M. PUESCHEL, T. BEN-NUN

Lecture 5: Languages and Locks
GPUs: Recap

- Massively parallel
  - 84x256 ALUs

- Composed of SMs
  - Separate L1/scratch-pad caches

- CUDA programming model
  - Create as many threads as there are data

- Execution follows SIMT
  - 32-thread lock-step

```c
__global__ void MatMulKernel(Matrix A, Matrix B, Matrix C) {
    // Each thread computes one element of C
    // by accumulating results into Cvalue
    float Cvalue = 0;
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    for (int e = 0; e < A.width; ++e)
        Cvalue += A.data[row * A.width + e] * B.data[e * B.width + col];
    C.data[row * C.width + col] = Cvalue;
}
```

```c
if (threadIdx.x < 4) {
    A;
    B;
} else {
    X;
    Y;
}
Z;
```
High-Level Languages: OpenMP and OpenACC

- Directive-based programming

- Implicitly wrap for loops
  - Advantages:
    - *Code can run without #pragmas*
    - *Loop construct clearer than kernel*

- Portable across platforms
  - ...but has to be tuned for each platform separately

- Directives nest into each other for increased control
  - Disadvantage: It’s possible to have more #pragmas than code

Source: Gayatri and Yang, Optimizing Large Reductions in BerkeleyGW on GPUs Using OpenMP and OpenACC
Scratch-pad (shared) memory
Matrix Multiplication with Shared Memory

```c
__global__ void MatMulKernel(Matrix A, Matrix B, Matrix C) {
    int blockRow = blockIdx.y; int blockCol = blockIdx.x;
    Matrix Csub = GetSubMatrix(C, blockRow, blockCol);
    float Cvalue = 0;
    int row = threadIdx.y; int col = threadIdx.x;

    for (int m = 0; m < (A.width / BLOCK_SIZE); ++m) {
        Matrix Asub = GetSubMatrix(A, blockRow, m);
        Matrix Bsub = GetSubMatrix(B, m, blockCol);
        __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
        __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];
        As[row][col] = Asub.data[row][col];
        Bs[row][col] = Bsub.data[row][col];

        __syncthreads();

        for (int e = 0; e < BLOCK_SIZE; ++e)
            Cvalue += As[row][e] * Bs[e][col];

        __syncthreads();
    }
    Csub.data[row][col] = Cvalue;
}
```

Figures and code courtesy of nvidia.com
GPU Synchronization

- Across warps – No need

- Across thread-blocks – __syncthreads()

- Across grid
  - Simplest solution: Queue another kernel
  - Less simple: Grid barriers

- Across GPU (multiple streams), multiple GPUs
  - Paired synchronization (event + stream-wait-event)

- Between host and GPU
  - cuda{Stream,Event,Device}Synchronize()
  - cudaMemcpy() – Not async (also not recommended).
Class Discussion

- How do we define a memory model for GPUs?
  - Things to remember: Different memory spaces, logical thread hierarchy, SMs, cache model

- Can GPUs guarantee SC? Should they?

- Open research question: Can we port the benefits of GPU fast thread-switching to CPUs?
Review of last lecture

- Memory models
  - Ordering between accesses to different variables
  - Sequential consistency – nice but unrealistic
    
    *Demonstrate how it prevents compiler and architectural optimizations*

- Practical memory models
  - Overview of various models (TSO, PSO, RMO, ... existing CPUs)
  - Case study of x86 (8 principles, TLO + CC)
  - Case study of NVIDIA GPUs (continuing today)
DPHPC Overview

DPHPC
- locality
- parallelism
  - vector ISA
  - shared memory
  - distributed memory

concepts & techniques
- caching
- memory hierarchy

Memory Models
- cache coherency
- distributed algorithms
- group communications
- locks
- lock free
- wait free
- linearizability

Models
- memory
- Amdahl's and Gustafson's law
- α - β
- PRAM
- LogP
- I/O complexity
- balance principles I
- Little's Law
- balance principles II
- scheduling
Goals of this lecture

