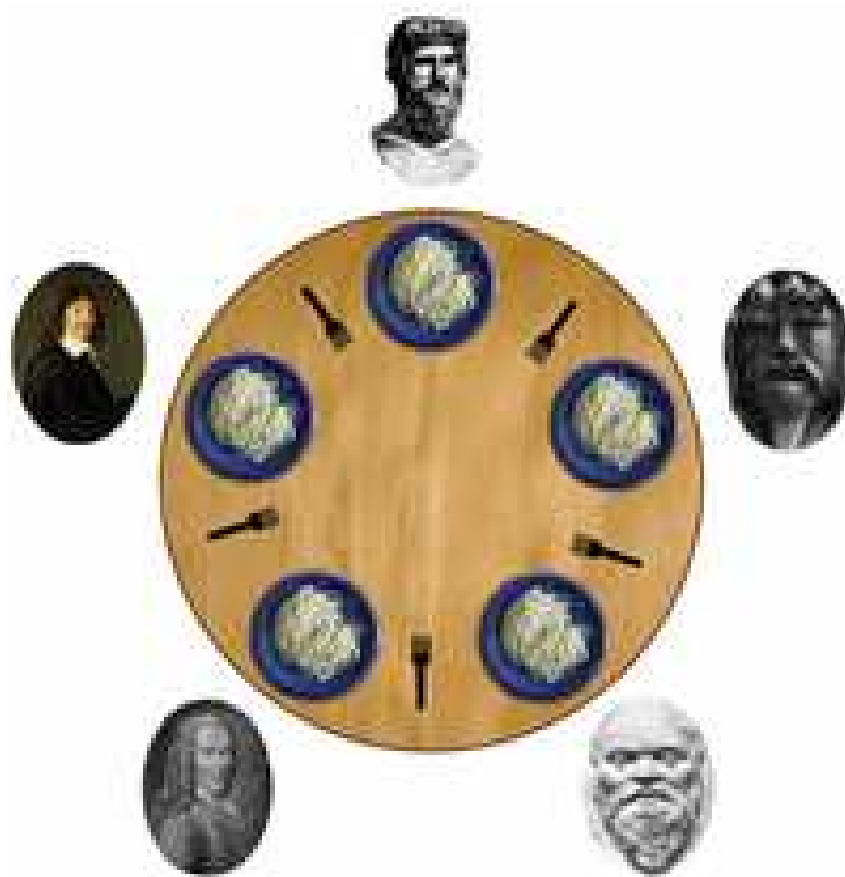


Dining Philosophers

What is it?





- A philosopher thinks and eats.
- 5 philosophers sitting around a table.
- 5 forks - 1 shared between each pair of philosophers.
- A philosopher needs the fork on either side in order to eat.
- We don't really want philosophers starving to death.

Examples

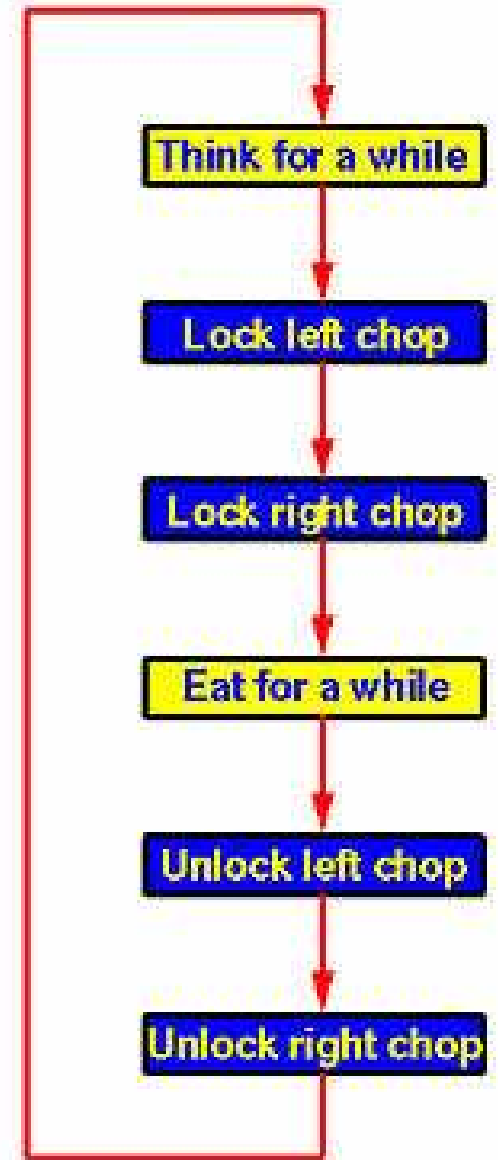
- <http://www.doc.ic.ac.uk/~jnm/concurrency/classes/Diners/Diners.html>
- <http://web.engr.oregonstate.edu/~quinn/education/JavaApplets/applets/DiningPhilosophers.html>

