Operating System Concepts

Lecture 19: Synchronization Primitives — Part 2

Omid Ardakanian
oardakan@ualberta.ca
University of Alberta

Example of conditional synchronization

Condition variables are used with a mutex lock and in a loop to check the condition

```
Class CokeMachine{
    …
    storage for cokes (buffer) of size n
    Lock lock;
    int count = 0;
    Condition notFull, notEmpty;
}

CokeMachine::Deposit(){
    lock->acquire( );  // entering the critical section
    while(count == n)
       notFull.wait(&lock); // release lock before blocking; reacquire when waking up
    add coke to the machine;
    count++;
    notEmpty.notify();
    lock->release();
}

CokeMachine::Remove(){
    lock->acquire();  // entering the critical section
    while(count == 0)
       notEmpty.wait(&lock); // release lock before blocking; reacquire when waking up
    remove coke from the machine;
    count--;
    notFull.notify(); // always hold a lock while signalling to avoid a race condition
    lock->release();
}
```

Today's class

- Synchronization primitives
  - Semaphore
  - Monitor

Semaphores

- Generalized locks invented by Dijkstra in 1965
  - they are a (non-negative) integer variable that supports two **atomic** operations: `wait` and `signal` (or `down` and `up`)
- **Binary semaphore**: used for mutual exclusion (just like mutex locks)
  - guarantees mutually exclusive access to a shared resource, i.e., only one thread in the critical section at a time
  - value can change from 0 to 1 and vice versa; it is initialized to free (value = 1)
- **Counting semaphore**: used for conditional synchronization
  - useful when multiple instances of a resource are available
  - counting semaphore is usually initialized to the number of instances of a resource that are available; a thread can enter the critical section to access resources as long as at least one instance is available

Atomic operations with semaphores

- Each semaphore supports a queue of processes waiting to access the critical section (e.g., to check/buy milk)
- If a thread executes `Wait( )` and the semaphore is free (non-zero), it continues executing after **decrementing** the semaphore's variable. But if it is not free, OS puts the thread on the wait queue for that semaphore
- `Signal( )` unblocks **one thread** on the semaphore's wait queue (if any) after **incrementing** the semaphore's variable

Using semaphores to implement mutual exclusion

```
Semaphore milksemaphore; // suppose it's initialized to 1

milksemaphore.Wait()
<critical section>
milksemaphore.Signal()
```

acquired before accessing shared data

released after accessing shared data

Implementing signal and wait by disabling interrupts

```
class Semaphore {
public:
  void Wait();
  void Signal();
private:
  int value;
  Queue Q;
}

Semaphore::Semaphore(int val) {
  value = val; // initialized to the number of available resources
}

Semaphore::Wait() {
  intr_disable();
  value = value - 1;
  if(value < 0) {    // |value| is the number of waiting threads
    queue_add(Q, gettid());
    thread_block();
  }
  intr_enable();
}

Semaphore::Signal() {
  intr_disable();
  value = value + 1;
  if(value <= 0)     // if there is a waiting thread
    thread_unblock(queue_remove(Q));
  intr_enable();
}
```

Example: counting semaphore

	Semaphore value	Semaphore queue	Thread 1's state	Thread 2's state
	2	empty		
`Thread 1: SM.wait()`	1	empty		
`Thread 2: SM.wait()`	0	empty		
`Thread 1: SM.wait()`	-1	T1	waiting	
`Thread 2: SM.signal()`	0	empty		
`Thread 1: SM.signal()`	1	empty		
`Thread 1: SM.signal()`	2	empty		

Example: counting semaphore

Suppose each thread needs two units of the resource

	Semaphore value	Semaphore queue	Thread 1's state	Thread 2's state
	2	empty		
`Thread 1: SM.wait()`	1	empty		
`Thread 2: SM.wait()`	0	empty		
`Thread 1: SM.wait()`	-1	T1	waiting	
`Thread 2: SM.wait()`	-2	T1, T2	waiting	waiting

