18CSC207J – Advanced Programming Practice

C.Arun, Asst. Prof. Dept of Software Engineering, School of Computing, SRMIST

Imperative
Programming Paradigm

Topics

- Program state, instructions to change the program state
- Combining Algorithms and Data Structures
- Imperative Vs Declarative Programming
- Other Languages: PHP, Ruby, Perl, Swift
- Demo: Imperative Programming in Python

INTRODUCTION

- In a computer program,a **variable** stores the data. The contents of these locations at any givenpoint 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 **machinelanguages.**

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.
- For example:
  - – *rs*, *rt*, and *rd* indicate register operands
  - – *shamt* gives a shift amount
  - – the *address* or *immediate* fields contain an operand directly

So, adding the registers 1 and 2 and placing the result in register 6 is encoded:

[op | rs | rt | rd |shamt| funct]

0 1 2 6 0 32 decimal

000000 00001 00010 00110 00000 100000 binary

Assembly Code

- 10110000 01100001
- **Equivalent Assembly code**
- B0 61
  - *B0 - 'Move a copy of the following valueinto AL' (AL is a register)*
  - *61 is a hexadecimal representation of the value01100001 – 97*
- **Intel assembly language**
- MOV AL, 61h; Load AL with 97 decimal (61 hex)

Other Languages

- **FORTRAN**(FORmula TRANslation) was the first high level language to gain wide acceptance. It was designed for **scientific applications** and featured an algebraic notation, types, subprograms, and formatted input/output.
- **COBOL** (COmmon Business Oriented Language) was designed at the initiative of the U. S. Department of Defence in 1959 and implemented in 1960 to meet the need for **business data processing applications.**
- **ALGOL 60** (ALGorithmic Oriented Language) was designed in 1960 by an international committee for use in **scientific problem solving**

Evolutionary developments

Algol to PL /I	PL / I to FORTRAN	COBOL to PL/I	LISP to PL/I
• Block structure • Control statements • Recursion	• Subprograms • Formatted IO	• File manipulation • Record	• Dynamic storage allocation • Linked structures

