6SENG002W Concurrent Programming

Lecture 10

Semaphores & Java Semaphores

Semaphores & Java Semaphores

Aim of this lecture is to:

- Introduce the concept of a *semaphore*: *binary* & *general*;
- Describe semaphore *operations*:
  - *initialise*
  - *“claim”*
  - *“release”*
- Present an overview of Java’s `Semaphore` class;
- Using Semaphores to achieve:
  - *ordering* of the actions of processes,
  - *mutual exclusion* of a critical section,
  - *conditional synchronization*.
- Solve the concurrency *resource sharing* problems using *semaphores*:
  - *Producer/Consumer* problem
  - *Readers & Writers* problem
  - *Dining Philosophers* problem.

PART I

The Semaphore Mechanism

(Edsger W. Dijkstra 1968)

What is a Semaphore?

Invented by *Edsger W. Dijkstra* during research into Operating Systems in mid 1960s.

A *semaphore* is a programming language mechanism that is used in concurrent programming to achieve the following:

- *resolve race conditions*
- *achieve mutual exclusion*
- *achieve processes synchronization & coordination*.

By combining these basic elements together, a programmer can ensure that a collection of concurrent processes *safely* share resources & coordinate their activities.

Semaphores were the first specific & most primitive form of synchronization mechanism introduced in to programming languages to achieve these tasks.

You can think of a semaphore as a special kind of *"lock"* that has operations that allow it to be either "locked" or "unlocked"

Definition of a Semaphore

A *semaphore* $\mathtt{s}$ is an integer variable that can have *only non–negative values*, i.e. $\mathtt{s} \geq \mathtt{0}$.

There are two types of semaphores:

- *Binary* semaphores can only have the values 0 or 1.

  A binary semaphore is also called a "*mutex*", from mutual exclusion.

- *General* semaphores can have any value greater than 0.

  However, general semaphores are (almost) always defined to have a *maximum* value, that *must not be exceed*.

  A general semaphore is also called a "*counting*" semaphore.

Semaphore Operations

The only operations permissible on a semaphore `s` are the following:

Claim: **"lock"** a semaphore: `s.claim()`

Synonyms: `s.acquire()`, `wait(s)` & `P(s)`

Release: **"unlock"** a semaphore: `s.release()`

Synonyms: `signal(s)` & `V(s)`

Initialise: set an initial value for a semaphore as either **"locked"** (`s = 0`) or **"unlocked"** ($s \geq 1$):

Synonyms: `s = v`

```
s = new BinarySemaphore( v )          [0 ≤ v ≤ 1]
s = new GeneralSemaphore( MAX, v )    [0 ≤ v ≤ MAX]
```

Overview of Semaphore Operations

The standard definition of the semaphore operations are:

- **`s.claim()`** — if `s > 0` then `s = s - 1` & the **process continues**, otherwise the **process is suspended** on `s`.

  ```
  while ( s == 0 )
  {
      skip ;         // do nothing
  } ;
  s = s - 1 ;
  ```

- **`s.release()`** — if some process `P` has been suspended by a previous `s.claim()` then **wake up** `P`, otherwise `s = s + 1`.

  If a number of processes have been *suspended* on `s` then select one of them to be woken up, the others remain suspended.

  For binary semaphores the definition of `s.release()` is changed to `s = 1`.

- **`s = new Semaphore( v )`** — initialise the semaphore `s` with the value `v`, i.e. `s = v`.

Implementing Semaphores

There are a number of issues that need to be resolved when implementing the semaphore mechanism:

Atomicity: of the *claim* & *release* operations.

Waiting: within a *claim* operation.

Blocking & Waking: *claiming* processes.

Atomicity of Claim & Release operations – Resolving Race Conditions

Semaphores are implemented in the *kernels* of many operating systems & *real-time executives*.

This is because the operations:

- `s.claim()` &
- `s.release()`

MUST BE: *atomic*, *primitive* & *uninterruptible*.

This means that if several `s.claim()` &/or `s.release()` operations are attempted simultaneously on a single semaphore `s`, then

- **only one of them would succeed**, &
- **the others would be blocked**.

This is how *race conditions* are resolved when several processes attempt to `claim` the same semaphore at the same time.

Waiting within a Claim operation

The definition of semaphores seems to imply some sort of *“busy wait”* by `s.claim()` until `s` becomes non-zero.

