Independent vs. cooperating threads

- Independent threads
  - no state shared with other threads
  - deterministic --- input state determines result
  - reproducible
  - scheduling order does not matter
  - still not fully isolated (may share files)

- Cooperating threads
  - shared state
  - non-deterministic
  - non-reproducible

Non-reproducibility and non-determinism means that bugs can be intermittent. This makes debugging really hard!
Example: two threads, one counter

- A web site gets millions of hits a day. Uses multiple threads (on multiple processors) to speed things up.
- Simple shared state error: each thread increments a shared counter to track the number of hits today:

```
... 
  hits = hits + 1;
... 
```
- What happens when two threads execute this code concurrently?

Problem with shared counters

- One possible result: lost update!

```
  hits = 0
  read hits (0)
  hits = 0 + 1
  T1
  T2
  read hits (0)
  hits = 0 + 1
  hits = 1
```

- One other possible result: everything works.
  - Bugs are frequently intermittent. Makes debugging hard.
  - This is called “race condition”
Race conditions

- Race condition: timing dependent error involving shared state.
  - whether it happens depends on how threads scheduled
- *Hard* because:
  - must make sure all possible schedules are safe. Number of possible schedules permutations is huge.

```c
if(n == stack_size) /* A */
  return full; /* B */
stack[n] = v; /* C */
n = n + 1; /* D */
```

* Some bad schedules aaccdd, acadcd, ... (how many?)
- they are intermittent. Timing dependent = small changes (adding a print stmt, different machine) can hide bug.

More race condition example:

Thread a:
```
i = 0;
while(i < 10)
i = i + 1;
print "A won!";
```

Thread b:
```
i = 0;
while(i > -10)
i = i - 1;
print "B won!";
```

- Who wins?
- Guaranteed that someone wins?
- What if both threads on its own identical speed CPU executing in parallel? will it go on forever?
Preventing race conditions: atomicity

- atomic unit = instruction sequence guaranteed to execute indivisibly (also, a “critical section”).
  - If two threads execute the same atomic unit at the same time, one thread will execute the whole sequence before the other begins.

  - How to make multiple inst’s seem like one atomic one?

Synchronization motivation

- When threads concurrently read/write shared memory, program behavior is undefined → race conditions
  - Two threads write to the same variable; which one should win?

- Thread schedule is non-deterministic
  - Behavior changes when re-run program

- Compiler/hardware instruction reordering

- Multi-word operations are not atomic
Question: can this panic?

Thread 1

\[ p = \text{someComputation}(); \]
\[ p\text{Initialized} = \text{true}; \]

Thread 2

\[ \text{while} (!p\text{Initialized}) ; \]
\[ q = \text{someFunction}(p); \]
\[ \text{if} (q \neq \text{someFunction}(p)) \]
\[ \text{panic} \]

Why reordering?

- Why do compilers reorder instructions?
  - Efficient code generation requires analyzing control/data dependency
  - If variables can spontaneously change, most compiler optimizations become impossible

- Why do CPUs reorder instructions?
  - Write buffering: allow next instruction to execute while write is being completed

Fix: memory barrier
  - Instruction to compiler/CPU
  - All ops before barrier complete before barrier returns
  - No op after barrier starts until barrier returns
## Example: the Too-Much-Milk problem

<table>
<thead>
<tr>
<th>Time</th>
<th>Event Description</th>
<th>Person A</th>
<th>Person B</th>
</tr>
</thead>
<tbody>
<tr>
<td>3:00</td>
<td>Look in fridge. Out of milk</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3:05</td>
<td>Leave for store</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3:10</td>
<td>Arrive at store</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3:15</td>
<td>Buy milk</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3:20</td>
<td>Arrive home, put milk away</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3:25</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3:30</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Goal:**
1. never more than one person buys
2. someone buys if needed

## Too much milk: solution #1

- **Basic idea:**
  - leave a note (kind of like “lock”)
  - remove note (kind of like “unlock”)
  - don’t buy if there is a note (wait)

```java
if (noMilk) {
  if (noNote) {
    leave Note;
    buy milk;
    remove Note
  }
}
```
Why solution #1 does not work?

Thread A
3:00 if (noMilk) {
3:05 if (noNote) {
3:10
3:15
3:20 leave Note;
3:25 buy milk;
3:30 remove Note} }

Thread B

Threads can get context-switched at any time!

