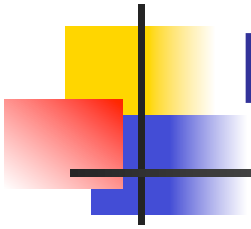
Operating Systems Design (CS 423)

Elsa L Gunter

2112 SC, UIUC

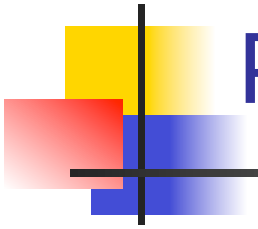
http://www.cs.illinois.edu/class/cs423/

Based on slides by Roy Campbell, Sam King, and Andrew S Tanenbaum



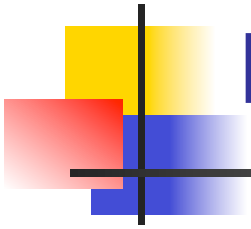
Programming with Monitors

- List shared data needed to solve problem
- Decide which locks protect which data
  - More locks allows different data to be accessed simultaneously, more complicated
  - One lock usually enough in this class
- Put lock...unlock calls around code that uses shared data



Programming with Monitors

- List ordering constraints
  - One condition variable per constraint
  - Condition variable's lock should be the lock that protects the shared data used to eval condition
- Call wait() when thread needs to wait for a condition to be true
  - Use a while loop



Programming with Monitors

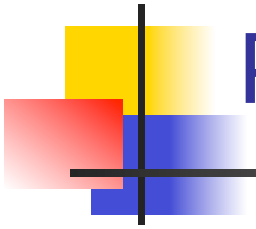
- Call signal when a condition changes
- Make sure invariant is established whenever lock is not held
  - E.g., before you call wait

Producer-consumer (bounded buffer)

- Problem: producer puts things in shared buffer, consumer takes them out.
- Synchronization for coordinating

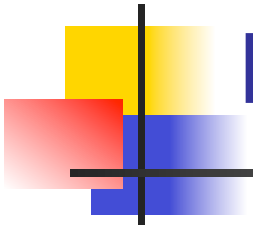
produce →  → consume

- Unix pipeline (gcc calls cpp | cc1 | cc2 | as)
- Buffer between allows them to operate independently
- What would execution be like without buffer?



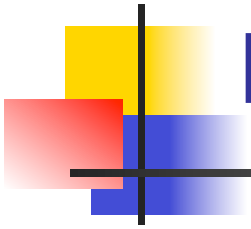
Producer-consumer: Example

- Coke machine
  - Delivery person (producer)
  - Customers buy cokes (consumer)
  - Coke machine has finite space (buffer)



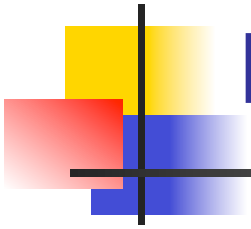
Producer-consumer using monitors

- Operations
  - Add coke to machine
  - Take coke out of machine
- Variables
  - Shared data for the coke machine
  - maxCokes (capacity of machine)
  - numCokes (number of cokes in machine)



Producer-consumer using monitors

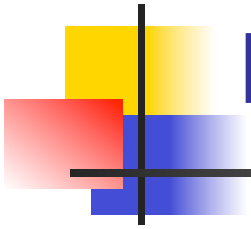
- One lock (cokeLock) to protect shared data
  - Fewer locks easier to program, less concur.
- Ordering constraints
  - Consumer must wait for producer to fill buffer if all buffers are empty (hasCoke)
  - Producer must wait for consumer to take from buffer if buffer is completely full (hasRoom)



Basic behavior – What's wrong?

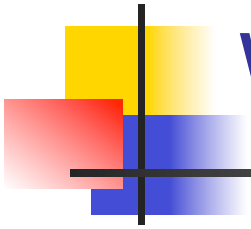
```
producer()
  lock(cokeLock);

  put one coke
  in machine;

  unlock(cokeLock);
}
```

```
consumer:
  lock(cokeLock);

  take one coke out
  of machine;

  unlock(cokeLock);
}
```



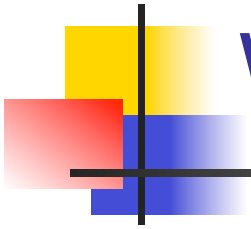
Basic behavior

```
producer()                          consumer()
  lock(cokeLock);                     lock(cokeLock);
  while (numCokes ==                  while(numCokes == 0){
          maxCokes){                    wait(cokrLock,hasCoke);
  wait(cokeLock,hasRoom);
  put one coke                          take one coke out
  in machine;                           of machine;
  signal(cokeLock;hasCoke);           signal(cokeLock,hasRoom);
  unlock(cokeLock);                     unlock(cokeLock);
}                                   }
```



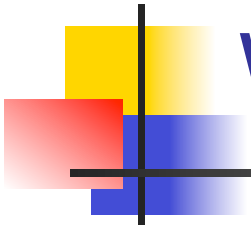
What if producer loops? Is it OK?

```
Producer() {
 lock(cokeLock);
 while(1) {
   while(numCokes == max) {
     wait(cokeLock, hasRoom);
   }
   add coke to machine;
   signal(hasCoke);
 }
 unlock(cokeLock);
}
```



What if we add sleep?

