Introduction to Synchronization

Ashvin Goel
ECE
University of Toronto
Overview

- Introduction to synchronization
- Producer-consumer problem
Problems with Concurrent Programming

- **Race Conditions**
  - Certain thread interleavings cause incorrect operation
  - Races can be avoided by using mutex locks

- **Synchronization**
  - Need to ensure operations are performed in particular order
  - E.g., T1 initializes a variable, T2 reads initialized variable

```c
T1() {
    int v1;
    A[0] = v1;
}
```

```c
T2() {
    int v1, v2;
    v2 = A[0];
}
```

initialize  Synchronize  read
Producer-Consumer Problem

- Threads communicate with each other using a shared buffer of a fixed size (i.e., bounded buffer)
  - One or more producers fill buffer
  - One or more consumers empty buffer

- Two synchronizations conditions
  - Producers wait if the buffer is full
  - Consumers wait if the buffer is empty

Applications: processes communicate using messages
Bounded Buffer Implementation

- Implementation uses a circular buffer

  - Number of elements in buffer: count = (in - out + n) % n
    - E.g., count = (7 - 4 + 8) % 8 = 3 // n is 8 since buffer has 8 slots
  - Buffer is empty when it has no elements, i.e., count = 0
  - Buffer is full when it has n-1 elements, i.e., count = 7
    - Why not n?

shared variables:
char buf[8]; // 7 usable slots
int in;     // place to write
int out;    // place to read

in = 7
out = 4
Try 1: Single Producer-Consumer

<table>
<thead>
<tr>
<th>void send(char elem) {</th>
</tr>
</thead>
<tbody>
<tr>
<td>int count = (in - out + n) % n;</td>
</tr>
<tr>
<td>while (count == n - 1) {</td>
</tr>
<tr>
<td>} // full</td>
</tr>
<tr>
<td>buf[in] = elem;</td>
</tr>
<tr>
<td>in = (in + 1) % n;</td>
</tr>
<tr>
<td>}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>char receive() {</th>
</tr>
</thead>
<tbody>
<tr>
<td>while (in == out) {</td>
</tr>
<tr>
<td>} // empty</td>
</tr>
<tr>
<td>elem = buf[out];</td>
</tr>
<tr>
<td>out = (out + 1) % n;</td>
</tr>
<tr>
<td>return elem;</td>
</tr>
<tr>
<td>}</td>
</tr>
</tbody>
</table>

- Is this code correct for a single producer, single consumer?
  - Code is incorrect, count is not recomputed in the while loop!
Try 2: Single Producer-Consumer

void send(char elem) {
    while ((in-out+n)%n == n - 1) {
    } // full
    buf[in] = elem;  // CS
    in = (in + 1) % n;
}

char receive() {
    while (in == out) {
    } // empty
    elem = buf[out];
    out = (out + 1) % n;
    return elem;
}

- Is this code correct for a single producer, single consumer?
  - Code is correct, even without mutex locks! But it is inefficient.

- Is this code correct with multiple producers, multiple consumers?
  - Code has a race
Try 3: Use Locking

<table>
<thead>
<tr>
<th>void send(char elem)</th>
</tr>
</thead>
<tbody>
<tr>
<td>while ((in-out+n)%n == n - 1) {</td>
</tr>
<tr>
<td>} // full</td>
</tr>
<tr>
<td>lock(l);</td>
</tr>
<tr>
<td>buf[in] = elem;</td>
</tr>
<tr>
<td>in = (in + 1) % n;</td>
</tr>
<tr>
<td>unlock(l);</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>char receive()</th>
</tr>
</thead>
<tbody>
<tr>
<td>while (in == out) {</td>
</tr>
<tr>
<td>} // empty</td>
</tr>
<tr>
<td>lock(l);</td>
</tr>
<tr>
<td>elem = buf[out];</td>
</tr>
<tr>
<td>out = (out + 1) % n;</td>
</tr>
<tr>
<td>unlock(l);</td>
</tr>
<tr>
<td>return elem;</td>
</tr>
</tbody>
</table>

- Is this code correct with multiple producers, multiple consumers?
  - Code has a race
Try 4: Use Locking for While Loop as well

