Patterns based on Semaphores

CS511

Review of Semaphores

- An Abstract Data Type with two operations
  - `acquire`
  - `release`
- Can be used to solve the mutual exclusion problem
- Can be used to synchronize cooperative threads

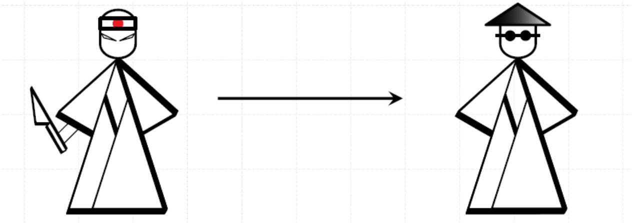
Today

- Recurring problems in the area
- Proven solution templates

Producers/Consumers

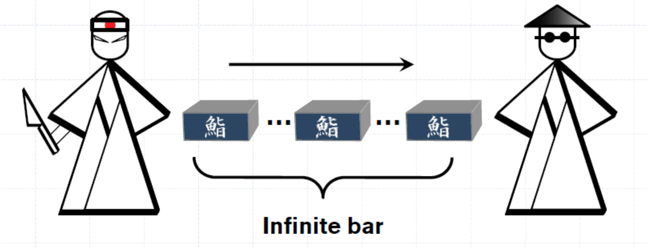
Readers/Writers

Producers/consumers



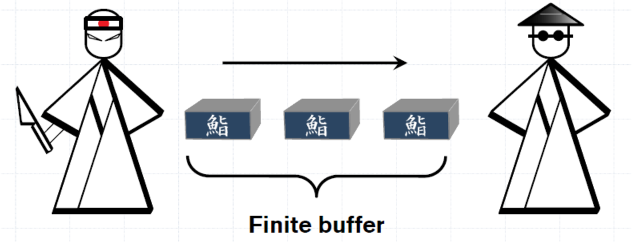
- A commmon pattern of interaction
- Must cater for difference in speed between each party

Unbounded Buffer



- The producer can work freely
- The consumer must wait for the producer to produce

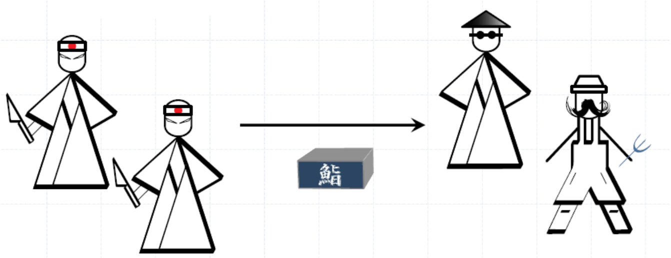
Bounded Buffer



- The producer must wait when the buffer is full
- The consumer must wait for the producer to produce

Buffer using Semaphores

- Capacity 1



- Various producers
- Various consumers
- Semaphores

Buffer using Semaphores – 1 producer and 1 consumer

```
Object buffer;
...
...
```

```
Thread.start { // Prod
  while (true) {
    ...
    buffer = produce();
    ...
  }
```

```
Thread.start { // Cons
  while (true) {
    ...
    consume(buffer);
    ...
  }
```

Split Binary Semaphores

- Two semaphores
  - one to hold permissions to produce
  - one to hold permissions to consume
- Initialization
  - `produce = 1`
  - `consume = 0`
- Invariant
  - `produce + consume <= 1`