However, since *“busy wait”* is inefficient, the kernel of an operating system, usually implements semaphores by using a *“blocking wait”* as shown below:

```
claim(s):
           if ( s > 0 ) then
               decrement s
           else
               block execution of the calling process
```

```
release(s):
            if ( processes blocked on s ) then
                wake up one of them
            else
                 increment s
```

Blocking & Waking Claiming Processes

Implementations of semaphores usually implement the:

- **"blocking"** (or "suspending") of a process claiming a **locked** semaphore,
- the **"wake-up"** strategy for blocked processes on a newly **unlocked** semaphore

by associating a *First-In-First-Out* (FIFO) queue with each semaphore.

This makes the above tasks straightforward & the two semaphore operations become:

- `s.claim()` — if `s > 0` then `s = s - 1` & the process can continue. Otherwise, the process is suspended & *appended to the end of the semaphore queue.*

- `s.release()` — if the semaphore queue is **NOT empty** then *wake–up the first process in the queue*.
  Otherwise, if the semaphore queue is empty then `s = s + 1`.

Notes about Implementations of Semaphores

1. Another approach would be to extend the FIFO queue method by assigning a *priority* to each process.
   Then when a `claim` operation causes a process to be blocked it is inserted into the queue in front of all other processes which have a lower priority.

2. A FIFO queuing should not be relied on in reasoning about the correctness of semaphore programs.

3. Java's `Semaphore` class can support this notion of FIFO queuing, if the *"fairness"* option is used, see the `Semaphore` class's constructor.

A Brief History of Semaphores

Invented by Edsger W. Dijkstra, as part of research into the development of *"THE"* operating system, & published in the following paper:

The Structure of the "THE"– Multiprogramming System,
Communications of the ACM, Volume 11, Number 5, pp 341–346, May 1968.

(Semaphores are described in the *Appendix* of this paper!)

In Dijkstra's original the `claim` operation was called **P**, from the Dutch word *passeren*, meaning "to pass".

The `release` operation was called **V**, from the Dutch word *vrijgeven*, meaning "to release".

Subsequently, many synchronization mechanisms have been developed & incorporated into programming languages, e.g.

- *conditional-critical-regions* (Java's `synchronized` statement)
- *co-routines* (Simula 67)
- *monitors* (Concurrent Pascal, Mesa, Java)
- *rendezvouses* (Ada)
- *synchronised message passing* (`occam`)

PART II

Java's `Semaphore` *Class*

`java.util.concurrent.Semaphore`

Java's `Semaphore` Class (1)

- The *claim* operation is called `acquire`:
  `s.claim()  -> s.acquire()`
- A semaphore *"maintains a set of permits"*, but in reality there are no permit objects.
- The value of the semaphore is the number of *permits available*, i.e. how many `acquire`s can be performed on it.
- `acquire()` **blocks** if necessary until a permit is available & then takes it.
- `release()` adds a permit, potentially releasing a blocked process waiting to complete an `acquire()`.
- Methods provided to *acquire & release multiple permits* at a time.
- A semaphore that is **"locked"** can be released by a thread other than the owner, as *semaphores have no notion of "ownership"*.

Java's `Semaphore` Class (2)

There are two `Semaphore` ***constructors***:

```
public Semaphore( int permits )

public Semaphore( int permits, boolean fair )
```

The first creates a `Semaphore` with the given number of permits & **non-fair** fairness setting.

The second creates a Semaphore as above but the fairness setting can be determined by the `fair`-ness parameter:

- `false`: no guarantees about the order in which threads acquire permits, this allows *"barging"*, i.e. **queue jumping**.
- `true`: threads are granted permits in the order they made the `acquire` calls, via a FIFO queue.

Java's `Semaphore` Class (3)

It is *strongly recommended* that:

> *(general) semaphores used to control resource access should be initialised as fair, to avoid threads being starved out from accessing a resource.*

```
final boolean FAIR = true ;

Semaphore free_space = new Semaphore( SIZE, FAIR ) ;
```

For more details see the `Semaphore` class API & its description at:

java.util.concurrent.Semaphore

PART III

Using Semaphores to achieve:

Action Ordering,
Mutual Exclusion
&
Conditional Synchronization

Using Semaphores: Ordering Process Actions

Enforcing a *strict order/interleaving* of the actions of two processes, can be achieved using *two binary semaphores*.

Each semaphore is used to *control the progress of one* of the two processes.

Each semaphore is *claimed* by only one process & is only *released* by the other process.

