Monitors and Blocking Synchronization

Adapted from the Companion slides for The Art of Multiprocessor Programming by Maurice Herlihy & Nir Shavit
Last Lecture: Spin-Locks

- Spin lock
- Critical section
- Resets lock upon exit
Spin-Locks

- Test-and-Set
- Test-and-Test-and-Set
- Bus-based architecture and cache coherence
Introduce delay

- Exponential Back-off Lock
  - Choosing parameters was tedious
  - Not suitable across platforms
- Anderson Queue Lock
  - Space hog…
  - One bit per thread $\Rightarrow$ one cache line per thread (What if unknown number of threads? Or What if small number of actual contenders?)
CLH Lock

• CLH Lock
  – Lock release affects predecessor only
  – Small, constant-sized space
  – Does not need to know number of threads
  – Provides fairness
  – Doesn’t work for uncached NUMA architectures
MCS Lock

- Releasing lock requires spinning
- Requires more reads, writes and compareAndSet() calls than CLHLock algorithm.
Abortable Lock

• **Time-out Lock**
  – Local spinning on cached location
  – Small, constant-sized space
  – Wait-free time out (as also in BackoffLock)
  – New node per lock access
  – Going up chain of timed-out node to access critical section.
Spin-Locks Summary

- Each better than others in some way
- There is no one solution
- Lock we pick really depends on:
  - the application
  - the hardware
  - which properties are important
Monitors

• Two threads: a producer and a consumer

```java
mutex.lock();
try {
    queue.enq(object);
}
finally {
    mutex.unlock();
}
```
Monitors

- Two threads: a producer and a consumer

```java
Object result = null;
mutex.lock();
try
{
    result = queue.deq();
}
finally
{
    mutex.unlock();
}
return result;
```
Monitors

• Suppose queue is bounded, attempt to add an item cannot proceed
• Decision depends on queue internal state
• Thread need to keep track of both the lock, and the queue objects
• Application is correct only if thread use same locking convention.
Monitors

• Alternatively, let each queue manage its own synchronization. (queue has internal lock acquired by methods)
• No need to ensure that threads using a queue follows a synchronization protocol
• If a thread calls enq() and queue is full, enq() should detect, suspend caller and resume later.
Monitors

• Monitors are a structured way of combining synchronization, data and methods.

• A **class** encapsulates:
  – data
  – methods (behaviours)

• A **monitor** adds synchronization mechanisms.
Monitors

• A lock is a basic mechanism for ensuring mutual exclusion.
• A monitor exports a collection of methods (that make use of locks)
Waiting Thread

- Release lock while waiting
- When “something happens”
  - Reacquire lock
  - Either
    - Rerelease lock & resume waiting
    - Finish up and return
Styles of Waiting

• Spinning
  – Repeatedly retest condition

• Blocking
  – Ask OS to run someone else
Styles of Waiting

• Spinning
  – Good for very short intervals
  – Expensive to call OS
  – Works only on multiprocessors!

• Blocking
  – Good for longer intervals
  – Processor can do work
Spinning and Blocking

• For example;
  – Consumer thread for deq()?

Block
Spinning and Blocking

• Clever libraries sometimes mix
• Combine Spinning and Blocking
  – Thread waiting to dequeue may spin then block after some time.
package java.util.concurrent.locks;

public interface Lock {
    void lock();
    void lockInterruptibly() throws InterruptedException;
    boolean tryLock();
    boolean tryLock(long time, TimeUnit unit);
    Condition newCondition();
    void unlock();
}
Conditions

Consider the following:

• a queue that links producers and consumers, and

• a consumer waiting on an empty queue.

