

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

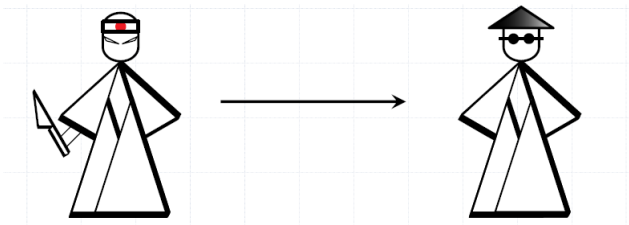
Today

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

Producers/Consumers

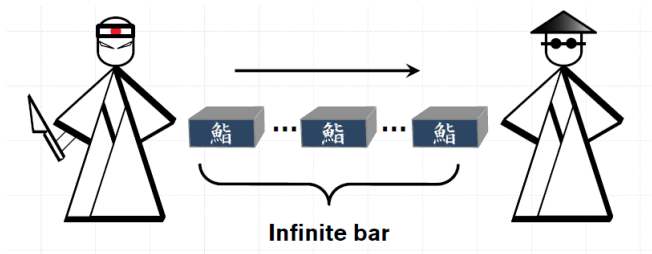
Readers/Writers

Producers/consumers



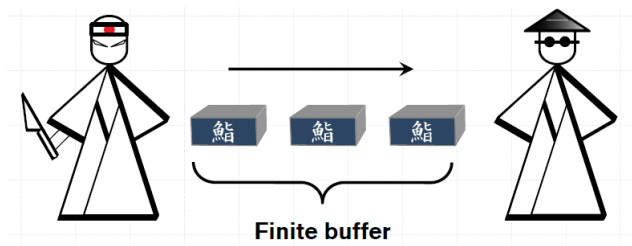
- ▶ A common pattern of interaction
- ▶ Must cater for difference in speed between each party

Unbounded Buffer



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

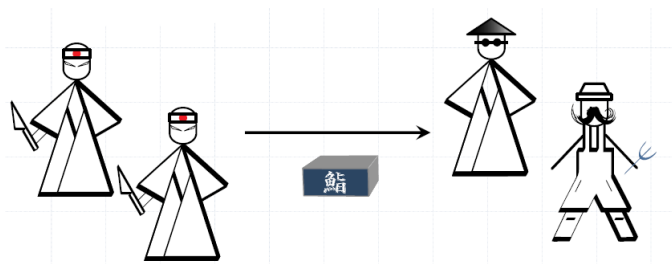
Bounded Buffer



- ▶ 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 global Object buffer;
2 ...
3 ...

4 thread Producer:           4 thread Consumer:
5     while (true) {         5     while (true) {
6         ...                 6         ...
7         buffer = produce(); 7         consume(buffer);
8         ...                 8         ...
9     }                       9     }
```

Split Binary Semaphores

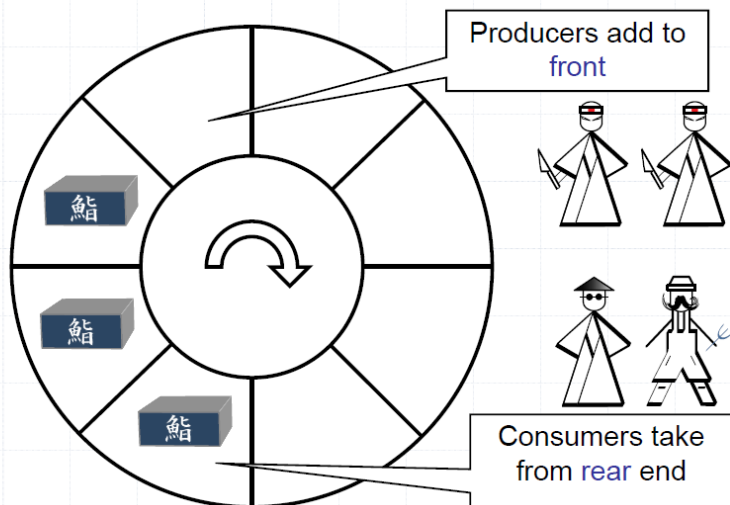
- ▶ Two semaphores
 - ▶ 1 to indicate when the buffer is empty
 - ▶ 1 to indicate when the buffer is full
- ▶ Initialization
 - ▶ `empty = 1`
 - ▶ `full = 0`
- ▶ Invariant
 - ▶ `empty + full <= 1`

Split Binary Semaphores

```
1 global Object buffer;
2 global Semaphore empty = new Semaphore(1);
3 global Semaphore full = new Semaphore(0);

4 thread Producer:          4 thread Consumer:
5   while (true) {          5   while (true) {
6     empty.acquire();      6     full.acquire();
7     buffer = produce();   7     consume(buffer);
8     full.release();       8     empty.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
 - ▶ $\text{empty} + \text{full} \leq N$

