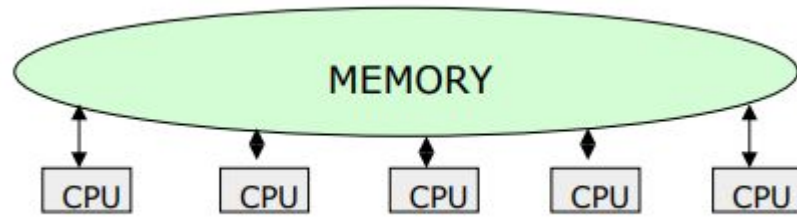# 18CSC207J – Advanced Programming Practice

# 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

**Message Passing Architecture**

**Shared memory Architecture**
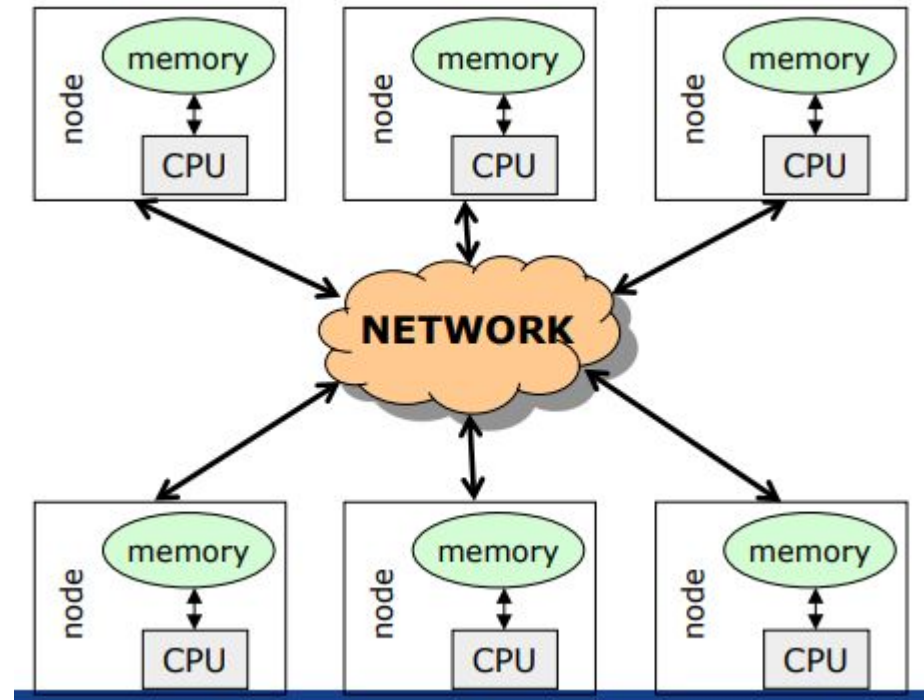
# Explicit parallelism

- 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)
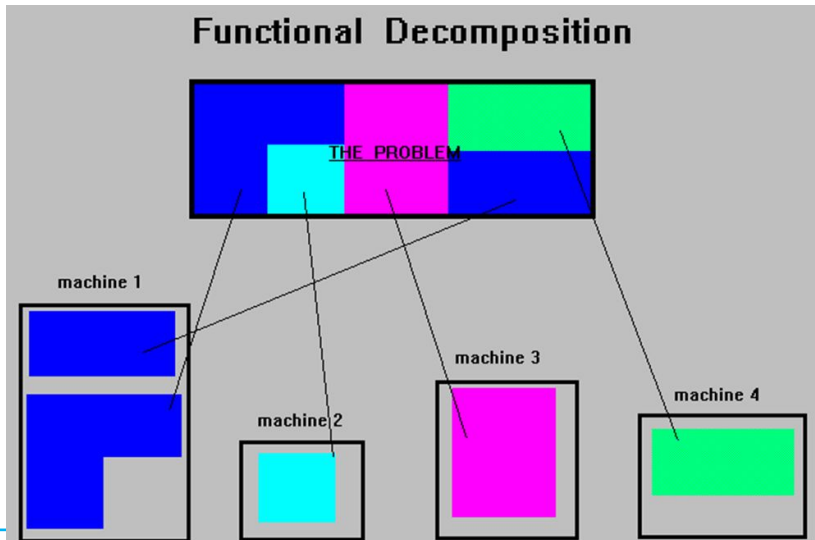
# Parallel Programming Paradigm

