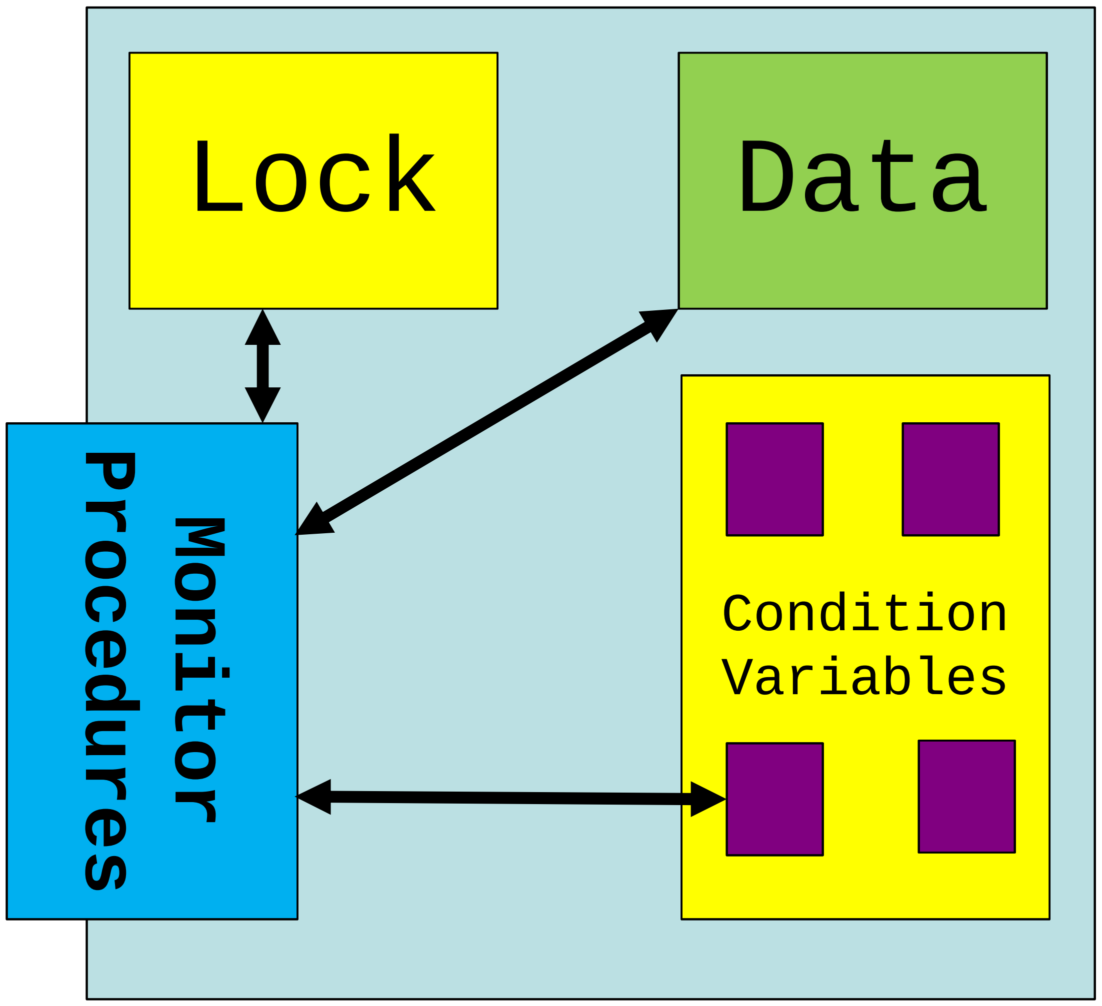
# G52OSC
# OPERATING SYSTEMS AND CONCURRENCY

## ThreadPools and Problems

Dr Jason Atkin

# Last lecture: Monitors

- Private data
- Public methods
- Locks
- Condition variables



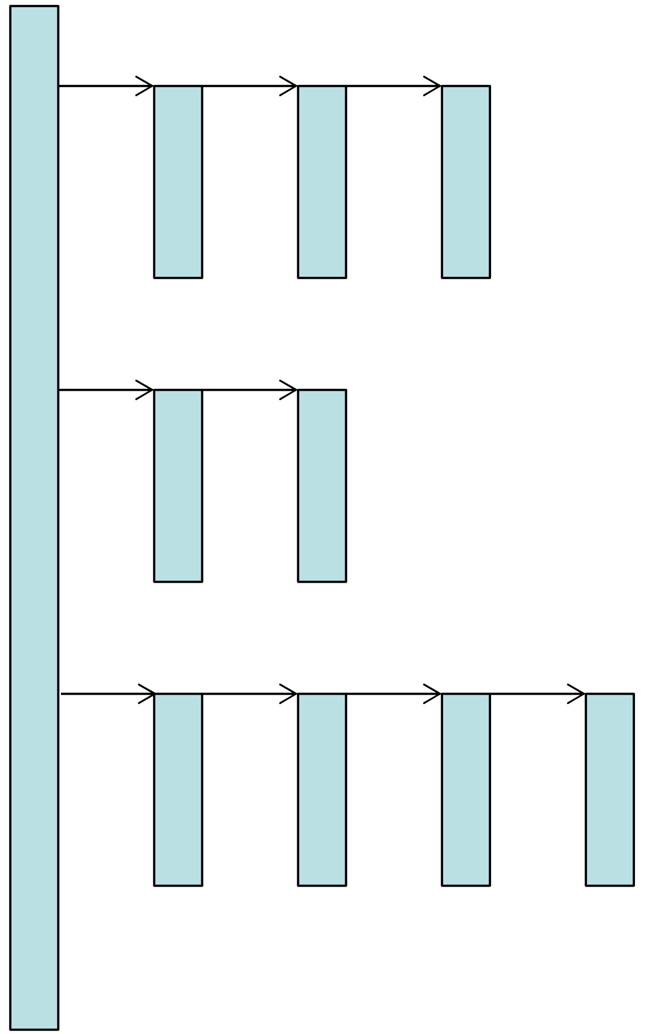
# This Lecture

- ThreadPools
  - And Futures
- Dining Philosophers Problem

# Threadpools

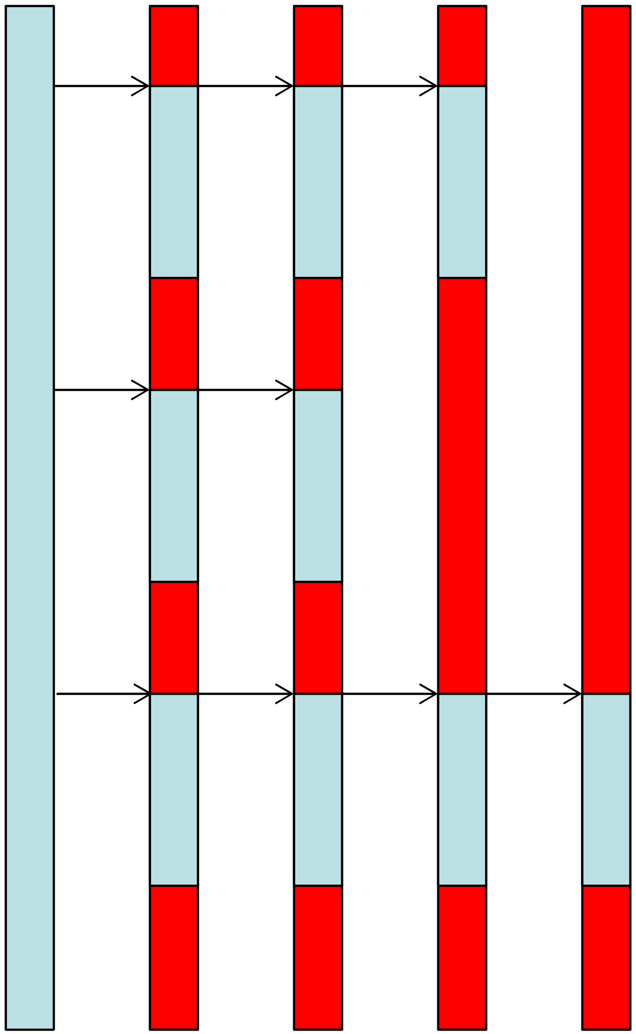
- It is common for us to need to do things in parallel for a short time
- Then to do so again
- And again
- And again…