Deadlock

Example: counting semaphore

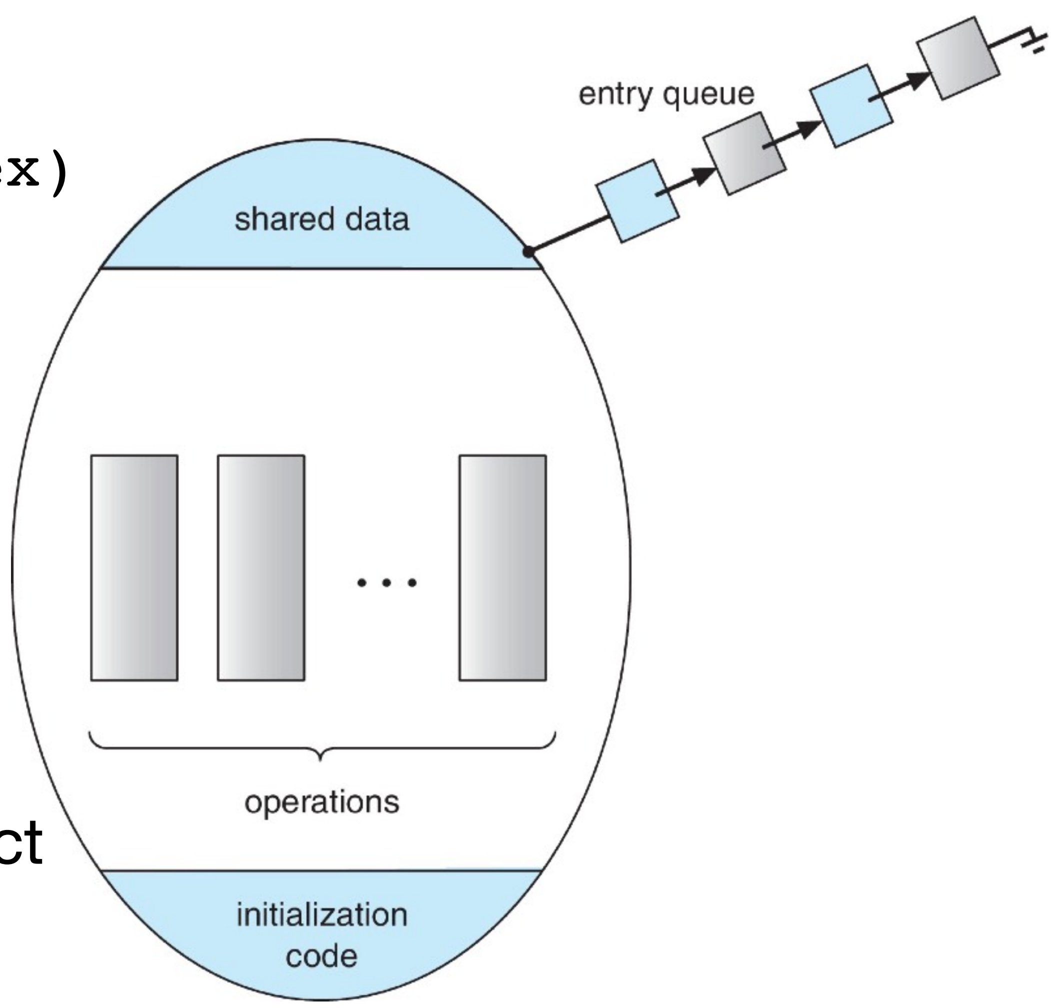
	Semaphore value	Semaphore queue	Thread 1's state	Thread 2's state	Thread 3's state
	2	empty			
`Thread 1: SM.wait()`	1	empty			
`Thread 1: SM.wait()`	0	empty			
`Thread 2: SM.wait()`	-1	T2		waiting	
`Thread 3: SM.wait()`	-2	T2, T3		waiting	waiting
`Thread 1: SM.signal()`	-1	T3			waiting
`Thread 1: SM.signal()`	0	empty			
`Thread 3: SM.signal()`	1	empty			
`Thread 2: SM.signal()`	2	empty			

