CS 202
Advanced Operating Systems

Winter 26

Lecture 4: Synchronization

Instructor: Chengyu Song

Need for Synchronization

- Access to share resources
  - Memory (threads, kernel)
  - OS resources: files, terminal, communication channel, network, etc.
  - Computer systems
  - Physical resources
  - …
- It is important to coordinate their progress to ensure proper outcome

One Example

- Suppose we have to implement a function to handle withdrawals from a bank account:

```
withdraw (account, amount) {
        balance = get_balance(account);
        balance = balance – amount;
        put_balance(account, balance);
        return balance;
}
```

- Now suppose that you and your father share a bank account with a balance of $1000
- Then you each go to separate ATM machines and simultaneously withdraw $100 from the account

Example Continued

- We'll represent the situation by creating a separate thread for each person to do the withdrawals
- These threads run on the same bank machine:

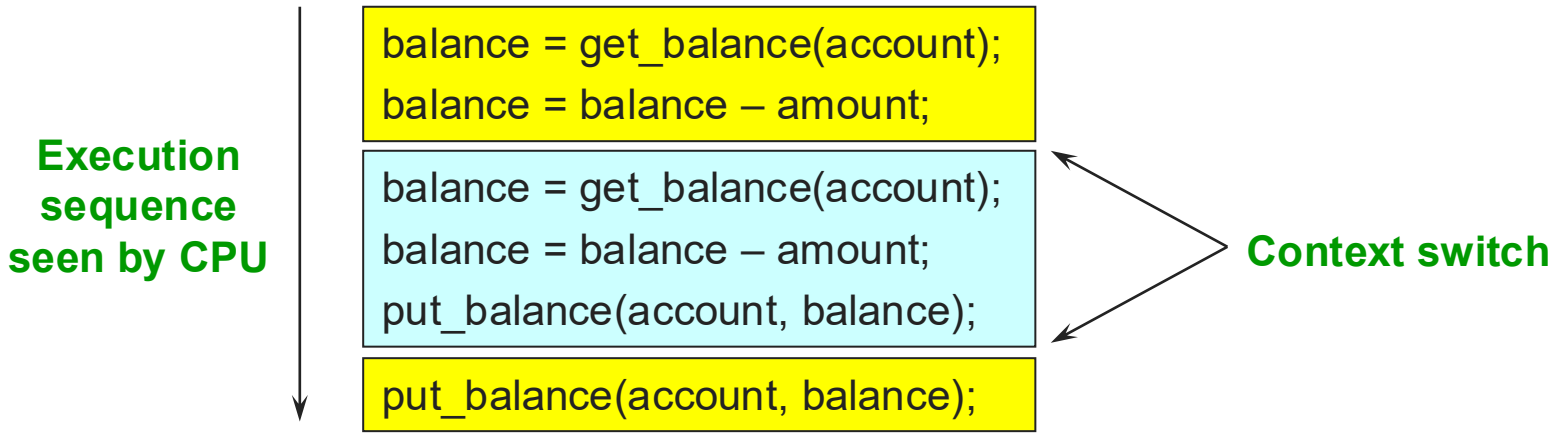
```
withdraw (account, amount) {
    balance = get_balance(account);
    balance = balance – amount;
    put_balance(account, balance);
    return balance;
}
```

```
withdraw (account, amount) {
    balance = get_balance(account);
    balance = balance – amount;
    put_balance(account, balance);
    return balance;
}
```

- What's the problem with this implementation?
  - Think about potential schedules of these two threads

Interleaved Schedules

- The problem is that the execution of the two threads can be interleaved:

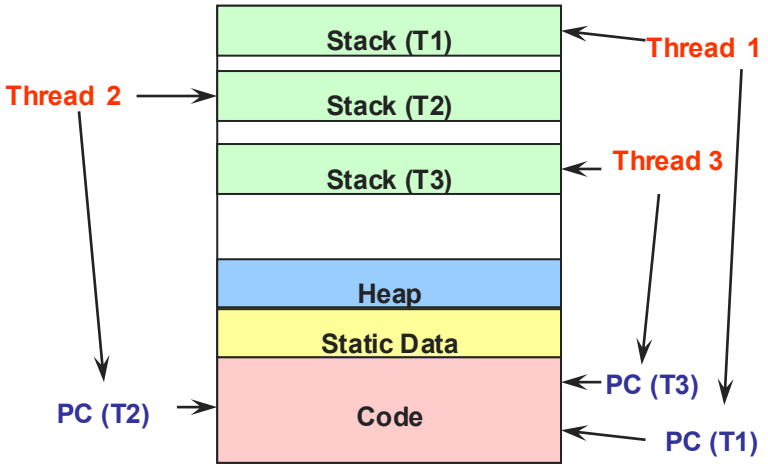


- What is the balance of the account now?

Shared Resources

- Problem: two or more threads accessed a shared resource (and the outcome depends on the order of their executions)
  - Known as a **race condition** (remember this buzzword!)
- Need mechanisms to control this access
  - So we can reason about how the program will operate
- Our example was updating a shared bank account
- Also necessary for synchronizing access to any shared data structure
  - buffers, queues, lists, hash tables, etc.

What Resources Are Shared?

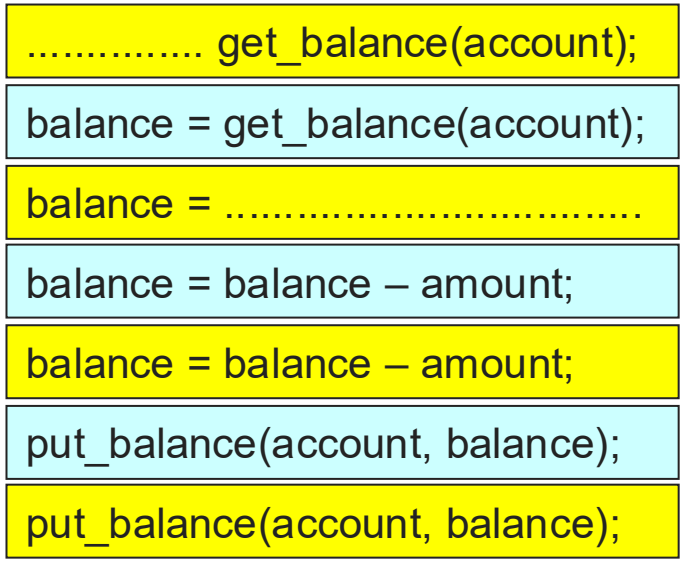


- Local variables?
  - Not shared: refer to data on the stack
  - Each thread has its own stack
  - Don't pass/share/store a pointer to a local variable on the stack for thread T1 to another thread T2
- Global variables and static objects?
  - Shared: in static data segment, accessible by all threads
- Dynamic objects and other heap objects?
  - Shared: Allocated from heap with malloc/free or new/delete

How Interleaved Can It Get?

How contorted can the interleaving be?

- We'll assume that the only atomic operations are reads and writes of individual memory locations
  - Some architectures don't even give you that!
- We'll assume that a context switch can occur at any time
- We'll assume that you can delay a thread as long as you like as long as it's not delayed forever



Threads: Sharing memory

```
int count = 0;

void twiddledee() {
  int i=0;
  for (i=0; i<2; i++) {
    count = count * count;
}}

void twiddledum() {
  int i=0;
  for(i=0; i<2; i++) { count = count - 1;}
}

void main() {
  thread_fork(twiddledee);
  thread_fork(twiddledum);
  print count;
}
```

- What are all the values that could be printed in main?

Behind the Scene

```
# count = count * count;
    mov  eax, DWORD PTR count[rip]   # read count from mem
    imul eax, eax                    # calculate new count
    mov  DWORD PTR count[rip], eax   # write count to mem

# count = count - 1;
    mov  eax, DWORD PTR count[rip]   # read count from mem
    sub  eax, 1                      # calculate new count
    mov  DWORD PTR count[rip], eax   # write count to mem
```

Atomic Operations

```
# count = count * count; <- not atomic
    mov  eax, DWORD PTR count[rip]   # read count from mem, atomic
    imul eax, eax                    # calculate new count, atomic
    mov  DWORD PTR count[rip], eax   # write count to mem, atomic

# count = count - 1; <- not atomic
    mov  eax, DWORD PTR count[rip]   # read count from mem, atomic
    sub  eax, 1                      # calculate new count, atomic
    mov  DWORD PTR count[rip], eax   # write count to mem, atomic
```

Operations whose semantic will always be executed as a whole and cannot be interrupted

Thread: Sharing memory

```
int count = 0; //shared variable since its global

void twiddledee() {
  int i=0; //not shared local variable
  for (i=0; i<2; i++) {
    count = count * count;
}}

void twiddledum() {
  int i=0; //not shared local variable
  for(i=0; i<2; i++) { count = count - 1;}
}

void main() {
  thread_fork(twiddledee);
  thread_fork(twiddledum);
  print count;
}
```

- What operations are atomic?
- What variable(s) are shared?
- What variables are not shared?

- What are all the values that could be printed in main?

What do we do about it?

- Does this problem matter in practice?
- Are there other concurrency problems?
- And, if so, how do we solve it?
  - Really difficult because behavior can be different every time
- How do we handle concurrency in real life?

Synchronization

- For correctness, we need to control this cooperation
  - Threads interleave executions arbitrarily and at different rates
  - Scheduling is not under program control

- We control cooperation using synchronization
  - Synchronization enables us to restrict the possible inter-leavings of thread executions

What about processes?

- Does this apply to processes too?
  - Yes!
- What synchronization system call you have seen?
  - `wait()`
- Do I need to learn this if I don't write multi-thread programs?
  - But share the OS structures and machine resources so we need to synchronize them too
  - Basically, the OS is a multi-threaded program

Mutual Exclusion

- Mutual exclusion to synchronize access to shared resources
  - This allows us to have larger “atomic” blocks
- Code that uses mutual called a critical section
  - Only one thread at a time can execute in the critical section
  - All other threads are forced to wait on entry
  - When a thread leaves a critical section, another can enter
  - Example: sharing an ATM with others
- What requirements would you place on a critical section?

Critical Section Requirements

1) Mutual exclusion (mutex)
   - If one thread is in the critical section, then no other is
2) Progress
   - A thread in the critical section will eventually leave the critical section
   - If some thread T is not in the critical section, then T cannot prevent some other thread S from entering the critical section
3) Bounded waiting (no starvation)
   - If some thread T is waiting on the critical section, then T will eventually enter the critical section
4) Performance
   - The overhead of entering and exiting the critical section is small with respect to the work being done within it

About Requirements

There are three kinds of requirements that we'll use

- Safety property: nothing bad happens
  - Mutex
- Liveness property: something good happens
  - Progress, Bounded Waiting
- Performance requirement
  - Performance
- Properties hold for each run, while performance depends on all the runs
  - Rule of thumb: When designing a concurrent algorithm, worry about safety first, but don't forget liveness!

