Interacting Processes and Concurrency
CS303 Operating Systems

Recap

- Concurrency: Multitasking Vs. Parallel
- Precedence graph construction lets to schedule modules/statements to be run in parallel
- Interacting processes continue execution with appropriate synchronisation in place
  - Ex: Producer – Consumer problem
- Synchronisation is important between interacting processes
  - Ex: In producer-consumer case:
    - If buffer is empty, consumer shall wait
    - If buffer is full, producer shall wait
- Waiting could be:
  - Busy waiting: Busy-wait Vs Blocking

Producer in C

Producer:

```
while(true)
   {
   _produce_pItem; // inits pItem
   while ((in+1) % n == out) { } //IF FULL
   buffer[in] = pItem;
   in = (in+1) % n;
   }
```

Producer is “BUSY WAITING”

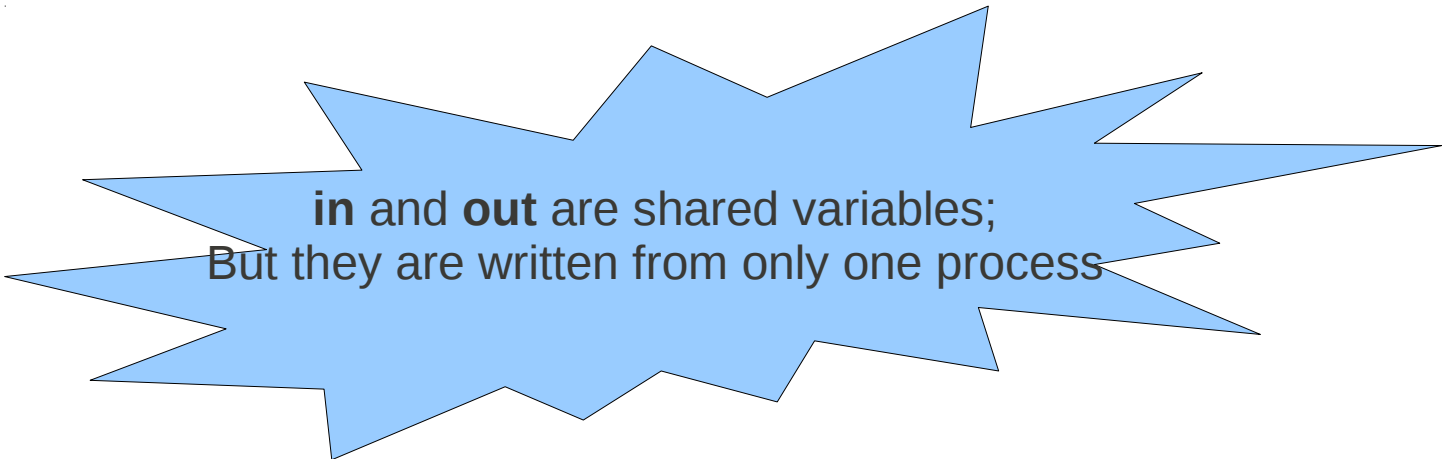
Producer:

```
while(true)
   {
   _produce_pItem; // inits pItem
   while ((in+1) % n == out) { } //IF FULL
   buffer[in] = pItem;
   in = (in+1) % n;
   }
```

Consumer

Consumer:

```
while(true)
   {
   while (in == out) { } //WAIT IF EMPTY
   cItem = buffer[out];
   out = (out+1) % n;
   _consume_cItem; // takes cItem
   }
```



