DEADLOCK

By Cristian Koliver
Cristian.koliver@ufsc.br

CONCEPTS

System consists of resources

* Resource types $R_1, R_2, \ldots, R_m$ (CPU cycles, memory space, I/O devices ...)
* Each resource type R_i has W_i instances.

CONCEPTS

* Each task utilizes a resource as follows:

1) request: for example the system calls open( ), malloc( ), new( ), and request( ).

2) use: e.g. prints to the printer or reads from the file.

3) release: for example, close( ), free( ), delete( ), and release( ).

CONCEPTS

Deadlock can arise if **four conditions** hold simultaneously:

1) **Mutual exclusion**: only one task at a time can use a resource.

CONCEPTS

Deadlock can arise if **four conditions** hold simultaneously:

2) **Hold and wait:** a task holding at least one resource is waiting to acquire additional resources held by other tasks.

CONCEPTS

Deadlock can arise if **four conditions** hold simultaneously:

3) **No preemption**: a resource can be released only voluntarily by the task holding it, after that task has completed its task.

CONCEPTS

Deadlock can arise if **four conditions** hold simultaneously:

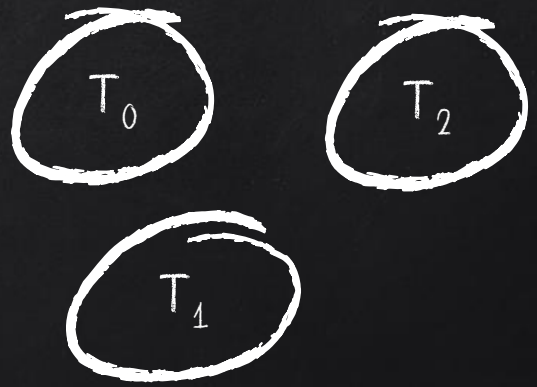
4) **Circular wait:** there exists a set $\{T_0, T_1, \ldots, T_n\}$ of waiting tasks such that T_0 is waiting for a resource that is held by T_1, T1 is waiting for a resource that is held by T_2, ..., T_{n-1} is waiting for a resource that is held by T_n, and T_n is waiting for a resource that is held by T_0.

RESOURCE-ALLOCATION GRAPH

A set of vertices **V** and a set of edges **E**.

V is partitioned into two types:

1) $T = \{T_1, T_2, \ldots, T_n\}$, the set consisting of all the tasks in the system;

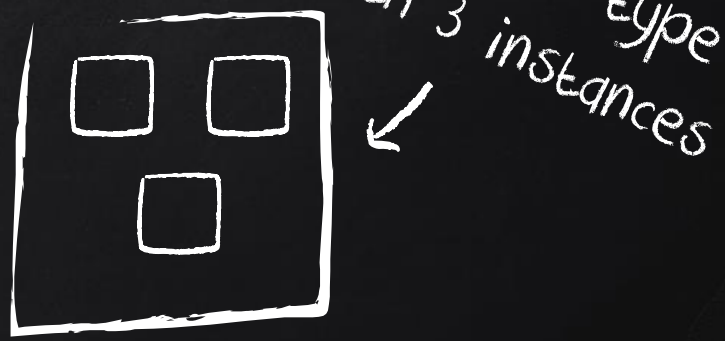


RESOURCE-ALLOCATION GRAPH

A set of vertices **V** and a set of edges **E**.

V is partitioned into two types:

2) $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system.



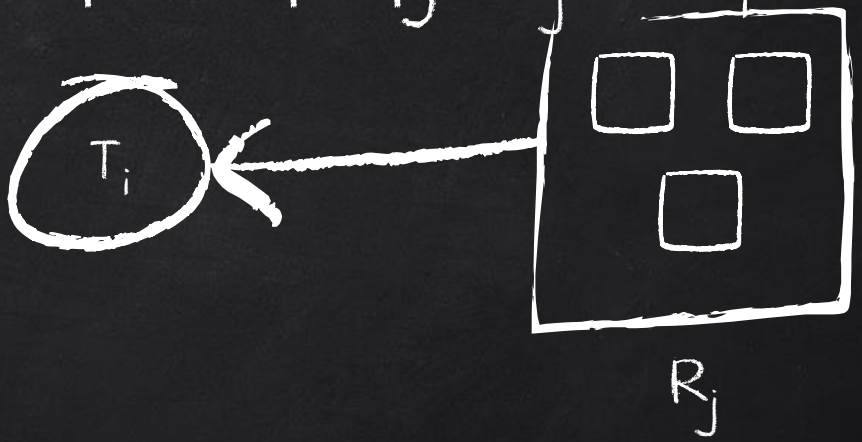
RESOURCE-ALLOCATION GRAPH

* **Request edge**: directed edge $T_i \rightarrow R_j$

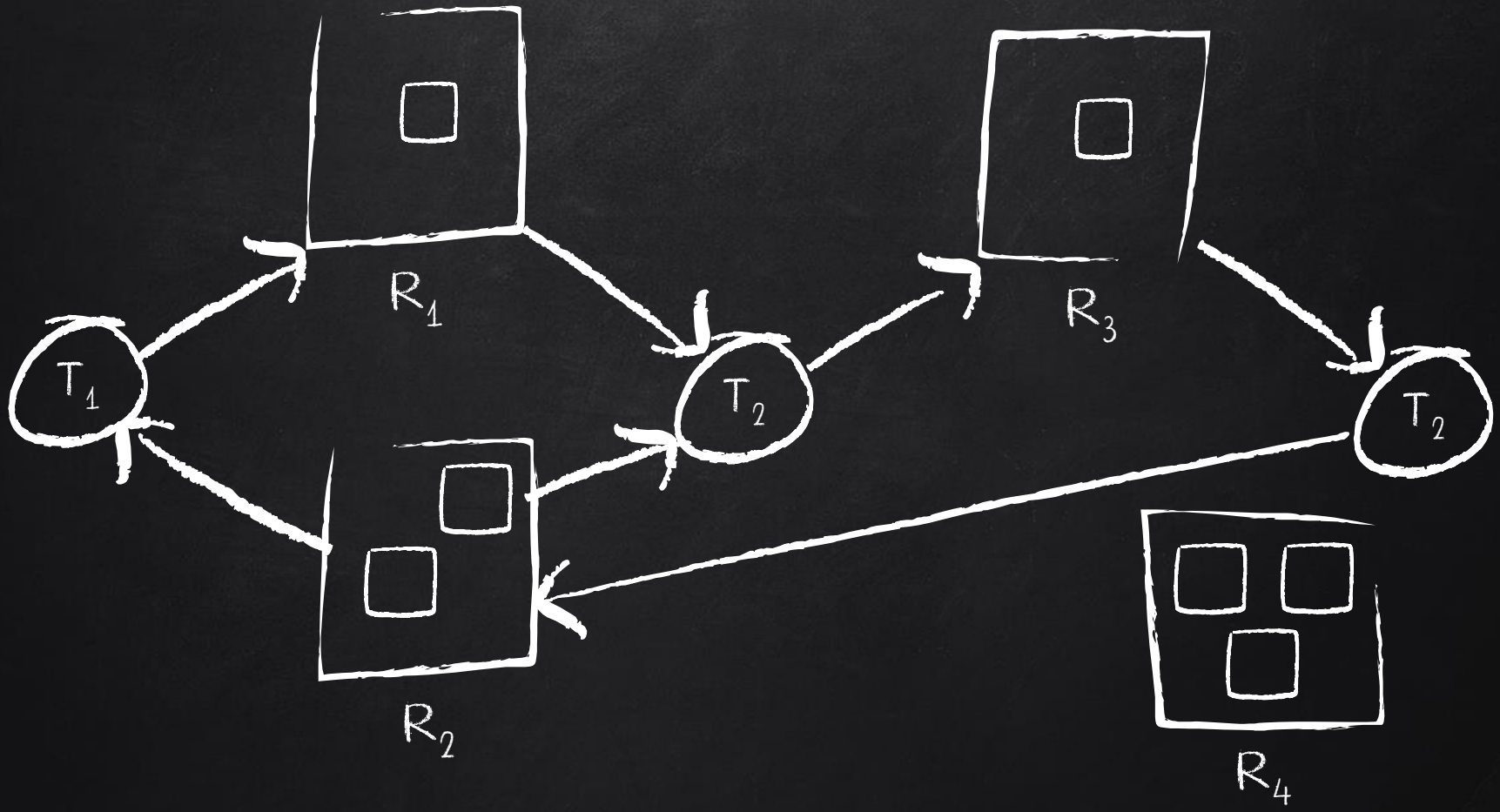
T_i requests instance of R_j

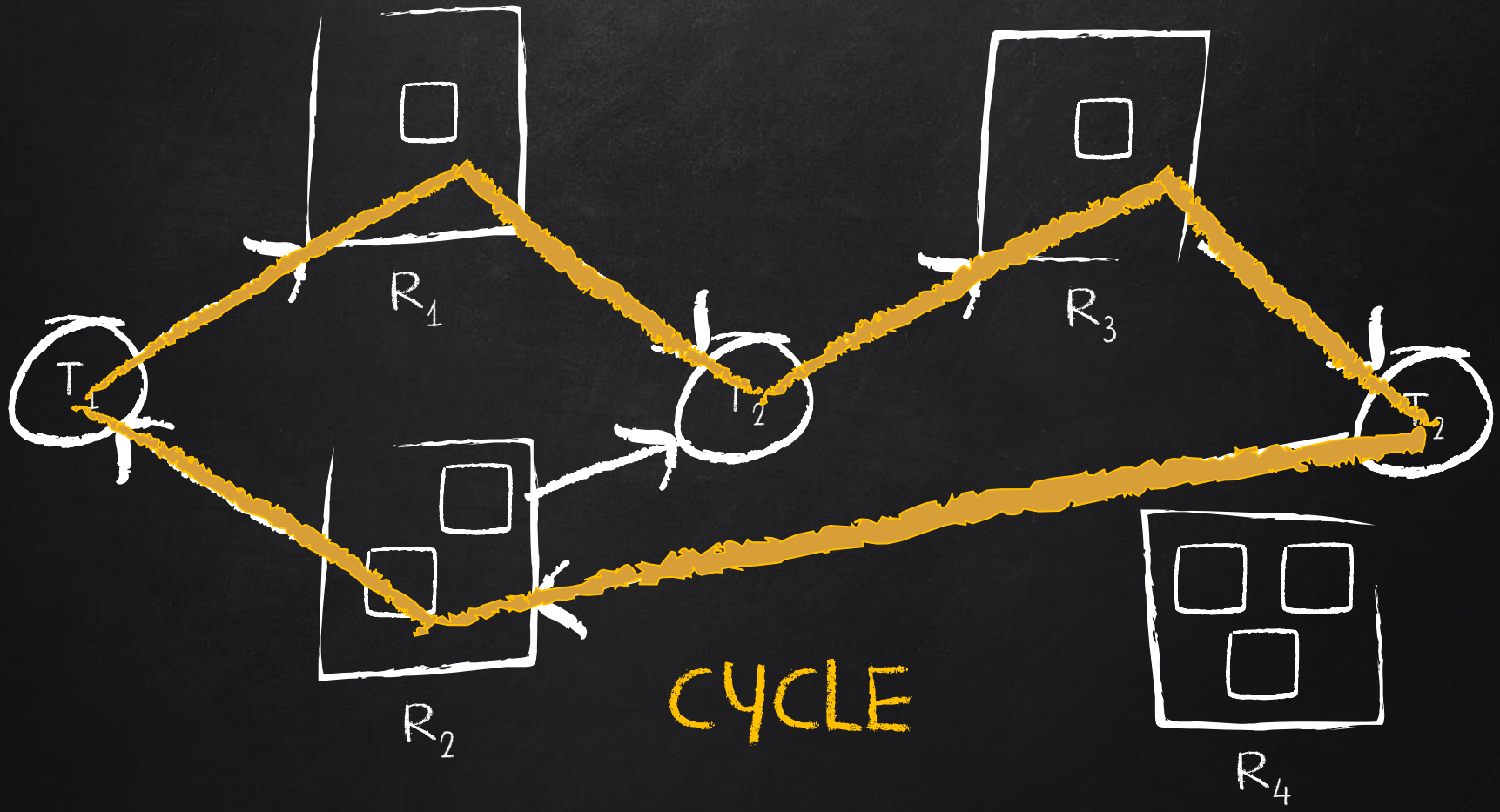
RESOURCE-ALLOCATION GRAPH

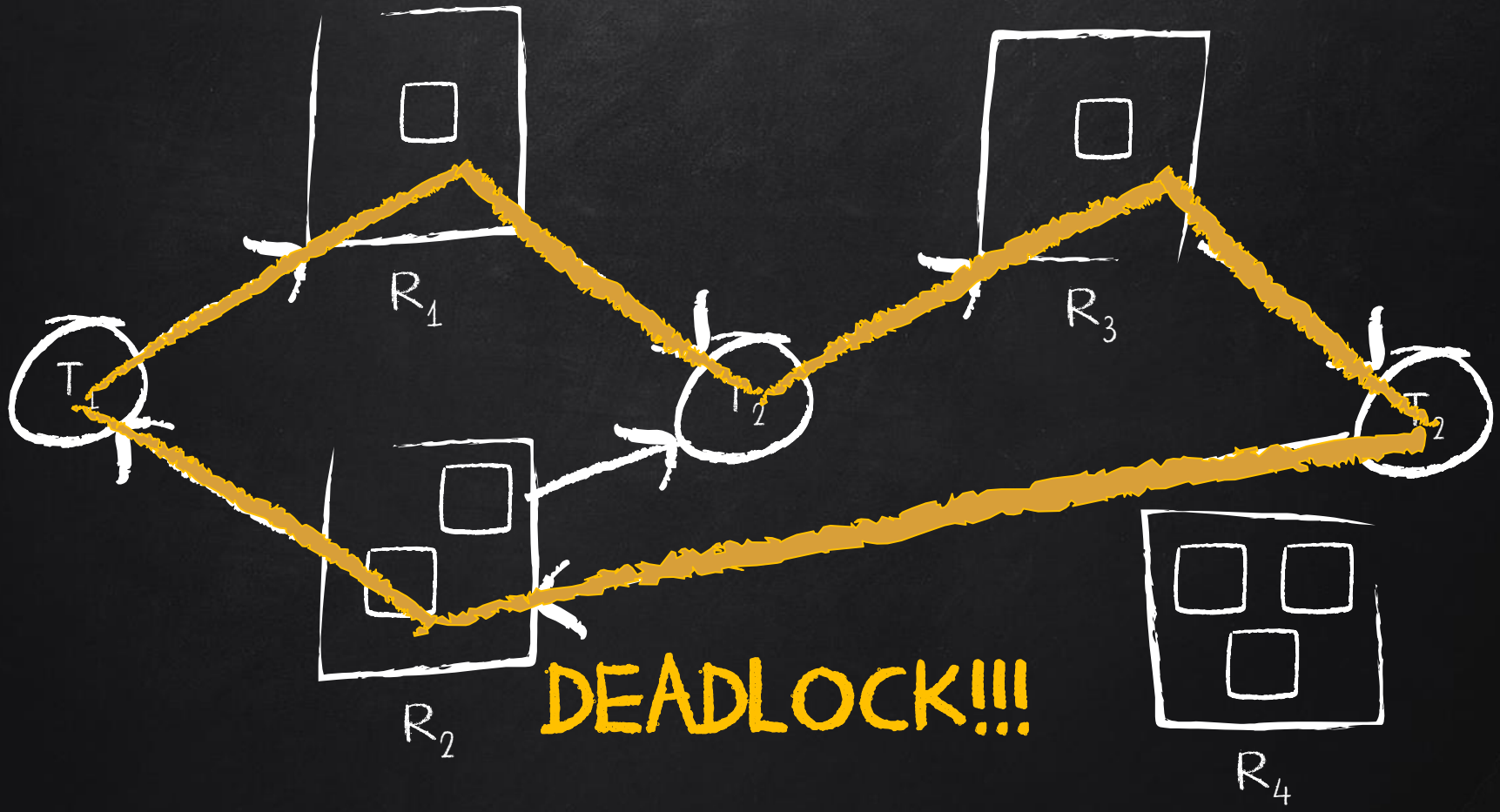
* **Assignment edge**: directed edge $R_j \rightarrow T_i$

