Introduction to Lock-Free Programming

Mainack Mondal
Sandip Chakraborty

CS 60203
Autumn 2024

Outline

- Locks in OS
  - What are locks and why are they required?
  - Different types of Locks
  - Why are Locks Bad?

- Lock-Free Programming
  - Definition, Different Lock Free Primitives
  - Examples of lockless Data Structures
  - Advantages
  - Problems

Locks in Operating Systems

The Synchronization Problem

In simple terms, it refers to keeping **different threads** on same page
Let's understand this with an example:

Consider a simple *banking* application:

- Basically, it allows you to withdraw/deposit money
- Multi-threaded, centralized architecture
- All deposits/withdrawal sent to central server

What do you think will happen, if two person try deposit money to the *same account* at the *same time* ?

The Synchronization Problem (contd.)

```
balance = balance + sum;

mov eax, balance
mov ebx, sum
add eax, ebx
mov balance, eax
```

What is the final amount stored in variable “*balance*”?

Thread 1 : deposit(Rs. 50)

```
mov eax, balance
mov ebx, sum
```

Context Switch

Thread 2 : deposit(Rs. 100)

```
mov eax, balance
mov ebx, sum
add eax, ebx
mov balance, eax
```

Context Switch

Thread 1 (continued):

```
add eax, ebx
mov balance, eax
```

The Synchronization Problem (contd.)

What **problem** did we see previously?

- A *Race Condition*
- Two Threads tries to update balance at the same time.
- Errors emerge based on the ordering of operations, and the scheduling of threads
- These errors are thus *non-deterministic*

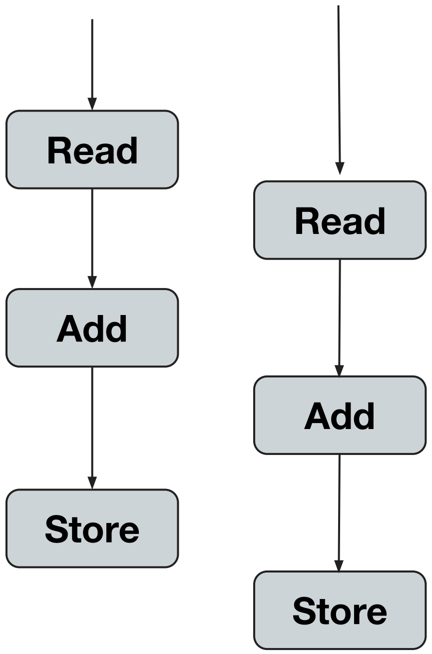
We call this problem **"The Synchronization Problem"**, in other words the **critical section problem.**

The question is … How do you **solve** it ?

