Lecture 10
Semaphores

Semaphores

- Dijkstra proposed more general solution to mutual exclusion

- Semaphore **S is an abstract data type** that is accessed only through two standard atomic operations
  - wait( ) (also called P(), from Dutch word *proberen* "to test")
    - somewhat equivalent to a test-and-set
    - also involves *decrementing* the value of S
  - signal( ) (V(), from Dutch word *verhogen* "to increment")
    - *increments* the value of S

- OS provides ways to create and manipulate semaphores atomically

Semaphores

```
typedef struct {
    int value;
    PID *list[ ];
} semaphore;
```

Both wait() and signal() operations are atomic

```
wait(semaphore *s) {
    s→value--;
    if (s→value < 0) {
        add this process to s→list;
        sleep ( );
    }
}
```

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

Binary Semaphore

```
semaphore S = 1;  // initial value of semaphore is 1
int counter;      // assume counter is set correctly somewhere in code
```

Process P1:

```
wait(S);
    // execute critical section
    counter++;
signal(S);
```

Process P2:

```
wait(S);
    // execute critical section
    counter--;
signal(S);
```

- Both processes atomically wait() and signal() the semaphore S, which enables mutual exclusion on critical section code, in this case protecting access to the shared variable ***counter***
- ***This solves mutual exclusion but will also eliminate busy wait***

Problems with semaphores

shared R1, R2;

semaphore Q = 1; // binary semaphore as a mutex lock for R1

semaphore S = 1; // binary semaphore as a mutex lock for R2

Process P1:

wait(S); (1)

wait(Q); (3)

modify R1 and R2;

signal(S);

signal(Q);

Process P2:

wait(Q); (2)

wait(S); (4)

modify R1 and R2;

signal(Q);

signal(S);

Potential for deadlock

Deadlock

- In the previous example,
  - Each process will block on a semaphore
  - The signal() statements will never get executed, so there is no way to wake up the two processes
  - There is no rule about the order in which wait( ) and signal( ) operations may be invoked
  - In general, with more processes sharing more semaphores, the potential for deadlock grows

Other problematic scenarios

- A programmer mistakenly follows a wait() with a second wait() instead of a signal()

- A programmer forgets and omits the wait(mutex) or signal(mutex)

- A programmer reverses the order of wait() and signal()

Another problem with synchronization

shared R1, R2;

semaphore S=1; // binary semaphore as a mutex lock for R1 & R2

Process P1:	*Process P2:*	*Process P3:*
wait(S)	wait(S)	wait(S);
modify R1 and R2;	modify R1 and R2;	modify R1 and R2;
Signal (S);	signal(S);	signal(S);

Potential for starvation

Starvation

- The possibility that a process would never get to run
- For example, in a multi-tasking system the resources could switch between two individual processes
- Depending on how the processes are scheduled, a third process may never get to run
- The third task is being starved of accessing the resource

Semaphore Solution for Mutual Exclusion

```
shared lock = 1; // initial value of semaphore 1
shared int count;
Shared data_type buffer [MAX];
```

Code for p_1

```
while(1) {
    produce (nextdata)
    while(count==MAX);
    wait(lock);
    buffer[count] = nextdata;
    count++;
    signal(lock);
}
```

Code for p_2

```
while (1) {
    while(count==0);
    wait(lock);
    data = buffer[count-1];
    count--;
    signal(lock);
    consume(data);
}
```

Semaphore Solution for Mutual Exclusion

```
shared lock = 1; // initial value of semaphore 1
shared int count;
Shared data_type buffer [MAX];
```

Code for p_1

```
while(1) {
    produce (nextdata)
    while(count==MAX);
    wait(lock);
    buffer[count] = nextdata;
    count++;
    signal(lock);
}
```

Code for p_2

```
while (1) {
    while(count==0);
    wait(lock);
    data = buffer[count-1];
    count--;
    signal(lock);
    consume(data);
}
```

- Busy waiting removed from the mutual exclusion when waiting on lock
- Does it solve all busy waiting issues?

