Part III

Synchronization

Deadlocks and Livelocks

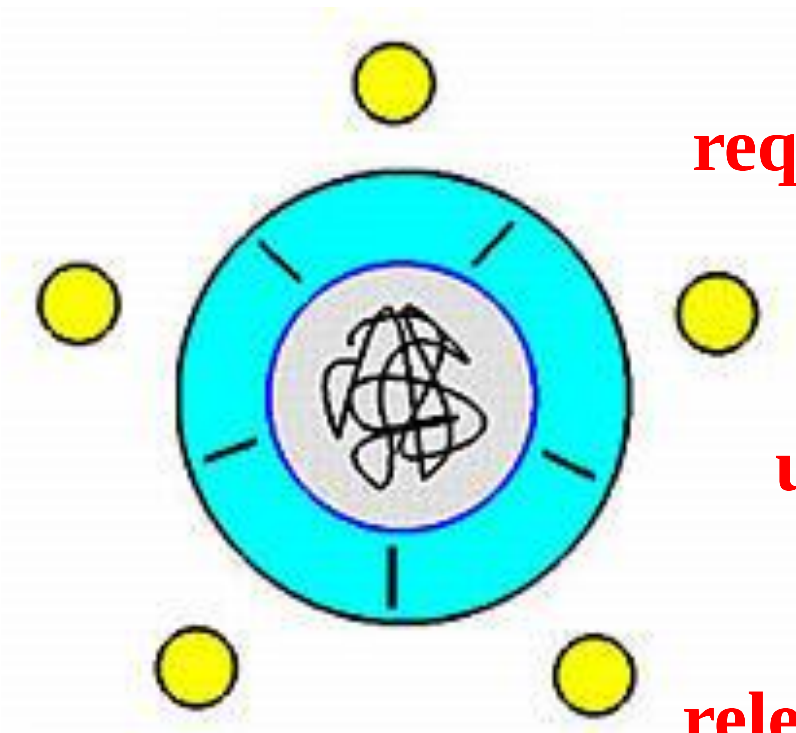
You think you know when you learn,
are more sure when you can write,
even more when you can teach,
but certain when you can program.

Alan J. Perlis

System Model: 1/2

- 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).

