DEADLOCK

Contents

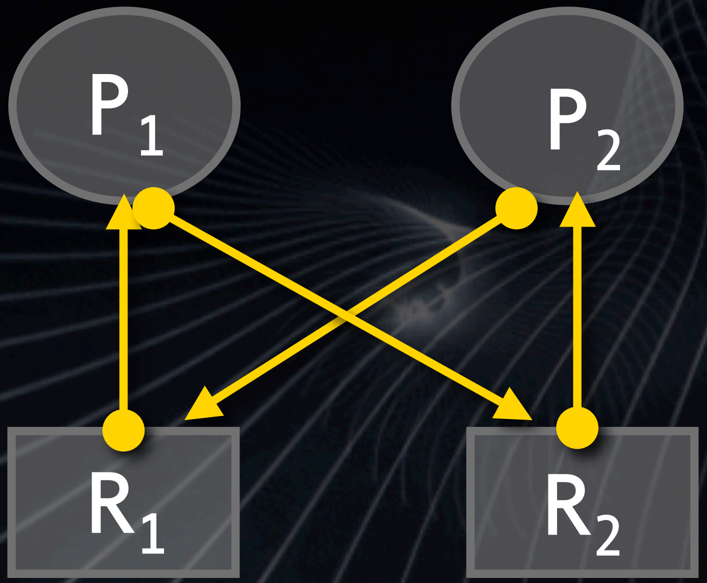
→ Principles of deadlock

→ Deadlock prevention

→ Deadlock detection

Deadlock

A set of processes is deadlocked if each process in the set is waiting for an event that only another process in the set can cause.

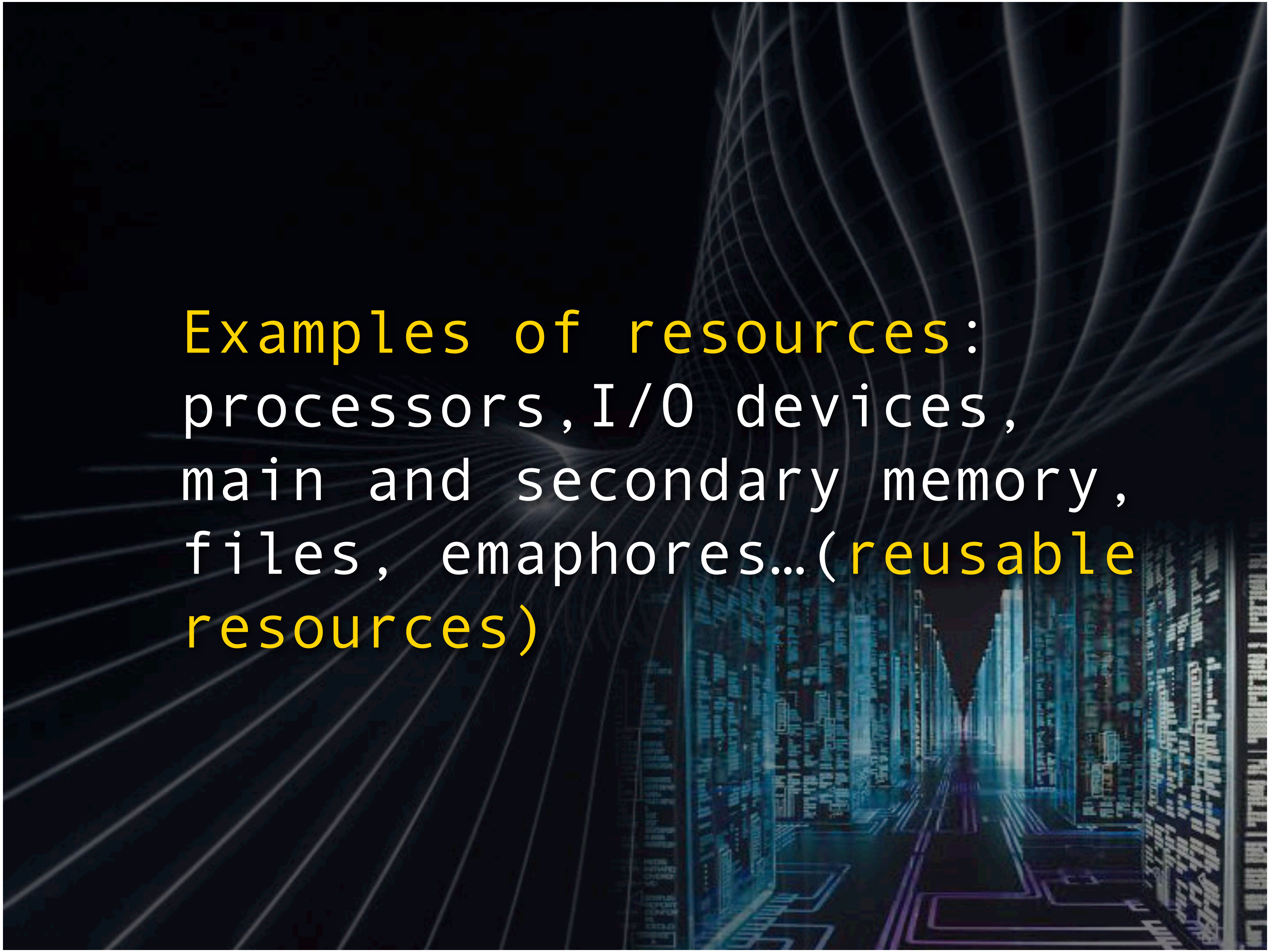


P_1

P_2

R_1

R_2



Examples of resources:

processors,I/O devices, main and secondary memory, files, emaphores…(reusable resources)

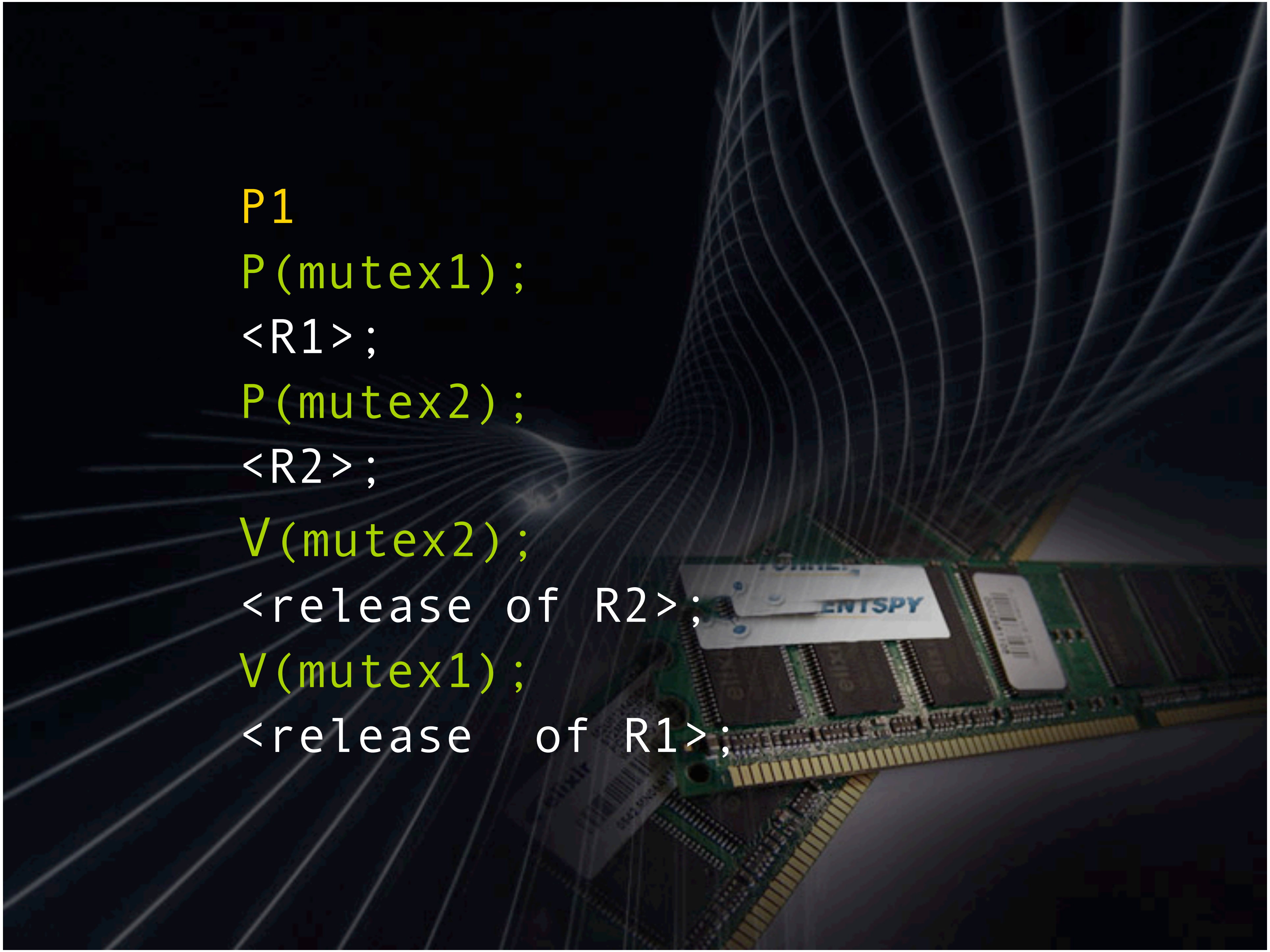
Utilization protocol

request

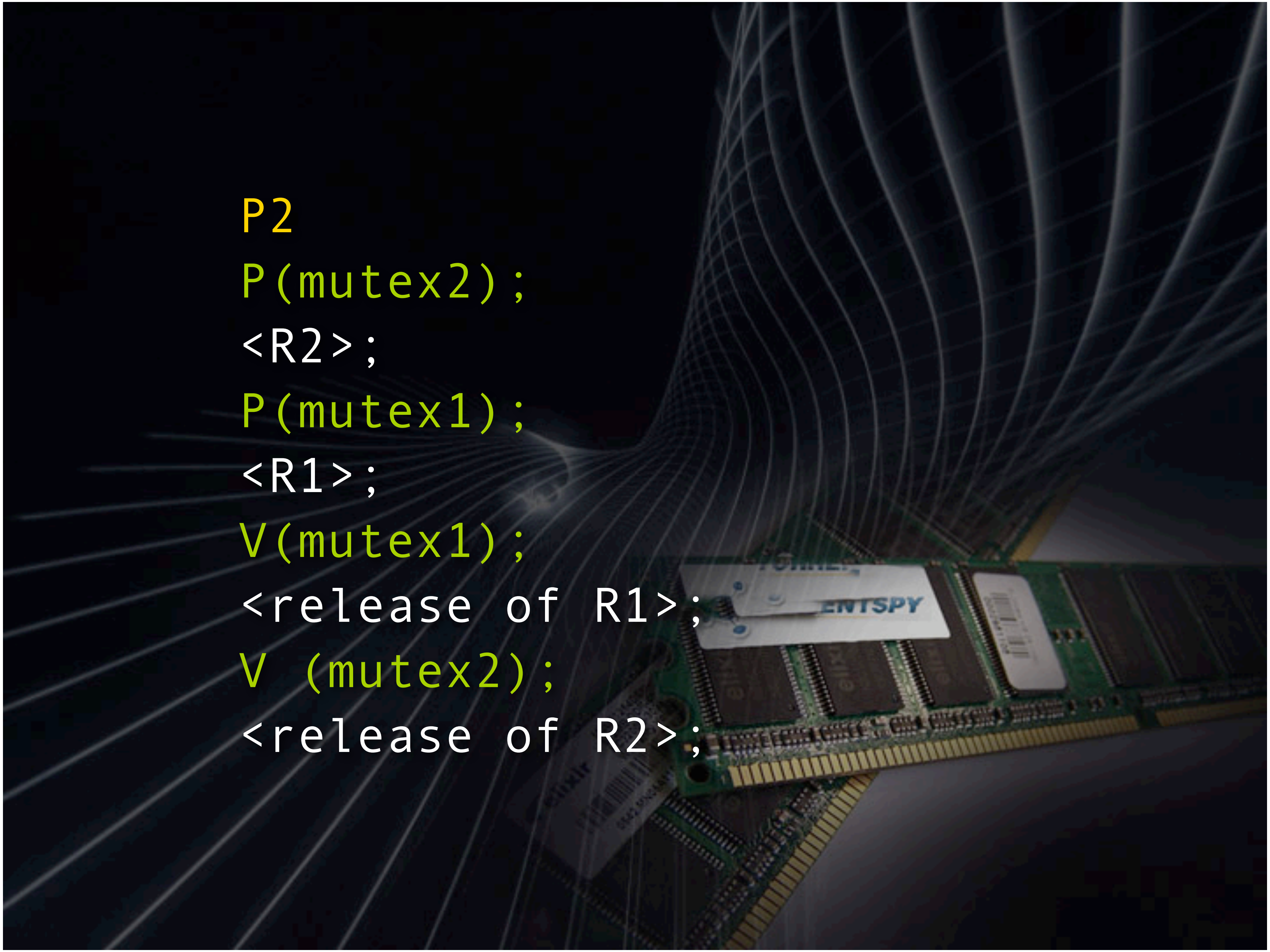
(the process can be blocked)

use

release



```
P1
P(mutex1);
<R1>;
P(mutex2);
<R2>;
V(mutex2);
<release of R2>;
V(mutex1);
<release  of R1>;
```



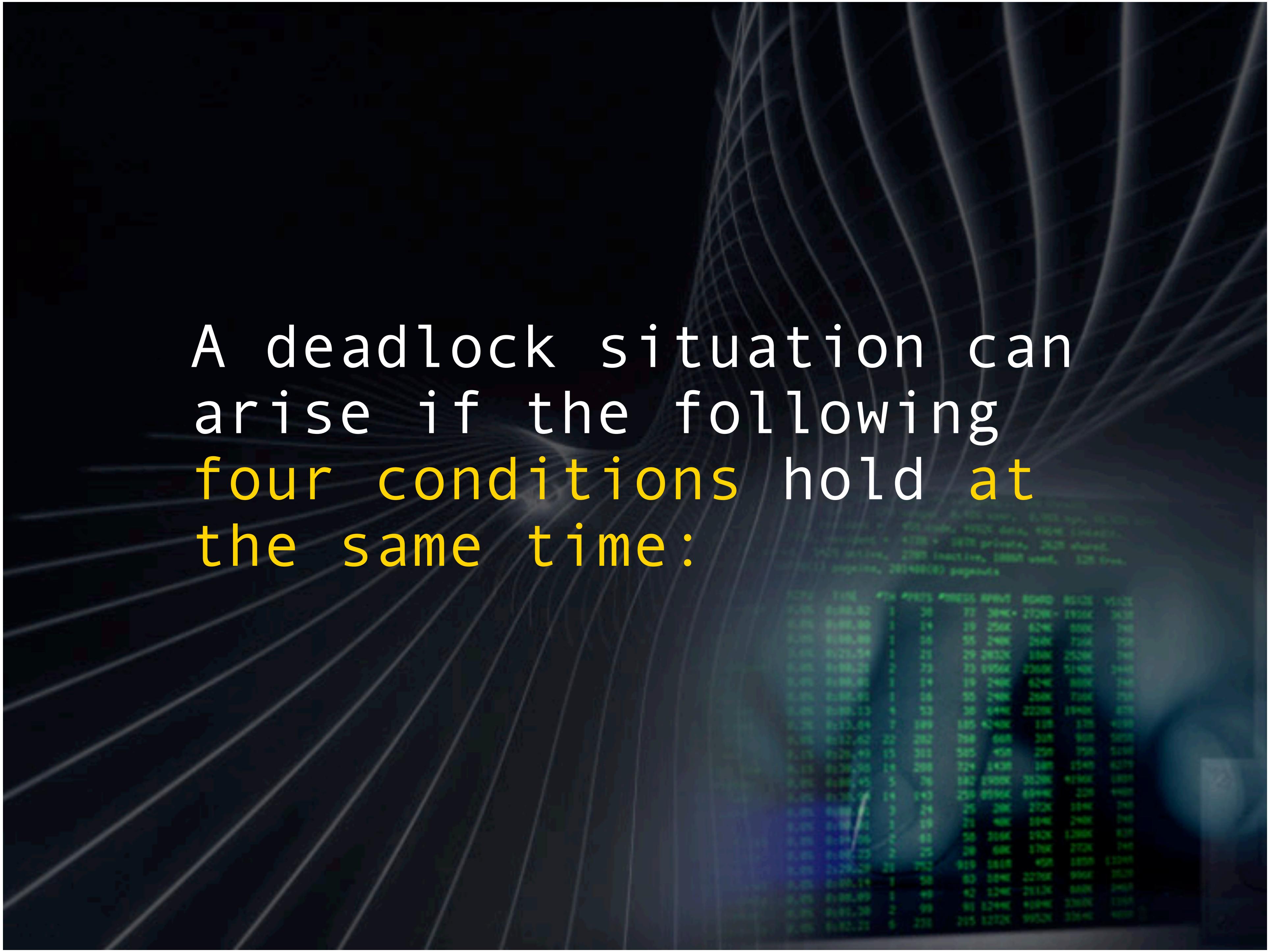
```
P2
P(mutex2);
<R2>;
P(mutex1);
<R1>;
V(mutex1);
<release of R1>;
V (mutex2);
<release of R2>;
```

A deadlock situation derives from a race condition occurred to some involved processes

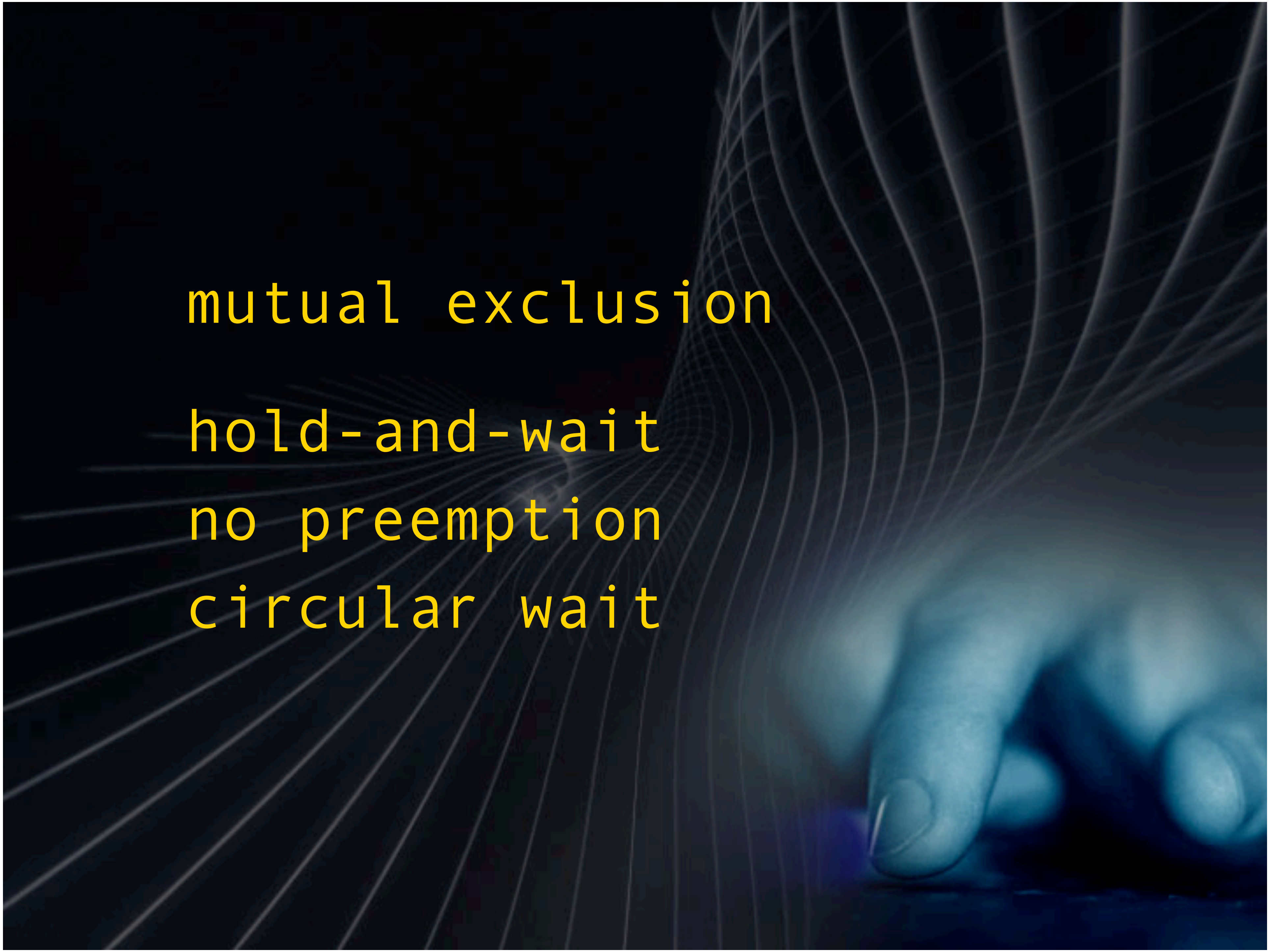
Conditions for deadlock

P1, P2, … , Pn:
a set of processes

R1, R2, … , Rm:
a set of resource types



A deadlock situation can arise if the following **four conditions** hold **at the same time:**



mutual exclusion

hold-and-wait

no preemption

circular wait

All four conditions must hold for deadlock to occur

System resource allocation graph

vertices:

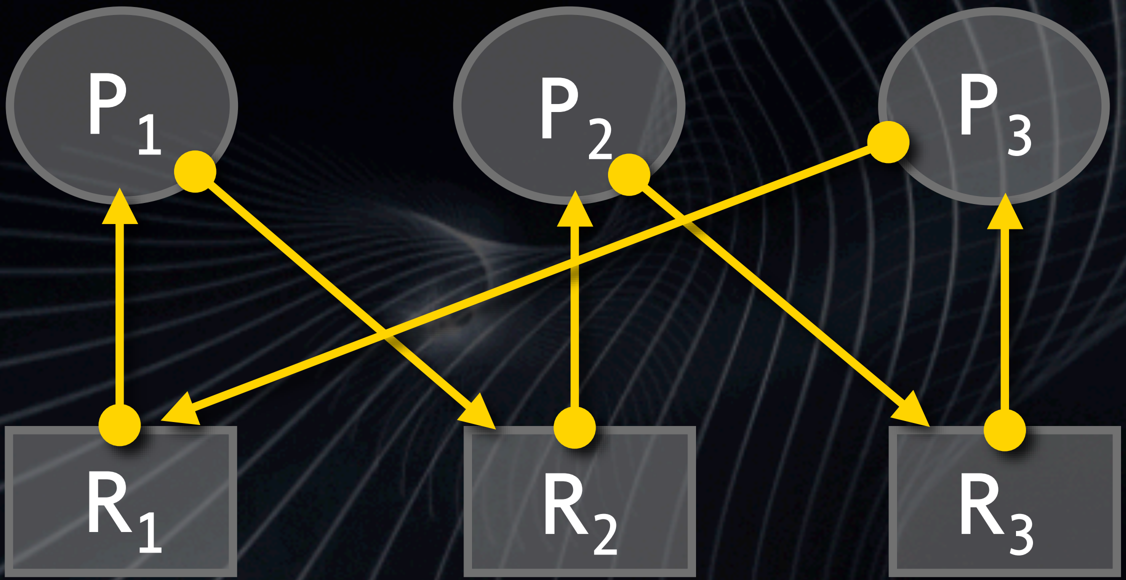
P=(P1,P2,...,Pn)

R=(R1,R2,...,Rm)

edges:

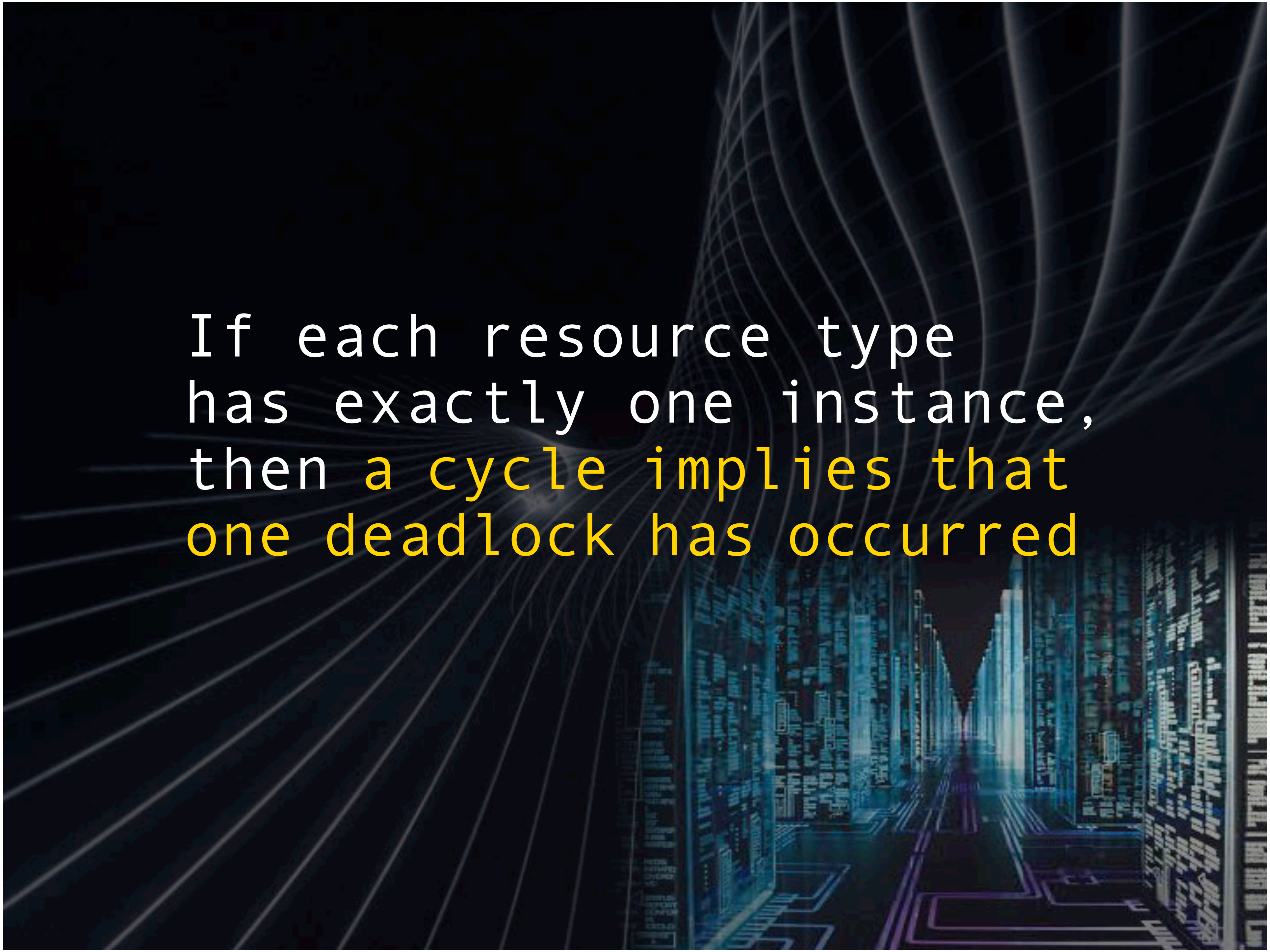
request edge Pi → Rj

assignment edge Rj → Pi


P_1
P_2
P_3
R_1
R_2
R_3

If the graph does not contains cycles, then no process is deadlocked

If the graph contain one cycle, then a deadlock may exist

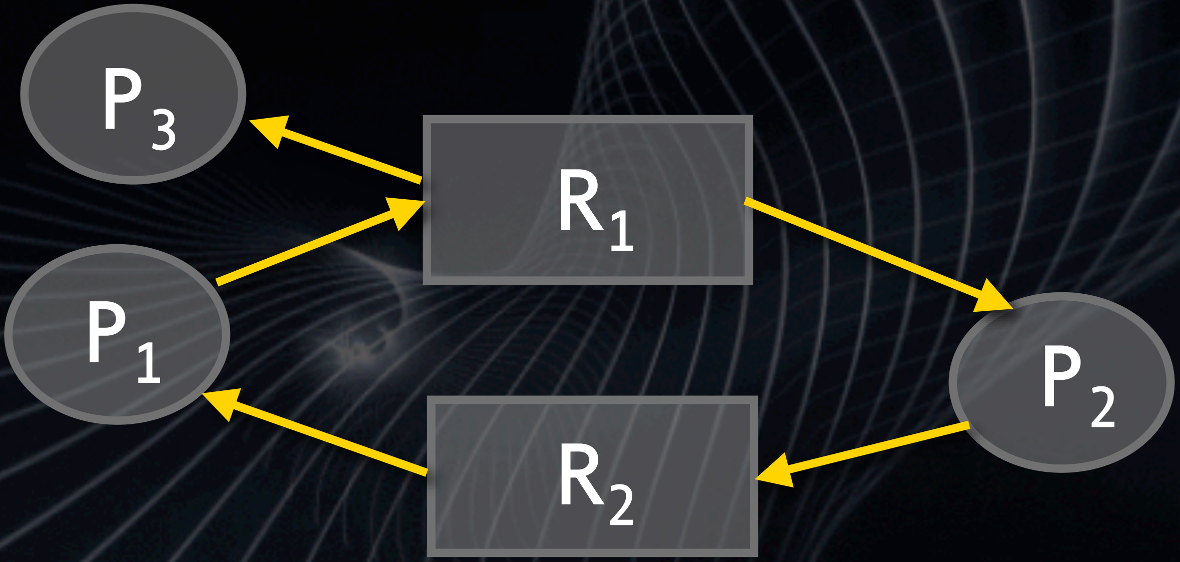


If each resource type has exactly one instance, then a cycle implies that one deadlock has occurred

Each process involved in the cycle is deadlocked

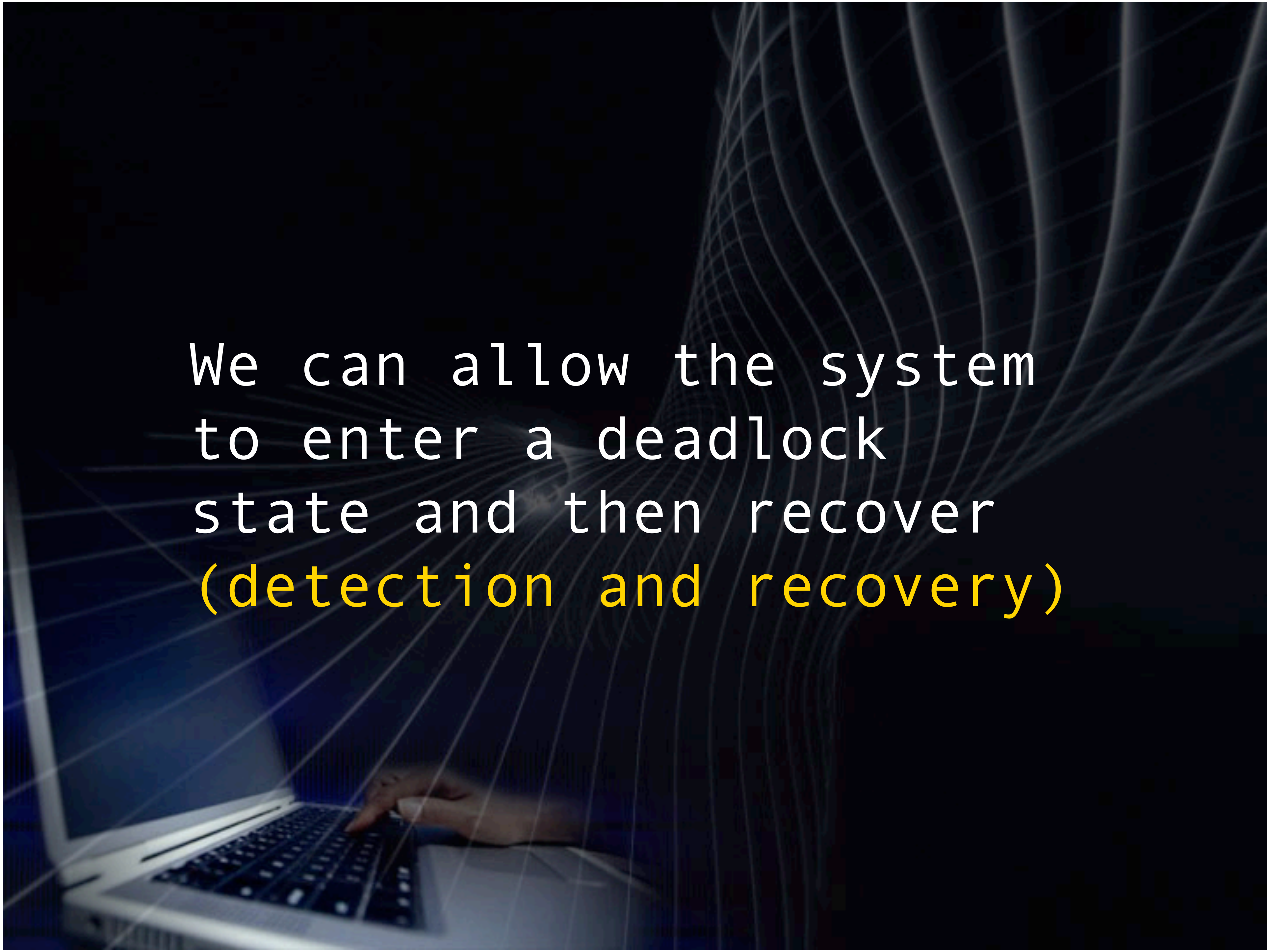
(a cycle in the graph is a necessary and sufficient condition for the existence of a deadlock)

If each resource type has several instances, then one cycle does not necessary imply that a deadlock occurred (the cycle is a necessary but not sufficient condition)


P_3
R_1
P_1
P_2
R_2

Methods for handling deadlock

We can use a protocol to ensure hat the system will never enter a deadlock state (deadlock prevention)



We can allow the system to enter a deadlock state and then recover (detection and recovery)

We can ignore the problem, and pretend that deadlocks never occur in the system

It is up to the application developer to write programs that handle deadlocks

Deadlock prevention

Deadlock prevention is a set of methods for ensuring that at least one of the necessary conditions can never occur

mutual exclusion

It is not possible to prevent deadlocks by denying the mutual exclusion condition

hold-and- wait

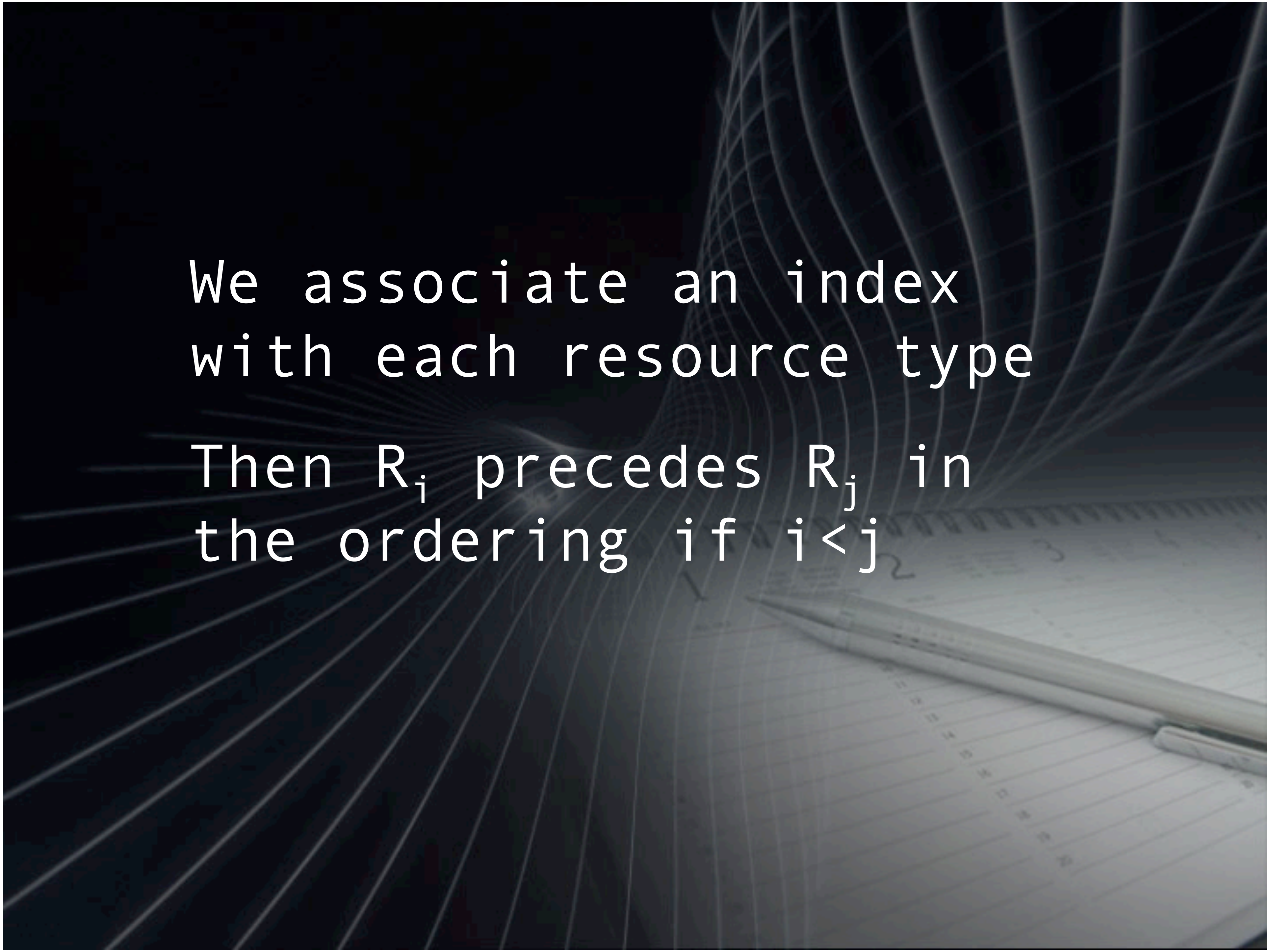
That condition may be prevented by requiring that each process must release all the resources currently allocated before it can request any additional resources.

no preemption

If a process that it is holding same resources request another resource that cannot be immediately allocated to it, then all resources currently being held are preempted

circular wait

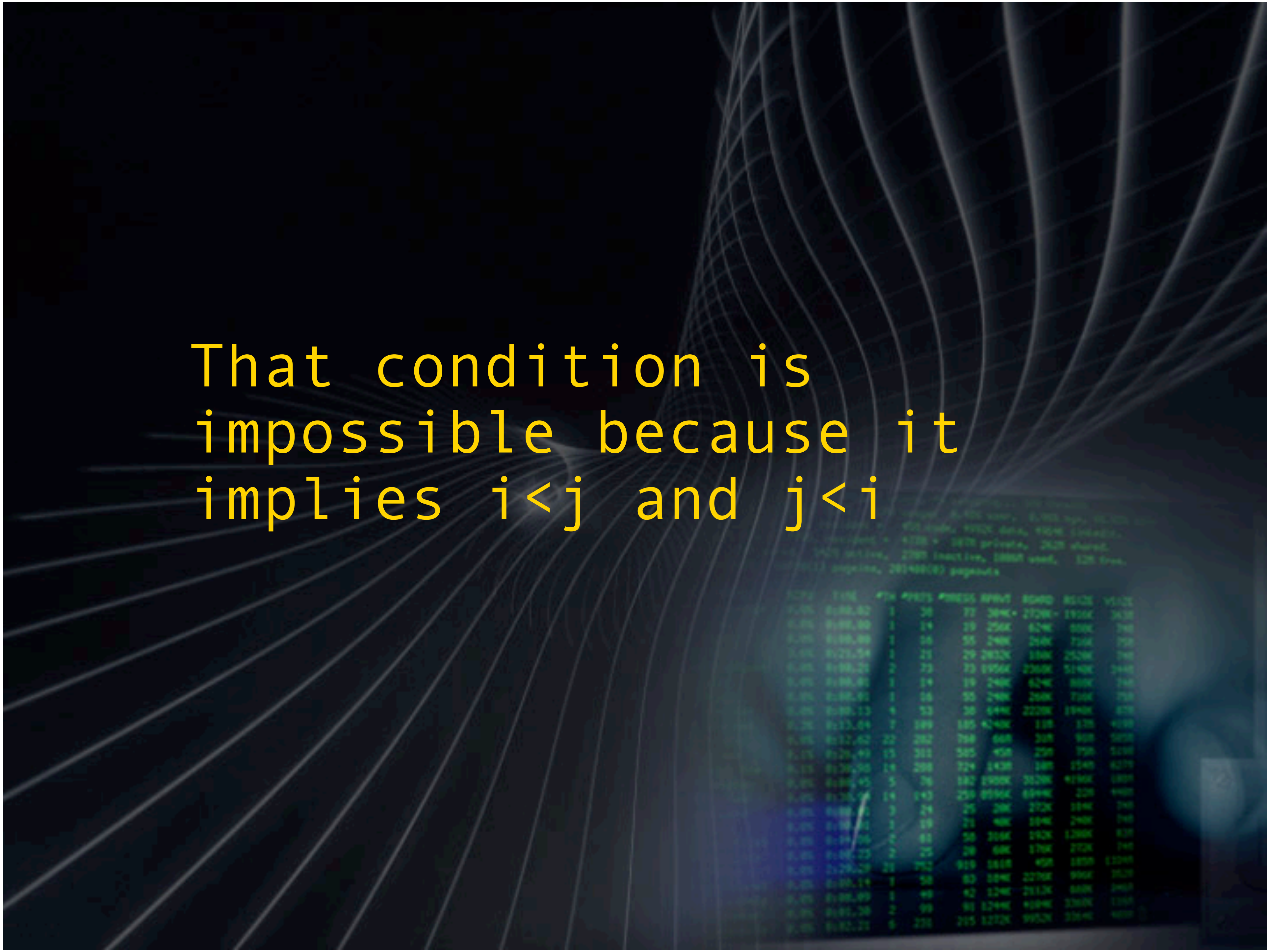
The condition can be prevented by defining a total ordering of all resource types and by requiring that each process requests resources in an increasing order



We associate an index with each resource type

Then R_i precedes R_j in the ordering if $i<j$

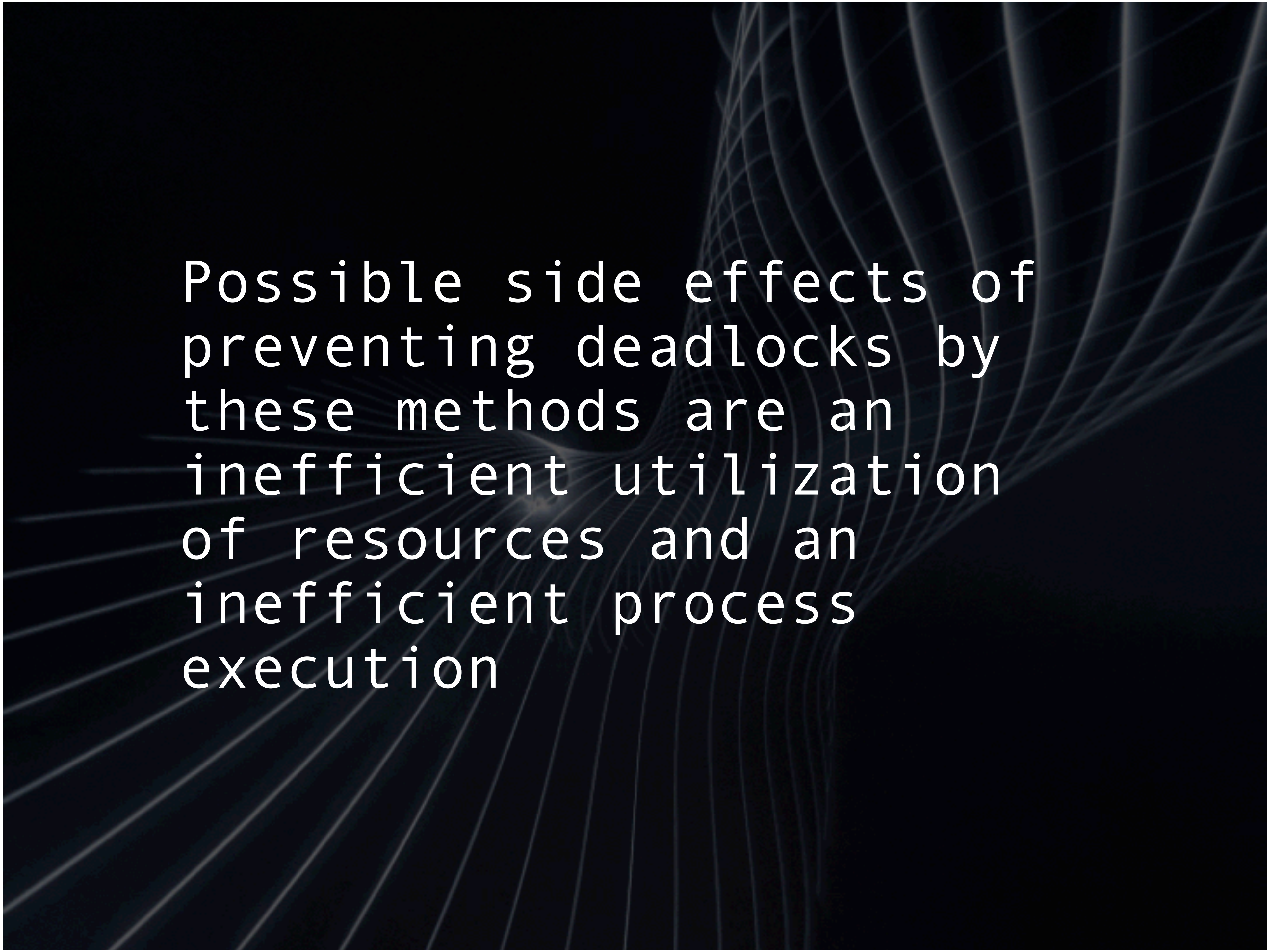
Two processes A and B, are deadlocked if A has acquired R_i and requests R_j, and B has acquired R_j and requests R_i



That condition is impossible because it implies $i<j$ and $j<i$

Deadlock avoidance

Deadlock-prevention algorithms prevent deadlocks by constraining the strategy on how requests can be made



Possible side effects of preventing deadlocks by these methods are an inefficient utilization of resources and an inefficient process execution

With deadlock avoidance, a decision is made dynamically whether current resource allocation requests, if granted, would potentially lead to deadlock

The resource allocation state is defined by the number of allocated and available resources and the maximum demands of processes

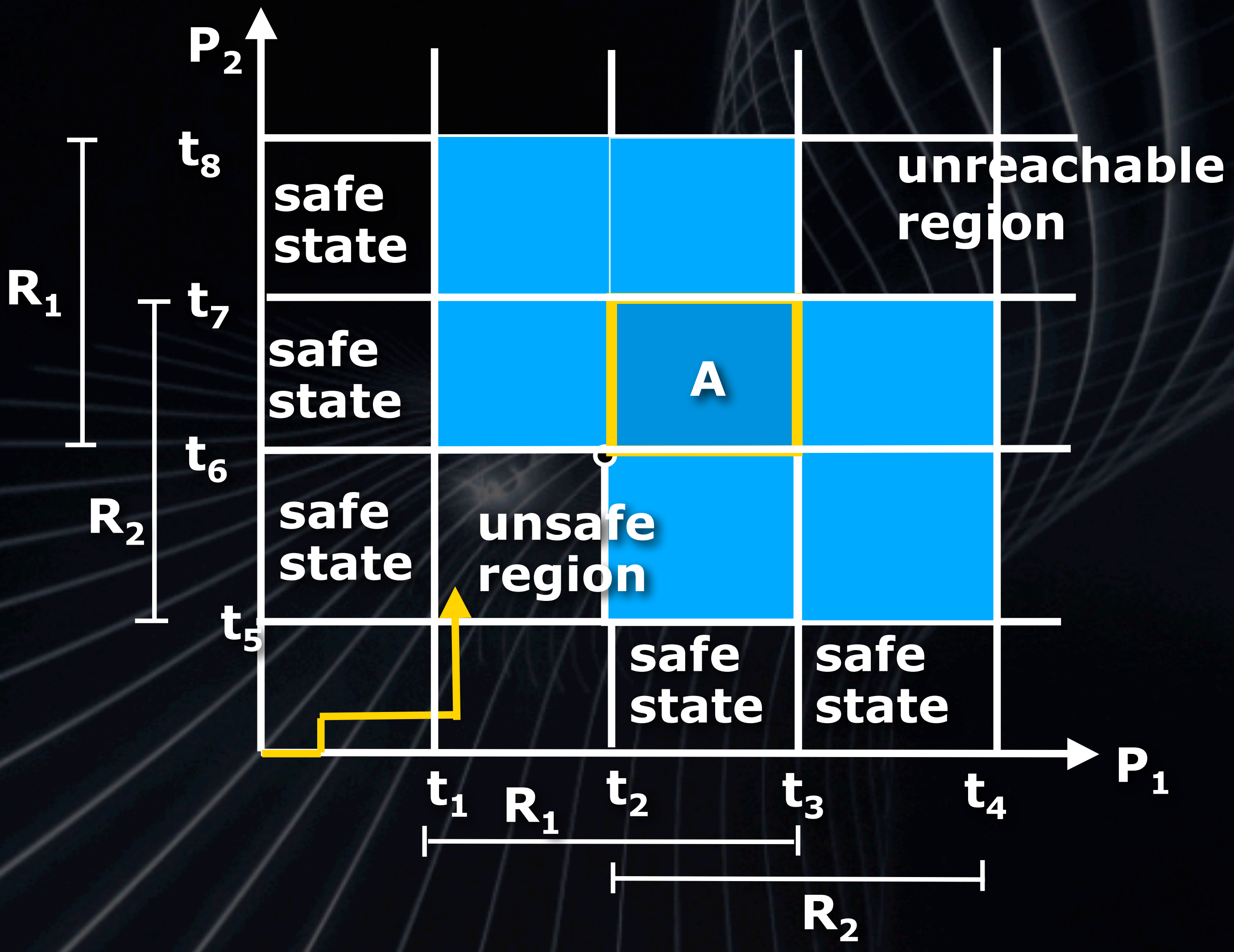
A **safe state** is one in which there is at least one process execution sequence such that all processes can be run to completion (**safe sequence**)

Banker's algorithm

When a process makes a request for a set of resources

assume that the request is granted, update the system state accordingly, and then determine if the result is still a safe state.

If so, grant the request,
if not, block the process
until it is safe
to grant the request



P2
P1
t8
t7
t6
t5
t1
t2
t3
t4
R1
R2
R1
R2
safe state
safe state
safe state
safe state
safe state
unreachable region
unsafe region
A

Deadlock detection

It requires:

an algorithm that examines the state of the system to determine whether a deadlock has occurred

an algorithm to recover from the deadlock

Recovery

Possible approaches:

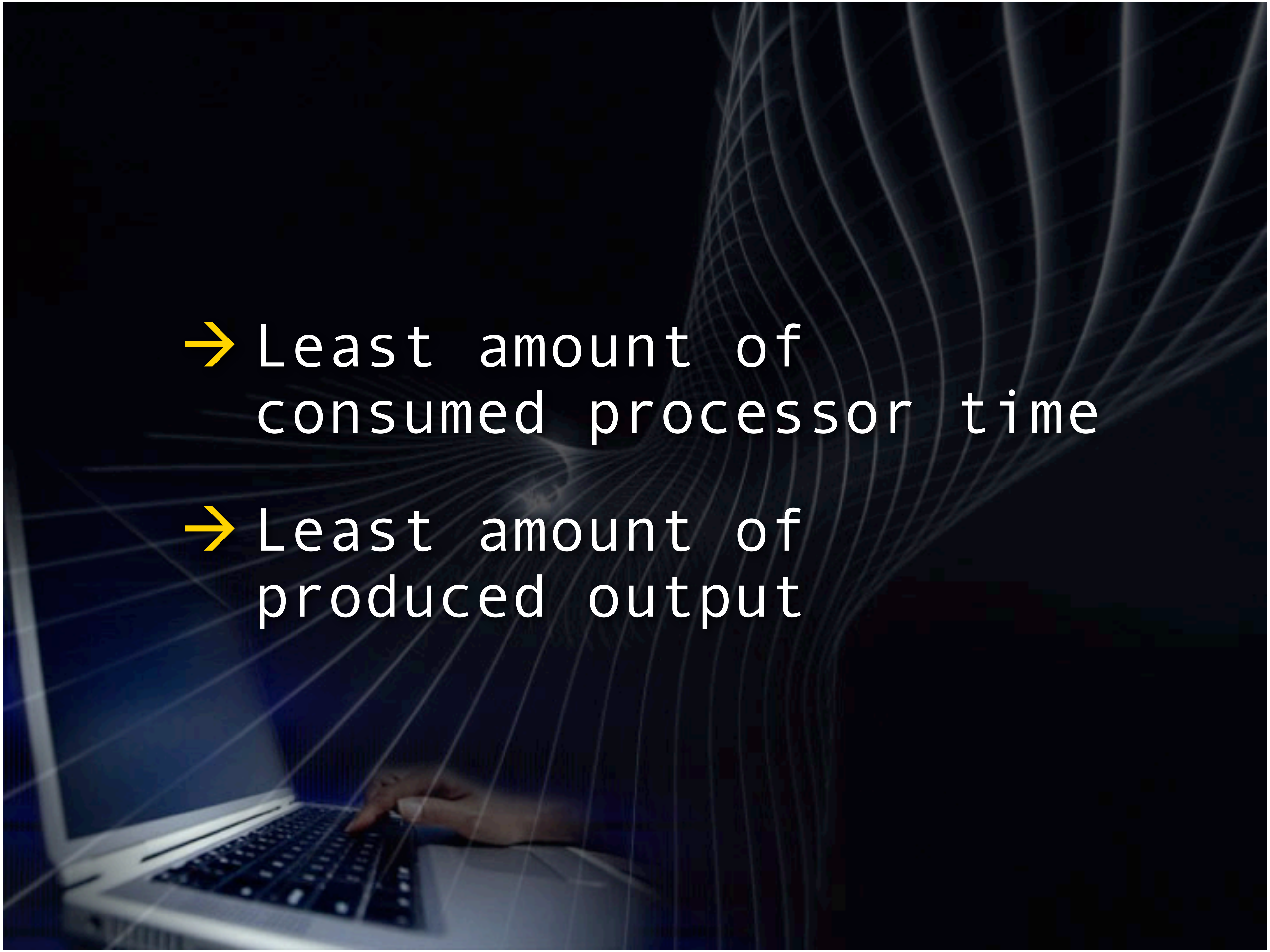
- Abort all deadlocked processes

- Back up each deadlocked process to some previously defined check points, and restart all processes form those checkpoints

3. Successively abort deadlocked processes until deadlock not longer exists

4. Successively preempt resources until deadlock not longer exists

For 3 and 4 the selection criteria could be one of the following. Choose the process with the:



- → Least amount of consumed processor time
- → Least amount of produced output

- Most estimated remaining time
- Least total resources allocated so far
- Lowest priority