IIT Madras

ONLINE DEGREE

Concurrency

What is concurrency?

- In all the procedures we have seen so far, only one step was being executed at a time

What is concurrency?

- In all the procedures we have seen so far, only one step was being executed at a time
  - In real life, several things are going on at the same time
  - This means that two or more steps could be executed simultaneously
  - This is called **concurrency**

What is concurrency?

- In all the procedures we have seen so far, only one step was being executed at a time
  - In real life, several things are going on at the same time
  - This means that two or more steps could be executed simultaneously
  - This is called **concurrency**
- Specifically, two procedure calls P() and Q() can be executed concurrently

What is concurrency?

- In all the procedures we have seen so far, only one step was being executed at a time
  - In real life, several things are going on at the same time
  - This means that two or more steps could be executed simultaneously
  - This is called **concurrency**
- Specifically, two procedure calls P() and Q() can be executed concurrently
- Concurrency means that the procedure calls have to be unbundled.
- The step $\mathrm{B} = \mathrm{P}(\mathrm{A})$ involves the following different actions:

What is concurrency?

- In all the procedures we have seen so far, only one step was being executed at a time
  - In real life, several things are going on at the same time
  - This means that two or more steps could be executed simultaneously
  - This is called **concurrency**
- Specifically, two procedure calls P() and Q() can be executed concurrently
- Concurrency means that the procedure calls have to be unbundled.
- The step B = P(A) involves the following different actions:
  - Start the procedure P and pass the parameter A to it
  - Wait for the procedure to complete its work
  - Accept the result of the procedure and store the result in B

What is concurrency?

- In all the procedures we have seen so far, only one step was being executed at a time
  - In real life, several things are going on at the same time
  - This means that two or more steps could be executed simultaneously
  - This is called **concurrency**
- Specifically, two procedure calls P() and Q() can be executed concurrently
- Concurrency means that the procedure calls have to be unbundled.
- The step $B = P(A)$ involves the following different actions:
  - Start the procedure P and pass the parameter A to it
  - Wait for the procedure to complete its work
  - Accept the result of the procedure and store the result in B
- Easier to visualise concurrency with objects.
  - For objects X and Y: X.P(A) and Y.P(A) may be executed concurrently

What is concurrency?

- In all the procedures we have seen so far, only one step was being executed at a time
  - In real life, several things are going on at the same time
  - This means that two or more steps could be executed simultaneously
  - This is called **concurrency**
- Specifically, two procedure calls P() and Q() can be executed concurrently
- Concurrency means that the procedure calls have to be unbundled.
- The step B = P(A) involves the following different actions:
  - Start the procedure P and pass the parameter A to it
  - Wait for the procedure to complete its work
  - Accept the result of the procedure and store the result in B
- Easier to visualise concurrency with objects.
  - For objects X and Y: X.P(A) and Y.P(A) may be executed concurrently
  - In this lecture, we will only look at concurrency within object oriented computing

Concurrent objects

- The step B $=$ X.P(A) involves the following different actions:
  - Start the procedure X.P and pass the parameter A to it
  - Wait for the procedure X.P to complete its work
  - Accept the result of the procedure and store the result in B

Concurrent objects

- The step B = X.P(A) involves the following different actions:
  - Start the procedure X.P and pass the parameter A to it
    - We can write this as X.start(P,A)
    - Object X is told to start its procedure P with parameter A
  - Wait for the procedure X.P to complete its work
  - Accept the result of the procedure and store the result in B

Concurrent objects

- The step $B = X.P(A)$ involves the following different actions:
  - Start the procedure X.P and pass the parameter A to it
    - We can write this as X.start(P,A)
    - Object X is told to start its procedure P with parameter A
  - Wait for the procedure X.P to complete its work
    - Object X can indicate completion by setting a flag which can be read using X.ready(P)
  - Accept the result of the procedure and store the result in B

Concurrent objects

- The step B = X.P(A) involves the following different actions:
  - Start the procedure X.P and pass the parameter A to it
    - We can write this as X.start(P,A)
    - Object X is told to start its procedure P with parameter A
  - Wait for the procedure X.P to complete its work
    - Object X can indicate completion by setting a flag which can be read using X.ready(P)
    - Object X can be **polled** by repeatedly checking if the condition X.ready(P) is true
  - Accept the result of the procedure and store the result in B