- Recap: Correctness in parallel programs
  - Covered in PP, here a slimmed down version to make the DPHPC lecture self-contained
    
    *Watch for the green bar on the right side*

- Languages and Memory Models
  - Java/C++ definition
  - Races (now in practice)
  - Synchronization variables (now in practice)

- Mutual exclusion
  - Recap – simple lock properties
  - Proving correctness in SC and memory models (x86)
  - Locks in practice – performance overhead of memory models!
Notions of Correctness

We discussed so far:

- Read/write of the same location
  - *Cache coherence (write serialization and atomicity)*
- Read/write of multiple locations
  - *Memory models (visibility order of updates by cores)*

Now one level up: objects (variables/fields with invariants defined on them)

- Invariants “tie” variables together
- Sequential objects
- Concurrent objects
Sequential Objects

- Each object has a type

- A type is defined by a class
  - Set of fields forms the state of an object
  - Set of methods (or free functions) to manipulate the state

- Remark
  - An Interface is an abstract type that defines behavior
    
    A class implementing an interface defines several types
Running Example: FIFO Queue

- Insert elements at tail
- Remove elements from head
  - Initial: head = tail = 0
  - enq(x)
  - enq(y)
  - deq() [x]
  - ...

capacity = 8
Sequential Queue

class Queue {
private:
    int head, tail;
    std::vector<Item> items;

public:
    Queue(int capacity) {
        head = tail = 0;
        items.resize(capacity);
    }

    // ...
};
Sequential Queue

class Queue {
    // ...

public:
    void enq(Item x) {
        if((tail+1)%items.size() == head) {
            throw FullException;
        }
        items[tail] = x;
        tail = (tail+1)%items.size();
    }

    Item deq() {
        if(tail == head) {
            throw EmptyException;
        }
        Item item = items[head];
        head = (head+1)%items.size();
        return item;
    }
};
Sequential Execution

- (The) one process executes operations one at a time
  - Sequential 😊

- Semantics of operation defined by specification of the class
  - Preconditions and postconditions
    e.g., Hoare logic
Design by Contract!

- **Preconditions:**
  - Specify conditions that must hold before method executes
  - Involve state and arguments passed
  - Specify obligations a client must meet before calling a method

- **Example: enq()**
  - Queue must not be full!

```java
class Queue {
    // ...
    void enq(Item x) {
        assert((tail+1)%items.size() != head);
        // ...
    }
};
```
Design by Contract!

- **Postconditions:**
  - Specify conditions that must hold after method executed
  - Involve previous state, state, and arguments passed

- **Example: enq()**
  - Queue must contain element!

```java
class Queue {
    // ...
    void enq(Item x) {
        // ...
        assert (tail == (old_tail + 1)%items.size()) && (items[old_tail] == x));
    }
}
```
Sequential specification

- if(precondition)
  - Object is in a specified state

- then(postcondition)
  - The method returns a particular value or
  - Throws a particular exception and
  - Leaves the object in a specified state

- Invariants
  - Specified conditions (e.g., object state) must hold anytime a client could invoke an object’s method!
Advantages of sequential specification

- **State between method calls is defined**
  - Enables reasoning about objects
  - Interactions between methods captured by side effects on object state

- **Enables reasoning about each method in isolation**
  - Contracts for each method
  - Local state changes global state

- **Adding new methods**
  - Only reason about state changes that the new method causes
  - If invariants are kept: **no need to check old methods**
  - **Modularity!**
Concurrent execution - State

- Concurrent threads invoke methods on possibly shared objects
  - At overlapping time intervals!

