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

⇒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

Requirements for Deadlock

1. **Mutex**: at least one held resource must be non-shareable
2. **No pre-emption:** resources cannot be pre-empted (no way to break priority). Locks have this priority.
3. **Hold and wait**: there exists a process holding a resource and waiting for another resource

Unfortunately these conditions make code more efficient and hence want them

4. **Circular wait**: there exists a set of processes P_1, P_2,…,P_N such that P_1 is waiting for P_2, P_2 is waiting for P_3,… and P_N is waiting for P_1

All 4 conditions must hold for deadlock to occur

$\Rightarrow$ Need strategies to avoid circular wait (as it guarantees deadlock)

If 3 conditions hold then you can get starvation but not deadlock

Sample Deadlock

- Acquire locks in different orders
- Example:

<u>Thread 1</u>	<u>Thread2</u>
lock(x);	lock(y);
A=A+10;	B=B+10;
lock(y);	lock(x);
B=B+20;	A=A+20;
A=A+30;	B=B+30;
unlock(y);	unlock(x);
unlock (x)	unlock(y);

Sample Deadlock – Check for Deadlock

- Example:

Thread 1	Thread2
lock(x);	lock(y);
A=A+10;	B=B+10;
lock(y);	lock(x);
B=B+20;	A=A+20;
A=A+30;	B=B+30;
unlock(y);	unlock(x);
unlock (x)	unlock(y);

1. Do we have mutex?
2. Do we have hold and wait?
3. Do we have no pre-emption?
4. Do we have a circular wait?

Deadlocks without Locks

- Deadlocks can occur for any resource or any time a thread waits, e.g.
  - Messages: waiting to receive a message before sending a message ie hold and wait
  - Allocation: waiting to allocate resources before freeing another resource ie hold and wait

Testing for Real World Deadlock

- How do cars do it?
  - Never block an intersection
  - Must backup if you find yourself doing so (a form of pre-emption)
- Why does this work?
  - Breaks a wait and hold
  - Shows that refusing to hold a resource while waiting for something else is a key element of avoiding deadlock

Dealing With Deadlocks: Ignore

- Strategy 1: Ignore the fact that deadlocks may occur
  - Write code, put nothing special in
  - Sometimes you have to re-boot the system
  - May work for some unimportant or simple applications where deadlock does not occur often
- Quite a common approach!

Dealing with Deadlock: Reactive

- Periodically check for evidence of deadlock
  - E.g. add timeouts to acquiring a lock, if you timeout then it implies deadlock has occurred and you must do something
  - Recovery actions:
    - Blue screen and re-boot computer
    - Pick a thread to terminate eg a low priority one
      - Only works with some types of applications
      - May corrupt data so thread needs to do cleanup when terminated
      - Then thread often re-tries from start
      - This **breaks the pre-emption condition**
        - » Databases often do this as state is generally well known

Dealing with Deadlock: Proactive

- Prevent 1 of the 4 necessary conditions for deadlock
- No single approach is appropriate (or possible) for all circumstances
  - Need techniques for each of the four conditions

Solution 1: No Mutual Exclusion

- Make resources shareable
- Example: read-only files
  - No need for locks
- Example: per-thread variables
  - Counters per thread instead of global counter

Fixing our Sample Deadlock Code

Thread 1	Thread2
lock(x);	lock(y);
A=A+10;	B=B+10;
lock(y);	lock(x);
B=B+20;	A=A+20;
A=A+30;	B=B+30;
unlock(y);	unlock(x);
unlock (x)	unlock(y);

Solution 1: Avoid Hold and Wait

- Only request a resource when you have none;
  Release a resource before requesting another

Thread 1	Thread2
lock(x);	lock(y);
A=A+10;	B=B+10;
unlock(x);	**unlock(y);**
lock(y);	lock(x);
B=B+20;	A=A+20;
unlock(y);	**unlock(x);**
lock(x);	**lock(y);**
A=A+30;	B=B+30;
unlock (x);	unlock(y);

Original code:

Thread 1	Thread2
lock(x);	lock(y);
A=A+10;	B=B+10;
lock(y);	lock(x);
B=B+20;	A=A+20;
A=A+30;	B=B+30;
unlock(y);	unlock(x);
unlock (x)	unlock(y);

$\Rightarrow$ Never hold x when want y. Works in many cases.

But if there is a requirement to maintain a relationship between x and y then you cannot do this.

Solution 2: Avoid Hold and Wait using Atomicity

- Only acquire all resources at once.;
  Eg use a single lock to protect all data

Thread 1	Thread2
lock(z);	**lock(z);**
A=A+10;	B=B+10;
B=B+20;	A=A+20;
A=A+30;	B=B+30;
unlock (z);	**unlock(z);**

Original code:

Thread 1	Thread2
lock(x);	lock(y);
A=A+10;	B=B+10;
lock(y);	lock(x);
B=B+20;	A=A+20;
A=A+30;	B=B+30;
unlock(y);	unlock(x);
unlock (x)	unlock(y);

Problem: low concurrency.

Now all threads accessing A or B cannot run at the same time (even if they don't access both).

Having fewer locks is called lock coarsening

Prevention: Adding Pre-emption

- Locks cannot be pre-empted but other pre-emptive methods are possible
- Strategy: preempt resources
- Example:
  - If thread A is waiting for a resource held by thread B, then take the resource from B and give it to A
- Problems:
  - Only works for some resources eg CPU and memory ie use virtual memory
  - Not possible if a resource cannot be saved and restored, otherwise taking away a lock causes issues
  - Also overhead cost of doing pre-empt and restore

Prevention: Eliminate Circular Waits

- Strategy: Impose an ordering on resources and threads must acquire the highest ranked resource first

<u>Thread 1</u>	<u>Thread2</u>
lock(x);	**lock(x);**
lock(y);	lock(y);
A=A+10;	B=B+10;
B=B+20;	A=A+20;
A = A+B;	A=A+B;
unlock(y);	**unlock(x);**
A=A+30;	B=B+30;
unlock (x);	unlock(y);

Original code:

<u>Thread 1</u>	<u>Thread2</u>
lock(x);	lock(y);
A=A+10;	B=B+10;
lock(y);	lock(x);
B=B+20;	A=A+20;
A=A+30;	B=B+30;
unlock(y);	unlock(x);
unlock (x)	unlock(y);

=> Locks are always acquired in the same order, have eliminated the circular dependency

Preventing Circular Wait: Lock Hierarchy

- Strategy
  - Define an ordering of all locks in your program
  - Always acquire locks in that order
- Problem: Sometimes you do not know the order that the events will be used
  - Recall our code for transferring money from 1 account to another

```
transfer(acc1, acc2, amount){
        acquire(acc1.a_lock);
        acquire(acc2.a_lock);
        acc1.balance -= amount;
        acc2.balance += amount;
        release(acc1.a_lock);
        release(acc2.a_lock);
}
```

How do we know the global order?