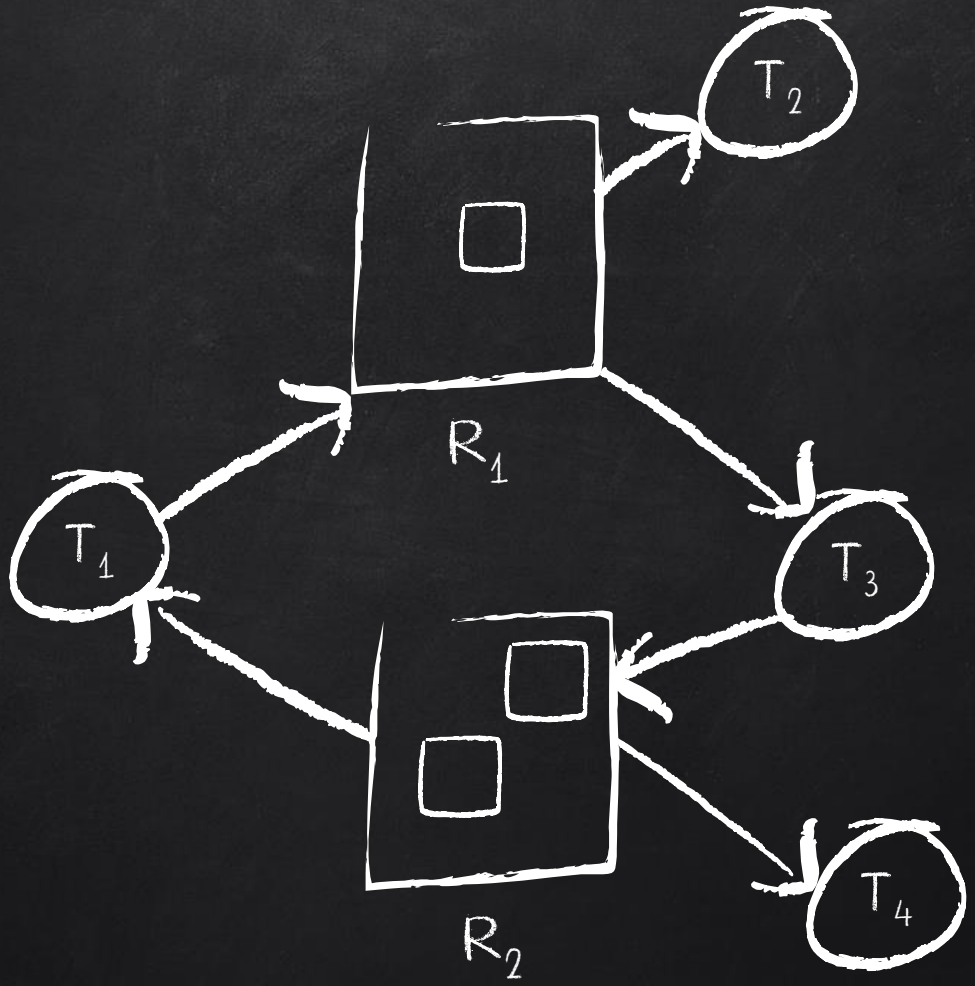


T_i is holding an instance of R_j

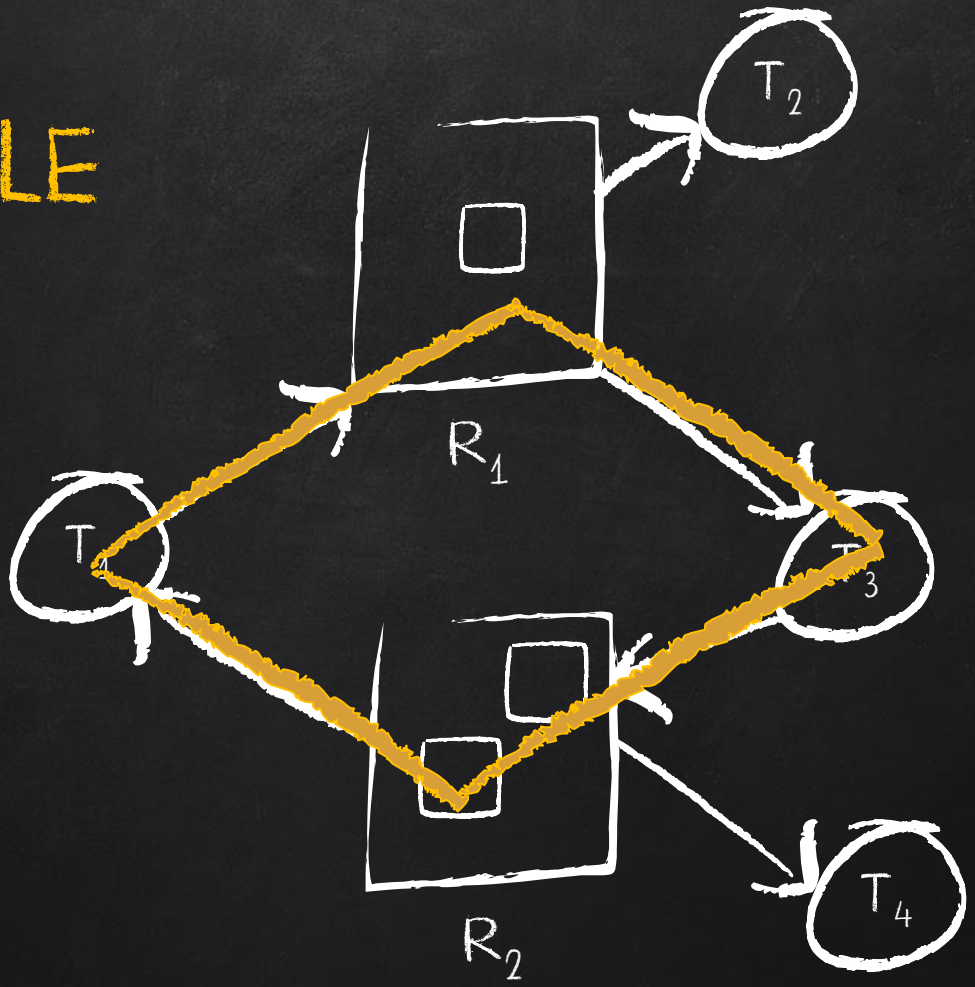

R1
R3
T1
T2
T2
R2
R4


R1
R3
T1
T2
T2
R2
CYCLE
R4

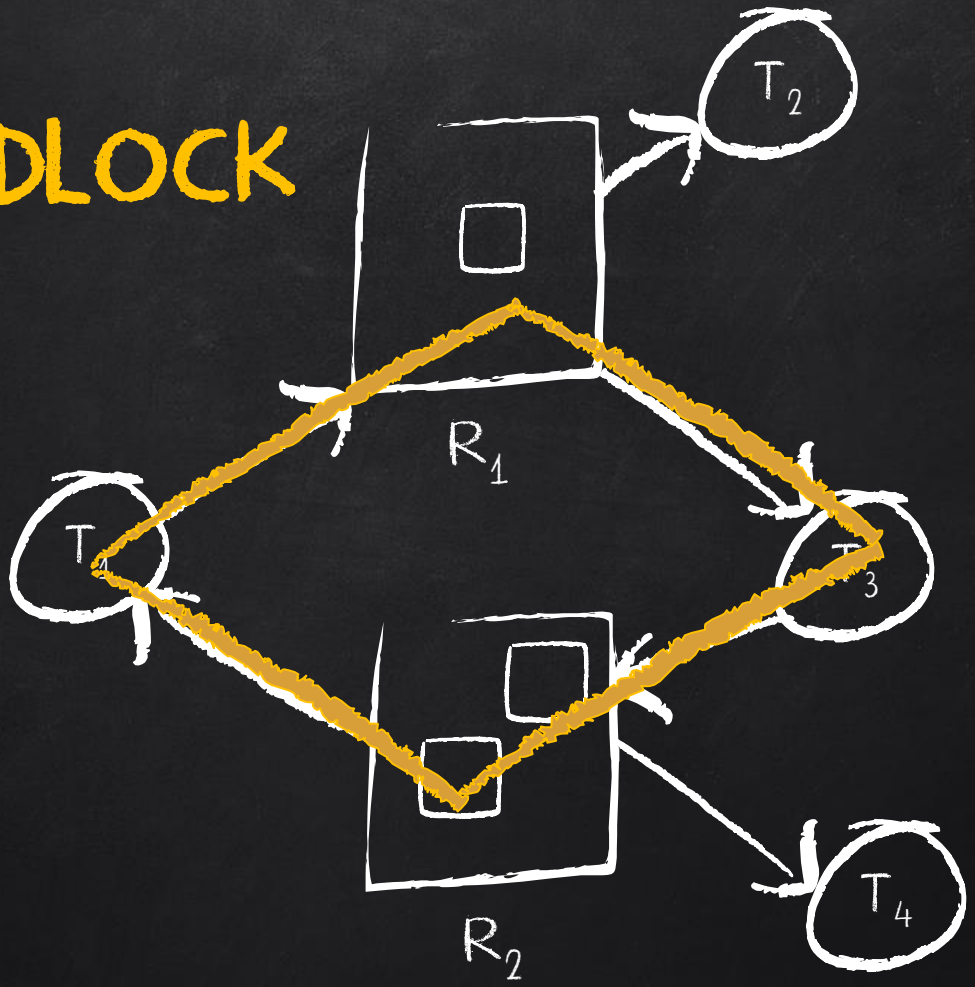

R1
R3
T1
T2
T2
R2
DEADLOCK!!!
R4

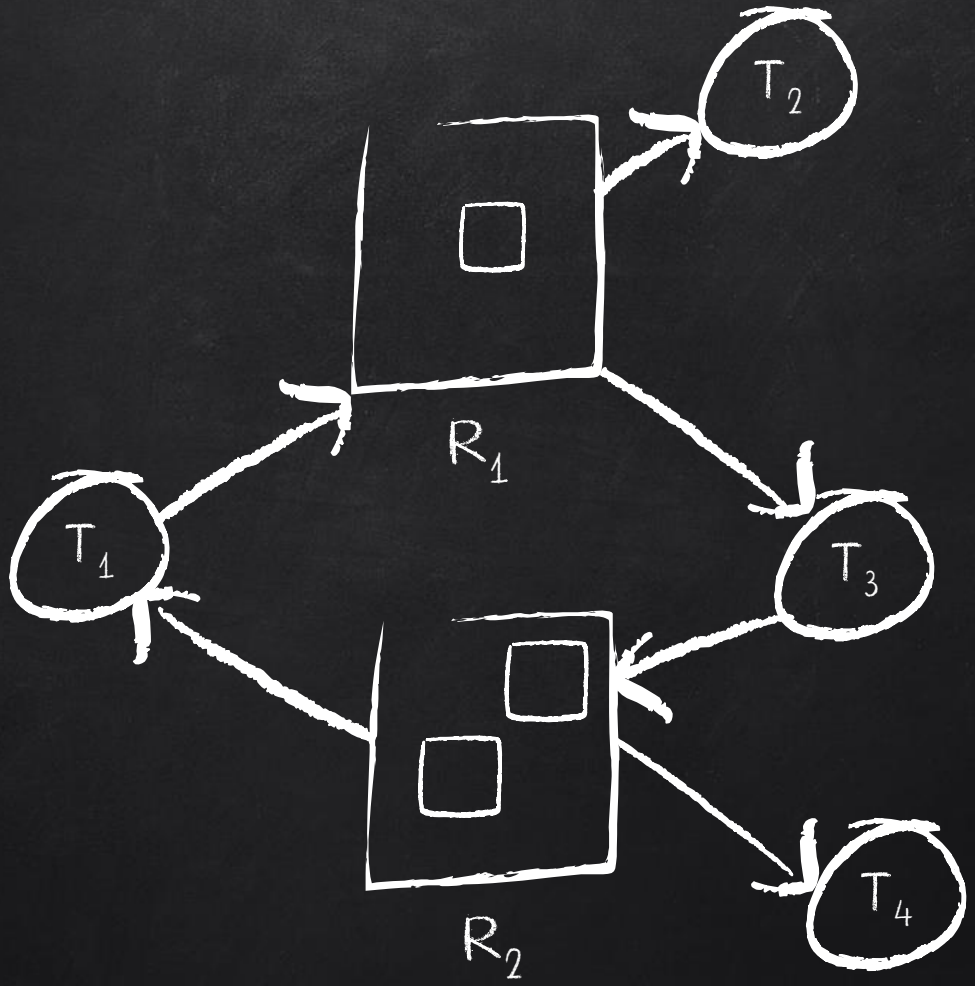

T2
R1
T1
T3
R2
T4

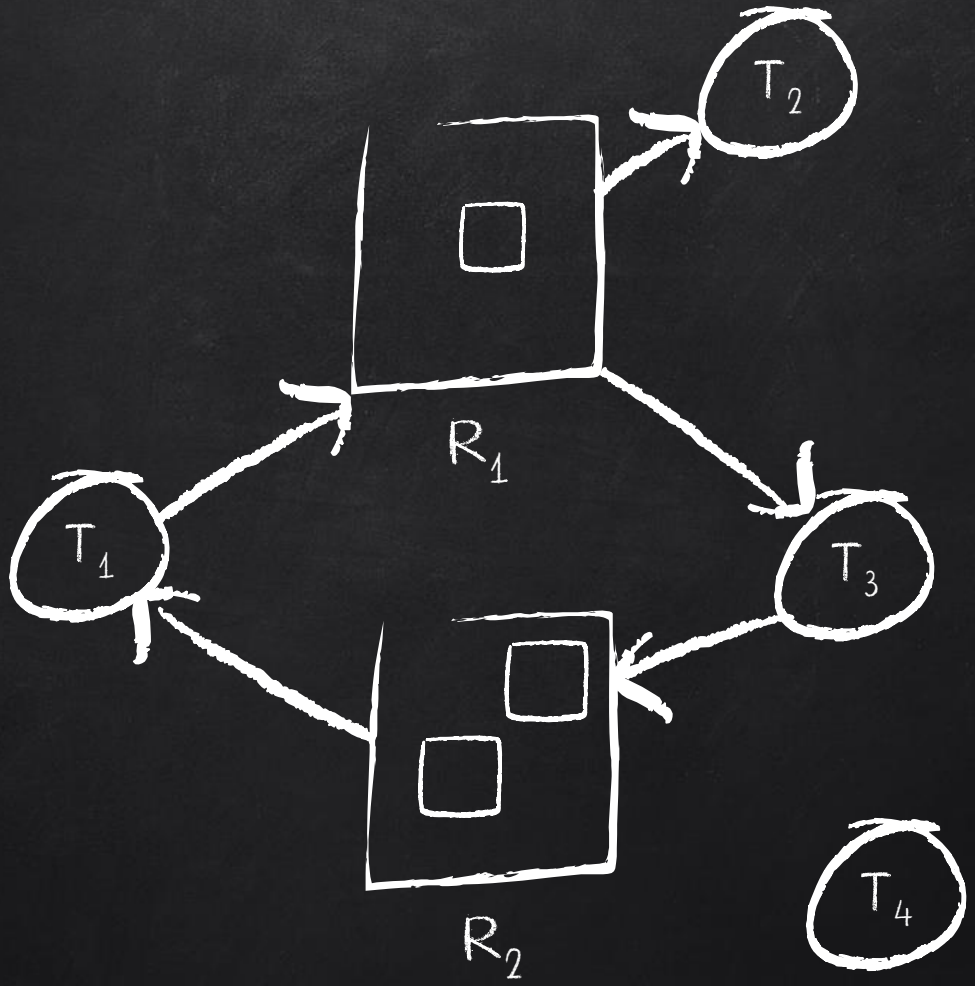
CYCLE

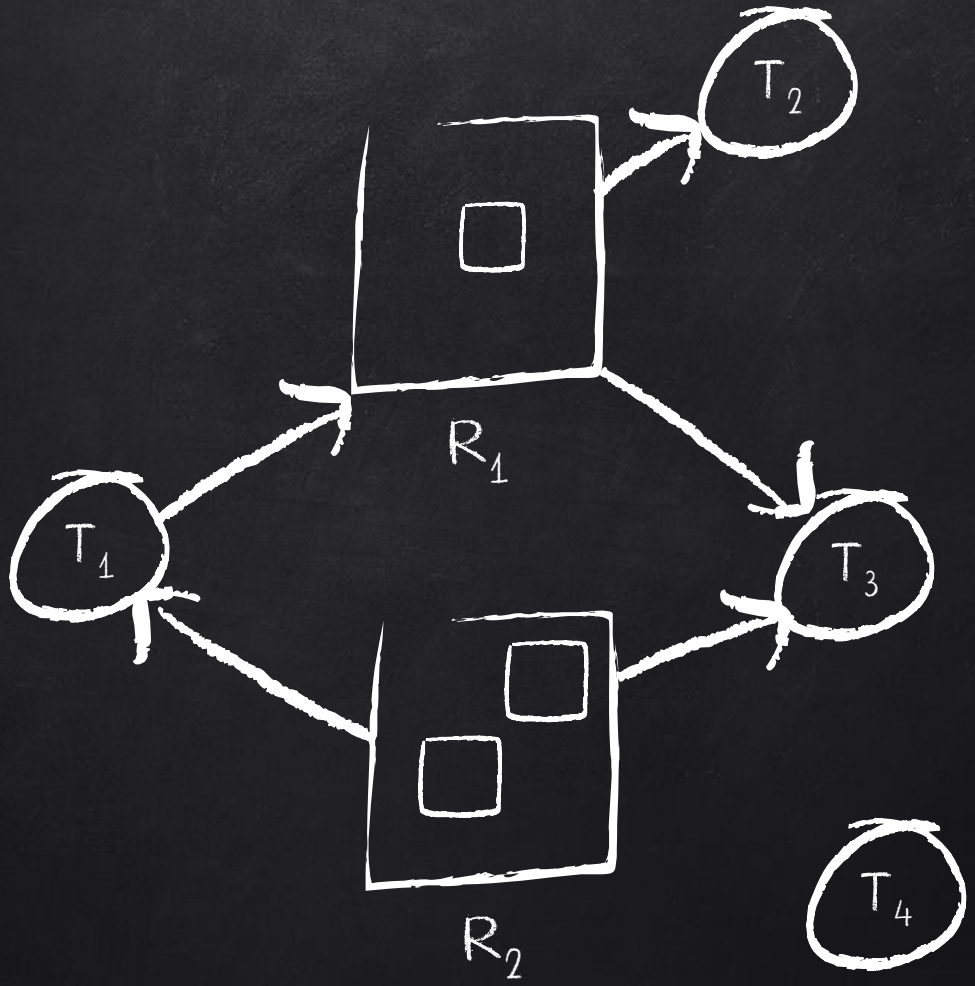


NO DEADLOCK




T2
R1
T1
T3
R2
T4


T2
R1
T1
T3
R2
T4


T2
R1
T1
T3
R2
T4

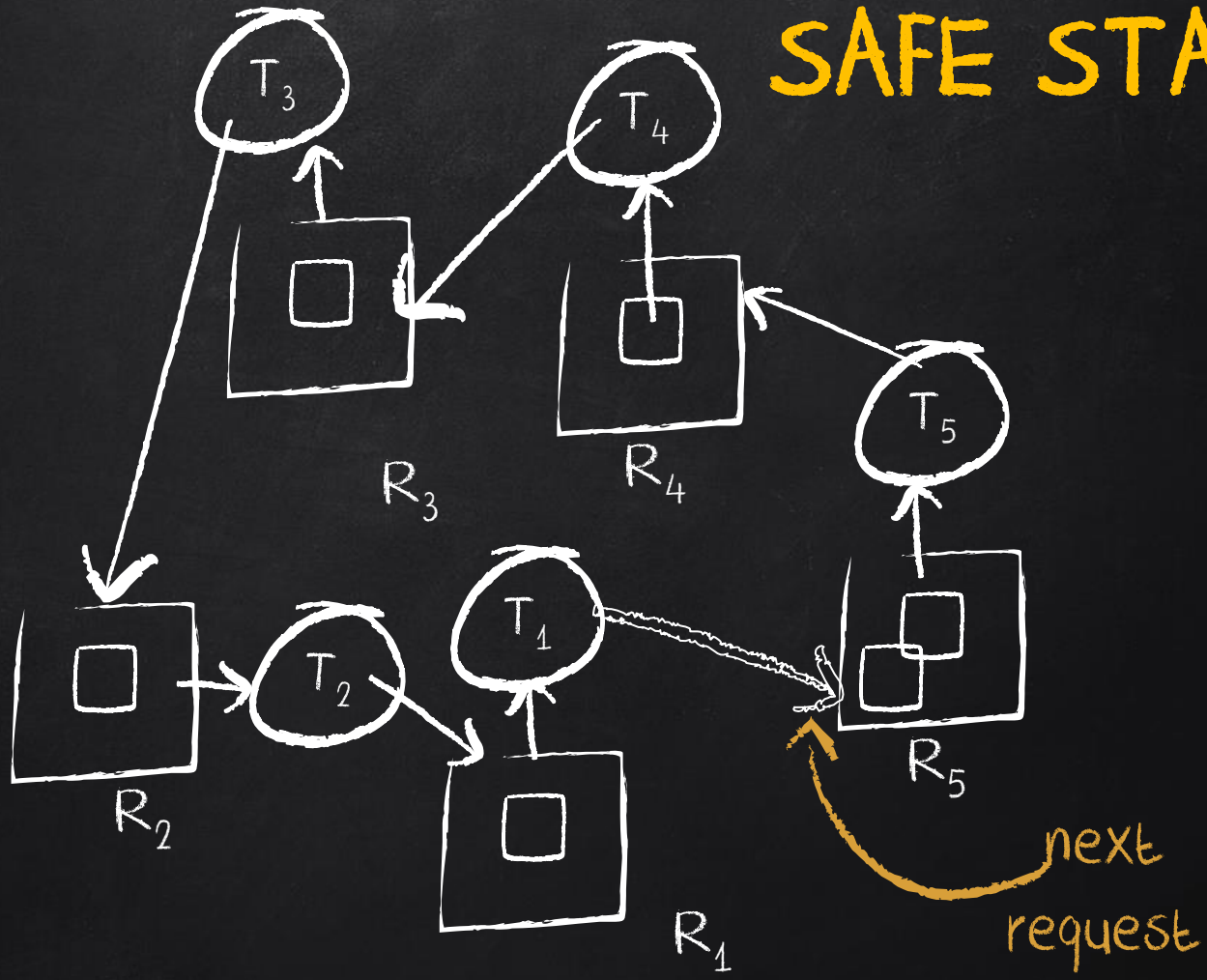
RESOURCE-ALLOCATION GRAPH

* If graph contains no cycles -> **no deadlock**.
* If graph contains a cycle:
    * if only one instance per resource type, then deadlock;
    * if several instances per resource type, possibility of deadlock.

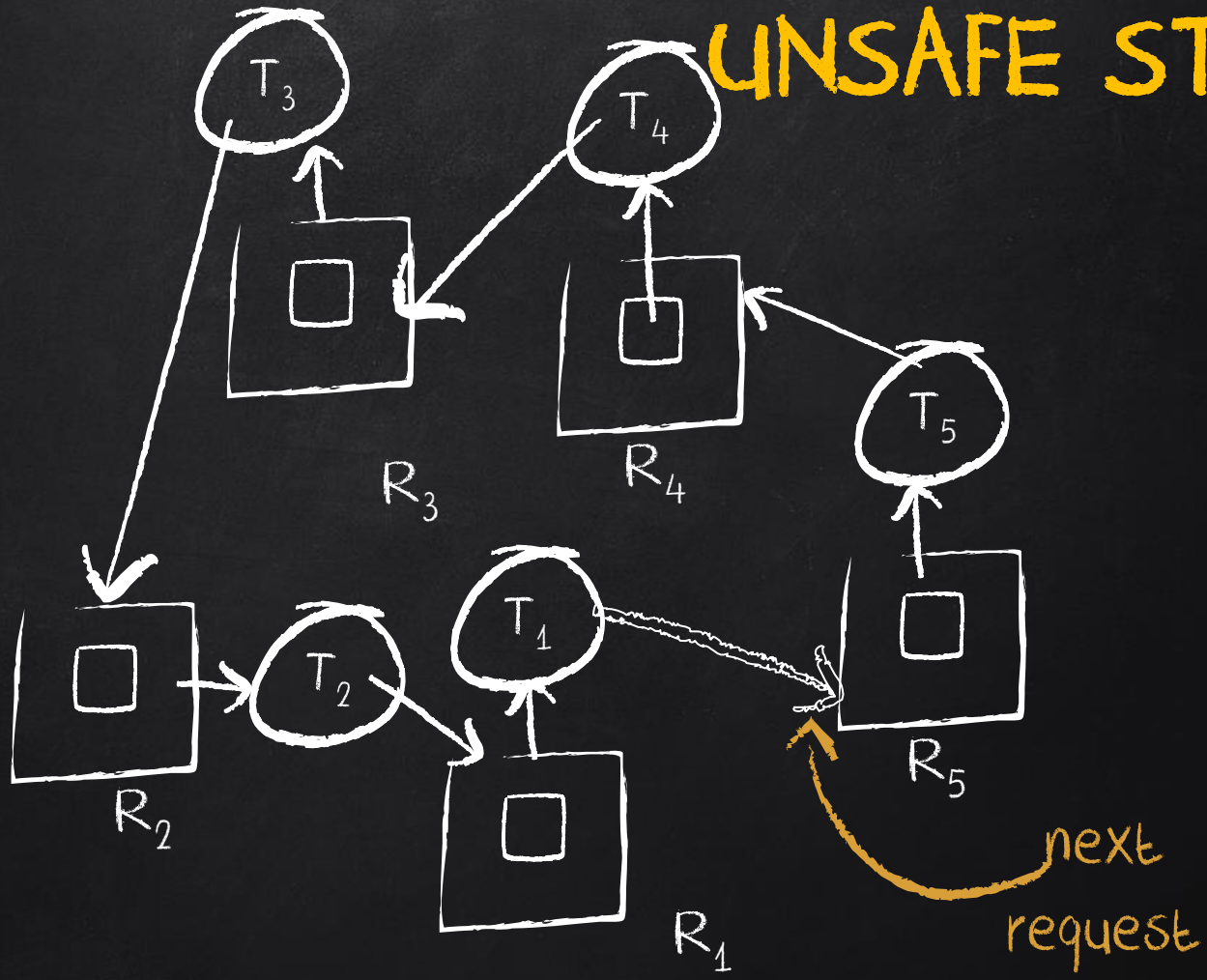
SAFE AND UNSAFE STATE

System is in **safe state** if there exists a sequence $\langle T_1, T_2, \ldots, T_n \rangle$ of ALL the tasks in the system such that for each T_i, the resources that T_i can still request can be satisfied by currently available resources + resources held by all the T_j, with $j < i$.