Mechanisms For Building Critical Sections

- Locks
  - Primitive, minimal semantics, used to build others
- Architecture help
  - Atomic test-and-set
- Semaphores
  - Basic, easy to get the hang of, but hard to program with
- Monitors
  - High-level, requires language support, operations implicit

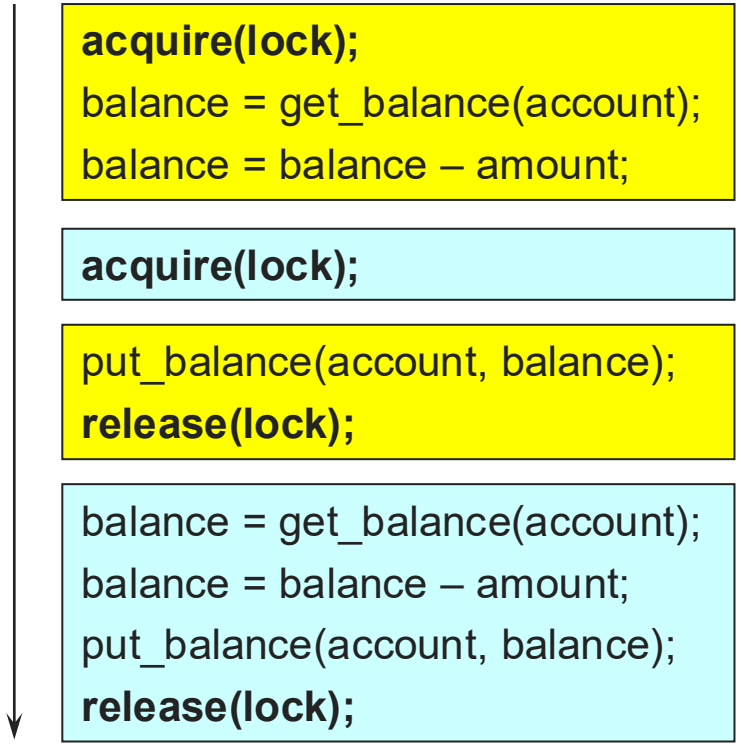
Locks

- A lock is an object in memory providing two operations
  - acquire(): before entering the critical section
  - release(): after leaving a critical section
- Threads pair calls to acquire() and release()
  - Between acquire()/release(), the thread holds the lock
  - acquire() does not return until any previous holder releases
  - What can happen if the calls are not paired?

Using Locks

```
withdraw (account, amount) {
    acquire(lock);
    balance = get_balance(account);
    balance = balance – amount;
    put_balance(account, balance);
    release(lock);
    return balance;
}
```

Critical Section



- Why is the “return” outside the critical section? Is this ok?
- What happens when a third thread calls acquire?

How do we implement a lock? First try

```
pthread_trylock(mutex) {
    if (mutex==0) {
     mutex= 1;
     return 1;
    } else return 0;
}
```

```
Thread 0, 1, …

…//time to access critical region
while(!pthread_trylock(mutex); // wait
<critical region>
pthread_unlock(mutex)
```

- Does this work? Assume reads/writes are **atomic**
- The lock itself is a critical region!
  - Chicken and egg
- Computer scientist struggled with how to create software locks

Second try

```
int turn = 1;
```

```
while (true) {
    while (turn != 1) ;
    critical section
    turn = 2;
    outside of critical section
}
```

```
while (true) {
    while (turn != 2) ;
    critical section
    turn = 1;
    outside of critical section
}
```

This is called alternation. It satisfies mutex:

- If blue is in the critical section, then turn == 1 and if yellow is in the critical section then turn == 2
- (turn == 1) ≡ (turn != 2)

Is there anything wrong with this solution?

Third try – two variables

```
bool flag[2] = {0, 0};
```

```
while (flag[1] != 0);
flag[0] = 1;
critical section
flag[0]=0;
outside of critical section
```

```
while (flag[0] != 0);
flag[1] = 1;
critical section
flag[1]=0;
outside of critical section
```

We added two variables to try to break the race for the same variable

Is there anything wrong with this solution?

Fourth try – set before you check

```
bool flag[2] = {0, 0};
```

```
flag[0] = 1;
while (flag[1] != 0);
critical section
flag[0]=0;
outside of critical section
```

```
flag[1] = 1;
while (flag[0] != 0);
critical section
flag[1]=0;
outside of critical section
```

Is there anything wrong with this solution?

Fifth try – double check and back off

```
bool flag[2] = {0, 0};
```

```
flag[0] = 1;
while (flag[1] != 0) {
        flag[0] = 0;
        wait a short time;
        flag[0] = 1;
}
critical section
flag[0]=0;
outside of critical section
```

```
flag[1] = 1;
while (flag[0] != 0) {
        flag[1] = 0;
        wait a short time;
        flag[1] = 1;
}
critical section
flag[1]=0;
outside of critical section
```

Six try – Dekker's Algorithm

```
bool flag[2] = {0, 0};
int turn = 1;
```

```
flag[0] = 1;
while (flag[1] != 0) {
        if (turn == 2) {
        flag[0] = 0;
        while (turn == 2);
        flag[0] = 1;
       } //if
} //while
critical section
flag[0]=0;
turn=2;
outside of critical section
```

```
flag[1] = 1;
while (flag[0] != 0) {
        if (turn == 1) {
        flag[1] = 0;
        while (turn == 1);
        flag[1] = 1;
       } //if
} //while
critical section
flag[1]=0;
turn=1;
outside of critical section
```

Peterson's Algorithm

```
int turn = 1;
bool try1 = false, try2 = false;
```

```
while (true) {
    try1 = true;
    turn = 2;
    while (try2 && turn != 1) ;
    critical section
    try1 = false;
    outside of critical section
}
```

```
while (true) {
    try2 = true;
    turn = 1;
    while (try1 && turn != 2) ;
    critical section
    try2 = false;
    outside of critical section
}
```

- This satisfies all the requirements
- Here's why...

Peterson's Algorithm: analysis

```
int turn = 1;
bool try1 = false, try2 = false;
```

```
while (true) {
   {¬ try1 ∧ (turn == 1 ∨ turn == 2) }
1  try1 = true;
   { try1 ∧ (turn == 1 ∨ turn == 2) }
2  turn = 2;
   { try1 ∧ (turn == 1 ∨ turn == 2) }
3  while (try2 && turn != 1) ;
   {  try1 ∧ (turn == 1 ∨ ¬ try2 ∨
        (try2 ∧ (yellow at 6 or at 7)) }
   critical section
4  try1 = false;
   {¬ try1 ∧ (turn == 1 ∨ turn == 2) }
   outside of critical section
}
```

```
while (true) {
   {¬ try2 ∧ (turn == 1 ∨ turn == 2) }
5  try2 = true;
   { try2 ∧ (turn == 1 ∨ turn == 2) }
6  turn = 1;
   { try2 ∧ (turn == 1 ∨ turn == 2) }
7  while (try1 && turn != 2) ;
   {  try2 ∧ (turn == 2 ∨ ¬ try1 ∨
        (try1 ∧ (blue at 2 or at 3)) }
   critical section
8  try2 = false;
   {¬ try2 ∧ (turn == 1 ∨ turn == 2) }
   outside of critical section
}
```

(blue at 4) ∧ try1 ∧ (turn == 1 ∨ ¬ try2 ∨ (try2 ∧ (yellow at 6 or at 7))
∧ (yellow at 8) ∧ try2 ∧ (turn == 2 ∨ ¬ try1 ∨ (try1 ∧ (blue at 2 or at 3))
... ⇒ (turn == 1 ∧ turn == 2)

Some observations

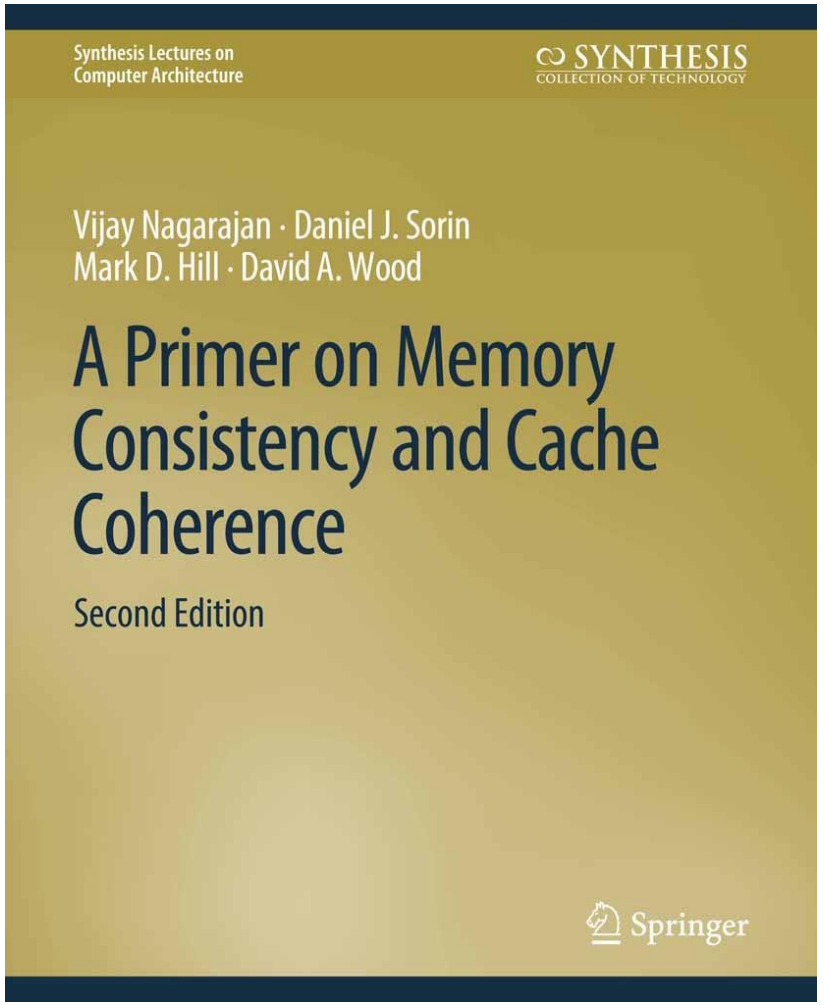
- This stuff (software locks) is hard
  - Hard to get right
  - Hard to prove right
- Even better, software locks don't really work
  - Compiler optimizations and hardware (out-of-order execution) can reorder memory accesses from different threads
    - Something called memory consistency model
    - Well beyond the scope of this class ☺
- So, we need to find a different way
  - Hardware help; more in a second

Hardware to the rescue

- Crux of the problem:
  - We get interrupted between checking the lock and setting it to 1
  - Software locks reordered by compiler/hardware
- Possible solutions?
  - Atomic instructions: create a new CPU instruction that checks and sets a variable atomically
    - » Cannot be interrupted! (toss the problem to hardware designers)
    - » How do we use them?
  - Disable interrupts altogether (no one else can interrupt us)

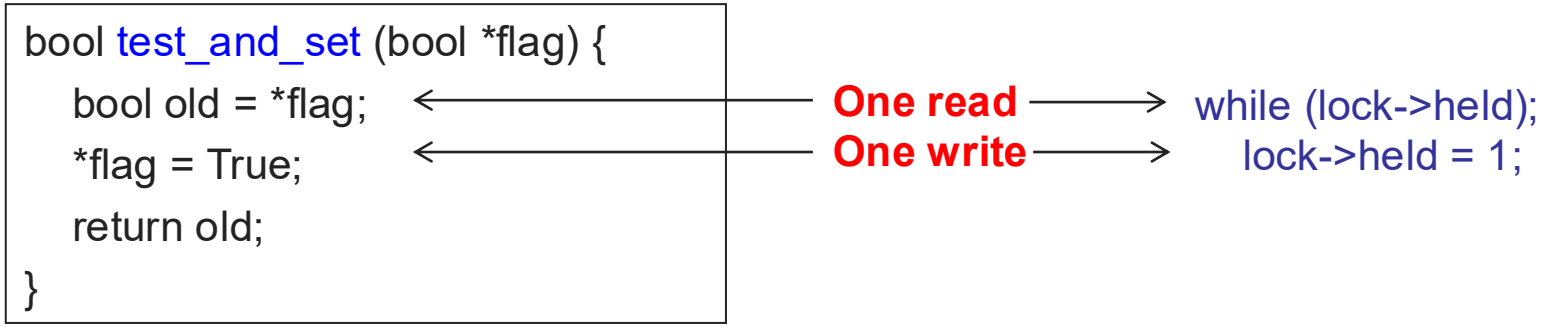
*How hardware solve the problem?

