Locking

Lecture 9
CS 390
2/26/08
Mutual Exclusion

- Given a collection of concurrently running threads, how do we guarantee a thread exclusive access to a resource?
  - Disable interrupts.
    - Prevents timer interrupts from triggering scheduling decisions

- What are the limitations of this approach?
Software Approaches

• First approach:

```c
int flag;

void enter_region(int process) {
    while ( flag == 1 );
    flag = 1;
}

void leave_region(int process) {
    flag = 0;
}
```

What’s wrong with this solution?
Alternation

```c
int turn;

void enter_region(int process) {
    while ( turn != process ) ;
}

void leave_process(int process) {
    int other = 1 - process;
    turn = other;
}
```

What’s wrong with this approach?
Software Approaches

Dekker’s algorithm (modified by Peterson 1981)

```c
int turn;
int interested[2];

void enter_region(int process) {
    int other;
    other = 1 - process;
    interested[process] = TRUE;
    turn = process;
    while ( turn == process && interested[other] == TRUE ) ;
}

void leave_region(int process) {
    interested[process] = FALSE;
}

Setting turn to the entering pid releases the other process from the while loop.
```
Bakery Algorithm

- Each process has an id. Ids are ordered.
- Before entering a critical section, process receives a number.
  - holder of the smallest number enters
- Tie break is done using process id
choosing[i] is true if Pi is choosing a number. number[i] holds this number, and is 0 if Pi is not trying to enter.

why is choosing necessary?
Hardware Approaches

- **Test and Set**

  ```c
  int flag;

  void enter_region(int process) {
    int my_flag = test_and_set(flag);
    while (my_flag == 1) {
      my_flag = test_and_set(flag);
    }
  }

  void leave_region(int process) {
    flag = 0;
  }

  int test_and_set(int lock) {
    int old;
    old = lock;
    lock = 1;
    return old;
  }
  ```
Hardware Approaches

- Compare and Swap
  - Three operands:
    - a memory location (V)
    - an expected old value (A)
    - new value (B)
  - Processor automatically updates location to new value if the value stored is the expected old value.
  - Using this for synchronization:
    - read a value A from location V
    - perform some computation to derive new value B
    - use CAS to change the value of V from A to B
public class SimulatedCAS {
    private int value;

    public synchronized int getValue() { return value; }

    public synchronized int compareAndSwap(int expectedValue, int newValue) {
        int oldValue = value;
        if (value == expectedValue) {
            value = newValue;
            return oldValue;
        }
        return oldValue;
    }
}

Lock-free counter:

public class CasCounter {
    private SimulatedCAS value;

    public int getValue() {
        return value.getValue();
    }

    public int increment() {
        int oldValue = value.getValue();
        while (value.compareAndSwap(oldValue, oldValue + 1) != oldValue) {
            oldValue = value.getValue();
        }
        return oldValue + 1;
    }
}
Lock-free algorithms

- An algorithm is said to be *wait-free* if every thread makes progress in the face of arbitrary delay (or even failure) of other threads.
- An algorithm is said to be *lock-free* if some thread always makes progress.
  - permits starvation
- An algorithm is said to be *obstruction-free* if at any point, a single thread executed in isolation for a bounded number of steps will complete.
Lock freedom

- Avoids priority inversion
- Avoids convoying: a process holding a lock is descheduled (e.g., page fault, I/O, timer interrupt)
- Avoids deadlock
Stack

- A stack is made up of linked cells.
- Last cell of the stack always points to NULL.

Operations: lifo-init, lifo-push, lifo-pop
First attempt

• Why is this wrong?

**lifo-push** (lf: pointer to lifo, cl: pointer to cell)

A1: cl->next = lf->top # set the cell next pointer to top of the lifo
A2: lf->top = cl # set the top of the lifo to cell
Using atomic operations

**lifo-push** (lf: pointer to lifo, cl: pointer to cell)

B1: loop
B2: cl->next = lf->top # set the cell next pointer to top of the lifo
B3: if CAS (&lf->top, cl->next, cl) # try to set the top of the lifo to cell
B4: break
B5: endif
B6: endloop
Using atomic operations

lifo-pop (lf: pointer to lifo): pointer to cell
C1:    loop
C2:      head = lf->top            # get the top cell of the lifo
C3:      if head == NULL
C4:          return NULL          # LIFO is empty
C5:      endif
C6:      next = head->next        # get the next cell of cell
C7:      if CAS (&lf->top, head, next)  # try to set the top of the lifo to the next cell
C8:          break
C9:      endif
C10:     endloop
C11:    return head
The ABA problem

- What happens if the contents of memory appear to have not changed when in fact they have?