SAFE STATE



UNSAFE STATE



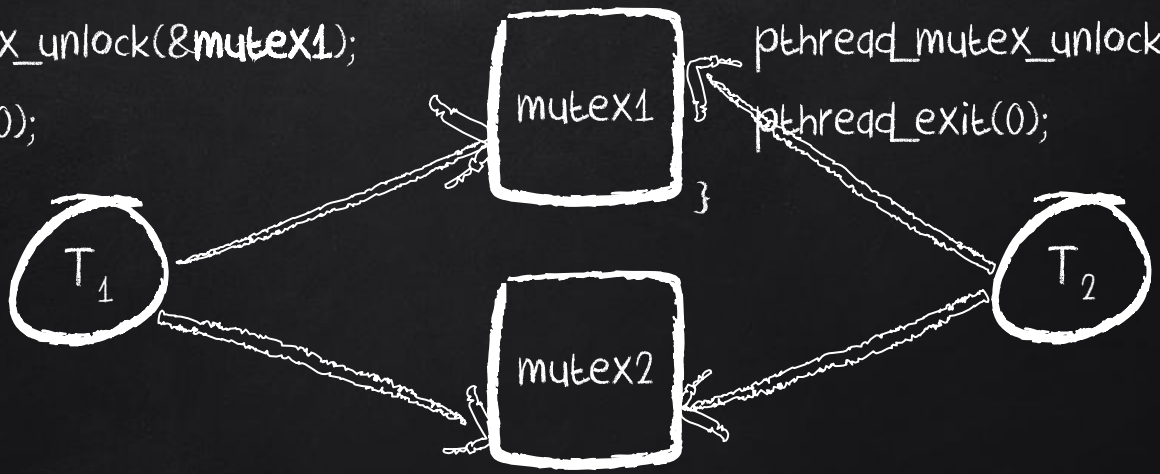
SAFE AND UNSAFE STATE

* If a system is in safe state -> no deadlocks.
* If a system is in unsafe state -> possibility of deadlock.
* Avoidance -> ensure that a system will never enter an unsafe state.


deadlock
unsafe
safe

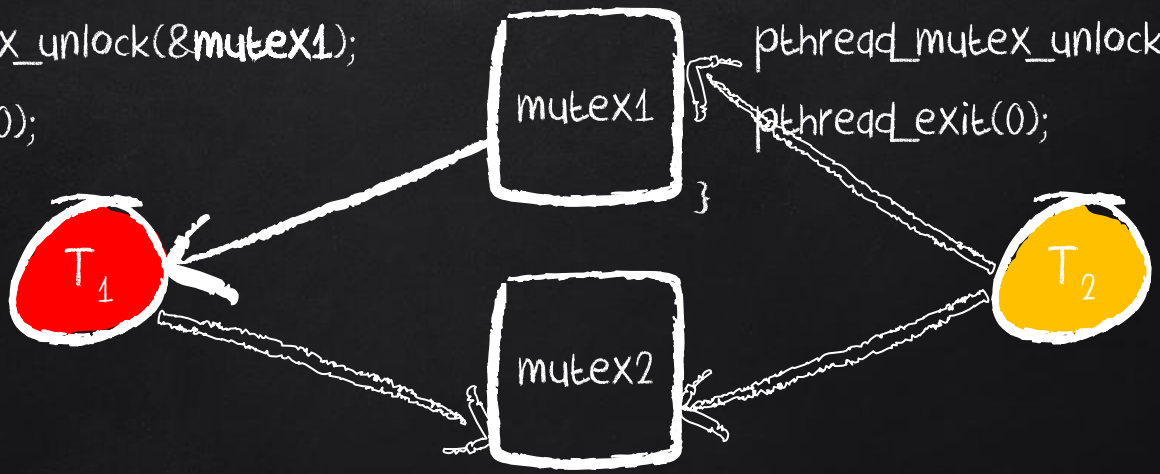
```
/* thread one runs in this function */
void *do_work_one(void *param)
{
    pthread_mutex_lock(&mutex1);
    pthread_mutex_lock(&mutex2);
    /** * Do some work */
    pthread_mutex_unlock(&mutex2);
    pthread_mutex_unlock(&mutex1);
    pthread_exit(0);
}
```

```
/* thread two runs in this function */
void *do_work_two(void *param)
{
    pthread_mutex_lock(&mutex2);
    pthread_mutex_lock(& mutex1);
    /** * Do some work */
    pthread_mutex_unlock(& mutex1);
    pthread_mutex_unlock(&mutex2);
    pthread_exit(0);
}
```



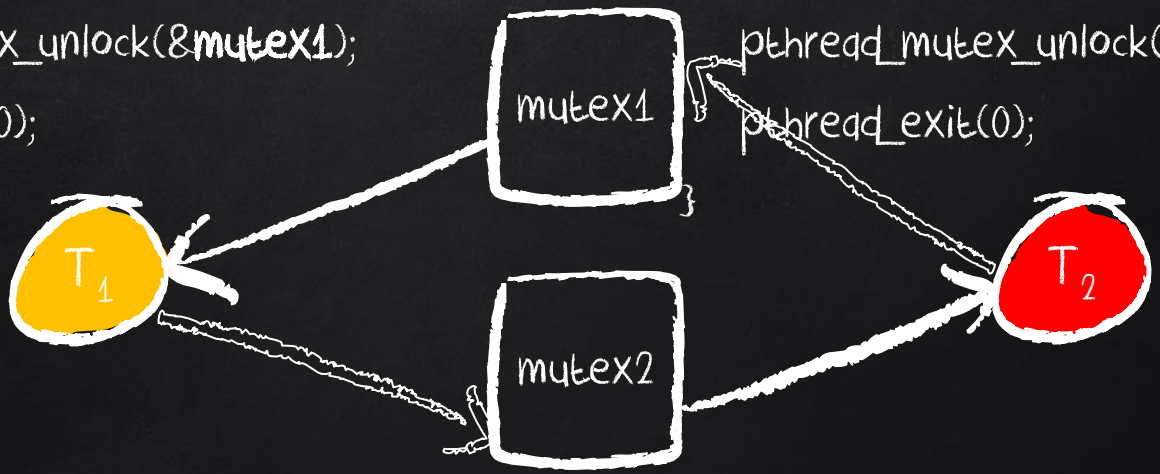
```
/* thread one runs in this function */
void *do_work_one(void *param)
{
    pthread_mutex_lock(&mutex1);
    pthread_mutex_lock(&mutex2);
    /** * Do some work */
    pthread_mutex_unlock(&mutex2);
    pthread_mutex_unlock(&mutex1);
    pthread_exit(0);
}
```

```
/* thread two runs in this function */
void *do_work_two(void *param)
{
    pthread_mutex_lock(&mutex2);
    pthread_mutex_lock(& mutex1);
    /** * Do some work */
    pthread_mutex_unlock(& mutex1);
    pthread_mutex_unlock(&mutex2);
    pthread_exit(0);
}
```



```
/* thread one runs in this function */
void *do_work_one(void *param)
{
    pthread_mutex_lock(&mutex1);
    pthread_mutex_lock(&mutex2);
    /** * Do some work */
    pthread_mutex_unlock(&mutex2);
    pthread_mutex_unlock(&mutex1);
    pthread_exit(0);
}
```

```
/* thread two runs in this function */
void *do_work_two(void *param)
{
    pthread_mutex_lock(&mutex2);
    pthread_mutex_lock(& mutex1);
    /** * Do some work */
    pthread_mutex_unlock(& mutex1);
    pthread_mutex_unlock(&mutex2);
    pthread_exit(0);
}
```

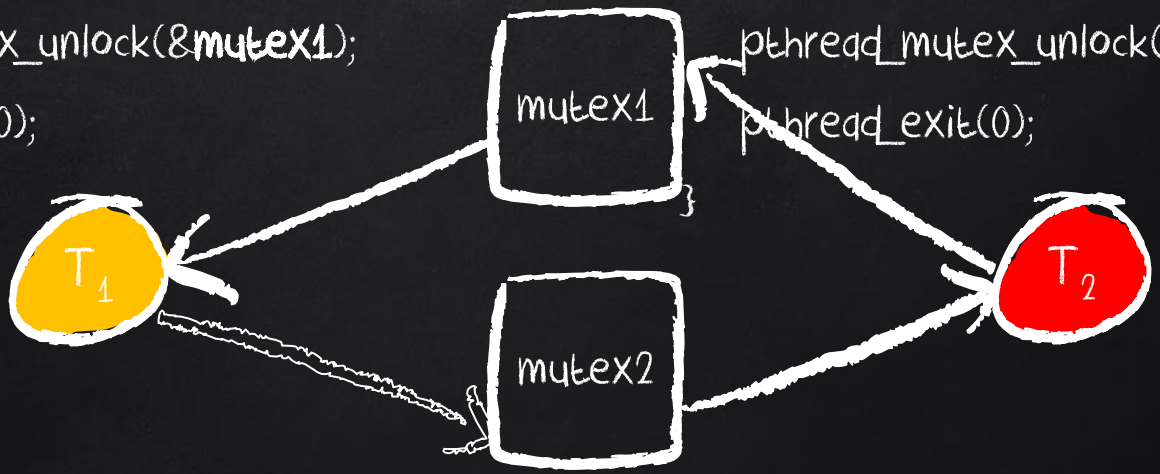


```
/* thread one runs in this function */
void *do_work_one(void *param)
{
    pthread_mutex_lock(&mutex1);
    pthread_mutex_lock(&mutex2);
    /** * Do some work */
    pthread_mutex_unlock(&mutex2);
    pthread_mutex_unlock(&mutex1);
    pthread_exit(0);
}
```

```
/* thread two runs in this function */
void *do_work_two(void *param)
{
    pthread_mutex_lock(&mutex2);
    pthread_mutex_lock(& mutex1);
    /** * Do some work */
    pthread_mutex_unlock(& mutex1);
    pthread_mutex_unlock(&mutex2);
    pthread_exit(0);
```

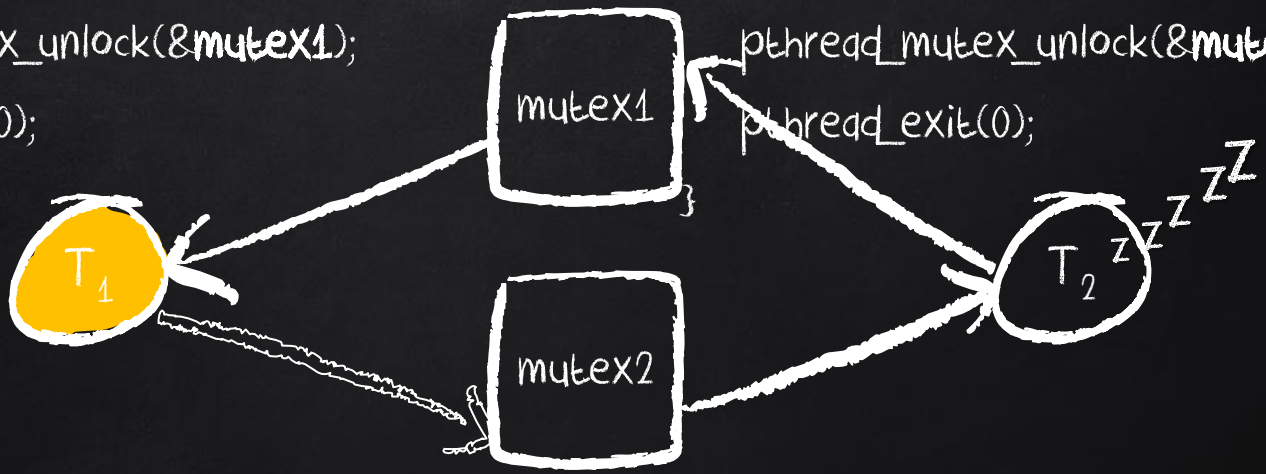
```
/* thread one runs in this function */
void *do_work_one(void *param)
{
    pthread_mutex_lock(&mutex1);
    pthread_mutex_lock(&mutex2);
    /** * Do some work */
    pthread_mutex_unlock(&mutex2);
    pthread_mutex_unlock(&mutex1);
    pthread_exit(0);
}
```

```
/* thread two runs in this function */
void *do_work_two(void *param)
{
    pthread_mutex_lock(&mutex2);
    pthread_mutex_lock(& mutex1);
    /** * Do some work */
    pthread_mutex_unlock(& mutex1);
    pthread_mutex_unlock(&mutex2);
    pthread_exit(0);
}
```



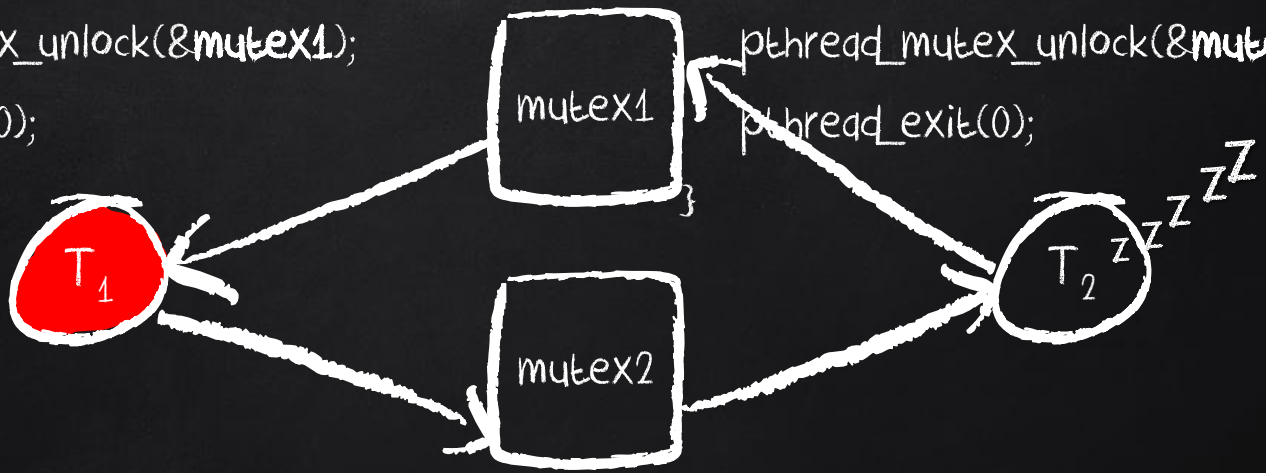
```
/* thread one runs in this function */
void *do_work_one(void *param)
{
    pthread_mutex_lock(&mutex1);
    pthread_mutex_lock(&mutex2);
    /** * Do some work */
    pthread_mutex_unlock(&mutex2);
    pthread_mutex_unlock(&mutex1);
    pthread_exit(0);
}
```

```
/* thread two runs in this function */
void *do_work_two(void *param)
{
    pthread_mutex_lock(&mutex2);
    pthread_mutex_lock(& mutex1);
    /** * Do some work */
    pthread_mutex_unlock(& mutex1);
    pthread_mutex_unlock(&mutex2);
    pthread_exit(0);
}
```



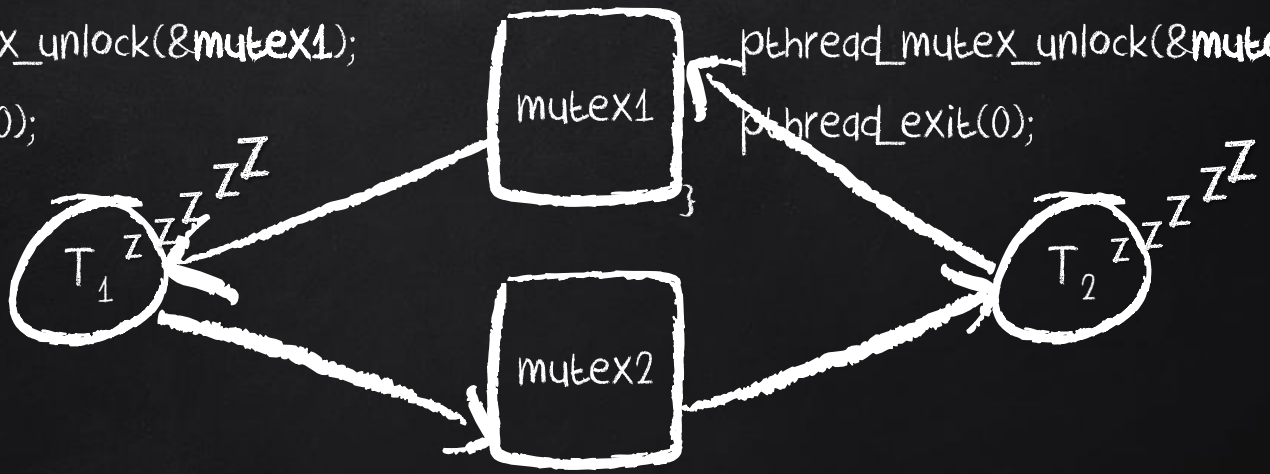
```
/* thread one runs in this function */
void *do_work_one(void *param)
{
    pthread_mutex_lock(&mutex1);
    pthread_mutex_lock(&mutex2);
    /** * Do some work */
    pthread_mutex_unlock(&mutex2);
    pthread_mutex_unlock(&mutex1);
    pthread_exit(0);
}
```

```
/* thread two runs in this function */
void *do_work_two(void *param)
{
    pthread_mutex_lock(&mutex2);
    pthread_mutex_lock(& mutex1);
    /** * Do some work */
    pthread_mutex_unlock(& mutex1);
    pthread_mutex_unlock(&mutex2);
    pthread_exit(0);
}
```



