PRODUCER-CONSUMER PROBLEM

OVERVIEW

- **producer-consumer problem** (also known as the **bounded-buffer problem**) is a multi-process synchronization problem.
- The problem describes two processes, the producer and the consumer, who share a common, fixed-size buffer.
- Producer :- The producer's job is to generate a piece of data, put it into the buffer and start again.
- Consumer :- The consumer is consuming the data (i.e., removing it from the buffer) one piece at a time.

PROBLEM

The problem is to make sure that the producer won't try to add data into the buffer if it's full and that the consumer won't try to remove data from an empty buffer.

SOLUTION FOR THE PRODUCER

- Producer either go to sleep or discard data if the buffer is full.
- The next time the consumer removes an item from the buffer and it notifies the producer who starts to fill the buffer again.

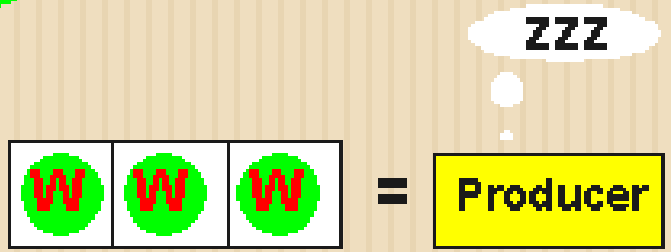
SOLUTION FOR THE CONSUMER

- consumer can go to sleep if it finds the buffer to be empty.
- The next time the producer puts data into the buffer, it wakes up the sleeping consumer.

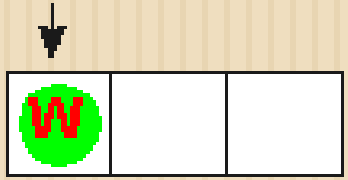
PRODUCER

1.Make new widget 

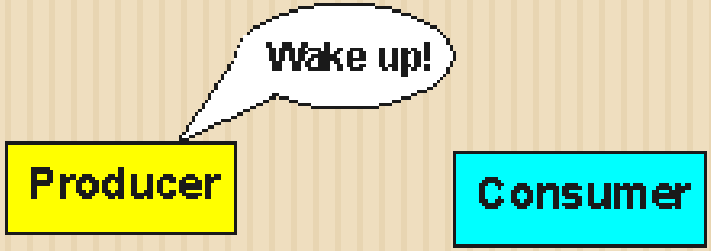
2.If buffer is full, go to sleep



3.Put widget in buffer

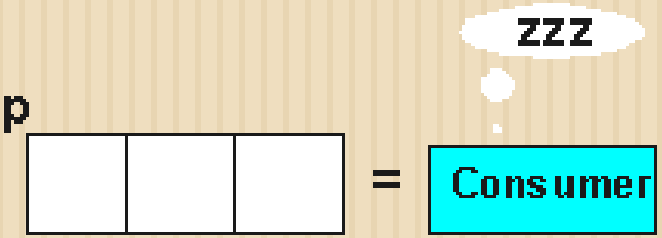


4.If buffer was empty, wake consumer

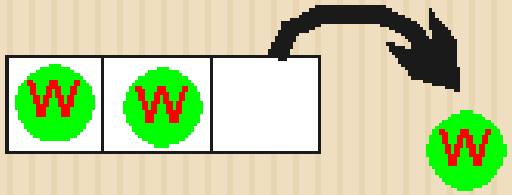


CONSUMER

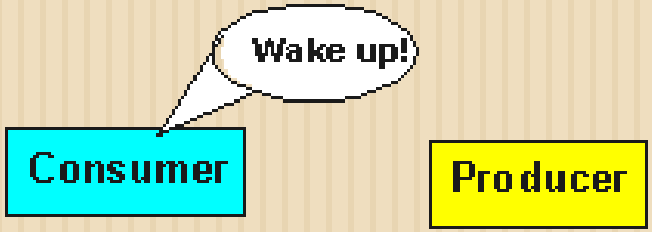
1.If buffer is empty, go to sleep



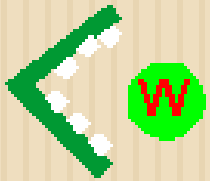
2. Take widget from buffer



3.If buffer was full, wake producer



4.Consume the Widget



INADEQUATE SOLUTION

```
BufferSize = 3;
 count = 0;

Producer() {
      int item;
      WHILE (true)
      {
            make_new(item);                          // create a new item to put in the buffer
            IF(count==BufferSize)  Sleep();          // if the buffer is full, sleep
            put_item(item);                  // put the item in the buffer
            count = count + 1;               // increment count of items
            IF (count==1)
            Wakeup(Consumer);                // if the buffer was previously empty, wake the consumer
      }
}
```

