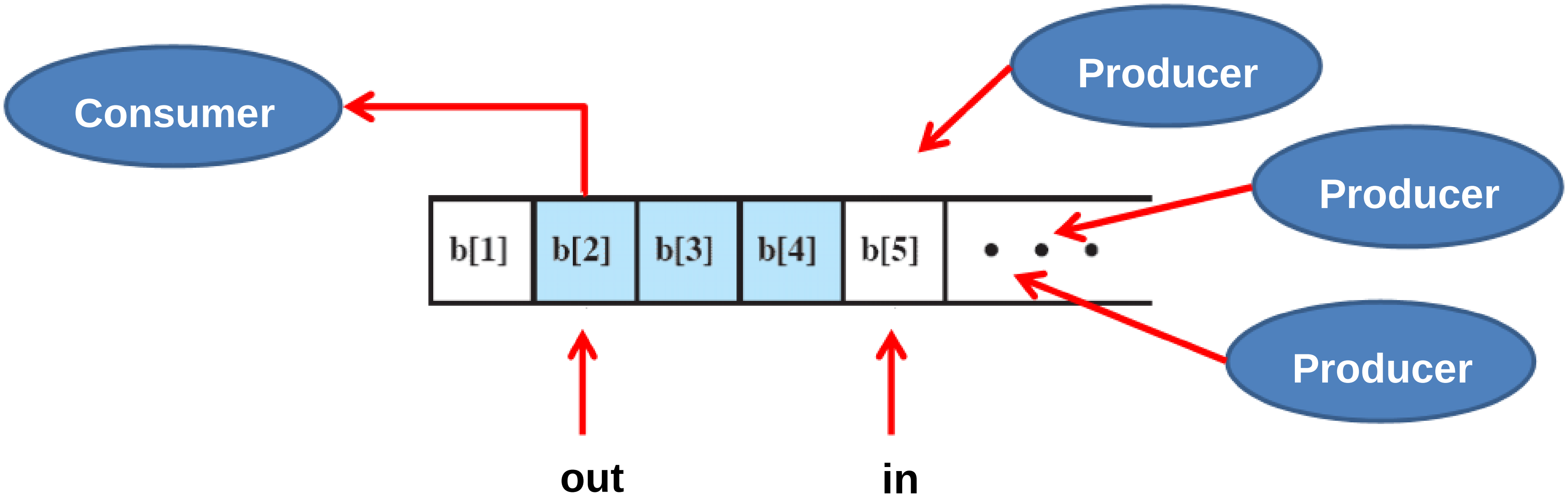
Producer – Consumer Problem

Many Producers – One Consumer

- Producer and consumer processes exchange data items via a buffer
- One or moreproducers put data into the buffer
- One consumer takes informationout of the buffer
- Objective: prevent any overlap of buffer operations !