⇒ Need extra code to find this out and then acquire them In the right order

⇒ It could get worse

Problem:

T1: transfer(rob,martin)

T2: transfer(martin,rob)

T1: acquire(rob.a_lock)

T2: acquire(martin.a_lock)

T1: acquire(martin.a_lock)

T2: acquire(rob.a_lock)

Lock Hierarchy Problems

- General problem: dynamically chosen locks
  - Hard to enforce order if don’t know the lock you will acquire

```
transX(acc1, acc2, acc3, amount){
   acquire(acc1.a_lock);
   acquire(acc2.a_lock);
   if(acc1.balance < acc2.
                     balance)
          acquire(acc3.a_lock)
          acc1.balance -= amount;
          acc3.balance += amount;
          release(acc3.a_lock);
          release(acc2.a_lock);
   } else {
          acc1.balance-=amount;
          acc2.balance+=amount;
   }
   release(acc1.a_lock);
   release(acc2.a_lock);
}
```

SECTION 2.3: Ye Classicale Problemes of Synchronization

1. The Producer-Consumer Problem

This type of problem has two types of processes:

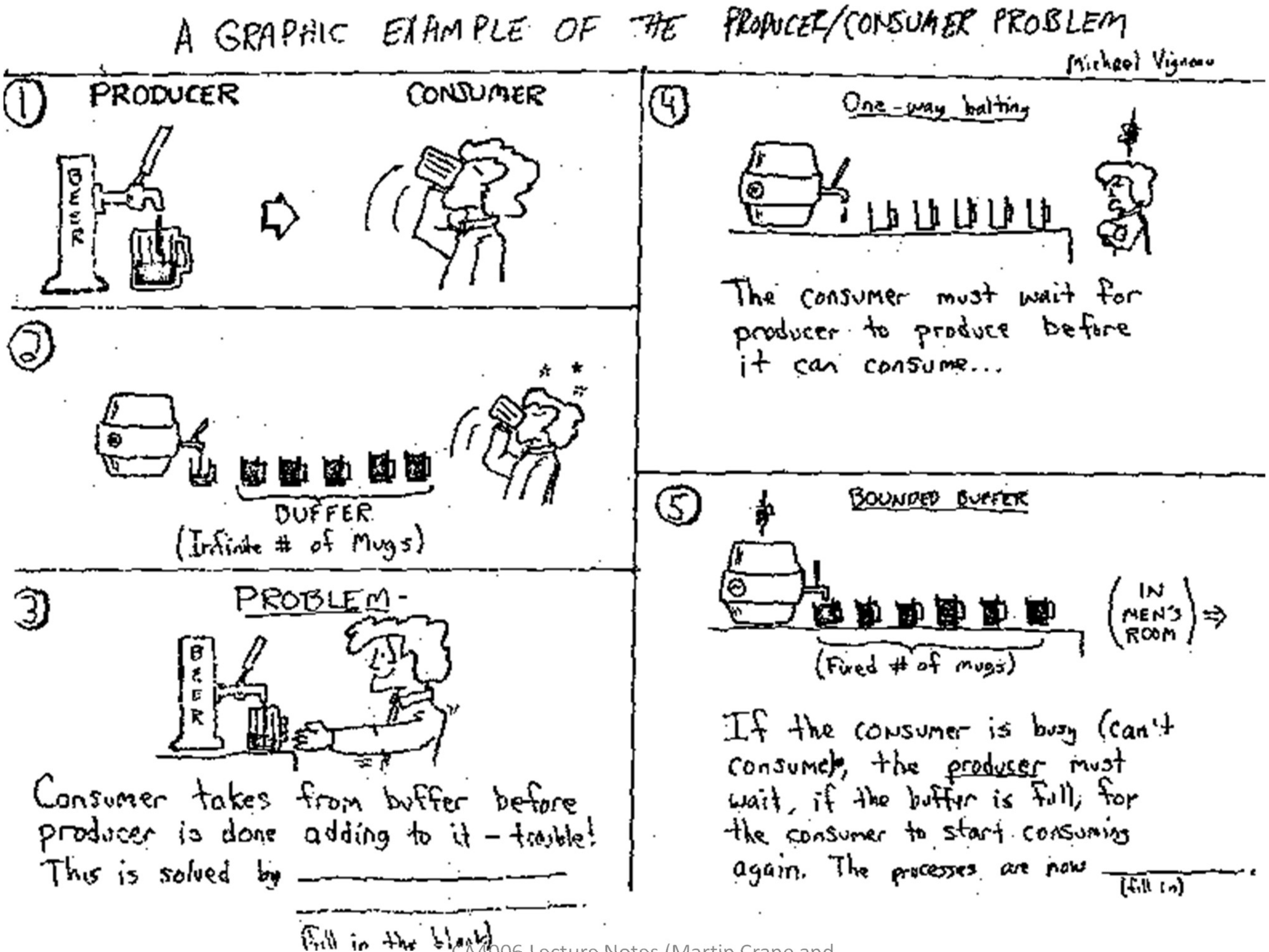
Producers	processes that, from some inner activity, produce data to send to consumers.
Consumers	processes that on receipt of a data element consume data in some internal computation.

- Can join processes synchronously, so data is only sent when producer can send it & consumer can receive.
- Better to connect them by a buffer (ie a *queue*)
- For an infinite buffer, the following invariants hold for the buffer:

$$\#elements \geq 0$$

$$\#elements = 0 + in\_pointer - out\_pointer$$

- These are the same as the semaphore invariants with a semaphore called *elements* and an initial value 0.



A GRAPHIC EXAMPLE OF THE PRODUCER/CONSUMER PROBLEM
Michael Vigneau
1
PRODUCER
CONSUMER
BEER
2
BUFFER
(Infinite # of Mugs)
3
PROBLEM -
BEER
Consumer takes from buffer before producer is done adding to it – trouble!
This is solved by ______________
(fill in the blank)
4
The consumer must wait for producer to produce before it can consume...
5
BOUNDED BUFFER
(Fixed # of mugs)
(IN MEN'S ROOM)
If the consumer is busy (can't consume), the producer must wait, if the buffer is full, for the consumer to start consuming again. The processes are now ______________.
(fill in)

The Producer-Consumer Problem (cont'd)

```
int in_pointer = 0, out_pointer = 0
semaphore elements = 0; // items produced
semaphore spaces = N;   //spaces left
void producer( int i) {
   while (1) {
      item = produceItem();
      wait(spaces);
      putItemIntoBuffer(item);
      in_pointer:=(in_pointer+1) mod N;
      signal(elements);
   }
}
void consumer( int i) {
    while (1) {
        wait(elements);
        item = removeItemFromBuffer();
        out_pointer:=(out_pointer+1)mod N
        signal(spaces);
        consumeItem(item);
     }
}
int main ( ) {
          cobegin {
producer(1); producer (2); consumer (1);
consumer (2); consumer (3); }
}
```