- But creating threads has an overhead



# Threadpools

- Threadpools are pools of threads which are available for you to use
- They will run the functions for you
  - At some point, when they get around to it
- And block when not running something
  - No CPU usage
- Decide how many you need in the pool



# Java threadpool example

- Create the thread pool and specify number of threads

```
ExecutorService executorConsumers =
    Executors.newFixedThreadPool(3);
```

- Or:

```
= newCachedThreadPool()
= newSingleThreadExecutor()
```

etc (there are others, see **Executors** class)

- **Then use the executor service to execute a runnable:**

```
executorConsumers.execute( myRunnable );
executorConsumers.execute( new Runnable()
      { public void run() { doConsumer(); } } );
```

# C/Windows API threadpool example

As usual, using C and the windows API is a bit more tricky, but the same principles apply and I have created a sample for you to illustrate this.

- Create the threadpool:

```
PTP_POOL pool = CreateThreadpool( NULL );
```

- Specify number of threads to run:

```
SetThreadpoolThreadMinimum( pool,1 );
SetThreadpoolThreadMaximum( pool,4 );
```

- Tell it to run the function:

```
SubmitThreadpoolWork( work[i] );
```

- If you want to know when some work finishes:

```
WaitForThreadpoolWorkCallbacks( work[i],FALSE );
```

- Close it the work when you have finished with it (before or after it runs):

```
CloseThreadpoolWork( work[i] );
```

- And close the threadpool when you will not use it any more:

```
CloseThreadpool( pool );
```

# ThreadPool Environment and Work

- Create the pool
- Associate environment with it

  InitializeThreadpoolEnvironment()

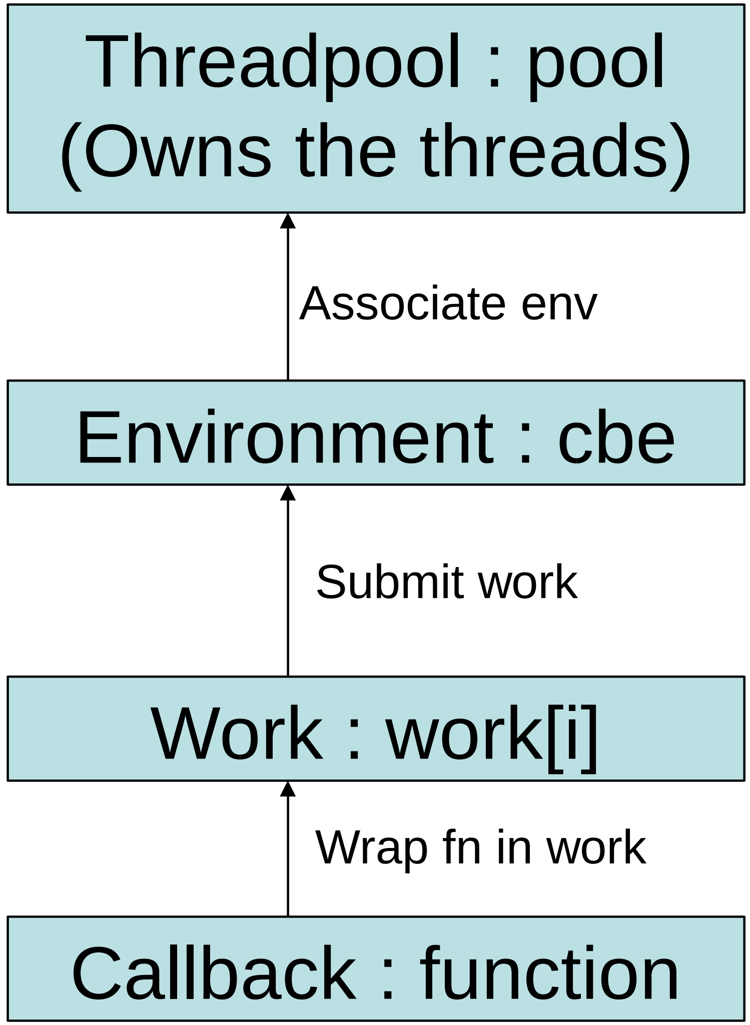
  SetThreadpoolCallbackPool()

- For every function you wish to execute, create a work object to wrap it up and associate it with the environment

  CreateThreadpoolWork(…)

- Then you submit the work object, not the function:

  SubmitThreadpoolWork( work[i] );



# Back to Java: futures

# Java futures example

- Posting to a threadpool, you may want to know:
  - When has the work finished?
  - What was the result?
- In C you would probably give it a pointer to some result structure to fill in (using the parameter that you can pass to a thread function/work)
- But there are nicer/more elegant ways to do this via futures
  - Java and C++ (since C++11) both support futures
    - So do other languages, or at least the concepts
  - We are ignoring C++ so will look at the Java version

# Submitting the callback

- Create the ThreadPool as usual

```
ExecutorService executor =
      Executors.newSingleThreadExecutor();
```

- Use submit() rather than execute()
  - Give it an object which implements Callable<> (call() function)

```
Future<String> myFutureString =
      executor.submit( MyCallableObject ) );
```

```
Future<String> myFutureString = executor.submit(
      new Callable<String>() {
            public String call() {
                  try {  Thread.sleep(1000); }
                  catch (InterruptedException e) { }
                  return "Finished…";
      } } );
```

# Submitting the callback

- Create the ThreadPool as usual

```
ExecutorService executor =
        Executors.newSingleThreadExecutor();
```

- Use submit() rather than execute()
  - Give it an object which implements Callable<> (call() function)

```
Future<String> myFutureString =
        executor.submit( MyCallableObject ) );
```

```
Future<String> myFutureString = executor.submit(
      new Callable<String>() {
            public String call() {
                  try {  Thread.sleep(1000); }
                  catch (InterruptedException e) { }
                  return "Finished…";
      } } );
```

# Using the Future

- This will create a future and capture various information in the future (for ‘future’ use by you)
- The future is returned from the submit() function:
  `Future<String> myFutureString = executor.submit(`