- Memory consistency
  - barriers
- Cache coherence
- CPU interconnects



Atomic Instruction: Test-and-Set

- The semantics of test-and-set are:
  1. Record the old value
  2. Set the value to indicate available
  3. Return the old value
- Hardware executes it atomically!



- When executing test-and-set on "flag"
  - What is value of flag afterwards if it was initially False? True?
  - What is the return result if flag was initially False? True?

Using Test-and-Set

- Here is our lock implementation with test-and-set:

```
struct lock {
    bool held = 0; // 0 === false
}
void acquire (lock) {
    while (test-and-set(&lock->held));
}
void release (lock) {
    lock->held = 0; // 0 === false
}
```

Does it satisfy critical section requirements?
(mutex, progress, bounded wait, performance?)

- When will the while return? What is the value of held?

Disabling Interrupts

- Another implementation of acquire/release is to disable interrupts:

```
struct lock {
}
void acquire (lock) {
    disable interrupts;
}
void release (lock) {
    enable interrupts;
}
```

- Note that there is no state associated with the lock
- Can two threads disable interrupts simultaneously?

On Disabling Interrupts

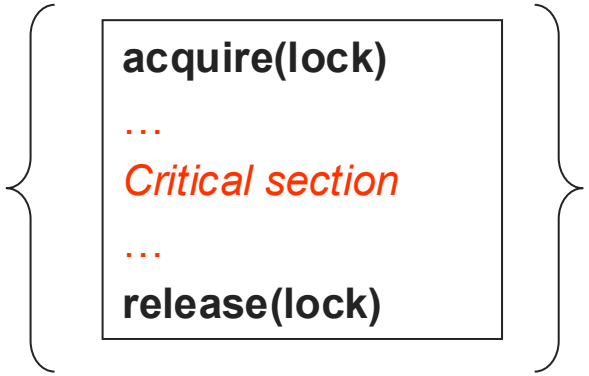
- Disabling interrupts blocks notification of external events that could trigger a context switch (e.g., timer)
- In a "real" system, this is only available to the kernel
  - Why?
- Disabling interrupts doesn't work on a multiprocessor system
  - Why? Back to atomic instructions
- Like spinlocks, only want to disable interrupts to implement higher-level synchronization primitives
  - Don't want interrupts disabled between acquire and release

Summarize Where We Are

- Goal: Use mutual exclusion to create critical sections of code that access shared resources without data races
- Method: Use locks (spinlocks or disable interrupts)
- Problem: Critical sections can be long

Spinlocks:

- Threads waiting to acquire lock spin in test-and-set loop
- May wastes CPU cycles
- Longer the CS, the longer the spin
- Greater the chance for lock holder to be interrupted
- Memory consistency model causes problems (out of scope of this class)



Disabling Interrupts:

- Should not disable interrupts for long periods of time
- Can miss or delay important events (e.g., timer, I/O)

Higher-Level Synchronization

- Spinlocks and disabling interrupts are useful for short and simple critical sections
  - Can be wasteful otherwise
  - These primitives are “primitive” – don’t do anything besides mutual exclusion
- Need higher-level synchronization primitives that:
  - Block waiters
  - Leave interrupts enabled within the critical section
- All synchronization requires atomicity
- So we’ll use our atomic locks as primitives to implement them

Implementing a Blocking Lock

- Can use a spinlock instead of disabling interrupts

```
struct lock {
    int held = 0;
    queue Q;
}
void acquire (lock) {
    spinlock->acquire();
    if (lock->held) {
        put current thread on lock Q;
        block current thread;
    }
    lock->held = 1;
    spinlock->release();
}
```

```
void release (lock) {
    spinlock->acquire();
    if (Q)
        remove and unblock a waiting thread;
    else
        lock->held = 0;
    spinlock->release();
}
```

```
acquire(lock)
…
Critical section
…
release(lock)
```

Spinning

Running or Blocked

Spinning

Mechanisms For Building Critical Sections

- Locks
  - Primitive, minimal semantics, used to build others
- Architecture help
  - Atomic test-and-set
- Semaphores
  - Basic, easy to get the hang of, but hard to program with
- Monitors
  - High-level, requires language support, operations implicit

Semaphores

- Semaphores are an abstract data type that provide mutual exclusion to critical sections
  - Block waiters, interrupts enabled within critical section
  - Described by Dijkstra in THE system in 1968
- Semaphores are **integers** that support two operations:
  - wait(semaphore): decrement, block until semaphore is open
    - » Also P(), after the Dutch word for test, or down()
  - signal(semaphore): increment, allow another thread to enter
    - » Also V() after the Dutch word for increment, or up()
  - That's it! No other operations – not even just reading its value

Blocking in Semaphores

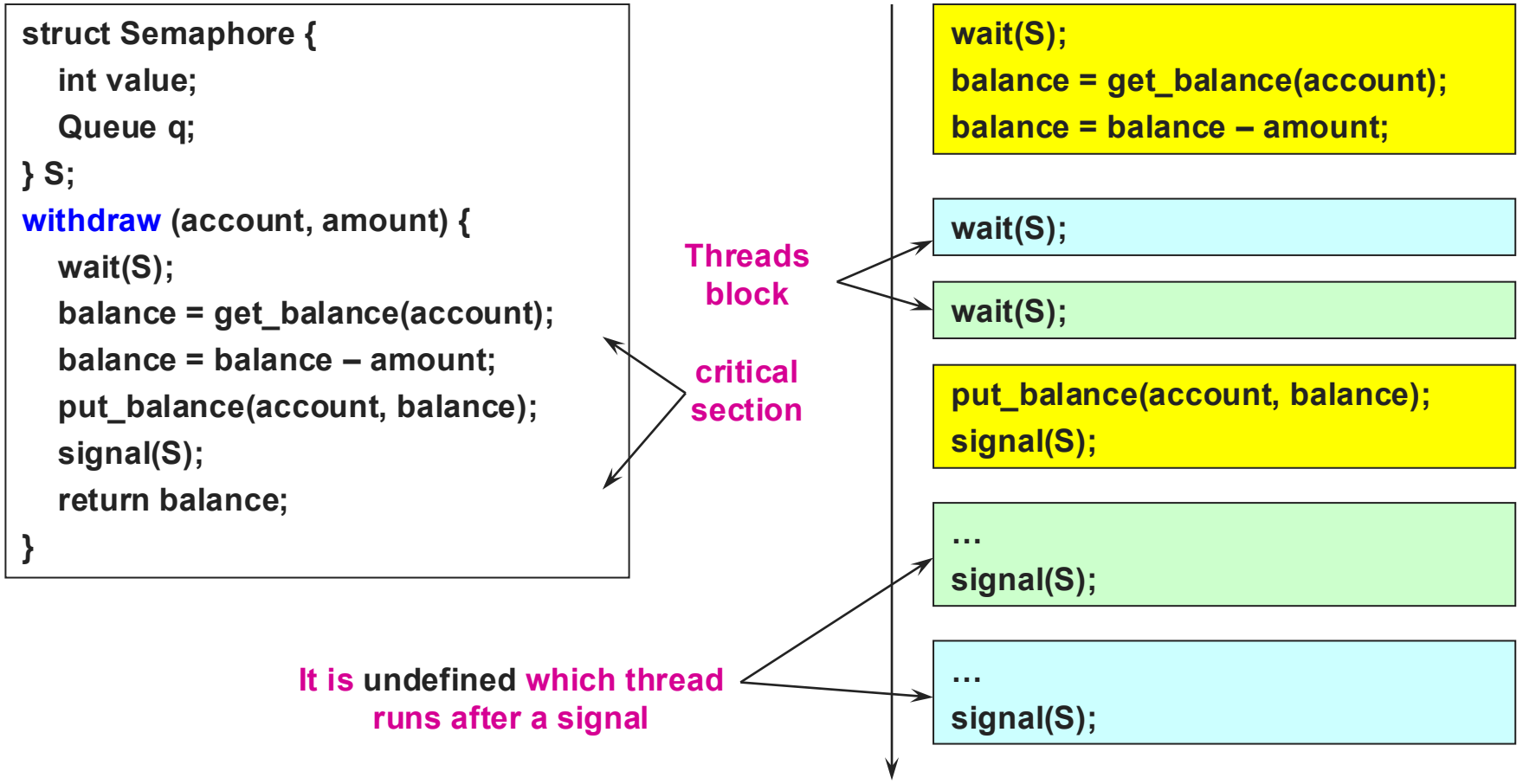
- Associated with each semaphore is a queue of waiting threads/processes
- When wait() is called by a thread:
  - If semaphore is open (> 0), and thread continues
  - If semaphore is closed (<= 0), thread blocks on queue
- Then signal() opens the semaphore:
  - If semaphore is still closed after increment (<= 0), a thread is waiting on the queue, the thread is unblocked
  - If semaphore becomes open (> 0), no threads are waiting on the queue, the signal is remembered for the next thread

Semaphore Types

- Semaphores come in two types
- Mutex semaphore (or binary semaphore)
  - Represents single access to a resource
  - Guarantees mutual exclusion to a critical section
- Counting semaphore (or general semaphore)
  - Multiple threads pass the semaphore determined by count
    - » mutex has count = 1, counting has count = N
  - Represents a resource with many units available
  - or a resource allowing some unsynchronized concurrent access (e.g., reading)

