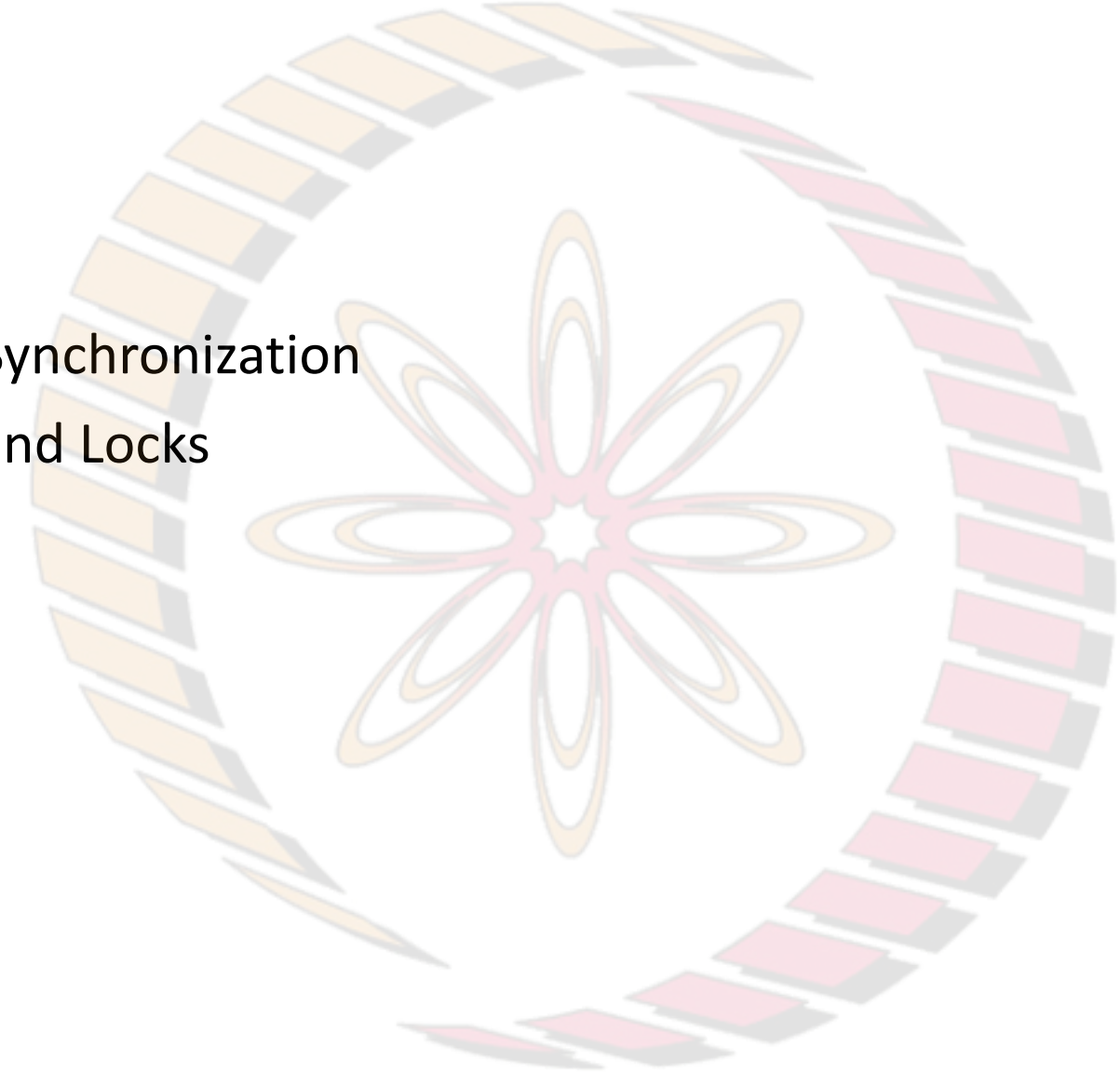
Work Sharing and Synchronization Primitives

Agenda

- Introduction
- Data races and Synchronization
- Critical section and Locks
- Atomics
- Semaphores
- Barriers
- Summary

