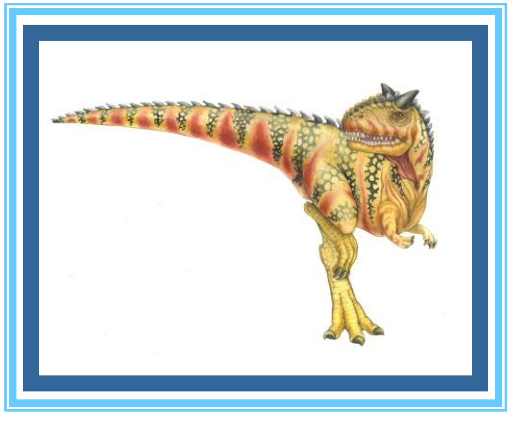
Threads



Day5: Threads and Deadlock

Multithreaded programming:

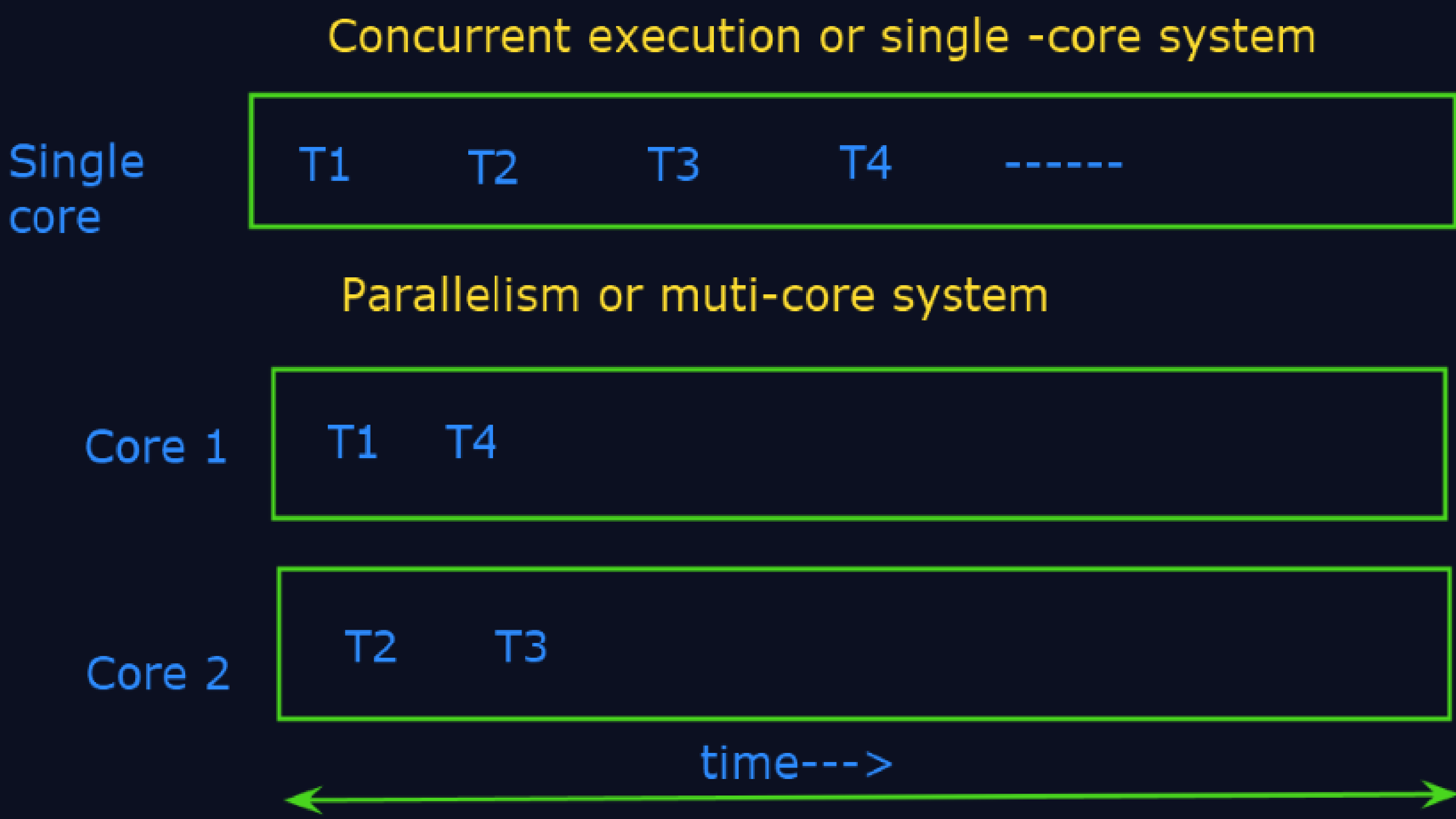
1.Responsiveness
2.Resource sharing
3.Economical
4.Utilization of multiprocessor architecture

Benefits

-Advantage ----> challenges

-Dividing activities
-Balancing
-Data splitting
-Data dependency
-Testing and debugging

1. Parallelism
2. Concurrency



Benefits

- **Responsiveness –** may allow continued execution if part of process is blocked, especially important for user interfaces
- **Resource Sharing –** threads share resources of process, easier than shared memory or message passing
- **Economy –** cheaper than process creation, thread switching lower overhead than context switching
- **Scalability –** process can take advantage of multiprocessor architectures

Multicore Programming

- **Multicore** or **multiprocessor** systems putting pressure on programmers, challenges include:
  - **Dividing activities**
  - **Balance**
  - **Data splitting**
  - **Data dependency**
  - **Testing and debugging**
- ***Parallelism*** implies a system can perform more than one task simultaneously
- ***Concurrency*** supports more than one task making progress
  - Single processor / core, scheduler providing concurrency

Multicore Programming (Cont.)

- Types of parallelism
  - **Data parallelism** – distributes subsets of the same data across multiple cores, same operation on each
  - **Task parallelism** – distributing threads across cores, each thread performing unique operation
- As # of threads grows, so does architectural support for threading
  - CPUs have cores as well as ***hardware threads***
  - Consider Oracle SPARC T4 with 8 cores, and 8 hardware threads per core

Thread: It is a basic unit of CPU utilization.

Thread Id

Program counter

Register set

stack

Share:

----

code section
data section
resources
signals

single thread: traditional approach
multiple thread: more than one task at a time