Using two processes `P1`, `P2` & *two semaphores* **`s1`** & **`s2`** we have:

	`s1`	**`s2`**
`P1`	claims	releases
`P2`	releases	claims

So:

- `P1`'s progress is **blocked** by having to claim **`s1`**
- `P1` allows `P2` to **progress** by releasing **`s2`**
- `P2`'s progress is **blocked** by having to claim **`s2`**
- `P2` allows `P1` to **progress** by releasing **`s1`**

Semaphores Ordering Process Actions: Processes

```
Program Strict_Interleaving
{
    process P1( BinarySemaphore s1, BinarySemaphore s2 )
    {
       s1.claim() ;
       first_actions ;
       s2.release() ;       // Claims s1, Releases s2
       s1.claim() ;
       third_actions ;
       s2.release() ;
    }


    process P2( BinarySemaphore s1, BinarySemaphore s2 )
    {
       s2.claim() ;
       second_actions ;
       s1.release() ;       // Claims s2, Releases s1
       s2.claim() ;
       fourth_actions ;
    }
```

Semaphores Ordering Process Actions: Main Program

```
  main()
  {
   final int UNLOCKED = 1 ;
   final int LOCKED   = 0 ;

    BinarySemaphore s1 = new BinarySemaphore( UNLOCKED ) ;

    BinarySemaphore s2 = new BinarySemaphore( LOCKED ) ;

    parbegin
         P1( s1, s2 ) ;
         P2( s1, s2 ) ;
    parend;
  }

} //  Strict_Interleaving
```

See a version using Java's `Semaphore` class on the module web site.

Interleaving of the Program

The program results in a strict interleaving of the actions of `P1` & `P2`:

Process	Action	s1	s2
	Initialisation	1	0
P2	**blocked** attempting `s2.claim()`	1	0
P1	`s1.claim()`	0	0
P1	`first_actions`	0	0
P1	`s2.release()`	0	1
P1	**blocked** attempting `s1.claim()`	0	1
P2	**unblocked** & completes `s2.claim()`	0	0
P2	`second_actions`	0	0
P2	`s1.release()`	1	0
P2	**blocked** attempting `s2.claim()`	1	0
P1	**unblocked** & completes `s1.claim()`	0	0
P1	`third_actions`	0	0
P1	`s2.release()`	0	1
P2	**unblocked** & completes `s2.claim()`	0	0
P2	`fourth_actions`	0	0

Using Semaphores: Mutual Exclusion

The abstract mutual exclusion problem has the general form:

```
locking-protocol
critical section
unlocking-protocol
```

Mutual exclusion is very easy to achieve using semaphores & can be done using a *single binary semaphore*.

In the following example, the two processes each compete to claim the **`mutex`** semaphore.

When a process is successful it can execute the critical section, the loosing process is blocked & must wait until the first process releases the semaphore.

Example: Mutual Exclusion using Semaphores

```
Program Mutual_Exclusion
{
    process P1( BinarySemaphore mutex )
    {
        while ( true )
        {
           mutex.claim() ;          // locking-protocol
           use_critical_section ;
           mutex.release() ;        // unlocking-protocol
        }
    }


    process P2( BinarySemaphore mutex )
    {
        while ( true )
        {
           mutex.claim() ;           // locking-protocol
           use_critical_section ;
           mutex.release() ;         // unlocking-protocol
        }
    }
```

Semaphores Mutual Exclusion: Main Program

```
  main()
  {
    final int UNLOCKED = 1 ;

    BinarySemaphore mutex = new BinarySemaphore( UNLOCKED ) ;

    parbegin
       P1( mutex ) ;
       P2( mutex ) ;
    parend;
  }

} // Mutual_Exclusion
```

See a version of this program using Java's `Semaphore` class on the module web site.

Interleavings of Program `Mutual_Exclusion`

Program has an *arbitrary interleaving* of use of the *critical section* by `P1` & `P2`.

Following is just an initial sequence of one of the many possible interleavings.

`P1` wins race with `P2` & *successfully claims* **`mutex`**, & `P2` is **blocked**.

