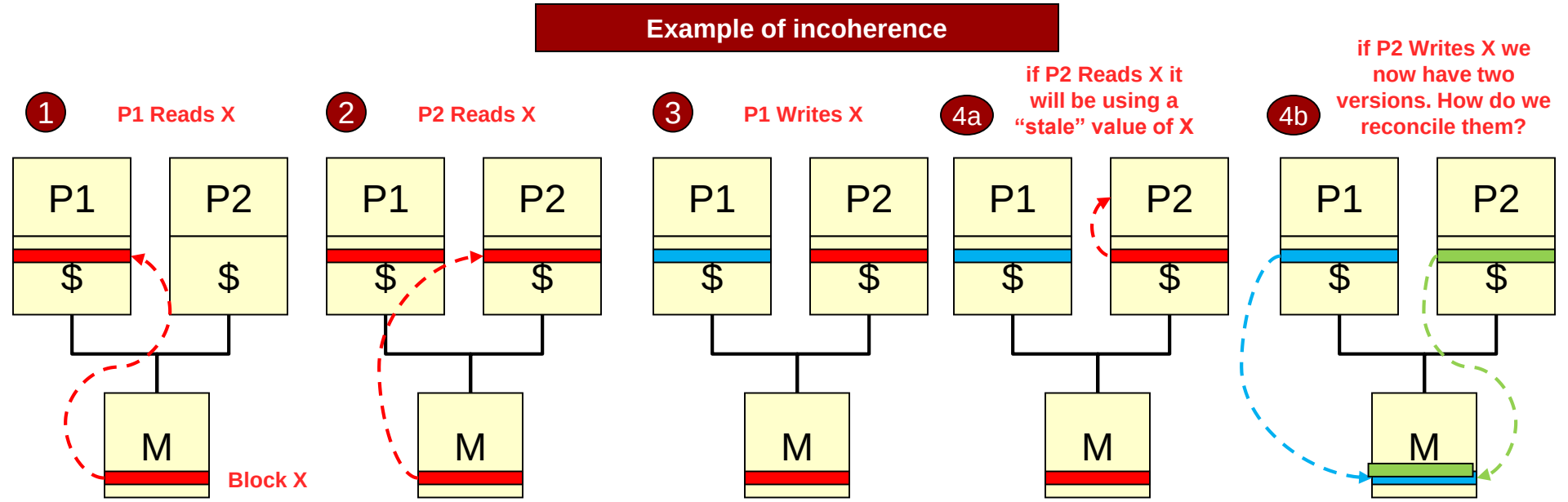
- Synchronizing a single shared object is not TOO hard
- Sometimes shared objects depend on others or require multiple resources each with their own lock
- When multiple locks become involved, new problems arise and reasoning about the system becomes more difficult
- In general, we need to be concerned about:
  - **Safety/correctness**: Ensure that atomicity is maintained correctly
  - **Multiprocessor performance**: Efficient performance is crucial for multiprocessors, especially because of cache effects
  - **Liveness**: Ensure that **deadlock**, **livelock** and **starvation** do NOT happen
    - Deadlock: No thread can run
    - Livelock: Threads can run but cannot make progress
    - Starvation: Some thread is consistently denied access to needed resources (deadlock implies starvation but starvation does not imply deadlock)

Effects of caching, false sharing, etc.

REVIEW OF CACHING & CONTENTION AND OTHER BACKGROUND MATERIAL

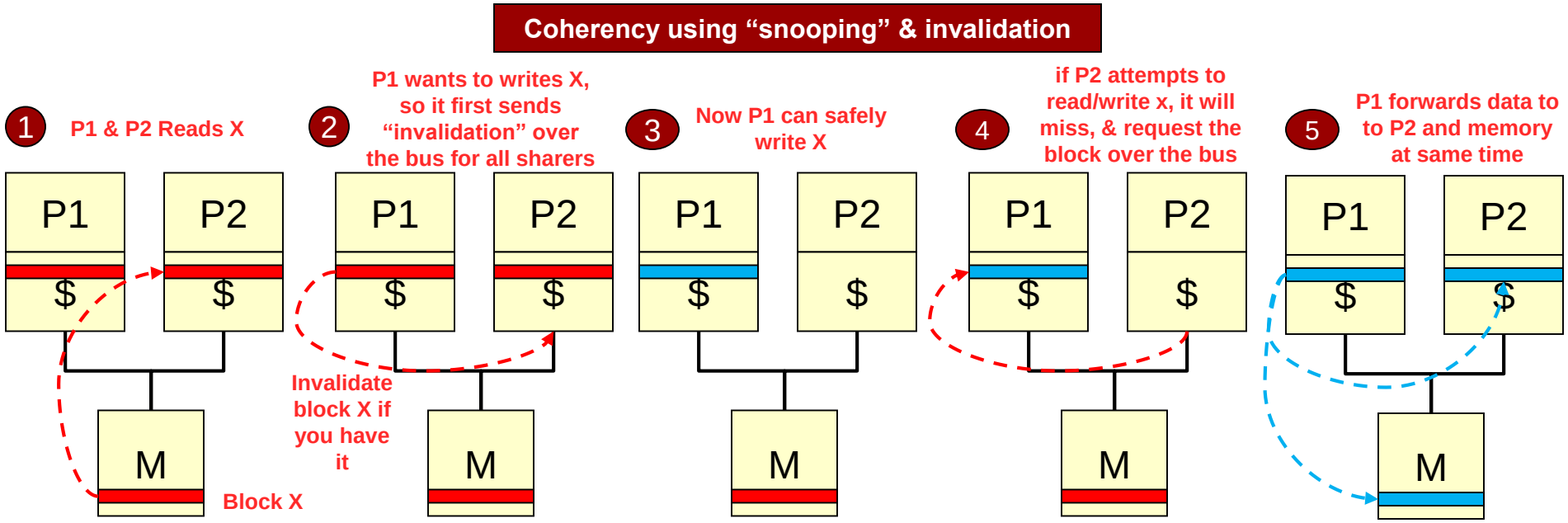
Cache Coherency

- Most multi-core processors are shared memory systems where each processor has its own cache
- Problem: Multiple cached copies of same memory block
  - Each processor can get their own copy, change it, and perform calculations on their own different values…INCOHERENT!
- Solution: Snoopy caches…



Solving Cache Coherency

- If no writes, multiple copies are fine
- Two options: When a block is modified
  - Go out and update everyone else's copy
  - Invalidate all other sharers and make them come back to you to get a fresh copy
- "Snooping" caches using invalidation policy is most common
  - Caches monitor activity on the bus looking for invalidation messages
  - If another cache needs a block you have the latest version of, forward it to mem & others



