Bounded-Buffer Problem

- How many semaphores do we need?
- Again -- 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 must at the same time

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 */
}
```

Bounded-Buffer Problem

- Semaphore **`mutex`**
- Semaphore **`full`**
- Semaphore **`empty`**

Initial values for three semaphores?

- Again -- 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 must at the same time

Bounded-Buffer Problem

- Semaphore **`mutex`** initialized to the value ***1***
  - To protect **`count`** operation
- Semaphore **`full`** initialized to the value ***0***
  - Number of full buffers
- Semaphore **`empty`** initialized to the value ***n***
  - Number of empty buffers

```
int n;
semaphore mutex = 1;
semaphore empty = n;
semaphore full = 0
```

Let’s Make Pseudocode First

Producer with Semaphore

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++;
}
```

Producer (w/ Semaphore)

```
while (true) {
      . . .
   /* produce an item in next_produced */
      . . .
   wait(empty);
   wait(mutex);
      . . .
   /* add next_produced to the buffer */
      . . .
   signal(mutex);
   signal(full);
}
```

Consumer with Semaphore

Consumer

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

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

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

Consumer (w/ Semaphore)

```
while (true) {
   wait(full);
   wait(mutex);
      . . .
   /* remove an item from buffer to next_consumed */
      . . .
   signal(mutex);
   signal(empty);
      . . .
   /* consume the item in next_consumed */
      . . .
}
```

Producer and Consumer

Producer (w/ Semaphore)

```
while (true) {
    . . .
  /* produce an item in next_produced */
    . . .
  wait(empty);
  wait(mutex);
    . . .
  /* add next_produced to the buffer */
    . . .
  signal(mutex);
  signal(full);
}
```

Consumer (w/ Semaphore)

```
while (true) {
  wait(full);
  wait(mutex);
    . . .
  /* remove an item from buffer to next_consumed */
    . . .
  signal(mutex);
  signal(empty);
    . . .
  /* consume the item in next_consumed */
    . . .
}
```

Let’s Implement “Bounded-Buffer Problem”

- Let’s use two semaphores and one mutex lock

```
int n;
semaphore mutex = 1;
semaphore empty = n;
semaphore full = 0
```

```
int n_items_in_buffer = 0;
pthread_mutex_t lock;
sem_t sem_empty;
sem_t sem_full;
```

- APIs
  - **pthread_mutex_init**
  - **pthread_mutex_lock & pthread_mutex_unlock**
  - **sem_init & sem_destroy**
  - **sem_wait & sem_post**

Base Code

nike.cs.uga.edu - PuTTY

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>
#include <semaphore.h>

int n_items_in_buffer = 0;
pthread_mutex_t lock;
sem_t sem_empty;
sem_t sem_full;

int MAX_SLOT = 5;
int NUM_CONSUMER = 3;

void *producer() {
  while(1) {
    printf("[Producer] produce an item\n");

    n_items_in_buffer++;
    printf("[Producer] added the item to the buffer. # items in buffer = %d\n"
          , n_items_in_buffer);
    usleep(rand()%500000); // processing overhead

  }
}

void *consumer(void *arg) {
  printf("[Consumer %d] joined.\n", syscall(186)-getpid());
  while(1) {

    n_items_in_buffer--;
    printf("[Consumer %d] removed an item from the buffer. # items in buffer = %d\n"
          , syscall(186)-getpid(), n_items_in_buffer);
    usleep(rand()%800000); // processing overhead
  }
}

```

14,1 Top

nike.cs.uga.edu - PuTTY

```c
int main(int argc, char *argv[])
{
  int i;

  pthread_t p;
  pthread_t c[10];

  for(i = 0; i < NUM_CONSUMER; i++)
  {
    pthread_create(&(c[i]), NULL, consumer, NULL);
  }
  pthread_create(&p, NULL, producer, NULL);

  for(i = 0; i < NUM_CONSUMER; i++)
    pthread_join(c[i], NULL);
  pthread_join(p, NULL);

  return 0;
}
```

57,1 Bot

Real-World Example of Bounded Buffer Problem

- IoT Data Processing
- Audio/Video player:
  - Thread #1: Network/data transmission
  - Thread #2: Decode packets and display data from shared buffer
- Web Server: Master threads and slave threads
  - Listener Threads: Producers
  - Worker Threads: Consumer

Classical Problems of Synchronization

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

Readers-Writers Problem

- Classic synchronization problem in data sharing/management system. e.g., DBMS
- A data set is shared among a number of concurrent processes
  - **Readers** – only read the data set; they do ***not*** perform any updates
  - **Writers** – can both read and write

Readers-Writers Problem - Requirements

1. Two or more readers access the shared data at the same time, no adverse effects will result
2. No simultaneous access from a writer and some other readers/writers -- no reading while data is being updated
3. Writers should have exclusive access to the shared data.
   - In other words, if a writer is updating (writing) data, no other readers or writers are allowed to access the data.

Readers-Writers Problem

- You have a global variable **`count`**
- Reader
  - Do "**`printf count`**"
- Writer
  - Do "**`count = count*10;`**"
- Let's use mutex or binary semaphore