Race Conditions

- It's a race to get to each chopstick
- = is a flaw in a system or process whereby the output of the process is unexpectedly and critically dependent on the sequence or timing of other events.

A solution?

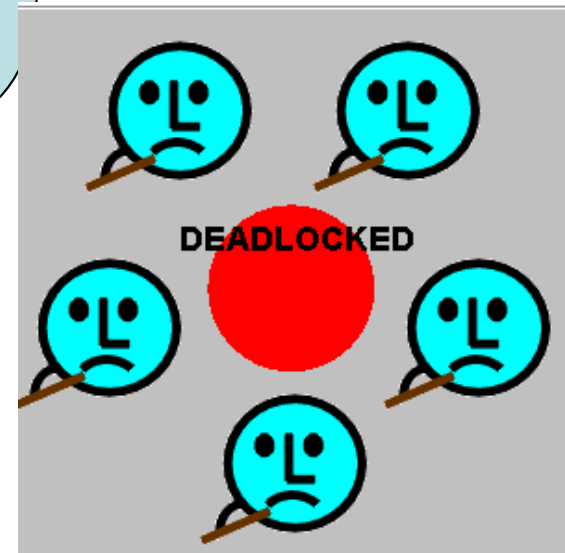
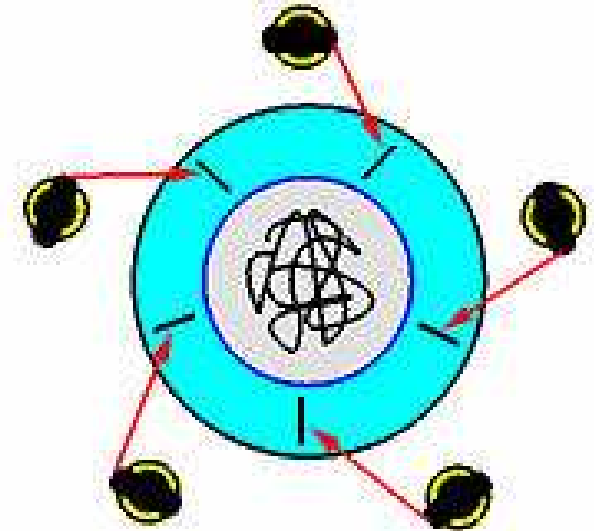
- A philosopher does this:
loop
 think
 eat
end
- definition eat
 @status = "waiting"
 @right.waitForLock
 @left.waitForLock
 @status = "eating"
 @right.freeLock
 @left.freeLock
end
- What's the problem?



A solution? – Deadlock

- A philosopher does this:
loop
 think
 eat
end
- definition eat
 @status = "waiting"
 @left.waitForLock
 @right.waitForLock
 @status = "eating"
 @left.freeLock
 @right.freeLock
end

Deadlock can occur –
Philosophers all lock left chopstick and all are stuck waiting for right chopstick



Deadlock

- Example: every philosopher holds a left fork and waits perpetually for a right fork (or vice versa).
- A set of threads is in a deadlock state when every thread in the set is waiting for an event that can only be caused by another (also waiting) thread in the set.
- Example: deadlock can occur when multiple processes acquire multiple resources (printers, disks, etc.), perform work, and then release their resources.
- What if one process acquires the disk and then the printer, and another process acquires the printer and then the disk?
- Circular wait

Deadlock Example

- Alphonse and Gaston are friends, and great believers in courtesy.
- A strict rule of courtesy is that when you bow to a friend, you must remain bowed until your friend has a chance to return the bow.
- Unfortunately, this rule does not account for the possibility that two friends might bow to each other at the same time.

