Anthony Ventresque

anthony.ventresque@ucd.ie

Operating Systems

COMP30640

Deadlock and Starvation (I)

School of Computer Science, UCD

Scoil na Ríomheolaíochta, UCD

Announcements

- ***Extra lab session***:
  - Friday 2nd November: 10 a.m. – 11 a.m.
  - H1.51 Science Hub

Last week…

Conditions for True Solution to CS Problem (Dijkstra)

1. ***Mutual exclusion***
   - One process <u>at most</u> inside the CS at any time
2. ***Progress***
   - A process in execution out of a CS cannot prevent other processes from entering it
   - If several processes are attempting to enter a CS simultaneously the decision on which one goes in cannot be indefinitely postponed
   - A process may not remain in its CS indefinitely (neither terminate inside it)
3. ***Bounded waiting*** (no starvation)
   - A process attempting to enter its CS will eventually do so

Notes:

- These are ***necessary and sufficient*** conditions, provided that basic operations are atomic
- No assumptions are made about: number of processes, relative speed of processes, or underlying hardware

Last Week…

Mechanisms for Implementing ME in a CS

Three basic mechanisms:

1. Semaphores
   – Simple, but hard to program with (low level)
2. Monitors
   – More abstract, higher level mechanism (language support)
3. Messages
   – Very flexible and simple method of interprocess communication (IPC) & synchronisation

Last Week···

Semaphores

- A CS may be protected by a semaphore → we may implement ME by means of semaphores; example:
  - Initialise S = 1
  - To enter the CS, execute ***P*** on its semaphore
  - When leaving the CS, execute ***V*** on its semaphore
  - Therefore, two or more processes sharing a CS and this semaphore achieve ME by executing

```
P(S);
   CS
V(S);
```

Last Week···

Monitors

```
monitor pr_co {
    int count;
    condition full_s,empty_s;

    void put(msg) {
        if(count==N) wait(empty_s);
        put_message(msg);
        count++;
        if(count==1) signal(full_s);
    }

    msg get() {
        if(count==0) wait(full_s);
        msg=get_message();
        count--;
        if(count==N-1) signal(empty_s);
    }
}
```

Producer

```
while(true) {
    msg=produce_message();
    pr_co.put(msg);
}
```

Consumer

```
while(true) {
    msg=pr_co.get();
    consume_message(msg);
}
```

- Simplest possible consumer and producer code
- Monitor takes care of any issues, not producer or consumer

Outline

- Deadlock and Starvation
- Four condition for deadlocks
- Resource-allocation Graphs (RAGs)

Take home message:

Deadlock (2+ processes waiting indefinitely for an event that can be caused only by a waiting process) can occur only if four necessary conditions hold simultaneously: mutual exclusion, hold and wait, no preemption and circular wait.

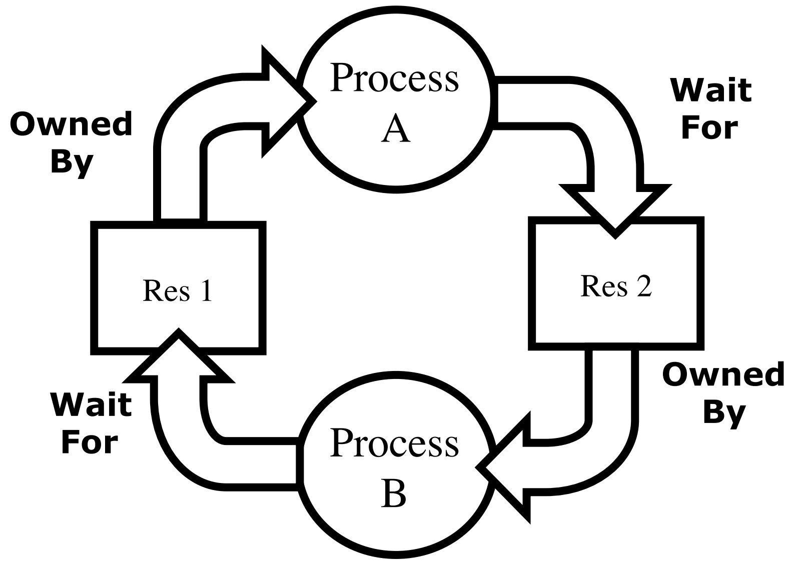
Deadlock and Starvation

- Mutual exclusion (ME) mechanisms for synchronisation guarantee that processes do not clash when using ***shared resources***
  - However, we saw that severe problems may still arise

