Concurrent Computing

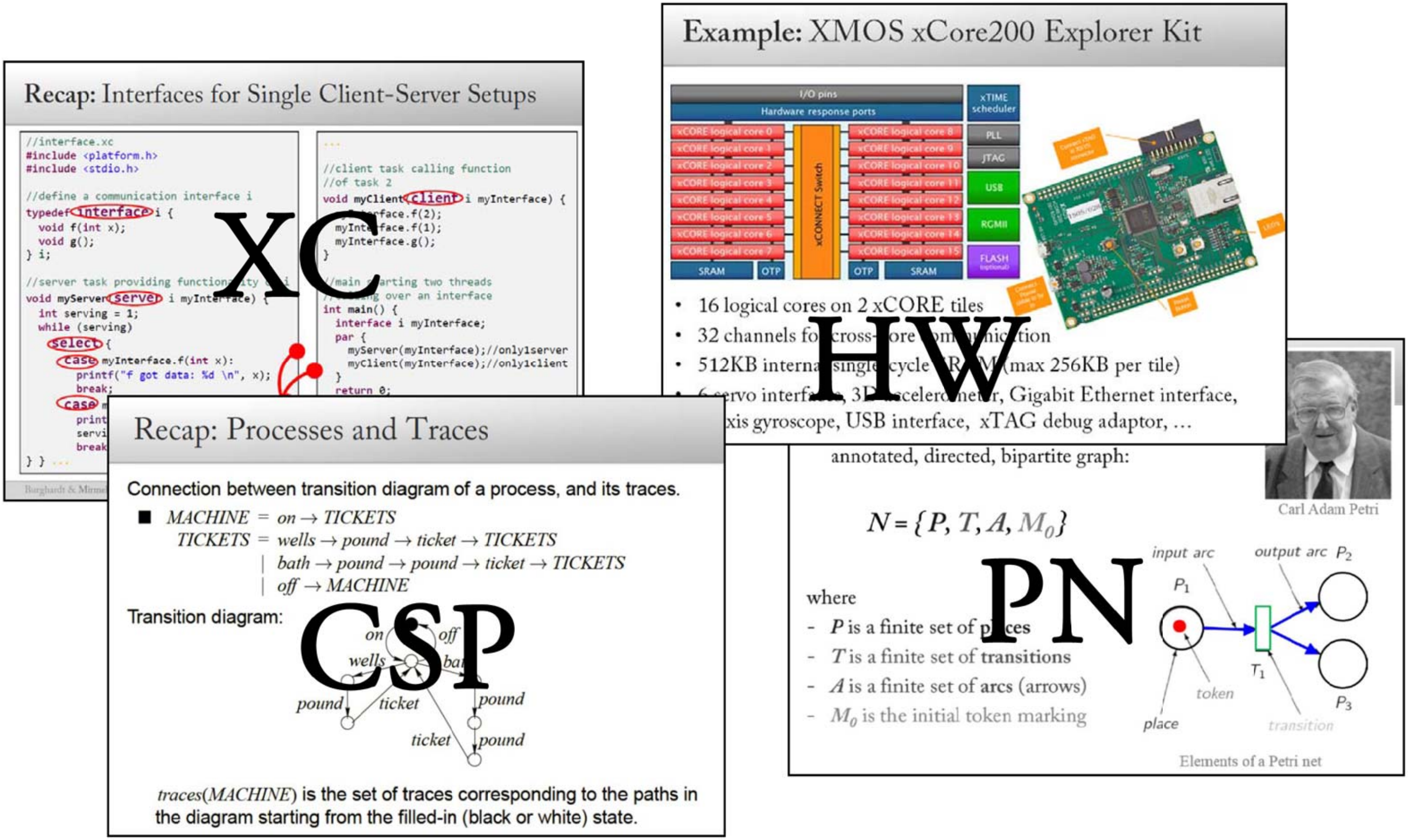
Lecturers: Prof. Majid Mirmehdi (majid@cs.bris.ac.uk)
Dr. Tilo Burghardt (tilo@cs.bris.ac.uk)
Dr. Daniel Page (page@cs.bris.ac.uk)

Web: http://www.cs.bris.ac.uk/Teaching/Resources/COMS20001

LECTURE 15

RACE CONDITIONS & CRITICAL SECTIONS

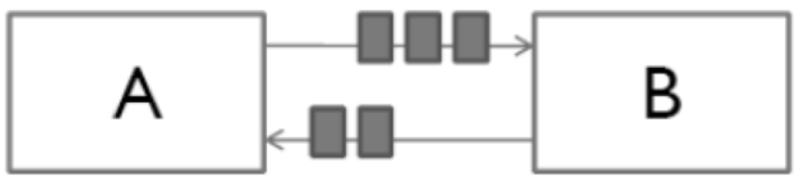
Concurrency Models so far...