Process	Action	**`mutex`**
	Initialisation	1
P1	`mutex.claim()`	0
P2	**blocked** attempting `mutex.claim()`	0
P1	use critical section	0
P1	`mutex.release()`	1
P2	**unblocked** & completes `mutex.claim()`	0
P1	**blocked** attempting `mutex.claim()`	0
P2	use critical section	0
P2	`mutex.release()`	1
P1	**unblocked** & completes `mutex.claim()`	0
P2	**blocked** attempting `mutex.claim()`	0
P1	use critical section	0
P1	`mutex.release()`	1
P1	`mutex.claim()`	0
P1	use critical section	0
P1	`mutex.release()`	1
⋮	⋮	⋮

Using Semaphores: Conditional Synchronization

When two processes share a data object which is in a state inappropriate for executing a particular operation, such an operation should be delayed.

To implement *conditional synchronization*:

- *shared variables* are used to represent the *condition* &
- a *semaphore* is associated with the *condition* & is used to achieve *synchronization*.

After a process has made the condition `true`, it signals this by **releasing** the semaphore.

A process is delayed until the condition has become `true`, by attempting to **claim** the semaphore.

For controlling resource allocation general semaphores are useful.

The semaphore is initialised to the *number of resources available*.

A resource is allocated by performing a **claim** operation & a resource is returned by performing a **release** operation.

Mapping "Conditions" to Semaphore Operations & States

When considering using *"Conditional Synchronization"* as part of a resource allocation algorithm the initial steps are to:

1. Identify the important *"conditions"* within the system.
2. Map "conditions" onto: collections of *shared variables* & *semaphore operations & states*.

For example, for each *condition*, it is useful to consider the following:

- Shared variable necessary for state information.
- If (*condition = true*) then a **claim** should be successful.
- If (*condition = false*) then a **claim** should be unsuccessful, i.e. a claim is blocked.
- Should a **claim** make: condition = false.
- Should a **release** make: condition = true.
- When using a *general semaphore*, how does its *value relate to the condition*?
  Since this is no longer a simple true/false condition.

Example of Conditional Synchronization

```
Program ConditionalSynchronization
{
    int P1s_result, P2s_result, result ;

    process P1( BinarySemaphore finished )
    {
       calculate(P1s_result) ;
       finished.release() ;
    }

    process P2( BinarySemaphore finished )
    {
       calculate(P2s_result);
       finished.release() ;
    }

    process P3( BinarySemaphore P1_finished,
                BinarySemaphore P2_finished  )
    {
       P1_finished.claim() ;     // These ``claims'' could
       P2_finished.claim() ;     // be in either order.

       result = combine( P1s_result, P2s_result ) ;
    }
```

Semaphores Conditional Synchronization: Main Program

```
main()
{
  final int LOCKED = 0 ;

  BinarySemaphore P1_finished = new BinarySemaphore( LOCKED ) ;
  BinarySemaphore P2_finished = new BinarySemaphore( LOCKED ) ;

  parbegin
       P1( P1_finished ) ;
       P2( P2_finished ) ;

       P3( P1_finished, P2_finished ) ;
  parend ;
}

} // ConditionalSynchronization
```

PART IV

Classic Problems Solved Using Semaphores

The Producer/Consumer problem

See Lectures 7 & 8 for full details of the Producer/Consumer problem.

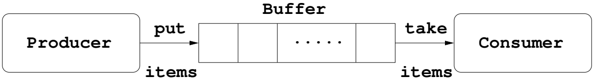

Figure : 10.1 Producer/Consumer Problem – with a Buffer.

Safety & Liveness Requirements

Flow control: use *synchronization* to prevent:

- producer from appending data to a **full buffer**, &
- consumer from reading data from an **empty buffer**.

Mutually Exclusive: use *synchronization* to ensure that reading from & writing to the buffer is *mutually exclusive*.

Semaphores used in Producer/Consumer Problem

`mutex`: is a `BinarySemaphore` used to ensure *mutual exclusion* for access to the buffer.

`free_space`: is a `GeneralSemaphore` which indicates how many *free slots* there are in the buffer.

`num_item`: is a `GeneralSemaphore` which indicates the *number of items* currently in the buffer.

Questions: What should the *initial values* of these semaphores be?

Can you map **"conditions"** to *semaphore states*?

