# Condition Variables & Monitors

Godmar Back

Virginia Tech

October 10, 2024

# Outline

- Condition variables provide another way of signaling between threads with the goal of allowing one thread to signal one or more others.
- Condition variables are easily one of the least understood mechanisms in multithreaded programming.
  - E.g., the most common question is: what mutex is passed to `pthread_cond_wait()` and why?
- Condition variables (along with mutexes) are *building blocks* of a larger abstraction called a *monitor*
  - In C, this abstraction must be implemented as an idiomatic pattern.

## Approach

To understand condition variables, we must understand the ideas behind monitors first.

VIRGINIA TECH

# Monitors

## Key Insight

When threads interact, they usually share both state and a need to signal each other regarding updates to this state. Updating and signaling are thus interlinked.

- Monitors combine a set of shared variables and operations on them
- Think of a properly encapsulated Java class: all fields are private and are accessed only via public methods
- A monitor is associated with a single mutex that is acquired and released upon entry & exit; thus methods form a critical section
- A monitor may use one or more signaling queues

VIRGINIA TECH

# Infinite Buffer implemented in a Monitor

```
monitor buffer {
  // implied: struct lock mlock;
private:
  char buffer[];
  int head, tail;
public:
  produce(item);
  item consume();
}
```

```
buffer::produce(item i)
{  // try { mlock.acquire()
  buffer[head++] = i;
  // } finally { mlock.release() }
}

buffer::consume()
{  // try { mlock.acquire()
  return buffer[tail++];
  // } finally { mlock.release() }
}
```

This (hypothetical) example shows the mutual exclusion facility that would be added by a classic monitor. Each instance is given an implicit (hidden) lock that is acquired upon method entry and released on all paths leaving a method.

VIRGINIA TECH

# Bounded Buffer implemented in a Monitor

- A bounded buffer may be empty or full. Thus, it is necessary for producers and consumers to coordinate and make sure that
  - consumers learn when a buffer switches from empty to non-empty
  - producers learn when a buffer switches from full to non-full
- As usual, we want a solution that is correct, efficient, and does not introduce unnecessary delay
- Solution: condition variables

VIRGINIA TECH

# Condition Variable

- Are signaling queues that support 3 operations:
  - Wait(): add current thread to the queue and enter BLOCKED state
  - Signal(): if any threads are on queue, remove first thread and make it READY
  - Broadcast(): remove all threads from the queue and make them READY

  Note that Signal() and Broadcast() have no effect if no threads are in the queue
- Misnomer: the condition variable doesn't represent any "condition" as such. Conditions are expressed as a general boolean predicate over the monitor's state: e.g. "is buffer empty?"

VIRGINIA TECH

# Finite Buffer with Condition Variables

```
monitor buffer {
  condition items_avail;
  condition slots_avail;

private:
  char buffer[];
  int head, tail;

public:
  produce(item);
  item consume();
}
```

```
buffer::produce(item i)
{ /* wait while buffer is full */
  while ((tail+1-head)%CAPACITY == 0)
     slots_avail.wait();

  buffer[head++] = i;
  items_avail.signal();
}

buffer::consume()
{  /* wait while buffer is empty */
   while (head == tail)
     items_avail.wait();

   item i = buffer[tail++];
   slots_avail.signal();
   return i;
}
```

Subtle requirement: to avoid deadlock, threads waiting on a condition variable must leave the monitor (e.g. release its lock).

VIRGINIA TECH

# Condition Variable – Wait Operation

- The Wait operation involves 3 steps
  - Release monitor lock
  - Add current thread to queue and enter BLOCKED state
  - (Once unblocked as a result of a Signal or Broadcast operation), reacquire monitor lock

  The release + block steps are atomic, that is, no other thread will be able to acquire the lock before the thread was added to the queue. This ensures that no wakeups are lost.
- But the unblock and reacquisition steps are not: a woken-up thread reentering the monitor is not given preference when reentering the monitor compared to threads entering elsewhere.

VIRGINIA TECH

# Condition Variable – Rechecking the condition after Wait()

The condition that caused a thread to wait must be rechecked after returning from Wait() – this implies a `while` loop, because:

1. Other threads may be acquiring the lock before the thread returning from Wait(). Example
   - Thread A (consumer) calls `consume()`, queue is empty, calls Wait() and blocks.
   - Thread B (producer) call `produce()`, adds item, signals. Thread A is woken up (READY), but has not reacquired lock.
   - Thread C (consumer) calls `consume()`
   - Thread C acquires monitor lock first, removes item produced by B
   - Thread A gets the monitor lock, returns from Wait() but no item is in the queue. Correct action now is to recheck and call Wait() again.
2. Spurious wakeups are allowed by the specification.

VIRGINIA TECH

## IEEE Std. 1003.1

When using condition variables there is always a Boolean predicate involving shared variables associated with each condition wait that is true if the thread should proceed. Spurious wakeups from the `pthread_cond_timedwait()` or `pthread_cond_wait()` functions may occur. Since the return from `pthread_cond_timedwait()` or `pthread_cond_wait()` does not imply anything about the value of this predicate, the predicate should be re-evaluated upon such return.

## POSIX.1 2008

An added benefit of allowing spurious wakeups is that applications are forced to code a predicate-testing-loop around the condition wait. This also makes the application tolerate superfluous condition broadcasts or signals on the same condition variable that may be coded in some other part of the application. The resulting applications are thus more robust. Therefore, POSIX.1-2008 explicitly documents that spurious wakeups may occur.