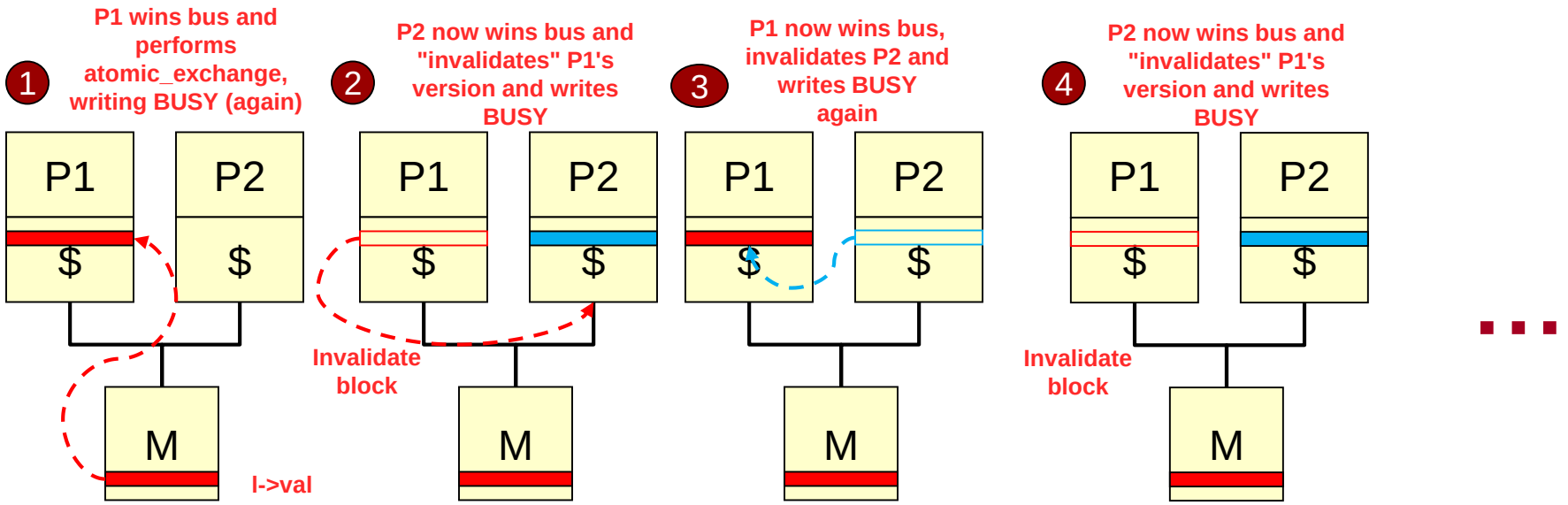
Lock Contention (Spinlocks)

- Consider a spinlock held by a thread on P3 (not shown) for a "long time" while thread 1 and 2 (on P1 and P2) try to acquire the lock
- Continuous invalidation of each other reduces access to the bus for others (especially P3 when it tries to release)

Thread1

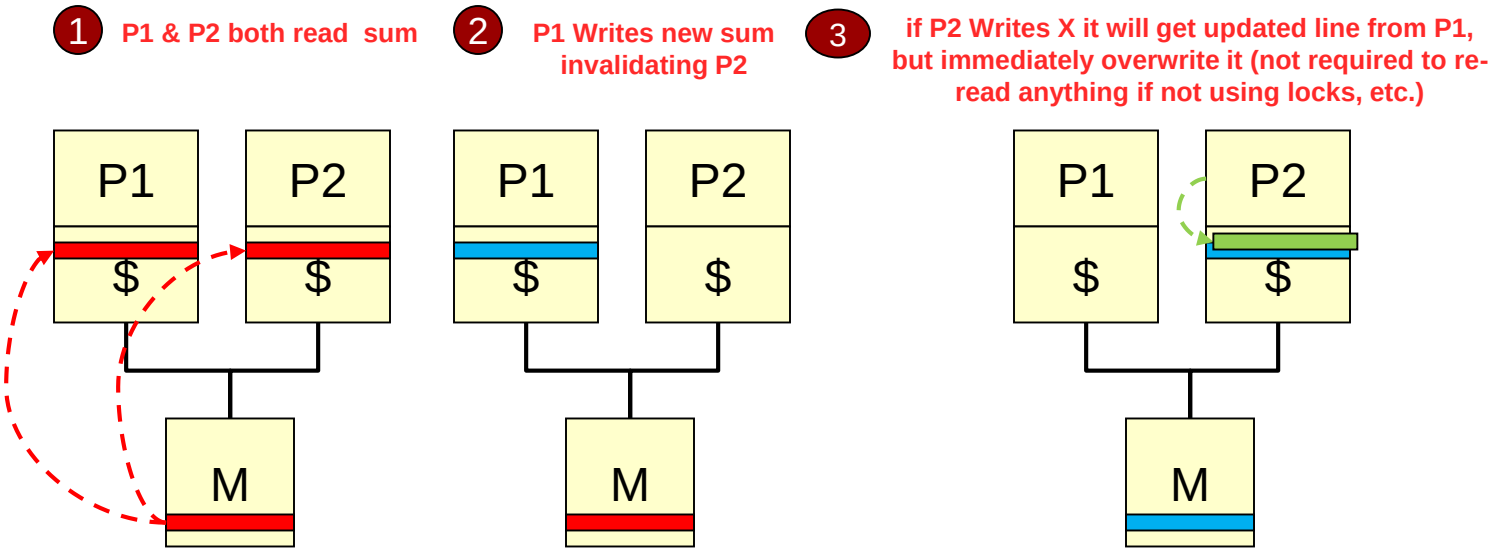
```
void acquire(lock* l)
{
  int val = BUSY;
  while( atomic_swap(
              val, l->val)
         == FREE);
}
```

Thread2



Is Cache Coherency = Atomicity?

- No, cache coherence only serializes writes and does not serialize entire read-modify-write sequences
  - Coherence simply ensures two processors don't read two different values of the same memory location
- Consider our sum example (sum = sum + 1;)



Amdahl's Law

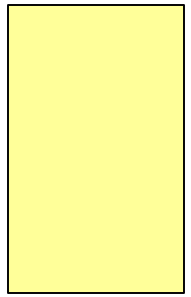
- Where should we put our effort when trying to enhance performance of a program
- Amdahl's Law => How much performance gain do we get by improving only a part of the whole

$$ExecTimeNew = ExecTimeUnaffected + \frac{ExecTimeAffected}{ImprovementFactor}$$

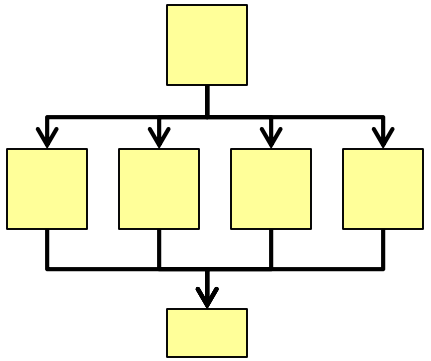
$$Speedup = \frac{ExecTimeOld}{ExecTimeNew} = \frac{1}{Percent_{Unaffected} + \frac{Percent_{Affected}}{ImprovementFactor}}$$

Amdahl's Law

- Holds for both HW and SW
  - HW: Which instructions should we make fast? The most used (executed) ones
  - SW: Which portions of our program should we work to optimize
- Holds for parallelization of algorithms (converting code to run multiple processors)



Original Sequential Program



Parallelized Program

Parallelization Example

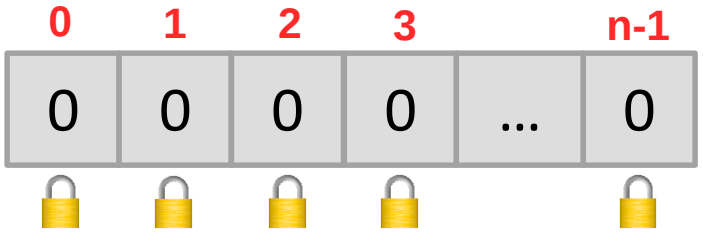
- A programmer parallelizes a function in her program to be run on 8 cores. The function accounted for 40% of the runtime of the overall program. What is the overall speedup of this enhancement?

$$Speedup =$$

FINE-GRAINED LOCKING

Locks and Contention

- The more threads compete for a lock the slower performance will be
  - Continuous sequence of invalidate, get exclusive access for ‘tsl’ or ‘cas’, check lock, see it is already taken, repeat
- Options
  - Use queueing locks
    - Go to sleep if lock is not available
  - Lock Granularity: Use locks for "pieces" of a data structure rather than the one lock for the whole structure
  - Others that you can explore as needed…

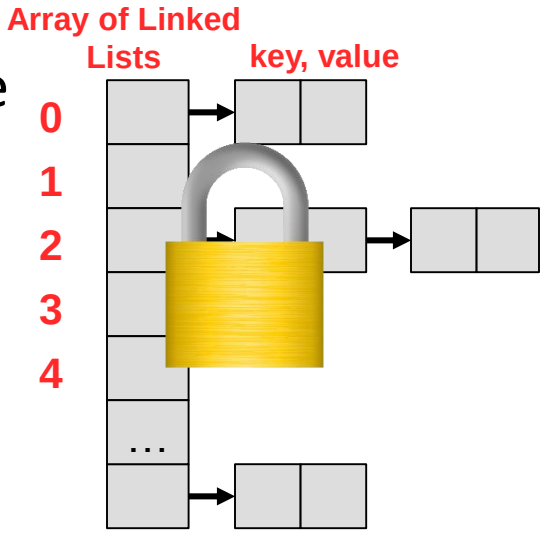


1 thread, 1 array	51.2
2 threads, 2 arrays	52.5
2 threads, 1 array	197.4
2 threads, 1 array (even/odd)	127.3

Example: Fig. 6.1 OS:PP 2nd Ed.