//Spaces = 4

T1 p1: produce item, putItemIntoBuffer //spaces= 3 , elements = 1

T2 p2: produce item, putItemIntoBuffer //spaces= 2 , elements = 2

T3 p3: produce item, putItemIntoBuffer //spaces= 1 , elements = 3

T4 c1: removetemFromBuffer //spaces= 2 , elements = 2

T1 p1: produce item, putItemIntoBuffer //spaces= 1 , elements = 3

T2 p2: produce item, putItemIntoBuffer //spaces= 0 , elements = 4

T2 p2: produce item, wait //BLOCKED spaces= 0 , elements = 4

The Producer-Consumer Problem (cont'd)

```
/* Copyright (C) Wikipedia */
/* Assumes various procedures e.g. wait */
int in_pointer = 0, out_pointer = 0
semaphore elements = 0; // items produced
semaphore spaces = N;   //spaces left

void producer( int i) {
    while (1) {
        item = produceItem();
        wait(spaces);
        putItemIntoBuffer(item);
        in_pointer:=(in_pointer+1) mod N;
         signal(elements);
     }
}
```

```
void consumer( int i) {
    while (1) {
        wait(elements);
        item = removeItemFromBuffer();
        out_pointer:=(out_pointer+1)mod N
        signal(spaces);
        consumeItem(item);
     }
}
int main ( ) {
        cobegin {
producer(1); producer (2); consumer (1);
consumer (2); consumer (3); }
}
```

- Shows the case of a real, bounded circular buffer to count empty places/spaces in the buffer.
- As an exercise prove the following:
  - (i) No deadlock, (ii) No starvation &
  - (iii) No data removal/appending from an empty/full buffer respectively

Check for Deadlock

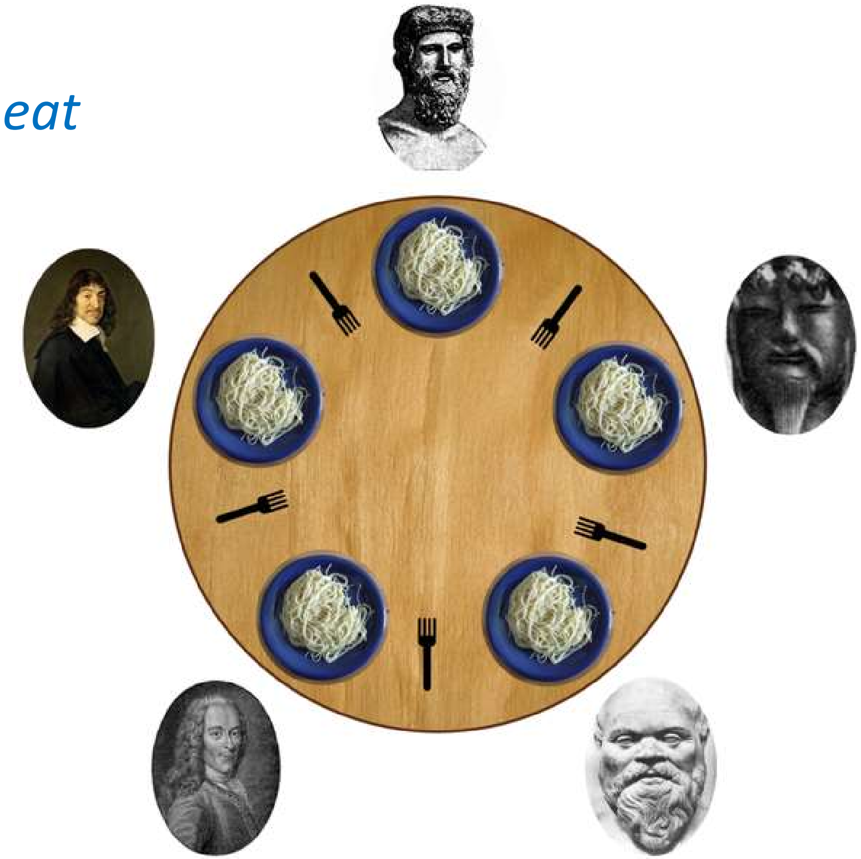
Do we have?

1. Mutex
2. No pre-emption:
3. Hold and wait:
4. Circular wait:

```
int in_pointer = 0, out_pointer = 0
semaphore elements = 0; // items produced
semaphore spaces = N;   //spaces left
void producer( int i) {
   while (1) {
      item = produceItem();
      wait(spaces);
      putItemIntoBuffer(item);
      in_pointer:=(in_pointer+1) mod N;
      signal(elements);
   }
}
void consumer( int i) {
    while (1) {
        wait(elements);
        item = removeItemFromBuffer();
        out_pointer:=(out_pointer+1)mod N
        signal(spaces);
        consumeItem(item);
     }
}
int main ( ) {
          cobegin {
producer(1); producer (2); consumer (1);
consumer (2); consumer (3); }
}
```

2. The Dining Philosophers Problem

- DCU hires 5 philosophers for hard problems
- Philosophers only have 2 states: *think* & *eat*
- Dining table has five plates & five forks*.
- Philosophers need 2 forks to eat
- Each plate is endlessly refilled.
- Philosopher may only pick up the forks immediately to his left or right.



*or five bowls and five chopsticks

Dining Philosophers (cont'd)

- For this system to operate correctly it is required that:
  1. A philosopher eats only if he has two forks.
  2. No two philosophers can hold the same fork simultaneously.
- Challenge: Develop an algorithm where no philosopher starves.

