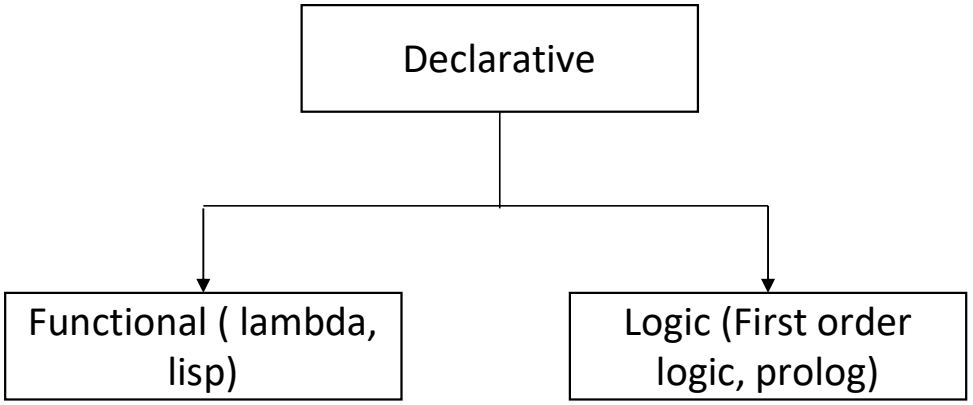
Declarative Programming Paradigm

Introduction

- Declarative programming is a programming paradigm that expresses the logic of a computation without describing its control flow.
- This paradigm often considers programs as theories of a formal logic, and computations as deductions in that logic space.
- Declarative programming is often defined as any style of programming that is not imperative.
- Common declarative languages include those of database query languages (SQL), logic programming, functional programming, etc.
- A program that describes what computation should be performed and not how to compute it. Non-imperative, non-procedural.
- Any programming language that lacks side effects(example: a function might modify a global variable or static variable, modify one of its arguments, raise an exception,).
- A language with a clear correspondence to mathematical logic.



Declarative Programming Paradigm

- A program that describes what computation should be performed and not how to compute it .
- Any programming language that lacks side effects (or more specifically, is referentially transparent).
- A language with a clear correspondence to mathematical logic
- Here, the term side effect was mentioned.
- A function or expression is said to have a side effect if, in addition to returning a value, it also modifies some state or has an observable interaction with calling functions or the outside world. ,
- Examples of declarative languages are HTML, XML, CSS, JSON and SQL, and there are more

SQL Elements

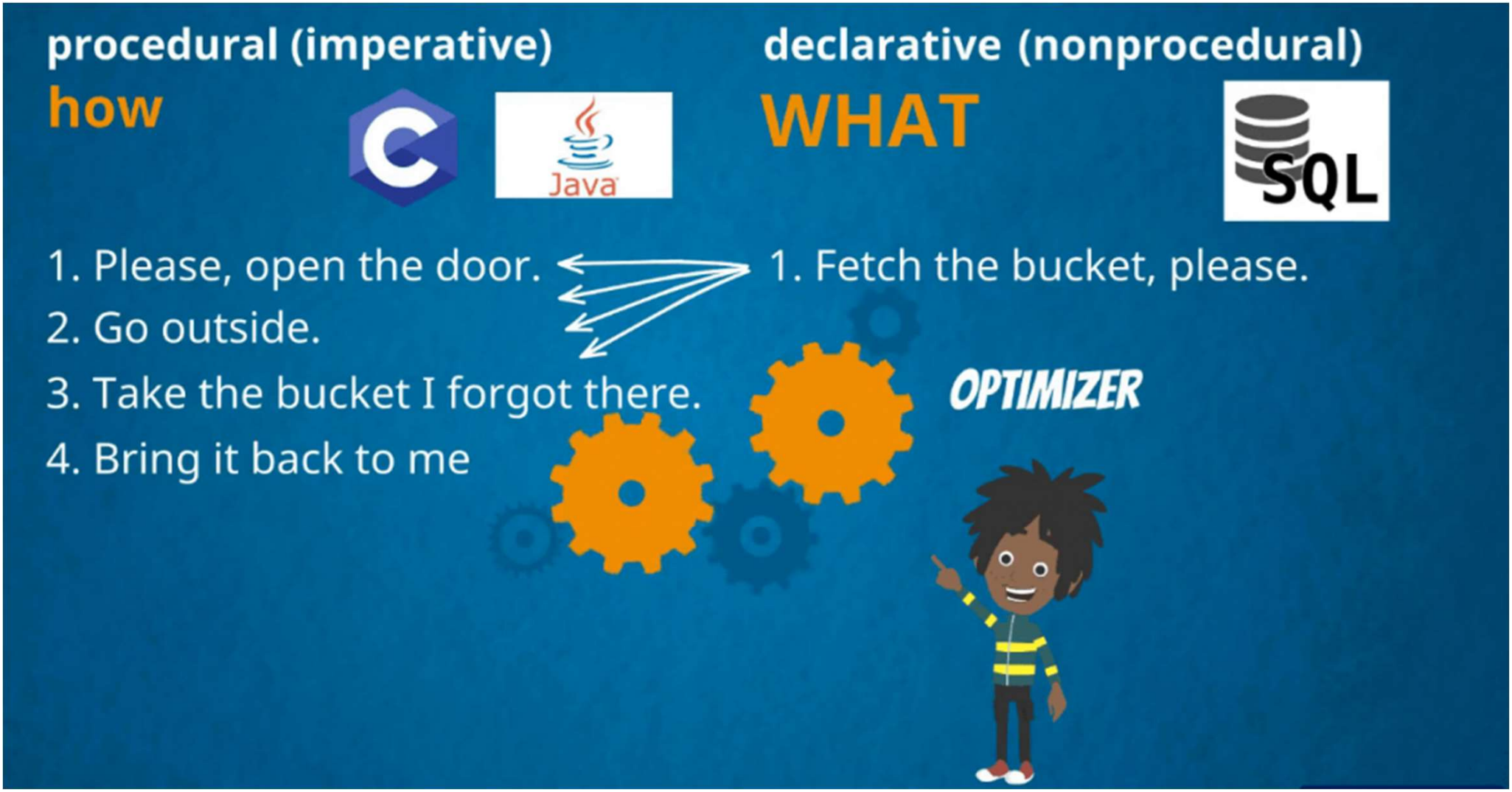
SQL is the standard language used to communicate with a relational database.

It can be used to retrieve data from a database using a query but it can also be used to create, manage, destroy as well as modify their structure and contents.