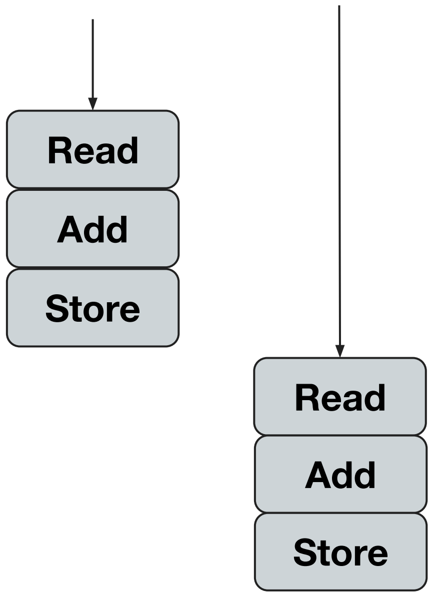
Atomicity

Race conditions lead to unexpected errors when sections of code are interleaved

These errors can be avoided by ensuring the code is executed atomically



Interleaved Execution



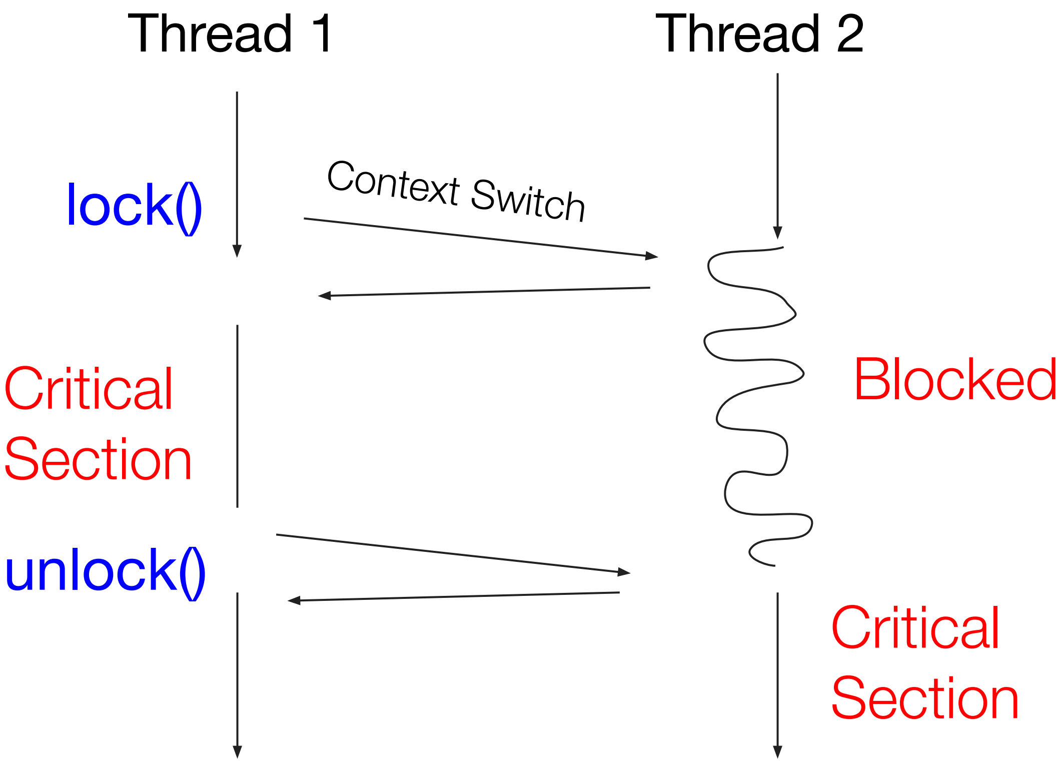
Non-Interleaved (atomic) Execution

How to ensure atomicity?

Ensuring Atomicity: Locks

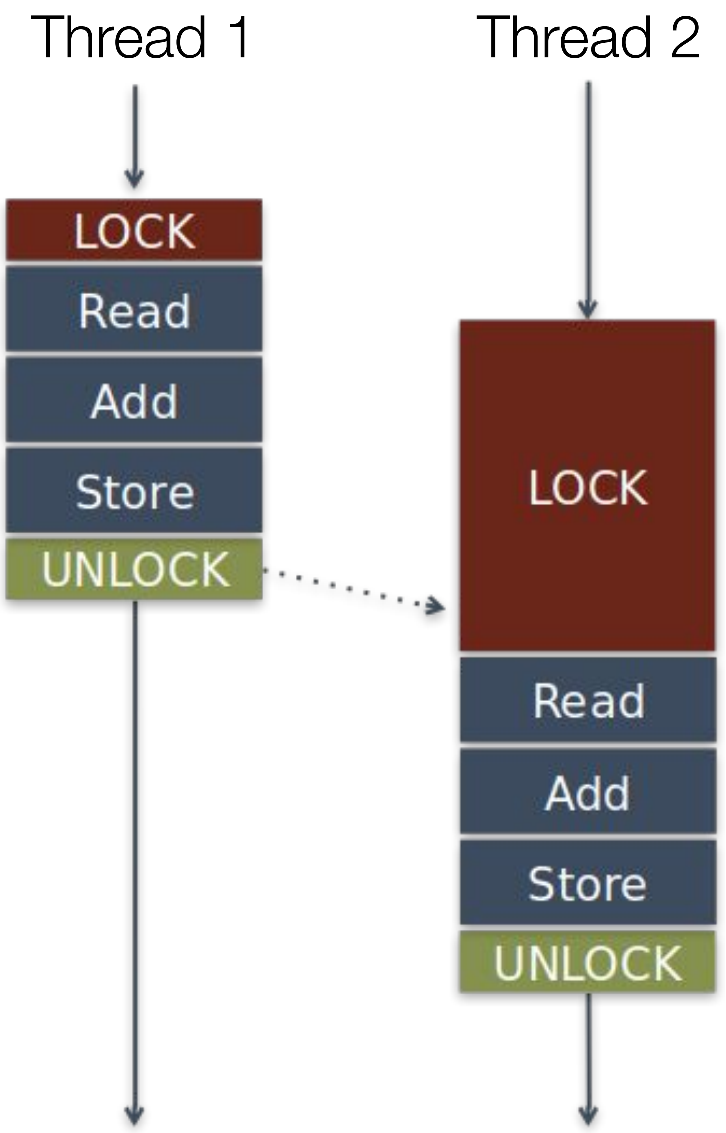
As the name suggests, locks:

- "lock" the critical section
- thus, barring other threads from entering it



Fixing the Bank Example

```
func deposit(int sum){
    lock(lock_ctx);
    balance = balance + sum;
    unlock(lock_ctx);
}
```



Types of Locks

- **Mutex Locks**
  - short for Mutual Exclusion
  - a type of *lockable* object, can be owned by **exactly one** thread at a time
  - When the mutex is locked, any attempt to acquire the lock **will fail**
  - The thread which has locked the mutex, can only **unlock** it

- **Spin Locks**
  - a special type of mutex
  - do **not** use **OS synchronization functions** when a lock operation has to wait
  - keeps trying to update the mutex data structure to take the lock in a loop
  - efficient if lock is not held very often, or is only held for very short periods (why?)

Types of Locks (contd.)

- **Semaphores**
  - relaxed type of *lockable* object
  - maintains a *counter*
  - allows threads to enter critical section unless, counter goes to zero
  - when counter goes to zero, thread has to wait
  - Two main operations (both atomic):
    - `wait` - decrements the counter
    - `signal` - increments the counter

How are binary semaphore (counter = 1) **different** from mutex lock ?

But …
Why switch to Lock-Free Programming?

Problems with Locks

- Locks cause **Deadlocks**

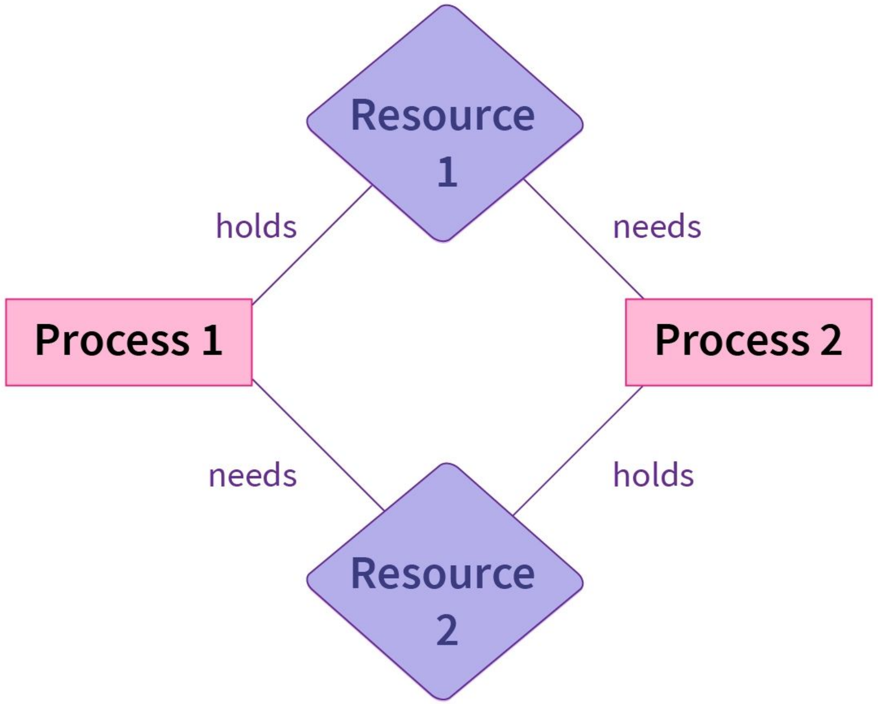


Image credits: scaler.com

- Locks sometimes causes **performance bottlenecks**
  (Source: why-mutex-lock-on-c-affects-multithreading-efficiency-so-badly)

Problems with Locks (Contd.)