- 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.
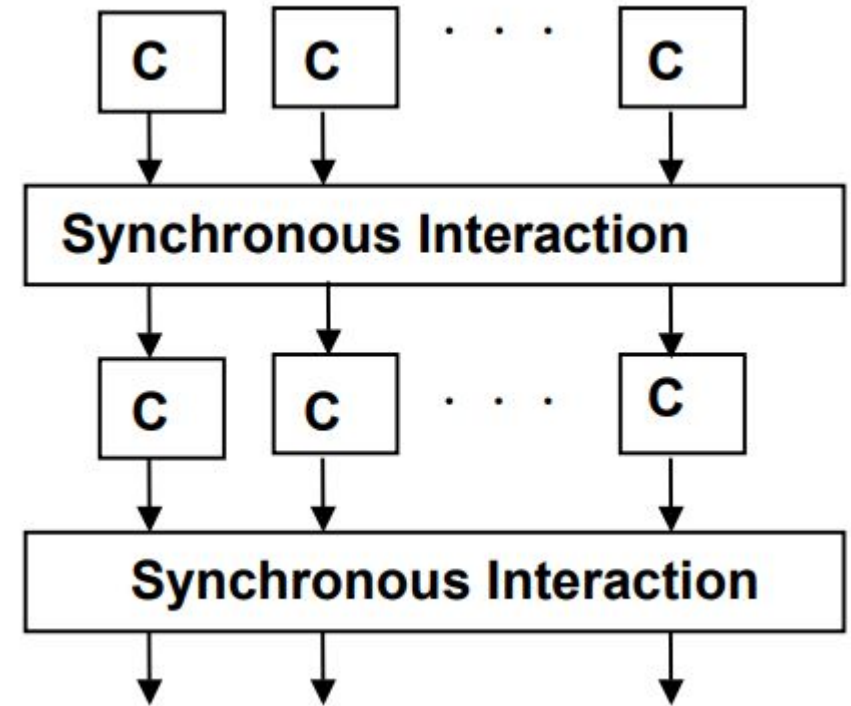
# Divide and Conquer & Pipeline model

- 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.
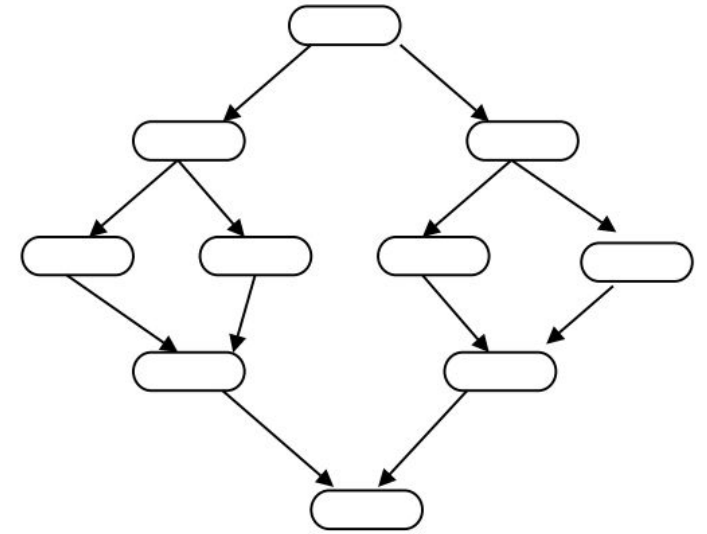
**Pipeline**

- 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.
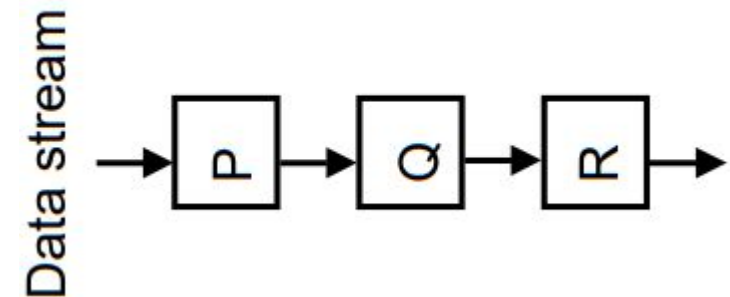
# Process Farm & Work Pool Model

- 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.