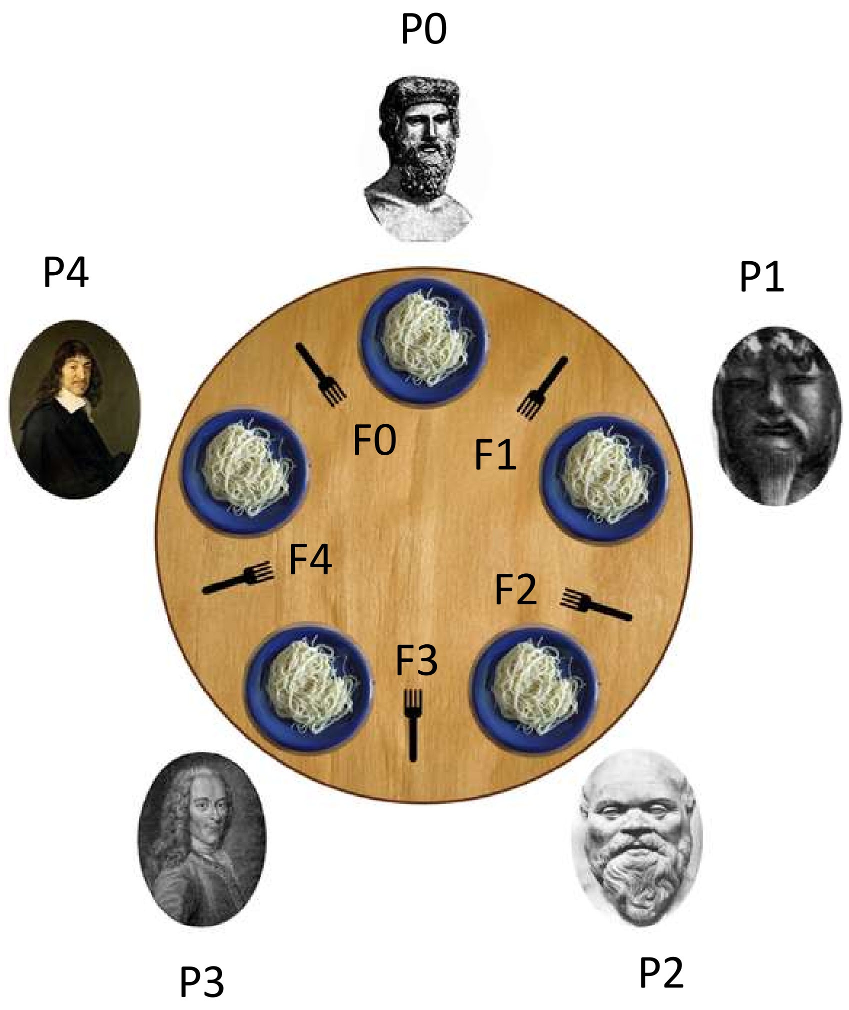
- Question: What could go wrong?

- This problem is a generalisation of multiple processes accessing a set of shared resources;
  – e.g. a network of computers accessing a bank of printers.

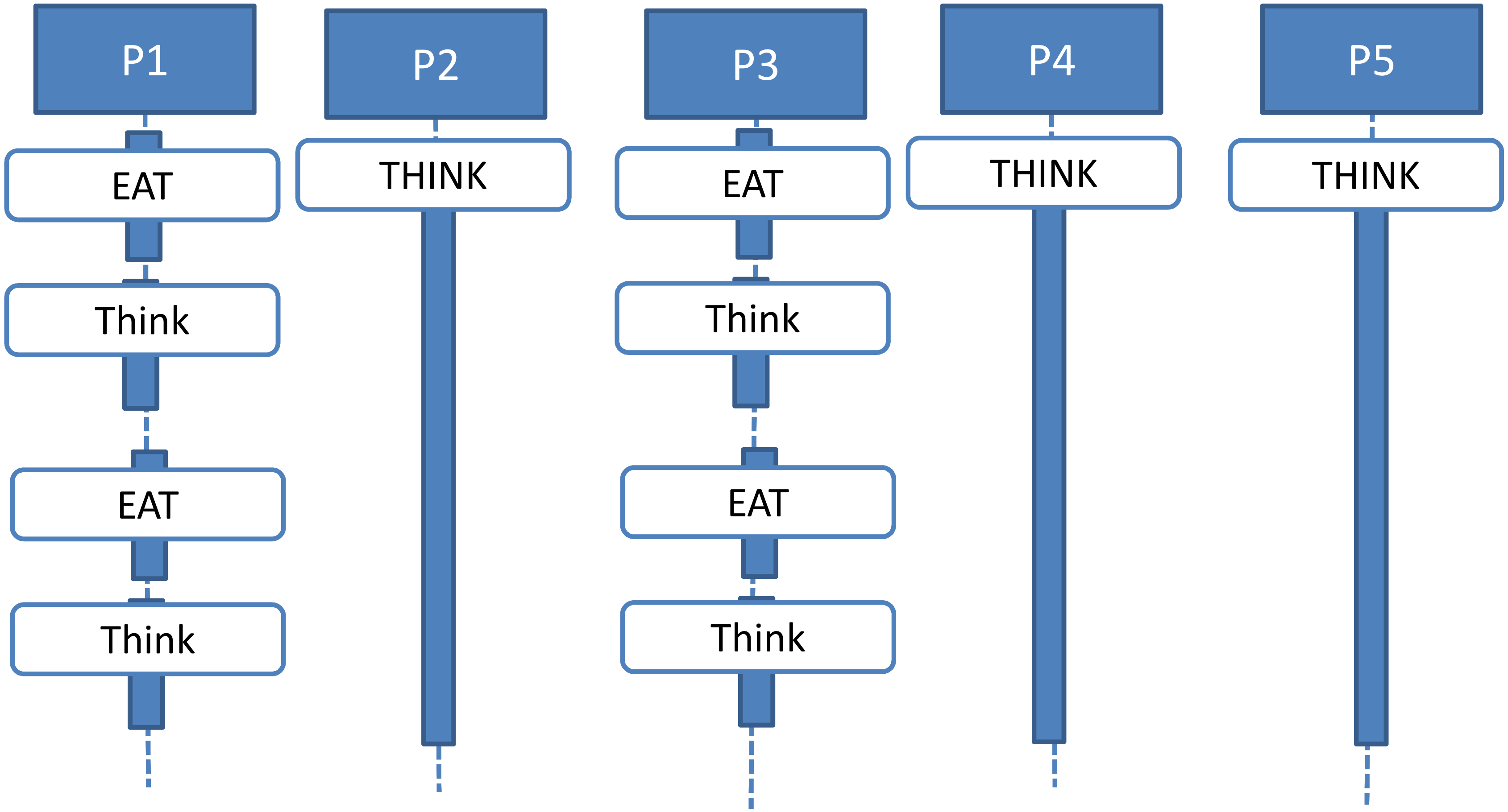
First Attempt

```
void philosopher(int id){
    while(TRUE){
        think(); //for some time
        take_fork(right);
        take_fork(left);
        eat();
        put_fork(left);
        put_fork(right);
    }
}
```

Q: Any issues?



