CS202 – Advanced Operating Systems

Concurrency

February 10, 2025

Need for Synchronization

- Activities share resources.
- It is important to coordinate their progress to ensure proper usage.

Examples of coordination

- John and Mary are each printing a different 10-page document on the same printer. You do not want their pages/output interleaving!
  – Exclusion
- John and Mary are sharing a bank account. John deposits $10. Mary should be allowed to withdraw only after this deposit

  - Ordering

Resources

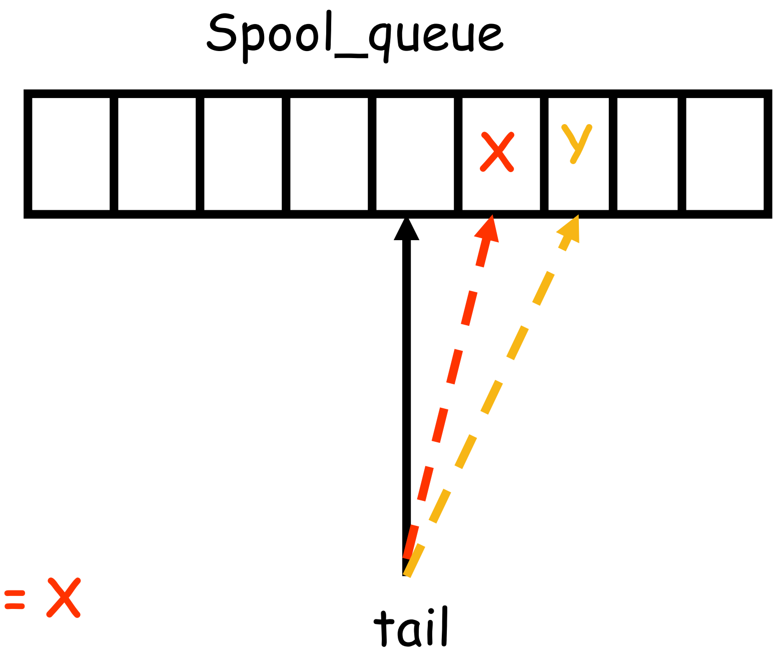
- There are multiple kinds of resources that are shared:
  - Physical (terminal, disk, network, …)
  - Logical (files, sockets, (virtual) memory, …)
- For the purposes of this discussion, let us focus on "memory" to be the shared resource
  - i.e., threads can all read and write into memory (variables) that are shared.

Problems due to sharing

- Consider a shared printer queue, spool_queue[N]
- 2 threads want to enqueue an element each to this queue.
- tail points to the current end of the queue
- Each thread needs to do

```
tail = tail + 1;
spool_queue[tail] = “element”;
```

What we are trying to do …



Thread 1

```
tail = tail + 1;
Spool_queue[tail] = X
```

Thread 2

```
tail = tail + 1;
Spool_queue[tail] =  Y
```

What is the problem?

- tail = tail + 1 is NOT one machine instruction
- It can translate as follows:

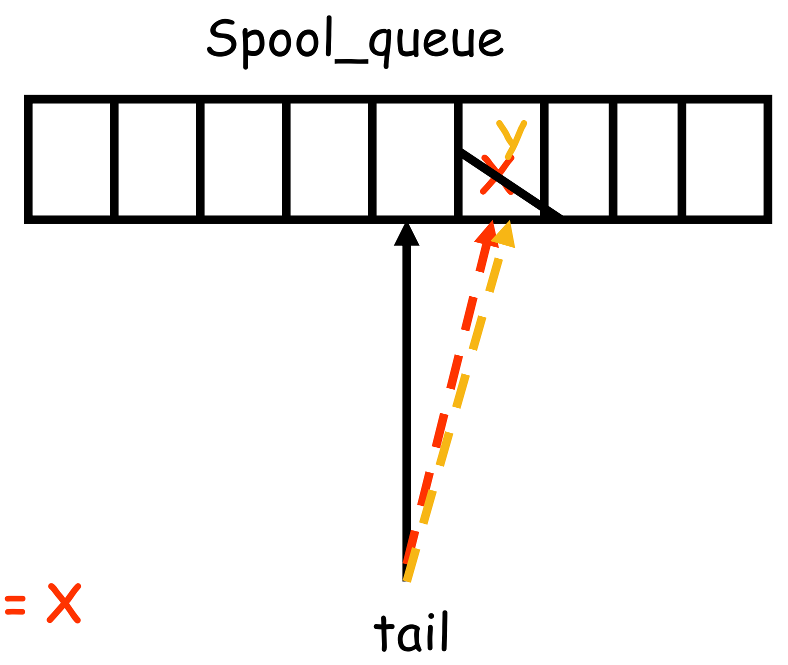
```
Load tail, R1
Add R1, 1, R2
Store R2, tail
```

- These 3 machine instructions may NOT be executed atomically.

- If each thread is executing this set of 3 instructions, context switching can happen at any time.
- Let us say we get the following resultant sequence of instructions being executed:

```
1: Load tail, R1
2: Load tail, R1
1: Add R1, 1, R2
2: Add R1, 1, R2
1: Store R2, tail
2: Store R2, tail
```

Leading to …



Thread 1

```
tail = tail + 1;
Spool_queue[tail] = X
```

Thread 2

```
tail = tail + 1;
Spool_queue[tail] =  Y
```

- Situations like this that can lead to erroneous execution, depending on who executes when, are called race conditions.

- Debugging race conditions is NOT easy! since errors can be non-repeatable.

Avoiding Race Conditions

- If we had a way of making those (3) instructions atomic – i.e., while one thread is executing those instructions, another cannot execute the same instructions – then we could have avoided the race condition.
- These 3 instructions are said to constitute a critical section.
- While one thread is in a critical section, another should NOT be allowed to execute the same critical section – mutual exclusion.

IMPLEMENT MUTUAL EXCLUSION: CONCEPTS

Implementing Mutual Exclusion

- Overall philosophy: Keep checking some state until it indicates other threads are not in critical section.
- However, this is a non-trivial problem.

```
locked = FALSE;
```

```
T1 {

while (locked == TRUE)
    ;
locked = TRUE;

/*************
(critical section code)
/*************

locked = FALSE;
}
```

```
T2 {

while (locked == TRUE)
    ;
locked = TRUE;

/*************
(critical section code)
/*************

locked = FALSE;
}
```