Recap: Paradigms of Concurrent Programming

so far covered: … **Message Passing**

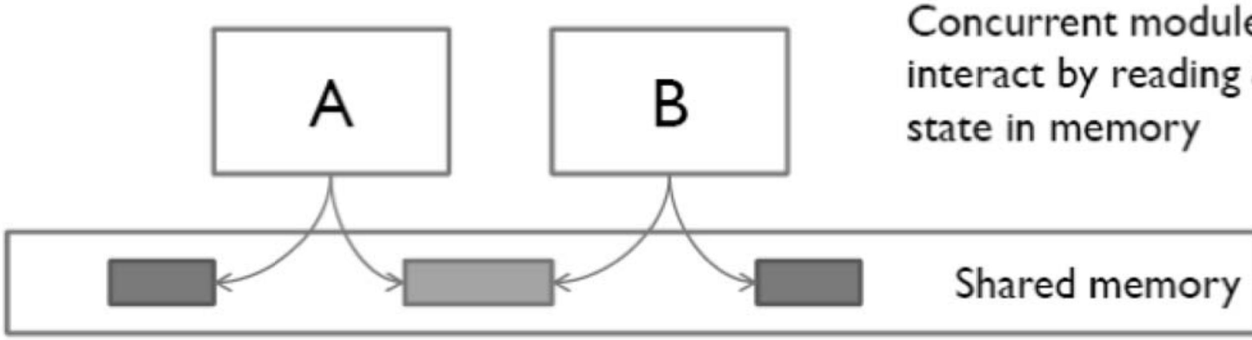
- Analogy: two computers in a network, communicating only by network connections



A and B interact by sending messages to each other through a communication channel

