CSCE 313-200
Introduction to Computer Systems
Spring 2024

Synchronization VIII

Dmitri Loguinov
Texas A&M University

March 1, 2024

Homework #2

- Request buffer allocated once per thread:

```
#define MAX_BATCH 10000
// set up initial buffer to hold header + MAX_BATCH rooms
char *request = new char [...];
CommandRobotHeader *crh = (...) request;
DWORD *roomArray = (...) (crh + 1);
```

- Then, batch-mode pop works as following:

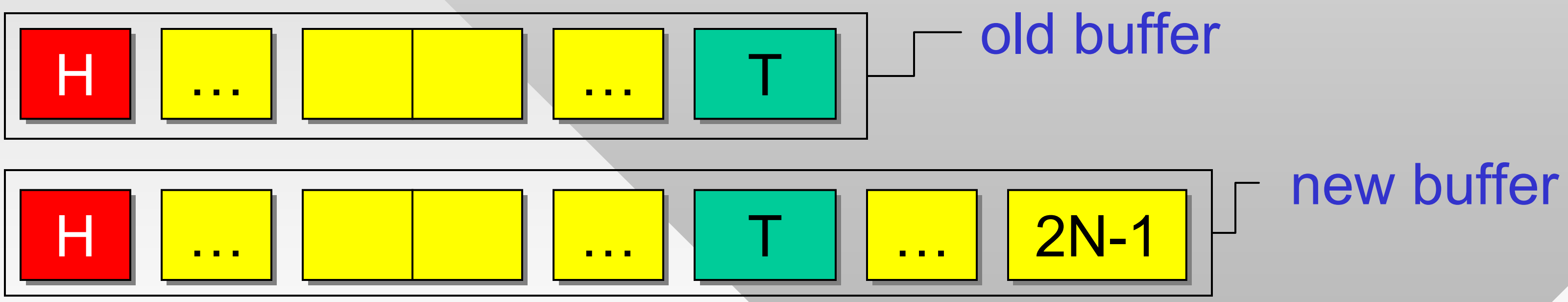
```
int nPopped = Q[cur].pop (roomArray, MAX_BATCH);
// compute msg size based on nPopped
pipe.SendMsg (request, requestSize);
```

- BFS queue class – needs to be written from scratch
  - Encapsulates a buffer with two offsets: head & tail
- Use a private heap inside the queue class
  - HeapCreate(), HeapAlloc(), HeapFree() instead of new/delete

Homework #2

```
// double queue size
size <<= 1;
buf = HeapReAlloc (heap, HEAP_NO_SERIALIZE,
                          buf, size);
```

- Simplified queue without concurrent push/pop
  - Push moves tail by batch size
  - Pop moves head similarly
- When buffer overflows, what operations are needed to double the queue size?



- Simplest is to use HeapReAlloc()
  - If realloc is not in place, the function copies your data

Homework #2

- Hash tables
  - 4B bits in a 512-MB buffer represent all possible nodes
  - InterlockedBitTestAndSet to access the bits
  - LONG array of $2^{32}/32 = 2^{27}$ words (each word is 4 bytes)
  - Make sure to memset to zero during initialization
- Given room ID x, what is the offset and bit # in array?
  - Offset = x >> 5 (equivalent to x / 32)
  - Bit = x & 0x1F (equivalent to x % 32)

  bit ops are faster
- Extra credit: devise a method to interlock less frequently when the number of unique rooms drops close to 0%
  - One line of code

Homework #2

- General structure, gets you ~260 sec runtime on ts

```
char *request = new char
          [sizeof(CommandRobotHeader) +
          MAX_BATCH * sizeof(DWORD)];
CommandRobotHeader *crh =
             (CommandRobotHeader*)request;
crh->command = MOVE;
DWORD *rooms = (DWORD *) (cr + 1);
while (true) {
    if (quit)         // flag set?
        break;

    int batch = 0;
    CS.lock();        // PC 3.4
    if (Q[cur].sizeQ > 0) {
        batch = Q[cur].pop (rooms, MAX_BATCH);
        activeThreads ++;
        // other stats go here
    }
    CS.unlock();
    if (batch == 0) {   // got nothing from Q?
        Sleep (100);
        continue;
    }
    pipe.SendMsg (...);       // send request[]
    pipe.RecvMsg (...);       // read response
    while (rooms left in response) {
        DWORD ID = ... // get next room
        DWORD offset = ...
        DWORD bit = ...
        if (InterlockedBitTestAndSet
         (hashTable + offset, bit) == 0)
          localQ.push (ID);
    }

    CS.lock();
    // batch-pop all elements from
    // localQ into Q[cur^1]
    activeThreads --;
    if (this BFS level is over)
        if (next level empty)
          quit = true;
        else
          cur ^= 1;
    CS.unlock();
}
```