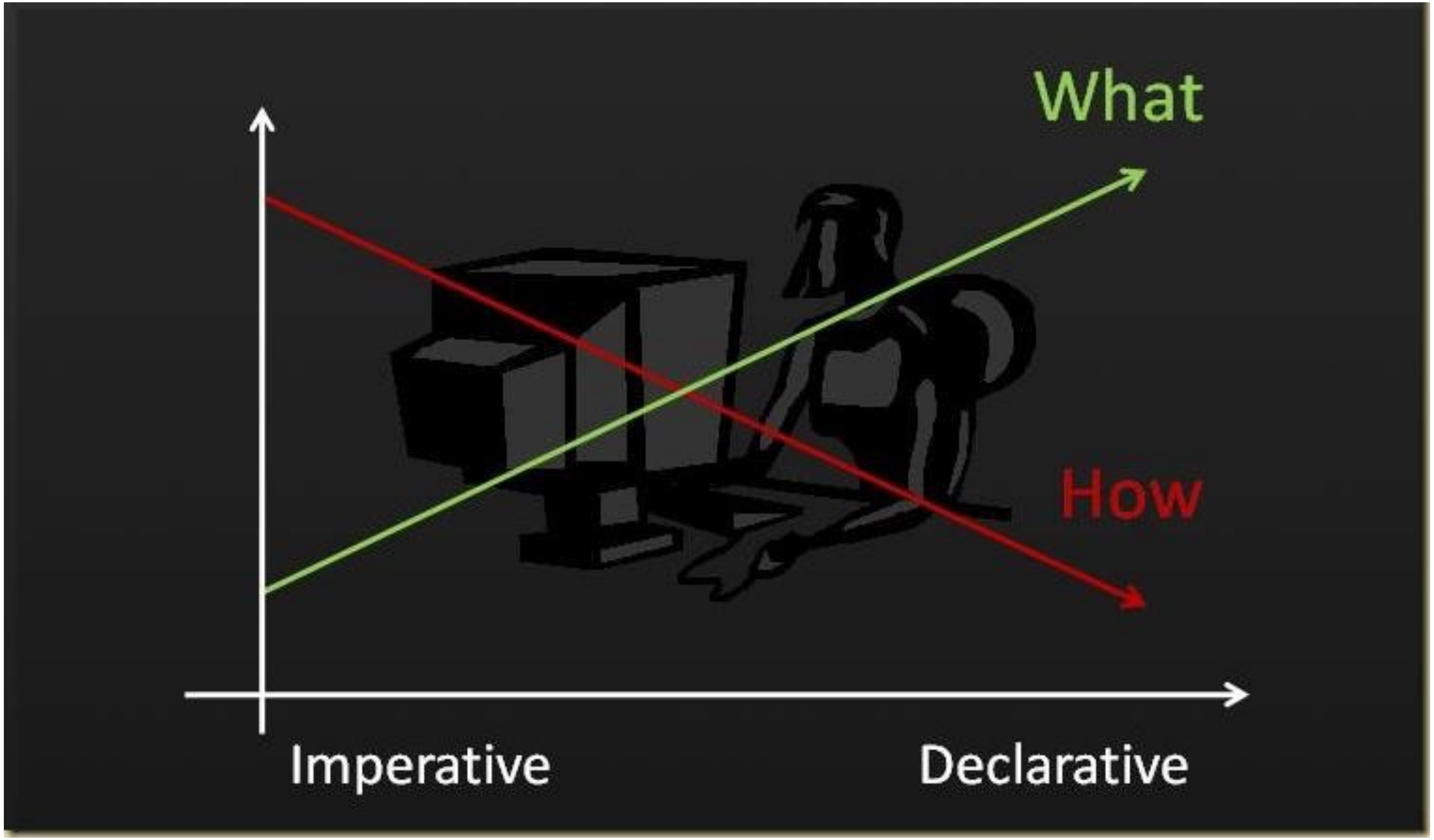
OVERVIEW

- In imperative programming, a name may be assigned to a value and later reassigned to another value.
- The collection of names and the associated values and the location of control in the program constitute the state.
- The state is a logical model of storage whichis an association between memory locations and values.
- A program in execution generates a sequence of states.
- The transition from one state to the next is determined by assignment operations and sequencing commands.

Highlights on

- Assignment,
- goto commands
- structured programming
- Command
- Statement
- Procedure
- Control-flow
- Imperative language
- Assertions
- Axiomatic semantics
- State
- Variables
- Instructions
- Control structures

Declarative Vs Imperative



Declarative Vs Imperative

Declarative

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

Imperative

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

```
0
1
2
3
4
```

DEMO

An algorithm to add two numbers entered by user

Step 1: Start

Step 2: Declare variables num1, num2 and sum.

Step 3: Read values num1 and num2.

Step 4: Add num1 and num2 and assign the result to sum. sum←num1+num2

Step 5: Display sum

Step 6: Stop

Addition two numbers entered by user

```
num1 = 0
num2 = 0
sum = 0
num1 = input("Enter the First number ")
num2 = input("Enter the Second number ")
sum = int(num1) + int(num2)
print("\nSum:", sum)
```

```
Enter the First number 10

Enter the Second number 20

Sum: 30
```

An Algorithm to Get n number, print the same and find Sum of n numbers

Step 1: Start

Step 2: Declare variable sum = 0.

Step 3: Get the value of limit “n”.

