Demonstrating Failure of Locks (or any Concurrent System)

1. **Check Safety properties** These must *always* be true.
   Mutual exclusion Two processes must not interleave certain sequences of instructions.
   Absence of deadlock Deadlock is when a non-terminating system cannot respond to any signal.

2. **Check Liveness properties** These must *eventually* be true.
   Absence of starvation Information sent is delivered.
   Fairness That any contention must be resolved.

- **If you can demonstrate any cases in which these properties do not hold => system is not correct.**

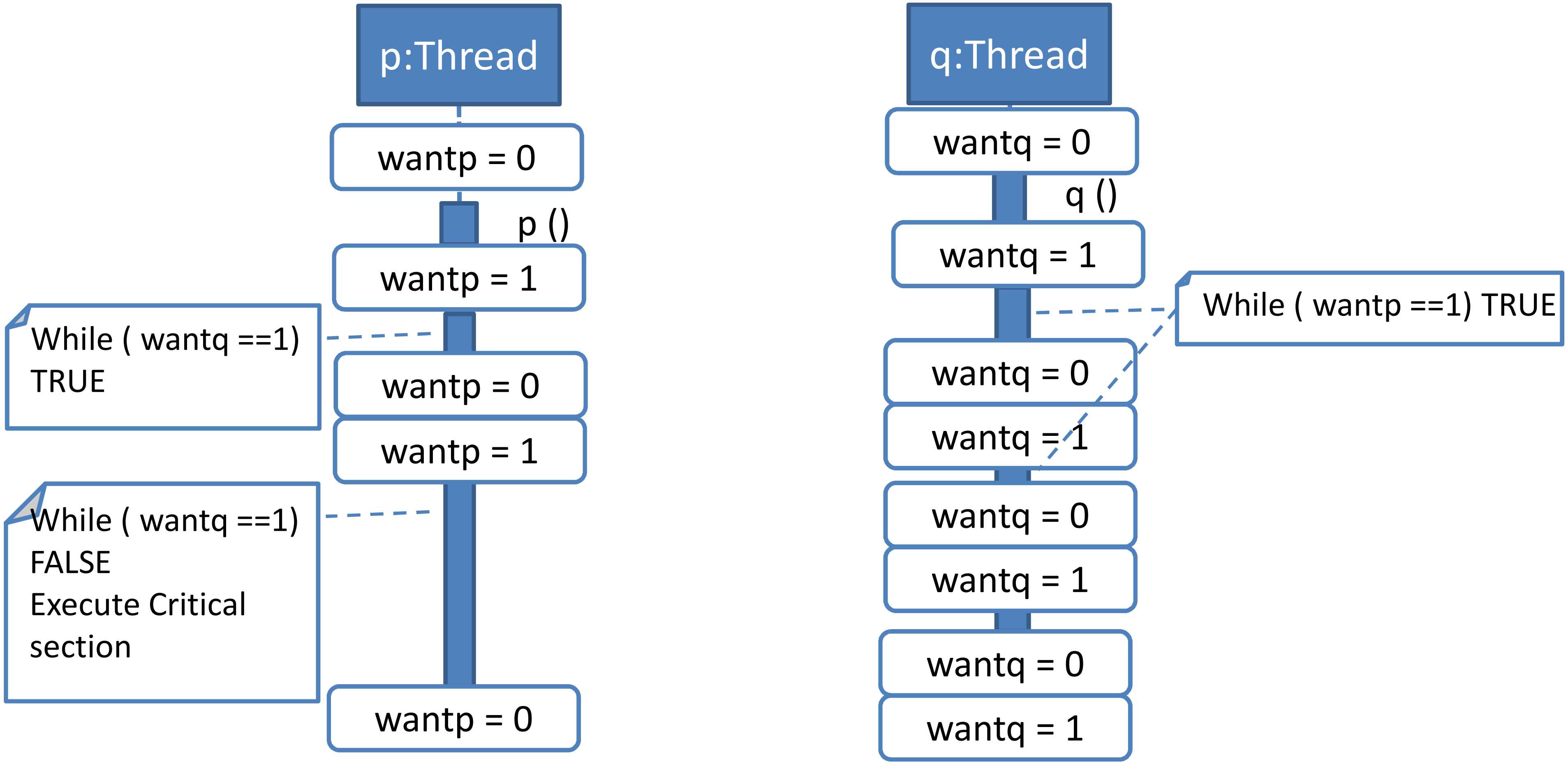
Example Software (not hardware) Solution to Mutual Exclusion Problem

```
/* Copyright © 2006 M. Ben-Ari. */

int wantp = 0;
int wantq = 0;

void p()
{
    while (1) {
      cout << "p non-critical section\n";
      wantp = 1;
      while (wantq == 1) {
          wantp = 0;
          wantp = 1;
      }
      cout << "p critical section\n";
      wantp = 0;
    }
}

void q()
{
    while (1) {
      cout << "q non-critical section\n";
      wantq = 1;
      while (wantp == 1) {
          wantq = 0;
          wantq = 1;
      }
      cout << "q critical section\n";
      wantq = 0;
    }
}
main() {
cobegin {
    p(); q();
    }

}
```

