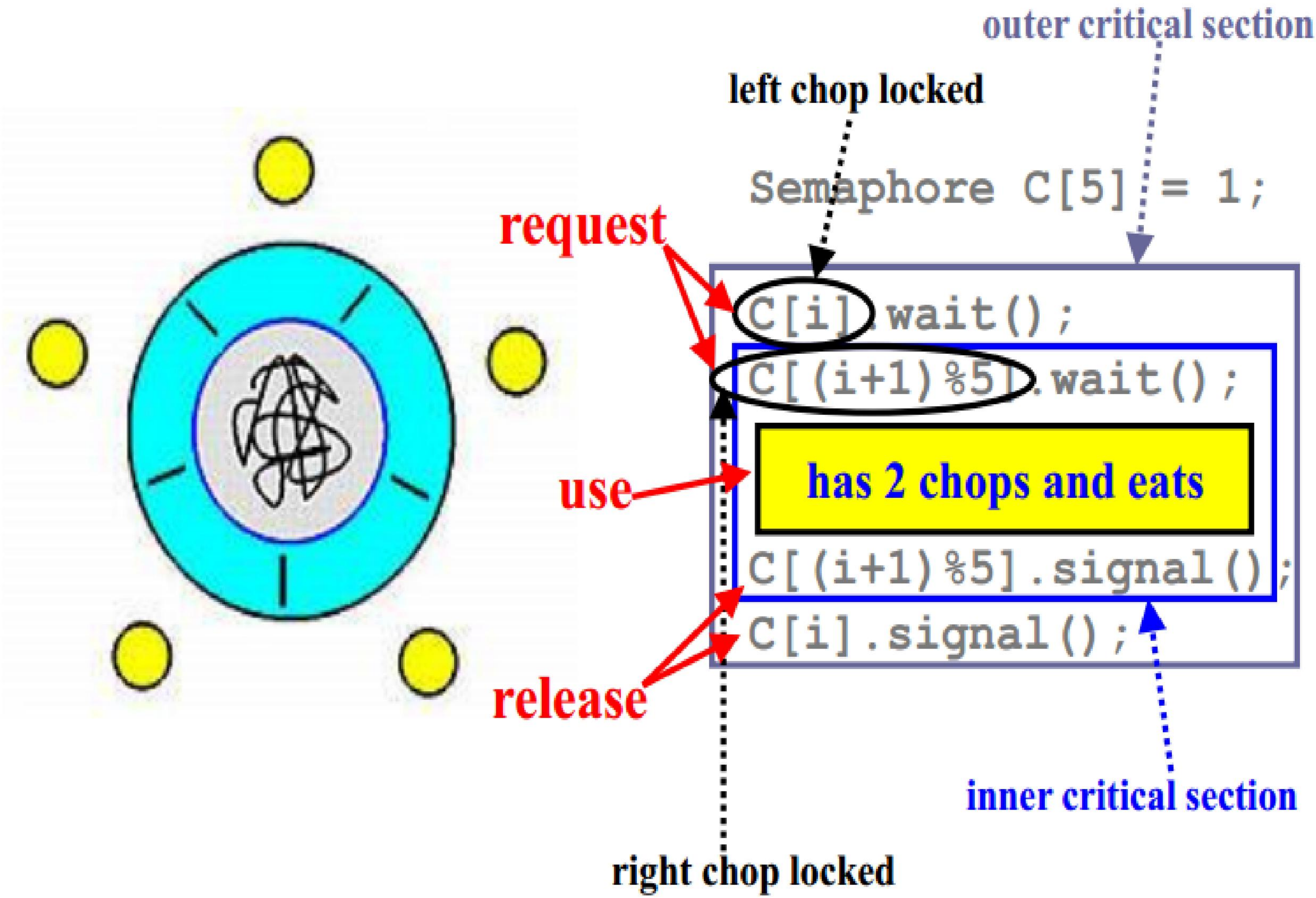
Synchronization -
DeadLocks and LiveLocks

