CS3211
AY22/23 Sem 2
github.com/SeekSaveServe

Lectures

L1: Introduction to Concurrency

Concurrency
- Concurency is pervasive when modern computers have several cores and types of memory
- $\geq$ 2 activities making progress at the same time (overlaping time periods)
- Involves interleaving of instructions from different activities

Parallelism
- $\geq$ 2 processes executing and making progress *simultaneously*
- Hardware dependent: requires hyperthreading (SMT), or multi-core and hardware threads

Processes Vs threads
- Independent Vs Shared memory (address) space
- Both use independet stack
- Expensive Vs Cheap context switch
- OS facilitated vs Non OS facilitated inter-process/thread Communication
- Expensive (copy on write mediates this somewhat) Vs Cheap creation

Interrupts
- Asynchronous (independent to program execution)
- Used by OS to interact with the programme
- Triggered by external events (e.g. I/O, timer, hardware failure)

Exceptions
- Synchronous (dependent on program execution)
- Used by process to interact with the OS
- Triggered by process error (e.g. underflow, overflow)

User thread
- Library created, linked to one kernel thread

Race condition
- Outcome depends on reltive ordering of operations on ge 2 Threads
- a flaw that occurs when the timing or ordering of events affects a program's correctness

Data Race
1. $\geq$ 2 concurrent threads concurrently access a shared resource without Synchronisation / fixed ordering
2. At least one modifies shared resource
3. Causes undefined behaviour

Mutex
- Creates critical section can be treated as a large atomic blocks
- Only one thread at a a time
- Supported by a hardware instruction (CAS, test and set etc)
- **Properties:** Mutex, progres, bounded wait, performance
- Provides **serialisation** (less concurrency)

Critical section
- Safety: nothing bad happens
- Liveness: Something good (progress) happens
- Performance: depends on aggregate performance of all threads

Locks
- Primitive that is provided b the hardware, minimal semantic
- E.g. Test and set

Deadlock iff
1. Mutex: One resource held in a non-shareable state
2. Hold and wait: One proces holding one resource and waiting for another resource
3. No-preemption: Resource and critical section cannot be aborted externally
4. Circular wait
5. Note: Lock free can deadlock

Dealing with deadlock
- Prevention: Eliminate one of the above conditions (E.g. hold all locks at the start)
- Detection and recovery: Look for cycles in dependencies (E.g. wait for graph)
- Avoidance: Control allocation of resources

Starvation
- One process cannot progress becausee another process is holding on a resource it needs
- Side effect of scheduling algorithm
- Wait-die and wound-wait are possible solutions, if priority of processes is preserved

Advantages of concurrency
- Performance
- Separation of concerns

Disadvanageous of concurrency
- Maintenance and debugging

Task parallelism
1. Do the **same type of** work faster
2. Task dependency graph can be parallel
3. Make tasks specialists: Same type of tasks are assigned to the same thread
4. Divide a sequence of tasks among threads to solve complexed task
5. **Pipeline:** 1 type of thread for one phase of execution

Data parallelism
1. Do **more work** in the same amount of time
2. Divide data to chunks and execute by different threads
3. Embarasingly parallel tasks

Challenges of concurrency
1. Finding enough parallelism: Amadahl's law
2. Granularity of tasks
3. Locality
4. Coordination and Synchronisation
5. debugging
6. Performance and monitroring

L2: Tasks, threads, synchronisation in modern C++

History of CPP
- 1998: No support for multithreading
  1. Effects of language model are assumed to be sequential and there are no established memory model
  2. Different libraries used different memory models
  3. Execution threads were not acknowledged
- 2011: C++11
  1. Standard threads are implemented
  2. Thread aware memory model.Do not rely on platform specific extensions to guarantee behaviour
  3. Atomic operations library, class to manage threads, protected shared data etc.

Four ways to manage threads
1. Declare a function that returns a thread

```
void hello() {
    std::cout << "Hello Concurrent 
        World\n";
}
int main() {
    std::thread t(hello);
    t.join();// existing thread waits for
        t to finish
}
```

2. Thread with a function object

```
class background_task {
public:
    void operator()() const {
        do_something();
        do_something_else();
    }
};
/* Calleable object */
background_task f;
std::thread my_thread(f);
```

- $std :: my\_thread(background\_task())$ declares a function that takes a single parameter (type *f() $\rightarrow$ object)
- This is not the same as using a function object!