Exercise:

1. Turn to a partner
2. Can you find any cases where this will not work?

Failure by Starvation



Software Solution (cont'd)

Proof of Failure of Software mutext Attempt:

1. By Starvation

p sets **wantp** to 1.

p completes a full cycle:

Checks **wantq** Enters CS
Resets **wantp** Does non-CS
Sets **wantp** to 1

q sets **wantq** to 1
q checks **wantp**, sees **wantq**=1 & resets **wantq** to 0

q sets **wantq** to 1 and back

2. By Livelock

p sets **wantp** to 1.
p tests **wantq** , remains in its **do** loop
p resets **wantp** to 0 to relinquish attempt to enter CS
p sets **wantp** to 1

q sets **wantq** to 1
q tests **wantp** , remains in its **do** loop
q resets **wantq** to 0 to relinquish attempt to enter CS
q sets **wantq** to 1
etc

Example Software (not hardware) Solution to Mutual Exclusion Problem

- This proposal has two drawbacks:

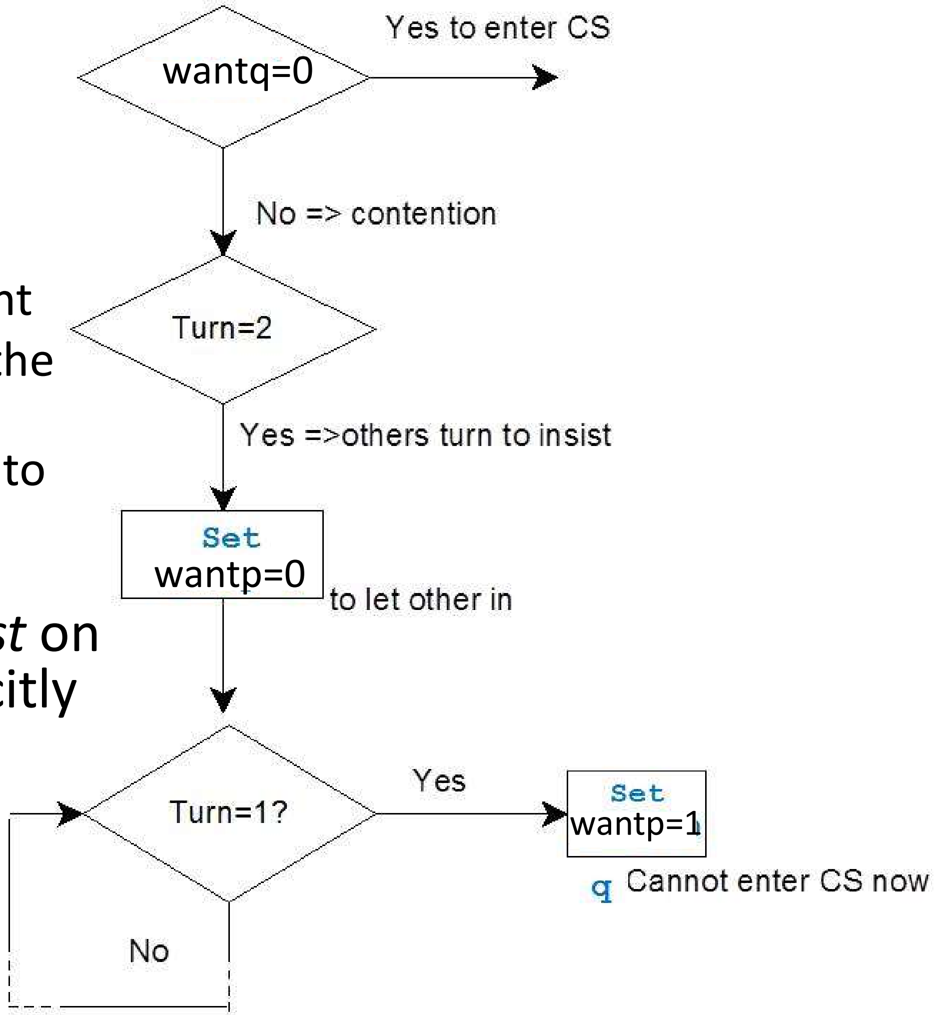
1. A process can be starved.

