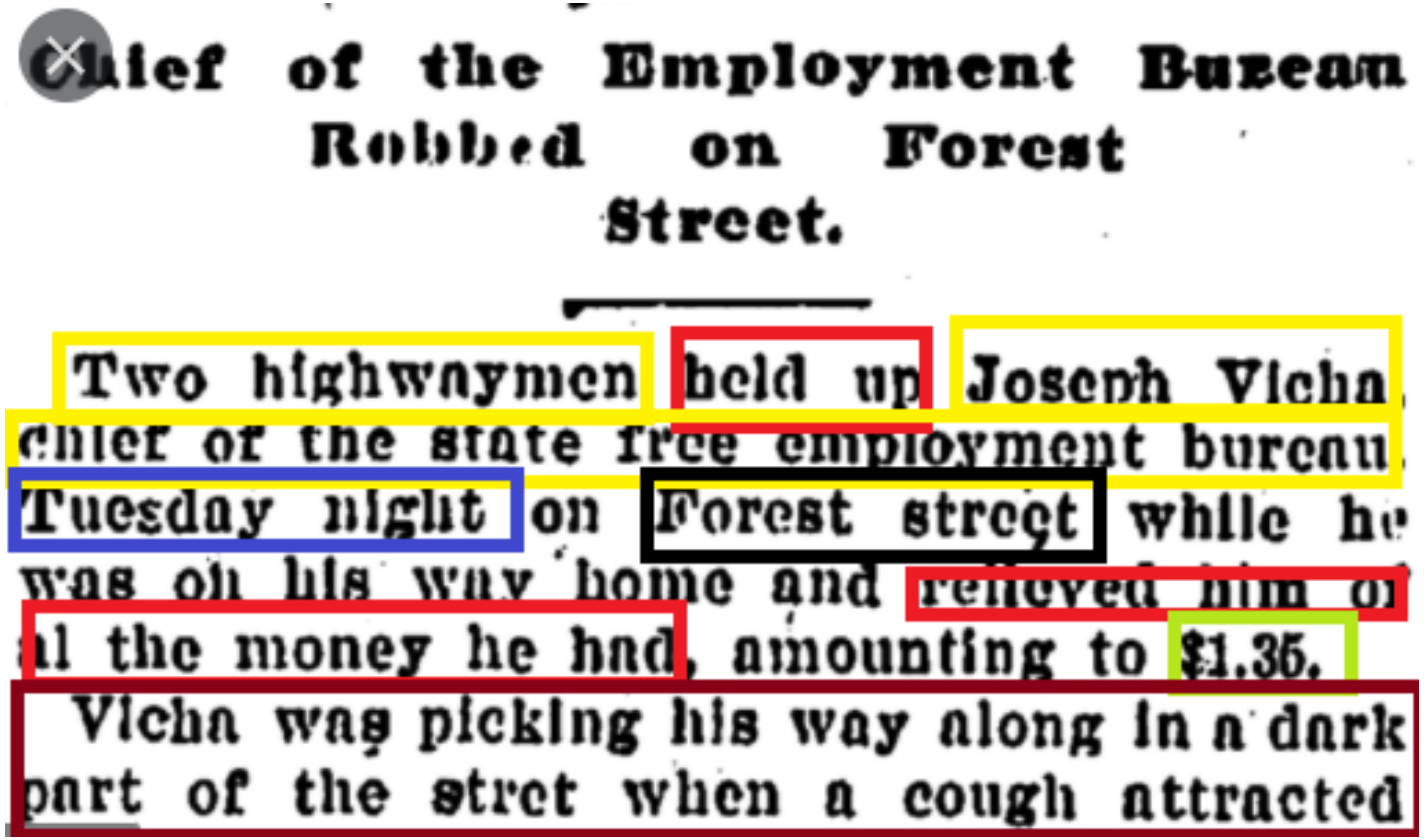
CPSC 5042: Week 4

Process Synchronization

Last Week

- Wrapped up Memory Management
- Look beyond Contiguous
- Paged Memory Allocation
- Demand Paged Memory Allocation
    - Algorithms for page removal: FIFO, LRU…
- Segmented Memory Allocation
- Paging & Segmentation Combined

WHO
WHY
WHERE
WHAT
HOW
WHEN



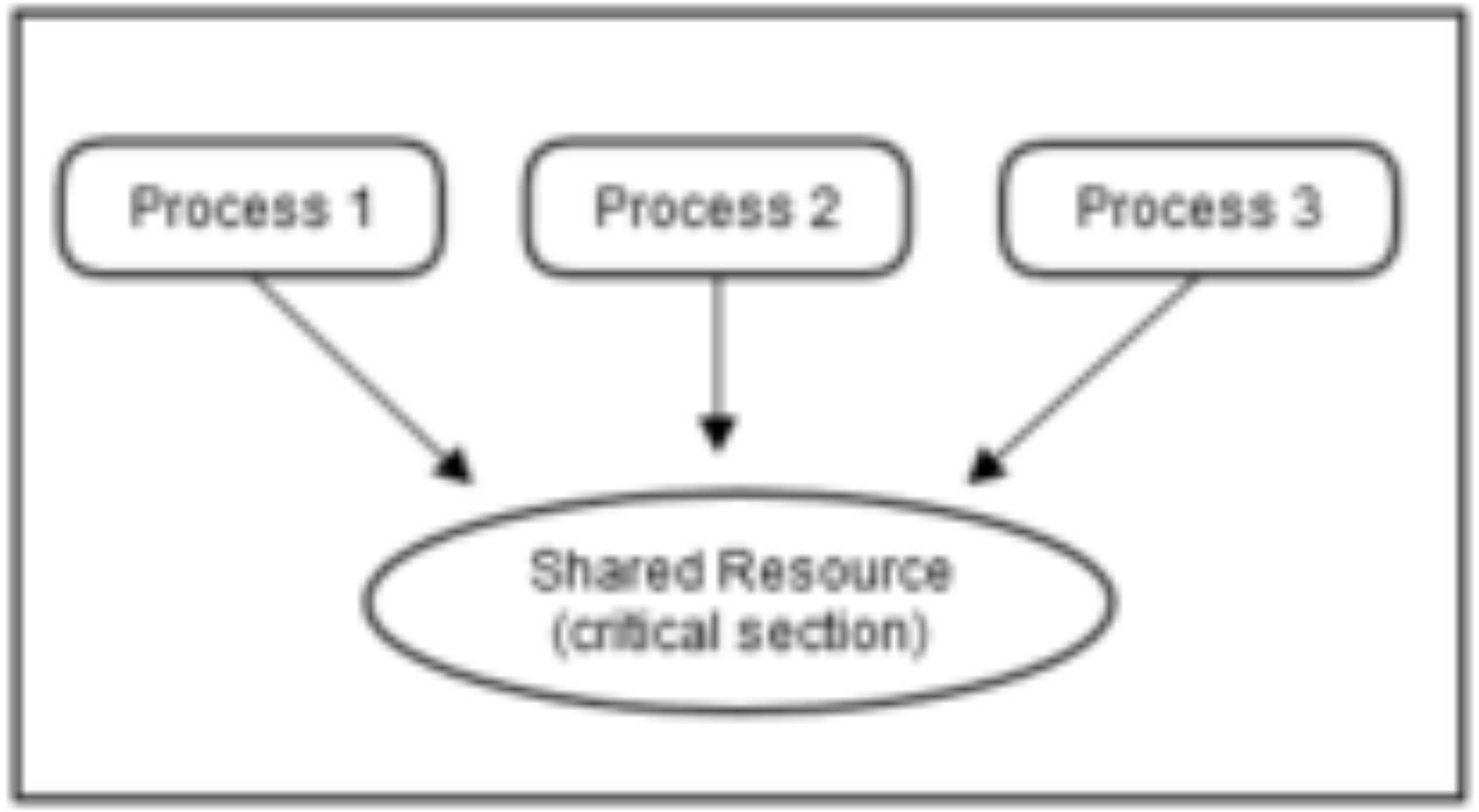
Chief of the Employment Bureau Robbed on Forest Street.

Two highwaymen held up Joseph Vicha, chief of the state free employment bureau, Tuesday night on Forest street while he was on his way home and relieved him of all the money he had, amounting to $1.35.

Vicha was picking his way along in a dark part of the stret when a cough attracted

Process Synchronization

- Process Synchronization is a way to coordinate processes that use shared data



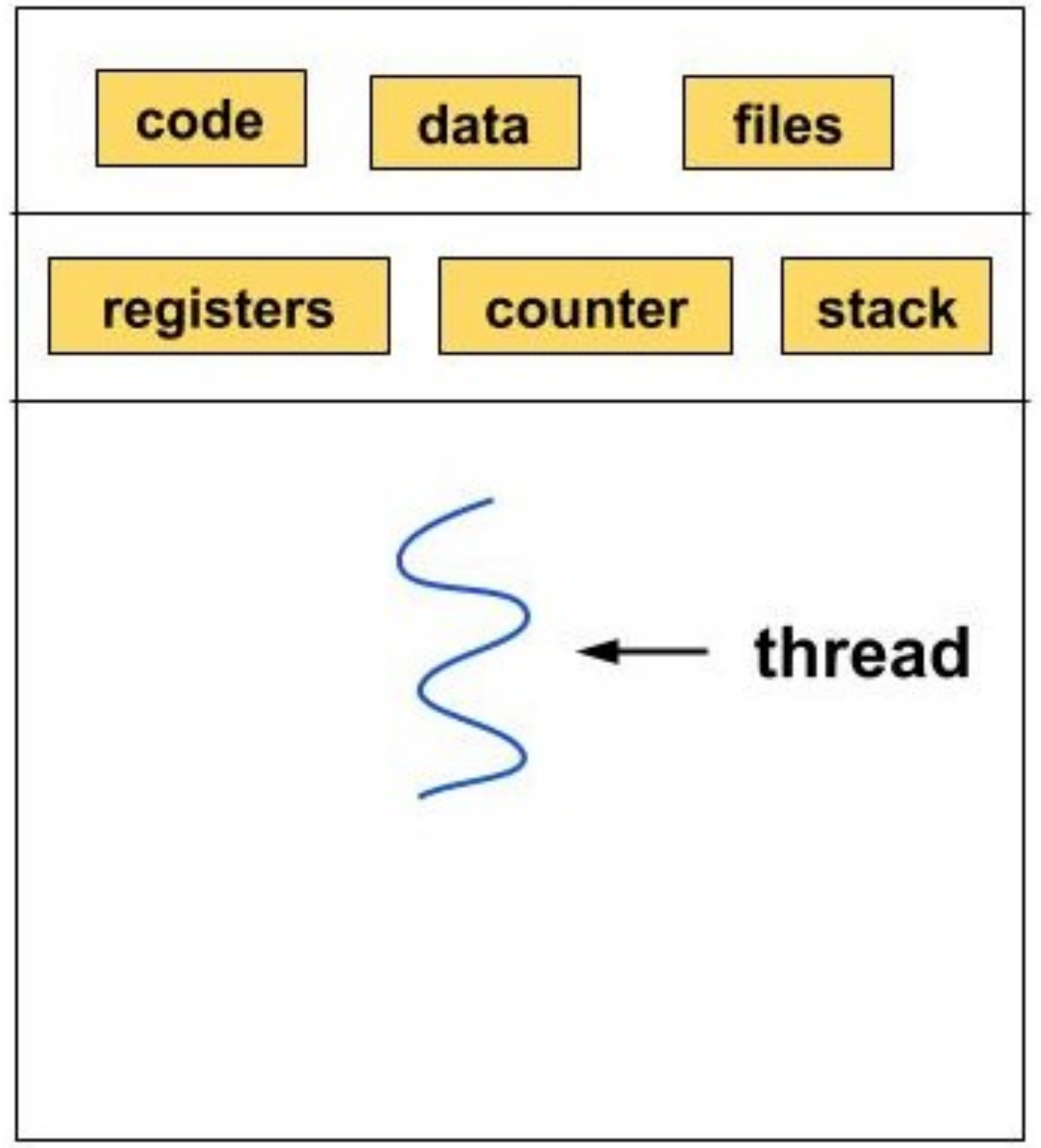
5W and H of Process Synchronization

- WHO are the main subjects in Process Synchronization?
- WHAT are the main characteristics of Process Synchronization?
- WHEN did Process Synchronization become important?
- WHERE is Process Synchronization used?
- WHY is it important?
- HOW is practiced?

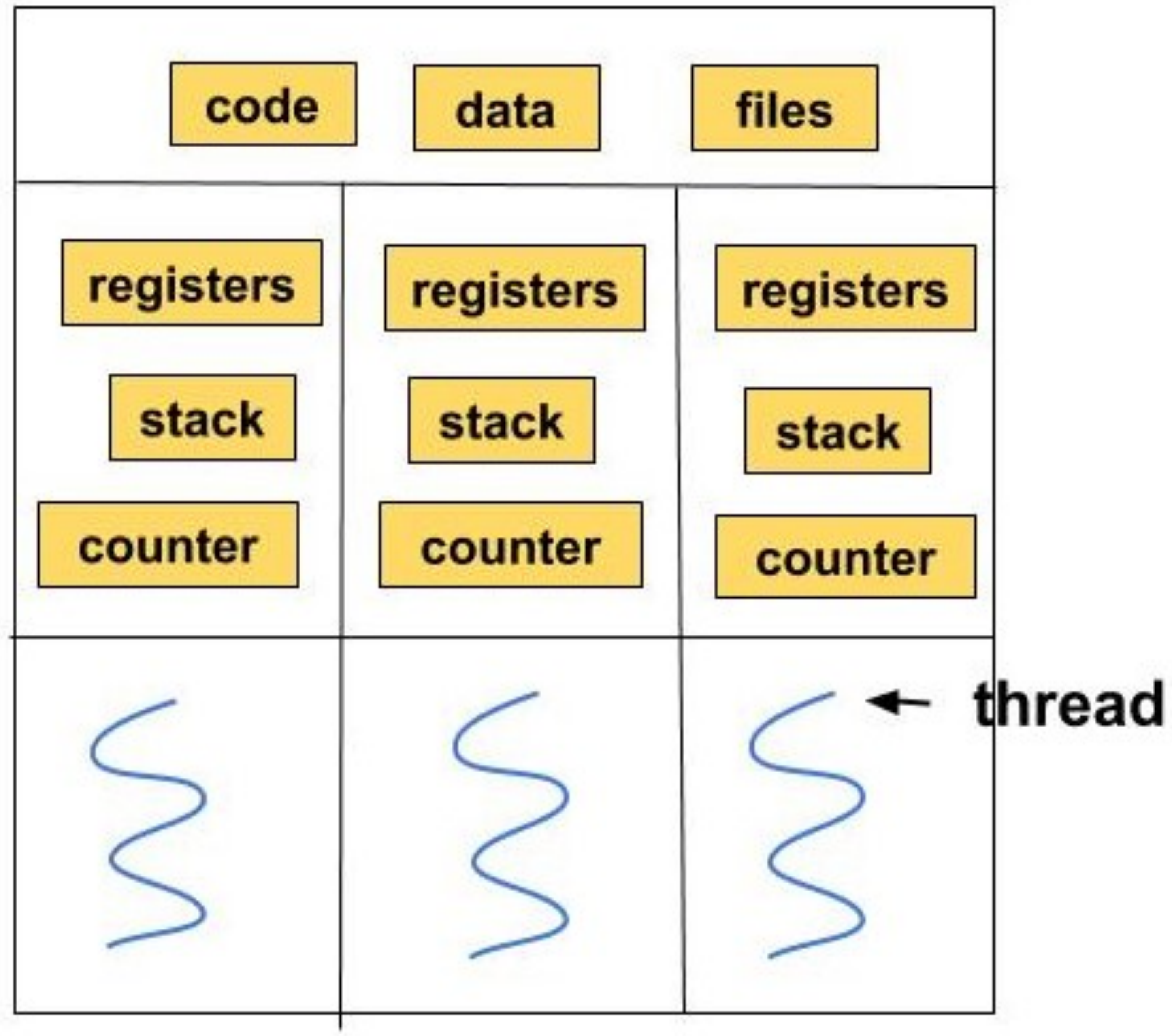
Who?