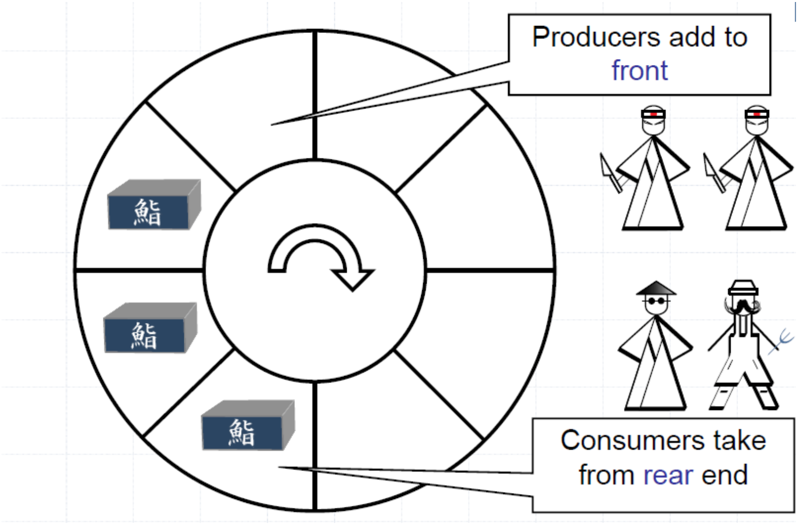
Split Binary Semaphores

```
Object buffer;
Semaphore produce = new Semaphore(1);
Semaphore consume  = new Semaphore(0);
```

```
Thread.start { // Prod
  while (true) {
    produce.acquire();
    buffer = produce();
    consume.release();
  }
```

```
Thread.start { // Cons
  while (true) {
    consume.acquire();
    consume(buffer);
    produce.release();
  }
```

N Size Buffer



General Semaphores

- Semaphores count the number of empty slots in the buffer
- Initialization
  - There are N empty slots
  - There are 0 full slots
- Invariant
  - `produce + consume <= N`

Unique Producer/Consumer

```
Object[]  buffer  = new Object[N];
Semaphore produce = new Semaphore(N);
Semaphore consume = new Sempahore(0);
int start = 0;
int end   = 0;
```

```
Thread.start { // Prod
  while (true) {
    ...
    Object item = produce();
    buffer[start] = item;
    start = (start+1) % N;
    ...
  }
```

```
Thread.start{ // Cons
  while (true) {
    ...
    Object item = buffer[end];
    end = (end+1) % N;
    consume(item);
    ...
  }
```

Unique Producer/Consumer

```java
Object[]  buffer = new Object[N];
Semaphore produce  = new Semaphore(N);
Semaphore consume   = new Sempahore(0);
int start = 0;
int end   = 0;
```

```java
Thread.start{ // Prod
  while (true) {
    produce.acquire();
    Object item = produce();
    buffer[start] = item;
    start = (start+1) % N;
    consume.release();
  }
```

```java
Thread.start { // Cons
  while (true) {
    consume.acquire();
    Object item = buffer[end];
    end = (end+1) % N;
    consume(item);
    produce.release();
  }
```

Multiple Producers

- We cannot simply add multiple instances of the producer
- Why? Justify with a trace
- What can we do about it?

```
// declarations: same as above...
```

```
Thread.start{ // ProdA
  while (true) {
    produce.acquire();
    Object item = produce();
    buffer[start] = item;
    start = (start+1) % N;
    consume.release();
  }
```

```
Thread.start { // ProdB
  while (true) {
    produce.acquire();
    Object item = produce();
    buffer[start] = item;
    start = (start+1) % N;
    consume.release();
  }
```

Multiple Producers

- Must guarantee mutual exclusion between producers:
  - We add a new semaphore

```
Semaphore mutexP = new Semaphore(1);

Thread.start { // Prod
  while (true) {
    produce.acquire();
    Object item = produce();
    mutexP.acquire();
    buffer[start] = item;
    start = (start+1) % N;
    mutexP.release();
    consume.release();
  }
}
```

Multiple Consumers

- Must guarantee mutual exclusion between consumers

```
Semaphore mutexC = new Semaphore(1);

Thread.start { // Cons
  while (true) {
    consume.acquire();
    mutexC.acquire();
    Object item = buffer[end];
    end = (end+1) % N;
    mutexC.release();
    consume(item);
    produce.release();
  }
}
```

Putting it all together

```
import java.util.concurrent.Semaphore

final int N = 10;
buffer = new int[N];
permToProduce  = new Semaphore(N);
permToConsume   = new Semaphore(0);
mutexP    = new Semaphore(1);
mutexC    = new Semaphore(1);
start = 0;
end   = 0;

5.times {
    int id = it
    Thread.start { // Consumer(id)
        while (true) {
            permToConsume.acquire();
            mutexC.acquire();
            int item = buffer[end];
            println(id+" consumed product "+ buffer[end] + " at "+
            end = (end+1) % N;
            mutexC.release();
            permToProduce.release();
            // consumeItem(item);
        }
    }
}
```

