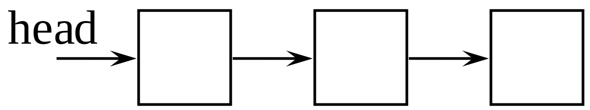
Critical Sections: Implementing Logical Atomicity

- Critical section of code is one that:
  - must be executed by one thread at a time
    - otherwise, there is a race condition
  - Example: 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?

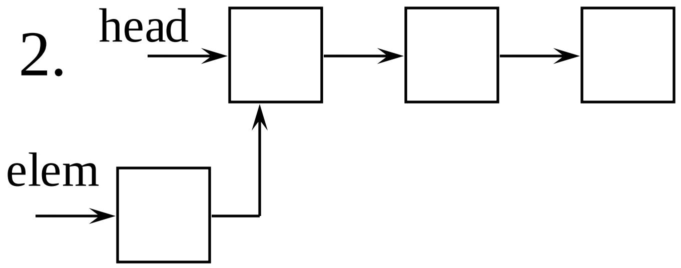
List Example

1.



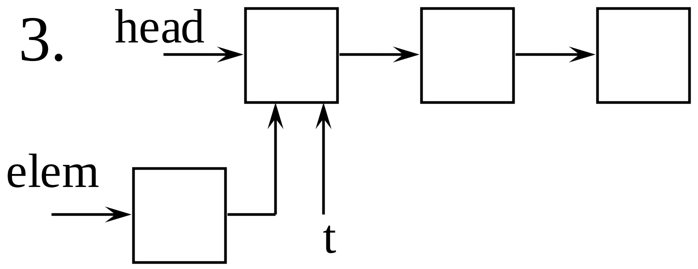
Insert: elem→next := head;

2.



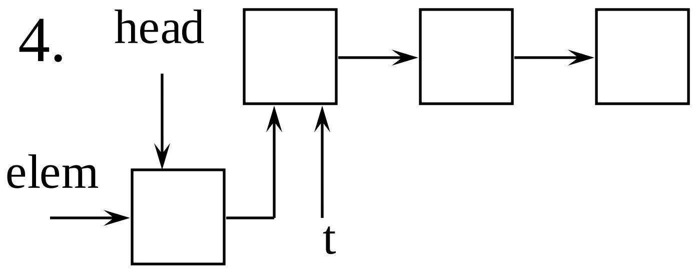
Remove: t := head;

3.



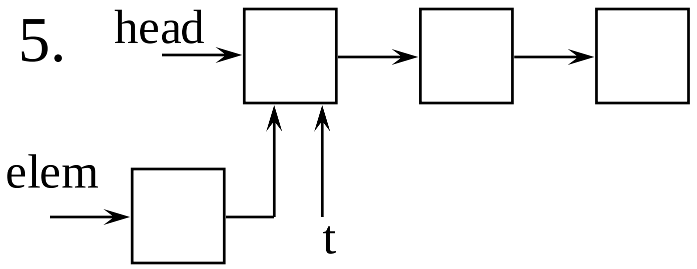
Insert: head := elem;

4.



Remove: head := head→next;

5.



Remove: return t;

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
  - a thread must not leave entry routine until it’s safe
- CS solution properties (red: safety; black: liveness)
  - Mutual exclusion: at most one thread is executing CS
  - Absence of deadlock: two or more threads trying to get into CS, and no thread is in → at least one succeeds
  - Absence of unnecessary delay: if only one thread tries to get into CS, and no thread is in CS, 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” (i.e., non-critical section)
    call CS entry routine
    execute CS
    call CS exit routine
}
```

Critical Section Assumptions

- Threads must call CS entry and exit routines
- Threads must not die or quit inside a critical section (or the entry and exit routines)
- Threads **can** be context switched inside a critical section
  - this does **not** mean that another thread may enter the critical section

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

```
turn = 0
entry(id)  {
    while (turn != id) ;
}
exit(id) {
    turn := 1-id;
}
```

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

```
flag[0] = flag[1] = false
entry(id)  {
    flag[id] := true;
    while (flag[1-id]) ;
}
exit(id) {
    flag[id] := false;
}
```

Critical Section Solution Attempt #3