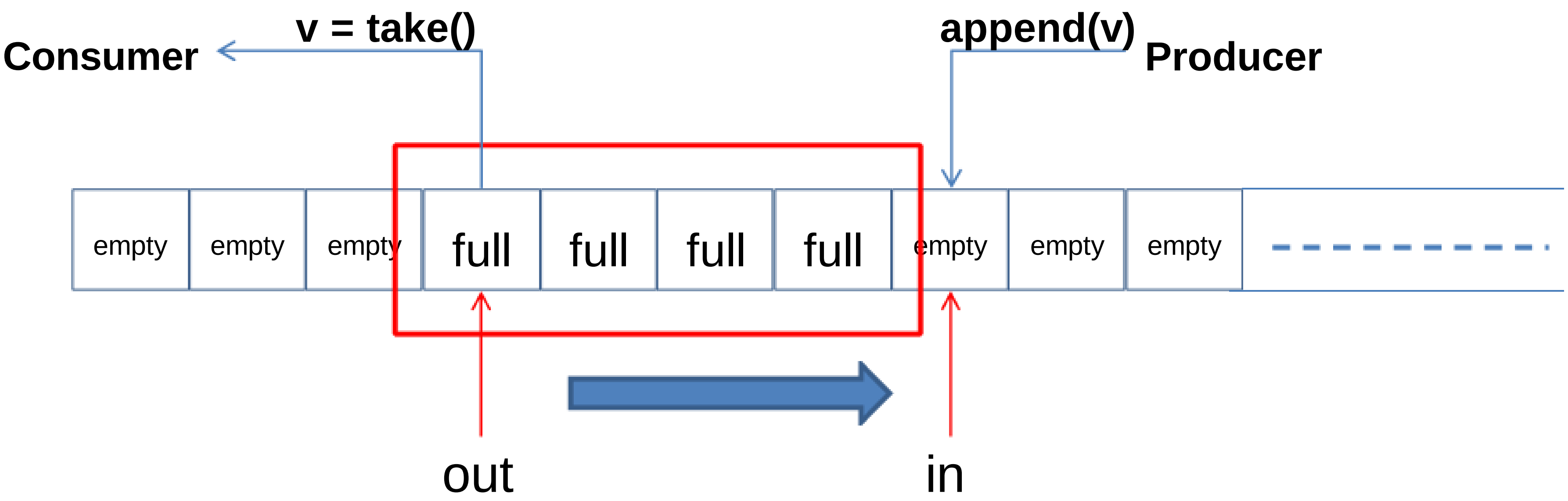
Producer Consumer

- Managing a shared buffer with a semaphore
- Many Producers:
  - –Writes data into buffer
  - –Can only write if there is space
- One Consumer:
  - –Read data from buffer
  - –Can only read if there is something in buffer

Producer – Consumer
Infinite Buffer

- Assumption: infinite Buffer
- Producer can append elements to buffer any time (because no buffer restrictions)
- Consumer has to wait for data
- Two buffer pointers
  - "in" : points to next free place in buffer
  - "out": points to next data element in buffer that can be read

Infinite Buffer



out < in

Producer – Consumer Implementation

- Combine mutual exclusion and condition synchronisation
- Uses two semaphores
  - Mutex semaphore:
    - mutual exclusion between producer and consumer
  - Counting semaphore:
    - Number of slots available in buffer
- Mutual exclusion
  - Only one process may access buffer at a time
- Condition synchronisation
