Introduction to System Programming

Deadlock

Minal Shah

Outline

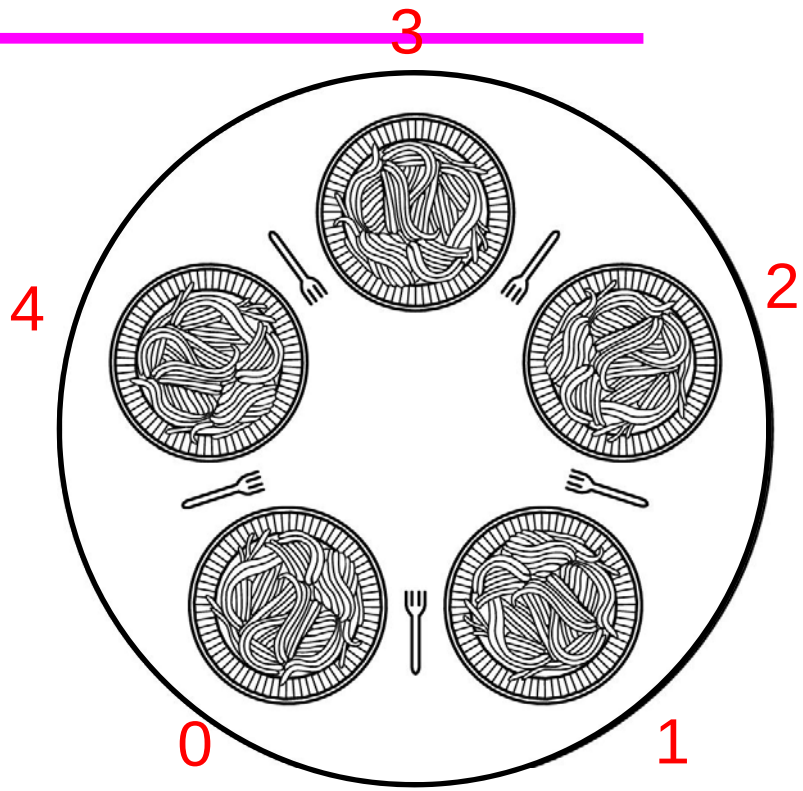
- Dinging Philosopher Problem
- Deadlock V/s Starvation

Classical IPC Problem
Dining Philosophers problem

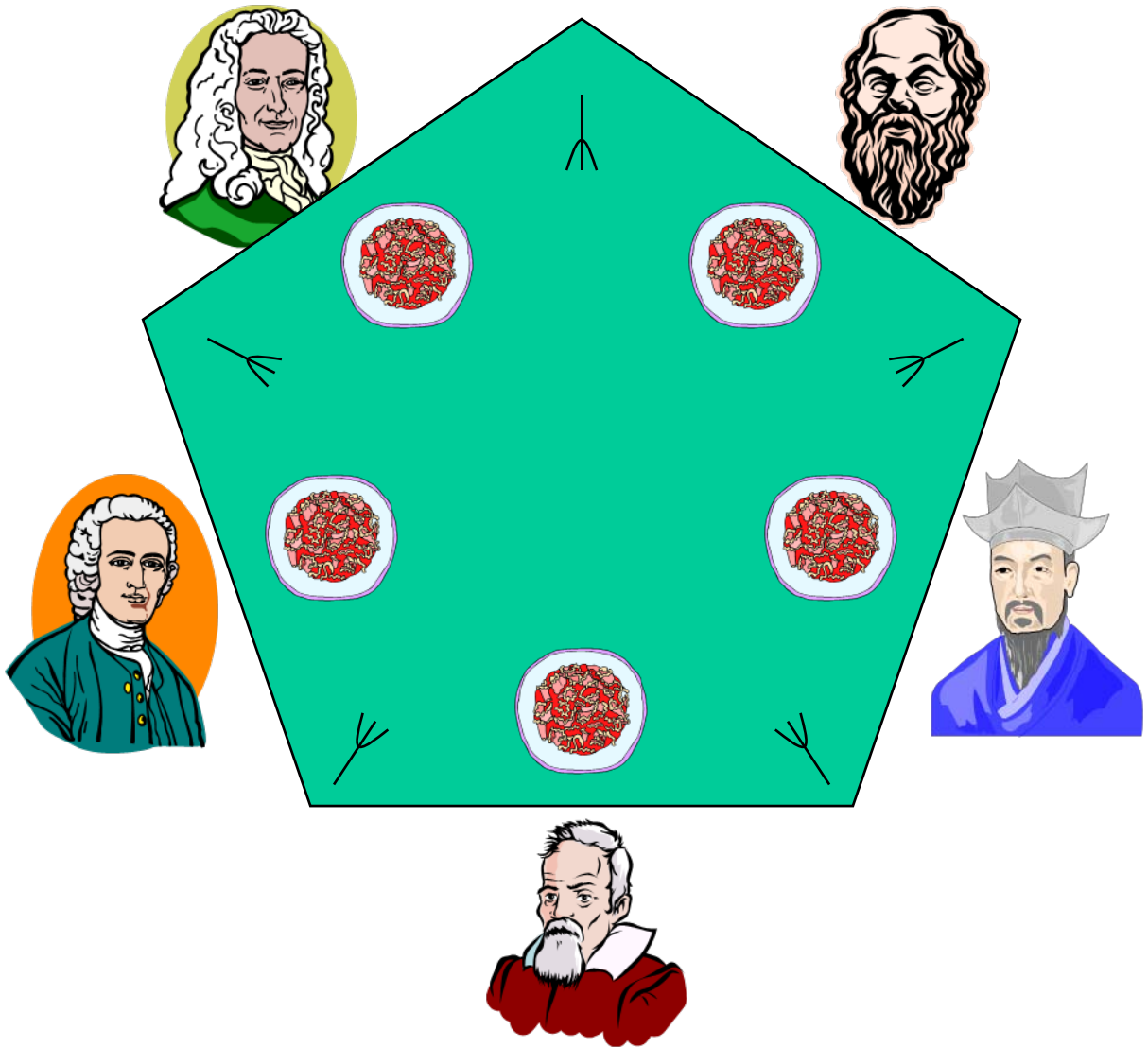
- Five philosophers are sitting around circular table. Each philosopher has one plate of rice.
- Between each pair of plate there is one fork.
- Philosopher has alternate period of eating and thinking.
- When philosopher is hungry, it tries to get two forks (from left and right) one at a time.
- If able to get then eats for sometime and keep down fork and continue.
- Each philosopher should get fork whenever required and should not stuck.
- Problem of multiple resources and multiple processes.
- Suppose philosopher 0 to 4.
- If philosopher is 2 then left philosopher is 1 and right is 3.