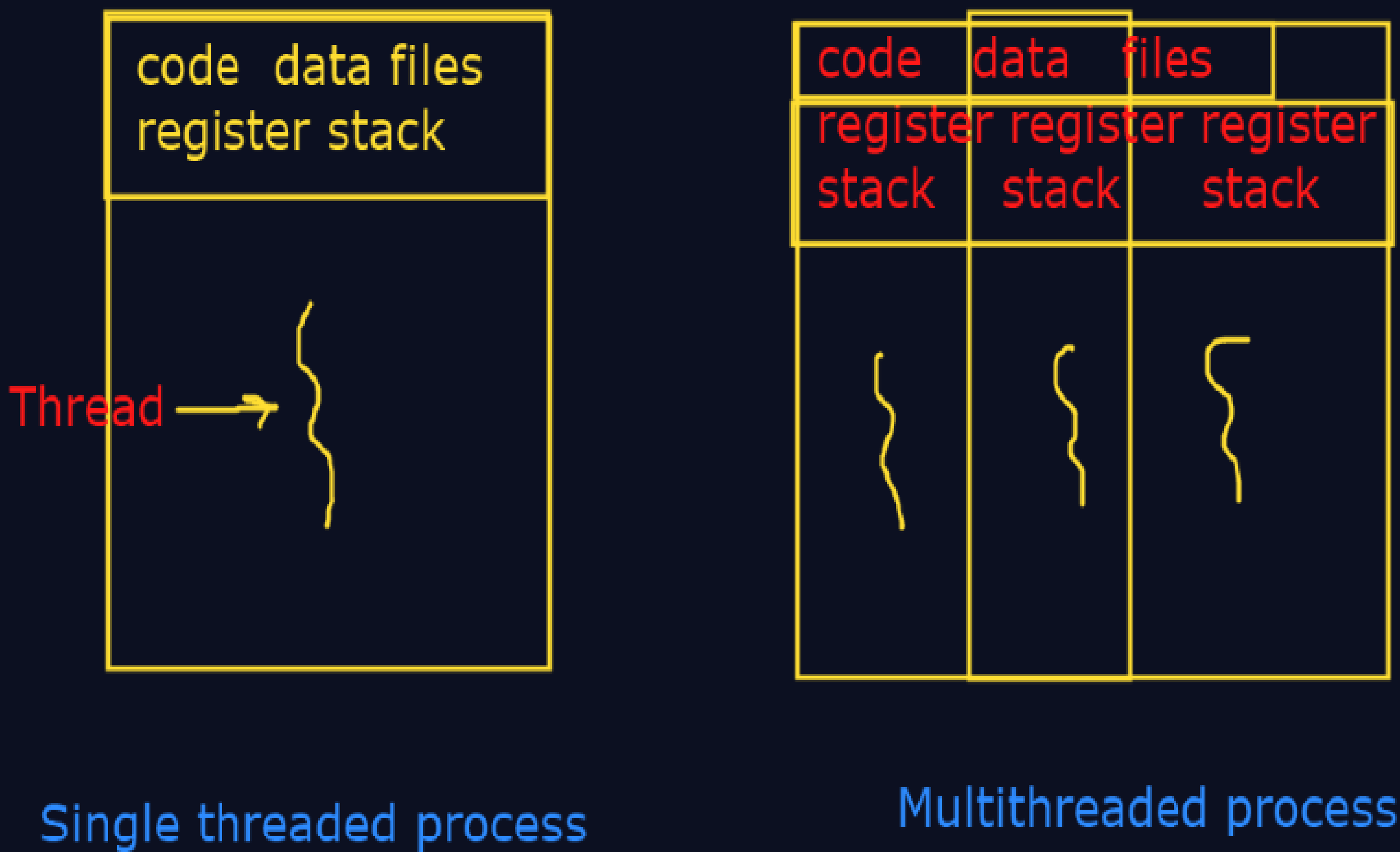
What is Thread?

- Thread is an **execution unit** that consists of its own program counter, a stack, and a set of registers where the program **counter mainly keeps track of which instruction to execute next**, a set of registers mainly hold its current working variables, and a stack mainly contains the history of execution.
- Threads are also known as **Lightweight processes.**
- Threads are a popular **way to improve the performance of an application** through parallelism.
- Threads are **mainly used to represent a software approach** in order to improve the performance of an operating system just by reducing the overhead thread that is mainly equivalent to a classical process.

Types of Thread

- There are two types of threads:
  - User Threads
  - Kernel Threads
- User threads are above the kernel and without kernel support. These are the **threads that application programmers use in their programs**.
- Kernel threads are **supported within the kernel of the OS itself**. All modern OSs support kernel-level threads, allowing the kernel to perform multiple simultaneous tasks and/or to service multiple kernel system calls simultaneously.

Thread: A thread is a basic unit of CPU utilization.



Amdahl's Law

- Identifies performance gains from adding additional cores to an application that has both serial and parallel components
- *S* is serial portion
- *N* processing cores

$$speedup \leq \frac{1}{S + \frac{(1-S)}{N}}$$

- That is, if application is 75% parallel / 25% serial, moving from 1 to 2 cores results in speedup of 1.6 times
- As *N* approaches infinity, speedup approaches 1 / *S*

Serial portion of an application has disproportionate effect on performance gained by adding additional cores

- But does the law take into account contemporary multicore systems?

Multithreading Models

- Many-to-One
- One-to-One
- Many-to-Many

2. Kernel thread

- -Windows
- -Linux
- -Mac os
- -Solaris

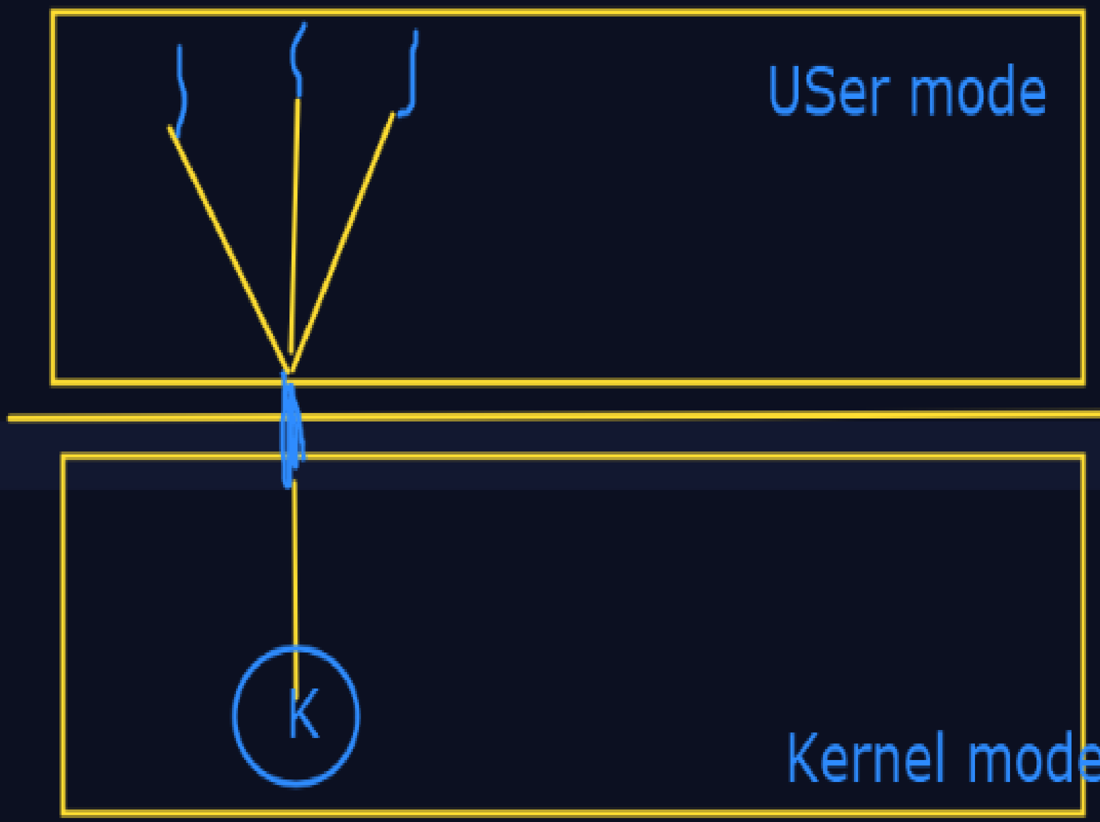
Amdahl's Law:

Thread Mapping:

Many to one

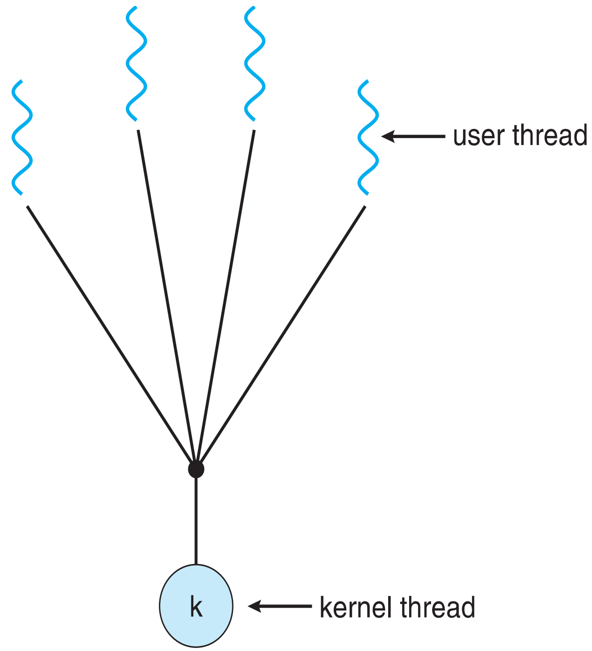
One to One

One to Many



Many-to-One

- Many user-level threads mapped to single kernel thread
- One thread blocking causes all to block
- Multiple threads may not run in parallel on muticore system because only one may be in kernel at a time
- Few systems currently use this model
- Examples:
  - **Solaris Green Threads**
  - **GNU Portable Threads**



2. Kernel thread

- -Windows
- -Linux
- -Mac os
- -Solaris

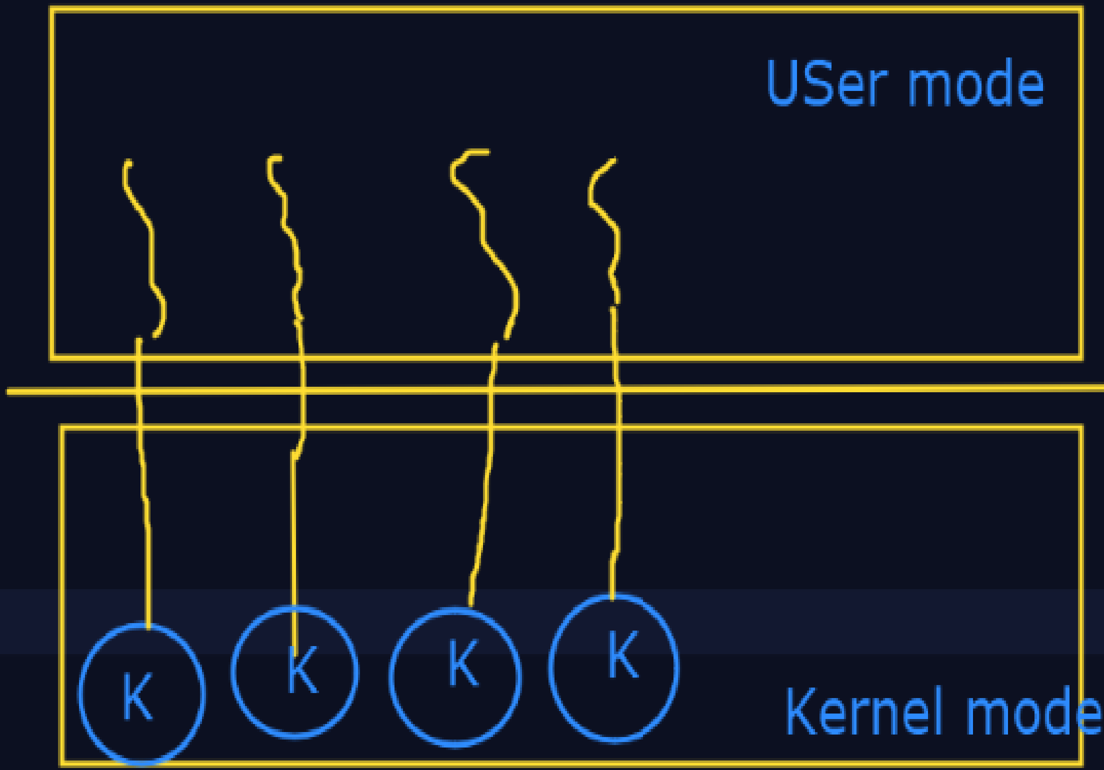
Amdahl's Law:

Thread Mapping:

Many to one

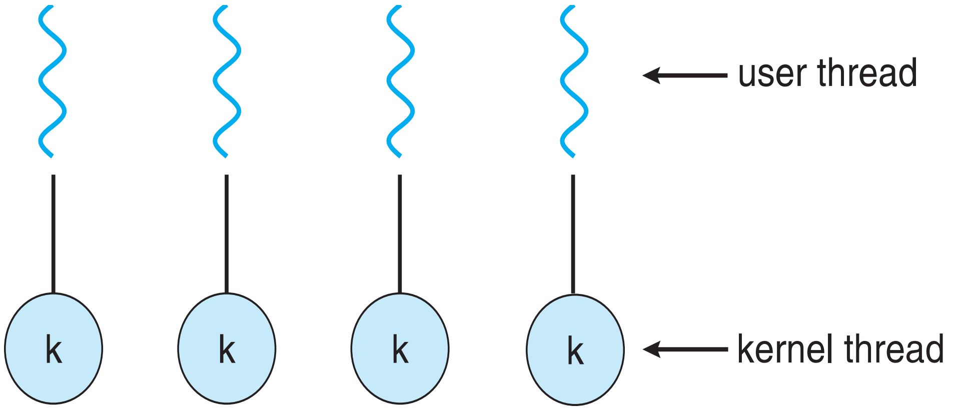
One to One ✓

One to Many



One-to-One

- Each user-level thread maps to kernel thread
- Creating a user-level thread creates a kernel thread
- More concurrency than many-to-one
- Number of threads per process sometimes restricted due to overhead
- Examples
  - Windows
  - Linux
  - Solaris 9 and later

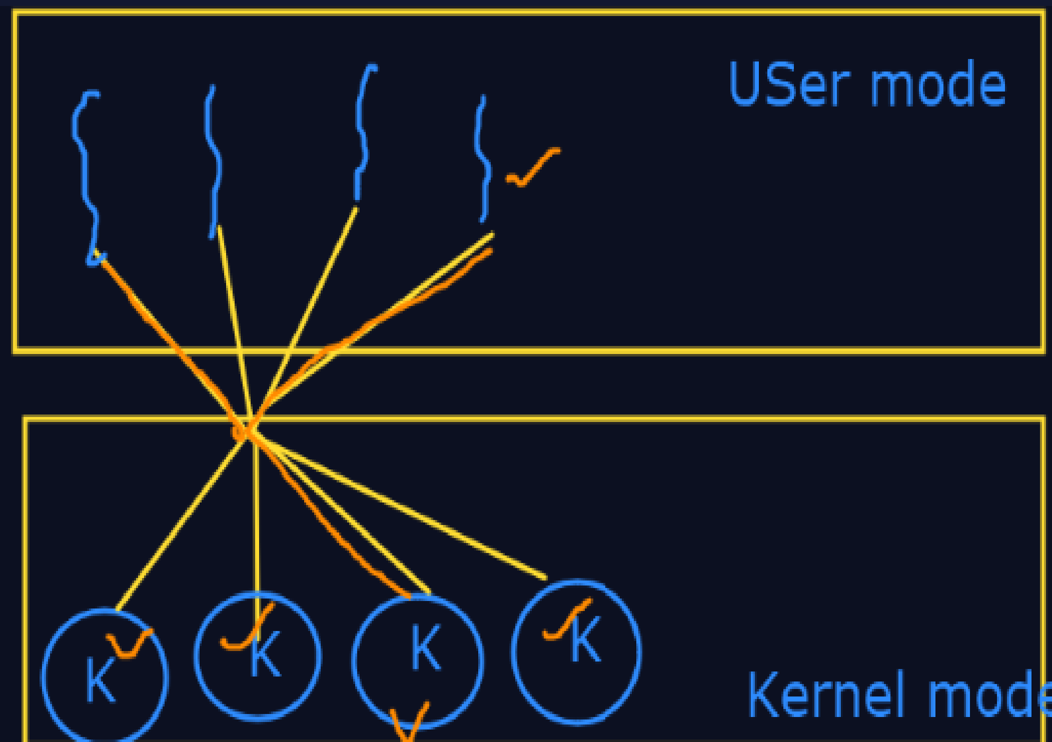


Types of thread:
----

1. User thread
    -Java threads
    -POSIX threads pThread :API
    -Windows threads

2. Kernel thread
    -Windows
    -Linux
    -Mac os
    -Solaris

Amdahl's Law:

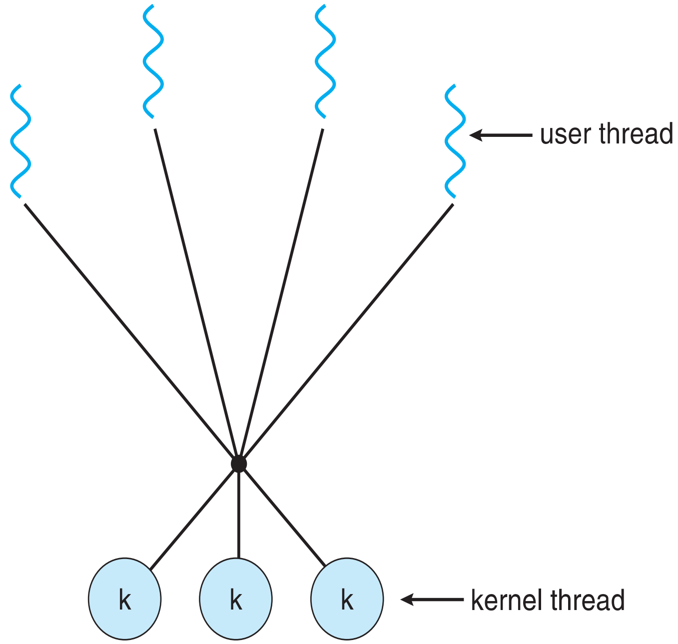


Thread Mapping:

Many to one
One to One
Many to Many

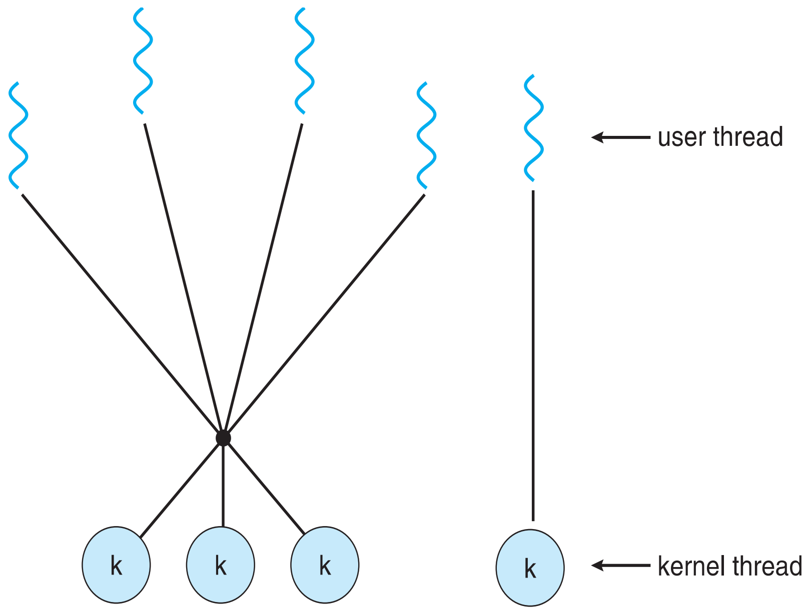
Many-to-Many Model

- Allows many user level threads to be mapped to many kernel threads
- Allows the operating system to create a sufficient number of kernel threads
- Solaris prior to version 9
- Windows with the *ThreadFiber* package

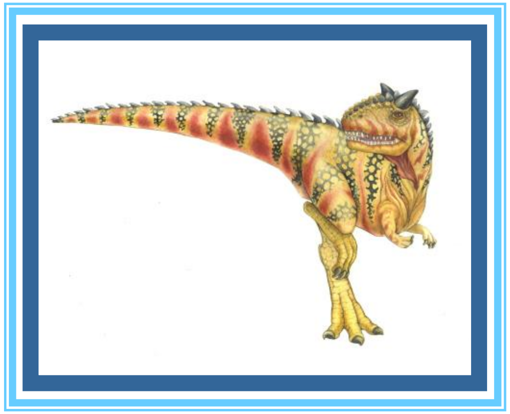


Two-level Model

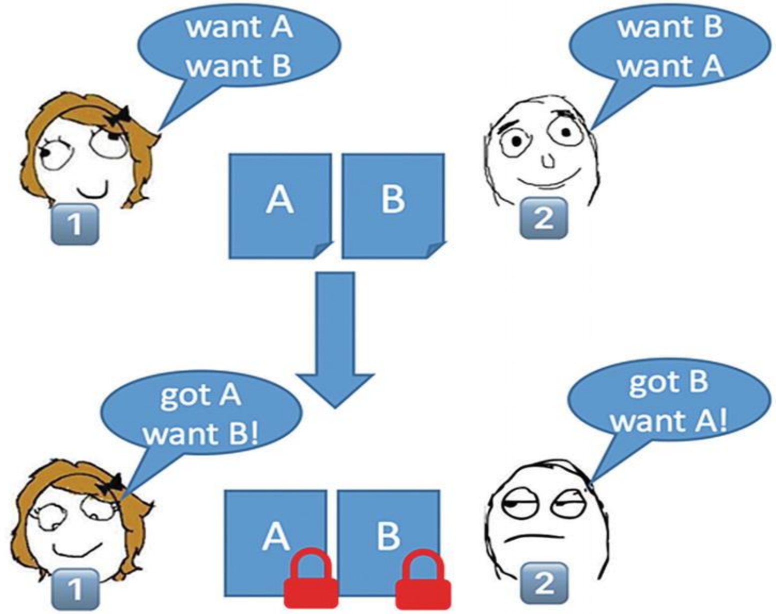
- Similar to M:M, except that it allows a user thread to be **bound** to kernel thread
- Examples
  - IRIX
  - HP-UX
  - Tru64 UNIX
  - Solaris 8 and earlier



Chapter 7: Deadlocks



(a) Deadlock in real life



(b) Deadlock in virtual life

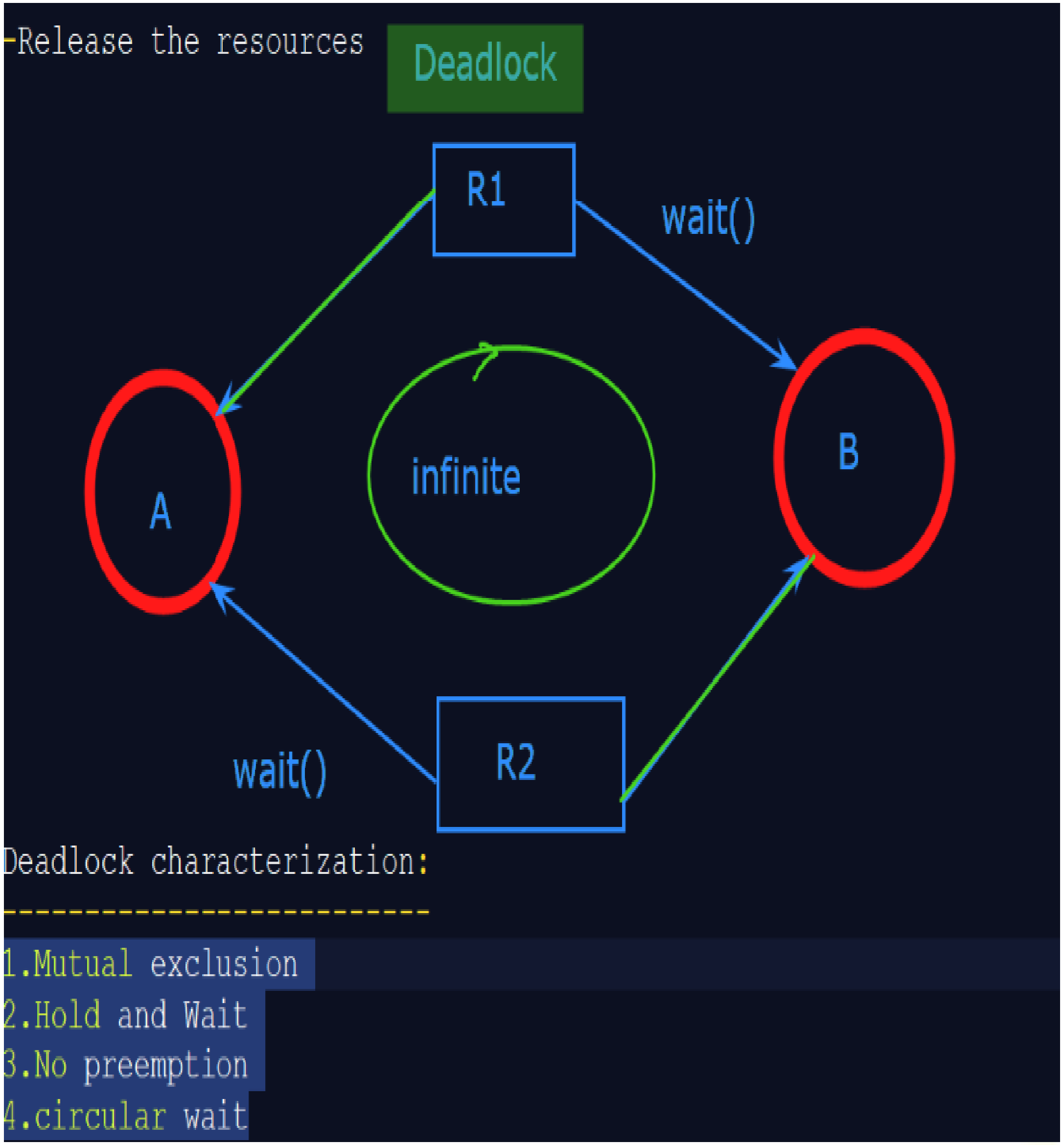
Who can see what you share here? Recording On

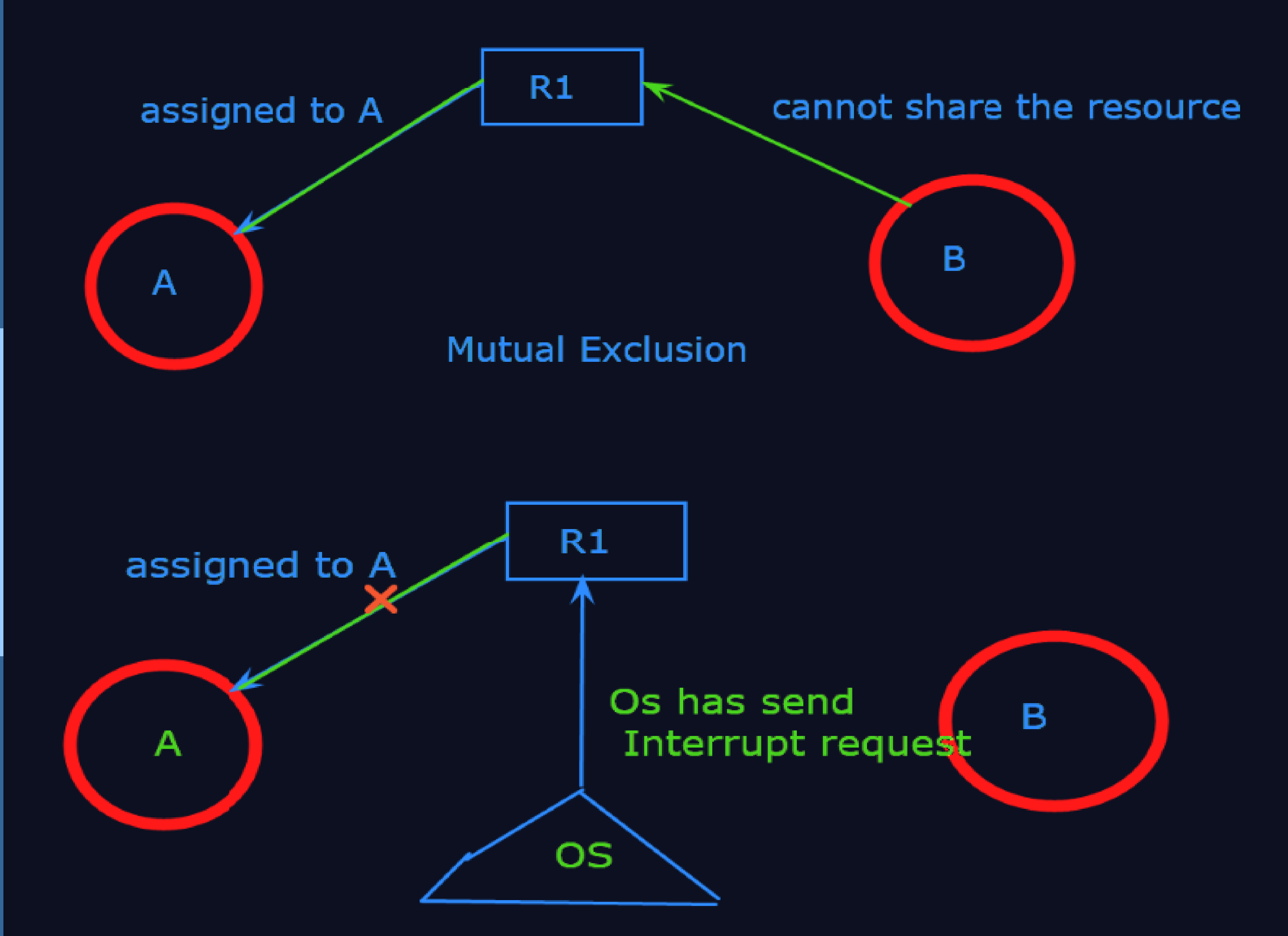
```
Deadlock:
--------------

Request a resource --->Use the resource ------>Release the resource
```

System Model

- 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.
- Each process utilizes a resource as follows:
  - **request**
  - **use**
  - **release**




R1
assigned to A
cannot share the resource
A
B
Mutual Exclusion
R1
assigned to A
A
Os has send
Interrupt request
B
OS

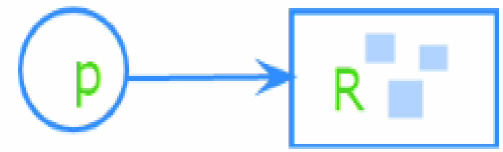
Deadlock Characterization

Deadlock can arise if four conditions hold simultaneously.

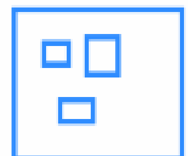
- **Mutual exclusion:** only one process at a time can use a resource
- **Hold and wait:** a process holding at least one resource is waiting to acquire additional resources held by other processes
- **No preemption:** a resource can be released only voluntarily by the process holding it, after that process has completed its task
- **Circular wait:** there exists a set $\{P_0, P_1, \ldots, P_n\}$ of waiting processes such that P_0 is waiting for a resource that is held by P_1, P_1 is waiting for a resource that is held by P_2, …, P_{n-1} is waiting for a resource that is held by P_n, and P_n is waiting for a resource that is held by P_0.

Resource-Allocation Graph

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



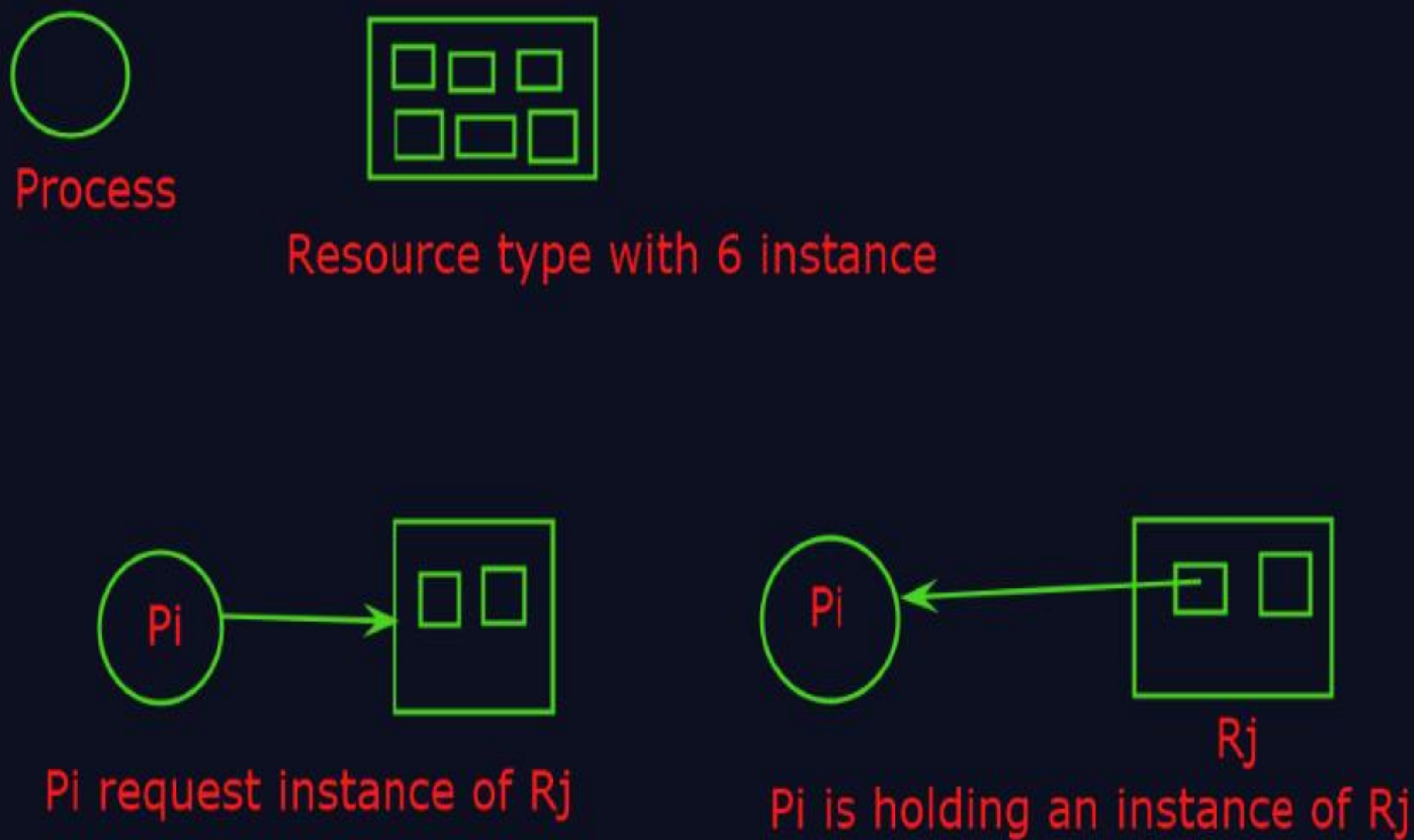
- V is partitioned into two types:
  - $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system



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

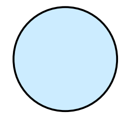
- **request edge** – directed edge $P_i \to R_j$
- **assignment edge** – directed edge $R_j \to P_i$

Resource allocation graph:

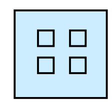


Resource-Allocation Graph (Cont.)

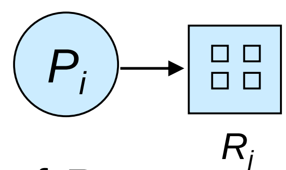
- Process



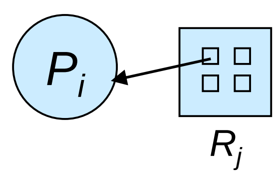
- Resource Type with 4 instances



- P_i requests instance of R_j

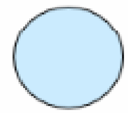


- P_i is holding an instance of R_j

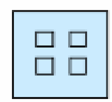


Resource-Allocation Graph (Cont.)

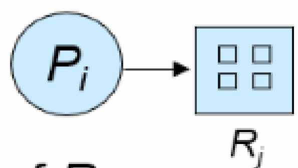
- Process



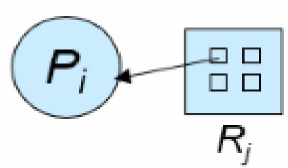
- Resource Type with 4 instances



- P_i requests instance of R_j

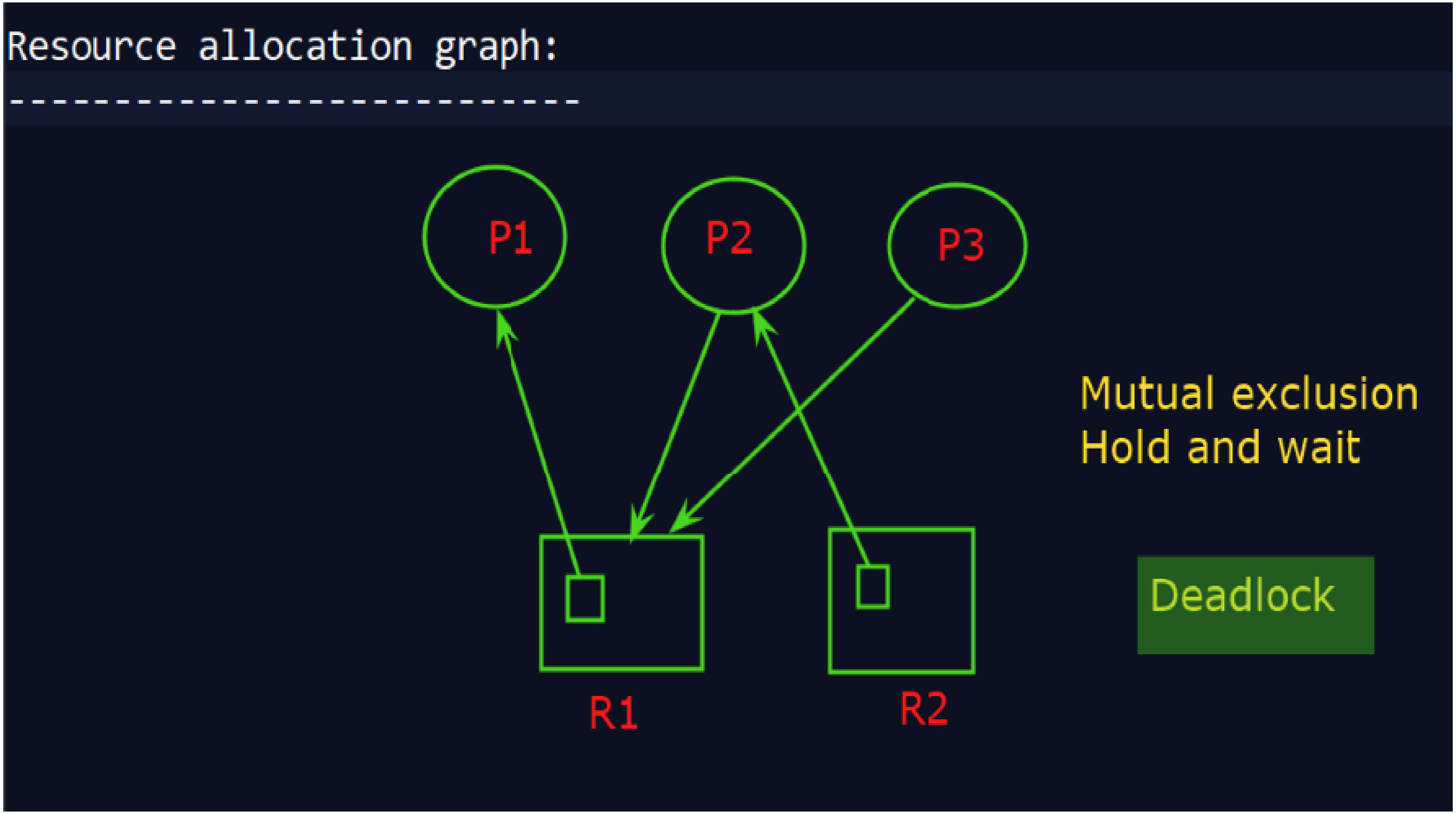


- P_i is holding an instance of R_j



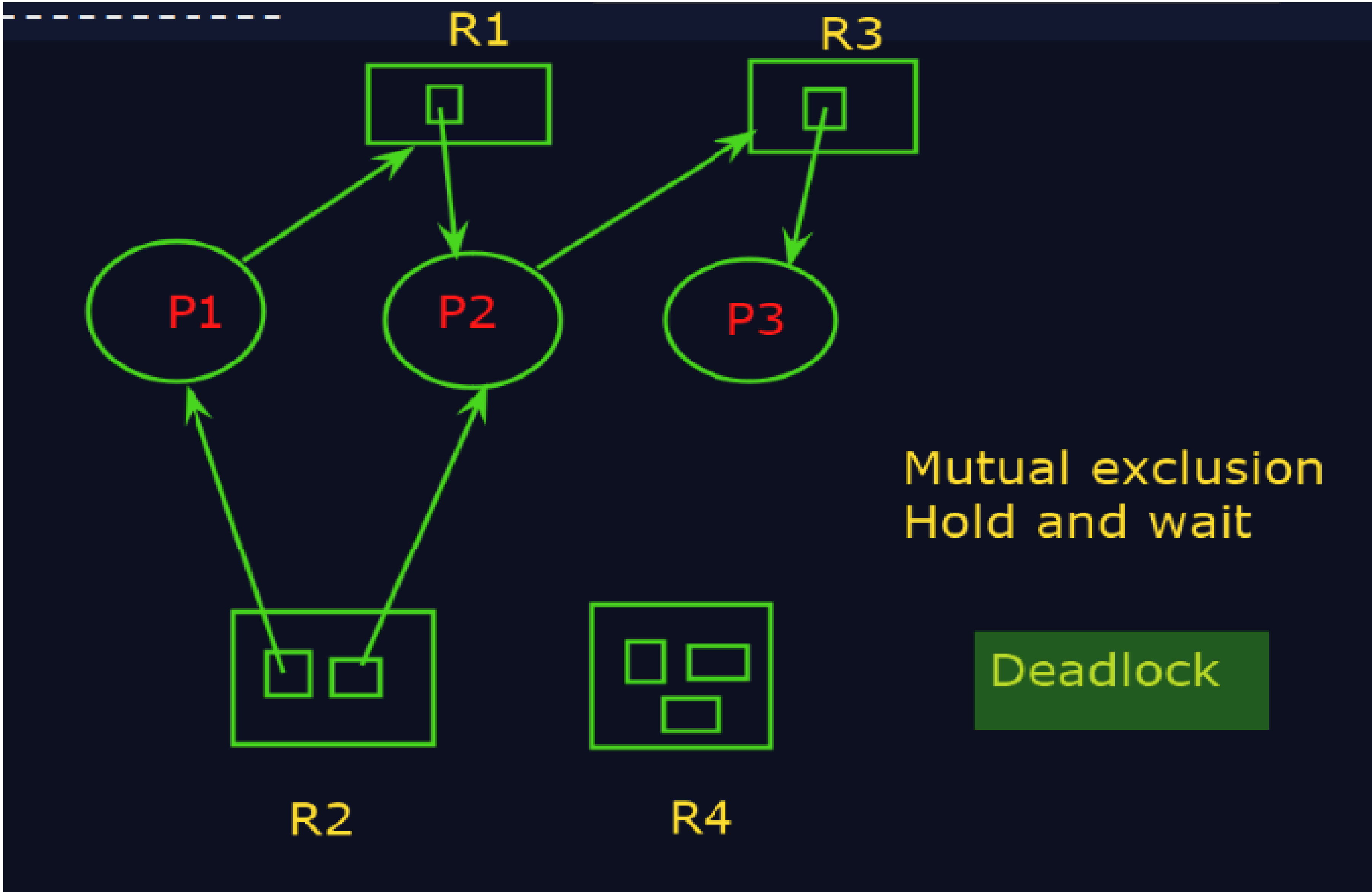
P1	3	4
P2	2	1
P3	1	4
P4	4	3

Example of a Resource Allocation Graph



Basic Facts

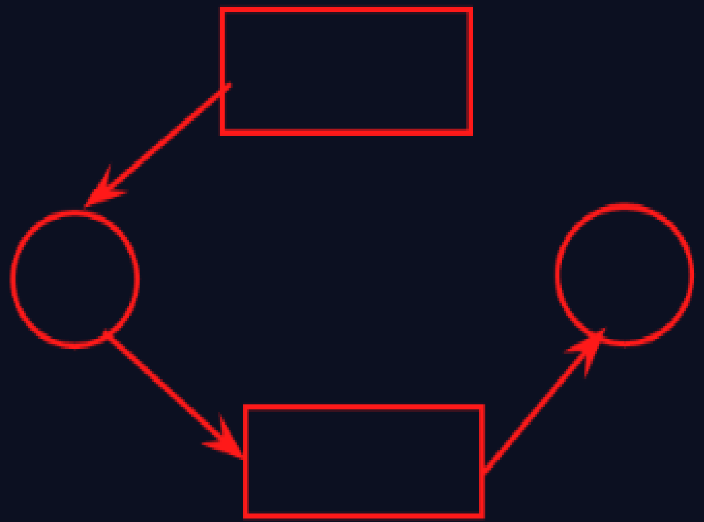
- If graph contains no cycles $\Rightarrow$ no deadlock
- If graph contains a cycle $\Rightarrow$
  - if only one instance per resource type, then deadlock
  - if several instances per resource type, possibility of deadlock


R1
R3
P1
P2
P3
Mutual exclusion
Hold and wait
R2
R4
Deadlock

Deadlock Handling

----

1. Pre-emption
2. Roll back
3. Kill the process



Graph allocation method:

Bankers Algorithm:Deadlock avaoidance

----

1. no of process to be excute, resource available
2. duration for resources
3. available resources

Deadlock prevention strategies:

-Mutual exclusion

-hold & wait

Methods for Handling Deadlocks

- Ensure that the system will ***never*** enter a deadlock state:
  - Deadlock prevention
  - Deadlock avoidence
- Allow the system to enter a deadlock state and then recover
- Ignore the problem and pretend that deadlocks never occur in the system; used by most operating systems, including UNIX

Deadlock Prevention

Restrain the ways request can be made

- **Mutual Exclusion** – not required for sharable resources (e.g., read-only files); must hold for non-sharable resources
- **Hold and Wait** – must guarantee that whenever a process requests a resource, it does not hold any other resources
  - Require process to request and be allocated all its resources before it begins execution, or allow process to request resources only when the process has none allocated to it.
  - Low resource utilization; starvation possible

Deadlock Prevention (Cont.)

- **No Preemption** –
  - If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released
  - Preempted resources are added to the list of resources for which the process is waiting
  - Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting
- **Circular Wait** – impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration

Deadlock Avoidance

Requires that the system has some additional ***a priori*** information available

- Simplest and most useful model requires that each process declare the ***maximum number*** of resources of each type that it may need
- The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition
- Resource-allocation *state* is defined by the number of available and allocated resources, and the maximum demands of the processes

Safe State

- When a process requests an available resource, system must decide if immediate allocation leaves the system in a safe state
- System is in **safe state** if there exists a sequence $<P_1, P_2, \ldots, P_n>$ of ALL the processes in the systems such that for each P_i, the resources that P_i can still request can be satisfied by currently available resources + resources held by all the P_j, with $j < I$
- That is:
  - If P_i resource needs are not immediately available, then P_i can wait until all P_j have finished
  - When P_j is finished, P_i can obtain needed resources, execute, return allocated resources, and terminate
  - When P_i terminates, P_{i+1} can obtain its needed resources, and so on

Basic Facts

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

Avoidance Algorithms

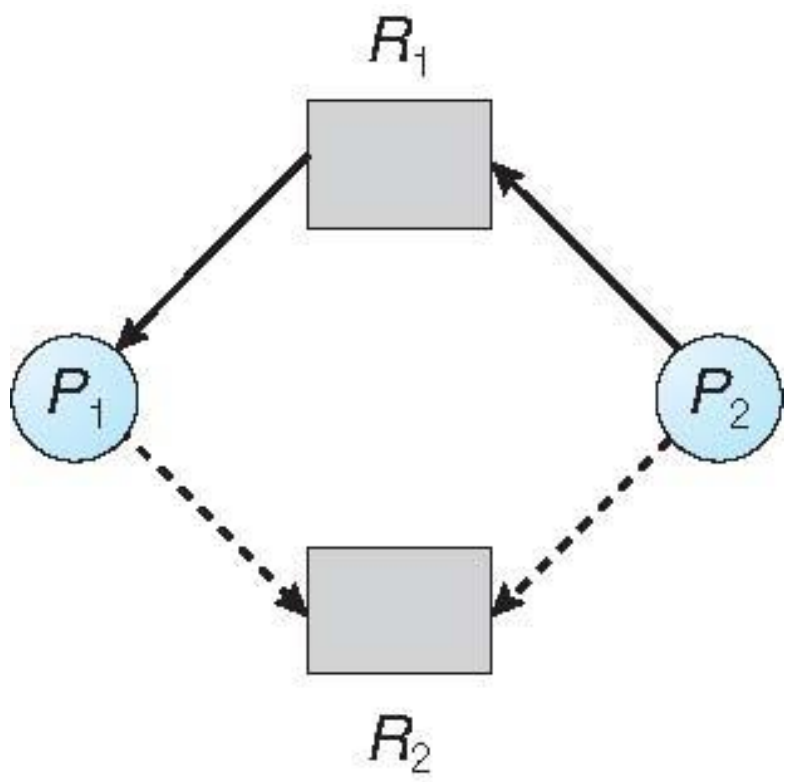
- Single instance of a resource type
  - Use a resource-allocation graph

- Multiple instances of a resource type
  - Use the banker's algorithm

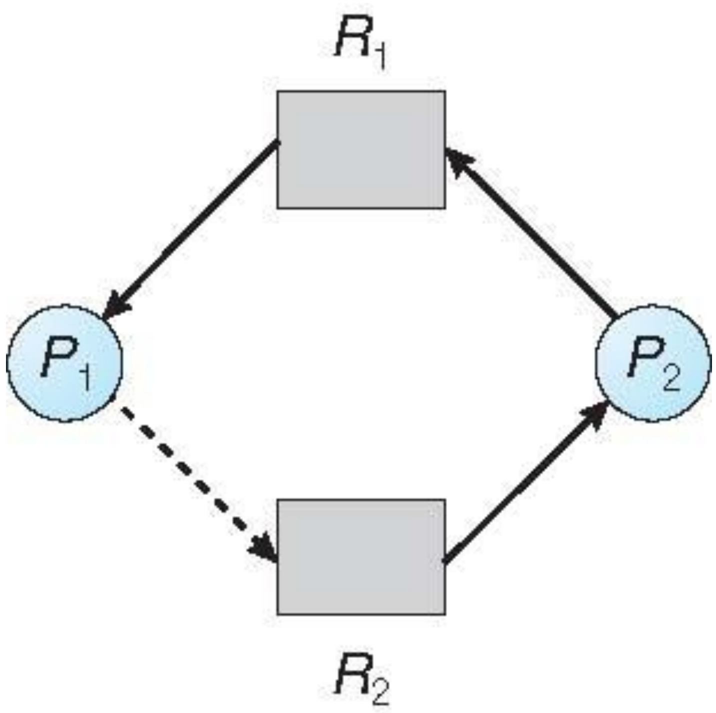
Resource-Allocation Graph Scheme

- **Claim edge** $P_i \rightarrow R_j$ indicated that process P_j may request resource R_j; represented by a dashed line
- Claim edge converts to request edge when a process requests a resource
- Request edge converted to an assignment edge when the resource is allocated to the process
- When a resource is released by a process, assignment edge reconverts to a claim edge
- Resources must be claimed *a priori* in the system

Resource-Allocation Graph



Unsafe State In Resource-Allocation Graph



Resource-Allocation Graph Algorithm

- Suppose that process P_i requests a resource R_j
- The request can be granted only if converting the request edge to an assignment edge does not result in the formation of a cycle in the resource allocation graph

Banker's Algorithm

- Multiple instances
- Each process must a priori claim maximum use
- When a process requests a resource it may have to wait
- When a process gets all its resources it must return them in a finite amount of time

Data Structures for the Banker's Algorithm

Let n = number of processes, and m = number of resources types.

- **Available***:* Vector of length m. If available [j] = k, there are k instances of resource type R_j available
- **Max***:* n x m matrix. If *Max* [i,j] = k, then process P_i may request at most k instances of resource type R_j
- **Allocation***:* n x m matrix. If Allocation[i,j] = k then P_i is currently allocated k instances of R_j
- **Need***:* n x m matrix. If *Need*[i,j] = k, then P_i may need k more instances of R_j to complete its task

$$Need\ [i,j] = Max[i,j] - Allocation\ [i,j]$$