Semaphores versus condition variables

- Semaphore can be implemented using a variable (counter), a condition variable, and a mutex lock
- Condition variables are **memoryless**, but semaphores have memory
  - on a condition.signal if no one is waiting, nothing happens
    - if a thread then does a condition.wait, it will wait
  - on a semaphore.signal if no one is waiting, the value of the semaphore is incremented
    - if a thread then does a semaphore.wait, then value is decremented and the thread continues
- Thus semaphore's `wait` and `signal` are commutative, i.e., the result is the same regardless of the order of execution. Condition variables are not, and as a result they must be in a critical section to access state variables and do their job

We have to be careful when using semaphores

- Semaphore operations can be used incorrectly by programmers → two threads may be in their critical sections simultaneously or may permanently block!
  - `signal(mutex)` … `wait(mutex)`
  - `wait(mutex)` … `wait(mutex)`
  - omitting `wait(mutex)` and/or `signal(mutex)`



Solution? a higher-level synchronization construct

Monitors

- A **thread-safe class** that ties (private) data and methods (including synchronization operations) together, introduced by Per Brinch Hansen in 1970s
  - guarantees mutual exclusion, i.e., only one thread may run a given monitor **method** at a time
  - provides a mechanism for threads to temporarily give up exclusive access to wait for a certain condition
- It defines a **lock** and zero, one, or more **condition variables** for managing concurrent access to shared data
  - the lock ensures mutual exclusion
    - used to ensure that only a single thread is active in the monitor at a time
  - condition variables allow threads to go to sleep **inside a critical section**, by releasing their lock at the same time they are put to sleep
    - used when an operation cannot complete (because the condition is not true)
- Many programming languages, such as C# and Java, support the notion of Monitor
  - synchronized class methods in Java; Monitor.Enter, Monitor.Exit, Monitor.Wait, Monitor.Pulse in C#

Revisiting the CokeMachine class

```
Class CokeMachine{
    …
    storage for cokes (buffer) of size n
    Lock lock;                        // a shared lock
    int count = 0;
    Condition notFull, notEmpty;
}

CokeMachine::Deposit(){
    lock->acquire();
    while(count == n)
       notFull.wait(&lock);
    add coke to the machine;
    count++;
    notEmpty.notify();
    lock->release();
}

CokeMachine::Remove(){
    lock->acquire();
    while(count == 0)
       notEmpty.wait(&lock);
    remove coke from the machine;
    count--;
    notFull.notify();
    lock->release();
}
```

What if a thread frees a resource needed by a waiting thread?

- Should the waiting thread be immediately awakened or the signalling thread finish first?
  - this gives rises to different versions of monitor semantics

```
CokeMachine::Deposit(){
    lock->acquire( );
    while(count == n)
       notFull.wait(&lock);
    add coke to the machine;
    count++;
    notEmpty.notify();
    lock->release();
}

CokeMachine::Remove(){
    lock->acquire();
    while(count == 0)
       notEmpty.wait(&lock);
    remove coke from the machine;
    count--;
    notFull.notify();
    lock->release();
}
```

thread A is put to sleep after releasing the lock (points to `notFull.wait(&lock);`)

thread B removes a coke so thread A can resume execution now (points to `notFull.notify();`)

