Synchronization

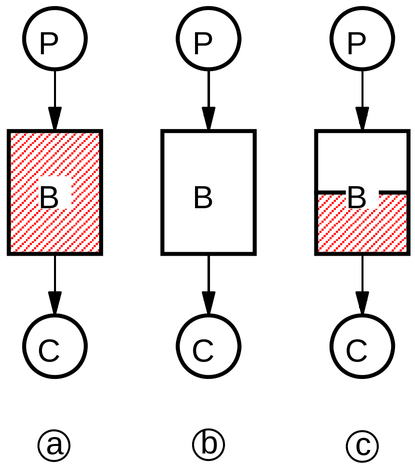
Synchronization protocols with semaphores

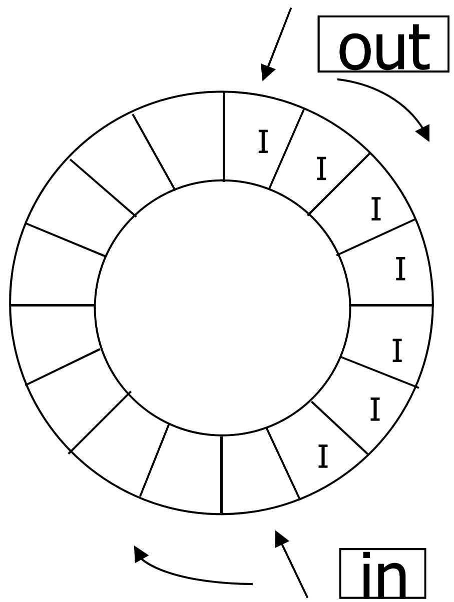
Stefano Quer and Pietro Laface
Dipartimento di Automatica e Informatica
Politecnico di Torino

Producer & Consumer with limited memory buffer

- Uses a circular buffer of size **MAX** for storing the produced elements to be consumed
- The circular buffer implements a FIFO (First-In First-Out) queue

FIFO
a) full
b) empty
c) partially full





Access functions

```
#define MAX ...
...
int buffer[MAX];
int in, out;
...
void init () {
  in = 0;
  out = 0;
}
```

```
void enqueue (int val) {
  queue[in] = val;
  in=(in+1)%MAX;
  return;
}
```

```
int dequeue (int *val) {
  *val=queue[out];
  out=(out+1)%MAX;
  return;
}
```

Concurrent access

number of full elements
number of empty elements

```
init (full, 0);
init (empty, MAX);
```

1 Producer
1 Consumer

```
Producer () {
  Message m;
  while (TRUE) {
    produces (m);
    wait (empty);
    enqueue (m);
    signal (full);
  }
}
```

```
Consumer () {
  Message m;

  while (TRUE) {

    wait (full);
    m = dequeue ();
    signal (empty);
    consumes (m);
  }
}
```

Considerations

- The solution is symmetric (dual)
- Producers and consumers **operate** on different indexes of the buffer, thus they can operate **in concurrency**
  - As long as the queue is not full or empty
  - Otherwise either a producer or a consumer is blocked
- The solution can be easily extended more than one producer and consumer process
  - Two producers or two consumers should instead act in mutual exclusion to protect their index (**in** or **out**, respectively)

Producers & Consumers

P Producers
C Consumers

```
init (full, 0);
init (empty, MAX);
init (MEp, 1);
init (MEc, 1);
```

For Mutual Exclusion among Producers (Consumers)

```
Producer () {
  Message m;
  while (TRUE) {
    produces m;
    wait (empty);
    wait (MEp);
    enqueue (m);
    signal (MEp);
    signal (full);
  }
}
```

```
Consumer () {
  Message m;
  while (TRUE) {
    wait (full);
    wait (MEc);
    m = dequeue ();
    signal (MEc);
    signal (empty);
    consumes m;
  }
}
```

Readers & Writers

- Sharing a database between two sets of concurrent threads
  - One class of such threads is called **Reader threads**
    - Readers are allowed access the database in concurrency
  - One class of such threads is called **Writer threads**
    - Writers must access the database is in Mutual Exclusion
      - with other Writers
      - with Readers

Readers & Writers

- There are two versions of the problem
  - Reader priority
  - Writer priority

Readers & Writers

- When a Writer is writing in the database, several Readers and Writers processes can be blocked outside their CSs waiting the end of the write operation 有人在写，其他人不能读写
- Readers precedence 读者优先
  - At the end of a writing operation, to give priority to the Readers means to favour the access of the waiting Readers rather than of the waiting Writers
- Writers precedence 写者优先
  - At the end of a writing operation, to give priority to the Writers means to favour the access of the waiting Writers rather than of the waiting Readers

Readers & Writers

- Common objectives
  - Respect the precedence protocol
  - Comply with the Bernstein conditions
  - Maximize concurrency