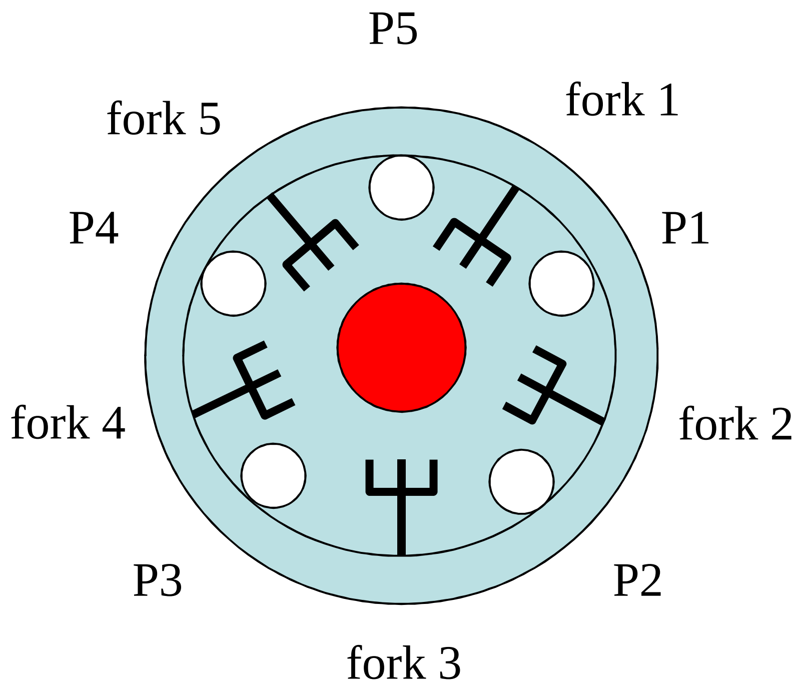
- You can use the future to see what is happening
  - Check whether the function call has completed:
    `myFutureString.isDone()`
  - Retrieve the returned value – type depends on the future type
    `myFutureString.get()`
  - The value from get() will be whatever you `return`ed from the function which was `submit`ted

# Dining Philosophers Problem

# Dining Philosophers Problem

- The *Dining Philosophers* problem illustrates mutual exclusion between processes which compete for overlapping sets of shared variables
  - five philosophers sit around a circular table
  - each philosopher alternately thinks and eats spaghetti from a dish in the middle of the table (think…eat…think…eat)
  - the philosophers can only afford five forks–one fork is placed between each pair of philosophers
  - to eat, a philosopher needs to obtain mutually exclusive access to both the fork on their left and right
- The problem is to avoid *starvation* / deadlock
  - e.g., each philosopher acquires one fork **and refuses to give it up until they have eaten something with it**, can you define a protocol such that all of them will get to each

# Dining Philosophers Problem




Examples: Each tries to take the one on their right then the one on their left, or each tries a random one each time then the other one.
Which policies will work to solve the problem (to the end of the meal)?

# Deadlock in the Dining Philosophers

- The key to the solution is to avoid *deadlock* caused by circular waiting
- E.g. everyone grabs the fork to their right then tries to grab the one on their left:
  - process 1 is waiting for a resource (i.e. a fork) held by process 2
  - process 2 is waiting for a resource held by process 3
  - process 3 is waiting for a resource held by process 4
  - process 4 is waiting for a resource held by process 5
  - process 5 is waiting for a resource held by process 1
- No process can make progress and all processes remain deadlocked

Some slides from 1st year Database Systems module on Concurrency/Deadlocks

# Deadlocks

- A deadlock is an impasse that may result when two or more ~~transactions~~ **threads** are waiting for locks to be released which are held by each other. E.g.:
  - T1 has a lock on X and is waiting for a lock on Y
  - T2 has a lock on Y and is waiting for a lock on X
  - **Both wait for each other**
- We can detect deadlocks that will happen in a schedule using a *wait-for graph* (WFG)
- We are tracking what is ‘waiting for’ something else
  - If there is any **cycle** of things waiting for other things, then they will **deadlock**

# Using Wait-For Graphs

**If you have a specific trace then you can use a wait-for graph to look for deadlock (already occurred)**

**Operating systems and databases do this a lot**

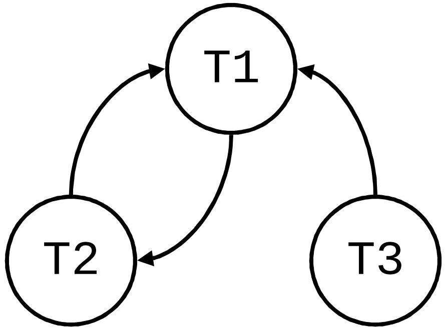
**What you would actually like to know though is whether any trace can result in deadlock later**

## Wait-for Graph

- Aim to find out if we (will) have a deadlock
- Each ~~transaction~~ thread is a vertex
- Looks at locks
- Edge from T2 to T1 if
  - T1 **read-locks** X then T2 tries to **write-lock** it
  - T1 **write-locks** X then T2 tries to **read-lock** it
  - T1 **write-locks** X then T2 tries to **write-lock** it

# Databases: Example

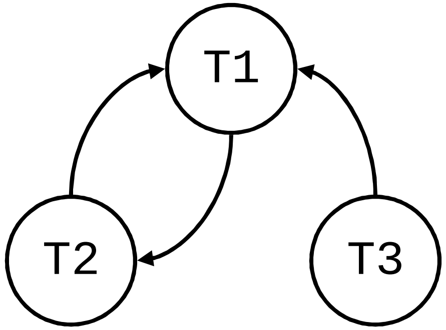
| Schedule | Locks |
|---|---|
| T1 Read(X) | write-lock(X) |
| T2 Read(Y) | read-lock(Y) |
| T1 Write(X) | |
| T2 Read(X) | tries read-lock(X) |
| T3 Read(Z) | write-lock(Z) |
| T3 Write(Z) | |
| T1 Read(Y) | tries write-lock(Y) |
| T3 Read(X) | tries read-lock(X) |
| T1 Write(Y) | |




Wait for graph

# Observations

- To get deadlock (i.e. to have a problem) you need each thread which is involved to:
  - Have a lock that another thread wants
  - Attempt to gain a lock that another thread has




- If you only ever want one lock at a time, this cannot happen

- If threads use different locks/resources you cannot have a problem

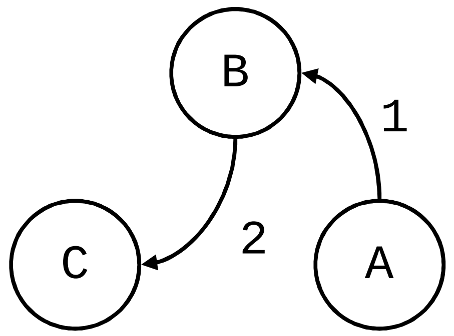
- If you have only one thread it is OK as long as locks have owners

Informal: can we use a graph to help us to spot potential problems in advance?