Dining Philosophers

- **Motivation is access to limited number of resources e.g. I/O device**
- Philosophers eat/think
- Eating needs 2 forks
- Pick one fork at a time
- How to prevent deadlock



Dining Philosopher's Problem (Dijkstra '71)

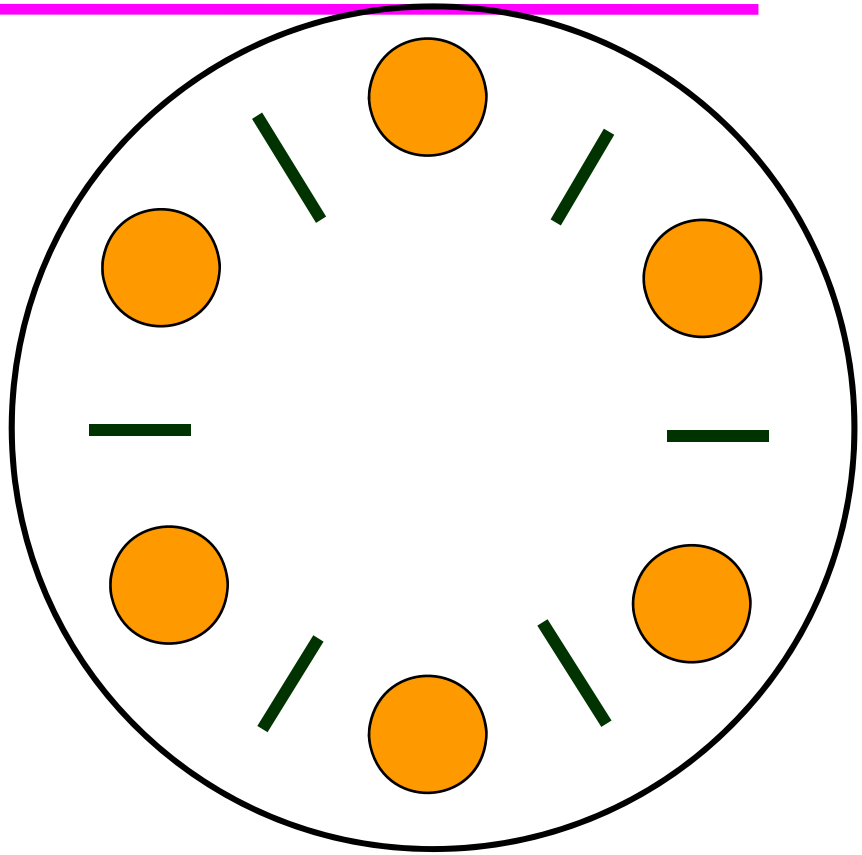


Dining philosophers: definition

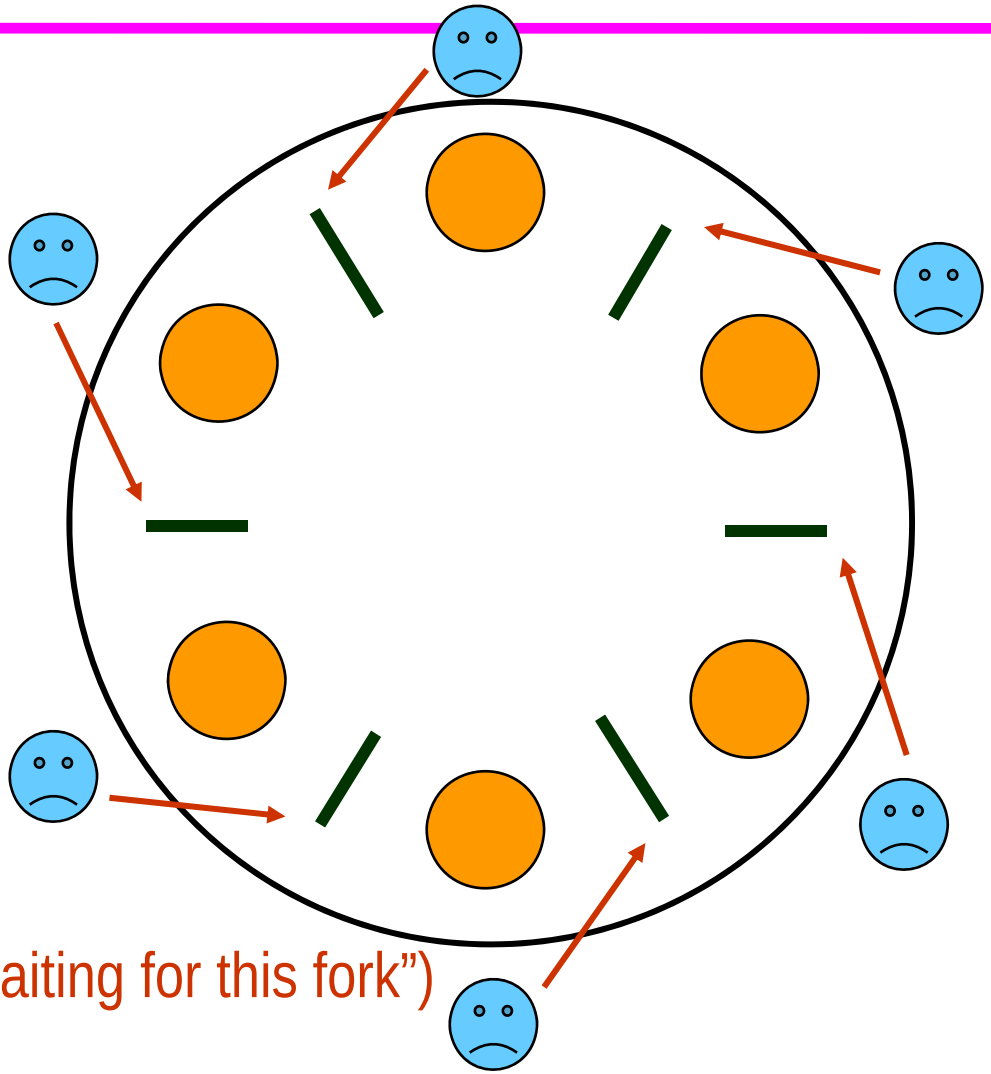
- Each process needs two resources
- Every pair of processes compete for a specific resource
- *A process may proceed only if it is assigned both resources*
- Every process that is waiting for a resource should sleep (be blocked)
- Every process that releases its two resources must wake-up the two competing processes for these resources, if they are interested

The Dining Philosophers Problem

- Philosophers
  - think
  - take forks (one at a time)
  - eat
  - put forks (one at a time)
- Eating requires 2 forks
- Pick one fork at a time
- How to prevent deadlock?
- What about starvation?
- What about concurrency?