Concurrent objects

- The step B = X.P(A) involves the following different actions:
  - Start the procedure X.P and pass the parameter A to it
    - We can write this as X.start(P,A)
    - Object X is told to start its procedure P with parameter A
  - Wait for the procedure X.P to complete its work
    - Object X can indicate completion by setting a flag which can be read using X.ready(P)
    - Object X can be **polled** by repeatedly checking if the condition X.ready(P) is true
    - Meanwhile function wait(C) can put the caller in a wait state till some condition C is true
  - Accept the result of the procedure and store the result in B

Concurrent objects

- The step B = X.P(A) involves the following different actions:
  - Start the procedure X.P and pass the parameter A to it
    - We can write this as X.start(P,A)
    - Object X is told to start its procedure P with parameter A
  - Wait for the procedure X.P to complete its work
    - Object X can indicate completion by setting a flag which can be read using X.ready(P)
    - Object X can be **polled** by repeatedly checking if the condition X.ready(P) is true
    - Meanwhile function wait(C) can put the caller in a wait state till some condition C is true
    - Putting it all together: wait(X.ready(P)) makes the caller wait for the result of P to be ready
  - Accept the result of the procedure and store the result in B

Concurrent objects

- The step B = X.P(A) involves the following different actions:
  - Start the procedure X.P and pass the parameter A to it
    - We can write this as X.start(P,A)
    - Object X is told to start its procedure P with parameter A
  - Wait for the procedure X.P to complete its work
    - Object X can indicate completion by setting a flag which can be read using X.ready(P)
    - Object X can be **polled** by repeatedly checking if the condition X.ready(P) is true
    - Meanwhile function wait(C) can put the caller in a wait state till some condition C is true
    - Putting it all together: wait(X.ready(P)) makes the caller wait for the result of P to be ready
  - Accept the result of the procedure and store the result in B
    - We can write this as B = X.result(P)

Example: Classroom dataset

- Consider two objects MaT and PhT of the ClassAve datatype
  - We wish to find MaT.average() and PhT.average() concurrently

Example: Classroom dataset

- Consider two objects MaT and PhT of the ClassAve datatype
  - We wish to find MaT.average() and PhT.average() concurrently
- Recall that ClassAve had the following:
  - private fields marksList and aValue
  - public procedures average() and addStudent(newMark)

Example: Classroom dataset

- Consider two objects MaT and PhT of the ClassAve datatype
  - We wish to find MaT.average() and PhT.average() concurrently
- Recall that ClassAve had the following:
  - private fields marksList and aValue
  - public procedures average() and addStudent(newMark)
- To make average concurrent:
  - We need to implement the procedures ready(average) and result(average)

Example: Classroom dataset

- Consider two objects MaT and PhT of the ClassAve datatype
  - We wish to find MaT.average() and PhT.average() concurrently
- Recall that ClassAve had the following:
  - private fields marksList and aValue
  - public procedures average() and addStudent(newMark)
- To make average concurrent:
  - We need to implement the procedures ready(average) and result(average)
  - But we already have the field aValue which does this !

Example: Classroom dataset

- Consider two objects MaT and PhT of the ClassAve datatype
  - We wish to find MaT.average() and PhT.average() concurrently
- Recall that ClassAve had the following:
  - private fields marksList and aValue
  - public procedures average() and addStudent(newMark)
- To make average concurrent:
  - We need to implement the procedures ready(average) and result(average)
  - But we already have the field aValue which does this !
  - So ready(average) can just return the boolean value (aValue ≥ 0)

Example: Classroom dataset

- Consider two objects MaT and PhT of the ClassAve datatype
  - We wish to find MaT.average() and PhT.average() concurrently
- Recall that ClassAve had the following:
  - private fields marksList and aValue
  - public procedures average() and addStudent(newMark)
- To make average concurrent:
  - We need to implement the procedures ready(average) and result(average)
  - But we already have the field aValue which does this !
  - So ready(average) can just return the boolean value (aValue $\geq$ 0)
  - Similarly, result(average) can just return aValue