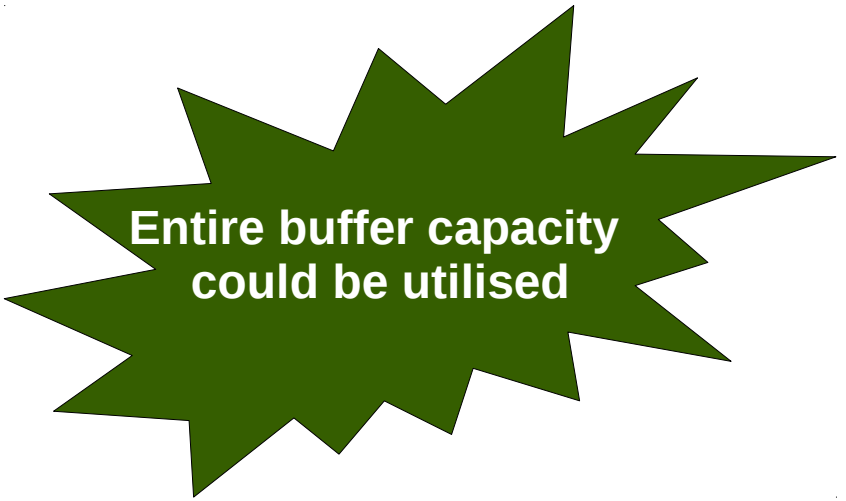
Prod-Cons ver.2 with counter

Producer:

```
int itemCount = 0;
     /*shared variable for no.
     of items in Buffer*/
while(true)
   {
   _produce_pItem;
        // creating pItem
   while (itemCount==n) { }
   buffer[in] = pItem;
   in = (in+1) % n;
   itemCount++;
   }
```

Consumer:

```
while(true)
   {
   while(itemCount==0) { }
   cItem = buffer[out];
   out = (out+1) % n;
   itemCount--;
   _consume_cItem; // takes
   cItem
   }
```



Prod-Cons ver.2 with counter

Producer:

```
int itemCount = 0;
      /*shared variable for no. of
      items in Buffer*/
while(true)
   {
   _produce_pItem;
         // creating pItem
   while (itemCount == n) { }
   //IF FULL
   buffer[in] = pItem;
   in = (in+1) % n;
   itemCount++;
   }
```

Consumer:

```
while(true)
   {
   while(itemCount==0) { }
   cItem = buffer[out];
   out = (out+1) % n;
   itemCount--;
   _consume_cItem; // takes
   cItem
   }
```

PC: sharedVariable updated in two different processes

Producer:

```
int itemCount = 0;
     /*shared variable for no. of
     items in Buffer*/
while(true)
   {
   _produce_pItem;
        // creating pItem
   while (itemCount == n) { }
   //IF FULL
   buffer[in] = pItem;
   in = (in+1) % n;
   itemCount++;
   }
```

Consumer:

```
while(true)
   {
   while(itemCount==0) { }
   cItem = buffer[out];
   out = (out+1) % n;
   itemCount--;
   _consume_cItem; // takes
   cItem
   }
```