  - Consumer may only read, if there is at least one data item stored in the buffer

Producer – Consumer
Semaphores

- Mutual exclusion
  - Only one process may access buffer at any time
  - Saveguarded with semaphore:
    - We use a semaphore “mutex” to control mutual exclusion
- Condition for consumer:
  - Consumer can only read/remove elements from buffer, if there is at least one unread data element stored in buffer
  - Semaphore
    - We use a counting semaphore “N” that counts the number of data elements stored in the buffer

Producer – Consumer
Unlimited Buffer

```
init(mutex,1);  init(N,0);  in = 0;  out = 0;  b[] = empty_list;
```

```
while(TRUE)
{
    x = produce();
    wait(mutex);
    append(x);
    signal(mutex);
    signal(N);
}
```

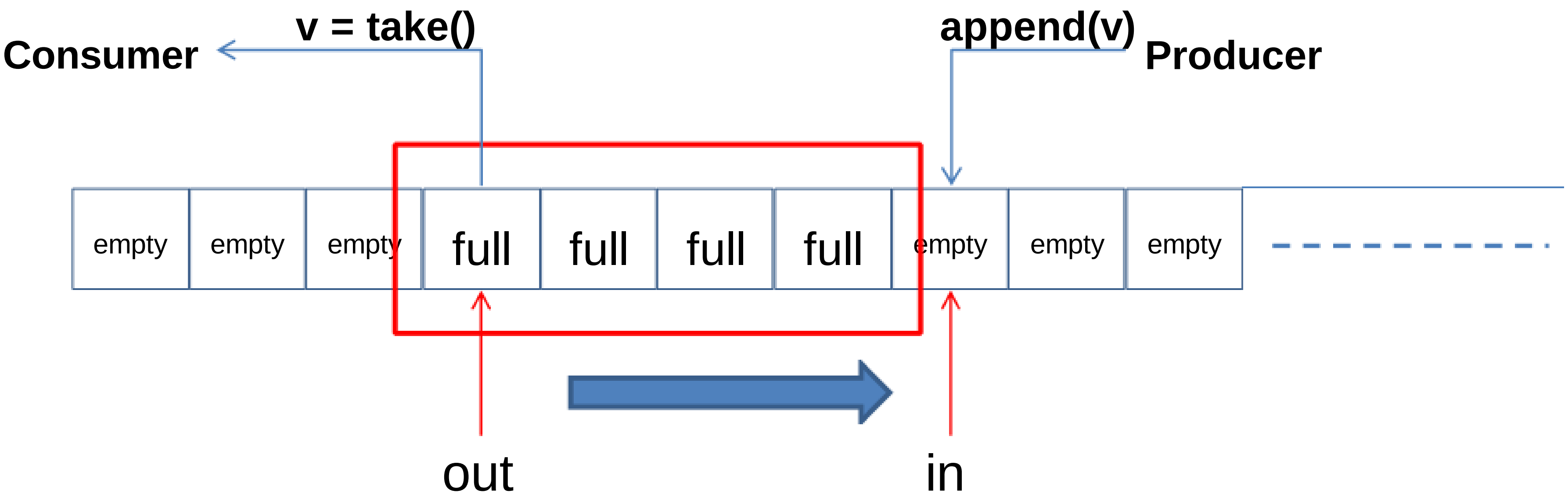
```
while(TRUE)
{
    wait(N);
    wait(mutex);
    y = take();
    signal(mutex);
    consume(y);
}
```

```
append(x)
{
    b[in] = x;
    in ++;
}
```

```
take()
{
    y = b[out];
    out ++;
    return y;
}
```