pthread Synchronization

- Mutex locks
  - Used to protect critical sections
- Some implementations provide semaphores through POSIX SEM extension
  - Not part of pthread standard

```
#include <pthread.h>
pthread_mutex_t m; //declare a mutex object
pthread_mutex_init (&m, NULL); // initialize mutex object
```

```
//thread 1
pthread_mutex_lock (&m);
   //critical section code for th1
pthread_mutex_unlock (&m);
```

```
//thread 2
pthread_mutex_lock (&m);
   //critical section code for th2
pthread_mutex_unlock (&m);
```

pthread mutex

- pthread mutexes can have only one of two states: lock or unlock
- Important restriction
  - Mutex ownership:
    Only the thread that locks a mutex can unlock that mutex
  - Mutexes are strictly used for mutual exclusion while binary semaphores can also be used for synchronization between two threads or processes

POSIX semaphores

#include <semaphore.h>

int sem_init(sem_t *sem, int pshared, unsigned int value);
// pshared: 0 (among threads); 1 (among processes)

int sem_wait(sem_t **sem*); //same as wait( )

int sem_post(sem_t *sem); //same as signal( )

sem_getvalue( ); // check the current value of the semaphore
sem_close( ); // done with the semaphore then close

Producer-Consumer Problem
also known as
Bounded Buffer Problem

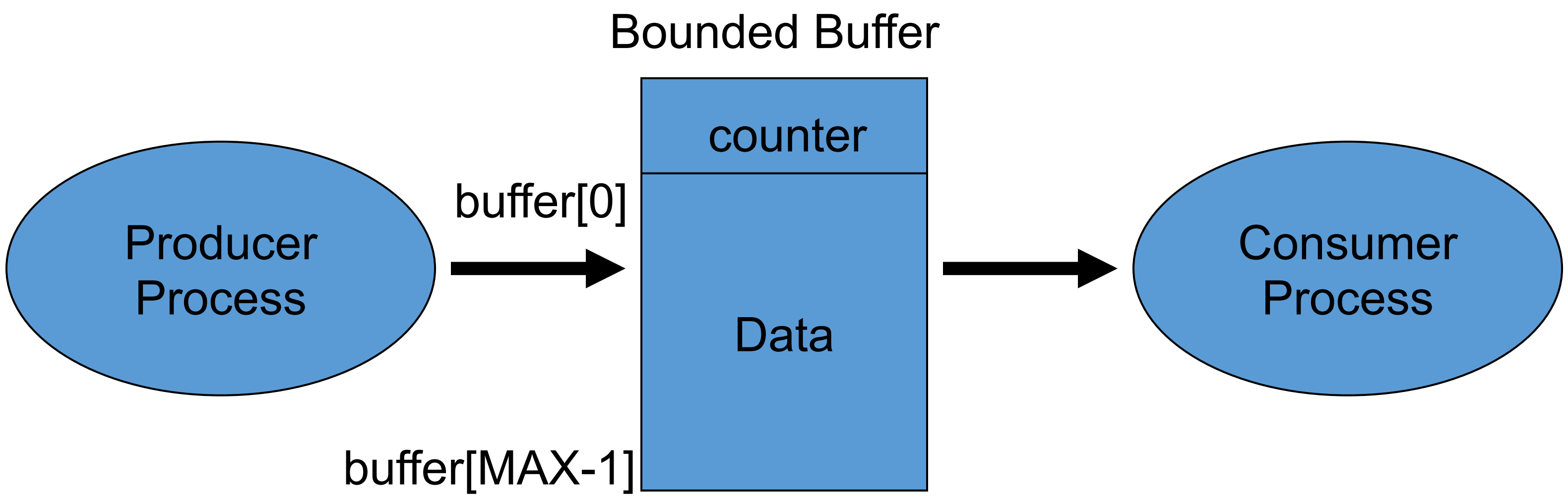
- We have already seen this problem with one producer and one consumer