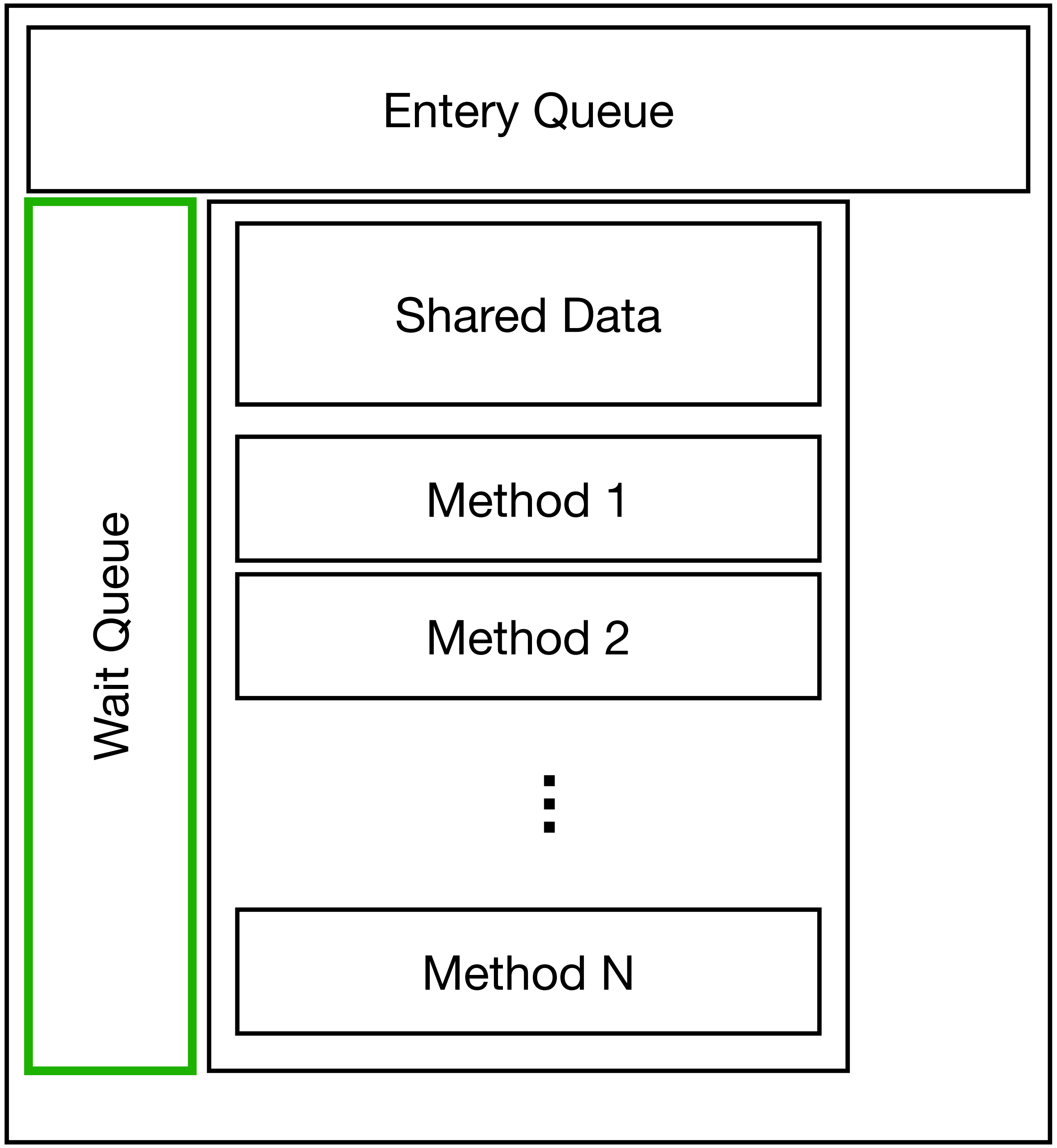
Two types of monitors

- Mesa-style monitors (used in most real operating systems)
  - signal puts a waiting thread on ready queue (with no special priority)
    but the signalling thread keeps the lock and thus the processor
    - some other thread could grab the lock before the waiting thread gets to run
    - hence, the waiting thread may have to wait again after it is woken up (the condition may not be satisfied at that time)!
- Hoare-style monitors (not commonly used but presented in some textbooks)
  - signalling thread gives **the processor and the lock** to the waiting thread which should immediately execute
  - when the waiting finishes or waits again, the processor and the lock are returned to the signalling thread
    - so the signalling thread should be kept in another queue known as the signal queue

Monitor implementation with semaphores

- An entry queue (binary semaphore) for the entire class
- A wait queue for every counting semaphore defined inside the class