   Can find interleavings where a process can never enter its critical section.

2. The program can *livelock* (a form of deadlock).

Dekker's Algorithm

- A extension last proposal:
  - Add a variable to explicitly pass right to enter Critical Sections between the processes (fairness, turn-taking),
  - Last proposal had its own variable to prevent problems in absence of contention.
- In Dekker's algorithm right *to insist* on entering a Critical Section is explicitly passed between processes.
- Not safe on all hardware (optimisers can compromise this)



Dekker's Algorithm (cont'd)

```
/* Copyright © 2006 M. Ben-Ari.
*/

int wantp = 0;
int wantq = 0;
int turn = 1; // NEW!

void p()
{
    while (1) {
      cout << "p non-CS \n";
      wantp = 1;
      while (wantq == 1) {
       wantp = 0;
       while (!(turn == 1));
              wantp = 1; }
      cout << "p CS\n";
      turn = 2;
      wantp = 0;
    }
}
```

```
void q()
{
    while (1) {
      cout << "q non-CS\n";
      wantq = 1;
      while (wantp == 1) {
       wantq = 0;
       while (!(turn == 2));
              wantq = 1; }
      cout << "q CS\n";
      turn = 1;
      wantq = 0;
    }
}
main() {
/* As before */
}
```

Mutual Exclusion for n Processes: The Bakery Algorithm

- Dekker's Algorithm solves mutual exclusion problem for 2 processes.
- Many algorithms solve N process ME problem; all are complicated and relatively slow to other methods.
- *The Bakery Algorithm* is one where processes take a numbered ticket (whose value constantly increases) when it wants to enter its CS.
- The process with the lowest current ticket gets to enter its CS.
- This algorithm is not practical because:
  - ticket numbers will be unbounded if a process is always in its critical section, and
  - even in the absence of contention it is very inefficient as each process must query the other processes for their ticket number.

Mutual Exclusion for *N* Processes: The Bakery Algorithm (cont'd)

```
/* Copyright (C) 2006 M. Ben-Ari. */

const int NODES = 3;
      int num[NODES];
      int choose[NODES];

  int Max() {
  int Current = 0;
  int i;
    for (i=0; i <NODES; i++)
      if (num[i] > Current) Current = num[i];
    return Current;
  }

    void p(int i) {
    int j;
        while (1)         {
          cout << "proc " << i << " non-CS\n";
          choose[i] = 1;
          num[i]= 1 + Max();
          choose[i] = 0;
          for (j=0; j <NODES; j++)
            if (j != i)          {
              while (choose[j]);
              while (!
               ((num[j]==0)||(num[i]<num[j])||
               ((num[i]==num[j])&&(i < j))) );
              }
          cout << "process " << i << " CS\n";
          num[i]=0;
          }
    }
```

```
main() {
  int j;
  for (j=0; j <NODES; j++) number[j]=0;
  for (j=0; j <NODES; j++) choose[j]=0;
  cobegin {
    p(0); p(1); p(2); // 3 processes here
  }
}
```