- General problem: **multiple** producers and multiple consumers

- **Producers** puts new information in the buffer

- **Consumers** takes out information from the buffer

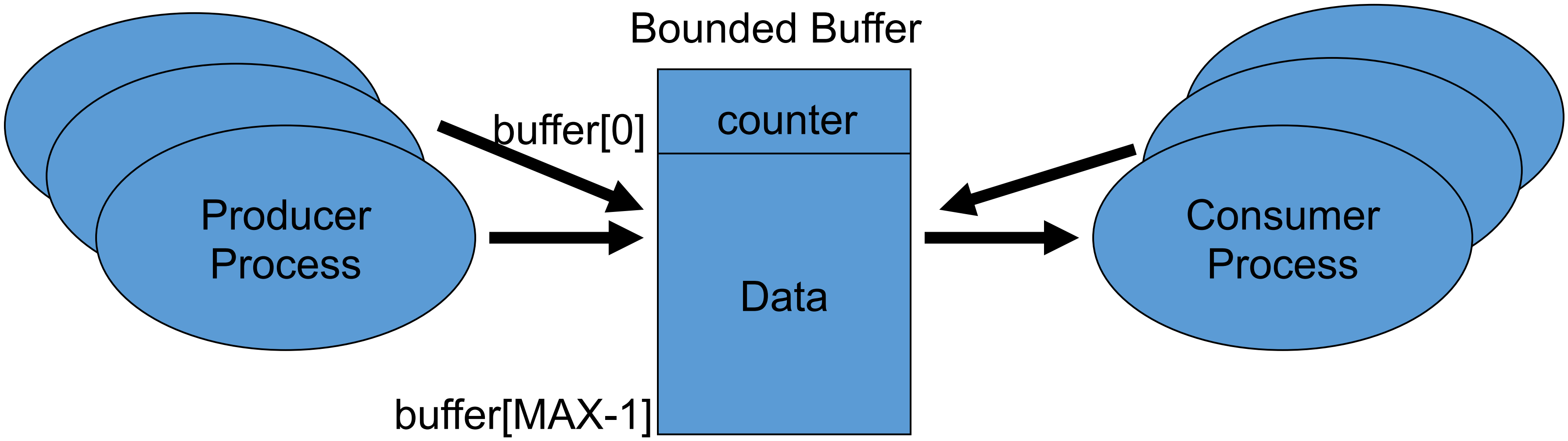
Prior Bounded-Buffer P/C Approach



- Producer places data into a buffer at the next available position

- Consumer takes information from the earliest item

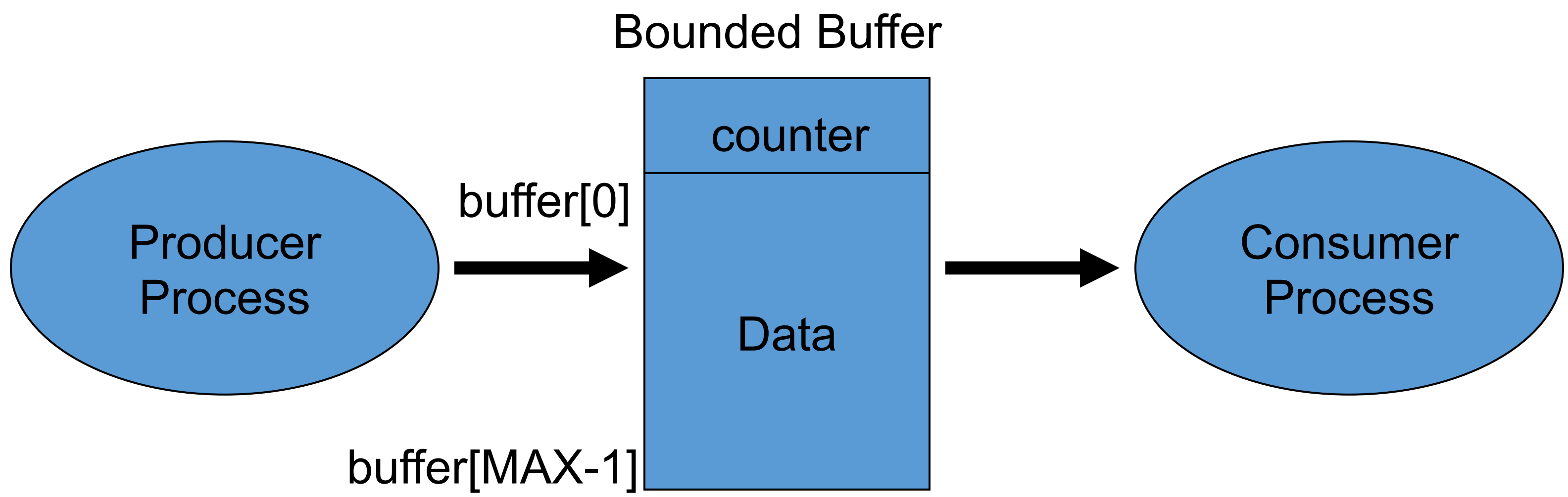
Prior Bounded-Buffer P/C Approach



- Producer places data into a buffer at the next available position

- Consumer takes information from the earliest item

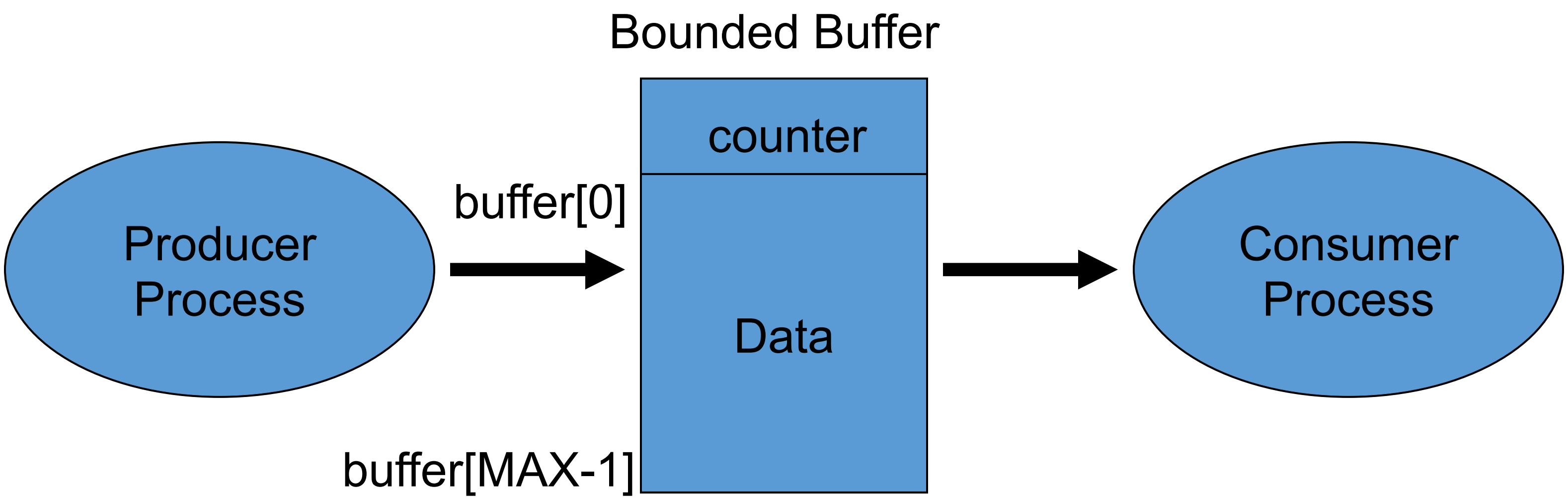
Prior Bounded-Buffer P/C Approach




```