- Who are the main subjects of Process Synchronization?
    - Process (High level)
    - Resource (High level)
        - File
        - Memory
        - Devices
    - Thread (low level)
    - Critical Sections (low level)
    - Semaphores (low level)

Threads

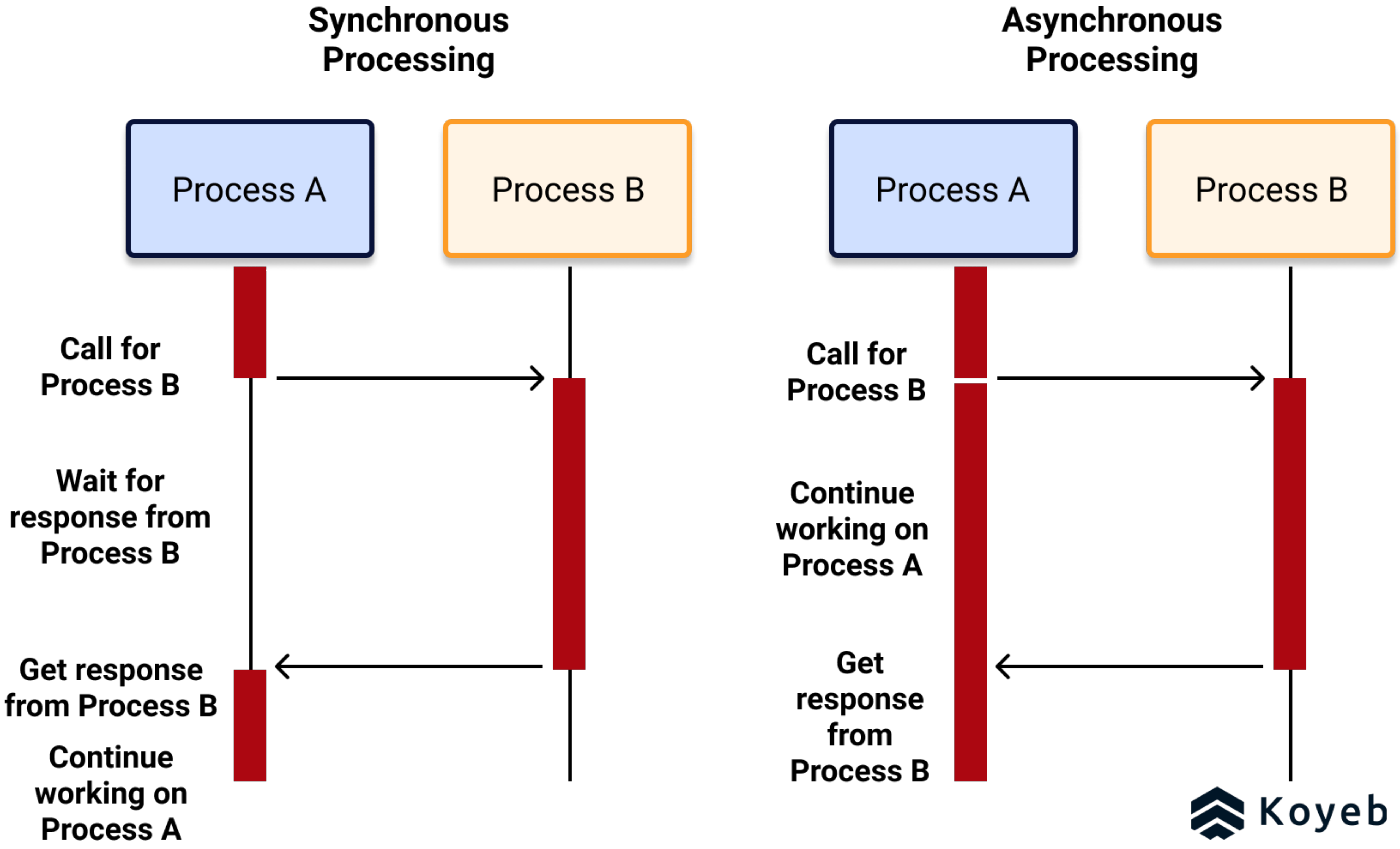


Single-threaded process



Multi-threaded process

Synchronous vs Asynchronous



Synchronous Processing
Process A
Process B
Call for Process B
Wait for response from Process B
Get response from Process B
Continue working on Process A
Asynchronous Processing
Process A
Process B
Call for Process B
Continue working on Process A
Get response from Process B

Koyeb

What?

- What is a standard definition for Process Synchronization?
- What are the various aspects of Process Synchronization?
- What are some personal experiences regarding Process Synchronization?

Standard Definition

- Process Synchronization:
  - Coordinate processes that use shared data.
  - Helps to maintain shared consistency and proper execution.
  - Processes have to be scheduled to ensure that concurrent access to shared data does not create inconsistencies.
  - Data inconsistency can result in what is called a race condition.
    - A race condition occurs when two or more operations are executed at the same time, not scheduled in the proper sequence, and not exited in the critical section correctly.

Consequences of Poor Synchronization

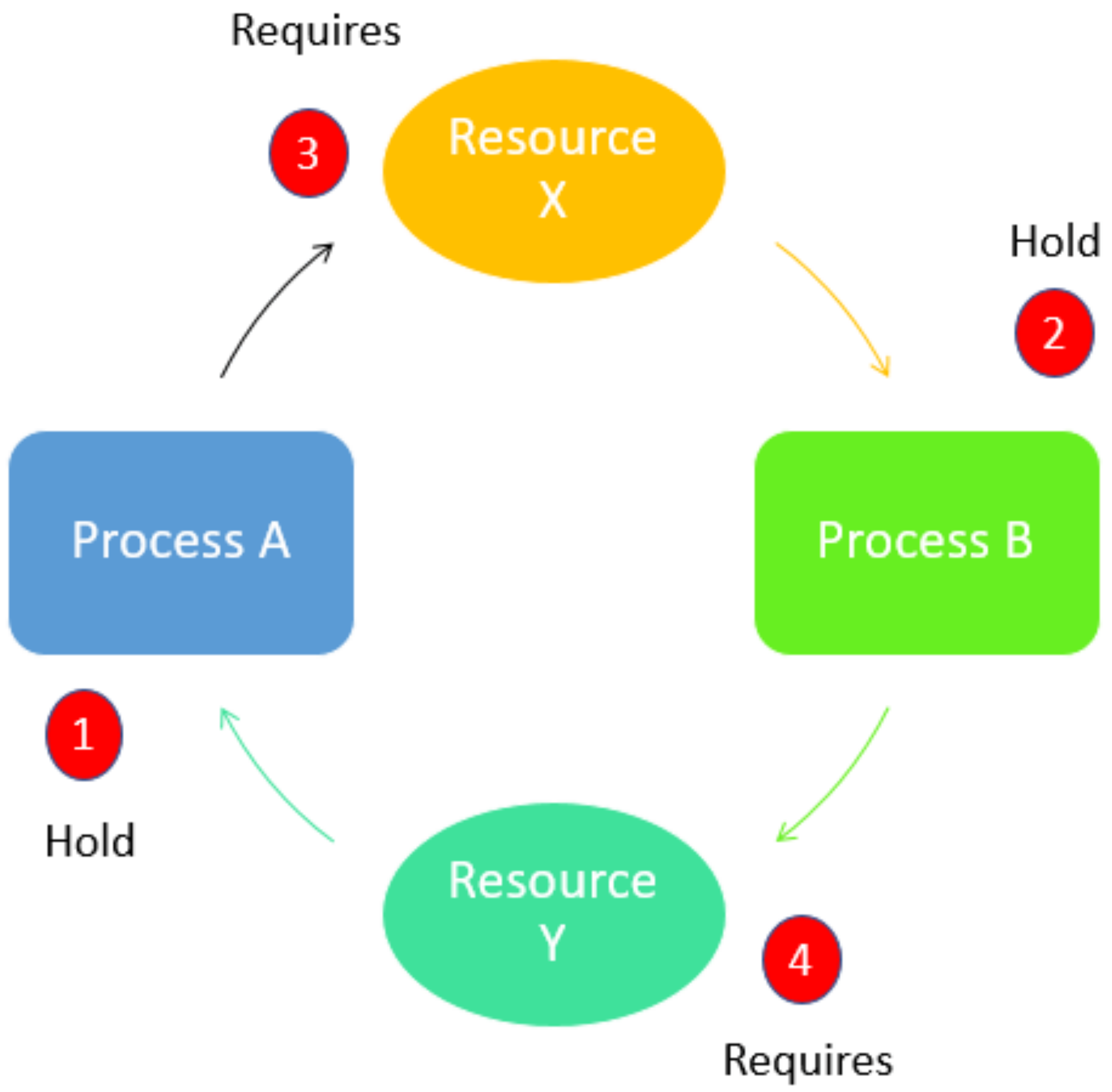
- Narrow staircase analogy
  - Staircase = system; steps and landings = resources
  - Stairs: only wide enough for one person
  - Landing at each floor: room for two people
- Deadlock: two people meet on the stairs
  - Neither retreats
- Livelock: two people on a landing
  - Each time one takes a step to the side, the other mirrors that step; neither moves forward
  - Or, if two people are calling each other at the same time & find the other one busy
- Starvation: people wait on landing for a break
  - Break never comes

Consequences of Poor Synchronization

- Affects entire system
    - Affects more than one job
        - Not just a few programs
    - All system resources become unavailable
- More prevalent in interactive systems
- Real-time systems
    - Deadlocks quickly become critical situations
- OS must **prevent** or **resolve**

Live Lock

- Looks like Deadlock, but because they are POLLING (check to see if it is available), and if it is not, they keep processing.

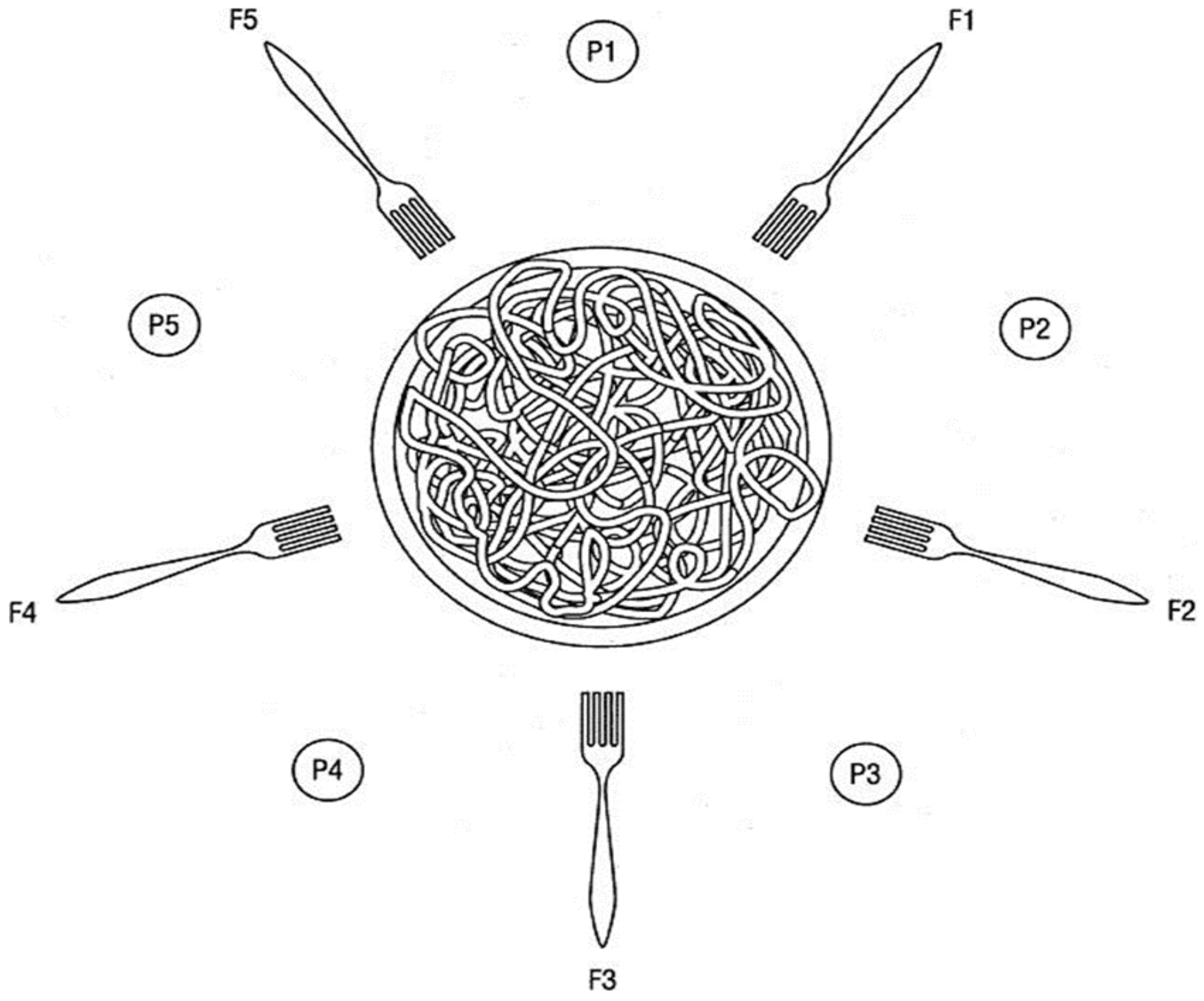


Code?

