System and Device Programming

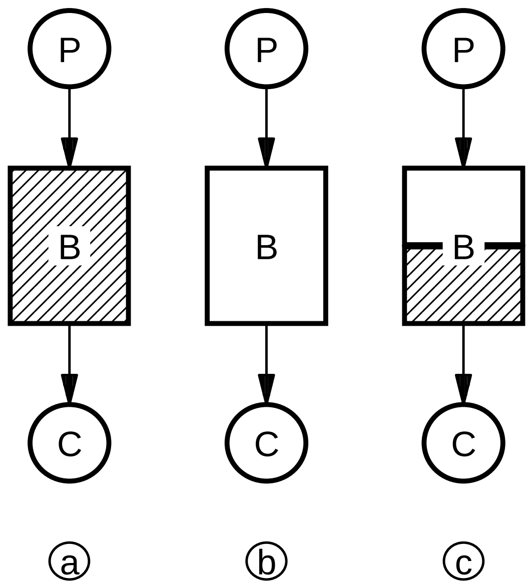
Classical Synchronization Problems

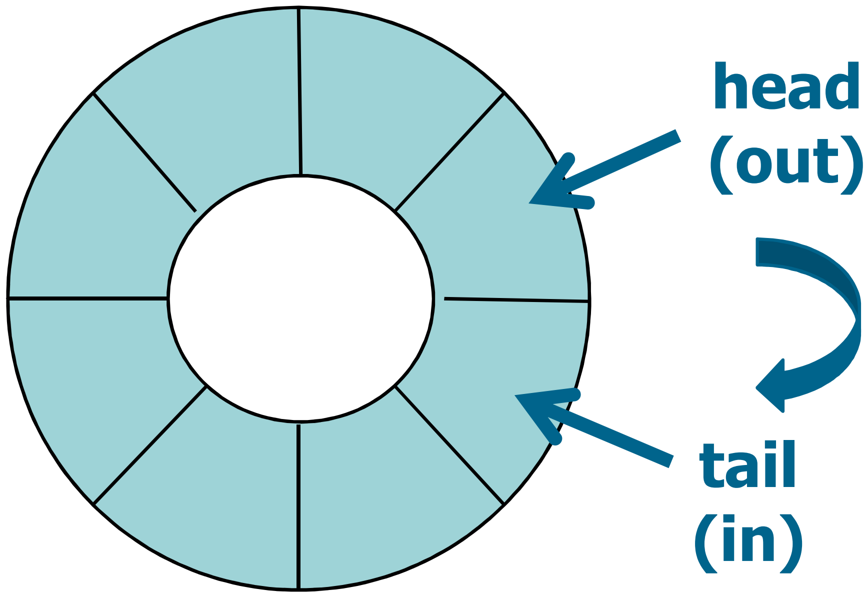
Stefano Quer
Dipartimento di Automatica e Informatica
Politecnico di Torino

Producer-Consumer

- Producer and consumer with limited memory
  - It uses a circular buffer of dimension **SIZE** to store the elements to be produced and consumed
  - The circular buffer implements a FIFO queue (First-In First-Out)

FIFO
full,
empty,
partially full





Solution

- In the sequential access enqueue and dequeue are concurrent
- In the parallel access we can have two cases
  - Only 1 producer and only 1 consumer
    - The operations enqueue and dequeue act on different extremes of the queue, however the n variable is shared
  - P producers and C consumers
    - In addition to the previous case, concurrent access operations to the same extreme of the queue are possible

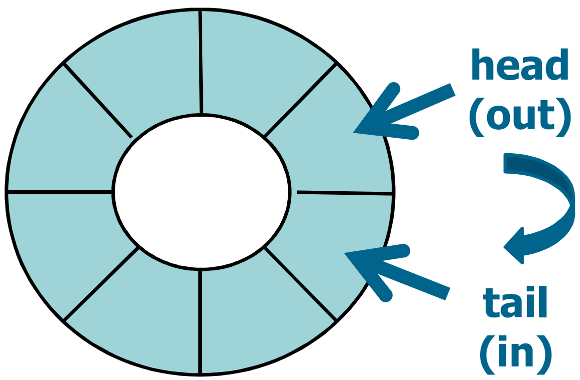
Solution

- With P producer and C consumer, we need
  - To count full and empty elements in the queue
    - A semaphore "full" counts the number of filled elements
    - A semaphore "empty" counts the number of empty elements
  - **Mutual exclusion** among producers and among consumers, as they act on opposite extremes of the buffer
    - Producers and consumers can work **concurrently**
    - As long as the queue is not completely full or completely empty

Solution

```
#define SIZE ...
...
int queue[SIZE];
int tail, head;
...
void init () {
  tail = 0;
  head = 0;
}
```

FIFO standard (non ADT) without the variable n




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

```
void dequeue (int *val) {
  *val=queue[head];
  head=(head+1)%SIZE;
  return;
}
```

Solution

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

full elements
empty elements

Mutual exclusion between P and C

```
Producer () {
  int val;
  while (TRUE) {
    produce (&val);
    wait (empty);
    wait (MEp);
    enqueue (val);
    signal (MEp);
    signal (full);
  }
}
```

```
Consumer () {
  int val;
  while (TRUE) {
    wait (full);
    wait (MEc);
    dequeue (&val);
    signal (MEc);
    signal (empty);
    consume (val);
  }
}
```