What should the consumer do?
Conditions

• While a thread is waiting
  – Release the lock on the queue
  – After the waiting thread has released lock, it must be notified to reacquire it and try again
Conditions

• Ability to release lock temporarily:
  Condition object associated with lock

```java
Condition condition =
    mutex.newCondition();

mutex.lock();
try{
    while (!property)
        condition.await();
}
catch (InterruptedException e) {}  // Property holds...
```
Conditions

• A thread that wants to wait for a condition property:
  – Tests for property while holding lock
  – If property does not hold, calls `await()` to release lock and sleep until it is awakened by another thread.
Conditions

• A thread that wants to wait for a condition property:
  – Tests for property while holding lock
  – If property does not hold, calls `await()` to release lock and sleep until it is awakened by another thread.

• There is no guarantee that the property will hold when the thread awakens
Conditions
(Variations of await())

public interface Condition {
    void await() throws InterruptedException;
    boolean await(long time, TimeUnit unit)
            throws InterruptedException;
    boolean awaitUntil(Date deadline)
            throws InterruptedException;
    long awaitNanos(long nanosTimeout)
            throws InterruptedException;
    void awaitUninterruptibly();
    void signal(); // wake up one waiting thread
    void signalAll(); // wake up all waiting threads
}

Art of Multiprocessor Programming
Monitor execution
Monitor execution
Monitor execution

Art of Multiprocessor Programming
Example: Bounded FIFO Queue

class LockedQueue<T> {
    final Lock lock = new ReentrantLock();
    final Condition notFull = lock.newCondition();
    final Condition notEmpty = lock.newCondition();
    final T[] items;
    int tail, head, count;
    public LockedQueue(int capacity) {
        items = (T[]) new Object[capacity];
    }
}
Example: Bounded FIFO Queue

```java
public void enq(T x) {
    lock.lock();
    try {
        while (count == items.length)
            notFull.await();
        items[tail] = x;
        if (++tail == items.length)
            tail = 0;
        ++count;
        notEmpty.signal();
    } finally {
        lock.unlock();
    }
}
```

notFull notifies waiting enqueuers when queue goes from full to not full.
Example: Bounded FIFO Queue

```java
public T deq() {
    lock.lock();
    try {
        while (count == 0)
            notEmpty.await();
        T x = items[head];
        if (++head == items.length)
            head = 0;
        --count;
        notFull.signal();
        return x;
    } finally {
        lock.unlock();
    }
}
```

notempty notifies waiting dequeuers when queue goes from empty to nonempty.
The Lost-wakeup Problem

• Condition objects are vulnerable to lost wakeup

• Example:
  – enq() must signal notEmpty each time an item is enqueued
  – otherwise a wakeup signal may be missed and a deq() operation blocks although data is available
The Lost-wakeup Problem - Example

```java
public void enq(T x) {
    lock.lock();
    try {
        while (count == items.length)
            notFull.await();
        items[tail] = x;
        if (++tail == items.length)
            tail = 0;
        ++count;
        if (count == 1)
            notEmpty.signal(); /* WRONG */
    } finally {
        lock.unlock();
    }
}
```
Why is it broken?

- One producer and one consumer
Empty queue, waiting dequeuers
Enqueuer puts item in queue
Since queue was empty, wakes dequeuer

Zzz …

Ouch!

notify() this!
1st Dequeuer slow, overtaken by enqueueers

Zzz …

Must have coffee …
1\textsuperscript{st} Dequeueuer finishes

\textit{Zzz ...}
Why is it broken?

- One producer and one consumer
- A and B both want to dequeue
- Queue is empty → both block on `notEmpty`
- C enqueues an item and wakes A
- Before A's turn, D enqueues another item, but doesn't wake anybody because the queue is not empty
- A dequeues an item, but B does not wake up (lost wakeup)
Minimize Lost Wakeup

Defensive programming practices:
• Always signal (wake) all processes waiting on a condition, not just one
• Specify a timeout when waiting

• Small performance penalty compared to cost of a lost wakeup.
Monitors in Java

- Provided in form of synchronized blocks and methods, as well as:
  - `wait()`
  - `notify()`
  - `notifyAll()`
Example - Call Center Scenario (Java Syntax)