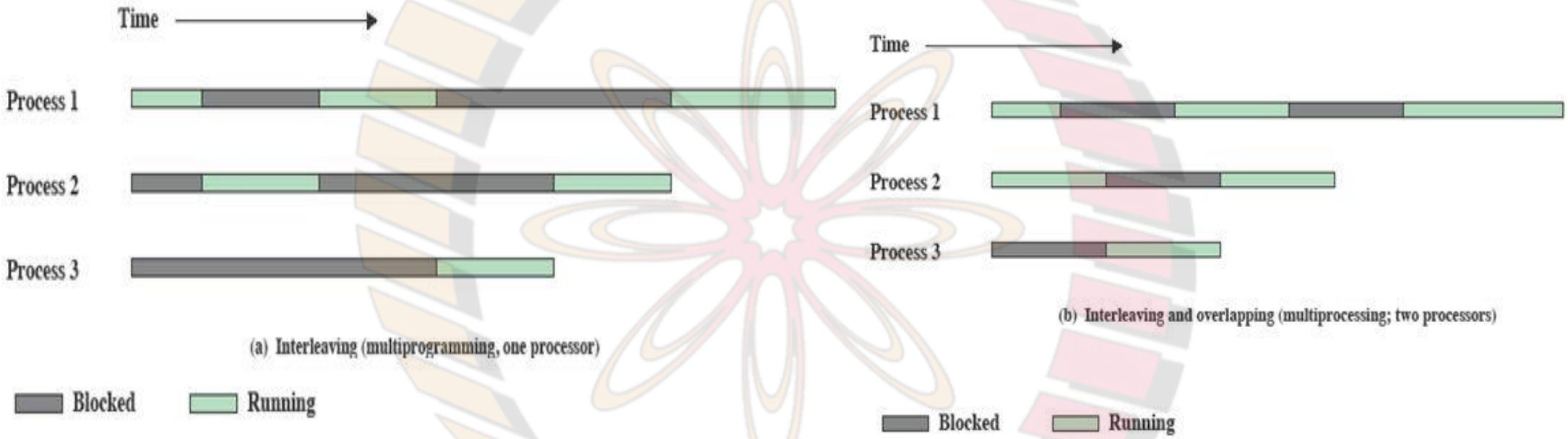


Introduction

Introduction

- Sequential programs involves execution of instructions in an order one after another as dictated by the programmer
- Supercomputers use parallel programming techniques to achieve higher performance
- Parallel programs make use multiple CPU cores, with each core performing a task independently.
  - However, these tasks may be required to interact with each other
- Concurrency allows multiple tasks of a program to run on a single CPU core
  - The core switches between these tasks which actually are threads
- Execution of these threads is interleaved over time, because the CPU core can execute only one thread at a time
- A system with multiple cores, can run the threads in parallel, because the system can assign a separate thread to each core

Principle of Concurrency



- Processes or threads can be interleaved on uniprocessor systems
- Processes or threads can be interleaved AND overlapped on multi-processor systems

Fig. Courtesy: William Stallings, OS Design Principles, 7th Ed.

Data Races and Synchronization

Data Race

- Race condition occurs when two thread try to access the same variable at the same time.
  - This is likely to cause problem sometimes
- Because of races, the result of a computation becomes:
  - Indeterminate (you can’t tell for sure what it will be)
  - Inconsistent (the result varies across different runs of the program)
- The value of a computation depends on the order of execution of multiple processes or threads
  - The output is determined by which thread finishes last
  - However, it is very difficult to know which thread would finish last