- Mutex locks cause **context switches**

  Eg: For a testbench with 100 threads with each updating a shared variable 1000 times

  `perf` benchmark:

Configuration	Context Switches	Task Clock	Instruction per cycle	CPU Frequency
With Lock	5,965	163.53 ms	0.51	1.849 GHz
Without Locks	9	6.98 ms	0.95	2.939 GHz

Other Issues:

- It also causes busy waiting
- Semaphores causes priority inversion problem
- Locks can cause starvation

Outline

- Locks in OS
  - What are locks and why are they required?
  - Different types of Locks
  - Why are Locks Bad?

- **Lock-Free Programming**
  - Definition, Different Lock Free Primitives
  - Examples of lockless Data Structures
  - Advantages
  - Problems

Lock-Free Programming (Slides partially taken from Geoff Lang, CMU)

What is Lock-Free Programming?

- Thread-safe access to shared data **without** the use of synchronization primitives such as mutexes
- Possible but not practical in the **absence** of hardware support
- Example: Lamport's "Concurrent Reading and Writing"
  - CACM 20(11), 1977
  - describes a non-blocking buffer
  - limitations on number of concurrent writers

How do you design **lock-free** algorithms?

Lock-Free Programming

The simple answer is you don't !!

- Usually, designing them is hard.

Rather, we design lock free data structures

- eg: stack, queue, buffer, map, deque etc.

But … how do you design them?
Well, you use **lock-free primitives**

Lock-Free Primitives

- **Compare and Swap (CAS)**
  - Most basic lock - free primitive
  - It's an instance of so-called atomic RMW (read-modify-write) operation
  - Pseudocode:

```
compare-and-swap(T* location, T cmp, T new){
    // do atomically (in hardware)
    {
        T val = *location;
        if (cmp == val)
            *location = new;
        return val;
    }
}
```

We will see how to use it with an example

Lock-Free Primitives (Contd.)

- **Fetch and Add**
  - another lock - free primitive
  - Basically, used for atomic addition (uses hardware support)
  - Pseudocode:

```
fetch-and-add(T* location, T x)
{
  // do atomically (in hardware)
  {
   T val = *location;
   *location = val + x;
   return val;
  }
}
```

Used in atomic counters

Lock-Free Primitives (Contd.)

- **Load-Linked (LL) and Store-Conditional (SC)**
  - Special Instructions in Hardware (MIPS)

  - Load Linked:
    - Similar to typical load operation
    - Fetches data from memory and puts in the register

```
load-linked(T* ptr){
    return *ptr;
}
```

Lock-Free Primitives (Contd.)

- Store Conditional:
  - It is different from normal store instruction
  - it succeeds if no intervening store to the address has taken place

```
store-conditional(T* ptr, T value){
    if (no update to *ptr since LL to this addr) {
        *ptr = value;
         return 1; // success!
    } else {
         return 0; // failed to update
    }
}
```

How do you use LL-SC to create locks?