3. Threads with a lambda expression (local fn instead of a calleable object)

```
std::thread my_thread([]{
    do_something();
    do_something_else();
});
```

Wait
- Uses join() on the thread instance exactly once
- Use joinable to check
- Local variables do not go out of scope
- Blocking

Detach()
- Local variable passed might go out of scope and 'disappear' during runtime, causing invalid access for the detached thread
- Example

```
void oops(){
    int local_state = 0;
    /* Reference passed might become
        invalid */
    func my_func(local_state);
    std::thread my_thread(my_func);
    my_thread.detach();
} /*oops ends here and local_state will
    be destroyed */
```

- Not blocking

Passing arguments
1. by value $std :: threadt(f, 3, "hello")$
2. by reference $std :: threadt(f, 3, buffer)$
   - Buffer is a charbuffer that only gets converted to str when we call f
   - Hence it is possible for buffer to go out of scope
   - Fix: Use explicit cast $std :: thread(f, 3, std :: string(buffer))$
   - **Major issue** with passing by reference is that threads outside of the scope can use it in **unsafe** ways. E.g. Not using mutex on shared data, deletion etc
3. by copy

```
void update_data_for_widget(widget_id w,
    widget_data& data);
void oops_again(widget_id w) {
    widget_data data;
    /* a copy of data is passed */
    std::thread t(update_data_for_widget,
        w, data);
    display_status();
    t.join();
    /* changes made to the copy is not
        reflected to other threads */
    process_widget_data(data);
}
```

- Fix: use reference $std :: threadt(update_data_for_widget, w, std :: ref(data))$

Ownership in C++
- Owner is an object containing a pointer to an object allocated by new for which the owner is responsible for deleting
- Every object on free store (heap, dynamic store) must have **exactly one** owner

C++ Resource Management
- For scoped objects, desctructor is implicit at scope exit
- Free store objects (created using new) requires explicit delete

RAII
- Binds the lifetime of a resource that must be acquired before use to the lifetime of an object

```
/* Handle interrupts using RAII */
void enqueue(Job job) {
    std::unique_lock lock{mut}; //
        constructor locks mutex
    jobs.push(job);// destructor unlocks
        mutex
}
```

Lifetime
- Lifetime begins when storage is obtained and its initialization is complete (except std::allocator::allocate)
- Lifetime ends when :
  - Non-class type (int): destroyed
  - Class type: When destructor is called
  - Reference: begins with initialisation and ends when destroyed. A dangling reference is possible.

Ownership of thread
- Moveable but not copyable

```cpp
void some_function();
void some_other_function();
std::thread t1(some_function);
/* t1 no longer references the thread */
std::thread t2 = std::move(t1);
/* t1 now owns a new thread */
t1 = std::thread(some_other_function);
std::thread t3;
/* t3 owns the thread running some
    function */
t3 = std::move(t2);
/* t1 already owns a thread, this will
    trigger a runtime error */
t1 = std::move(t3);
```

  - C++ compiler cannot catch this
- Ownerhship can be moved out of a function and moved into another function

```cpp
/* Transferrring out of a function */
std::thread g() {
    void some_function();
    std::thread t(some_function);
    return t; // ownership transferred
        out of g()
}

/* Transferring into a function */
void f(std::thread t);
void g() {
    void some_other_function();
    std::thread t(some_other_function);
    f(std::move(t)); // ownership
        transferred into f()
}
```

Mutex in C++
- $std :: lock_g uard$ locks the mutex upon initialisation, unlocks upon destruction
- $std :: lock_g uard < std :: mutex > some_m utex;$
- Group mutex and protected data together in a class rather than use global variables
- Never pass data or pointers (via returns, sotring in externally visible memory, as input to functions etc) when their usage is not guaranteed to be safe

Types of lock guards
- **lock guard** no manual lock, can lock many or one mutex at once without deadlock
- **Scoped lock** accepts and locks a list of mutexes. Can be unintentionally initialized without a mutex
- **unique lock** manual unlock, defers locking using $std :: defer_l ock$, only single mutex.

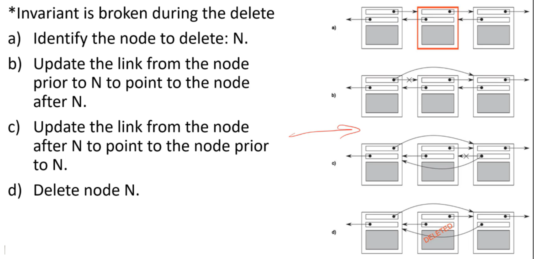
Condition Variable
- Use condition variables to wait for a an event to be triggered by another thread
- Avoids busy waiting

```cpp
std::condition_variable.wait(lock,
    []{return predicate});
```

- if condition is satisfied, returns
- unlocks the mutex and places the thread in block state if condition is not satisfied
- $std :: condition_v ariable.nofiy_o ne();$ to notify one thread waiting on the cond

Spurious Wake
- Thread wakes up from waiting, but is blocked again as the resource required is not available
- Leads to unnecessary context switching
- Use conditionals to prevent spurious wake

Shared DS - Invariants
- Often broken during an update
- **Case study: Doubly LL**



- invariant is temporarily broken during an update, and we need to prevent objects from accessing the DS during this time

L3: Atomics and Memory Model in C++

Reordering of operations
- Compiler may reorder (potentially conflicting) actions for performance
- Not visible to programmers

As-if rule
- Reordering is allowed as long as:
  1. At termination, data written to files is exactly as if the program was executed as written (same final state)
  2. Prompting text that is sent to interactive devices will be shown before the program waits for input
  3. Programs with undefined behaviour is exempted from these rules.