```
flag[0] = flag[1] = false, turn == 0
entry(id)  {
    flag[id] := true;
    turn := 1-id;
    while (flag[1-id] and turn == 1-id) ;
}
exit(id) {
    flag[id] := false;
}
```

Summary: Satisfying the 4 properties (on Attempt #3)

- 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 => the thread will get into the CS
- Eventual Entry
  - spinning thread will not modify turn
  - thread trying to go back in will set turn equal to spinning thread’s id

Next three slides are just for reference (include comments)

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

```
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)

```
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)

```
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 */
}
```

Hardware Support for Critical Sections

- All modern machines provide atomic instructions
- All of these instructions are *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**

Example: Test-and-Set

Some 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

```
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(bool &target) returns bool
  bool b := target
  target := true
  return b
```

A few example atomic instructions

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

Basic Idea with Atomic Instructions

- One shared variable plus per-thread flag
- Use atomic instruction with flag, shared variable
  - On a change, allow thread to go in
  - Other threads will not see this change
  - Should use “test and test and set” style (see next slide)
- When done with CS, set shared variable back to initial state
  - Some advanced solutions spin on exit (not discussed in 422)

CS using Test-and-Test-and-Set

```
Initially, s == false
entry( )  {
    bool spin;
    spin := TS(s);
    while (spin) {
        while (s) ;      ← This line is the difference from "regular" Test-and-Set
        spin := TS(s);
    }
}
exit( )
    s := false;
```

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

The (Fair) Ticket Algorithm-- High Level Solution

```
Init: number = next = 0
Entry: <ticket = number; number++>
            <await (ticket == next)>
Exit: <next++>
```

Fair Ticket Algorithm--Implementation

```
Function FA(ref int target, const int C) returns int
   int loc := target
   target := target + C
   return loc
```

```
number = next = 0
entry( )  {
     int ticket = FetchAndAdd(number, 1)
     while (ticket != next) ;        ← No atomicity needed---why?
}
exit( )
     next = next + 1                 ← No atomicity needed---why?
```

Works for any number of threads---and is fair (provides eventual entry)

N-Thread CS Solutions *Without* Atomic Instructions

- Much more complex
  - We will not address these, and you will not be responsible for them
- Most common solution is called the bakery algorithm
  - Essentially an implementation of the ticket algorithm, but without atomic instructions (other than load and store, of course)
  - If interested, see: https://en.wikipedia.org/wiki/Lamport%27s_bakery_algorithm

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

In some cases, want to block when waiting for CS

Locks

- Two operations:
  - Acquire (get it, if can't go to sleep or spin)
  - Release (give it up, possibly wake up a waiter or spinning thread)
- Acquire and Release are (logically) 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

Using Locks for Critical Sections (CSs)

Assume L1 and L2 are locks; x and y are shared variables

- Single CS, one location
  - Acquire(L1)
  - CS (e.g., $x += k$)
  - Release(L1)
- Single CS, two locations
  - Acquire(L1)
  - CS (e.g., $x += k_1$)
  - Release(L1)
  - …
  - Acquire(L1)
  - CS (e.g., $x += k_2$)
  - Release(L1)

- Two different CSs
  - Acquire(L1)
  - CS 1 (e.g., $x += k_1$)
  - Release(L1)
  - …
  - Acquire(L2)
  - CS 2 (e.g., $y += k_2$)
  - Release(L2)

Incorrect Blocking Lock Implementation in an OS kernel (single-core machine)

- 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 more skewed than typical, etc.

Correct Blocking Lock Implementation in an OS kernel (single-core machine)

lock class has queue, value

Initially:

    queue is empty
    value is free

```
Acquire(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

```
Acquire(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

```
Acquire(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”?

Items in red must be addressed

lock class has queue, value

Initially:

    queue is empty
    value is free

```
Acquire(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
```

Fair Ticket *Lock*, Revisited

```
number = next = 0
Acquire( )  {
    int ticket = FetchAndAdd(number, 1)
    while (ticket != next) ;
}
Release( ) {
    next = next + 1
}
```

This is known as a “spin-lock”.