Using Semaphores

- Use is similar to our locks, but semantics are different



Blocking in Semaphores

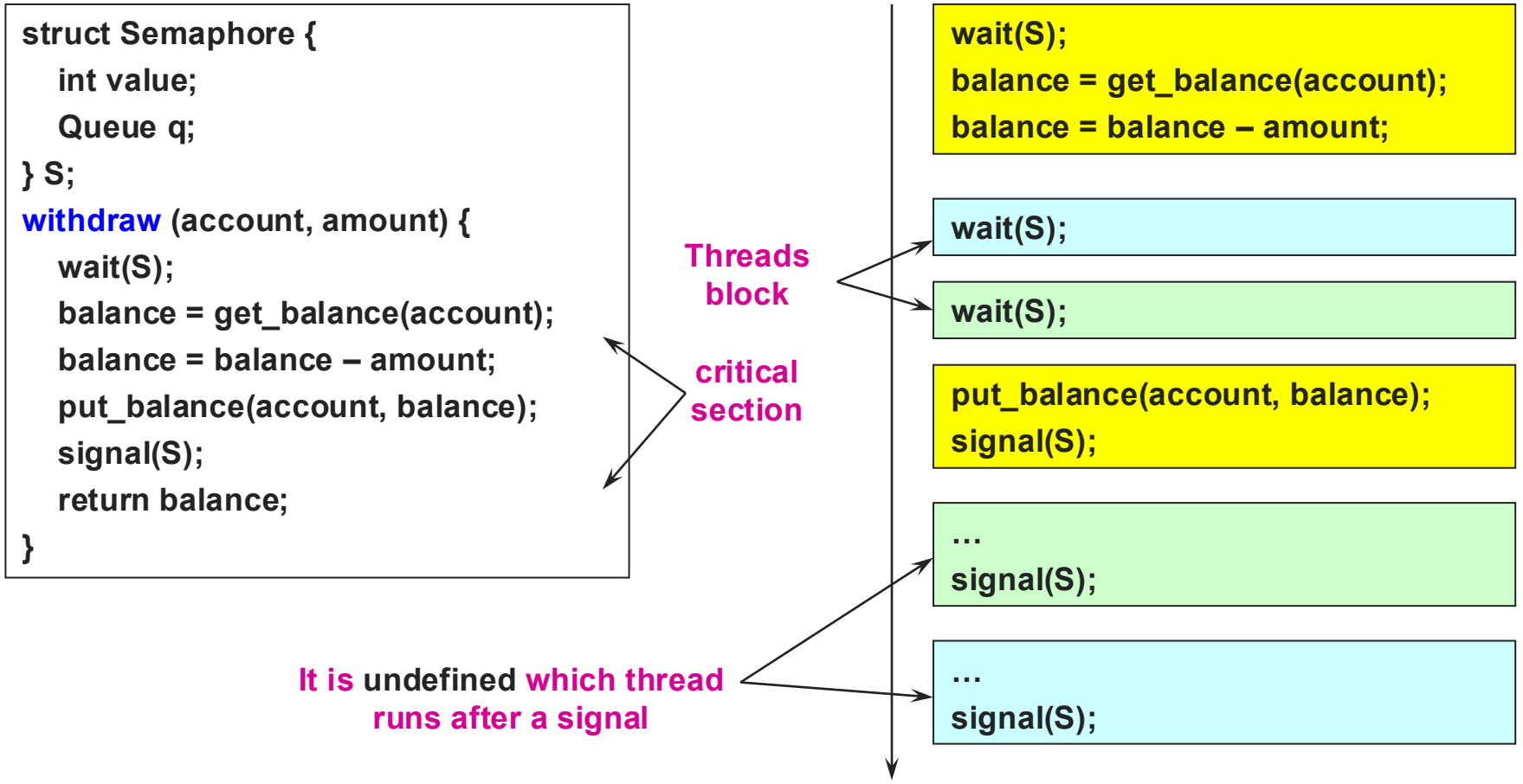
- Associated with each semaphore is a queue of waiting threads/processes
- When wait() is called by a thread:
  - If semaphore is open (> 0), and thread continues
  - If semaphore is closed (== 0), thread blocks on queue
- Then signal() opens the semaphore:
  - If semaphore is closed, a thread is waiting on the queue, the thread is unblocked
  - If no threads are waiting on the queue, the signal is remembered for the next thread

Semaphores

- Semaphores are an abstract data type that provide mutual exclusion to critical sections
  - Block waiters, interrupts enabled within critical section
  - Described by Dijkstra in THE system in 1968
- Semaphores are **integers** that support two operations:
  - wait(semaphore): decrement, block until semaphore is open
    - » Also P(), after the Dutch word for test, or down()
  - signal(semaphore): increment, allow another thread to enter
    - » Also V() after the Dutch word for increment, or up()
  - That's it! No other operations – not even just reading its value

Using Semaphores

- Use is similar to our locks, but semantics are different



Beyond Mutual Exclusion

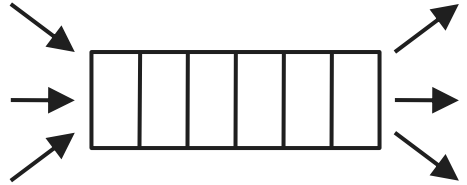
- We've looked at a simple example for using synchronization
  - Mutual exclusion while accessing a bank account
- We're going to use semaphores to look at more interesting examples
  - Counting critical region
  - Ordering threads
  - Readers/Writers
  - Producer consumer with bounded buffers
  - More general examples

Rendezvous

- A group of us go to a restaurant; we wait until the last person arrives before we go in.

```
int group_count = N; //initialized to the number of people in our party
Semaphore mutex(1); //protect the group_count
Semaphore barrier(0); //initialized to 0
arrive() {
  mutex.wait(); // prevent data race on updating and checking the counter
  group_count --;
  if (group_count == 0) {
    for (int i=0; i < N; i++)
      barrier.signal(); // wake up everyone waiting
  }
  mutex.signal();
  barrier.wait(); // wait until being wake up
  //party time!
}
```

Bounded Buffer



- Problem: Set of buffers shared by producer and consumer threads
  - Producer inserts jobs into the buffer set
  - Consumer removes jobs from the buffer set
- Producer and consumer execute at different rates
  - No serialization of one behind the other
  - Tasks are independent (easier to think about)
  - The buffer set allows each to run without explicit handoff
- Data structure should not be corrupted
  - Due to race conditions
  - Or producer writing when full
  - Or consumer deleting when empty

Bounded Buffer (2)

```
Semaphore mutex = 1;   // mutual exclusion to shared set of buffers
Semaphore empty = N;   // count of empty buffers (all empty to start)
Semaphore full = 0;    // count of full buffers (none full to start)
```

```
producer {
  while (1) {
    Produce new resource;
    wait(empty); // wait for empty buffer
    wait(mutex); // lock buffer list
    Add resource to an empty buffer;
    signal(mutex); // unlock buffer list
    signal(full);      // note a full buffer
  }
}
```

```
consumer {
  while (1) {
    wait(full);        // wait for a full buffer
    wait(mutex);    // lock buffer list
    Remove resource from a full buffer;
    signal(mutex); // unlock buffer list
    signal(empty); // note an empty buffer
    Consume resource;
  }
}
```

Bounded Buffer (3)

- Why need the mutex at all?

- The pattern of signal/wait on full/empty is a common construct often called an interlock

- Producer-Consumer and Bounded Buffer are classic examples of synchronization problems
  - We will see and practice others

Granularity of Locks

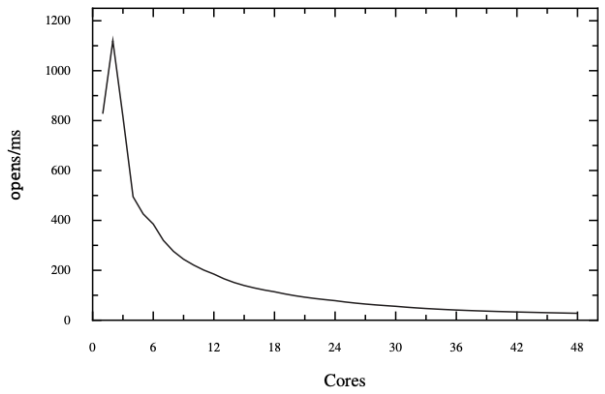
- Voting Machine:
  - You are writing code for the voting machines for an upcoming election. A central display shows the counter of each candidate as results come from different voting machines. You can consider each machine as a thread and counters are shared between different threads.
- Where to put the lock?
  - Option1: a single global lock, thread always update global counters
  - Option2: per-counter lock, thread always update global counters
  - Option3: a single global lock, thread updates local counters first, periodically update global counters
  - Option4: per-counter lock, thread updates local counters first, periodically update global counters
- Does it matter? **YES**

Lock Contention

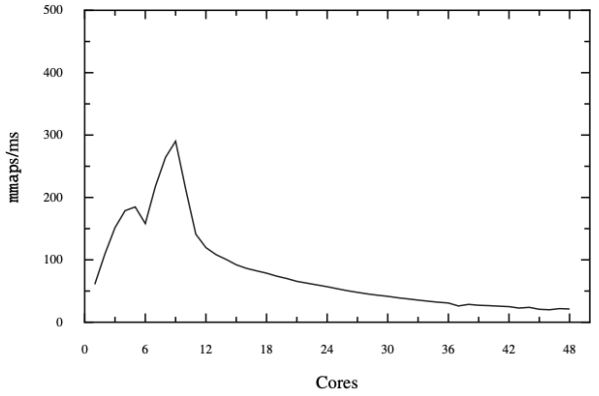
- Lock contention occurs when multiple threads or processes simultaneously attempt to acquire the same lock, causing some to wait or spin idly until the lock is released, which reduces parallelism and degrades performance.

```
struct spinlock_t {
  int current_ticket;
  int next_ticket;
}

void spin_lock(spinlock_t *lock)
{
  int t =
    atomic_fetch_and_inc(&lock->next_ticket);
  while (t != lock->current_ticket)
    ;   /* spin */
}

void spin_unlock(spinlock_t *lock)
{
  lock->current_ticket++;
}
```

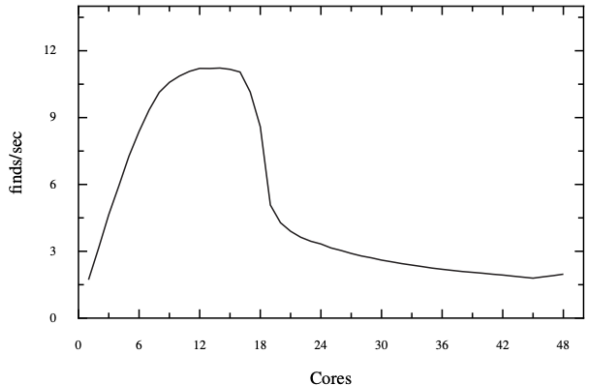
Figure 1: Pseudocode for ticket locks in Linux.



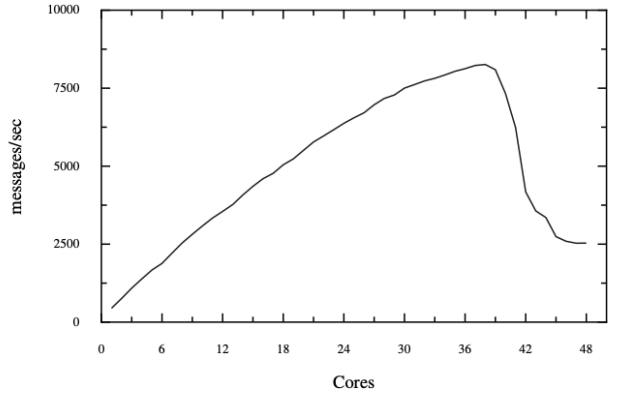
(a) Collapse for FOPS.



(b) Collapse for MEMPOP.