Multi-threading aware memory model
- Using synchronisation constructs (mutexes, barriers etc) should preclude the need for a memory model since they serialise threads
- Memory model gives us more flexibility and speed by getting us closer to the machine

Stucture of memory model
- Every object has a memory location, some occupy exactly one, some occupy many
- The changes in memory location / what is stored there affects other threads

Modification Order
- Compose of all writes to an object from all threads in the program
- MO varies between runs, each object has their own MO
- The programmer is responsible that threads agree on the MO (if not, race condition happens)

MO - Requirements
- The MO of each object is monotonic within a thread
- But the relative ordering of MO of different objects is not guaranteed

MO - Building Blocks
- **Sequenced-before (SB)**
  - Each lines of code in a thread is sequenced before the next line
  - There is **NO** sequenced before in a statement with many function calls
- **Synchronises-with (SW)**
  - Established by a $load$ from T_i reading T_j's $store$
  - Both T_i and T_j are synced with respect to the common value in the MO
- **Happens-before (HB)** When an operation happens before another operation due to SW or SB
- **Interthread Happens Before (IHB)**
  - When a $store$ in T_i established a sequenced before a $load$ in T_j, $sotre_i$ happens before $load_j$
  - IHB $\subseteq$ HB
- **Visible Side effects**
  - Side effect of write A on O is visible to a read B on O if:
    1. A HB B
    2. There is no other side effects to O that happens between A and B
  - If the side effect of A is visible to B then the longest contiguous subset of the side-effects to O (that B does not HB) is known as the visible sequence of side effects
- **Modification Order**

MO - Seq Const
- The default
- All threads must see the same ordering of operation
- Synchronises with a sequentially consistent load of the same variable that reads the value loaded
- Does not apply to atomic operations with relaxed ordering
- Performance penalty when working with weakly ordered machine instructions (common)
- Essentially a serailised monoversion - global total order enforced
- Only guaranteed for data-race free programs (which is difficult since C++ is not as safe as Rust)

MO - Relaxed
- Atomic operations don't conform with SW relationships
- Happens before still applies within the thread $\rightarrow$ monotonicity and SB within the thread is preserved
- No HB between load and store, different store operations from T1 can be viewed out of order by reads in T2
- T1: $x = 1, y = 0$. T2 can see y=0 without seeing x=1 since there is no SW between the two threads even though x=1 HB y=1 in T1.

MO - Acquire Release
- No total modification order, but there is a partial order
- Read - acquire updates about the memory order, load - release updates about the memory order
- A link between acquire and release acts like a barrier

MO - Mixing Models
- Seq const and Release Acquire: load and store of seq const behaves similar to release acquite
- any MO and relaxed: Relaxed behaves like relaxed but is bounded by the other more limiting MO

```cpp
// T1
x.store(true, std::memory_order_relaxed);
y.store(true, std::memory_order_release);
// T2
while (!y.load(std::memory_order_acquire) );
/* Never fires because acquire and release */
/* x.store HB y.store  & y.store SW y.load */
assert(x.load(std::memory_order_relaxed));
```

Atomic Operations
- Compiler ensures necesary synchronisation is in place and enforces MO
- Atomic ops are indivisible
- Atomic load loads either the initial value or the value stored by one of the modifications (cannot be half-done)
- Can be lock free or be implemented using mutex (which wipes off performance gains)
- Not necessarily race free

Fences
- Enforce MO constraints without modifying data, typically combined with atomic operations that uses relaxed MO
- Memory barriers: places a line in code that cannot be crossed
- E.g. atomic thread fence with memory order release prevents preceeding reads and writes from moving past subsequent stores

Tutorials

T1: Threads and Synchronisation

Why mutexes work - standard argument
- Define a critical section that contains all acesses to the shared resource
- Argue that mutex guarantees mutual exclusivity of threads - removes interleaving, data race precluded

Why mutexes work - theoretical argument
- Lock and unlock appears in a single total order
- Only one thread owns the lock at any pointer
- Unlock happens after lock, creating a synchronises with relationship between processes and **serialises the interleaving** - no concurrent access

Monitor
- Allows us to block until a condition becomes true
- The monitor has:
1. A mutex on the critical section
2. A condition variable
3. A condition to wait for

```
std::condition_variable cond;

Job dequeue() {
    std::unique_lock lock{mut};
    /* wait until there is a job */
    cond.wait(lock, [this]() { return
        !jobs.empty(); });
    Job job = jobs.front();
    jobs.pop();
    return job;
    }

void enqueue(Job job) {
    {
        std::unique_lock lock{mut};
        jobs.push(job);
    }
    /* notify one thread waiting on the
        condition variable */
    cond.notify_one();
    }
```

Classical Synchronisation Problems

Comparison between Rust, Go, C++

Ownership

- C++ has RAII to manage resources, moveable but not copyable reference