Data stream

Master

Slave    Slave    Slave

Work Pool

P    P    P
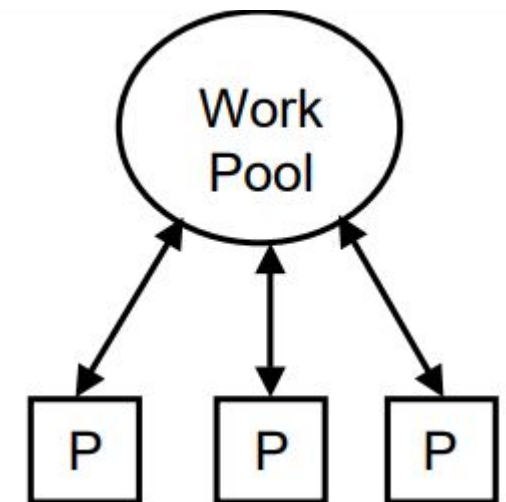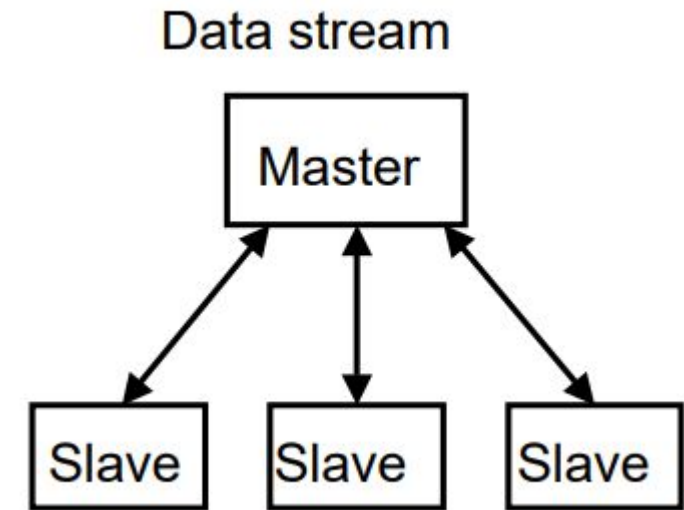
# Parallel Program using Python

- 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

```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())
```

```python
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

```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

Thread 1

Thread 2

115

115

116

116

Shared state

Cash machines

| A | B | C | D |
|---|---|---|---|
| deposit $100 to account 1 | withdraw $100 from account 2 | deposit $100 to account 1 | get balance of account 1 |

