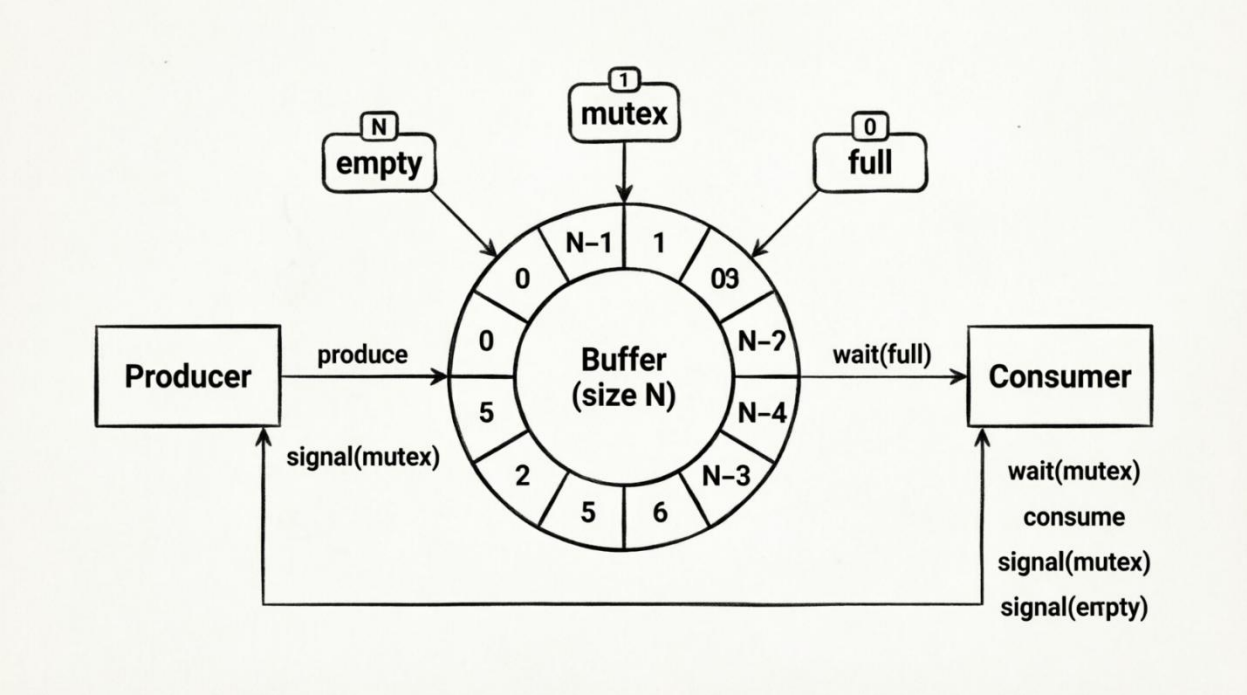
Producer-Consumer Problem

Processes share a bounded buffer. Synchronization ensures producers don't overwrite full slots and consumers don't read empty ones. Priority variants control which side wins contention when the buffer is neither full nor empty.

1 Standard Bounded Buffer (FIFO/Fair)

Behavior: Access follows resource availability. No artificial bias; OS scheduler typically handles tie-breaking.



1.1 Semaphores

```
semaphore empty = N;  // counts empty slots
semaphore full  = 0;  // counts filled slots
semaphore mutex = 1;  // protects buffer index manipulation

Producer() {
   wait(empty);
   wait(mutex);
   // --- INSERT ITEM INTO BUFFER ---
   signal(mutex);
   signal(full);
}
```

```
Consumer() {
   wait(full);
   wait(mutex);
   // --- REMOVE ITEM FROM BUFFER ---
   signal(mutex);
   signal(empty);
}
```

