# COMP 4735: Operating Systems

## Lesson 8.4: Monitors

Rob Neilson

rneilson@bcit.ca

# Monitors

- Monitors are high-level synchronization abstractions
- The are essentially a specialized form of an ADT (object)
  - Monitor encapsulates implementation (just like an object)
  - Monitor exposes a public interface (just like an object)
- ***Only one process can be executing in the monitor at a time***
- This is similar to Java *synchronized* methods
  - a Java class can have a number of *synchronized* methods
  - the JVM guarantees that when a synchronized method is running, no other synchronized methods in the same class will start executing

# Why Monitors??

- monitors are easier (conceptually) to code with
- as a programmer you just need to ensure that any critical sections are coded in a monitor, and you are pretty much done
- Of course this only makes sense if the language and programming environment you are working with supports the monitor abstraction!

# Monitors (cont)

*Conceptually, a monitor adds a critical section capability to the idea of an object*

For example, we could change a critical section that is guarded by a semaphore into a critical section guarded by a monitor:

```
class anADT {
   private:
   semaphore mutex = 1;
   <ADT data structs>
   ...
public proc_i() {
      P(mutex);
         <processing for proc_i>
      V(mutex);
   }
   ...
}
```

***this**, would be the same as **this**, using a monitor, or **this**, in Java*

```
monitor anADT {
   <ADT data structs>
   ...
public proc_i()
   {
   <processing for proc_i>
   }
   ...
}
```

```
class anADT {
   <ADT data structs>
   ...
public synchronized proc_i()
   {
   <processing for proc_i>
   }
   ...
}
```

# Example: Shared Balance Problem

- In the shared balance problem we have a shared variable that can be updated by multiple threads.
- We need to synchronize actions so that updates to the shared variable are mutually exclusive.
- With monitors we do this as follows:

```
monitor sharedBalance {
  double balance;
public:
  credit(double amount) {balance += amount;};
  debit(double amount) {balance -= amount;};
  . . .
}
```

- Note that because it is a monitor, access to all methods (credit/debit) are run mutually exclusively.

# Readers-Writers Problem Statement

There are ***many threads*** that need access to a shared variable (or shared resource).

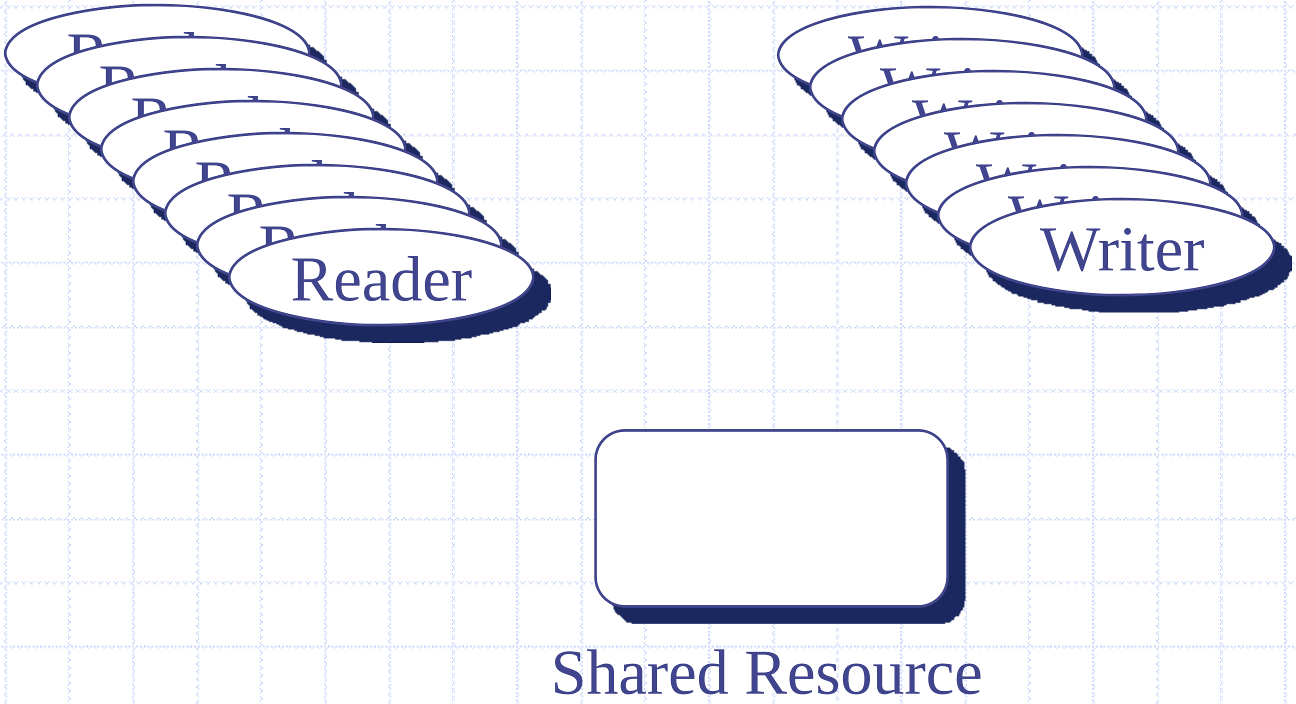
- some of the threads just require read access
- some threads just require write access