Lock-Free Primitives (Contd.)

```
lock(lock_t *lock) {
    while (1) {
        while (load-linked(&lock->flag) == 1); // spin until it's zero

        if (store-conditional(&lock->flag, 1) == 1)
            return; // if set-to-1 was success: done
                    // otherwise: try again
    }
}
```

```
unlock(lock_t *lock) {
    lock->flag = 0;
}
```

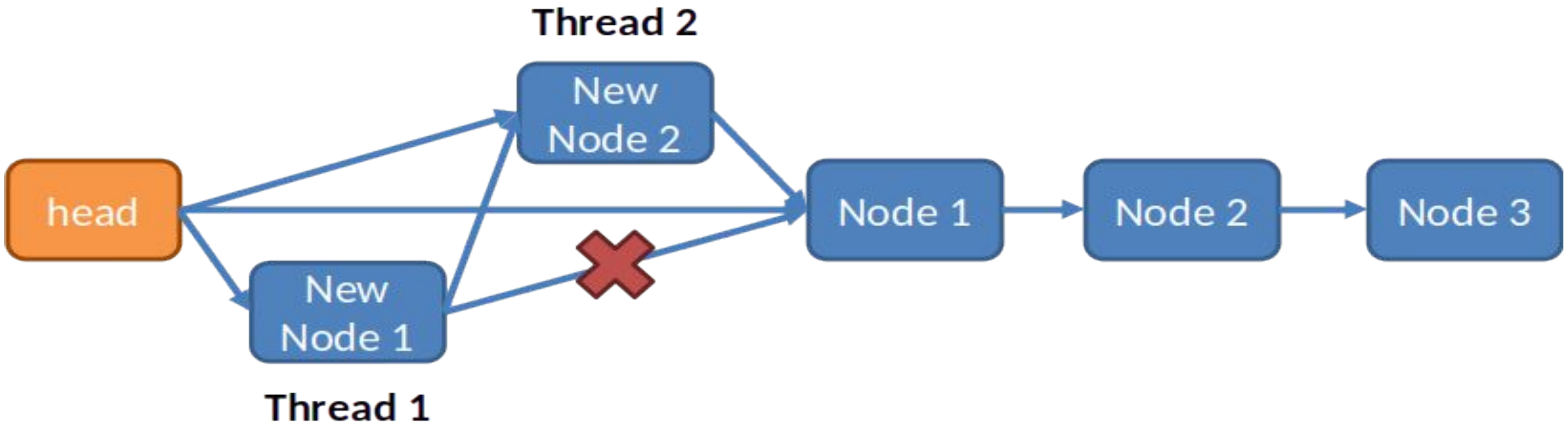
What is this code doing?

Lock-Free Data Structures: Stack

```
struct Node
{
    Node * next;
    int data;
};

Node * head;
// points to the first node
```

```
void push(int t)
{
    Node* node = new Node(t);
    do {
        node->next = head;
    }
    while (!cas(&head, node->next, node));
}
```



Lock-Free Stack (Contd.)

```
bool pop(int& t) {
    Node* current = head;
    while(current) {
        if(cas(&head, current, current->next)) {
          t = current->data;
          delete current;
          return true;
        }
        current = head;
    }
    return false;
}
```

Do you see any problem here?

ABA Problem

- Thread 1 looks at some shared variable, finds that it is ‘A’
- Thread 1 calculates some interesting thing based on the fact that the variable is ‘A’
- Thread 2 executes, changes variable to B
  (if Thread 1 wakes up now and tries to compare-and-set, all is well – compare and set fails and Thread 1 retries)
- Instead, Thread 2 changes variable back to A!
- OK if the variable is just a value, but…

ABA Problem (Contd.)

In our example, variable in question is the stack head

- It’s a pointer, not a plain value!

Thread 1: pop()	Thread 2:
read A from head	
store A.next `somewhere’	
	pop()
	pops A, discards it
	First element becomes B
	memory manager recycles ‘A’ into new variable
	Pop(): pops B
cas with A suceeds	Push(head, A)

ABA Problem (Contd.)

How do solve this problem?

- **Work-arounds**
  - Keep a ‘update count’ (needs ‘doubleword CAS’)
  - Don’t recycle the memory ‘too soon’

- Theoretically not a problem for LL/SC-based approaches
  - However, note the term “Theoretically”
  - Practically, no ideal implementation of LL-SC
  - Hence, leads to spurious failures

References: Lock-Free Data Structures

Advantages of Lock-Free Programming

- No/Less Context-Switches
- Higher CPU frequency and throughput
- No Deadlocks or Priority Inversions
- Faster Multicore Programming