Too much milk: solution #2

Thread A
leave NoteA
if (noNoteB) {
if (noMilk)
  buy milk
}
remove NoteA

Thread B
leave NoteB
if (noNoteA) {
if (noMilk)
  buy milk
}
remove NoteB

Problem: neither thread to buy milk --- think other is going to buy --- starvation!
Too much milk: solution #3

Thread A
leave NoteA
while (NoteB) // X
do nothing;
if (noMilk)
  buy milk;
remove NoteA

Thread B
leave NoteB
if (noNoteA) { // Y
  if (noMilk)
    buy milk;
}
remove NoteB

Either safe for me to buy or others will buy!

It works but:
- it is too complex
- A’s code different from B’s (what if lots of threads?)
- A busy-waits --- consumes CPU!

A better solution

- Have hardware provide better primitives than atomic load and store.
- Build higher-level programming abstractions on this new hardware support.
- Example: using locks as an atomic building block
  
  Acquire --- wait until lock is free, then grabs it
  Release --- unlock, waking up a waiter if any

  These must be atomic operations --- if two threads are waiting for the lock, and both see it is free, only one grabs it!
Too much milk: using a lock

◆ It is really easy!

  lock -> Acquire();
  if (nomilk)
    buy milk;
  lock -> Release();

◆ What makes a good solution?
- Only one process inside a critical section
- No assumption about CPU speeds
- Processes outside of critical section should not block other processes
- No one waits forever
- Works for multiprocessors

◆ Future topics:
- hardware support for synchronization
- high-level synchronization primitives & programming abstraction
- how to use them to write correct concurrent programs?

A few definitions

◆ Synchronization:
  - using atomic operations to ensure cooperation between threads

◆ Mutual exclusion:
  - ensuring that only one thread does a particular thing at a time. One thread doing it excludes the other, and vice versa.

◆ Critical section:
  - piece of code that only one thread can execute at once. Only one thread at a time will get into the section of code.

◆ Lock: prevents someone from doing something
  - lock before entering critical section, before accessing shared data
  - unlock when leaving, after done accessing shared data
  - wait if locked
A quick recap

- We talked about critical section
  ```java
  Acquire(lock);
  if (noMilk)
      buy milk;
  Release(lock);
  ```
  Critical section
- We also talked about what is a good solution
  - Only one process inside a critical section
  - No assumption about CPU speeds
  - Processes outside of critical section should not block other processes
  - No one waits forever
  - Works for multiprocessors

How to write concurrent programs?

Use shared objects (aka concurrent objects) --- always encapsulate (hide) its shared state
The big picture

Concurrent Applications

Shared Objects

Bounded Buffer  Barrier

Synchronization Variables

Semaphores  Locks  Condition Variables

Atomic Instructions

Interrupt Disable  Test-and-Set

Hardware

Multiple Processors  Hardware Interrupts

The big picture (cont’d)

- **Shared object layer**: all shared objects appear to have the same interface as those for a single-threaded program

- **Synchronization variable layer**: a synchronization variable is a data structure used for coordinating concurrent access to shared state

- **Atomic instruction layer**: atomic processor-specific instructions
The big picture

Concurrent Applications

Shared Objects

Bounded Buffer  Barrier

Synchronization Variables

Semaphores  Locks  Condition Variables

Atomic Instructions

Interrupt Disable  Test-and-Set

Hardware

Multiple Processors  Hardware Interrupts

Locks

- Lock::acquire
  - wait until lock is free, then take it
- Lock::release
  - release lock, waking up anyone waiting for it

1. At most one lock holder at a time (safety)
2. If no one holding, acquire gets lock (progress)
3. If all lock holders finish and no higher priority waiters, waiter eventually gets lock (progress)
Question: why only Acquire/Release

- Suppose we add a method to a lock, to ask if the lock is free. Suppose it returns true. Is the lock:
  - Free?
  - Busy?
  - Don’t know?

Lock example: malloc/free

```c
char *malloc (n) {
    heaplock.acquire();
    p = allocate memory
    heaplock.release();
    return p;
}

void free(char *p) {
    heaplock.acquire();
    put p back on free list
    heaplock.release();
}
```
Rules for using locks

- Lock is initially free
- Always acquire before accessing shared data structure
  - Beginning of procedure!
- Always release after finishing with shared data
  - End of procedure!
  - Only the lock holder can release
  - DO NOT throw lock for someone else to release
- Never access shared data without lock
  - Danger!