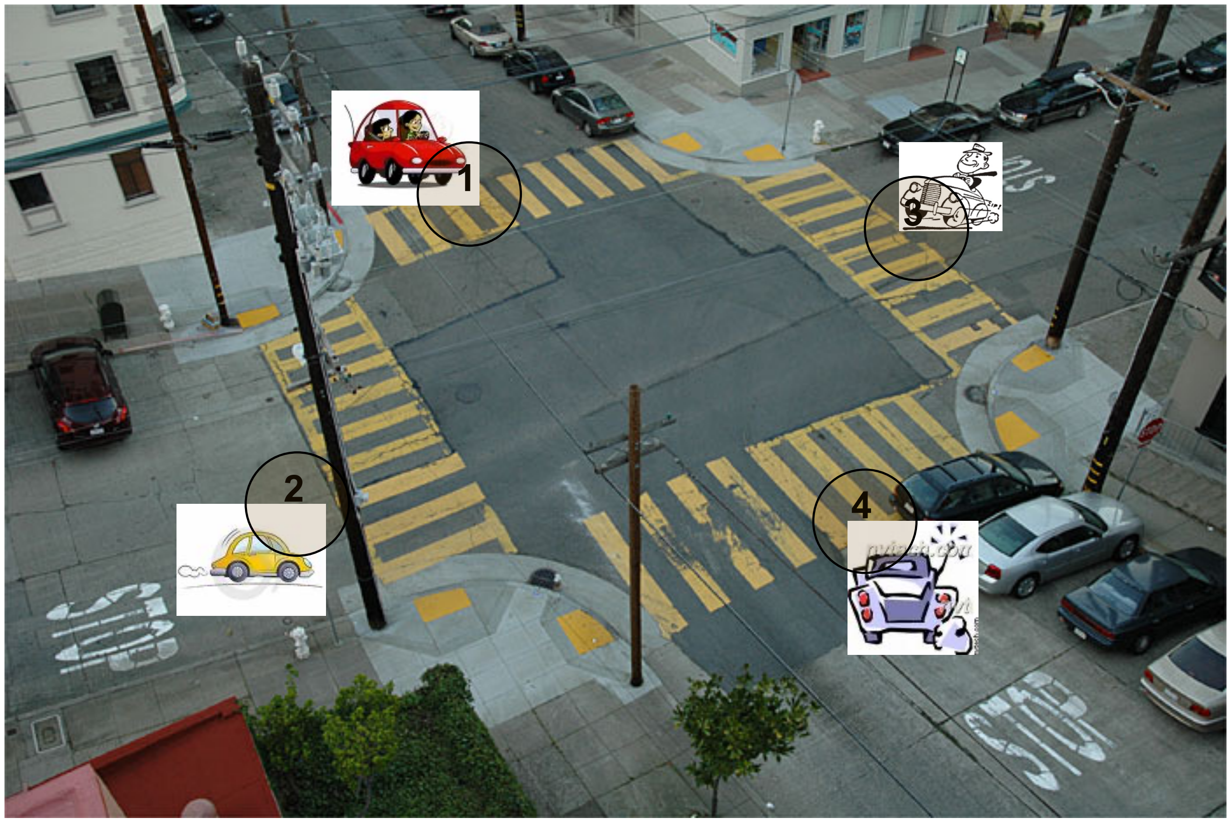
We have a race condition again since there is a gap between detection locked is FALSE, and setting locked to TRUE.

Desirables

- Should achieve mutual exclusion – Correctness
- A process that requests to access a critical section should not be blocked if no one else is using the critical section – Progress
- No process should have to wait forever to enter its critical section – Bounded Waiting

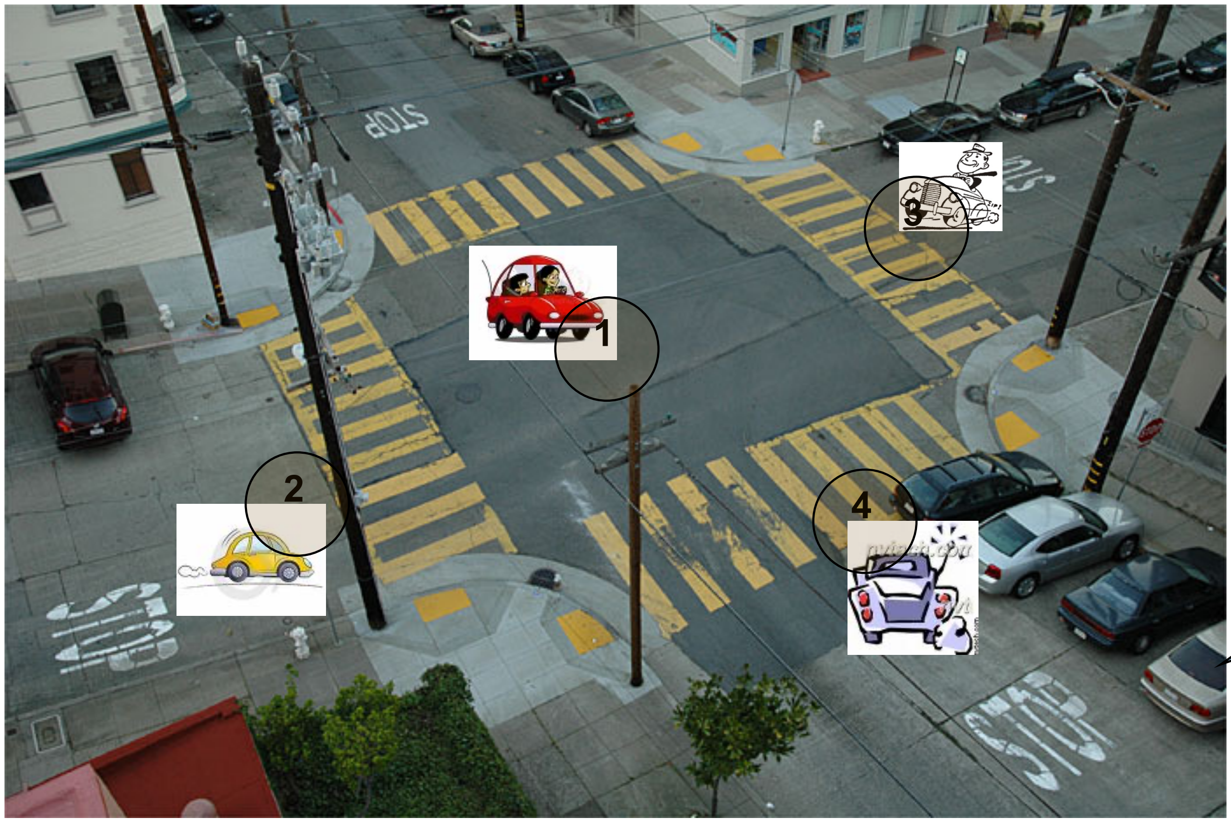
Mutual Exclusion: Traffic Analogy

If process P_i is executing in its critical section, then no other processes can be executing in their critical sections



Mutual Exclusion: Traffic Analogy

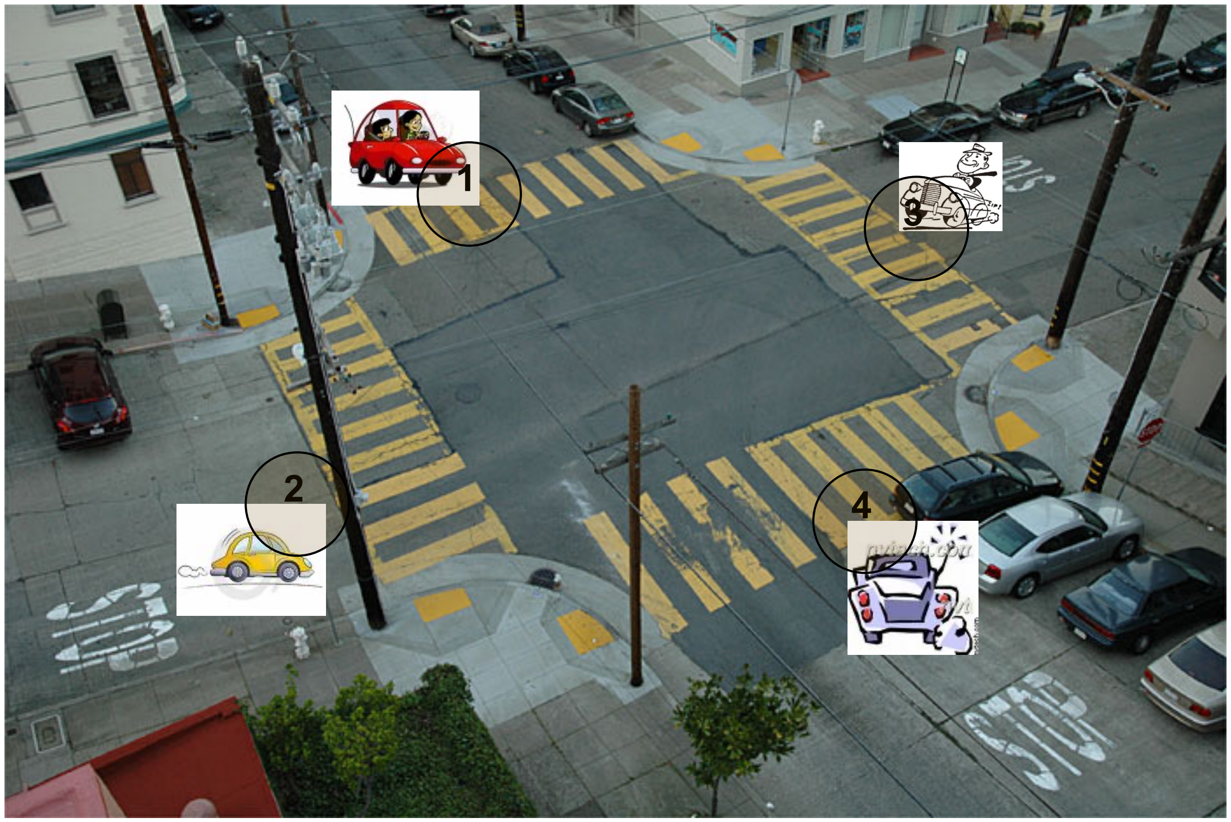
If process P_i is executing in its critical section, then no other processes can be executing in their critical sections