SECTION 2.2: HIGHER LEVEL SUPPORT FOR MUTUAL EXCLUSION: SEMAPHORES & MONITORS

Example Scenario: Producer/Consumer

- Producer: creates a resource (data)
- Consumer: Uses a resource (data)
- E.g. `ps | grep “gcc” | wc`
- Don’t want producers and consumers to operate in lockstep (ie atomicity)
  - Each cmd must wait for the previous output
  - Implies lots of context switching (v expensive)
- Solution (Pattern): place a fixed size buffer between producers and consumers
  - Synchronise access to buffer
  - Producer waits of buffer full; consumer waits if buffer empty

Semaphores

- Semaphore = higher level synchronisation primitive
  - Invented by Dijkstra in 1965 as part of THE O/S project
- Implement with
  - A **counter** that is manipulated atomically via 2 operations **signal** and **wait**
  - `wait(semaphore)`: decrement, if counter is zero then block until semaphore is signalled (AKA down() or P())
  - `signal (semaphore)`: increment counter, wake up one waiter if any (AKA up() or V())
  - `sem_init(semaphore, counter)`: set initial counter value

Semaphore Pseudocode

```
struct semaphore {
        int value;
        queue L; // list of processes
}
wait (S) {
        if (s.value > 0)
                s.value = s.value -1;
        else {
                add this process to s.L;
                block;
        }
}
signal (S) {
        if (S.L != EMPTY){
                remove a process P from S.L;
                wakeup(P);
        } else
                s.value = s.value + 1;
}
```

`wait()/signal()` are critical sections! Hence, they must be Executed atomically With respect to each Other.

Blocking Semaphores

- Each semaphore has an associated queue of threads
  - When `wait()` is called by a thread
    - If semaphore is available => thread continues
    - If semaphore is unavailable, thread blocks, waits on queue
  - `signal()` opens the semaphore
    - If threads are waiting on a queue, one thread is unblocked
    - If no threads are on the queue, the signal is remembered for the next time wait() is called
  - NB Blocking threads are not spinning, they release the CPU to do other work

Semaphore Initialisation

- If semaphore initialised to 1
  - First call to wait goes through
    - Semaphore value goes from 1 to 0
  - Second call to wait() blocks
    - Semaphore value stays at zero, thread goes on queue
  - If first thread calls signal()
    - Semaphore value stays at 0
    - Wakes up second thread

$\Rightarrow$Acts like a mutex lock

$\Rightarrow$Can use semaphores to implement locks

This is called a **binary semaphore**

What happens if we initialise to 2?

```
struct semaphore {
          int value;
          queue L; // list of processes
}
wait (S) {
          if (s.value > 0)
                    s.value = s.value -1;
          else {
                    add this process to
s.L;
                    block;
          }
}
signal (S) {
          if (S.L != EMPTY){
                    remove a process P
from S.L;
                    wakeup(P);
          } else
                    s.value = s.value + 1;
}
```

Sem_init(sem, 2)

Consider multiple threads:

Thread1: wait(sem)

Thread2: wait(sem)

Thread2: wait(sem) –blocks

Observations:

Initial value of semaphore = number of threads that can be active at once

Uses of Semaphores

- Allocating a number of resources
  - Shared buffers: each time you want to access a buffer, call wait() => you are queued if there is no buffer available
  - Devices
- Counter is initialised to N = number of resources
- Called a **counting semaphore**
- Useful for conditional synchronisation
  - I.e. one thread is waiting for another thread to finish a piece of work before it continues