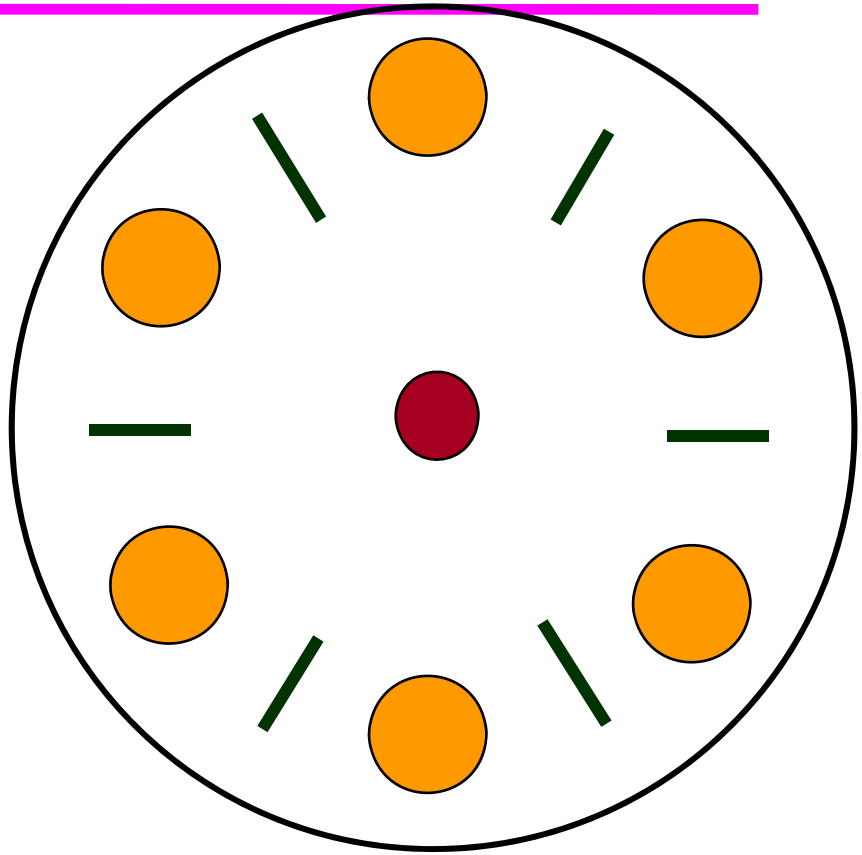
An incorrect naïve solution



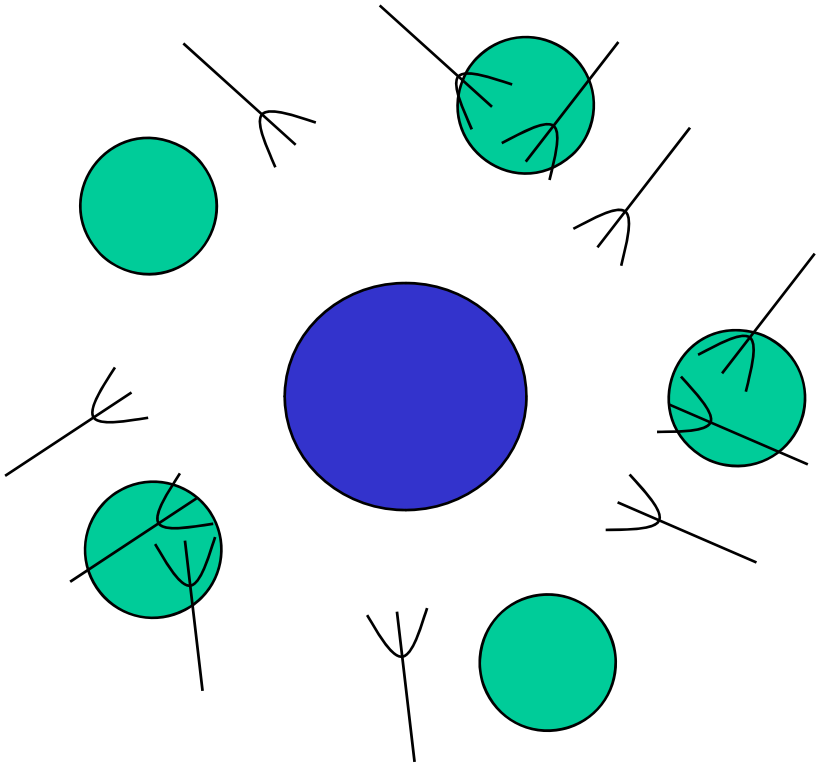
(⟶ means “waiting for this fork”)