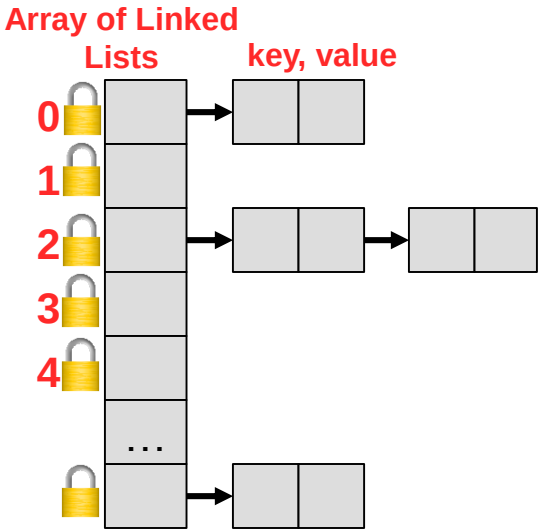
Hashtable Example

- Consider a shared data-structure like a hashtable (using chaining) supporting insert, remove, and find/lookup
  - We could protect concurrent access with one master lock for the whole data structure
  - This limits concurrency/performance
  - Consider an application where requests spend 20% of their time looking up data in a hash table. We can add N processors to serve requests in parallel but all requests must access the 1 hash table. What speedup can we achieve? How many processors should we use?
    - Even if we get rid of the other 80% of the access time we can at most achieve a 5x speedup since 20% of the time must be spent performing sequential work



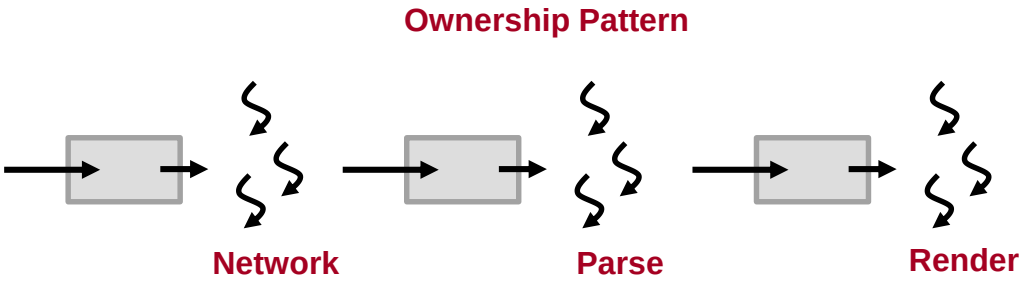
Fine Grained Locking Example

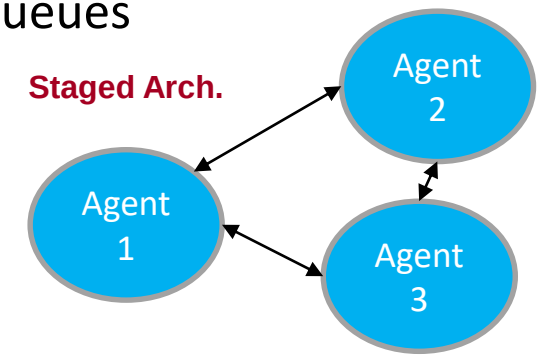
- However, remember keys hash to one chain where we will perform the insert/remove/find
  - We could consider one lock per chain so that operations that hash to a different chain can be performed in parallel
  - This is known as **fine-grained** locking
- But what if we need to resize the table and rehash all items? What do we have to do?
- One solution:
  - A Reader/Writer lock for the whole table and then fine-grained locks per chain
  - To resize, we acquire a writer lock on the hashtable



Other Ideas

- Separate/replicate data structures on each processor
  - Web server's cache of webpages
- Object ownership
  - Objects are queued for processing and whichever thread dequeues the object assumes exclusive access
  - Queue becomes the point of synchronization, not the object
- Staged Architecture (More general ownership pattern)
  - Shared state is private to the stage (and only the worker threads in that stage contend for it)
  - Messages/object passed between stages via queues





General Advice

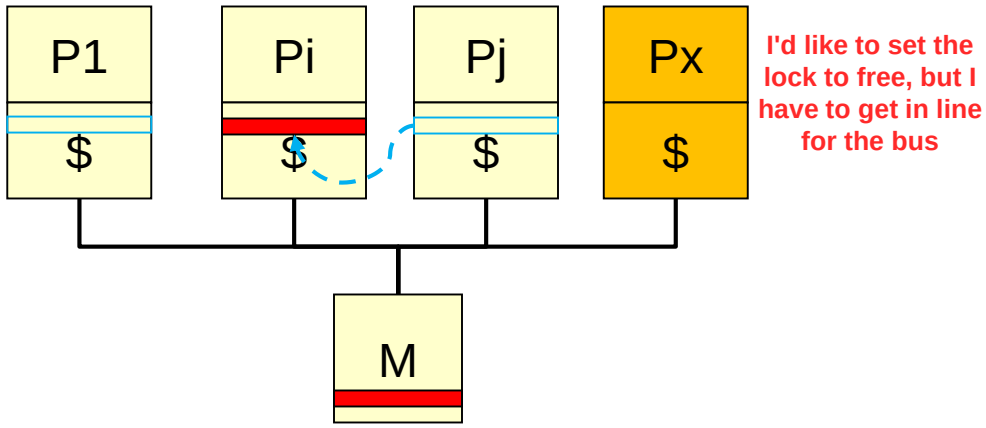
- *Premature optimization:* Avoid the temptation of writing the most fine-grained locks to begin with.
  - "It is easier to go from a working system to a working, fast system than to go from a fast system to a fast, working system".
  - Early versions of Linux used to have one big kernel lock (BKL), but over the years more and more fine-grained locking has been introduced.