Deadlock

- **Deadlock exists iff following conditions hold:**
- [Mutual Exclusion] at least one thread must hold a resource in nonsharable mode -- i.e., the resource may only be used by one thread at a time.
- [Hold and Wait] at least one thread holds a resource and is waiting for other resource(s) to become available. A different thread holds the resource(s).
- [No Preemption] a thread can only release a resource voluntarily; another thread or the OS cannot force the thread to release the resource.
- [Circular Wait] must exist a set of threads t_1, t_2, \dots, t_n where t_1 is waiting on t_2 , t_2 is waiting on t_3 , ..., and t_n is waiting on t_1 .

How to handle Deadlock

- ***Prevention***
 - statically make deadlocks structurally impossible
- ***Avoidance***
 - avoid deadlocks by allocating resources carefully (run-time)
 - “compromise” between “prevention” and “detection and recovery”
- ***Detection and Recovery***
 - let deadlocks occur, detect them, and try to recover
- **Ignore the problem (ostrichizing) -- most OSs**
 - It’s an “application-level” concern

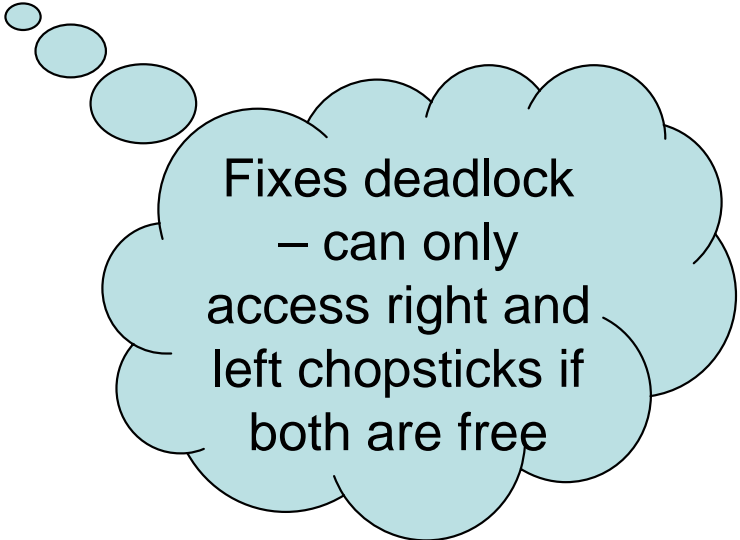
Second Solution

- def eat
 - @status = "waiting"
 - simultaneous_wait(@leftForLock, @rightForLock)
 - @status = "eating"
 - simultaneous_free(@left, @right)
- end
- What's the problem?

Second Solution

- def eat
 @status = "waiting"
 simultaneous_wait(@leftForLock, @rightForLock)
 @status = "eating"
 simultaneous_free(@left, @right)
end

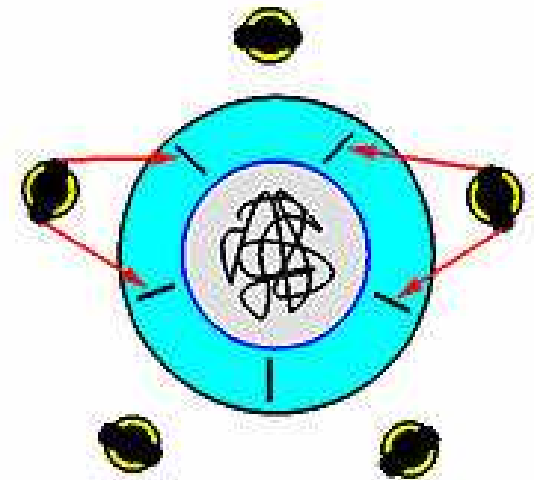
- What's the problem?