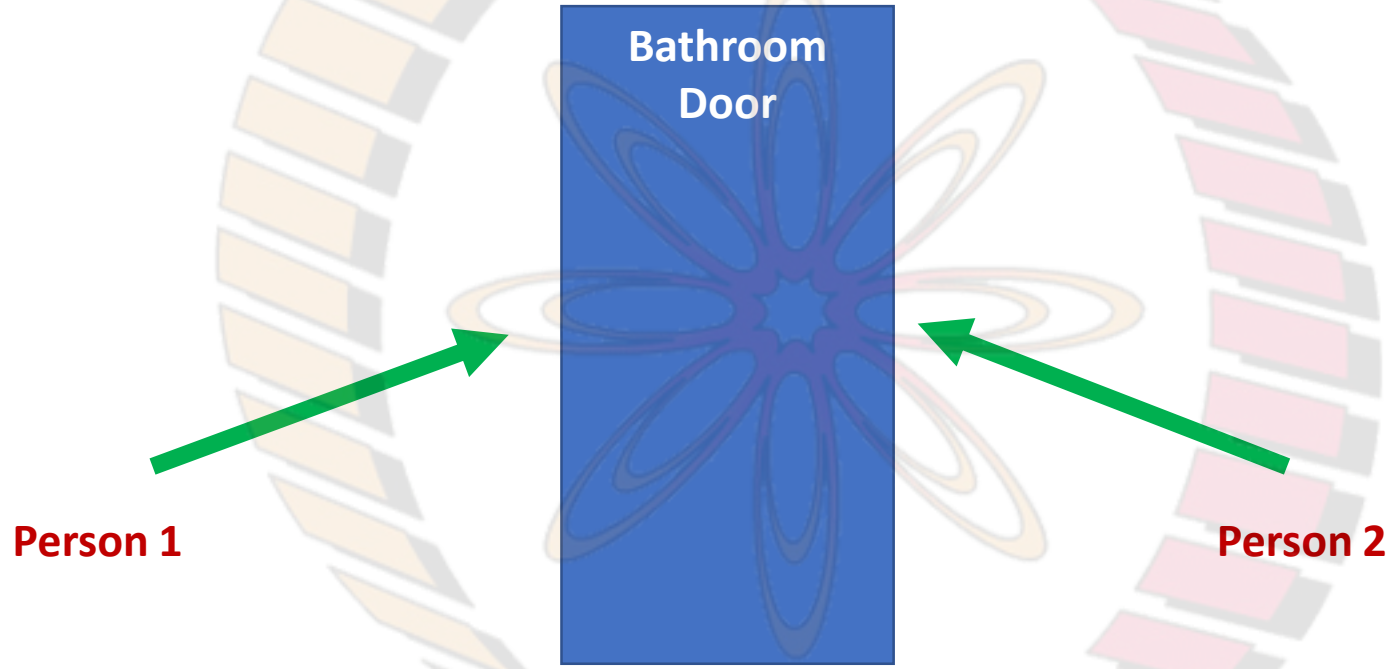
Data Race - Example

- Assume two threads accessing a global variable X
- Each thread increments the existing value of X

Thread 1	Thread 2	Integer value
		0
read value		0
increase value		0
write back		1
	read value	1
	increase value	1
	write back	2

Thread 1	Thread 2	Integer value
		0
read value		0
	read value	0
increase value		0
	increase value	0
write back		1
	write back	1