an alternative: … **Shared Memory**

- Analogy: two processors in a computer, sharing the same physical memory

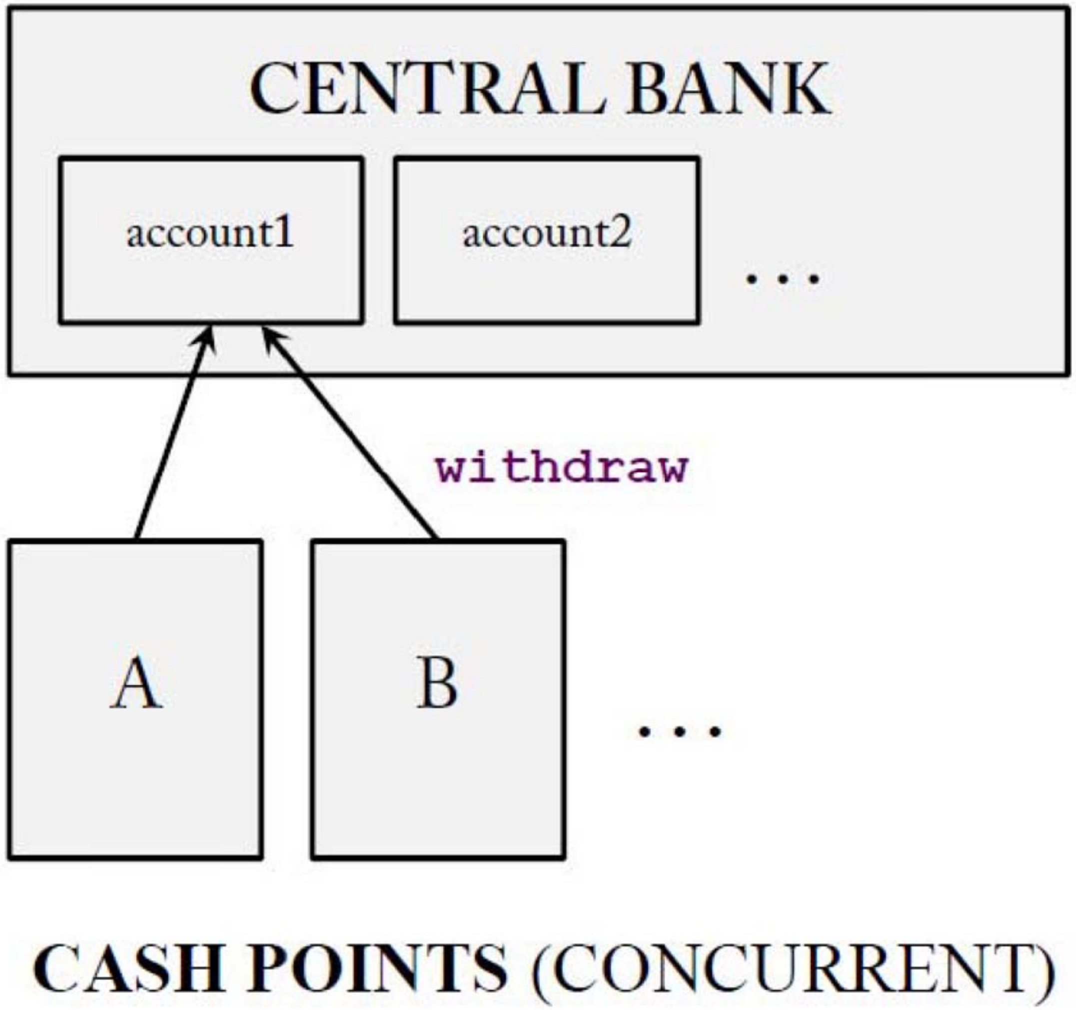


Concurrent modules A and B interact by reading & writing shared state in memory

Race Conditions: Bank Example (in C)

```c
...
// shared global memory
// at bank
int account1 = 99;
int account2 = ...
...
```

```c
...
// code fragment for
// withdrawing cash
...
int withdraw1( int amount ) {
  if (amount <= account1) {
    account1 -= amount;
    return 1;
  } else return 0;
}
...
```



Race Conditions: Concurrent Withdrawals 1

```
// shared global memory held at bank
int account1 = 99;
```

```
// started from CASH POINT A
 withdraw1(20);
```

```
// started from CASH POINT B
 withdraw1(90);
```

```
...
// THREAD AT CASH POINT A

  if (amount <= account1) {
    account1 -= amount;
    return 1;
  } else return 0;
...
```

```
...
// THREAD AT CASH POINT B
```

```
...
  if (amount <= account1) {
    account1 -= amount;
    return 1;
  } else return 0;
...
```

runtime

£20 withdrawn, £90 payout rejected, new balance is £79 – *ok*

Race Conditions: Concurrent Withdrawals 2

```
// shared global memory held at bank
int account1 = 99;
```

```
// started from CASH POINT A
 withdraw1(20);
```

```
// started from CASH POINT B
 withdraw1(90);
```

...
// THREAD AT CASH POINT A
 	
`if (amount <= account1) {`
	...
	// THREAD AT CASH POINT B

	`if (amount <= account1) {`
`account1 -= amount;`	

	`account1 -= amount;`
	`return 1;`
	`} else return 0;`

`return 1;` `} else return 0; ...`	...

runtime

£110 withdrawn

Race Conditions: Critical Section (CS)

Critical Sections are…

code fragments that interact with a shared resource and should not be accessed by more than one thread at any one time.

```
...
// shared global memory held at bank
int account1 = 99;
int account2 = ...
...
// code fragment for withdrawing cash
...
int withdraw1( int amount ) {
  if (amount <= account1) {
    account1 -= amount;
    return 1;
  } else return 0;
}
...
```

Demands on Critical Sections

(SECURITY)

no two threads can be within the critical section at the same time (mutual exclusion)

(LIVENESS / NO STARVATION)

any thread attempting to enter the critical section is able to enter it after some finite time

(FAIRNESS)