Starvation- Dining Philosopher's Table

- The dining philosophers' table, before the meal begins. Each philosopher is indicated by a circled P and each fork resource is shown as an F.

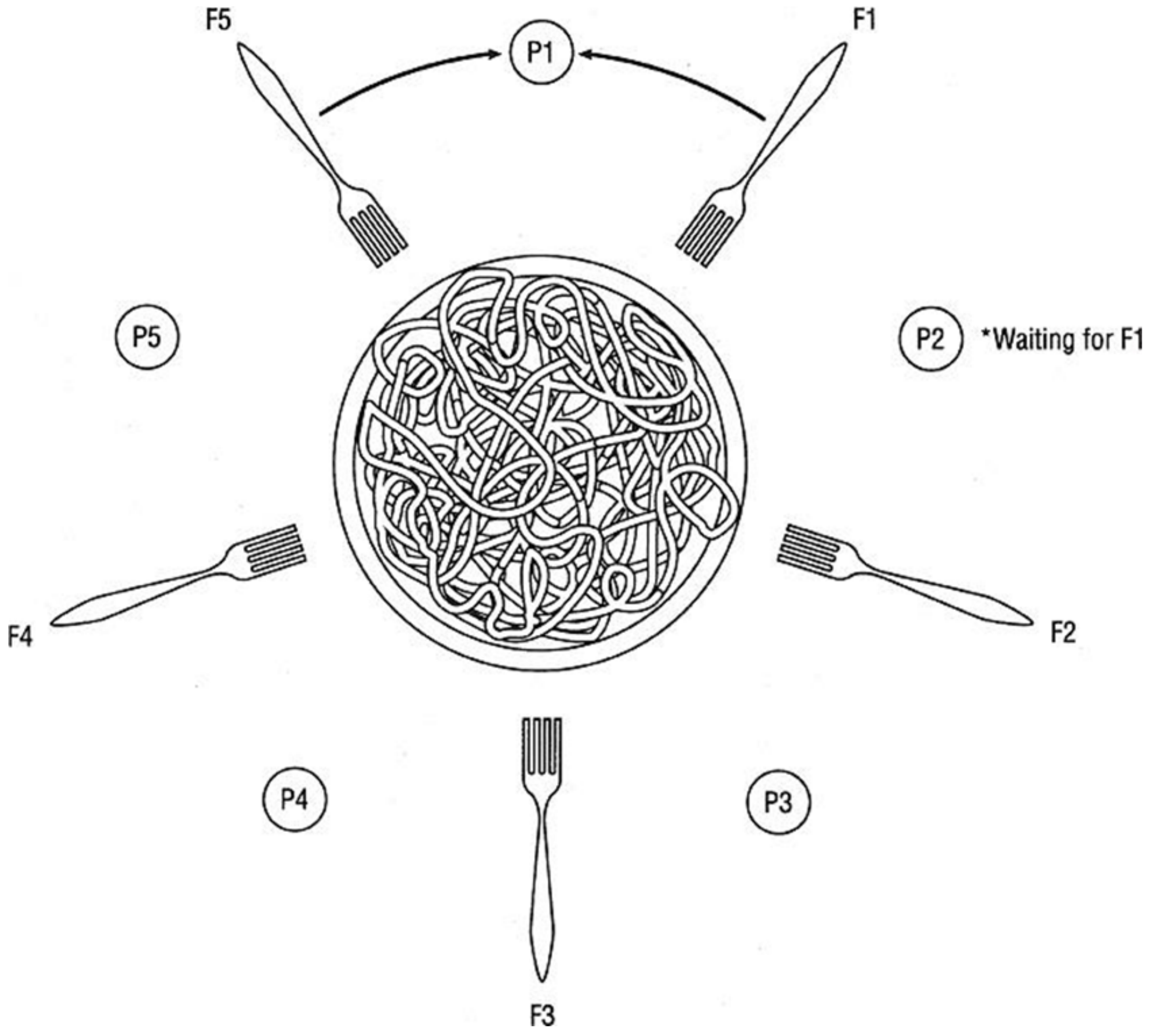
- Each philosopher must have both forks to begin eating, the one on the right and the one on the left. Unless the resources, the forks, are allocated fairly, some philosophers may starve.



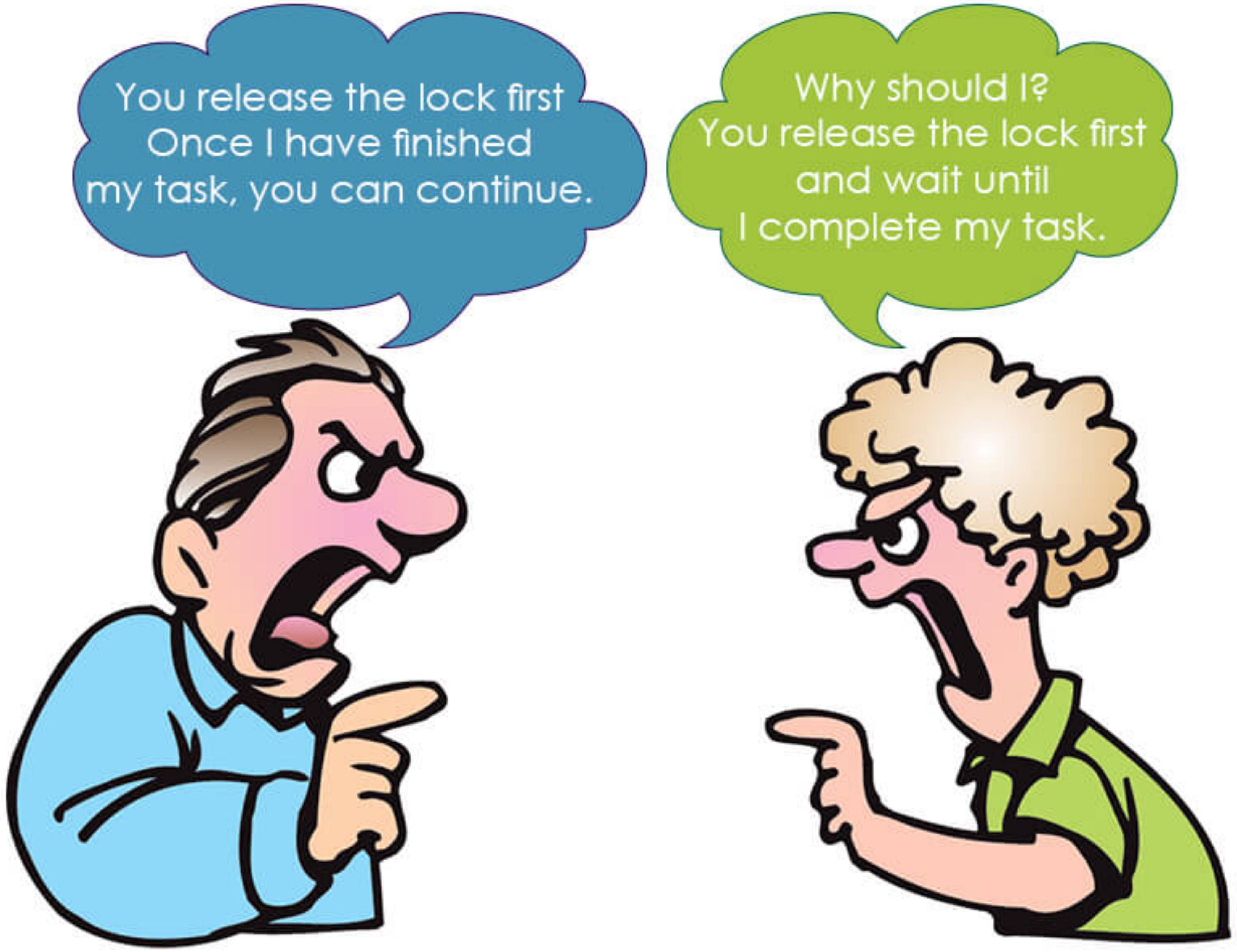
Starvation- Dining Philosopher's Table

- Situation: P1 and P3 grab two forks and start eating. P2 wants to eat, but no forks available. P3 finishes. P2 again wants to eat, but F1 not available. P3 goes back to eat, and P1 finishes. P2 again wants to eat but BOTH forks are not available.

  This is a Starvation issue, as the request can't get granted!!! Is their an algorithm we can use?



Deadlock - Process Synchronization

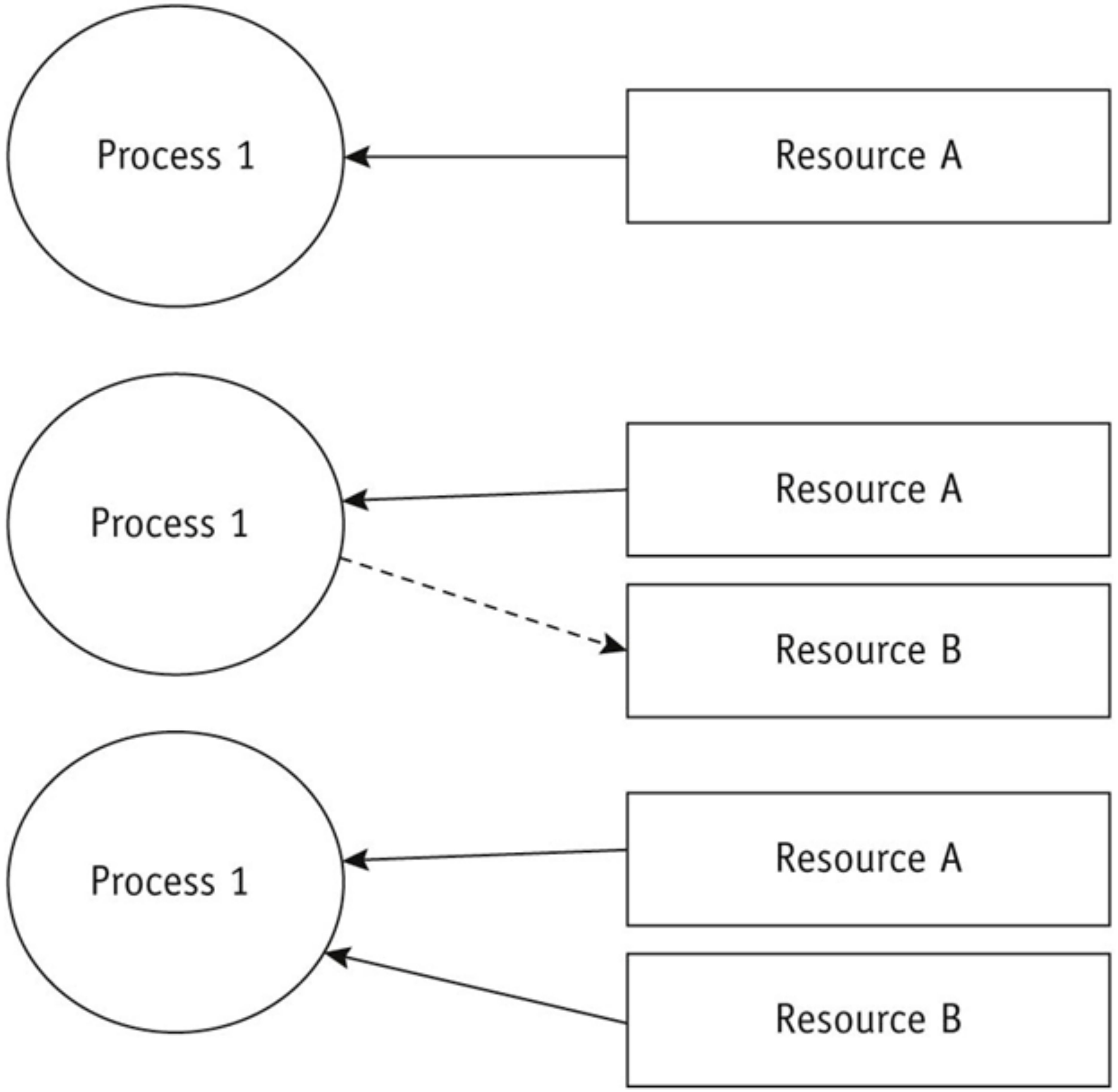

You release the lock first
Once I have finished
my task, you can continue.
Why should I?
You release the lock first
and wait until
I complete my task.