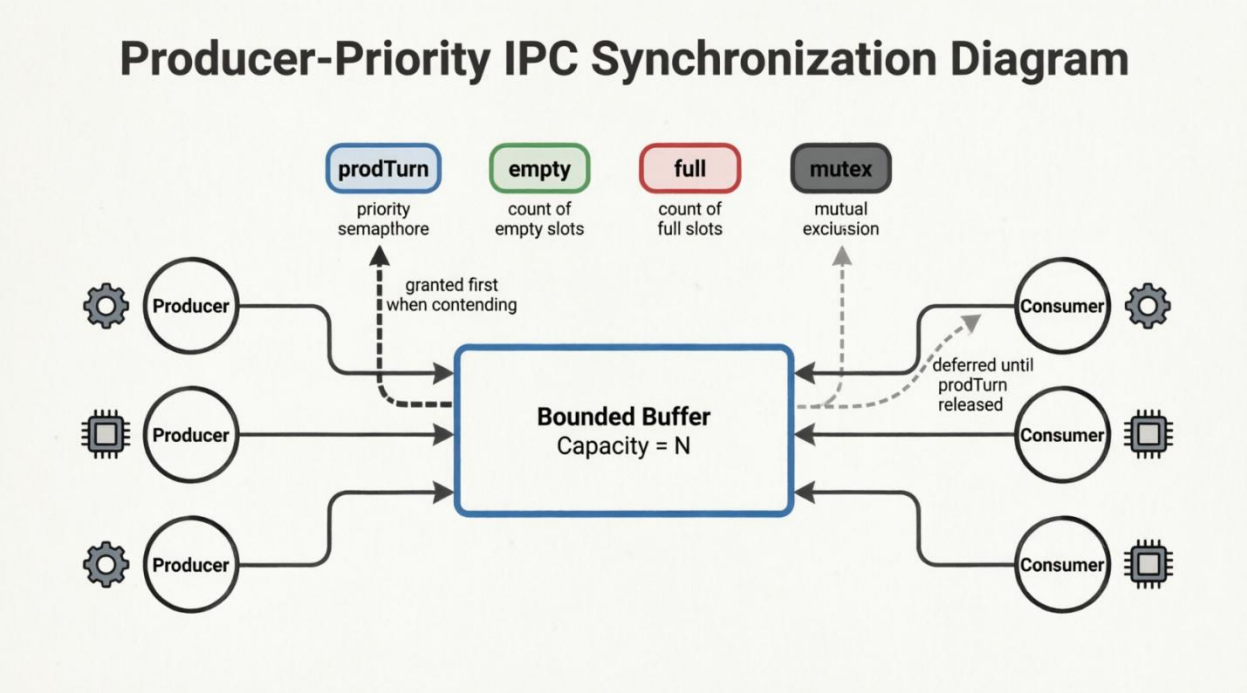
1.2 Monitors

```
monitor PC_Standard {
   int count = 0;
   int buffer[N];
   condition notFull, notEmpty;

   void insert(item) {
      while (count == N) wait(notFull);
      // add item to buffer
      count++;
      signal(notEmpty);
   }
   item remove() {
      while (count == 0) wait(notEmpty);
      // remove item from buffer
      count--;
      signal(notFull);
      return item;
   }
}
```

2 Producer-Priority

Behavior: When both a producer and consumer are ready and the buffer has space, the producer runs first. Useful in data-ingestion pipelines where fresh data must be prioritized.



2.1 Semaphores

```
semaphore empty    = N;
semaphore full     = 0;
semaphore mutex    = 1;
semaphore prodTurn = 1;  // priority gate

Producer() {
  wait(prodTurn);     // acquire priority turn first
  wait(empty);
  wait(mutex);
  // --- INSERT ITEM ---
  signal(mutex);
  signal(full);
  signal(prodTurn);   // release turn
}

Consumer() {
  wait(full);
  wait(prodTurn);     // yield if producer is contending
  wait(mutex);
  // --- REMOVE ITEM ---
  signal(mutex);
  signal(empty);
  signal(prodTurn);
}
```