while(1) {
    produce (nextdata)
    while(count==MAX);
    Acquire(lock);
    buffer[count] = nextdata;
    count++;
    Release(lock);
}
```

```
while (1) {
    while(count==0);
    Acquire(lock);
    data = buffer[count-1];
    count--;
    Release(lock);
    consume(data);
}
```

Prior Bounded-Buffer P/C Approach

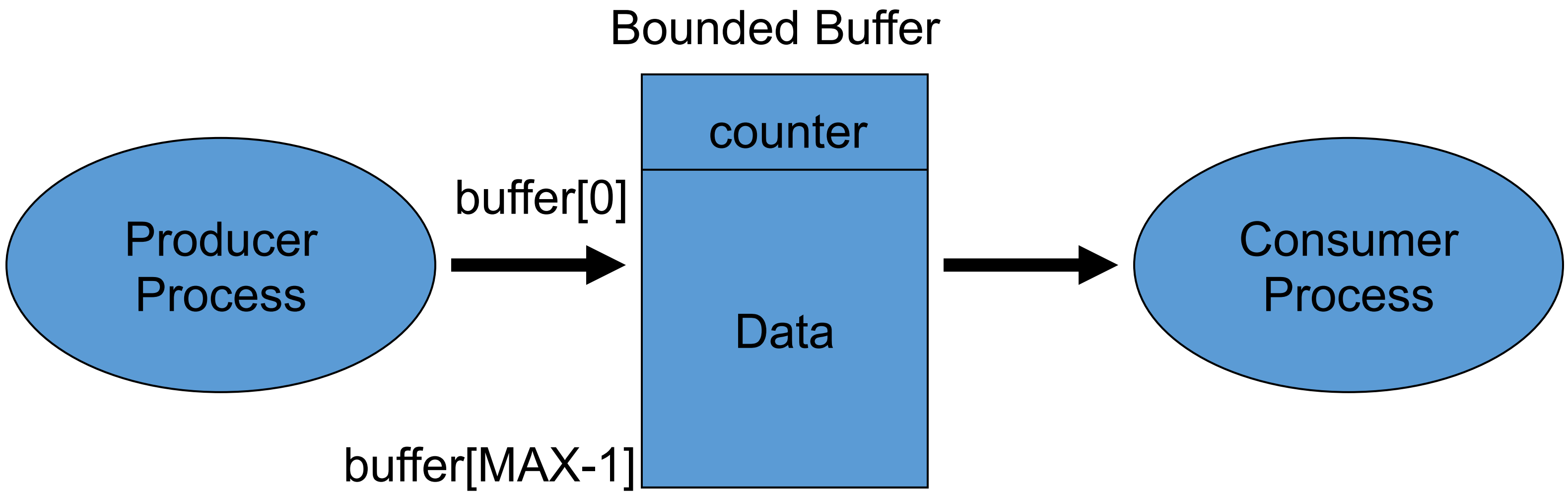



```
while(1) {
    produce (nextdata)
    while(count==MAX);
    Acquire(lock);
    buffer[count] = nextdata;
    count++;
    Release(lock);
}
```

```
while (1) {
    while(count==0);
    Acquire(lock);
    data = buffer[count-1];
    count--;
    Release(lock);
    consume(data);
}
```