Possible Scenario



Dining Philosophers: Solution #2

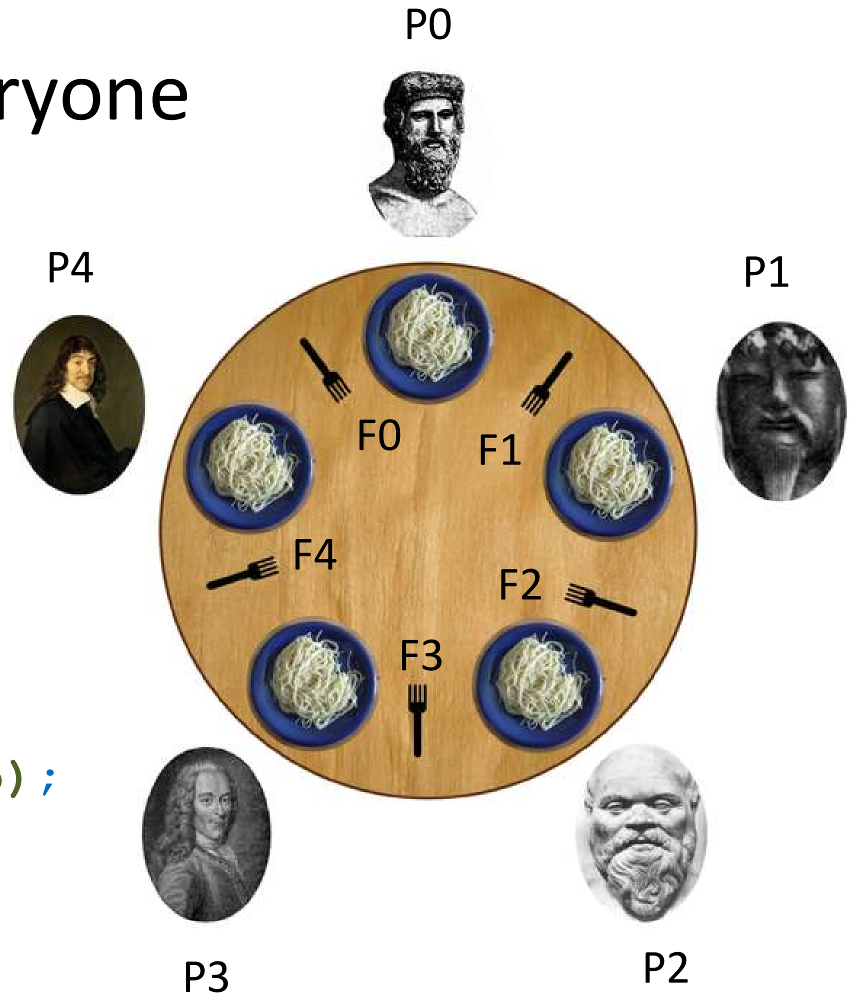
- Model each fork as a semaphore.
- Then each philosopher must wait() on both the left and right forks before eating.

```
semaphore fork [5] := ((5) 1)
/* pseudo-code */
/* fork is array of semaphores all initialised to have value 1 */
process philosopher (i := 0 to 4) {
     while (1) {
            think ( );
            wait(fork (i));                        // grab fork[i]
            wait(fork ((i+1) mod 5); // mod allows for circular
                               //table where P4 grabs fork 4 and fork 0
            eat ( );
            signal(fork (i));                      // release fork[i]
            signal(fork ((i+1) mod 5);             // release rh fork
     }
}
```

Look at this in Action

- What happens when everyone picks up left fork?

```
semaphore fork [5] := ((5) 1)
process philosopher (i := 0 to 4) {
    while (1) {
            think ( );
            wait(fork (i));
            wait(fork ((i+1) mod 5);
            eat ( );
            signal(fork (i));
            signal(fork ((i+1) mod 5);
    }
}
```



Dining Philosphers: Solution #2

- Called a *symmetric solution* as each task is identical.
- Symmetric solutions have advantages, e.g. for load-balancing.
- Can prove no 2 philosophers hold same fork as `Eat()` is fork's critical section.
  - If $\#P_i$ is number of philos with fork *i* then $\boldsymbol{Fork(i) + \#P_i = 1}$

    (ie either philo has the fork or sem is 1)
- Since a semaphore is non-negative then $\boldsymbol{\#P_i \leq 1}$.
- But deadlock possible (i.e none can eat) when all philos pick up their left forks together;
  - i.e. all execute `P(fork[i])` before `P(fork[(i+1)mod 5]`
- Two solutions:
  - Make one philosopher take a right fork first (asymmetric solution);
  - Only allow four philosophers into the room at any one time.

Dining Philosophers#2: Symmetric Solution

```
/* pseudo-code for room solution to dining philosophers */
/* fork is array of semaphores all initialised to have value 1 */

semaphore Room := 4
semaphore fork (5) := ((5) 1)
process philosopher (i := 0 to 4) {
     while (1) {
            Think ( );      // thinking not a CS!
            wait (Room);
            wait(fork (i));
            wait(fork ((i+1) mod 5);

            Eat ( )         // eating is the CS

            signal(fork (i));
            signal(fork ((i+1) mod 5);
            signal (Room);
      }
}
```

- This solution solves the deadlock problem.
- It is also symmetric (i.e. all processes execute same code).

Dining Philosophers#2: Asymmetric Solution

```
/* pseudo-code for asymentric solution to dining philosophers */
/* fork is array of semaphores all initialised to have value 1 */

semaphore fork (5) := ((5) 1)
process philosopher (i := 0 to 4) {
     while (1) {
          Think ( );     // thinking not a CS!

          wait(min (fork (i), fork ((i+1) mod 5 );
          wait(max (fork (i), fork ((i+1) mod 5 );

          Eat ( )        // eating is the CS

          signal (max (fork (i), fork ((i+1) mod 5 );
          signal (min (fork (i), fork ((i+1) mod 5 );

     }
}
```

- This solution solves the deadlock problem.
- It is also asymmetric as the last philosopher now picks up his right fork first to preserve the order of resources (forks)).

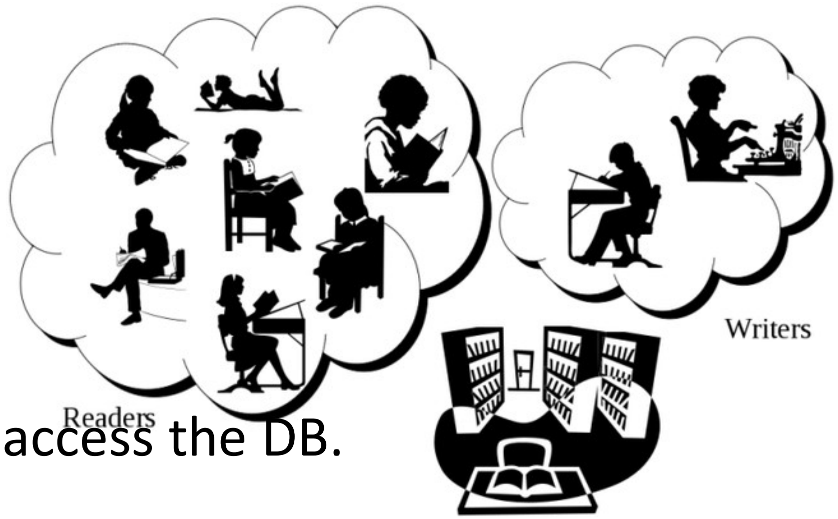
Dining Philosophers (cont'd)

- For this system to operate correctly it is required that:
  1. A philosopher eats only if he has two forks.
  2. No two philosophers can hold the same fork simultaneously.
  3. There can be no deadlock.
  4. There can be no individual starvation.
  5. There must be efficient behaviour under the absence of contention.