High-level statement translates to several Asm-instructions

Producer Assembly-prog

```
itemCount = itemCount + 1;
```

- Translates to:

```
R1 = itemCount;
R1 = R1 + 1;
itemCount = R1;
```

Consumer Assembly-prog

```
itemCount = itemCount - 1;
```

- Translates to:

```
R2 = itemCount;
R2 = R2 + 1;
itemCount = R2;
```

Atomicity: single asm instruction

Producer Assembly-prog

```
itemCount = itemCount + 1;
```

- Translates to:

```
R1 = itemCount;
R1 = R1 + 1;
itemCount = R1;
```

Consumer Assembly-prog

```
itemCount = itemCount - 1;
```

- Translates to:

```
R2 = itemCount;
R2 = R2 + 1;
itemCount = R2;
```

Atomicity: single asm instruction

Producer Assembly-prog

```
itemCount = itemCount + 1;
```

- Translates to:

```
R1 = itemCount;
R1 = R1 + 1;
itemCount = R1;
```

Consumer Assembly-prog

```
itemCount = itemCount - 1;
```

- Translates to:

```
R2 = itemCount;
R2 = R2 + 1;
itemCount = R2;
```

High-level statement translates to several Asm-instructions

Producer Assembly-prog

```
itemCount = itemCount + 1;
```

- Translates to:

```
R1 = itemCount;
R1 = R1 + 1;
itemCount = R1;
```

Consumer Assembly-prog

```
itemCount = itemCount - 1;
```

- Translates to:

```
R2 = itemCount;
R2 = R2 + 1;
itemCount = R2;
```

Producer Assembly-prog

```
itemCount = itemCount + 1;
```

- Translates to:

```
R1 = itemCount;
R1 = R1 + 1;
itemCount = R1;
```

Consumer Assembly-prog

```
itemCount = itemCount - 1;
```

- Translates to:

```
R2 = itemCount;
R2 = R2 + 1;
itemCount = R2;
```

Producer Assembly-prog

```
itemCount = itemCount + 1;
```

- Translates to:

```
R1 = itemCount;
R1 = R1 + 1;
itemCount = R1;
```

Consumer Assembly-prog

```
itemCount = itemCount - 1;
```

- Translates to:

```
R2 = itemCount;
R2 = R2 + 1;
itemCount = R2;
```

Producer Assembly-prog

```
itemCount = itemCount + 1;
```

- Translates to:

```
R1 = itemCount;
R1 = R1 + 1;
itemCount = R1;
```

Consumer Assembly-prog

```
itemCount = itemCount - 1;
```

- Translates to:

```
R2 = itemCount;
R2 = R2 + 1;
itemCount = R2;
```

Shared Variables' Operations: Problems

Producer Assembly-prog

```
itemCount = itemCount + 1;
```

- Executes as:

```
P1: R1 = itemCount;
P2: R1 = R1 + 1;
P3: itemCount = R1;
```

Consumer Assembly-prog

```
itemCount = itemCount - 1;
```

- Executes as:

```
C1: R2 = itemCount;
C2: R2 = R2 + 1;
C3: itemCount = R2;
```

1. Consider itemCount = 12 at some instance

2. If executions continues:
   P1, P2, P3 then itemCount = 13

3. then If:
   C1, C2, C3 then itemCount = 12

No problem!

Shared Variables' Operations: Problems

Producer Assembly-prog

```
itemCount = itemCount + 1;
```

- Executes as:

```
P1: R1 = itemCount;
P2: R1 = R1 + 1;
P3: itemCount = R1;
```

Consumer Assembly-prog

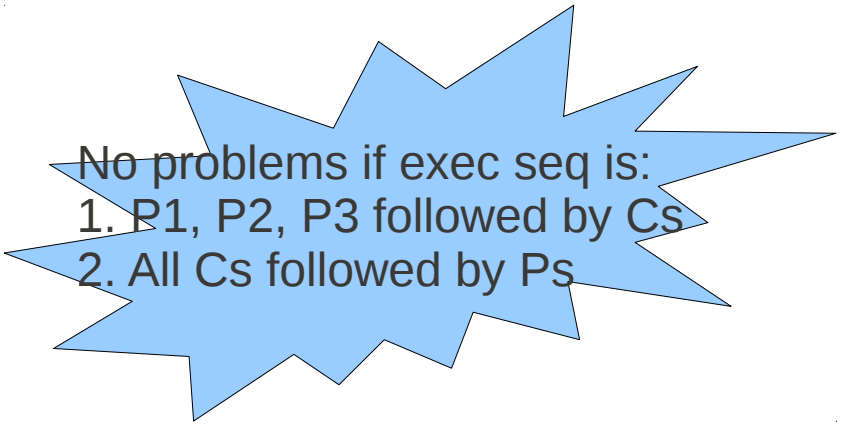
```
itemCount = itemCount - 1;
```

- Executes as:

```
C1: R2 = itemCount;
C2: R2 = R2 + 1;
C3: itemCount = R2;
```

But problem arises if there is interleaving between P-steps and C-steps!!!
Consider the exec sequence:

P1 R1 = 12
P2 R1 = 13
C1 R2 = 12
C2 R2 = 11
P3 itemCount = 13
C3 itemCount = 11



Solutions to Critical Section Problem

Producer Assembly-prog

```
itemCount = itemCount + 1;
```

- Executes as:

```
P1: R1 = itemCount;
P2: R1 = R1 + 1;
P3: itemCount = R1;
```

Consumer Assembly-prog

```
itemCount = itemCount - 1;
```

- Executes as:

```
C1: R2 = itemCount;
C2: R2 = R2 + 1;
C3: itemCount = R2;
```

CRITICAL SECTION: Those sections of two or more programs where shared variables are modified.