Bank

| $50 | $200 | $50 | Shared memory |
|---|---|---|---|
| account 1 | account 2 | account 3 | |
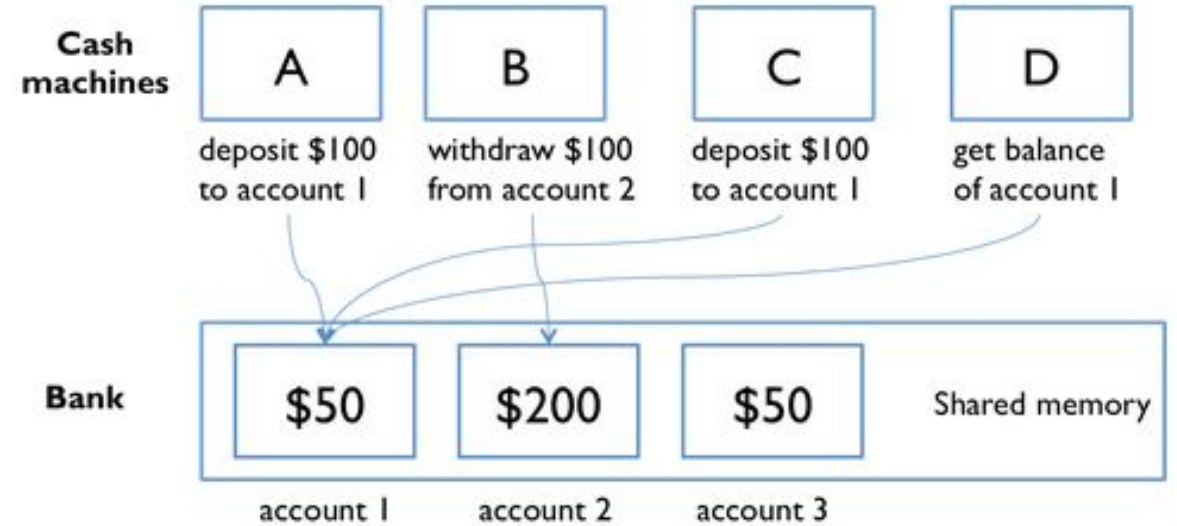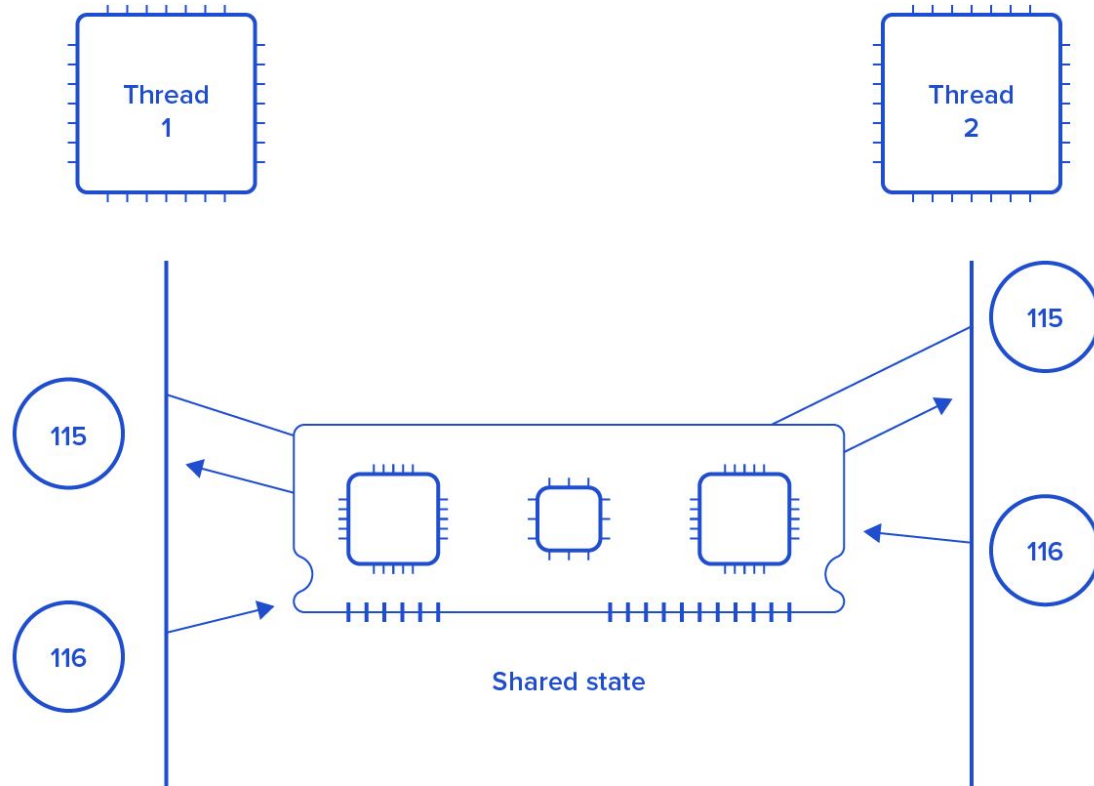
# Race Condition

```python
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().
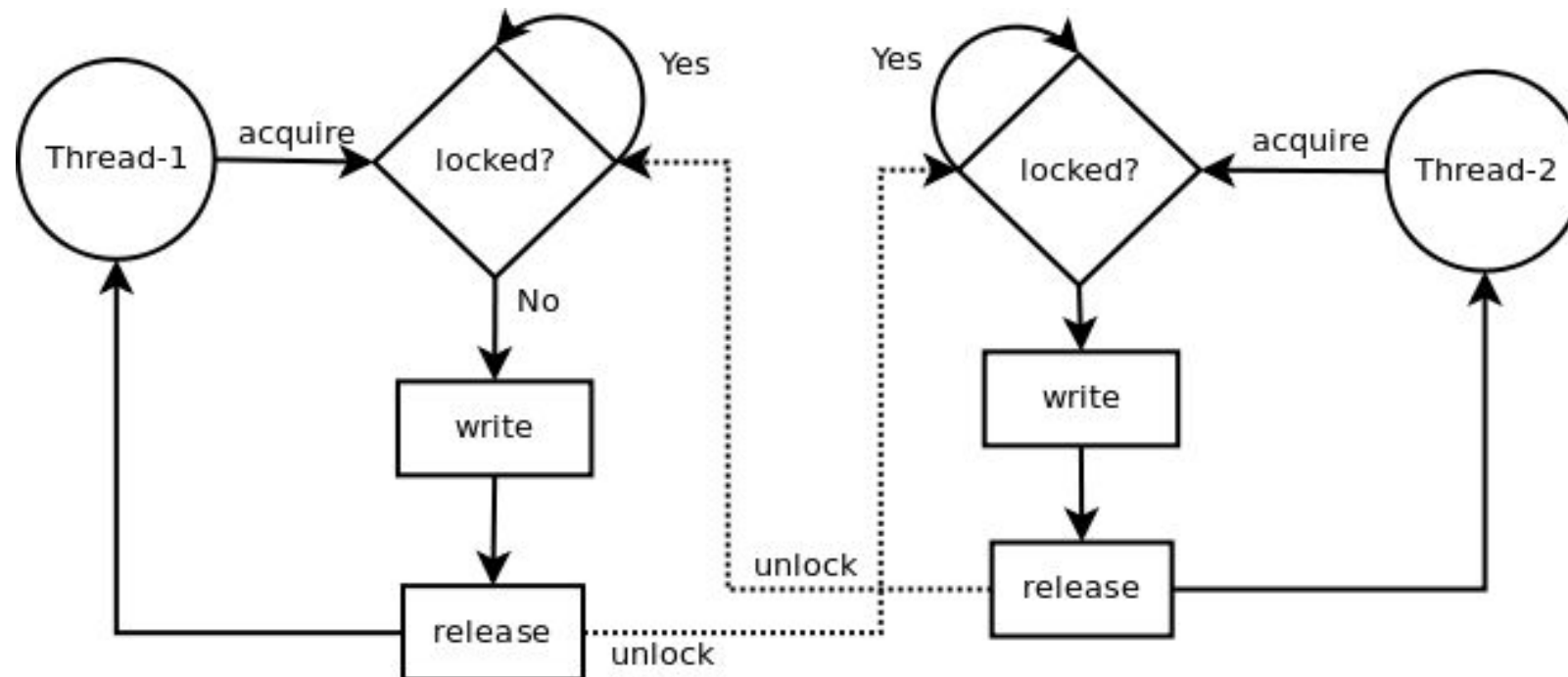
# Synchronization in Python using Lock