- Extra-credit runtime: <130 sec on P30

Chapter 5: Roadmap

5.1 Concurrency

5.2 Hardware mutex

5.3 Semaphores

5.4 Monitors

5.5 Messages

5.6 Reader-Writer

Performance

Windows APIs

- GetCurrentProcess() and GetCurrentProcessId()
  - Return a handle and PID, respectively
- EnumProcesses(), OpenProcess()
  - Enumerates PIDs in the system, opens access to them
- TerminateProcess() kills another process by its handle
  - ExitProcess() voluntarily quits (similar to C-style exit())
- GetProcessTimes()
  - Time spent on the CPU (both in kernel-mode and user-mode)
- Available resources
  - GlobalMemoryStatus(): physical RAM, virtual memory
  - GetActiveProcessorCount(): how many CPUs
- CPU utilization: see cpu.cpp in sample project

Windows APIs

```
CRITICAL_SECTION cs;
InitializeCriticalSection (&cs);
// mutex.Lock()
EnterCriticalSection (&cs);
// mutex.Unlock()
LeaveCriticalSection (&cs);
```

- WaitForSingleObject
  - Always makes a kernel-mode transition and is pretty slow
  - Mutexes, semaphores, events all rely on this API
- A faster mutex is CRITICAL_SECTION (CS)
  - Busy-spins in user mode on interlocked exchange for a fixed number of CPU cycles
  - If unsuccessful, gives up and locks a kernel mutex
- While kernel objects (i.e., mutexes, semaphores, events) can be used between processes, CS works only between threads *within* a process

Windows APIs

```
CONDITION_VARIABLE cv;
InitializeConditionVariable (&cv);
```

- Condition variables in Windows
  - In performance, similar to CS (i.e., spins in user mode)
  - Secret (monitor) mutex is explicit pointer to some CS
- PC 3.0 that actually works in Windows

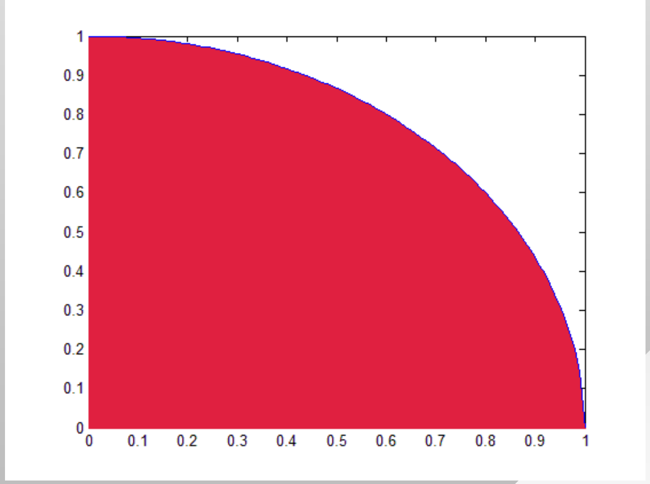
```
pcQueue::push (Item x) {
    EnterCriticalSection (&cs);
    while ( Q.isFull () )
          SleepConditionVariable (&cvNotFull, &cs, ...);
    Q.add (x);
    LeaveCriticalSection (&cs);
    WakeConditionVariable (&cvNotEmpty);
}
```

pop() is similar

- Slim RW locks
  - AcquireSRWLockShared (reader)
  - AcquireSRWLockExclusive (writer)

Performance

- Example 1: compute π in a Monte Carlo simulation
  - Generate N random points in 1x1 square and compute the fraction of them that falls into unit circle at the origin
  - Probability to hit the red circle?
- This probability is the visible area of the circle divided by the area of the square (i.e., 1)
  - Quarter of a circle gives us π/4




```
DWORD WINAPI ThreadPi (LONG *hitCircle) {
    for (int i=0; i < ITER; i++) {
        // uniform in [0,1]
        x = rand.Uniform(); y = rand.Uniform();
        if (x*x + y*y < 1)
            IncrementSync (hitCircle);
    }
}
```

```
main () {
    // run N ThreadPi() threads
    // wait to finish
    double pi =
        4*hitCircle/ITER/nThreads;
}
```

