Multithreading - I

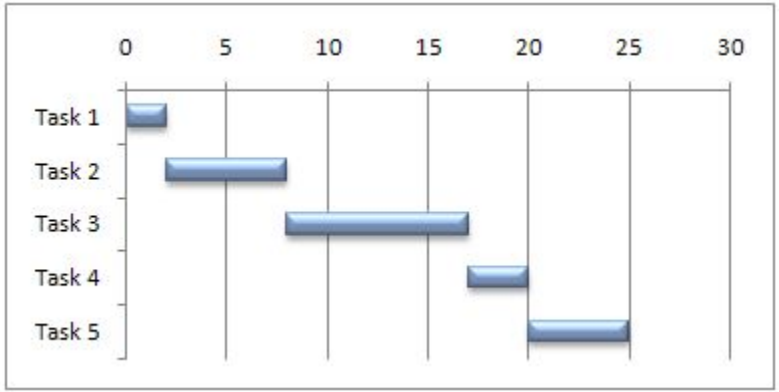
Ashutosh Bohara

Outline

- Threads
- Atomic Operations
- Volatile variables
- Mutex
- Semaphores

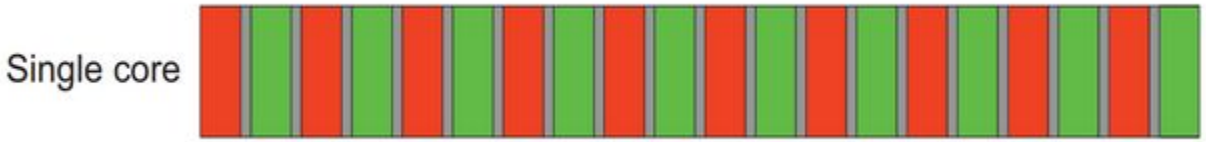
Scheduling

- Selecting a “job” for the CPU to perform
- Performed by the OS
- Several popular strategies:
  - CFS (Linux)
  - Round Robin
  - FCFS



Context Switching

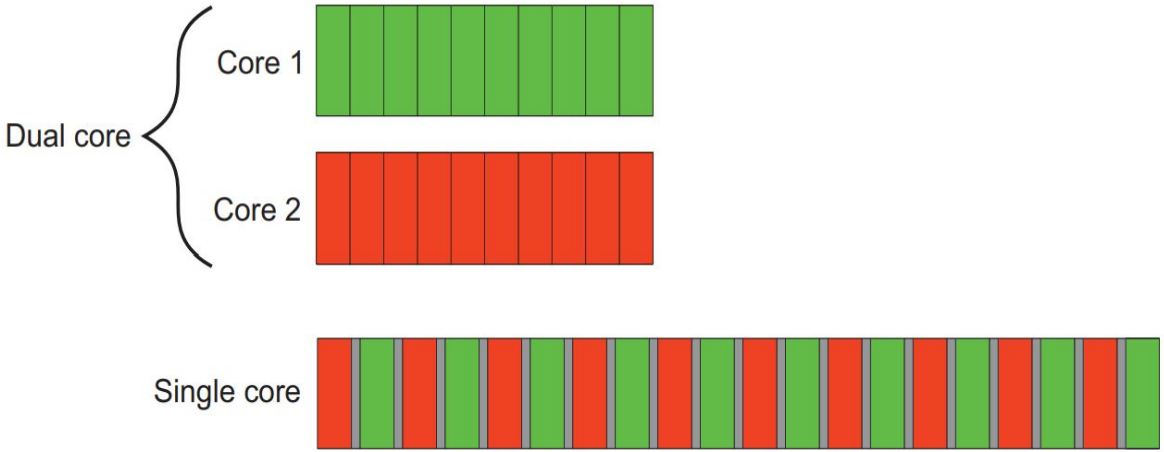
- ➔ Performed by the scheduler(a part of the OS) when changing the thread currently running in a CPU core
- ➔ Saves all the relevant information like program counter, registers, page tables, etc
- ➔ Cost of context switching



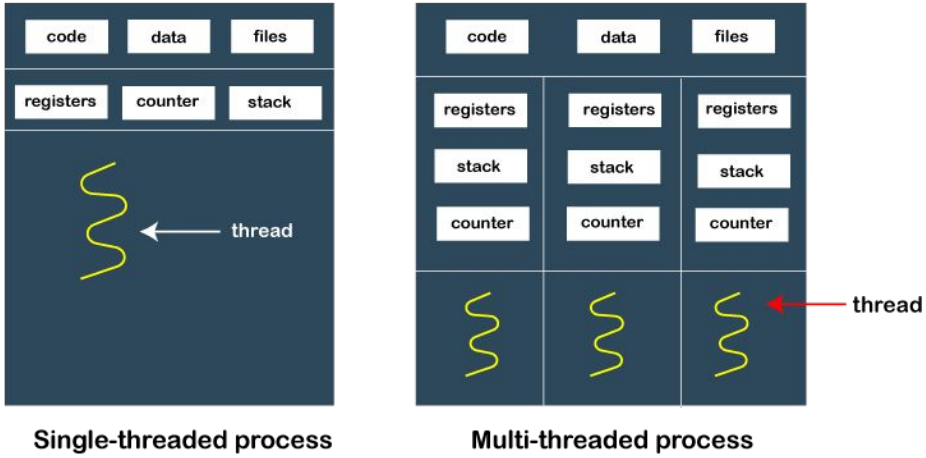
Why Multithreading ?

