10/18/23

Monitors

(block/unblock)

→ design pattern for synchronizing threads based on events & conditions

→ condition, condition variable, lock

→ cv_wait, cv_signal, cv_broadcast

xk ⇓ sleep(chan, lk) × ⇓ wakeup(chan)

Basic Pattern

```
function() {
  lock.acquire();
  while (!condition){
     cv_wait(lock);
  }
  // consume condition
  lock.release();
}
```

within cv_wait:
① atomically releases the lock & blocks,
② reacquires the lock upon a signal

```
function() {
  lock.acquire();
  // update condition
  cv_signal();
  lock.release();
}
```

Synchronization Problem: Coffee

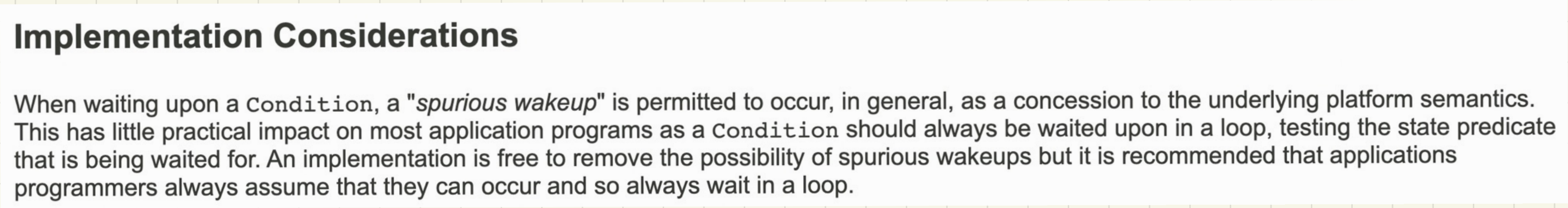
"Spurious wakeups"

```
int coffee=0;
condvar coffee_cv;
lock lk;
```

wait in a while loop b/c the condition might not be true anymore when we wake up.

```
function get_coffee() {
  lk.acquire();
  while (coffee==0){
      coffee_cv.wait(lk);
  }
    coffee--;
    lk.release();
}
```

```
function make_coffee() {
    lk.acquire();
    coffee++;
    coffee_cv.signal();
    lk.release();
}
```



Implementation Considerations

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. An implementation is free to remove the possibility of spurious wakeups but it is recommended that applications programmers always assume that they can occur and so always wait in a loop.

Synchronization problem: making breakfast pancake

Starter code:

```
Lock lk;
Condvar breakfast_cv;
int pancake = 0;
int berries = 0;
```

```
// needs 1 pancake & 2 berries for breakfast
function plate_breakfast() { }

// produce 1 pancake at a time
function make_pancake() { }

// refill 10 berries at a time
function refill_berries() { }
```

```
Lock lk;
Condvar breakfast_cv;
int pancake =0;
int berries =0;


function plate_breakfast(){
   lk.acquire();
   while (pancake <1 || berries<2){
      breakfast_cv.wait(lk);
   }
   // assert(!pancake<1 && !berries<2);
              pancake>=1     berries>=2

    pancake --;
    berries -=2;
    lk.release();
  }
```

```
make_pancake(){
   lk.acquire();
   pancake ++;
    breakfast_cv.signal();
     lk.release();
}

                                   ↑
   // similar code for berries
```

Bounded Buffer Problem

fixed size buffer : **N elements**

Producer: produce item and put into an empty slot, blocks if no room to put item (buffer is full)

Consumer: consume item from a slot, blocks if no item to consume

starter code

```
char buffer[N];
int consume_ofs = 0; // consumer reads here
int produce_ofs = 0; // producer writes here
int count = 0; // # of items in the buffer.
```

```
function produce(item)
```

```
// returns consumed item
function consume()
```