Directed Graphs

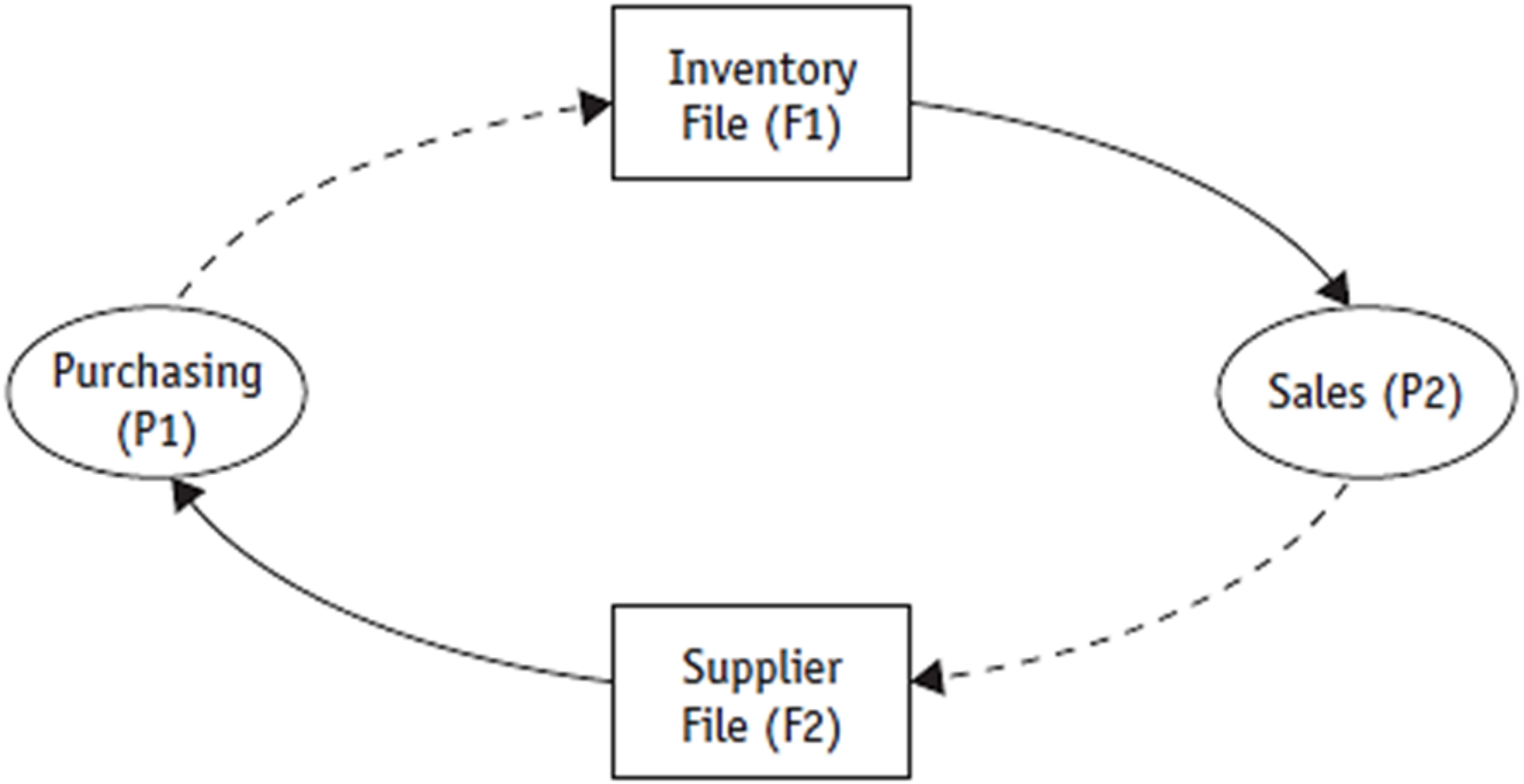
- In 1972,Richard Hold published a visual tool to show how deadlocks can be modified using directed graphs
- Circles represented processes, and squares represented resources.
- A solid line represents that is has been allocated to
- A dashed line represents that it has been requested

Modeling Deadlocks with Directed Graphs

- Solid=Allocated
  Dashed=Requested
- Resource = (File, Printer, Lock, Database, …)



Is this a Deadlock?



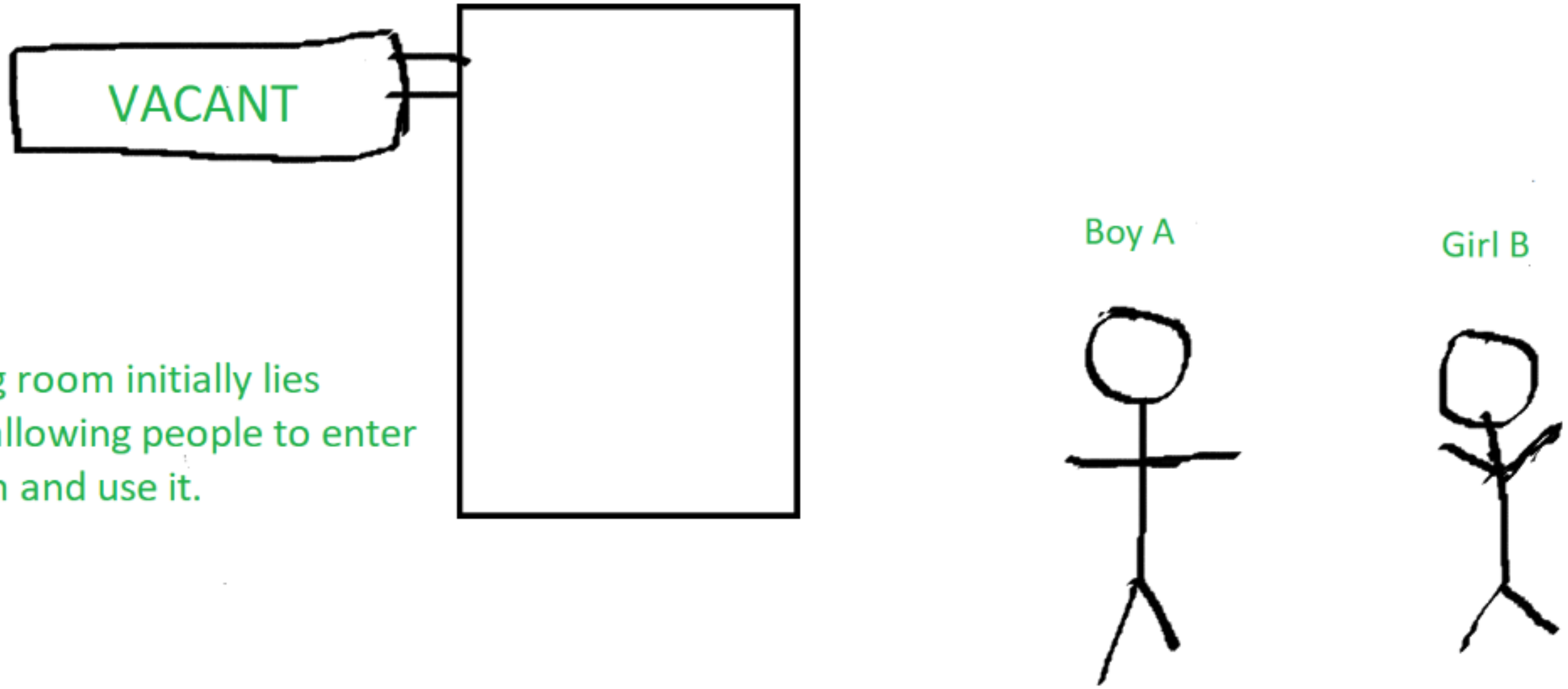
Necessary Conditions for Deadlock

- Four conditions required for a locked system
    - Mutual Exclusion
        - Two want the same thing, but only one can have it. On the tower, Person A has the step I want, but they are not going to give it up. The stair step can only be occupied by 1 person
    - Hold & Wait
        - You are both coming from two directions on a sidewalk. You are suddenly within 6 feet of that person during COVID. Both people stop, but nobody takes any action. They keep waiting for the other person to back off, but nobody does
    - No Preemption
        - No-one can ask you to give up your resource. You must do it yourself voluntarily.
    - Circular Wait
        - Circular dependency chain
- **All conditions required for deadlock**
- Resolving deadlock: ?

Necessary Conditions for Deadlock

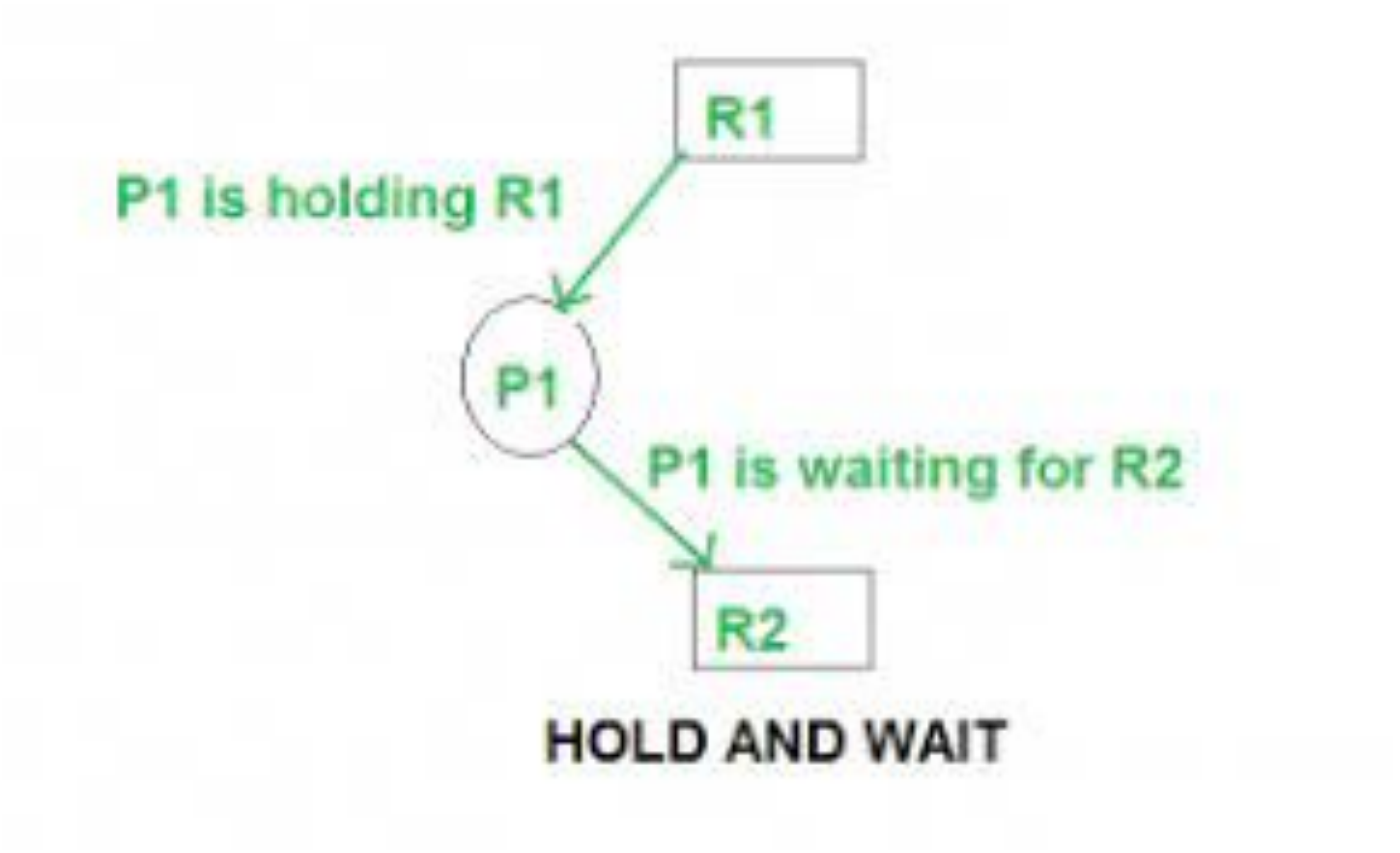
- Four conditions required for a locked system
  - Mutual Exclusion
    - Two want the same thing, but only one can have it. On the tower, Person A has the step I want, but they are not going to give it up. The stair step can only be occupied by 1 person
  - Hold & Wait
    - You are both coming from two directions on a sidewalk. You are suddenly within 6 feet of that person during COVID. Both people stop, but nobody takes any action. They keep waiting for the other person to back off, but nobody does
  - No Preemption
    - No-one can ask you to give up your resource. You must do it yourself voluntarily.
  - Circular Wait
    - Circular dependency chain
- **All conditions required for deadlock**
- Resolving deadlock: remove one of the conditions.

Mutual Exclusion



Changing room initially lies vacant, allowing people to enter the room and use it.

