Busy-Waiting

Recall the consumer and producer threads:

- Producer thread:

```
for (;;) {
  synchronized (q) {
    q.queue(10);
  }
}
```

- Consumer thread:

```
for (;;) {
  synchronized (q) {
    if (!q.empty()) q.unqueue();
  }
}
```

Problem: consumer hogs computer while waiting.

This is called *busy-waiting*.

Want to block a thread if it waits for something.

Towards a Solution

Assume a complementary pair of statements:

- *wake*: wakes up [all] waiting threads
- *wait*: waits until waken up

Producer thread:

```
for (;;) {
  synchronized (q) {
    q.queue(10);
    wake;
  }
}
```

But consumer is harder to write.

Non-Solutions

Wrong consumer solutions:

- Deadlock:

```
for (;;) {
  synchronized (q) {
    if (!q.empty()) q.unqueue();
    wait;
  }
}
```

  While waiting, the lock on q is not released.

- Race condition:

```
for (;;) {
  synchronized (q) {
    if (!q.empty()) q.unqueue();
  }
  wait;
}
```

  If producer calls *wake* just before consumer calls *wait*, consumer will miss it, then wait indefinitely.

Ideal Solution

The deadlock case will work if *wait*:

1. *atomically* releases lock and starts waiting
2. waits until waken up
3. re-gains lock

This will then work:

```
for (;;) {
  synchronized (q) {
    if (!q.empty()) q.unqueue();
    wait;  // releases lock before blocking
  }
}
```

Java's Wait and Notify

Each object `x` has these methods (they are from `Object`):

- `public void notify()`

  Wakes up *one* thread that is waiting on `x`.

- `public void notifyAll()`

  Wakes up *all* threads that are waiting on `x`.

- `public void wait() throws InterruptedException`

  Releases lock of `x`, then wait on `x` until waken, then re-gains lock of `x`

If you call these methods on object `x`, you must first get the lock of `x`. (This eliminates race conditions as we have seen.)

Solutions in Wait and Notify

Producer:

```
for (;;) {
  synchronized (q) {
    q.queue(10);
    q.notifyAll();
  }
}
```

Consumer:

```
try {
  for (;;) {
    synchronized (q) {
      if (!q.empty()) q.unqueue();
      q.wait();
    }
  }
}
catch (InterruptedException e) {}
```


Wait and Notify Are Per Object

What is meant by "wait on `x`"? Isn't "wait" just wait?

If `x` and `y` are different objects, calling `x.wait()` and calling `y.wait()` are different. Consider two threads:

- thread 1:

```
synchronized (x) {
  ...
  x.wait();
  ...
}
```

- thread 2:

```
synchronized (y) {
  ...
  y.wait();
  ...
}
```

Calling `x.notifyAll()` will only wake up thread 1.

Unbounded Buffer Example

Collecting the above ideas into a class.

A thread can call `send` to send an item.

Another thread can call `recv` to wait and get an item.

```
public class UnboundedBuffer {
  private IntQueue q;

  public UnboundedBuffer() {
    q = new IntQueue();
  }

  public synchronized send(int m) {
    q.queue(m);
    notifyAll();  // could be notify
  }

  public synchronized int recv() throws
      InterruptedException {
    while (q.empty()) wait();
    return q.unqueue();
  }
}
```

Bounded Buffer Example

```
public class BoundedBuffer {
  private IntQueue q;
  private int bound;

  public BoundedBuffer(int b) {
    q = new IntQueue();
    bound = b;
  }

  public synchronized send(int m) throws
  InterruptedException {
    while (q.length() == bound) wait();
    q.queue(m);
    notifyAll();
  }

  public synchronized int recv() throws
  InterruptedException {
    while (q.empty()) wait();
    notifyAll();
    return q.unqueue();
  }
}
```