```python
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

```python
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

```python
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

```python
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

```python
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

```python
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()
```

# Network Programming Paradigm

# Introduction

The Network paradigm involves thinking of computing in terms of a client, who is essentially in need of some type of information, and a server, who has lots of information and is just waiting to hand it out. Typically, a client will connect to a server and query for certain information. The server will go off and find the information and then return it to the client.

In the context of the Internet, clients are typically run on desktop or laptop computers attached to the Internet looking for information, whereas servers are typically run on larger computers with certain types of information available for the clients to retrieve. The Web itself is made up of a bunch of computers that act as Web servers; they have vast amounts of HTML pages and related data available for people to retrieve and browse. Web clients are used by those of us who connect to the Web servers and browse through the Web pages.

Network programming uses a particular type of network communication known as sockets. A socket is a software abstraction for an input or output medium of communication.

# What is Socket?

- A socket is a software abstraction for an input or output medium of communication.

- Sockets allow communication between processes that lie on the same machine, or on different machines working in diverse environment and even across different continents.

- A socket is the most vital and fundamental entity. Sockets are the end-point of a two-way communication link.

- An endpoint is a combination of IP address and the port number.

For Client-Server communication,

- Sockets are to be configured at the two ends to initiate a connection,

- Listen for incoming messages

- Send the responses at both ends

- Establishing a bidirectional communication.

# Socket Types

**Datagram Socket**

- A datagram is an independent, self-contained piece of information sent over a network whose arrival, arrival time, and content are not guaranteed. A datagram socket uses User Datagram Protocol (UDP) to facilitate the sending of datagrams (self-contained pieces of information) in an unreliable manner. Unreliable means that information sent via datagrams isn't guaranteed to make it to its destination.

**Stream Socket:**

- A stream socket, or connected socket, is a socket through which data can be transmitted continuously. A stream socket is more akin to a live network, in which the communication link is continuously active. A stream socket is a "connected" socket through which data is transferred continuously.