Semaphores for Mutual Exclusion

- With semaphores, guaranteeing mutual exclusion for N processes is trivial

```
semaphore mutex = 1;

void P (int i) {
while (1) {
       // Non Critical Section Bit
       wait(mutex) // grab the mutual exclusion semaphore
       // Do the Critical Section Bit
       signal(mutex) //grab the mutual exclusion semaphore
       }
   }

int main ( ) {
       cobegin {
              P(1);  P(2);
       }
}
```

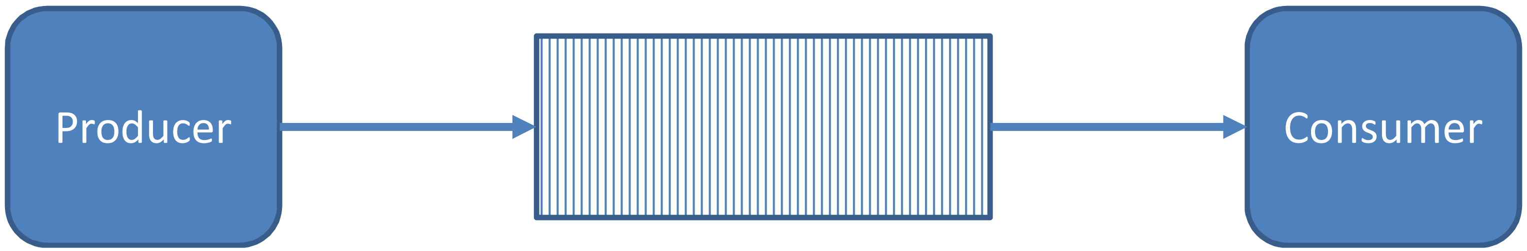
Example bounded buffer problem

- AKA producer/consumer problem
  - Buffer in memory
    - Finite size of N entries
  - A producer process inserts an entry into it
  - A consumer process removes an entry from it
- Processes are concurrent

  => Must use a synchronisation mechanism to control access to shared variables describing buffer state

Producer/consumer single buffer

- Simplest case
  - Single producer thread, single consumer thread
  - Single shared buffer between producer and consumer
- Requirements
  - Consumer must wait for producer to fill buffer
  - Producer must wait for consumer to empty buffer (if filled)



Exercise:

- How many shared data values do you have?

  3 : lock, emptyBuffer flag, sharedBuffer flag

- How many semaphores do we need?
- How should we initialise them?

Producer

```
while(1) {
        wait(&emptyBuffer);
        wait(&lock);

        fill(&buffer);

        signal(&lock);
        signal(&fullBuffer);
}
```

Consumer

```
while(1) {
        wait(&fullBuffer);
        wait(&lock);

        use(&buffer);

        signal(&lock);
        signal(&emptyBuffer);
}
```

Types of Semaphores

- Defined above is a general semaphore. A *binary semaphore* is a semaphore that can only take the values 0 and 1.
- Choice of which suspended process to wake gives the following definitions:
  - *Blocked-set semaphore* — Wakes any one suspended process
  - *Blocked-queue semaphore* — Suspended processes are kept in FIFO & woken in order of suspension
  - *Busy-wait semaphore* — semaphore value is tested in a busy-wait loop, with atomic test. Some loop cycles may be interleaved.

Semaphores can be hard to Use

- Complex patterns of resource usage
  - Cannot capture relationships with semaphores alone
  - Need extra state variables to record information (see Sleeping Barber later)
  - Often use semaphores such that
    - One is for mutex around state variables
    - One for each class of waiting

$\Rightarrow$Produce buggy code that is hard to write

- If one coder forgets to do **`V()/signal()`** after critical section, the whole system can deadlock

Monitors

- Need a higher level construct that groups the responsibility for correctness.
  - Supports controlled access to shared data
    - Synchronisation code added by compiler, enforced at runtime
- *Monitors* do this. They're an extension of the monolithic monitor used in OS to allocate memory etc.
  - *Encapsulate*
    - Shared data structures
    - Procedures that operate on shared data
    - Synchronization between concurrent processes that invoke these proceedures
  - Ensure only one process execute a monitor procedure at once (=>ME).
  - Guarantees only way to access the shared data is through proceedures.
- Native language support in Java