Step 4: If limit is reached, goto Step 7 else goto Step 5

Step 5: Get the number from user and add it to sum

Step 6: Goto Step 4

Step 7: If limit is reached, goto Step 9 else goto Step 8

Step 8: Print the numbers

Step 9: Goto Step 7

Step 9: Display sum

Step 10: Stop

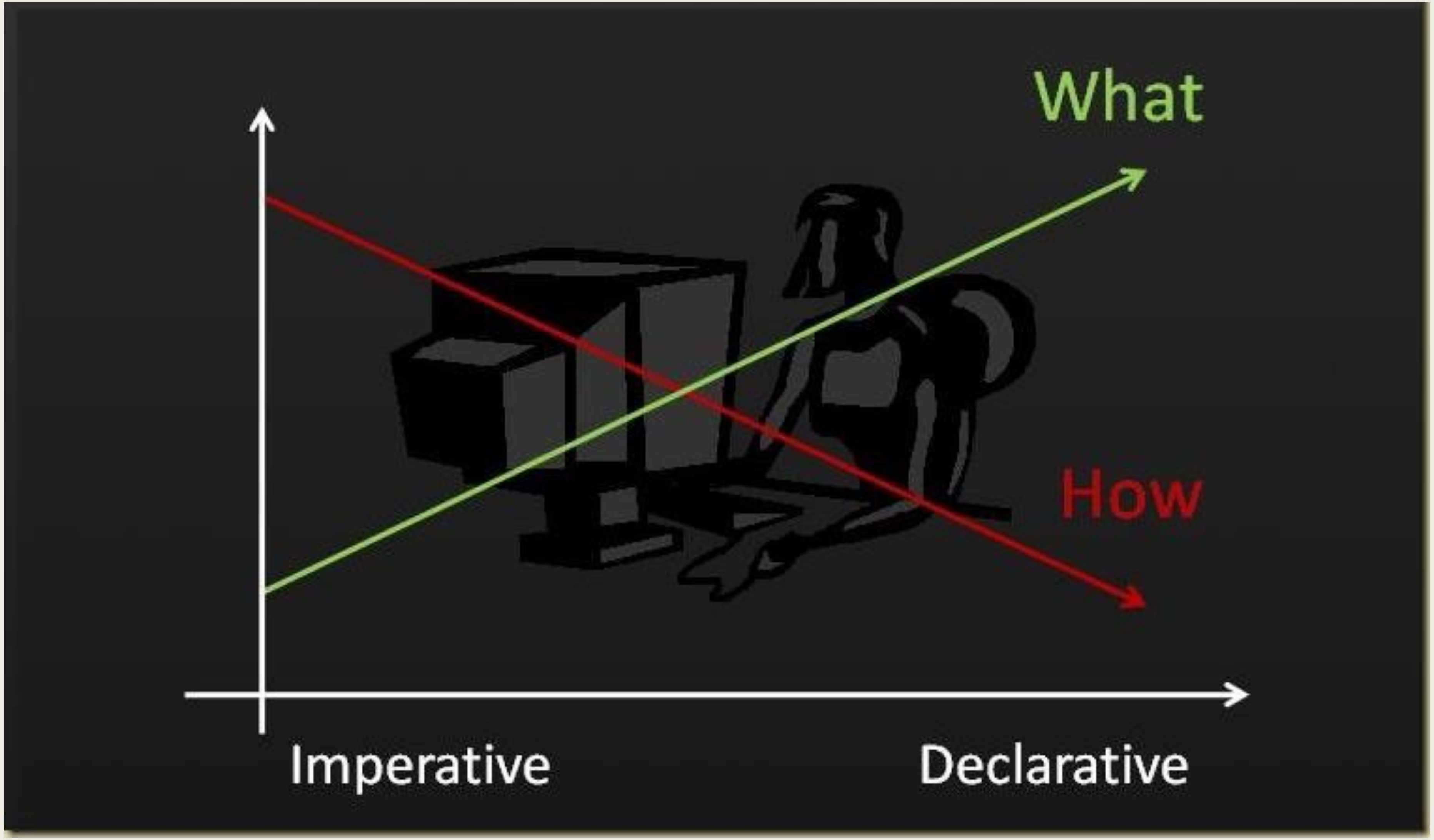
```python
sum = 0
num=[]
n = input("Enter the Total number of values ")
num = [ int(input("Enter value ")) for i in range(int(n))]
for i in range(int(n)):
    sum = sum + num[i]
print("\nYou have entered")
for i in range(int(n)):
    print(num[i])
print("..and the sum is", sum)
```

```
Enter the Total number of values 5

Enter value 55

Enter value 62

Enter value 12

Enter value 34

Enter value 20

You have entered
55
62
12
34
20
..and the sum is 183
```

```python
Dict = {1: 'Song A', 2: 'Song B', 3: 'Song C'}

n=input("Select the number to play your favorite song ")
# accessing a element using key
print("You are listening to ")
print(Dict[int(n)])
```

```
Select the number to play your favorite song 3
You are listening to
Song C
```