any thread has a fair chance of entering the critical section

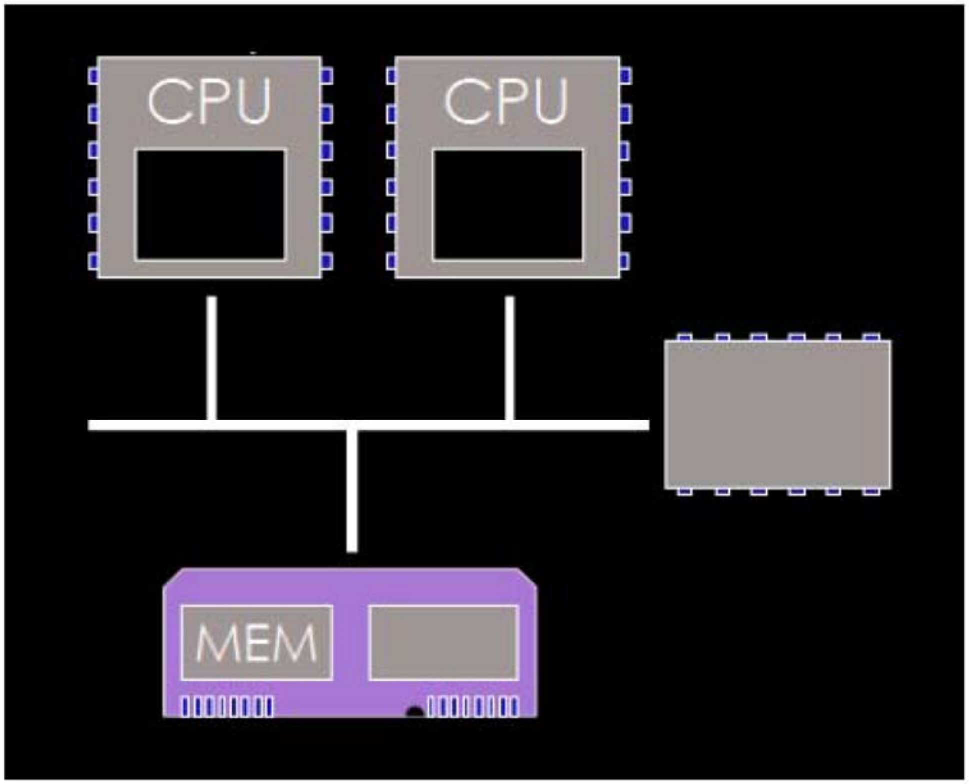
Insecure Implementation Attempt for Locks

```
...
// UNSUCCESSFUL LOCKING ATTEMPT
...
int Lock = 0;
//Lock == 0: free
//Lock == 1: locked

void lock(int *Lock) {
  while (*Lock != 0);
  //busy waiting
  //CRITICAL SECTION EXPOSED HERE
  *Lock = 1;
}

void unlock(int *lock) {
  *Lock = 0;
}
```

- **Why does this attempt fail in a concurrent system?**
- Which atomic operation would solve the problem?

NO MUTUAL EXCLUSION!

Hardware Support for Atomic Operations



- bus logic implements atomic operations:
  - read
  - write
  - read and write

```
test_and_set R, lock
//write to a memory location and
//return its old value, i.e.
//R = lock; lock = 1;

exchange R, lock
//X = lock; lock = R; R = X;
```

Implementation using HW Support

```
...
// SUCCESSFUL LOCKING ATTEMPT
// (to LOCK spin until lock
//  is found to be 0, write 1
//  to lock after any test)

LOCK:
  test_and_set R, lock
  //atomic: R = lock; lock = 1;
  cmp R, #0
  //compare result R to zero
  jnz LOCK
  //if (R != 0) goto lock
  ret
  //return

UNLOCK:
  mov lock, #0
  //set lock to zero
  ret
  //return
```

achieves mutual exclusion

Disadvantages:

- requires HW support
  (What happens if two threads are on same processor?)
- busy waiting
- heavy bus load
- thread starvation possible (no fairness)

Mutual Exclusion via Peterson's Algorithm

```
...
// THREAD 0
...

  // register interest
  interested[0] = true;

  // secure next available turn
  turn = 0;

  while (
    (interested[1]==true) &&
    (turn == 0)) {
    // busy wait
  }

  // CRITICAL SECTION

  interested[0] = false;
```

```
...
// THREAD 1
...

  // register interest
  interested[1] = true;

  // secure next available turn
  turn = 1;

  while (
    (interested[0]==true) &&
    (turn == 1)) {
    // busy wait
  }

  // CRITICAL SECTION

  interested[1] = false;
```

achieves mutual exclusion

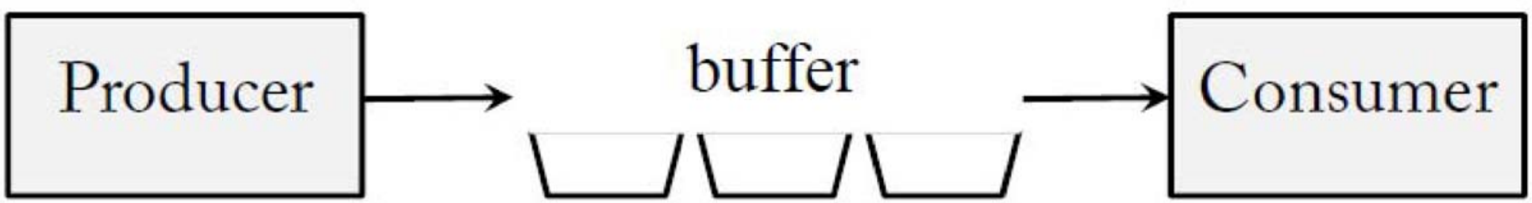
Busy Waiting vs. Suspension

- **'Busy Waiting' (also known as Spinning) …**
  a thread repeatedly checks to see if a condition is true

