Assignment No. : A4

1. TITLE

In an embedded system application Dining Philosopher's problem algorithm is used to design a software that uses shared memory between neighboring processes to consume the data. The Data is generated by different Sensors/WSN system Network and stored in MOngoDB (NoSQL). Implementation be done using Scala/ Python/ C++/ Java. Design using Client-Server architecture. Perform Reliability Testing. Use latest open source software modeling, Designing and testing tool/Scrum-it/KADOS, NoSQLUnit and Camel.

2. PREREQUISITES

- 64-bit Fedora or equivalent OS with 64-bit Intel-i5/i7
- Python

3. OBJECTIVE

- Solve the Dining philosopher's problem using Python.

4. THEORY

Dining philosopher's problem

The problem was designed to illustrate the challenges of avoiding deadlock, a system state in which no progress is possible. To see that a proper solution to this problem is not obvious, consider a proposal in which each philosopher is instructed to behave as follows:

- think until the left fork is available; when it is, pick it up;
- think until the right fork is available; when it is, pick it up;
- when both forks are held, eat for a fixed amount of time;
- then, put the right fork down;
- then, put the left fork down; repeat from the beginning.

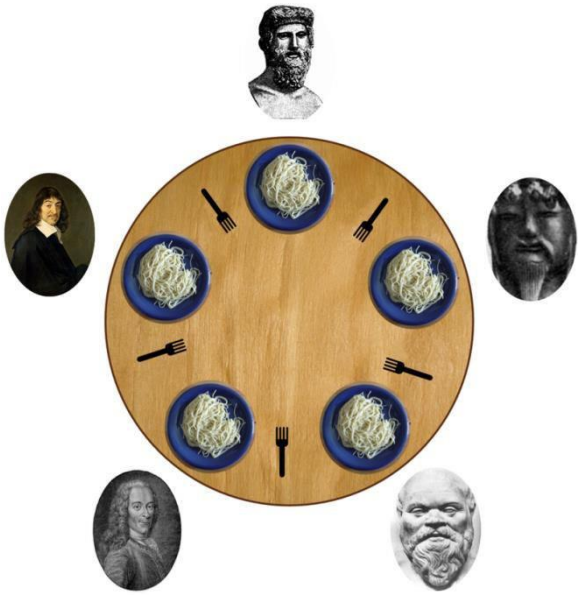

Fig. Dining philosopher's problem

This attempted solution fails because it allows the system to reach a deadlock state, in which no progress is possible. This is a state in which each philosopher has picked up the fork to the left, and is waiting for the fork to the right to become available. With the given instructions, this state can be reached, and when it is reached, the philosophers will eternally wait for each other to release a fork.

Resource starvation might also occur independently of deadlock if a particular philosopher is unable to acquire both forks because of a timing problem. For example there might be a rule that the philosophers put down a fork after waiting ten minutes for the other fork to become available and wait a further ten minutes before making their next attempt. This scheme eliminates the possibility of deadlock (the system can always advance to a different state) but still suffers from the problem of livelock. If all five philosophers appear in the dining room at exactly the same time and each picks up the left fork at the same time the philosophers will wait ten minutes until they all put their forks down and then wait a further ten minutes before they all pick them up again.

Mutual exclusion is the basic idea of the problem; the dining philosophers create a generic and abstract scenario useful for explaining issues of this type. The failures these philosophers may experience are analogous to the difficulties that arise in real computer programming when multiple programs need exclusive access to shared resources. These issues are studied in the branch of concurrent programming. The original problems of Dijkstra were related to external devices like tape drives. However, the difficulties exemplified by the dining philosophers problem arise far more often when multiple processes access sets of data that are being updated. Systems such as operating system kernels use thousands of locks and synchronizations that require strict adherence

to methods and protocols if such problems as deadlock, starvation, or data corruption are to be avoided.

Resource hierarchy solution

This solution to the problem is the one originally proposed by Dijkstra. It assigns a partial order to the resources (the forks, in this case), and establishes the convention that all resources will be requested in order, and that no two resources unrelated by order will ever be used by a single unit of work at the same time. Here, the resources (forks) will be numbered 1 through 5 and each unit of work (philosopher) will always pick up the lower-numbered fork first, and then the higher-numbered fork, from among the two forks he plans to use. The order in which each philosopher puts down the forks does not matter. In this case, if four of the five philosophers simultaneously pick up their lower-numbered fork, only the highest numbered fork will remain on the table, so the fifth philosopher will not be able to pick up any fork. Moreover, only one philosopher will have access to that highest-numbered fork, so he will be able to eat using two forks.

While the resource hierarchy solution avoids deadlocks, it is not always practical, especially when the list of required resources is not completely known in advance. For example, if a unit of work holds resources 3 and 5 and then determines it needs resource 2, it must release 5, then 3 before acquiring 2, and then it must re-acquire 3 and 5 in that order. Computer programs that access large numbers of database records would not run efficiently if they were required to release all higher-numbered records before accessing a new record, making the method impractical for that purpose.

5. MATHEMATICAL MODEL

Let, S be the System Such that,
S={ S, E, I,O, F, DD, NDD, success, failure }
Where,
S = Start state,
E = End state,
I = Input
O = Output
DD = Deterministic Data
NDD = Non-deterministic Data s1 =
Initialization of database/server.
s2 = Initialization of connection with the server.
e1 = Successful completion of the program and output written in database.
F : Functions:
F1 = main() : Here we create the threads and assign them to each philosopher and start their execution.

