Example Concurrent Program
(x is shared, initially 0)

- code for Thread 0

  foo( )

  x := x+1

- code for Thread 1

  bar( )

  x := x+2

Assume both threads execute at about the same time.

What's the output?

Example Concurrent Program (cont.)

- One possible execution order is:
  - Thread 0: R1 := x (R1 == 0)
  - Thread 1: R2 := x (R2 == 0)
  - Thread 1: R2 := R2 + 2 (R2 == 2)
  - Thread 1: x := R2 (x == 2)
  - Thread 0: R1 := R1 + 1 (R1 == 1)
  - Thread 0: x := R1 (x == 1)
- Final value of x is 1 (!!)
- Question: what if Thread 1 also uses R1?

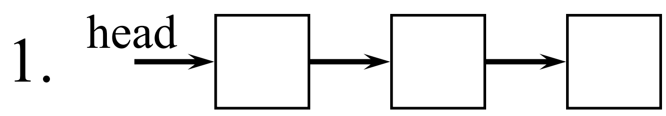
More Concurrent Programming: Linked Lists (head is shared)

```
Insert(head, elem) {
    elem-> next := head;
    head := elem;
}
```

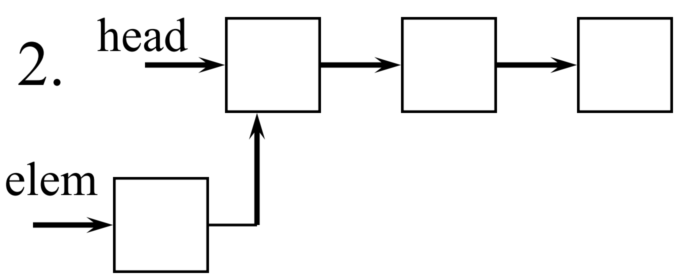
```
Void *Delete(head) {
    Void *t;
    t:= head;
    head := head->next;
    return t;
}
```

(Assume one thread calls Insert and one calls Delete)

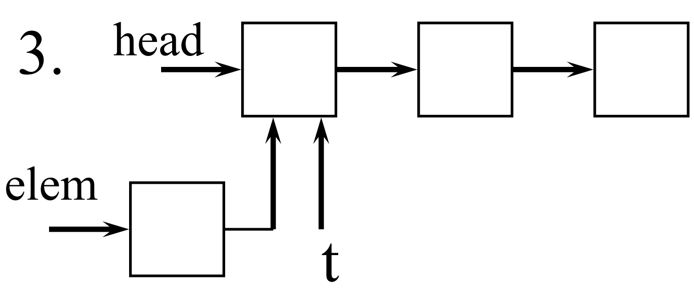
Example Execution



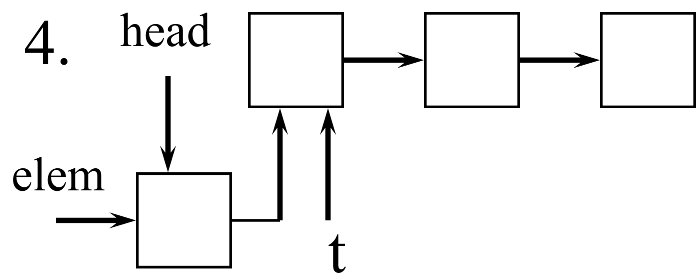
Insert: elem->next := head;



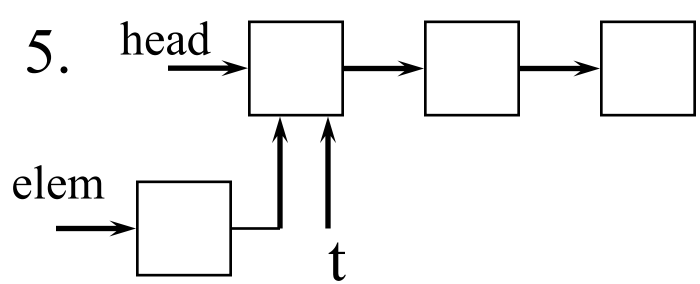
Delete: t := head;



Insert: head := elem;



Delete: head := head->next;



Delete: return t;

Some Definitions

- Race condition
  - when output depends on ordering of thread execution
  - more formally:
    - (1) two or more threads access a shared variable with no synchronization (*or incorrect synchronization*), and
    - (2) at least one of the threads writes to the variable

More Definitions

- Atomic Operation
  - an operation that, once started, runs to completion
    - **note: more precisely, logically runs to completion**
  - indivisible
  - in this class: loads and stores
    - meaning: if thread A stores "1" into variable x and thread B stores "2" into variable x about about the same time, result is either "1" or "2"

Critical Section

- section of code that:
  - must be executed by one thread at a time
  - if more than one thread executes at a time, have a race condition
  - ex: linked list from before
    - Insert/Delete code forms a critical section
    - What about just the Insert *or* Delete code?
      - is that enough, or do both procedures belong in a single critical section?