Example: Classroom dataset

- So to execute MaT.average() and PhT.average() concurrently:

  MaT.start(average)
  PhT.start(average)
  wait(MaT.ready(average) and PhT.ready(average))
  AveMaths = MaT.result(average)
  AvePhysics = PhT.result(average)

Example: Classroom dataset

- So to execute MaT.average() and PhT.average() concurrently:

  MaT.start(average)
  PhT.start(average)
  wait(MaT.ready(average) and PhT.ready(average))
  AveMaths = MaT.result(average)
  AvePhysics = PhT.result(average)

- Note that:
  - start is called with only average, as it has no parameters
  - caller waits for both MaT and PhT to be ready

Example: Classroom dataset

- Now consider the same two objects MaT and PhT of the ClassAve datatype
  - We wish to add a new student to both MaT and PhT
  - MaT.addStudent(newMark1) and PhT.addStudent(newMark2) are executed concurrently

Example: Classroom dataset

- Now consider the same two objects MaT and PhT of the ClassAve datatype
  - We wish to add a new student to both MaT and PhT
  - MaT.addStudent(newMark1) and PhT.addStudent(newMark2) are executed concurrently
- To make addStudent concurrent:
  - We need to implement the procedure ready(newStudent)

Example: Classroom dataset

- Now consider the same two objects MaT and PhT of the ClassAve datatype
  - We wish to add a new student to both MaT and PhT
  - MaT.addStudent(newMark1) and PhT.addStudent(newMark2) are executed concurrently
- To make addStudent concurrent:
  - We need to implement the procedure ready(newStudent)
  - We could use the same field aValue to implement this !

Example: Classroom dataset

- Now consider the same two objects MaT and PhT of the ClassAve datatype
  - We wish to add a new student to both MaT and PhT
  - MaT.addStudent(newMark1) and PhT.addStudent(newMark2) are executed concurrently
- To make addStudent concurrent:
  - We need to implement the procedure ready(newStudent)
  - We could use the same field aValue to implement this !
  - ready(addStudent) can just return the boolean value (aValue == -1)

Example: Classroom dataset

- Now consider the same two objects MaT and PhT of the ClassAve datatype
  - We wish to add a new student to both MaT and PhT
  - MaT.addStudent(newMark1) and PhT.addStudent(newMark2) are executed concurrently
- To make addStudent concurrent:
  - We need to implement the procedure ready(newStudent)
  - We could use the same field aValue to implement this !
  - ready(addStudent) can just return the boolean value (aValue == -1)
  - Note that addStudent does not return anything, so we don't need the result implementation

Example: Classroom dataset

- So to execute MaT.addStudent(newMark1) and PhT.addStudent(newMark2) concurrently:

  MaT.start(addStudent, newMark1)
  PhT.start(addStudent, newMark2)
  wait(MaT.ready(addStudent) and PhT.ready(addStudent))

Example: Classroom dataset

- So to execute MaT.addStudent(newMark1) and PhT.addStudent(newMark2) concurrently:

  MaT.start(addStudent, newMark1)
  PhT.start(addStudent, newMark2)
  wait(MaT.ready(addStudent) and PhT.ready(addStudent))

- Note that:
  - start is called with parameters
  - caller waits for both MaT and PhT to be ready

Example: Classroom dataset

- Now consider the situation where while MaT.average() is executing, a new student is added to MaT:

Example: Classroom dataset

- Now consider the situation where while MaT.average() is executing, a new student is added to MaT:

  MaT.start(average)
  MaT.start(addStudent,newMark)
  wait(MaT.ready(average) and MaT.ready(addStudent))
  AveMaths = MaT.result(average)

Example: Classroom dataset

- Now consider the situation where while MaT.average() is executing, a new student is added to MaT:

  MaT.start(average)
  MaT.start(addStudent,newMark)
  wait(MaT.ready(average) and MaT.ready(addStudent))
  AveMaths = MaT.result(average)

- There is potential for conflict here !
  - average will set aValue to 0 or a positive value
  - addStudent will set aValue to -1

Example: Classroom dataset

- Now consider the situation where while MaT.average() is executing, a new student is added to MaT:

  MaT.start(average)
  MaT.start(addStudent,newMark)
  wait(MaT.ready(average) and MaT.ready(addStudent))
  AveMaths = MaT.result(average)

- There is potential for conflict here !
  - average will set aValue to 0 or a positive value
  - addStudent will set aValue to -1
  - So both cannot be ready at the same time !
  - wait(MaT.ready(average) and MaT.ready(addStudent)) will just wait forever !!

