Semaphores & Monitors

Arvind Krishnamurthy
Spring 2004

Semaphores (Dijkstra 1965)

- Semaphores have a non-negative integer value, and support two operations:
  - **semaphore->P():** an atomic operation that waits for semaphore to become positive, then decrements it by 1
  - **semaphore->V():** an atomic operation that increments semaphore by 1, waking up a waiting P, if any.
- Semaphores are like integers except:

  (1) non-negative values;

  (2) only allow P&V --- can't read/write value except to set it initially;

  (3) operations must be atomic:

  -- two P's that occur together can't decrement the value below zero.

  -- thread going to sleep in P won't miss wakeup from V,
  even if they both happen at about the same time.

Implementing semaphores

P means “test” (proberen in Dutch)
V means “increment” (verhogen in Dutch)

- Binary semaphores:
  - Like a lock; can only have value 0 or 1 (unlike the previous “counting semaphore” which can be any non-negative integers)
- How to implement semaphores? Standard tricks:
  - Can be built using interrupts disable/enable
  - Or using test-and-set
    - Use a queue to prevent unnecessary busy-waiting
  - Question: Can we build semaphores using just locks?

Using interrupts

```
class Semaphore { int value = 0; }

Semaphore::P()  {
   Disable interrupts;
   while (value == 0) {
     Enable interrupts;
     Disable interrupts;
   }
   value = value - 1;
   Enable interrupts;
}
```

```
Semaphore::V()  {
   Disable interrupts;
   value++;
   Enable interrupts;
}
```

Using test&set

```
class Semaphore { int value = 0;
                  int guard = 0; }


Semaphore::P()  {
  while (test&set(guard))  // short wait time
      ;

  if (value == 0) {
     Put on queue of threads waiting for lock;
     Go to sleep and set guard to 0
  } else {
     value = value - 1;
     guard = 0;
  }
}
```

```
Semaphore::V()  {
   while (test&set(guard))
         ;
   if  anyone on wait queue {
      Take a waiting thread off wait
      queue and put it at the front
      of the ready queue;
   } else {
      value = value + 1;
   }
   guard = 0;
}
```

How to use semaphores

- Binary semaphores can be used for mutual exclusion:

  initial value of 1; P() is called before the critical section; and V() is called after the critical section.

  ```
  semaphore->P();
  // critical section goes here
  semaphore->V();
  ```

- Scheduling constraints
  - having one thread to wait for something to happen
    - Example: Thread::Join, which must wait for a thread to terminate. By setting the initial value to 0 instead of 1, we can implement waiting on a semaphore
- Controlling access to a finite resource

Scheduling constraints

- Some thread must wait for some event
- For instance, thread join can be implemented using semaphores

```
Initial value of semaphore = 0;
Fork a child thread
Thread::Join calls P      // will wait until something
                          // makes the semaphore positive
-----------------------------------------------------------------------

Thread finish calls V     // makes the semaphore positive
                          // and wakes up the thread
                          // waiting in Join
```

- Question: Can we implement thread join with just locks and unlocks?

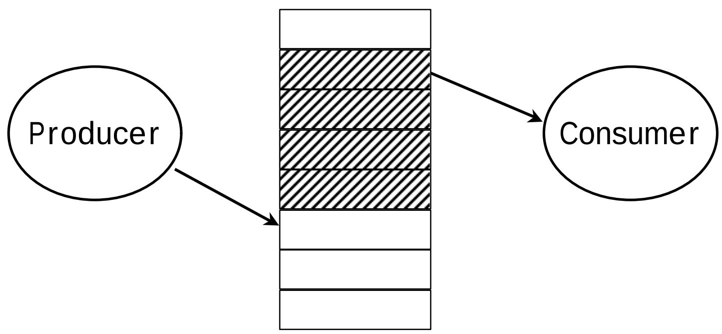
Scheduling with Semaphores

- In general:
  - scheduling dependencies between threads T1, T2, …, Tn can be enforced with n-1 semaphores, S1, S2, …, Sn-1
  - T1 runs and signals V(S1) when done.
  - Tm waits on Sm-1 (using P) and signals V(Sm) when done.
- Contrived example: schedule print(f(x,y))

```
float  x, y, z;
sem   Sx = 0, Sy = 0, Sz = 0;
T1:              T2:              T3:
  x = …;           P(Sx);           P(Sz);
  V(Sx);           P(Sy);           print(z);
  y = …;           z = f(x,y);      …
  V(Sy);           V(Sz);
  …                …
```