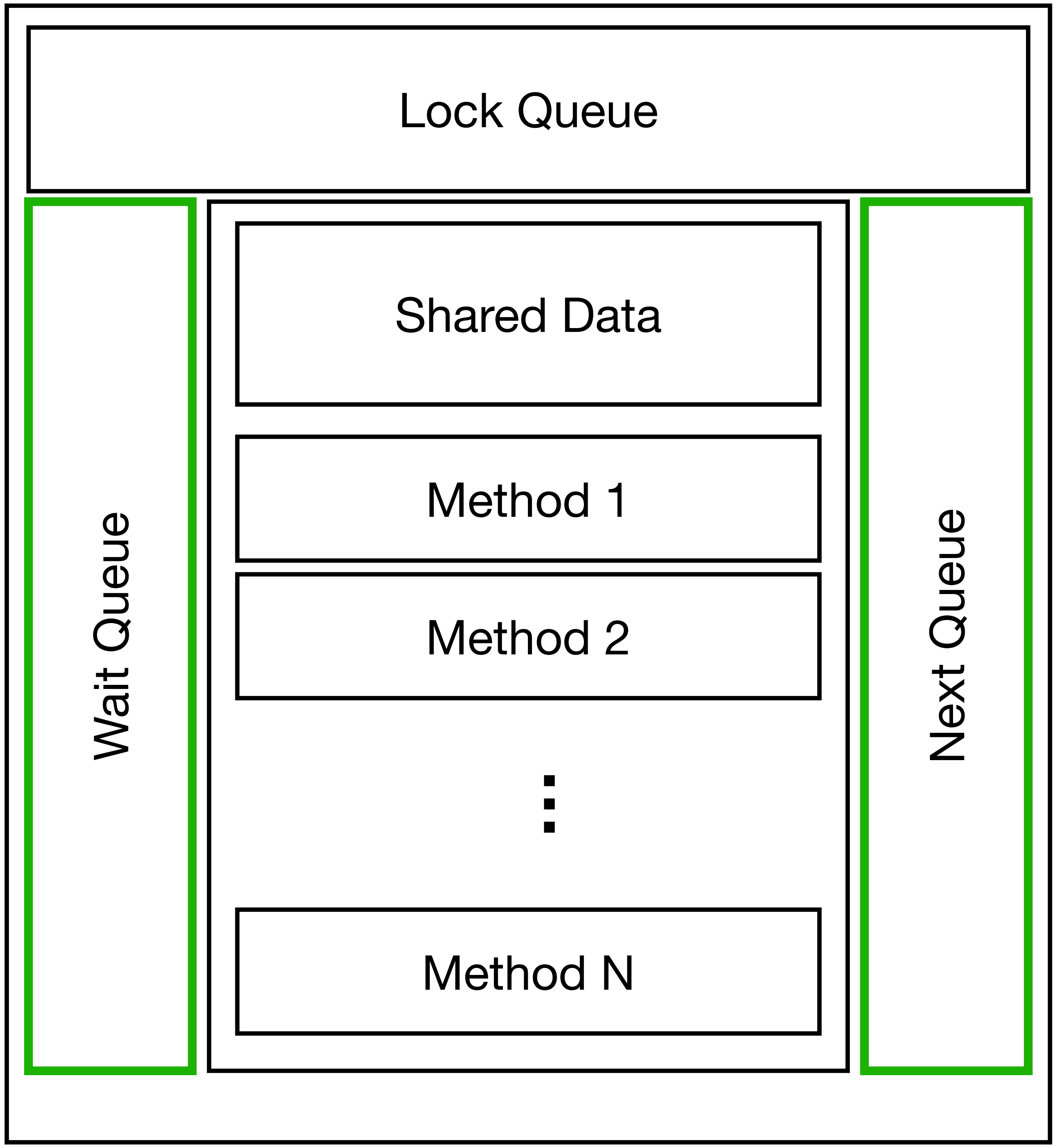
Mesa-style monitor (signal and continue)



Monitor implementation with semaphores

- A lock queue (binary semaphore) for the entire class
- A wait queue for every counting semaphore defined inside the class
- A next queue for every counting semaphore defined inside the class

Hoare-style monitor (signal and wait)