- System resources are used in the following way:
- Request: If a process makes a request (i.e., semaphore wait or monitor acquire) to use a system resource which cannot be granted immediately, then the requesting process blocks until it can acquire the resource successfully.
- Use: The process operates on the resource (i.e., in critical section).
- Release: The process releases the resource (i.e., semaphore signal or monitor release).



outer critical section

left chop locked

```
Semaphore C[5] = 1;

C[i].wait();
C[(i+1)%5].wait();
```

request

use

has 2 chops and eats

```
C[(i+1)%5].signal();
C[i].signal();
```

release

inner critical section

right chop locked

Deadlock: Definition

- A set of processes is in a **deadlock** state when every process in the set is waiting for an event that can only be caused by another process in the same set.
- The key here is that processes are all in the **waiting state.**

Deadlock Necessary Conditions

- If a deadlock occurs, then each of the following four conditions must hold .
  - **Mutual Exclusion:** At least one resource must be held in a non-sharable way.
  - **Hold and Wait**: A process must be holding a resource and waiting for another.
  - **No Preemption**: Resource cannot be Preempted.
  - **Circular Waiting**: P_1 waits for P_2, P_2 waits for P_3, …. P_{n-1} waits for P_n and P_n waits for P_1.

Deadlock Necessary Conditions

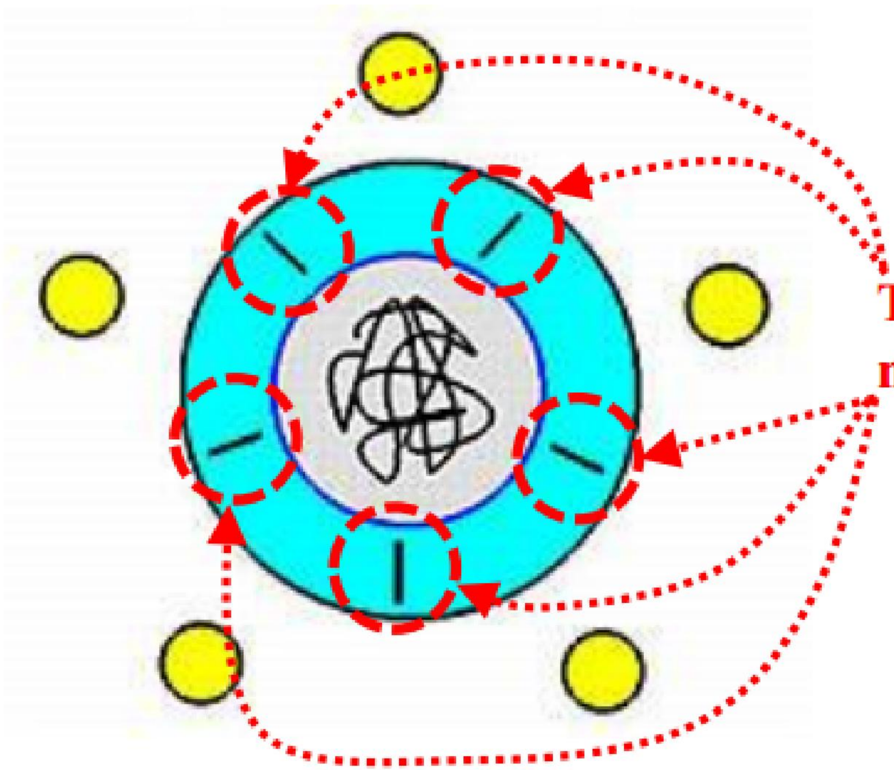
- Note that the conditions are necessary.
- This means if a deadlock occurs ALL conditions are met.
- Since $p \Rightarrow q$ is equivalent to $\neg q \Rightarrow \neg p$, where $\neg q$ means not all conditions are met and $\neg p$ means no deadlock, as long as one of the four conditions fails there will be no deadlock.

Deadlock Prevention

- **Deadlock Prevention** means making sure deadlocks never occur.
- To this end, if we are able to make sure at least one of the four conditions fails, there will be no deadlock.