Only one car can be in the square area between the stop signs at a given time

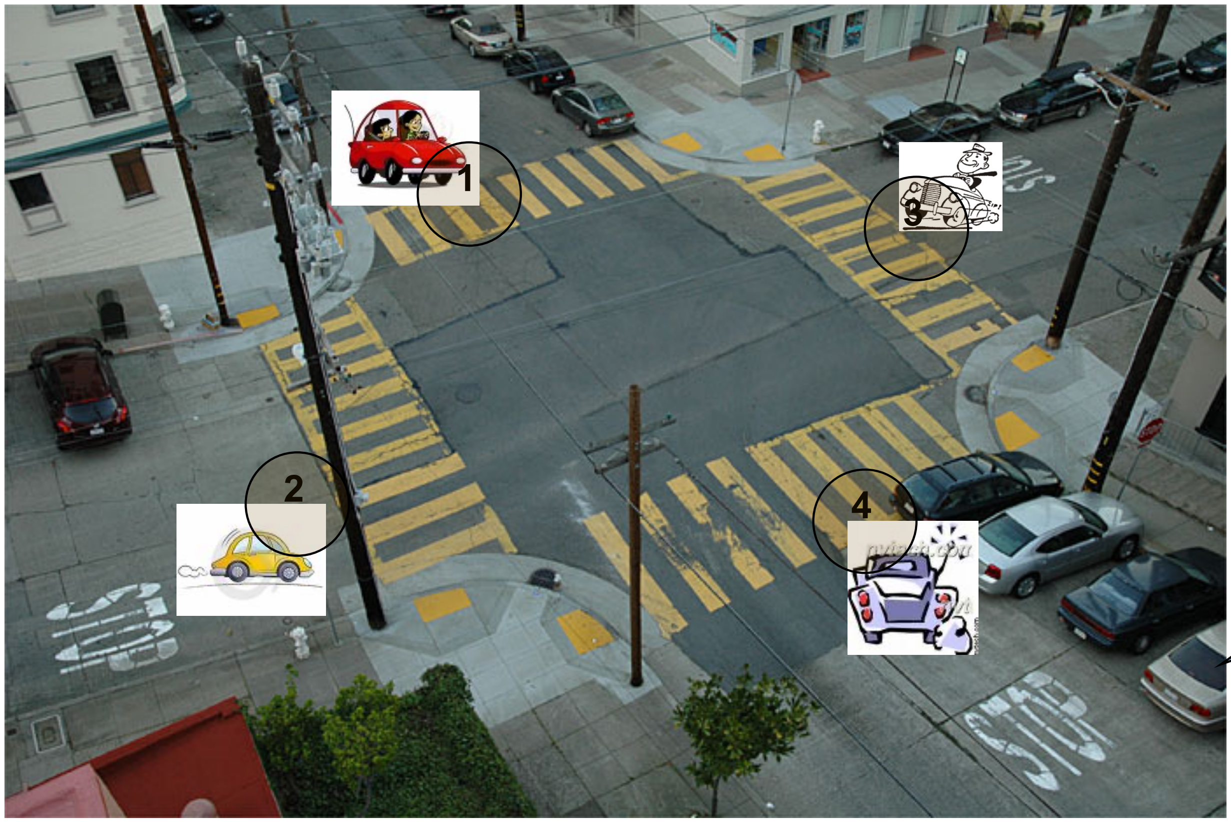
Progress: Traffic Analogy

If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely



Progress: Traffic Analogy

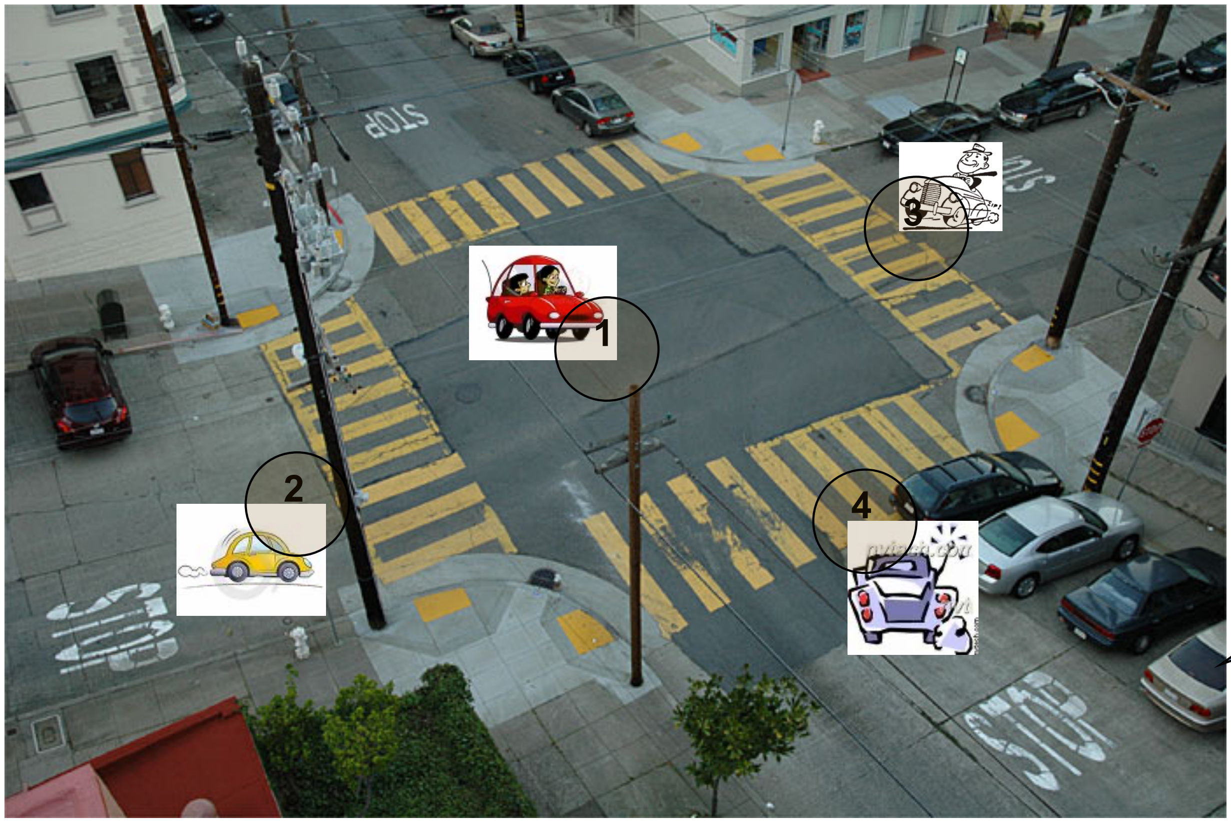
If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely



The first vehicle to the intersection has the right of way. In the event of a tie, the vehicle to the right has the right of way.