Readers priority

- Giving priority to the Readers means that
  - A Reader does not wait unless a Writer is writing
- Access protocol
  - While Readers are reading (they can access the database in concurrency), new Readers are allowed to read, and Writers are blocked
  - When the last Reader terminates, a waiting Writer can access the database

Readers priority

```
nR = 0;
init (meR, 1); init (meW,1);
init (w, 1);
```

Reader

```
wait (meR);
  nR++;
  if (nR==1)
    wait (w);
signal (meR);
...
read
...
wait (meR);
  nR--;
  if (nR==0)
    signal (w);
signal (meR);
```

Writer

```
wait(meW)
wait (w);
...
write
...
signal (w);
signal(meW)
```

Analysis

- The solution uses
  - A shared variable (**nR**) that counts the number of Readers inside their CS (reading)
  - A Mutual Exclusion semaphore the protects variable **nR** (**meR**)
  - A Mutual Exclusion semaphore (**w**) among Writers, or among Readers and Writers
  - A Mutual Exclusion semaphore (**meW**) among Writers, (only writers can queue on this semaphore)

Analysis

- Writers are subject to starvation, since they can wait forever
  - More complex solutions are possible that avoid starvation of the Writers

Writers priority

- Giving priority to the Writers means
  - A Writer has priority over all Readers
- Access protocol
  - A Writer trying to enter its CS blocks **new** Readers, but the Readers that are inside their CS are allowed to complete their reading task

Writers priority

```
nR = nW = 0;
init (w, 1); init (r, 1);
init (meR, 1); init (meW, 1);
```

Reader

```
wait (r);
  wait (meR);
    nR++;
    if (nR == 1)
      wait (w);
  signal (meR);
signal (r);
...
read
...
wait (meR);
  nR--;
  if (nR == 0)
    signal (w);
signal (meR);
```

Writer

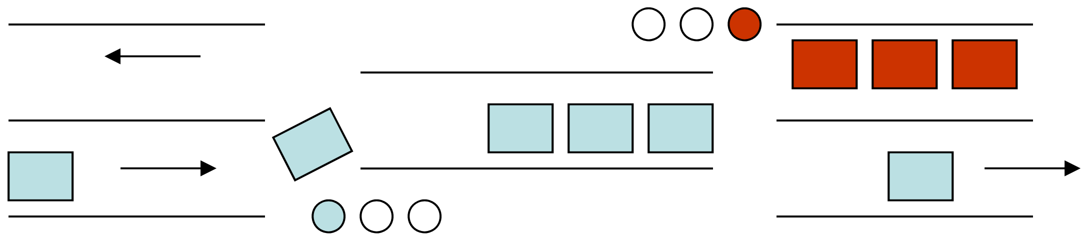
```
wait (meW);
  nW++;
  if (nW == 1)
    wait (r);
signal (meW);
wait (w);
  ...
  write
  ...
signal (w)
wait (meW);
  nW--;
  if (nW == 0)
    signal (r);
signal (meW);
```

Analysis

- The solution uses
  - Two shared variables (**nR** and **nW**) for counting the Readers inside their CS, and the Writers that need to write (one of them possibly writing)
  - Two Mutual Exclusion semaphores (**meR** and **meW**) for protecting the variables **nR** and **nW**
  - Two Mutual Exclusion semaphores (**r** and **w**) to enforce Readers and Writers to wait on different queues
- The Reades are subject to starvation, since they can wait forever
  - More complex solutions are possible that avoid starvation for the Readers

Single lane tunnel

- A tunnel has a single lane, and cars can proceed only in alternate directions
- Access protocol
  - Enable any number of cars (threads) to proceed in the same direction
  - If there is traffic in one direction, block traffic in the opposite direction



Single lane tunnel

- Similar to the Readers & Writers problem, but for two sets of Readers
- Data structure
  - Two shared count variables (**`n1`** and **`n2`**), one for each travel direction
  - Two semaphores (**`s1`** and **`s2`**), one for each travel direction
  - A global semaphore wait (**`busy`**)
- In its basic implementation can result in starvation of cars in one direction

Solution

```
n1 = n2 = 0;
init (s1, 1); init (s2, 1);
init (busy, 1);
```

left2right

```
wait (s1);
  n1++;
  if (n1 == 1)
    wait (busy);
signal (s1);
...
Run (left to right)
...
wait (s1);
  n1--;
  if (n1 == 0)
    signal (busy);
signal (s1);
```

right2left

```
wait (s2);
  n2++;
  if (n2 == 1)
    wait (busy);
signal (s2);
...
Run (left to right)
...
wait (s2);
  n2--;
  if (n2 == 0)
    signal (busy);
signal (s2);
```