Monitors

Arvind Krishnamurthy
Spring 2004

Monitors Recap

- Monitors contain:
  - “lock” for mutual exclusion
  - “condition variables” for scheduling constraints
- Monitor usage:
  - Obtain lock
  - Perform tasks. If certain scheduling constraints are not met, release lock and sleep till appropriate conditions are met.
  - Sleeping threads are woken up by “signal” and “broadcast” operations
  - Release lock when thread exits critical section

Synchronized queue

- **Rule:** must hold lock when doing condition variable operations

```
AddToQueue()
{
   lock.Acquire();

   put item on queue;
   condition.signal();

   lock.Release();
}
```

```
RemoveFromQueue()
{
   lock.Acquire();

   while nothing on queue
     condition.wait(&lock);
              // release lock; go to
              // sleep; reacquire lock

   remove item from queue;
   lock->Release();
   return item;
}
```

Mesa-style vs. Hoare-style

- Mesa-style (Nachos, most real OS):
  - Signaler keeps lock, processor
  - Waiter simply put on ready queue, with no special priority
    (in other words, waiter may have to wait for lock again)
- Hoare-style (most theory, textbook):
  - Signaler passes lock, CPU to waiter; waiter runs immediately
  - Waiter gives lock, processor back to signaler when it exits critical section or if it waits again
- For Mesa-semantics, you always need to check the condition after wait (use “while”). For Hoare-semantics you can change it to “if”

Producer-consumer with semaphores

```
Semaphore fullBuffers = 0;      // initially no coke
Semaphore emptyBuffers = MAX_BUFFER;
          // initially, # of empty slots semaphore used to
          // count how many resources there are
Semaphore mutex = 1;            // no one using the machine

Producer() {
   emptyBuffers.P();        // check if there is space
                            //  for more coke
   mutex.P();               // make sure no one else
                            // is using machine

   put 1 Coke in machine;

   mutex.V();               // ok for others to use machine
   fullBuffers.V();         // tell consumers there is now
                            // a Coke in the machine
}
```

```
Consumer() {
   fullBuffers.P();         // check if there is
                            // a coke in the machine
   mutex.P();               // make sure no one
                            // else is using machine

   take 1 Coke out;

   mutex.V();               // next person's turn
   emptyBuffers.V();        // tell producer
                            // we need more
}
```

Producer-consumer with monitors

```
Condition full;
Condition empty;
Lock lock;

int numInBuffer = 0;

Producer() {
   lock.Acquire();

   while (numInBuffer == MAX_BUFFER)
          full.wait(&lock);

   put 1 Coke in machine; numInBuffer++;

   empty.signal();
   lock.Release();
}
```

```
Consumer() {
   lock.Acquire();

   while (numInBuffer == 0)
          empty.wait(&lock);

   take 1 Coke;  numInBuffer--;

   full.signal();
   lock.Release();
}
```

Monitor Summary

General template for using monitors:

```
lock.Acquire();

while (!ready) {
    wait(cond);
}

lock.Release();
```

```
lock.Acquire();

ready = 1;

signal(cond);

lock.Release();
```

Issue 1:

- Wait = release lock; sleep; obtain lock
- "release lock + sleep" needs to be atomic

Thread T1

```
lock.Acquire();

ready = 1;

signal(cond);

lock.Release();
```

Thread T2

```
lock.Acquire();

while (!ready) {
    lock.Release();
    sleep on cond;
}

lock.Release();
```

Issue 2:

- If wait does not automatically acquire the lock when it returns, does that lead to errors?
- Is it ok for wait to be just an atomic “release lock + sleep”

Thread T1

```
lock.Acquire();

ready = 1;

signal(cond);

lock.Release();
```

Thread T2

```
lock.Acquire();

while (!ready) {
    wait(cond);
    lock.Acquire();
}

lock.Release();
```

Issue 3:

- Does the waker require mutex?

Thread T1

```
ready = 1;

signal(cond);
```

Thread T2

```
lock.Acquire();

while (!ready) {
    wait(cond);
}

lock.Release();
```

Issue 4:

- Is it correct to: change state with mutex, but signal without the lock?