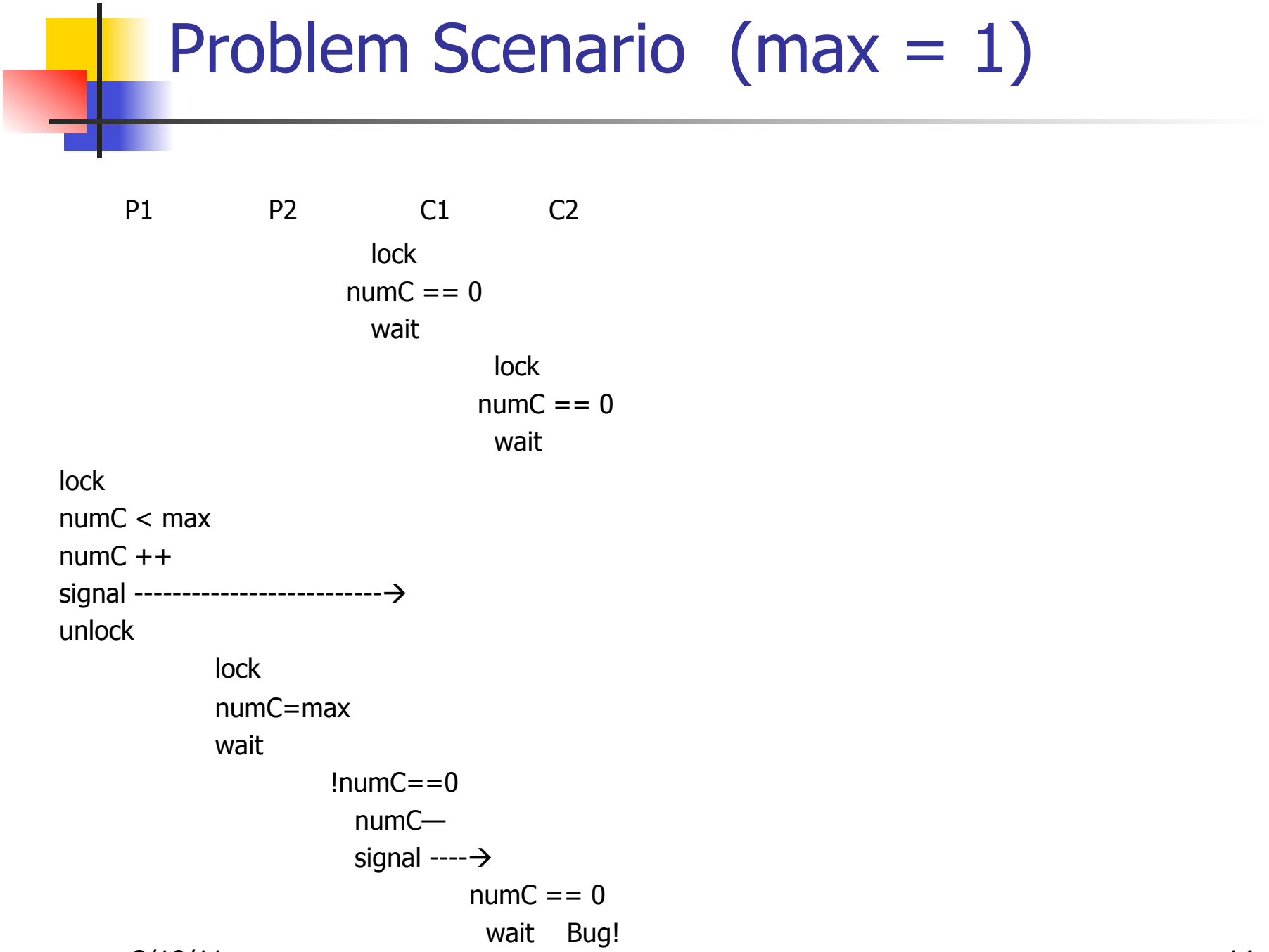
```
Producer() {
 lock(cokeLock);
 while(1) {
   sleep(1 hour);
   while(numCokes == max) {
     wait(cokeLock, hasRoom);
   }
   add coke to machine;
   signal(hasCoke);
 }
 unlock(cokeLock);
}
```

