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

Producer-consumer: Example

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

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);
}
```

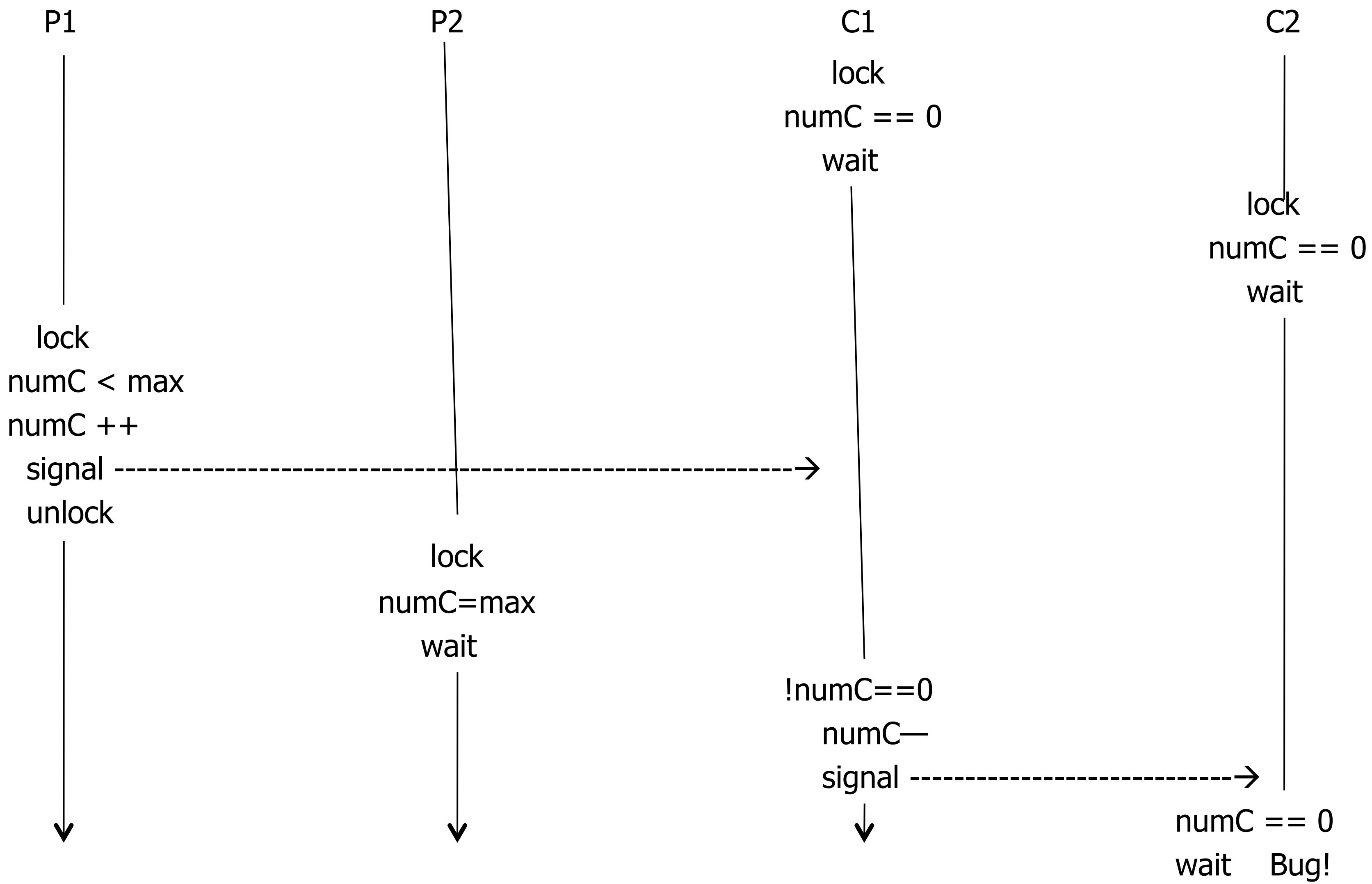
What if we add sleep?

```
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)



P1

lock
numC < max
numC ++
signal
unlock

P2

lock
numC=max
wait

C1

lock
numC == 0
wait

!numC==0
numC—
signal

C2

lock
numC == 0
wait

numC == 0
wait   Bug!