Hold & Wait

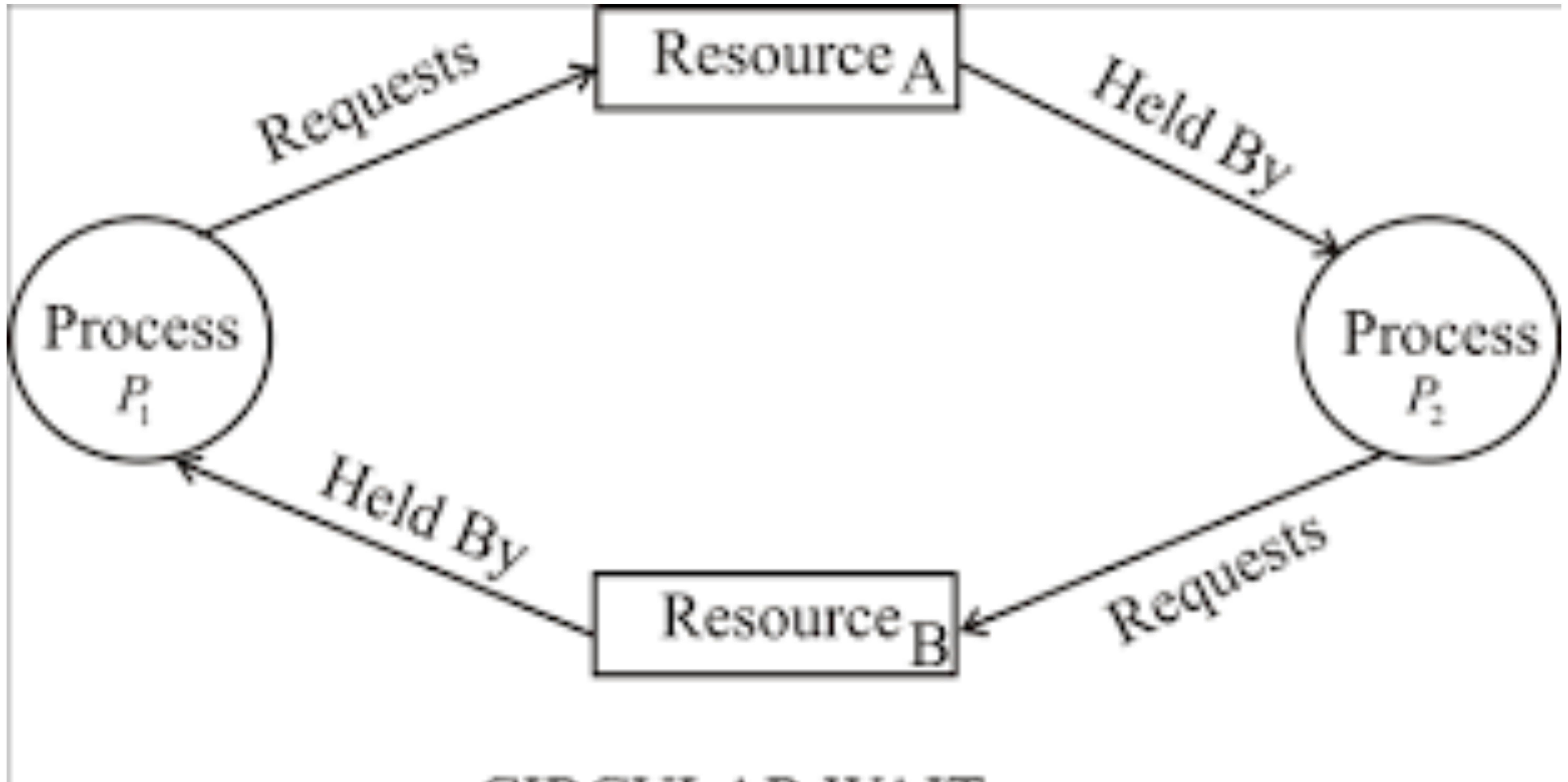


HOLD AND WAIT

No Preemption

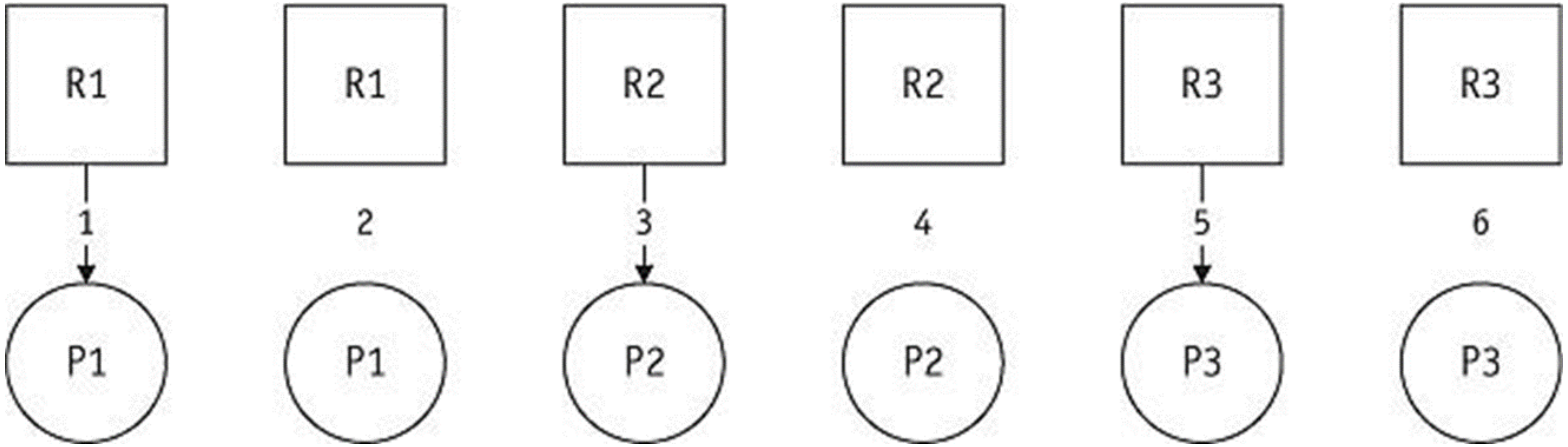
- No Preemption suggests that the life preserver would not be able to save a deadlocked situation.

Circular Wait



CIRCULAR WAIT

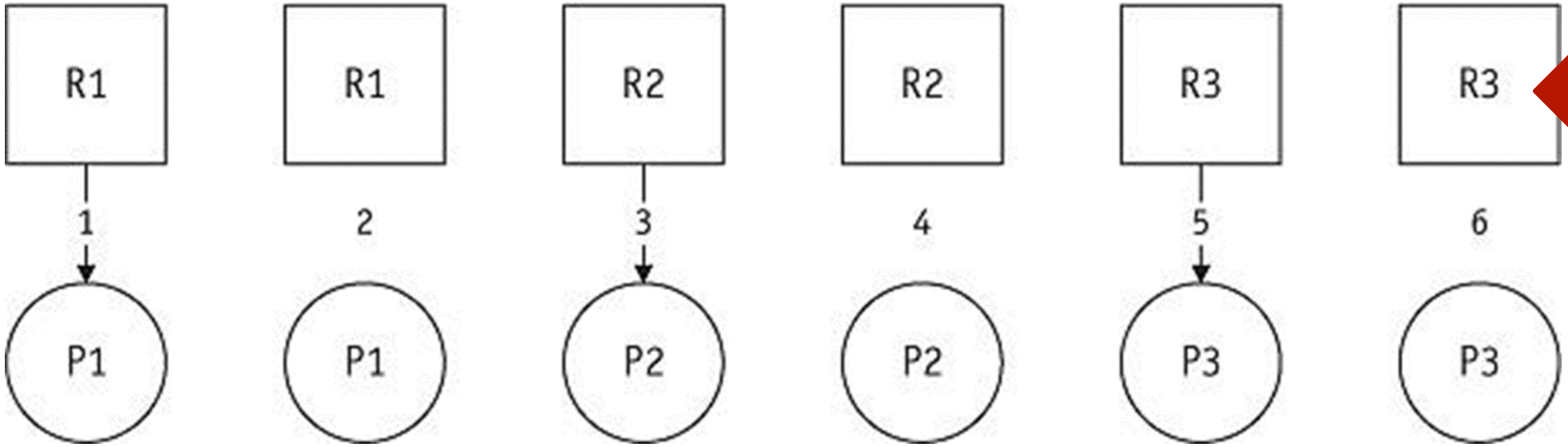
Directed Graphs - Do we have a Deadlock here?



Event	Action
1	Process 1 (P1) requests and is allocated the printer (R1).
2	Process 1 releases the printer.
3	Process 2 (P2) requests and is allocated the disk drive (R2).
4	Process 2 releases the disk drive.
5	Process 3 (P3) requests and is allocated the plotter (R3).
6	Process 3 releases the plotter.

Directed Graphs - Do we have a Deadlock

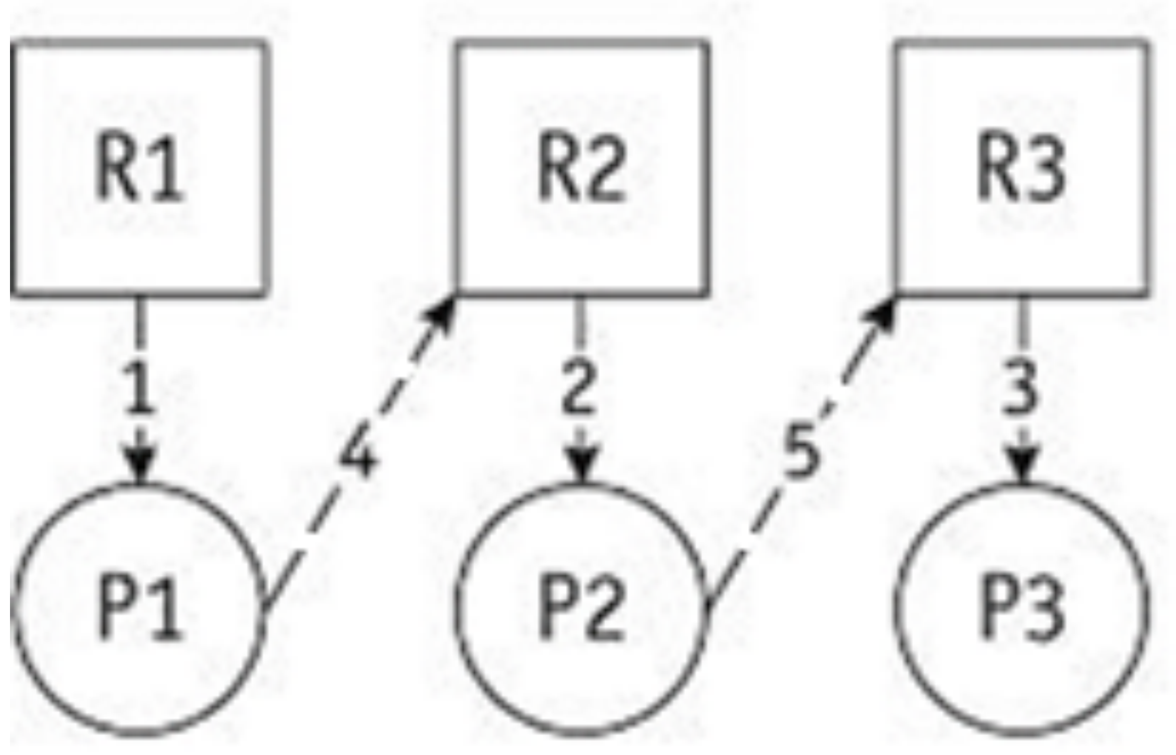
No Circular Dependency



Event	Action
1	Process 1 (P1) requests and is allocated the printer (R1).
2	Process 1 releases the printer.
3	Process 2 (P2) requests and is allocated the disk drive (R2).
4	Process 2 releases the disk drive.
5	Process 3 (P3) requests and is allocated the plotter (R3).
6	Process 3 releases the plotter.

Directed Graphs - Do we have a Deadlock here?

- Scenario 2: resource holding
- Processes waiting for resource held by another



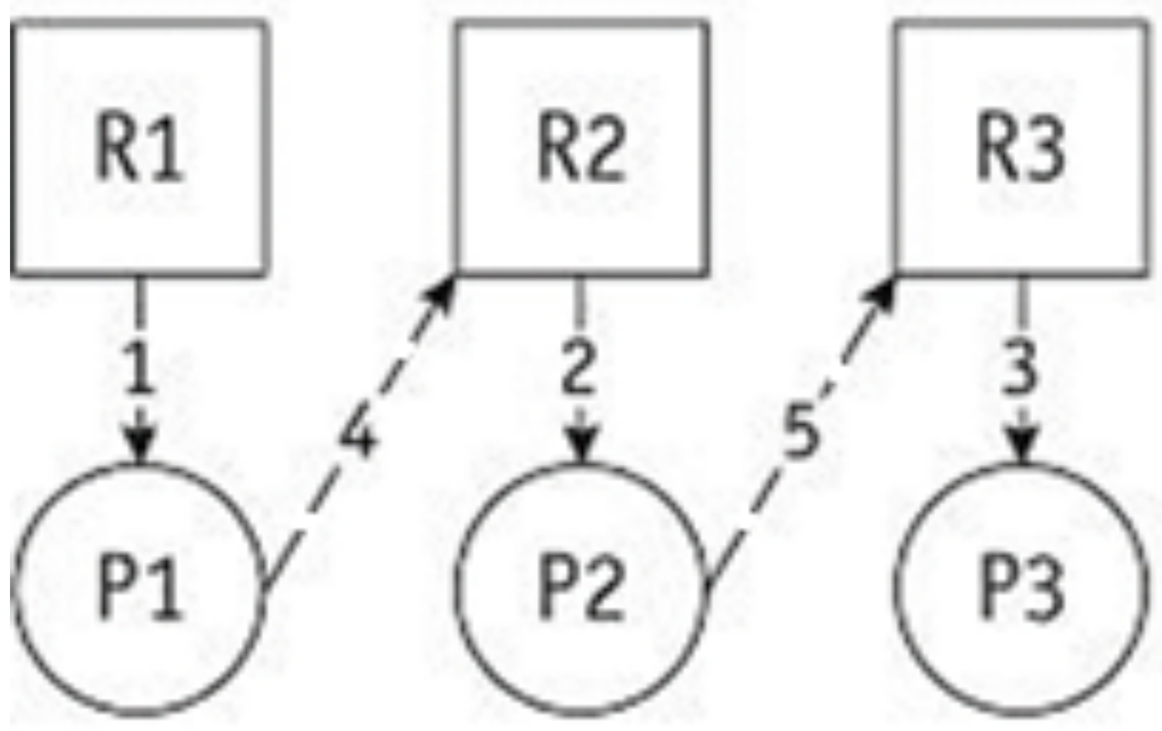
(a)

Event	Action
1	P1 requests and is allocated R1.
2	P2 requests and is allocated R2.
3	P3 requests and is allocated R3.
4	P1 requests R2.
5	P2 requests R3.

Directed Graphs - Do we have a Deadlock

No Circular Dependency

- Scenario 2: resource holding
- Processes waiting for resource held by another

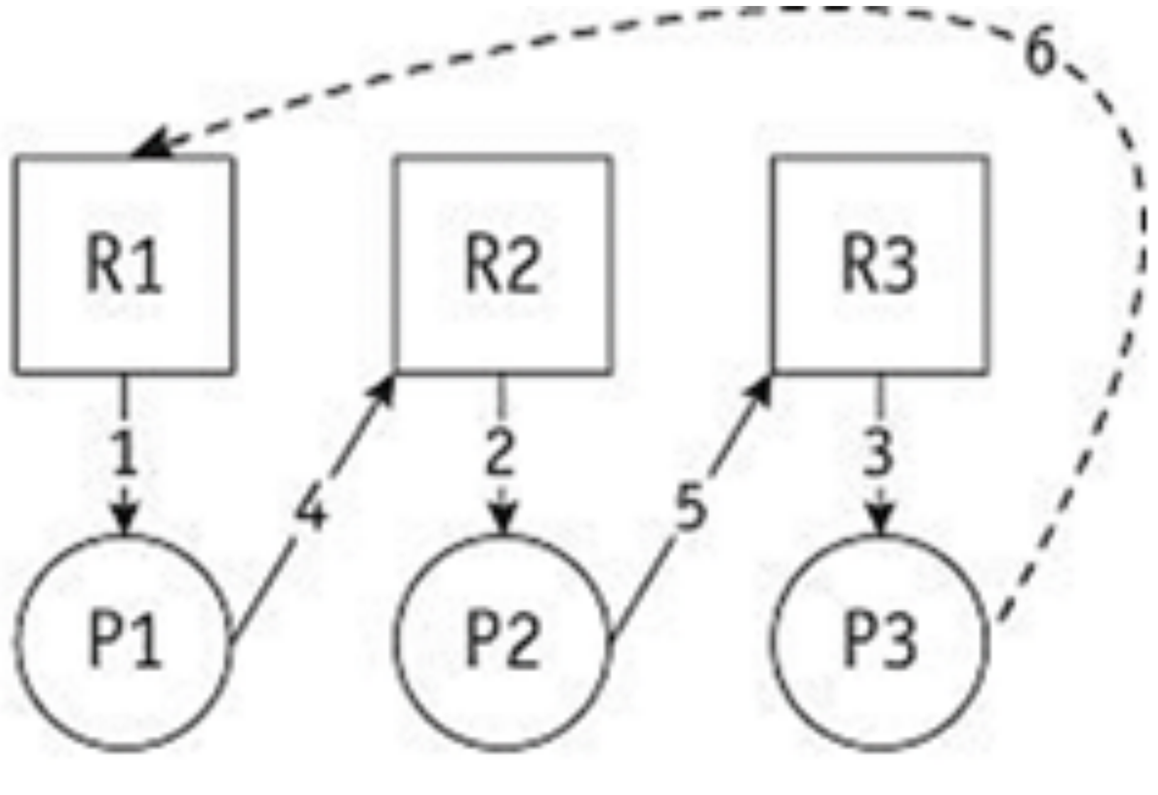


(a)

Event	Action
1	P1 requests and is allocated R1.
2	P2 requests and is allocated R2.
3	P3 requests and is allocated R3.
4	P1 requests R2.
5	P2 requests R3.