- Peterson's Algorithm uses 'Busy Waiting'

```
...
  while (
    (interested[1]==true) &&
    (turn == 0)) {
    // busy wait
  }
...
```

- **Consider a Producer-Consumer System with Limited Buffer and Permanent Operation**



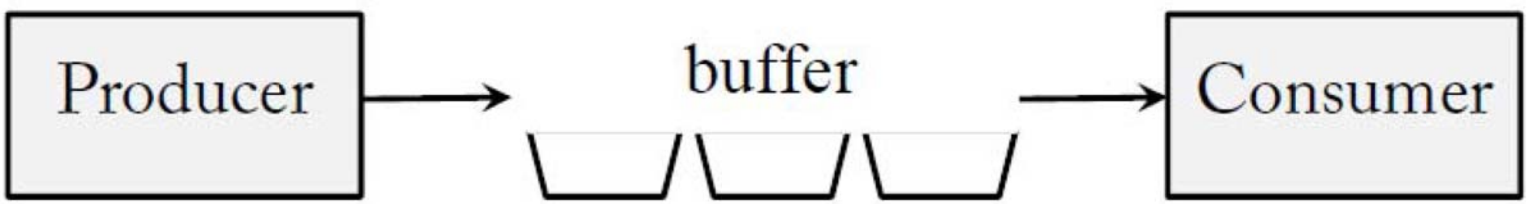
→ explore suspension instead of busy waiting

Consumer-Producer Solution via Semaphores

achieves mutual exclusion

```
...
// PRODUCER FRAGMENT
...

void produce() {
  while (true) {
    item = produce_item();
    noOfEmpty.P(); //wait&decr buffer
    critSec.P();   //enter critSec

    // send to buffer
    enter_item(item);

    critSec.V();   //leave critSec
    noOfFull.V();  //incr items
  }
}
```

```
...
// CONSUMER FRAGMENT
...

void consume() {
  while (true) {
    noOfFull.P(); //wait&decr item
    critSec.P();  //enter critSec

    // receive from buffer
    item = remove_item();

    critSec.V();   //leave critSec
    noOfEmpty.V(); //incr free buffer
    consume_item(item);
  }
}
```

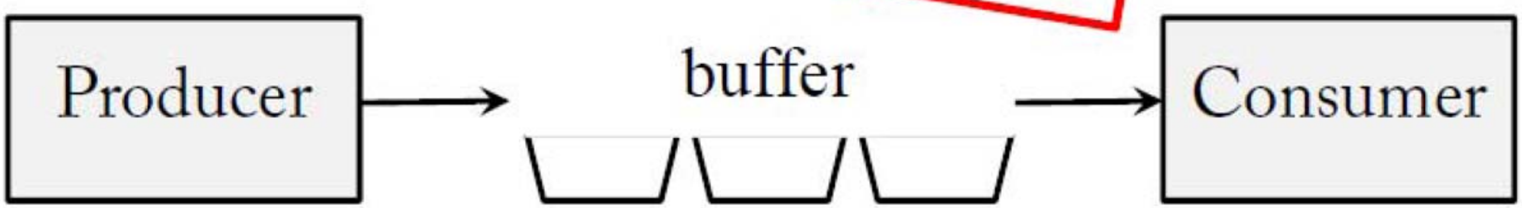


Deadlocking Consumer-Producer Implmentation

```
...
// PRODUCER FRAGMENT
...

void produce() {
  while (true) {
    item = produce_item();
    noOfEmpty.P();
    critSec.P();

    // send to buffer
    enter_item(item);

    critSec.V();
    noOfFull.V();
  }
}
```

```
...
// CONSUMER FRAGMENT
...

void consume() {
  while (true) {
    critSec.P();
    noOfFull.P();

    // receive from buffer
    item = remove_item();

    noOfEmpty.V();
    critSec.V();
    consume_item(item);
  }
}
```

DEADLOCK POSSIBLE!



Scheduler-supported Semaphore

```
class SemaphoreT {
  int       count;
  QueueType queue;
  public:
  SemaphoreT(int howMany);
  void P();
  void V();
}

SemaphoreT::SemaphoreT(
  int howMany) {
  count = howMany;
}

SemaphoreT::P() {
  if (count <= 0)
    sleep(queue);
  count--;
}

SemaphoreT::V() {
  count++;
  wakeup(queue);
}
```

- Need to implement all these methods as Critical Sections themselves !!!!

- **sleep(queue)**

  ```
  thread_current.state = sleeping;
  queue.enter(thread_current);
  schedule;
  ```

- **wakeup(queue)**

  ```
  thread_current.state = ready;
  switch_to(queue.take);
  ```