<table>
<thead>
<tr>
<th>Property</th>
<th>Sequential</th>
<th>Concurrent</th>
</tr>
</thead>
<tbody>
<tr>
<td>State</td>
<td>Meaningful only between method executions</td>
<td>Overlapping method executions → object may never be “between method executions”</td>
</tr>
</tbody>
</table>

Each method execution takes some non-zero amount of time!
Concurrent execution - Reasoning

- Reasoning must now include all possible interleavings
  - Of changes caused by methods themselves

<table>
<thead>
<tr>
<th>Property</th>
<th>Sequential</th>
<th>Concurrent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reasoning</td>
<td>Consider each method in isolation; invariants on state before/after execution.</td>
<td>Need to consider all possible interactions; all intermediate states during execution</td>
</tr>
</tbody>
</table>

That is, now we have to consider what will happen if we execute:
- `enq()` concurrently with `enq()`
- `deq()` concurrently with `deq()`
- `deq()` concurrently with `enq()`

Each method execution takes some non-zero amount of time!
Concurrent execution - Method addition

- Reasoning must now include all possible interleavings
  - Of changes caused by and methods themselves

<table>
<thead>
<tr>
<th>Property</th>
<th>Sequential</th>
<th>Concurrent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add Method</td>
<td>Without affecting other methods; invariants on state before/after execution.</td>
<td>Everything can potentially interact with everything else</td>
</tr>
</tbody>
</table>

- Consider adding a method that returns the last item enqueued

```java
Item peek() {
  if (tail == head) throw EmptyException;
  return items[head];
}
```

```java
void enq(Item x) {
  items[tail] = x;
  tail = (tail+1) % items.size();
}
```

```java
Item deq() {
  Item item = items[head];
  head = (head+1) % items.size();
}
```

- If `peek()` and `enq()` run concurrently: what if tail has not yet been incremented?
- If `peek()` and `deq()` run concurrently: what if last item is being dequeued?
Concurrent objects

- How do we describe one?
  - No pre-/postconditions 😊

- How do we implement one?
  - Plan for quadratic or exponential number of interactions and states

- How do we tell if an object is correct?
  - Analyze all quadratic or exponential interactions and states

Is it time to panic for (parallel) software engineers?
Who has a solution?
Lock-based queue

class Queue {
private:
    int head, tail;
    std::vector<Item> items;
    std::mutex lock;

public:
    Queue(int capacity) {
        head = tail = 0;
        items.resize(capacity);
    }
    // ...
};

We can use the lock to protect Queue’s fields.
Lock-based queue

class Queue {
    // ...
    public:
        void enq(Item x) {
            std::lock_guard<std::mutex> l(lock);
            if(((tail+1)%items.size()==head) {
                throw FullException;
            }
            items[tail] = x;
            tail = (tail+1)%items.size();
        }

        Item deq() {  
            std::lock_guard<std::mutex> l(lock);
            if(tail == head) {
                throw EmptyException;
            }
            Item item = items[head];
            head = (head+1)%items.size();
            return item;
        }
    };

Class question: how is the lock ever unlocked?

One of C++'s ways of implementing a critical section
C++ Resource Acquisition is Initialization

- RAII – suboptimal name
- Can be used for locks (or any other resource acquisition)
  - Constructor grabs resource
  - Destructor frees resource
- Behaves as if
  - Implicit unlock at end of block!
- Main advantages
  - Always unlock/free lock at exit
  - No “lost” locks due to exceptions or strange control flow (goto 😃)
  - Very easy to use

```cpp
template <typename mutex_impl>
class lock_guard {
  mutex_impl& _mtx; // ref to the mutex

public:
  lock_guard(mutex_impl& mtx) : _mtx(mtx) { _mtx.lock(); // lock mutex in constructor }

  ~lock_guard() { _mtx.unlock(); // unlock mutex in destructor }
};
```
Example execution

```c++
void enq(Item x) {
    std::lock_guard<std::mutex> l(lock);
    if((tail+1)%items.size()==head) {
        throw FullException;
    }
    items[tail] = x;
    tail = (tail+1)%items.size();
}

Item deq() {
    std::lock_guard<std::mutex> l(lock);
    if(tail == head) {
        throw EmptyException;
    }
    Item item = items[head];
    head = (head+1)%items.size();
    return item;
}
```

Methods effectively execute one after another, sequentially.
Correctness – end of interlude

- Is the locked queue correct?
  - Yes, only one thread has access if locked correctly
  - Allows us again to reason about pre- and postconditions
  - Smells a bit like sequential consistency, no?