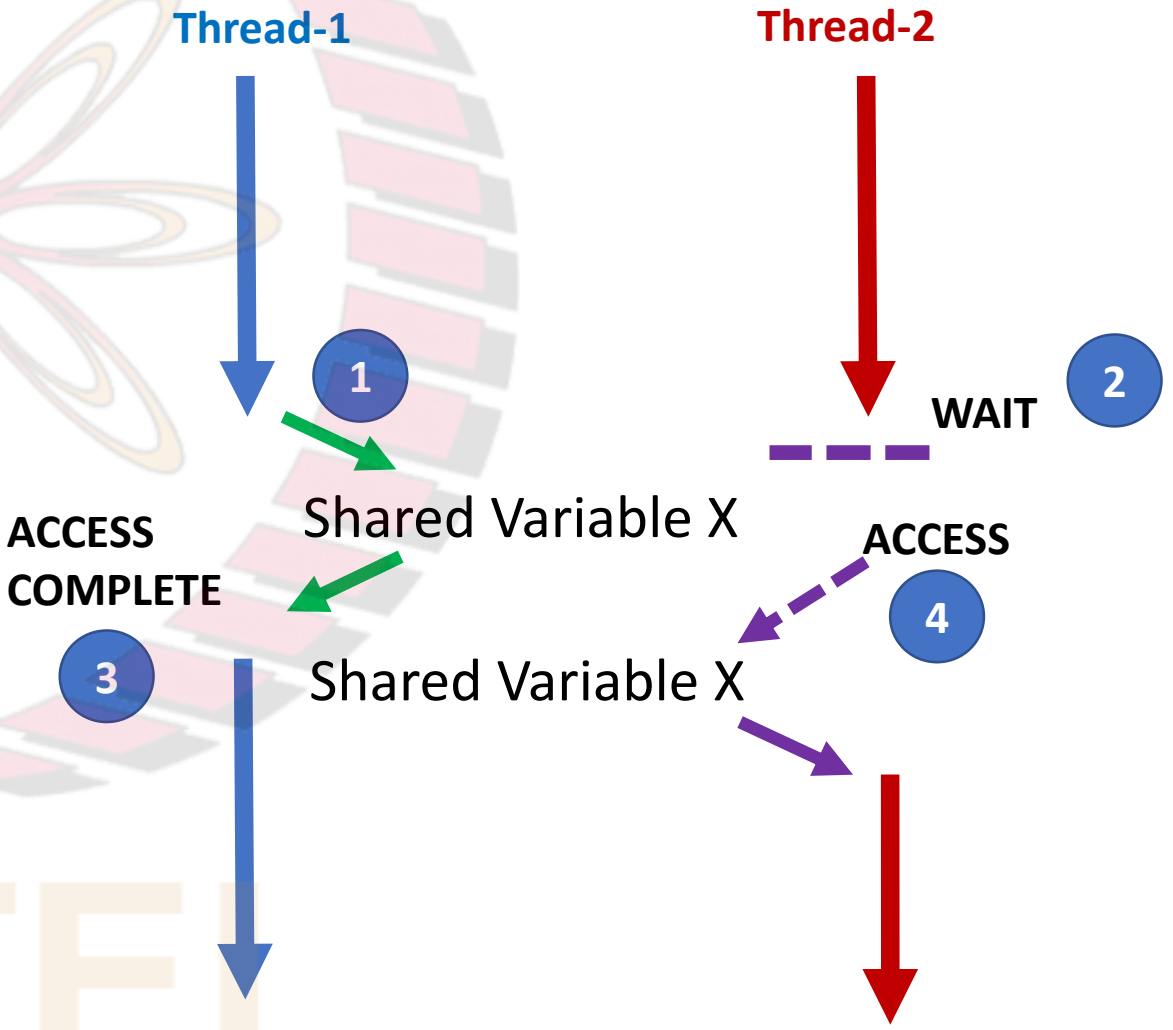
Races: Real Life Example



- So, how do we ensure that there is no contention and both persons get to access the bathroom?

Synchronization – Method to deal with Races

- Take the bathroom door as an example – how do you guarantee proper access?
  - Use Lock and Key
- Take the example of the shared variable “X”