Critical Section Problem

Producer Assembly-prog

```
itemCount = itemCount + 1;
```

- Executes as:

```
P1: R1 = itemCount;
P2: R1 = R1 + 1;
P3: itemCount = R1;
```

Consumer Assembly-prog

```
itemCount = itemCount - 1;
```

- Executes as:

```
C1: R2 = itemCount;
C2: R2 = R2 + 1;
C3: itemCount = R2;
```

CRITICAL SECTION: Those sections of two or more programs where shared variables are modified.

Shared Variables' Operations: Problems

Producer Assembly-prog

```
itemCount = itemCount + 1;
```

- Executes as:

```
P1: R1 = itemCount;
P2: R1 = R1 + 1;
P3: itemCount = R1;
```

Consumer Assembly-prog

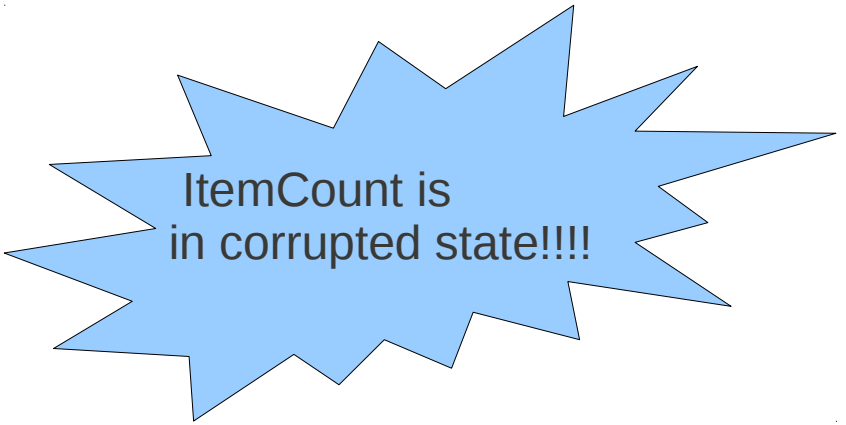
```
itemCount = itemCount - 1;
```

- Executes as:

```
C1: R2 = itemCount;
C2: R2 = R2 + 1;
C3: itemCount = R2;
```

But problem arises if there is interleaving between P-steps and C-steps!!!
Consider the exec sequence:

P1 R1 = 12
P2 R1 = 13
C1 R2 = 12
C2 R2 = 11
P3 itemCount = 13
C3 itemCount = 11



Solution to Critical Section Problem

- Many solutions could be there, but there are three important requirements on such solutions:
  - Mutual exclusion: atmost only one process could be in the critical section (CS);
  - Progress:
    - Who enters the critical section is decided by the processes that want to enter the critical section
    - Decision needs to be taken within finite time
  - Bounded waiting:
    - From the instance of expressing interest to enter CS
    - It shall be given access within no more than certain number of times other processes access the CS
      - Remember there could be several processes interested into CS
    - Meaning there shall be bounded time duration the process be given access into

Critical Section Problem

- CS is characterised by:
  - An Entry Section
  - Critical Section
  - Exit Section
  - Remainder Section

  Solution

  P1 and P2 two processes

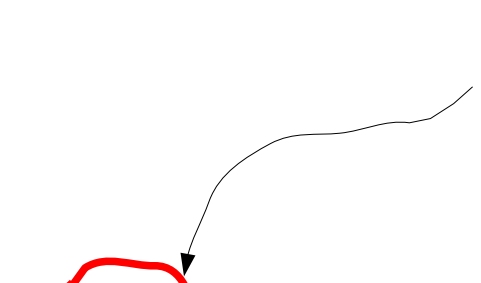
  turn = {1, 2} //enumerated type

  P_i:

```
while turn <> I do { }
   CS
turn = j
```

Consumer

Consumer:




```
while(true)
   {
   while (in == out) { } //WAIT IF EMPTY
   cItem = buffer[out];
   out = (out+1) % n;
   _consume_cItem; // takes cItem
   }
```

In busy-waiting CPU time is completely wasted!!!!!