Example: Classroom dataset

- Now consider the situation where while MaT.average() is executing, a new student is added to MaT:

  MaT.start(average)
  MaT.start(addStudent,newMark)
  wait(MaT.ready(average) and MaT.ready(addStudent))
  AveMaths = MaT.result(average)

- There is potential for conflict here !
  - average will set aValue to 0 or a positive value
  - addStudent will set aValue to -1
  - So both cannot be ready at the same time !
  - wait(MaT.ready(average) and MaT.ready(addStudent)) will just wait forever !!
- To prevent this, we can use explicit aveReady and addReady fields:
  - addReady is set to false when addStudent starts and to true when it finishes
  - ready(addStudent) returns the value of addReady

Example: Classroom dataset

- Now consider the situation where while MaT.average() is executing, a new student is added to MaT:

  MaT.start(average)
  MaT.start(addStudent,newMark)
  wait(MaT.ready(average) and MaT.ready(addStudent))
  AveMaths = MaT.result(average)

- There is potential for conflict here !
  - average will set aValue to 0 or a positive value
  - addStudent will set aValue to -1
  - So both cannot be ready at the same time !
  - wait(MaT.ready(average) and MaT.ready(addStudent)) will just wait forever !!
- To prevent this, we can use explicit aveReady and addReady fields:
  - addReady is set to false when addStudent starts and to true when it finishes
  - ready(addStudent) returns the value of addReady
  - Similarly for aveReady which is set after average completes

Race conditions

- addStudent needs to set two fields after it finishes
  - aValue has to be set to -1, so that the average is recomputed
  - addReady has to be set to true

Race conditions

- addStudent needs to set two fields after it finishes
  - aValue has to be set to -1, so that the average is recomputed
  - addReady has to be set to true
- Similarly, average needs to set two fields after it finishes
  - aValue has to be set to the computed average value
  - aveReady has to be set to true

Race conditions

- addStudent needs to set two fields after it finishes
  - aValue has to be set to -1, so that the average is recomputed
  - addReady has to be set to true
- Similarly, average needs to set two fields after it finishes
  - aValue has to be set to the computed average value
  - aveReady has to be set to true
- While in the addStudent case, the two values can be written in any order, in the average case, aValue must be updated before aveReady is set
  - Otherwise, we could have a race condition: aveReady allows the wait condition to be true, so the wait exits
  - caller could then execute result(average) before the aValue is set: this will result in -1 being returned as the average !

Need for atomicity

- Concurrent execution of average and addStudent:

  MaT.start(average)
  MaT.start(addStudent,newMark)
  wait(MaT.ready(average) and MaT.ready(addStudent))
  AveMaths = MaT.result(average)

- What is the average returned? Does the average include the new student?

Need for atomicity

- Concurrent execution of average and addStudent:

  MaT.start(average)
  MaT.start(addStudent,newMark)
  wait(MaT.ready(average) and MaT.ready(addStudent))
  AveMaths = MaT.result(average)

- What is the average returned? Does the average include the new student?
  - If average completes processing the list before the new student is added, then the average does not include the new student marks

Need for atomicity

- Concurrent execution of average and addStudent:

  MaT.start(average)
  MaT.start(addStudent,newMark)
  wait(MaT.ready(average) and MaT.ready(addStudent))
  AveMaths = MaT.result(average)

- What is the average returned? Does the average include the new student?
  - If average completes processing the list before the new student is added, then the average does not include the new student marks
  - On the other hand, if the new student has been added before average starts processing the list, then the average will include the new student marks

Need for atomicity

- Concurrent execution of average and addStudent:

  MaT.start(average)
  MaT.start(addStudent,newMark)
  wait(MaT.ready(average) and MaT.ready(addStudent))
  AveMaths = MaT.result(average)

- What is the average returned? Does the average include the new student?
  - If average completes processing the list before the new student is added, then the average does not include the new student marks
  - On the other hand, if the new student has been added before average starts processing the list, then the average will include the new student marks
  - But something worse can happen ! What if average is at the end of the list, just as the new element is being appended to the end of the list?