Will this code work?

```
if (p == NULL) {
    lock.acquire();
    if (p == NULL) {
        p = newP();
    }
    lock.release();
}
use p->field1
```

```
newP() {
    p = malloc(sizeof(p));
    p->field1 = ...;
    p->field2 = ...;
    return p;
}
```
Example: thread-safe bounded queue

```cpp
// Thread-safe queue interface
const int MAX = 10;
class TSQueue {
    // Synchronization variables
    Lock lock;
    // State variables
    int items[MAX];
    int front;
    int nextEmpty;

public:
    TSQueue();
    ~TSQueue();
    bool tryInsert(int item);
    bool tryRemove(int *item);
};
```

// Initialize the queue to empty
// and the lock to free.
TSQueue::TSQueue() {
    front = nextEmpty = 0;
}

// Try to insert an item.
// If the queue is full, return false;
// otherwise return true.
bool TSQueue::tryInsert(int item) {
    bool success = false;
    lock.acquire();
    if ((nextEmpty - front) % MAX) {
        items[nextEmpty % MAX] = item;
        nextEmpty++;
        success = true;
    }
    lock.release();
    return success;
}

// Try to remove an item. If the queue
// is empty, return false;
// otherwise return true.
bool TSQueue::tryRemove(int *item) {
    bool success = false;
    lock.acquire();
    if (front < nextEmpty) {
        *item = items[front % MAX];
        front++;
        success = true;
    }
    lock.release();
    return success;
}
Example: thread-safe bounded queue

The lock holder always maintain the following invariants when releasing the lock:

- The total number of items ever inserted in the queue is `nextEmpty`.
- The total number of items ever removed from the queue is `front`.
- `front <= nextEmpty`
- The current number of items in the queue is `nextEmpty - front`
- `nextEmpty - front <= MAX`

Example: thread-safe bounded queue

```c
// TSQueueMain.cc
// Test code for TSQueue.
int main(int argc, char **argv) {
    TSQueue *queues[3];
    thread_t workers[3];
    int i, j;
    // Start worker threads to insert.
    for (i = 0; i < 3; i++) {
        queues[i] = new TSQueue();
        thread_create(&workers[i], putSome, queues[i]);
    }
    // Wait for some items to be put.
    thread_join(workers[0]);
    // Remove 20 items from each queue.
    for (i = 0; i < 3; i++) {
        printf("Queue %d:
", i);
        testRemoval(&queues[i]);
    }
}

// Insert 50 items into a queue.
void *putSome(void *p) {
    TSQueue *queue = (TSQueue *)p;
    int i;
    for (i = 0; i < 50; i++) {
        queue->tryInsert(i);
    }
    return NULL;
}

// Remove 20 items from a queue.
void testRemoval(TSQueue *queue) {
    int i, item;
    for (i = 0; i < 20; i++) {
        if (queue->tryRemove(&item)) {
            printf("Removed %d\n", item);
        } else {
            printf("Nothing there.\n");
        }
    }
}
```
The big picture

Concurrent Applications

Shared Objects

Bounded Buffer  Barrier

Synchronization Variables

Semaphores  Locks  Condition Variables

Atomic Instructions

Interrupt Disable  Test-and-Set

Hardware

Multiple Processors  Hardware Interrupts

How to use the lock?

- The lock provides mutual exclusion to the shared data
- Rules for using a lock:
  - Always acquire before accessing shared data structure
  - Always release after finishing with shared data
  - Lock is initially free.

- Simple example: a synchronized queue

```cpp
bool tryInsert()
{
    lock.Acquire();  // lock before use
    ... put item on queue;  // ok to access
    lock.Release();  // unlock after done
    return success;
}

bool tryRemove()
{
    lock.Acquire();  // can we wait?
    if something on queue
        remove it;
    lock->Release();
    return success;
}
```
Condition variables

- How to make `tryRemove` wait until something is on the queue?
  - can't sleep while holding the lock
  - Key idea: make it possible to go to sleep inside critical section, by atomically releasing lock at same time we go to sleep.