REDUCING LOCK CONTENTION

Recall

- Consider a spinlock held by a thread on Px (not shown) while ***n*** other threads spin on the lock, trying to get exclusive access to the bus, and invalidating everyone else
- When Px wants to release the lock it is just 1 of the ***n*** threads contending for the bus
  - **Potentially requires O(n) time to release**

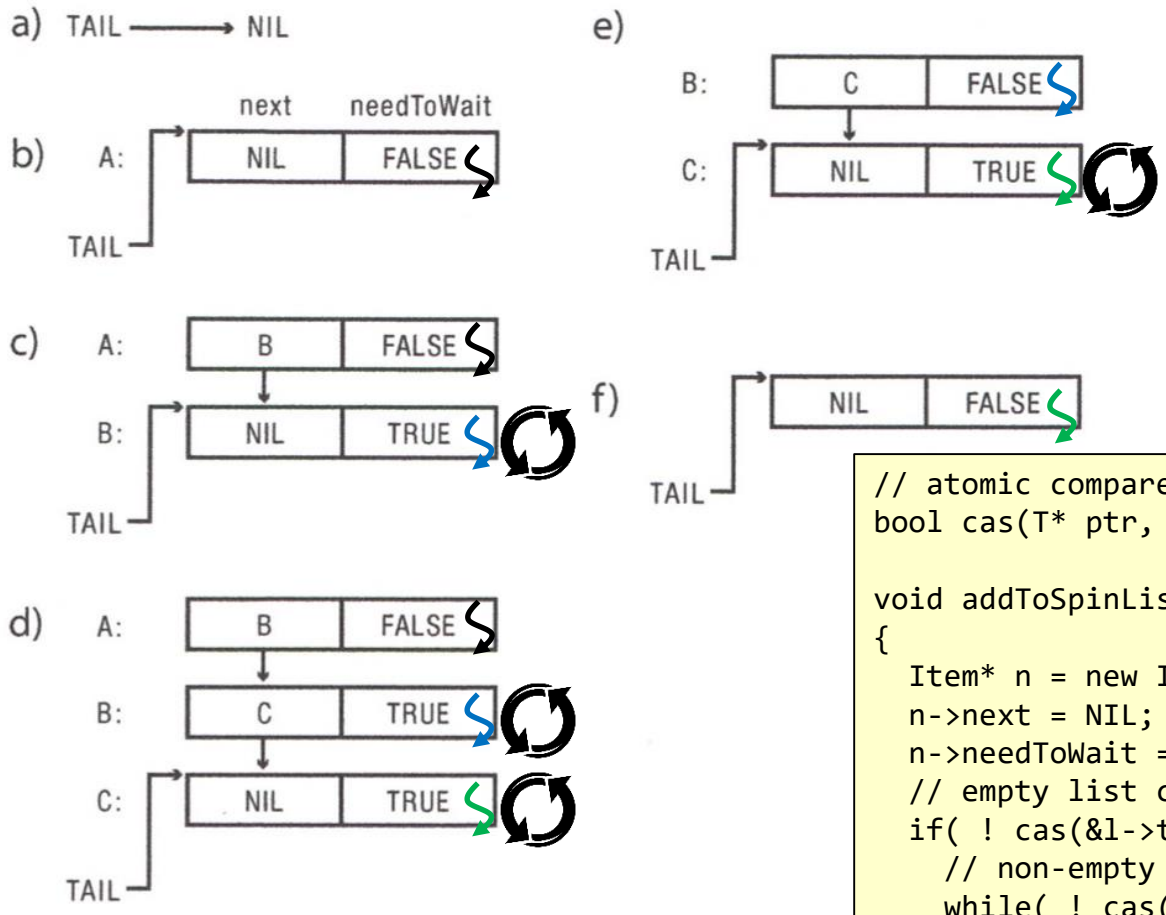
```
void acquire(lock* l)
{
  int val = BUSY;
  while( atomic_swap(
              val, l->val)
         == FREE);
}
```



MCS Locks

- Mellor-Crummey and Scott
- Better performance when MANY contenders
  - Main idea: Have each thread spin on a "different" piece of memory (to avoid cache coherency issues)
  - Create a new entry in a queue each with a different "flag" variable to spin on
  - When a thread releases the lock it will set the next thread's flag (i.e. flag in the queue's head item) causing that thread to "acquire" the lock
- Requires atomic update to tail/next pointer of the queue
  - Using a compare_and_swap atomic instruction

Illustration of MCS Locks



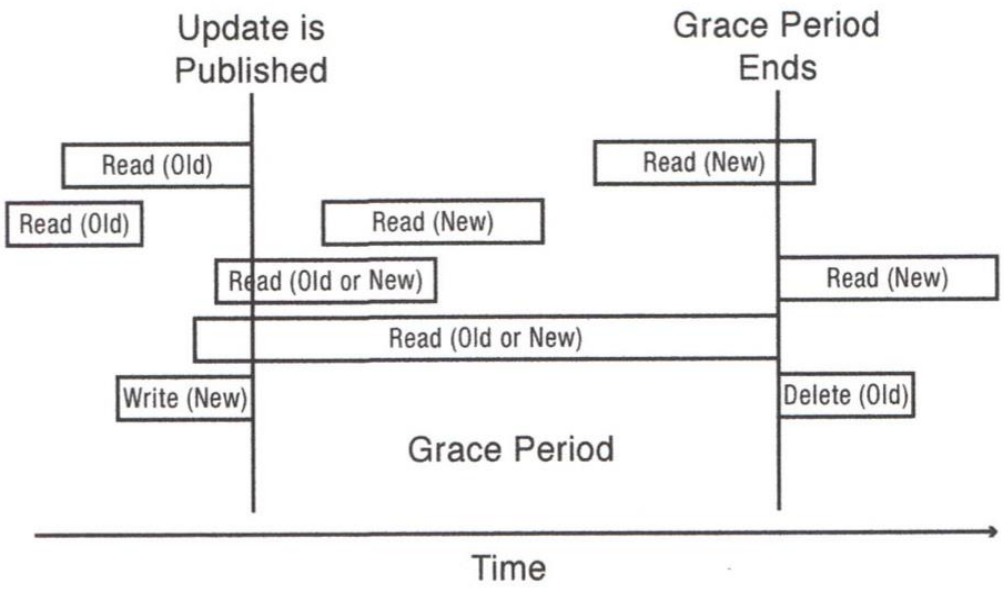

```
// atomic compare and swap
bool cas(T* ptr, T oldval, T newval);

void addToSpinList(MCSLock* l)
{
  Item* n = new Item;
  n->next = NIL;
  n->needToWait = true;
  // empty list case
  if( ! cas(&l->tail, NIL, n) ) {
    // non-empty case
    while( ! cas(&l->tail->next, NIL, n) );
  }
  else { n->needToWait = false; }
}
```

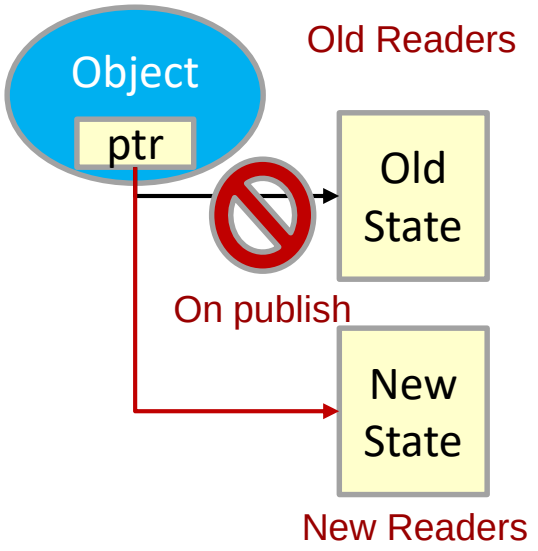
See OS:PP 2nd Ed. Fig. 6.3 for code implementation

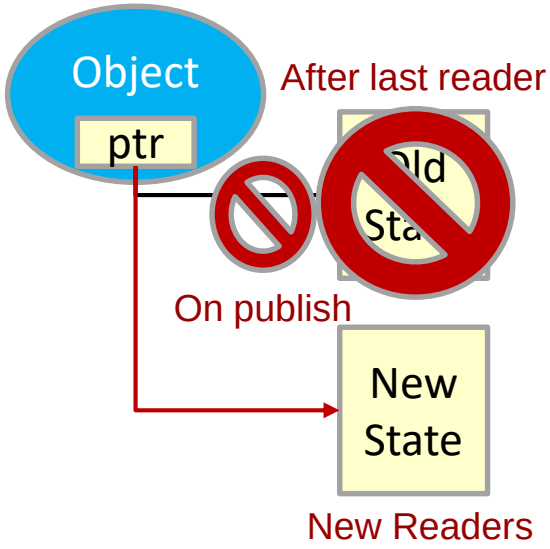
RCU Locks

- Read-Copy-Update Locks
  - An optimized Reader/Writer lock (optimizing the reader case)
  - Readers can be concurrent with at most 1 writer
    - Important: Can be writing during read
  - Writer creates a new "version" (updated copy) of the data, publishing the new version in an atomic compare_and_swap (usually a pointer update)
    - Concurrent readers will see a coherent version of the data, either old or new version (but not some mixture)
  - Once all readers that were looking at the old version finish, the old version can be deleted
    - Time from when the new data is published until the old version is deleted is known as the **grace period**
    - Uses information from the thread scheduler to know when readers of the old data are done (requires integration with the thread scheduler).
- Used in Linux kernel and Java