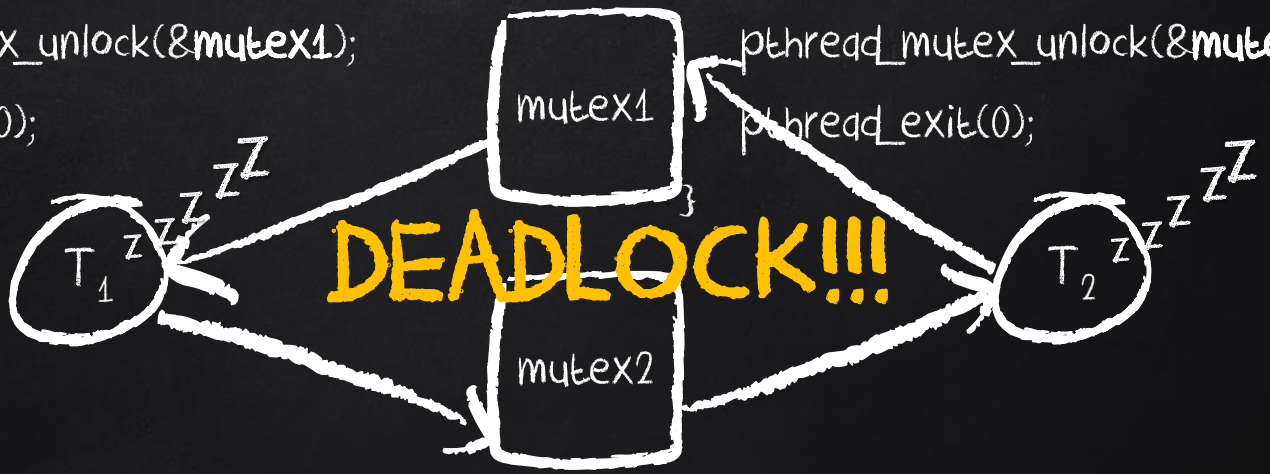
```
/* thread one runs in this function */
void *do_work_one(void *param)
{
    pthread_mutex_lock(&mutex1);
    pthread_mutex_lock(&mutex2);
    /** * Do some work */
    pthread_mutex_unlock(&mutex2);
    pthread_mutex_unlock(&mutex1);
    pthread_exit(0);
}
```

```
/* thread two runs in this function */
void *do_work_two(void *param)
{
    pthread_mutex_lock(&mutex2);
    pthread_mutex_lock(& mutex1);
    /** * Do some work */
    pthread_mutex_unlock(& mutex1);
    pthread_mutex_unlock(&mutex2);
    pthread_exit(0);
}
```



```
/* thread one runs in this function */
void *do_work_one(void *param)
{
    pthread_mutex_lock(&mutex1);
    pthread_mutex_lock(&mutex2);
    /** * Do some work */
    pthread_mutex_unlock(&mutex2);
    pthread_mutex_unlock(&mutex1);
    pthread_exit(0);
}
```

```
/* thread two runs in this function */
void *do_work_two(void *param)
{
    pthread_mutex_lock(&mutex2);
    pthread_mutex_lock(& mutex1);
    /** * Do some work */
    pthread_mutex_unlock(& mutex1);
    pthread_mutex_unlock(&mutex2);
    pthread_exit(0);
}
```



HANDLING DEADLOCKS

1) Deadlock prevention
2) Deadlock avoidance
3) Ignore deadlocks

HANDLING DEADLOCKS

1) Prevention

Eliminate one or more conditions needed for deadlock

HANDLING DEADLOCKS

1) Prevention

* **Eliminate mutual exclusion**: not required for sharable resources (e.g., read-only files); must hold for non-sharable resources

HANDLING DEADLOCKS

1) Prevention

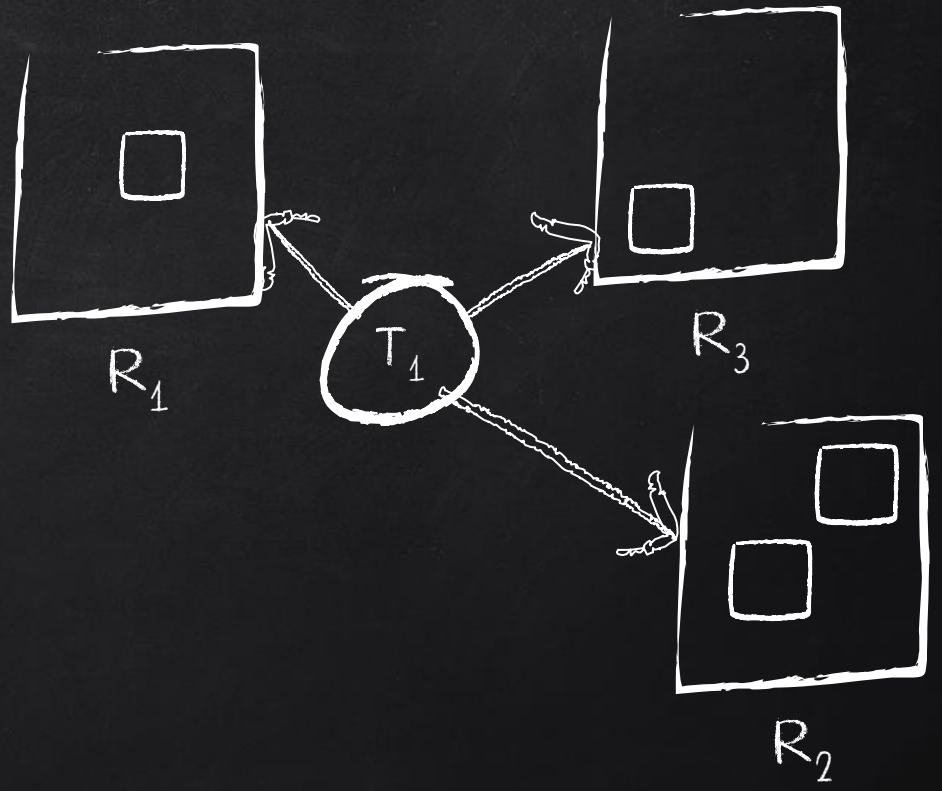
* **Eliminate mutual exclusion**: not viable for some resources, such as printers and tape drives, that require exclusive access by a single task.

HANDLING DEADLOCKS

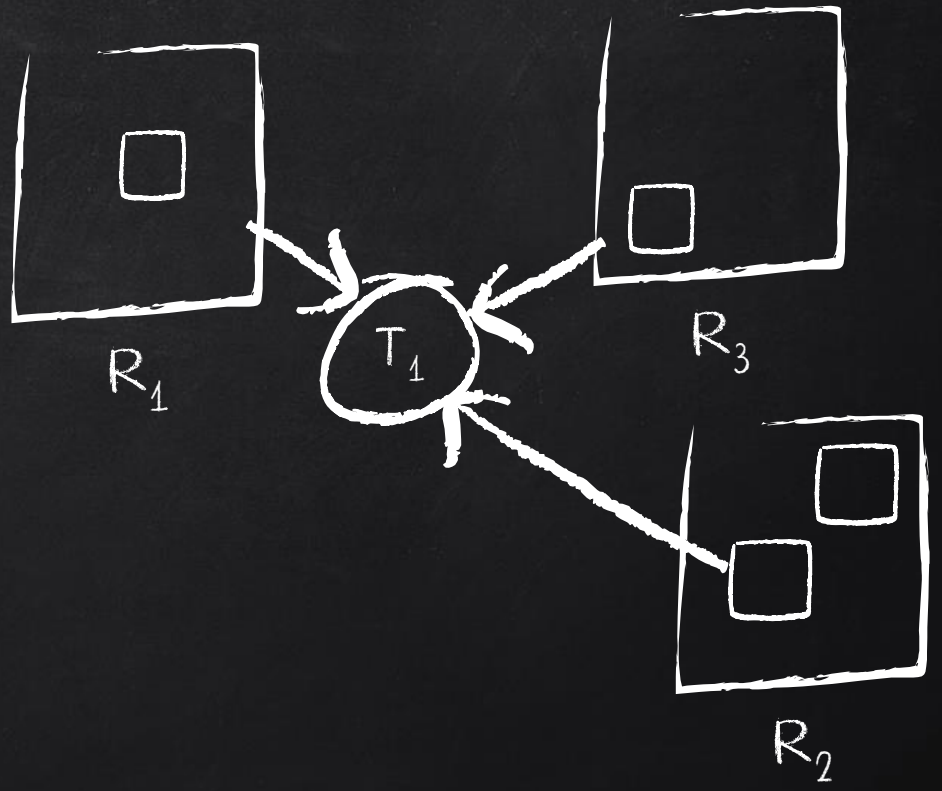
1) Prevention

* Eliminate hold and wait

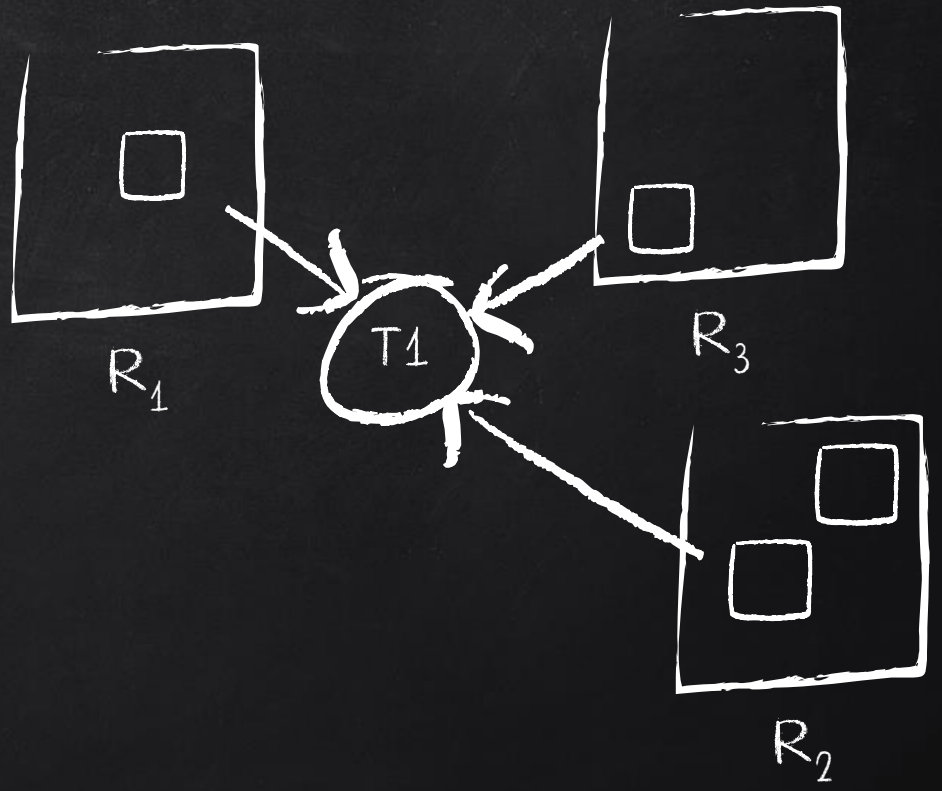
Approach 1: task requests all its resources before it begins execution. It is triggers iff it holds all needed resources



Approach 1: task requests all its resources before it begins execution. It is triggers iff it holds all needed resources

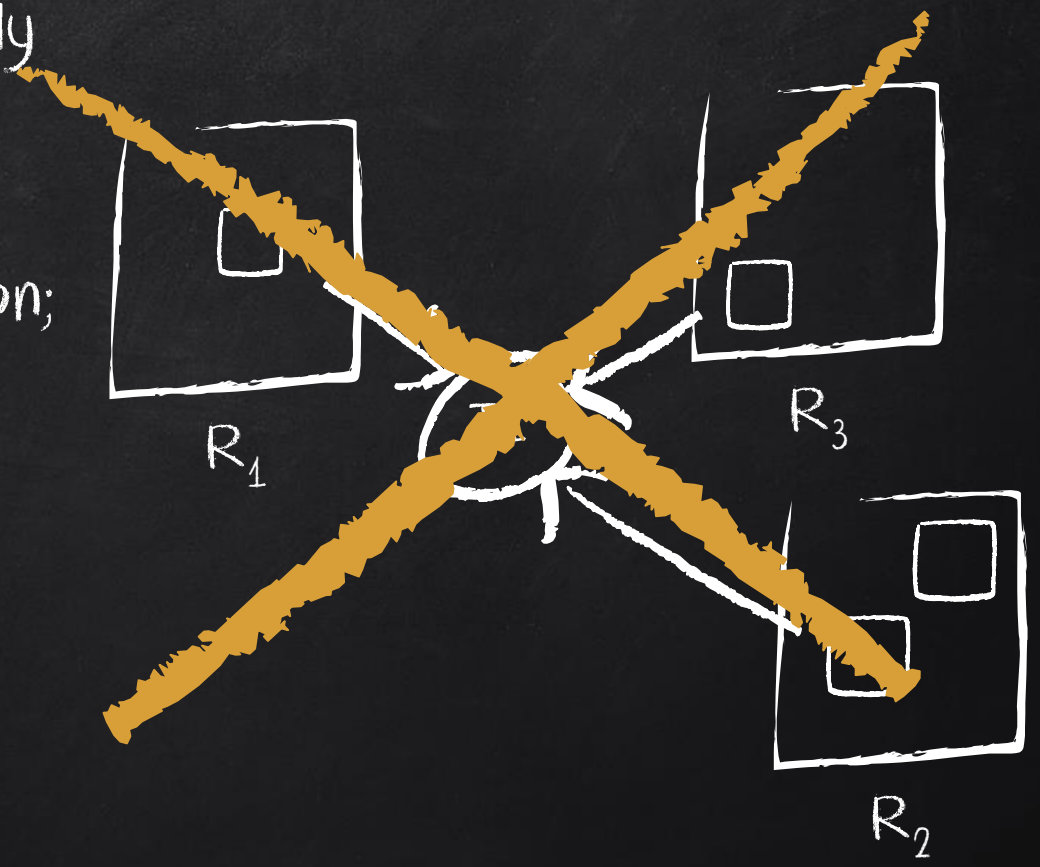


Approach 2: task can request resources only when it has none allocated to it.
Low resource utilization; starvation possible

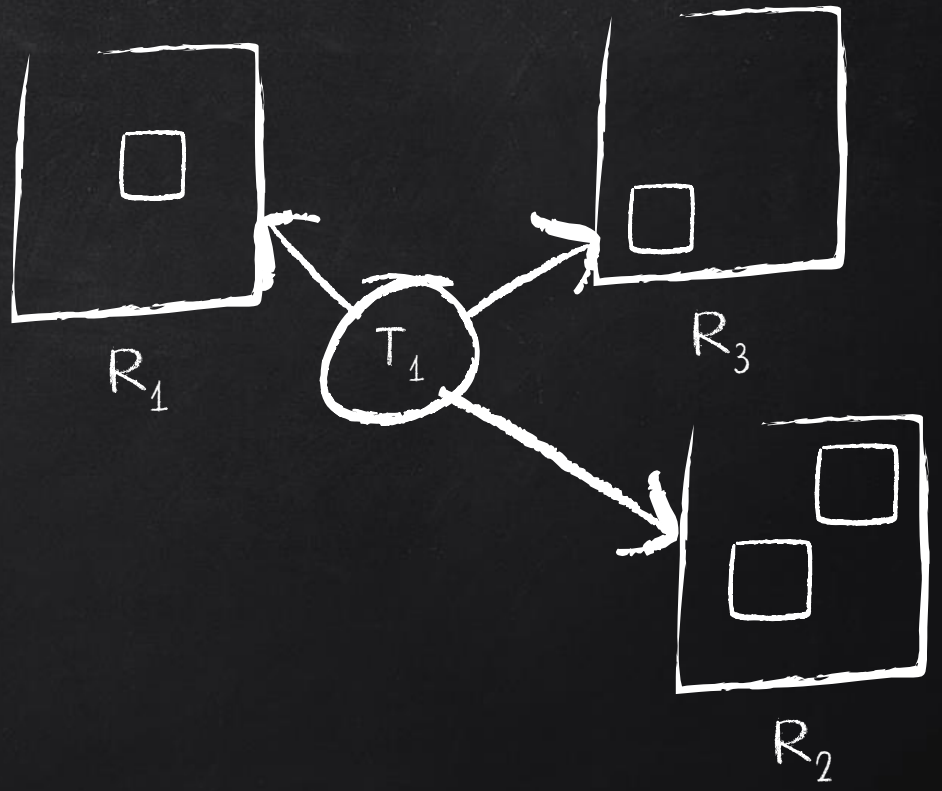


Approach 2: task can request resources only when it has none allocated to it.

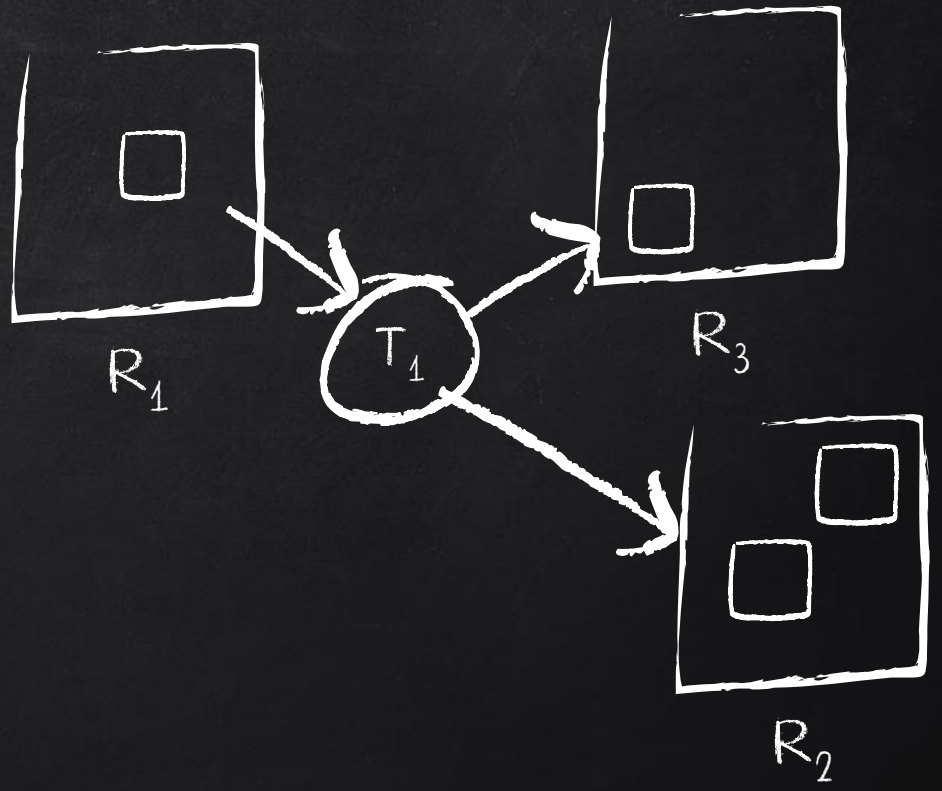
Low resource utilization; starvation possible



Approach 2: task can request resources only when it has none allocated to it.
Low resource utilization; starvation possible

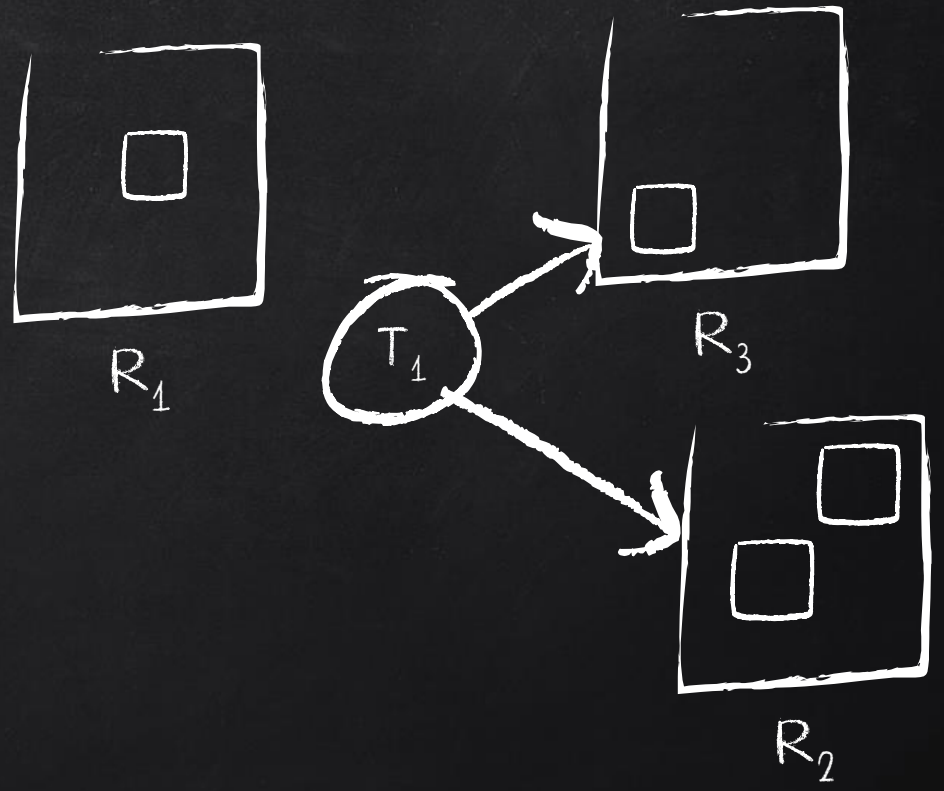


Approach 2: task can request resources only when it has none allocated to it.
Low resource utilization; starvation possible

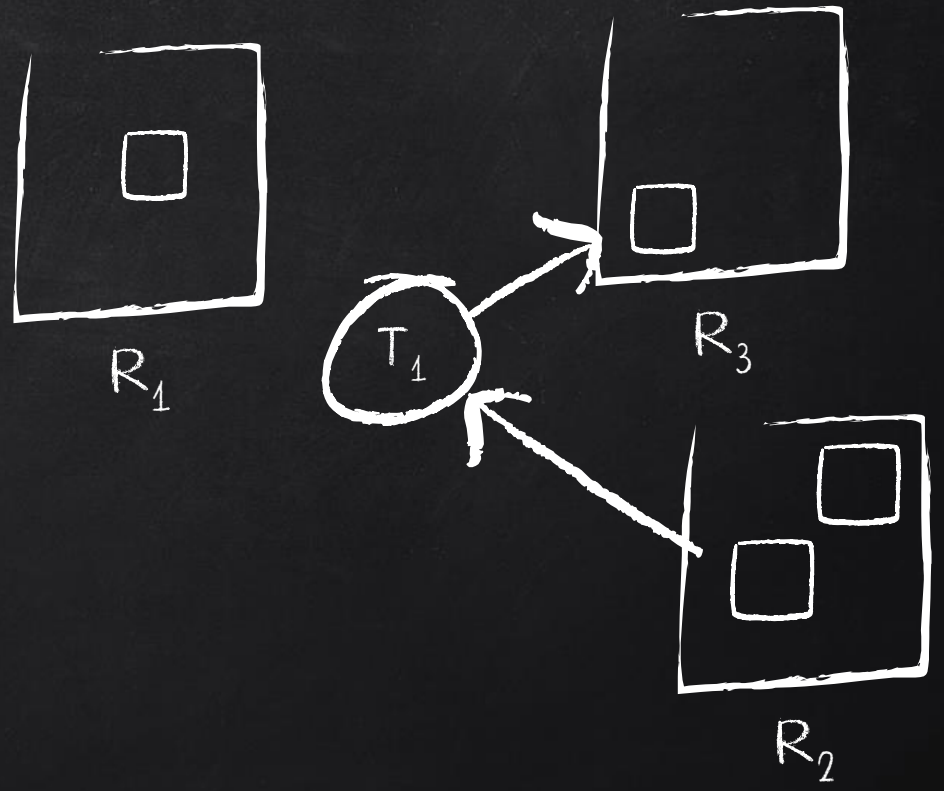


Approach 2: task can request resources only when it has none allocated to it.