# INFORMAL similar graph for locks

- Build a graph showing which threads may lock which resources, in which order
- Thread 1:
  - Lock A, Lock B, Unlock B, Unlock A
- Thread 2:
  - Lock B, Lock C, Unlock C, Unlock B
- More complicated rules, so only an informal analysis
  - This is an **informal** way to visualise what you would do with traces

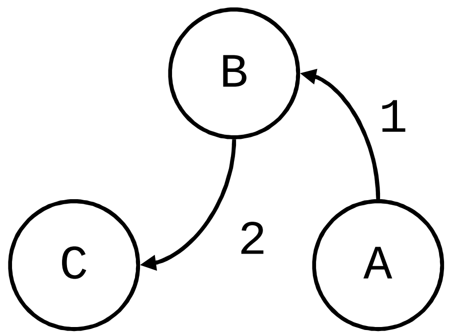
- Switch arcs and vertices
  - Vertices of the graph are the locks
  - Arcs of the graph are the order in which a thread requests the locks



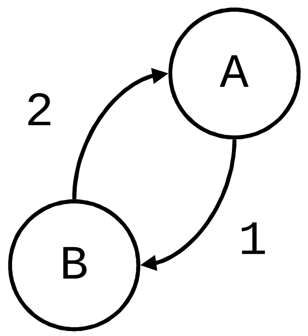

- Note that this transformation makes no difference topologically

# INFORMAL similar graph for locks

- Add the **paths** for the order in which locks can occur for each thread
  - If a process 1 requests lock A then lock B, add an arc from A to B (label it 1)
  - If a process 2 requests lock B, then lock A then add an arc from B to A
- Cycles represent the possibility that deadlock **could** occur **if all of the locks and requests can occur at the same time**



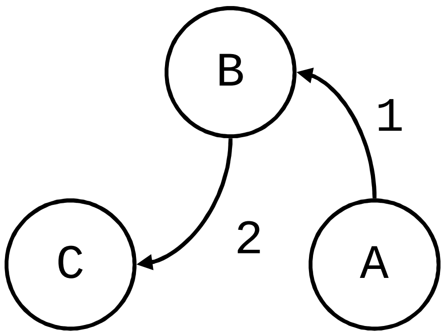

- Classic problem:
  - T1: A then B
  - T2: B then A

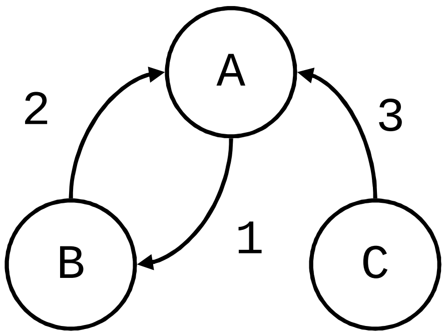


# INFORMAL similar graph for locks

- If there is no cycle there can be no deadlock
- If every thread locks everything in the same order, you cannot possibly get a cycle

- If there is a cycle then you **may** be able to get deadlock (if all of the locks **can occur at once**)
  - **Really you need to check all of the traces** but this may be easier visually

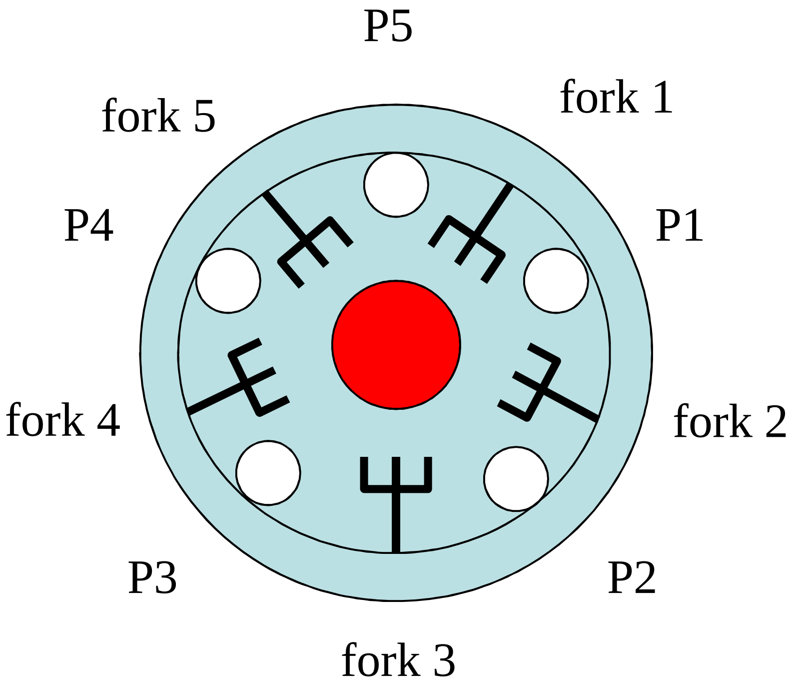


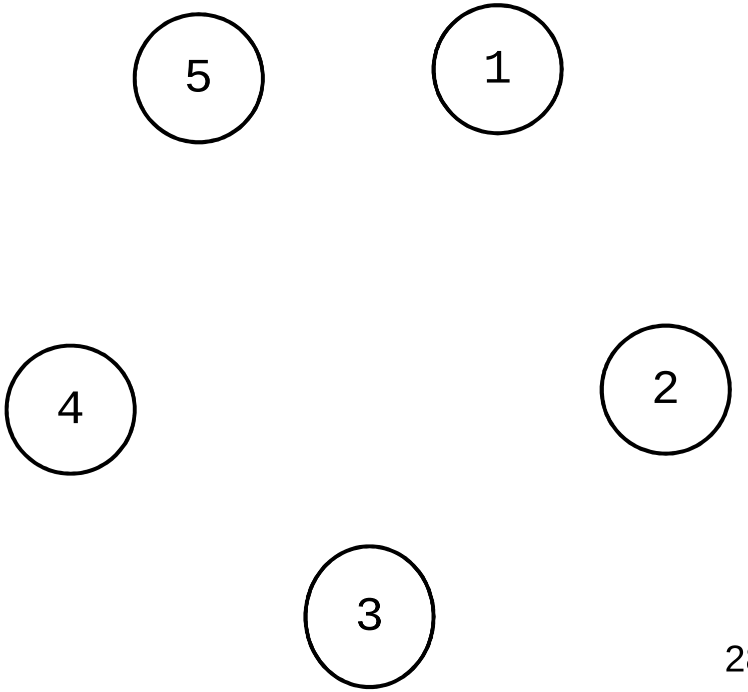



# Applying this to the Dining Philosophers Problem

# Dining Philosophers Problem

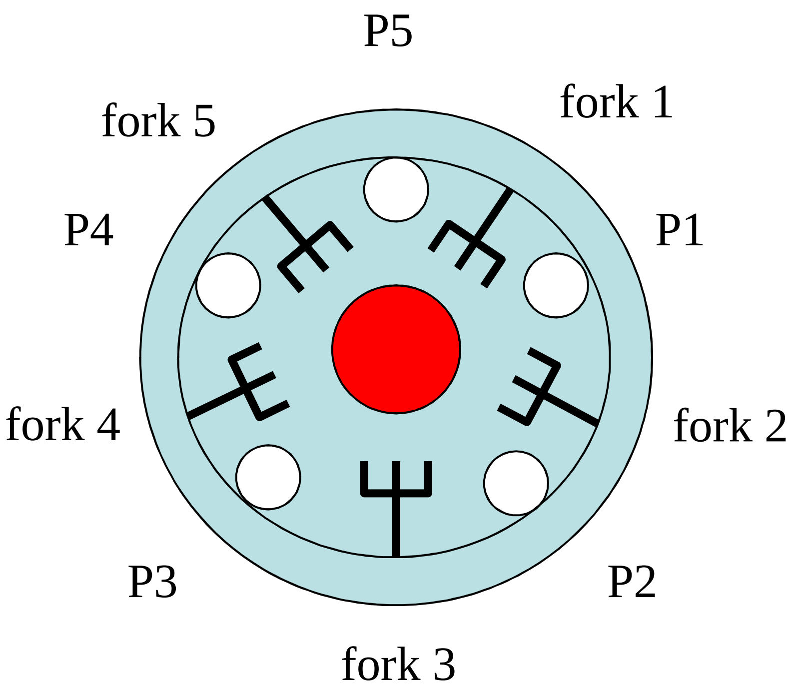
- When we trace the potential paths, what does it look like?
- Philosophers all take the fork on their right then the fork on their left