Illustration of RCU Locks







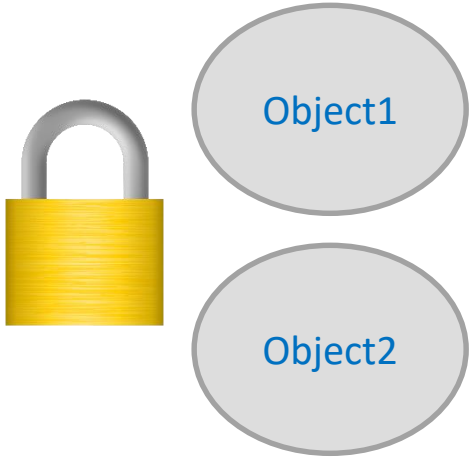
- Readers interrupt/check-in upon read completion or once per grace period
- Grace period ends when all "check-ins" have been received
  - No check-in => still reading

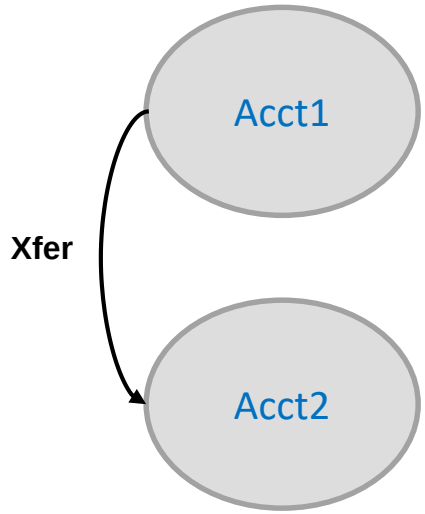
http://www.rdrop.com/users/paulmck/RCU/rclock_OLS.2001.05.01c.pdf

MULTIOBJECT SYNCHRONIZATION

Multiobject Synchronization

- RMW cycle involving multiple objects
  - A change in object1 necessitates a change in object2
- Consider a payment service like PayPal™
  - Transaction of transfer funds from account1 to account2
  - Several transactions may occur on an account at the same time
    - I could pay someone else at the same time a friend pays me





Options

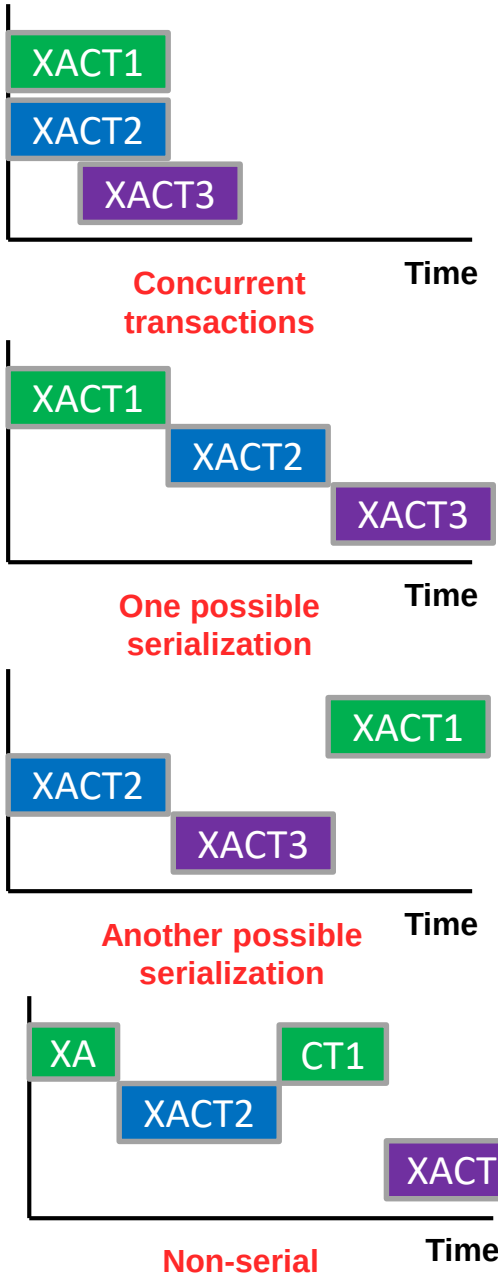
- 1 lock for all accounts
  - Linux's BKL
  - Limits Parallelism
- Fine-grained locking strategy
  - 1 lock per object / owner
  - Note: When multiple locks need to be held, deadlock may be a concern
  - Let's explore this option more
- Lock-free approaches
  - See later in the slides

```
void transact(
   Acct* from, Acct* to, int amount)
{
  allAccountsLock->acquire();

  from->deduct(amount);
  to->credit(amount);

  allAccountsLock->release();
}
```

```
void transact(
   Acct* from, Acct* to, int amount)
{
  from->lock->acquire();
  to->lock->acquire();

  from->deduct(amount);
  to->credit(amount);

  to->lock->release();
  from->lock->release();
}
```

Serializability

- **(Def.)** *The result of any program execution (of concurrent transactions) is equivalent to an execution in which transactions are processed one at a time in some order.*
- Example
  - Assume each person starts with $100
  - XACT1: Bob pays Alice $20
    - R_{11}(Bob),R_{12} (Alice),W_{13}(Bob),W_{14}(Alice)
  - XACT2: Bob deposits $50
    - R_{21}(Bob),W_{22}(Bob)
  - Non-serial ordering
    - R_{11},R_{21},W_{22},R_{12},W_{13},W_{14} => Bob ends with $80
  - Proper locking is meant to ensure serializability on shared data

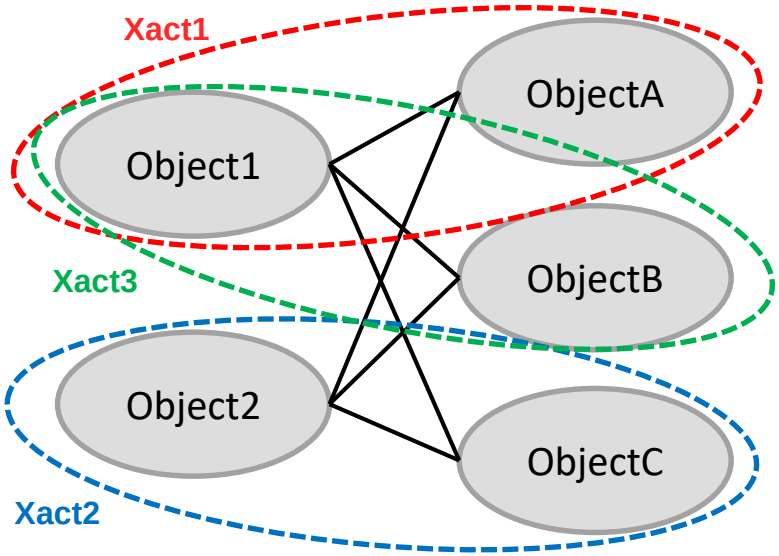


https://courses.cs.washington.edu/courses/cse344/11au/lectures/lecture19-transactions.txt
http://www.cburch.com/cs/340/reading/serial/