- Two semaphores
  - Saveguard update on buffer (mutex)
  - Count elements currently stored in buffer (N)

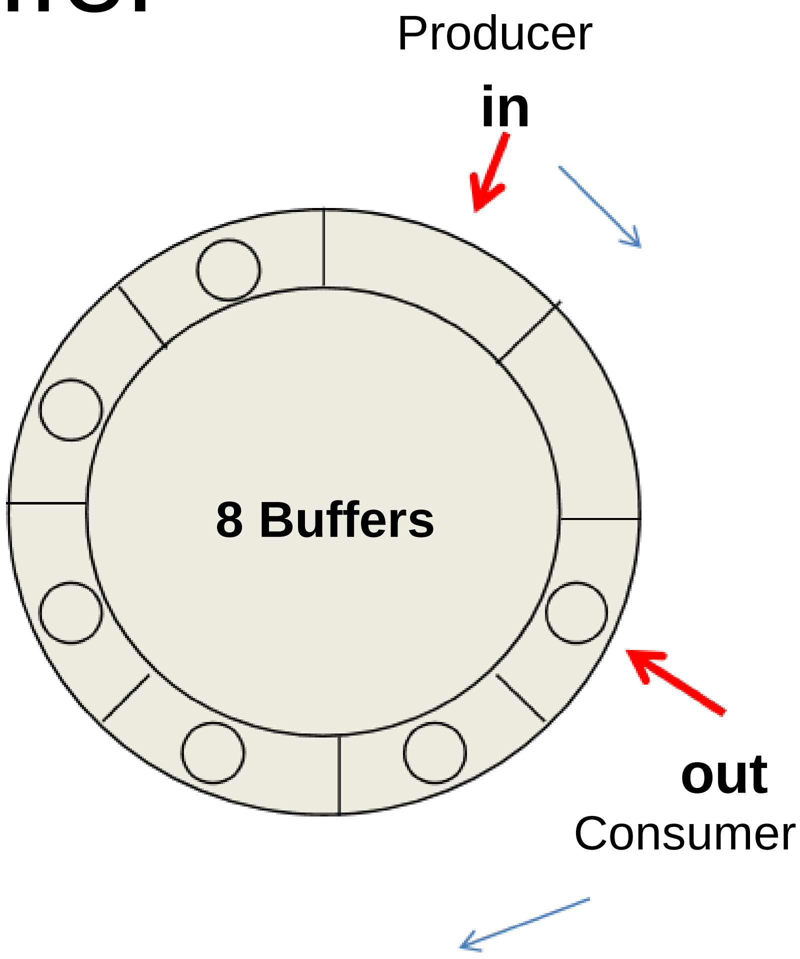
Infinite Buffer

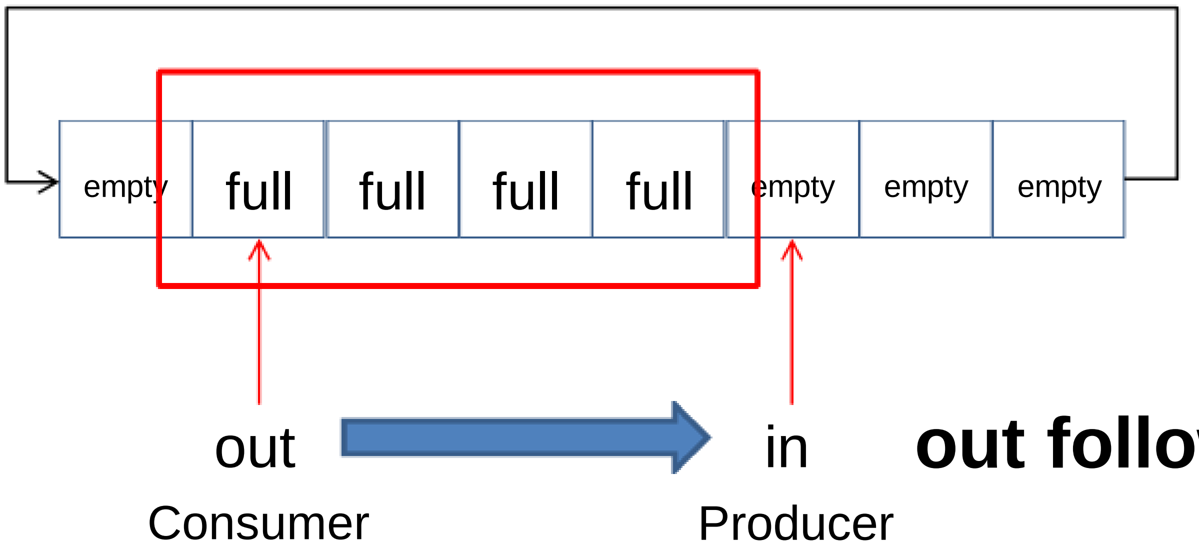


- If we assume an infinite buffer, a “window of full places” appears to move across the buffer
  - Producer always takes a new empty place
  - Places that become free again, are never reused
- Infinite buffer can be implemented with a bounded buffer or “ring buffer”

Bounded Buffer

- Implemented as a Ring Buffer
  - If "in" pointer reaches last location in buffer, it is reset to first location
  - If "out" pointer reaches last location in buffer, it is reset to first location
  - out and in move clockwise

