Week 4 Homework due on Friday Oct 12, 18:59 Hour

Group 5

Wong Ann Yi (1004000)
Liu Bowen (1004028)
Tan Chin Leong Leonard (1004041)

Exercise 1

The dining philosophers' problem was proposed by Edgar Dijkstra and described here: http://www.cs.utexas.edu/users/EWD/transcriptions/EWD10xx/EWD1000.html. The problem says that there are N of philosophers sitting at a round table and thinking about problems. Each philosopher has a plate of spaghetti in front of him/her and his/her own fork to the left of the plate. However, to eat spaghetti one has to use 2 forks at the same time, which means that a philosopher has to borrow the fork of his/her neighbor in order to eat. After eating, a philosopher puts both forks down in their corresponding places and then continues to think. As explained in the class, this will cause the deadlock problem. Provide a solution (with detailed steps) to address this problem.

Answer:

The dinning philosopher problem is used in concurrent algorithm to resolve synchronization issues. In this problem, a deadlock will occur if all the philosophers pick up a fork and do not put down. Alternatively, a few philosophers may be rotating the forks among themselves leaving others to starve. Therefore, we have studied the solutions which are developed in the academic world and derived two such algorithms to solve the dinning philosopher problem.

In the first algorithm, we will use semaphores to prevent deadlock and starvation. This solution is also known as the Arbitrator solution (found in wiki). In this scenario, there are 5 philosophers and 5 forks. Every fork is represented by a semaphore and there will be 5 semaphores. We will use another semaphore called mutex to be a waiter (or arbitrator) which is usually 1. In order to pick up the forks, a philosopher must ask permission from the waiter. The waiter (mutex) will reduce the counter to 0 and check the forks availability. The waiter gives permission to only one philosopher at a time until the philosopher has picked up both of their forks. Putting down a fork is always allowed. This solution can resolve deadlock and starvation.

In addition to introducing a new central entity (the waiter), this approach can result in reduced parallelism: if a philosopher is eating and one of their neighbors is requesting the forks, all other philosophers must wait until this request has been fulfilled even if forks for them are still available.

Below are the suggested codes:

```
do {
```

/*Every philosopher will execute the below coding if he wants to eat*/

wait(mutex) /*The waiter(mutex) will assign the forks to a philosopher it will reduce from 1 to 0 and the rest of the philosophers will need to wait*/

wait(fork[i]); /*The philosopher will try to lock the left fork (e.g. Acquire fork semaphore 0)*/

wait(fork[i+1]%5]; /*The philosopher will try to lock the right fork (e.g. Acquire fork semaphore 4)*/

signal(mutex) /*After the philosopher is done eating he will signal the waiter increase the mutux back to 1*/

eat

signal(fork[i]); /*The philosopher will release the left fork (e.g. Release fork semaphore 0)*/

signal(fork[(i+1)%5]; /*The philosopher will release the right fork(e.g. Release fork semaphore 4)*/

think

}while (true) /*Upon successful execution the program will keep looping*/

==========

For the second algorithm to address the Dinning Philosophers problem, we tried to use a mix of Condition and Semaphore objects in Python to develop a solution[1]. Condition objects allows us to use a "lock" to synchronize access to some shared resources (or states and in our case the forks!). Threads (which are tasks and in this case represent the Philosophers wanting to eat!) that are interested in the shared resources will call notify() repeatedly until they see the availability of the desired resources via the semaphore. On the other hand, threads (or a satisfied Philosopher) that have eaten will call notify() to modify the semaphore so that other threads would find it available (to eat!). The function Notify() "wakes" up threads who are waiting for a change in the semaphore state. We have constructed the solution with the following parts:

1) threading.Condition() allows one or more threads (representing the philosophers) to wait until they are notified by another thread.
2) introduce a semaphore is_eating. When a philosopher starts to eat, the semaphore is_eating is set to false, otherwise it will be set to true.
3) We will use the value of is_eating to represent the presence of a fork. Each philosopher need to maintain a value of is_eating. An example is shown in below Figure 1, when P_1

[1] Higher-level threading interface - https://docs.python.org/2/library/threading.html

starts thinking, he will set is_eating to false and then call the notify() function. The notify() method wakes up one of the threads (P_0 or P_2) and trigger it to eat.

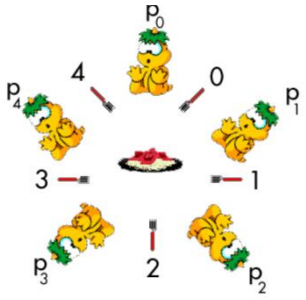


Figure 1

Before P_1 starts to eat, he will check if the neighboring philosophers (either P_0 or P_2) is eating. If one of them is eating, P_1 will set is_eating semaphore to false and call wait(). If neither P_0 nor P_2 is eating, then P_1 will start to eat. This is shown in def eating() in the codes below. In general, the above actions will apply to philosopher P_i, P_{i-1} and P_{i+1} where i is 0 $\leq i \leq 4$.