Need for atomicity

- Concurrent execution of average and addStudent:

  MaT.start(average)
  MaT.start(addStudent,newMark)
  wait(MaT.ready(average) and MaT.ready(addStudent))
  AveMaths = MaT.result(average)

- What is the average returned? Does the average include the new student?
  - If average completes processing the list before the new student is added, then the average does not include the new student marks
  - On the other hand, if the new student has been added before average starts processing the list, then the average will include the new student marks
  - But something worse can happen ! What if average is at the end of the list, just as the new element is being appended to the end of the list?
  - May lead to a erroneous list being created, or may crash !

Need for atomicity

- Concurrent execution of average and addStudent:

  MaT.start(average)
  MaT.start(addStudent,newMark)
  wait(MaT.ready(average) and MaT.ready(addStudent))
  AveMaths = MaT.result(average)

- To prevent corruption of the list, we have to make sure that we do only one list operation at a time. The list operations cannot be concurrent. We say that the list field is **atomic**.

Need for atomicity

- Concurrent execution of average and addStudent:

  MaT.start(average)
  MaT.start(addStudent,newMark)
  wait(MaT.ready(average) and MaT.ready(addStudent))
  AveMaths = MaT.result(average)

- To prevent corruption of the list, we have to make sure that we do only one list operation at a time. The list operations cannot be concurrent. We say that the list field is **atomic**.
  - If we are doing append on the list, then any first, rest, ... operation will have to wait

Need for atomicity

- Concurrent execution of average and addStudent:

  MaT.start(average)
  MaT.start(addStudent,newMark)
  wait(MaT.ready(average) and MaT.ready(addStudent))
  AveMaths = MaT.result(average)

- To prevent corruption of the list, we have to make sure that we do only one list operation at a time. The list operations cannot be concurrent. We say that the list field is **atomic**.
  - If we are doing append on the list, then any first, rest, ... operation will have to wait
  - Reverse also holds: append will have to wait for any other list operation to complete

Need for atomicity

- We could argue: why do we need to execute average and addStudent concurrently. Just do them in sequence. For instance:

  MaT.start(addStudent,newMark)
  wait(MaT.ready(addStudent))
  MaT.start(average)
  wait(MaT.ready(average))
  AveMaths = MaT.result(average)

Need for atomicity

- We could argue: why do we need to execute average and addStudent concurrently. Just do them in sequence. For instance:

  MaT.start(addStudent,newMark)
  wait(MaT.ready(addStudent))
  MaT.start(average)
  wait(MaT.ready(average))
  AveMaths = MaT.result(average)

- Will clearly return the average including the new student.

Need for atomicity

- We could argue: why do we need to execute average and addStudent concurrently. Just do them in sequence. For instance:

  MaT.start(addStudent,newMark)
  wait(MaT.ready(addStudent))
  MaT.start(average)
  wait(MaT.ready(average))
  AveMaths = MaT.result(average)

- Will clearly return the average including the new student.
- The issue is while designing the concurrent object of datatype ClassAve, we cannot control who will call the procedures and in what order

Need for atomicity

- We could argue: why do we need to execute average and addStudent concurrently. Just do them in sequence. For instance:

  MaT.start(addStudent,newMark)
  wait(MaT.ready(addStudent))
  MaT.start(average)
  wait(MaT.ready(average))
  AveMaths = MaT.result(average)

- Will clearly return the average including the new student.
- The issue is while designing the concurrent object of datatype ClassAve, we cannot control who will call the procedures and in what order
  - One caller X may call addStudent, and a different caller Y may call average concurrently

Need for atomicity

- We could argue: why do we need to execute average and addStudent concurrently. Just do them in sequence. For instance:

  MaT.start(addStudent,newMark)
  wait(MaT.ready(addStudent))
  MaT.start(average)
  wait(MaT.ready(average))
  AveMaths = MaT.result(average)

- Will clearly return the average including the new student.
- The issue is while designing the concurrent object of datatype ClassAve, we cannot control who will call the procedures and in what order
  - One caller X may call addStudent, and a different caller Y may call average concurrently
  - It is very difficult to have X and Y co-ordinate on their use of the shared object MaT

The Producer-Consumer model

- So far, the caller started the procedures and then just waited for them to finish (polling).

