Process Synchronization

COMP 3361: Operating Systems I

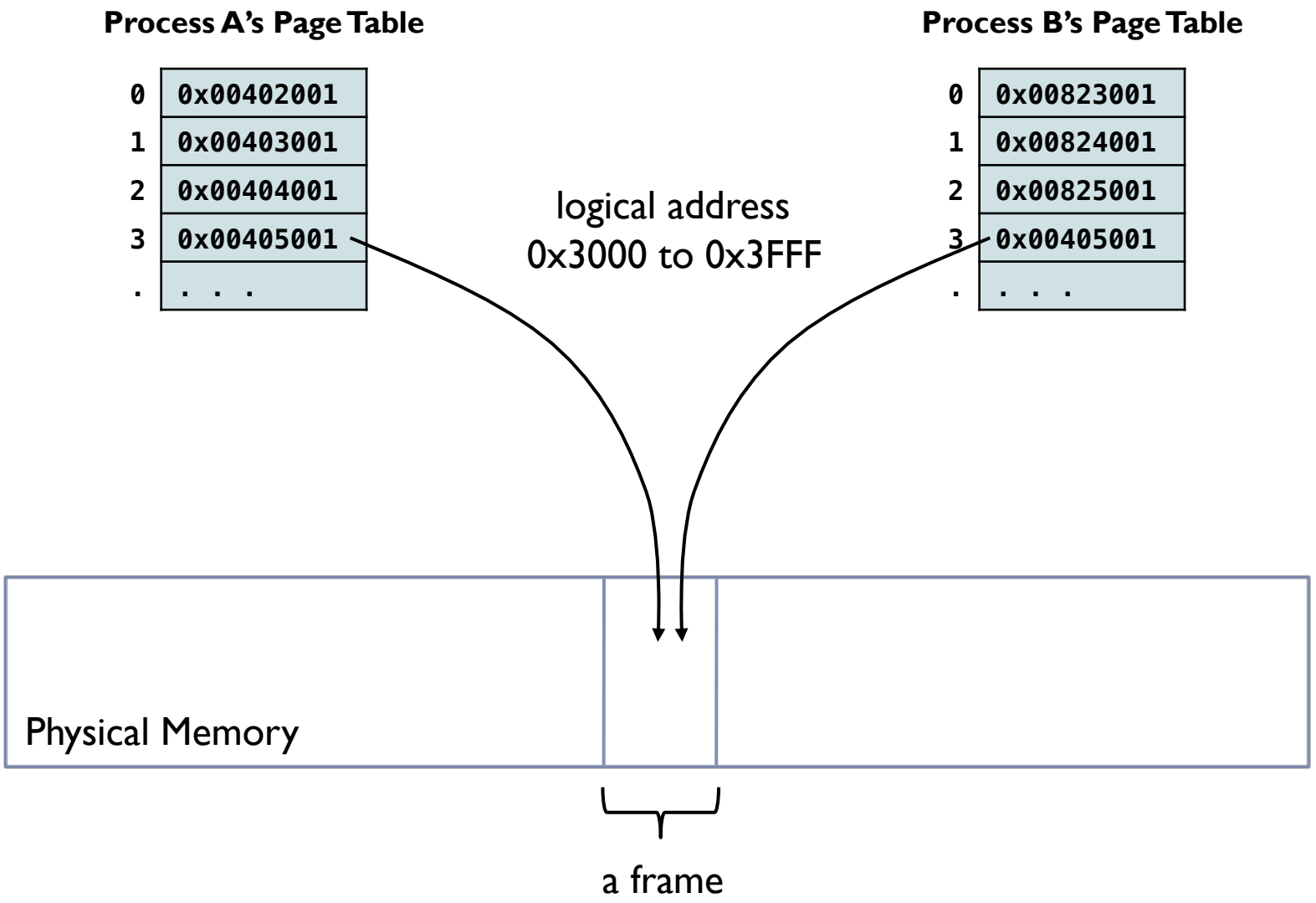
Winter 2015

http://www.cs.du.edu/3361

1 Shared Memory

- Two or more processes (or threads) need access to the same data
- Threads
  - by design, they share data
- Processes
  - by design, each process has its own address space (therefore separate data section)
  - how can they share data?

2 Shared Memory in Processes



Why Synchronization?

- Concurrent access to shared data may result in data inconsistency
  - imagine two processes writing to the same array at the same time
- Maintaining **data consistency** requires mechanisms to ensure the orderly execution of cooperating processes