INADEQUATE SOLUTION

```
Consumer() {
    Int item;
    WHILE(true)
    {
        IF(count==0) Sleep();                          // if the buffer is empty, sleep
        remove_item(item);                             // take an item from the buffer
        count = count – 1;                             // decrement count of items
        IF(count==N-1) Wakeup(Producer);  // if buffer was previously full, wake
                                                                the producer
        Consume_item(item);                            // consume the item
    }
}
```

Problem with the above Solution

- The problem with this solution is that it contains a race condition that can lead into a deadlock. Consider the following scenario:
- The consumer has just read the variable itemCount, noticed it's zero and is just about to move inside the if-block.
- Just before calling sleep, the consumer is interrupted and the producer is resumed.
- The producer creates an item, puts it into the buffer, and increases itemCount.
- Because the buffer was empty prior to the last addition, the producer tries to wake up the consumer.
- Unfortunately the consumer wasn't yet sleeping, and the wakeup call is lost. When the consumer resumes, it goes to sleep and will never be awakened again. This is because the consumer is only awakened by the producer when itemCount is equal to 1.
- The producer will loop until the buffer is full, after which it will also go to sleep.
- Since both processes will sleep forever, we have run into a deadlock. This solution therefore is unsatisfactory.

Solution using Semaphore

```
BufferSize = 3;
semaphore mutex = 1;                    // Controls access to critical section
semaphore empty = BufferSize;           // counts number of empty buffer slots
semaphore full = 0;                     // counts number of full buffer slots

Producer()
{
    int item;
    while (TRUE)
    {
        make_new(item);                 // create a new item to put in the buffer
        down(&empty);                   // decrement the empty semaphore
        down(&mutex);                   // enter critical section
            put_item(item);             // put item in buffer
        up(&mutex);                     // leave critical section
        up(&full);                      // increment the full semaphore
    }
}
```

Solution using Semaphore

```
Consumer(){
    int  item;
     while (TRUE) {
        down(&full);                    // decrement the full semaphore
        down(&mutex);                   // enter critical section
                    remove_item(item);          // take a item from the buffer
       up(&mutex);                  // leave critical section
       up(&empty);                      // increment the empty semaphore
       consume_item(item);      // consume the item
    }
}
```

Solution using Semaphore

In the above example there are three semaphores.

- ***Full***, used for counting the number of slots that are full;
- ***empty***, used for counting the number of slots that are empty;
- ***mutex***, used to enforce mutual exclusion.

Solution using Monitors

- The problem with the semaphores described in the previous page is that they require the programmer to use low-level system calls. It is easy to put these system calls in the wrong order, resulting in a **deadlock**. A higher level solution would make implementing mutual exclusion and synchronization a little easier.
- **Monitors** They are a high level construct found in some programming languages which contain variables, procedures, and data structures. They are found in languages like C++ and Ada, they only allow access to the variables they contain through the functions they contain. **Only one process may execute a function within a monitor at any one time.** This is what allows monitors to enforce mutual exclusion.

Solution using Monitors

```
monitor ProducerConsumer
condition full, empty;
int count;
procedure enter(item item) {
    if (count == N)  wait(full);          // if buffer is full,
       put_item(item);                    // put item in buffer
    count = count + 1;                    // increment count of full
                                          //slots
    if (count == 1)  signal(empty); // if buffer was empty,
                                          //wake consumer
}
```

Solution using Monitors

```
    procedure remove() {
        if (count == 0) wait(empty);   // if buffer is empty, block
        item = remove_item(item);                  // remove item from
                                                   //buffer
        count = count - 1;             // decrement count of full slots
        if (count == N-1) signal(full); // if buffer was full,
                                                   //wake producer
        return item;
}
end monitor;
```

Solution using Monitors

```
Producer();
 {
    while (TRUE)
    {
       Item = make_item();              // make a new item
       ProducerConsumer.enter(item);    // call enter function in
                                        //monitor
    }
}
```

Solution using Monitors

```
Consumer();
{
    while (TRUE)
        {
        Item = ProducerConsumer.remove(); // call remove
                                          //function in monitor
        consume_item(item);               // consume an item
    }
}
```

Other Problems to Study for QUIZ

- Cigarette smokers problem
- Dining philosophers problem
- Readers-writers problem
- Sleeping barber problem