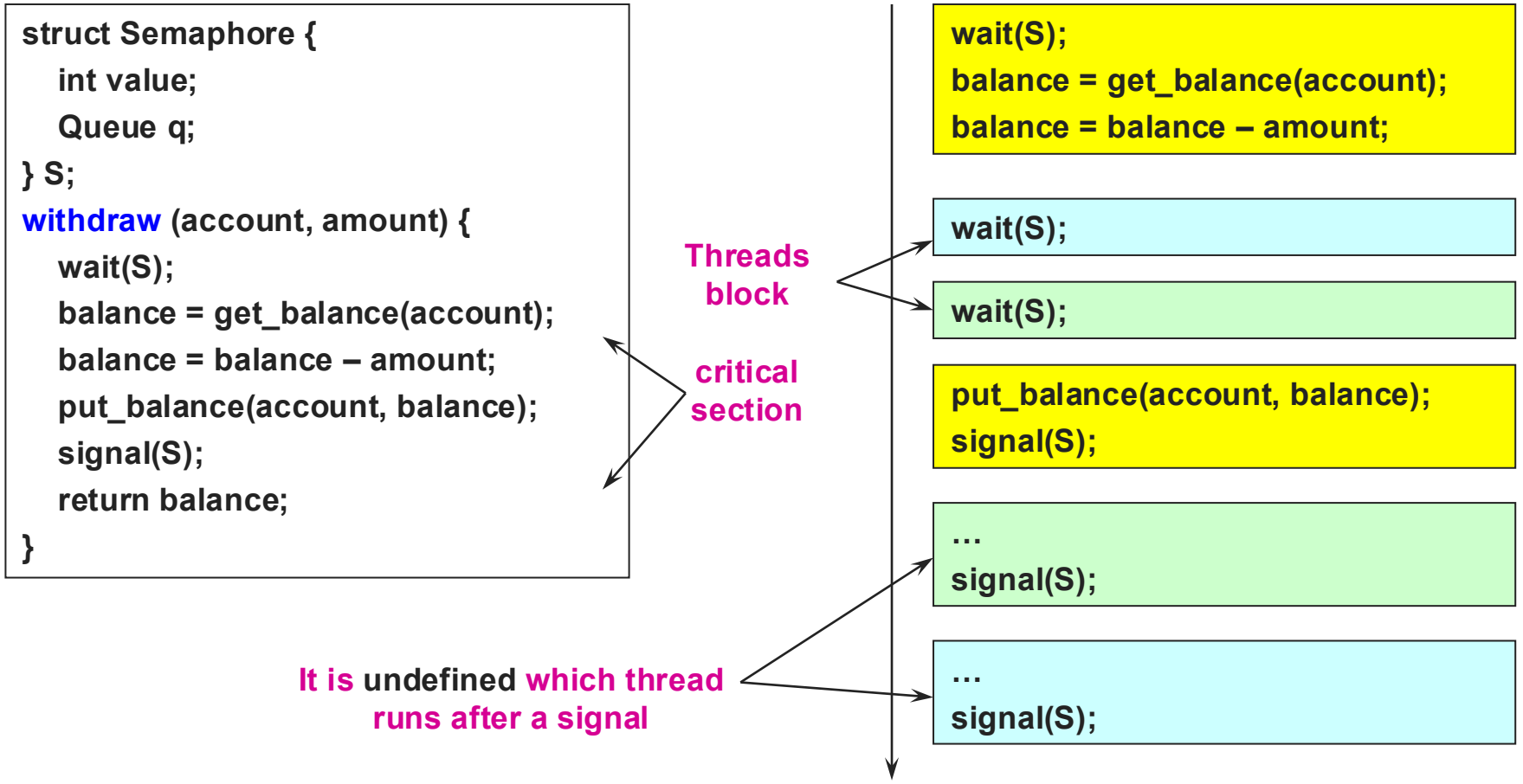
(c) Collapse for PFIND.



(d) Collapse for EXIM.

Revisit our withdraw method

- What's wrong with this implementation?



Readers/Writers Problem

- Readers/Writers Problem:
  - An object is shared among several threads
  - Some threads only read the object, others only write it
  - We can allow multiple readers but only one writer
    - Let #r be the number of readers, #w be the number of writers
    - Safety: $(\#r \geq 0) \wedge (0 \leq \#w \leq 1) \wedge ((\#r > 0) \Rightarrow (\#w = 0))$
- Use three variables
  - int readcount – number of threads reading object
  - Semaphore mutex – control access to readcount
  - Semaphore w_or_r – exclusive writing or reading

Readers/Writers

```
1: // number of readers
2: int readcount = 0;
3: // mutual exclusion to readcount
4: Semaphore mutex = 1;
5: // exclusive writer or reader
6: Semaphore w_or_r = 1;
7:
8: writer {
9:    wait(w_or_r); // lock out readers
10:  Write;
11:  signal(w_or_r); // up for grabs
12:}
```

```
1: reader {
2:    wait(mutex);       // lock readcount
3:    readcount += 1; // one more reader
4:    if (readcount == 1)
5:       wait(w_or_r); // synch w/ writers
6:    signal(mutex);    // unlock readcount
7:    Read;
8:    wait(mutex);       // lock readcount
9:    readcount -= 1; // one less reader
10:  if (readcount == 0)
11:    signal(w_or_r); // up for grabs
12:  signal(mutex);    // unlock readcount
13: }
```

Readers/Writers

```
1: // number of readers
2: int readcount = 0;
3: // mutual exclusion to readcount
4: Semaphore mutex = 1;
5: // exclusive writer or reader
6: Semaphore w_or_r = 1;
7:
8: writer {
9:    wait(w_or_r); // lock out readers
10:  Write;
11:  signal(w_or_r); // up for grabs
12:}
```

```
1: reader {
2:    wait(mutex);       // lock readcount
3:    readcount += 1; // one more reader
4:    if (readcount == 1)
5:       wait(w_or_r); // synch w/ writers
6:    signal(mutex);    // unlock readcount
7:    Read;
8:    wait(mutex);       // lock readcount
9:    readcount -= 1; // one less reader
10:  if (readcount == 0)
11:    signal(w_or_r); // up for grabs
12:  signal(mutex);    // unlock readcount
13: }
```

Avoid Starvation

```
// number of readers
int readcount = 0;
// mutual exclusion to readcount
Semaphore mutex = 1;
// exclusive writer or reader
Semaphore w_or_r = 1;
// turnstile for everyone
Semaphore turnstile = 1;

writer {
    wait(turnstile); // get in the queue
    wait(w_or_r); // lock out readers
    Write;
    signal(w_or_r); // up for grabs
    signal(turnstile); // next
}
```

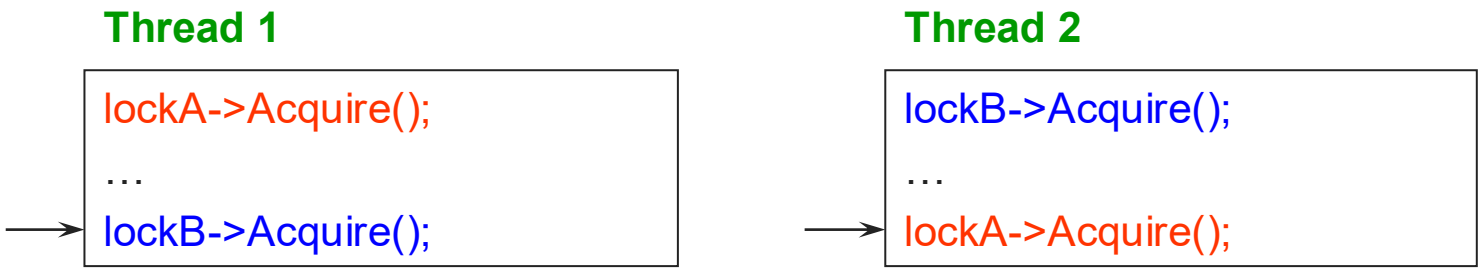
```
reader {
    wait(turnstile);  // get in the queue
    signal(turnstile); // next
    wait(mutex);       // lock readcount
    readcount += 1; // one more reader
    if (readcount == 1)
        wait(w_or_r); // synch w/ writers
    signal(mutex);    // unlock readcount
    Read;
    wait(mutex);       // lock readcount
    readcount -= 1; // one less reader
    if (readcount == 0)
        signal(w_or_r); // up for grabs
    signal(mutex);    // unlock readcount
}
```

Deadlock — the deadly embrace!

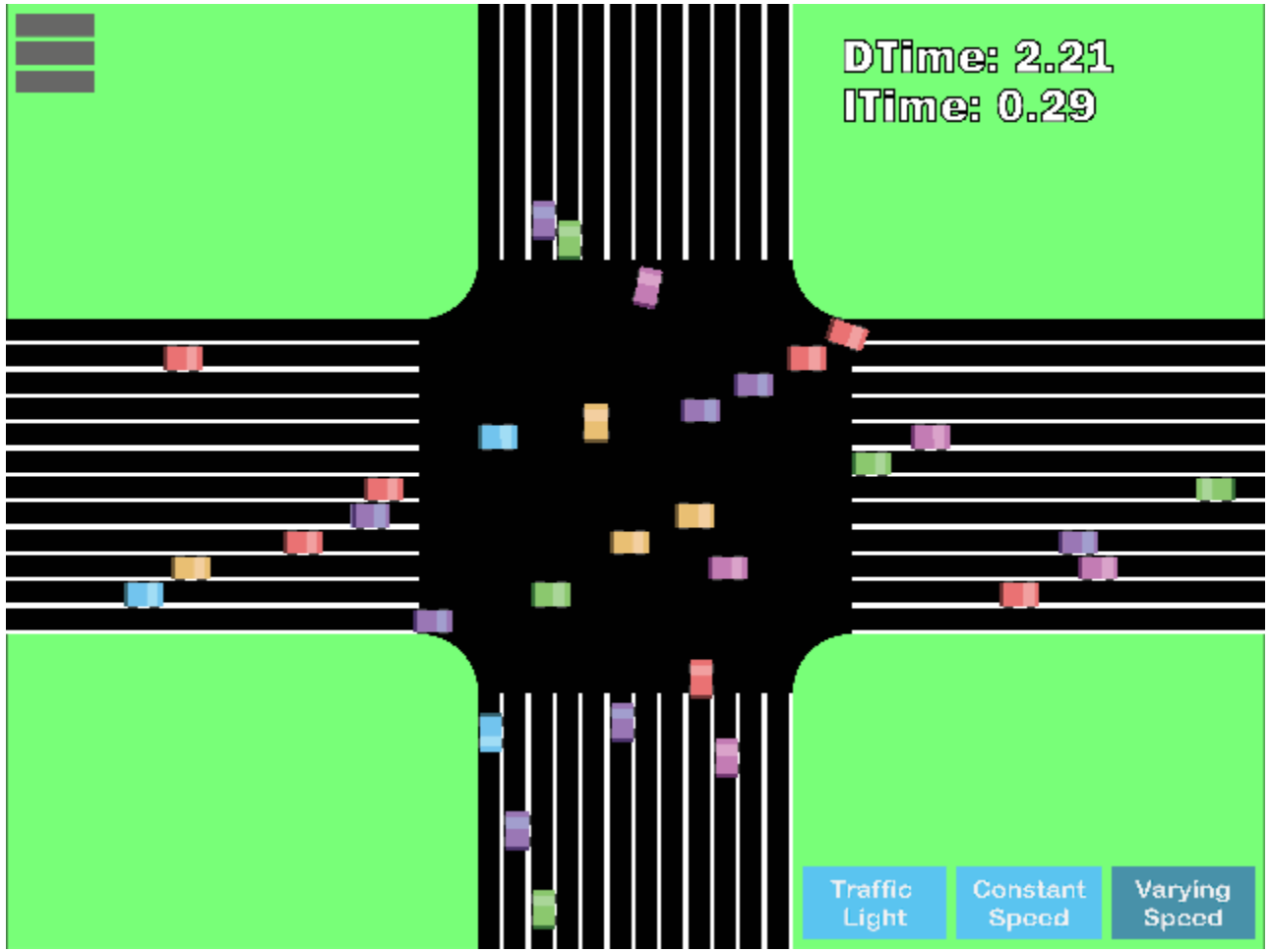
- Synchronization – we can easily shoot ourselves in the foot
  - Incorrect use of synchronization can block all processes
  - You have likely been intuitively avoiding this situation already
- Consider: threads that use multiple critical sections/need different resources
  - If one thread tries to access a resource that a second thread holds, and vice-versa, they can never make progress
- We call this situation deadlock, and we'll look at:
  - Definition and conditions necessary for deadlock
  - Representation of deadlock conditions
  - Approaches to dealing with deadlock

Deadlock Definition

- Deadlock is a problem that can arise:
  - When threads/processes compete for access to limited resources
  - When threads/processes are incorrectly synchronized
- Definition:
  - Deadlock exists among a set of threads if every thread is waiting for an event that can be caused only by another thread in the set



Real example!