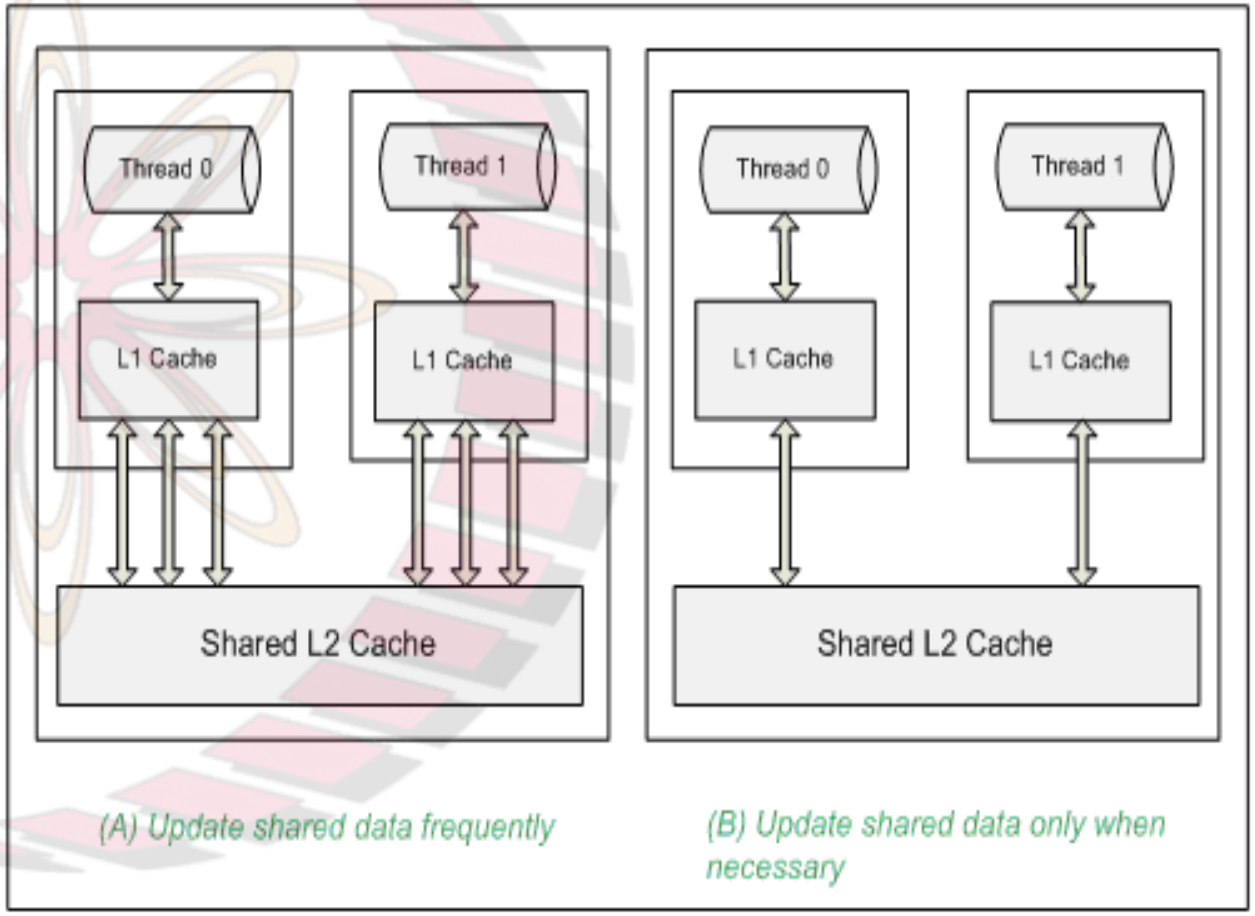
Synchronization Overhead

- Synchronization mechanisms may create overheads
  - Because threads have to WAIT!
  - May lead to reduction in application performance

- Hence, Minimizing the amount of synchronization is desirable by restructuring the code/algorithm

Synchronization – Less Overhead Approach

- Fig. (A) shows shared data being updated frequently in shared L2 cache
- Fig. (B) shows update of shared data only when necessary
- This way, the threads are not unnecessarily burdened with updating cache
  - They hold private copy and update only when needed



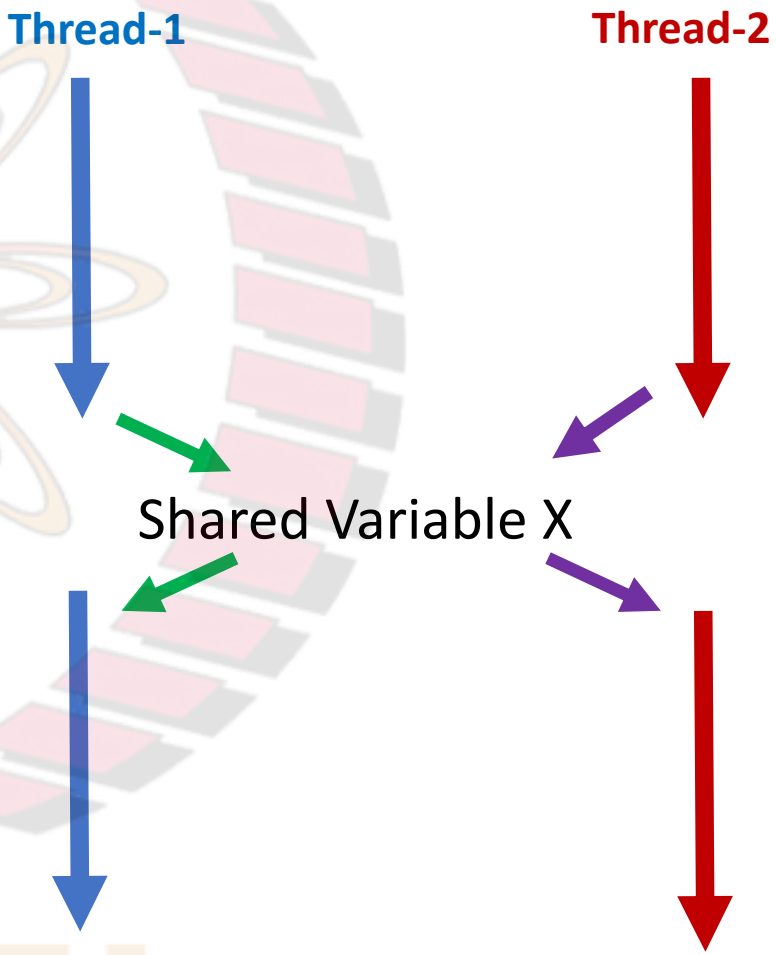
Source:
https://software.intel.com/content/www/us/en/develop/articles/software-techniques-for-shared-cache-multi-core-systems.html