See the two versions of this program using Java's `Semaphore` class on the module web site:

- 1 Producer & 1 Consumer
- 3 Producer & 3 Consumer

Producer/Consumer Problem Using Semaphores

```
Program ProducerConsumer
{
   final int SIZE  = 10 ; // size of the buffer ;

   Items[] buffer = new Items[SIZE] ;
   int in  = 0 ;
   int out = 0 ;

    process Producer( BinarySemaphore  mutex,
                      GeneralSemaphore free_space,
                      GeneralSemaphore num_items   )
    {
       Items item ;

       while ( true )
       {
           produce(item);

           free_space.claim() ;      // check for empty slots
           mutex.claim() ;
           buffer[in] = item ;
           mutex.release() ;
           in = (in + 1) % SIZE ;
           num_items.release() ;     // indicate new data
       }
    } ;
```

Consumer

```
process Consumer( BinarySemaphore  mutex,
                  GeneralSemaphore free_space,
                  GeneralSemaphore num_items    )
{
    Items item ;

    while ( true )
    {
       num_items.claim() ;          // check for new data
       mutex.claim() ;
       item = buffer[out] ;
       mutex.release() ;
       out = (out + 1) % SIZE ;
       free_space.release() ;       // indicate a free slot
       consume(item) ;
    } ;
} ;
```

Main Program

```
main()
{
  final int UNLOCKED = 1 ;
  final int LOCKED   = 0 ;

  // buffer is claimable -> "unlocked" (UNLOCKED)
  BinarySemaphore mutex = new BinarySemaphore( UNLOCKED ) ;

  // nothing in buffer -> "locked" (LOCKED)
  GeneralSemaphore num_items = new GeneralSemaphore( SIZE, LOCKED ) ;

  // all (SIZE) slots empty -> "unlocked" (SIZE)
  GeneralSemaphore free_space = new GeneralSemaphore( SIZE, SIZE ) ;


  parbegin
        Producer( mutex, free_space, num_items ) ;
        Consumer( mutex, free_space, num_items ) ;
  parend;
 }

} // ProducerConsumer
```

Analysis of the `ProducerConsumer` Program

Questions:

1. Would this solution work if there were several Producers & Consumers?

```
parbegin
  Producer( m, fs, ni ); ... Producer( m, fs, ni );
  Consumer( m, fs, ni ); ... Consumer( m, fs, ni );
parend
```

2. What happens if the order of the `release`s in the Producer were changed?

```
mutex.release() ;                num_items.release() ;

in = (in + 1) % N ;   --->    in = (in + 1) % N ;

num_items.release() ;            mutex.release() ;
```

3. What happens if the order of the `claim`s in the Consumer were changed?

```
num_items.claim() ;              mutex.claim() ;
                      --->
mutex.claim() ;                  num_items.claim() ;
```

The Readers & Writers Problem

For details of the Readers & Writers Problem see Lecture 8.

Invariants:

1. $0 \leq \#Writers \leq 1$
2. $0 \leq \#Readers \leq m$
3. $\#Writers = 1 \Rightarrow \#Readers = 0$
4. $\#Readers > 0 \Rightarrow \#Writers = 0$

Readers & Writers Behaviour

1. *permission*(`Reader`, `read_data_base`) when $\#Readers \geq 0$ & $\#Writers = 0$
2. *non–permission*(`Reader`, `read_data_base`) when $\#Writers = 1$
3. *permission*(`Writer`, `write_data_base`) when $\#Readers = 0$ & $\#Writers = 0$
4. *non–permission*(`Writer`, `write_data_base`) when $\#Readers > 0$ or $\#Writers = 1$

Readers/Writers Problem using Semaphores

In this version of the Readers/Writers Problem *two binary semaphores* are used:

`mutex`: used to ensure *mutual exclusion* for access to the variable `Number_of_Readers`, which is shared between the Readers.

`writing`: controls when writing to the database is allowed.

How it works

It helps to think of the database as having three possible "logical states" – *"reading"*, *"writing"* & *"neutral"*.

If no Writer is currently writing then the first Reader prohibits writing to the database & the system switches from *neutral* into *reading* state.

In *reading* state subsequent Readers can just read from the database.

The last Reader to stop reading & leave the database switches from *reading* into *neutral* state.

In *writing* state subsequent Writers are blocked & Readers are also blocked.

Readers/Writers Database States

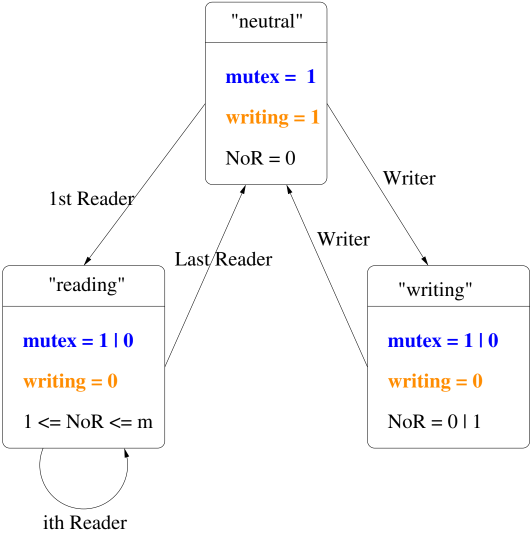

Figure : 10.2 Readers & Writers “Logical” States.