Declarative Vs Imperative



Declarative Vs Imperative

Declarative

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

Imperative

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

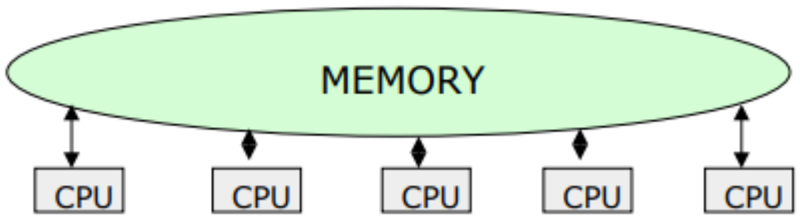
```
0
1
2
3
4
```

Parallel & Concurrent Programming Paradigm

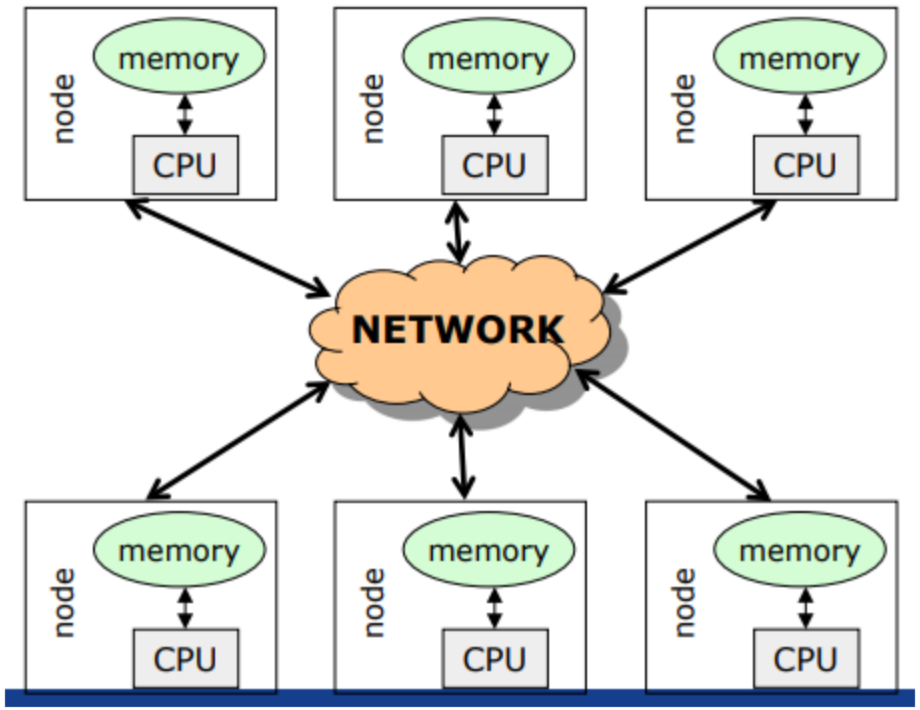
Introduction

- 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).

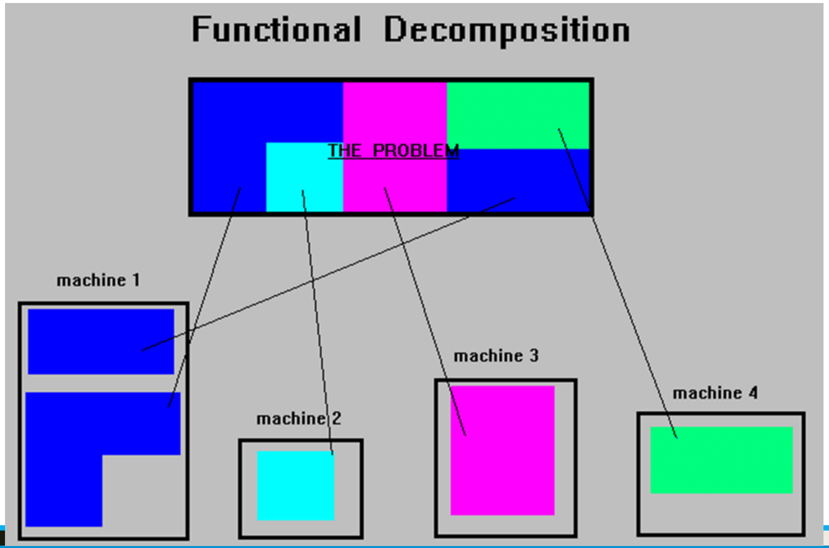
Ways for Parallelism

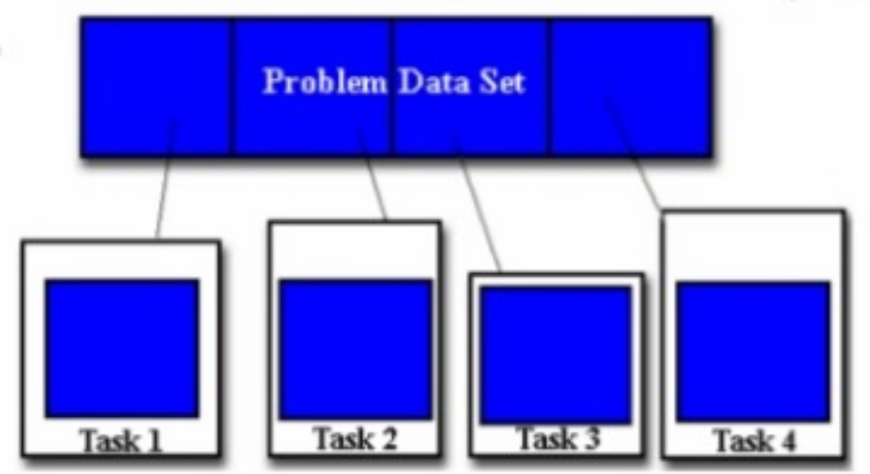
Functional Decomposition (Functional Parallelism)

