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

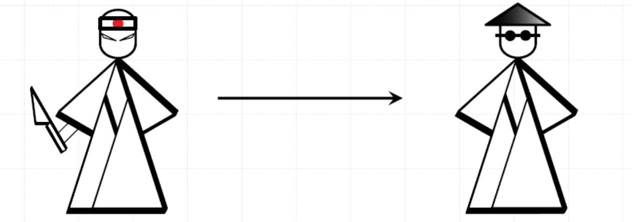
Today

- Recurring problems in the area
- Proven solution templates

Producers/Consumers

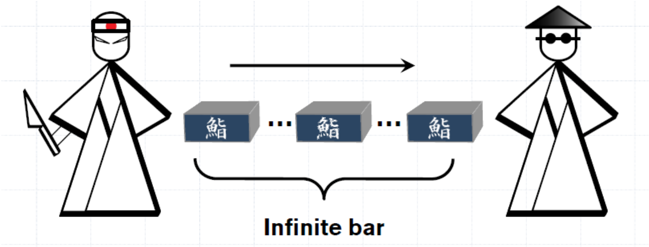
Readers/Writers

Producers/consumers



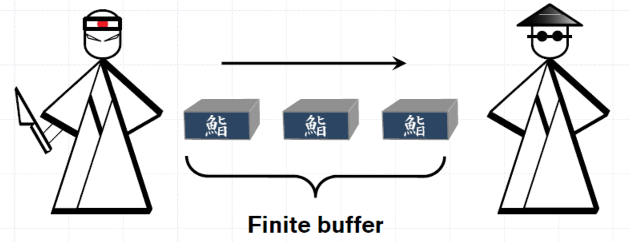
- A commmon pattern of interaction
- Must cater for difference in speed between each party

Unbounded Buffer



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

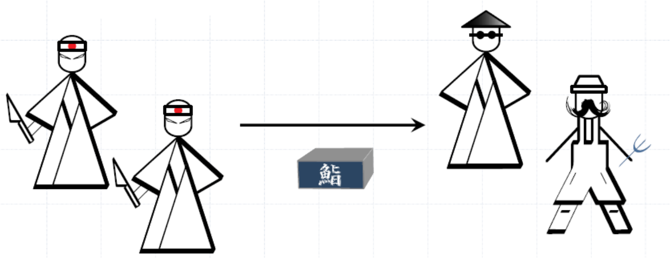
Bounded Buffer



- 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

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

```
thread Producer:
  while (true) {
    ...
    buffer = produce();
    ...
  }
```

```
thread Consumer:
  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`

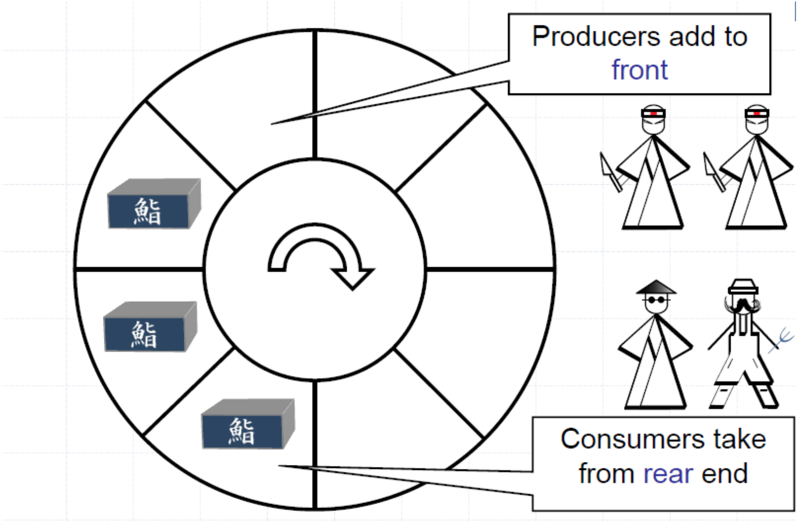
Split Binary Semaphores

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

```
thread Producer:
  while (true) {
    produce.acquire();
    buffer = produce();
    consume.release();
  }
```

```
thread Consumer:
  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

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

```
thread Producer:
  while (true) {
    ...
    buffer[start] = produce();
    start = (start+1) % N;
    ...
  }
```

```
thread Consumer:
  while (true) {
    ...
    consume(buffer[end]);
    end = (end+1) % N;
    ...
  }
```

Unique Producer/Consumer

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

```
thread Producer:
  while (true) {
    produce.acquire();
    buffer[start] = produce();
    start = (start+1) % N;
    consume.release();
  }
```

```
thread Consumer:
  while (true) {
    consume.acquire();
    consume(buffer[end]);
    end = (end+1) % N;
    produce.release();
  }
```

Multiple Producers

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

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

```
thread ProducerA:
  while (true) {
    produce.acquire();
    buffer[start] = produce();
    start = (start+1) % N;
    consume.release();
  }
```

```
thread ProducerB:
  while (true) {
    produce.acquire();
    buffer[start] = produce();
    start = (start+1) % N;
    consume.release();
  }
```

Multiple Producers

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

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

thread Producer: {
  while (true) {
    produce.acquire();
    mutexP.acquire();
    buffer[start] = produce();
    start = (start+1) % N;
    mutexP.release();
    consume.release();
  }
}
```

Multiple Consumers

- Must guarantee mutual exclusion between consumers

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

thread Consumer: {
  while (true) {
    consume.acquire();
    mutexC.acquire();
    consume(buffer[end]);
    end = (end+1) % N;
    mutexC.release();
    produce.release();
  }
}
```

Putting it all together

```
int N = 10;
global int[]  buffer = new int[N];
global Semaphore produce   = new Semaphore(N);
global Semaphore consume    = new Semaphore(0);
global int start = 0;
global int end   = 0;
int counter = 0;

void consume(int i) { }

int produce () { return counter++; }

global Semaphore mutexP = new Semaphore(1);
global Semaphore mutexC = new Semaphore(1);

// continues in next slide
```

Putting it all together

```
Consumer(int id) {
  while (true) {
    consume.acquire();
    mutexC.acquire();
    consume(buffer[end]);
    print(id+" consumed product "+ buffer[end] + " at "+ end);
    end = (end+1) % N;
    mutexC.release();
    produce.release();
  }
}


Producer(int id) {
  while (true) {
    produce.acquire();
    mutexP.acquire();
    buffer[start] = produce();
    print(id+" add product "+ buffer[start]+ " at "+ start);
    start = (start+1) % N;
    mutexP.release();
    consume.release();
  }
}
```

Putting it all together

```

for (i=0; i<5; i++) {
    int id = i;
    thread Producer(id);
    thread Consumer(id);
    }
```

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

```
global Semaphore resource = new Semaphore(1);
global Semaphore numReadersMutex  = new Semaphore(1);
global 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

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

```
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