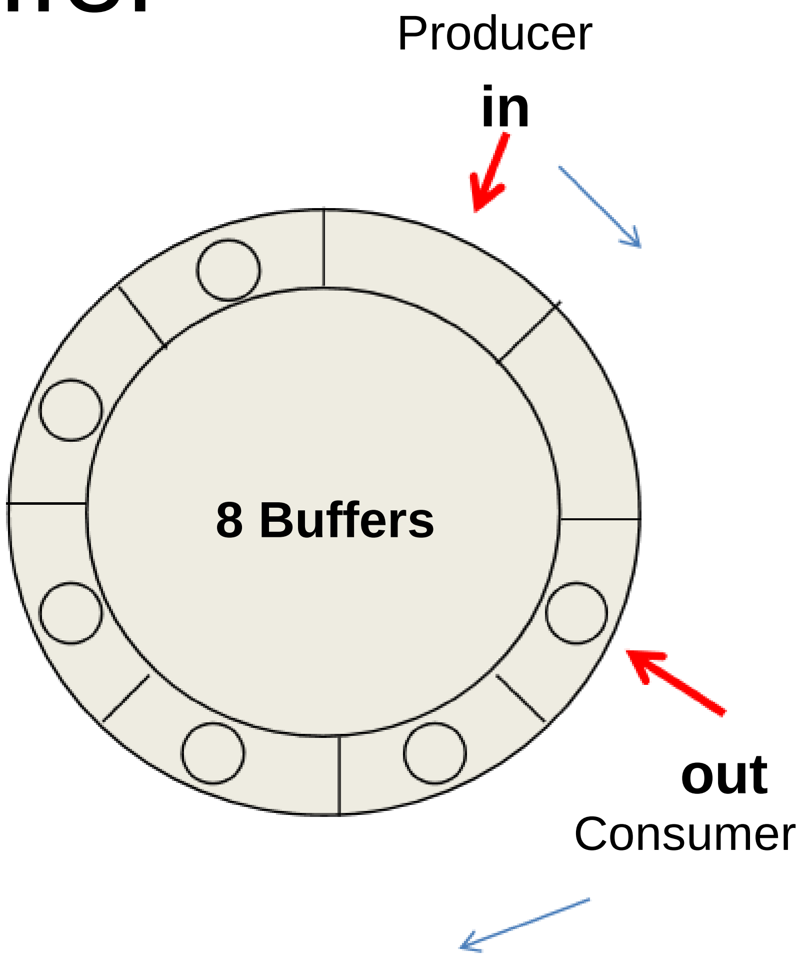


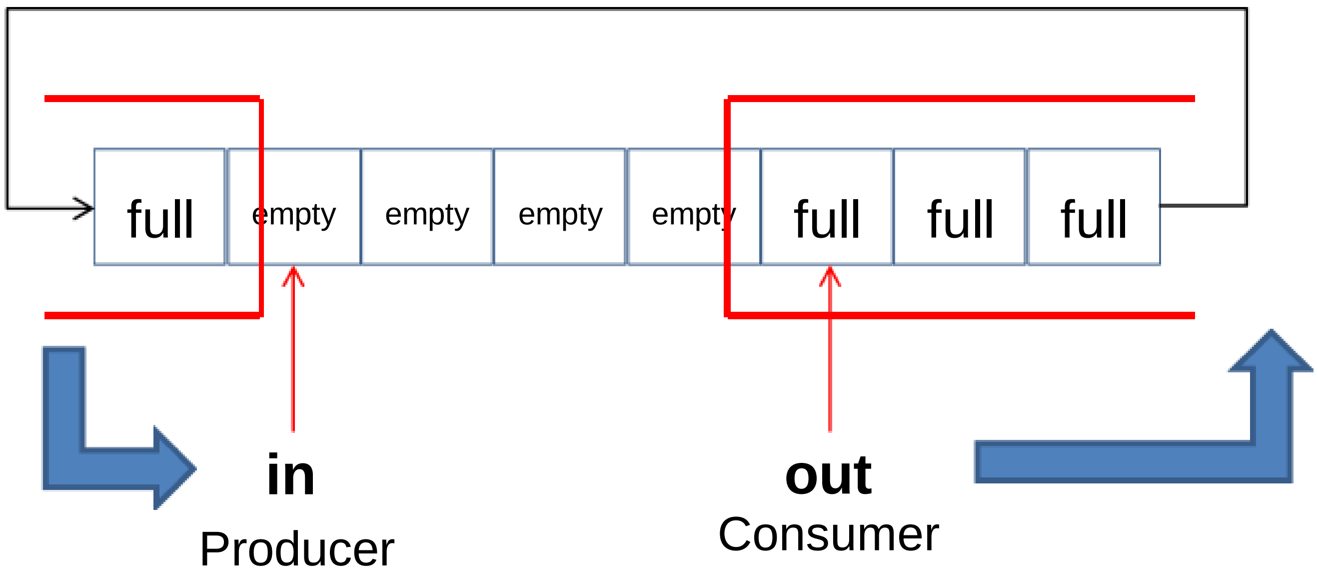


out follows in

Bounded Buffer

- Implemented as a Ring Buffer
  - If “in” pointer reaches last location in buffer, it is reset to first location
  - If “out” pointer reaches last location in buffer, it is reset to first location
  - out and in move clockwise





out follows in

Producer – Consumer Ringbuffer

- A ring buffer is a finite array of elements
- Consequence:
  - Producers can only write, if there is at least one empty slot
- Uses three semaphores
  - Mutex semaphore:
    - mutual exclusion between producer and consumer
  - Counting semaphore – control consumer:
    - Number of data items in buffer: consumer can only read, if there is at least one new data item in buffer
  - Counting semaphore – control producer:
    - Number of data items in buffer: producer can only write, if there is at least one empty slot

Producer – Consumer
Ring Buffer

```
init(mutex,1);  init(full,0);  init(empty,  N);
in  =  0;  out  =  0;  buffer[N]  ;
```

Producer

```
while(TRUE)
{
    x  =  produce();
    wait(empty);
    wait(mutex);
    append(x);
    signal(mutex);
    signal(full);
}
```

