Operating Systems

Monitors & Condition Variables

Hongliang Liang, BUPT

Sep. 2023

Producer-consumer using semaphores

```
semaphore mutex = 1, empty = N, full = 0
send(message msg)
  down(empty)
  down(mutex)        //*
  buffer[in % N] = msg
  in = in + 1
  up(mutex)      //*
  up(full)
message receive()
  down(full)
  down(mutex)        //*
  msg = buffer[out % N]
  out = out + 1
  up(mutex)      //*
  up(empty)
  return msg
```

Monitors: the motivation

It is difficult to produce correct programs using semaphores

- correct ordering of `down()` is **tricky**
- avoiding race conditions and deadlocks is **tricky**

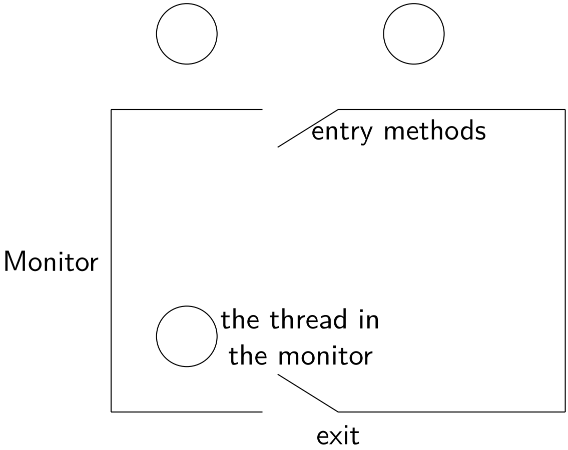
Is it possible to ask a **compiler** to generate the correct semaphore code for us?

- If so, what are the suitable higher level abstraction?

Monitors: one thread at a time

- Monitors are like objects in object-oriented programs
- Compiler enforces encapsulation and mutual exclusion
- Encapsulation
  - Local data variables are accessible only via the monitor's entry methods
- Mutual exclusion
  - Each monitor has an associated mutex lock
  - Threads must acquire the lock when invoking any of the **entry methods**
  - Automatically, only **one thread** can be active in a monitor at any time

A monitor illustrated



How BLITZ implements a monitor

```
class AMonitor
  fields
    monitorLock: Mutex

  method MyEntryMethod
    monitorLock.Lock()
    ...
    if ...
      monitorLock.Unlock()
      return
    endIf
    ...
    monitorLock.Unlock()
  endMethod
```

Implementing producer-consumer with monitors

```
monitor ProducerConsumer
  send(message msg)
    while in - out == N do
      sleep()
    buffer[in % N] = msg
    if in == out then
      in = in + 1
      wakeup(receiverThread)    //
    else in = in + 1

  message receive()
    while in == out do
      sleep()
    msg = buffer[out % N]
    if in - out == N then
      out = out + 1
      wakeup(senderThread)  //
    else out = out + 1
    return msg
```

Problems with using a monitor

When the sender thread sleeps and blocks itself in case of buffer full, no one else will be able to wake it up

Why?

- The sender thread goes to sleep inside a monitor
- No other threads are able to enter the monitor to wake it up!

The root of the problem

We have two concurrency problems to solve

The need for **mutual exclusion**

- Only one thread in the critical section at a time
- Handled by the definition of a monitor: one thread in the monitor at any time

The need for **synchronization**

- Wait (sleep) until a certain condition holds
- Signal (wake up) waiting threads when the condition holds

Revisiting a previous idea without monitors

```
send(message msg)
  acquire(buffer_lock)
  while in - out == N do
    release(buffer_lock)
    sleep()     //
    acquire(buffer_lock)
  buffer[in % N] = msg
  if in == out then
    in = in + 1
    wakeup(receiverThread)//
  else
    in = in + 1
  release(buffer_lock)
```