- A CPU has many cores
- Why not use all of them?
- Parallelizable tasks
  - Matrix manipulation (used in games, ML)
  - Image processing
  - Blockchain
  - Servers

True Parallelism vs Task Interleaving



Processes and Threads



Single-threaded process

Multi-threaded process

Thread

- ➔ Single execution unit that can get scheduled by the OS to run on a CPU core
- ➔ Single sequential flow of control
- ➔ A process can create many threads, all of which share its virtual address space and system resources

Hardware threads vs Software threads

- ➔ CPU with many cores and threads
- ➔ One hardware thread per CPU core (with the exception of hyperthreading and SMT)
- ➔ Software threads are a virtualization over the hardware threads
- ➔ When there are more software threads than hardware threads, at least one of the software threads is always sleeping

From here onwards “thread” will mean a software thread

Thread Operations

- ➔ Creating threads
- ➔ Joining threads
- ➔ Terminating threads
- ➔ Detaching threads
- ➔ Sleep

Let's create a thread

Infinite threads, Infinite Power ??

Thread Support in C

- ➔ C11 added support for multithreading with threads.h
- ➔ Since it was termed optional, MS still hasn't added support for the library
- ➔ Introducing "threading.h"

Passing Arguments

- ➔ Provided during thread creation
- ➔ void* is used to pass arguments of arbitrary type to the function
- ➔ Arguments passed to the child thread must be valid while it is being used in the child thread (code sample)

Adding upto 100000 using multiple threads

Data race/ race condition

- ➔ Conflicting accesses to the same memory location from multiple threads
- ➔ In the previous program, the incrementation operation creates conditions for data race.

Ideal Case

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

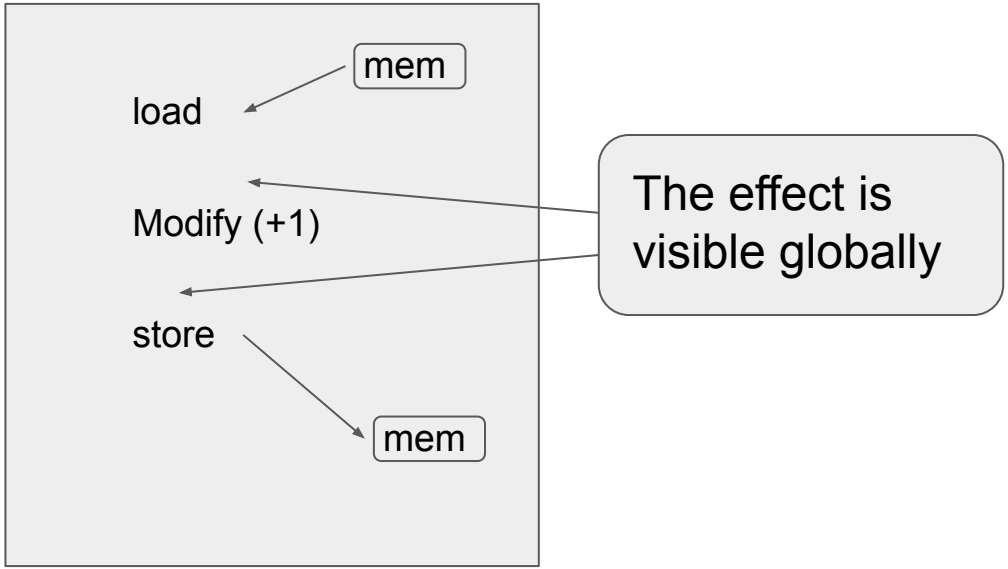
Without Synchronization

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

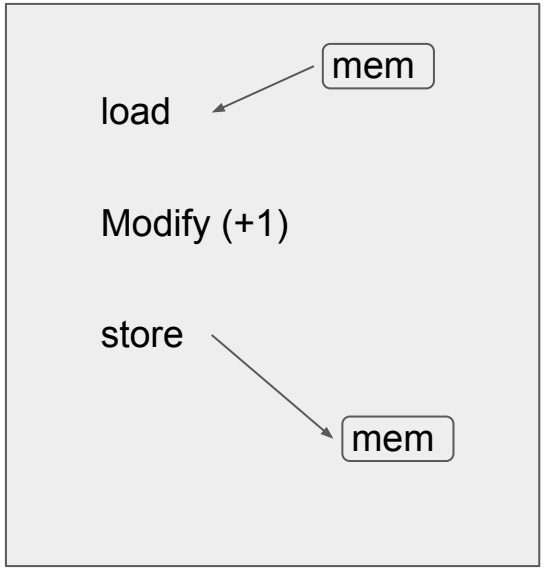
Atomic Operations

- CPU operations that cannot be broken down into smaller steps having an observable effect
- Once started, an atomic operation cannot be left incomplete
- x86 has ‘lock’ prefix added before certain instructions to make them atomic.
  - e.g. ***lock add, lock inc, lock sub***

inc



atomic inc



Performed all at once

Compiler Intrinsics

- Atomic Operations are available in C through intrinsics, which are compiler specific
- But ‘threading.h’ wraps these into a common interface
  - InterlockedIncrement
  - InterlockedDecrement
  - InterlockedAdd
  - InterlockedCompareExchange