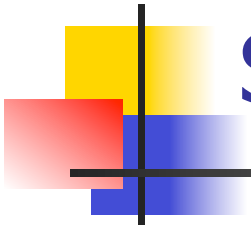


What is wrong here? (hard)

```
producer()                          consumer()
  lock(cokeLock);                     lock(cokeLock);
  while (numCokes                     while(numCokes == 0){
         == maxCokes){                  wait(cokeLock,condVar);
  wait(cokeLock,condVar)};
  put one coke                          take one coke out
  in machine;                           of machine;
  signal(cokeLock;condVar);             signal(cokeLock,condVar);
  unlock(cokeLock);                     unlock(cokeLock);
}                                   }
```

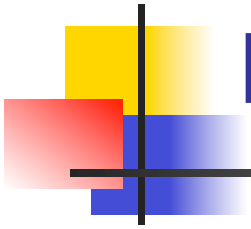
Problem Scenario (max = 1)





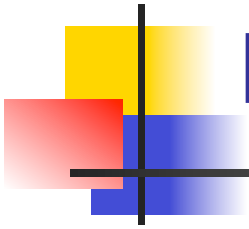
Solution to too few Cond Vars

- Use broadcast
- Will wake everyone up
- Each will check its own progress condition
- First one to check and get true will go
- Much more inefficient than signal and multiple condition variables



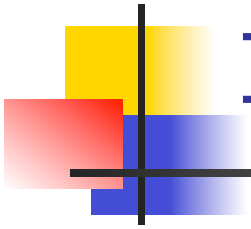
Reader – Writer Locks

- Problem: With standard locks, threads acquire lock to read shared data
- Prevents other reader threads from accessing data
- Can we allow more concurrency?



Reader – Writer Locks

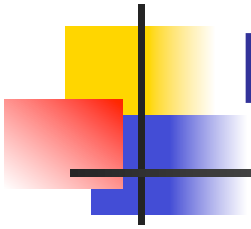
- Problem definition:
  - Shared data that will be read and written by multiple threads
  - Allow multiple readers to access shared data when no threads are writing data
  - A thread can write shared data only when no other thread is reading or writing the shared data



Interface

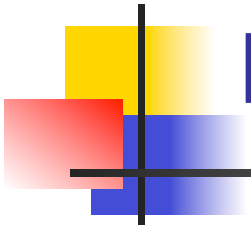
- readerStart()
- readerFinish()
- writerStart()
- writerFinish()

- Many threads can be in between a readStart and readerFinish
- Only one thread can be between writerStart and writitierFinish



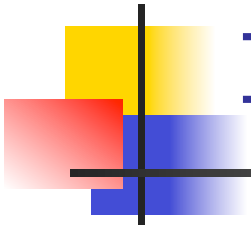
Example: Calendar

- Goal: online calendar for a class
- Lots of people may read it at the same time
- Only one person updates it (prof, Tas)
- Shared data
- map<date, listOfEvents> EventMap
- listOfEvents GetEvents(date)
- AddEvent(data, newEvent)



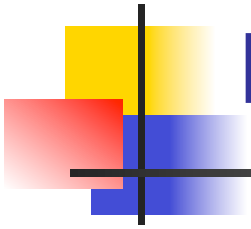
Basic Code – Single Threaded

```
GetEvents(date) {
 List events = EventMap.find(date).copy();
 return events;
}

AddEvent(data, newEvent) {
 EventMap.find(date) += newEvent;
}
```



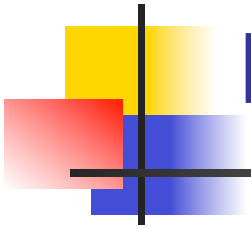
Inefficient Multi-threaded code

```
GetEvents(date) {
 lock(mapLock);
 List events = EventMap.find(date).copy();
 unlock(mapLock);
 return events;
}
AddEvent(data, newEvent) {
 lock(mapLock);
 EventMap.find(date) += newEvent;
 unlock(mapLock);
}
```



How to do with reader – write locks?