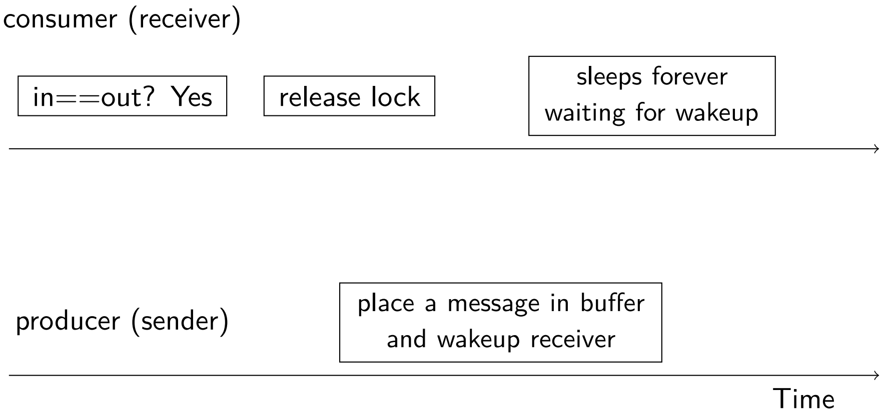
```
message receive()
  acquire(buffer_lock)
  while in == out do
    release(buffer_lock)
    sleep()     //
    acquire(buffer_lock)
  msg = buffer[out % N]
  if in - out == N then
    out = out + 1
    wakeup(senderThread)//
  else
    out = out + 1
  release(buffer_lock)
  return msg
```

The problem in this idea

In the first try of solving synchronization problem in producer-consumer, the solution suffers from the **lost wakeup problem**

```
while in == out do
   release(buffer_lock)
   sleep()        //
   acquire(buffer_lock)
```

Revisiting the lost wakeup problem



Solving this in the context of monitors

It will be good to make the `release()/sleep()/acquire()` trio before-or-after atomic

- In the context of a monitor, the thread exits the monitor, blocks itself to wait for an event to occur, and enter the monitor when it wakes up
- No interruption between it exits the monitor and blocks itself
- No interruption between it wakes up and re-enters the monitor

Condition variables

A **condition variable** represents a condition that a thread is waiting for and signaling

It supports three operations

- condition.wait(): a thread exits the monitor, waits for the condition variable to hold, and enters the monitor again when it does
- condition.signal(): signals (wakes up) a waiting thread on the condition variable, so that it can try to enter the monitor
- condition.broadcast(): signals (wakes up) all waiting threads on the condition variable, so that they can all try to enter the monitor

`wait()`, `signal()`, and `broadcast()` are made before-or-after **atomic actions** in order to avoid the lost wakeup problem

Revisiting the producer-consumer problem

```
monitor ProducerConsumer
  Condition full // Sender threads wait when buffer is full
  Condition empty // Receiver threads wait when buffer is empty
  send(message msg)
    if in - out == N then
      full.wait() // buffer is full, so I wait outside monitor
    buffer[in % N] = msg
    if in == out then
      in = in + 1
      empty.signal() // wake up a receiver waiting outside
    else in = in + 1
  message receive()
    if in == out then
      empty.wait() // buffer is empty, so I wait outside monitor
    msg = buffer[out % N]
    if in - out == N then
      out = out + 1
      full.signal() // wake up a sender waiting outside
    else out = out + 1
    return msg
```

How BLITZ implements condition variables

You really get it by reading/writing the code!

```
class Condition
  fields
    waitingThreads: List [Thread]
method Init()
  waitingThreads = new List [Thread]
endMethod
method Wait(mutex: ptr to Mutex)
  disable interrupts
  mutex.Unlock()  // release and exit the monitor
  waitingThreads.AddToEnd(currentThread)
  currentThread.Sleep()
  mutex.Lock()  // acquire to re-enter the monitor
  restore interrupts
endMethod
```

How BLITZ implements condition variables

```
method Signal(mutex: ptr to Mutex)
  disable interrupts
  t = waitingThreads.Remove()
  if t
    t.status = READY
    readyList.AddToEnd(t)
  endIf
  restore interrupts
endMethod
```

How BLITZ implements condition variables

```
method Broadcast(mutex: ptr to Mutex)
  disable interrupts
  while true
    t = waitingThreads.Remove()
    if t == null
       break
    endIf
    t.status = READY
    readyList.AddToEnd(t)
  endWhile
  restore interrupts
endMethod
```