1) A → B → C → X → NULL

2) A → N → X → NULL

3) B → ?
Solution

- Keep a count of the cells popped from the stack:

  ```
  structure lifo {
    top: a pointer to a cell
    ocount: total count of pop operations
  }
  ```

- Two locations must be atomically updated:
  - the top of stack
  - the count of pops

2.2. The ABA problem

However, the above implementation of the LIFO pop operations doesn't catch the ABA problem. Assume that a process is preempted while dequeuing a cell after C6: several concurrent push and pop operations may result in a situation where the top cell remains unchanged but points to a different next cell as shown in figure 7.

![Figure 7: 1) state at the beginning of the pop operation, 2) state after preemption, 3) state after pop completion](image)

The LIFO change won't prevent the CAS operation to operate in C7, allowing to put a wrong cell on top of the stack. The solution to the ABA problem consists in adding a count of the cells popped from the stack to the LIFO structure as shown in figure 8 and to make use of the CAS2 primitive.

```structure lifo {
  top: a pointer to a cell
  ocount: total count of pop operations
}
```

Figure 8: extended lifo structure

The push operation remains unchanged and the pop operation is now implemented as shown in figure 9: it checks both for lifo top and output count changes when trying to modify the lifo top.

```lifo-pop (lf: pointer to lifo): pointer to cell
SC1: loop
SC2: head = lf->top # get the top cell of the lifo
SC2: oc = lf->ocount # get the pop operations count
SC3: if head == NULL
SC4: return NULL # LIFO is empty
SC5: endif
SC6: next = head->next # get the next cell of cell
SC7: if CAS2 (&lf->top, head, oc, next, oc + 1) # try to change both the top of the lifo and pop count
SC8: break
SC9: endif
SC10: endloop
SC11: return head
```

Figure 9: lifo-pop catching the ABA problem

3. Lock-free FIFO stacks

The FIFO queue is implemented as a linked list of cells with head and tail pointers. Each pointer have an associated counter, ocount and icount, which maintains a unique modification count of operations on head and tail. The cell structure is the same as above (figure 1) and the fifo structure is shown in figure 10.

```structure fifo {
  head: a pointer to head cell
  ocount: total count of pop operations
  tail: a pointer to tail cell
  icount: total count of push operations
}
```

Figure 10: the fifo structure

As in Michael-Scott [4] and Valois [3], the FIFO always contains a dummy cell, only intended to maintain the consistency. An empty FIFO contains only this dummy cell which points to an end fifo marker unique to the system: a trivial solution consists in using the FIFO address itself as a unique marker. All along the operations, head always points to the dummy cell which is the first cell in the list and tail always points to the last or the second last cell in the list. The double-word compare-and-swap increments the modification counters to avoid the ABA problem.

The queue consistency is maintained by cooperative concurrency: when a process trying to enqueue a cell detects a pending enqueue operation (tail is not the last cell of the list), it first tries to complete the pending operation before enqueuing the cell. The dequeue operation also ensures that the tail pointer does not point to the dequeued cell and if necessary, tries to complete any pending enqueue operation. Figure 11 to 13 presents the commented pseudo-code for the fifo queue operations.
Extend Compare-and-Swap

- CAS2 (mem, old1, old2, new1, new2)
  - available on Pentium and x86 processors
  - 64-bit processors running in 32 bit applications

 lifo-pop (lf: pointer to lifo): pointer to cell

 SC1: loop
 SC2: head = lf->top # get the top cell of the lifo
 SC2: oc = lf->ocount # get the pop operations count
 SC3: if head == NULL # LIFO is empty
 SC4: return NULL
 SC5: endif
 SC6: next = head->next # get the next cell of cell
 SC7: if CAS2 (&lf->top, head, oc, next, oc + 1) # try to change both the top of the lifo and pop count
 SC8: break
 SC9: endif
 SC10: endloop
 SC11: return head
2.2. The ABA problem

However, the above implementation of the LIFO pop operations doesn't catch the ABA problem. Assume that a process is preempted while dequeuing a cell after C6: several concurrent push and pop operations may result in a situation where the top cell remains unchanged but points to a different next cell as shown in figure 7.