Critical section and Locks

Shared Variable & Threads

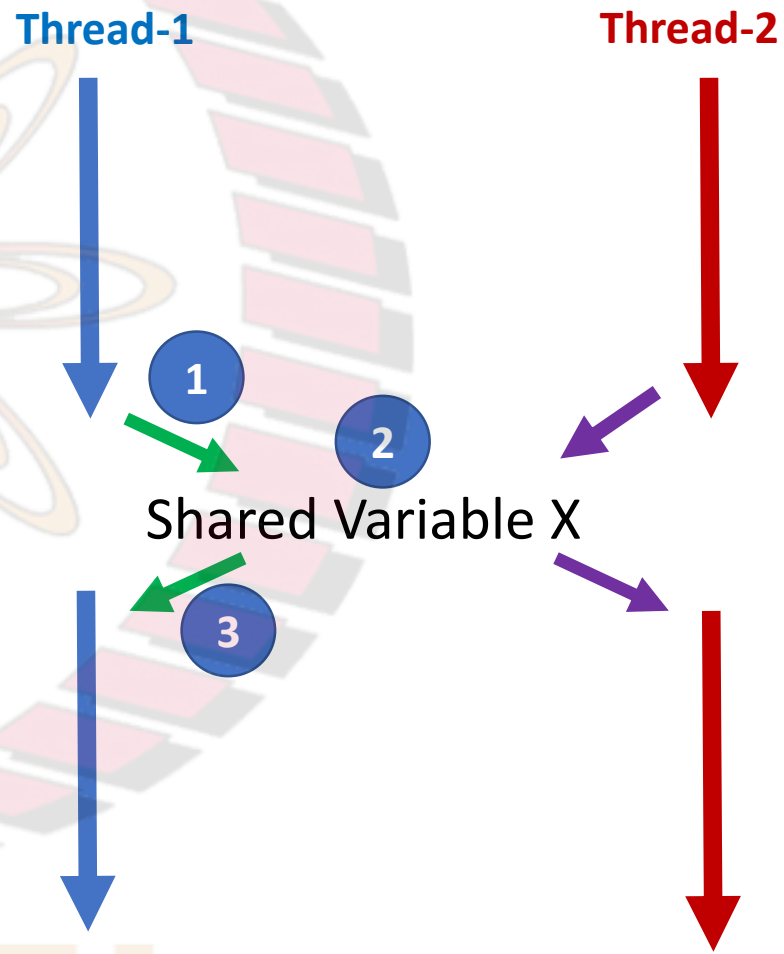
- Contention-Only Threads
  - Threads share cache but do not share data

- Data-Sharing Threads
  - Threads share data
  - Where is the shared variable X?
    - In Shared Memory like L2 cache shown on the previous slide



