Synchronization and Bounded Buffer

Semaphore

- Synchronization tool that provides more sophisticated ways (than Mutex locks) for process to synchronize their activities.
- Semaphore ***S*** – integer variable
- Can only be accessed via two indivisible (atomic) operations
  - **`wait()`** and **`signal()`**
    - Originally called **`P()`** and **`V()`**
- Definition of the **`wait()` operation**

```
wait(S) {
    while (S <= 0)
        ; // busy wait
    S--;
}
```

- Definition of the **`signal()` operation**

```
signal(S) {
    S++;
}
```

Semaphore Usage

- **Counting semaphore** – integer value can range over an unrestricted domain
- **Binary semaphore** – integer value can range only between 0 and 1
  - Same as a **mutex lock**
- Can solve various synchronization problems
- Consider P_1 and P_2 that require S_1 to happen before S_2

  Create a semaphore "`synch`" initialized to 0

```
P1:
    S1;
    signal(synch);
P2:
    wait(synch);
    S2;
```

- Can implement a counting semaphore ***S*** as a binary semaphore

Semaphore Implementation

- Must guarantee that no two processes can execute the `wait()` and `signal()` on the same semaphore at the same time
- Thus, the implementation becomes the critical section problem where the `wait` and `signal` code are placed in the critical section
  - Could now have **busy waiting** in critical section implementation
    - But implementation code is short
    - Little busy waiting if critical section rarely occupied
- Note that applications may spend lots of time in critical sections and therefore this is not a good solution

Semaphore Implementation with no Busy waiting

- With each semaphore there is an associated waiting queue
- Each entry in a waiting queue has two data items:
  - value (of type integer)
  - pointer to next record in the list
- Two operations:
  - **block** – place the process invoking the operation on the appropriate waiting queue
  - **wakeup** – remove one of processes in the waiting queue and place it in the ready queue
- ```
  typedef struct{
  int value;
  struct process *list;
  } semaphore;
  ```
  ```

Implementation with no Busy waiting (Cont.)

```
wait(semaphore *S) {
   S->value--;
   if (S->value < 0) {
      add this process to S->list;
      block();
   }
}

signal(semaphore *S) {
   S->value++;
   if (S->value <= 0) {
      remove a process P from S->list;
      wakeup(P);
   }
}
```

Deadlock and Starvation

- **Deadlock** – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes
- Let *S* and *Q* be two semaphores initialized to 1

P_0	P_1
`wait(S);`	`wait(Q);`
`wait(Q);`	`wait(S);`
`...`	`...`
`signal(S);`	`signal(Q);`
`signal(Q);`	`signal(S);`

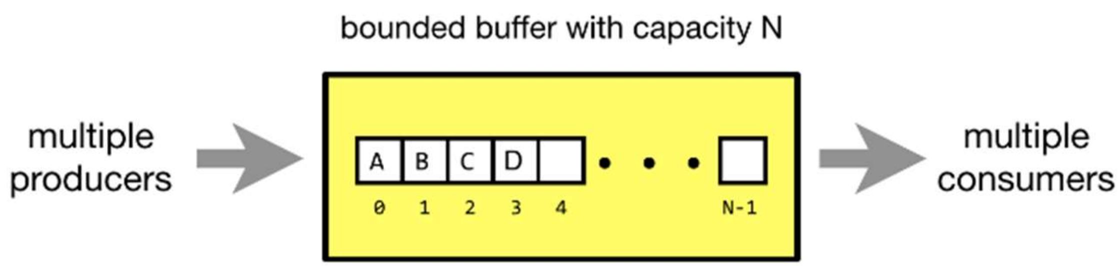
- **Starvation** – **indefinite blocking**
  - A process may never be removed from the semaphore queue in which it is suspended
- **Priority Inversion** – Scheduling problem when lower-priority process holds a lock needed by higher-priority process
  - Solved via **priority-inheritance protocol**

Classical Problems of Synchronization

- Classical problems used to test newly-proposed synchronization schemes
  - Bounded-Buffer Problem
  - Readers and Writers Problem
  - Dining-Philosophers Problem

Bounded-Buffer Problem

- *n* buffers, each can hold one item
- Semaphore **`mutex`** initialized to the value 1
- Semaphore **`full`** initialized to the value 0
- Semaphore **`empty`** initialized to the value n



Bounded Buffer Problem

- A bounded buffer with capacity N has can store N data items. The places used to store the data items inside the bounded buffer are called slots.

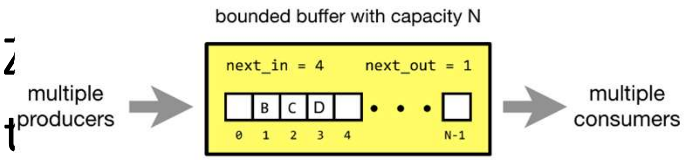
Without proper synchronization the following errors may occur.

- The producers doesn’t block when the buffer is full.
- A Consumer consumes an empty slot in the buffer.
- A consumer attempts to consume a slot that is only half-filled by a producer.
- Two producers writes into the same slot.
- Two consumers reads the same slot.

Bounded Buffer

- typedef struct {