Readers/Writers Program

```
Program Readers_Writers
{
  int Number_of_Readers = 0 ;

  process Reader ( BinarySemaphore mutex, BinarySemaphore writing )
  {
    while ( true )
    {
      mutex.claim() ;                  // enter CS

         Number_of_Readers++ ;
         // 1st Reader disables writing
         if ( Number_of_Readers == 1 )  writing.claim() ;

      mutex.release();                 // leave CS

      // READ FROM DATABASE

      mutex.claim();                   // enter CS

         Number_of_Readers-- ;
         // last Reader enables writing
         if ( Number_of_Readers == 0 )  writing.release() ;

      mutex.release() ;                // leave CS
    }
  } // Reader
```

Writers & Main Program

```
process Writer( BinarySemaphore writing )
{
  while ( true )
  {
     writing.claim() ;           // enter CS
         // WRITE TO DATABASE
     writing.release() ;         // leave CS
  } ;
 } // Writer


 main()   // Main Program
 {
   final int UNLOCKED = 1 ;

   BinarySemaphore mutex   = new BinarySemaphore( UNLOCKED ) ;
   BinarySemaphore writing = new BinarySemaphore( UNLOCKED ) ;

   parbegin
      Reader( mutex, writing ) ; ... Reader( mutex, writing ) ;

      Writer( writing ) ; ... ; Writer( writing ) ;
   parend ;
 }
} // Readers_Writers
```

Dining Philosophers Problem

For details of the Dining Philosophers Problem see previous Lectures.

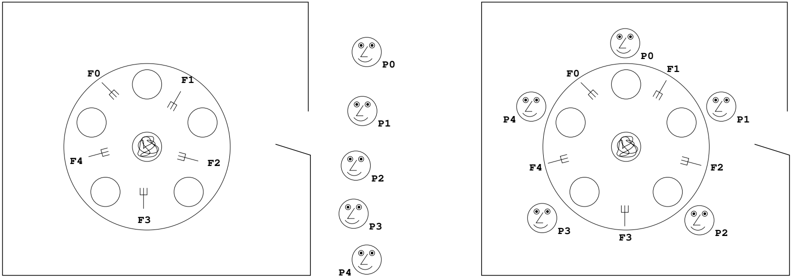

Figure : 10.3 The Dining Philosopher's Table

Each philosopher needs to pick up two forks to eat the spaghetti.

With unconstrained behaviour it is possible for **deadlock** to occur when all the philosophers are seated & have the fork to their right.

Deadlocking Dining Philosophers Problem using Semaphores

```
Program Dining_Philosophers_DEADLOCK
{
   final int NUM_PHILS = 5 ;

   process Philosopher( int i,
                        BinarySemaphore leftFork,
                        BinarySemaphore rightFork )
   {
      while ( true )
      {
        // Think ;

        rightFork.claim() ;      // pick up 2 forks
        leftFork.claim() ;

        // Eat ;

        rightFork.release() ;    // put down 2 forks
        leftFork.release() ;

      }

   }
```

Main Program

```
main()
{
   final int UNLOCKED = 1 ;

   BinarySemaphore Fork[] = new BinarySemaphore[NUM_PHILS] ;

   for ( int i = 0 ; i < NUM_PHILS ; i++ )
   {
        // forks can be picked up -> "unlocked"
        Fork[i] = new BinarySemaphore( UNLOCKED ) ;
   }

    parbegin
       // Philosopher( id,  leftFork, rightfork )

          Philosopher(  0,  Fork[1],  Fork[0] ) ;
          Philosopher(  1,  Fork[2],  Fork[1] ) ;
          Philosopher(  2,  Fork[3],  Fork[2] ) ;
          Philosopher(  3,  Fork[4],  Fork[3] ) ;
          Philosopher(  4,  Fork[0],  Fork[4] ) ;
    parend;
  }

} // Dining_Philosophers_DEADLOCK
```

Interleaving of the Program

Program results in **deadlock** when all 5 philosophers have entered the room & picked up their own forks.