- 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

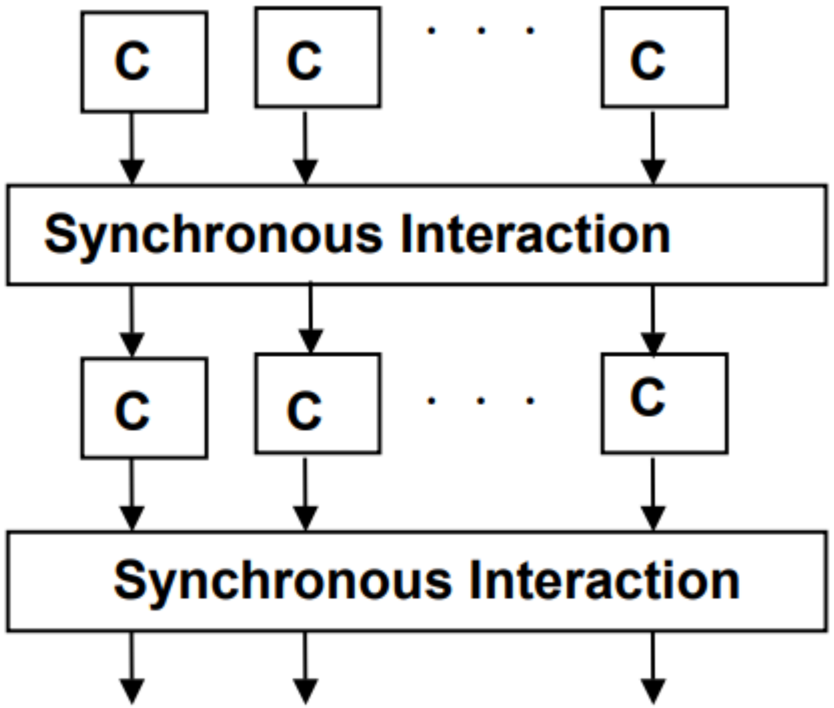
- Phase parallel
- Divide and conquer
- Pipeline
- Process farm
- Work pool

Note:

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

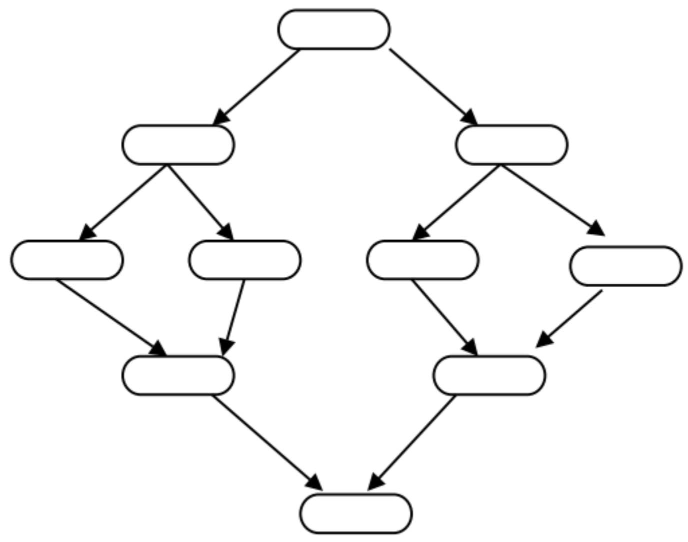
Phase Parallel Model

- 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.



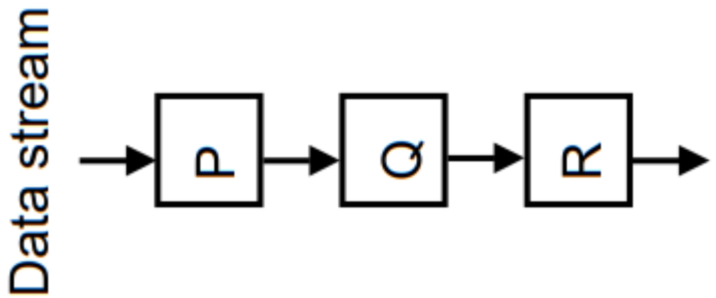
Divide and Conquer & Pipeline model

- 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.