Performance

```
SetThreadAffinityMask (GetCurrentThread(),
                       1 << (threadID % nCPUs));
```

- Six-core AMD Phenom II X6, 2.8 GHz
- Two modes of operation
  - No affinity set (threads run on the next available core)
  - Each thread is permanently bound to one of the 6 cores
- Total k threads
- The basic kernel Mutex
  - $\pi \approx 3.13$
  - CPU $\approx$ 16%
  - Requires 2 kernel-mode switches per increment
  - Runs almost twice as slow with 20K threads

```
IncrementSync (LONG *hitCircle) {
    WaitForSingleObject (mutex, INFINITE);
    (*hitCircle) ++;
    ReleaseMutex (mutex);
}
```

k = 60		k = 20K	
No affinity	Affinity	No affinity	Affinity
384K/s	447K/s	278K/s	220K/s

Performance

- AtomicSwap
  - $\pi \approx 3.1405$
  - CPU = 100% (locks up the computer)
  - Unable to start more than 7K threads since the CPU is constantly busy

```
LONG taken = 0;    // shared flag
IncrementSync (LONG *hitCircle) {
    while (InterlockedExchange (&taken, 1)
                                == 1)
          ;
    (*hitCircle) ++;
    taken = 0;
}
```

k = 60		k = 20K	
No affinity	Affinity	No affinity	Affinity
448K/s	485K/s	–	–

- AtomicSwap and yield
  - When cannot obtain mutex, yield to other threads if they are ready to run
  - $\pi \approx 3.1412$
  - CPU = 100%, but computer much more responsive

```
LONG taken = 0;    // shared flag
IncrementSync (LONG *hitCircle) {
    while (InterlockedExchange (&taken, 1)
                                == 1)
          SwitchToThread();
    (*hitCircle) ++;
    taken = 0;
}
```

k = 60		k = 20K	
No affinity	Affinity	No affinity	Affinity
6.8M/s	6.8M/s	12M/s	11.9M/s

Performance

- CRITICAL_SECTION
  - $\pi \approx 3.1417$
  - CPU = 36%
- Interlocked increment
  - $\pi \approx 3.1416$
  - CPU = 100%
  - Fastest method so far
- No sync (naive)
  - CPU = 100%
  - Concurrent updates lost due to being held in registers and cache

```
CRITICAL_SECTION cs;
IncrementSync (LONG *hitCircle) {
    EnterCriticalSection (&cs);
    (*hitCircle) ++;
    LeaveCriticalSection(&cs);
}
```

k = 60		k = 20K	
No affinity	Affinity	No affinity	Affinity
6.9M/s	15.9M/s	7.3M/s	12.8M/s

```
IncrementSync (LONG *hitCircle) {
    InterLockedIncrement (hitCircle);
}
```

k = 60		k = 20K	
No affinity	Affinity	No affinity	Affinity
19.4M/s	19.2M/s	19.1M/s	19.0M/s

```
IncrementSync (LONG *hitCircle) {
    (*hitCircle)++;
}
```

k = 60		k = 20K	
No affinity	Affinity	No affinity	Affinity
25.5M/s	19.9M/s	20.6M/s	20.2M/s
$\pi \approx 1.21$	$\pi \approx 1.03$	$\pi \approx 0.96$	$\pi \approx 1.33$

Performance

```
DWORD WINAPI ThreadPi (LONG *hitCircle) {
    LONG counter = 0;
    for (int i=0; i < ITER; i++) {
          // uniform in [0,1]
          x = rand.Uniform(); y = rand.Uniform();
          if (x*x + y*y < 1)
              counter ++;
    }
    InterlockedAdd (hitCircle, counter);
}
```

- No sync (correct approach)
  - $\pi \approx 3.1415$
  - 202M/s, 100% CPU, bottlenecked by rand.Uniform()
- Lessons
  - Kernel mutex is slow, should be avoided
  - CRITICAL_SECTION is the best general mutex
  - Interlocked operations are best for 1-line critical sections
  - Affinity mask makes a big difference in some cases
- If you can write code only using local variables and synchronize rarely, it can be 1000x faster than kernel mutex and 10x faster than Interlocked