- *many reader threads can access the resource at the same time*
- *writer threads require exclusive access to the resource*

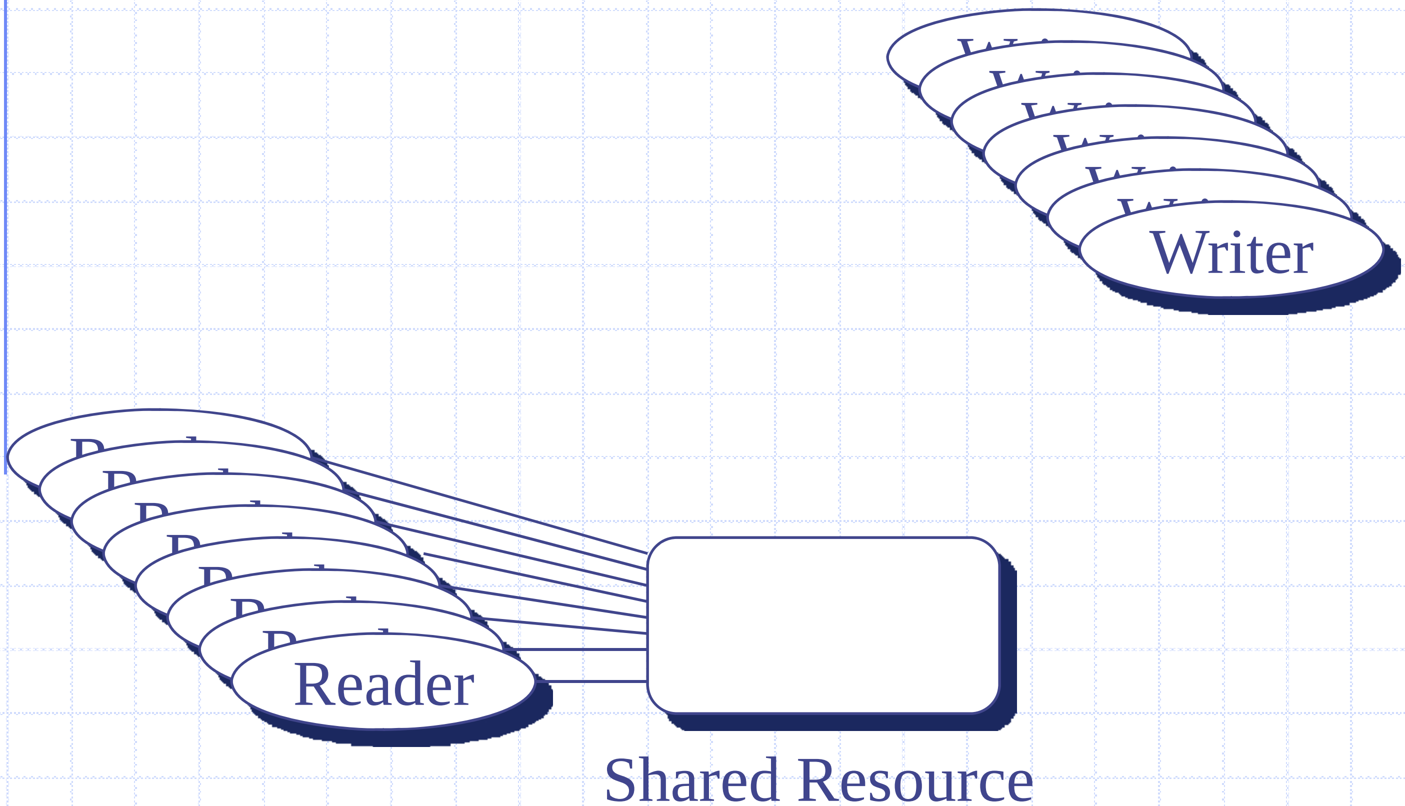
**The challenge is to find a solution that:**