Unique Producer/Consumer

```
1 global Object[]  buffer = new Object[N];
2 global Semaphore empty  = new Semaphore(N);
3 global Semaphore full   = new Semaphore(0);
4 global int start = 0;
5 global int end    = 0;

6 thread Producer:
7     while (true) {
8         ...
9         buffer[start] = produce();
10        start = (start+1) % N;
11        ...
12    }

6 thread Consumer:
7     while (true) {
8         ...
9         consume(buffer[end]);
10        end = (end+1) % N;
11        ...
12    }
```

Unique Producer/Consumer

```
1 global Object[] buffer = new Object[N];
2 global Semaphore empty = new Semaphore(N);
3 global Semaphore full = new Semaphore(0);
4 global int start = 0;
5 global int end = 0;

6 thread Producer:
7     while (true) {
8         empty.acquire();
9         buffer[start] = produce();
10        start = (start+1) % N;
11        full.release();
12    }

6 thread Consumer:
7     while (true) {
8         full.acquire();
9         consume(buffer[end]);
10        end = (end+1) % N;
11        empty.release();
12    }
```

Multiple Producers

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

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

```
6 thread ProducerA:
7   while (true) {
8     empty.acquire();
9     buffer[start] = produce();
10    start = (start+1) % N;
11    full.release();
12  }

6 thread ProducerB:
7   while (true) {
8     empty.acquire();
9     buffer[start] = produce();
10    start = (start+1) % N;
11    full.release();
12  }
```


Multiple Producers

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

```
1 global Semaphore mutexP = new Semaphore(1);
2
3 thread Producer: {
4     while (true) {
5         empty.acquire();
6         mutexP.acquire();
7         buffer[start] = produce();
8         start = (start+1) % N;
9         mutexP.release();
10        full.release();
11    }
12 }
```

Multiple Consumers

- ▶ Must guarantee mutual exclusion between consumers

```
1 global Semaphore mutexC = new Semaphore(1);
2
3 thread Consumer: {
4     while (true) {
5         full.acquire();
6         mutexC.acquire();
7         consume(buffer[end]);
8         end = (end+1) % N;
9         mutexC.release();
10        empty.release();
11    }
12 }
```

Putting it all together

```
1 int N = 10;
2 global int[] buffer = new int[N];
3 global Semaphore empty = new Semaphore(N);
4 global Semaphore full = new Semaphore(0);
5 global int start = 0;
6 global int end = 0;
7 int counter = 0;
8
9 void consume(int i) { }
10
11 int produce () { return counter++; }
12
13 global Semaphore mutexP = new Semaphore(1);
14 global Semaphore mutexC = new Semaphore(1);
15
16 // continues in next slide
```

Putting it all together

```
1 Consumer(int id) {
2     while (true) {
3         full.acquire();
4         mutexC.acquire();
5         consume(buffer[end]);
6         print(id+" consumed product "+ buffer[end] + " at "+ end);
7         end = (end+1) % N;
8         mutexC.release();
9         empty.release();
10    }
11 }
12
13
14 Producer(int id) {
15     while (true) {
16         empty.acquire();
17         mutexP.acquire();
18         buffer[start] = produce();
19         print(id+" add product "+ buffer[start]+ " at "+ start);
20         start = (start+1) % N;
21         mutexP.release();
22         full.release();
23    }
24 }
```

Putting it all together

```
1
2 for (i=0; i<5; i++) {
3     int id = i;
4     thread Producer(id);
5     thread Consumer(id);
6 }
```

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 Writer() {  
2  
3     ...  
4     write();  
5     ...  
6  
7 }
```

```
1 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 global Semaphore resource = new Semaphore(1);
2 global Semaphore readersCountAccess
  = new Semaphore(1);
3 global int readersCount = 0;

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

1 Reader() {
2     readersCountAccess.acquire();
3     readersCount++;
4     if (readersCount == 1)
5         resource.acquire();
6     readersCountAccess.release();
7
8     read();
9
10    readersCountAccess.acquire();
11    readersCount--;
12    if (readersCount == 0)
13        resource.release();
14    readersCountAccess.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 Writer() {
2
3     writersCountAccess.acquire();
4     writersCount++;
5     if (writersCount == 1)
6         readTry.acquire();
7     writersCountAccess.release();
8
9
10    resource.acquire();
11    write();
12    resource.release();
13
14    writersCountAccess.acquire();
15    writersCount--;
16    if (writersCount == 0)
17        readTry.release();
18    writersCountAccess.release();
19 }

1 Reader() {
2     readTry.acquire();
3     readersCountAccess.acquire();
4     readersCount++;
5     if (readersCount == 1)
6         resource.acquire();
7     readersCountAccess.release();
8     readTry.release();
9
10
11    read();
12
13
14    readersCountAccess.acquire();
15    readersCount--;
16    if (readersCount == 0)
17        resource.release();
18    readersCountAccess.release();
19 }
```

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

Third Solution

```
1 global int readersCount;
2 global Semaphore resource = new Semaphore(1);
3 global Semaphore readCountAccess = new Semaphore(1);
4 global Semaphore serviceQueue = new Semaphore(1);
```

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

```
1 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