Monitors

\[ \rightarrow \text{design pattern for synchronizing threads based on events \\& conditions} \]

\[ \rightarrow \text{condition, condition variable, lock} \]

\[ \rightarrow \text{cv-wait, cv-signal, cv-broadcast} \]

\[ \times \downarrow \]

sleep(chan, 1k) \[ \times \uparrow \]

wakeup(chan)

Basic Pattern

function() {
  lock.acquire();
  while (!condition) {
    cv-wait(lock);
  }
  // consume condition
  lock.release();
}

function() {
  lock.acquire();
  // update condition
  cv-signal(lock);
  lock.release();
}
Synchronization Problem: Coffee

```c
int coffee = 0;
condvar coffee-cv;
lock lk;

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

Wait in a while loop blo the condition might not be true anymore when we wake up.

```c
unlock lk acquire();
coffee++;
coffee-wait(lk);
}```
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 pancakes

Starter code:

```c
// Lock
lock lk;

// Condvar breakfast[C0];

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;
lock breakfast_cuv;

int pancake = 0;
int berries = 0;


int pancake = 0;
int berries = 0;

function plate_breakfast()

lk.acquire();

while (pancake < 1 || berries < 2)

breakfast_cuv.wait(lk);

lk.acquire();

3

{11 assert (!pancake < 1 && !berries < 2);

pancake --;

berries -= 2;

lk.release();

lk.release();

make_pancake();

lk.acquire();

pancake++;  

breakfast_cuv.signal();

lk.release();

3

ll 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

```c
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

// Returns consumed item
function produce(item)
function consume()
```