# Bounded Buffer (working C implementation)

```c
#define CAPACITY 10
// internal bounded buffer state
static int buffer[CAPACITY];
static int tail, head;

// protects the monitor: buffer, head, tail
static pthread_mutex_t buffer_lock = PTHREAD_MUTEX_INITIALIZER;

// items available for consumption
static pthread_cond_t items_avail = PTHREAD_COND_INITIALIZER;

// buffer slots available for production
static pthread_cond_t slots_avail = PTHREAD_COND_INITIALIZER;

void produce(int item);

void consume();
```

VIRGINIA TECH

# Bounded Buffer, cont'd

```
void produce(int item)
{
    pthread_mutex_lock(&buffer_lock);        // Enter monitor
    while ((tail + 1 - head) % CAPACITY == 0)
        // Add calling thread to 'slots_avail' queue
        // Leave monitor (release buffer_lock)
        // Block
        // Enter monitor (acquire buffer_lock)
        pthread_cond_wait(&slots_avail, &buffer_lock);
    // invariant: (a) buffer_lock is held
    //            (b) slot is available
    printf("thread %p: produces item %d\n", (void *)pthread_self(), item);
    buffer[head] = item;                     // update buffer state
    head = (head + 1) % CAPACITY;
    pthread_cond_signal(&items_avail);       // wake up consumer (if any)
    pthread_mutex_unlock(&buffer_lock);      // Leave monitor
}
```

VIRGINIA TECH

# Bounded Buffer, cont'd

```
void consume()
{
    pthread_mutex_lock(&buffer_lock);       // Enter monitor
    while (head == tail)
        // Add thread to `items_avail` queue
        // Leave monitor (release buffer_lock)
        // Block
        // Enter monitor (acquire buffer_lock)
        pthread_cond_wait(&items_avail, &buffer_lock);
    // invariant: (a) buffer_lock is held
    //            (b) item is available
    int item = buffer[tail];
    tail = (tail + 1) % CAPACITY;
    printf("thread %p: consumes item %d\n", (void *)pthread_self(), item);
    pthread_cond_signal(&slots_avail);      // wake up producer (if any)
    pthread_mutex_unlock(&buffer_lock);     // Leave monitor
}
```

VIRGINIA TECH

# Comparison to Semaphores

Semaphores

- Signal()/Post() are remembered even when no thread is currently blocked in Wait()
- Wait() may or may not block the calling thread
- Use when
  - number of waits()/posts() matches
  - for one-off "rendezvous"
- Require separate measures for safe access to shared state

Condition Variables

- Signal() has no effect when no thread is currently blocked in Wait()
- Wait() always blocks the calling thread
- Must be used in conjunction with the lock protecting the shared state that may change
- Use when
  - Coordinating about arbitrary/complex state changes

VIRGINIA TECH

# History

- Classic monitors were invented by C.A.R. Hoare & Per Brinch-Hansen in 1972/73 [3]. If integrated into a programming language, they provide "safe" parallelism: the programmer cannot forget to lock variables before accessing them - the compiler enforces this
- Condition variables were used in the Mesa/Cedar System @ Xerox PARC 1978
- Later influenced the design of Java/C# (albeit without the safety [2])
- Further reading: Arpaci-Dusseau 2018 [1]

VIRGINIA TECH

# Monitors/Condition Variables in Java

- Every Java object can be used as a monitor with exactly one condition variable (via `java.lang.Object.*` methods `wait()` and `notify()`, `notifyAll()`.)
- `synchronized` marks methods that enter the monitor.
- In hindsight, not a good design choice: inflexible, and imposes huge implementation cost on JVM.

## Java Monitor Example

```
class buffer {
  private char buffer[];
  private int head, tail;
  public synchronized produce(item i) {
     while (buffer_full())
          this.wait();
     buffer[head++] = i;
     this.notifyAll();
  }
  public synchronized item consume() {
     while (buffer_empty())
          this.wait();
     i = buffer[tail++];
     this.notifyAll();
     return ;
  }
}
```

VIRGINIA TECH

# Condition Variables in java.util.concurrent

- `java.util.concurrent.*` provides more flexible means to use the monitor pattern.
- Multiple condition variables may be associated with a lock.
- Unfortunately, syntactic support for monitor blocks was then lost, requiring calls to `lock()` and `unlock()` in `try/finally`.

```
import java.util.concurrent.locks.*;
class buffer {
  private ReentrantLock monitorlock
    = new ReentrantLock();
  private Condition items_available
    = monitorlock.newCondition();
  private Condition slots_available
    = monitorlock.newCondition();
  public /* NO SYNCHRONIZED here */
     void produce(item i) {
      monitorlock.lock();
      try {
       while (buffer_full())
          slots_available.await();
       buffer[head++] = i;
       items_available.signal();
      } finally {
        monitorlock.unlock();
      }
   } /* consume analogous */
}
```

VIRGINIA TECH

# References


[1] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau.
*Operating Systems: Three Easy Pieces*.
Arpaci-Dusseau Books, 1.00 edition, August 2018.

[2] Per Brinch Hansen.
Java's insecure parallelism.
*SIGPLAN Not.*, 34(4):38–45, April 1999.

[3] C.A.R. Hoare.
Monitors: An operating system structuring concept.
*Communications of the ACM*, 17:549–557, October 1974.


VIRGINIA TECH