- This problem is a generalisation of multiple processes accessing a set of shared resources;
  – e.g. a network of computers accessing a bank of printers.

3. The Readers-Writers Problem

- Two kinds of processes, readers & writers, share a DB.
- Readers run transactions that examine the DB, writers can examine/update the DB.
- Given initial DB consistency, to ensure that it stays so, writer process must have exclusive access.



- Any number of readers may concurrently access the DB.
- Obviously, for writers, writing is a CS; cannot interleave with any other process.

The Readers-Writers Problem (cont'd)

```
int M:= 20; int N:= 5;
int nr:=0; //numReaders
sem mutexR := 1; sem rw := 1
process reader (i:= 1 to M) {
   while (1)   {
        wait (mutexR);
        nr := nr + 1;
        if (nr = 1)
               wait (rw);
        signal (mutexR);
        Read_Database ( );
        wait (mutexR);
        nr := nr - 1;
        if (nr = 0 )
               signal (rw)
        signal (mutexR);
   }
}
```

```
process writer(i:=1 to N) {
   while (1)
        wait (rw);
        Update_Database ( );
        signal (rw);
   }
}
```

- Called *readers' preference* solution:

  If a reader accesses DB then reader & writer arrive at their entry protocols then readers always have preference over writers.

Readers-Writers: Ballhausen's Solution

- Readers' Preference isn't fair.
- A continual flow of readers blocks writers from updating the database.
- Ballhausen's solution tackles this:
  - Solution idea: Efficiency: one reader takes up the same space as all readers reading together.
  - A semaphore **`access`** is used for readers to enter DB, with a value initially equalling the total number of readers.
  - Every time a reader accesses the DB, the value of **`access`** is decremented and when one leaves, it is incremented.
  - Writer wants to enter DB, occupies all spaces step by step by waiting for all old readers to leave and blocking entry to new ones.
  - The writer uses a semaphore **`mutex`** to prevent deadlock between two writers trying to occupy half available space each.

Readers-Writers: Ballhausen's Solution (cont'd)

```
sem mutex = 1;
sem access = m;

void reader ( int i ) {
   while (1)
         P(access);

         // ... reading ...

         V(access);
         // other operations
   }
}
```

```
void writer ( int i ) {
   while (1)  {
         P(mutex);
         for k = 1 to m {
                P(access);
         }
         //... writing ...
         for k = 1 to m {
                V(access);
         }
         // other operations
         V(mutex);
   }
}
int main ( ) {
cobegin
   reader (1);reader (2);reader (3);
   writer (1); writer (2);
}
```

Monitors

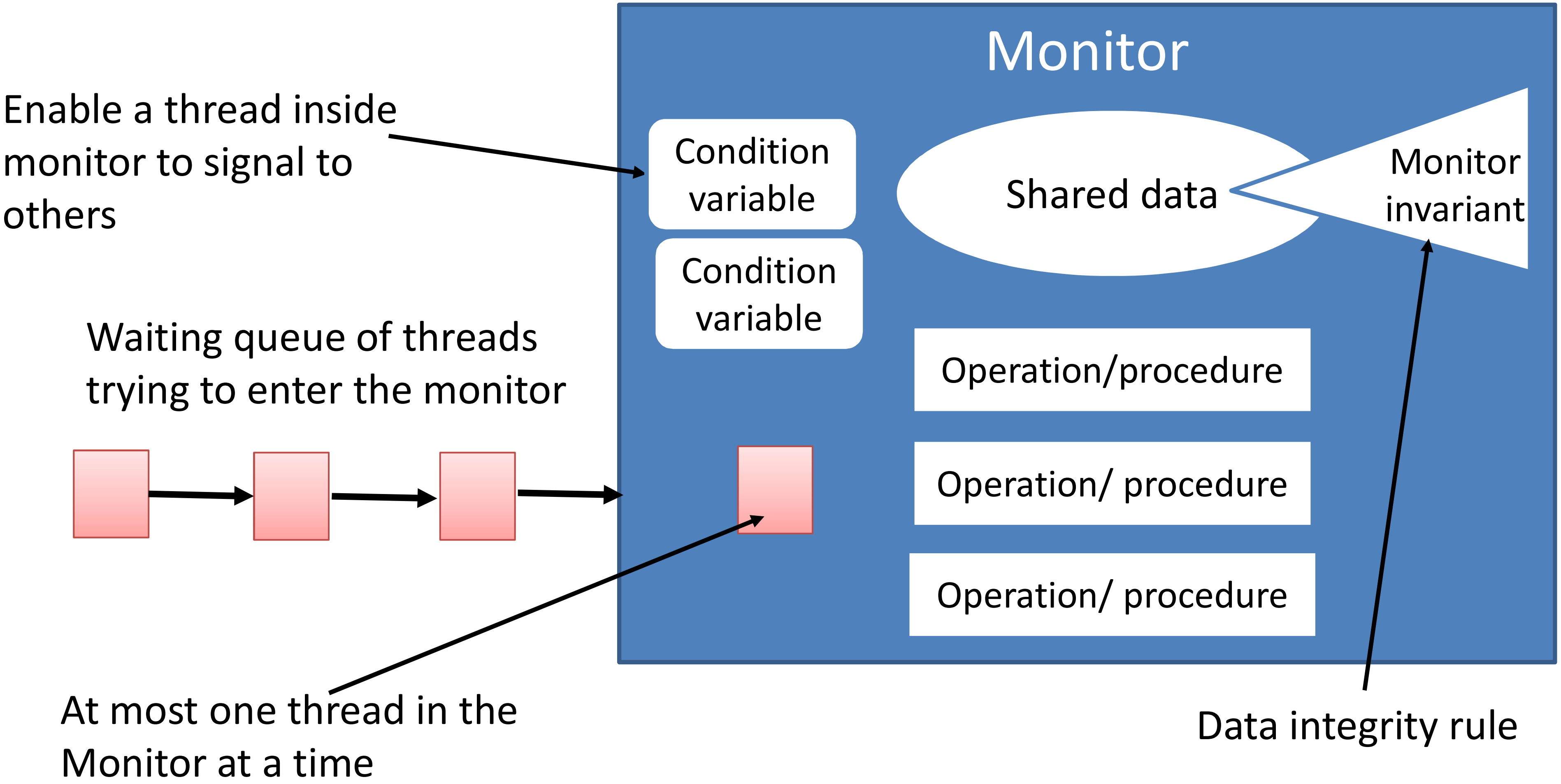
- A programming language construct that supports controlled access to shared data
  - Synchronization code added by compiler, enforced at runtime i.e. less work for programmer!
- Monitor is a software module that encapsulates
  - **Shared data** structures (ie multiple threads)
  - **Procedures** that operate on the shared data
  - **Synchronization** between concurrent processes that invoke those procedures
  - Ie Monitor keeps track of <u>who</u> is allowed to access the shared data and <u>when</u> they can do it
- Monitor protects the data from unstructured access
  - Guarantees can only access data through procedures, hence in legitimate ways