Fixes deadlock
– can only
access right and
left chopsticks if
both are free

Second Soln Problem - Starvation

- def eat
 - @status = "waiting"
 - simultaneous_wait(@leftForLock, @rightForLock)
 - @status = "eating"
 - simultaneous_free(@left, @right)
- end



Starvation

- Occurs if a philosopher is unable to acquire both forks due to a timing issue.
- Deadlock **IS** a starvation problem
(but not all starvations are deadlock)
- A thread waits indefinitely (for example, because some other threads are using a resource)
- This happens when shared resources are made unavailable for long periods by "greedy" threads.

Starvation

- Imagine that two philosophers are fast thinkers and fast eaters. They think fast and get hungry fast.
- Then, they sit down in opposite chairs as shown below.
- Because they are so fast, it is possible that they can lock their chopsticks and eat.
- After finish eating and before their neighbors can lock the chopsticks and eat, they come back again and lock the chopsticks and eat.
- In this case, the other three philosophers, even though they have been sitting for a long time, they have no chance to eat.
- This is a ***starvation***.
- Note that it is not a deadlock because there is no circular waiting, and every one has a chance to eat!

So What?

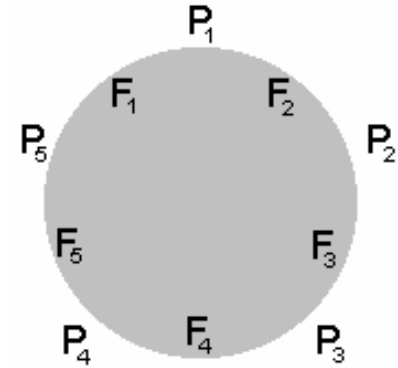
So What?

- Example of a common computing problem in concurrency.
- The lack of available chopsticks is an analogy to the locking of shared resources in real computer programming
- **Concurrency** = is a property of systems which consist of computations that execute overlapped in time, and which may permit the sharing of common resources between those overlapped computations.
- Common in multi-process synchronization

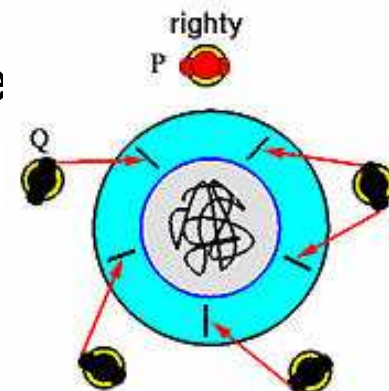
So What?

- Locking a resource is a common technique to ensure the resource is accessed by only one program or chunk of code at a time.
- When the resource the program is interested in is already locked by another one, the program waits until it is unlocked.
- When several programs are involved in locking resources, deadlock might happen, depending on the circumstances.
- For example, one program needs two files to process. When two such programs lock one file each, both programs wait for the other one to unlock the other file, which will never happen.

Dijkstra's Solution



- Positions of philosophers and forks in ordering solution.
- One solution is to order the forks and require the philosophers to pick up the forks in increasing order.
- To illustrate this solution, label the philosophers P₁, P₂, P₃, P₄, and P₅, and label the forks F₁, F₂, F₃, F₄, and F₅.
- Each philosopher must pick up forks in a prescribed order and cannot pick up a fork another philosopher already has.
- Upon acquiring two forks, a philosopher may eat.
- Philosophers P₁ through P₄ follow the rule that P_x must pick up fork F_x first and then may pick up fork F_{x+1}. For example, P₁ must pick up F₁ first and F₂ second.
- Philosopher P₅ follows a slightly different rule, picking up fork F₁ before picking up fork F₅. If P₅ followed the same rule as the others, F₅ would come before F₁. This change in behavior for P₅ creates an asymmetry that prevents deadlock.



Another Solution

- **Chandy / Misra Solution**
- allow for arbitrary agents (numbered P_1, \dots, P_n) to contend for an arbitrary number of resources (numbered R_1, \dots, R_m).
- For every pair of philosophers contending for a resource, create a fork and give it to the philosopher with the lower ID. 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.
- This solution also allows for a large degree of concurrency.