• Calls arrive faster than they can be answered
  – Play recorded message
    • “your call is very important to us …”
  – Put call in queue
    • Play insipid music …
  – Operators dequeue call when ready
Call Queue Implementation

class Queue<T> {
    int head = 0, tail = 0;
    T[] items = new T[QSIZE];
    public T deq() {
        return items[(head++) % QSIZE]
    }
    public T deq() {
        return items[(head++) % QSIZE]
    }

class Queue<T> {
    int head = 0, tail = 0;
    T[] items = new T[QSIZE];
    public T dequeue() {
        return items[(head++) % QSIZE];
    }
    public T enq(T x) {
        items[(tail++) % QSIZE] = x;
    }
}
Works for generic type T
class Queue<T> {
    int head = 0, tail = 0;
    T[] items = new T[QSIZE];
    public enq(T x) {
        items[(tail++) % QSIZE] = x;
    }
    public T deq() {
        return items[(head++) % QSIZE]
    }
}
class Queue<T> {
    int head = 0, tail = 0;
    T[] items = new T[QSIZE];
    public T deq() {
        return items[(head++) % QSIZE]
    }
    public enq(T x) {
        items[(tail++) % QSIZE] = x;
    }
}

next slot to dequeue, 1\textsuperscript{st} empty slot
class Queue<T> {
    int head = 0, tail = 0;
    T[] items = new T[QSIZE];
    public enq(T x) {
        items[(tail++) % QSIZE] = x;
    }
    public T deq() {
        return items[(head++) % QSIZE]
    }
}
Call Queue Implementation

class CallQueue {
    int head = 0, tail = 0;
    call[] calls = new call[QSIZE];
    public enq(call x) {
        calls[(tail++) % QSIZE] = x;
    }
    public call deq() {
        return calls[(head++) % QSIZE];
    }
}
Of course, this doesn’t work
Mutual Exclusion

- Only one thread modifying queue fields at a time
- Use *synchronized* methods
  - Locks object on call
  - Releases lock on return
Mutual Exclusion

head
Synchronized Method

class Queue<T> {
    ...  
    public synchronized enq(T x) {
            items[(tail++) % QSIZE];
    }
    ...
    ...
}
Synchronized Method

class Queue<T> {
    ...
    synchronized public enq(T x) {
        items[(head++) % QSIZE];
    }
    ...
}

Lock acquired on entry, released on exit
Syntactic Sugar

class Queue<T> {
    ...
    public enq(T x) {
        synchronized (this) {
            items[(tail++) % QSIZE];
        }
    }

    ...
}

Syntactic Sugar

class Queue<T> {
    ...
    public enq(T x) {
        synchronized (this) {
            items[(tail++) % QSIZE];
        }
    }...
}

Same meaning, more verbose
Vocabulary

• *A synchronized* method locks the object

• No other thread can call another *synchronized* method for that same object

• Code in middle is critical section
Re-entrant Locks

• What happens if you lock the same object twice?
  – In Java, no deadlock
  – Keeps track of number of times locked and unlocked
  – Unlock occurs when they balance out
Will this work?

class Queue<T> {
    ...
    public synchronized enq(T x) {
        items[(tail++) % QSIZE] = x;
    }
    ...
}
Still Doesn’t Work

class Queue<T> {
    ...
    public synchronized enq(T x) {
        items[(tail++) % QSIZE] = x;
    }
    ...
}

Still Doesn’t Work

class Queue<T> {
    ...
    public synchronized enq(T x) {
        items[(tail++) % QSIZE] = x;
    }
    ...
}
Waiting