Implementation with Monitors

- Similar to a class in Java
  - Encapsulates a set of shared data
  - Set of operations to manipulate it
  - Similar to accessing a private member variable
- Using Monitors: When you identify a set of shared data being used by multiple threads:
  1. Create a monitor to contain the data
  2. Create a set of operations to work on the data
  3. Define a set of synchronisation rules between threads that invoke the operations

     $\Rightarrow$ You can control when a thread should + should not execute
- Protects you against synchronization issues

Monitor Concept



=> Provides **mutex** for the operations/procedures acting on the data

Monitor Affordances

- Mutual exclusion
  - Only one thread can be executing inside at any time (synchronization is implicitly associated with monitor)
    => Every time a thread invokes an operation, the monitor code will acquire a lock preventing other monitor operations (+ the same one) from being called
  - If a second thread tries to enter a monitor procedures, it blocks until the first has left the monitor
    => More restrictive than semaphores, but usually easier to use
  - Note mutex is explicitly associated with data, with semaphore/lock programmer had to remember the relationship
- Once inside a thread may discover it can't continue and may wish to sleep eg a counsumer thread may have to wait for a producer to produce data
  - Or allow another thread to continue
  - **Condition variables** provided within monitor
    - Have 2 operations:
      - Wait(): waits for condition variable to be signalled
      - Signal():signals a condition variable and can wake up other threads
    - Condition variable can only be accessed from inside monitor

Monitor Invariant

- Consistency rules for shared data
  - Bounded Buffer example:
    - 0 <= Count of items <= size of buffer
  - Linked List example
    - All list items have both forward and back link for a doubly linked list
  - Priority queue example
    - For all items I, priority(i) <= priority(successor(i))
- Monitor invariant must hold **whenever monitor lock is free**
  - While held, can violate to manipulate data structures
  - Nobody else can see intermediate states

$\Rightarrow$ A thread manipulating the data must always ensure the invariant is true before it releases the lock

$\Rightarrow$ A thread entering the monitor can always assume the invariant is true

Monitors in Java

- Built-in language feature
  - Use synchronized keyword on a method

```
Class Counter{
   private int count = 0;
   public void synchronized increment(){
      int n = count;
      count = n + 1;
   }
}
```

- Compiler automatically creates a lock for increment() and generates code to acquire this lock on entry to increment()
- You don't have to
  - Write the code
  - Remember that this is shared state
  - Acquire or release the lock

Condition Variables

- Key feature of monitors
  - Replace wait/signal of semaphores but different semantics
- A place to wait, sometimes called a rendezvous point
  - Always used with a monitor lock
  - No value (history) associated with condition variable ie unlike a semaphore
- Three operations on condition variables
  - Wait(c)
    - Release monitor lock, so another thread can get in

    $\Rightarrow$Can check variables without holding the lock

    $\Rightarrow$Eliminates many hold and wait conditions
    - Wait for someone else to signal condition

    $\Rightarrow$Condition variables have wait queues
    - Must ensure invariant is true before you call wait

Condition Signalling

- Signal(c) (or notify(c)) means
  - Wake **one thread** waiting on this condition variable (**if any**)
    - Signaller can keep lock and CPU
  - Waiter is made ready*, but the signaller continues
    - Waiter runs when signaller leaves monitor or waits
    - Condition is not necessarily true when waiter runs again
  - Signaller need not restore invariant until it leaves the monitor
- Broadcast(c) (or NotifyAll)
  - Wake **all threads** waiting on condition variable
  - Avoids need for multiple condition variables

* I.e. it joins the queue waiting for the lock outside the monitor

Waiting

- Waiting on a condition variable releases the lock and puts the thread on the condition variable's wait queue
  - Guarantees no other thread can enter the monitor before this thread is on the queue
  - i.e. no thread can call signal before it is put on queue
- Threads stay on the queue until signalled (and at head of queue) or broadcast
- After signalled, threads wait to acquire the monitor lock before running
  - There could be other threads ahead of it in the queue

General use of Signal

```
Class Counter{
//Prints value every time count>0
   private int count = 0;
   public void synchronized increment(){
      int n = count;
      count = n + 1;
       If (count > 0)
             notify(); //let print know
   }
   public void synchronized decrement(){
      int n = count;
      count = n - 1;
   }
   public void synchronized printVal(){
      if (count <= 0)
         wait();
      System.out.println("Count=" + count);
   }
}
```

Problem: no guarantee notified thread runs right away
Eg
Count = -1
T1: increment //count = 0
T2 printVal // count=0, waits
T3: increment //count=1, signals
T1 decrement //count = 0
T2: printVal wakes up
//count = 0 WRONG

Solution: Treat Waking as a Hint

- Woken up => something has changed
- **Another thread may have entered monitor between the signal() and the wake up**
- Implication: must re-check conditional after waking
  - Test in a while loop
  - When thread wakes it will re-test

```
public void synchronized printVal(){
      while (count <= 0)
         wait();
      System.out.println("Count=" + count);
   }

}
```

Monitors (cont'd): Signal & Continue

- If a monitor guarantees mutual exclusion:
  - A process uses the *signal* operation
  - So wakes up another process suspended in the monitor,
  - So 2 processes in same monitor at once????
  - Yes.
- To solve: a few signalling constructs: simplest *signal & continue (previously described)*.
  - This is the Mesa semantics
  - There are other solutions e.g. **Hoare Semantics**

Readers-Writers Using Monitors in C

```
/* Copyright (C) 2006 M. Ben-Ari */
monitor RW {
  int NR = 0, NW = 0;
  condition OK2Rd, OK2Wr;

  void StartRead() {
    if (NW || !empty(OK2Wr))
        waitc(OK2Rd);
    NR := NR + 1;
    signalc(OK2Rd);   }

  void EndRead() {
    NR := NR - 1;
    if (NR == 0) signalc(OK2Wr); }

  void StartWrite() {
    if (NW || (NR! = 0))
        waitc(OK2Wr);
    NW = 1;
  }
```

```
  void EndWrite() {
     NW = 0;
     if (empty(OK2Rd))
         signalc(OK2Wr);
     else signalc(OK2Rd); } }

void Reader(int N) { int i;
  for (i = 1; i < 10; i++) {
     StartRead();
     cout << N << "reading" << '\n';
     EndRead();   } }

void Writer(int N) { int i;
 for (i = 1; i < 10; i++) {
     StartWrite();
     cout << N << "writing" << '\n';
     EndWrite();    } }

void main() {
  cobegin { Reader(1); Reader(2);
Reader(3); Writer(1); Writer (2);}
}
```