Progress: Traffic Analogy

If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely

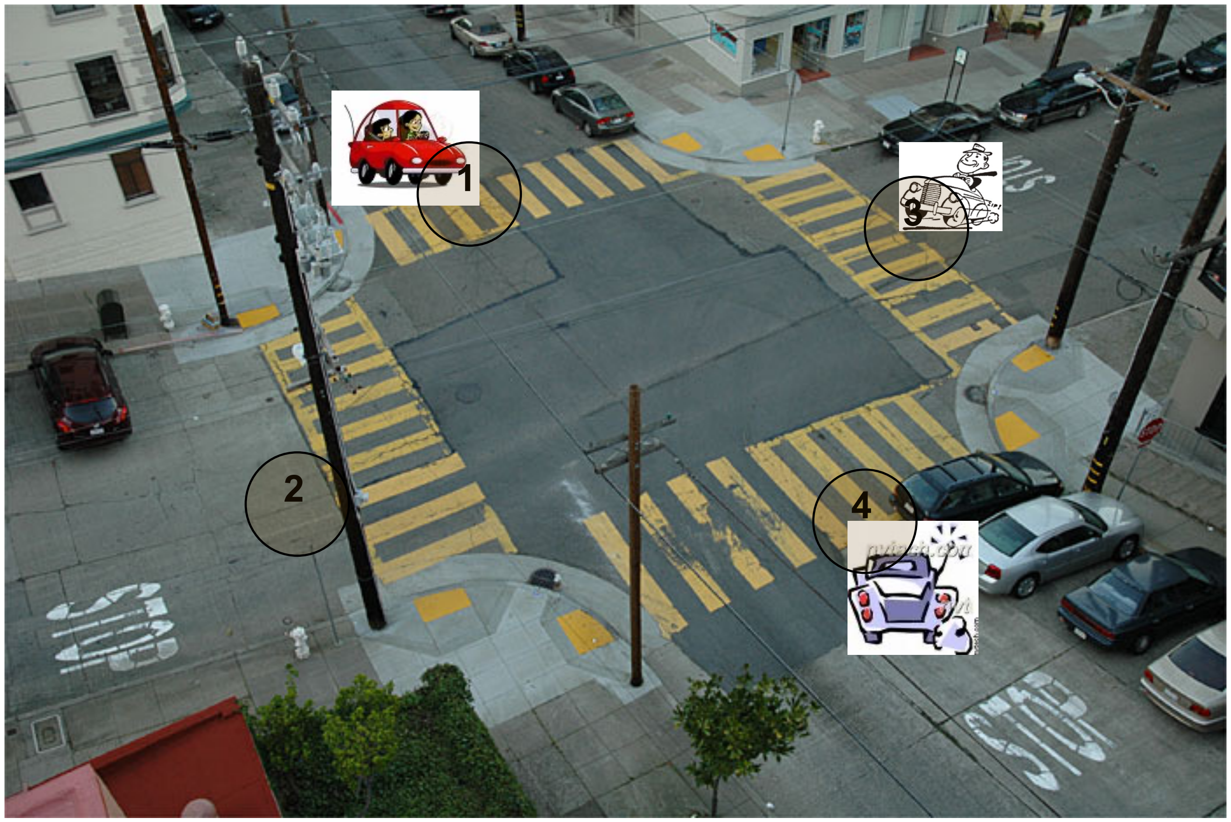


Still need to break a tie if all reach the intersection at the same time.
Coin flip?

Bounded Waiting: Traffic Analogy

A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted

- Assume that each process executes at a nonzero speed
- No assumption concerning relative speed of the N processes



Any car has to wait at most for 3 others to pass after it stops (fairness)

IMPLEMENT MUTUAL EXCLUSION: SPINNING

Algorithms

- Describe the requirements to implement mutual exclusion
  - Satisfying progress and bounded waiting as well
- Understanding these algorithms can be like solving a puzzle
  - Note: "`while (x);`" means we "spin" waiting for `x` to become false – implementing spin locks
- Also, hardware implementations of atomic operations, such as "test & set" (see Chapter 28)

1. Strict Alternation

```
turn = 0;

T0 {
  while (turn != 0);
  /**********/
  critical section
  /**********/
  turn = 1;
}

T1 {
  while (turn != 1);
  /**********/
  critical section
  /**********/
  turn = 0;
}
```

It works!

Problems:

- requires threads to alternate getting into CS
- does NOT meet Progress requirement.

Fixing the “progress” requirement

```
bool flag[2]; // initialized to FALSE

T0 {
  flag[0] = TRUE;
  while (flag[1] == TRUE)
     ;
  /* critical section */
  flag[0] = FALSE;
}

T1 {
  flag[1] = TRUE;
  while (flag[0] == TRUE)
     ;
  /* critical section */
  flag[1] = FALSE;
}
```

Problem:
  Both can set their flags to true and wait indefinitely for the other

2. Peterson's Algorithm

```
int turn;
int interested[N];          /* all set to FALSE initially */

enter_CS(int myid) {   /* param. is 0 or 1 based on T0 or T1 */
        int other;

        otherid = 1 – myid; /* id of the other thread*/
        interested[myid] = TRUE;
        turn = myid;
        while (turn == myid && interested[otherid] == TRUE)
            ;
}

leave_CS(int myid) {
        interested[myid] = FALSE;
}
```

Intuitively …

- This works because a thread can enter CS, either because
  - **Other thread is not even interested in critical section**
  - **Or even if the other thread is interested, this process did the “turn = myid” first.**

Prove that

- It is correct (achieves mutex)
  - If both are in critical section, then 1 condition at least must have been false for both threads.
  - This has to be the “turn == myid” which cannot be false for both threads.

Prove that

- There is progress
  - If a thread is waiting in the loop, the other person has to be interested.
  - One of the two will definitely get in during such scenarios.

Prove that

- There is bounded waiting
  - When there is only one thread interested, it gets through
  - When there are two threads interested, the first one that did the “turn = myid” statement goes through.
  - When the current thread is done with CS, the next time it requests the CS, it will get it only after any other thread waiting at the loop.

- We have looked at only 2 thread solutions.
- How do we extend for multiple threads?

More than 2 thread solution

- Analogy to serving different customers in some serial fashion.
  - **Make them pick a number/ticket on arrival.**
  - **Service them in increasing tickets**
  - **Need to use some tie-breaker in case the same ticket number is picked (e.g. larger thread id wins).**

3. Bakery Algorithm

Notation: (a,b) < (c,d) if a<c or a=c and b<d

Every thread has a unique id (integer) Pi

```
bool choosing[0..n-1];
int number[0..n-1];

enter_CS(myid) {
        choosing[myid] = TRUE;
        number[myid] = max(number[0],number[1], .... number[n-1]) + 1;
        choosing[myid] = FALSE;
        for (j=0 to n-1) {
                while (choosing[j])
                        ;
                while (number[j] != 0) && ((number[j])<(number[myid]) ||
                        (((number[j])==(number[myid]) && (myid <= Pj)))
                        ;
        }
}
leave_CS(myid) {
        number[myid] = 0;
}
```

CLASSIC SYNCHRONIZATION PROBLEMS: BLOCKING

- In blocking solutions, you relinquish the CPU at the time you cannot proceed, i.e., you are put in the blocked queue.
- It is the job of the activity changing the condition to wake you up (i.e., move you from blocked back to ready queue).
- This way you do not unnecessarily occupy CPU cycles.