Thread T1

```
lock.Acquire();

ready = 1;

lock.Release();

signal(cond);
```

Thread T2

```
lock.Acquire();

while (!ready) {
    wait(cond);
}

lock.Release();
```

Announcements

- Deadline reminders
  - Design spec for as1 due tomorrow
  - Review for Scheduler Activations due on Wednesday

Readers/writers problem

- Motivation
  - shared database (e.g., bank balances / airline seats)
  - Two classes of users:
    - Readers --- never modify database
    - Writers --- read and modify database
  - Using a single lock on the database would be overly restrictive
    - want many readers at the same time
    - only one writer at the same time
- Constraints
    - Readers can access database when no writers (Condition okToRead)
    - Writers can access database when no readers or writers (Condition okToWrite)
    - Only one thread manipulates state variable at a time

Design Specification

- Reader
  - wait until no writers
  - access database
  - check out - wake up waiting writer
- Writer
  - wait until no readers or writers
  - access database
  - check out --- wake up waiting readers or writer
- Lock and condition variables: **okToRead, okToWrite**

Solving readers/writers

```
Reader() {
  lock.Acquire();
  WR++;
  while (AW > 0)
     okToRead.Wait(&lock);
  WR--;
  AR++;
  lock.Release();

  Access DB;

  lock.Acquire();
  AR--;
  if (AR == 0 && WW > 0)
        okToWrite.Signal(&lock);
  lock.Release();

}
```

```
Writer() {
  lock.Acquire();
  WW++;
  while ((AW+AR) > 0)
     okToWrite.Wait(&lock);
  WW --;
  AW++;
  lock.Release();

  Access DB;

  lock.Acquire();
  AW--;
  if (WW > 0)  okToWrite.Signal(&lock);
    else if (WR > 0) okToRead.Broadcast(&lock);
  lock.Release();
}
```

One-way-bridge problem

- Problem definition
  - a narrow light-duty bridge on a public highway
  - traffic cross in one direction at a time
  - at most 3 vehicles on the bridge at the same time (otherwise it will collapse)
- Each car is represented as one thread:

```
OneVechicle (int direc)
{
   ArriveBridge(direc);
   ... cross the bridge ...
   ExitBridge(direc);
}
```

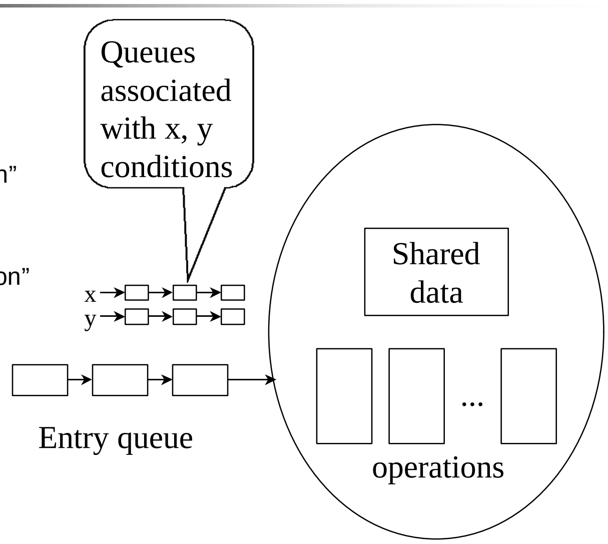
One-way bridge solution

```
Lock lock;
Condition safe;          // safe to cross bridge
int currentNumber;   // # of cars on bridge
int currentDirec;      // current direction

ArriveBridge(int direc) {
  lock.Acquire();
  while (! safe-to-cross(direc)) {
     safe.wait(lock)
  }
  currentNumber++;
  currentDirec = direc;
  lock.Release();
}
```

```
ExitBridge(int direc) {
   lock.Acquire();
   currentNumber--;
   safe.signal(lock);
   lock.Release();
}

safe-to-cross(int direc) {
    if (currentNumber == 0)
        return TRUE;     // always safe if empty
    else if ( (currentNumber < 3) &&
                (currentDirec == direc) )
        return TRUE;
    else
        return FALSE;
}
```

Implementing Monitors

- Wait()
  - Block on “condition”
- Signal()
  - Wakeup a blocked thread on “condition”



Implementing Monitors

- Can we use semaphores to implement condition variables?
- Simple attempt:

```
Wait()     {   semaphore->P();   }
Signal()   {   semaphore->V();   }
```

- Solution is not relinquishing the lock:

```
lock.Acquire();

while (!condition)
    Wait();

lock.Release();
```

```
lock.Acquire();

Signal();

lock.Release();
```

Second Attempt

- Use one semaphore for each condition variable
- Release the lock during wait:

```
Wait(Lock *lock) {
    lock->Release();
    semaphore->P();
    lock->Acquire();
}
Signal()  {   semaphore->V();   }
```

- Is this solution correct?

Peek at the waiting queue

- Perform a check during signal:

```
Wait(Lock *lock) {
    lock->Release();
    semaphore->P();
    lock->Acquire();
}
Signal()
{
    if semaphore queue is not empty
        semaphore->V();
}
```

- Well, it is cheating! But is it correct?