The language is subdivided into several language elements, including:

- Clauses
- Expressions
- Predicates
- Queries
- Statements

Procedure vs declarative



SQL-String data types

CHAR(size)

A FIXED length string (can contain letters, numbers, and special characters). The size parameter specifies the column length in characters - can be from 0 to 255. Default is 1

VARCHAR(size)

A VARIABLE length string (can contain letters, numbers, and special characters). The size parameter specifies the maximum column length in characters - can be from 0 to 65535

BINARY(size)

Equal to CHAR(), but stores binary byte strings. The size parameter specifies the column length in bytes. Default is 1

VARBINARY(size)

Equal to VARCHAR(), but stores binary byte strings. The size parameter specifies the maximum column length in bytes.

TINYBLOB-For BLOBs (Binary Large OBjects). Max length: 255 bytes

TINYTEXT-Holds a string with a maximum length of 255 characters

TEXT(size)- Holds a string with a maximum length of 65,535 bytes

BLOB(size)-For BLOBs (Binary Large OBjects). Holds up to 65,535 bytes of data

String data types

MEDIUMTEXT

Holds a string with a maximum length of 16,777,215 characters

MEDIUMBLOB

For BLOBs (Binary Large OBjects). Holds up to 16,777,215 bytes of data

LONGTEXT

Holds a string with a maximum length of 4,294,967,295 characters

LONGBLOB For BLOBs (Binary Large OBjects).

Holds up to 4,294,967,295 bytes of data

Numeric data types

BIT(size)A bit-value type.

The number of bits per value is specified in size. The size parameter can hold a value from 1 to 64. The default value for size is 1.

TINYINT(size)

A very small integer. Signed range is from -128 to 127. Unsigned range is from 0 to 255. The size parameter specifies the maximum display width (which is 255)

BOOL

Zero is considered as false, nonzero values are considered as true.

BOOLEAN **Equal to BOOL**

SMALLINT(size)

A small integer. Signed range is from -32768 to 32767. Unsigned range is from 0 to 65535. The size parameter specifies the maximum display width (which is 255)

MEDIUMINT(size)

A medium integer. Signed range is from -8388608 to 8388607. Unsigned range is from 0 to 16777215. The size parameter specifies the maximum display width (which is 255)

Numeric data types

INT(size)

A medium integer. Signed range is from -2147483648 to 2147483647. Unsigned range is from 0 to 4294967295. The size parameter specifies the maximum display width (which is 255)

BIGINT(size) A large integer. Signed range is from -9223372036854775808 to 9223372036854775807. Unsigned range is from 0 to 18446744073709551615. The size parameter specifies the maximum display width (which is 255)

Numeric data types

FLOAT(size, d)

A floating point number.

If p is from 0 to 24, the data type becomes FLOAT(). If p is from 25 to 53, the data type becomes DOUBLE()

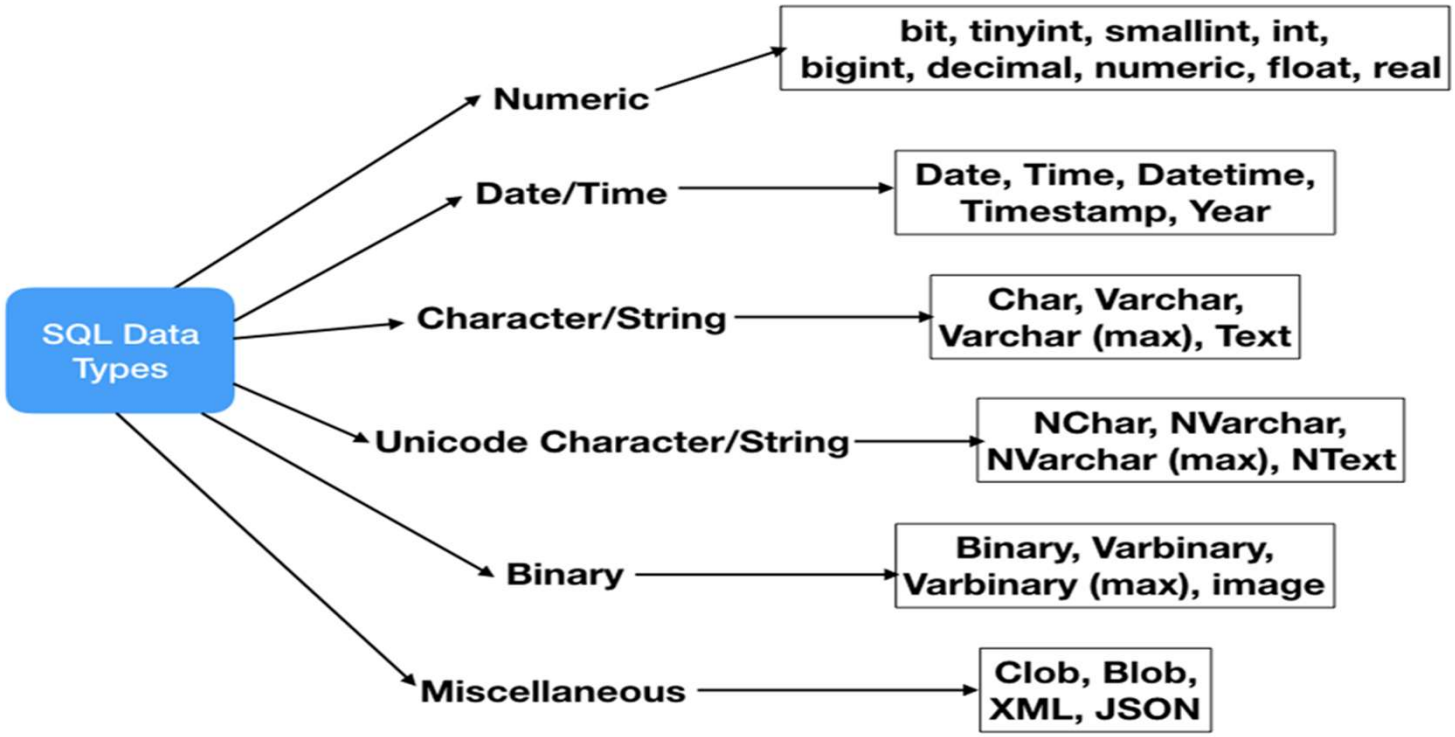
DOUBLE(size, d)

A normal-size floating point number. The total number of digits is specified in size. The number of digits after the decimal point is specified in the d parameter

DECIMAL(size, d)

An exact fixed-point number. The total number of digits is specified in size. The number of digits after the decimal point is specified in the d parameter. The maximum number for size is 65. The maximum number for d is 30. The default value for size is 10. The default value for d is 0.

