

# 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

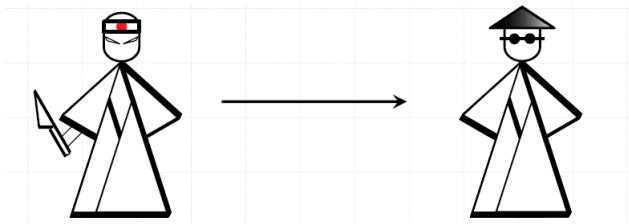
# Today

- ▶ Recurring problems in the area
- ▶ Proven solution templates

Producers/Consumers

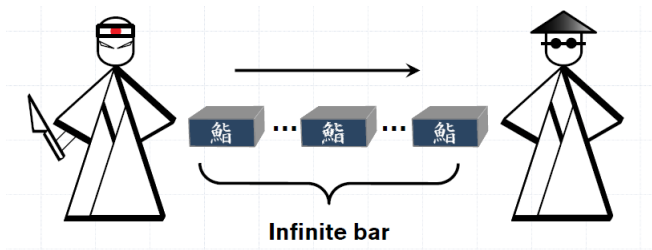
Readers/Writers

# Producers/consumers



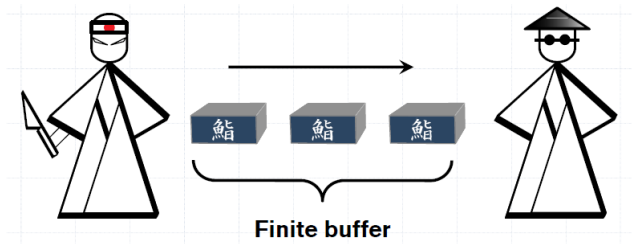
- ▶ A common pattern of interaction
- ▶ Must cater for difference in speed between each party

# Unbounded Buffer



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

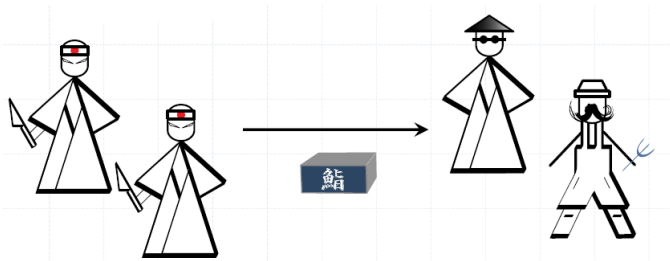
# Bounded Buffer



- ▶ 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

```
1 Integer buffer;
2 ...
3 ...

4 Thread.start { // Prod 4 Thread.start { // Cons
5     while (true) {      5     while (true) {
6         ...              6         ...
7         buffer = produce(); 7         consume(buffer);
8         ...              8         ...
9     }                    9     }
```

# Split Binary Semaphores

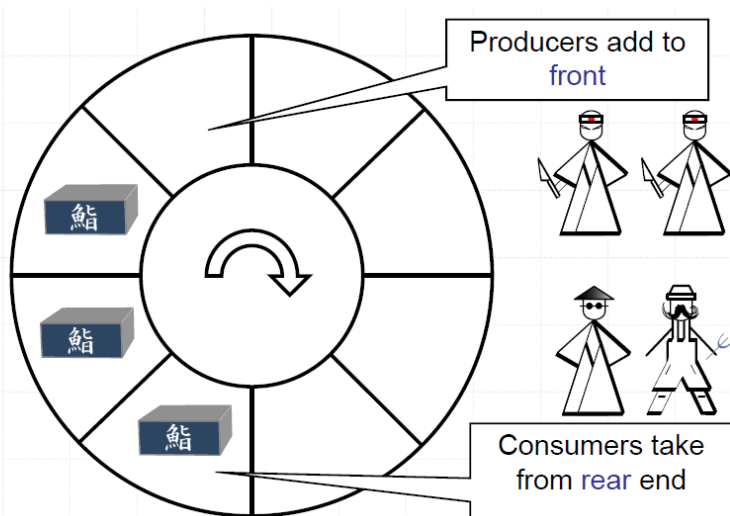
- ▶ Two semaphores
  - ▶ one to hold permissions to produce
  - ▶ one to hold permissions to consume
- ▶ Initialization
  - ▶ `produce = 1`
  - ▶ `consume = 0`
- ▶ Invariant
  - ▶ `produce + consume <= 1`

## Split Binary Semaphores

```
1 Integer buffer;
2 Semaphore produce = new Semaphore(1);
3 Semaphore consume = new Semaphore(0);

4 Thread.start { // Prod 4 Thread.start { // Cons
5     while (true) {      5     while (true) {
6         produce.acquire(); 6         consume.acquire();
7         buffer = produce(); 7         consume(buffer);
8         consume.release(); 8         produce.release();
9     }                    9     }
```

# 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
  - ▶  $\text{produce} + \text{consume} \leq N$

# Unique Producer/Consumer

```
1 Integer[]  buffer  = new Integer[N];
2 Semaphore produce = new Semaphore(N);
3 Semaphore consume = new Semaphore(0);
4 int start = 0;
5 int end    = 0;

6 Thread.start { // Prod          6 Thread.start{ // Cons
7     while (true) {              7     while (true) {
8         ...                      8         ...
9         Integer item = produce(); 9         Integer item = buffer[end];
10        buffer[start] = item;      10        end = (end+1) % N;
11        start = (start+1) % N;    11        consume(item);
12        ...                      12        ...
13    }                            13    }
```