Directed Graphs - Do we have a Deadlock here?

- Scenario 2: resource holding
- Processes waiting for resource held by another



(b)

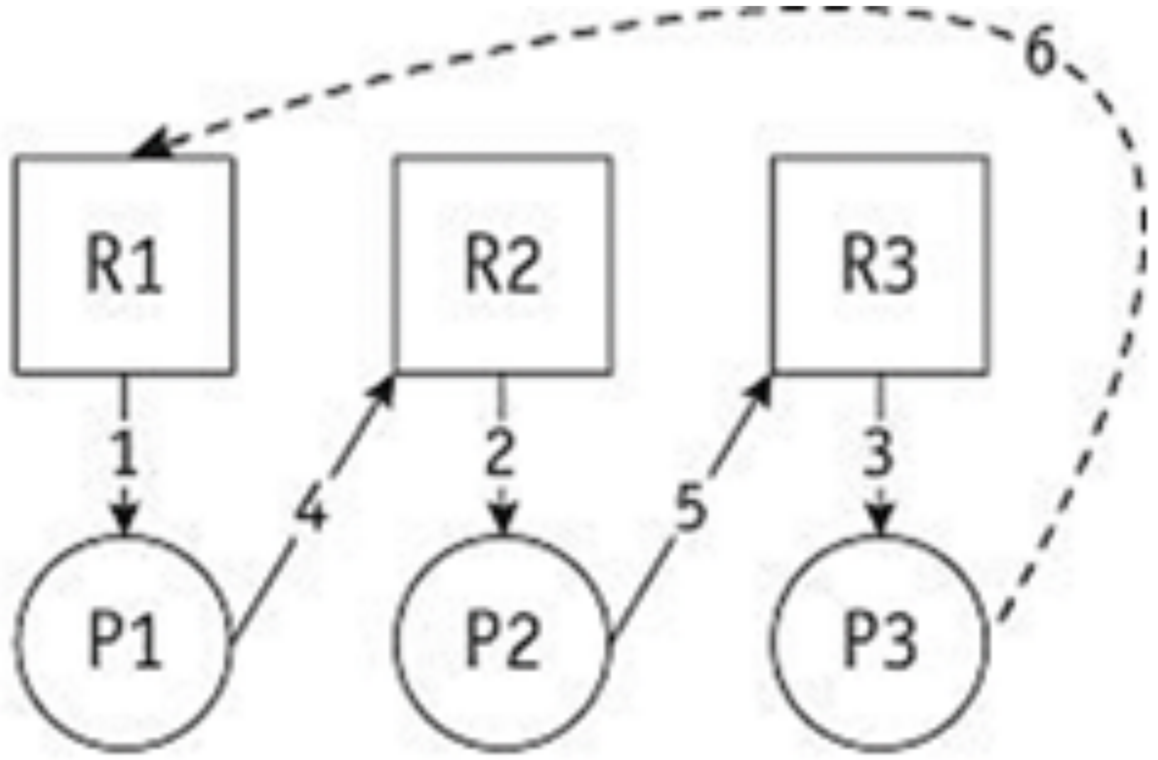
Event	Action
1	P1 requests and is allocated R1.
2	P2 requests and is allocated R2.
3	P3 requests and is allocated R3.
4	P1 requests R2.
5	P2 requests R3.
6	P3 requests R1.

Directed Graphs - Do we have a Deadlock here?

All conditions met

- Scenario 2: resource holding

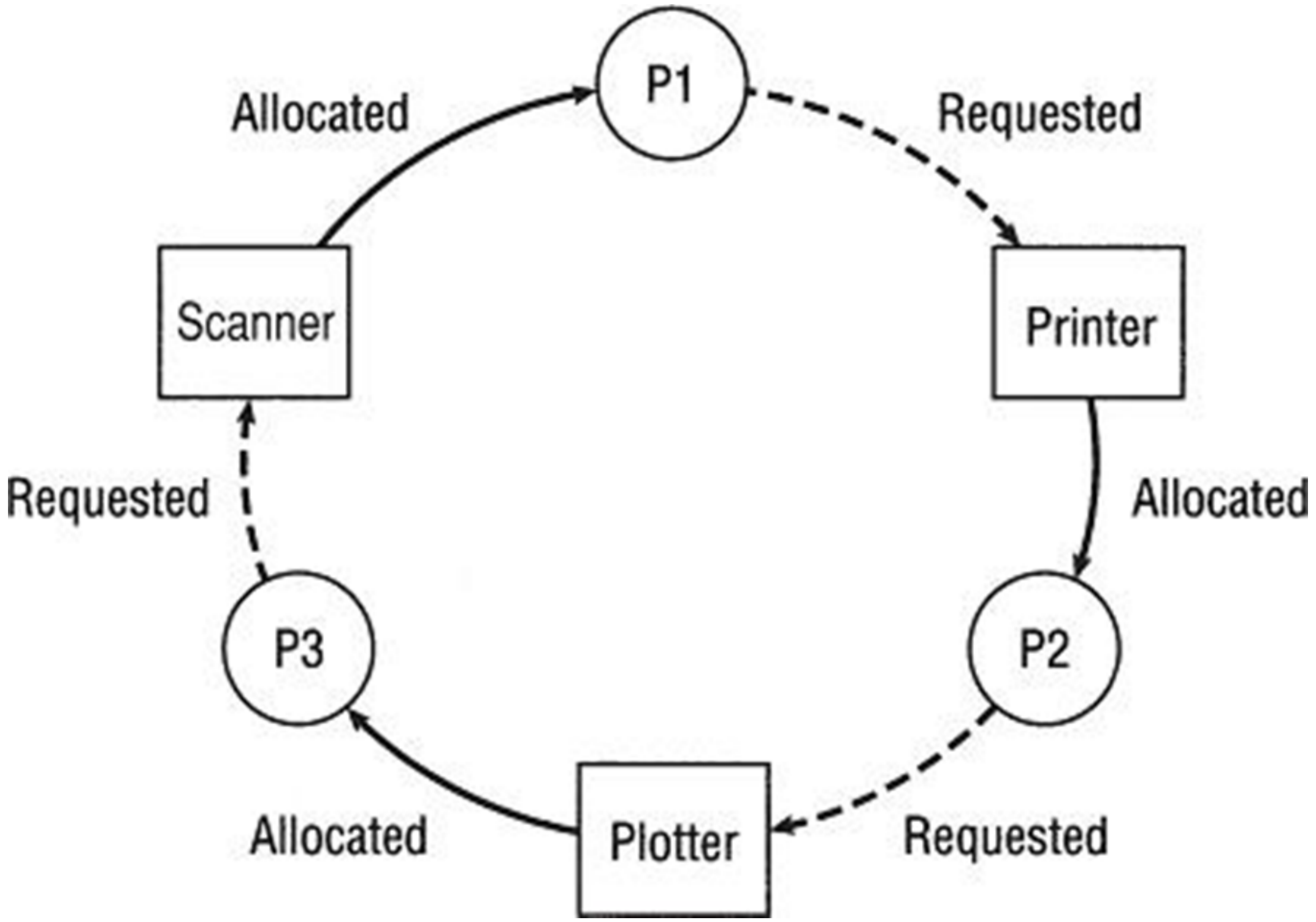
- Processes waiting for resource held by another



(b)

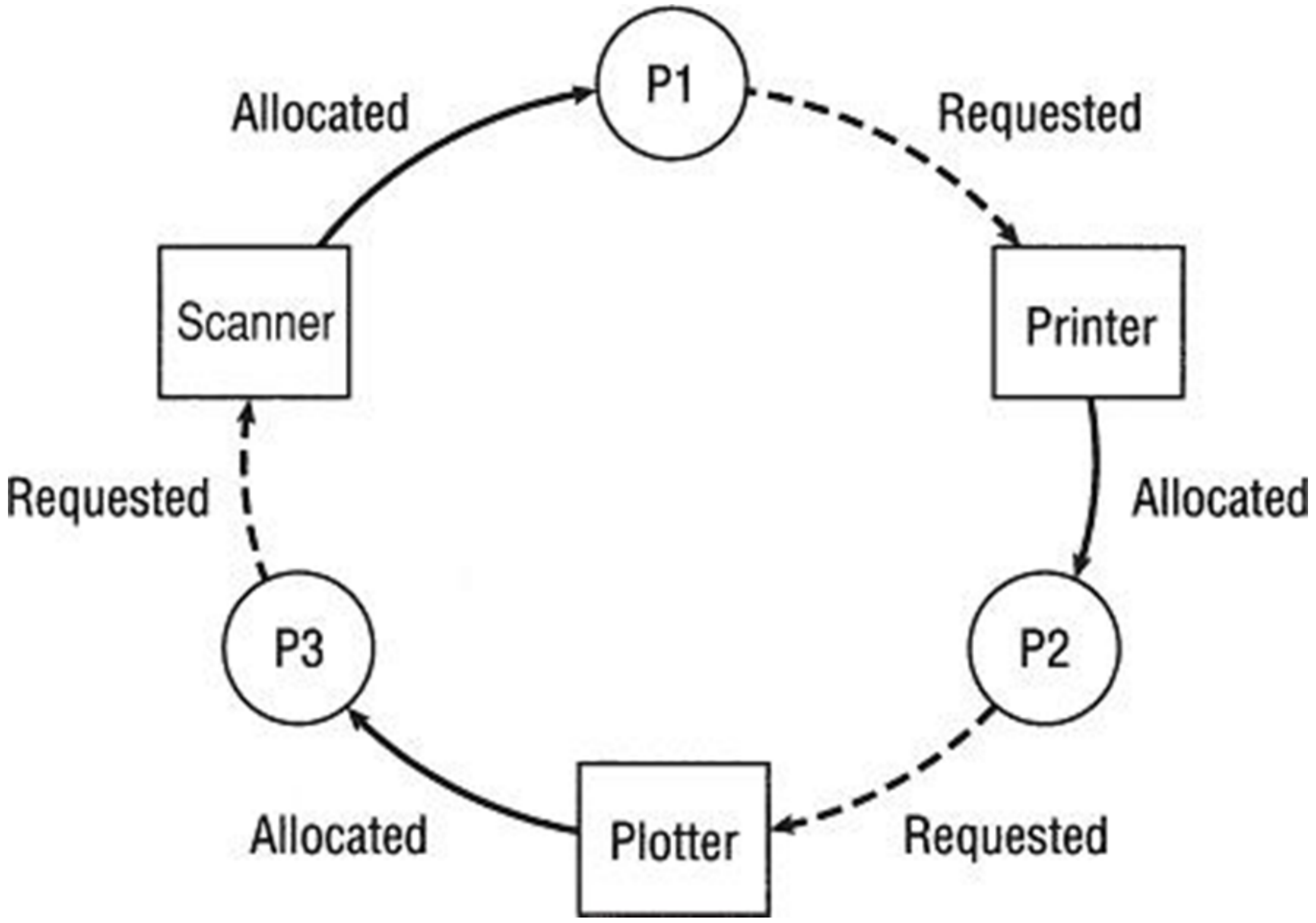
Event	Action
1	P1 requests and is allocated R1.
2	P2 requests and is allocated R2.
3	P3 requests and is allocated R3.
4	P1 requests R2.
5	P2 requests R3.
6	P3 requests R1.

Directed Graphs - Do we have a Deadlock here?



Directed Graphs - Do we have a Deadlock here?

All conditions met



Code?

When?

- When did Process Synchronization get introduced to Computer Science?

Edsger Dijkstra in 1965

- Mutual exclusion is a property of concurrency control, which is instituted for the purpose of preventing race conditions. It is the requirement that one thread of execution never enters a critical section while a concurrent thread of execution is already accessing critical section

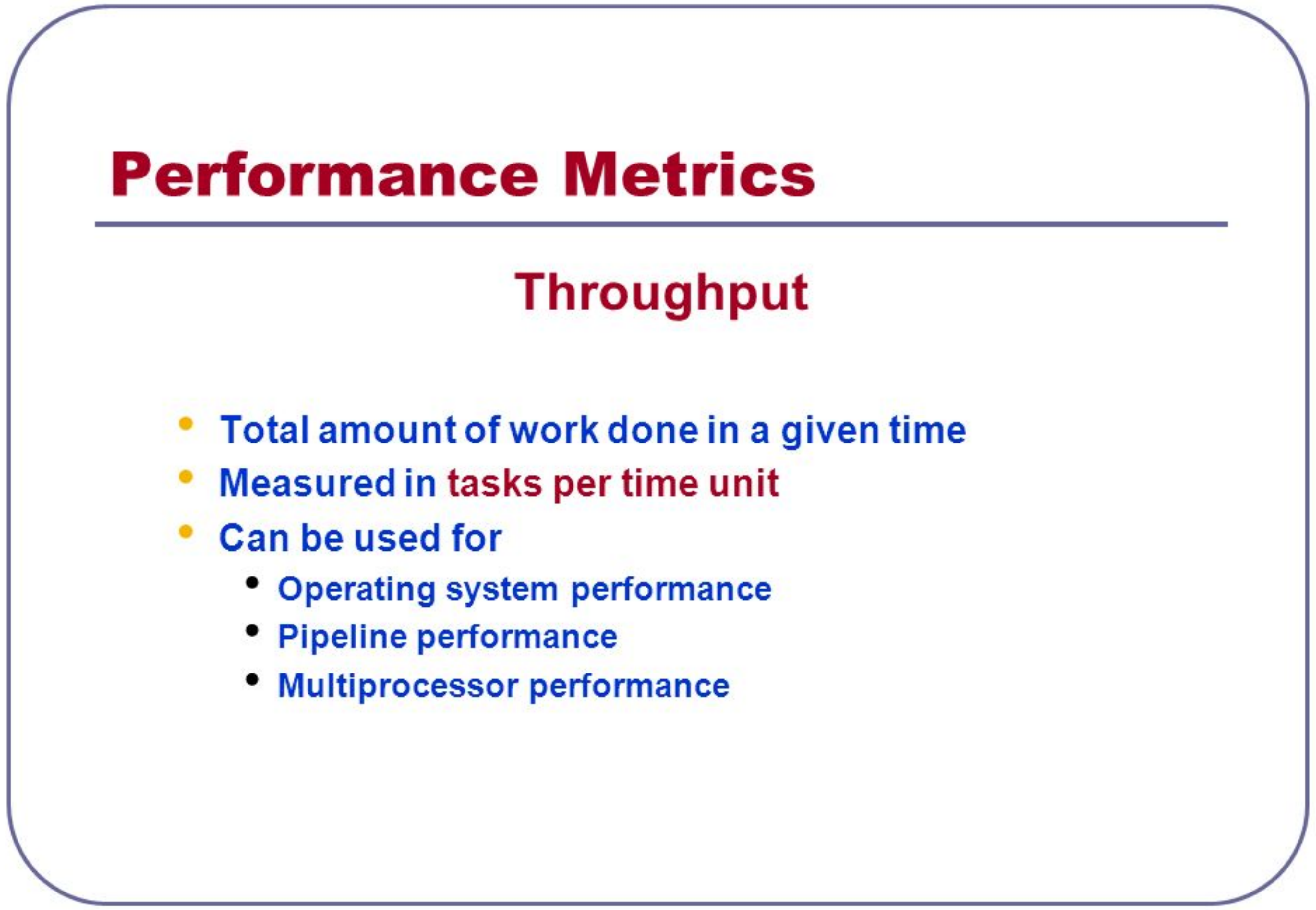
- 1965 paper "Solution of a problem in concurrent programming control", which is credited as the first topic in the study of concurrent algorithms.