Implementing Monitors

Using one semaphore for each waiting thread --- making sure it indeed gets the message when it is signalled.

```
class Condition { List waitQueue; }

Condition::Wait(Lock* lock)   {
    Semaphore *w;

    w = new Semaphore (0);
    add w to the waitQueue;
    lock->Release();
    w->P();
    lock->Acquire();
    delete w;
}
```

```
Condition::Signal(Lock* lock) {
    Semaphore *w;

    if  anyone on waitQueue {
        Take a waiting element off
        and name it w;
        w->V();
    }
}
```

Announcements

- Sign up for design review meetings at the end of class
  - Meetings will take place tomorrow
- Read lottery scheduling paper for Friday
  - No review required

Multiprocessors & Parallel Programs

- Difficulties of developing parallel programs
  - Hard to design & debug
  - Application characteristics might limit parallelism
    - What is the inherent parallelism in the program?
  - Latency of communication and overheads
    - Threads need to communicate with each other
    - Suppose a thread creates a new task – is it easier to just execute it than passing the task to a different thread?

Approach #1

- Operating system level approach
- Create kernel threads
  - Communicate priorities to kernel
- Use kernel's communication and synchronization primitives
- But kernel threads are too expensive to create
  - Solution: keep a pool of kernel threads, reuse within application

- Problems:
  - Context switches are still slow
  - Kernel keeps lot more state around and is not aware of user-level program properties
  - Cannot customize the scheduling policy

Approach #2

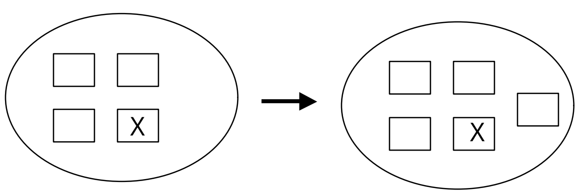
- One kernel thread for each processor in the system
- Implement user-level threads entirely at the user-level in the runtime system
  - Any user thread can run on any kernel thread
  - Very fast thread creation and context switch
  - Fast synchronization
- Can support much finer-grained parallelism
- Problem: Two schedulers!
  - What if a task does blocking I/O
    - Loses CPU
  - What if there are other applications running on the machine?
    - Application might lose a kernel thread at a "bad time"
  - Application might sometimes need fewer threads

Scheduler Activations

- Mechanism of communicating between the two schedulers
- Scheduler activation:
  - Vessel for running user threads (acts like a kernel thread)
  - Can think of it as a virtual processor
  - Notifies the user-level runtime system of interesting kernel events
  - Provides space for saving processor context of the currently running user thread when the thread is stopped in the kernel
- Old world: fixed # of kernel threads
  - New world: fixed number of “running” threads

Scenario 1

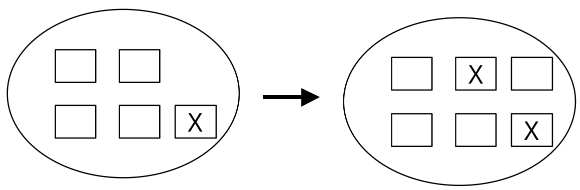
- Application has certain number of activations running
- If an activation is blocked:
  - New activation is created
  - Allows the user-level scheduler to run in this new activation



  - Runtime scheduler can schedule another user thread to run on the new activation

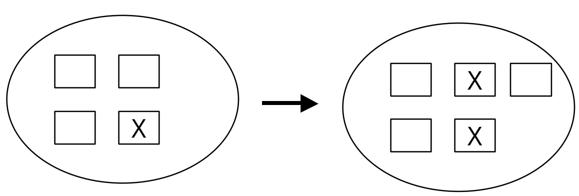
Scenario 2

- One of the blocked activations wakes up
- Kernel notifies the application
  - Preempt another activation
  - Create a new activation and tell the user level scheduler:
    - Previous activation is now unblocked
    - Existing activation has been preempted
  - User level scheduler decides what to run where
  - Has access to all of the register state



Scenario 3

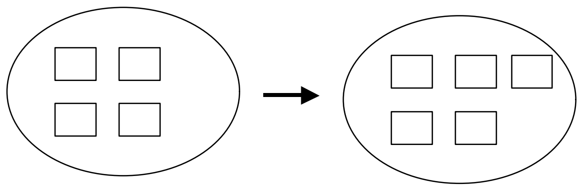
- Kernel takes away an activation
- Notifies the application
- Must preempt activation to report
- Same as before:



- User-level scheduler again makes a decision on which threads to run

Scenario 4

- New CPU becomes available
- Kernel creates a new activation



- User-level scheduler picks a new thread to execute on the new activation
- In addition, at any point the application can tell kernel it doesn't need extra CPUs

Summary

- Three key features about this approach:
  - Goal is the get user-level threads performance with the scheduling consistency provided by kernel-level threads in multiprocessors
  - The problem to solve: coordinating two independent thread schedulers
  - Scheduler activations used to transmit information between the two as well as to provide virtual processors
- Lesson: export your functionality (in this case, threads) out of the kernel for improved performance and flexibility
  - Figure out how to interact with the kernel "just enough"