- Is this code correct with multiple producers, multiple consumers?
  - Code has a deadlock!
Try 5: Release Locks Before Spinning

- Is this code correct?
  - Code is correct

- But code is inefficient
Try 6: Sleep After Unlocking

```c
// shared variable
wait_queue full, empty;

void send(char elem) {
    lock(l);
    while ((in-out+n)%n == n - 1) {
        unlock(l);
        thread_sleep(full);
        lock(l);
    } // full
    buf[in] = elem;
    in = (in + 1) % n;
    thread_wakeup(empty);
    unlock(l);
} // full

char receive() {
    lock(l);
    while (in == out) {
        unlock(l);
        thread_sleep(empty);
        lock(l);
    } // empty
    elem = buf[out];
    out = (out + 1) % n;
    thread_wakeup(full);
    unlock(l);
    return elem;
}
```

- Is this code correct?
  - Code has lost wakeup problem
Try 7: Sleep Before Unlocking

// shared variable
wait_queue full, empty;

void send(char elem) {
    lock(l);
    while ((in-out+n)%n == n - 1) {
        unlock(l);
        thread_sleep(full);
        lock(l);
    } // full
    buf[in] = elem;
    in = (in + 1) % n;
    thread_wakeup(empty);
    unlock(l);
}

char receive() {
    lock(l);
    while (in == out) {
        unlock(l);
        thread_sleep(empty);
        lock(l);
    } // empty
    elem = buf[out];
    out = (out + 1) % n;
    thread_wakeup(full);
    unlock(l);
    return elem;
}

- Is this code correct?
  - Code has a deadlock!
Synchronization Challenges

- Have shown 7 different producer-consumer implementations!
- Some work correctly, but spin
- The last two implement blocking synchronization
  - Can’t release lock and then sleep
    - Causes race because wakeup notification can be lost!
  - Can’t sleep while holding lock
    - Causes deadlock!
- Need to release the lock and sleep in a critical section!
  - Will avoid race and deadlock
  - Next lecture shows how …
Summary

- Threads use synchronization to order their operations
  - A thread waits on a condition to occur, e.g., data to arrive
  - When another thread makes the condition occur, e.g., delivers data, it wakes up waiting thread

- Used producer-consumer problem to motivate need for synchronization
  - Producers wait when buffer is full
  - Consumers wait when buffer is empty

- Showed that implementing synchronization using ad-hoc methods generally leads to incorrect results

- Next lecture describes primitives for solving synchronization problems
Think Time

- What is the difference between mutual exclusion and synchronization?

- In Try 4, suppose send and receive use different locks. Would that work?

- In Try 6, producer woke up consumer before releasing the lock. Wouldn’t that cause a problem?

- In Try 6, suppose the producer woke up consumer BEFORE incrementing the IN variable. Would that cause a problem?

- In Try 6, suppose the producer woke up consumer AFTER releasing the lock. Would that cause a problem?
Think Time Answers

- What is the difference between mutual exclusion and synchronization?
  - Mutex is used to ensure that only one thread accesses a critical section at a time, helping avoid races. Synchronization is used to ensure that threads wait on some condition, which helps with ordering operations.

- In Try 4, suppose send and receive use different locks. Would that work?
  - In this case, the producers are running in a critical section and the consumers are running in a critical section, so the problem becomes equivalent to a single producer-consumer problem, and so this solution will work correctly, similar to Try 2. However, this solution will also cause spinning when the buffer is full/empty.
In Try 6, producer woke up consumer before releasing the lock. Wouldn’t that cause a problem?

- No, there is no problem because consumers will try to acquire a lock after they wakeup, which will only succeed after the producer releases the lock.

In Try 6, suppose the producer woke up consumer BEFORE incrementing the IN variable. Would that cause a problem?

- That will not cause a problem for the same reason as above. Consumers will need to acquire a lock before entering their critical section, and the producer holds the lock.
In Try 6, suppose the producer woke up consumer AFTER releasing the lock. Would that cause a problem?

- That would cause a problem similar to the lost wakeup problem because the wakeup is not being performed in a critical section.