Synchronization VI

Dmitri Loguinov
Texas A&M University

February 23, 2024
Previous version of search was slow
- CPU utilization 14%, clearly system can handle more, but…
- Lots of time spent on context switches, not doing useful work

Delays in the CC are per command, not per room
- Improvement #1: batching
  (multiple rooms per request)

Next problem: STL set is a major bottleneck
- Improvement #2: write a non-STL hash table

Next problem: out of RAM on STL queue
- Improvement #3: write a non-STL queue with batching

Goal: caves w/4 billion rooms @ 10M rooms per sec
Homework #2

- Suggestion: develop incrementally from hw #1
  - 2a: Introduce CC 2.0 batching (push/pop up to 10K rooms, send them in one message), but keep the rest
  - Confirm correctness; run benchmarks for report question 2
  - 2b: Replace D with bit hash table; confirm result matches 2a
  - 2c: Replace U with custom queue (single push/pop); confirm result matches 2a-2b
  - 2d: Introduce batch-mode push/pop; confirm result
  - 2e: Optimize synchronization; confirm result

- Make sure to print commas in large numbers

------- Switching to level 11 with 421,068,639 nodes
------- Switching to level 12 with 471,263,881 nodes
*** Thread [1080]: found exit room 1C63A9F, distance 12, steps 619,225,089
Chapter 5: Roadmap

5.1 Concurrency
5.2 Hardware mutex
5.3 Semaphores
5.4 Monitors
5.5 Messages
5.6 Reader-Writer
Bounded Producer-Consumer

- Now assume the buffer has some fixed size B
  - Often the queue is a circular array of this size
- Classical version
  - **PC 2.0**

```java
Queue Q;
Mutex m;
Semaphore semaFullSlots = {0, B};
Semaphore semaEmptySlots = {B, B};
Producer() {
    while (true) {
        // make item x
        semaEmptySlots.Wait();
        m.Lock();
        Q.add (x);
        m.Unlock();
        semaFullSlots.Release(1);
    }
}
Consumer() {
    while (true) {
        // no need to check Q.size
        x = Q.pop();
        m.Unlock();
        semaEmptySlots.Release(1);
        // consume x outside
        // the critical section
    }
}
```

- What if bursty consumer or producer?
Bounded Producer-Consumer

- PC 2.0 requires two waits before item can be consumed or produced, potentially inefficient?
  - PC 2.1

```c
Queue Q;
Mutex m;
Semaphore semaFullSlots = {0, B};
Semaphore semaEmptySlots = {B, B};
Producer() {
    while (true) {
        // make item x
        WaitAll (semaEmptySlots, m);
        Q.add (x);
        m.Unlock();
        semaFullSlots.Release(1);
    }
}
```

- Drawback: does not work with eventQuit
  - Need a timeout in WaitAll to check for termination events
Bounded Producer-Consumer

- MSDN says STL objects can never be safely modified from multiple threads
  - Always need a mutex
- Can producer-consumer be implemented completely without synchronization?
  - Suppose we’re allowed to write our own circular queue
- Yes, but only if one thread of each type
  - Producer modifies only Q.tail, while consumer only Q.head

```c
void Q::push (Item x){
    newTail = (tail + 1) % B;
    while (newTail == head)
        Sleep (SOME_DELAY);
    buf [tail] = x;
    tail = newTail;
}
```

```c
Item Q::pop (void){
    while (tail == head)
        Sleep (SOME_DELAY);
    tmp = buf [head];
    head = (head + 1) % B;
    return tmp;
}
```
Bounded Producer-Consumer

• More complex designs are possible
  – One internal mutex for K producers (modifying Q.tail) and another for M consumers (modifying Q.head)

• What if the buffer gets reallocated periodically?
  – Then, whoever is allocating the new buffer needs to obtain both mutexes simultaneously

```cpp
void Q::push (Item x) {
    producerMutex.Lock();
    if (buffer too small)
        consumerMutex.Lock();
        // change buffer to be bigger
        consumerMutex.Unlock();
    deposit x, modify tail
    producerMutex.Unlock();
}
```

```cpp
Item Q::pop (void){
    consumerMutex.Lock();
    if (buffer too large)
        producerMutex.Lock();
        // change buffer to be smaller
        producerMutex.Unlock();
    remove x, modify head
    consumerMutex.Unlock();
}
```

potential for a deadlock
Chapter 5: Roadmap

5.1 Concurrency
5.2 Hardware mutex
5.3 Semaphores
5.4 Monitors
5.5 Messages
5.6 Reader-Writer
• The concept, invented in 1974, is now used in certain programming languages
  - Concurrent Pascal, Modula-2/3, Java, Ada, Ruby

• **Definition**: monitor is a class with two properties
  - No external access to internal objects (all data is private)
  - Each member function is protected by compiler to ensure that only one thread can execute inside

• Compiler locks some **hidden class-specific mutex** on entry and unlocks it on exit

• Mutex is not accessible directly in the code, so a wait for another event inside the monitor may deadlock the whole program

```cpp
class monitor MyClass {
private:
  // some variables
public:
  F1(); F2(); ... // some functions
};
```
Monitors

• **Example**: producer-consumer queue as a monitor
  - How about this:

```cpp
pcQueue::push(Item x) mutex.Lock(); {
    semaEmptySlots.Wait();
    Q.add(x);
    semaFullSlots.Release(1);
} mutex.Unlock();
```

deadlock!

• Obviously a problem

• To fix this, a new type of synchronization primitive was invented that is similar to an event
  - When blocked waiting on this primitive, the compiler secretly unlocks the mutex and when the event is signaled, the compiler secretly locks it again

we want this, but can’t have it because the mutex is invisible to the programmer
Monitors

• **Definition:** condition variable is a class with two ops:
  - Sleep: unlocks the secret mutex of the monitor and blocks on the event; then acquires mutex when event is signaled
  - Wake: signals the event if threads are sleeping; otherwise, does nothing

```cpp
class CondVar {
  Event waitEvent;
  Sleep (); Wake ();
};
```

```cpp
CondVar::Sleep () {
  UnlockWaitLock (mutex, waitEvent);
}
```

```cpp
CondVar::Wake () {
  if (threads are blocked)
    waitEvent.Signal();
  // if nobody is blocked,
  // the wake-up is lost
}
```

• Function UnlockWaitLock():
  - Unlocks compiler mutex and blocks on event
  - Once event is signaled, it blocks on mutex

• Wake is guaranteed to unblock one thread
Monitors

- Producer-consumer with monitors
  - PC 3.0

```c
void pcQueue::push (Item x) { mutex.Lock ();
    while ( Q.isFull () )
        cvNotFull.Sleep ();
    Q.add (x);
    cvNotEmpty.Wake ();
}  mutex.Unlock();
```

```c
Item pcQueue::pop () { mutex.Lock ();
    while ( Q.isEmpty () )
        cvNotEmpty.Sleep ();
    x = Q.remove ();
    cvNotFull.Wake ();
    return x;
}  mutex.Unlock();
```

- When `pop()` finishes, producers compete for `mutex`
  - New threads wanting to enter `push()` and those asleep
- Why is there a while loop around `Q.isFull()`?
  - In certain monitor implementations, `Sleep()` allows new threads to enter the monitor and **steal a wake-up**
  - Thus, awakened thread must check if the queue is still not full before attempting to add to it