The codes (in Python) are as follows:

##

```
import threading
import random
import signal
class Philosopher(threading.Thread):
    def __init__(self, con):
        threading.Thread.__init__(self)
        self.is_eating = False
        self.con = con
#"left" denotes philosopher on the left and "right" denotes philosopher on the right
    def set_both_sides(self, left, right):
        self.left = left
        self.right = right
    def thinking(self):
        with self.con:
            self.is_eating = False
            # notify() method wakes up one of the threads waiting for the change in semaphore
condition
            self.left.con.notify() #if Philosopher is thinking, he wakes up both left and right
philosophers
            self.right.con.notify()
            time.sleep(random.random())
    def eating(self):
```

```
        with self.con:
            #if one Philosopher notices either left or right philosopher is eating, he releases the lock
            while self.left.is_eating or self.right.is_eating:
                # The wait() method releases the lock, and then blocks until it is awakened by a
notify() call
                self.con.wait()
            self.is_eating = True #if one Philosopher notices both the left philosopher and right
philosopher are not eating, he will start to eat
            print '%s is eating %s' % (threading.currentThread().getName())
    def run(self):
        while True:
            self.thinking()
            self.eating()
if __name__ == '__main__':
    num = 5
    con = threading.Condition()#allows one or more threads to wait until they are notified by
another thread.
    philosophers = []
    for _ in range(num):
        philosophers.append(Philosopher(con))
    for i in range(num):
        philosophers[i].set_both_sides(philosophers[(i - 1) % num], philosophers[i % num])
    for p in philosophers:
        p.start()
```

Exercise 2

The Byzantine generals' problem was proposed by Lamport et al. in the following paper: http://research.microsoft.com/en-us/um/people/lamport/pubs/byz.pdf. The problem says that a reliable computer system must be able to cope with the failure of one or more components. A failing component may exhibit a type of behavior that confuses other components, i.e. sending conflicting information to other components.

a) Describe an algorithm which can guarantee that all properly functioning components can reach a common decision/state given a certain number of failing components.

Answer:

The algorithm is derived by solving the Byzantine Generals' Problem. The problem simulates the computer component(s) failure by imagining several divisions of a Byzantine army attacking a city. The generals of each division are dispersed around the city and they must agree on a strategy to attack or to retreat. However, there are traitors (they represent the failed components in the computer system) amongst the generals who will transmit false information on the battle strategy. The solution of the algorithm will ensure that all loyal lieutenants generals decide upon the same plan of action; and a small number of traitors cannot cause the loyal generals to adopt a bad plan.

We will describe an algorithm which the generals communicate with oral messages with no signature. The following conditions are required in the algorithm:

- That every message that is sent is delivered correctly;
- That the receiver of a message knows the sender; and
- That the absence of a message can be detected.

In addition, a commander must send an order to his n-1 lieutenant generals such that interactive consistency conditions hold in the presence of at most m traitors where n>=3m+1. They are:

IC1: all loyal lieutenants obey the same order;
IC2: if the commander is loyal, then all lieutenants obey the order he sends.

At a high level, the algorithm essentially finds the majority vote of the decision made by all the generals and assuming that only a small number of traitors are present, their votes will not cause an impact.

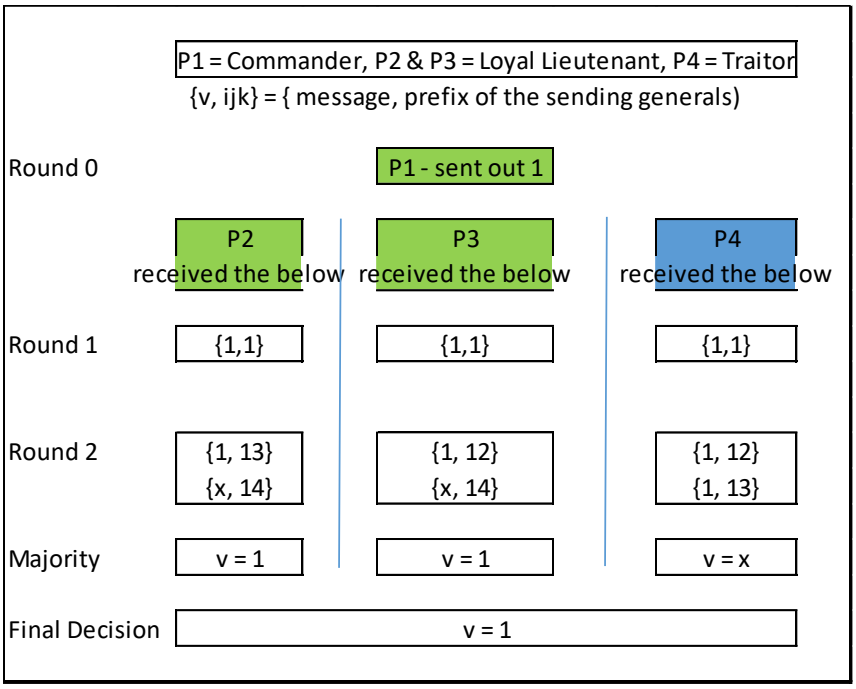
The algorithm contains the following steps with OM(m) referring to Oral Message algorithm for all non-negative integers m (which is also the number of traitors); v(i) the value receives by lieutenant (i); and n is the total number of general/lieutenants. It starts with a base case of m=0 and recursive steps for m>0. In addition, the function majority(v(1), …, v(n)) returns the majority value among v(i) if it exists or otherwise the value "retreat".

- The lieutenants recursively forward orders to all the other lieutenants;

- Algorithm OM(0)
  - The General sends his value to every lieutenant.
  - Each lieutenant uses the value he receives from the general; or uses the value "retreat" if he receives no value.
- Algorithm OM(m), m>0
  - The General sends his value to every lieutenant.
  - Lieutenant (i) acts as the general in in Algorithm (m-1) and send value v(i) to n-2 other lieutenants.
  - Let v(j) (where $j \neq i$) be the value Lieutenant (i) received from Lieutenant (j) in previous step (when using Algorithm (m-1)), or else "retreat" if he received no value. Lieutenant (i) then use the value majority(v(1), … v(n-1)).
  - The majority value tallied by all the generals will be the plan of action agreed.

This is illustrated in the below example using m= 1 (m represents the number of traitor or failing components) and n = 4 (n represents the total number of Generals or components). We use Pi (i=1,2,3,4) or P1, P2, P3 and P4 to represent the 4 generals with P1 being the commander, P2 and P3 being the loyal lieutenants and P4 being the traitor. The message sent by the commander will be a value v = 1 or 0. In this example, the commander will send a value 1 while the traitor will send a wrong value say x. We will denote {v, ij} as the message v sent by Pi and resent by Pj. See below figure which shows the message received by each lieutenant at each round and the final tally of the message(s) based on majority. By running the algorithm and following the capacity relationship of n>= 3m+1, the final decision is the value 1 which is sent by the loyal commander.

Do note that P2 and P3 each received two 1 and one x values and using majority principle, value 1 is the vote. P4 being a traitor will provide a different value x. Comparing two 1 and one x from the generals, final value is 1 which is the same value transmitted by the commander.



b) What is the maximum number of failing components that this algorithm can tolerate?

Answers:

In order for the algorithm to work, the maximum number of failing components must follow the formula of n >= 3m+1 where n is the total number of components (or processes) and m is number of the failing components (or processes). This will allow the algorithm to achieve interactive consistency among the non-faulty processes.

c) Why can't the algorithm tolerate more failing components?

Answer:

As per b), the number of failing components (m) must follow the formula of n >= 3m+1 where n is the total number of components otherwise if n is less than 3m+1, the algorithm will not work. We can prove this by inference by assuming n=3 and m=1 and reduce this to a 3 Generals problem where there is 1 traitor and 2 loyal Generals. A loyal lieutenant cannot distinguish who is the traitor when he gets equal amount of conflicting messages from the two Generals.

A more rigorous proof is by contradiction where we assume a solution exists for a group of 3m or fewer generals and use it to construct a three-general solution. Each general will simulate one third of 3m generals. Since at most only 1 general is the traitor, there are at most 1m traitors among 3m generals. The assumed solution guarantees that IC1 and IC2

hold for all the generals. By IC1, all lieutenants obey the same order which is the order he is to obey. Therefore, conditions IC1 and IC2 which apply to the generals would imply that all the generals are loyal which is incorrect and in contradiction in this case. This is also written by Lamport, Shostak and Pease in the paper “Reaching Agreement in the Presence of Faults” published in Journal of the Association for Computing Machinery, Vol 27, No.2 April 1980.

Exercise 3

The following pair of processes share a common variable X, and use a shared binary semaphore S:

```
[Process A]          [Process B]
int Y;               int Z;
wait(S);             wait(S);
A1: Y = X*2;         B1: Z = X+1;
A2: X = Y;           B2: X = Z;
signal(S);           signal(S);
```

S is set to 1 before either process begins execution and X is set to 5. Statements within a process are executed sequentially, but statements in process A may execute in any order with respect to statements in process B and vice versa. Each process will only execute once.

a) How many different values of X are possible after both processes finish executing? Why?

As there is semaphore S in Process A and Process B which guarantee each process execute without any halt and stuck. Hence, only two sequences are supposed to execute:

Process A -> Process B: A1-A2-B1-B2, for this circumstance X equals 11
Process B -> Process A: B1-B2-A1-A2, and X equals 12

b) Suppose the programs are modified as follows to use a shared binary semaphore T:

```
[Process A]          [Process B]
int Y;               int Z;
                     wait(T);
A1: Y = X*2;         B1: Z = X+1;
A2: X = Y;           B2: X = Z;
signal(T);
```

T is set to 0 before either process begins execution and, as before, X is set to 5. Now, how many different values of X are possible after both processes finish executing? Why?

Answer:

Different from a), T is set to 0 and therefore T guarantees that Process B starts to execute if and only if Process A finishes and add a value to T. Hence, there is only one circumstance:

Process A -> Process B: A1-A2-B1-B2, for this circumstance X equals 11