The Producer-Consumer model

- So far, the caller started the procedures and then just waited for them to finish (polling).
  - What if the caller also does something concurrently with the procedures ?
  - Then the called procedure will have to wait till the caller is ready before it can return the result. This can become quite complicated

The Producer-Consumer model

- So far, the caller started the procedures and then just waited for them to finish (polling).
  - What if the caller also does something concurrently with the procedures ?
  - Then the called procedure will have to wait till the caller is ready before it can return the result. This can become quite complicated
- There are two ways this can be remedied:

The Producer-Consumer model

- So far, the caller started the procedures and then just waited for them to finish (polling).
  - What if the caller also does something concurrently with the procedures ?
  - Then the called procedure will have to wait till the caller is ready before it can return the result. This can become quite complicated
- There are two ways this can be remedied:
  - The procedure when it finishes execution can **pre-empt** the caller and return its value. We will not discuss this method further here.

The Producer-Consumer model

- So far, the caller started the procedures and then just waited for them to finish (polling).
  - What if the caller also does something concurrently with the procedures ?
  - Then the called procedure will have to wait till the caller is ready before it can return the result. This can become quite complicated
- There are two ways this can be remedied:
  - The procedure when it finishes execution can **pre-empt** the caller and return its value. We will not discuss this method further here.
  - The procedure when it finishes places the result in a pre-agreed place (called a **buffer**). This gives rise to a model called the **producer-consumer** model of concurrency,

The Producer-Consumer model

- So far, the caller started the procedures and then just waited for them to finish (polling).
  - What if the caller also does something concurrently with the procedures ?
  - Then the called procedure will have to wait till the caller is ready before it can return the result. This can become quite complicated
- There are two ways this can be remedied:
  - The procedure when it finishes execution can **pre-empt** the caller and return its value. We will not discuss this method further here.
  - The procedure when it finishes places the result in a pre-agreed place (called a **buffer**). This gives rise to a model called the **producer-consumer** model of concurrency,
- In the producer-consumer model, the caller is the consumer and the object whose procedure is called is the producer.
  - The consumer may also queue up tasks (procedure calls) for the producer to execute one after the other.

The Producer-Consumer model

- So far, the caller started the procedures and then just waited for them to finish (polling).
  - What if the caller also does something concurrently with the procedures ?
  - Then the called procedure will have to wait till the caller is ready before it can return the result. This can become quite complicated
- There are two ways this can be remedied:
  - The procedure when it finishes execution can **pre-empt** the caller and return its value. We will not discuss this method further here.
  - The procedure when it finishes places the result in a pre-agreed place (called a **buffer**). This gives rise to a model called the **producer-consumer** model of concurrency,
- In the producer-consumer model, the caller is the consumer and the object whose procedure is called is the producer.
  - The consumer may also queue up tasks (procedure calls) for the producer to execute one after the other.
  - To ensure that the producer knows where to write the results for each of these tasks, the consumer can pass the result buffer as an argument in the procedure call.

Classroom example: Producer-Consumer

- Our first example: Find MaT.average() and PhT.average() concurrently

Classroom example: Producer-Consumer

- Our first example: Find MaT.average() and PhT.average() concurrently
- Let Env be an environment object representing the consumer:
  - Env has mBuff and pBuff fields for storing the results of MaT and PhT.