System Model: 2/2



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

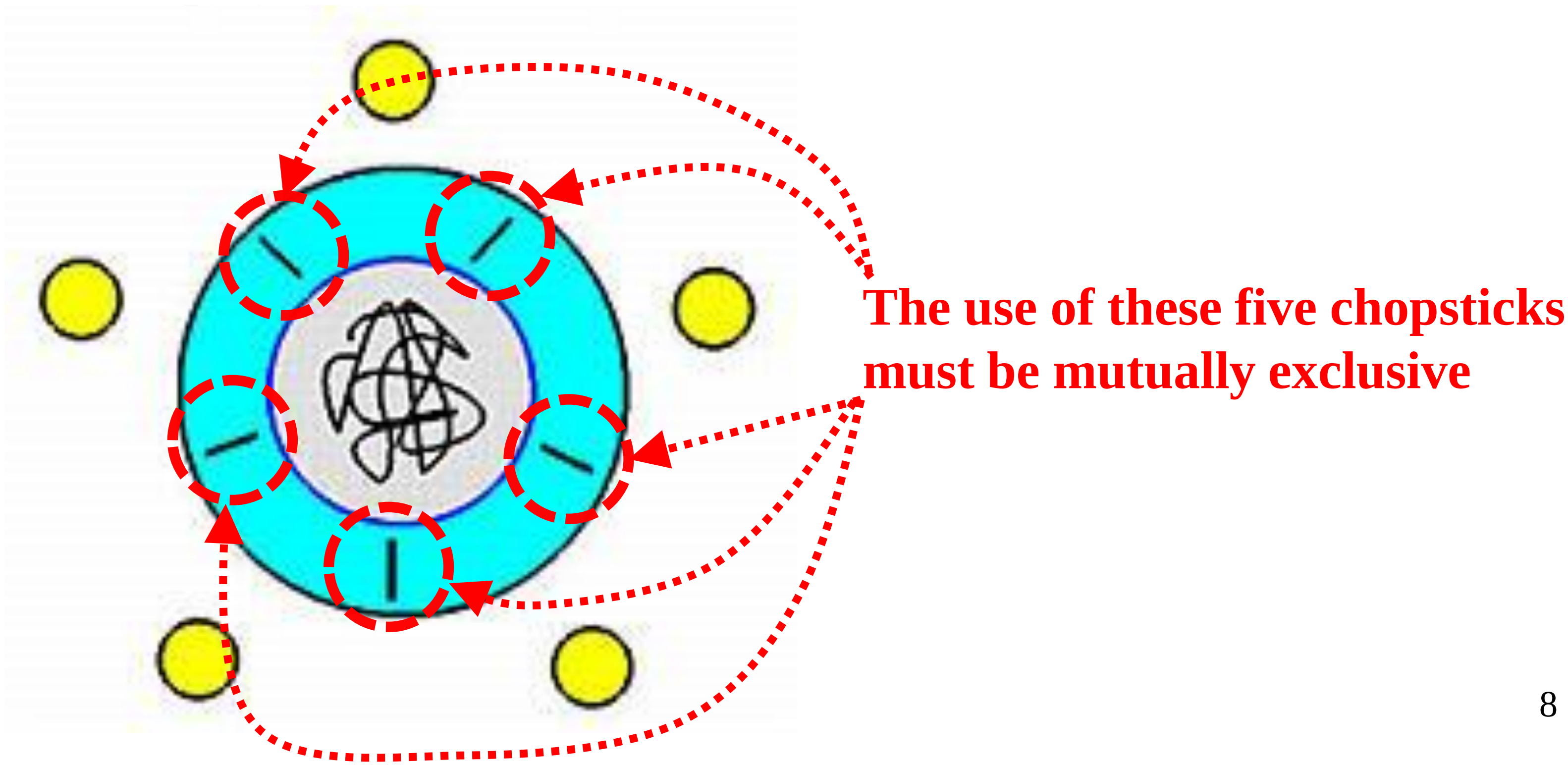
- ***Note that the conditions are necessary.***
- ***This means if a deadlock occurs ALL conditions are met.***
- ***Because $p \Rightarrow q$ is equivalent to $\neg q \Rightarrow \neg p$, where $\neg q$ means not all conditions are met and $\neg p$ means no deadlock, as long as one of the four conditions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

Deadlock Prevention: 1/7

- ***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.**

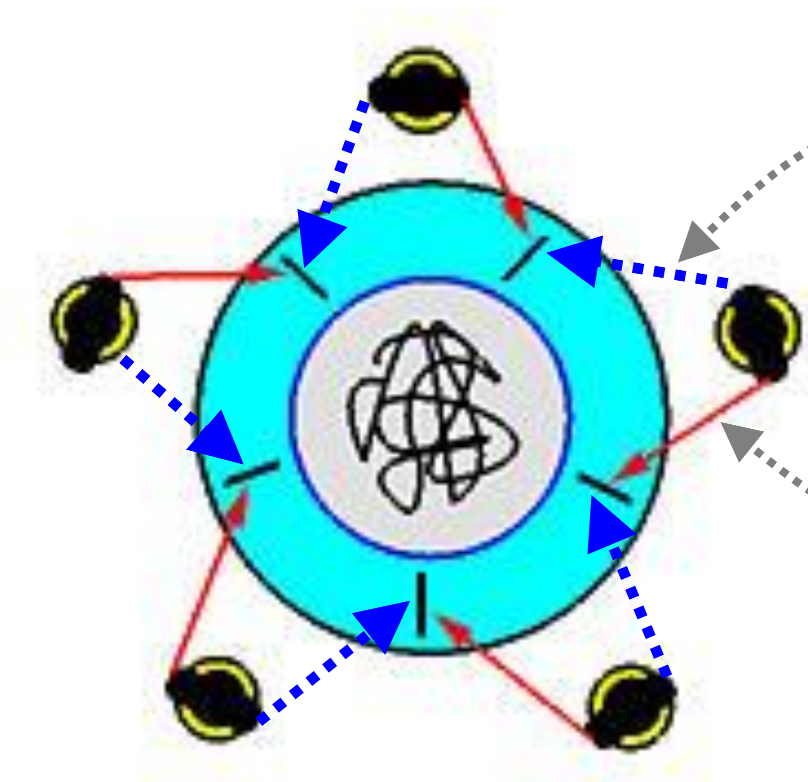
Deadlock Prevention: 2/7
Mutual Exclusion