To see this consider the following interleaving.

Philosopher	Action	Forks				
	Initialisation	1	1	1	1	1
P0	Think	1	1	1	1	1
P1	Think	1	1	1	1	1
P2	Think	1	1	1	1	1
P3	Think	1	1	1	1	1
P4	Think	1	1	1	1	1
P0	Fork[0].claim()	0	1	1	1	1
P1	Fork[1].claim()	0	0	1	1	1
P2	Fork[2].claim()	0	0	0	1	1
P3	Fork[3].claim()	0	0	0	0	1
P4	Fork[4].claim()	0	0	0	0	0
P0	**blocked** by Fork[1].claim()	0	0	0	0	0
P1	**blocked** by Fork[2].claim()	0	0	0	0	0
P2	**blocked** by Fork[3].claim()	0	0	0	0	0
P3	**blocked** by Fork[4].claim()	0	0	0	0	0
P4	**blocked** by Fork[0].claim()	0	0	0	0	0
		DEADLOCK!!				

First Deadlock Free Solution to the Dining Philosophers Problem

In this version of the Dining Philosophers Problem **deadlock is avoided** by adding a *general semaphore* called `Butler`.

- `Butler` represents the idea of a *butler*, who limits the number of philosophers entering the dining room to 4, i.e. 1 less than their number.
- It thus has the effect of limiting the number of philosophers that compete for the 5 forks to 4.
  This *ensures that at least one philosopher can always pick up two forks* & **thus avoids deadlock**.
- It is initialised to 4 – the number of philosophers that can *"safely"* be allowed into the dining room at once.
- Every philosopher must **claim** it before entering the dining room & **release** it on leaving.

See a version of this program using Java's `Semaphore` class on the module web site.

Deadlock Free Philosopher Process

```
Program Dining_Philosophers_BUTLER
{
   final int NUM_PHILS = 5 ;

   process Philosopher( int i,
                        BinarySemaphore  leftFork,
                        BinarySemaphore  rightFork
                        GeneralSemaphore butler    )
   {
      while ( true )
      {
          // Think ;

          butler.claim() ;       // enter dining room

              rightFork.claim() ;
              leftFork.claim() ;

              // Eat ;

              rightFork.release() ;
              leftFork.release() ;

          butler.release() ;     // leave dining room
       } ;
   }
```

Deadlock Free Main Program

```
 main()
 {
   final int UNLOCKED = 1 ;
   final int CAPACITY = NUM_PHILS - 1 ;  // Max 4 Phils in room

   BinarySemaphore Fork[] = new BinarySemaphore[NUM_PHILS] ;

   for ( int i = 0 ; i < NUM_PHILS ; i++ )
   {
      // forks can be picked up -> "unlocked"
      Fork[i] = new BinarySemaphore( UNLOCKED ) ;
   }

   GeneralSemaphore Butler
                 = new GeneralSemaphore( CAPACITY, CAPACITY ) ;

   parbegin
    // Philosopher( id,  leftFork, rightfork, butler )
       Philosopher(  0,  Fork[1],  Fork[0],   Butler ) ;
       Philosopher(  1,  Fork[2],  Fork[1],   Butler ) ;
       Philosopher(  2,  Fork[3],  Fork[2],   Butler ) ;
       Philosopher(  3,  Fork[4],  Fork[3],   Butler ) ;
       Philosopher(  4,  Fork[0],  Fork[4],   Butler ) ;
    parend;
 }
} // Dining_Philosophers_BUTLER
```

Dining Philosophers Problem: Second Deadlock Free Solution

In this version of the Dining Philosophers Problem **deadlock is avoided** by using a *non-symmetric* solution, without the `Butler` semaphore.

- The first 4 philosophers (0 – 3) behave as before, i.e. are *right handed* & pick up their own fork then their neighbours.
- But now the last philosopher – 4 behaves differently, i.e. is *left handed* & picks up their neighbours fork then their own.
- The result of this is that the first right handed philosopher & the left handed philosopher immediately try to claim their shared fork **`Fork[0]`**:

```
RightHanded_Philosopher( 0, Fork[1], Fork[0] )

LeftHanded_Philosopher( 4, Fork[0], Fork[4] )
```

- Since only one of them can be successful one of them is immediately blocked & then at most 4 philosophers will be competing for the 4 forks; & thus ensuring **no deadlock**.