• What if
  – Enqueuer finds a full array?
  – Dequeuer finds an empty array?
• Throw an exception?
  – What can caller do?
  – Repeated retries wasteful
• Wait for something to happen
Waiting Synchronized Method

class Queue<T> {
    ...
    public synchronized enq(T x) {
        while (tail - head == QSIZE) {};
        items[(tail++) % QSIZE] = x;
    }
    ...
}
Waiting Synchronized Method

class Queue<T> {
    ...
    public synchronized enq(T x) {
        while (tail - head == QSIZE) {};
        items[(tail++) % QSIZE] = x;
    }
    ...
}

Spin while the array is full
Deadlock

• Enqueueuer is
  – Waiting for a dequeueuer
  – While holding the lock

• Dequeueuer
  – Waiting for enqueueuer to release lock

• Nothing will ever happen
The wait() Method

```java
q.wait();
```

- Releases lock on `q`
- Sleeps (gives up processor)
- Awakens (resumes running)
- Reacquires lock & returns
class Queue<T> {
    ...
    public synchronized enq(T x) {
        while (tail - head == QSIZE) {
            wait();
        };
        items[(head++) % QSIZE] = x;
    }
    ...
}
Waiting Synchronized Method

class Queue<T> {
    ...
    public synchronized enq(T x) {
        while (tail - head == QSIZE) {
            wait();
        }
        items[(head++) % QSIZE] = x;
    }
    ...
}
Waiting Synchronized Method

class Queue<T> {
    ...
    public synchronized enq(T x) {
        while (tail - head == QSIZE) {
            wait();
        }
        items[(head++) % QSIZE] = x;
    }
    ...
}
Wake up!

• When does a waiting thread awaken?
  – Must be notified by another thread
  – when something has happened

• Failure to notify in a timely way is called a “lost wakeup”
The notify() Method

q.notify();

• Awakens one waiting thread
• Which will reacquire lock & returns
The wait() Method

```java
public synchronized enq(call x) {
    while (tail - head == QSIZE) {
        wait();
    }
    calls[(head++) % QSIZE] = x;
    if (tail-head == QSIZE-1) {
        notify();
    }
}
```

Will this work?
public synchronized enq(call x) {
    while (tail - head == QSIZE) {
        wait();
    }
    calls[(head++) % QSIZE] = x;
    if (tail - head == QSIZE - 1) {
        notify();
    }
}

Wait for empty slot
The wait() Method

public synchronized enq(call x) {
    while (tail - head == QSIZE) {
        wait();
    }
    calls[(head++) % QSIZE] = x;
    if (tail - head == QSIZE - 1) {
        notify();
    }
}

Stuff item into array
The wait() Method

```java
public synchronized enq(call x) {
    while (tail - head == QSIZE) {
        wait();
    }
    calls[(head++) % QSIZE] = x;
    if (tail-head == QSIZE-1) {
        notify();
    }
}
```

If the queue was empty, wake up a dequeueuer

Lost-wakeup problem
The notifyAll() Method

q.notifyAll();

- Awakens all waiting threads
- Which will reacquire lock & return
The wait() Method

```java
class Queue {
    private int head, tail;
    private final int QSIZE;
    private final Object[] calls;

    public synchronized void enq(Call x) {
        while (tail - head == QSIZE) {
            wait();
        }
        calls[(head++) % QSIZE] = x;
        if (tail - head == QSIZE - 1) {
            notifyAll();
        }
    }
}
```
public interface ReadWriteLock
{
    Lock readLock();
    Lock writeLock();
}
Readers-Writers Locks

```java
public interface ReadWriteLock {
    Lock readLock();
    Lock writeLock();
}
```

Who needs synchronization? Readers or writers?
Readers-Writers Locks

```java
public interface ReadWriteLock {
    Lock readLock();
    Lock writeLock();
}
```

- A *readers–writers* lock allows multiple readers or a single writer to enter the critical section concurrently.
Readers-Writers Locks

The read and write lock satisfy the following safety properties:

• No thread can acquire the write lock while any thread holds either the write lock or the read lock.
• No thread can acquire the read lock while any thread holds the write lock.
SimpleReadWriteLock

public class SimpleReadWriteLock implements ReadWriteLock {
    int readers;
    boolean writer;
    Lock lock;
    Condition condition;
    Lock readLock, writeLock;
    public SimpleReadWriteLock() {
        writer = false;
        readers = 0;
        lock = new ReentrantLock();
        readLock = new ReadLock();
        writeLock = new WriteLock();
        condition = lock.newCondition();
    }
}
SimpleReadWriteLock

```java
public Lock readLock()
{
    return readLock;
}

public Lock writeLock()
{
    return writeLock;
}
```
SimpleReadWriteLock

private class ReadLock
    implements Lock
{
    public void lock()
    {
        lock.lock();
        try
        {
            while (writer)
                condition.await();

            readers++;
        }
        finally { lock.unlock() }
    }
}
SimpleReadWriteLock

```java
public void unlock()
{
    lock.lock();
    try
    {
        readers--;
        if (readers == 0)
            condition.signalAll();
    }
    finally
    {
        lock.unlock();
    }
}
```
SimpleReadWriteLock

private class WriteLock
{
    public void lock()
    {
        lock.lock();
        try
        {
            while (readers > 0 || writer)
                condition.await();

            writer = true;
        }
        finally { lock.unlock(); }
    }
}
SimpleReadWriteLock

class SimpleReadWriteLock {
    public void unlock()
    {
        writer = false;
        condition.signalAll();
    }
}

Fair Readers-Writers Lock

• SimpleReadWriteLock is correct but is not satisfactory. Why?
• If readers are more frequent than writers then writers can be locked out for long.
reader/writer locks

- Increases concurrency
- When readers and writers both queued up, who gets lock?
  - Favor readers
    - Improves concurrency
    - Can starve writers
  - Favor writers
  - Alternate
    - Avoids starvation
FifoReadWriteLock

```java
public class FifoReadWriteLock implements ReadWriteLock {
    int readAcquires, readReleases;
    boolean writer;
    Lock lock;
    Condition condition;
    Lock readLock, writeLock;
    public FifoReadWriteLock() {
        readAcquires = readReleases = 0;
        writer = false;
        lock = new ReentrantLock();
        condition = lock.newCondition();
        readLock = new ReadLock();
        writeLock = new WriteLock();
    }
}
```
public Lock readLock() {
    return readLock;
}

public Lock writeLock() {
    return writeLock;
}

...
private class ReadLock implements Lock {
    public void lock() {
        lock.lock();
        try {
            readAcquires++;,
            while (writer) {
                condition.await();
            }
        } finally {
            lock.unlock();
        }
    }
}
FifoReadWriteLock

```java
public void unlock() {
    lock.lock();
    try {
        readReleases++;
        if (readAcquires == readReleases)
            condition.signalAll();
    } finally {
        lock.unlock();
    }
}
```
FifoReadWriteLock

```java
private class WriteLock implements Lock {
    public void lock() {
        lock.lock();
        try {
            while (readAcquires != readReleases)
                condition.await();
            writer = true;
        } finally {
            lock.unlock();
        }
    }
    public void unlock() {
        writer = false;
    }
}
```
Re-entrant Lock – Our Version

• Filter lock, Perterson lock, Queue locks and others cannot reacquire a lock: it deadlocks – can arise with nested calls.
• A lock is re-entrant if it can be acquired multiple times by the same thread.
• How do you create a re-entrant lock from a non-reentrant lock?
Re-entrant Lock – Our Version