SQL Data Types



SQL Data Types

```sql
CREATE TABLE Student (
        StudID int,
        LastName varchar(255),
        FirstName varchar(255),
        Gender varchar(255),
        DOB DATE
        );

INSERT INTO Student (StudID, Lastname, Firstname, Gender, DOB)
VALUES ('1000', 'H', 'John', 'Male','1990-01-01');

select * from Student;
```

```sql
select * from Scientist where Gender = 'Male';
```

```sql
UPDATE Scientist SET Firstname = 'Sundar',
    Lastname = 'Pitchai', DOB = '1972-06-10'
    where SciID = 1020;
select * from Scientist;
```

```sql
delete from Scientist where SciID = 1015;
select * from Scientist;
```

SQL Data Types

```
    conn.commit()
    print("Record Inserted")

def update(rno,name):
    cur.execute("update stud set name='"+name+"' where rno='"+rno+"'")
    conn.commit()
    print('Record updated')

def select(rno):
    cur.execute("select * from stud")
    for row in cur.fetchAll():
            if(row[0]==rno):
                    print("Name ="+name)
def delete(rno):
    cur.execute("delete from stud where rno='"+rno+"'")
    conn.commit()
    print('Record deleted')



conn=sqlite3.connect("univ.db")
cur=conn.cursor()
sql ="create table stud23 ( regno varchar(10),name varchar(20));"
cur.execute(sql)
insert('123,','bob')
insert('124,','danny')
update('124','daniel')
```

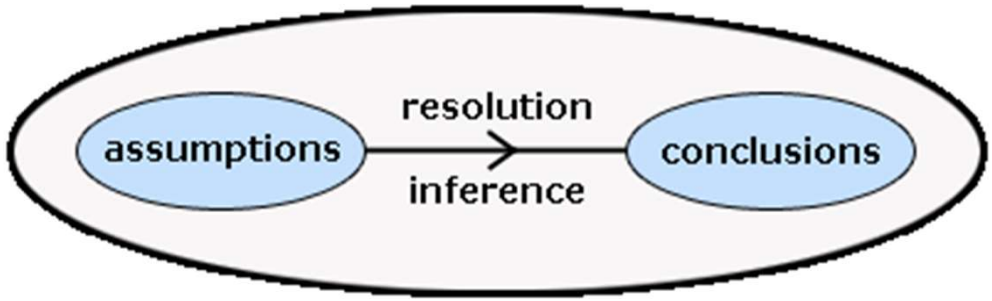
Logical Programming Paradigm

Logical Programming Paradigm

- It can be an abstract model of computation.
- Solve logical problems like puzzles
- Have knowledge base which we know before and along with the question you specify knowledge and how that knowledge is to be applied through a series of rules
- The Logical Paradigm takes a declarative approach to problem-solving.
- Various logical assertions about a situation are made, establishing all known facts.
- Then queries are made.

Logical Programming Paradigm

Logic programming is a paradigm where computation arises from proof search in a logic according to a fixed, predictable strategy. A logic is a language. It has syntax and semantics. It. More than a language, it has inference rules.



Syntax:

Syntax: the rules about how to form formulas; usually the easy part of a logic.

Semantics:

About the meaning carried by the formulas, mainly in terms of logical consequences.

Inference rules:

Inference rules describe correct ways to derive conclusions.

Logical Programming Paradigm

Logic :

A Logic program is a set of predicates. Ex parent, siblings

Predicates :

Define relations between their arguments. Logically, a Logic program states what holds. Each predicate has a name, and zero or more arguments. The predicate name is a atom. Each argument is an arbitrary Logic term. A predicate is defined by a collection of clauses.

Example: Mother(x,y)

Clause :

A clause is either a rule or a fact. The clauses that constitute a predicate denote logical alternatives: If any clause is true, then the whole predicate is true.

Example:

Mother(X,Y) <= female(X) & parent(X,Y) # implies X is the mother of Y, if X has to female

Parts of Logical Programming Paradigm

- A series of definitions/declarations that define the problem domain (fact)
- Statements of relevant facts (rules)
- Statement of goals in the form of a query (query)

Example:

Given information about fatherhood and motherhood, determine grand parent relationship

E.g. Given the information called facts

John is father of Lily

Kathy is mother of Lily

Lily is mother of Bill

Ken is father of Karen

Who are grand parents of Bill?

Who are grand parents of Karen?

Logical Programming Paradigm

Fact

A fact must start with a predicate (which is an atom). The predicate may be followed by one or more arguments which are enclosed by parentheses. The arguments can be atoms (in this case, these atoms are treated as constants), numbers, variables or lists.

Facts are axioms; relations between terms that are assumed to be true.

Example facts:

+big(‘horse')

+big('elephant')

+small('cat')

+brown(‘horse')

+black('cat')

+grey('elephant')

Consider the 3 fact saying ‘cat’ is a smallest animal and fact 6 saying the elephant is grey in color

Rule

Rules are theorems that allow new inferences to be made.

dark(X)<=black(X)

dark(X)<=brown(X)

Consider rule 1 saying the color is black its consider to be dark color.

Logical Programming Paradigm

Queries

A query is a statement starting with a predicate and followed by its arguments, some of which are variables. Similar to goals, the predicate of a valid query must have appeared in at least one fact or rule in the consulted program, and the number of arguments in the query must be the same as that appears in the consulted program.

```
print(pyDatalog.ask('father_of(X,jess)'))
```

Output:

```
{('jack',)}
X
```

```
print(father_of(X,'jess'))
```

Output:

```
jack
X
```

Logical Programming Paradigm

```
from pyDatalog import pyDatalog
pyDatalog.create_atoms('parent,male,female,son,daughter,X,Y,Z')
+male('adam')
+female('anne')
+female('barney')
+male('james')
+parent('barney','adam')
+parent('james','anne')

#The first rule is read as follows: for all X and Y, X is the son of Y if there exists X and Y such that Y is the parent of X and X is male.
#The second rule is read as follows: for all X and Y, X is the daughter of Y if there exists X and Y such that Y is the parent of X and X is female.
son(X,Y)<= male(X) & parent(Y,X)
daughter(X,Y)<= parent(Y,X) & female(X)
print(pyDatalog.ask('son(adam,Y)'))
print(pyDatalog.ask('daughter(anne,Y)'))
print(son('adam',X))
```

Logical Programming Paradigm

```
pyDatalog.create_terms('factorial, N')

factorial[N] = N*factorial[N-1]

factorial[1] = 1

print(factorial[3]==N)
```

Logical Programming Paradigm

```
from pyDatalog import pyDatalog
pyDatalog.create_terms('X,Y,Z, works_in, department_size, manager, indirect_manager, count_of_indirect_reports')
# Mary works in Production
+works_in('Mary', 'Production')
+works_in('Sam',  'Marketing')
+works_in('John', 'Production')
+works_in('John', 'Marketing')
+(manager['Mary'] == 'John')
+(manager['Sam']  == 'Mary')
+(manager['Tom']  == 'Mary')

indirect_manager(X,Y) <= (manager[X] == Y)
print(works_in(X,  'Marketing'))
indirect_manager(X,Y) <= (manager[X] == Z) & indirect_manager(Z,Y)
print(indirect_manager('Sam',X))
```

Logical Programming Paradigm

Lucy is a Professor

Danny is a Professor

James is a Lecturer

All professors are Dean

Write a Query to reterive all deans?