Dining philosophers

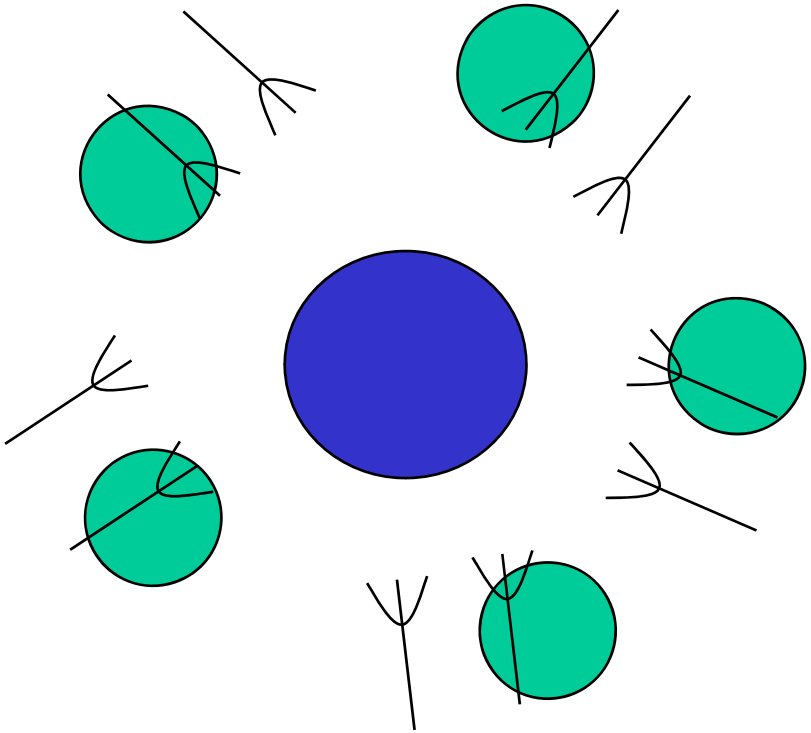
- The solution
  - A philosopher first gets
  - only then it tries to take the 2 forks.



Starvation - Pictorially



Deadlock - Pictorially



Dining Philosophers: Solutions

- Simple: "waiting" state
  - Enter waiting state when neighbors eating
  - When neighbors done, get forks
  - Neighbors can't enter waiting state if neighbor waiting
- Problem:
  - Doesn't prevent starvation
  - Requires checking both neighbors at once
    - Race condition

Dining Philosophers

```
#define N 5                                     /* number of philosophers */

void philosopher(int i)                         /* i: philosopher number, from 0 to 4 */
{
      while (TRUE) {
            think( );                           /* philosopher is thinking */
            take_fork(i);                       /* take left fork */
            take_fork((i+1) % N);               /* take right fork; % is modulo operator */
            eat( );                             /* yum-yum, spaghetti */
            put_fork(i);                        /* put left fork back on the table */
            put_fork((i+1) % N);                /* put right fork back on the table */
      }
}
```

A nonsolution to the dining philosophers problem

Dining Philosophers: Solutions

- Above solution is wrong, because if all 5 philosophers take left fork simultaneously.
- So right fork will not be available and cause deadlock.
- Modification:
  - Take left fork and check whether right is available or not.
  - If not, put down left fork and wait for some time.
  - Repeat the process.

Dining Philosophers: Solutions

- Problem – All picks up left fork simultaneously.
  - Looking that right is not available, put down left and wait for some time.
  - After some time again all picks it up simultaneously and so on forever.
  - This situation i.e. program running but cannot progress is called starvation.
  - This is very rare situation because generally waiting time is random. But not perfect solution.

Solution using semaphores

- To avoid deadlock and starvation, binary semaphore can be used. i.e. before getting fork down on mutex and after putting fork up on mutex.
- In that case only one philosopher can eat at a time even though maximum 2 is possible. i.e. lower the performance
- Another solution (deadlock free and for any number of philosophers)
  - Array called state is used.
  - Philosopher can be in one of the state. i.e. eating, thinking, hungry
  - Philosopher can move to eating state only when neither neighbor is eating.