Deadlock Prevention
Mutual Exclusion

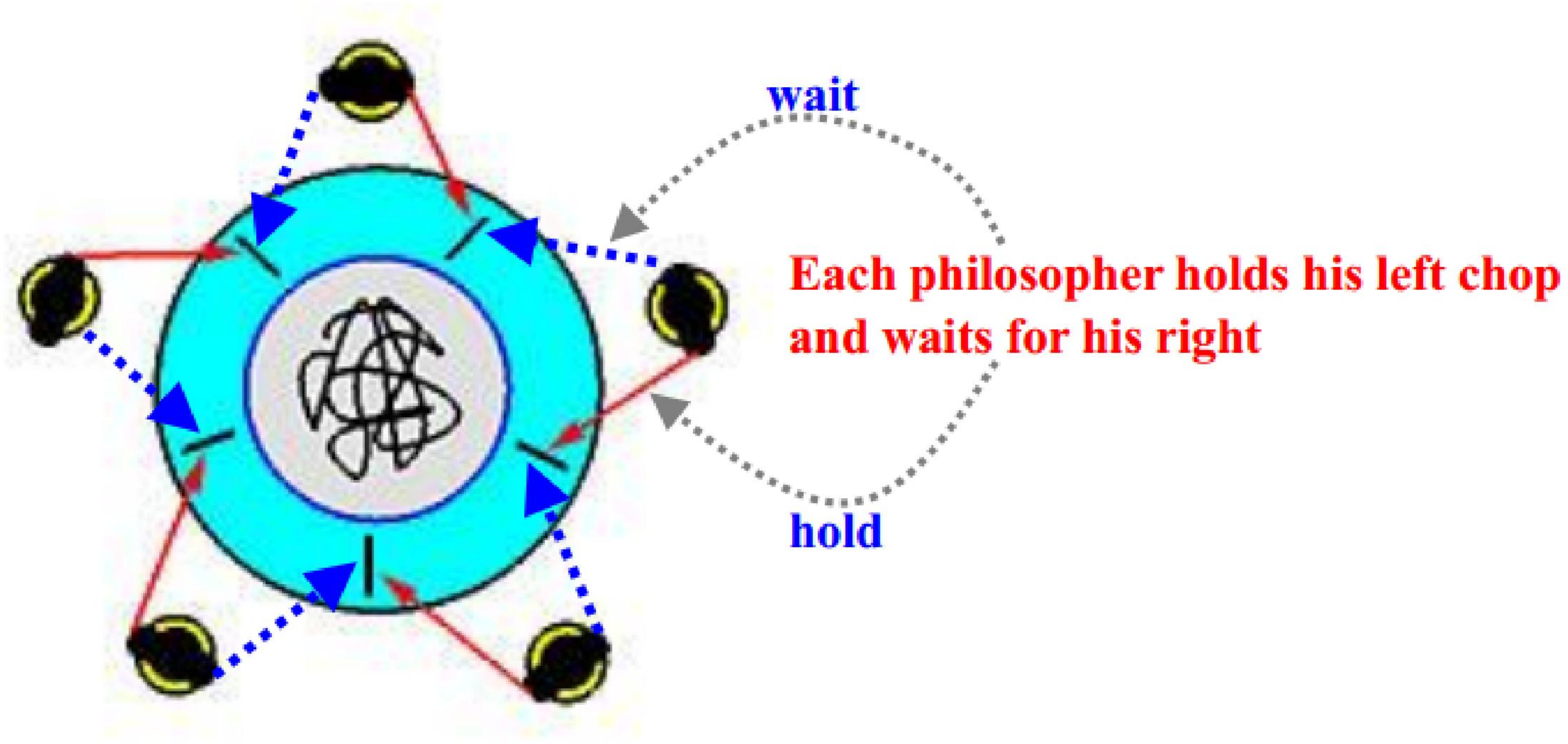
- **Mutual Exclusion:** Some sharable resources must be accessed exclusively, which means we cannot deny the mutual exclusion condition.