- **Condition variable**: a queue of threads waiting for something inside a critical section.
  - `Wait()` --- Release lock, go to sleep, re-acquire lock
    * release lock and going to sleep is atomic
  - `Signal()` --- Wake up a waiter, if any
  - `Broadcast()` --- Wake up all waiters

Synchronized queue using condition variables

- **Rule**: must hold lock when doing condition variable operations

```java
addToQueue()
{
    lock.acquire();
    put item on queue;
    condition.signal();
    lock.release();
}
```

```java
removeFromQueue()
{
    lock.acquire();

    while nothing on queue
        condition.wait(&lock);
        // release lock; got to
        // sleep; reacquire lock

    remove item from queue;
    lock.release();
    return item;
}
```
Condition variable design pattern

```java
methodThatWaits() {
    lock.acquire();

    // Read/write shared state
    while (!testSharedState()) {
        cv.wait(&lock);
    }

    // Read/write shared state
    lock.release();
}
```

```java
methodThatSignals() {
    lock.acquire();

    // Read/write shared state
    if (testSharedState())
        cv.signal(&lock);

    // Read/write shared state
    lock.release();
}
```

Example: blocking bounded queue

```java
// Thread-safe blocking queue.
const int MAX = 10;

class BBQ{
    // Synchronization variables
    Lock lock;
    CV itemAdded;
    CV itemRemoved;

    // State variables
    int items[MAX];
    int front;
    int nextEmpty;

    public:
    BBQ();
    ~BBQ() {};
    void insert(int item);
    int remove();
};
```
Example: blocking bounded queue

```cpp
// Wait until there is room and // then insert an item.
void BBQ::insert(int item) {
    lock.acquire();
    while ((nextEmpty - front) == MAX) {
        itemRemoved.wait(&lock);
    }
    items[nextEmpty % MAX] = item;
    nextEmpty++;
    itemAdded.signal();
    lock.release();
}

// Wait until there is an item and // then remove an item.
int BBQ::remove() {
    int item;
    lock.acquire();
    while (front == nextEmpty) {
        itemAdded.wait(&lock);
    }
    item = items[front % MAX];
    front++;
    itemRemoved.signal();
    lock.release();
    return item;
}

// Initialize the queue to empty, // the lock to free, and the // condition variables to empty.
BBQ::BBQ() {
    front = nextEmpty = 0;
}
```

Pre/Post conditions & invariants

- What is state of the blocking bounded queue at lock acquire?
  - front <= nextEmpty
  - front + MAX >= nextEmpty

- These are also true on return from wait

- And at lock release

- Allows for proof of correctness
Pre/Post conditions & invariants

```java
methodThatWaits() {
    lock.acquire();
    // Pre-condition: State is consistent
    // Read/write shared state
    while (!testSharedState()) {
        cv.wait(&lock);
    }
    // WARNING: shared state may have changed! But
    // testSharedState is TRUE
    // and pre-condition is true
    // Read/write shared state
    lock.release();
}

methodThatSignals() {
    lock.acquire();
    // Pre-condition: State is consistent
    // Read/write shared state
    // If testSharedState is now true
    cv.signal(&lock);
    // NO WARNING: signal keeps lock
    // Read/write shared state
    lock.release();
}
```

Condition variables

- **ALWAYS** hold lock when calling wait, signal, broadcast
  - Condition variable is sync FOR shared state
  - **ALWAYS** hold lock when accessing shared state

- **Condition variable is memoryless**
  - If signal when no one is waiting, no op
  - If wait before signal, waiter wakes up

- **Wait atomically releases lock**
  - What if wait, then release?
  - What if release, then wait?
**Question 1:** *wait replaced by unlock + sleep?*

```java
methodThatWaits() {
    lock.acquire();
    // Read/write shared state
    while (!testSharedState()) {
        lock.release();
        cv.sleep(&lock);
    }
    // Read/write shared state
    lock.release();
}

methodThatSignals() {
    lock.acquire();
    // Read/write shared state
    if (testSharedState) { // Should be true here
        cv.signal(&lock);
        // Read/write shared state
        lock.release();
    }
}
```

**Question 2:** *wait does not acquire lock?*

```java
methodThatWaits() {
    lock.acquire();
    // Read/write shared state
    while (!testSharedState()) {
        cv.wait(&lock);
        lock.acquire();
    }
    // Read/write shared state
    lock.release();
}

methodThatSignals() {
    lock.acquire();
    // Read/write shared state
    if (testSharedState) { // Should be true here
        cv.signal(&lock);
        // Read/write shared state
        lock.release();
    }
}
```
Condition variables, cont’d

◆ When a thread is woken up from `wait`, it may not run immediately
  - Signal/broadcast put thread on `ready list`
  - When lock is released, anyone might acquire it

◆ Wait MUST be in a loop
  ```c
  while (needToWait()) {
    condition.Wait(lock);
  }
  ```