![Figure 7: 1) state at the beginning of the pop operation, 2) state after preemption, 3) state after pop completion](image)

The LIFO change won't prevent the CAS operation to operate in C7, allowing to put a wrong cell on top of the stack. The solution to the ABA problem consists in adding a count of the cells popped from the stack to the LIFO structure as shown in figure 8 and to make use of the CAS2 primitive.

```c
structure lifo {
    head: a pointer to head cell
    ocount: total count of pop operations
    tail: a pointer to tail cell
    icount: total count of push operations
}
```

Figure 8: extended lifo structure

The push operation remains unchanged and the pop operation is now implemented as shown in figure 9: it checks both for lifo top and output count changes when trying to modify the lifo top.

```c
lifo-pop(lf: pointer to lifo): pointer to cell
SC1:
    loop
    SC2:
        head = lf->top # get the top cell of the lifo
        SC2: oc = lf->ocount # get the pop operations count
        SC3:
            if head == NULL
                SC4: return NULL # LIFO is empty
            endif
            SC5:
            SC6:
                next = head->next # get the next cell of cell
                SC7:
                    if CAS2(&lf->top, head, oc, next, oc + 1) # try to change both the top of the lifo and pop count
                        SC8:
                            break
                        SC9:
                    endif
                    SC10:
            endloop
            SC11:
            return head
```

Figure 9: lifo-pop catching the ABA problem

3. Lock-free FIFO stacks

The FIFO queue is implemented as a linked list of cells with head and tail pointers. Each pointer have an associated counter, ocount and icount, which maintains a unique modification count of operations on head and tail. The cell structure is the same as above (figure 1) and the fifo structure is shown in figure 10.

```c
structure fifo {
    head: a pointer to head cell
    ocount: total count of pop operations
    tail: a pointer to tail cell
    icount: total count of push operations
}
```

Figure 10: the fifo structure

As in Michael-Scott [4] and Valois [3], the FIFO always contains a dummy cell, only intended to maintain the consistency. An empty FIFO contains only this dummy cell which points to an end fifo marker. All along the operations, head always points to the dummy cell which is the first cell in the list and tail always points to the last or the second last cell in the list. The double-word compare-and-swap increments the modification counters to avoid the ABA problem.

The queue consistency is maintained by cooperative concurrency: when a process trying to enqueue a cell detects a pending enqueue operation (tail is not the last cell of the list), it first tries to complete the pending operation before enqueuing the cell. The dequeue operation also ensures that the tail pointer does not point to the dequeued cell and if necessary, tries to complete any pending enqueue operation. Figure 11 to 13 presents the commented pseudo-code for the fifo queue operations.
Lock-free queues

Diagram showing a lock-free queue with a tail, head, dummy, and two elements.
Issues

• More complicated that a stack
  - Need fast access to two variables, the head and the tail
  - Two pointers refer to the tail:
    • the next to last element, and the tail pointer
    • successful insertion requires updates to both these pointers
  - How can both updates occur atomically?
    • separate CAS operations on both pointers
      - what if one succeeds and the other fails?
      - even if both succeed, another thread might access the queue between the first and second updates
Invariants

- Queue must always be in a consistent state, even in the middle of a multi-step update.
- If A is in the middle of an update when thread B arrives, B should be able to identify this situation.
  - B waits.
- What happens if a thread fails?
  - If B finds that A is in the middle of an update, rather than wait for A to finish, it can do A’s work for it.
Insertion

- Involves updating two pointers
  - First links the new node to the end of the list by updating the next pointer of the current last element
  - Second swings the tail pointer to point to the new last element
Observation

• If queue is in a quiescent state, next field of the link node pointed to by tail is null.
• If queue is an intermediate state, next field is non-null.
• Can transition from intermediate to quiescent state by advancing tail to point to the next node, even if some other thread is in the middle of doing the operation
Lock-free Queues: Summary

- Maintain consistency using cooperative concurrency:
  - when a thread trying to enqueue detects another thread is already in the middle of trying to perform an enqueue (tail is not the end of the list), it first tries to complete the pending operation.
  - dequeue operation also ensures that the tail pointer does not point to the dequeued cell.
Lock-free Queues