Implications

- With Mesa semantics, signal is a **hint** that the condition may be true so you always have to check the condition after waking up (should be done in a while loop)
  - efficient implementation but can lead to concurrency bugs if the condition is not checked again
- With Hoare semantics, you can assume that the condition holds after waking up (‘while’ can be replaced by ‘if’)
  - inefficient and much more complicated, but leads to a nicer proof of correctness

```
CokeMachine::Deposit(){
     lock->acquire( );
     while(count == n)
        notFull.wait(&lock);
     add coke to the machine;
     count++;
     notEmpty.notify();
     lock->release();
}
```

```
CokeMachine::Deposit(){
     lock->acquire( );
     if(count == n)
        notFull.wait(&lock);
     add coke to the machine;
     count++;
     notEmpty.notify();
     lock->release();
}
```

EXAMPLES OF MESA-STYLE MONITOR

Monitor in the Producer-Consumer problem

- use a binary semaphore for mutual exclusion
- use counting semaphores **for each constraint**
- putting them together:

```
Semaphore mutex;
Semaphore full_buffer;
Semaphore empty_buffer;
```

Monitor implementation with semaphores

```
class BoundedBuffer {
  public:
    void Producer();
    void Consumer();
  private:
    /* shared data */
    Items buffer;
    int last, count;
    /* shared data */

    Semaphore mutex; // control access to buffers
    Semaphore empty; // number of free slots
    Semaphore full;  // number of used slots
}

BoundedBuffer::BoundedBuffer(int N){
  mutex.value = 1;   // initially free
  empty.value = N;   // initially all slots are empty
  full.value  = 0;   // initially all slots are empty

  buffer = new Items[N];
  last = 0;
  count = 0;
}
```

Monitor implementation with semaphores

```
BoundedBuffer::Producer(){
  <produce item>
  empty.Wait();   // one fewer full slot, or wait
  mutex.Wait();   // entering critical section to access buffer
  <add item to buffer>
  mutex.Signal(); // leaving critical section
  full.Signal();  // one more used slot
}

BoundedBuffer::Consumer(){
  full.Wait();    // one fewer full slot, or wait
  mutex.Wait();   // entering critical section to access buffer
  <remove item from buffer>
  mutex.Signal(); // leaving critical section
  empty.Signal(); // one more free slot
}
```

What if there are multiple producers and/or consumers?

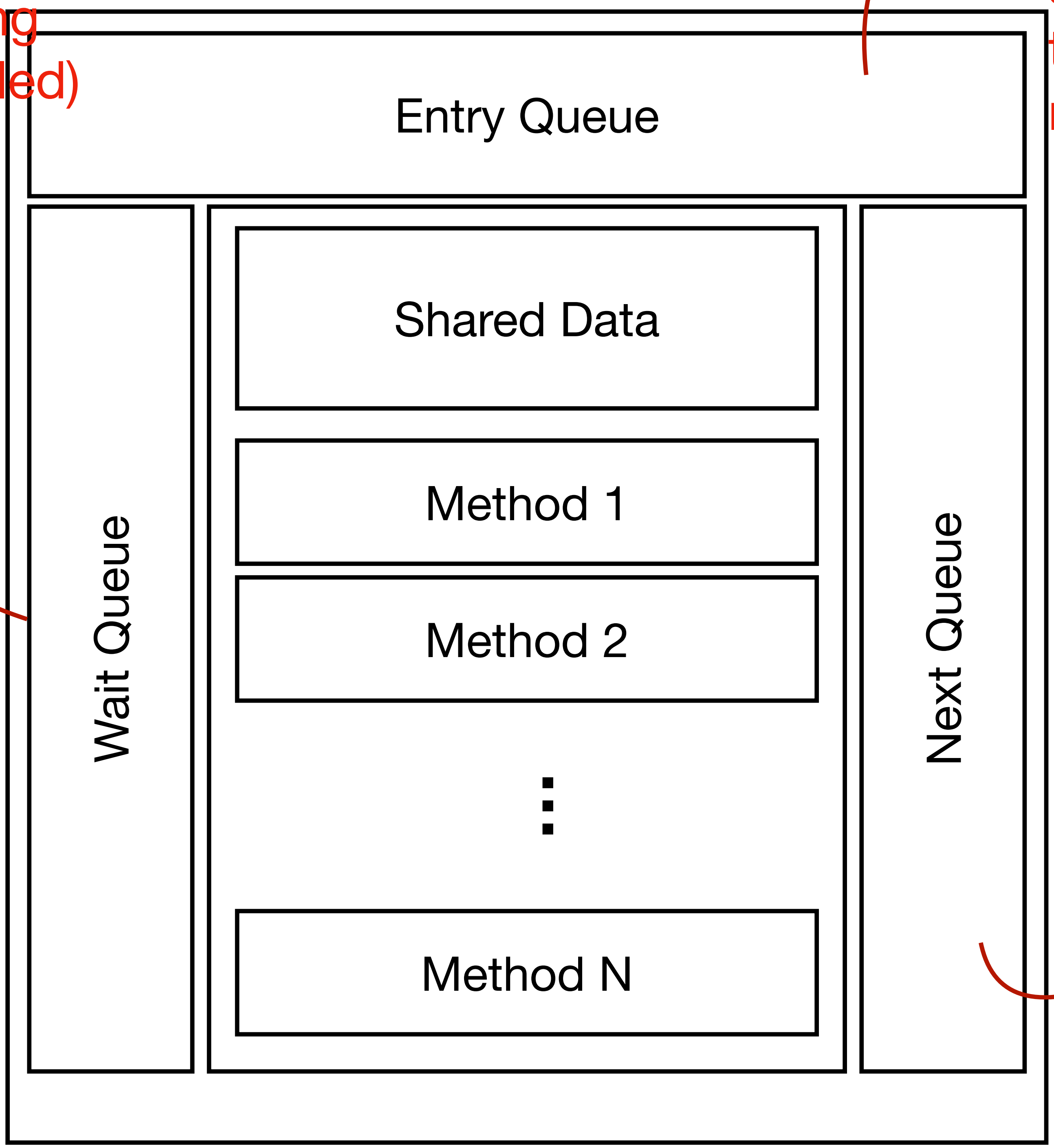
it still works!