Soln

```
from pyDatalog import pyDatalog
pyDatalog.create_terms('X,Y,Z,professor,lecturer, dean')
+professor('lucy')
+professor('danny')
+lecturer('james')
dean(X)<=professor(X)
print(dean(X))
```

Logical Programming Paradigm

```
likes(john, susie).                 /* John likes Susie */
likes(X, susie).                    /* Everyone likes Susie */
likes(john, Y).                     /* John likes everybody */
likes(john, Y), likes(Y, john).     /* John likes everybody and everybody likes John */
likes(john, susie); likes(john,mary). /* John likes Susie or John likes Mary */
not(likes(john,pizza)).             /* John does not like pizza */
likes(john,susie) :- likes(john,mary)./* John likes Susie if John likes Mary.


rules
friends(X,Y) :- likes(X,Y),likes(Y,X).          /* X and Y are friends if they like each other */
hates(X,Y) :- not(likes(X,Y)).                  /* X hates Y if X does not like Y. */
enemies(X,Y) :- not(likes(X,Y)),not(likes(Y,X)).  /* X and Y are enemies if they don't like each other */
```

Imperative Programming Paradigm

Imperative Programming Paradigm

- It's a programming paradigm that describes computation as statements that change a program state.
- Imperative programs are a sequence of commands for the computer to perform.
- imperative programming paradigm assumes that the computer can maintain through environments of variables any changes in a computation process.
- Computations are performed through a guided sequence of steps, in which these variables are referred to or changed. The order of the steps is crucial, because a given step will have different consequences depending on the current values of variables when the step is executed.
- Popular programming languages are imperative more often than they are any other paradigm studies because the imperative paradigm most closely resembles the actual machine itself, so the programmer is much closer to the machine;
- Imperative programs define sequences of commands/statements for the computer that change a program state (i.e., set of variables)
    - Commands are stored in memory and executed in the order found
    - Commands retrieve data, perform a computation, and assign the result to a memory location

Imperative Programming Paradigm

- In a computer program, a variable stores the data. The contents of these locations at any given point in the program's execution are called the program's state. Imperative programming is characterized by programming with state and commands which modify the state.
- The first imperative programming languages were machine languages.
- Machine Language : Each instruction performs a very specific task, such as a load, a jump, or an ALU operation on a unit of data in a CPU register or memory.
    - MOV AL, BL
    - MOV A, B

Imperative vs Declarative Programming Paradigm

- Declarative programming is a programming paradigm … that expresses the logic of a computation without describing its control flow. Its focus how to do a task
- Imperative programming is a programming paradigm that uses statements that change a program's state. It focus on what to day rather than how to do it.

Example:

#declarative

```
small_nums = [ x for x in range(20) if x<5]
```

#Imperative

```
small = []
for I in range(20):
        if i<5:
                small.append(i)
```

Imperative Programming Paradigm

- Central elements of imperative paradigm:
  - Assignment statement: assigns values to memory locations and changes the current state of a program
  - Variables refer to memory locations
  - Step-by-step execution of commands
  - Control-flow statements: Conditional and unconditional (GO TO) branches and loops to change the flow of a program
- Example of computing the factorial of a number:

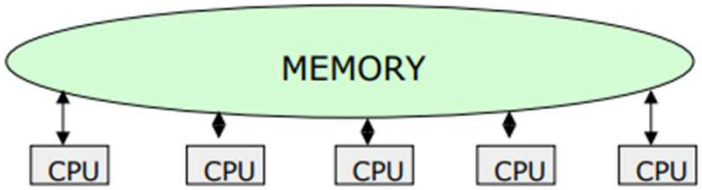
```
unsigned int n = 5;
unsigned int result = 1;
while(n > 1)
{
        result *= n;
        n--;
}
```

Parallel & Concurrent Programming Paradigm

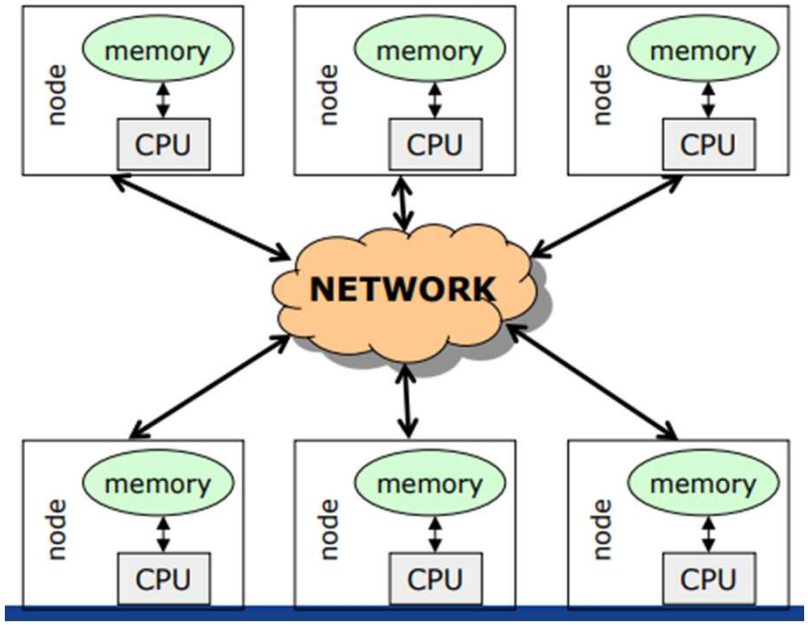
Introduction

- A system is said to be parallel if it can support two or more actions executing simultaneously i.e., multiple actions are simultaneously executed in parallel systems.
- The evolution of parallel processing, even if slow, gave rise to a considerable variety of programming paradigms.
- Parallelism Types:
  - Explicit Parallelism
  - Implicit Parallelism

Shared memory Architecture



Message Passing Architecture



Explicit parallelism

- Explicit Parallelism is characterized by the presence of explicit constructs in the programming language, aimed at describing (to a certain degree of detail) the way in which the parallel computation will take place.
- A wide range of solutions exists within this framework. One extreme is represented by the ``ancient'' use of basic, low level mechanisms to deal with parallelism--like fork/join primitives, semaphores, etc--eventually added to existing programming languages. Although this allows the highest degree of flexibility (any form of parallel control can be implemented in terms of the basic low level primitives gif), it leaves the additional layer of complexity completely on the shoulders of the programmer, making his task extremely complicate.