Example Blocking Implementation

```
Mutex_Lock(L) {
   Disable Interrupts/Use Spinlock
   Check if anyone is using L
   If not {
        Set L to being used
   }
   else {
        Move this TCB to Blocked
             queue for L
        Select another activity to run
             from Ready queue
        Context switch to that activity
   }
   Enable Interrupts/Use Spinlock
}
```

```
Mutex_Unlock(L) {
   Disable Interrupts/Use spinlock
   if (blocked queue of L == NULL)
        Set L to free
   }
   else {
        Move TCB from head of
             Blocked queue of L to
             Ready queue
   }
   Enable Interrupts/Use spinlock
}
```

NOTE: These are OS system calls (where Disable/Enable are available), or Library calls (where Spinlocks are only option for implementation)

Until now …

- Exclusion synchronization/constraint
  - **Typical construct mutual exclusion lock**
    - **Mutex_lock(m)**
    - **Mutex_unlock(m)**
  - Do a “man” on pthread_mutex_lock() for further syntactic/semantic information.

Classic Synchronization Problems

- Bounded-buffer problem
- Readers-writers problem
- Dining Philosophers problem
- ….

- We will compose solutions using semaphores
- See "Little Book of Semaphores" for context

Mutex + Counting

- A lot of concurrency problems require tracking counts as conditions
  - Can we design a primitive that implements mutual exclusion while tracking counts?
- Yes – they are called semaphores
  - After the visual signal method called “semaphores”

Semaphores

- You are given a data-type Semaphore_t.
- On a variable of this type, you are allowed
  - P(Semaphore_t) -- wait
  - V(Semaphore_t) – signal
- Intuitive Functionality:
  - Logically one could visualize the semaphore as having a counter initially set to 0.
  - When you do a P(), you decrement the count, and need to block if the count becomes negative.
  - When you do a V(), you increment the count and you wake up 1 process from its blocked queue if not null.

Semaphore Implementation

```
typedef struct {
   int value;
   struct TCB *L;
} semaphore_t;
```

```
void P(semaphore_t S) {
   Disable Interrupts/Use spinlock
   S.value--;
   if (S.value < 0) {
      add this thread to S.L and
         remove from ready queue
      context switch to another
   }
   Enable Interrupts/Use spinlock
}
```

```
void V(semaphore_t S) {
   Disable Interrupts/Use Spinlock
   S.value++;
   if (S.value <= 0) {
      remove a thread from S.L
      put it in ready queue
   }
   Enable Interrupts/Use Spinlock
}
```

Semaphores can implement mutex

```
Semaphore_t m;    // Initialize its count/value to 1

Mutex_lock() {
   P(m);
}

Mutex_unlock() {
   V(m);
}
```

Bounded Buffer problem

- A queue of finite size implemented as an array.
- You need mutual exclusion when adding/removing from the buffer to avoid race conditions
- Also, you need to wait when appending to buffer when it is full or when removing from buffer when it is empty.

Bounded Buffer using Semaphores

```
int BB[N];
int count, head, tail = 0;
Semaphore_t m; // value initialized to 1
Semaphore_t notfull; // value initialized to N
Semaphore_t notempty; // value initialized to 0
```

```
Append(int elem) {
   P(notfull);
   P(m);

   BB[tail] = elem;
   tail = (tail + 1)%N;
   count = count + 1;

   V(m);
   V(notempty);
}
```

```
int Remove () {
   P(notempty);
   P(m);

   int temp = BB[head];
   head = (head + 1)%N;
   count = count - 1;

   V(m);
   V(notfull);
   return(temp);
}
```

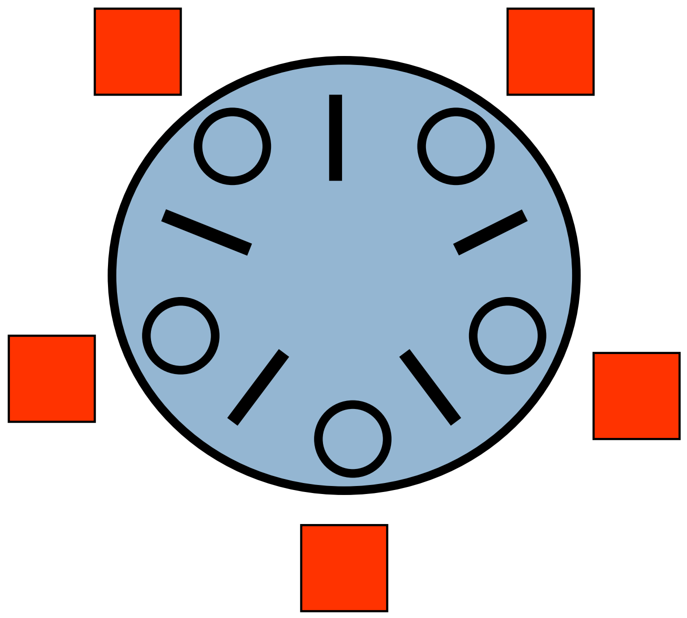
Readers-Writers Problem

- There is a database to which there are several readers and writers.
- The constraints to be enforced are:
  - **When there is a reader accessing the database, there could be other readers concurrently accessing it.**
  - **However, when there is a writer accessing it, there cannot be any other reader or writer.**

Readers-writers using Semaphores

```
Database db;
int nreaders = 0;
Semaphore_t m; // value initialized to 1
Semaphore_t wrt; // value initialized to 1

Reader() {
   P(m);
   nreaders++;
   if (nreaders == 1) P(wrt);
   V(m);

   .... Read db here ...

  P(m);
  nreaders--;
  if (nreaders == 0) V(wrt);
  V(m);
}

Writer() {
      P(wrt);

      ... Write db here ...

      V(wrt);
}
```

Dining Philosophers Problem



Philosophers alternate between thinking and eating.

When eating, they need both (left and right) chopsticks.

A philosopher can pick up only 1 chopstick at a time.

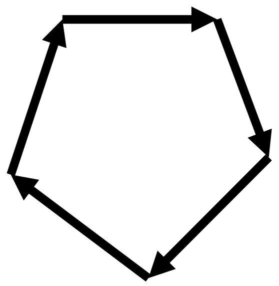
After eating, the philosopher puts down both chopsticks.

```
Semaphore_t chopstick[5];

Philosopher(i) {
  while () {
      P(chopstick[i]);
      P(chopstick[(i+1)%5];

      … eat …

      V(chopstick[i]);
      V(chopstick[(i+1)%5];

      … think …
  }
}
```

This is NOT correct!

Though no 2 philosophers use the same chopstick at any time, it can so happen that they all pick up 1 chopstick and wait indefinitely for another.

This is called a deadlock,



- Note that putting

  P(chopstick[i]);

  P(chopstick[(i+1)%5];

  within a critical section (using say P(mutex) / V(mutex)) can avoid that deadlock.
- But then,
  - only 1 philosopher can pick up at a time (even if you don't depend on philosophers across the table).
  - Can still deadlock if you hold the mutex and block for a chopstick

```
int state[N];
Semaphore_t s[N]; // init. to 0
Semaphore_t mutex; // init. to 1

#define LEFT     (i-1)%N
#define RIGHT  (i+1)%N

philosopher(i) {
   while () {
            take_chopsticks(i);
            eat();
            put_chopsticks(i);
            think();
   }
}
```

```
take_chopsticks(i) {
            P(mutex);
            state[i] = HUNGRY;
            test(i);
            V(mutex);
            P(s[i]);
}

put_chopsticks(i) {
            P(mutex);
            state[i] = THINKING;
            test(LEFT);
            test(RIGHT);
            V(mutex);
}
```

```
test(i) { /* can phil i eat? if so, signal that philosopher */
            if (state[i] == HUNGRY &&
                state[LEFT] != EATING && state[RIGHT] != EATING) {
                        state[i] = EATING;
                        V(s[i]);
            }
}
```

- But you also need synchronization constructs for an arbitrary condition to become true
  - E.g., If printer queue is full, I need to wait until there is at least 1 empty slot
  - Note that mutex_lock()/mutex_unlock() are not very suitable to implement such synchronization
    - You can use semaphores in this case, but not for any condition

Condition Variables

- C_wait() and c_signal() operations.
- A thread blocked on c_wait() returns when another performs a c_signal().

```
C_wait(C) {
   Disable Interrupts /Use spinlock
   Move this TCB to Blocked queue for C
   Select another thread to run
             from Ready queue
   Enable Interrupts /Use spinlock
   Context switch to that thread
}
```

```
C_signal(C) {
   Disable Interrupts/Use spinlock
   Check if blocked queue
      for L is empty
   else {
       Move TCB from head of
            Blocked queue of C to
            Ready queue
   }
   Enable Interrupts/Use spinlock
}
```

```
Cond_t  not_full, not_empty;;
Int count == 0;

Append() {
            if count == N   c_wait(not_full);

            ... ADD TO BUFFER, UPDATE COUNT ...

            c_signal(not_empty);
}

Remove() {
            if count == 0       c_wait(not_empty);

            ... REMOVE FROM BUFFER, UPDATE COUNT

            c_signal(not_full);
}
```

However, there is something wrong with this code!