Classroom example: Producer-Consumer

- Our first example: Find MaT.average() and PhT.average() concurrently
- Let Env be an environment object representing the consumer:
  - Env has mBuff and pBuff fields for storing the results of MaT and PhT.
- Concurrent execution of MaT.average and PhT.average:

  MaT.start(average,mBuff)
  PhT.start(average,pBuff)
  ... Do domething else here ...
  wait(available(mBuff) and available(pBuff))
  AveMaths = mBuff
  AvePhysics = pBuff

Classroom example: Producer-Consumer

- Our first example: Find MaT.average() and PhT.average() concurrently
- Let Env be an environment object representing the consumer:
  - Env has mBuff and pBuff fields for storing the results of MaT and PhT.
- Concurrent execution of MaT.average and PhT.average:

  MaT.start(average,mBuff)
  PhT.start(average,pBuff)
  ... Do domething else here ...
  wait(available(mBuff) and available(pBuff))
  AveMaths = mBuff
  AvePhysics = pBuff

- ClassAve does not have to implement ready, result procedures. average() is the same as before, except that the result will also have to be written into the (remote) buffer.

Classroom example: Producer-Consumer

- Our first example: Find MaT.average() and PhT.average() concurrently
- Let Env be an environment object representing the consumer:
  - Env has mBuff and pBuff fields for storing the results of MaT and PhT.
- Concurrent execution of MaT.average and PhT.average:

  MaT.start(average,mBuff)
  PhT.start(average,pBuff)
  ... Do domething else here ...
  wait(available(mBuff) and available(pBuff))
  AveMaths = mBuff
  AvePhysics = pBuff

- ClassAve does not have to implement ready, result procedures. average() is the same as before, except that the result will also have to be written into the (remote) buffer.
- available(buff) returns true only if buff has been fully written by the producer
  - This prevents unwanted race conditions between read and write of the buffer

Classroom example: Producer-Consumer

- The next example: MaT.average() and MaT.addStudent(newMark) concurrently

Classroom example: Producer-Consumer

- The next example: MaT.average() and MaT.addStudent(newMark) concurrently
- Let Env be an environment object representing the consumer:
  - Env has aveBuff field for average, and addBuff field which is just a boolean flag

Classroom example: Producer-Consumer

- The next example: MaT.average() and MaT.addStudent(newMark) concurrently
- Let Env be an environment object representing the consumer:
  - Env has aveBuff field for average, and addBuff field which is just a boolean flag
- Concurrent execution of MaT.average and MaT.addStudent:

```
MaT.start(average,aveBuff)
MaT.start(addStudent,newMark,addBuff)
... Do domething else here ...
wait(available(aveBuff) and available(addBuff))
AveMaths = aveBuff
```

Classroom example: Producer-Consumer

- The next example: MaT.average() and MaT.addStudent(newMark) concurrently
- Let Env be an environment object representing the consumer:
  - Env has aveBuff field for average, and addBuff field which is just a boolean flag
- Concurrent execution of MaT.average and MaT.addStudent:

  MaT.start(average,aveBuff)
  MaT.start(addStudent,newMark,addBuff)
  ... Do domething else here ...
  wait(available(aveBuff) and available(addBuff))
  AveMaths = aveBuff

- addStudent will have to set addBuff to true after it finishes

Classroom example: Producer-Consumer

- The next example: MaT.average() and MaT.addStudent(newMark) concurrently
- Let Env be an environment object representing the consumer:
  - Env has aveBuff field for average, and addBuff field which is just a boolean flag
- Concurrent execution of MaT.average and MaT.addStudent:

  MaT.start(average,aveBuff)
  MaT.start(addStudent,newMark,addBuff)
  ... Do domething else here ...
  wait(available(aveBuff) and available(addBuff))
  AveMaths = aveBuff

- addStudent will have to set addBuff to true after it finishes
- There is no need for the two flags aveReady and addReady anymore. The write to aveBuff and addBuff serve as their equivalents

Classroom example: Producer-Consumer

- The next example: MaT.average() and MaT.addStudent(newMark) concurrently
- Let Env be an environment object representing the consumer:
  - Env has aveBuff field for average, and addBuff field which is just a boolean flag
- Concurrent execution of MaT.average and MaT.addStudent:

  MaT.start(average,aveBuff)
  MaT.start(addStudent,newMark,addBuff)
  ... Do domething else here ...
  wait(available(aveBuff) and available(addBuff))
  AveMaths = aveBuff

- addStudent will have to set addBuff to true after it finishes
- There is no need for the two flags aveReady and addReady anymore. The write to aveBuff and addBuff serve as their equivalents
- The race conditions with the list can still occur within MaT, so the need for the list field to be atomic continues