Implicit Parallelism

- Allows programmers to write their programs without any concern about the exploitation of parallelism. Exploitation of parallelism is instead automatically performed by the compiler and/or the runtime system. In this way the parallelism is transparent to the programmer maintaining the complexity of software development at the same level of standard sequential programming.
- Extracting parallelism implicitly is not an easy task. For imperative programming languages, the complexity of the problem is almost prohibitively and allows positive results only for restricted sets of applications (e.g., applications which perform intensive operations on arrays.
- Declarative Programming languages, and in particular Functional and Logic languages, are characterized by a very high level of abstraction, allowing the programmer to focus on what the problem is and leaving implicit many details of how the problem should be solved.
- Declarative languages have opened new doors to automatic exploitation of parallelism. Their focusing on a high level description of the problem and their mathematical nature turned into positive properties for implicit exploitation of parallelism.

Methods for parallelism

There are many methods of programming parallel computers. Two of the most common are message passing and data parallel.

1. Message Passing - the user makes calls to libraries to explicitly share information between processors.
2. Data Parallel - data partitioning determines parallelism
3. Shared Memory - multiple processes sharing common memory space
4. Remote Memory Operation - set of processes in which a process can access the memory of another process without its participation
5. Threads - a single process having multiple (concurrent) execution paths
6. Combined Models - composed of two or more of the above.

Methods for parallelism

Message Passing:

- Each Processor has direct access only to its local memory
- Processors are connected via high-speed interconnect
- Data exchange is done via explicit processor-to-processor communication i.e processes communicate by sending and receiving messages : send/receive messages
- Data transfer requires cooperative operations to be performed by each process (a send operation must have matching receive)

Data Parallel:

- Each process works on a different part of the same data structure
- Processors have direct access to global memory and I/O through bus or fast switching network
- Each processor also has its own memory (cache)
- Data structures are shared in global address space
- Concurrent access to shared memory must be coordinate
- All message passing is done invisibly to the programmer

Steps in Parallelism

- Independently from the specific paradigm considered, in order to execute a program which exploits parallelism, the programming language must supply the means to:
  - Identify parallelism, by recognizing the components of the program execution that will be (potentially) performed by different processors;
  - Start and stop parallel executions;
  - Coordinate the parallel executions (e.g., specify and implement interactions between concurrent components).

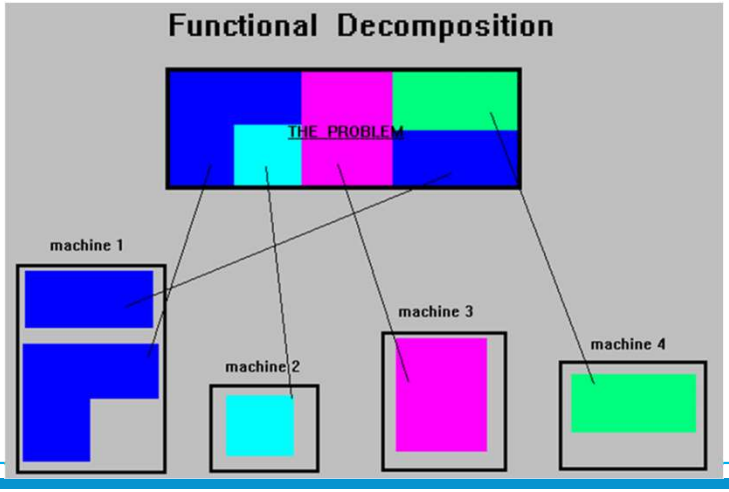
Ways for Parallelism

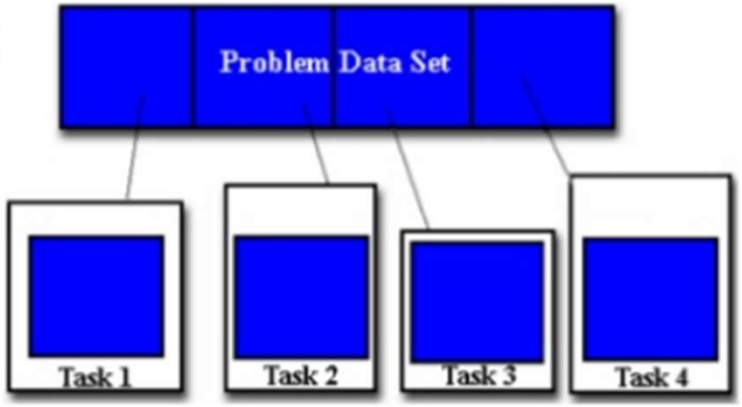
Functional Decomposition (Functional Parallelism)

- Decomposing the problem into different tasks which can be distributed to multiple processors for simultaneous execution
- Good to use when there is not static structure or fixed determination of number of calculations to be performed

Domain Decomposition (Data Parallelism)

- Partitioning the problem's data domain and distributing portions to multiple processors for simultaneous execution
- Good to use for problems where:
- data is static (factoring and solving large matrix or finite difference calculations)
- dynamic data structure tied to single entity where entity can be subsetted (large multi-body problems)
- domain is fixed but computation within various regions of the domain is dynamic (fluid vortices models)





Parallel Programming Paradigm

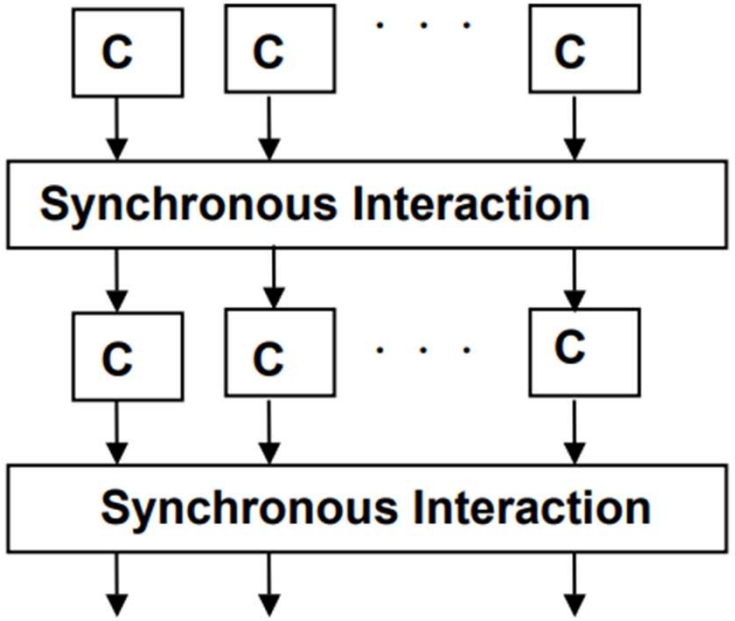
- Phase parallel
- Divide and conquer
- Pipeline
- Process farm
- Work pool

Note:

- The parallel program consists of number of super steps, and each super step has two phases : computation phase and interaction phase