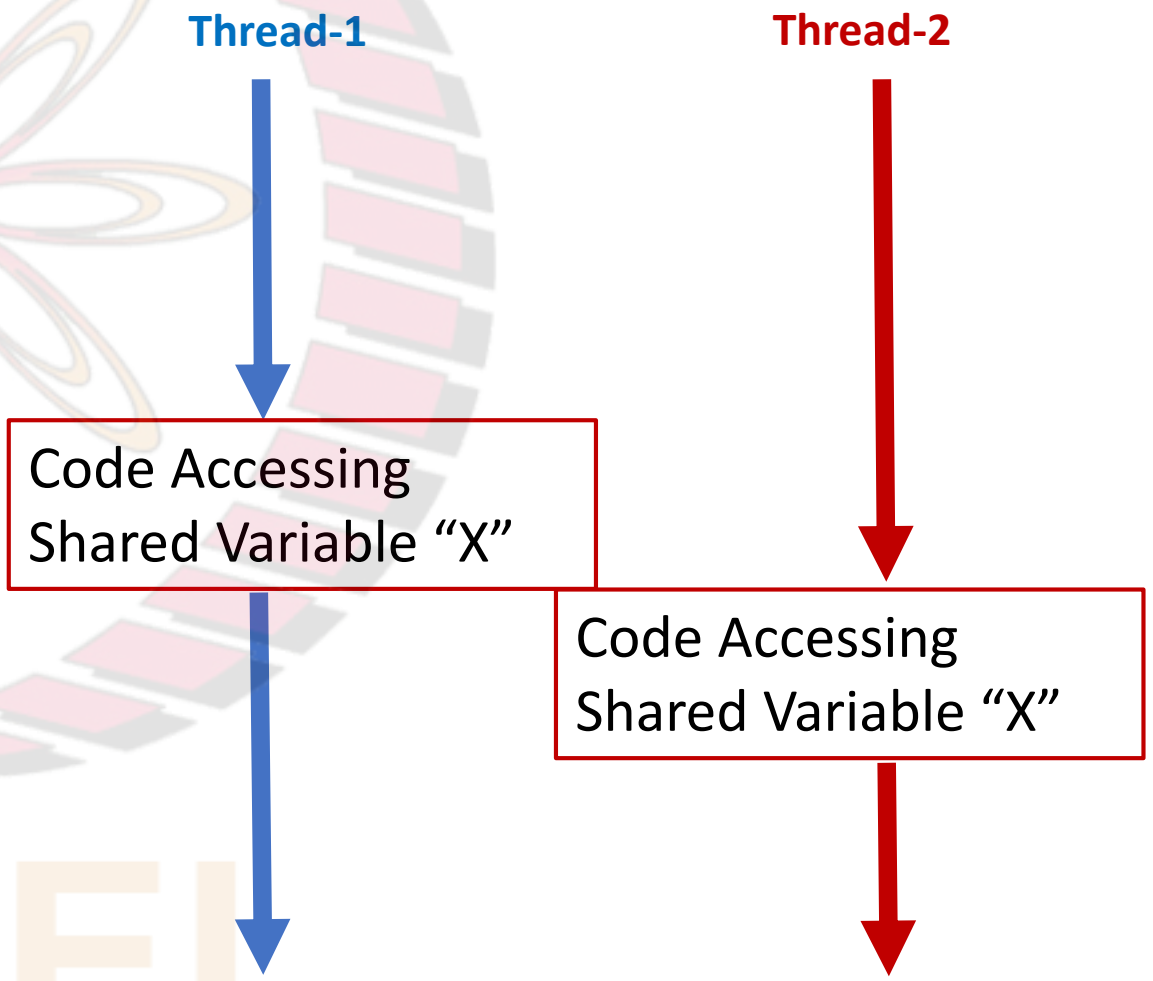
Shared Variable & Threads

- Every thread is performing the following three tasks on the shared variable “X”
  - Read the variable 1
  - Do some operation on the variable 2
    - Like increment it
  - Write the variable 3
- A thread may ONLY READ also



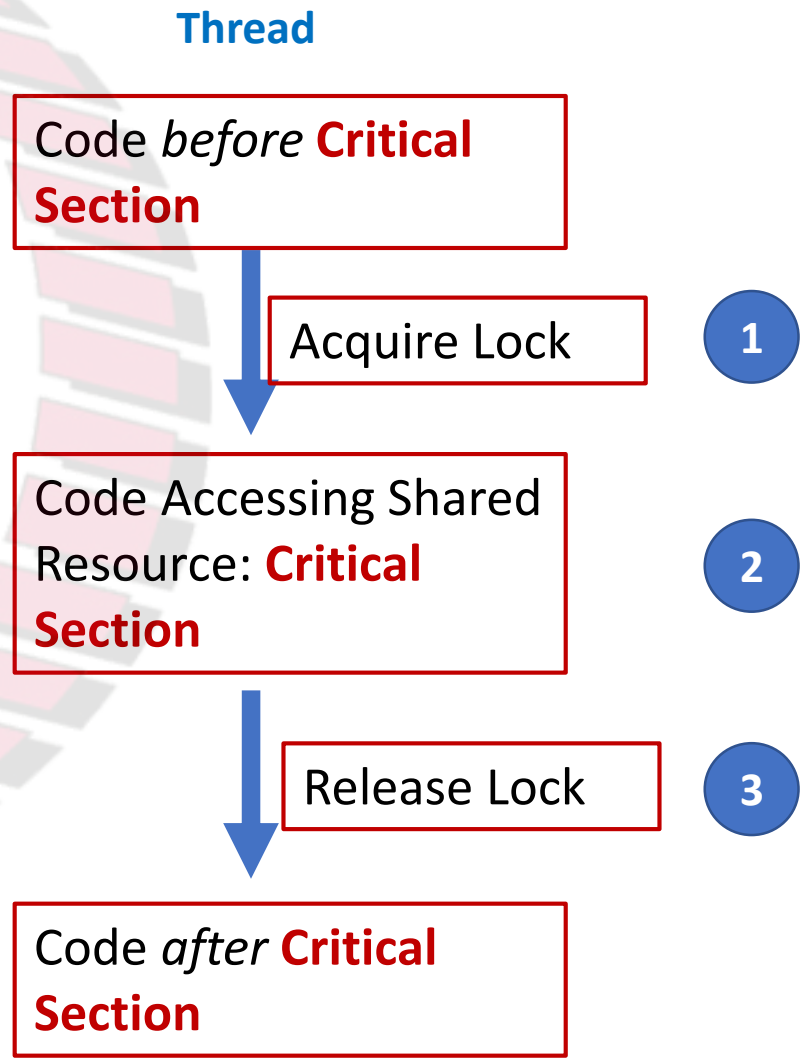
Thread's access to shared variable: Critical Section

- So how does a thread do those three operations?
  - By using code of course!
  - That code is called "Critical Section (CS)"
  - It is called critical because its execution should not be interrupted by any other thread
  - Once a thread enters its CS, does its job and exits CS, only then any other thread can enter its own CS for the same shared variable



Critical Section and Lock

- So, how can one guarantee non-interruption of a Critical Section?
  - Through a **Lock** mechanism
  - This ensures *exclusive access to the shared resource*
  - This could be a memory location, a peripheral device or a network connection or a data structure