Acquire-All / Release-All

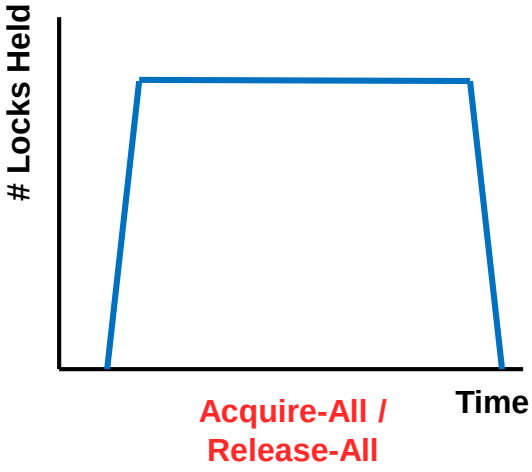
- Acquire all needed locks prior to updating ANY data
- Ensures **serializability**
- Pro: All benefits of fine-grained locking
  - Good parallelism when non-overlapping sets (e.g. XACT1 || XACT2)
- Con: May not know what locks are needed in advance
  - In that case we may be waiting for or holding locks that we don't even need
  - Example: If Bob has enough $$, pay Alice. Else Bob pays all he can, Charlie pays the balance
    - Don't know if we need Charlie's lock until we look at Bob

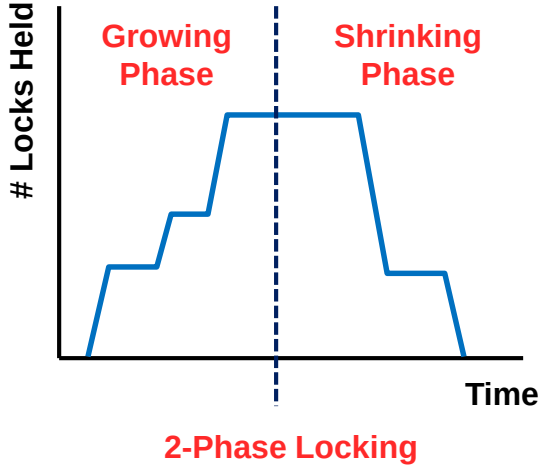
```
void transact(
   Acct* from, Acct* to, int amount)
{
  from->lock->acquire();
  to->lock->acquire();

  from->deduct(amount);
  to->credit(amount);

  to->lock->release();
  from->lock->release();
}
```



2-Phase Locking

- A slight relaxation on acquire-all/release-all
  - Can acquire locks at different times and release locks at different times
  - But *once any lock is released*, **no more lock acquisitions can be made**
- Example: If Bob has enough $$, pay Alice. Else Bob pays all he can, Charlie pays the balance
  - Acquire lock on Charlie's acct. only if needed
  - Non-serializable: Lock(Bob), Lock(Alice), transfer some $$ from Bob->Alice, Unlock(Bob), Unlock(Alice), Lock(Charlie), Lock(Alice), etc.
- Still ensures serializability
  - Giving up and then reacquiring locks allows non-serializable transactions





DEADLOCK & ITS MITIGATION

Deadlock

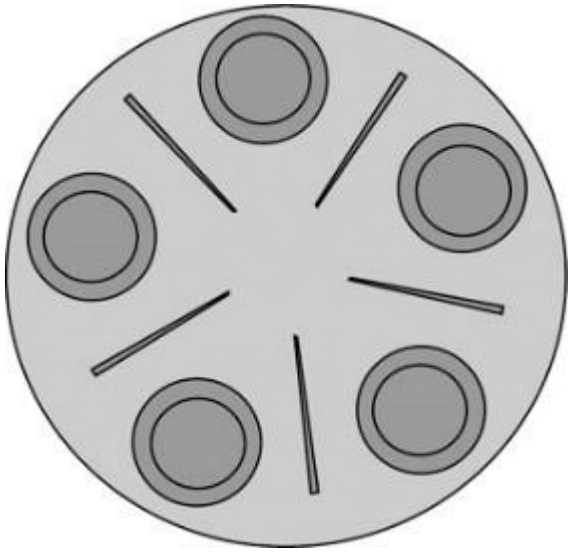
- When multiple locks are involved, deadlock becomes an issue
- **Deadlock**: No thread is able to make progress
- Causes
  - Mutually Recursive Waiting
  - Nested Waiting
  - ALL use a HOLD & WAIT strategy
- Examples:
  - Busy intersection
  - Dining Philosophers

```
void myTask(void* arg)
{
  lock1.acquire();
  lock2.acquire();
  ...
}
void yourTask(void* arg)
{
  lock2.acquire();
  lock1.acquire();
  ...
}
```

Recursive Waiting

Dining Philosophers Problem

- Classical "toy" example of deadlock
- ***n*** philosophers having dinner together
  - Like to talk for a while and then take a bite of food
- ***n*** chopsticks available on the table
  - Pick up left chopstick
  - Pick up right chopstick
  - Eat
  - Return chopsticks
- How can deadlock occur?



Dining Philosophers Problem

```
1. think for a while
2. get left chopstick
3. get right chopstick
4. eat for a while
5. return left chopstick
6. return right chopstick
7. return to 1
```

http://www.chegg.com/homework-help/questions-and-answers/dining-philosophers-problem-invented-e-w-dijkstra-concurrency-pioneer-clarify-notions-dead-q9351133

Deadlock vs. Starvation

- Deadlock implies starvation but not vice versa
- Starvation example
  - Reader/writer lock (a reader that keeps being held off)
  - But no deadlock
- Deadlock is usually non-deterministic
  - May work fine for many "runs" of the program
  - Deadlock occurs only if the right-sequence / interleaving occurs

Necessary Conditions

- **Four** necessary conditions:
  - Bounded resources/mutual exclusion: for at least one resource, there must be mutual exclusion (or a limit on the number of threads that can concurrently use the resource)
  - Hold and wait: threads can hold a resource and wait for another
  - No preemption: no way to revoke a resource from a thread
  - Circular wait (cyclical wait): a set of waiting threads such that each thread waits for another
- Are these sufficient conditions?
- No, necessary but not sufficient
  - Philosopher's can eat happily for a long time provided they don't all pick up a chopstick on their left (or right) at the same time

Dining Philosophers Example

- **Bounded Resources:** Limited chopsticks
- **Hold and Wait:** Philosopher picked up one and waited for 2nd
- **No preemption:** Philosopher won't put down a chopstick until they eat (get a second chopstick)
- **Cycle in Dependencies**: Each philosopher waits for the philosopher to their right (around a circular table).