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



Deadlock Prevention: 3/7
Hold and Wait

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



wait

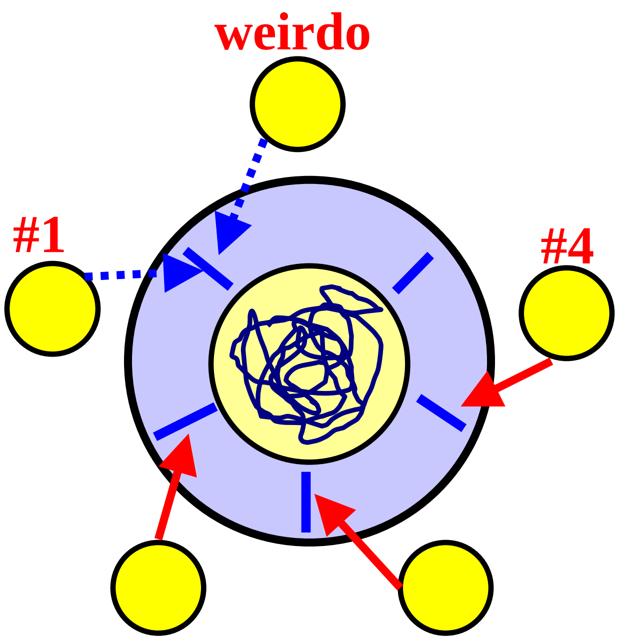
Each philosopher holds his left chop and waits for his right

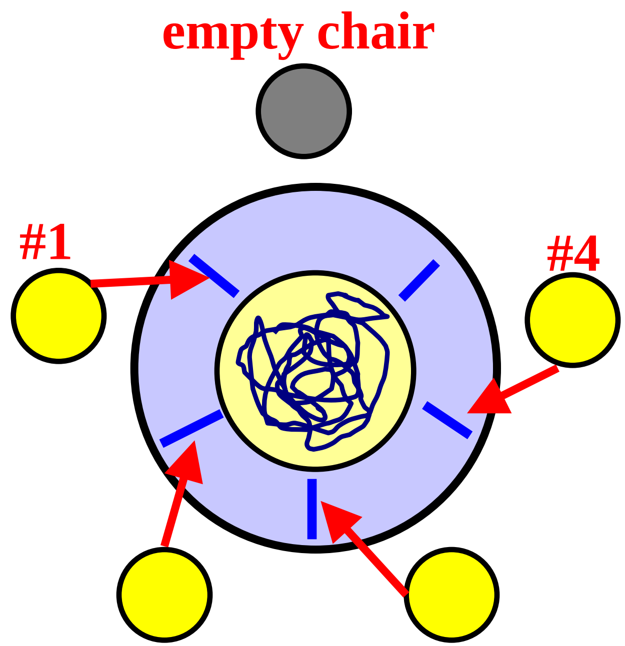
hold

Deadlock Prevention: 4/7
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 my have to wait indefinitely.**

Deadlock Prevention: 5/7
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.