“Right Handed” Philosopher Processes

```
Program Dining_Philosophers_LEFT_HANDED
{
   final int NUM_PHILS = 5 ;

   process RightHanded_Philosopher( int i,
                                    BinarySemaphore leftFork,
                                    BinarySemaphore rightFork )
   {
      while ( true )
      {
        // Think ;

        rightFork.claim() ;  // pick up RIGHT fork first
        leftFork.claim() ;

        // Eat ;

        rightFork.release() ;
        leftFork.release() ;

      } ;
   }
```

“Left Handed” Philosopher Process

```
process LeftHanded_Philosopher( int i,
                                BinarySemaphore  leftFork,
                                BinarySemaphore  rightFork )
 {
    while ( true )
    {
      // Think ;

      leftFork.claim()] ;   // pick up LEFT fork first
      rightFork.claim() ;

      // Eat ;

      leftFork.release() ;
      rightFork.release() ;
   } ;

 }
```

Deadlock Free “Left Handed” Main Program

```
  main()
  {
    final int UNLOCKED = 1 ;

    BinarySemaphore Fork[] = new BinarySemaphore[NUM_PHILS] ;

    for ( int i = 0 ; i < NUM_PHILS ; i++ )
    {
      Fork[i] = new BinarySemaphore( UNLOCKED ) ;
    }

    parbegin
      // P0 & P4 both attempt to pick up Fork[0] first
      // RightHanded_Philosopher( id,  leftFork, rightfork )

         RightHanded_Philosopher(  0,  Fork[1],  Fork[0] ) ;
         RightHanded_Philosopher(  1,  Fork[2],  Fork[1] ) ;
         RightHanded_Philosopher(  2,  Fork[3],  Fork[2] ) ;
         RightHanded_Philosopher(  3,  Fork[4],  Fork[3] ) ;

      // LeftHanded_Philosopher( id,  leftFork, rightfork )

         LeftHanded_Philosopher(  4,  Fork[0],  Fork[4] ) ;
    parend ;
  }
} // Dining_Philosophers_LEFT_HANDED
```

PART V

Disadvantages of using Semaphores

Practical Disadvantages of using Semaphores (1)

The use of semaphores can easily lead to errors.

The most common types of errors are caused by making mistakes in the ordering of the `claim` & `release` operations.

This *slight* mistake can lead to serious program errors such as **deadlock**, **failure to maintain mutual exclusion**, etc.

For example, if a process interchanges the operations on a semaphore `mutex` & we have:

```
mutex.release() ;
// use critical section ;
mutex.claim() ;
```

several processes could enter the critical section at once!

If we use the wrong operation *deadlock* can occur:

```
mutex.claim() ;
// use critical section ;
mutex.claim() ;
```

Practical Disadvantages of using Semaphores (2)

Also if we get the ordering of two or more `claim` operations on different semaphores wrong, *deadlock* or some other error could occur:

```
mutex_1.claim() ;     -->   mutex_2.claim() ;
mutex_2.claim() ;           mutex_1.claim() ;
```

Other potential errors can be caused by:

- omitting a `claim` or a `release` altogether;
- by initialising a semaphore to an **incorrect value**, e.g. **locked** when it should be **unlocked** or vice versa, etc;
- claiming &/or releasing the wrong semaphore.

Semaphores Can Seriously Damage Your Program!

- The KEY point is that all of the above errors result in the **incorrect "behaviour"** of the processes concerned.

- BUT in general, due to the inherently huge number of possible nondeterministic behaviours, they may be **impossible to detect**, **until they occur**.

Philosophical Disadvantages of using Semaphores

The philosophical disadvantages of using semaphores are that they are a *very low level mechanism* which are **unstructured**.

- *Mutual exclusion*, *process ordering* & *conditional synchronization* are all achieved using the same primitive mechanism.
- A process can *only test one semaphore at a time*, it is not possible to test groups of semaphores all in one go.
- Once a process has committed itself to testing a semaphore (i.e. performing a **`claim`**), if the semaphore is 0 then it is stuck & can not withdraw & attempt to do something else such as try another semaphore.

Another reason for not using them is that they become **VERY cumbersome** when dealing with many processes. (*Many* often means **more than 2!**)

Also they are **not a very good** *Software Engineering* approach, in that any "resource management" algorithm designed using them is **"distributed"** across the client processes of the shared resource.

Thus making it very difficult to manage & ensure it is correct.