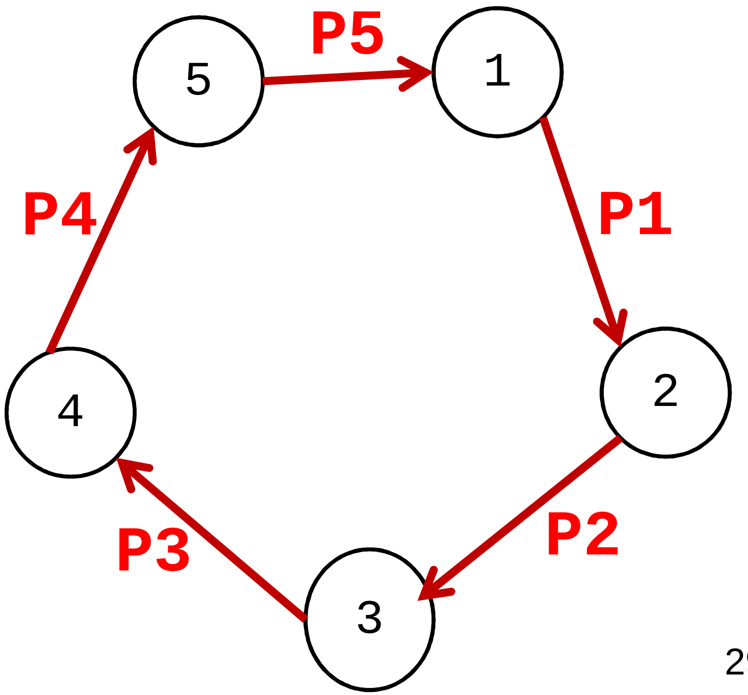


# Dining Philosophers Problem

- Philosophers all take the fork on their right then the fork on their left
- Each gets one fork and has to wait for the other … forever

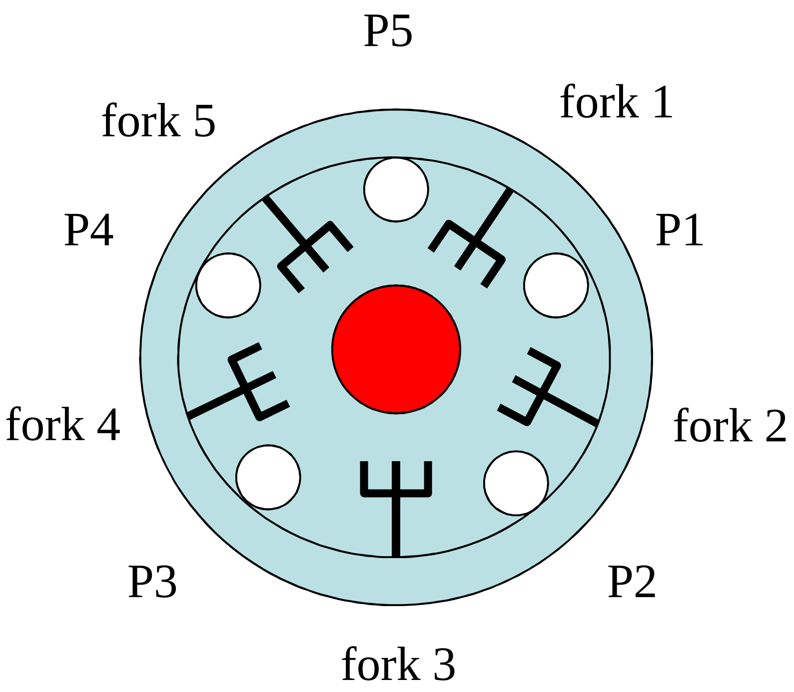


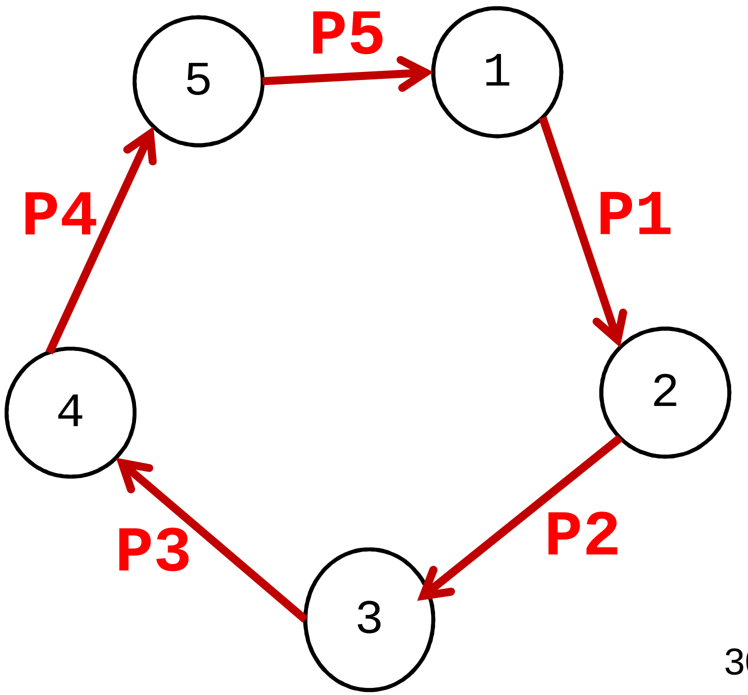



# Dining Philosophers Problem

- Philosophers all take the fork on their right then the fork on their left
- Each gets one fork and has to wait for the other … forever