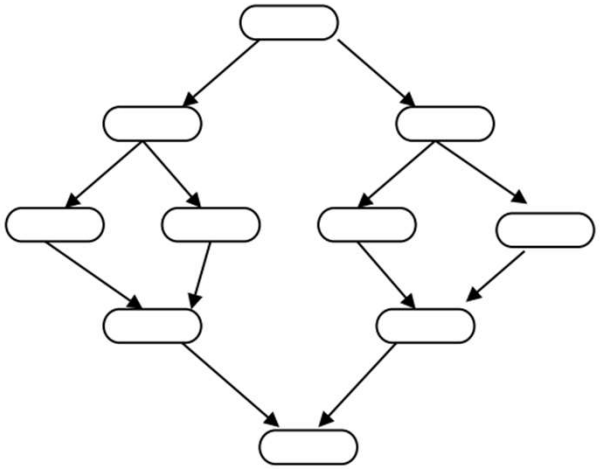
Phase Parallel Model

- The phase-parallel model offers a paradigm that is widely used in parallel programming.
- The parallel program consists of a number of supersteps, and each has two phases.
  - In a computation phase, multiple processes each perform an independent computation C.
  - In the subsequent interaction phase, the processes perform one or more synchronous interaction operations, such as a barrier or a blocking communication.
  - Then next superstep is executed.



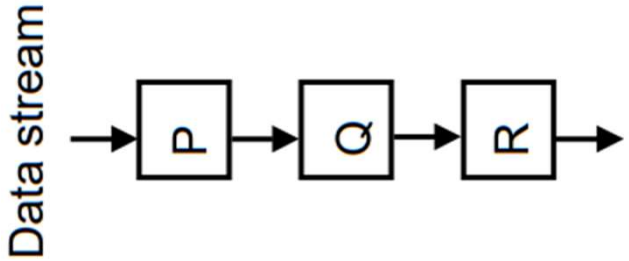
Divide and Conquer & Pipeline model

- A parent process divides its workload into several smaller pieces and assigns them to a number of child processes.
- The child processes then compute their workload in parallel and the results are merged by the parent.
- The dividing and the merging procedures are done recursively.
- This paradigm is very natural for computations such as quick sort.



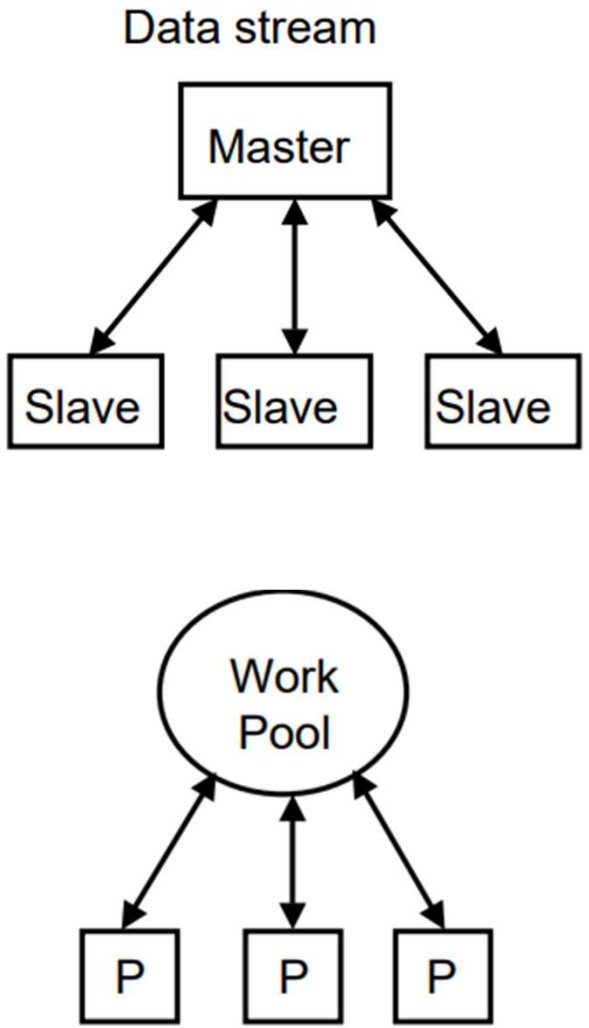
Pipeline

- In pipeline paradigm, a number of processes form a virtual pipeline.
- A continuous data stream is fed into the pipeline, and the processes execute at different pipeline stages simultaneously in an overlapped fashion.



Process Farm & Work Pool Model

- This paradigm is also known as the master-slave paradigm.
- A master process executes the essentially sequential part of the parallel program and spawns a number of slave processes to execute the parallel workload.
- When a slave finishes its workload, it informs the master which assigns a new workload to the slave.
- This is a very simple paradigm, where the coordination is done by the master.
- This paradigm is often used in a shared variable model.
- A pool of works is realized in a global data structure.
- A number of processes are created. Initially, there may be just one piece of work in the pool.
- Any free process fetches a piece of work from the pool and executes it, producing zero, one, or more new work pieces put into the pool. The parallel program ends when the work pool becomes empty.
- This paradigm facilitates load balancing, as the workload is dynamically allocated to free processes.



Parallel Program using Python

- A thread is basically an independent flow of execution. A single process can consist of multiple threads. Each thread in a program performs a particular task. For Example, when you are playing a game say FIFA on your PC, the game as a whole is a single process, but it consists of several threads responsible for playing the music, taking input from the user, running the opponent synchronously, etc.
- Threading is that it allows a user to run different parts of the program in a concurrent manner and make the design of the program simpler.
- Multithreading in Python can be achieved by importing the threading module.

Example:

```
import threading
from threading import *
```

Parallel program using Threads in Python

```
# simplest way to use a Thread is to instantiate it with a target function and call start() to let it begin working.
from threading import Thread,current_thread
print(current_thread().getName())
def mt():
    print("Child Thread")
    for i in range(11,20):
        print(i*2)
def disp():
   for i in range(10):
        print(i*2)
child=Thread(target=mt)
child.start()
disp()
print("Executing thread name :",current_thread().getName())
```

```
from threading import Thread,current_thread
class mythread(Thread):
    def run(self):
        for x in range(7):
            print("Hi from child")
a = mythread()
a.start()
a.join()
print("Bye from",current_thread().getName())
```

Parallel program using Process in Python

```
import multiprocessing
def worker(num):
    print('Worker:', num)
    for i in range(num):
        print(i)
    return

jobs = []
for i in range(1,5):
    p = multiprocessing.Process(target=worker, args=(i+10,))
    jobs.append(p)
    p.start()
```

Concurrent Programming Paradigm

- Computing systems model the world, and the world contains actors that execute independently of, but communicate with, each other. In modelling the world, many (possibly) parallel executions have to be composed and coordinated, and that's where the study of concurrency comes in.
- There are two common models for concurrent programming: shared memory and message passing.
  - **Shared memory.** In the shared memory model of concurrency, concurrent modules interact by reading and writing shared objects in memory.
  - **Message passing.** In the message-passing model, concurrent modules interact by sending messages to each other through a communication channel. Modules send off messages, and incoming messages to each module are queued up for handling

Issues Concurrent Programming Paradigm

Concurrent programming is programming with multiple tasks. The major issues of concurrent programming are:

- Sharing computational resources between the tasks;
- Interaction of the tasks.