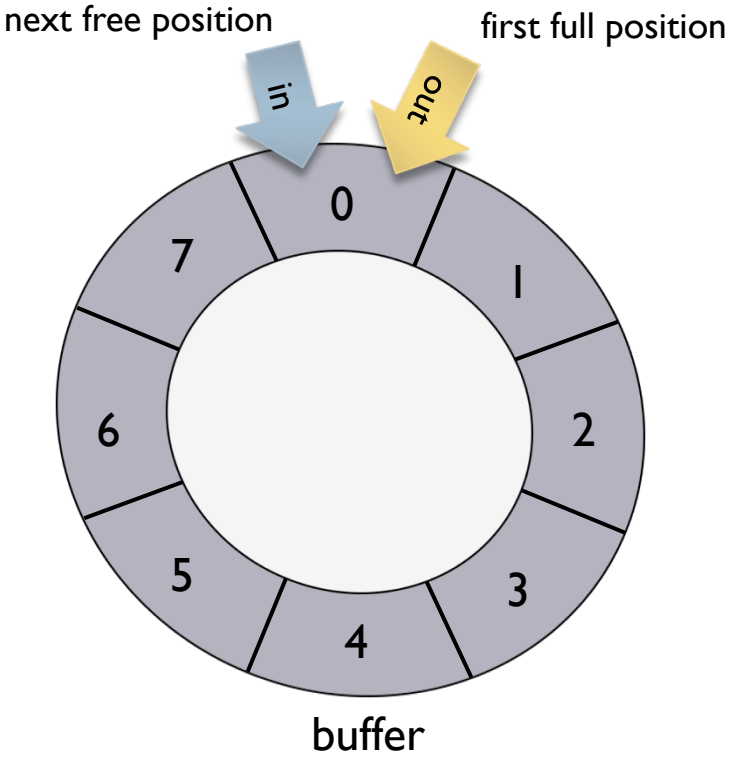
4 Producer-Consumer Problem

- A **producer** process produces information that is consumed by a **consumer** process
- If producer has access to an unlimited amount of storage (*unbounded buffer*), it can keep producing
  - do not have to worry if consumer is consuming the information or not
- The consumer may have to wait for new items to be produced
- What happens when the storage is limited (*bounded buffer*)?
  - producer also may have to wait until some items are consumed

A Shared Memory Solution

- Shared data

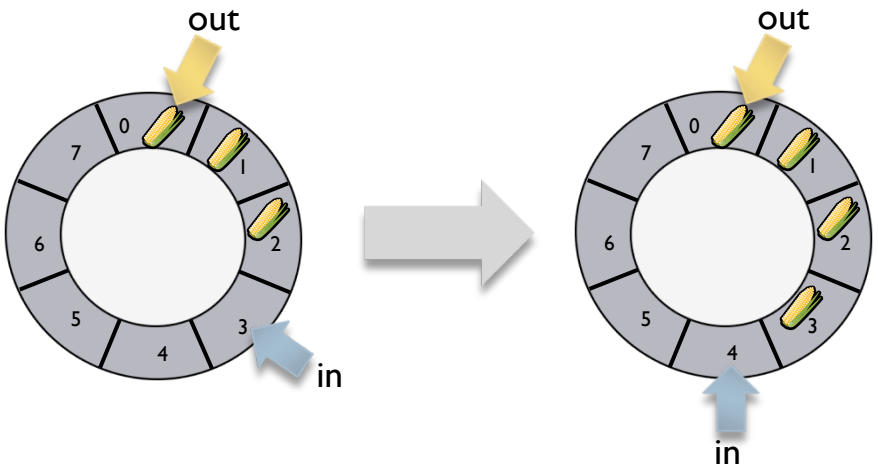
```
#define BUFFER_SIZE 8
typedef struct {
   . . .
} item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

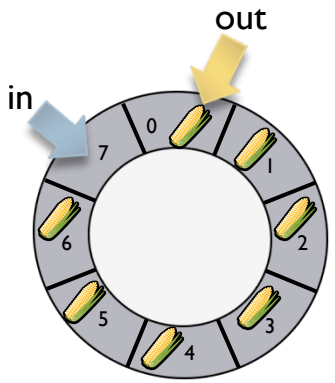


buffer is empty when **`in == out`**
buffer is full when **`((in+1)%BUFFER_SIZE) == out`**

Producer