- **maximizes concurrency** among the threads that need to access the resource
- **avoids starvation**

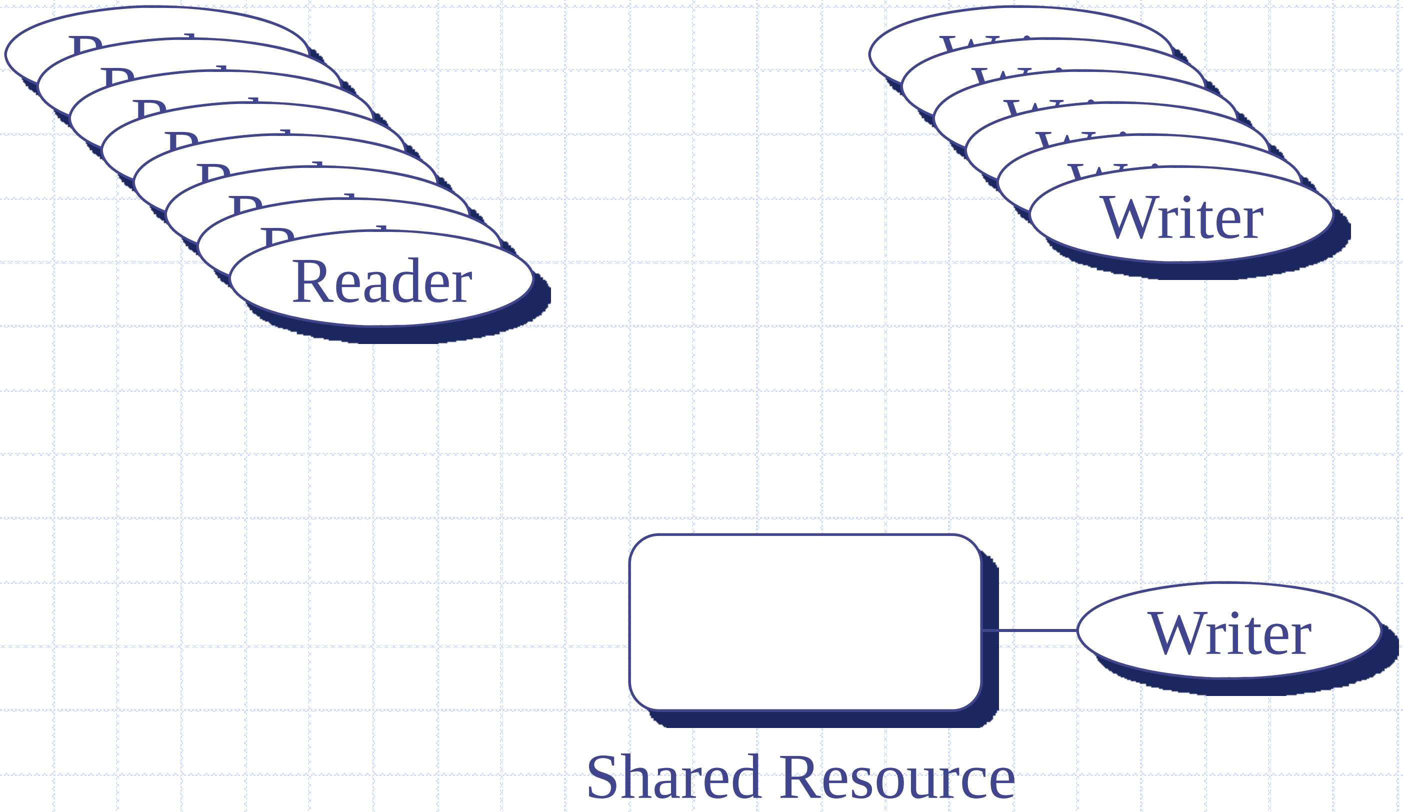
# Readers-Writers Problem Depicted



# Readers-Writers Problem: Simultaneous Reading



# Readers-Writers Problem: Exclusive Writes



# Readers and Writers (Solution A)

```
monitor readerWriter_A {
public:
   mon_read() {
      <read the resource>
   };
   mon_write() {
      <write the resource>
   };
};
```