Objects shared by multiple tasks have to be safe for concurrent access. Such objects are called protected. Tasks accessing such an object interact with each other indirectly through the object.

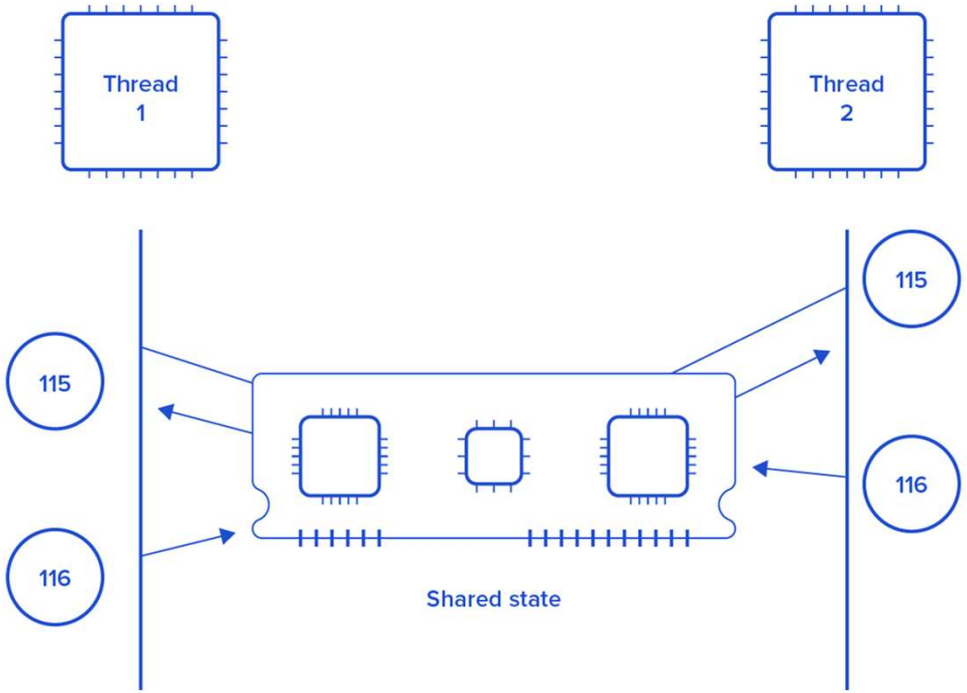
An access to the protected object can be:

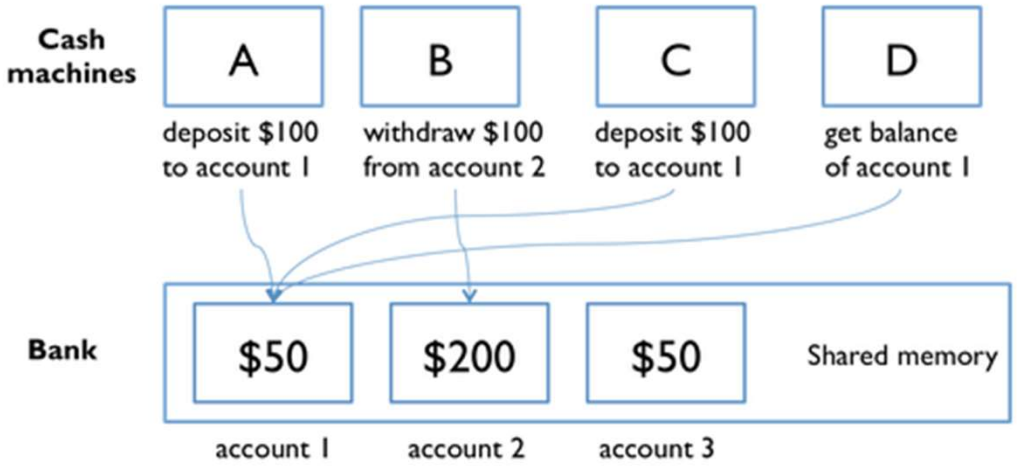
- Lock-free, when the task accessing the object is not blocked for a considerable time;
- Blocking, otherwise.

Blocking objects can be used for task synchronization. To the examples of such objects belong:

- Events;
- Mutexes and semaphores;
- Waitable timers;
- Queues

Issues Concurrent Programming Paradigm





Race Condition

```
import threading
x = 0     # A shared value
COUNT = 100

def incr():
    global x
    for i in range(COUNT):
        x += 1
        print(x)

def decr():
    global x
    for i in range(COUNT):
        x -= 1
        print(x)
```

```
t1 = threading.Thread(target=incr)
t2 = threading.Thread(target=decr)
t1.start()
t2.start()
t1.join()
t2.join()
print(x)
```

Synchronization in Python

Locks:

Locks are perhaps the simplest synchronization primitives in Python. A Lock has only two states — locked and unlocked (surprise). It is created in the unlocked state and has two principal methods — acquire() and release(). The acquire() method locks the Lock and blocks execution until the release() method in some other co-routine sets it to unlocked.

R-Locks:

R-Lock class is a version of simple locking that only blocks if the lock is held by another thread. While simple locks will block if the same thread attempts to acquire the same lock twice, a re-entrant lock only blocks if another thread currently holds the lock.

Semaphore:

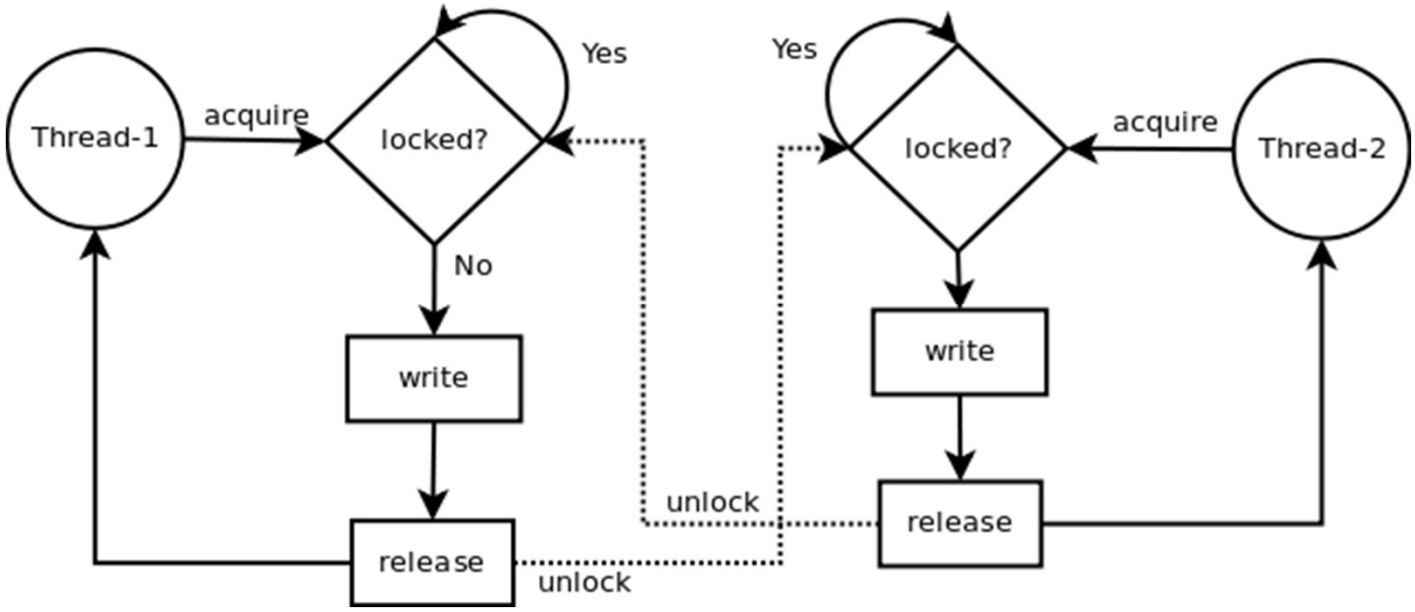
A semaphore has an internal counter rather than a lock flag, and it only blocks if more than a given number of threads have attempted to hold the semaphore. Depending on how the semaphore is initialized, this allows multiple threads to access the same code section simultaneously.

LOCK in python

Synchronization using LOCK

Locks have 2 states: locked and unlocked. 2 methods are used to manipulate them: acquire() and release(). Those are the rules:

1. if the state is unlocked: a call to acquire() changes the state to locked.
2. if the state is locked: a call to acquire() blocks until another thread calls release().
3. if the state is unlocked: a call to release() raises a RuntimeError exception.
4. if the state is locked: a call to release() changes the state to unlocked().



Synchronization in Python using Lock

```
import threading
x = 0     # A shared value
COUNT = 100
lock = threading.Lock()

def incr():
    global x
    lock.acquire()
    print("thread locked for increment cur x=",x)
    for i in range(COUNT):
        x += 1
        print(x)
    lock.release()
    print("thread release from increment cur x=",x)
def decr():
    global x
    lock.acquire()
    print("thread locked for decrement cur x=",x)
    for i in range(COUNT):
        x -= 1
        print(x)
    lock.release()
    print("thread release from decrement cur x=",x)
t1 = threading.Thread(target=incr)
t2 = threading.Thread(target=decr)
t1.start()
t2.start()
t1.join()
t2.join()
```

Synchronization in Python using RLock

```
import threading

class Foo(object):
    lock = threading.RLock()
    def __init__(self):
        self.x = 0
    def add(self,n):
        with Foo.lock:
            self.x += n
    def incr(self):
        with Foo.lock:
            self.add(1)
    def decr(self):
        with Foo.lock:
            self.add(-1)


def adder(f,count):
    while count > 0:
        f.incr()
        count -= 1

def subber(f,count):
    while count > 0:
        f.decr()
        count -= 1

# Create some threads and make sure it works
COUNT = 10
f = Foo()
t1 = threading.Thread(target=adder,args=(f,COUNT))
t2 = threading.Thread(target=subber,args=(f,COUNT))
t1.start()
t2.start()
t1.join()
t2.join()
print(f.x)
```

Synchronization in Python using Semaphore

```
import threading
import time

done = threading.Semaphore(0)
item = None

def producer():
    global item
    print "I'm the producer and I produce data."
    print "Producer is going to sleep."
    time.sleep(10)
    item = "Hello"
    print "Producer is alive. Signaling the consumer."
    done.release()

def consumer():
    print "I'm a consumer and I wait for data."
    print "Consumer is waiting."
    done.acquire()
    print "Consumer got", item

t1 = threading.Thread(target=producer)
t2 = threading.Thread(target=consumer)
t1.start()
t2.start()
```

Synchronization in Python using event

```
import threading
import time
item = None
# A semaphore to indicate that an item is available
available = threading.Semaphore(0)
# An event to indicate that processing is complete
completed = threading.Event()
# A worker thread
def worker():
    while True:
        available.acquire()
        print "worker: processing", item
        time.sleep(5)
        print "worker: done"
        completed.set()
# A producer thread
def producer():
    global item
    for x in range(5):
        completed.clear()       # Clear the event
        item = x                # Set the item
        print "producer: produced an item"
        available.release()     # Signal on the semaphore
        completed.wait()
        print "producer: item was processed"
t1 = threading.Thread(target=producer)
t1.start()
t2 = threading.Thread(target=worker)
t2.setDaemon(True)
t2.start()
```

Producer and Consumer problem using thread

```
import threading,time,Queue
items = Queue.Queue()
# A producer thread
def producer():
    print "I'm the producer"
    for i in range(30):
        items.put(i)
        time.sleep(1)
# A consumer thread
def consumer():
    print "I'm a consumer", threading.currentThread().name
    while True:
        x = items.get()
        print threading.currentThread().name,"got", x
        time.sleep(5)
# Launch a bunch of consumers
cons = [threading.Thread(target=consumer)
        for i in range(10)]
for c in cons:
    c.setDaemon(True)
    c.start()
# Run the producer
producer()
```

Producer and Consumer problem using thread

```
import threading
import time
# A list of items that are being produced.  Note: it is actually
# more efficient to use a collections.deque() object for this.
items = []
# A condition variable for items
items_cv = threading.Condition()
def producer():
    print "I'm the producer"
    for i in range(30):
        with items_cv:          # Always must acquire the lock first
            items.append(i)     # Add an item to the list
            items_cv.notify()   # Send a notification signal
        time.sleep(1)
def consumer():
    print "I'm a consumer", threading.currentThread().name
    while True:
        with items_cv:          # Must always acquire the lock
            while not items:    # Check if there are any items
                items_cv.wait()  # If not, we have to sleep
            x = items.pop(0)    # Pop an item off
        print threading.currentThread().name,"got", x
        time.sleep(5)
cons = [threading.Thread(target=consumer)
        for i in range(10)]
for c in cons:
    c.setDaemon(True)
    c.start()
producer()
```