◆ Simplifies implementation
  - Of condition variables and locks
  - Of code that uses condition variables and locks

Structured synchronization

◆ Identify objects or data structures that can be accessed by multiple threads concurrently
◆ Add locks to object/module
  - Grab lock on start to every method/procedure
  - Release lock on finish
◆ If need to wait
  - while(needToWait()) { condition.Wait(lock); }
  - Do not assume when you wake up, signaller just ran
◆ If do something that might wake someone up
  - Signal or Broadcast
◆ Always leave shared state variables in a consistent state
  - When lock is released, or when waiting
Monitors and condition variables

- **Monitor definition:**
  - a lock and zero or more condition variables for managing concurrent access to shared data

- **Monitors make things easier:**
  - “locks” for mutual exclusion
  - “condition variables” for scheduling constraints

Monitors embedded in prog. languages (1)

- **High-level data abstraction that unifies handling of:**
  - Shared data, operations on it, synch and scheduling
    - All operations on data structure have single (implicit) lock
    - An operation can relinquish control and wait on condition
    - // only one process at time can update instance of Q
      class Q {
        int head, tail;  // shared data
        void enq(v) { locked access to Q instance }
        int deq() { locked access to Q instance }
      }
  - Java from Sun; Mesa/Cedar from Xerox PARC

- **Monitors easier and safer than semaphores**
  - Compiler can check, lock implicit (cannot be forgotten)
### Monitors embedded in prog. languages (2)

- **Wait()**
  - Block on "condition"
- **Signal()**
  - Wakeup a blocked process on "condition"

### Java language manual

When waiting upon a Condition, a "spurious wakeup" is permitted to occur, in general, as a concession to the underlying platform semantics. This has little practical impact on most application programs as a Condition should always be waited upon in a loop, testing the state predicate that is being waited for.
Remember the rules

- Use consistent structure
- Always use locks and condition variables
- Always acquire lock at beginning of procedure, release at end
- Always hold lock when using a condition variable
- Always wait in while loop
- Never spin in sleep()

Mesa vs. Hoare semantics

- Mesa
  - Signal puts waiter on ready list
  - Signaller keeps lock and processor
- Hoare
  - Signal gives processor and lock to waiter
  - When waiter finishes, processor/lock given back to signaller
  - Nested signals possible!

- For Mesa-semantics, you always need to check the condition after wait (use “while”). For Hoare-semantics you can change it to “if”
The big picture: more examples

- Concurrent Applications
- Shared Objects
- Bounded Buffer
- Barrier
- Synchronization Variables
- Semaphores
- Locks
- Condition Variables
- Atomic Instructions
- Interrupt Disable
- Test-and-Set
- Hardware
- Multiple Processors
- Hardware Interrupts

Producer-consumer with monitors

```
Condition full;
Condition empty;
Lock lock;

Producer() {
    lock.Acquire();
    while (the buffer is full)
        full.wait(&lock);
    put 1 Coke in machine;
    if (the buffer was empty)
        empty.signal();
    lock.Release();
}

Consumer() {
    lock.Acquire();
    while (the buffer is empty)
        empty.wait(&lock);
    take 1 Coke;
    if (the buffer was full)
        full.signal();
    lock.Release();
}
```
Example: the readers/writers problem

◆ Motivation
- shared database (e.g., bank balances / airline seats)
- Two classes of users:
  * Readers --- never modify database
  * Writers --- read and modify database
- Using a single lock on the database would be overly restrictive
  * want many readers at the same time
  * only one writer at the same time

◆ Constraints
  * Readers can access database when no writers (Condition okToRead)
  * Writers can access database when no readers or writers (Condition okToWrite)
  * Only one thread manipulates state variable at a time

Design specification (readers/writers)

◆ Reader
  - wait until no writers
  - access database
  - check out - wake up waiting writer
◆ Writer
  - wait until no readers or writers
  - access database
  - check out --- wake up waiting readers or writer
◆ State variables
  - # of active readers (AR); # of active writers (AW);
  - # of waiting readers (WR); # of waiting writers (WW);
◆ Lock and condition variables: okToRead, okToWrite
Solving readers/writers

```
Reader() {
    // first check self into system
    lock.Acquire();
    while ((AW+WW) > 0) {
        WR ++;
        okToRead.Wait(&lock);
        WR --;
    }
    AR++;
    lock.Release();
    Access DB;
    // check self out of system
    lock.Acquire();
    AR--;
    if (AR == 0 & & WW > 0)
        okToWrite.Signal(&lock);
    lock.Release();
}

Writer() {
    // first check self into system
    lock.Acquire();
    while ((AW+AR) > 0) {
        WW ++;
        okToWrite.Wait(&lock);
        WW --;
    }
    AW++;
    lock.Release();
    Access DB;
    // check self out of system
    lock.Acquire();
    AW--;
    if (WW > 0) okToWrite.Signal(&lock);
    else if (WR > 0) okToRead.Broadcast(&lock);
    lock.Release();
}
```