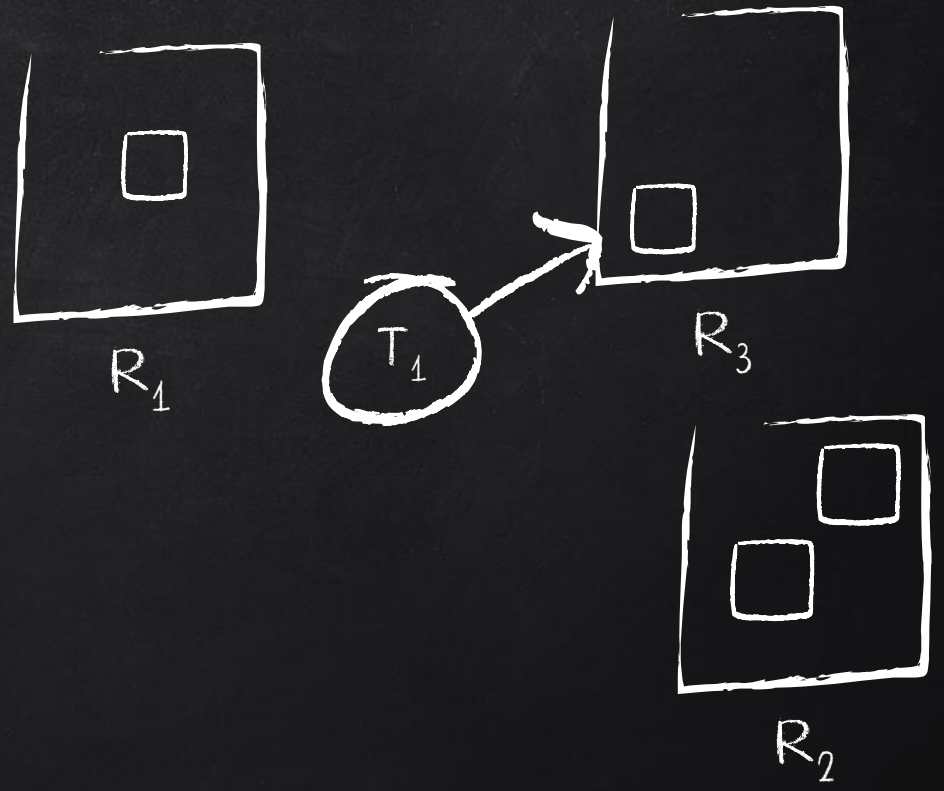
Low resource utilization; starvation possible



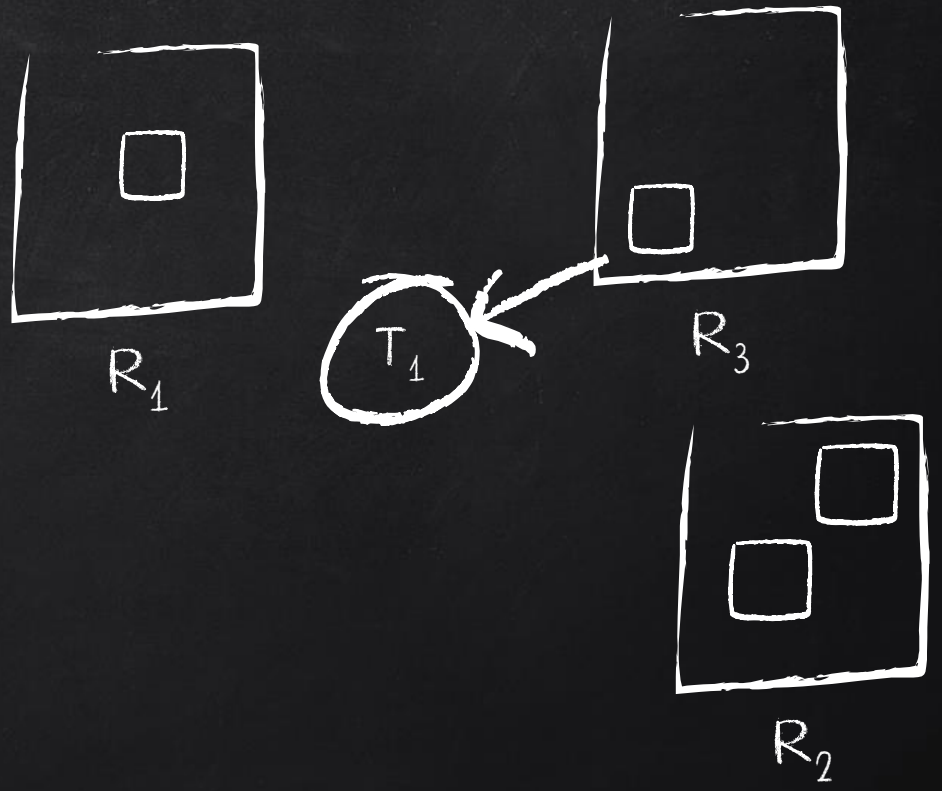
Approach 2: task can request resources only when it has none allocated to it.
Low resource utilization; starvation possible



Approach 2: task can request resources only when it has none allocated to it.
Low resource utilization; starvation possible



Approach 2: task can request resources only when it has none allocated to it.
Low resource utilization; starvation possible

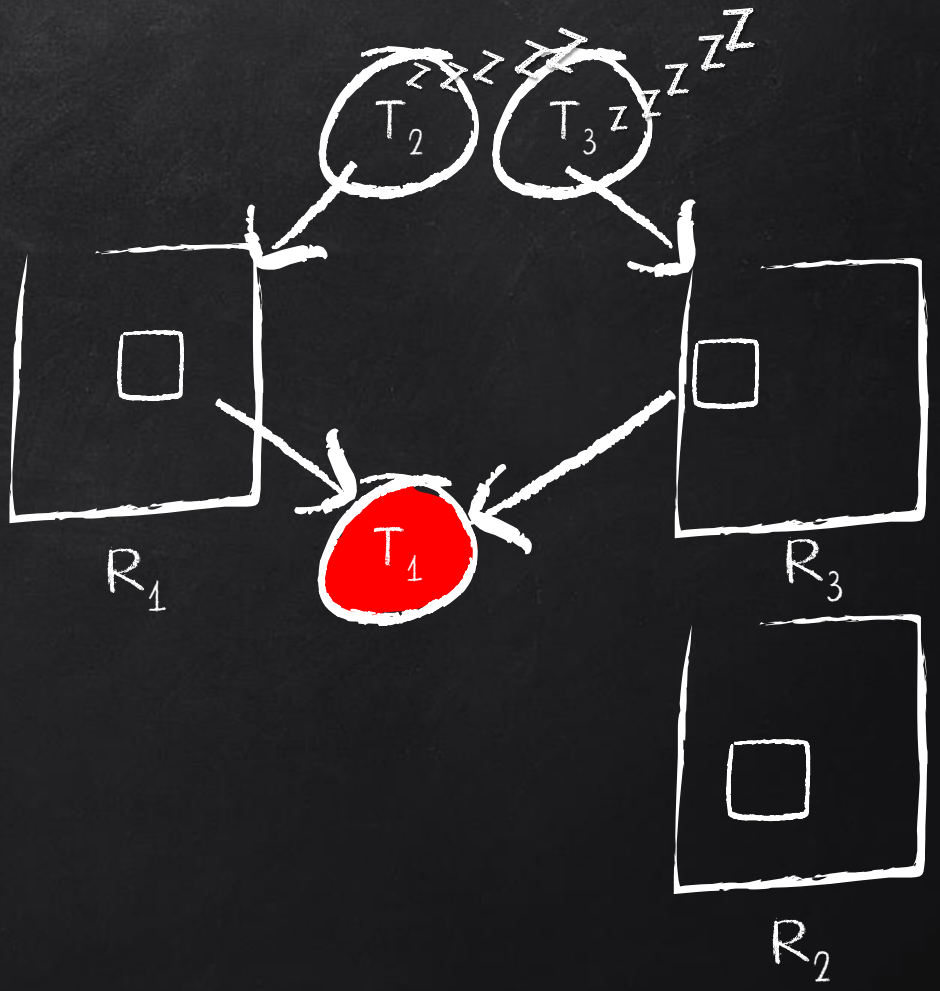


HANDLING DEADLOCKS

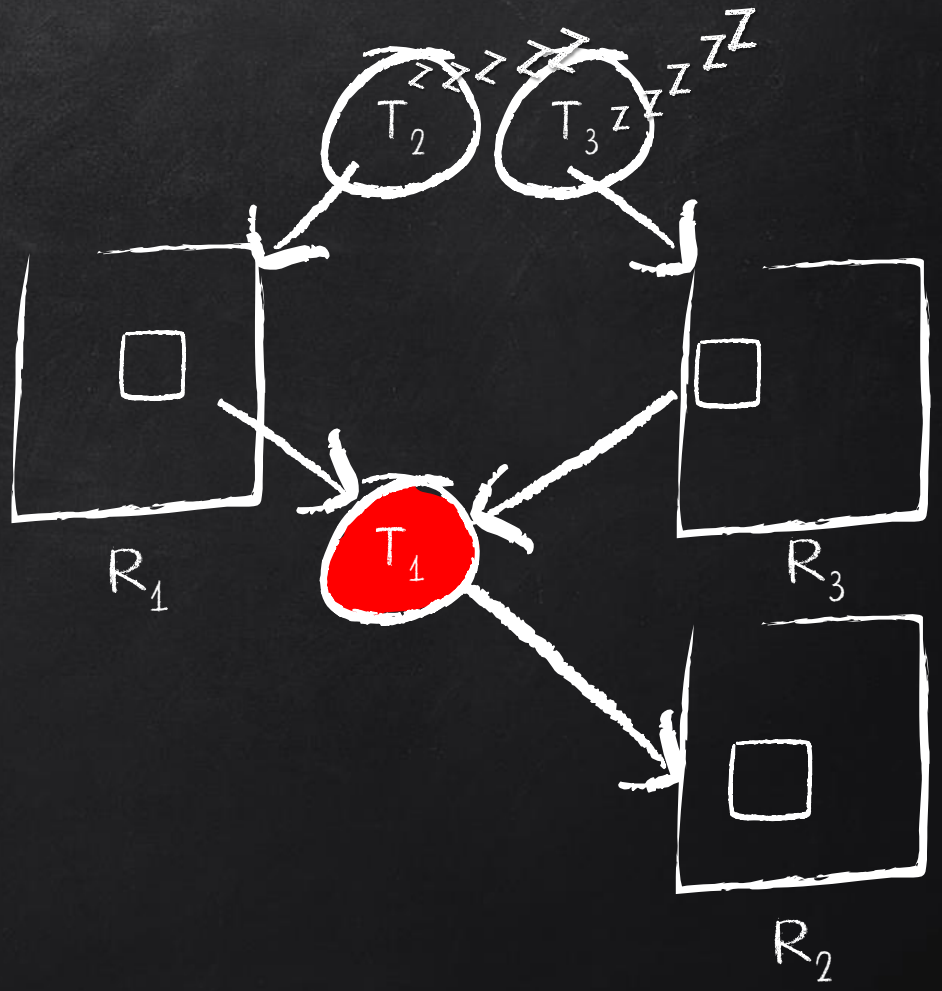
1) Prevention

* Eliminate no preemption

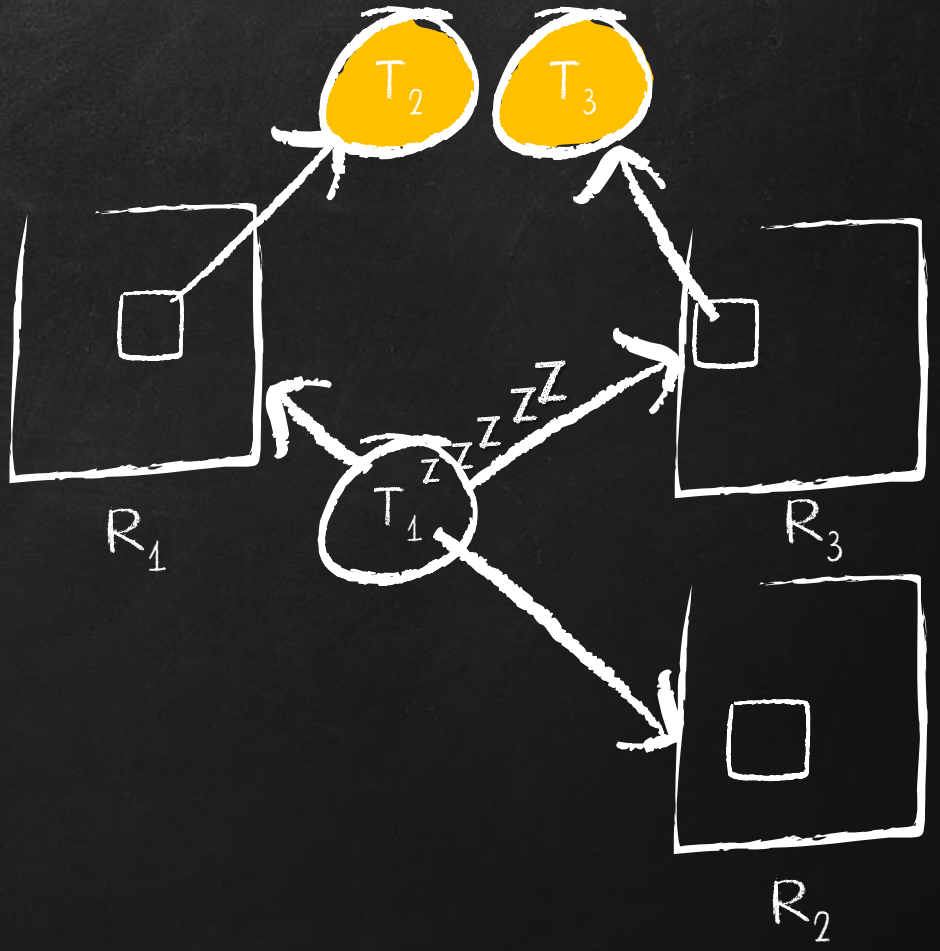
Approach 1: if a task that is holding some resources requests another resource that cannot be immediately allocated to it



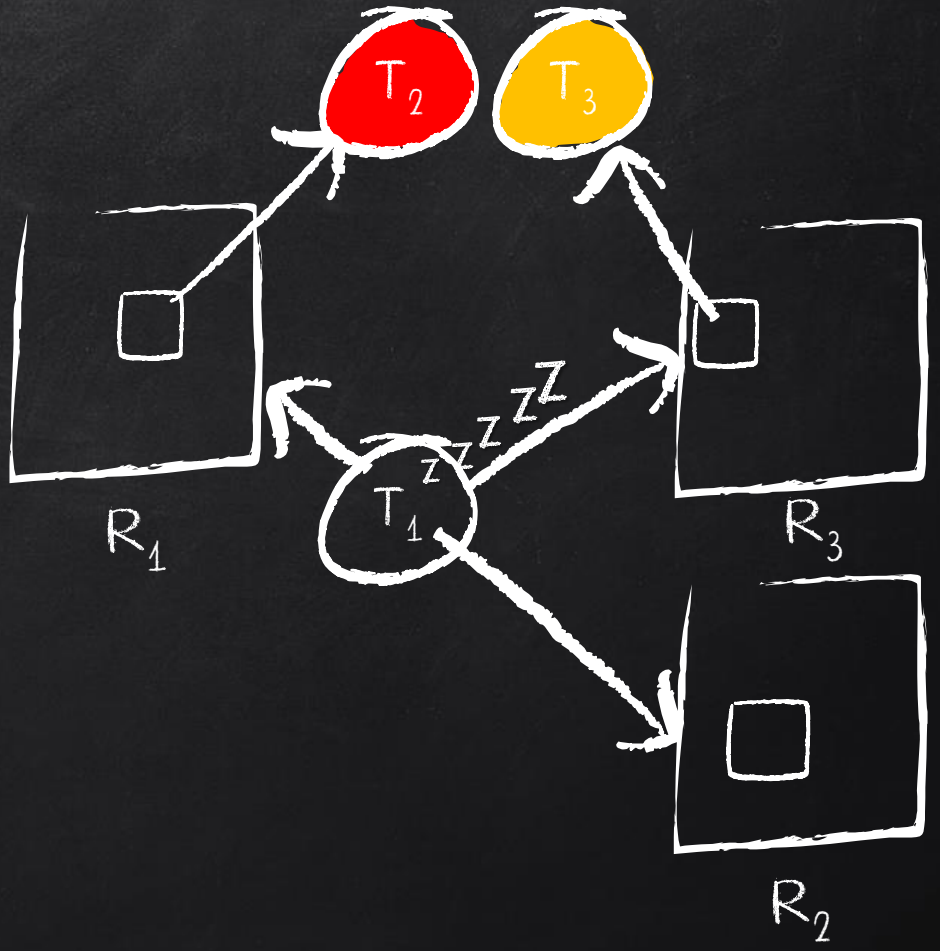
then all resources currently being held are released