Critical Section (CS) Problem

- Provide entry and exit routines:
  - all threads must call entry before executing CS
  - all threads must call exit after executing CS
  - thread must not leave entry routine until it's safe
- CS solution properties
  - *Mutual exclusion*: at most one thread is executing CS
  - *Absence of deadlock*: two or more threads trying to get into CS, and no threads in => at least one succeeds
  - *Absence of unneccessary delay*: if only one thread trying to get into CS, and no thread is in, it succeeds
  - *Eventual entry*: thread eventually gets into CS

Structure of threads for Critical Section problem

Threads do the following:

```
while (1) {
    do other stuff (non-critical section)
    call enter
    execute CS
    call exit
    do other stuff (non-critical section)
}
```

Critical Section Assumptions

- Threads must call enter and exit
- Threads must not die or quit inside a critical section
- Threads **can** be context switched inside a critical section
  - this does **not** mean that the newly running thread may enter the critical section

Critical Section Solution Attempt #1 (2 thread version, with id's 0 and 1)

```
Initially, turn == 0  /* turn is shared */
entry(id)  {  /* note id local to each thread */
    while (turn != id) ;  /* if not my turn, spin */
}
exit(id) {
    turn := 1-id;  /* other thread's turn */
}
```

Critical Section Solution Attempt #2 (2 thread version, with id's 0 and 1)

Initially, flag[0] = flag[1] = false

/* flag is a shared array */

```
entry(id)  {
    flag[id] := true;  /* I want to go in */
    while (flag[1-id]) ; /*proceed if other not trying*/
}
exit(id) {
    flag[id] := false;  /* I'm out */
}
```

Critical Section Solution Attempt #3 (2 thread version, with id's 0 and 1)

Initially, flag[0] == flag[1] == false, turn == 0

/* flag and turn are shared variables */

```
entry(id)  {
    flag[id] := true;  /* I want to go in */
    turn := 1-id;  /* in case other thread wants in */
    while (flag[1-id] and turn == 1-id) ;
}
exit(id) {
    flag[id] := false;  /* I'm out */
}
```

Satisfying the 4 properties

- Mutual exclusion
  - turn must be 0 or 1 => only one thread can be in CS
- Absence of deadlock
  - turn must be 0 or 1 => one thread will be allowed in
- Absence of unnecessary delay
  - only one thread trying to get into CS => flag[other] is false => will get in
- Eventual Entry
  - spinning thread will not modify turn
  - thread trying to go back in will set turn equal to spinning thread

Hardware Support

- Provide instruction that is:
  - atomic
  - fairly easy for hardware designer to implement
- Read/Modify/Write
  - atomically read value from memory, modify it in some way, write it back to memory
- Use to develop simpler critical section solution for any number of threads

Test-and-Set

Many machines have it

```
function TS(ref target: bool) returns bool
   bool b := target;  /* return old value */
   target := true;
   return b;
```

Executes atomically

CS solution with Test-and-Set

```
Initially, s == false  /* s is a shared variable */
entry( )  {
    bool spin;  /* spin is local to each thread! */
    spin := TS(s);
    while (spin)
      spin := TS(s);
}
exit( ) {
    s := false;
}
```

```
Function TS(ref target: bool) returns bool
  bool b := target
  target := true
  return b
```

Partial List of Atomic Instructions

- Compare and Swap (x86)
- Load linked and conditional store (RISC)
- Fetch and Add (Ultracomputer)
- Atomic Swap
- Atomic Increment

Basic Idea with Atomic Instructions

- Each thread has a local flag
- One variable shared by all threads
- Use the atomic instruction with flag, shared variable
  - on a change, allow thread to go in
  - other threads will not see this change
- When done with CS, set shared variable back to initial state

Problems with busy-waiting CS solution

- Complicated
- Inefficient
  - consumes CPU cycles while spinning
- Priority inversion problem
  - low priority thread in CS, high priority thread spinning can end up causing deadlock
  - example: Mars Pathfinder problem

May want to block when waiting for CS

Locks

- Two operations:
  - Acquire (get it, if can’t go to sleep)
  - Release (give it up, possibly wake up a waiter)
- Acquire and Release are atomic
- A thread can only release a previously acquired lock
- entry( ) is then just Acquire(lock)
- exit( ) is just Release(lock)

Lock is shared among all threads

First Attempt at Blocking Lock Implementation

- Acquire(lock) disables interrupts
- Release(lock) enables interrupts
- Advantages:
  - is a blocking solution; can be used inside OS in some situations
- Disadvantages:
  - CS can be in user code [could infinite loop], might need to access disk in middle of CS, system clock could be skewed, etc.