Summary

- Concurrency allows several steps to be executed simultaneously

Summary

- Concurrency allows several steps to be executed simultaneously
  - To keep control of this, we only considered concurrent execution of procedures within objects

Summary

- Concurrency allows several steps to be executed simultaneously
  - To keep control of this, we only considered concurrent execution of procedures within objects
- In the polling model, the caller starts all the procedures to be executed concurrently, and then waits for them to complete

Summary

- Concurrency allows several steps to be executed simultaneously
  - To keep control of this, we only considered concurrent execution of procedures within objects
- In the polling model, the caller starts all the procedures to be executed concurrently, and then waits for them to complete
  - The object can be polled to check if it has finished. The result can then be retrieved from the object (we used ready and result for this).

Summary

- Concurrency allows several steps to be executed simultaneously
  - To keep control of this, we only considered concurrent execution of procedures within objects
- In the polling model, the caller starts all the procedures to be executed concurrently, and then waits for them to complete
  - The object can be polled to check if it has finished. The result can then be retrieved from the object (we used ready and result for this).
- In the producer-consumer model, the caller object starts all the procedures to be executed concurrently, and passes result buffers to them to write into when they finish

Summary

- Concurrency allows several steps to be executed simultaneously
  - To keep control of this, we only considered concurrent execution of procedures within objects
- In the polling model, the caller starts all the procedures to be executed concurrently, and then waits for them to complete
  - The object can be polled to check if it has finished. The result can then be retrieved from the object (we used ready and result for this).
- In the producer-consumer model, the caller object starts all the procedures to be executed concurrently, and passes result buffers to them to write into when they finish
- In both cases:

Summary

- Concurrency allows several steps to be executed simultaneously
  - To keep control of this, we only considered concurrent execution of procedures within objects
- In the polling model, the caller starts all the procedures to be executed concurrently, and then waits for them to complete
  - The object can be polled to check if it has finished. The result can then be retrieved from the object (we used ready and result for this).
- In the producer-consumer model, the caller object starts all the procedures to be executed concurrently, and passes result buffers to them to write into when they finish
- In both cases:
  - caller used wait(C) to wait for the boolean condition C to become true

Summary

- Concurrency allows several steps to be executed simultaneously
  - To keep control of this, we only considered concurrent execution of procedures within objects
- In the polling model, the caller starts all the procedures to be executed concurrently, and then waits for them to complete
  - The object can be polled to check if it has finished. The result can then be retrieved from the object (we used ready and result for this).
- In the producer-consumer model, the caller object starts all the procedures to be executed concurrently, and passes result buffers to them to write into when they finish
- In both cases:
  - caller used wait(C) to wait for the boolean condition C to become true
  - In our polling model, C used ready() to check if the object has finished its task

Summary

- Concurrency allows several steps to be executed simultaneously
  - To keep control of this, we only considered concurrent execution of procedures within objects
- In the polling model, the caller starts all the procedures to be executed concurrently, and then waits for them to complete
  - The object can be polled to check if it has finished. The result can then be retrieved from the object (we used ready and result for this).
- In the producer-consumer model, the caller object starts all the procedures to be executed concurrently, and passes result buffers to them to write into when they finish
- In both cases:
  - caller used wait(C) to wait for the boolean condition C to become true
  - In our polling model, C used ready() to check if the object has finished its task
  - In our producer-consumer model, C called available(B) to check if B has been fully written

Summary

- Concurrency allows several steps to be executed simultaneously
  - To keep control of this, we only considered concurrent execution of procedures within objects
- In the polling model, the caller starts all the procedures to be executed concurrently, and then waits for them to complete
  - The object can be polled to check if it has finished. The result can then be retrieved from the object (we used ready and result for this).
- In the producer-consumer model, the caller object starts all the procedures to be executed concurrently, and passes result buffers to them to write into when they finish
- In both cases:
  - caller used wait(C) to wait for the boolean condition C to become true
  - In our polling model, C used ready() to check if the object has finished its task
  - In our producer-consumer model, C called available(B) to check if B has been fully written
- In both cases, race conditions have to be handled by ensuring that access to shared objects (such as lists) are made atomic (i.e. non-concurrent).