Revisiting the producer-consumer problem

```
monitor ProducerConsumer
  Condition full // Sender threads wait when buffer is full
  Condition empty // Receiver threads wait when buffer is empty
  send(message msg)
    if in - out == N then
      full.wait() // buffer is full, so I wait outside monitor
    buffer[in % N] = msg
    if in == out then
      in = in + 1
      empty.signal() // wake up a receiver waiting outside
    else in = in + 1
  message receive()
    if in == out then
      empty.wait() // buffer is empty, so I wait outside monitor
    msg = buffer[out % N]
    if in - out == N then
      out = out + 1
      full.signal() // wake up a sender waiting outside
    else out = out + 1
    return msg
```

One more (last) problem

- The sender thread is running in the monitor
- It adds a message to an empty shared buffer
- It signals a waiting receiver thread, waking it up
- At this time, the sender and receiver thread **cannot both run inside the monitor**
- Which one runs (in the monitor), and which one blocks (outside of the monitor)?

Design choices of monitors

Only one thread is active in the monitor at a time, so what should we do when a thread is unblocked on **signal**?

Two choices when thread A signals a condition unblocking thread B:

1. B enters the monitor, A waits for B to exit the monitor
2. A remains in the monitor, B waits for A to exit the monitor

Which one do you prefer to?

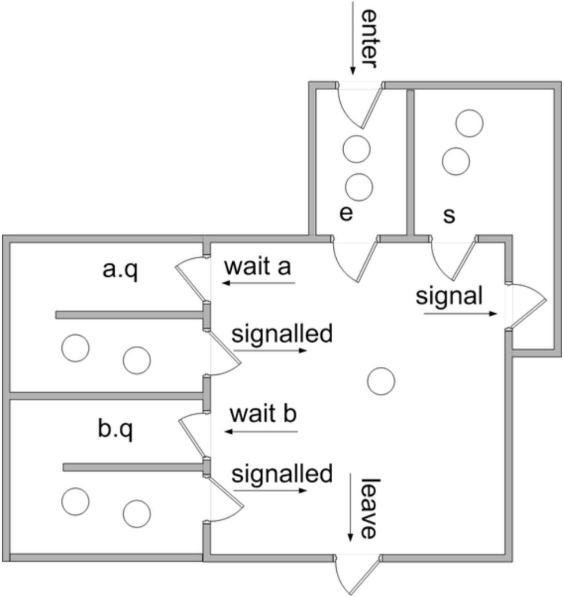
Option 1: Hoare Semantics

Tony Hoare, who invented the monitor, proposed Hoare Semantics

- The signaling thread, A, always leaves and waits
- The signaled thread, B, enters the monitor **immediately**
- No other threads can enter the monitor between the execution of the signal operation by the signaling thread A, and the acquisition of the lock by the signaled thread B

Now the signaled thread **B** can have a **guarantee** that the condition holds when it enters the monitor

Option 1: Hoare Semantics



Option 2: MESA Semantics

MESA Semantics is more **relaxed**

- The only guarantee: the signaled thread is awakened
- It will have to compete against all other threads for the monitor lock

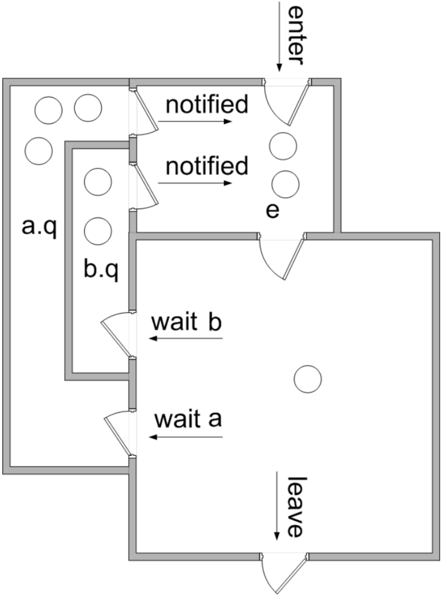
The signaling thread is allowed to continue its execution

- When it leaves the monitor, the awakened thread, and perhaps other threads, will try to acquire the monitor lock

The signaled thread will **eventually** enter the monitor

- but there are no guarantee that the condition still holds!

