Part 1 – Fine-grained Operations

As we learned on Monday, CMPXCHG can be used to implement other primitives, such as TestAndSet.

```
int CMPXCHG (int* loc, int oldval, int newval)
{
  ATOMIC();
  int old_reg_val = *loc;
  if (old_reg_val == oldval)
     *loc = newval;
  END_ATOMIC();
  return old_reg_val;
}
```

During the performance monitoring tools lecture, we saw another use of CMPXCHG, where the following C code was compiled with OpenMP:

```
...
float add, *pnext;
...
#pragma omp atomic
   pnext[d] += add;
```

And it compiled into (with pseudo-code on right):

```
160:   mov    %eax,%edx                do { float old = pnext[d];
       mov    %edx,0x18(%rsp)
       mov    %edx,%eax
       movss  0x18(%rsp),%xmm2
       addss  %xmm0,%xmm2                  float new = old + add;
       movss  %xmm2,0x18(%rsp)
       mov    0x18(%rsp),%r15d
       lock   cmpxchg %r15d,(%rcx)     } while (CMPXCHG(pnext + d, old, new)
       cmp    %eax,%edx                          != old);
       jne    160
```

Q1) Describe how the pseudo-code will atomically perform the floating point addition.

Q2) Complete the following function using a similar pattern:

void atomic_min(int* addr, int x) {

}

Similar to the CUDA list, there are some operations on x86 that can be made atomic using the lock prefix in the assembly. Originally exposed through compiler intrinsics, now available via including <stdatomic.h> or <atomic> for C or C++ respectively, x86 provides the following operations:

INC, DEC, NOT, NEG, ADD, SUB, AND, OR, and XOR (although some are not exposed).

Q3) Give an example of a simple C operation that needs locking.

Q4) The atomic operations are exposed as atomic_fetch_***op***(volatile *object, operand) that returns the old value. Revise Q3 using an appropriate operation.

Part 2 – Fine-grained Locking

While Monday's lecture discussed lock implementations and we have seen that single memory locations can avoid locks, data structures pose particular problems for locking and synchronization.

For the following questions we will consider an ordered, linked list implementation:

```
struct Node {
   int value;
   Node* next;
};
```

```
struct List {
  Node* head;
};
```

Q5) Given the following list, is it conceptually possible to insert 4 and 12 simultaneously?

Q6) Modify List to guarantee that two or more threads modifying the list will do so safely.

Q7) How many threads can modify the list given your implementation?

Q8) To insert the value 4, what existing linked list nodes must be accessed or modified?

Q9) When inserting the value 12, what existing linked list nodes must be accessed or modified? What overlap, if any, is between this answer and Q8?

Q10) Modify `struct Node` to support simultaneous insert.

Q11) Describe how a linked list insertion algorithm would be modified based on Q10.

Q12) What is one drawback to this approach (there are several)?

Q13) (BONUS) How would a binary tree be similarly modified?

```
struct Tree {
  Node* root;
};
struct Node {
   int value;
   Node* left;
   Node* right;
};
```

Part 3 – Lock-free

Q14) What happens if a thread holding a lock is preempted? Can progress be made?

Lock-free algorithms are defined as those in which there is no possible preemption of one thread that can prevent the other threads from making progress. For the following code you may assume sequential consistency (either implicitly or via fences, atomics, etc).

Q15) Write a single-threaded insert routine for a stack:

```
struct Node {
  Node* next;
  int value;
};
struct Stack {
  Node* top;
};
```

```
void push(Stack* s, Node* n)
{




}
```

Q16) While ultimately the `push` operation makes a Node globally available, which statement(s) can be done prior to the Node becoming global?

Q17) What statement in Q15 makes the Node globally available?

Q18) Using CMPXCHG (a version also exists for 8 byte values) to make the Node available, rewrite the code in Q15:

```
void push(Stack* s, Node* n) {









}
```

Q19) How does your code in Q18 handle the case where a thread is preempted while attempting to push a Node?

Q20) Assume for the moment that a stack `pop` operation also exists, the code in Q18 has a problem. The stack contains addresses A, B, and C. Two threads now call `pop`. One completes successfully and the other is preempted. What is the state of the list?

Q21) Continuing with Q20, `push(D)` happens. `push(A)` happens. What does the list contain? What is the value of top?

Q22) The other thread from Q20 now resumes. What was top when it started executing? What is it now? Can the thread detect that the list is different?

This problem is known as the ABA problem. It is usually addressed by adding timestamp information with the pointer address. The timestamp could be literal or a wrap-around counter. Some architectures have a double compare-and-swap that can process two separate locations atomically. While x86 does not have this instruction, it can swap values up to 16 bytes.

Q23) Modify stack from Q15 so that a 16 byte swap will handle both the pointer and a "timestamp".

Q24) Often part of the pop routine is a sequence of statements, such as the following:

```
Node* top = s->top;
if (top == NULL)
  return NULL;
Node* new_top = top->next;
```

Identify one issue that might occur when two pops calls interleave their execution.