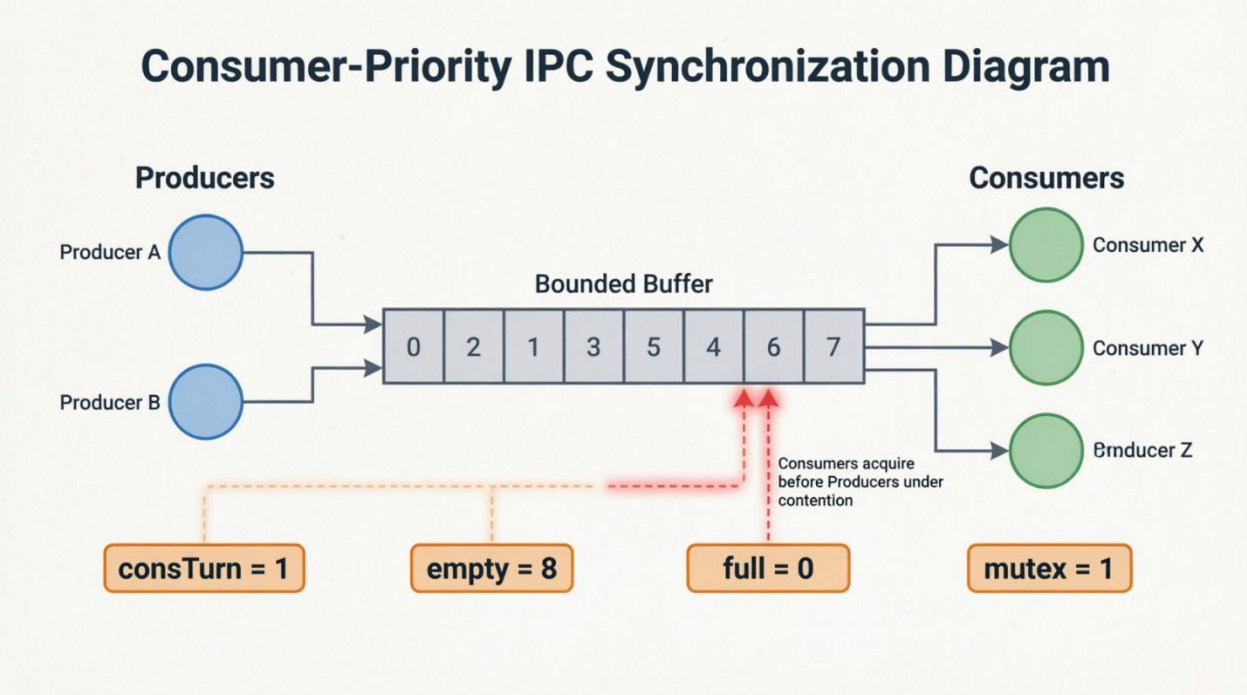
2.2 Monitors

```
monitor PC_ProdPriority {
  int count = 0;
  int waitProd = 0, waitCons = 0;
  condition notFull, notEmpty;

  void insert(item) {
    waitProd++;
    while (count == N) wait(notFull);
    waitProd--;
    // add item
    count++;
    signal(notEmpty);
  }
  item remove() {
    waitCons++;
    while (count == 0 || (count < N && waitProd > 0)) wait(notEmpty);
    waitCons--;
    // remove item
    count--;
    signal(notFull); // Always wake producer first if space exists
    return item;
  }
}
```

3 Consumer-Priority

Behavior: When both are ready and the buffer contains data, the consumer runs first. Common in real-time processing or low-latency systems where draining the buffer quickly is critical.



3.1 Semaphores

```
semaphore empty   = N;
semaphore full    = 0;
semaphore mutex   = 1;
semaphore consTurn = 1;  // priority gate

Producer() {
  wait(empty);
  wait(consTurn);     // yield priority to consumers
  wait(mutex);
  // --- INSERT ITEM ---
  signal(mutex);
  signal(full);
  signal(consTurn);
}

Consumer() {
  wait(consTurn);     // acquire priority turn first
  wait(full);
  wait(mutex);
  // --- REMOVE ITEM ---
  signal(mutex);
  signal(empty);
  signal(consTurn);
}
```

3.2 Monitors

```
monitor PC_ConsPriority {
   int count = 0;
   int waitProd = 0, waitCons = 0;
   condition notFull, notEmpty;

   void insert(item) {
      waitProd++;
      while (count == N || (count > 0 && waitCons > 0)) wait(notFull);
      waitProd--;
      // add item
      count++;
      signal(notEmpty); // Always wake consumer first if data exists
   }
   item remove() {
      waitCons++;
      while (count == 0) wait(notEmpty);
      waitCons--;
      // remove item
      count--;
      signal(notFull);
      return item;
   }
}
```