Where?

- Where is Process Synchronization found?

Why?

- Why do we care about Process Synchronization?



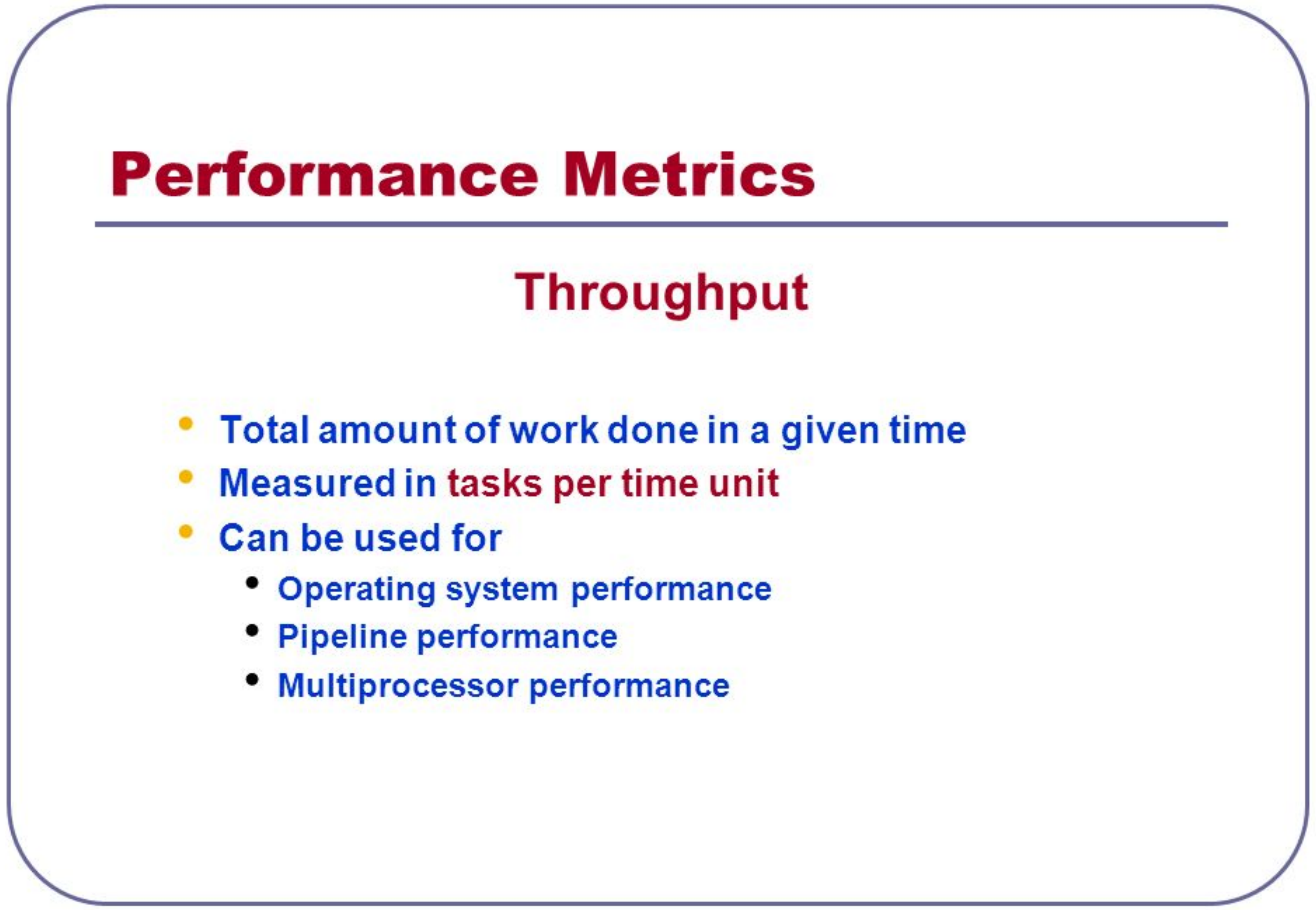
Performance Metrics

Throughput

- Total amount of work done in a given time
- Measured in tasks per time unit
- Can be used for
  - Operating system performance
  - Pipeline performance
  - Multiprocessor performance

Why?

- Why do we care about Process Synchronization?
  - Reusability, Flexibility, and Value



Performance Metrics

Throughput

- Total amount of work done in a given time
- Measured in tasks per time unit
- Can be used for
  - Operating system performance
  - Pipeline performance
  - Multiprocessor performance

How?

- How do Developers use Process Synchronization in code.
    - Much of this will come next week, as we dive into Multi-Threading, Critical Sections, Semaphores, Mutual Exclusion.
    - This week, we focused on the problems that occur when you don’t have it.
        - Live lock
        - Deadlock
        - Starvation

Strategies for Handling Deadlocks

- Prevention
  - Prevent occurrence of one condition
    - Mutual exclusion, resource holding, no preemption, circular
- Avoidance
  - Avoid deadlock if it becomes probable
- Detection
  - Detect deadlock when it occurs
- Recovery
  - Resume system normalcy quickly and gracefully

Prevention

- **Mutual exclusion**
    - Some resources must allocate exclusively. You can’t request. You either get it or you don’t. Kind of strict, but it will prevent things from being deadlocked.
    - E.g. Printers - requires exclusive access
    - However, NOT all shared resources cause deadlock
    - e.g. Multiple processes can read a file. But multiple processes may not write to it.
    - Spooling: Simultaneous Peripheral Operations Online
        - A Printer has associated memory which can be used as a spooler directory (memory that is used to store files that are to be printed next).
        - In spooling, when multiple processes request the printer, their jobs (instructions of the processes that require printer access) are added to the queue in the spooler directory.
        - The printer is allocated to jobs on a first come first serve (FCFS) basis. In this way, the process does not have to wait for the printer and it continues its work after adding its job to the queue.
        - May cause race conditions: if process A overwrites the job of process B in the queue, then process B will never receive the output.
        - Not full-proof: after the queue becomes full, incoming processes go into a waiting or rejected state.

Prevention

- **Resource holding**
  - #1 Eliminate Wait
    - Jobs request every necessary resource at creation time
    - But Multiprogramming degree significantly decreased
    - Idle peripheral devices
  - #2 Eliminate Hold
    - Process has to release all resources it is currently holding before making a new request
      - But, releasing ALL resources might be unnecessary as no-one else was waiting for it

Prevention

- No pre-emption
    - #1 Process has to release all resources it is currently holding before making a new request
        - But, releasing ALL resources might be unnecessary as no-one else was waiting for it
    - #2 Operating system allowed to deallocate resources from jobs (so preemption happens), but
        - Okay if job state easily saved and restored
        - Pre-empting dedicated I/O device or files during modification: extremely unpleasant recovery tasks & data inconsistency

Prevention

- **Circular wait**
  - Bypassed if operating system prevents circle formation
    - Uses hierarchical ordering scheme: Assign priority to each resource
      - Priorities probably are process dependent e.g. media player will give lesser priority to printer but Microsoft Word will give it higher priority
    - Requires jobs to anticipate resource request order, but
      - Difficult to satisfy all users

Is Deadlock Prevention feasible?

- Mutual exclusion cannot be eliminated completely because some resources are inherently non-shareable

- Hold and wait cannot be eliminated as we cannot know in advance about the required resources to prevent waiting. It is inefficient to prevent a hold by releasing all the resources while requesting a new one

- Preempting processes can cause inconsistency and starting the process over by putting requests for all resources again is inefficient.

- Eliminating circular wait is the only practical way to prevent deadlock.

Avoidance

- Avoidance: use if condition cannot be removed
    - System knows ahead of time
        - Sequence of requests associated with each active process
- Dijkstra’s Bankers Algorithm (Dijkstra, 1965)
    - Regulates resource allocation to avoid deadlocks
        - No customer granted loan exceeding bank’s total capital
        - All customers given maximum credit limit
        - No customer allowed to borrow over limit
        - Sum of all loans will not exceed bank’s total capital

Banker’s Algorithm

- The bank started with $10,000 and has remaining capital of $4,000 after these loans. Therefore, it’s in a **safe** state (barely).

Customer	**Loan Amount**	**Maximum Credit**	**Remaining Credit**
Customer #1	0	4,000	4,000
Customer #2	2,000	5,000	3,000
Customer #3	4,000	8,000	4,000
Total loaned: $ 6,000			
Total capital fund: $10,000			

Banker’s Algorithm

- A few weeks later, the bank has made some loans, and (gotten some money), and is in this state. The bank only has remaining capital of $1,000 after these loans and, therefore, is in an **unsafe** state, since it cannot handle a condition where any of those customers could come back and ask for more.

Customer	**Loan Amount**	**Maximum Credit**	**Remaining Credit**
Customer #1	2,000	4,000	2,000
Customer #2	3,000	5,000	2,000
Customer #3	4,000	8,000	4,000
Total loaned: $ 9,000			
Total capital fund: $10,000			

Bankers Algorithm

- Resource assignments after initial allocations. This is a **safe** state: Six devices are allocated and four units are still available.

Job No.	**Device Allocated**	**Maximum Required**	**Remaining needs**
Job 1	0	4	4
Job 2	2	5	3
Job 3	4	8	4
Total number of device allocated : 6			
Total number of device is system: 10			

Bankers Algorithm

- Resource assignments after later allocations. This is an **unsafe** state: Only one unit is available, but every job requires at least two to complete its execution.

Job No.	**Device Allocated**	**Maximum Required**	**Remaining needs**
Job 1	2	4	2
Job 2	3	5	2
Job 3	4	8	4
Total number of device allocated : 9			
Total number of device is system: 10			

Banker's Algorithm

- A few weeks later, the bank has made some loans, and (gotten some money), and is in this state. The bank only has remaining capital of $1,000 after these loans and, therefore, is in an unsafe state, since it cannot handle a condition where any of those customers could come back and ask for more.

Customer	**Loan Amount**	**Maximum Credit**	**Remaining Credit**
Customer #1	2,000	4,000	2,000
Customer #2	3,000	5,000	2,000
Customer #3	4,000	8,000	4,000
Total loaned: $ 9,000			
Total capital fund: $10,000			

Do you see the concept? We don't want to be in a position of not being able to satisfy request

So what do we learn?

- Never satisfy request if job state moves from safe to unsafe
    - Identify job with smallest number of remaining resources
    - Number of available resources >= number needed for selected job to complete
    - Block request jeopardizing safe state.

Code?