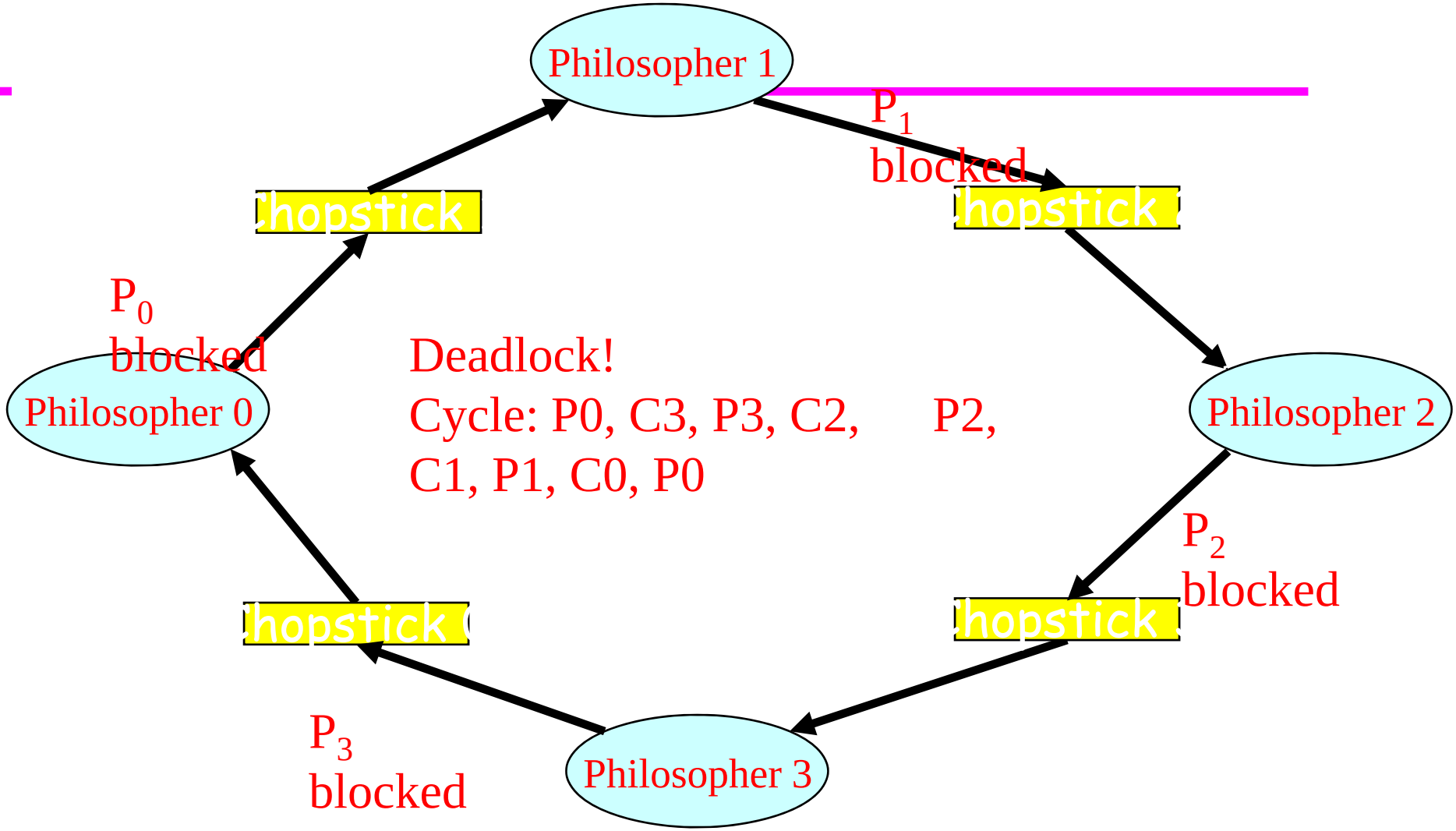
Deadlock

- Deadlock exists in a set of processes/threads when all processes/threads in the set are waiting for an event that can only be caused by another process in the set (which is also waiting!).
- Dining Philosophers is a perfect example. Each holds one chopstick and will wait forever for the other.

Resource Allocation Graph

- Deadlock can be described through a resource allocation graph
- Each node in graph represents a process/thread or a resource
- An edge from node P to R indicates that process P had requested resource R
- An edge from node R to node P indicates that process P holds resource R
- If graph has cycle, deadlock *may* exist. If graph has no cycle, deadlock **cannot** exist.

Cycle in Resource Allocation Graph



Interrupt & Context switch →

wait(chopstick[i]);

wait(chopstick[(i+1)%N]);

Fixing Dining Philosophers

- Make philosophers grab both chopsticks they need atomically
  - Maybe pass around a token (lock) saying who can grab chopsticks
- Make a philosopher give up a chopstick
- Others?

Better Semaphore Solution to Dining Philosophers

```
void philosophersLife(int i){
   while (1) {
      think();
      if ( i < ((i-1) % NUM_PHILOSOPHERS))}{
           wait(chopstick[i]);
           wait(chopstick[(i-1) % NUM_PHILOSOPHERS]);
      } else {
           wait(chopstick[(i-1) % NUM_PHILOSOPHERS]);
           wait(chopstick[i]);
      }

      eat();

      signal(chopstick[i]);
      signal(chopstick[(i-1) %
         NUM_PHILOSOPHERS]);
   }
}
```

Always wait for low chopstick first

Why better?