There is a gap between checking (count == N) and c_wait()!

Similarly, for Remove.

Solution: Put c_wait() within a mutex_lock()

```
Cond_t  not_full, not_empty;
Mutex_lock m;
Int count == 0;

Append() {
            mutex_lock(m);
            if count == N   c_wait(not_full,m);

            … ADD TO BUFFER, UPDATE COUNT …

            c_signal(not_empty);
            mutex_unlock(m);
}

Remove() {
            mutex_lock(m);
            if count == 0   c_wait(not_empty,m);

            … REMOVE FROM BUFFER, UPDATE COUNT

            c_signal(not_full);
            mutex_unlock(m);
}
```

C_wait(c,m) : You give up “m” before waiting,

and you regain “m” when you are signaled.

Issue

- But, this “solution” does not really work if there are two threads that run “remove” (or “append”)
  - Can you identify why not?

Solution: Put c_wait() within a mutex_lock()

```
Cond_t  not_full, not_empty;
Mutex_lock m;
Int count == 0;

Append() {
        mutex_lock(m);
        if count == N   c_wait(not_full,m);

        … ADD TO BUFFER, UPDATE COUNT …

        c_signal(not_empty);
        mutex_unlock(m);
}

Remove() {
        mutex_lock(m);
        if count == 0   c_wait(not_empty,m);

        … REMOVE FROM BUFFER, UPDATE COUNT

        c_signal(not_full);
        mutex_unlock(m);
}
```

Suppose that one thread performs a “c_wait”.

But, another thread runs “remove” after a “c_signal” and **steals** the buffer element.

Three Easy Pieces

- The textbook goes through these scenarios in detail in Chapter 30

Solution: Check the condition again

```
Cond_t  not_full, not_empty;
Mutex_lock m;
Int count == 0;

Append() {
        mutex_lock(m);
        while count == N   c_wait(not_full,m);

        … ADD TO BUFFER, UPDATE COUNT …

        c_signal(not_empty);
        mutex_unlock(m);
}

Remove() {
        mutex_lock(m);
        while count == 0   c_wait(not_empty,m);

        … REMOVE FROM BUFFER, UPDATE COUNT

        c_signal(not_full);
        mutex_unlock(m);
}
```

Need to check that the condition still holds when a thread is finally scheduled.

READ-COPY UPDATE

Why are we reading this paper?

- Example of a synchronization primitive that is:
  - Lock free (mostly/for reads)
  - Tuned to a common access pattern
  - Making the common case fast
- What is this common pattern?
  - A lot of reads
  - Writes are rare
    - Prioritize writes
  - Stale copies are short lived – time heals all wounds
  - Ok to read a slightly stale copy
    - But that can be fixed too

Lock free data structures

- Do not require locks
- Good if contention is rare
- But difficult to create and error prone
- RCU is a mixture
  - Concurrent changes to pointers a challenge for lock-free
  - RCU serializes writers using locks
  - Win if most of our accesses are reads

Traditional OS locking designs

- Poor concurrency
  - Especially if mostly reads
- Fail to take advantage of event-driven nature of operating systems
- Locks have acquire and release cost
  - Use atomic operations which are expensive
  - Can dominate cost for short critical regions
  - Locks become the bottleneck

Why RCU?

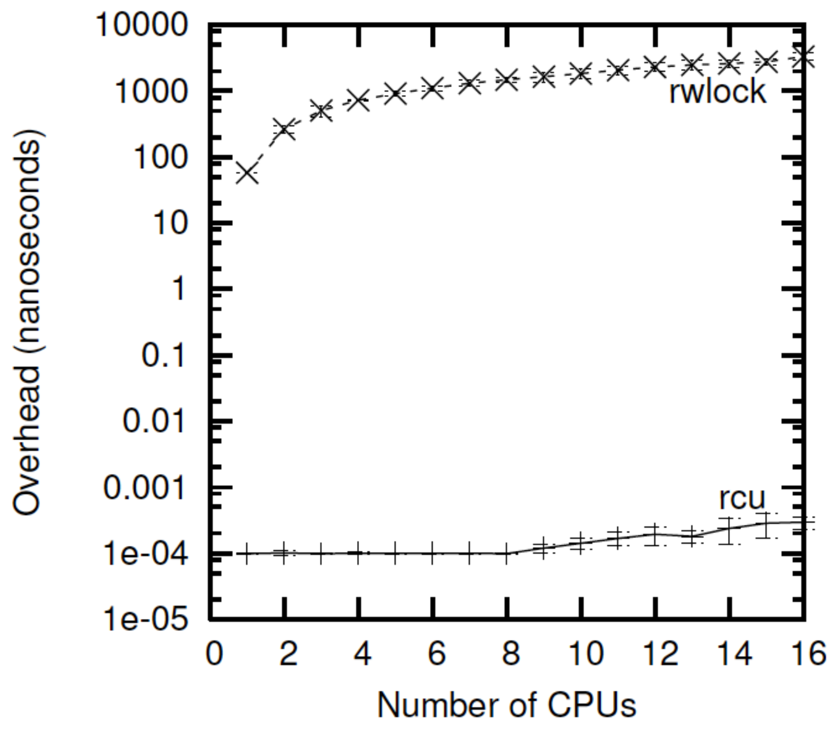


Figure 8: The overhead of entering and completing an RCU critical section, and acquiring and releasing a read-write lock.

Typical RCU update sequence

- Replace pointers to a data structure with pointers to a new version
  - Is this replacement atomic?
- Wait for all previous readers to complete their RCU read-side critical sections.
- at this point, there cannot be any readers who hold reference to the old data structure, so it now may safely be reclaimed.

Without RCU

- Operation 1A, 1B, and 2A access the module

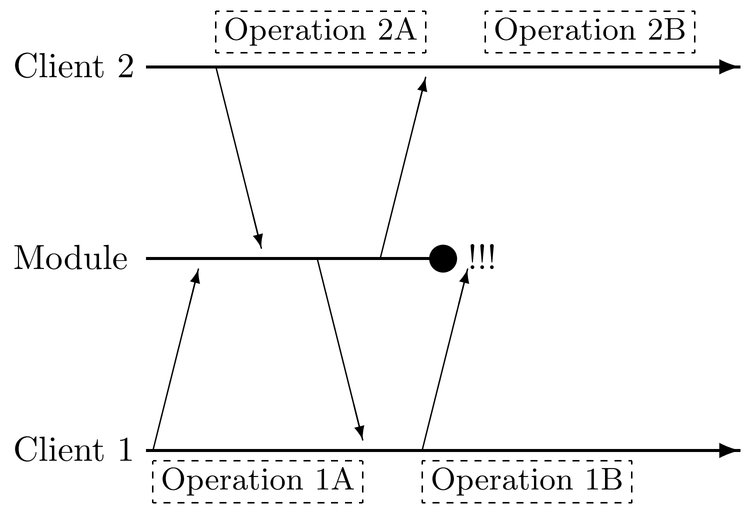


Figure 1: Race Between Teardown and Use of Service

- If the module is freed before 1B invokes – oops!