Busy-wait!

Prior Bounded-Buffer P/C Approach




```
while(1) {
    produce (nextdata)
    while(count==MAX);
    Acquire(lock);
    buffer[count] = nextdata;
    count++;
    Release(lock);
}
```

```
while (1) {
    while(count==0);
    Acquire(lock);
    data = buffer[count-1];
    count--;
    Release(lock);
    consume(data);
}
```

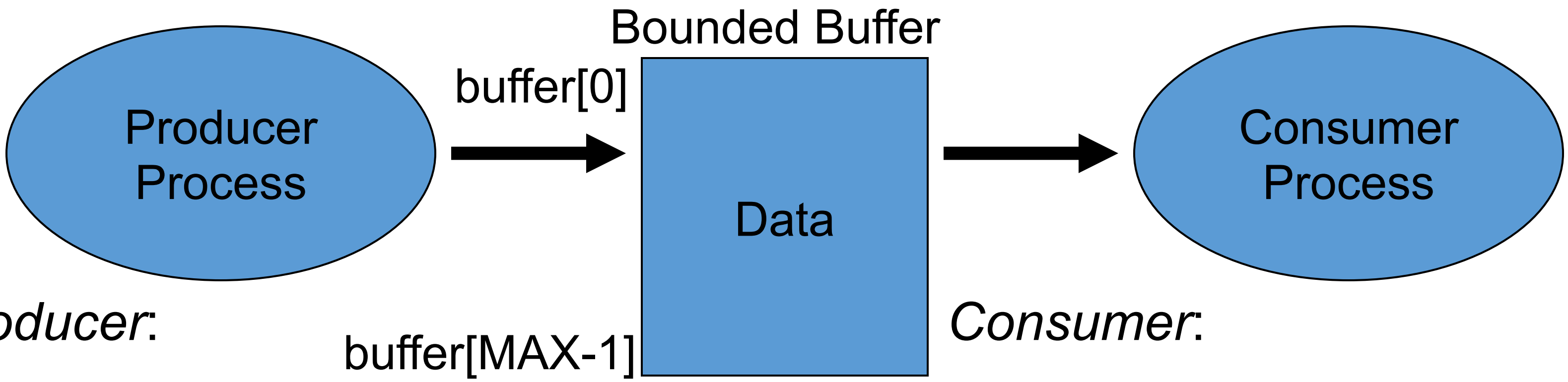
while (TS(&lock));

Busy-wait!