- Class question: What is the problem with this approach?
  - Same as for SC 😊

It does not scale!
What is the solution here?
Back to memory models: Language Memory Models

- Which transformations/reorderings can be applied to a program
- Affects platform/system
  - Compiler, (VM), hardware
- Affects programmer
  - What are possible semantics/output
  - Which communication between threads is legal?
- Without memory model
  - Impossible to even define “legal” or “semantics” when data is accessed concurrently
- A memory model is a contract
  - Between platform and programmer
History of Memory Models

- **Java’s original memory model was broken [1]**
  - Difficult to understand => widely violated
  - Did not allow reorderings as implemented in standard VMs
  - Final fields could appear to change value without synchronization
  - Volatile writes could be reordered with normal reads and writes
    => counter-intuitive for most developers

- **Java memory model was revised [2]**
  - Java 1.5 (JSR-133)
  - Still some issues (operational semantics definition [3])

- **C/C++ didn’t even have a memory model until much later**
  - Not able to make any statement about threaded semantics!
  - Introduced in C++11 and C11
  - Based on experience from Java, much more conservative

---

Everybody wants to optimize

- **Language constructs for synchronization**
  - Java: volatile, synchronized, ...
  - C++: atomic, *(NOT volatile!)*, mutex, ...

- **Without synchronization (defined language-specific)**
  - Compiler, (VM), architecture
  - Reorder and appear to reorder memory operations
  - Maintain *sequential semantics* per thread
  - Other threads may observe any order (have seen examples before)
Java and C++ High-level overview

- Relaxed memory model
  - No global visibility ordering of operations
  - Allows for standard compiler optimizations

- But
  - Program order for each thread (sequential semantics)
  - Partial order on memory operations (with respect to synchronizations)
  - Visibility function defined

- Correctly synchronized programs
  - Guarantee sequential consistency

- Incorrectly synchronized programs
  - Java: maintain safety and security guarantees
    - Type safety etc. (require behavior bounded by causality)
  - C++: undefined behavior
    - No safety (anything can happen/change)
Communication between threads: Intuition

- Not guaranteed unless by:
  - Synchronization
  - Volatile/atomic variables
  - Specialized functions/classes (e.g., java.util.concurrent, ...)

```
x = 10
y = 5
flag = true

if(flag)
    print(x+y)
```

Flag is a synchronization variable (atomic in C++, volatile in Java), i.e., all memory written by T1 must be visible to T2 after it reads the value true for flag!
Recap: Memory Model (Intuition)

- Abstract relation between threads and memory
  - Local thread view!

- Does not talk about classes, objects, methods, ...
  - Linearizability is a higher-level concept!

When are values transferred?

Abstraction of caches and registers.
Locks synchronize threads and memory!

- **Java**
  
  ```java
  synchronized (lock) {
    // critical region
  }
  
  - Synchronized methods as syntactic sugar
  ```

- **C++ (RAII)**
  
  ```cpp
  {
    unique_lock<mutex> l(lock);
    // critical region
  }
  
  - Many flexible variants
  ```

- **Semantics:**
  - mutual exclusion
  - at most one thread may hold a lock at a time
  - a thread B trying to acquire a lock held by thread A blocks until thread A releases the lock
  - note: threads may wait forever (no progress guarantee!)
Memory model semantics of locks

- Similar to synchronization variables

- All memory accesses **before** an unlock ...
- are ordered before and are visible to ...
- any memory access **after** a matching lock!
Memory model semantics of synchronization variables

- Variables can be declared volatile (Java) or atomic (C++)

- Reads and writes to synchronization variables
  - Are totally ordered with respect to all threads
  - Must not be reordered with normal reads and writes

- Compiler
  - Must not allocate synchronization variables in registers
  - Must not swap variables with synchronization variables
  - May need to issue memory fences/barriers
  - ...

Memory model semantics of synchronization variables

- **Write to a synchronization variable**
  - Similar memory semantics as unlock (no process synchronization!)