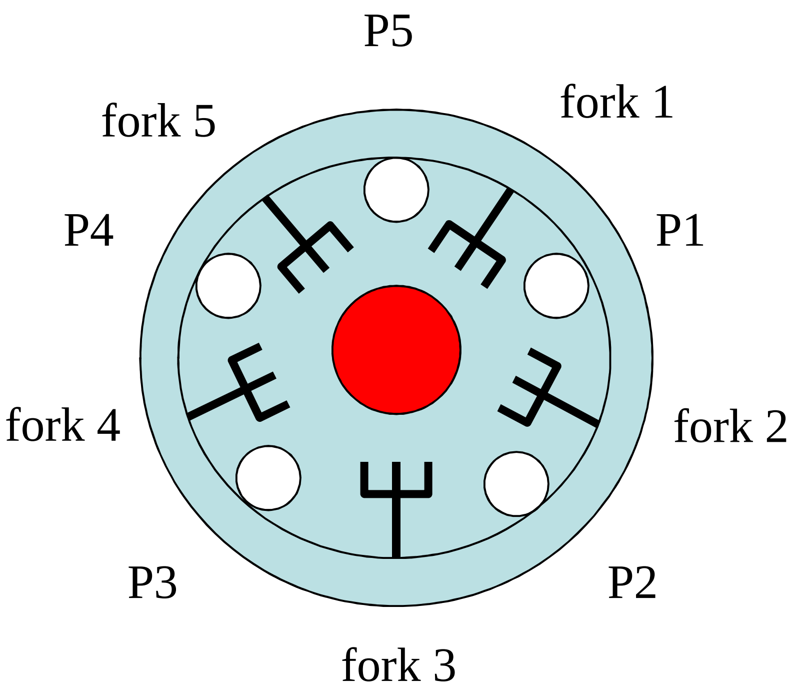
**How could we fix this?**



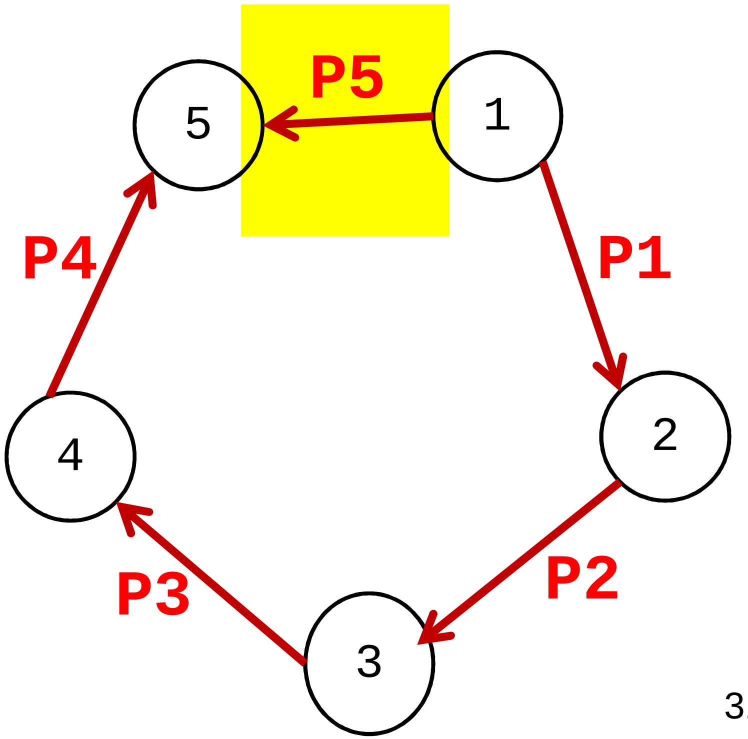



# Dining Philosophers Problem

Switching the order for one philosopher fixes the problem




`Order: 1 2 3 4 5`



# Semaphore Solution

```
// Philosopher i, i == 1-4

while(true) {
    //get right fork then left
    P(fork[i]);
    P(fork[i+1]);
    // eat ...
    V(fork[i]);
    V(fork[i+1]);
    // think ...
}
```

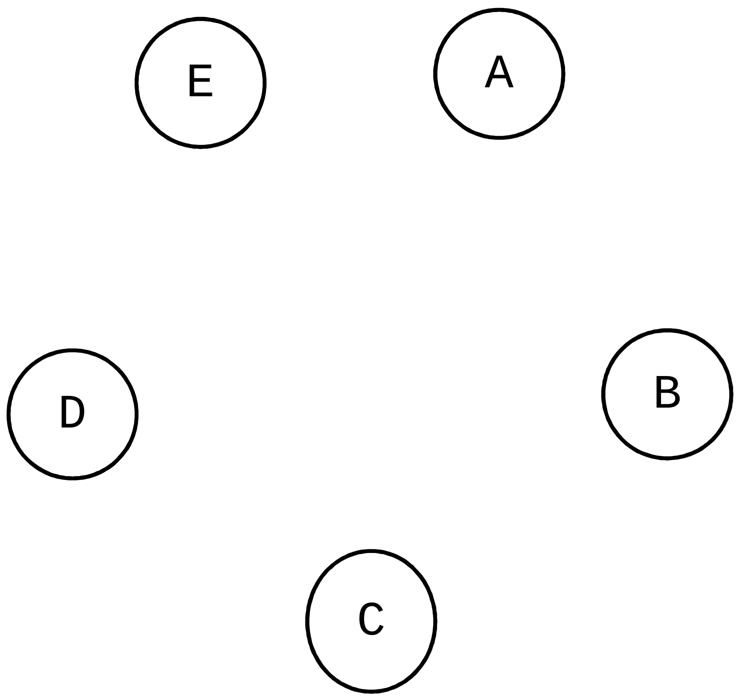
```
// Philosopher 5

while(true) {
    //get left fork then right
    P(fork[1]);
    P(fork[5]);
    // eat ...
    V(fork[1]);
    V(fork[5]);
    // think ...
}
```

```
// Shared variables
binary semaphores fork[5] = {1, 1, 1, 1, 1};
```

# Further examples

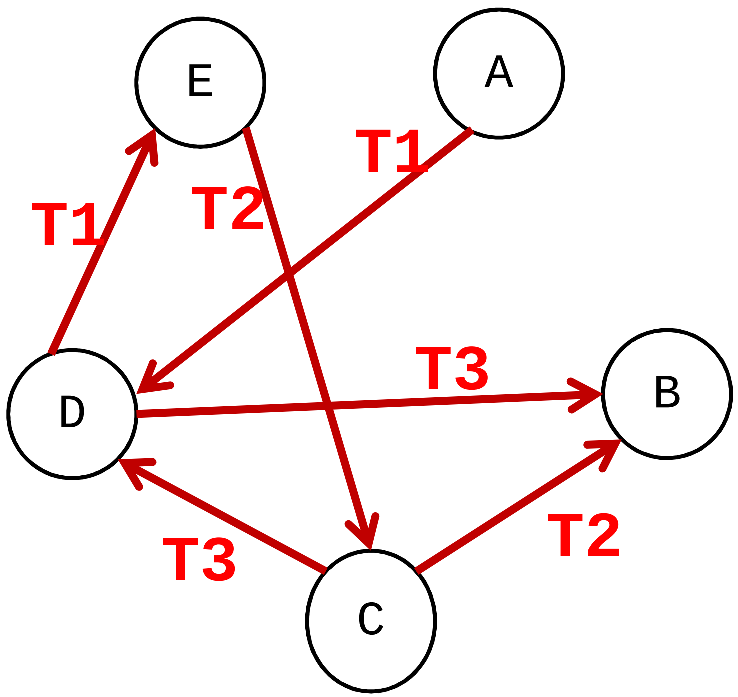
# Example Locks

- T1:
  - Lock A
  - Lock D
  - Lock E
- T2:
  - Lock E
  - Lock C
  - Lock B
- T3:
  - Lock C
  - Lock D
  - Lock B



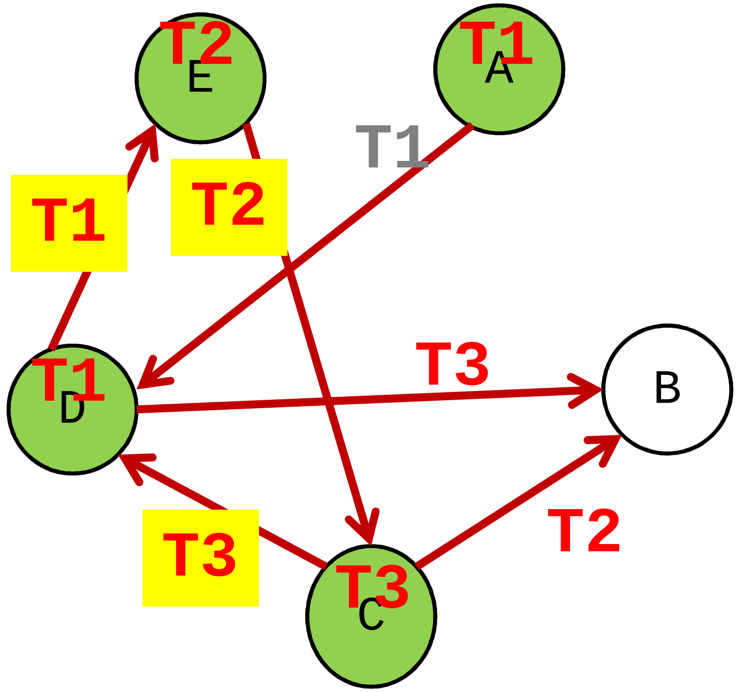
# Can this deadlock?