Bounded-Buffer Goals

- In the prior approach, both the producer and consumer are ***busy-waiting*** using locks
- Instead, want both to sleep as necessary
  - Goal #1: Producer should block when buffer is full
  - Goal #2: Consumer should block when the buffer is empty
  - Goal #3: mutual exclusion when buffer is partially full
    - Producer and consumer should access the buffer in a synchronized mutually exclusive way

Bounded Buffer Solution



Producer:

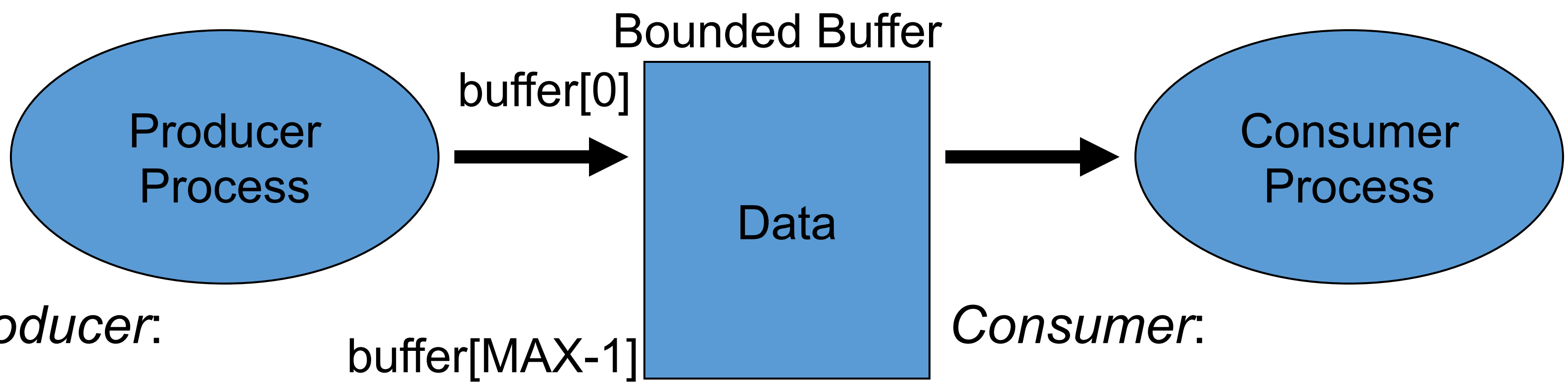
```
while(1) {
    …
    P(mutex)   // Goal 3: mutual
                  exclusion
       add item to buffer
    V(mutex)
    …
}
```

Consumer:

```
while(1) {
    ...
    P(mutex)   // Goal 3: mutual
                  exclusion
       remove item from buffer
    V(mutex)
    ...
}
```

Assume *mutex*$_{init}$ = 1

Bounded Buffer Solution (2)



Producer:

Consumer:

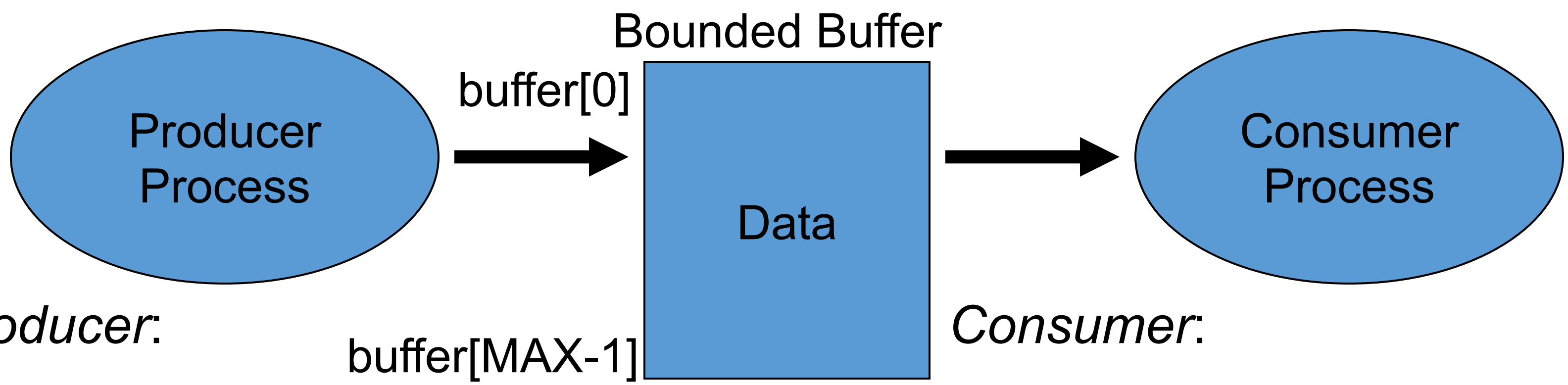
Goal #1: Producer should block when buffer is full

```
while(1) {
  P(empty) // Goal 1: full buffer
              blocks producer
    P(mutex)
      add item to buffer
    V(mutex)
    …
}
```