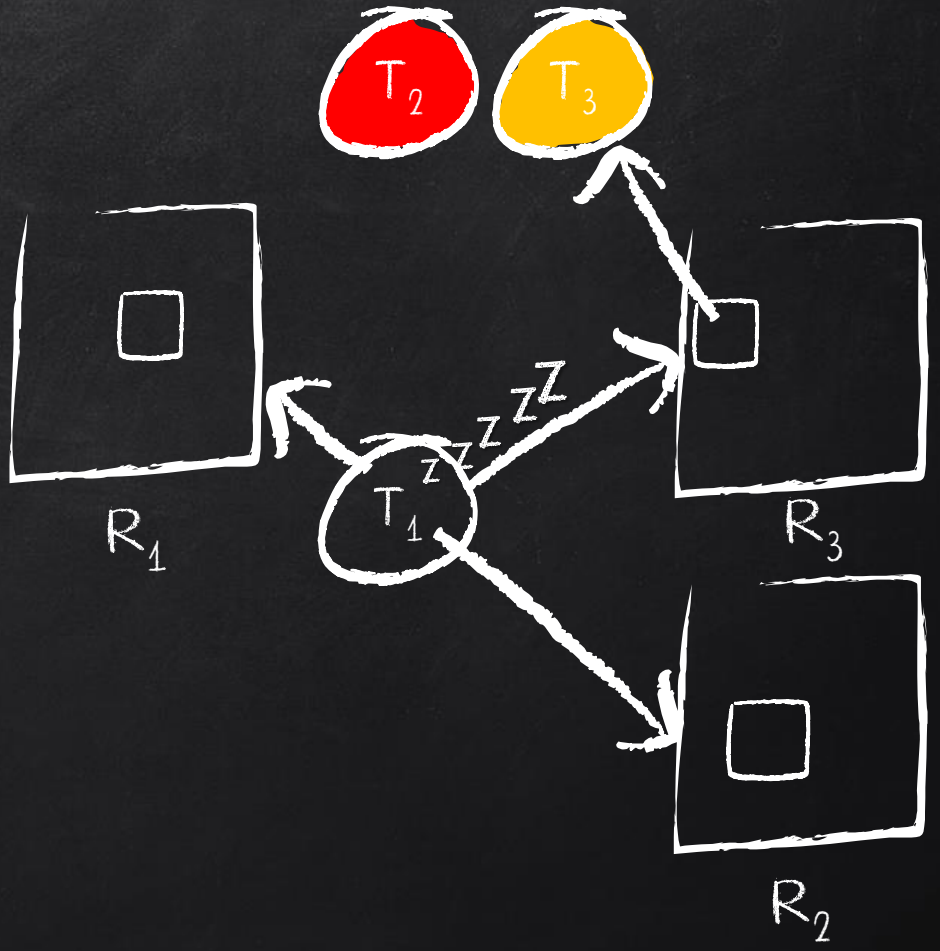
if a task that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released



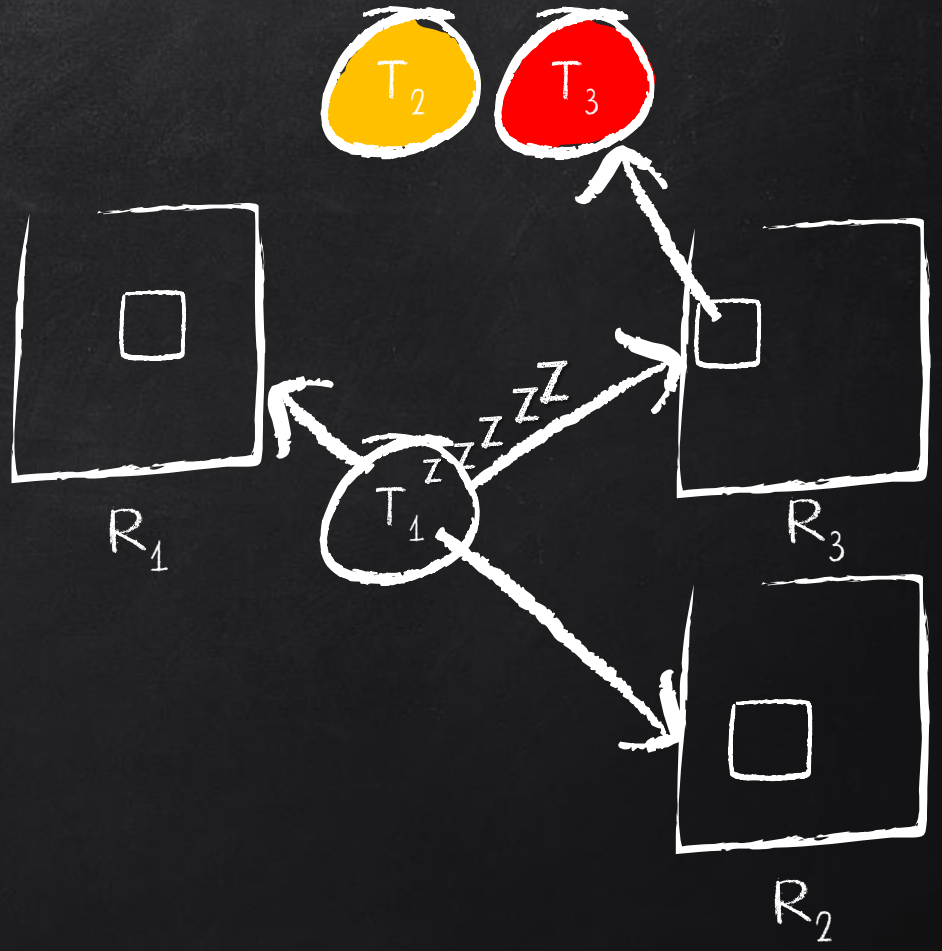
if a task that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released



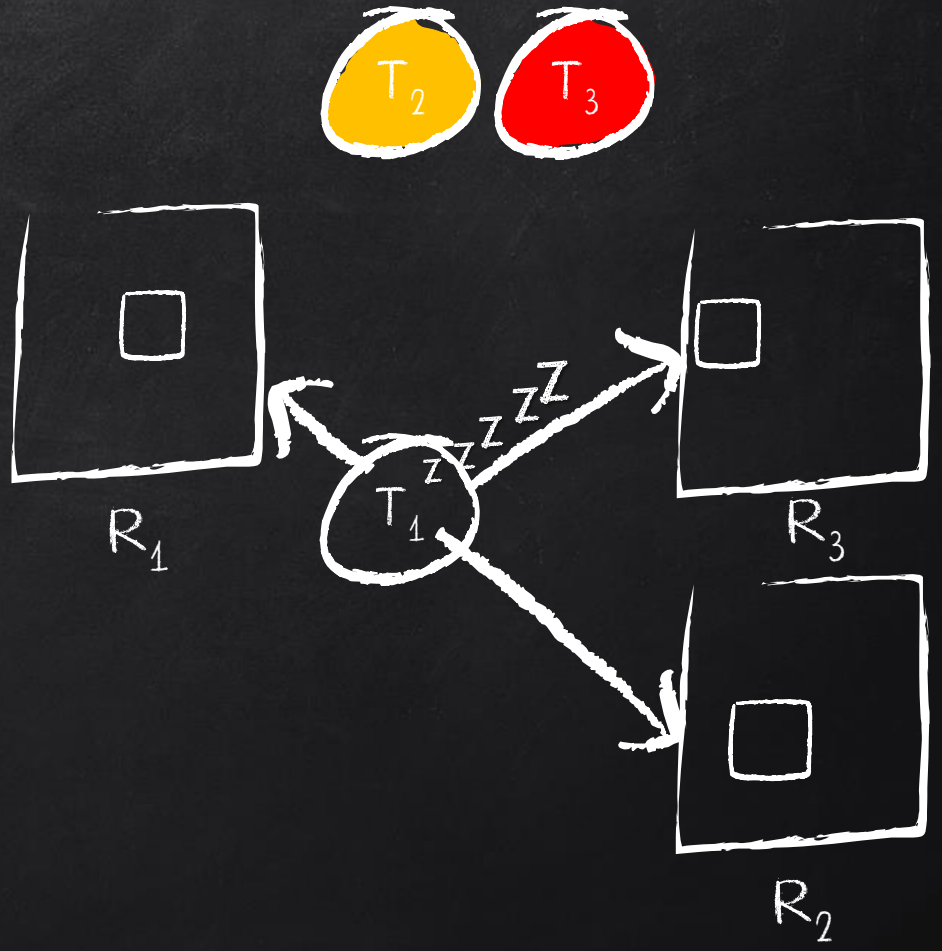
if a task that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released



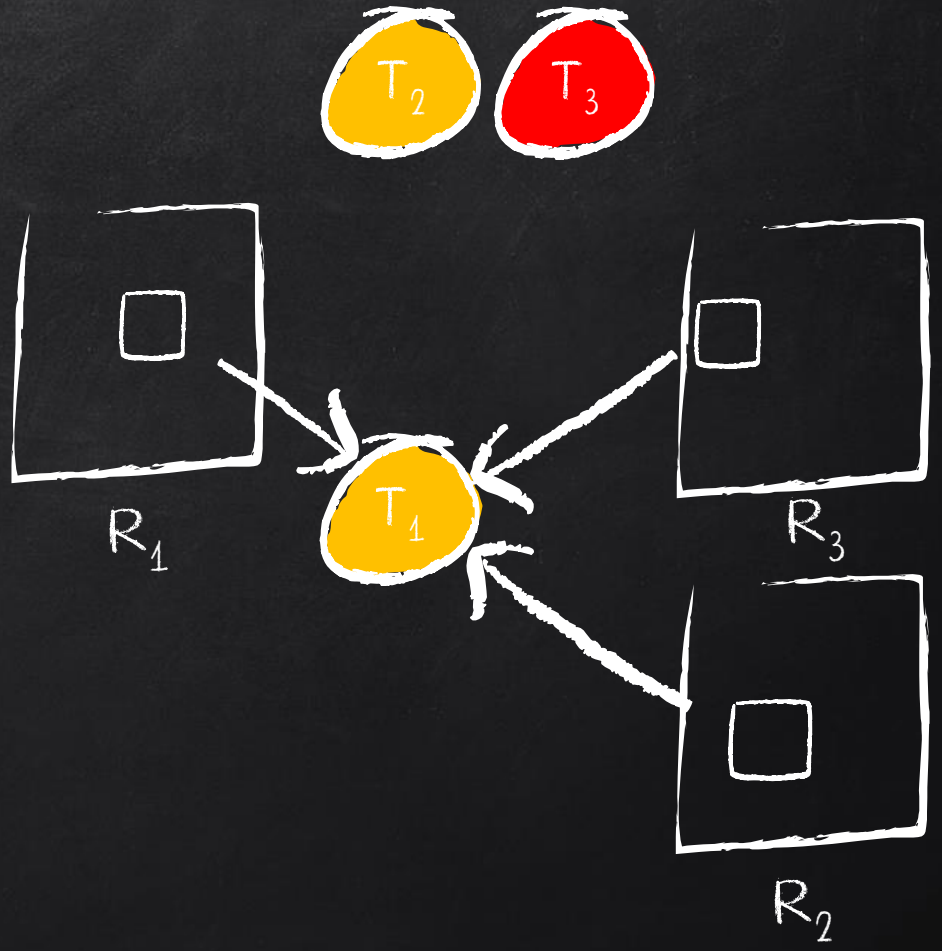
if a task that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released



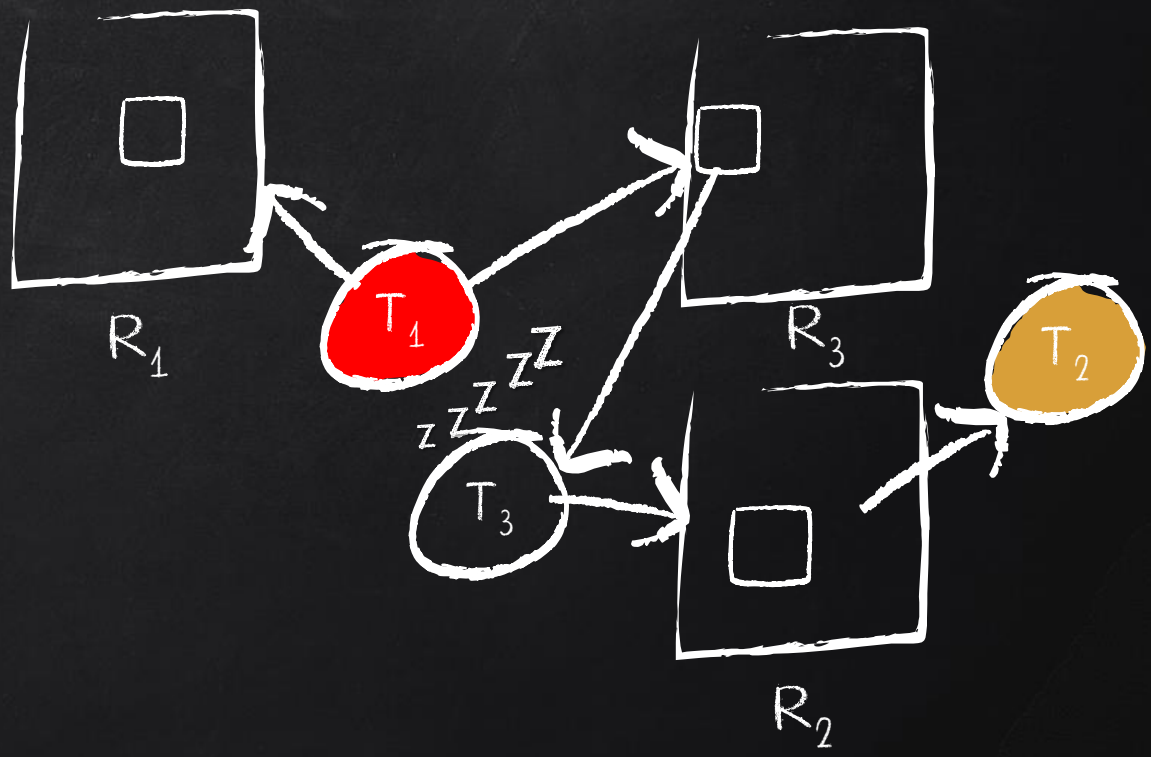
if a task that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released



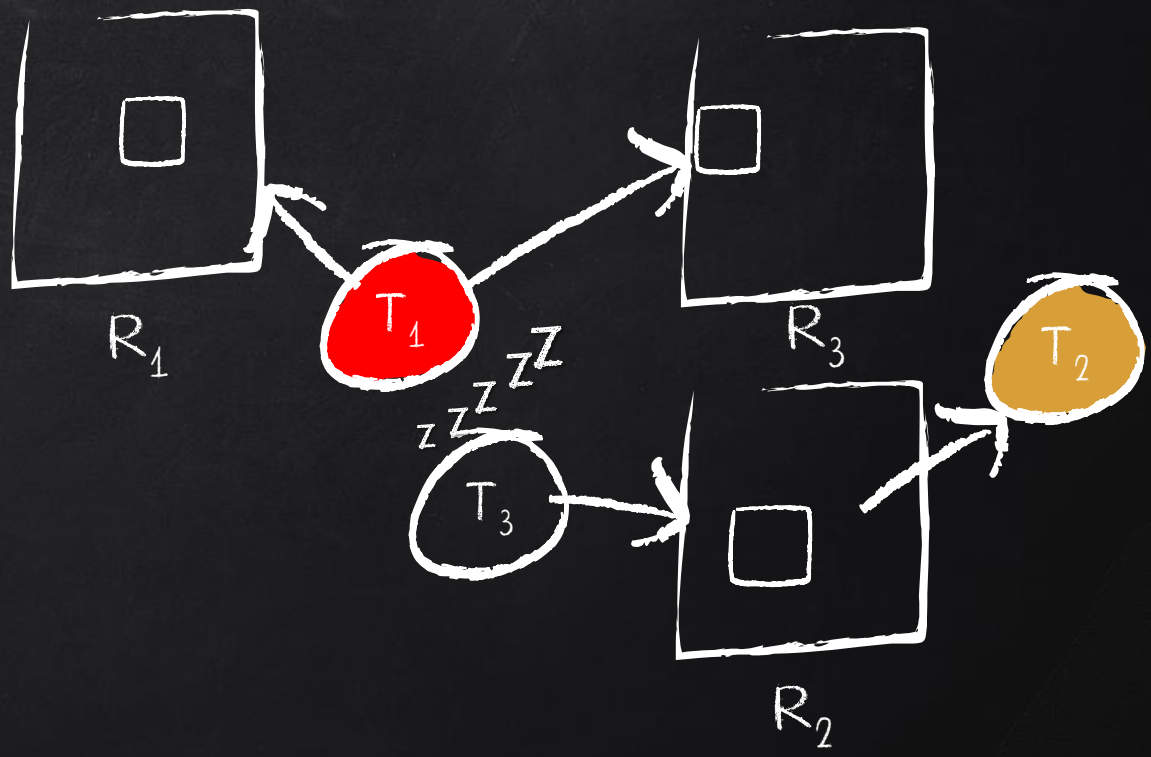
if a task that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released



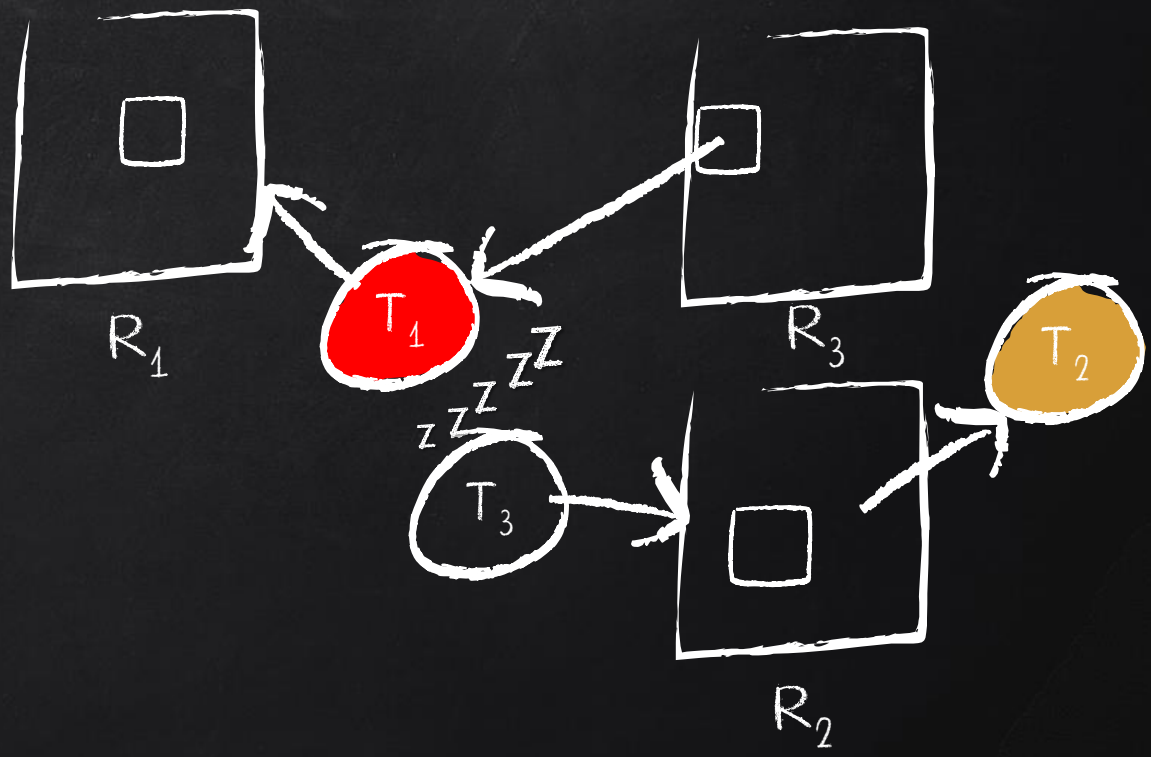
Approach 2: when a resource is requested and not available, then the system looks to see what other processes currently have those resources and are themselves blocked waiting for some other resource.



Approach 2: If such a process is found, then some of their resources may get preempted and added to the list of resources for which the process is waiting.



Approach 2: If such a process is found, then some of their resources may get preempted and added to the list of resources for which the process is waiting.