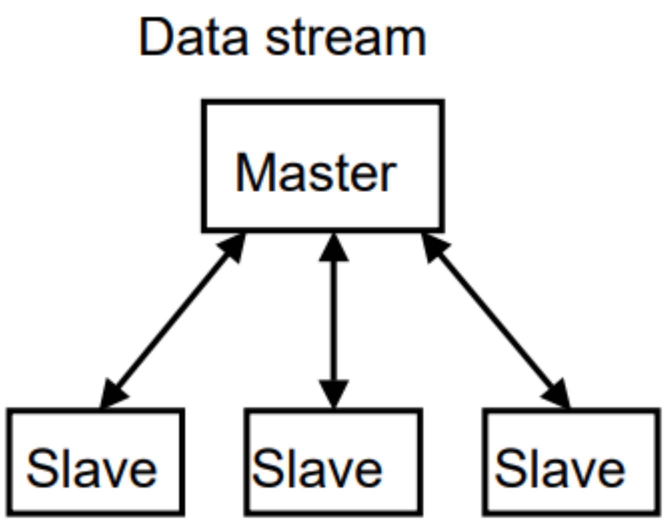
Pipeline

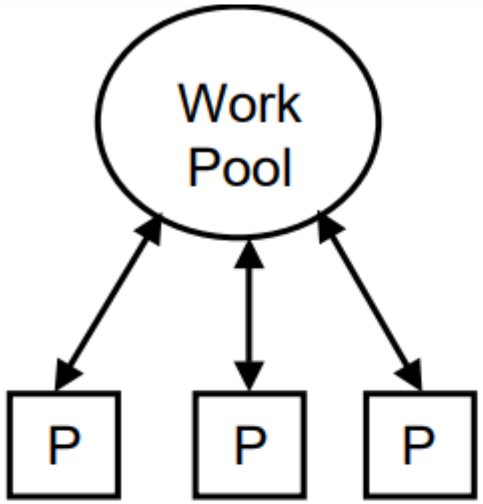
- In pipeline paradigm, a number of processes form a virtual pipeline.
- A continuous data stream is fed into the pipeline, and the processes execute at different pipeline stages simultaneously in an overlapped fashion.



Process Farm & Work Pool Model

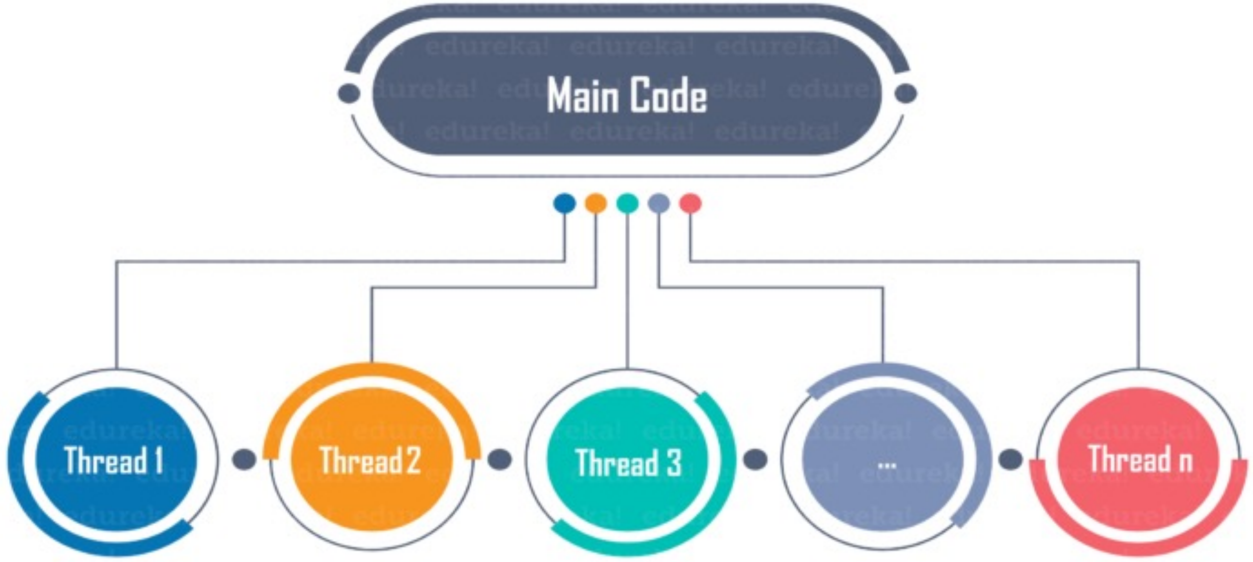
- This paradigm is also known as the master-slave paradigm.
- A master process executes the essentially sequential part of the parallel program and spawns a number of slave processes to execute the parallel workload.
- When a slave finishes its workload, it informs the master which assigns a new workload 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

- Multitasking, in general, is the capability of performing multiple tasks simultaneously. Multithreading refers to concurrently executing multiple threads by rapidly switching the control of the CPU between threads (called context switching).
- Python Global Interpreter Lock limits one thread to run at a time even if the machine contains multiple processors.
- There are two types of multitasking in an OS:
  1. Process-based
  2. Thread-based
- Threads in Python can be created in three ways:
  - Without creating a class
  - By extending Thread class
  - Without extending Thread class



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 Python

Model 1

Without creating a class

step1 : Import modules

step2 : Define the role of thread

step3 : Instantiate the thread by specifying target

step4 : Start the Thread.

```
from threading import *
def fun():
        .......
child = Thread(target = fun)
child.start()
```

Model 2:

By Extending thread class

step1 : import Modules

Step2 : Extending (inherit) a class from Thread

step3 : Override Run method

step4 : Instantitate the obj for extended class

step5 : call the start method

```
import threading
class test(Thread):
        def run(self):
                print("User threads")

ob=test()
ob.start()
```

Model 3

Without Extending thread class:

step1 : create class

step2 : define method

step3 : Create object for the class

step4 : Instantiate the Thread by passing target method

step5 : class Start method

```
from threading import *
class Test:
    def func(self):
        print("Child Thread")

ob=Test()
child = Thread(target= ob.func)
child.start()
```

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.