- **Read from a synchronization variable**
  - Similar memory semantics as lock (no process synchronization!)

```java
class example {
    int x = 0;
    atomic<bool> v = false;

    public void writer() {
        x = 42;
        v = true;
    }

    public void reader() {
        if(v) {
            print(x);
        }
    }
}
```

Without atomic or volatile, a platform may reorder these accesses!
Intuitive memory model rules

- **Java/C++**: Correctly synchronized programs will execute sequentially consistent.
- **Correctly synchronized = data-race free**
  - iff all sequentially consistent executions are free of data races
- **Two accesses to a shared memory location form a data race in the execution of a program if**
  - The two accesses are from different threads
  - At least one access is a write and
  - The accesses are not synchronized
Conflicting Accesses

- (recap) two memory accesses conflict if they can happen at the same time (in happens-before) and one of them is a write (store)
- Such a code is said to have a “race condition”
  - Also data-race
  - Trivia around races:
    - The Therac-25 killed three people due to a race
    - A data-race lead to a large blackout in 2003, leaving 55 million people without power causing $1bn damage
- Can be avoided by critical regions
  - Mutually exclusive access to a set of operations
Case Study: Implementing locks - lecture goals

- Among the simplest concurrency constructs
  - Yet, complex enough to illustrate many optimization principles

- Goal 1: You understand locks in detail
  - Requirements / guarantees
  - Correctness / validation
  - Performance / scalability

  Why you do not want to use them in many cases!

- Goal 2: Acquire the ability to design your own locks (and other constructs)
  - Understand techniques and weaknesses/traps
  - Extend to other concurrent algorithms

  Issues are very much the same

- Goal 3: Understand the complexity of shared memory!
  - Memory models in realistic settings
Preliminary Comments

- All code examples are in C/C++ style
  - Neither C (<11) nor C++ (<11) had a clear memory model
  - C++ is one of the languages of choice in HPC
  - Consider source as exemplary (and pay attention to the memory model)!
    
    In fact, many/most textbook examples are incorrect in anything but sequential consistency!

    In fact, you’ll most likely not need those algorithms, but the principles will be useful!

- x86 is really only used because it is common
  - This does not mean that we consider the ISA or memory model elegant!
  - We assume atomic memory (or registers)!
    
    Usually given on x86 (easy to enforce)

- Number of threads/processes is p, tid is the thread id
Recap Concurrent Updates

- Multi-threaded execution!
  - Demo: value of a for p=1?
  - Demo: value of a for p>1?
    Why? Isn’t it a single instruction?

```
const int n=1000;
volatile int a=0;
for (int i=0; i<n; ++i)
a++;
```

```
const int n=1000;
std::atomic<int> a;
a=0;
for (int i=0; i<n; ++i)
a++;
```

```
movl $1000, %eax  // i=n=1000
.L2:
  movl (%rdx), %ecx  // ecx = *a
  addl $1, %ecx    // ecx++
  subl $1, %eax    // i--
  movl %ecx, (%rdx) // *a = ecx
  jne .L2          // loop if i>0
```

```
movl $1000, %eax  // i=n=1000
movl $0, -24(%rsp) // a = 0
mfence          // a is visible!
.L2:
  lock addl $1, -24(%rsp) // (*a)++
  subl $1, %eax    // i--
  jne .L2         // loop if i>0
```
One instruction less! Performance!?

- Run with larger n ($10^8$)
- Compiler: gcc version 7.3.0 (enabled c++11 support, -O3)
- Single-threaded execution only!

```cpp
const int n = 1e8;
volatile int a=0;
for (int i=0; i<n; ++i)
    a++;
```

```cpp
const int n = 1e8;
std::atomic<int> a;
a=0;
for (int i=0; i<n; ++i)
    a++;
```

Demo: 0.17s

Demo: 0.55s

Some Statistics
More formal: Mutual Exclusion

- Control access to a critical region
  - Memory accesses of all processes happen in program order (a partial order, many interleavings)
    
    An execution history defines a total order of memory accesses
  
  - Some subsets of memory accesses (issued by the same process) need to happen **atomically** (thread a’s memory accesses may not be interleaved with other thread’s accesses)
    
    To achieve linearizability!
    