Real example!

Conditions for Deadlock

- Deadlock can exist if and only if the following four conditions hold **simultaneously**:
  1. Mutual exclusion – At least one resource must be held in a non-sharable mode
  2. Hold and wait – There must be one process holding one resource and waiting for another resource
  3. No preemption – Resources cannot be preempted (critical sections cannot be aborted externally)
  4. Circular wait – There must exist a set of threads $[T_1, T_2, T_3, \ldots, T_n]$ such that T_1 is waiting for T_2, T_2 for T_3, etc.

Deadlock Prevention

- Prevention – Ensure that at least one of the necessary conditions cannot happen
  - Mutual exclusion
    - » Make resources sharable (not generally practical)
  - Hold and wait
    - » Process/thread cannot hold one resource when requesting another
  - Preemption
    - » OS can preempt resource (costly)
  - Circular wait
    - » Impose an ordering (numbering) on the resources and request them in order (popular implementation technique)

Deadlock Prevention

- One shot allocation: ask for all your resources in one shot; no more resources can be requested
  - What ingredient does this prevent?
  - Comments?
- Preemption
  - Nice: Give up a resource if what you want is not available
  - Aggressive: steal a resource if what you want is not available
- Hierarchical allocation:
  - Assign resources to classes
  - Can only ask for resources from a higher number class than what you hold now

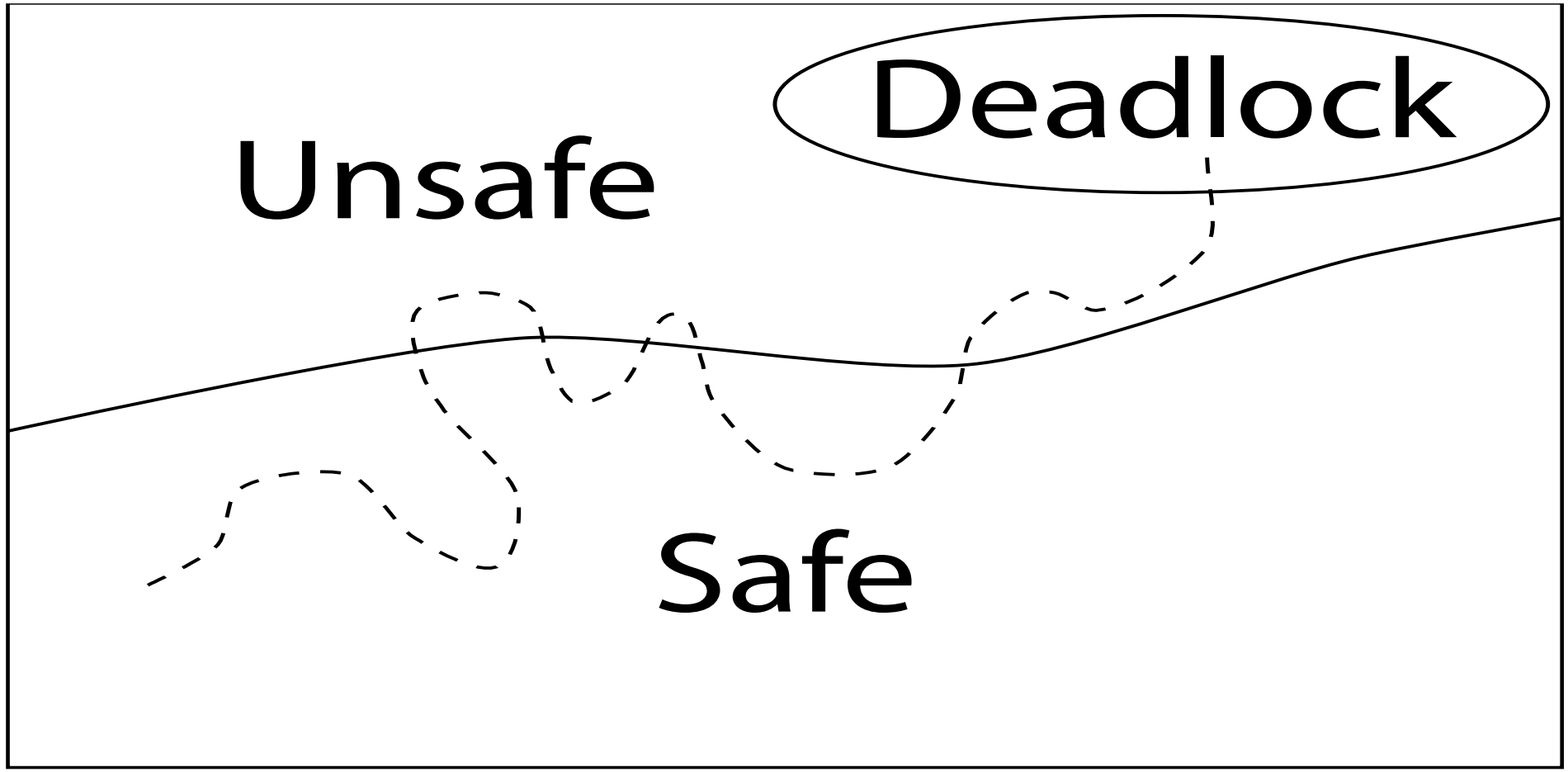
Deadlock Avoidance

- Prevention can be too conservative – can we do better?
- Avoidance
  - Provide information in advance about what resources will be needed by processes
  - System only grants resource requests if it knows that deadlock cannot happen
  - Avoids circular dependencies
- Tough
  - Hard to determine all resources needed in advance
  - Good theoretical problem, not as practical to use

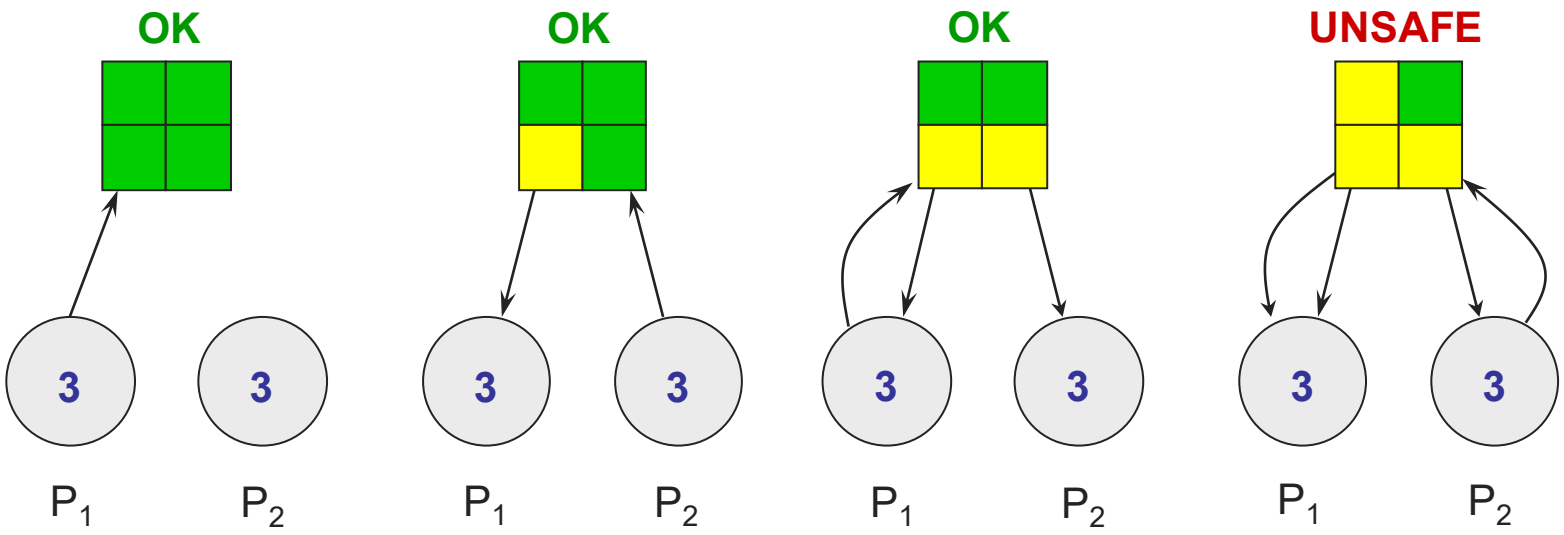
Banker's Algorithm

- The Banker's Algorithm is the classic approach to deadlock avoidance for resources with multiple units
  1. Assign a credit limit to each customer (process)
     - » Maximum credit claim must be stated in advance
  2. Reject any request that leads to a dangerous state
     - » A dangerous state is one where a sudden request by any customer for the full credit limit could lead to deadlock
     - » A recursive reduction procedure recognizes dangerous states
  3. In practice, the system must keep resource usage well below capacity to maintain a resource surplus
     - » Rarely used in practice due to low resource utilization

Possible System States



Banker's Algorithm Simplified



Detection and Recovery

- Detection and recovery
  - If we don't have deadlock prevention or avoidance, then deadlock may occur
  - In this case, we need to detect deadlock and recover from it
- To do this, we need two algorithms
  - One to determine whether a deadlock has occurred
  - Another to recover from the deadlock
- Possible, but expensive (time consuming)
  - Implemented in VMS
  - Run detection algorithm when resource request times out

Deadlock Detection

- Detection
  - Traverse the resource graph looking for cycles
  - If a cycle is found, preempt resource (force a process to release)
- Expensive
  - Many processes and resources to traverse
- Only invoke detection algorithm depending on
  - How often or likely deadlock is
  - How many processes are likely to be affected when it occurs

Deadlock Recovery

Once a deadlock is detected, we have two options…

1. Abort processes
   - Abort all deadlocked processes
     - » Processes need to start over again
   - Abort one process at a time until cycle is eliminated
     - » System needs to rerun detection after each abort
2. Preempt resources (force their release)
   - Need to select process and resource to preempt
   - Need to rollback process to previous state
   - Need to prevent starvation

Optimistic Concurrent Control (OCC)

- Goals: no data races, serializable execution of concurrent transactions/operations without conflicts corrupting data
- Approaches:
  - "Prevention" (pessimistic): assume conflicts are **likely** → acquire locks early to prevent issues (e.g., Two-Phase Reader-Writer Locking)
    - » Need to avoid deadlock
    - » Bad performance
  - "Hope" (optimistic): assume conflicts are **rare** → proceed without locking, detect conflicts late, and abort/restart if needed
    - » Cannot have deadlock
    - » Reduces blocking overhead in low-contention workloads

OCC: in the context of DBMS

- Even read only queries are slowed down as they must acquire read locks. Locking is pessimistic.
- Locking needs deadlock detection/prevention methods.
- If a database has a lock on an object that is on disk, performance suffers. Especially, if that object is “hot.”
- Need to hold locks till the end-of-transaction (assuming serializable mode).
- Locking is needed only in the “worst” case even for writers. But we always pay this locking overhead.
  - Not just lock contention, but also cache contention

```
reader {
    wait(turnstile);  // get in the queue
    signal(turnstile); // next
    wait(mutex);       // lock readcount
    readcount += 1; // one more reader
    if (readcount == 1)
        wait(w_or_r); // synch w/ writers
    signal(mutex);    // unlock readcount
    Read;
    wait(mutex);       // lock readcount
    readcount -= 1; // one less reader
    if (readcount == 0)
        signal(w_or_r); // up for grabs
    signal(mutex);    // unlock readcount
}
```

OCC: Preliminaries

Object can be anything: table, page, record, …

Object Interface:

Transactions use:

tcreate	• create(): Create a new object (returned by this call).
tdelete	• delete(n): delete object n.
tread	• read(n, i): Read item *i* of object *n* (return the value).
twrite	• write(n, i, v): Write object *v* as item *i* object n.