philosopher 0 gets chopstick 0

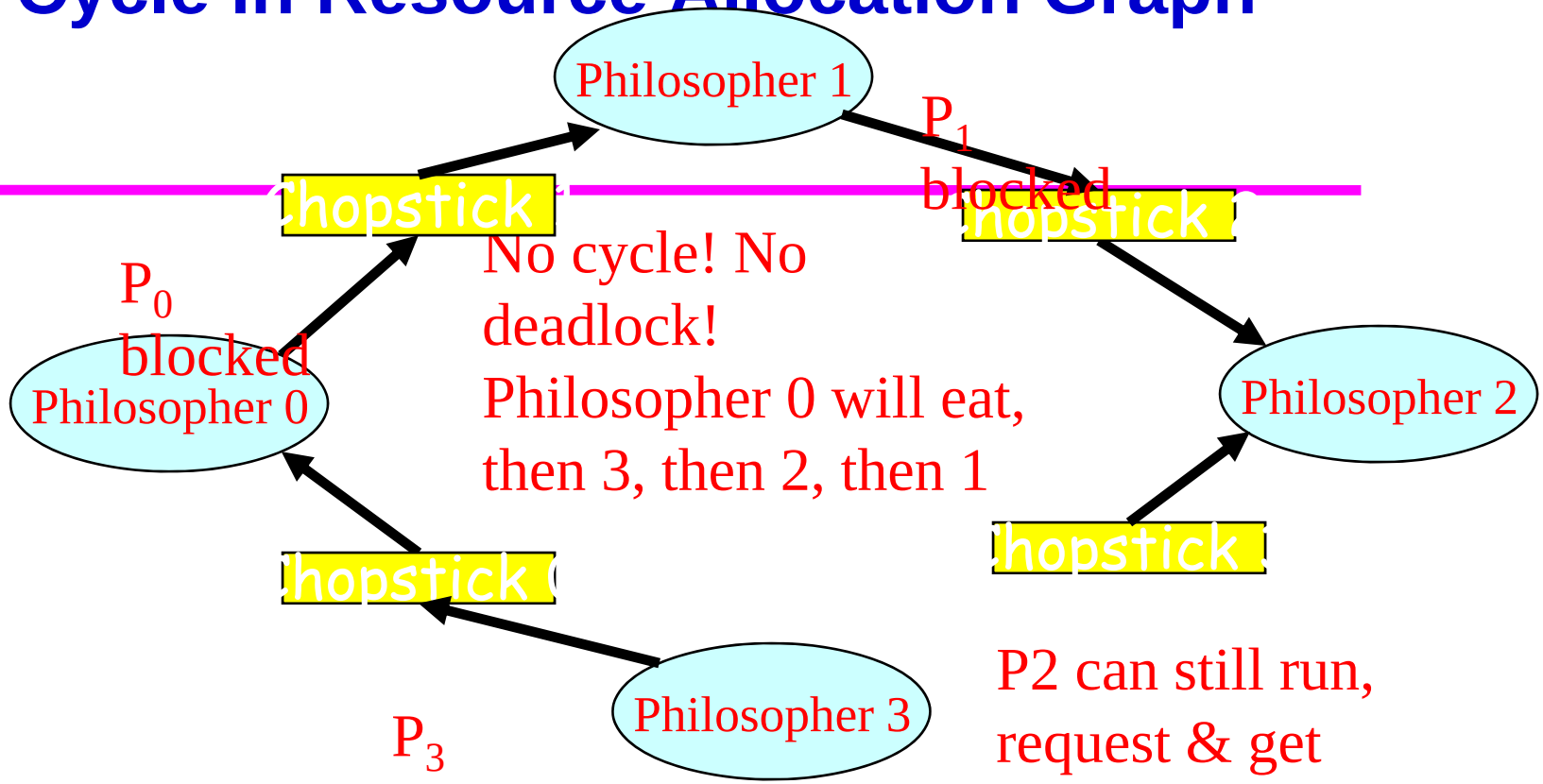
philosopher 1 gets chopstick 1

….

philosopher N waits for chopstick 0

No circular wait! No deadlock!!