    We need to restrict the valid executions

- Requires synchronization of some sort
  - Many possible techniques (e.g., TM, CAS, T&S, …)
  
  - We first discuss locks which have wait semantics

```
 movl $1000, %eax     // i=1000
.L2:                  
  movl (%rdx), %ecx  // ecx = *a
  addl $1, %ecx     // ecx++
  subl $1, %eax     // i--
  movl %ecx, (%rdx) // *a = ecx
  jne .L2           // loop if i>0
```
Fixing it with locks

- What must the functions lock and unlock guarantee?
  - #1: prevent two threads from simultaneously entering CR
    *i.e., accesses to CR must be mutually exclusive!*
  - #2: ensure consistent memory
    *i.e., stores must be globally visible before new lock is granted!*
- Any performance guesses (remember, 0.17s \(\rightarrow\) 0.55s for atomics)
  - 2.26s
Lock Overview

- **Lock/unlock or acquire/release**
  - Lock/acquire: before entering CR
  - Unlock/release: after leaving CR

- **Semantics:**
  - Lock/unlock pairs must match
  - Between lock/unlock, a thread holds the lock
Desired Lock Properties

- **Mutual exclusion**
  - Only one thread is in the critical region
- **Consistency**
  - Memory operations are visible when critical region is left
- **Progress**
  - If any thread a is not in the critical region, it cannot prevent another thread b from entering
- **Starvation-freedom (implies deadlock-freedom)**
  - If a thread is requesting access to a critical region, then it will eventually be granted access
- **Fairness**
  - A thread a requested access to a critical region before thread b. Was it also granted access to this region before b?
- **Performance**
  - Scaling to large numbers of contending threads
Simplified Notation (cf. Histories)

- **Time defined by precedence (a total order on events)**
  - Events are instantaneous (linearizable)
  - Threads produce sequences of events $a_0, a_1, a_2, \ldots$
  - Program statements may be repeated, denote $i$-th instance of $a$ as $a^i$
  - Event $a$ occurs before event $b$: $a \rightarrow b$
  - An interval $(a,b)$ is the duration between events $a \rightarrow b$
  - Interval $I_1=(a,b)$ precedes interval $I_2=(c,d)$ iff $b \rightarrow c$

- **Critical regions**
  - A critical region $CR$ is an interval $(a,b)$, where $a$ is the first operation in the $CR$ and $b$ the last

- **Mutual exclusion**
  - Critical regions $CR_A$ and $CR_B$ are mutually exclusive if:
    
    *Either* $CR_A \rightarrow CR_B$ *or* $CR_B \rightarrow CR_A$ *for all valid executions!*

- **Assume atomic registers (for now)**
Simple Two-Thread Locks

- A first simple spinlock

```c
volatile int flag = 0;

void lock() {
    while(flag);
    flag = 1;
}

void unlock() {
    flag = 0;
}
```

Busy-wait to acquire lock (spinning)

Is this lock correct?

Why does this not guarantee mutual exclusion?
Simple Two-Thread Locks

- Another two-thread spin-lock: LockOne

```c
volatile int flag[2];

void lock() {
    int j = 1 - tid;
    flag[tid] = true;
    while (flag[j]) {} // wait
}

void unlock() {
    flag[tid] = false;
}
```

When and why does this guarantee mutual exclusion?
Correctness Proof

- In sequential consistency!
- Intuitions:
  - Situation: both threads are ready to enter
  - Show that situation that allows both to enter leads to a schedule violating sequential consistency

  *Using transitivity of program and synchronization orders*
Simple Two-Thread Locks

- Another two-thread spin-lock: LockOne

```c
volatile int flag[2];

void lock() {
    int j = 1 - tid;
    flag[tid] = true;
    while (flag[j]) {} // wait
}

void unlock() {
    flag[tid] = false;
}
```

When and why does this guarantee mutual exclusion?