Read and Write phase functions:

- copy(n): Copy of object n, and return it.
- exchange(n1, n2): Exchange the names of two objects.

OCC: Preliminaries

"Private" new object.

```
tcreate = (
    n := create;
    create set := create set ∪ {n};
    return n)
```

Write directly to new objects that I have created.

Write to my private/local copy.

Copy-on-write.

```
twrite(n, i, v) = (
    if n ∈ create set
        then write(n, i, v)
    else if n ∈ write set
        then write(copies[n], i, v)
    else (
        m := copy(n);
        copies[n] := m;
        write set := write set ∪ {n};
        write(copies[n], i, v)))
```

Remember what we read.

Read from my private write set.

Actually, read the object.

```
tread(n, i) = (
    read set := read set ∪ {n};
    if n ∈ write set
        then return read(copies[n], i)
    else
        return read(n, i))
```

Remember what we delete.

```
tdelete(n) = (
    delete set := delete set ∪ {n}).
```

OCC: Preliminaries

tend(): **for** $n \in$ *write set* **do** *exchange*$(n, copies[n])$.

Make changes permanent in the "global" database.

(**for** $n \in$ *delete set* **do** *delete*(n);
for $n \in$ *write set* **do** *delete*$(copies[n])$).

Cleanup local state. Does not impact other transactions (txns) as this state was private to this txn.

OCC: Transaction model

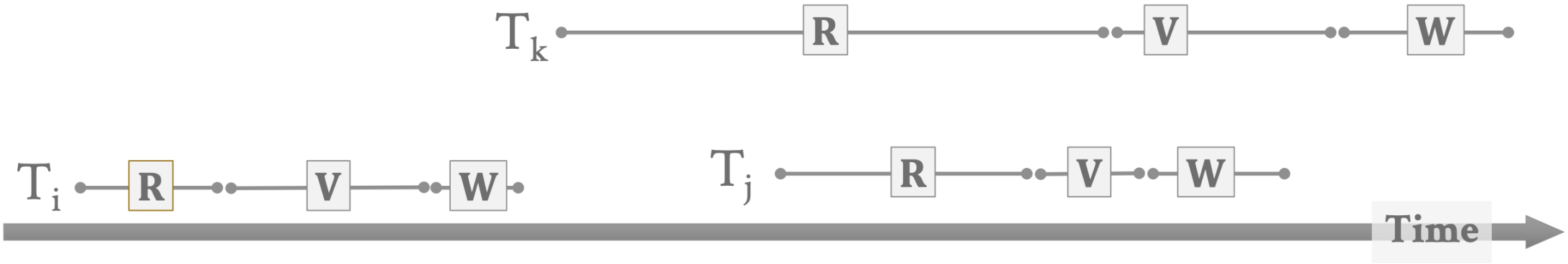
$$d_{final} = T_{\pi(n)} \circ T_{\pi(n-1)} \circ \cdots \circ T_{\pi(2)} \circ T_{\pi(1)}(d_{initial})$$

Goal: The permutation π is the sequence {1, 2, … n}, which is the serial order of the transactions. Note that we can number the transactions in any order; e.g., T1 does not need to be the first transaction that entered the system. Determining this permutation to maximize the concurrency is the problem we are solving.

OCC: Core idea

- Three Phases per Transaction
  1. Read Phase: Transaction reads data into a private workspace (local copies); tracks read/write sets. No locks or global visibility.
  2. Validation Phase: Check for conflicts with other committed transactions since the start.
  3. Write Phase: If validation succeeds, apply local writes to the global database; otherwise, abort and restart.
- **Core Assumption**: Most transactions won't conflict → waste is minimal on aborts.
- **No Blocking During Execution**: Readers/writers proceed freely until commit time.
- **Deadlock-Free**: No locks → no cycles

OCC: Examples



1. **READ** Phase: Read and write objects, making local copies.
2. **VALIDATION** Phase: Check for serializable schedule-related anomalies.
3. **WRITE** Phase: It is safe. Write the local objects, making them permanent.

OCC: Validations

- The critical step to make OCC work.
  - **Transaction Numbers**: Each transaction is assigned a unique integer transaction number (*t*). These are assigned at the end of the read phase to allow transactions to be validated immediately and improve response times.
  - **Three conditions**: For a transaction T_j to succeed, the system must guarantee that for all earlier transactions T_i (where $t(i)<t(j)$), one of the following three conditions is met.
- Two flavors: Serial and Parallel.
- If validation failed: abort and restart

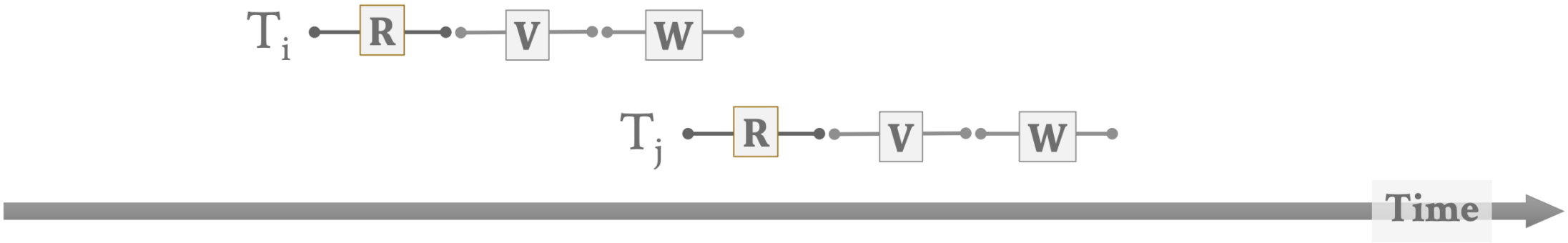
OCC: Validation condition 1

T_i completes its write phase before T_j starts its read phase.

No conflict as all of T_i's actions happen before T_j's.

OCC: Validation condition 2

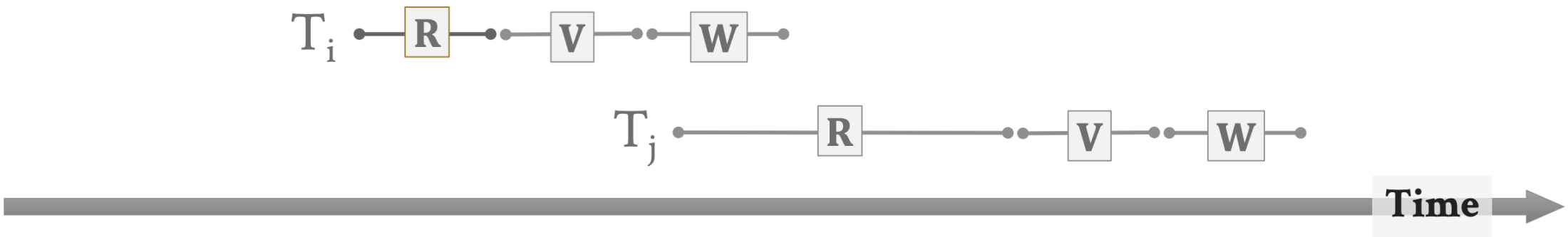
T_i completes its write phase before T_j starts its write phase.



Check that the write set of T_i does not intersect the read set of T_j, namely: $\texttt{WriteSet}(T_i) \cap \texttt{ReadSet}(T_j) = \emptyset$

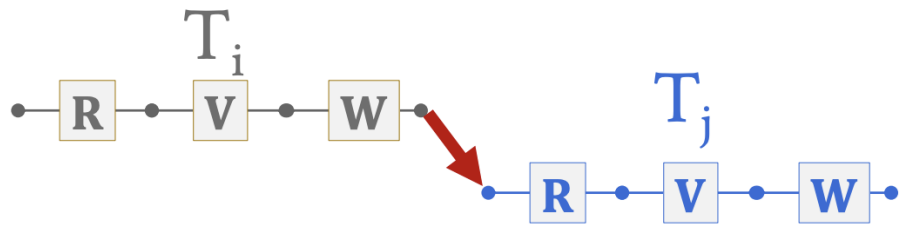
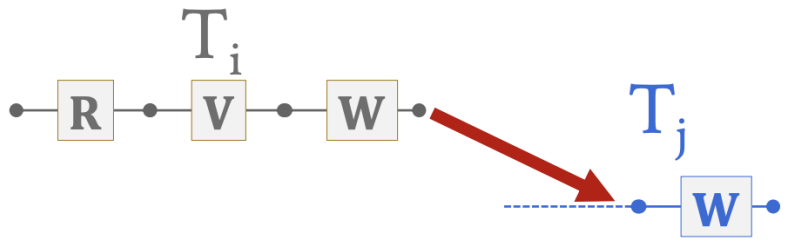
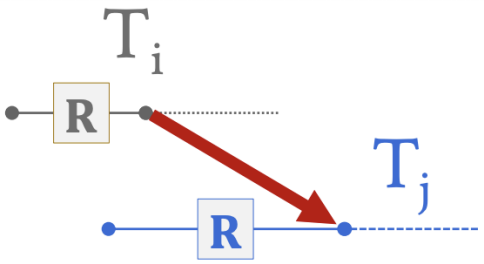
OCC: Validation condition 3

T_i completes its **read** phase before T_j completes its **read** phase.