The use of these five chopsticks must be mutually exclusive

Deadlock Prevention
Hold and Wait

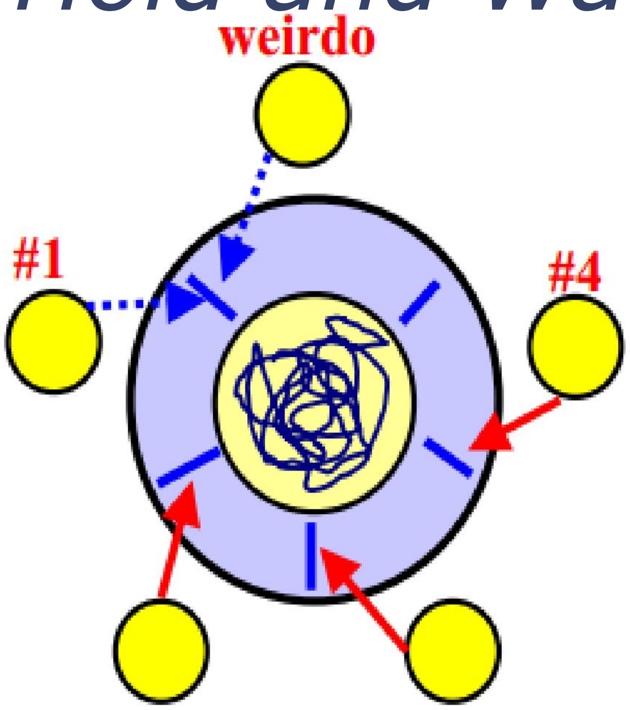
- **Hold and Wait:** A process holds some resources and requests for other resources


wait
Each philosopher holds his left chop and waits for his right
hold

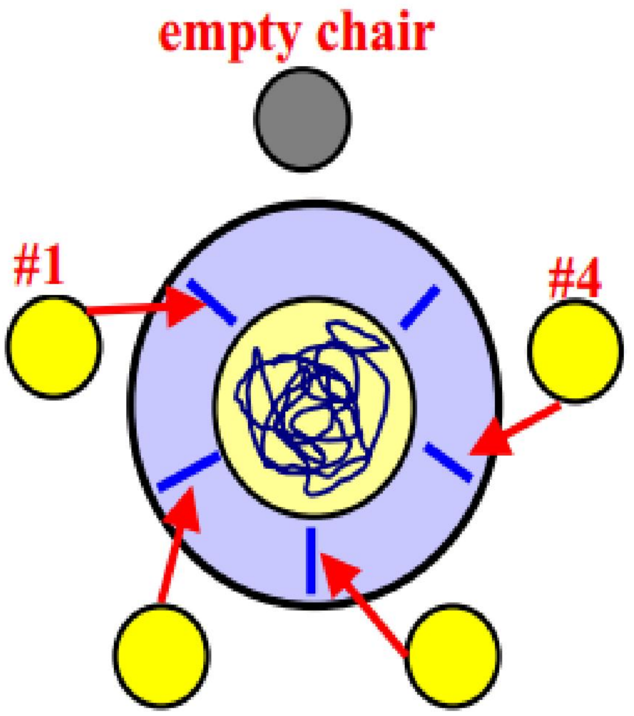
Deadlock Prevention
Hold and Wait

- **Solution:** Make sure no process can hold some resources and then request for other resources.
- Two strategies are possible (the monitor solution to
- the philosophers problem):
  - A process must acquire all resources before it runs.
  - When a process requests for resources, it must hold none (i.e., returning resources before requesting for more).
- **Resource utilization** may be low, since many resources will be held and unused for a long time.
- **Starvation** is possible. A process that needs some popular resources may have to wait indefinitely.

Deadlock Prevention
Hold and Wait



If weirdo is faster than #1, #1 cannot eat and the weirdo or #4 can eat but not both.
If weirdo is slower than #1, #4 can eat
Since there is no hold and wait, there is no deadlock.



In this case, #4 has no right neighbor and can take his right chop.
Since there is no hold and wait, there is no deadlock.

The monitor solution with `THINKING-HUNGRY-EATING` states forces a philosopher to have both chops before eating. Hence, no hold-and-wait.