Example: the one-way-bridge problem

- **Problem definition**
  - a narrow light-duty bridge on a public highway
  - traffic cross in one direction at a time
  - at most 3 vehicles on the bridge at the same time (otherwise it will collapses)

- **Each car is represented as one thread:**

  ```
  OneVehicle (int direc) {
      ArriveBridge (direc);
      ... crossing the bridge ...;
      ExitBridge(direc);
  }
  ```
One-way bridge with condition variables

```
Lock lock;
Condition safe;   // safe to cross bridge
int currentNumber;  // # of cars on bridge
int currentDirec;   // current direction

ArriveBridge(int direc) {
lock.Acquire();
while (! safe-to-cross(direc)) {
    safe.wait(lock)
}
currentNumber++;
currentDirec = direc;
lock.Release();
}

ExitBridge(int direc) {
lock.Acquire();
currentNumber--;
safe.signal(lock);
lock.Release();
}

safe-to-cross(int direc) {
    if (currentNumber == 0)
        return TRUE;  // always safe if empty
    else if ((currentNumber < 3) &&
             (currentDirec == direc))
        return TRUE;
    else
        return FALSE;
}
```

The mating-whales problem

◆ You have been hired by Greenpeace to help the environment. Because unscrupulous commercial interests have dangerously lowered the whale population, whales are having synchronization problems in finding a mate.

◆ To have children, three whales are needed, one male, one female, and one to play matchmaker --- literally, to push the other two whales together (I'm not making this up!).

◆ Write the three procedures:

```
void Male()
void Female()
void Matchmaker()
```

using locks and Mesa-style condition variables. Each whale is represented by a separate thread. A male whale calls Male() which waits until there is a waiting female and matchmaker; similarly, a female whale must wait until a male whale and a matchmaker are present. Once all three are present, all three return.
Step 1 --- two-way rendezvous

Lock* lock;
Condition* malePresent;
Condition* maleToGo;
int numMale = 0;
bool maleCanGo = FALSE;

void Male() {
    lock->Acquire();
    numMale++;
    malePresent->Signal();
    while (!maleCanGo) {
        maleToGo->Wait(lock);
    }
    maleCanGo = FALSE;
    lock->Release();
}

void MatchMaker() {
    lock->Acquire();
    while (numMale == 0) {
        malePresent->Wait(lock);
    }
    maleCanGo = TRUE;
    maleToGo->Signal();
    numMale--;
    lock->Release();
}

Step 2 --- three-way rendezvous

Lock* lock;
Condition* malePresent;
Condition* maleToGo;
int numMale = 0;
bool maleCanGo = FALSE;

Condition* femalePresent;
Condition* femaleToGo;
int numFemale = 0;
bool femaleCanGo = FALSE;

void Male() {
    lock->Acquire();
    numMale++;
    malePresent->Signal();
    while (!maleCanGo) {
        maleToGo->Wait(lock);
    }
    maleCanGo = FALSE;
    lock->Release();
}

void MatchMaker() {
    lock->Acquire();
    while (numMale == 0) {
        malePresent->Wait(lock);
    }
    maleCanGo = TRUE;
    maleToGo->Signal();
    numMale--;
    lock->Release();
}

void Female() {
    lock->Acquire();
    numFemale++;
    femalePresent->Signal();
    while (!femaleCanGo) {
        femaleToGo->Wait(lock);
    }
    femaleCanGo = FALSE;
    lock->Release();
}

while (numFemale == 0) {
    femalePresent->Wait(lock);
}
    femaleCanGo = TRUE;
    femaleToGo->Signal();
    numFemale--;
    lock->Release();