# Socket in Python

    sock_obj = socket.socket( socket_family, socket_type, protocol=0)

**socket_family:** - Defines  family of protocols used as transport mechanism.

    Either AF_UNIX, or

    AF_INET (IP version 4 or IPv4).

**socket_type:** Defines the types of communication between the two end-points.

    SOCK_STREAM (for connection-oriented protocols, e.g., TCP), or

    SOCK_DGRAM (for connectionless protocols e.g. UDP).

**protocol**: We typically leave this field or set this field to zero.

**Example:**

```
#Socket client example in python

import socket

#create an AF_INET, STREAM socket (TCP)

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

print 'Socket Created'
```
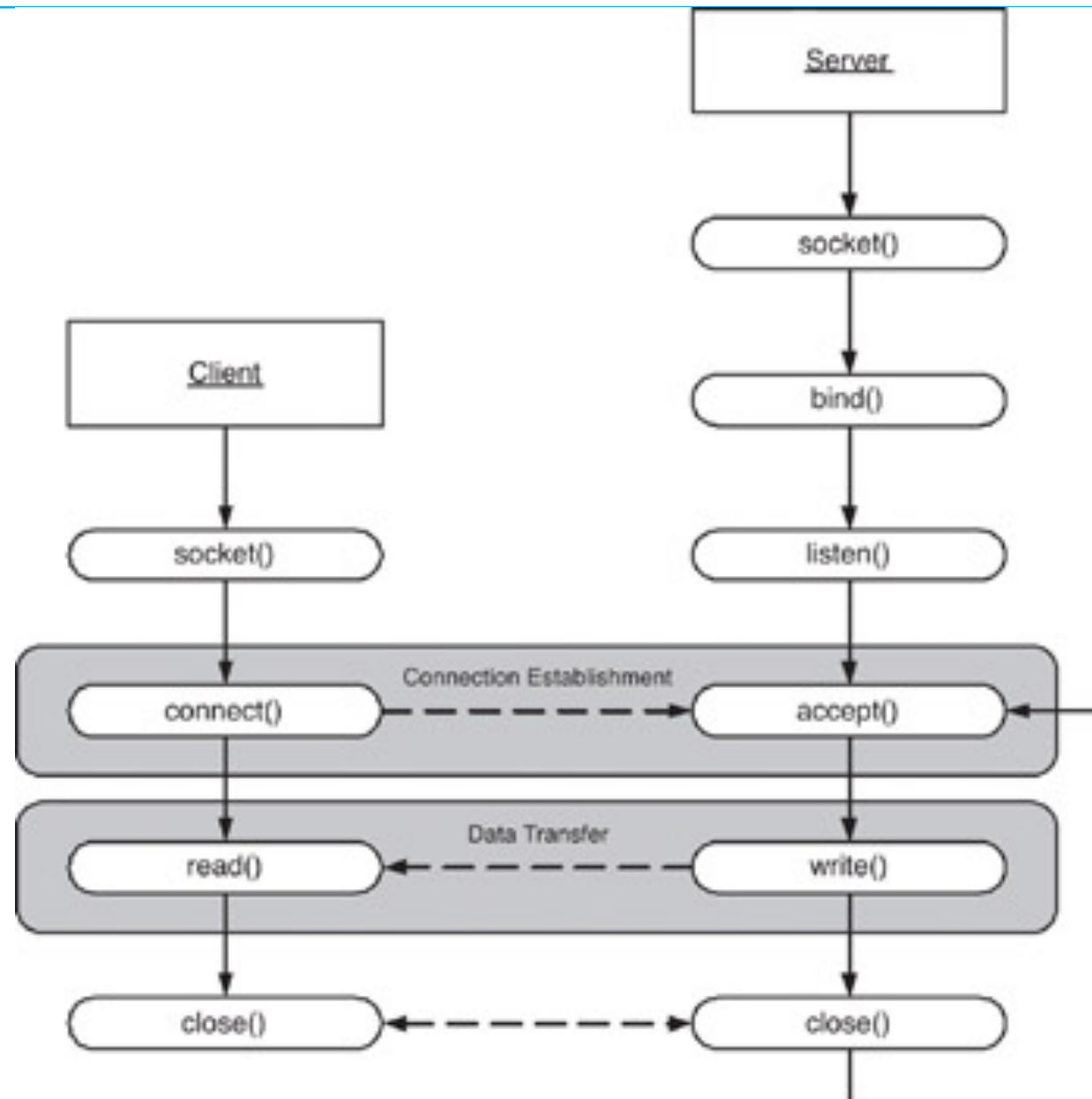
# Socket Creation

```python
import socket

import sys

try:

    #create an AF_INET, STREAM socket (TCP)

    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

except socket.error, msg:

    print 'Failed to create socket. Error code: ' + str(msg[0]) + ' , Error message : ' + msg[1]

    sys.exit();


print 'Socket Created'
```