- *this is just the monitor part … we still need some code to create the threads and call the monitor …*

# Reader and Writers (A)

- This code spawns a bunch of readers and writers, and lets them access the shared resource using the monitor from the previous slide

```
reader(){                    writer(){
  while(TRUE) {                while(TRUE) {
    . . .                        . . .
    mon_read();                  mon_write();
    . . .                        . . .
}                            }
fork(reader, 0);
. . .
fork(reader, 0):
fork(writer, 0);
. . .
fork(writer, 0);
```

# Readers and Writers

The problem with solution A:

- the monitor ensures that the reading and writing code is accessed sequentially, but …
- ***we need to allow multiple readers while blocking writers***
- maybe we could **just put the write code in the monitor, and leave the reading in the main program** …

# Reader and Writers (Solution B)

```
monitor writer_B {
public:
   mon_write() {
      <write the resource>
   };
};
```

```
class reader_B {
public:
   unprotected_read() {
      <read the resource>
   };
};
```

```
reader(){
  while(TRUE) {
    . . .
    unprotected_read()
    . . .
}
fork(reader, 0);
. . .
fork(reader, 0):
fork(writer, 0);
. . .
fork(writer, 0);
```

```
writer(){
  while(TRUE) {
    . . .
    mon_write();
    . . .
}
```

# Reader / Writer Revisited

Idea: What we need is some way to indicate that a read or write action is in progress, ie, we need to know when is has started

This way, *if a write has started we can block readers* … and …

…. *if there are readers we can block the writer* and …

…. *if the writer wants to write but there are readers, we can stop allowing new readers* …. (to prevent writer starvation)

ie: our solution needs to block readers / writers when another action has already started

# Readers and Writers (Solution C)

A framework for reader/writer that works with start and finish events ...

```
monitor readerWriter_C {

  int numReaders = 0;
  int numWriters = 0;
  boolean busy = FALSE;

public:
  startRead() {
  }
  finishRead() {
  }
  startWrite() {
  }
  finishWrite() {
  }
}
```

```
reader() {                  writer() {
  while(TRUE) {               while(TRUE) {
    . . .                       . . .
    startRead();                startWriter();
        <read>                      <write>
    finishRead();               finishWriter();
    . . .                       . . .
}                           }
fork(reader, 0);
. . .
fork(reader, 0):
fork(writer, 0);
. . .
fork(writer, 0);
```

# Readers and Writers (C)

```
monitor readerWriter_C {
  int numReaders = 0;
  int numWriters = 0;
  boolean busy = FALSE;
public:
  startRead() {
    while(numberOfWriters != 0);
    numReaders++;
  }
  finishRead() {
    numReaders--;
  }
  startWrite() {
    numWriters++;
    while( busy || (numReaders > 0) ) ;
    busy = TRUE;
  }
  finishWrite() {
    numWriters--;
    busy = FALSE;
  }
}
```

# Readers and Writers (C)

```
monitor readerWriter_C {
  int numReaders = 0;
  int numWriters = 0;
  boolean busy = FALSE;
public:
  startRead() {
    while(numberOfWriters != 0);
    numReaders++;
  }
  finishRead() {
    numReaders--;
  }
  startWrite() {
    numWriters++;
    while( busy || (numReaders > 0) ) ;
    busy = TRUE;
  }
  finishWrite() {
    numWriters--;
    busy = FALSE;
  }
}
```

# Readers and Writers (deadlock)

```
monitor readerWriter_C {
  int numReaders = 0;
  int numWriters = 0;
  boolean busy = FALSE;
public:
  startRead() {
    while(numberOfWriters != 0);
    numReaders++;
  }
  finishRead() {
    numReaders--;
  }
  startWrite() {
    numWriters++;
    while( busy || (numReaders > 0) );
    busy = TRUE;
  }
  finishWrite() {
    numWriters--;
    busy = FALSE;
  }
}
```

# Condition Variables

- With monitors there is a often a need to wait on an event
- For example:
  - readers need to wait until writers are finished before entering the monitor
- So we need a mechanism by which:
  - threads can “set” a condition when they enter a particular state
  - other threads can block and wait for the condition to be cleared
  - threads can “signal” waiting threads when the condition has been cleared
- This is accomplished with a construct known as a