File rw_control.c

Emulating Semaphores Using Monitors

- Semaphores/monitors are concurrent programming primitives of equal power: Monitors are just a higher level construct.

```
/* Copyright (C) 2006 M. Ben-Ari. */
monitor monsemaphore {
int semvalue = 1;
condition notbusy;

void monp()   {
        if (semvalue == 0)
                waitc(notbusy);
        else
                semvalue = semvalue - 1;
        }

void monv()   {
        if (empty(notbusy)) /* none susp'd? */
                semvalue = semvalue + 1;
        else
                signalc(notbusy); /* wake susp'd*/
        }

}
```

```
int n;

void inc(int i)
{
  monp();
  n = n + 1;
  monv();
}

main() {
  cobegin {
  inc(1); inc(2);
  }
  cout << n;
}
```

Dining Philosophers Using Monitors

```
monitor (fork_mon)
/* Assumes: wait( ), signal( )*/
/* and condition variables    */
   int fork:= ((5) 2);
   condition (ok2eat, 5)
/* array of condition variables */

   void (take_fork (i)) {
     if ( fork (i) != 2 )
        waitc (ok2eat(i));

        fork ((i-1) mod 5):=
                fork((i-1) mod 5)-1;
        fork ((i+1) mod 5) :=
                fork((i+1) mod 5)-1;
   }

   void release_fork (i)        {
       fork ((i-1) mod 5):=
                fork((i-1) mod 5)+1;
       fork ((i+1) mod 5) :=
                fork((i+1) mod 5)+1;

    }
```

```
        if ( fork((i+1)mod 5) ==2 )
          signalc(ok2eat((i+1)mod 5));
                //rh phil can eat

        if ( fork ((i-1)mod ) == 2 )
          signalc(ok2eat((i-1)mod 5));
                //lh phil can eat

    }
}

void philo ( int i )     {
    while (1)   {
        Think ( );
        take_fork (i);
        Eat ( );
        release_fork (i);
    }
}
void main( ) {
    cobegin { philo(1); philo(2);
philo(3); philo(4); philo(5); }
}
```

The Sleeping Barber Problem (cont'd)

- The barber and customers are interacting processes,
- The barber shop is the monitor in which they interact.

Monitors: The Sleeping Barber Problem

- A small barber shop has two doors, an entrance and an exit.
- Inside, barber spends all his life serving customers, one at a time.

1. When there are none in the shop, he sleeps in his chair.
2. If a customer arrives and finds the barber asleep:
   - he awakens the barber,
   - sits in the customer's chair and sleeps while hair is being cut.
3. If a customer arrives and the barber is busy cutting hair,
   - the customer goes asleep in one of the two waiting chairs.
4. When the barber finishes cutting a customer's hair,
   - he awakens the customer and holds the exit door open for him.
5. If there are waiting customers,
   - he awakens one and waits for the customer to sit in the barber's chair,
   - otherwise he sleeps.

Sleeping Barber Using Monitors (cont'd)

- For the Barbershop, the monitor provides an environment for the customers and barber to rendezvous
- There are four synchronisation conditions:
  – Customers must wait for barber to be available to get a haircut
  – Customers have to wait for barber to open door for them
  – Barber needs to wait for customers to arrive
  – Barber needs to wait for customer to leave
- Processes
  – wait on conditions using **`wait()`**s in loops
  – **`signal()`** at points when conditions are true

Monitors: The Sleeping Barber Problem (cont'd)

- Use three counters to synchronize the participants:
  - `barber`, `chair` and open (all initialised to zero)
- Variables alternate between zero and unity:
  1. `barber==1` the barber is ready to get another customer
  2. `chair==1` customer sitting on chair but no cutting yet
  3. `open==1` exit is open but customer not gone yet,
- The following are the synchronization conditions:
  - Customer waits until barber is available
  - Customer remains in chair until barber opens it
  - Barber waits until customer occupies chair
  - Barber waits until customer leaves

Monitors: Sleeping Barbers (cont'd)

```
monitor (barber_shop)
    int barber:=0; int chair :=0; int open :=0;
    condition (barber_available) ;        // signalled when barber > 0
    condition (chair_occupied) ;          // signalled when chair > 0
    condition (door_open) ;               // signalled when open > 0
    condition (customer_left) ;           // signalled when open = 0
```

```
void (get_haircut()) {
    do
      waitc(barber_available)
    while ( barber==0)

    barber := barber – 1;
    chair := chair + 1;

    signalc (chair_occupied);
    do
      waitc (door_open)
    while (open==0)

    open := open – 1;
    signalc (customer_left);
} // called by customer
```

```
void (get_next_customer( ))  {
      barber := barber +1;
      signalc(barber_available);

      do
        waitc(chair_occupied)
      while ( chair == 0 )

      chair := chair -1;
} // called by barber

void (finished_cut( )) {
      open := open +1;
      signalc (door_open);

      do
        waitc(customer_left)
      while (open==0)
}  // called by barber
}
```

Sleeping Barber Using Monitors (cont'd)

```
void customer ( i ) {
    while (1) {
         get_haircut ( );
         // let it grow
    }
}

void barber ( i ) {
    while (1) {
         get_next_customer ( );
         // cut hair
         finished_cut ( )
    }
}

int main ( ) {
  cobegin {
         barber (1); barber (2);
         customer (1); customer (2);
  }
}
```

Summary

- Can define a concurrent program as the interleaving of sets of sequential atomic instructions.
- Ensuring correctness of concurrent programs is tough even for two process systems as need to ensure both *Safety* & *Liveness* properties.
- Semaphores & Monitors facilitate synchronization among processes.
- Monitors are higher level but can emulate either one by other.
- Monitors provide a shared environment for processes to rendezvous.
- Both have been used to solve classical synchronization Problems:
  - Producers & Consumers
  - Readers & Writers
  - Dining Philosophers
  - Sleeping Barber

Go Concurrency Exercises

- No lecture tomorrow
- Please work on these instead (not graded)
- Submit solutions through Loop by 24th Feb
- http://whipperstacker.com/2015/10/05/3-trivial-concurrency-exercises-for-the-confused-newbie-gopher/
- https://github.com/loong/go-concurrency-exercises
- Will discuss on 25th Feb