- T1:
  - Lock A
  - Lock D
  - Lock E
- T2:
  - Lock E
  - Lock C
  - Lock B
- T3:
  - Lock C
  - Lock D
  - Lock B



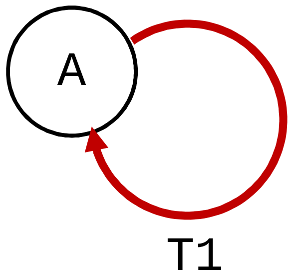
# Can this deadlock?

- T1:
  - Lock A
  - Lock D
  - Lock E
- T2:
  - Lock E
  - Lock C
  - Lock B
- T3:
  - Lock C
  - Lock D
  - Lock B



# Re-entrant locks

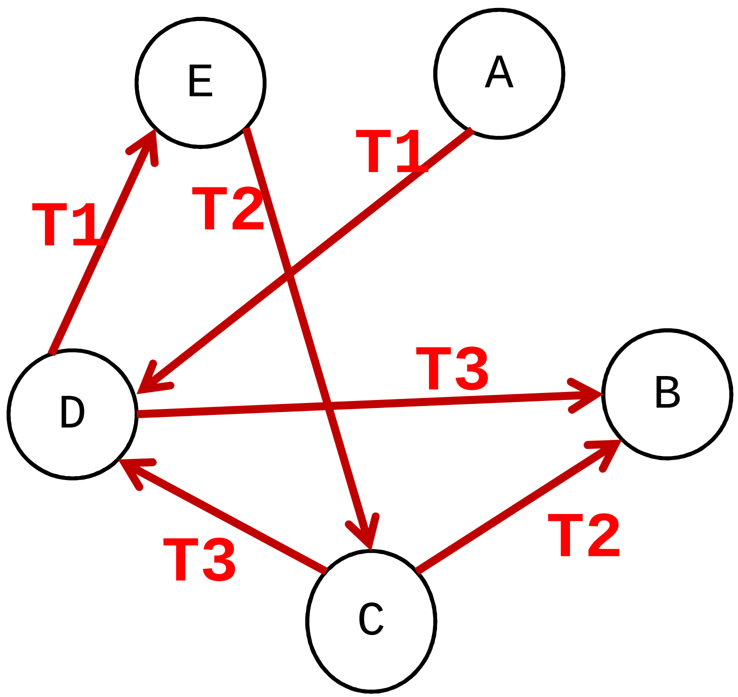
- Re-entrant lock: if you lock it again you still have it
  - E.g. Mutex
- Non-reentrant: If you lock it again you count as a new locker
  - Deadlock if it is a binary lock
  - E.g. binary semaphore



# Returning to example locks

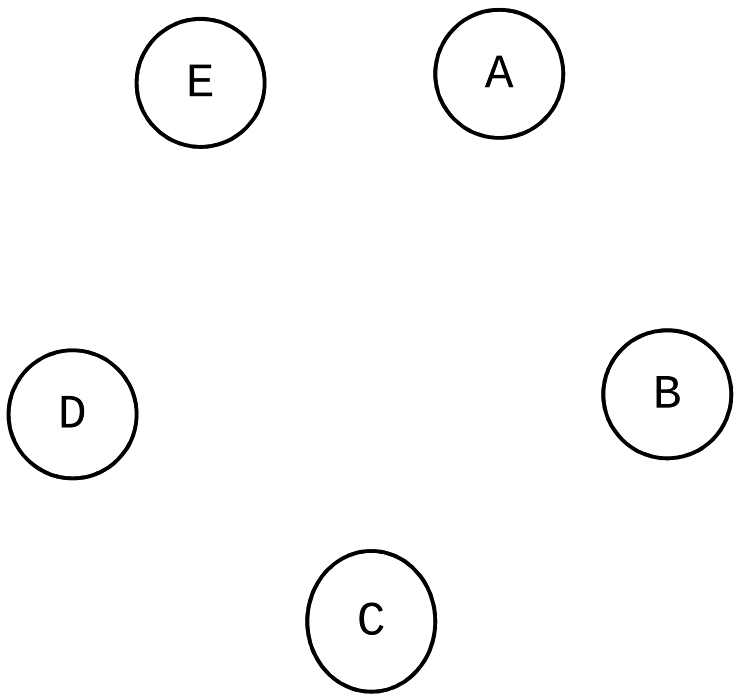
- T1:
  - Lock A
  - Lock D
  - Lock E
- T2:
  - Lock E
  - Lock C
  - Lock B
- T3:
  - Lock C
  - Lock D
  - Lock B

Can we fix this problem?
While still keeping the locks
when necessary?



# Assume reentrant locks

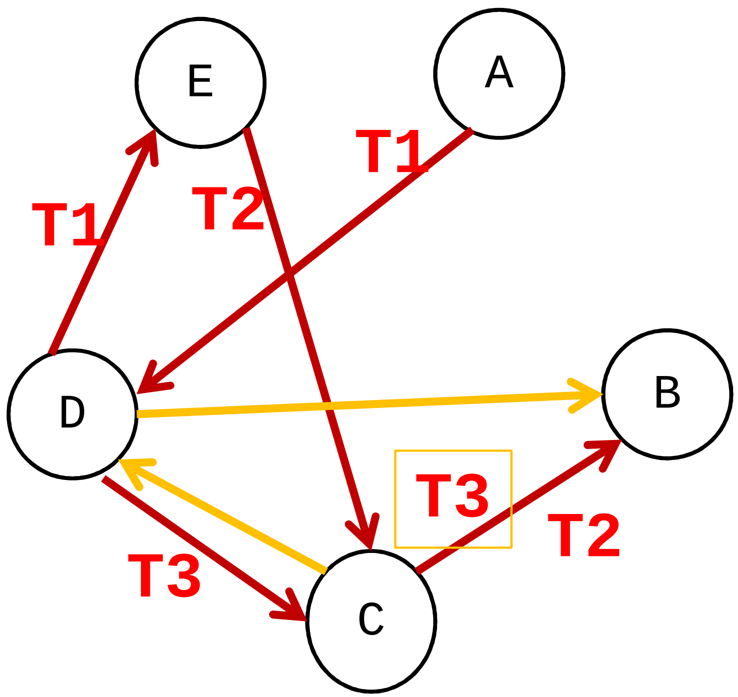
- **Add T3 lock D first**
- T1:
  - Lock A
  - Lock D
  - Lock E
- T2:
  - Lock E
  - Lock C
  - Lock B
- T3:
  - **Lock D**
  - Lock C
  - ~~Lock D~~
  - Lock B