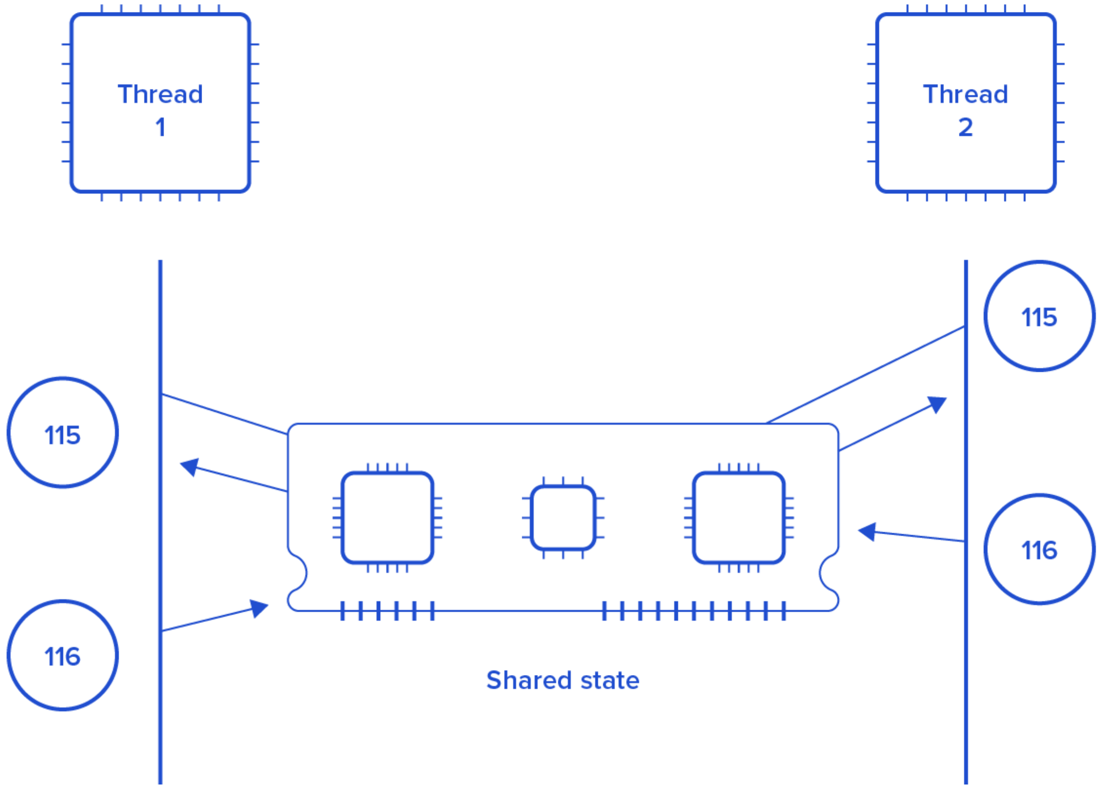
An access to the protected object can be:

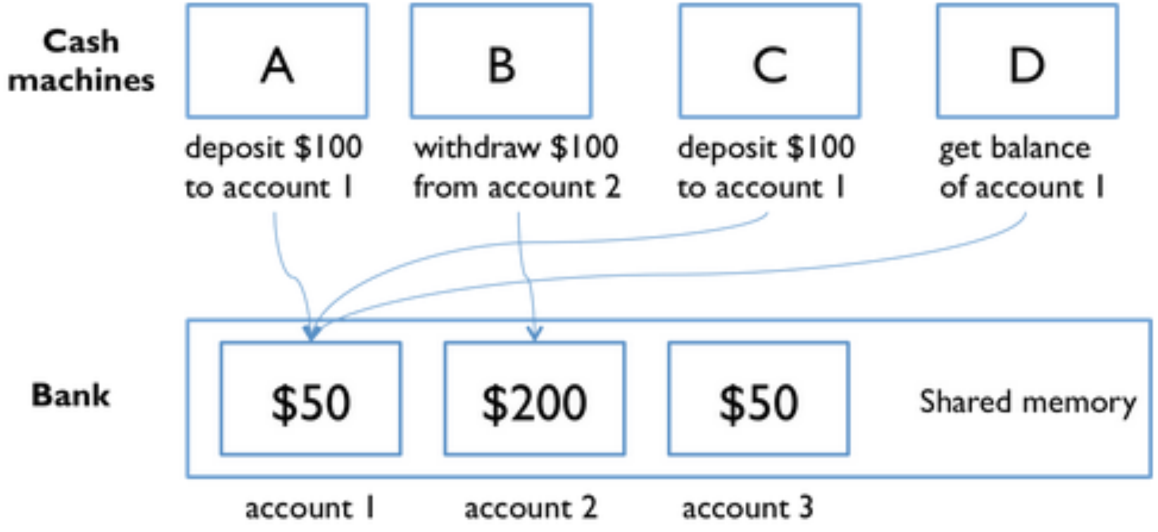
- Lock-free, when the task accessing the object is not blocked for a considerable time;
- Blocking, otherwise.

