CSCI 4730/6730 OS
(Chap #7 Sync. Examples)

In Kee Kim

Department of Computer Science

University of Georgia

Announcement

- Next week TA’s office hour
  - No office hour on Monday
  - Instead, he will hold two hours on Friday (10 to noon)

Chapter 7: Synchronization Examples

- Classical Problems in Synchronization
  - Bounded-Buffer Problem
  - Readers-Writers Problem
  - Dining-Philosophers Problem

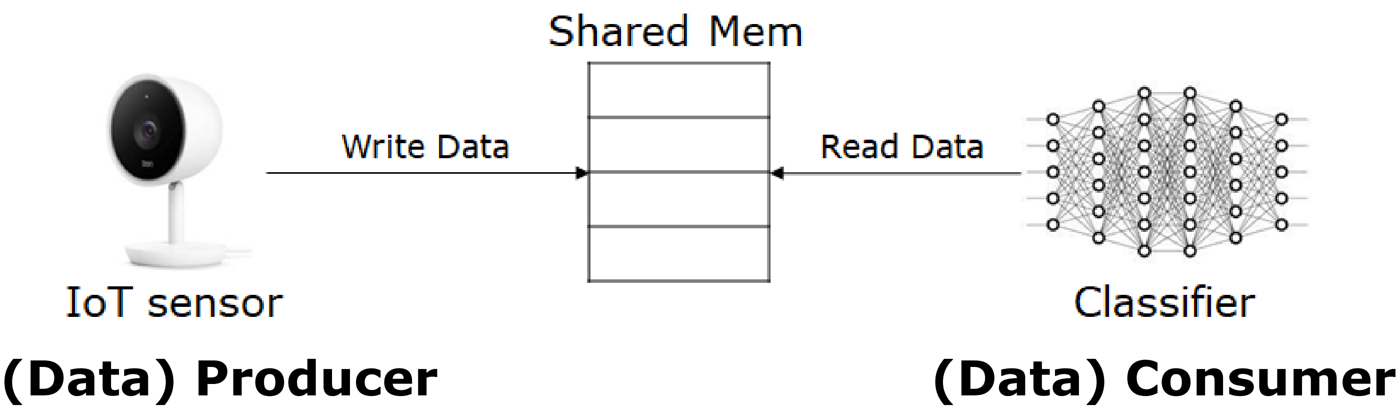
- Classical problems used to test newly-proposed synchronization schemes

Classical Problems of Synchronization

- Classical Problems in Synchronization
  - **Bounded-Buffer Problem**
  - Readers and Writers Problem
  - Dining-Philosophers Problem

Bounded-Buffer Problem

- It is also called “Producer-Consumer” problem

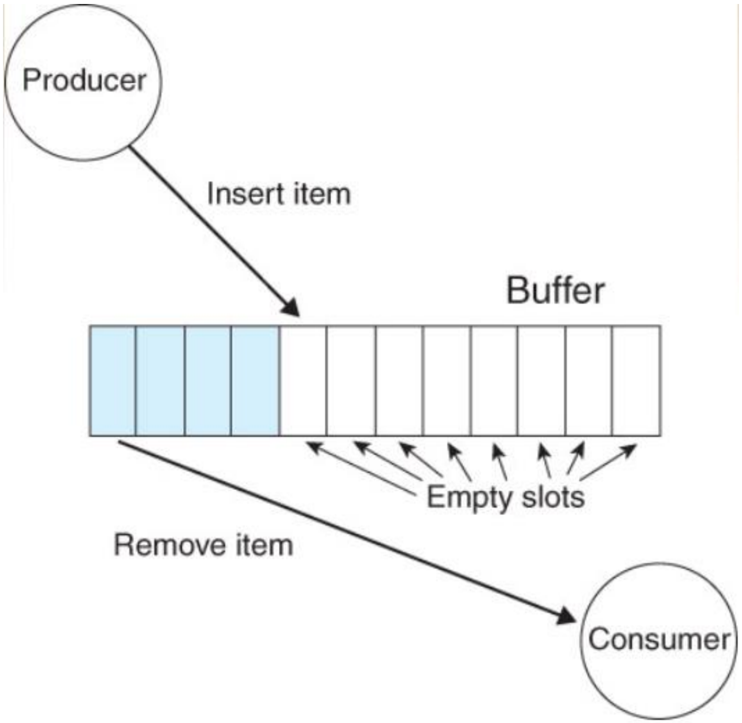


Bounded-Buffer Problem

- ***n*** buffers, each can hold one item
- There are two processes – Producer and Consumer
  - **Producer** tries to insert an item into an empty slot in the buffer
  - **Consumer** tries to remove the item from a filled slot in the buffer

Bounded-Buffer Problem

- Three requirements
  - The Producer (**P**) must not insert item when buffer is full
  - The Consumer (**C**) must not remove item when buffer is empty
  - **P** and **C** should not insert and remove at the same time



Bounded-Buffer Problem

Producer:

```
while (true) {
    /* produce an item in
    next produced */

    while (counter == BUFFER_SIZE);
    /* do nothing */

    buffer[in] = next_produced;
    in = (in + 1) % BUFFER_SIZE;
    counter++;
}
```

Consumer:

```
while (true) {
    while (counter == 0);
    /* do nothing */

    next_consumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;

    counter--;
    /* consume the item in next
    consumed */
}
```

Potential Issue?

- **Race Condition!**
- What is Race Condition?
  - Output of a concurrent program depends on the order of operations between threads

Producer:

```
while (true) {
    /* produce an item in
    next produced */

    while (counter == BUFFER_SIZE);
    /* do nothing */

    buffer[in] = next_produced;
    in = (in + 1) % BUFFER_SIZE;
    counter++;
}
```

Consumer:

```
while (true) {
    while (counter == 0);
    /* do nothing */

    next_consumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;

    counter--;
    /* consume the item in next
    consumed */
}
```

Race Condition

- **`counter++`** can be implemented as

```
register1 = counter
register1 = register1 + 1
counter = register1
```

- **`counter--`** can be implemented as

```
register2 = counter
register2 = register2 - 1
counter = register2
```

```
Producer:
while (true) {
    /* produce an item in
    next produced */

    while (counter == BUFFER_SIZE);
    /* do nothing */

    buffer[in] = next_produced;
    in = (in + 1) % BUFFER_SIZE;
    counter++;
}
```

```
Consumer:
while (true) {
    while (counter == 0);
    /* do nothing */

    next_consumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;

    counter--;
    /* consume the item in next
    consumed */
}
```

Race Condition

- Consider this execution interleaving with "count = 5" initially:

```
S0: producer executes register1 = counter         {register1 = 5}
S1: producer executes register1 = register1 + 1 {register1 = 6}
S2: consumer executes register2 = counter         {register2 = 5}
S3: consumer executes register2 = register2 – 1 {register2 = 4}
S4: producer executes counter = register1         {counter = 6 }
S5: consumer executes counter = register2         {counter = 4}
```

How to address this problem?

- Yes, Semaphore!

Recap: Semaphore

- Definition: a Semaphore has an integer value and supports the following two operations:
  - **`Wait()`**
  - **`Signal()`**
  - Only time you can set integer directly is at initialization time

Recap: Semaphore

- Semaphore from railway analogy
  - Here is a (counting) semaphore initialized to 2 for resource control: