Dining-Philosophers Problem

- Shared data
  - Bowl of rice (data set)
  - Semaphore chopstick [5] initialized to 1

Dining-Philosophers Problem (Cont.)

- The structure of Philosopher *i*:

```
Do  {
      wait ( chopstick[i] );
      wait ( chopStick[ (i + 1) % 5] );

            //  eat

      signal ( chopstick[i] );
      signal (chopstick[ (i + 1) % 5] );

           //  think

} while (true) ;
```

Dining-Philosophers Problem (Cont.)

- The structure of Philosopher *i*:

```
Do  {
      wait ( chopstick[i] );
      wait ( chopStick[ (i + 1) % 5] );

            //  eat

      signal ( chopstick[i] );
      signal (chopstick[ (i + 1) % 5] );

           //  think

} while (true) ;
```

Deadlock!- no progress
Each picks
Left fork at same time
And all wait forever

Conductor solution

- do
  {wait(mutex)
    wait(i)
    wait((i+1)%5)
   signal(mutex)
  //eat
   signal(i)
   signal((i+1)%5)
   think
  }while(true)

Conductor solution

- do
  {wait(mutex)
  wait(i)
  wait((i+1)%5)
  signal(mutex)
  //eat
  signal(i)
  signal((i+1)%5)
  think
  }while(true)

Lets say P1 got both his forks and is eating.
P2 reaches step 3 after getting one fork and is waiting for the other which is being used by P1.
➔P2 is still inside the mutex lock which does not permit P4 to request forks though both his forks are idle.

Concurrent execution not Possible due to mutex lock

Move with precaution

Imagine,

- All philosophers start at the same time
- Run simultaneously
- And think for the same time

This could lead to philosophers taking fork and putting it down continuously. a deadlock.

A better alternative

- Philosophers wait a random time before take_fork(i)
- Less likelihood of deadlock.
- Used in schemes such as Ethernet

```
void philosopher(int i){
  while(TRUE){
     think();
     take_fork(i);
     if (available((i+1)%N){
        take_fork((i + 1) % N);
        eat();
     }else{
         put_fork(i);
     }
}
```

Move with precaution

Imagine,

- All philosophers start at the same time
- Run simultaneously
- And think for the same time

This could lead to philosophers taking fork and putting it down continuously. a deadlock.

A better alternative

- Philosophers wait a random time before take_fork(i)
- Less likelihood of deadlock.
- Used in schemes such as Ethernet

```
void philosopher(int i){
  while(TRUE){
     think();
     take_fork(i);
     if (available((i+1)%N){
        take_fork((i + 1) % N);
        eat();
     }else{
         put_fork(i);
     }
}
```

All pick fork together to realise other fork is not available. So all put back fork together. This goees on.

Tannenbaum solution

Uses N semaphores (s[0], s[1], …., s[N]) all initialized to 0, and a mutex
Philosopher has 3 states: HUNGRY, EATING, THINKING
A philosopher can only move to EATING state if neither neighbor is eating

```
void philosopher(int i){
    while(TRUE){
        think();
        take_forks(i);
        eat();
        put_forks();
    }
}
```

```
void take_forks(int i){
    lock(mutex);
    state[i] = HUNGRY;
    test(i);
    unlock(mutex);
    down(s[i]);
}
```

```
void put_forks(int i){
    lock(mutex);
    state[i] = THINKING;
    test(LEFT);
    test(RIGHT)
    unlock(mutex);
}
```

```
void test(int i){
    if (state[i] = HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING){
            state[i] = EATING;
            up(s[i]);
    }
}
```

- Up(fork) is like signal(fork)
- Down(fork) is like wait(fork)
  wait(fork){ fork --; while (fork <0);}

Chandy Misra solution

- For every pair of philosophers contending for a resource, create a fork and give it to the philosopher with the lower ID (*n* for agent P_n). Each fork can either be *dirty* or *clean.* Initially, all forks are dirty.
- When a philosopher wants to use a set of resources (*i.e.* eat), he must obtain the forks from his contending neighbors. For all such forks he does not have, he sends a request message.
- When a philosopher with a fork receives a request message, he keeps the fork if it is clean, but gives it up when it is dirty. If he sends the fork over, he cleans the fork before doing so.
- After a philosopher is done eating, all his forks become dirty. If another philosopher had previously requested one of the forks, he cleans the fork and sends it.
- https://ycpcs.github.io/cs420-fall2017/lectures/lecture12_classic_synchronization_problems.pdf

Rules

- Initially forks dirty
- After eating forks dirty
- If someone had requested for your fork when you were eating, then clean and give it after you finish eating
- If you have both forks start eating
- I need to give a fork on request only if its dirty. I clean it and give it. If the fork is clean it means I have requested and got it and have not eaten yet. In such a case I do not give it up.
- Initially the one with lower id gets preference. So if P1 requests forks initially, then he will get both forks. Deadlock situation wont arise.
- Starvation wont arise since a philosopher who has eaten has to wait if his neighbors have requested for fork.