Preventing Deadlock

- Cause of deadlock may occur much earlier than the actual moment the deadlock occurs
  - Indirect, future resource needs that are grabbed much earlier
- 3 general strategies for prevention:
  - Change structure of program
  - Predict the future (know necessary resources in advance)
  - Detect and recover (undo / rollback when deadlock occurs)

Changing Structure of the Program

PREVENTING DEADLOCK 1

Avoiding Deadlock By Changing Program

- Since we know the necessary conditions we can simply ensure that one of them is not met
- **1. Circular wait (cyclical wait):** a set of waiting threads such that each thread waits for another
  - Total ordering of locks
  - Linux src: mm/filemap.c

```
/*
 * Lock ordering:
 *
 *  ->i_mmap_rwsem              (truncate_pagecache)
 *    ->private_lock            (__free_pte->__set_page_dirty_buffers)
 *      ->swap_lock             (exclusive_swap_page, others)
 *        ->mapping->tree_lock
```

Linux src: mm/filemap.c

```
void myTask(void* arg)
{
  lock1.acquire();
  lock2.acquire();
  ...
}
void yourTask(void* arg)
{
  lock1.acquire();
  lock2.acquire();
  ...
}
```

Reorder

```
void myTask(void* arg)
{
  if(&lock1 < &lock2){
    lock1.acquire();
    lock2.acquire();
  }
  else {
    lock2.acquire();
    lock1.acquire();
  }
  /* Do some computation/updates */
}
```

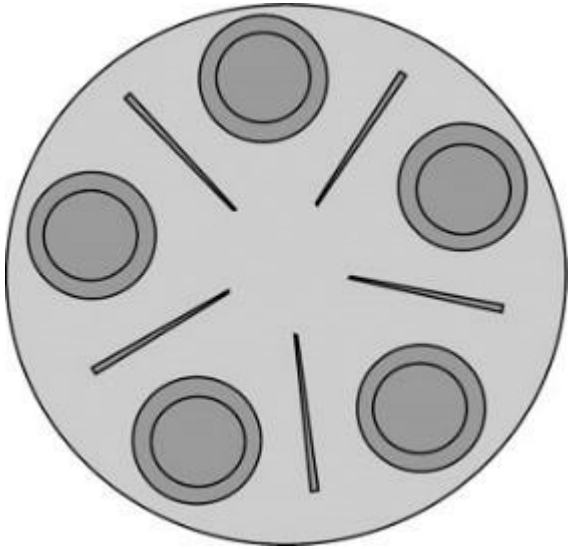
Trick: Use lock addresses to order
(OSTEP, Ch. 32 Concurrency Bugs)

Revisiting Necessary Conditions

- **2. Bounded resources/mutual exclusion**: Provide ample provisioning of resources (enough memory, etc.)
  - N+1 chopsticks for the dining philosophers (i.e. 1 spare)
- **3. Hold and wait**: threads can hold a resource and wait for another
  - Release resources before waiting
  - lock1.acquire(); lock2.tryAcquire()…If fail, release lock1 & start again
- **4. No preemption**: no way to revoke a resource from a thread
  - Take away resources (e.g. pages of memory) from one task and give to another

Livelock

- Livelock
  - **(Def.) Threads running but not making progress**
- We could modify the dining philosophers problem to avoid deadlock
  - If can't get both chopsticks put the other one down
- Explain a scenario where livelock occurs?



Dining Philosophers Problem

```
1. think for a while
2. get left chopstick
3. try to get right chopstick
4. if successful
   5. eat for a while
   6. return right chopstick
7. return left chopstick
8. goto to 1
```

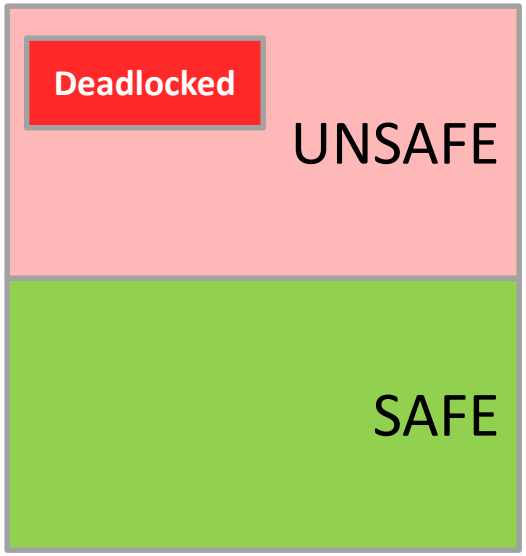
http://www.chegg.com/homework-help/questions-and-answers/dining-philosophers-problem-invented-e-w-dijkstra-concurrency-pioneer-clarify-notions-dead-q9351133

Controlling resource allocation

PREVENTING DEADLOCK 2

State Space of a System

- **Safe**: Deadlock cannot occur
  - For all possible requests there is at least one ordering for processing those requests that will succeed in granting those and other future requests
- **Unsafe**: Deadlock is possible but may not happen
  - There is a possible set of requests where no possible processing order can satisfy the requests
- **Deadlocked**: Deadlock has occurred



Safe or Unsafe?

- Suppose we have M resources where Available[k] (0 <= k <= M-1) represents number of free resources of type k exist
- N processes exist and declare in advance the max number of each type of resource they will need (i.e. MaxNeed[i][j] is the maximum number of type j resources that process i needs)
- For each of the states on the bottom indicate if they are safe or unsafe?

Avail	R1	R2
	8	6

Total Resources Available

Proc	R1	R2
A	5	3
B	4	2
C	4	3

Max Resource Requests

Proc	R1	R2
A	2	1
B	2	1
C	1	1

Is this state safe/unsafe/deadlocked?

Proc	R1	R2
A	3	1
B	2	2
C	2	1

Is this state safe/unsafe/deadlocked?

Safe or Unsafe?

- Consider the available and max resource request tables to the right
- For each of the states on the bottom indicate if they are safe or unsafe?
  - A: Safe – Even if a process requests the remainder of its max resource allocation we can satisfy one of those processes and then others
  - B: Unsafe – If no one returns resources before they request more we cannot satisfy any processes request. Could lead to deadlock

Avail	R1	R2
	8	6

Total Resources Available

Proc	R1	R2
A	5	3
B	4	2
C	4	3

Max Resource Requests

Proc	R1	R2
A	2	1
B	2	1
C	1	1

SAFE

Proc	R1	R2
A	3	1
B	2	2
C	2	1

UNSAFE

Deadlock is not guaranteed for 2nd option until all processes block on requests that are unable to be satisfied.

Banker's Algorithm Setup

- What method should we use to determine whether to grant a resource request?
- We could use an **acquire-all/release-all** strategy such that any new process receives its maximum needed resources or is blocked until it can
  - Remember *maximum* needed may not be *actual* needed
  - Could be overly conservative
  - **Would ensure a safe state** (A and B are guaranteed to finish at some point and return their resources allowing others to make progress)
- **Requires resource needs known in advance!**

Avail	R1	R2
	8	6

Total Resources Available

Proc	R1	R2
A	5	3
B	2	2
C	3	1

Max Resource Requests
C will be blocked A or B finishes

Banker's Algorithm & Example 1

- Banker's algorithm (proposed by E. Dijsktra) allows greater concurrency while still ensuring a safe state is maintained
- Upon a request, ensure there is a sequence of grants that can be made that will allow all processes to eventually finish, otherwise have the request wait (block)

Avail	R1	R2
	9	6

Total Resources Available

Proc	R1	R2
A	5	3
B	4	2
C	4	5

Max Resource Requests

Req	R1	R2
A	3	1

Grant / Block

Proc	R1	R2
A	1	0
B	3	1
C	2	3

Current state

Req	R1	R2
C	1	2

Grant / Block

Req	R1	R2
A	1	1

Grant / Block

Banker's Algorithm & Example 1

- Banker's algorithm (proposed by E. Dijsktra) allows greater concurrency while still ensuring a safe state is maintained
- Upon a request ensure there is a sequence of grants that can be made that will allow all processes to eventually finish, otherwise have the request wait (block)

Avail	R1	R2
	9	6

Total Resources Available

Proc	R1	R2
A	5	3
B	4	2
C	4	5

Max Resource Requests

Proc	R1	R2
A	1	0
B	3	1
C	2	3

Current state

Req	R1	R2
A	3	1

Block – No one can finish if all request more

Req	R1	R2
C	1	2

Grant – C can finish later & then give up resources

Req	R1	R2
A	1	1

Grant – B can still get necessary resources, finish, and free up enough resources for others

Banker's Algorithm & Example 2

- Is it safe to grant the following request?