Let's add Atomic Increment to our program

Compiler Optimizations

- Compilers are very sophisticated and impressive programs
- What ?, not How ?
- Some of the basic optimizations performed by a compiler are
  - Loop unrolling
  - Compile time calculation
  - Variable caching
  - Function inlining
  - Dead code elimination
  - Code reordering

```cpp
int foo(){
    int res = 0;
    for(int i = 0; i <= 10000; ++i) {
        res += i;
    }
    return res;
}
```

```asm
1   _foo    PROC
2           xor     eax, eax
3           mov     ecx, eax
4   $LL4@foo:
5           add     eax, ecx
6           inc     ecx
7           cmp     ecx, 10000
8           jle     SHORT $LL4@foo
9           ret     0
10  _foo    ENDP
```

```asm
1   foo:
2           mov     eax, 50005000
3           ret
```

Variable Caching ??
Seems problematic!

Volatile

- ➔ Type qualifier keyword in C
- ➔ Information for the compiler that its value may change at any time.
- ➔ It is widely used in embedded programming where a register that can be modified by an I/O operation is directly mapped to a variable
- ➔ Value of a variable may be changed by some other thread

Let's add Volatile to our program

Questions?

Memory Barriers

- ➔ Compiler can reorder segments of code
- ➔ But that's a problem for multithreaded programs (let's see how)
- ➔ A ***software*** memory barrier tells the compiler the compiler to not reorder memory operations across the barrier.


BUT WAIT
THERE'S MORE

Hardware Barriers

- ➔ The CPU can also reorder instructions (out-of-order processing)
- ➔ We need a ***hardware*** memory barrier that prevents the reordering of CPU instructions.
- ➔ Unlike software barriers, hardware barriers generate actual assembly code
  - ◆ lfence, mfence, sfence ,
- ➔ Compiler specific, available as intrinsics

Limitations of Atomic Operations

- ➔ There are only so many instructions a CPU can perform atomically
- ➔ Writing to a file from multiple threads?
- ➔ Read/write from/to a console from different threads?
- ➔ “atomic like” behavior, but more generalized

Mutex

- Mutex (Mutual Exclusion) is a simple data structures with states:
  - Locked
  - Unlocked
- Threads that share data use mutex to control access to it
- Critical section is executed only if mutex is not locked by another thread by locking the mutex first

Critical Section

- ➔ Must execute in isolation from rest of the program.
- ➔ Should be no interference from other threads
- ➔ Access to the shared data.

Mutex Example

Mutex Implementation

```
typedef struct Mutex {
    uint32_t value;
} Mutex;
```

Mutex Implementation

```c
static void MutexInit(Mutex *mutex) {
    mutex->value = 0;
}

static void MutexLock(Mutex *mutex) {
    while(mutex->value) {}
    mutex->value = 1;
}

static void MutexUnlock(Mutex *mutex) {
    mutex->value = 0;
}
```

What happened ??

```
MutexInit PROC                                          ; COMDAT
        mov     DWORD PTR [rcx], 0
        ret     0
MutexInit ENDP

mutex$ = 8
MutexLock PROC                                          ; COMDAT
        mov     DWORD PTR [rcx], 1
        ret     0
MutexLock ENDP

mutex$ = 8
MutexUnlock PROC                                        ; COMDAT
        mov     DWORD PTR [rcx], 0
        ret     0
MutexUnlock ENDP
```

How do we fix that ??

What now ??

- ➔ The problem is same as before.
- ➔ The access to the variable value is not atomic.
- ➔ **How do we make it atomic ??**

C++ Copy

```
LONG InterlockedCompareExchange(
  [in, out] LONG volatile *Destination,
  [in]      LONG          ExChange,
  [in]      LONG          Comperand
);
```

Questions?

Semaphores

- ➔ Generalised mutex
- ➔ Used to manage concurrent processes by using a simple integer value
- ➔ An integer variable that is shared between threads
- ➔ Binary semaphores vs counting semaphores

Mutex vs Binary Semaphore ??

Mutex Ownership

- ➔ Mutexes have a concept of ownership
- ➔ Unlocking a mutex by someone not its owner is undefined
- ➔ But semaphores are used to do exactly that
- ➔ A semaphore locked by a worker thread will be unlocked by the work creating thread

Let’s look at an example

Deadlocks

- ➔ When two or more threads are waiting for each other
- ➔ Two mutexes waiting for each other to release its lock
- ➔ When a thread is deadlocked, it can not run

Avoiding Deadlocks

- ➔ Timeouts
- ➔ Changing locking order
- ➔ Synchronisation techniques / Condition Variables
- ➔ Lock-free programming

Thank You !!