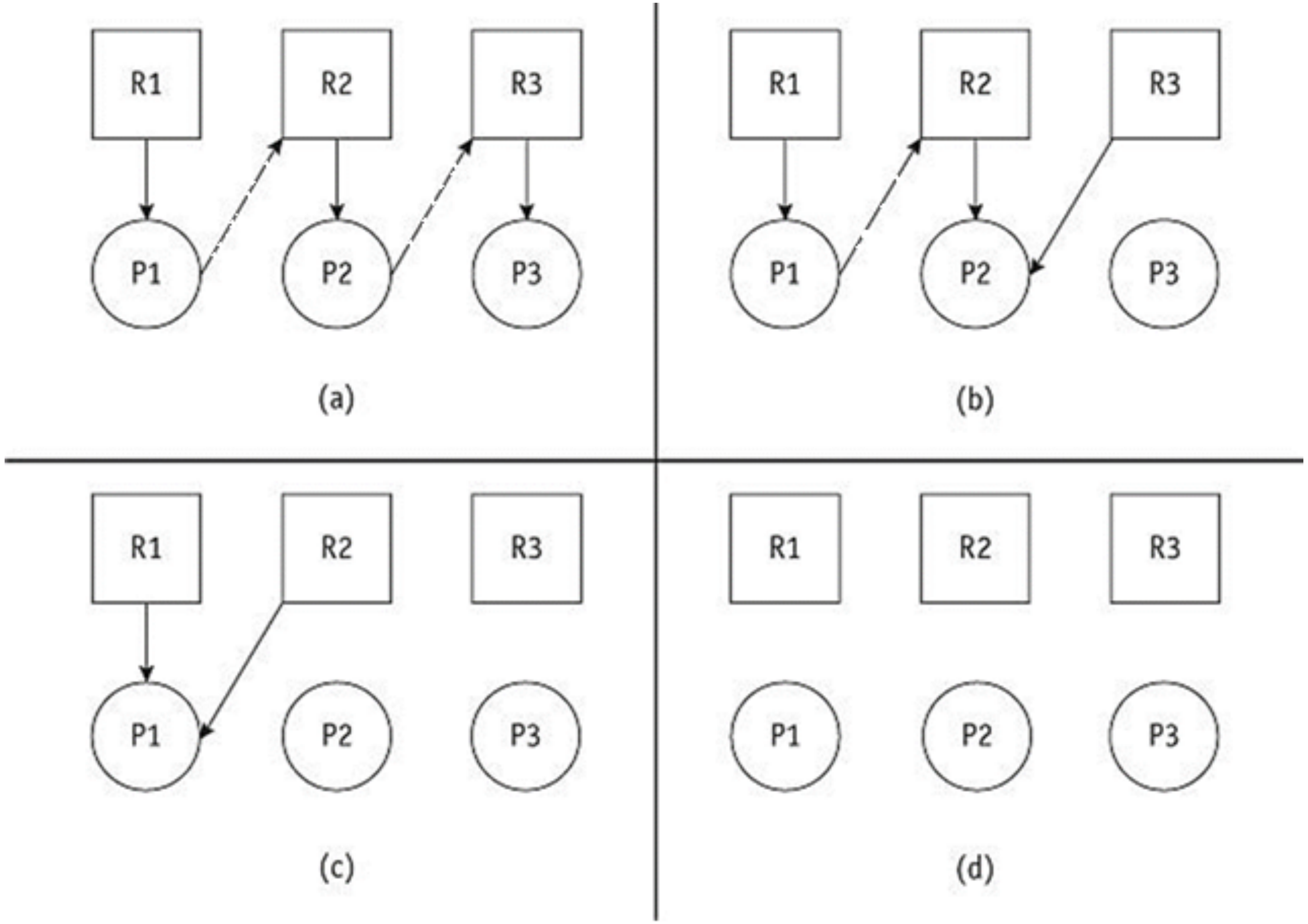
Problems with Bankers Algorithm

- Jobs must state maximum number needed resources
- Requires constant number of total resources for each class
- Number of jobs must remain fixed (Not possible with interactive systems)
- Possible high overhead cost incurred
- Resources not well utilized
  - Algorithm assumes worst case
  - Scheduling suffers
    - Result of poor utilization
    - Jobs kept waiting for resource allocation

Detection

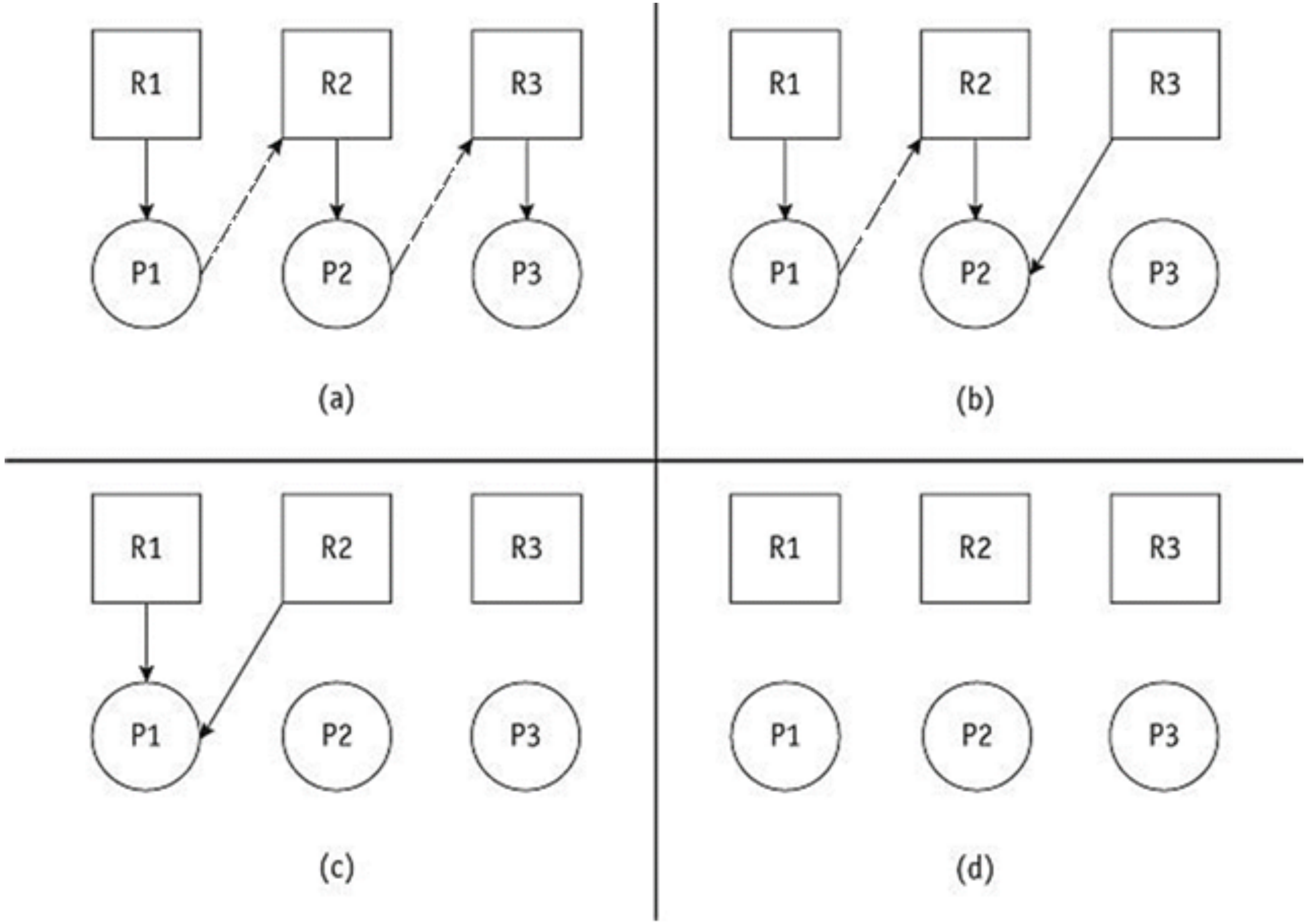
- Detection: build directed resource graphs
  - Look for cycles
- Algorithm detecting circularity
  - Executed whenever appropriate
- Detection algorithm
  - Remove process using current resource and not waiting for one
  - Remove process waiting for one resource class
    - Not fully allocated
- Go back to step 1
  - Repeat steps 1 and 2 until all connecting lines removed

Detection


R1
R2
R3
P1
P2
P3
(a)
R1
R2
R3
P1
P2
P3
(b)
R1
R2
R3
P1
P2
P3
(c)
R1
R2
R3
P1
P2
P3
(d)

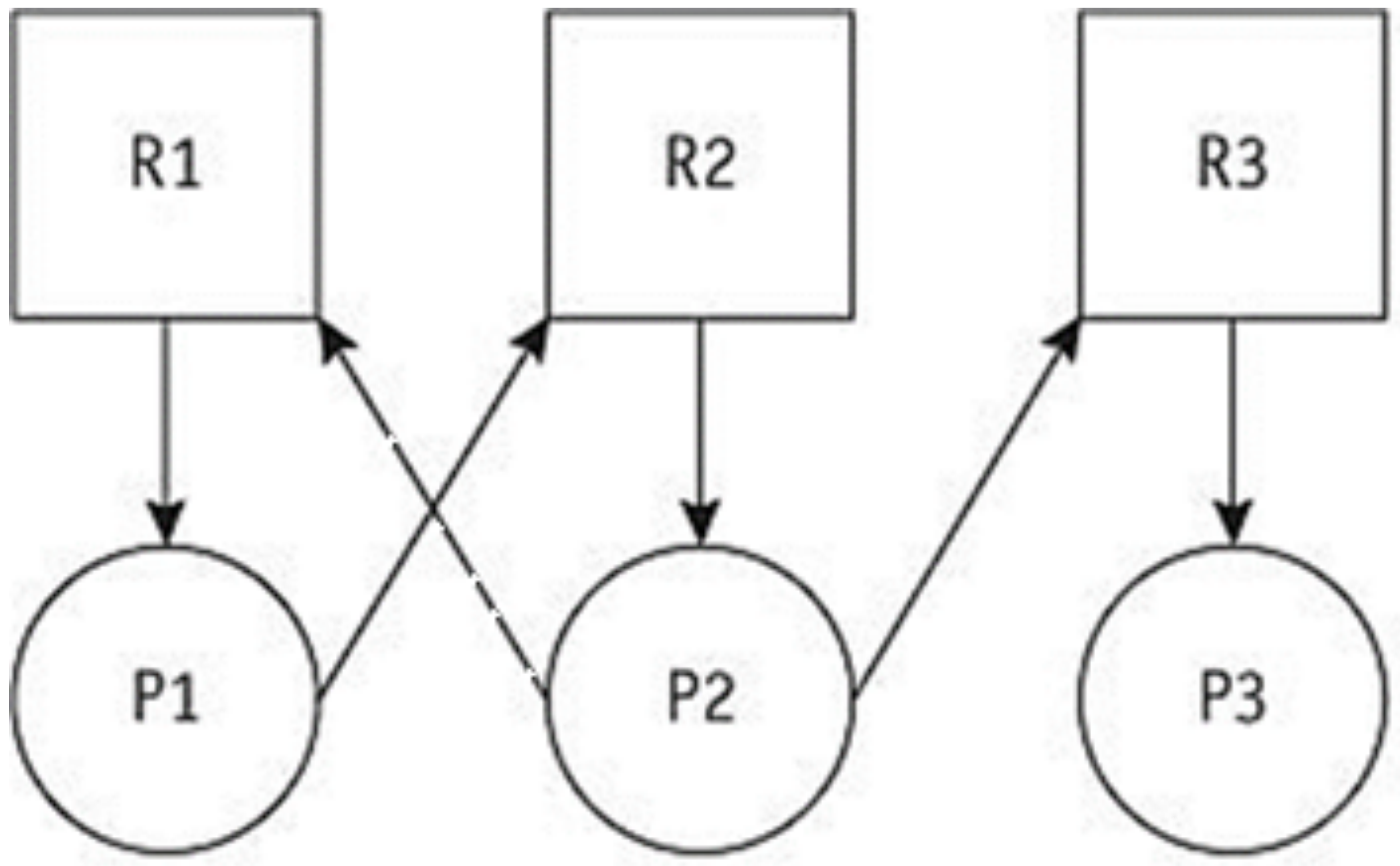
Detection

No Circular Dependency



R1
R2
R3
P1
P2
P3
(a)
R1
R2
R3
P1
P2
P3
(b)
R1
R2
R3
P1
P2
P3
(c)
R1
R2
R3
P1
P2
P3
(d)

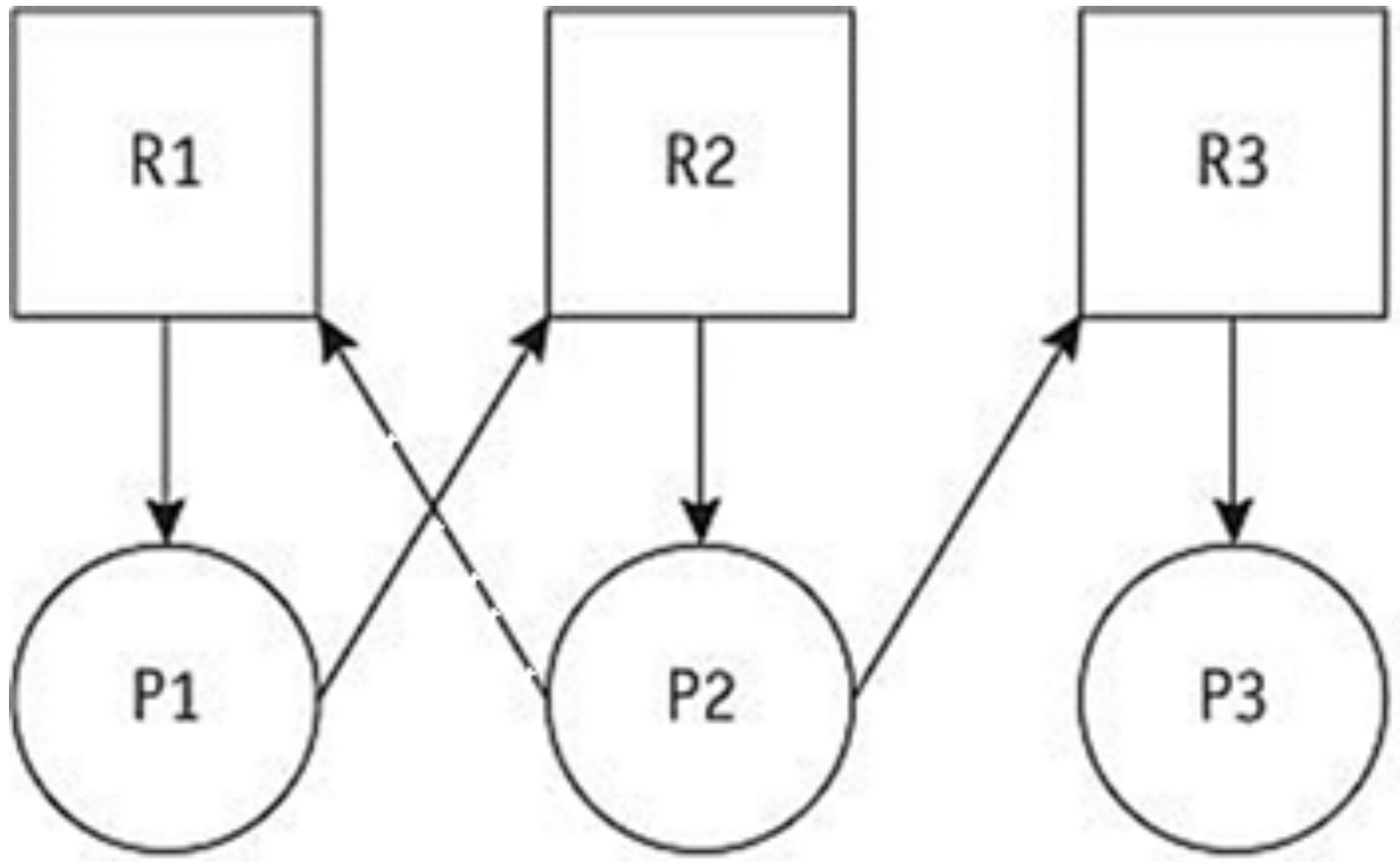
Detect?



(a)

Detect?

✓ Circular Dependency Exists



(a)