What about *blocking*, OS-level implementation on multiple cores?

- More complicated
  - Cannot use disabling/enabling interrupts
- Can be solved by using atomic instructions and spin locks together
  - If interested, consult the internet and/or AI tools

What about *blocking*, **user-level** implementation on multiple cores?

- Typically solved with atomic instructions and futex (**F**ast **U**serspace mu**TEX**) support
  - Atomic instruction used for a thread wanting to acquire the lock
  - Futex provides wait and wake operations that use the OS to block the thread if lock is held by another thread
    - All the complication is pushed into the OS
- Again, consult the internet and/or AI tools for details

Blocking Locks vs Spin Locks

- Blocking Locks
  - Advantages: do not consume CPU cycles when spinning
  - Disadvantage: context switches to block and unblock
- Spin Locks
  - Advantages: faster when multiple cores and no competing jobs
  - Disadvantage: consumes CPU cycles when spinning; priority inversion possible

Problem with Locks

- Not general
  - Locks only solve critical section problem
  - Locks do not allow any more general synchronization
    - Specifically, locks cannot enforce strict orderings between threads

Condition synchronization

- Need to wait until some condition is true
- Example: thread join
- Example: bounded buffer (later)

Barriers

- Points in program at which all threads must arrive before any can proceed
- Barriers provide *sequence control*
- In CSC 422, we will assume that all threads must participate in a barrier
  – It is possible to have “local” barriers, in which a subset of threads participate
    - In the message passing paradigm, we call these subcommunicators (will not discuss in this class)

Challenges With Barriers

- Must implement barriers
- Must insert barriers into programs
  - Want as few barriers as possible
    - Barriers are overhead
  - However, must *ensure* correct program behavior

Centralized Barrier

Shared variable: count = 0; assume *n* threads

High-level code for barrier for a given thread:

```
<count++>
<await (count == n) ;>
```

Actual implementation for barrier for a given thread:

```
FetchAndAdd(count, 1)
while (count != n) ;
```

Problem?

Centralized Barrier

Shared variable: count = 0; assume *n* threads

High-level code for barrier for a given thread:

```
    <count++>
    <await (count == n) ;>
```

Actual implementation for barrier for a given thread:

```
    FetchAndAdd(count, 1)
    while (count != n) ;
```

Problem: reset

Centralized Barrier, Suggested by Past Students

Shared variable: count = 0

Code for barrier for a given thread:

```
FetchAndAdd(count, 1)
while (count mod n != 0) ;
```

Problem?

Centralized Barrier, Suggested by Past Students

Shared variable: count = 0

Code for barrier for a given thread:

```
FetchAndAdd(count, 1)
while (count mod n != 0) ;
```

Problem: one or more threads might not see barrier exit condition (count mod n == 0)

Centralized Barrier, Suggested by Past Students, Version 2.0

Shared variable: count = 0

Code for barrier for a given thread:

```
if (FetchAndAdd(count, 1) == n-1)
   count = 0
else
   while (count != 0) ;
```

FetchAndAdd returns the old value

Problem?

Centralized Barrier, Suggested by Past Students, Version 2.0

Shared variable: count = 0

Code for barrier for a given thread:

```
if (FetchAndAdd(count, 1) == n-1)
    count = 0
else
    while (count != 0) ;
```

FetchAndAdd returns the old value

Problem: same as previous attempt: a thread may not see the barrier exit condition

Centralized Barrier, Suggested by Current Students

Shared variable: count = 0

```
Barrier(ref int k) {      // separate k for each thread
    FetchAndAdd(count, 1)
    while (count < k * n) ;
    k = k + 1
}
```

Thread invokes via Barrier(barrierNum), where barrierNum is kept by each thread and initially 0

This works, but has disadvantage that each thread must keep track of barrier state

Centralized Barrier (handles reset)