Req	R1	R2
A	1	0

Grant / Block

Avail	R1	R2
	8	4

Total Resources Available

Proc	R1	R2
A	6	2
B	2	1
C	6	2

Max Resource Requests

Proc	R1	R2
A	2	1
B	0	0
C	3	2

Current state

Banker's Algorithm & Example 2

- Unsafe!
  - You might think it is okay to grant the request since there would be enough resources for B to request and be granted resources and then complete
  - But even if B completes A and C by themselves would now be in an unsafe state (each potentially needing 3 more when only 2 would be available)

Avail	R1	R2
	8	4

Total Resources Available

Proc	R1	R2
A	6	2
B	2	1
C	6	2

Max Resource Requests

Proc	R1	R2
A	2	1
B	0	0
C	3	2

Current state

Req	R1	R2
A	1	0

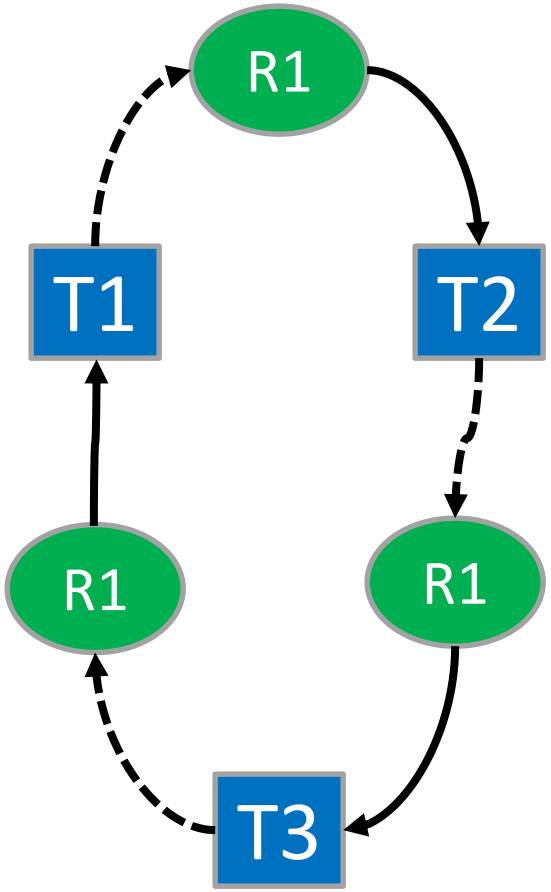
Block

Detect and Recover

PREVENTING DEADLOCK 3

Detecting Deadlock

- Detect cyclical resource dependency
  - Maintain a graph of threads and their "hold" and "need" relationship
- Threads that have not made progress in a "long" time



Recovering From Deadlock

- Rollback or kill/restart some threads
- Use "transactional system"
  - Computation can be "rewound" or rolled back to a checkpointed state
  - If deadlock occurs, pick some involved thread and roll it back
  - Allow other(s) to proceed
  - Generally, abort the 'youngest' thread

```
void threadTask(void* arg)
{
  /* Do local computation */

  /* checkpoints/saves state */
  begin_transaction(val1,val2) {

  lock1.acquire();

  /* Do some computation/updates */
  read(val1); write(val1);

  /* Could deadlock..if so,
     abort_transaction */
  lock2.acquire();

  read(val2);
  write(val2); write(val1);

  } // end_transaction
  abort {
    // release lock1
    // restore/re-read val1, val2
    // restart
  }
  lock1.release();
  lock2.release();
}
```

Selecting Who Rollsback/Retries

- Assume 2 threads are requesting a lock already held by each other
- Wait-die (non-preemptive)
  - If an older thread needs a lock held by a younger thread, the older can wait
  - If a younger thread needs a lock held by an older, it chooses itself to rollback
- Wound-wait (preemptive)
  - If an older thread needs a lock held by a younger thread, the younger is preemptively aborted
  - If a younger thread needs a lock held by an older, it can wait (may be prempted later)

```
void threadTask(void* arg)
{
  /* Do local computation */

  /* checkpoints/saves state */
  begin_transaction(val1,val2) {

  lock1.acquire();

  /* Do some computation/updates */
  read(val1); write(val1);

  /* Could deadlock..if so,
     abort_transaction */
  lock2.acquire();

  read(val2);
  write(val2); write(val1);

  } // end_transaction
  abort {
    // release lock1
    // restore/re-read val1, val2
    // restart
  }
  lock1.release();
  lock2.release();
}
```

http://www.mathcs.emory.edu/~cheung/Courses/554/Syllabus/8-recv+serial/deadlock-compare.html

What Do Real OSs Do?

- Not much
  - Up to programmer to write code that doesn't produce deadlock
  - Some might do detection

LOCK FREE STRUCTURES

Locking/Atomic Instructions

- TSL (Test and Set Lock)
  - tsl reg, addr_of_lock_var
  - Atomically stores const. '1' in lock_var value & returns lock_var in reg
    - Atomicity is ensured by HW not releasing the bus during the RMW cycle
- CAS (Compare and Swap)
  - cas addr_to_var, old_val, new_val
  - Atomically performs:
    - if (*addr_to_var != old_val) return false
    - else *addr_to_var = new_val; return true;
  - x86 Implementation
    - old_value always in $eax
    - CMPXCH r2, r/m1
      - if(%eax == *r/m1) ZF=1; *r/m1 = r2;
      - else { ZF = 0; %eax = *r/m1; }

```
ACQ: tsl     (lock_addr), %reg
     cmp   $0,%reg
     jnz     ACQ
     ret

REL: move  $0,(lock_addr)
```

```
ACQ:  move   $1, %edx
L1:   move   $0, %eax
      lock   cmpxchg %edx, (lock_addr)
      jnz      L1
      ret

REL:  move   $0, (lock_addr)
```

Lockless Atomic Updates

- Write data structures or code to avoid separate lock variables but to update data structures in a "transactional" way
  - Read and modify data w/o locks
  - Write only if data hasn't been accessed by another thread
- CAS (Compare and Swap) [x86]
- LL and SC (MIPS & others)
  - Lock-free atomic RMW
  - LL = Load Linked
    - Normal lw operation but tells HW to track any external accesses to addr.
  - SC = Store Conditional
    - Like sw but only stores if no other r/w to that addr. since LL & returns 0 in reg. if failed, 1 if successful

```
// High-level implementation
synchronized {
    sum += local_sum;
}
```

```
// x86 implementation
INC: move    (sum_addr), %edx
     move    %edx, %eax
     add     (local_sum),%edx
lock cmpxchg %edx, (sum_addr)
     jnz     INC
     ret
```

```
// MIPS implementation
     LA   $t1,sum
INC: LL   $5,0($t1)
     ADD  $5,$5,local_sum
     SC   $5,0($t1)
     BEQ  $5,$zero,UPDATE
```

TRANSACTIONS

Extending Lock-Free Structures with "Transactional Memory"

- No need to acquire lock
- Just indicate shared data
- HW & OS monitor no other access to shared data DURING the transaction
- If so, either rollback/retry some or all of the threads accessing the shared data
- Updates made "locally" during the transaction and are made visible if the transaction succeeds or destroyed if the transaction aborts
  - Otherwise, no computation (intermediate results) will be visible and computation restarts fresh

```
void threadTask(void* arg)
{
  /* Do local computation */

  /* checkpoints/saves state */
  begin_transaction(val1,val2) {

  /* Do some computation/updates */
  val1 -= amount;
  val2 += amount;
  } // end_transaction
  abort {
    // restore/re-read val1, val2
    // restart
  }
  lock1.release();
  lock2.release();
}
```

Active research in computer architecture & systems about ***Transactional Memory***

ANSWERS

Parallelization Example

- A programmer parallelizes a function in his program to be run on 8 coR The function accounted for 40% of the runtime of the overall program. What is the speedup of the enhancement?

$$Speedup = \frac{1}{0.6 + \frac{0.4}{8}} = \frac{1}{0.65} = 1.53$$