Option 2: MESA Semantics



How BLITZ implements condition variables

```
method Wait(mutex: ptr to Mutex)
  disable interrupts
  mutex.Unlock()
  waitingThreads.AddToEnd(currentThread)
  currentThread.Sleep()
  mutex.Lock()
  enable interrupts
endMethod
```

How BLITZ implements condition variables

```
method Signal(mutex: ptr to Mutex)
  disable interrupts
  t = waitingThreads.Remove()
  if t
    t.status = READY
    readyList.AddToEnd(t)
  endIf
  enable interrupts
endMethod
```

What semantics does BLITZ implement?

What semantics does BLITZ implement?

MESA Semantics

Revisiting the producer-consumer problem

```
monitor ProducerConsumer
  Condition full // Sender threads wait when buffer is full
  Condition empty // Receiver threads wait when buffer is empty
  send(message msg)
    if in - out == N then
      full.wait() // buffer is full, so I wait outside monitor
    buffer[in % N] = msg
    if in == out then
      in = in + 1
      empty.signal() // wake up a receiver waiting outside
    else in = in + 1
  message receive()
    if in == out then
      empty.wait() // buffer is empty, so I wait outside monitor
    msg = buffer[out % N]
    if in - out == N then
      out = out + 1
      full.signal() // wake up a sender waiting outside
    else out = out + 1
    return msg
```

How to change the code to use MESA semantics?

Producer-consumer with MESA Semantics

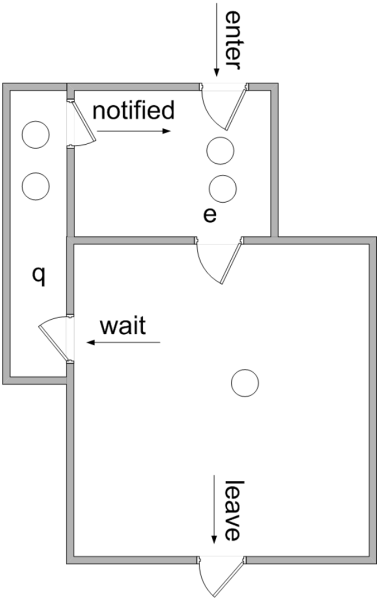
```
monitor ProducerConsumer
  Condition full // Sender threads wait when buffer is full
  Condition empty // Receiver threads wait when buffer is empty
  send(message msg)
    while in - out == N do
      full.wait() // buffer is full, so I wait outside monitor
    buffer[in % N] = msg
    if in == out then
      in = in + 1
      empty.signal() // wake up a receiver waiting outside
    else in = in + 1
  message receive()
    while in == out do
      empty.wait() // buffer is empty, so I wait outside monitor
    msg = buffer[out % N]
    if in - out == N then
      out = out + 1
      full.signal() // wake up a sender waiting outside
    else out = out + 1
    return msg
```

A simplified monitor in Java

Each object may be used as a monitor

- Entry methods requiring mutual exclusion must be explicitly marked as **synchronized**
- Rather than having explicit condition variables, each monitor (i.e., object) is equipped with a single wait queue, in addition to its entrance queue
- All waiting is done on this single wait queue, by calling `wait()`
- All `notify()` and `notifyAll()` operations apply to this queue

A Java-style monitor



Semaphores/spin locks/CVs are interchangeable (4 slides optional)

- Each is implementable through a combination of the others
- Depending on the use-case one is faster than the other
  - How often is the critical section executed?
  - How many threads compete for a critical section?
  - How long is the lock taken?