Assume *mutex*$_{init}$=1, *empty*$_{init}$=MAX

```
while(1) {
    ...
    P(mutex)
      remove item from buffer
    V(mutex)
  V(empty) // Goal 1: full
   buff
                blocks producer
}
```

Bounded Buffer Solution (3)



Producer:

Consumer:

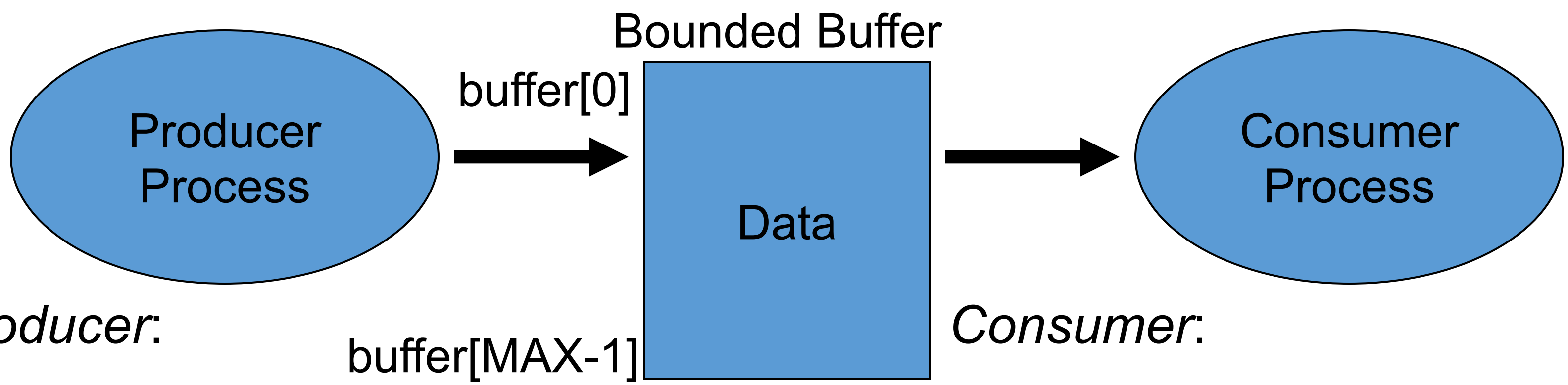
Goal #2: Consumer should block when the buffer is empty

```
while(1) {
  P(empty)
    P(mutex)
      add item to buffer
    V(mutex)
  V(full) // Goal 2: empty buffer
             blocks consumer
}
```

```
while(1) {
  P(full) // Goal 2: empty
                buff blocks consumer
    P(mutex)
      remove item from buffer
    V(mutex)
  V(empty)
}
```

Assume *mutex*$_{init}$=1, *empty*$_{init}$=MAX, *full*$_{init}$=0

Bounded Buffer Solution (4)



Producer:

```
while(1) {
  P(empty)
    P(mutex)
      add ite
    V(mutex)
  V(full)
}
```

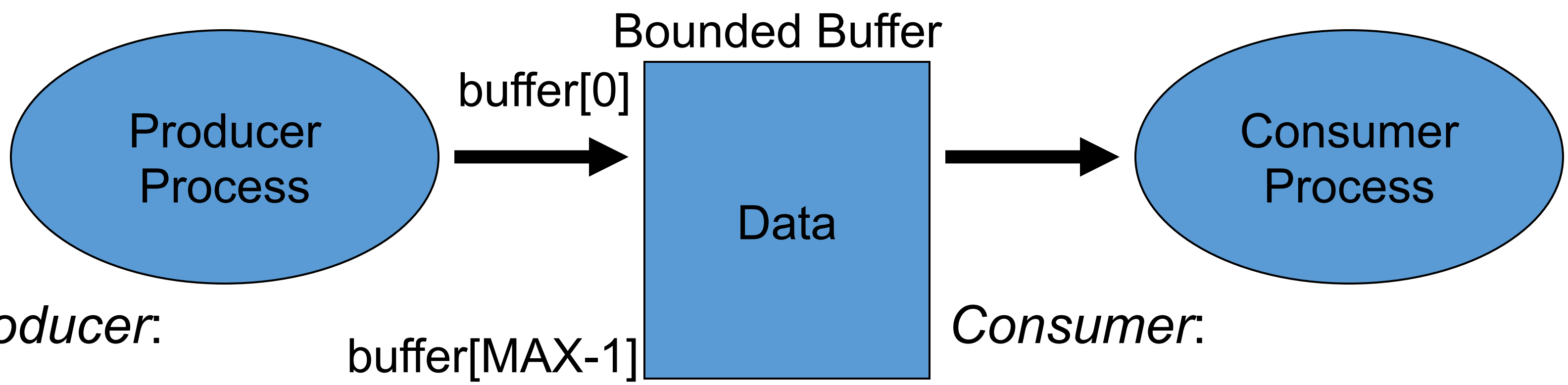
Consumer:

```
while(1) {
      ex)
      emove item from buffer
      ex)
      )
}
```