Does it work in practice?
Simple Two-Thread Locks

- A third attempt at two-thread locking: LockTwo

```c
volatile int victim;

void lock() {
    victim = tid; // grant access
    while (victim == tid) {} // wait
}

void unlock() {}
```

Does this guarantee mutual exclusion?
Correctness Proof

- **Intuition:**
  - Victim is only written once per `lock()`
  - A can only enter after B wrote
  - B cannot enter in any sequentially consistent schedule
Simple Two-Thread Locks

- A third attempt at two-thread locking: LockTwo

```c
volatile int victim;

void lock() {
    victim = tid; // grant access
    while (victim == tid) {} // wait
}

void unlock() {}
```

Does this guarantee mutual exclusion?

Does it work in practice?
Simple Two-Thread Locks

- The last two locks provide mutual exclusion
  - LockOne succeeds iff lock attempts do not overlap
  - LockTwo succeeds iff lock attempts do overlap
- Combine both into one locking strategy!
  - Peterson’s lock (1981)
Peterson’s Two-Thread Lock (1981)

- Combines the first lock (request access) with the second lock (grant access)

```c
volatile int flag[2];
volatile int victim;

void lock() {
    int j = 1 - tid;
    flag[tid] = 1; // I’m interested
    victim = tid; // other goes first
    while (flag[j] && victim == tid) {}; // wait
}

void unlock() {
    flag[tid] = 0; // I’m not interested
}
```
Proof Correctness

- **Intuition:**
  - Victim is written once
  - Pick thread that wrote victim last
  - Show thread must have read flag==0
  - Show that no sequentially consistent schedule permits that
Starvation Freedom

- **(recap) definition:** Every thread that calls lock() eventually gets the lock.
  - Implies deadlock-freedom!
- Is Peterson’s lock starvation-free?

```c
volatile int flag[2];
volatile int victim;

void lock() {
    int j = 1 - tid;
    flag[tid] = 1; // I’m interested
    victim = tid; // other goes first
    while (flag[j] && victim == tid) {}; // wait
}

void unlock() {
    flag[tid] = 0; // I’m not interested
}
```
Proof Starvation Freedom

- Intuition:
  - Threads can only wait/starve in while()
    \[\text{Until flag==0 or victim==other}\]
  - Other thread enters lock() → sets victim to other
    \[\text{Will definitely “unstuck” first thread}\]
  - So other thread can only be stuck in lock()
    \[\text{Will wait for victim==other, victim cannot block both threads \rightarrow one must leave!}\]
Lock Fairness

- Starvation freedom provides no guarantee on how long a thread waits or if it is “passed”!
- To reason about fairness, we define two sections of each lock algorithm:
  - Doorway D (bounded # of steps)
  - Waiting W (unbounded # of steps)

- FIFO locks:
  - If $T_A$ finishes its doorway before $T_B$ then $CR_A \rightarrow CR_B$
  - Implies fairness

void lock() {
  int j = 1 - tid;
  flag[tid] = true; // I’m interested
  victim = tid;    // other goes first
  while (flag[j] && victim == tid) {};
}
Lamport’s Bakery Algorithm (1974)

- Is a FIFO lock (and thus fair)
- Each thread takes a number in the doorway and threads enter in the order of their number!

```java
volatile int flag[n] = {0,0,...,0};
volatile int label[n] = {0,0,...,0};

void lock() {
    flag[tid] = 1; // request
    label[tid] = max(label[0], ..., label[n-1]) + 1; // take ticket
    while ((∃k != tid)(flag[k] && (label[k],k) <* (label[tid],tid))) {};
}

public void unlock() {
    flag[tid] = 0;
}
```
Lamport’s Bakery Algorithm (1974)

- **Advantages:**
  - Elegant and correct solution
  - Starvation free, even FIFO fairness

- **Not used in practice!**
  - Why?
  - Needs to read/write N memory locations for synchronizing N threads
  - Can we do better?
    
    *Using only atomic registers/memory*
A Lower Bound to Memory Complexity

- Theorem 5.1 in [1]: “If S is a [atomic] read/write system with at least two processes and S solves mutual exclusion with global progress [deadlock-freedom], then S must have at least as many variables as processes”

- So we’re doomed! Optimal locks are available and they’re fundamentally non-scalable. Or not?