Shared variables: countEven = countOdd = nB = 0

Code for barrier for a given thread (assume *n* threads):

```
if  (nB mod 2 == 0) {
    if  (FetchAndAdd(countEven, 1) == n-1) {
        nB = nB + 1
        countEven = 0
    }
    else
      while (countEven != 0) ;
else {
     // same code, but with countOdd instead of countEven
}
```

Barrier from centralized version 2.0

FetchAndAdd returns the old value

Centralized Barrier (handles reset) Full code

Shared variables: countEven = countOdd = nB = 0

Code for barrier for a given thread (assume *n* threads):

```
if  (nB mod 2 == 0) {
  if  (FetchAndAdd(countEven, 1) == n-1) {
      nB = nB + 1
      countEven = 0
  }
  else
     while (countEven != 0) ;
else {
   if  (FetchAndAdd(countOdd, 1) == n-1) {
      nB = nB + 1
      countOdd = 0
  }
  else
     while (countOdd != 0) ;
}
```

FetchAndAdd returns the old value

Centralized Barrier (handles reset) Sense-reversing version

Shared variables: count = 0, globalSense = false

Code for barrier for a given thread (assume *n* threads):

```
bool localSense = not globalSense
if (FetchAndAdd(count, 1) == n-1) {
    count = 0
    globalSense = localSense
}
else
  while (globalSense != localSense) ;
```

FetchAndAdd returns the old value

Centralized Barrier (with reset) works, but…

- May not perform well
  - If we are implementing a barrier, we should assume it will be used frequently and possibly with a large number of threads
  - Spinning on a shared flag (*count*) is not ideal on some multicore architectures
    - Typically ones with higher core counts
  - We want to investigate barriers where each thread spins on a *different* variable

Symmetric Barrier, 2 threads (not quite correct)

```
int arrive[2] = {0,0}
```

Thread 0’s code

```
arrive[0] = 1
while (arrive[1] != 1)
   ;
arrive[1] = 0
```

Thread 1’s code

```
arrive[1] = 1
while (arrive[0] != 1)
   ;
arrive[0] = 0
```

Flag synchronization principle (1): a thread resets the flag on which it spins.

Symmetric Barrier, 2 threads (not quite correct)

```
              int arrive[2] = {0,0}

Thread 0's code                  Thread 1's code


arrive[0] = 1                    arrive[1] = 1
while (arrive[1] != 1)           while (arrive[0] != 1)
   ;                  Problem!      ;
arrive[1] = 0                    arrive[0] = 0
```

Flag synchronization principle (1): a thread resets the flag on which it spins.

Flag synchronization principle (2): a thread should not set a flag until it's known to be clear.

Symmetric Barrier, 2 threads (correct)

```
int arrive[2] = {0,0}
```

Thread 0's code

```
while (arrive[0] != 0)
   ;
arrive[0] = 1
while (arrive[1] != 1)
   ;
arrive[1] = 0
```

Thread 1's code

Implements flag synchronization principle (2)

```
while (arrive[1] != 0)
   ;
arrive[1] = 1
while (arrive[0] != 1)
   ;
arrive[0] = 0
```

Symmetric Barrier, 2^p threads

- Conceptually, just glue multiple two-thread barriers together

Symmetric Barrier, 2^p threads

- Conceptually, just glue multiple two-thread barriers together
  - Problem: flags meant for one thread might be seen by another thread
  - Two solutions to this:
    - use more storage (e.g., a two-dimensional array of arrive flags, with the first index as the round), or
    - set the arrive array values to the round id (simpler and more space efficient)

Dissemination Barrier

(Hensgen, Finkel, and Manber)

- Similar to symmetric barrier in that it has a logarithmic number of steps (in the number of participating threads)
- Different from symmetric barrier in that at each round, a thread will signal one thread and wait for a *different* thread's signal

Dissemination Barrier

int arrive[0:P-1] = {0, 0, …, 0}   in C, need *volatile*

Thread i's code (local vars: *j* and *waitFor*):

```
for j = 1 to ceiling($\log_2(P)$) {
    while (arrive[i] != 0)  ;
    arrive[i] = j
    waitFor = (i + $2^{j-1}$) mod P
    while (arrive[waitFor] != j)  ;
    arrive[waitFor] = 0
}
```