HANDLING DEADLOCKS

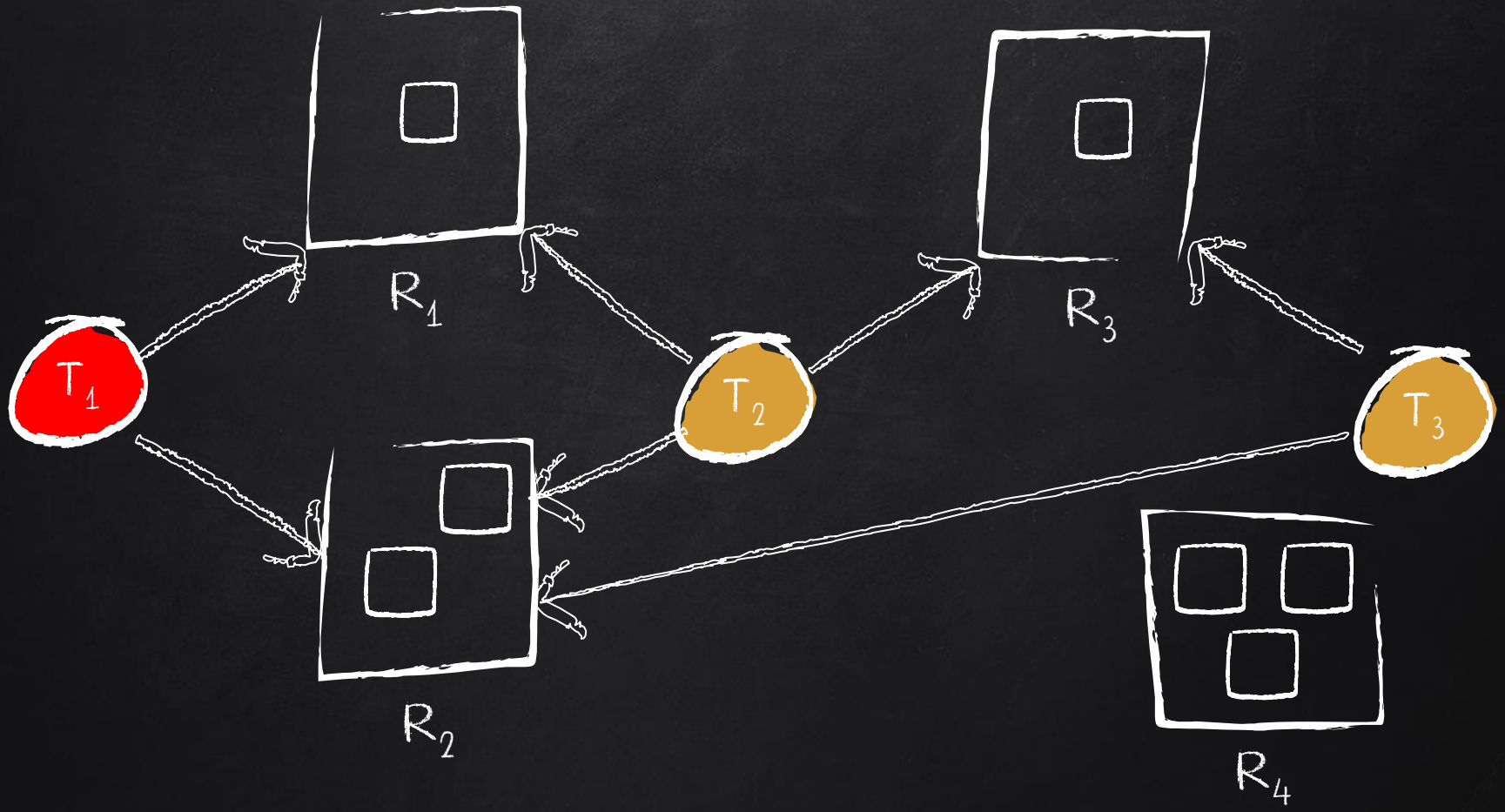
1) Prevention

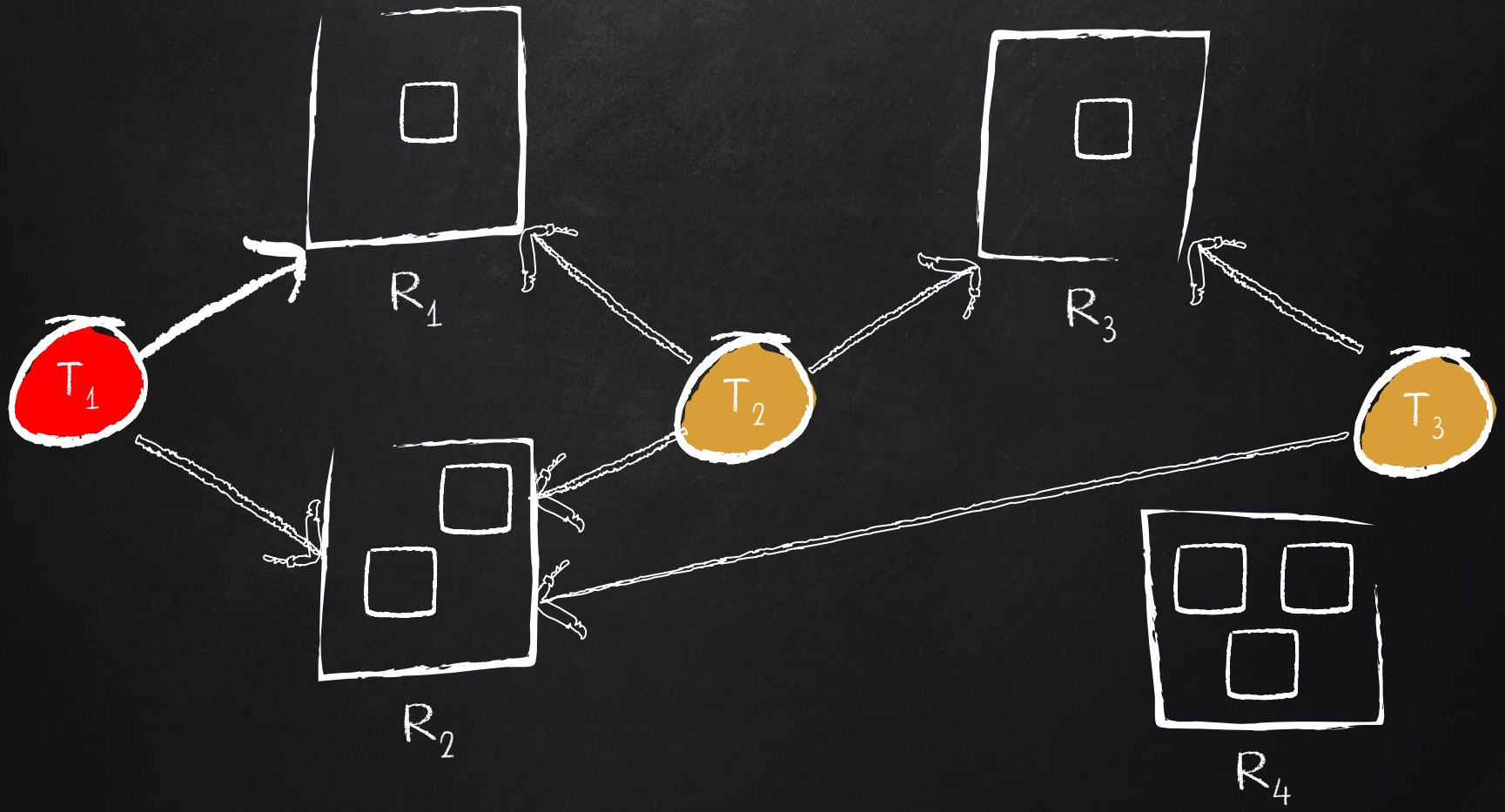
* **Circular Wait:** impose a total ordering of all resource types, and require that each task requests resources in an increasing order of enumeration.

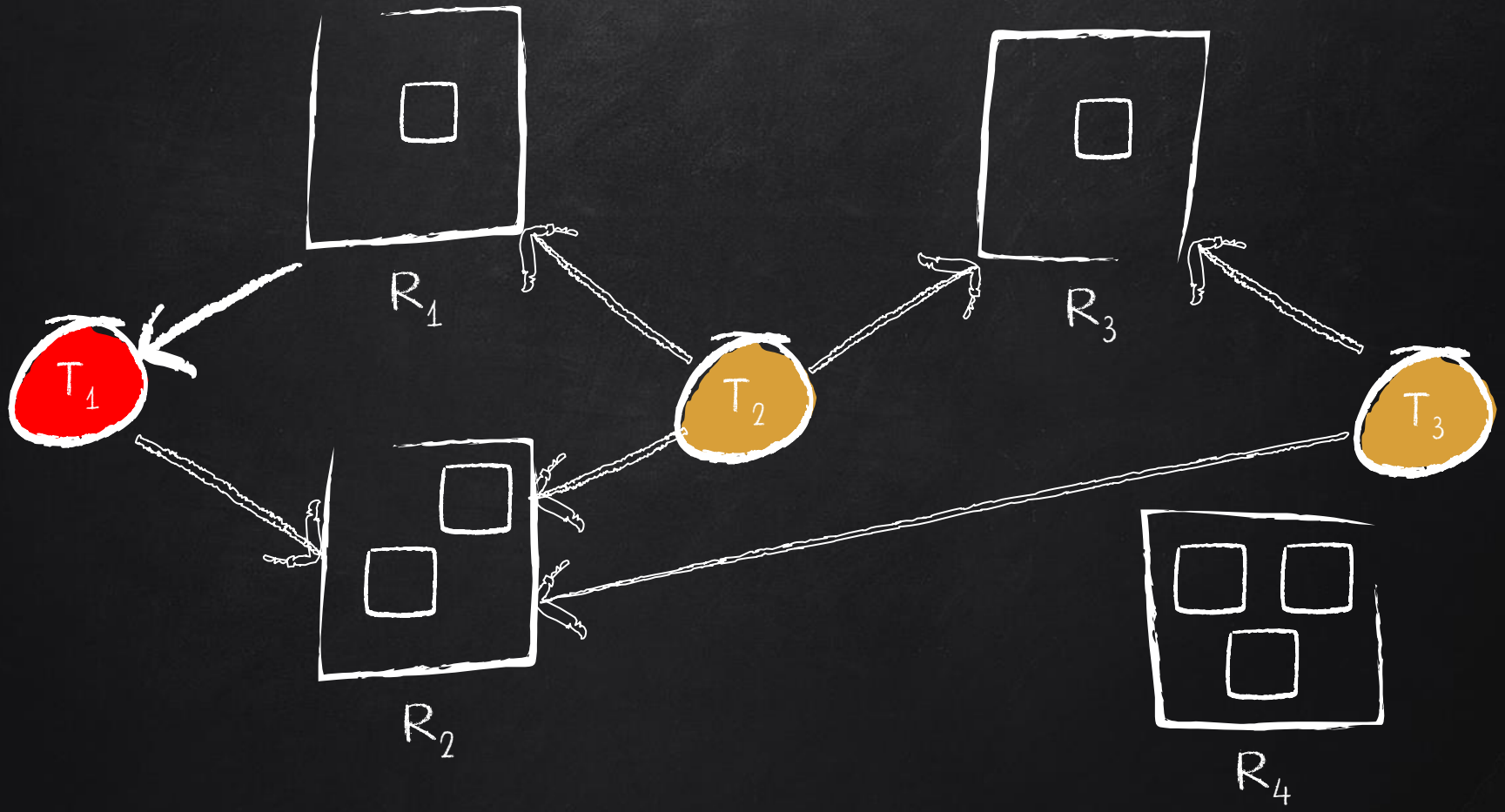
HANDLING DEADLOCKS

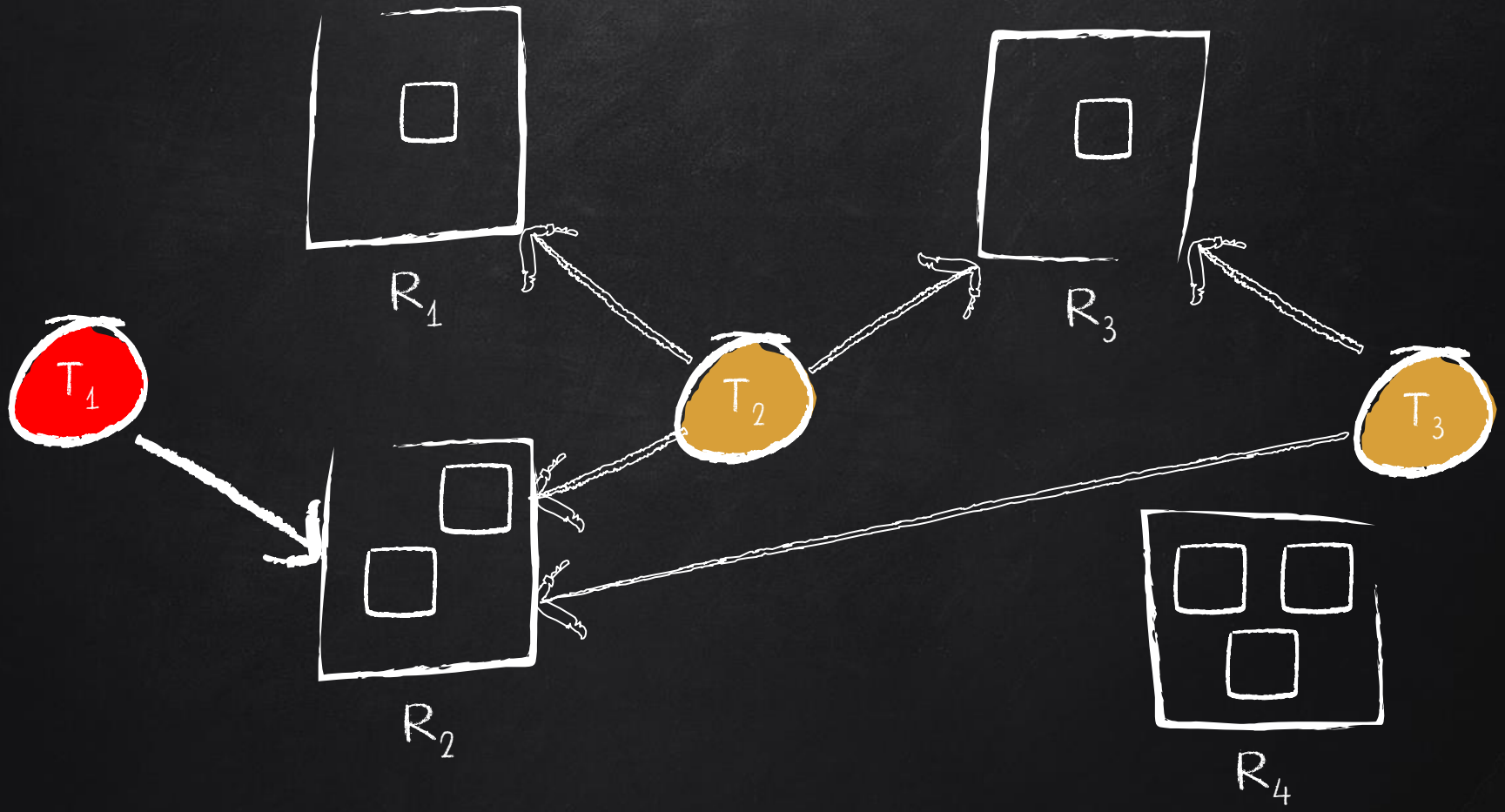
1) Prevention

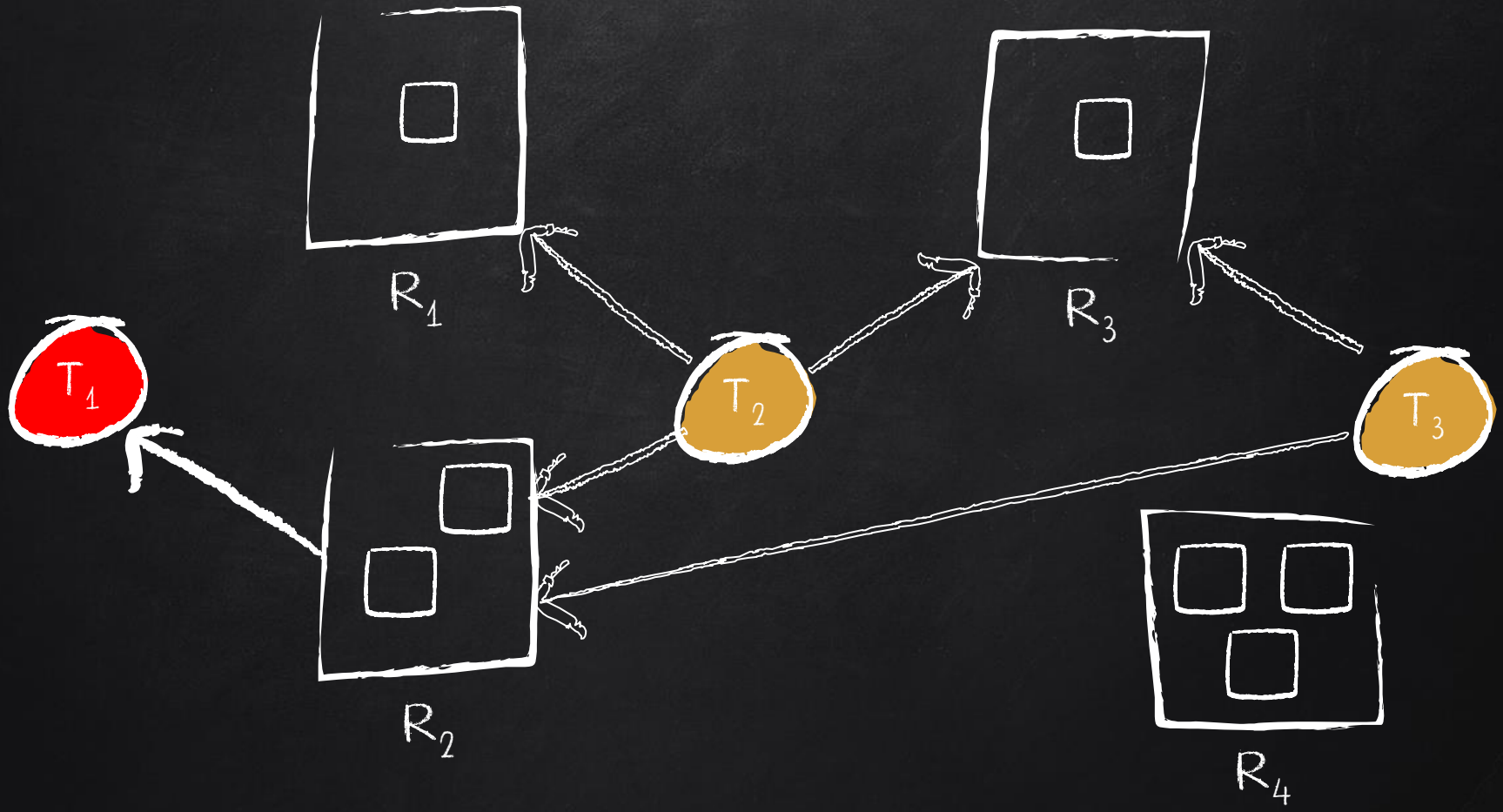
* **Circular Wait:** in other words, in order to request resource R_j, a process must first release all R_i such that $i >= j$.