With RCU

- Operation 1A, 1B, and 2A access the module

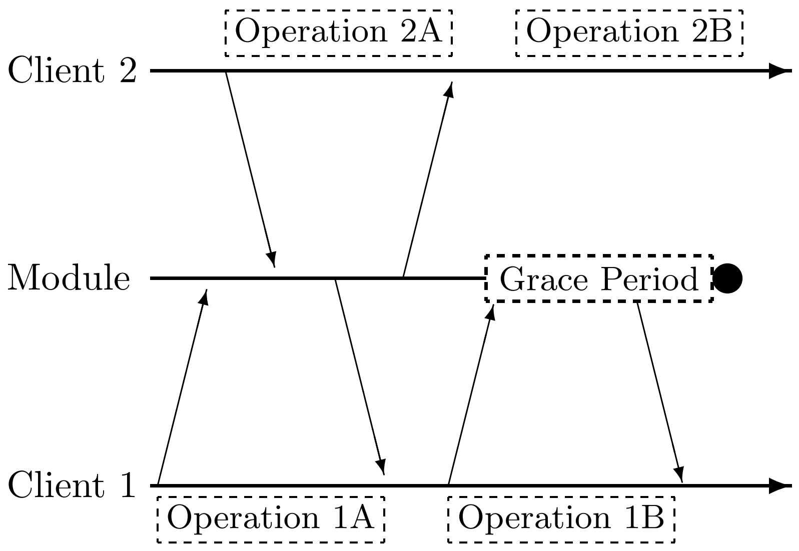


Figure 2: Read-Copy Update Handling Race

- Grace period before 1B finishes with the module
- 2B is told the module is gone – no problem

Delete implementation

```
void delete(struct el *p)
{
    spin_lock(&list_lock);
    p->next->prev = p->prev;
    p->prev->next = p->next;
    spin_unlock(&list_lock);
    call_rcu(&p->my_rcu_head,
    my_free, p);
}
```

Phase 1

Schedule phase 2
after quiescence

Figure 4: Read-Copy Dequeue From Doubly-Linked List

Read-Copy Deletion (delete B)