```
while (true) {
   /* Produce an item */
    while (((in + 1) % BUFFER_SIZE) == out)
      ;   /* do nothing -- no free buffers */
     buffer[in] = item;
     in = (in + 1) % BUFFER_SIZE;
 }
```

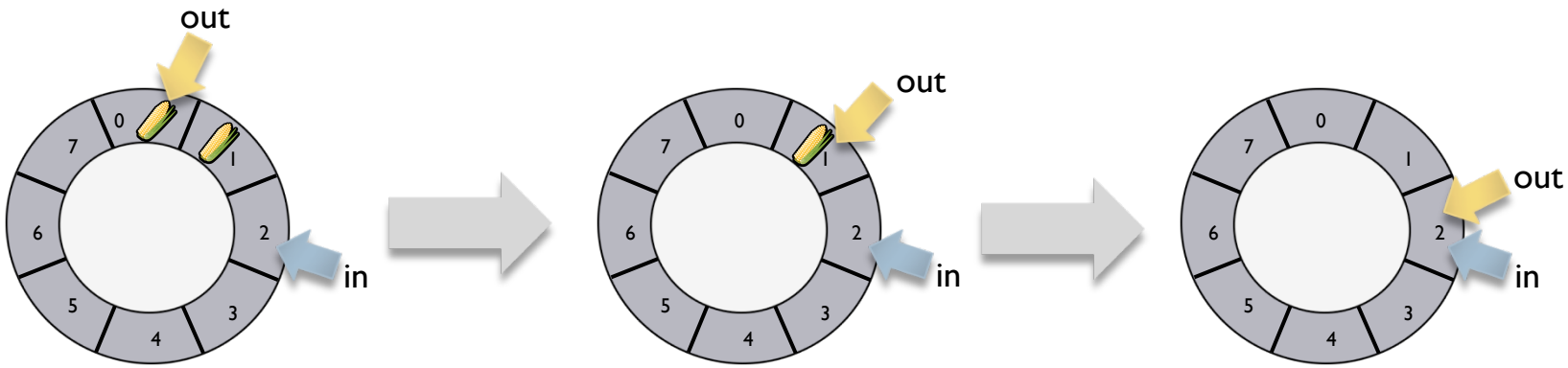




producer waits in this case

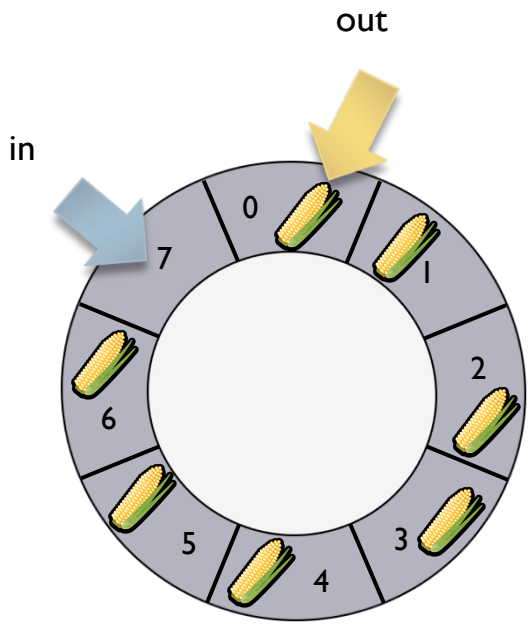
Consumer

```
while (true) {
        while (in == out)
                ; // do nothing -- nothing to consume

        // remove an item from the buffer
        item = buffer[out];
        out = (out + 1) % BUFFER SIZE;
 }
```



empty buffer
consumer must wait

8 Producer-Consumer Revisited



buffer is full since

```
(in+1) % BUFFER_SIZE == out
```

To use all available space in the buffer:

- use a **count** variable to track the number of occupied slots
  - initialize **count = 0**
- producer increments **count**
- consumer decrements **count**

Producer

Before

```
while (true) {
   /* Produce an item */
     while (((in + 1) % BUFFER_SIZE) == out)
     ;   /* do nothing -- no free buffers */
    buffer[in] = item;
    in = (in + 1) % BUFFER_SIZE;
  }
```

After

```
while (true) {
   /* Produce an item */
     while (count == BUFFER_SIZE)
     ; /* do nothing -- no free buffers */
     buffer [in] = item;
     in = (in + 1) % BUFFER_SIZE;
     count++;
  }
```

Consumer

Before

```
while (true) {
     while (in == out)
       ; // do nothing -- nothing to consume
     // remove an item from the buffer
     item = buffer[out];
     out = (out + 1) % BUFFER SIZE;
 }
```

After

```
while (true) {
     while (count == 0)
     ; /* do nothing -- nothing to consume */
     item = buffer [out];
     out = (out + 1) % BUFFER_SIZE;
     count--;
  }
```

Data Inconsistency Example

- **count++** could be implemented as

```
register1 = count
register1 = register1 + 1
count = register1
```

- **count--** could be implemented as

```
register2 = count
register2 = register2 - 1
count = register2
```

Data Inconsistency Example

- Say **count = 5**
- Say producer produces an item and consumer consumes one
  - **count** should still be 5 at the end of it
- Assume the following interleaving of instructions:

T0: producer executes register1 = count {register1 = 5}
T1: producer executes register1 = register1 + 1 {register1 = 6}
T2: consumer executes register2 = count {register2 = 5}
T3: consumer executes register2 = register2 - 1 {register2 = 4}
T4: producer executes count = register1 {count = 6 }
T5: consumer executes count = register2 {count = 4}

- We arrive at an incorrect value for **count**
  - the value may be different for a different execution order

Race Condition

- An incorrect state for **count** was achieved because both processes were allowed to manipulate it concurrently

- **Race condition**
  - several processes access and manipulate the same data concurrently
  - outcome of the execution depends on the particular order of access

- Process synchronization is all about the prevention of race conditions

Critical Section

- Segment of code in which a process may be changing shared data
  - common variables, tables, files, etc.
- Two processes **should not** be executing in their critical sections at the same time
  - one (or more) of the processes must be made to wait
  - also called having **mutual exclusion**

A Good Solution

- No two processes may be simultaneously inside their critical regions
- No assumptions may be made about speeds or the number of CPUs
- No process running outside its critical region may block other processes
- No process should have to wait forever to enter its critical region

References

- Chapter 2.3 and 2.5, Modern Operating Systems, A. Tanenbaum and H. Bos, 4th Edition.