Consumer

```
while(TRUE)
{
    wait(full);
    wait(mutex);
    y  =  take();
    signal(mutex);
    signal(empty);
    consume(y);
}
```

```
append(x)
{
    b[in]  =  x;
    in  =  (in+1)  mod  N;
}
```

```
take()
{
    y  =  b[out];
    out  =  (out+1)  mod  N;
    return  y;
}
```

- Bounded buffer:
  - Bufferlimited to N places, is managed as a circular buffer

Sequence of wait() and signal()

- The sequence of signal() call can be arbitrary
- The sequence of wait() calls is essential

```
Producer:
      v = produce();
      wait(empty);
      wait(mutex);
      append(v);
```

```
Consumer:

      wait(mutex);
      wait(full);
      v = take() ;
```

WRONG

- Calling first “wait(mutex)” allows consumer to enter critical section without testing whether bufferis empty
- Leads to Deadlock: consumer never leaves critical section, because it is blocked by “wait(full)”

Deadlock Situation

- The sequence of signal() call can be arbitrary
- The sequence of wait() calls is essential

```
while(TRUE)
{
      x = new_data();
      wait(empty);
      wait(mutex);
      append(x);
      signal(mutex);
      signal(full);
}
```

```
while(TRUE)
{
      wait(mutex);
      wait(full);
      y = take();
      signal(mutex);
      signal(empty);
      consume(y);
}
```

WRONG !!

ReaderWriterProblem Shared Read Access

- Onewriter (producer), many readers (consumers)
- Write access must be exclusive
  - When writer writes, no reader is allowed to access shared resource
- Read access isshared
  - All readers are allowed to read at the same time
  - Read access is noncritical, as long as there is no writer involved

ReaderWriterProblem Shared Read Access

- Processes that share a resource, e.g. a database, may perform read and write operations
- Write operations are critical
  - As it is a change to the shared data object, only one writer at a time may access the data object
  - All other processes, readers and writers, must be excluded from access
- Read operations are not critical
  - Many readers at the same time may read a shared data object
- Writers must have exclusive access to shared data
- Readers can access shared data simultaneously

ReaderWriterProblem Readers have Priority

- Reader processes share the following controlling data structures
  - Mutex Semaphores: mutex,W
  - rCount: counts readers
- Readers share mutex semaphore W with writers
  - Acts as amutual exclusion semaphore for readers and writers
  - Manipulated by the first or last reader when they enter / exit critical section
    - Blocking by first Reader, unblocking by last Reader, all other readers are not manipulating semaphore W
- Readers share semaphore mutex to allow exclusive manipulation of rCount

ReaderWriterProblem
Shared Read Access, ReaderPriority

Global Variables

```
init(mutex,1);  init(W,1);
rCount = 0; // counts the readers
```

Reader

```
while(TRUE)
{
    wait(mutex);
    rCount ++;
    if(rCount == 1)    wait(W);
    signal(mutex);
    read();
    wait(mutex);
    rCount --;
    if(rCount == 0)    signal(W);
    signal(mutex);
}
```

Writer

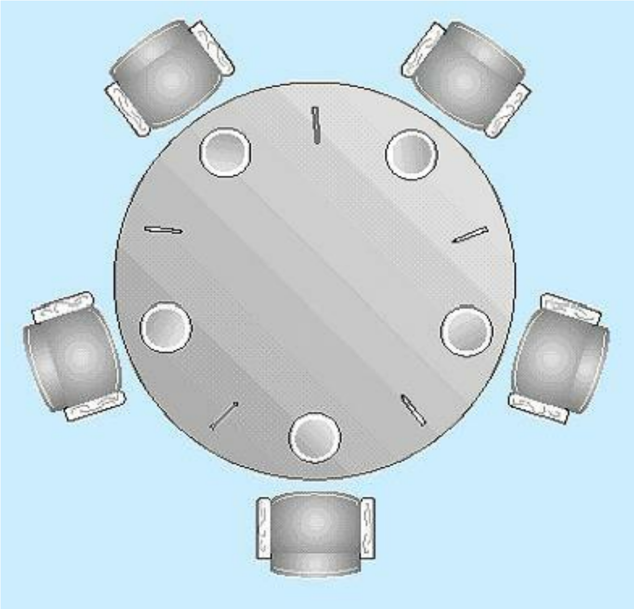
```
while(TRUE)
{
    wait(W);
    write();
    signal(W);
}
```