No Cycle in Resource Allocation Graph

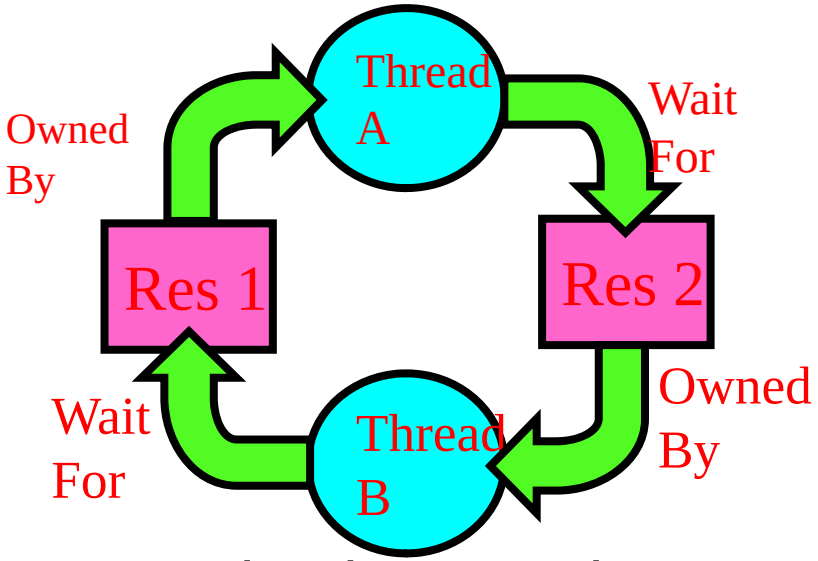


P2 can still run, request & get chopstick 3, eat, release chopsticks 2 & 3

```
if ( i < ((i+1)%N))){
      wait(chopstick[i]);
      wait(chopstick[(i+1)%N]);
} else {
      wait(chopstick[(i+1)%N]);
      wait(chopstick[i]);
}
```

Starvation vs Deadlock

- Starvation vs. Deadlock
  - Starvation: thread waits indefinitely (Example, low-priority thread waiting for resources constantly in use by high-priority threads)
  - Deadlock: circular waiting for resources (Thread A owns Res 1 and is waiting for Res 2, Thread B owns Res 2 and is waiting for Res 1)



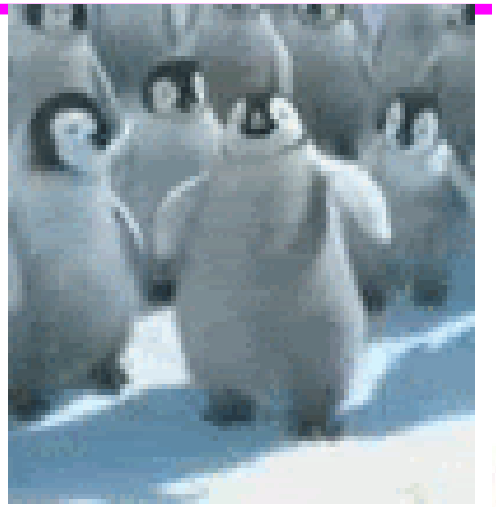
- Deadlock ⇒ Starvation but not vice versa
  - Starvation can end (but doesn't have to)
  - Deadlock can't end without external intervention

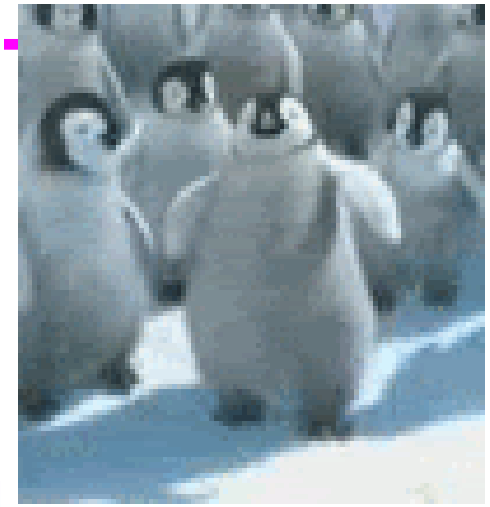
Key Differences Between Deadlock and Starvation in OS

- In a deadlock, none of the processes proceeds for execution, each process get blocked waiting for the resources acquired by the another process. On the other hand, starvation is a condition where the processes that possess higher priority is allowed to acquire the resources continuously by preventing the low priority processes to acquire resources resulting in indefinite blocking of low priority processes.
- Deadlock arises when four conditions **Mutual exclusion, Hold and wait, No preemption, and Circular wait** occurs simultaneously. However, starvation occurs when process **priorities have been enforced** while allocating resources, or there is uncontrolled resource management in the system.

Key Differences Between Deadlock and Starvation in OS

- Deadlock is often called by the name **circular wait** whereas, the starvation is called **Lived lock**.
- In Deadlock the resources are blocked by the process whereas, in starvation, the processes are continuously being used by the processes with high priorities.
- Deadlock can be prevented by the avoiding the conditions like mutual exclusion, Hold and wait, and circular wait and by allowing the preemption of the processes that are holding resources for a long time. On the other hand, Starvation can be prevented by **aging**.





Minal Shah