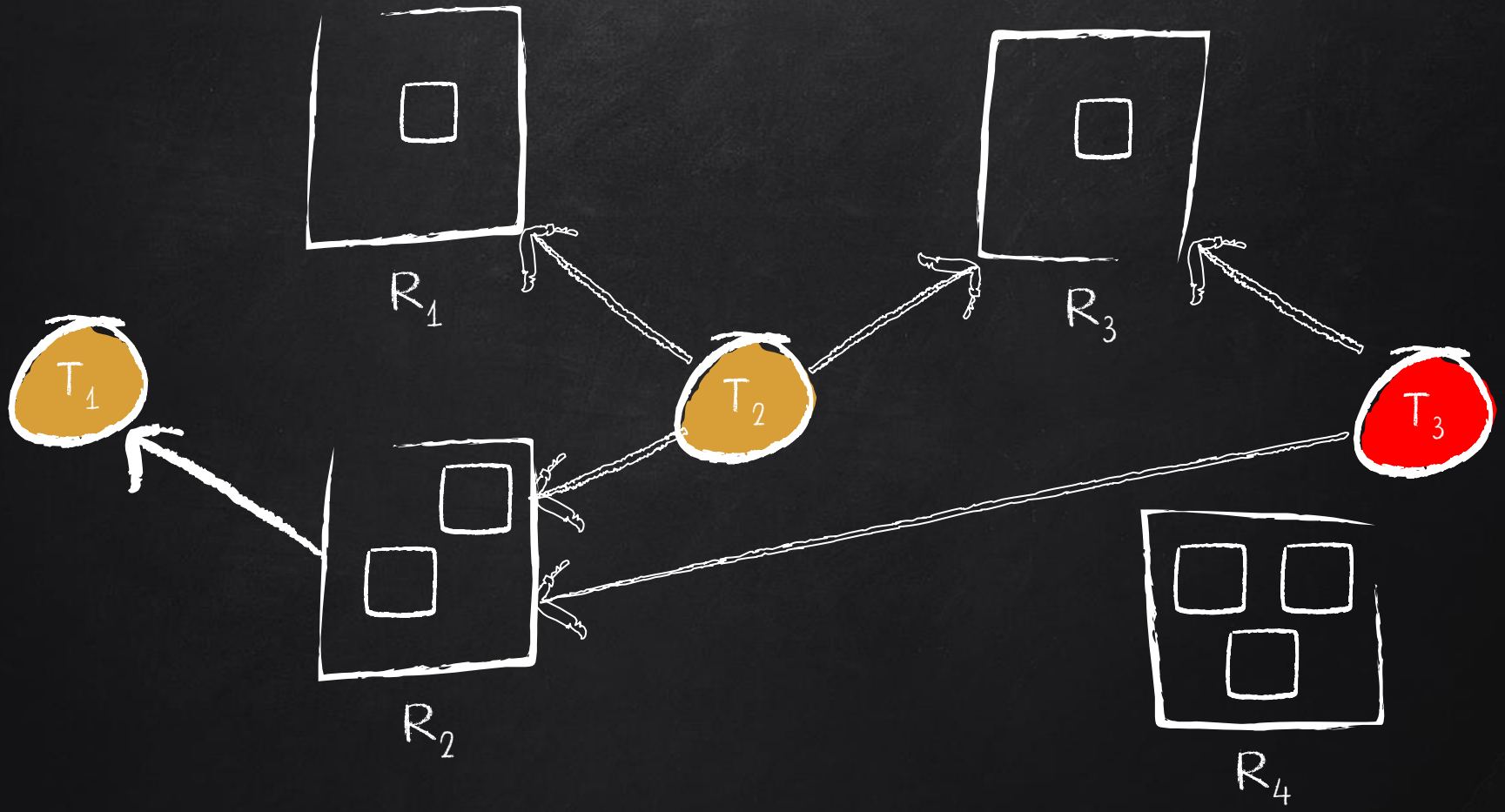

R1
R3
T1
T2
T3
R2
R4

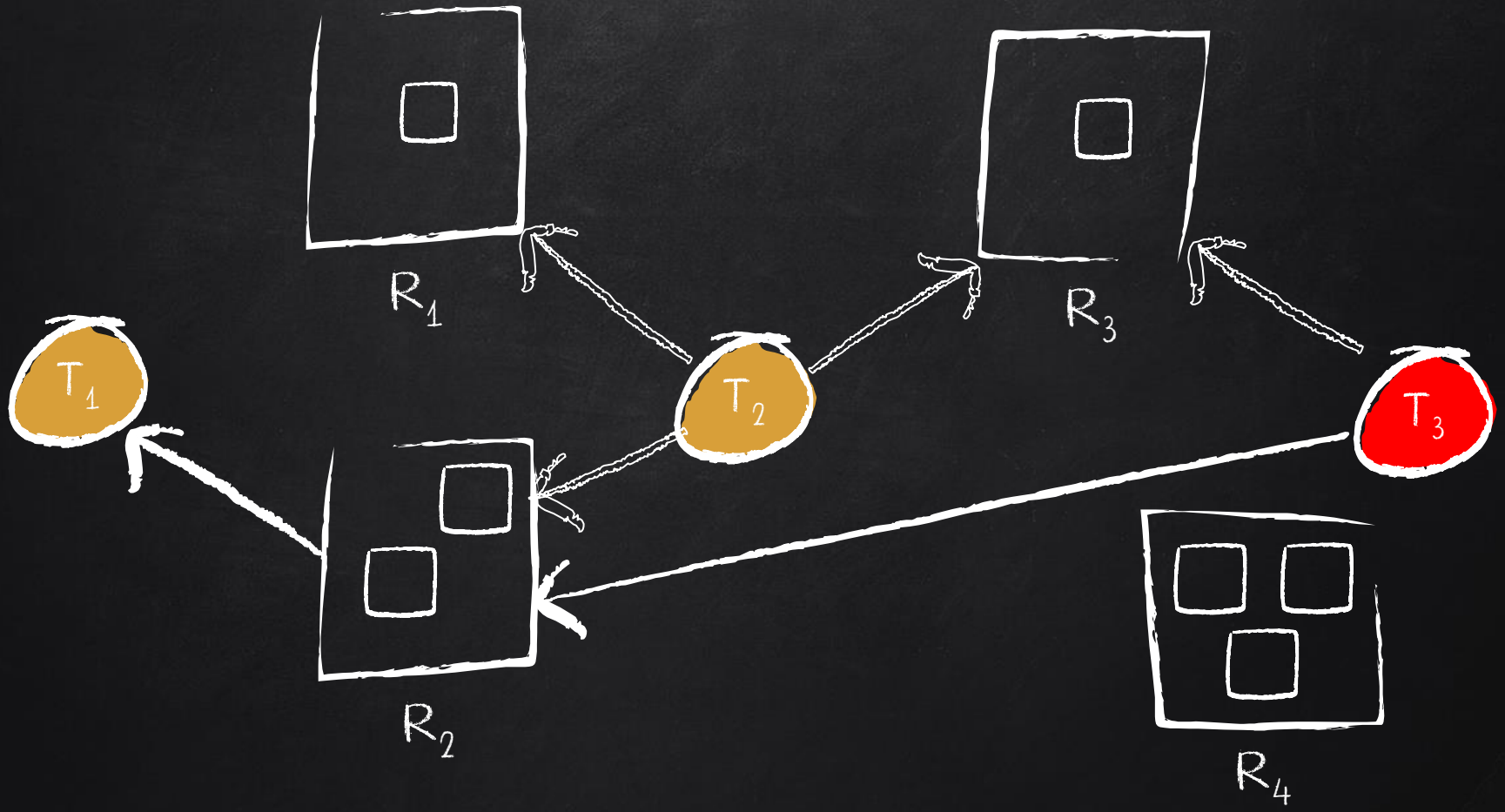

R1
R3
T1
T2
T3
R2
R4

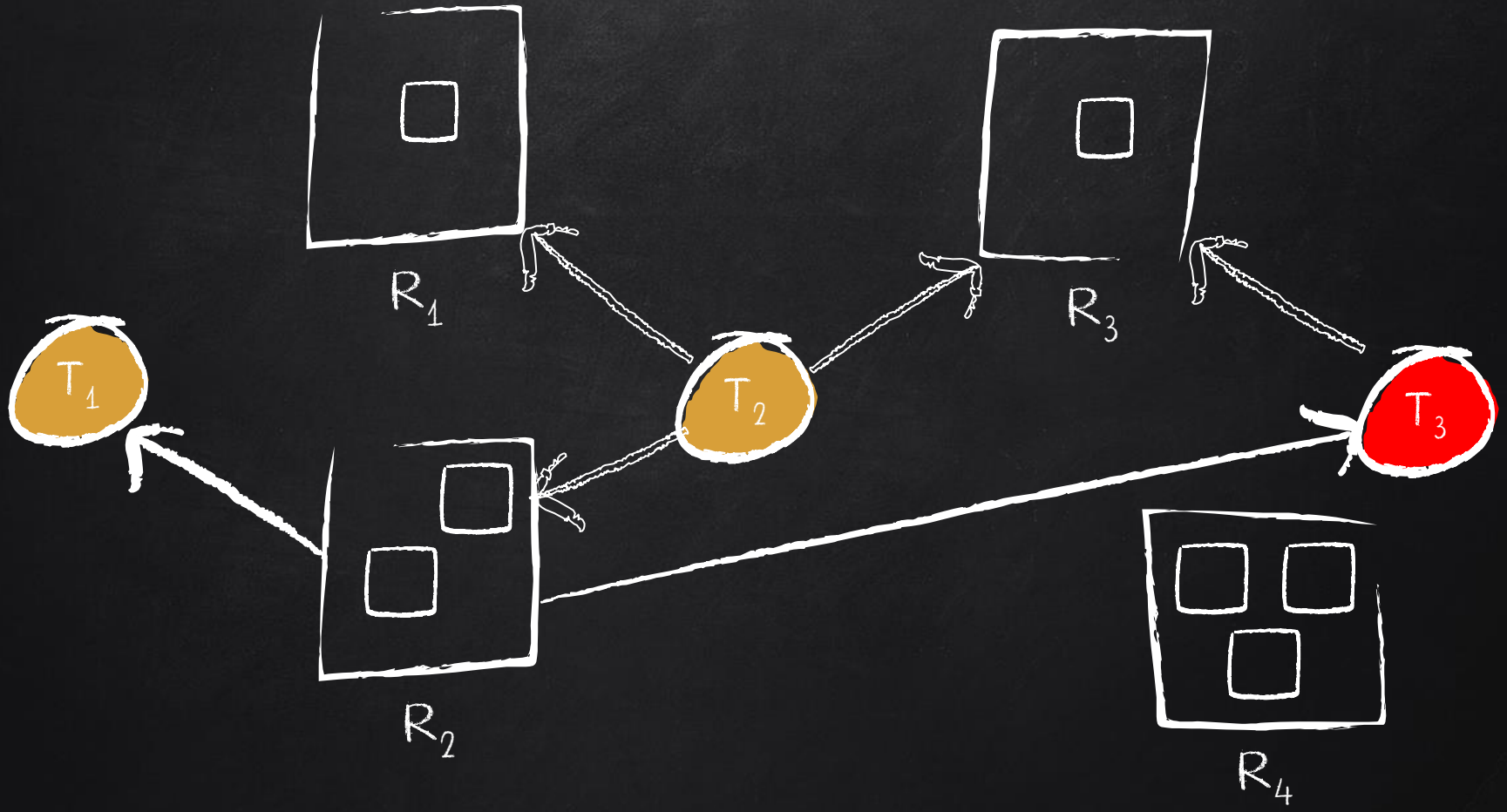

R1
R3
T1
T2
T3
R2
R4

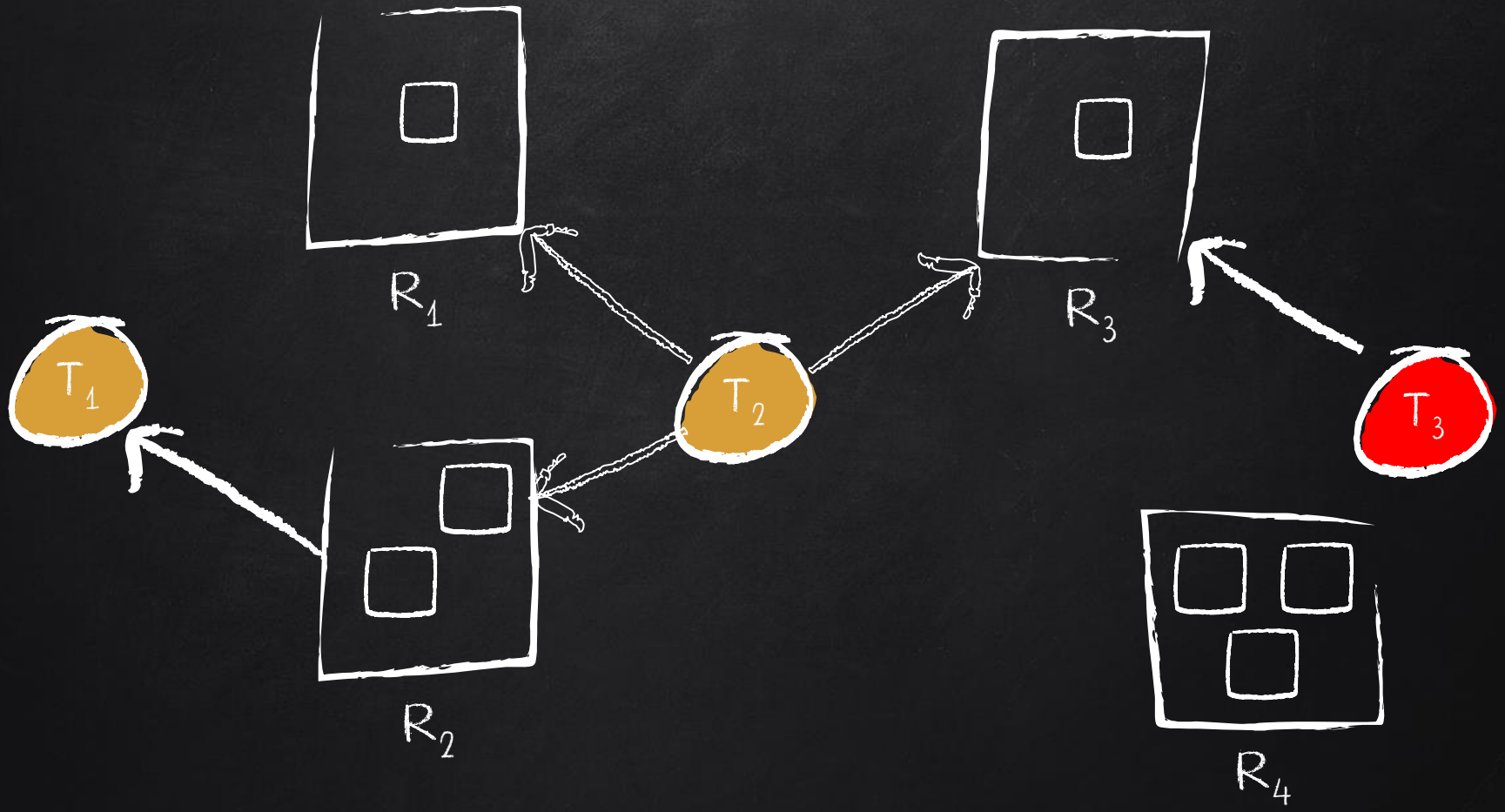

R1
R3
T1
T2
T3
R2
R4

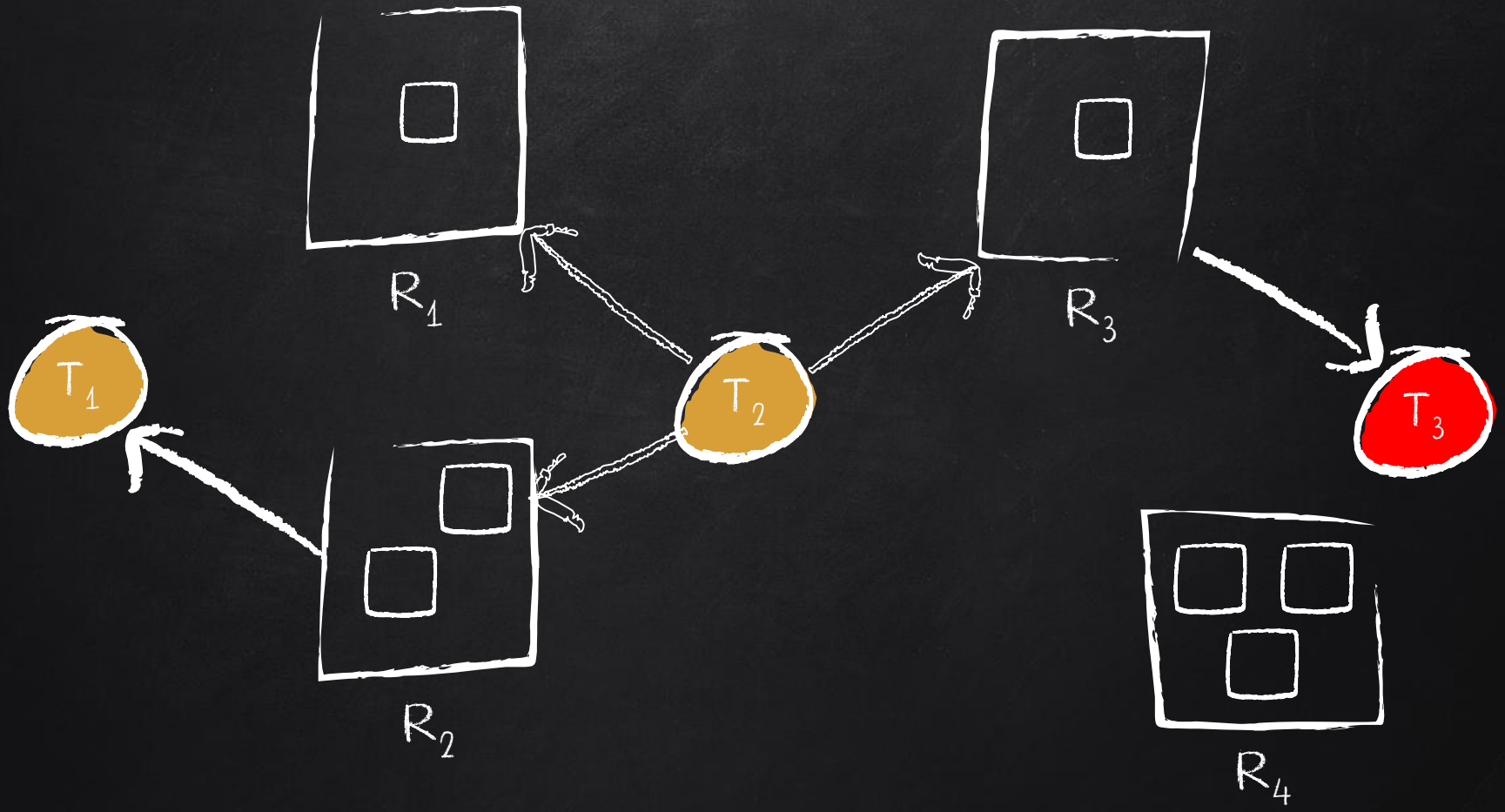

R1
R3
T1
T2
T3
R2
R4


R1
R3
T1
T2
T3
R2
R4


R1
R3
T1
T2
T3
R2
R4


R1
R3
T1
T2
T3
R2
R4


R1
R3
T1
T2
T3
R2
R4


R1
R3
T1
T2
T3
R2
R4

HANDLING DEADLOCKS

2) Avoidance

The general idea is to prevent deadlocks from ever happening, by preventing at least one of the aforementioned conditions.

This requires more information about each process, AND tends to lead to low device utilization.

HANDLING DEADLOCKS

2) Avoidance

The scheduler needs to know the maximum number of each resource that a process might potentially use or exactly what resources may be needed in what order.

When a scheduler sees that starting a process or granting resource requests may lead to future deadlocks, then that process is just not started or the request is not granted.

HANDLING DEADLOCKS

3) Ignore deadlocks

HANDLING DEADLOCKS

3) Ignore deadlocks: it is assumed that a deadlock will never occur. Used when the time intervals between occurrences of deadlocks are large and the data loss incurred each time is tolerable; UNIX-like OSs and Windows.

LIVELOCK

Similar to a deadlock, except that the states of the processes involved in the livelock constantly change with regard to one another, none progressing.

LIVELOCK

Special case of resource starvation; the general definition only states that a specific process is not progressing.

```
... // task 0
flag[0] = true;
while (flag[1]);
... // Critical section
flag[0] = false;
... // Remainder section
```

```
... // task 1
flag[1] = true;
while (flag[0]);
... // Critical section
flag[1] = false;
... // Remainder section
```

```
... // task 0
flag[0] = true;
while (flag[1]){
  flag[0] = false;
  flag[0] = true;
}
... // Critical section
flag[0] = false;
... // Remainder section
```

```
... // task 1
flag[1] = true;
while (flag[0]){
  flag[1] = false;
  flag[1] = true;
}
... // Critical section
flag[0] = false;
... // Remainder section
```

```
... // task 0
flag[0] = true;
while (flag[1]);
... // Critical section
flag[0] = false;
... // Remainder section
```

```
... // task 1
flag[1] = true;
while (flag[0]);
... // Critical section
flag[1] = false;
... // Remainder section
```

DEADLOCK POSSIBILITY

LIVELOCK POSSIBILITY

```
... // task 0
flag[0] = true;
while (flag[1]){
  flag[0] = false;
  flag[0] = true;
}
... // Critical section
flag[0] = false;
... // Remainder section
```

```
... // task 1
flag[1] = true;
while (flag[0]){
  flag[1] = false;
  flag[1] = true;
}
... // Critical section
flag[0] = false;
... // Remainder section
```

DEADLOCK POSSIBILITY

```
... // task 0
flag[0] = true;
while (flag[1]);
... // Critical section
flag[0] = false;
... // Remainder section
```

```
... // task 1
flag[1] = true;
while (flag[0]);
... // Critical section
flag[1] = false;
... // Remainder section
```

```
... // task 0
flag[0] = true;
while (flag[1]){
  flag[0] = false;
  flag[0] = true;
}
... // Critical section
flag[0] = false;
... // Remainder section
```

```
... // task 1
flag[1] = true;
while (flag[0]){
  flag[1] = false;
  flag[1] = true;
}
... // Critical section
flag[0] = false;
... // Remainder section
```

```
... // task 0
flag[0] = true;
while (flag[1]);
... // Critical section
flag[0] = false;
... // Remainder section
```

```
... // task 1
flag[1] = true;
while (flag[0]);
... // Critical section
flag[1] = false;
... // Remainder section
```

DEADLOCK POSSIBILITY

LIVELOCK POSSIBILITY

```
... // task 0
flag[0] = true;
while (flag[1]){
  flag[0] = false;
  flag[0] = true;
}
... // Critical section
flag[0] = false;
... // Remainder section
```

```
... // task 1
flag[1] = true;
while (flag[0]){
  flag[1] = false;
  flag[1] = true;
}
... // Critical section
flag[0] = false;
... // Remainder section
```

thanks!

Any questions?

You can find me at
Cristian.koliver@ufsc.br