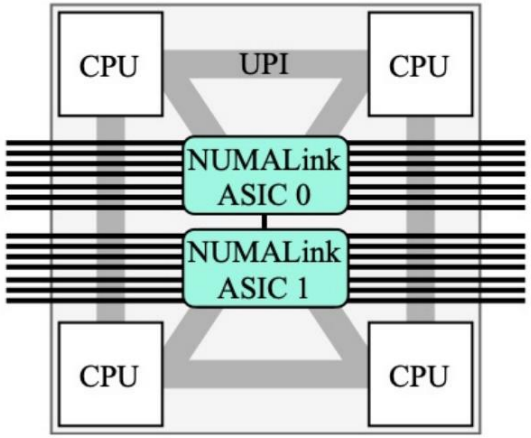


Check that the write set of T_i does not intersect the read or write sets of T_j, namely: $\texttt{WriteSet}(T_i) \cap \texttt{ReadSet}(T_j) = \emptyset$ AND $\texttt{WriteSet}(T_i) \cap \texttt{WriteSet}(T_j) = \emptyset$

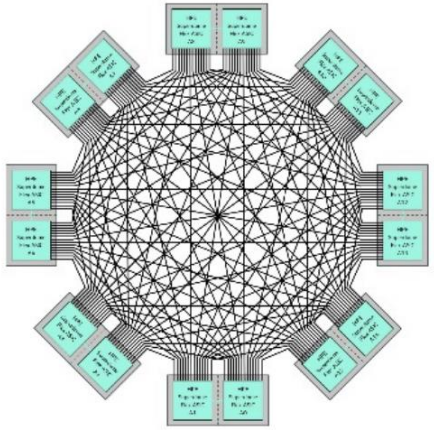
OCC: All three check

	R → W	W → R	W → W
Case 1 	✓	✓	✓
Case 2 	✓	$WriteSet(T_i) \cap ReadSet(T_j) = \emptyset$	✓
Case 3 	✓	$WriteSet(T_i) \cap ReadSet(T_j) = \emptyset$	$WriteSet(T_i) \cap WriteSet(T_j) = \emptyset$

OCC: Performance impacts



(a) Chassis topology



(b) NUMALink topology

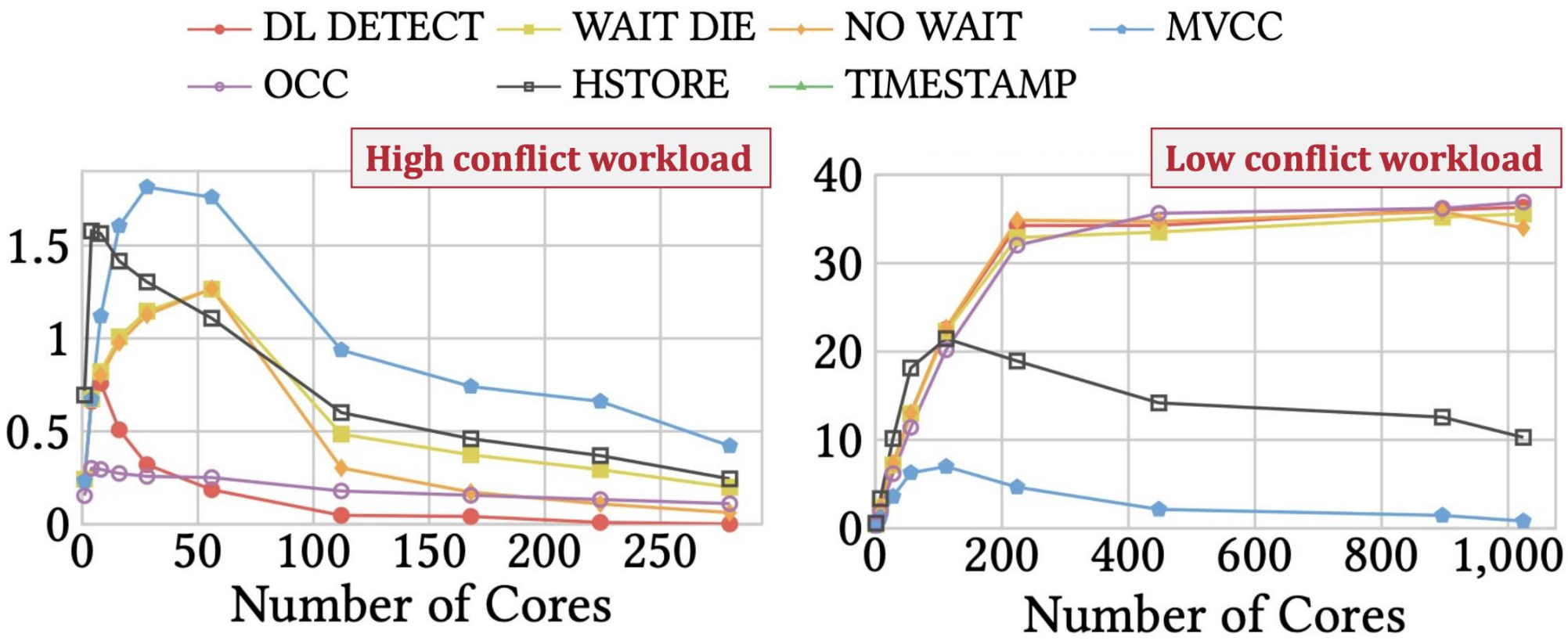
28 CPUs; 1568 logical cores; 20 TB of DRAM

NUMA level	Latency	Bandwidth
local	102 ns	95.1 GB/s
1 hop UPI	150 ns	17.2 GB/s
2 hop UPI	208 ns	16.6 GB/s
NUMALink	380 ns	11.2 GB/s

DL_DETECT	2PL with deadlock detection [2]
NO_WAIT	2PL with non-waiting deadlock prevention [2]
WAIT_DIE	2PL with wait-and-die deadlock prevention [2]
MVCC	Multi-version T/O [3]
OCC	Optimistic concurrency control [19]
H-STORE	T/O with partition-level locking [16]
TIMESTAMP	Basic T/O algorithm [2]
SILO	Epoch-based T/O [25]
TICTOC	Data-driven T/O [28]

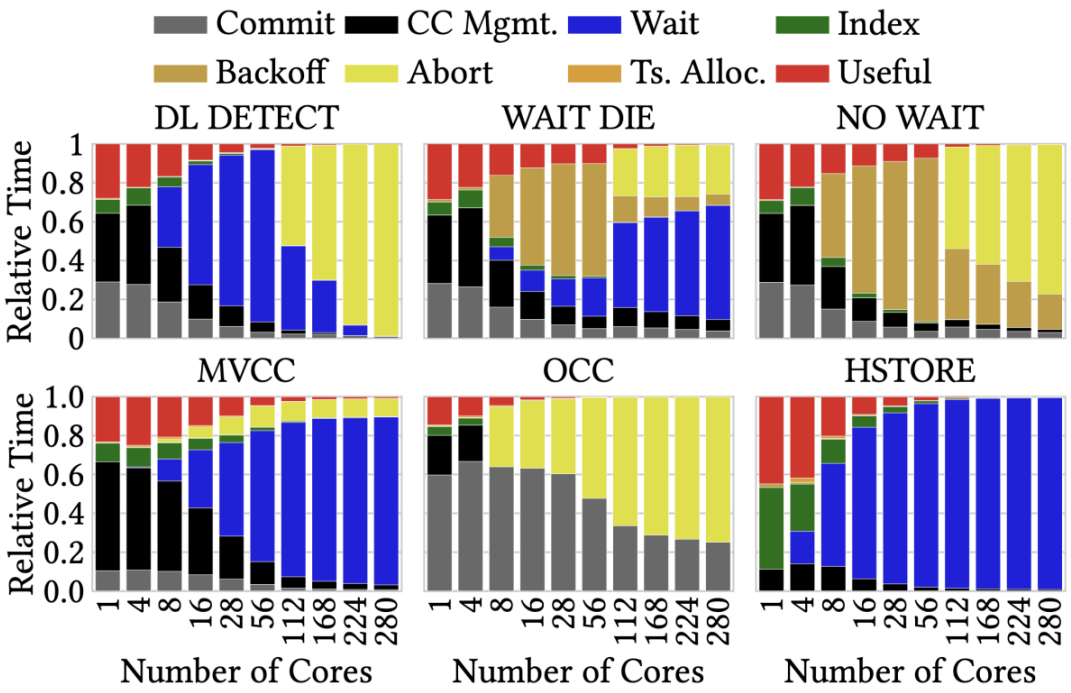
Tiemo Bang, Norman May, Ilia Petrov, Carsten Binnig: The tale of 1000 Cores: an evaluation of concurrency control on real(ly) large multi-socket hardware. DaMoN 2020: 3:1-3:9

OCC: Performance impacts

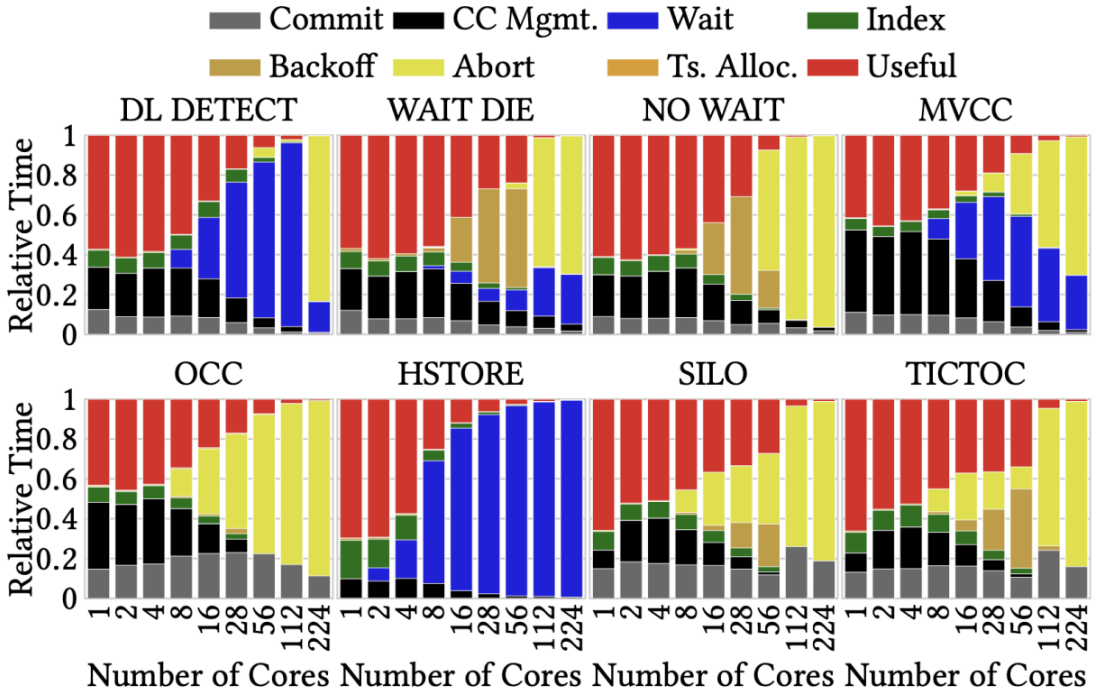


Tiemo Bang, Norman May, Ilia Petrov, Carsten Binnig: The tale of 1000 Cores: an evaluation of concurrency control on real(ly) large multi-socket hardware. DaMoN 2020: 3:1-3:9

OCC: Performance impacts



Software-based timestamp allocation



Using hardware clocks

Contention shifts to latches.

Tiemo Bang, Norman May, Ilia Petrov, Carsten Binnig: The tale of 1000 Cores: an evaluation of concurrency control on real(ly) large multi-socket hardware. DaMoN 2020: 3:1-3:9

OCC: Key differences

Aspect	Pessimistic (e.g., 2PL)	Optimistic (OCC)
Conflict Handling	Prevent early (lock before access)	Detect late (validate at commit)
Overhead	High blocking/waiting in contention	Low during execution; aborts in contention
Deadlocks	Possible (needs detection/prevention)	Impossible
Performance Sweet Spot	High contention workloads	Low contention, read-heavy workloads
Aborts/Restarts	Rare (blocked instead)	Possible (wasted work on conflict)
Implementation	Locks + queues	Read/Write sets + validation logic

OCC: Kernel perspective

- While OCC was initially introduced for DBMS, it has big influences on OS kernel synchronizations too
  - Seqlock (sequence locks): provide the closest kernel analog to OCC. They allow lock-free reads with optimistic retry.
  - RCU (read-copy-update): a high-performance synchronization mechanism that allows multiple readers to access a data structure simultaneously, even while an updater is modifying it

Withdraw method revisited

- How to implement OCC-style synchronization for the withdraw method?

```
withdraw (account, amount) {
    balance = get_balance(account);
    balance = balance – amount;
    put_balance(account, balance);
    return balance;
}
```