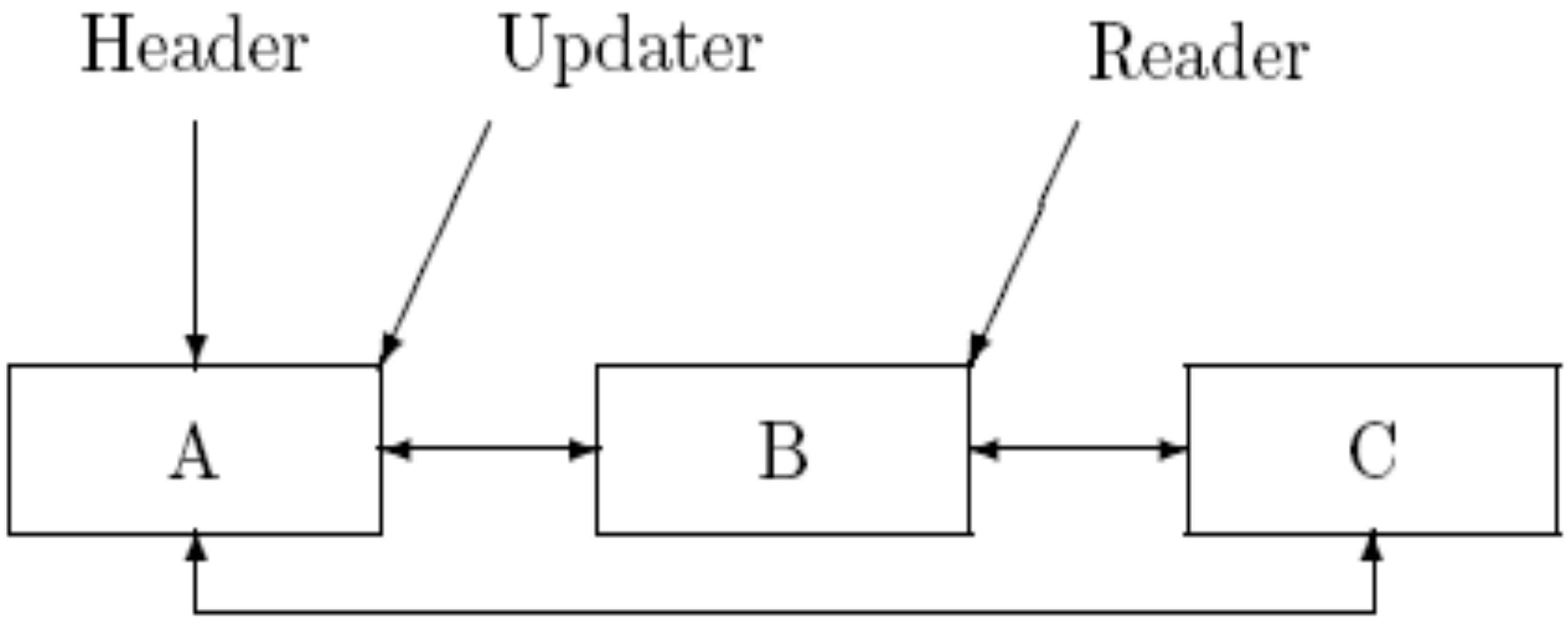


Figure 11: List Initial State

the first phase of the update

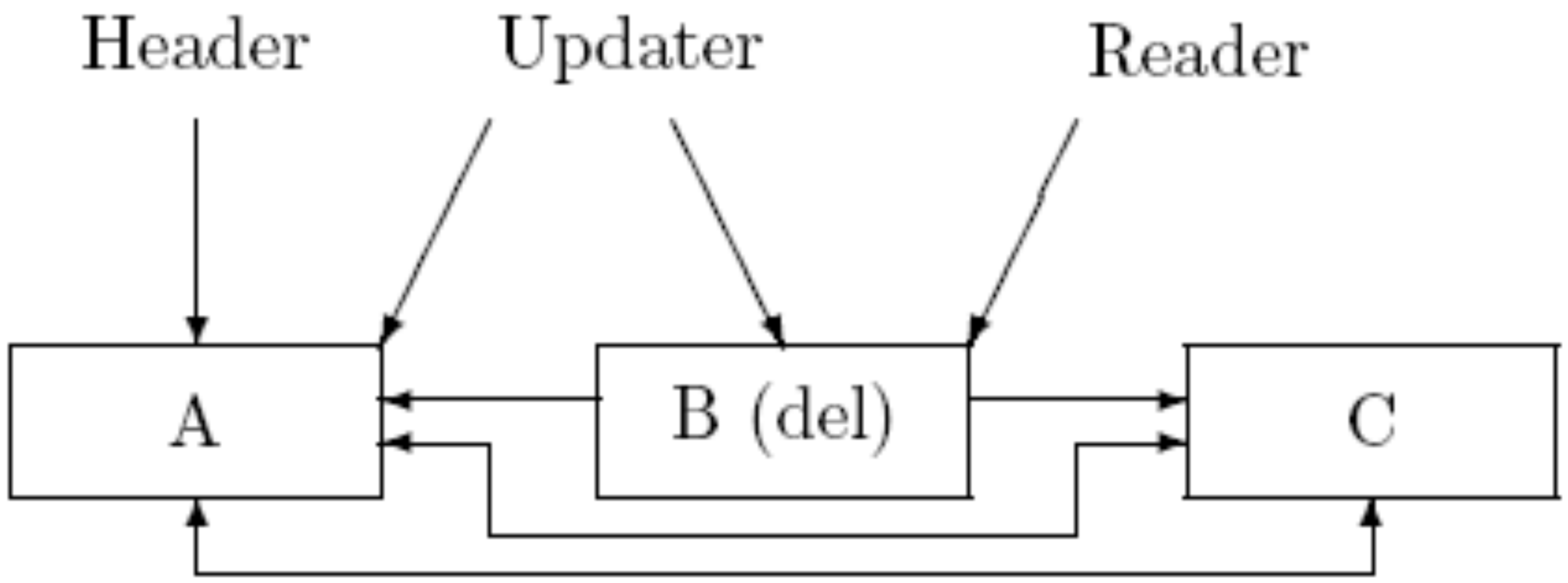


Figure 12: Element B Unlinked From List

18

Read-Copy Deletion

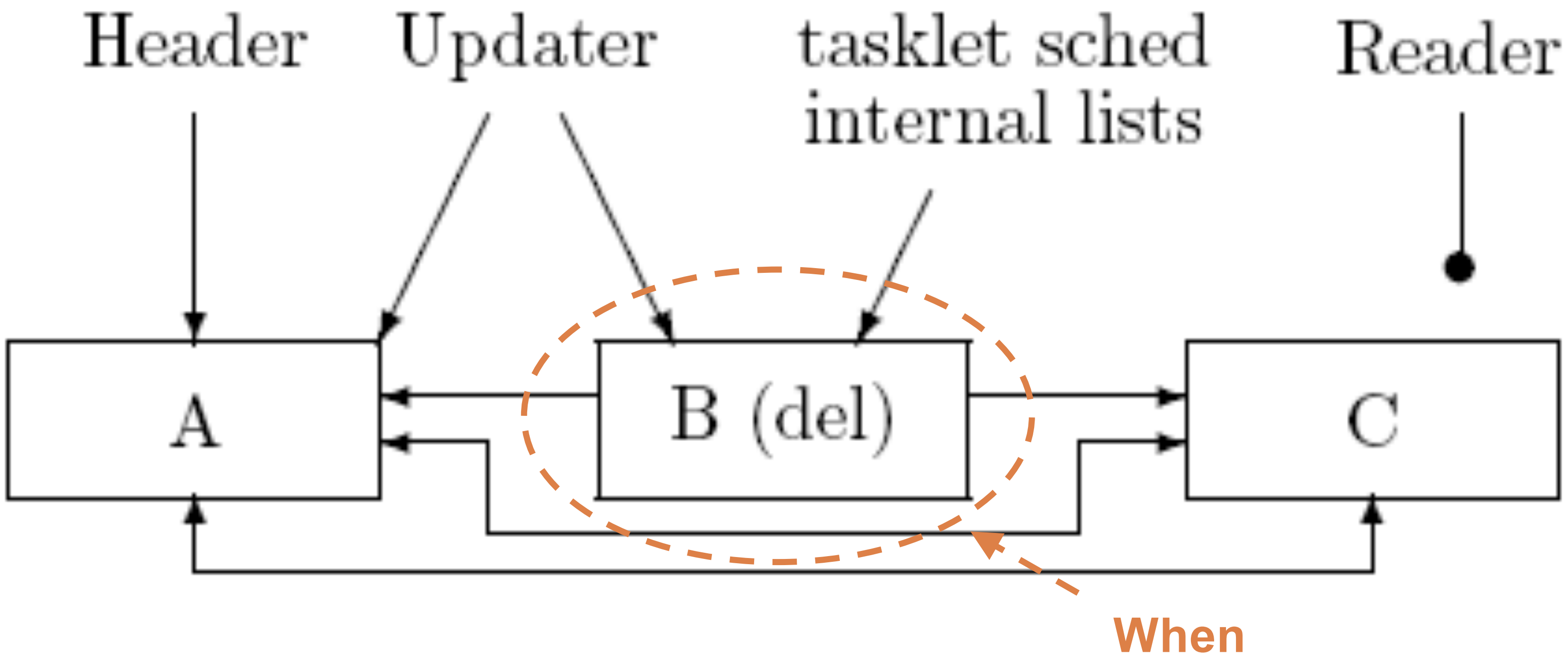


Figure 13: List After Grace Period

Implemented through the call_rcu()

Read-Copy Deletion

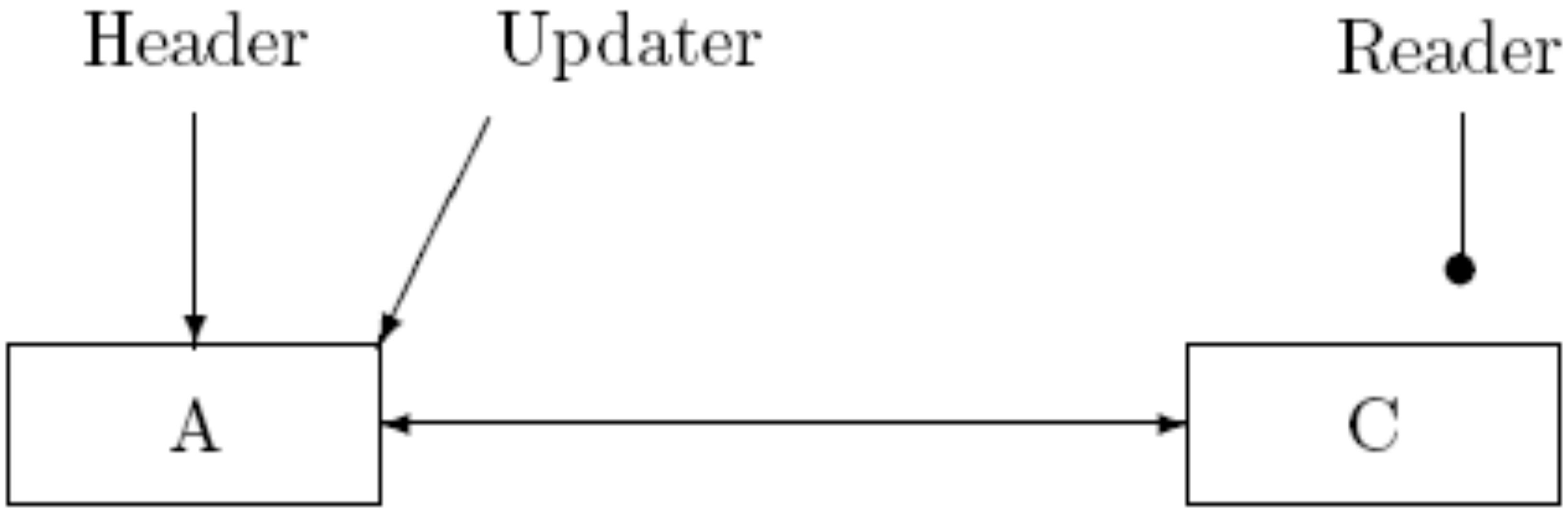


Figure 14: List After Element B Returned to Freel-
ist

How to detect quiescence?

- Idea of grace periods
  - Readers of old information will eventually leave
  - Exploit context switches
    - Threads do not hold OS locks across context switches
- How do we identify?
  - Paper goes into many alternatives and evaluates them (polling; counters; …)
  - Batching to reduce cost
  - Force context switch?
    - Expensive and some tasks are not preemptible

Discussion of paper

- Really challenging paper to read
  - Written by OS hackers (good thing!)
  - Mixes fundamentals and implementations
    - We have to try to step back and identify them
  - Too many ideas/alternatives
    - Better just to focus on one or two?
  - Back end of the paper is a survey
- What are your thoughts?

Conclusions

- The last few days have been a review of concurrency
- Programs/threads may share access to resources
  - Uncontrolled shared access can lead to race conditions
  - When can that happen?
- We review spin lock solutions and blocking solutions
  - Spin locks continue to check for a condition to be met
  - Blocking solutions (semaphores, mutex locks, condition variables) cause threads to be queued until the condition changes – may have to recheck condition
- Read-copy update is a new concurrency primitive
  - Exploits infrequent updates by delaying the effects of a change (e.g., free) until all readers of the old are ready

Questions