Implementing a mutex with a semaphore

```c
sem_t sem;
sem_init(&sem, 1);

sem_wait(&sem);
... // critical section
sem_post(&sem);
```

Implementing a semaphore with CV/locks

```
typedef struct {
 int value;              // sem value
 pthread_mutex_t lock; // access to sem
 pthread_cond_t cond;  // wait queue
} sem_t;

void sem_init(sem_t *s, int val) {
 s->value = val;
 pthread_mutex_init(&(s->lock), NULL);
 pthread_cond_init(&(s->cond), NULL);
}
```

Implementing a semaphore with CV/locks

```c
void sem_wait(sem_t *s) {
 pthread_mutex_lock(&(s->lock));
 while (s->value <= 0)
     pthread_cond_wait(&(s->cond), &(s->lock));
 s->value--;
 pthread_mutex_unlock(&(s->lock));
}

void sem_post(sem_t *s) {
 pthread_mutex_lock(&(s->lock));
 s->value++;
 pthread_cond_signal(&(s->cond));
 pthread_mutex_unlock(&(s->lock));
 }
```

Reader/writer locks

- A single (exclusive) writer, multiple (N) concurrent readers
- Implement using two semaphores: `lock` for reader count (`readers`), `wlock` for the writer
  - Both semaphores initialized with (1)
  - Writer only waits/posts on `wlock` when acquiring/releasing
  - Reader waits on `lock`, increments/decrements `readers`, if `readers==0`, must wait/post on `wlock`

Reader/writer locks

```
void acquire_readlock(rwlock_t *rw) {
 sem_wait(&rw->lock);
 rw->readers++;
 if (rw->readers == 1)
    sem_wait(&rw->wlock); // first r, also grab wlock
 sem_post(&rw->lock);
}

void release_readlock(rwlock_t *rw) {
 sem_wait(&rw->lock);
 rw->readers--;
 if (rw->readers == 0)
   sem_post(&rw->wlock); // last r, also release wlock
 sem_post(&rw->lock);
}
```

Bugs in concurrent programs

- Writing concurrent programs is hard!
- ***Atomicity bug:*** concurrent, unsynchronized modification (lock!)
- ***Order-violating bug:*** data is accessed in wrong order (use CV!)
- ***Deadlock:*** program no longer makes progress (locking order)

Atomicity bugs

One thread checks value and prints it while another thread concurrently modifies it.

```
int shared = 24;

void T1() {
 if (shared > 23) {
     printf("Shared is >23: %d\n", shared);
 }
}
void T2() {
 shared = 12;
}
```

Atomicity bugs

One thread checks value and prints it while another thread concurrently modifies it.

```c
int shared = 24;

void T1() {
 if (shared > 23) {
     printf("Shared is >23: %d\n", shared);
 }
}
void T2() {
 shared = 12;
}
```

- T2 may modify `shared` between `if` check and `printf` in T1.
- Fix: use a common mutex between both threads when accessing the shared resource.

Order-violating bug

One thread assumes the other has already updated a value.

```
Thread 1::
void init() {
  mThread = PR_CreateThread(mMain, ...);
  mThread->State = ...;
}

Thread 2::
void mMain(...) {
  mState = mThread->State;
}
```

Order-violating bug

One thread assumes the other has already updated a value.

```
Thread 1::
void init() {
  mThread = PR_CreateThread(mMain, ...);
  mThread->State = ...;
}

Thread 2::
void mMain(...) {
  mState = mThread->State;
}
```

- Thread 2 may run before `mThread` is assigned in T1.
- Fix: use a CV to signal that `mThread` has been initialized.

Deadlock

Locks are taken in conflicting order.

```
void T1() {
    lock(L1);
    lock(L2);
}

void T2() {
    lock(L2);
    lock(L1);
}
```

Deadlock

Locks are taken in conflicting order.

```
void T1() {
    lock(L1);
    lock(L2);
}

void T2() {
    lock(L2);
    lock(L1);
}
```

- Threads 1/2 may be stuck after taking the first lock, program makes no more progress
- Fix: acquire locks in increasing (global) order.

Summary

- Spin lock, CV, and semaphore synchronize multiple threads
  - Spin lock: atomic access, no ordering, spinning
  - Condition variable: atomic access, queue, OS primitive
  - Semaphore: shared access to critical section with (int) state
- All three primitives are equally powerful
  - Each primitive can be used to implement both other primitives
  - Performance may differ!
- Synchronization is challenging and may introduce different types of bugs such as atomicity violation, order violation, or deadlocks.