Readers & Writers

- Classical problem (1971) in which data is shared between two sets of concurrent processes
  - A set of **Readers**, which can access **concurrently** to the data
  - A set of **Writers**, which can access in **mutual exclusion**, both with other Writers and Readers processes, to the data
- There are two versions of the problem
  - Precedence to Readers
  - Precedence to Writers

Precedence to Readers

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

Reader

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

To enforce the precedence to R (the signal(w) unblocks an R)

Writer

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

Precedence to Writers

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

Reader

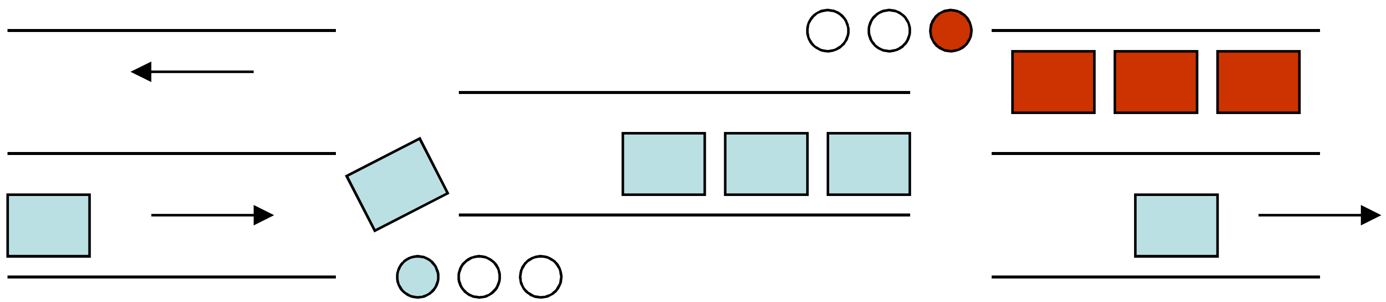
```
wait (r);
  wait (meR);
    nR++;
    if (nR == 1)
      wait (w);
  signal (meR);
signal (r);
...
reading
...
wait (meR);
  nR--;
  if (nR == 0)
    signal (w);
signal (meR);
```

Writer

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

The "Alternate direction tunnel"

- In an alternate direction tunnel
  - Allow any number of cars (processes) to proceed in the same direction
  - If there is traffic in one direction, block traffic in the opposite direction



The "Alternate direction tunnel"

- Extension to the Readers-Writers problem, with two sets of Readers
- Data structure
  - Two global counters (n1 and n2), one for each direction
  - Two semaphores (s1 and s2), one for each direction
  - A global semaphore for wait (busy)
- In its basic implementation, it can cause starvation of cars (in one direction with respect to the other)

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);
```

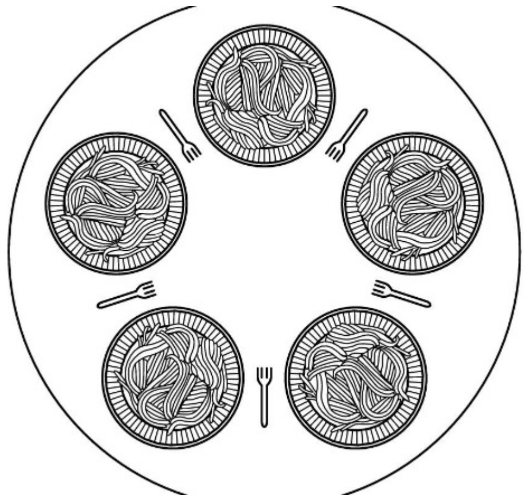
Dining (5) philosophers problem

- Model in which different resources are common to different concurrent processes
- Due to Dijkstra [1965]
- Definition of the problem
  - A table is set with
    - 5 rice dishes
    - 5 (Chinese) chopsticks each between two plates
  - Around the table sit 5 philosophers
  - Philosophers **think** or **eat**
    - To eat each philosopher needs two chopsticks
    - Chopsticks can be obtained one at a time

Solution

- Data structures
  - A state for each philosopher (THINKING, HUNGRY, EATING)
  - A semaphore for each philosopher (for access to food)
  - Another semaphore to manage the access in mutual exclusion to the philosopher state variable

```
while (TRUE) {
  Think ();
  takeForks (i);
  Eat ();
  putForks (i);
}
```



Solution

```
int state[N]
init (mutex, 1);
init (sem[0], 0); ...; init (sem[4], 0);
```

```
takeForks (int i) {
  wait (mutex);
  state[i] = HUNGRY;
  test (i);
  signal (mutex);
  wait (sem[i]);
}
```

```
putForks (int i) {
  wait (mutex);
  state[i] = THINKING;
  test (LEFT);
  test (RIGHT);
  signal (mutex);
}
```

```
test (int i) {
  if (state[i]==HUNGRY && state[LEFT]!=EATING &&
      state[RIGHT]!=EATING) {
    state[i] = EATING;
    signal (sem[i]);
  }
}
```