Achieves:
1) mutual exclusion
2) blocked producer if buffer full
3) blocked consumer if buffer empty
4) no deadlock
5) is efficient (no busy wait)

Assume $mutex_{init}$=1, $empty_{init}$=MAX, $full_{init}$=0

Bounded Buffer Solution (5)



Producer:

```
while(1) {
  P(space_avail)
    P(mutex)
      add item to buffer
    V(mutex)
  V(items_avail)
}
```

Consumer:

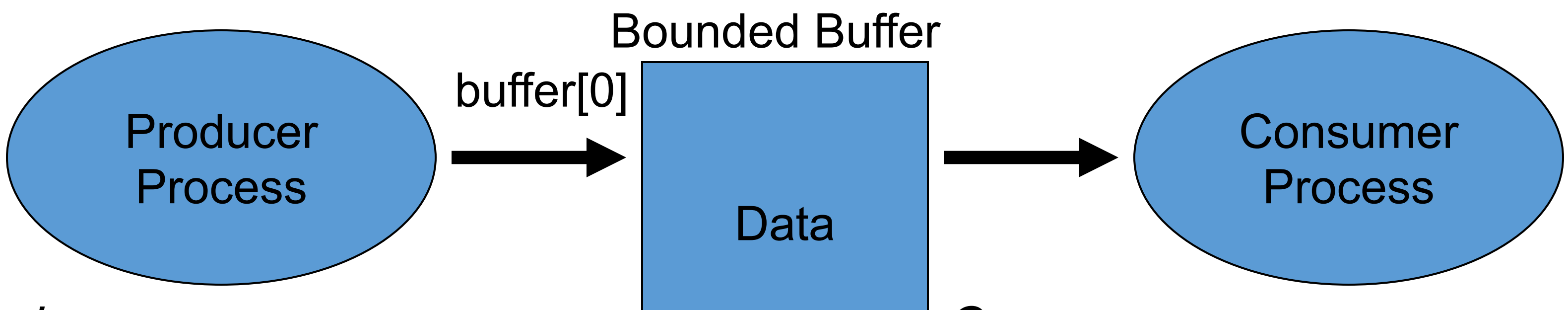
```
while(1) {
  P(items_avail)
    P(mutex)
      remove item from buffer
    V(mutex)
  V(space_avail)

}
```

Assume $mutex_{init}$=1,
$space\_avail_{init}$=MAX,
$items\_avail_{init}$=0

Bounded-Buffer Design

- **Goal #1: Producer should block when buffer is full**
  - Use a counting semaphore called *empty* that is initialized to $empty_{init}$ = MAX
  - Each time the producer adds an object to the buffer, this decrements the # of empty slots, until it hits 0 and the producer blocks
- **Goal #2: Consumer should block when the buffer is empty**
  - Define a counting semaphore *items_avail* that is initialized to $items\_avail_{init}$ = 0
  - *items_avail* tracks the # of full slots and is incremented by the producer
  - Each time the consumer removes a full slot, this decrements *items_avail*, until it hits 0, then the consumer blocks
- **Goal #3: Mutual exclusion when buffer is partially full**
  - Use a mutex semaphore to protect access to buffer manipulation, $mutex_{int}$ = 1

Bounded Buffer Solution (6)



Producer:

```
while(1) {
  P(space_avai
    P(mutex)
      add item to buffer
    V(mutex)
  V(items_avail)
}
```

Will this solution work for multiple Producers and multiple customers?

```
il)
      remove item from buffer
    V(mutex)
  V(space_avail)
}
```

Assume $mutex_{init}$=1,
$space\_avail_{init}$=MAX,
$items\_avail_{init}$=0