Example: producer-consumer with a bounded buffer

- Example:
  - cpp file.S | as



Producer-consumer: problem definition

- Producer puts things into a shared buffer; consumer takes them out.
  - Need synchronization for coordinating producer and consumer
- Don't want producer and consumer to have to operate in lockstep
  - so put a fixed-size buffer between them
  - need to synchronize access to this buffer
  - Producer needs to wait if buffer is full
  - Consumer needs to wait if buffer is empty
  - Semaphores are used for both mutex and scheduling
- Example coke vending machine:
  - Consumers are students/faculty
  - Producer is the delivery person

Producer-consumer with semaphores (1)

- Correctness constraints
  - consumer must wait for producer to fill buffers, if all empty (scheduling constraint)
  - producer must wait for consumer to empty buffers, if all full (scheduling constaint)
  - Only one thread can manipulate buffer queue at a time (mutual exclusion)

- General rule of thumb: use a separate semaphore for each constraint

```
Semaphore fullBuffers;      // consumer's constraint
                            // if 0, no coke in machine
Semaphore emptyBuffers;     // producer's constraint
                            // if 0, nowhere to put more coke
Semaphore mutex;            // mutual exclusion
```

Announcements

- Zheng Ma's office hours
- Design document structure and scope
- Paper review for "Scheduler Activations" paper: next Wednesday

Monitors & condition variables

- Locks provide mutual exclusion to shared data
- Semaphores help handle scheduling constraints

- Semaphore utility is overloaded:
  - dual purpose: mutual exclusion and scheduling constraints.

- Monitors make things easier:
  - "locks" for mutual exclusion
  - "condition variables" for scheduling constraints

- Monitor definition:
  - *a lock and zero or more condition variables for managing concurrent access to shared data*

Synchronized Lists

```
AddToQueue()
{
    lock.Acquire();          // lock before use
    put item on queue;       // ok to access
    lock.Release();          // unlock after done
}
```

```
RemoveFromQueue()
{
    lock.Acquire();
    if something on queue          // can we wait?
        remove it;
    lock->Release();
    return item;
}
```

- With semaphores, you could maintain a counter on number of elements in the list
  - Perform a semaphore-decrement on the counter before trying to obtain the lock
  - What if you wanted to support a "peek" operation on the list? Multiple threads could be waiting for an element to appear; need to wake them all up
  - What if a thread wants to wait for a general non-counter based program condition
  - Can be done using semaphores, but would like a better high level construct

Condition variables

- How to make RemoveFromQueue wait until something is on the queue?
  - can't sleep while holding the lock
  - Key idea: make it possible to go to sleep inside critical section, by atomically releasing lock at same time we go to sleep.

- **Condition variable**: *a **queue** of threads waiting for something inside a critical section.*
  - **Wait()** --- Release lock, go to sleep, re-acquire lock
    - release lock and going to sleep is atomic
  - **Signal()** --- Wake up a waiter, if any
  - **Broadcast()** --- Wake up all waiters

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 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();
}
```

Monitors Support in Languages

- High-level data abstraction that unifies handling of:
  - Shared data, operations on it, synch and scheduling
    - All operations on data structure have single (implicit) lock
    - An operation can relinquish control and wait on condition

```
// only one process at time can update instance of Q
class Q {
        int head, tail;  // shared data
        synchronized void enq(v) { locked access to Q instance }
        synchronized int deq() { locked access to Q instance }
}
```

  - Java from Sun; Mesa/Cedar from Xerox PARC

- Monitors easier and safer than semaphores
  - Compiler can check, lock implicit (cannot be forgotten)