F2 = run() : This is a function to be written when we use runnable interface for parallel processing of threads.

F3 = eat() : In this function, we lock the neighboring philosopher and the calling thread(philosopher) stars eating operation.

F4 = thing() : In this function, the calling thread(philosopher) starts thinking.

Success Case: It is the case the connection is Successful with the MongoDB server and threads start running parallely.

Failure Case: It is the case when the connection is failed or there is an exception in threads.

6. THE RELIABILITY TESTING IN SOFTWARE

Reliability Testing is about exercising an application so that failures are discovered and removed before the system is deployed. The purpose of reliability testing is to determine product reliability, and to determine whether the software meets the customer's reliability requirements.

According to ANSI, Software Reliability is defined as: the probability of failure-free software operation for a specified period of time in a specified environment. Software Reliability is not a direct function of time. Electronic and mechanical parts may become "old" and wear out with time and usage, but software will not rust or wear-out during its life cycle. Software will not change over time unless intentionally changed or upgraded.

- Reliability refers to the consistency of a measure. A test is considered reliable if we get the same result repeatedly. Software Reliability is the probability of failure-free software operation for a specified period of time in a specified environment. Software Reliability is also an important factor affecting system reliability.
- Reliability testing will tend to uncover earlier those failures that are most likely in actual operation, thus directing efforts at fixing the most important faults.
- Reliability testing may be performed at several levels. Complex systems may be tested at component, circuit board, unit, assembly, subsystem and system levels.
- Software reliability is a key part in software quality. The study of software reliability can be categorized into three parts:

1. Modeling
2. Measurement
3. Improvement

1. Modeling: Software reliability modeling has matured to the point that meaningful results can be obtained by applying suitable models to the problem. There are many models exist, but no single model can capture a necessary amount of the software characteristics. Assumptions and abstractions must be made to simplify the problem. There is no single model that is universal to all the situations.

2. Measurement: Software reliability measurement is naive. Measurement is far from commonplace in software, as in other engineering field. "How good is the software, quantitatively?" As simple as the question is, there is still no good answer. Software reliability cannot be directly measured, so other related factors are measured to estimate software reliability and compare it among products. Development process, faults and failures found are all factors related to software reliability.

3. Improvement: Software reliability improvement is hard. The difficulty of the problem stems from insufficient understanding of software reliability and in general, the characteristics of software. Until now there is no good way to conquer the complexity problem of software. Complete testing of a moderately complex software module is infeasible. Defect-free software product cannot be assured. Realistic constraints of time and budget severely limits the effort put into software reliability improvement.

7. CONCLUSION

The Dining philosopher problem is solved successfully in Python.

```
import threading
import random
import time
# Dining philosophers, 5 Phillies with 5 forks. Must have two forks to eat.
#
# Deadlock is avoided by never waiting for a fork while holding a fork (locked)
# Procedure is to do block while waiting to get first fork, and a nonblocking
# acquire of second fork.  If failed to get second fork, release first fork,
# swap which fork is first and which is second and retry until getting both.

class Philosopher(threading.Thread):
   running = True
   def __init__(self, xname, forkOnLeft, forkOnRight):
       threading.Thread.__init__(self)
       self.name = xname
       self.forkOnLeft = forkOnLeft
       self.forkOnRight = forkOnRight
   def run(self):
       while(self.running):
           #  Philosopher is thinking (but really is sleeping).
           time.sleep( random.uniform(3,13))
           print '%s is hungry.' % self.name
           self.dine()
   def dine(self):
       fork1, fork2 = self.forkOnLeft, self.forkOnRight
       while self.running:
           fork1.acquire(True)
           locked = fork2.acquire(False)
           if locked: break
           fork1.release()
           print '%s swaps forks' % self.name
           fork1, fork2 = fork2, fork1
       else:
           return
       self.dining()
       fork2.release()
       fork1.release()
   def dining(self):
       print '%s starts eating '% self.name
       time.sleep(random.uniform(1,10))
       print '%s finishes eating and leaves to think.' % self.name
def DiningPhilosophers():
   forks = [threading.Lock() for n in range(5)]
   philosopherNames = ('Aristotle','Kant','Buddha','Marx', 'Russel')
   philosophers= [Philosopher(philosopherNames[i], forks[i%5], forks[(i+1)%5]) \
           for i in range(5)]
   random.seed(507129)
   Philosopher.running = True
   for p in philosophers: p.start()
   time.sleep(20)
   Philosopher.running = False
   print ("Now we're finishing.")
   exit()
DiningPhilosophers()
```

```
'''
cipher@blackfury-HP-eNVy:~/be-2$ python diningphilo.py
Russel is hungry.
Russel starts eating
Aristotle is hungry.
Marx is hungry.
Marx swaps forks
Kant is hungry.
Kant starts eating
Buddha is hungry.
Russel finishes eating and leaves to think.
Aristotle swaps forks
Marx starts eating

Kant finishes eating and leaves to think.
Buddha swaps forks
Aristotle starts eating

Marx finishes eating and leaves to think.
Buddha starts eating
Russel is hungry.
Russel swaps forks
Now we're finishing.
Aristotle finishes eating and leaves to think.
Russel starts eating
Buddha finishes eating and leaves to think.
Russel finishes eating and leaves to think.
Marx is hungry.
Kant is hungry.
'''
```