### Step 3 --- a simplified version

```c
Lock* lock;
Condition* malePresent;
Condition* maleToGo;
int numMale = 0;
Condition* femalePresent;
Condition* femaleToGo;
int numFemale = 0;

void MatchMaker() {
    lock->Acquire();
    while (numMale == 0) {
        malePresent->Wait(lock);
    }
    while (numFemale == 0) {
        femalePresent->Wait(lock);
    }
    maleToGo->Signal();
    numMale--;
    femaleToGo->Signal();
    numFemale--;
    lock->Release();
}

void Male() {
    lock->Acquire();
    numMale++;
    malePresent->Signal();
    maleToGo->Wait(lock);
    lock->Release();
}

void Female() {
    lock->Acquire();
    numFemale++;
    femalePresent->Signal();
    femaleToGo->Wait(lock);
    lock->Release();
}
```

### Example: A MapReduce single-use barrier

```c
// A single use synch barrier.
class Barrier{
    private:
        // Synchronization variables
        Lock lock;
        CV allCheckedIn;
        // State variables
        int numEntered;
        int numThreads;

    public:
        Barrier(int n);
        ~Barrier();
        void checkin();
    };

    Barrier::Barrier(int n) {
        numEntered = 0;
        numThreads = n;
    }

    // No one returns until all threads
    // have called checkin.
    void checkin() {
        lock.acquire();
        numEntered++;
        if (numEntered < numThreads) {
            while (numEntered < numThreads)
                allCheckedIn.wait(&lock);
        } else { // last thread to checkin
            allCheckedIn.broadcast();
        }
        lock.release();
    }
```

Create n threads; Create barrier; Each thread executes map operation; barrier.checkin(); Each thread sends data to reducers; barrier.checkin(); Each thread executes reduce operation; barrier.checkin();
Example: A reusable synch barrier

class Barrier{
    private:
        // Synchronization variables
        Lock lock;
        CV allCheckedIn;
        CV allLeaving;

        // State variables
        int numEntered;
        int numLeaving;
        int numThreads;

    public:
        Barrier(int n);
        ~Barrier();
        void checkin();
    };

    Barrier::Barrier(int n) {
        numEntered = 0;
        numLeaving = 0;
        numThreads = n;
    }

    // No one returns until all threads have called checkin.
    void checkin() {
        lock.acquire();
        numEntered++;
        if (numEntered < numThreads) {
            while (numEntered < numThreads)
                allCheckedIn.wait(&lock);
        } else {
            // no threads in allLeaving wait
            numLeaving = 0;
            allCheckedIn.broadcast();
        }
        numLeaving++;
        if (numLeaving < numThreads) {
            while (numLeaving < numThreads)
                allLeaving.wait(&lock);
        } else {
            // no threads in allCheckedIn wait
            numEntered = 0;
            allLeaving.broadcast();
        }
        lock.release();
    }

Example: blocking bounded queue

    // Thread-safe blocking queue.
    const int MAX = 10;

    class BBQ{
        // Synchronization variables
        Lock lock;
        CV itemAdded;
        CV itemRemoved;

        // State variables
        int items[MAX];
        int front;
        int nextEmpty;

    public:
        BBQ();
        ~BBQ() {};
        void insert(int item);
        int remove();
    };

    // No one returns until all threads have called checkin.
Example: blocking bounded queue

```c
void BBQ::insert(int item) {
    lock.acquire();
    while ((nextEmpty - front) == MAX) {
        itemRemoved.wait(&lock);
    }
    items[nextEmpty % MAX] = item;
    nextEmpty++;
    itemAdded.signal();
    lock.release();
}
```

```c
// Wait until there is room and
// then insert an item.

void BBQ::insert(int item) {
    lock.acquire();
    while ((nextEmpty - front) == MAX) {
        itemRemoved.wait(&lock);
    }
    items[nextEmpty % MAX] = item;
    nextEmpty++;
    itemAdded.signal();
    lock.release();
}
```

```c
// Wait until there is an item and
// then remove an item.

int BBQ::remove() {
    int item;
    lock.acquire();
    while (front == nextEmpty) {
        itemAdded.wait(&lock);
    }
    item = items[front % MAX];
    front++;
    itemRemoved.signal();
    lock.release();
    return item;
}
```

```
// Wait until there is an item and
// then remove an item.

int BBQ::remove() {
    int item;
    lock.acquire();
    while (front == nextEmpty) {
        itemAdded.wait(&lock);
    }
    item = items[front % MAX];
    front++;
    itemRemoved.signal();
    lock.release();
    return item;
}
```