Correct Blocking Lock Implementation

lock class has queue, value

Initially:

- queue is empty
- value is free

```
Aquire(lock)
    Disable interrupts
    if (lock.value == busy)
      enQ(lock.queue,thread)
      go to sleep
    else
      lock.value := busy
    Enable interrupts
```

```
Release(lock)
    Disable interrupts
    if notEmpty(lock.queue)
      thread := deQ(lock.queue)
      enQ(readyList, thread)
    else
      lock.value := free
    Enable interrupts
```

Can interrupts be enabled before sleep?

lock class has queue, value

Initially:

queue is empty

value is free

```
Aquire(lock)
    Disable interrupts
    if (lock.value == busy)
        Enable interrupts
        enQ(lock.queue,thread)
        go to sleep
    else
        lock.value := busy
    Enable interrupts
```

```
Release(lock)
    Disable interrupts
    if notEmpty(lock.queue)
        thread := deQ(lock.queue)
        enQ(readyList, thread)
    else
        lock.value := free
    Enable interrupts
```

Can interrupts be enabled before sleep?

lock class has queue, value

Initially:

queue is empty

value is free

```
Aquire(lock)
    Disable interrupts
    if (lock.value == busy)
        enQ(lock.queue,thread)
        Enable interrupts
        go to sleep
    else
        lock.value := busy
    Enable interrupts
```

```
Release(lock)
    Disable interrupts
    if notEmpty(lock.queue)
        thread := deQ(lock.queue)
        enQ(readyList, thread)
    else
        lock.value := free
    Enable interrupts
```

What about a "spin-lock"?

Need to fix all items in red

lock class has queue, value

Initially:

  queue is empty

  value is free

```
Aquire(lock)
    Disable interrupts
    if (lock.value == busy)
      enQ(lock.queue,thread)
      go to sleep
    else
      lock.value := busy
    Enable interrupts
```

```
Release(lock)
    Disable interrupts
    if notEmpty(lock.queue)
      thread := deQ(lock.queue)
      enQ(readyList, thread)
    else
      lock.value := free
    Enable interrupts
```

Spin Lock Implementation (should look familiar)

```
Initially, s == false  /* s is a shared variable */
Acquire(lock)  {
    bool spin;  /* spin is local to each thread! */
    spin := TS(s);
    while (spin)
        spin := TS(s);
}
Release (lock) {
    s := false;
}
```

Many more algorithms in MCS paper!

Problems with Locks

- Not general
  - only solve simple critical section problem
  - can’t do any more general synchronization
  - often we want to enforce strict orderings between threads
- Condition synchronization
  - need to wait until some condition is true
  - example: bounded buffer (next slide)
  - example: thread join

Bounded Buffer Problem

- Consider 2 threads:
  - one producer, one consmer
  - real OS example: ps | grep dkl
    - shell forks a thread for “ps” and a thread for “grep dkl”
  - “ps” writes its output into a fixed size buffer; “grep” reads the buffer
  - access to a specific buffer slot a critical section, but:
  - between buffer slots, not a critical section
    - also may need to wait for buffer to be empty or full

Bounded Buffer Cont.

- Have the following:
  - buffer of size n (i. e., char buffer[n])
  - one producer thread
  - one consumer thread
- Locks are hard to use here
  - example: producer grabs lock, but must release it if buffer is full
  - example: producer and consumer access distinct locations -- can be concurrent!
- Need something more general

Semaphores (Dijkstra)

- Semaphore is an object
  - contains a (private) value and 2 operations
- **Semaphore value must be nonnegative**
- P operation (atomic):
  - if value is 0, block; else decrement value by 1
- V operation (atomic):
  - if thread blocked, wake up; else value++
- Semaphores are "resource counters"

Critical Sections with Semaphores

sem mutex := 1

entry( )

- P(mutex)

exit( )

- V(mutex)

- Semaphores are more powerful than locks
- For mutual exclusion, initialize semaphore to 1

Bounded Buffer
(1 producer, 1 consumer)

char buf[n], int front := 0, rear := 0

sem empty := n, full := 0

```
Producer( )
    do forever...
      produce message m
      P(empty)
      buf[rear] := m;
      rear := rear “+” 1
      V(full)
```

```
Consumer()
    do forever...
      P(full)
      m := buf[front]
      front := front “+” 1
      V(empty)
      consume m
```

Bounded Buffer (multiple producers and consumers)

char buf[n], int front := 0, rear := 0

sem empty := n, full := 0, mutexC := 1, mutexP := 1

Producer( )

```
do forever...
  produce message m
  P(empty); P(mutexP)
  buf[rear] := m;
  rear := rear "+" 1
  V(mutexP); V(full)
```

Consumer()

```
do forever...
  P(full); P(mutexC)
  m := buf[front]
  front := front "+" 1
  V(mutexC); V(empty)
  consume m
```

Readers/Writers

- Given a database
  - can have multiple “readers” at a time
    - don’t ever modify database
  - can only have one “writer” at a time
    - will modify database
    - readers not allowed in while writer is
- Problem has many variations

Idea of Readers/Writers Solution

- Need mutual exclusion in both entry and exit
  - use mutex semaphore, initialized to one
- Keep state of database, enforce constraints
  - number of delayed readers and writers
  - number of readers and writers in database
  - Ex: better not have nr, nw simultaneously > 0
- One semaphore blocks readers, different semaphore blocks writers
- Readers going in can let other readers go in