EXAMPLES OF HOARE-STYLE MONITOR

How does it work?

queue of threads waiting for a condition inside a monitor method (i.e., counting semaphore is signalled)

queue of a binary semaphore ensuing that one thread may execute a given monitor method at a time



queue of threads signalled a counting semaphore (they have to wait now if there was a thread waiting for that condition)

Hoare Monitor implementation with semaphores

```
class Monitor {
  public:
    void ConditionWait();   // calls cvar.wait() and implements Hoare semantics
    void ConditionSignal(); // calls cvar.singal() and implements Hoare semantics
    void someMethod();      // user-defined methods working on shared data

  private:
    <shared data>;     // data being protected by monitor
    semaphore lock;    // controls entry to monitor

    semaphore cvar;    // suspends a thread on a wait
    int waiters_cvar;  // number of threads waiting on a cvar

    semaphore next;    // suspends this thread when signalling another
    int waiters_next;  // number of threads suspended on next
}

Monitor::Monitor(int N) {
  cvar = N; // initialized to N
  lock = 1; // initialized to 1 as nobody is in the monitor
  next = 0; // initialized to 0 as nobody is suspended because of signalling
  waiters_next = 0;
  waiters_cvar = 0;
}
```

Using the monitor class

```
void Monitor::someMethod() {
  lock.Wait();     // lock the monitor
  <ops on data and calls to ConditionWait() and ConditionSignal()>
  if (waiters_next > 0)
    next.Signal(); // resume a suspended thread
  else
    lock.Signal(); // allow a new thread into the monitor
}
```

Monitor implementation with semaphores

```
void Monitor::ConditionWait() {
  waiters_cvar += 1;// increment the number of waiters
  if(waiters_next > 0)
    next.Signal();  // resume a suspended thread
  else
    lock.Signal();  // allow a new thread in the monitor
  cvar.wait();      // wait on the condition
  waiters_cvar -= 1;// on waking up decrement the number of waiters
}

void Monitor::ConditionSignal() {
  if (waiters_cvar > 0) { // don't signal cvar if nobody is waiting
    waiters_next += 1; // increment the number of suspended threads
    cvar.Signal();     // awaken a waiting thread
    next.Wait();       // wait for it to finish or wait again
    waiters_next -= 1; // decrement the number of suspended threads
  }
}
```