- Semaphore W checks whether the writer is in critical section
- Semaphore ‘mutex’ protects increment / decrement of rCount
- Readershave priority, writer has to wait until there is no reader

```
/* program readersandwriters */
int  readcount, writecount;
semaphore x = 1, y = 1, z = 1, wsem = 1, rsem = 1;
void reader()
{
    while (true)
    {
     semWait (z);
          semWait (rsem);
               semWait (x);
                    readcount++;
                    if (readcount == 1)
                         semWait (wsem);
               semSignal (x);
          semSignal (rsem);
     semSignal (z);
     READUNIT();
     semWait (x);
          readcount--;
          if (readcount == 0)
               semSignal (wsem);
     semSignal (x);
    }
}
void writer ()
{
    while (true)
    {
     semWait (y);
          writecount++;
          if (writecount == 1)
               semWait (rsem);
     semSignal (y);
     semWait (wsem);
     WRITEUNIT();
     semSignal (wsem);
     semWait (y);
          writecount--;
          if (writecount == 0)
               semSignal (rsem);
     semSignal (y);
    }
}
void main()
{
    readcount = writecount = 0;
    parbegin (reader, writer);
}
```

Dining Philosophers Problem

- Five philosophers sit arounda table for dinner
- There are only 5 forks on the table, two neighbouring philosophers share one fork



```
Philosopher:
while(TRUE)  {
   Think();
   Grab  first  fork;
   Grab  second  fork;
   Eat();
   Put  down  first  fork;
   Put  down  second  fork;
}
```

- Each philosopher needs two forks to eat
  - How many of them can eat at the same time?

Dining Philosophers Problem

- First attempt
  - –One process per philosopher
  - –One semaphore per fork
  - –Each semaphore initialised to 1
- Leads to deadlock,if all philosophers grab their left fork
  - –Will wait forever for a right fork to become available!

```
Semaphore  fork[5]  =  {1,1,1,1,1};
```

Process i

```
while(TRUE)
{
    think();
    wait(fork[i]);
    wait(fork[(i+1)  mod  5]);
    eat();
    signal(fork[(i+1)  mod  5]);
    signal(fork[i]);
}
```

Wait for left fork

Wait for right fork

Dining Philosophers Problem

- Problemoccurs if all philosophers want to eat at the same time
- We can introducetimeout:
  - –All philosophers pick up their left fork simultaneously
  - –They see that right fork is not available and put left fork down again
  - –They wait for a set time,pick up left fork again simultaneously
  - –Etc.
- Situation of starvation: they will never eat

Dining Philosophers Problem

- Practical Solution
  - Each philosopher waits arandom time, before trying again to acquire forks
  - E.g.: Ethernet protocol: if two computers want to send a packet at the same time – collision, both computers wait a random time to try again, hopefully no collision next time
  - Problem: although random time delay, we cannot guarantee that there is no collision next time

Dining Philosophers Problem, Solution

- Allow only 4 philosophers to pick up left fork at a time (only 4 are “seated” at the table)
- One philosopher has to wait
- One fork left free
- Of the 4 seated philosophers, at least one will have access to two forks
  - Philosophers with two forks can eat
  - All others have to wait

Dining Philosophers Problem

- 5 philosophers try to eat, allow only 4 philosophers at the table
- at least one philosopher hasaccess to two forks at a time,
- extra semaphore "seated" set to 4
- No deadlock, no starvation

```
Initialisation:
    init(seated,4);
    init(fork[1..5],1);
```

```
Philosopher  i:
void  philosopher(  int  i  )  {
    while(TRUE)  {
        think()  ;
        wait(seated);
        wait(fork[i]);
        wait(fork[i+1]  mod  5);
        eat();
        signal(fork[i+1]  mod  5);
        signal(fork[i]);
        signal(seated);
    }
}
```

Dining Philosophers Problem

- Other possible remedies
  - Goal: avoid deadlock / starvation, at least one philosopher should be able to eat
  - Allow a philosopherto pick up the two forks only if both are available at the same time
    - We need extra critical section for this
  - Use an asymmetric solution
    - Odd philosophers pickfirst theleft andthen the right fork
    - Even philosophers pick first the right and then the left fork