---

Starvation-Free (FIFO) BBQ

```
ConditionQueue insertQueue, removeQueue;
int numRemoveCalled = 0;  // # of times remove has been called
int numInsertCalled = 0;  // # of times insert has been called

int FIFOBBQ::remove() {
    int item, myPosition;
    CV *myCV, *nextWaiter;
    lock.acquire();
    myPosition = numRemoveCalled++;
    myCV = new CV;  // Create a new condition variable to wait on.
    removeQueue.append(myCV);

    // Even if I am woken up, wait until it is my turn.
    while (front < myPosition || front == nextEmpty) {
        myCV->Wait(&lock);
    }

    delete myCV;  // The condition variable is no longer needed.
    item = items[front % MAX];
    front++;

    // Wake up the next thread waiting in insertQueue, if any.
    nextWaiter = insertQueue.removeFromFront();
    if (nextWaiter != NULL) nextWaiter->Signal(&lock);

    lock.release();
    return item;
}
```

```
ConditionQueue insertQueue, removeQueue;
int numRemoveCalled = 0;  // # of times remove has been called
int numInsertCalled = 0;  // # of times insert has been called

int FIFOBBQ::remove() {
    int item, myPosition;
    CV *myCV, *nextWaiter;
    lock.acquire();
    myPosition = numRemoveCalled++;
    myCV = new CV;  // Create a new condition variable to wait on.
    removeQueue.append(myCV);

    // Even if I am woken up, wait until it is my turn.
    while (front < myPosition || front == nextEmpty) {
        myCV->Wait(&lock);
    }

    delete myCV;  // The condition variable is no longer needed.
    item = items[front % MAX];
    front++;

    // Wake up the next thread waiting in insertQueue, if any.
    nextWaiter = insertQueue.removeFromFront();
    if (nextWaiter != NULL) nextWaiter->Signal(&lock);

    lock.release();
    return item;
}
```
**Starvation-Free (FIFO) BBQ (cont’d)**

```cpp
ConditionQueue insertQueue, removeQueue;
int numRemoveCalled = 0; // # of times remove has been called
int numInsertCalled = 0; // # of times insert has been called

void FIFOBBQ::insert(int item) {
    int myPosition;
    lock.acquire();
    myPosition = numInsertCalled++;
    myCV = new CV;
    insertQueue.append(myCV);

    while (nextEmpty < myPosition || (nextEmpty - front) == MAX) {
        myCV->wait(&lock);
    }

delete myCV;
    items[nextEmpty % MAX] = item;
    nextEmpty ++;

    nextWaiter = removeQueue.removeFromFront();
    if (nextWaiter != NULL) nextWaiter->Signal();
}
```

**Starvation-Free (FIFO) BBQ**

- **Bug 1**: keeping destroyed CVs inside the `removeQueue`
  - Buffer size MAX=1, one producer and one consumer
  - Producer inserts one item when the buffer is empty
  - Producer tries to insert again and sleep on a 2nd allocated CV
  - Consumer calls remove successfully and wakes up the first CV in the `insertQueue`; the CV is NULL, so Consumer moves on;
  - Consumer calls removes again but had to sleep because the buffer is empty.

- **Bug 2**: starvation when multiple CVs are waken up
  - Buffer size MAX=2; one producer and two consumers (C1,C2)
  - Two consumers run first and sleeps on empty buffer
  - Producer inserts one item and wakes up C1; P inserts another one and wakes up C2;
  - C2 is scheduled first; but (front < myPosition), so it is not C2’s turn; so it goes to sleep; then C1 finishes; C2 will never wake up
Starvation-Free (FIFO) BBQ [Bug Fixed]

```cpp
int FIFOBBQ::remove () {
    int item, myPosition;
    CV *myCV, *nextWaiter;
    lock.acquire();
    myPosition = numRemoveCalled++;
    myCV = new CV;
    removeQueue.append(myCV);
    while (front < myPosition || front == nextEmpty) {
        myCV->wait(&lock);
    }
    delete myCV;
    item = items[front % MAX];
    front ++;
    nextWaiter = insertQueue.peekFront();
    if (nextWaiter != NULL) nextWaiter->Signal();
    removeQueue.removeFromFront(); // the remover now responsible for removing itself from the removeQueue
    nextWaiter = removeQueue.peekFront(); // the remover responsible for waking up the next in the removeQueue
    if (nextWaiter != NULL) nextWaiter->Signal();
    lock.release();
    return item;
}
```