**fifo-init** (ff: pointer to fifo, dummy: pointer to dummy cell)
- dummy->next = NULL  # makes the cell the only cell in the list
- ff->head = ff->tail = dummy  # both head and tail point to the dummy cell

**fifo-push** (ff: pointer to fifo, cl: pointer to cell)
1. cl->next = ENDFIFO(ff)  # set the cell next pointer to end marker
2. loop  # try until enqueue is done
3. icount = ff->icount  # read the tail modification count
4. tail = ff->tail  # read the tail cell
5. if CAS (&tail->next, ENDFIFO(ff), cl)  # try to link the cell to the tail cell
6. break;  # enqueue is done, exit the loop
7. else  # tail was not pointing to the last cell, try to set tail to the next cell
8. CAS2 (&ff->tail, tail, icount, tail->next, icount+1)
9. endif
10. endloop
11. CAS2 (&ff->tail, tail, icount, cl, icount+1)  # enqueue is done, try to set tail to the enqueued cell

CS390C: Principles of Concurrency and Parallelism
Lock-free Queues

**fifo-pop** (ff: pointer to fifo): pointer to cell

D1: loop # try until dequeue is done
D2: ocount = ff->ocount # read the head modification count
D3: icount = ff->icount # read the tail modification count
D4: head = ff->head # read the head cell
D5: next = head->next # read the next cell
D6: if ocount == ff->oc # ensures that next is a valid pointer
   # to avoid failure when reading next value
D7: if head == ff->tail # is queue empty or tail falling behind ?
D8: if next == ENDFIFO(ff) # is queue empty ?
D9: return NULL # queue is empty: return NULL
D10: endif # tail is pointing to head in a non empty queue, try to set tail to the next cell
D11: CAS2 (&ff->tail, head, icount, next, icount+1)
D12: else if next <> ENDFIFO(ff) # if we are not competing on the dummy next
D13: value = next->value # read the next cell value
D14: if CAS2 (&ff->head, head, ocount, next, ocount+1) # try to set head to the next cell
D15: break # dequeue done, exit the loop
D16: endif
D17: endif
D18: endloop
D19: head->value = value # set the head value to previously read value
D20: return head # dequeue succeed, return head cell

4 Correctness of the FIFO operations

Traditional sequential programs may be viewed as functions from inputs to outputs which may be specified as a pair consisting of a precondition describing the allowed inputs and postcondition describing the desired results for these inputs. However for concurrent programs, this approach is too limited and numerous work has been done for formal verification of concurrent systems. Although informal, two properties introduced by Lamport [11] are required for correctness of concurrent programs:

- **safety property**: states that "something bad never happens",
- **liveness property**: states that "something good eventually happens".

Formalizing this classification has been a main motivation for much of the work done on specification and verification of concurrent systems [12]. Formal methods successfully applied to sequential programs have also been extended to consider concurrent programming: Herlihy proposed a correctness condition for concurrent objects called "Linearizability" [13, 14]. It states that a concurrent computation is linearizable if it is equivalent to a legal sequential computation. An object (viewed as the aggregate of a type, which defines a set of possible values, and a set of primitive operations), is linearizable if each operation appears to take effect instantaneously at some point between the operation's invocation and response. It implies that processes appear to be interleaved at the granularity of complete operations and that the order of non-overlapping operations is preserved.

Correctness of the FIFO operations formal proof is beyond the scope of this paper, however it will be examined according to the properties mentionned above.
Properties

• Safety: nothing bad ever happens
  - linked list is always connected
  - cells are only inserted after the last cell in the linked list
  - cells are only deleted from the beginning of the list
  - head always points to the first node in the list
  - tail always points to a node in the list
Properties

• Liveness: something good eventually happens
  
  supposing a thread is trying to enqueue a cell.
  
  • failure means the thread is looping through E8
  • But, then the other thread would have succeeded in completing an enqueue operation, or dequeuing the tail cell.

  supposing a thread is trying to dequeue a cell.
  
  • failure means the process is looping through D11 or D14
  • Failure in D11 means that another thread must have succeeded in completing an enqueue operation, or in dequeuing the tail cell.
  • Failure in D14 means another thread must have succeeded in completing a dequeue operation.