Mutex Lock

- **Mutex** means “Mutual Exclusion”
- A mutex lock uses “lock” and “unlock” functions to protect critical section
- Here, *pthread.h* is a header file to create threads
- *count_mutex* is a mutex
- *pthread_mutex_lock* locks *count_mutex*
- *Pthread_mutex_unlock* unlocks (released) *count_mutex*

```
#include <pthread.h>

pthread_mutex_t count_mutex;
long count;

void
increment_count()
{
    pthread_mutex_lock(&count_mutex);
    count = count + 1;
    pthread_mutex_unlock(&count_mutex);
}
```

Internals of Mutex

- With every mutex is associated a binary variable i.e a variable that can take a value 0 or 1.
- A mutex is initialized to 1
- When a process or thread enters a critical section, it decreases the mutex value to 0 – this *acquiring lock on the shared resource*
- Any other process or thread trying to enter critical section for the same shared resource will see the mutex value to be 0 – hence, it would wait
- When the process or thread that acquired the lock, exits the critical section it increments mutex to 1 – thus, making it available for acquiring by any other thread or process

Atomics

Atomics

- Are instructions that execute without any interruption
- Examples:
  - TestAndSet()
  - CompareAndSwap ()
- Fig. A: Defintion of testAndSet() instruction
- Fig. B: Implementing mutual exclusion using testAndSet()
- Boolean variable lock is initialized to false

```
boolean test_and_set(boolean *target) {
   boolean rv = *target;
   *target = true;

   return rv;
}
```

Fig. A

```
do {
   while (test_and_set(&lock))
      ; /* do nothing */

      /* critical section */

   lock = false;

      /* remainder section */
} while (true);
```

Fig. B

Fig. Courtesy: Abraham Silberschatz – Operating Systems Concepts, 9th Ed.

Atomics

- CompareAndSwap()
- Fig. A: Definition of CAS
- Fig. B: Mutual exclusion using CAS
- Always returns the original value of the variable *value*

```
int compare_and_swap(int *value, int expected, int new_value) {
   int temp = *value;

   if (*value == expected)
      *value = new_value;

   return temp;
}
```

Fig. A

```
do {
   while (compare_and_swap(&lock, 0, 1) != 0)
      ; /* do nothing */

      /* critical section */

   lock = 0;

      /* remainder section */
} while (true);
```

Fig. B

Fig. Courtesy: Abraham Silberschatz – Operating Systems Concepts, 9th Ed.

Atomics

- Atomic operations are performed on a memory location i.e. an *address*
- The hardware ensures that no other thread can access the memory location until the atomic operation is complete
  - Any other waiting threads are held in a queue
  - All threads perform atomic operation in *serially*

Semaphore

Semaphore – a synchronization mechanism

- A **Semaphore** is an integer variable that, after initialization, can only be accessed through two standard atomic operations: `wait ( )` and `signal ( )`
- A semaphore is a signaling mechanism
  - It uses signals to help threads signal each other when they are entering or leaving a critical section

Semaphores

- An integer value used for signalling among processes.
- Only three operations may be performed on a semaphore, all of which are atomic:
  - Initialize,
  - Decrement (semWait)
  - increment. (semSignal)

Barriers

Barriers

- **Example of a barrier in real life:**
  - Consider the railway crossings in India
  - All vehicles have to wait at the crossing if the barrier is down
    - The vehicles may arrive at different times
    - The barrier acts as a synchronization method – all arriving vehicles get synchronized to the arrival/crossing of the train at the railway crossing

Photo Credit: www.irfca.org

Photo Credit: The Hindu

Barriers

- All threads must wait together at some point
- Program is divided into phases by a *barrier*
- All threads in a phase reach the barrier before they can proceed any further
- Thus, the program phase before and after the barrier cannot be concurrent
- This is useful for
  - Enforcing dependencies
  - Maintaining consistency across shared data
  - Implementing checkpointing
- In this way, barrier is a synchronization mechanism

Summary

Summary

- Concurrency is an essential aspect of parallel programming
- However, it introduces certain side-effects
- These can introduce errors, leading to incorrect results
  - Higher performance can never be at the cost of accuracy
- Hence mitigation of these side effects is necessary
- Synchronization methodologies help in overcoming race conditions
- Locks are used for implementing critical conditions
- Atomics are useful for implementing mutual exclusion
- Barriers is another useful technique for maintaining keeping thread synchronous
- Parallel programming frameworks like OpenMP, MPI and CUDA provides primitives based on these basic techniques

Thank You