“**condition variable**”

# Condition Variables

- Condition variables are essentially **an implementation of events**
- Condition variables are **<u>only</u> used inside monitors**
- There is a data structure that is **global to all methods in the monitor**, and its value is manipulated by three methods:

**`wait()`**: Suspend invoking thread until another executes a signal. The monitor is released.

**`signal()`**: Resume one thread if any are suspended, otherwise do nothing.

**`queue()`**: Return TRUE if there is at least one thread suspended on the condition variable.

# Active vs Passive `signal`

- Hoare semantics (signal and exit):
  - $t_0$ executes signal while $t_1$ is waiting
    - $\Rightarrow$ $t_0$ yields the monitor to $t_1$
  - Typically the signal() is the last statement before exiting monitor
  - Assume the signal is only TRUE the instant it happens.
- Brinch Hansen semantics (signal and continue)
  - $t_0$ executes signal while $t_1$ is waiting
    - $\Rightarrow$ $t_0$ continues to execute
  - later, when $t_0$ exits the monitor $t_1$ can receive the signal
  - java monitors (synchronized methods) use signal-and-continue

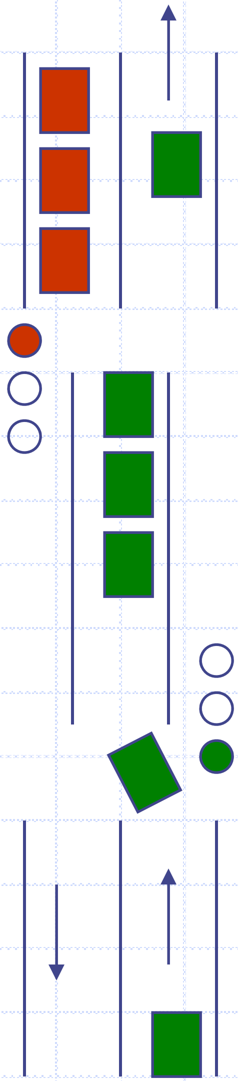
# Readers and Writers (Solution D)

```
monitor readerWriter_D {
  int numReaders = 0;
  boolean writing = FALSE;
  condition okToRead, okToWrite;

public:

 startRead() {
  if(writing || (okToWrite.queue())
     okToRead.wait();
  numReaders++;
  okToRead.signal();
 }

 finishRead() {
  numReaders--;
  if(numReaders == 0)
    okToWrite.signal();
 }
```

```
 startWrite() {
   if((numReaders != 0) || writing)
     okToWrite.wait();
   writing = TRUE;
 }

 finishWrite() {
   writing = FALSE;
   if(okToRead.queue())
     okToRead.signal()
   else
     okToWrite.signal()
 }
}
```

# Example: Synchronizing Traffic

- One-way tunnel
- Sensors detect when cars arrive
- Can only use tunnel if no oncoming traffic
- OK to use tunnel if traffic is already flowing the right way



# Example: Synchronizing Traffic

```
monitor tunnel {
  int northbound = 0, southbound = 0;
  trafficSignal nbSignal = RED, sbSignal = GREEN;
  condition busy;

public:

  northboundArrival() {         // called when a nb car arrives
    if(southbound > 0)
      busy.wait();              // wait if there are sb cars
    northbound++;               // OK to proceed in nd direction
    nbSignal = GREEN;
    sbSignal = RED;
  }

  southboundArrival() {         // called when a sb car arrives
    if(northbound > 0)
      busy.wait();              // wait if there are nb cars
    southbound++;               // OK to proceed in sb durection
    nbSignal = RED;
    sbSignal = GREEN;
  }
```

# Example: Synchronizing Traffic

```
  northboundDepart() {
      northbound--;
      if(northbound == 0)        // this was last nb car in tunnel
        while(busy.queue())      // signal all waiting sb cars ...
          busy.signal();         // ... that it is OK to proceed
  }

  southboundDepart() {
      southbound--;              // ... but we signal nb cars
      if(southbound == 0)
        while(busy.queue())
          busy.signal();
  }

}
```

# Tunnel Solution Problems

- if the traffic is heavy the lights will never change because tunnel will never be empty
- need a timer that turns both lights red (ie: stop cars from entering so that other direction can proceed)
- this would be a good exercise for you (the student) to complete on your own

# The End