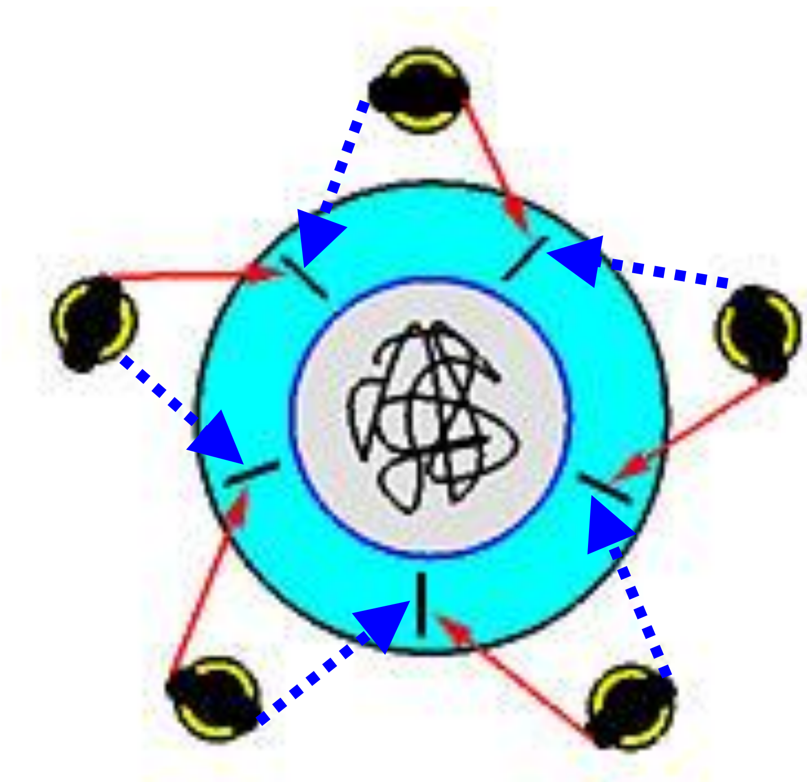
The monitor solution with `THINKING-HUNGRY-EATING` states forces a philosopher to have both chops before eating. Hence, no hold-and-wait.

Deadlock Prevention: 6/7
No Preemption

- This means resources being held by a process cannot be taken away (i.e., no preemption).
- To negate this no preemption condition, a process may deallocate all resources it holds so that the other processes can use.
- This is sometimes not doable. For example, while philosopher *i* is eating, his neighbors cannot take *i*'s chops away forcing *i* to stop eating.
- Moreover, some resources cannot be reproduced cheaply (e.g., printer).

Deadlock Prevention: 7/7
Circular Waiting

- ***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.



The weirdo, 4-chair, and monitor solutions all avoid circular waiting and there is no deadlock.

Resources can be ordered in a hierarchical way.
A process can only acquire resources higher than those it has.
To acquire lower order resources a process must release all resources higher than or equal to that of the acquiring one.
As a result, no deadlock can happen.
Prove this yourself.

Livelock: 1/3

- ***Livelock*: If two or more processes continually repeat the same interaction in response to changes in the other processes without doing any useful work.**
- **These processes are *not* in the waiting state, and they are running concurrently.**
- **This is different from a deadlock because in a deadlock all processes are in the waiting state.**

Livelock: 2/3

```
MutexLock Mutex1, Mutex2;

Mutex1.Lock();                    // lock Mutex1
while (Mutex2.isLocked()) {       // loop until Mutex2 is open
   Mutex1.Unlock();               //    release Mutex1 (yield)
   // wait for a while            //    wait for a while
   Mutex1.Lock();                 //    reacquire Mutex1
}                                 // OK, Mutex2 is open
Mutex2.Lock();                    // lock Mutex2. have both

Mutex2.Lock();
while (Mutex1.isLocked()) {
   Mutex2.Unlock();
   // wait for a while
   Mutex2.Lock();
}
Mutex1.Lock();
```

Both processes try to acquire two locks and they yield to each other

Livelock: 3/3

- **Process 1 locks `Mutex1` first. If `Mutex2` is not locked, process 1 acquires it. Otherwise, process 1 yields `Mutex1`, waits for a while (for process 2 to take `Mutex1` and finish its task), reacquires `Mutex1`, and checks again `Mutex2` is open.**
- **Process 2 does this sequence the same way with the role of `Mutex1` and `Mutex2` switched.**
- **To avoid this type of livelock, *order the locking sequence in a hierarchical way* (i.e., both lock `Mutex1` *first* followed by `Mutex2`). Thus, only one process can lock both locks successfully.**

The End