# Client/server symmetry in Sockets applications

# Socket in Python

To create a socket, we must use socket.socket() function available in the Python socket module, which has the general syntax as follows:

> ***S = socket.socket(socket_family, socket_type, protocol=0)***

socket_family: This is either AF_UNIX or AF_INET. We are only going to talk about INET sockets in this tutorial, as they account for at least 99% of the sockets in use.

socket_type: This is either SOCK_STREAM or SOCK_DGRAM.

Protocol: This is usually left out, defaulting to 0.

**Client Socket Methods**

Following are some client socket methods:

connect( ) : To connect to a remote socket at an address. An address format(host, port) pair is used for AF_INET address family.

# Socket in Python

**Server Socket Methods**

bind( ):  This method binds the socket to an address. The format of address depends on socket family mentioned above(AF_INET).

listen(backlog) : This method listens for the connection made to the socket. The backlog is the maximum number of queued connections that must be listened before rejecting the connection.

accept( ) : This method is used to accept a connection. The socket must be bound to an address and listening for connections. The return value is a pair(conn, address) where conn is a new socket object which can be used to send and receive data on that connection, and address is the address bound to the socket on the other end of the connection.

# General Socket in Python

sock_object.recv():

Use this method to receive messages at endpoints when the value of the protocol parameter is TCP.

sock_object.send():

Apply this method to send messages from endpoints in case the protocol is TCP.

sock_object.recvfrom():

Call this method to receive messages at endpoints if the protocol used is UDP.

sock_object.sendto():

Invoke this method to send messages from endpoints if the protocol parameter is UDP.

sock_object.gethostname():

This method returns hostname.

sock_object.close():

This method is used to close the socket. The remote endpoint will not receive data from this side.

# Simple TCP Server

```python
#!/usr/bin/python

#This is tcp_server.py script

import socket                              #line 1: Import socket module

s = socket.socket()                        #line 2: create a socket object
host = socket.gethostname()                #line 3: Get current machine name
port = 9999                                #line 4: Get port number for connection

s.bind((host,port))                        #line 5: bind with the address

print "Waiting for connection..."
s.listen(5)                                #line 6: listen for connections

while True:
    conn,addr = s.accept()                 #line 7: connect and accept from client
    print 'Got Connection from', addr
    conn.send('Server Saying Hi')
    conn.close()                           #line 8: Close the connection
```

# Simple TCP Client

```python
#!/usr/bin/python

#This is tcp_client.py script

import socket

s = socket.socket()
host = socket.gethostname()      # Get current machine name
port = 9999                      # Client wants to connect to server's
                                 # port number 9999

s.connect((host,port))
print s.recv(1024)               # 1024 is bufsize or max amount
                                 # of data to be received at once

s.close()
```

# Simple UDP Server

```python
#!usr/bin/python

import socket

sock = socket.socket(socket.AF_INET,socket.SOCK_DGRAM)        # For UDP

udp_host = socket.gethostname()              # Host IP
udp_port = 12345                             # specified port to connect

#print type(sock) ============> 'type' can be used to see type
                # of any variable ('sock' here)


sock.bind((udp_host,udp_port))

while True:
    print "Waiting for client..."
    data,addr = sock.recvfrom(1024)          #receive data from client
    print "Received Messages:",data," from",addr
```

# Simple UDP Client

```python
#!usr/bin/python

import socket

sock = socket.socket(socket.AF_INET,socket.SOCK_DGRAM)        # For UDP

udp_host = socket.gethostname()        # Host IP
udp_port = 12345                       # specified port to connect

msg = "Hello Python!"
print "UDP target IP:", udp_host
print "UDP target Port:", udp_port


sock.sendto(msg,(udp_host,udp_port))        # Sending message to UDP server
```