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:

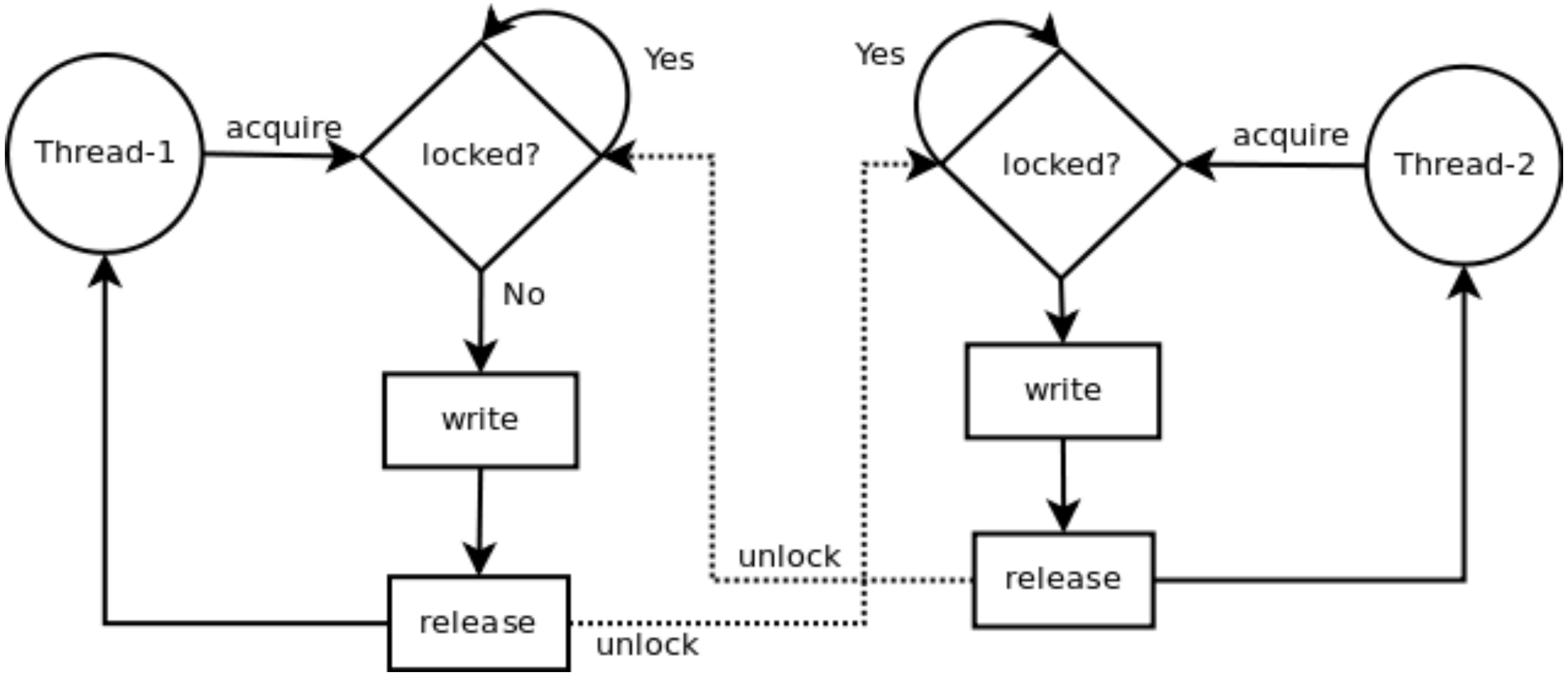
A semaphore has an internal counter rather than a lock flag, and it only blocks if more than a given number of threads have attempted to hold the semaphore. Depending on how the semaphore is initialized, this allows multiple threads to access the same code section simultaneously.

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()
```