• In practice, it is provided in `java.util.concurrent.locks`
Re-entrant Lock – Our Version

```java
public class SimpleReentrantLock implements Lock{
    Lock lock;
    Condition condition;
    int owner, holdCount;
    public SimpleReentrantLock() {
        lock = new SimpleLock();
        condition = lock.newCondition();
        owner = 0;
        holdCount = 0;
    }
}
```
Re-entrant Lock – Our Version

```java
public void lock() {
    int me = ThreadID.get();
    lock.lock();
    if (owner == me) {
        holdCount++;
        return;
    }
    while (holdCount != 0) {
        condition.await();
    }
    owner = me;
    holdCount = 1;
}
```
Re-entrant Lock – Our Version

```java
public void unlock() {
    lock.lock();
    try {
        if (holdCount == 0 || owner != ThreadID.get())
            throw new IllegalMonitorStateException();
        holdCount--;
        if (holdCount == 0) {
            condition.signal();
        }
    } finally {
        lock.unlock();
    }
}
```
Semaphores

- Mutual exclusion lock guarantees only one thread enter CS.
- If another thread wants to enter the CS, it blocks, and suspends itself till another thread notifies it to try again.
Semaphores

- Semaphores are one of the earliest forms of synchronization.
- A semaphore is a generalization of mutual exclusion locks.
- Instead of allowing one thread at a time to the critical section, a semaphore allows at most \( c \) threads, where \( c \) is the capacity of the semaphore determined when initialized.
Semaphores

A semaphore is a variable that controls access:
• by multiple threads
• to one or more resources.
A semaphore consists of:
- an initial value
- a wait() operation
- a signal() operation.
Semaphores

A semaphore consists of:
• an initial value
• a wait() operation: decrement, block until semaphore is open
• a signal() operation: increment, allow another thread to enter
Semaphores - Operation
Semaphores - Operation

![Diagram of semaphore operation with '1' and 'wait()' labels]
Semaphores - Operation

![Diagram of wait() operation with a semaphore value of 0]
Semaphores - Operation
Semaphores - Operation

![Diagram of semaphore operation]

- `signal()`: Increases the semaphore value.
- `wait()`: Decreases the semaphore value and blocks the process if it becomes negative.

Semaphore value: 0
Semaphores - Operation
Semaphores - Analogy

Consider a library that provides rooms in which students can study:

• Students request rooms at the front desk.
• The librarian at the front desk maintains a count of how many rooms are free.
• Each time a room is requested, the count is decreased and the student may use a room.
Semaphores - Limitations

- Semaphores do not keep track of which resources are used - only how many are used.
- Without an additional protocol, for instance, to enforce fairness, the system will be susceptible to starvation and even deadlock.
- A thread can misbehave by:
  - Using a resource without requesting it.
  - Asymmetrical requests and releases.
  - Holding a resource without using it.
Semaphore and Locks (Mutex)

• A binary semaphore is essentially a mutex except for the following, key differences:
  – A mutex implements the concept of an owner.
  – A mutex provides deletion safety.
  – A mutex provides priority inversion safety.
Semaphore - Implementation

```java
public class Semaphore {
    final int capacity;
    int state;
    Lock lock;
    Condition condition;
    public Semaphore(int c) {
        capacity = c;
        state = 0;
        lock = new ReentrantLock();
        condition = lock.newCondition();
    }
}```
Semaphore - Implementation

```java
public void acquire() {
    lock.lock();
    try {
        while (state == capacity) {
            condition.await();
        }
        state++;
    } finally {
        lock.unlock();
    }
}
```
Semaphore - Implementation

```java
public void release() {
    lock.lock();
    try {
        state--;
        condition.signalAll();
    } finally {
        lock.unlock();
    }
}
```
Semaphore - Applications

Producer-Consumer; Consider a queue that is modified by producers and consumers:

• A count of **empty spaces** is kept for producers.
• A count of **occupied spaces** is kept for consumers.
• When there are no occupied spaces, consumers block.
• 4. When there are no empty spaces, producers block.
Semaphore - Applications

Locks; Consider the previously described queue with producers and consumers:

• 1. A binary semaphore is decremented each time that either a producer or a consumer intends to modify the queue.

• 2. While the semaphore is at zero, other threads block.

• 3. When a thread finishes modifying the queue, it increments the semaphore, unblocking one other thread.