

Lecture #8: Semaphores and Bounded buffer

Review -- 1 min

- Hardware support for synchronization
- Building higher-level synchronization programming
- abstractions on top of hardware support (e.g., Lock)

Outline - 1 min

Definition of semaphore

Example of programming w. semaphore

Semaphore expresses 2 types of synchronization

- mutex (like lock)
- synchronization (wait for some event)

Simple implementation (time permitting)

Preview - 1 min

other programming abstractions (e.g., condition variables/monitor)

Lecture - 32 min

1. Motivation

writing concurrent programs hard – coordinate updates to shared memory

synchronization – coordinating multiple concurrent activities that are using shared state

Question: what are the right synchronization abstractions to make it easy to build concurrent programs?

Answer will necessarily be a compromise :

- between making it easy to modify shared variables any time you want and controlling when you can modify shared variables.
- between really flexible primitives that can be used in a lot of different ways and simple primitives that can only be used one way (but are more difficult to misuse)

Rules will seem a bit strange – why one definition and not another?

- no absolute answer
- history has shown that they are reasonably good – if you follow these definitions, you will find writing correct code easier.
- for now just take them as a given; use it for a while; then, if you can come up with something better, be my guest!

2. 2 “types” of synchronization

Convenient to break synchronization into two cases

- (1) **Mutual exclusion** – only allow one thread to access a given set of shared state at a time
Each shared object has lock and shared state variables
Public methods acquire the lock before reading/writing member state variables
- (2) **Scheduling constraints** – wait for some other thread to do something
e.g., wait for other thread to finish, wait for other thread to produce work, wait for other thread to consume work, wait for other thread to accept a connection, wait for other thread to get bytes off disk, ...

3. Definition of Semaphores

like a generalized lock
first defined by Dijkstra in late 60's
originally main synchronization primitive in Unix (now others available)

semaphore – has a positive integer value and supports the following two operations:
 semaphore->P() – an atomic operation that waits for the semaphore to become positive; then decrements it by 1
 semaphore->V() – an atomic operation that increments the semaphore by 1, waking up a waiting P if any

Like integers, except:

- 1) No negative values
- 2) Only operations are P() and V() – can't read or write the value (except to set it initially)
- 3) operations must be atomic – two P's that occur together can't decrement the value below zero. Similarly, thread going to sleep in P won't miss wakeup from V, even if they both happen at about the same time

binary semaphore – instead of an integer value, has a boolean value. P waits until value is 1, then sets it to 0
 V sets value to 1, waking up a waiting P if any

4. Two uses of semaphores

4.1 mutual exclusion

When semaphores are used for mutual exclusion, the semaphore has an initial value of 1, and P() is called before the critical section, and V() is called after the critical section

```
semaphore = new Semaphore(1);
...
semaphore->P();
// critical section goes here
semaphore->V();
```

4.2 scheduling constraints

semaphores can be used to describe general scheduling constraints – e.g. they provide a way to wait for something

usually in this case (but not always) the initial value for the semaphore is 0

Example: Wait for another thread to get done processing a request

```
*****
```

```
Admin - 3 min
```

```
*****
```

```
*****
```

```
Lecture - 30 min
```

```
*****
```

5. Producer-consumer with bounded buffer

5.1 problem definition

producer puts things into a shared buffer
 consumer takes them out

need synchronization for coordinating producer and consumer

e.g. `cpp | cc1 | cc2 | as`
 e.g., read/write network/disk (e.g., web server reads from disk, sends to network while your web client reads from network and draws to screen)

Don't want producer and consumer to operate in lock-step, so put a fixed sized **buffer** between them.
 Synchronization – producer must wait if buffer is full; consumer must wait if buffer is empty

e.g. coke machine
 producer is delivery person
 consumer is students and faculty

Notice: *shared object* (coke machine) *separate from threads* (delivery person, students, faculty). Shared object coordinates activity of threads.

Common confusion on project – try to do the synchronization within the threads' code. No, the synchronization happens within the shared objects. "Let the shared objects do the work."

Solution uses semaphores for both mutex and scheduling

5.2 Correctness constraints for solution

Synchronization problems have semaphores represent 2 types of constraint

- *mutual exclusions*
- *wait for some event*

When you start working on a synchronization problem, first define the mutual exclusion constraints, then ask "when does a thread wait", and create a separate synchronization variable representing each constraint

QUESTION: what are the constraints for bounded buffer?

- 1) only one thread can manipulate buffer queue at a time
mutual exclusion
- 2) consumer must wait for producer to fill buffers if none full
scheduling constraint
- 3) producer must wait for consumer to empty buffers if all full
scheduling constraint

Use a separate semaphore for each constraint

```
Semaphore mutex;
Semaphore fullBuffers; // consumer's constr
                        // if 0 no coke
```

```
Semaphore emptyBuffers; // producer's constr.
                        // if 0, nowhere to put more coke
```

5.3 Solution

```
Class CokeMachine{

Semaphore new mutex(1); // no one using machine
Semaphore new fullBuffers(0); // initially no coke!
Semaphore new emptyBuffers(numBuffers);
                        // initially # empty slots
                        // semaphore used to count how many
                        // resources there are

Produce(Coke *coke){
  emptyBuffers.P(); // check if there is space
                  // for more coke
  mutex.P();       // make sure no one else
                  // 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 machine
}

Coke *Consume(){
  fullBuffers.P(); // check if there's a coke
  mutex.P();       // make sure no one else
                  // using the machine
  coke = take a coke out
  mutex.V();       // next person's turn
  emptyBuffers.V(); // tell producer we're
                  // ready for more
  return coke;
}
}
```

5.4 Questions

Why does producer P and V different semaphores than consumer?

Is order of Ps important?

Is order of V's important?

What if we have 2 producers or 2 consumers? Do we need to change anything?

6. implementing semaphores

last time: implement locks by turning off interrupts (or test&set)

Question: how would you implement semaphores? (let's solve problem with the "turning off interrupts" technique:

Here was lock code:

```
member variables:
    int value
    queue *queue;
```

```
Lock::Lock()
    value = FREE;
    queue = new Queue();
```

```
Lock::Acquire()
    disable interrupts
    if (value == BUSY)
        put thread's TCB on queue of threads
        waiting for lock
        switch
    else
        value = BUSY
    enable interrupts
```

```
Lock::Release()
    disable interrupts
    if anyone on wait queue{
        take a waiting thread's TCB off queue
        put it on ready queue
```

```
else
    value = FREE;
    enable interrupts
```

Fill in the semaphore code:
Member variables:

```
Semaphore::Semaphore() // constructor
```

```
Semaphore::P()
//
// Thread that calls P() should wait for the
// semaphore to become positive and then
// decrement it by 1
//
```

```
Semaphore::V()
//
// A thread that calls V() should increment
// the semaphore by 1, waking up a thread
// waiting in P() if any
//
```

Summary - 1 min

2 types of synchronization

mutual exclusion

scheduling/waiting

semaphore can be used for both (is this good?)

Semaphore operations

P()

V()

Note: you can't ask the value of a semaphore – only can do P()
and V()

Semaphore built on same hardware primitives as lock using
essentially same techniques