```
int value[BUFFER_SIZ
int next_in, next_out
        } buffer_t;
```



Here,

- value used to store integer values in the buffer
- Index next_in is used to keep track of where to write the next data item to the buffer.
- Index next_out is used to keep track of from where to read the next data item from the buffer.

In the example, three data items B, C and D are currently in the buffer. On the next write data will be written to index next_in = 4. On the next read data will be read from index next_out = 0.

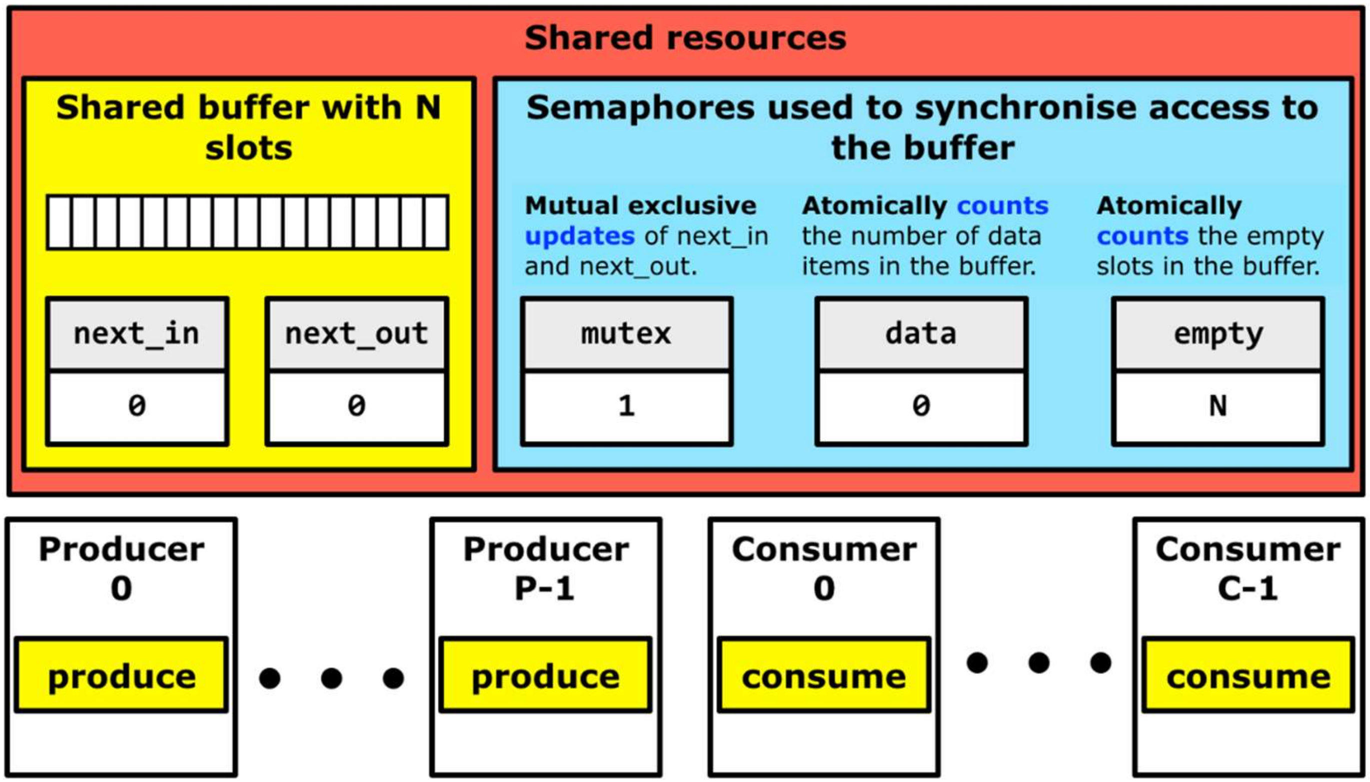
Bounded Buffer - Critical Section and mutual Exclusion

- All updates to the buffer state must be done in a critical section. More specifically, mutual exclusion must be enforced between the following critical sections:
  - A producer writing to a buffer slot and updating next_in.
  - A consumer reading from a buffer slot and updating next_out.
  - A binary semaphore can be used to protect access to the critical sections.

Synchronize producers and consumers

- Use one **semaphore** named **empty** to count the empty slots in the buffer. **Initialise** this semaphore to **N**.
  - A **producer** must **wait** on this semaphore before writing to the buffer.
  - A **consumer** will **signal** this semaphore after reading from the buffer.
- Use one **semaphore** named **data** to count the number of data items in the buffer. **Initialise** this semaphore to **0**.
  - A **consumer** must **wait** on this semaphore before reading from the buffer.
  - A **producer** will **signal** this semaphore after writing to the buffer.

Bounded Buffer



Bounded Buffer Problem (Cont.)

- The structure of the producer process

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

Bounded Buffer Problem (Cont.)

- The structure of the consumer process

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

Readers-Writers Problem

- 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
- Problem – allow multiple readers to read at the same time
  - Only one single writer can access the shared data at the same time
- Several variations of how readers and writers are considered – all involve some form of priorities
- Shared Data
  - Data set
  - Semaphore **`rw_mutex`** initialized to 1
  - Semaphore **`mutex`** initialized to 1
  - Integer **`read_count`** initialized to 0