**Order: ?**

# Assume reentrant locks

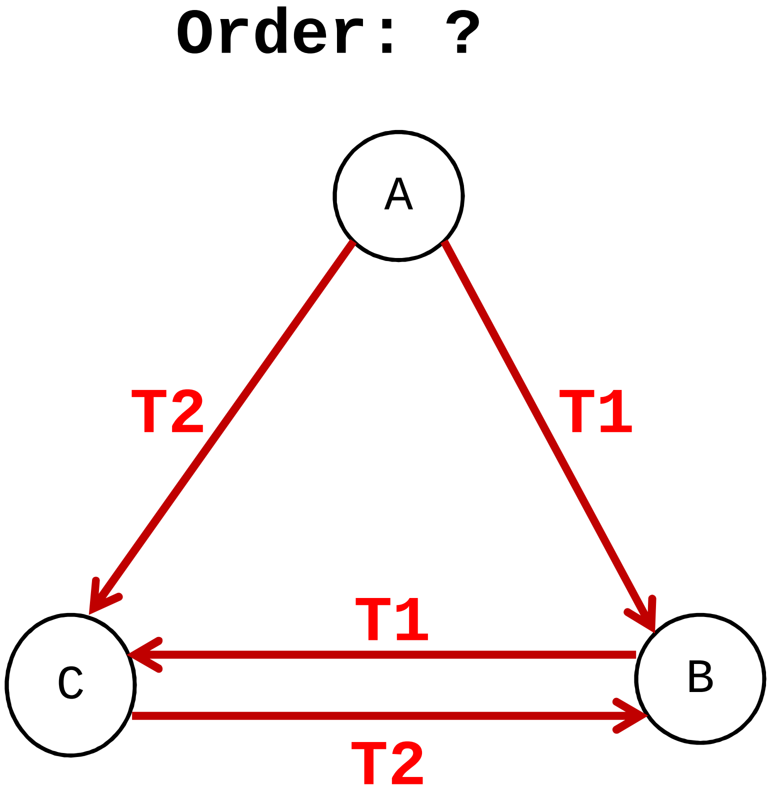
- **Add T3 lock D first**
- T1:
  - Lock A
  - Lock D
  - Lock E
- T2:
  - Lock E
  - Lock C
  - Lock B
- T3:
  - **Lock D**
  - Lock C
  - ~~Lock D~~
  - Lock B




**`Order: A D E C B`**

# More examples (1)

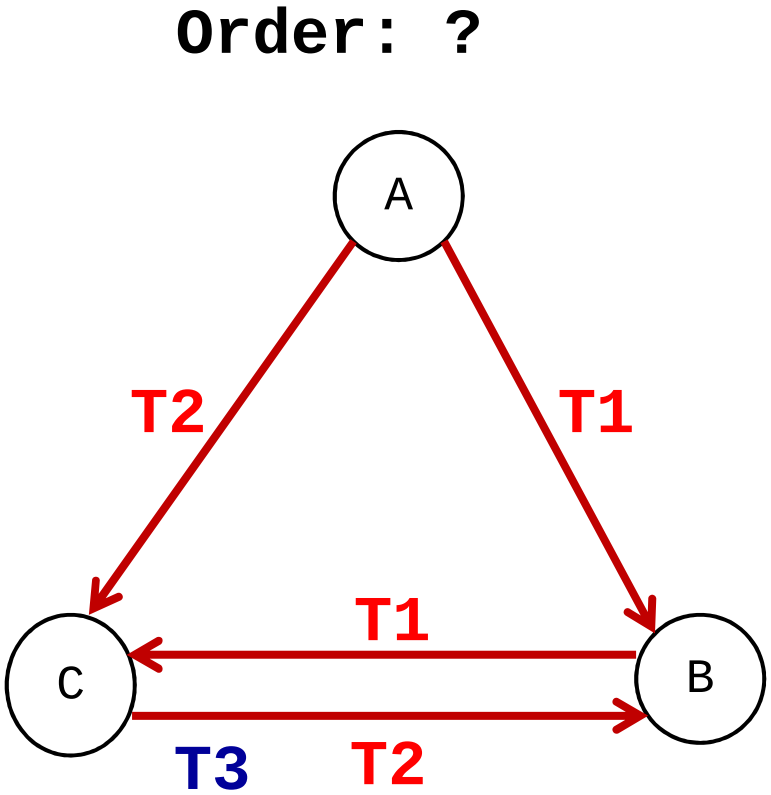
- T1:
  - Lock A
  - Lock B
  - Lock C
- T2:
  - Lock A
  - Lock C
  - Lock B




- Remember that the question is: could any trace end in deadlock?

# More examples (2)

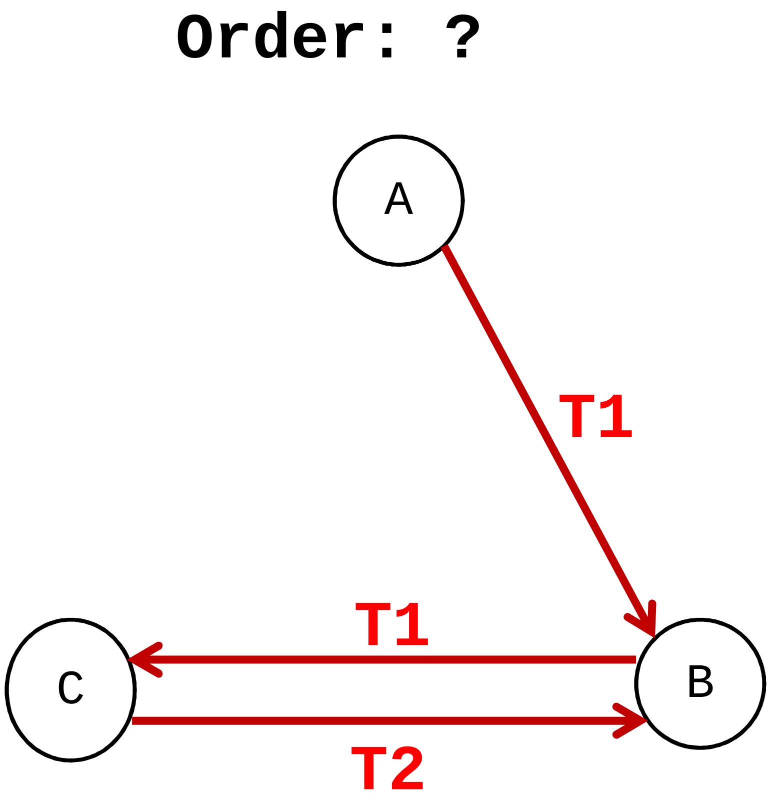
- T1:
  - Lock A
  - Lock B
  - Lock C
- T2:
  - Lock A
  - Lock C
  - Lock B
- T3:
  - Lock C
  - Lock B




- Remember that the question is: could any trace end in deadlock?

# More examples (3)

- Remove lock A from T2:
- T1:
  - Lock A
  - Lock B
  - Lock C
- T2:
  - ~~Lock A~~
  - Lock C
  - Lock B




- Remember that the question is: could any trace end in deadlock?

# Using graphs

- Wait-for-graphs are a common means for detecting deadlock
- We can use an informal variant to try to find whether deadlock is possible
- Eliminating cycles is useful
  - In which case you have an order for your locks
- You can use whatever method you wish to use in the coursework or exam
  - I only showed you the informal method I use

# Summary

- In general, putting an order on your locks is very useful
- You can **guarantee** that you do not get deadlock
- Only having one lock at a time is a very simple example of this, but often not practical
- Sometimes there are advantages to being able to lock out-of-order (for efficiency)
  - It is risky – a small change could break it

# Next week

- Another classic pattern :
  Readers and Writers

- Summary/review
  – The changing face of concurrency
  – Overview: communication and data protection
  – Why was this important?
  – How do we apply what we have seen?
  – Hardware support, now and the future?