Putting it all together

```
5.times {
    int id = it;
    Thread.start { // Producer(id) {
        Random r = new Random()
        while (true) {
            int item = r.nextInt(10000) // produceItem();
            permToProduce.acquire();
            mutexP.acquire();
            buffer[start] = item;
            println(id+" added product "+ buffer[start]+ " at "+ s
            start = (start+1) % N;
            mutexP.release();
            permToConsume.release();
        }
    }
}
```

Producers/Consumers

Readers/Writers

Readers/Writers

- There are shared resources between two types of threads
  - Readers: access the resource without modifying it
  - Writers: access the resource and may modify it
- Mutual exclusion is too restrictive
  - Readers: can access simultaneously
  - Writers: at most one at any given time

Properties a Solution should Possess

- Each read/write operation should occur inside the critical region
- Must guarantee mutual exclusion between the writers
- Must allow multiple readers to execute inside the critical region simultaneously

First Solution: Priority Readers

```
Writer() {

  ...
  write();
  ...

}
```

```
Reader() {

  ...
  read();
  ...

}
```

First Solution: Priority to Readers

- One semaphore for controlling write access
- Before writing, the permission must be obtained and then released when done
- The first reader must “steal” the permission to write and the last one must return it
  - We must count the number of readers inside the CS
  - This must be done inside its own CS

First Solution: Prioity Readers

```
Semaphore resource = new Semaphore(1);
Semaphore numReadersMutex  = new Semaphore(1);
int numReaders = 0;
```

```
Writer() {
  resource.acquire();
  write();
  resource.release();
}
```

```
Reader() {
  numReadersMutex.acquire();
  numReaders++;
  if (numReaders == 1)
    resource.acquire();
  numReadersMutex.release();

  read();

  numReadersMutex.acquire();
  numReaders--;
  if (numReaders == 0)
    resource.release();
  numReadersMutex.release();
}
```

Note: Is this solution free from starvation?

Second Solution: Priority Writers

- The readers can potentially lock out all the writers
  - We need to count the number of writers that are waiting
  - Also, this counter requires its own CS
- Before reading the readers must obtain a permission to do so

Second Solution: Priority Writers

```
Writer() {
  numWritersMutex.acquire();
  numWriters++;
  if (numWriters == 1)
    readTry.acquire();
  numWritersMutex.release();

  resource.acquire();
  write();
  resource.release();

  numWritersMutex.acquire();
  numWriters--;
  if (numWriters == 0)
    readTry.release();
  numWritersMutex.release();
}
```

```
Reader() {
  readTry.acquire();
  numReadersMutex.acquire();
  numReaders++;
  if (numReaders == 1)
    resource.acquire();
  numReadersMutex.release();
  readTry.release();

  read();

  numReadersMutex.acquire();
  numReaders--;
  if (numReaders == 0)
    resource.release();
  numReadersMutex.release();
}
```

- Readers might starve
- Solution in which neither readers nor writers starve?
  - Hint: Common service queue for both readers and writers

Third Solution

```
int numReaders;
Semaphore resource = new Semaphore(1);
Semaphore readCountAccess = new Semaphore(1);
Semaphore serviceQueue = new Semaphore(1,true);
```

```
Writer() {
  serviceQueue.acquire();
  resource.acquire();
  serviceQueue.release();

  writeResource();

  resource.release();
}
```

```
Reader() {
  serviceQueue.acquire();
  readCountAccess.acquire();
  readCount++;
  if (readCount == 1)
      resource.acquire();
  readCountAccess.release();
  serviceQueue.release();

  readResource();

  readCountAccess.acquire();
  readCount--;
  if (readCount == 0)
      resource.release();
  readCountAccess.release();
}
```

Summary

1. Semaphores are elegant and efficient for solving problems in concurrent programs
2. Still, they are low-level constructs since they are not structured
3. Monitors will provide synchronization by encapsulation