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 writierFinish

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);
}
```

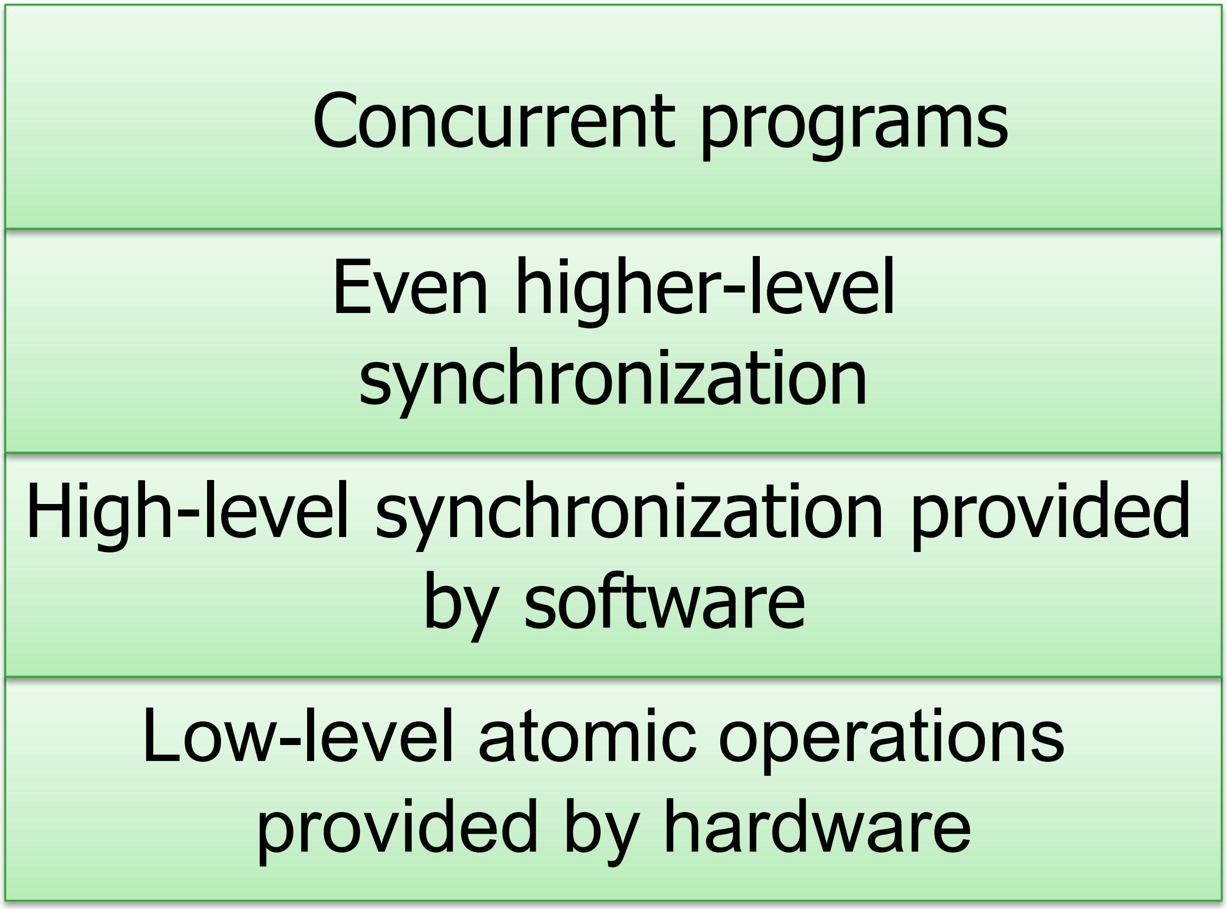
How to do with reader – write locks?

```
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



Concurrent programs

Even higher-level synchronization

High-level synchronization provided by software

Low-level atomic operations provided by hardware

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?
  - How many Condition Variables?

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?
  - How many Condition Variables?

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?
    - readerStart must wait if there are writers
    - writerStart must wait if there are readers or writes
  - How many Condition Variables?

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?
    - readerStart must wait if there are writers
    - writerStart must wait if there are readers or writes
  - How many Condition Variables?
    - One: condRW (no readers or writers)

Basic Implementation

```
readerStart() {









}
```

```
readerFinish() {








}
```

Basic Implementation

```
readerStart() {
  lock(lockRW);

  while(numWriters > 0){
    wait(lockRW,condRW);
  };

  numReaders++;

  unlock(lockRW);
}
```

```
readerFinish() {
  lock(lockRW);

  numReaders--;


  broadcast(lockRW,condWR);

  unlock(lockRW);
}
```

Basic Implementation

```
writerStart() {
  lock(lockRW);

  while(numReaders > 0||
           numWriters >0){
    wait(lockRW,condRW);
  };

  numWriters++;

  unlock(lockRW);
}
```

```
writerFinish() {
  lock(lockRW);

  numWriters--;


  broadcast(lockRW,condWR);

  unlock(lockRW);
}
```

Better Implementation

```
readerStart() {
  lock(lockRW);

  while(numWriters > 0){
    wait(lockRW,condRW);
  };

  numReaders++;

  unlock(lockRW);
}
```

```
readerFinish() {
  lock(lockRW);

  numReaders--.

  if(numReaders == 0){
    signal(lockRW,condWR);
  };
  unlock(lockRW);
}
```

Better Implementation

- Can we change broadcast to signal in writerFinish() in a similar way?

- Many Readers at a time, but only one Writer
- How long will one writer wait?
  - Starvation – process never gets a turn
- How to give priority to writer?

Write Priority

```
readerStart() {
  lock(lockRW);

  while(activeWriters + waitingWriters > 0){
    wait(lockRW,condRW);
  };

  numReaders++;

  unlock(lockRW);
}
```

Write Priority

```
writerStart() {
  lock(lockRW);
  waitingWriters ++;
  while(numReaders > 0||
           numWriters >0){
    wait(lockRW,condRW);
  };
  waitingWriters--;
  numWriters++;

  unlock(lockRW);
}
```

Lock and Reader / Writer Locks

- Reader-writer functions are similar to standard locks
  - Call readerStart before read shared data
  - Call readerFinish after done reading data
  - Call writerStart before writing shared data
  - Call writerFinish after done writing data
- These are known as “reader-writer locks”
  - Thread in between readerStart and readerFinish “holds a read lock”
  - Thread in between writerStart and writerFinish “holds a write lock”
- Compare reader-writer locks with standard locks

Semaphores (not used in this class)

- Like a generalized lock
- Semaphore has a non-negative integer value (>= 0) and supports
  - Down(): wait for semaphore to become positive, decrement semaphore by 1 (originally called “P” for Dutch “proberen”)
  - Up(): increment semaphore by 1 (originally called “V” for Dutch “verhogen”). This wakes up a thread waiting in down(), if there are any.
  - Can also set the initial value for the semaphore

Semaphores – Quick Review

- The key parts in down() and up() are atomic
  - Two down calls at the same time cannot decrement the value below 0
- Binary semaphore
  - Value is either 0 or 1
  - Down() waits for value to become 1, then sets to 0
  - Up() sets value to 1, waking up waiting down