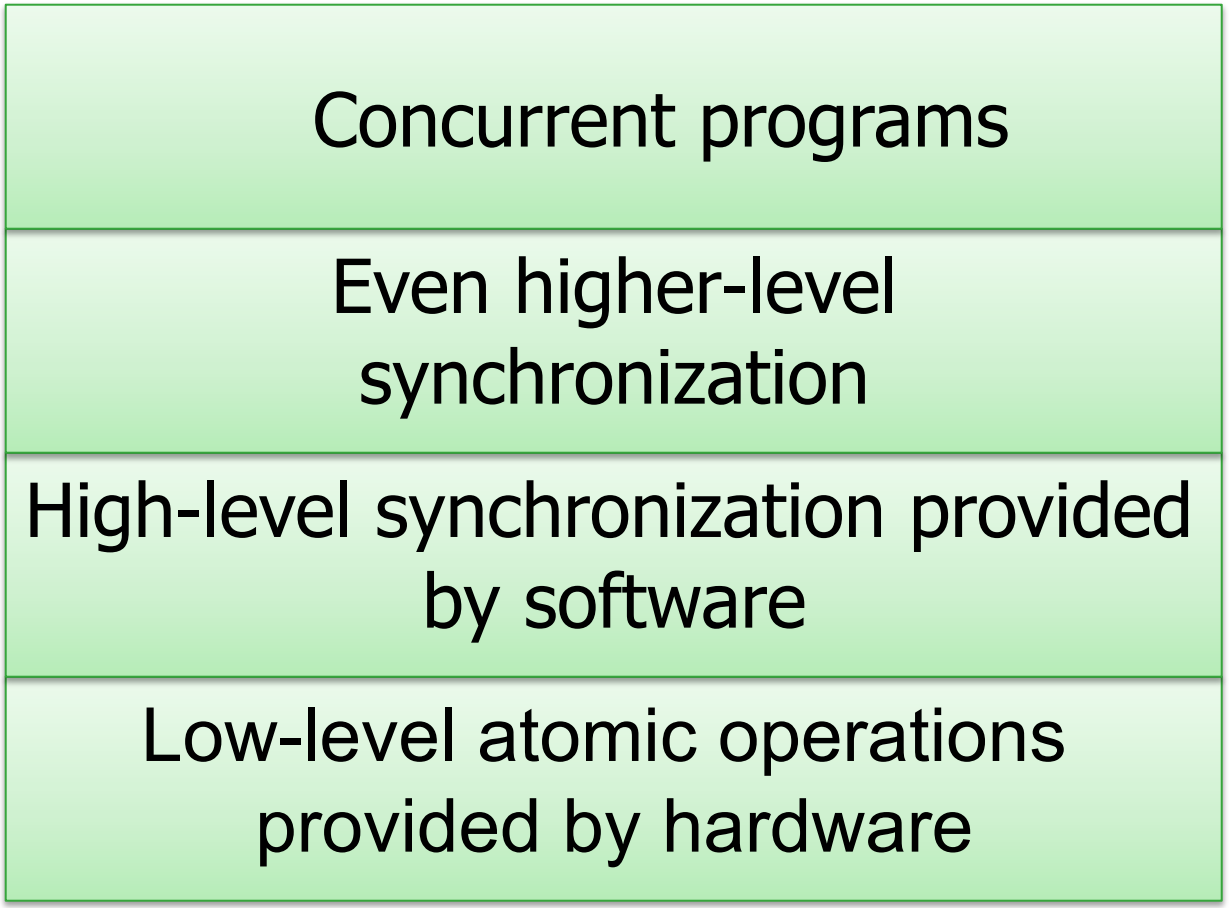
```
GetEvents(date) {

 List events = EventMap.find(date).copy();

 return events;
}
AddEvent(data, newEvent) {

 EventMap.find(date) += newEvent;

}
```

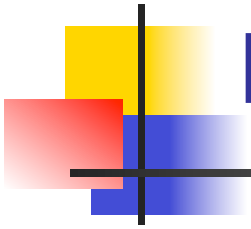


How to do with reader – write locks?

```
GetEvents(date) {
 readerStart(maRWLock);
 List events = EventMap.find(date).copy();
 readerFinish(mapRWLock);
 return events;
}
AddEvent(data, newEvent) {
 writerStart(maRWLock);
 EventMap.find(date) += newEvent;
 writerFinish(mapRWLock);
}
```

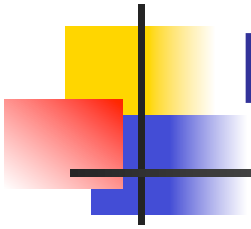
Additional Layer of Synchronization





Reader – Writer Locks using Monitors

- Note: Implement Reader/Writer Locks as an abstractions, *not* as an integrated part of code
- Central Questions:
  - Shared Data?
  - Ordering Constraints?
  - Invariants?



Reader – Writer Locks using Monitors

- Note: Implement Reader/Writer Locks as an abstractions, *not* as an integrated part of code
- Central Questions:
  - Shared Data?

    NumReaders

    NumWriters
  - Ordering Constraints?
  - Invariants?