# Unique Producer/Consumer

```
1 Integer[]  buffer = new Integer[N];
2 Semaphore produce = new Semaphore(N);
3 Semaphore consume = new Semaphore(0);
4 int start = 0;
5 int end = 0;

6 Thread.start{ // Prod
7     while (true) {
8         produce.acquire();
9         Integer item = produce();
10        buffer[start] = item;
11        start = (start+1) % N;
12        consume.release();
13    }

6 Thread.start { // Cons
7     while (true) {
8         consume.acquire();
9         Integer item = buffer[end];
10        end = (end+1) % N;
11        consume(item);
12        produce.release();
13    }
```

# Multiple Producers

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

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

```
6 Thread.start{ // ProdA
7   while (true) {
8     produce.acquire();
9     Integer item = produce();
10    buffer[start] = item;
11    start = (start+1) % N;
12    consume.release();
13  }
```

```
6 Thread.start { // ProdB
7   while (true) {
8     produce.acquire();
9     Integer item = produce();
10    buffer[start] = item;
11    start = (start+1) % N;
12    consume.release();
13  }
```



# Multiple Producers

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

```
1 Semaphore mutexP = new Semaphore(1);
2
3 Thread.start { // Prod
4     while (true) {
5         produce.acquire();
6         Integer item = produce();
7         mutexP.acquire();
8         buffer[start] = item;
9         start = (start+1) % N;
10        mutexP.release();
11        consume.release();
12    }
13 }
```

# Multiple Consumers

- ▶ Must guarantee mutual exclusion between consumers

```
1 Semaphore mutexC = new Semaphore(1);
2
3 Thread.start { // Cons
4     while (true) {
5         consume.acquire();
6         mutexC.acquire();
7         Integer item = buffer[end];
8         end = (end+1) % N;
9         mutexC.release();
10        consume(item);
11        produce.release();
12    }
13 }
```

# Putting it all together

```
1 import java.util.concurrent.Semaphore
2
3 final int N = 10;
4 buffer = new int[N];
5 permToProduce = new Semaphore(N);
6 permToConsume = new Semaphore(0);
7 mutexP = new Semaphore(1);
8 mutexC = new Semaphore(1);
9 start = 0;
10 end = 0;
11
12 5.times {
13     int id = it
14     Thread.start { // Consumer(id)
15         while (true) {
16             permToConsume.acquire();
17             mutexC.acquire();
18             int item = buffer[end];
19             println(id+" consumed product "+ buffer[end] + " at "+
20                 end = (end+1) % N;
21             mutexC.release();
22             permToProduce.release();
23             // consumeItem(item);
24         }
25     }
26 }
```

## Putting it all together

```
1 5.times {
2   int id = it;
3   Thread.start { // Producer(id) {
4       Random r = new Random()
5       while (true) {
6           int item = r.nextInt(10000) // produceItem();
7           permToProduce.acquire();
8           mutexP.acquire();
9           buffer[start] = item;
10          println(id+" added product "+ buffer[start]+ " at "+ s
11          start = (start+1) % N;
12          mutexP.release();
13          permToConsume.release();
14      }
15  }
16 }
```

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

```
1 Thread.start { // Writer
2
3   ...
4   write();
5   ...
6
7 }
```

```
1 Thread.start { // Reader
2
3   ...
4   read();
5   ...
6
7 }
```



## 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: Priority Readers

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

1 Thread.start{ // Writer
2     resource.acquire();
3     write();
4     resource.release();
5 }

1 Thread.start { // Reader
2     numReadersMutex.acquire();
3     numReaders++;
4     if (numReaders == 1)
5         resource.acquire();
6     numReadersMutex.release();
7
8     read();
9
10    numReadersMutex.acquire();
11    numReaders--;
12    if (numReaders == 0)
13        resource.release();
14    numReadersMutex.release();
15 }
```

**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

```
1 Thread.start { // Writer
2   numWritersMutex.acquire();
3   numWriters++;
4   if (numWriters == 1)
5     readTry.acquire();
6   numWritersMutex.release();
7
8   resource.acquire();
9   write();
10  resource.release();
11
12  numWritersMutex.acquire();
13  numWriters--;
14  if (numWriters == 0)
15    readTry.release();
16  numWritersMutex.release();
17 }

1 Thread.start { // Reader
2   readTry.acquire();
3   numReadersMutex.acquire();
4   numReaders++;
5   if (numReaders == 1)
6     resource.acquire();
7   numReadersMutex.release();
8   readTry.release();
9
10  read();
11
12  numReadersMutex.acquire();
13  numReaders--;
14  if (numReaders == 0)
15    resource.release();
16  numReadersMutex.release();
17 }
```

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

## Third Solution

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

1 Thread.start { // Writer
2
3
4 serviceQueue.acquire();
5 resource.acquire();
6 serviceQueue.release();
7
8
9
10 writeResource();
11
12 resource.release();
13 }

1 Thread.start { // Reader
2 serviceQueue.acquire();
3 readCountAccess.acquire();
4 readCount++;
5 if (readCount == 1)
6     resource.acquire();
7 readCountAccess.release();
8 serviceQueue.release();
9
10 readResource();
11
12 readCountAccess.acquire();
13 readCount--;
14 if (readCount == 0)
15     resource.release();
16 readCountAccess.release();
17 }
```

# 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