- Definitions:
  - ***Deadlock*****:** a set of processes is in a deadlock state when every process in the set is blocked forever, waiting for the availability of resources held by other processes in the set
  - ***Starvation (indefinite postponement)*** occurs when a process waits for resources that periodically become available, but are never allocated to that process due to some ***scheduling policy***

Starvation vs. Deadlock

- Starvation: process/thread waits indefinitely
  - Example, low-priority process/thread waiting for resources constantly in use by high-priority process/threads
- Deadlock: circular waiting for resources
  - Process A owns Res 1 and is waiting for Res 2
  - Process B owns Res 2 and is waiting for Res 1



- Deadlock ⇒ Starvation but not vice versa
  - Starvation can end (but doesn't have to)
  - Deadlock can't end without external intervention

Resources

Processes request and are assigned resources; two types:

Reusable resources

- Those used by only one process at a time, and not depleted by that use
- After their use, they are released for reuse by other processes
- *Examples:* processors, main and secondary memory, devices, data structures such as databases and semaphores

Consumable resources

- Those created (produced) by one process and destroyed (consumed) by another
- Unbounded number of instances
- No need to release them
- *Examples:* interrupts, signals, messages

Deadlock and starvation are possible with both types of resources

Example 1 (Reusable Resource)

- Two processes P1 and P2 requesting memory allocation
- Assume that the available memory space is 200MB, and that the processes execute the following pseudocode:

P_1

```
...
Request 80MB
...
Request 60MB
```

P_2

```
...
Request 70MB
...
Request 80MB
```

- The processes are correctly designed, since neither requests more than the total space in the system
- <u>However</u>: deadlock occurs if both processes have gone through their first requests and progress to their second requests (without having released the memory initially allocated)

Example 2 (Consumable Resource)

- Consumable resource: message (we may send and receive as many as we want: they are produced and consumed)
- Assume that the receive operation is blocking (i.e. not asynchronous), and that two processes execute the pseudocode below:

P_1

```
...
receive(mbox_2,M);
...
send(mbox_1,N);
```

P_2

```
...
receive(mbox_1,N);
...
send(mbox_2,M);
```

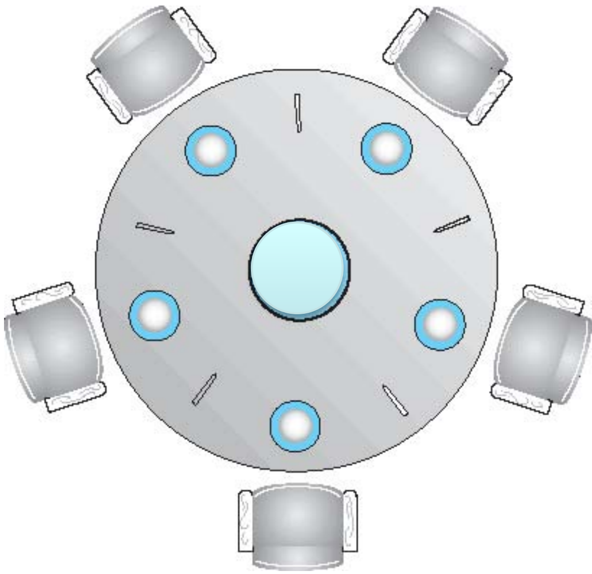
- Deadlock occurs, because each process requests a resource held by the other process, while at the same time holding a resource requested by the other process

Dining Philosophers

- Classic example proposed by Dijkstra (1971) to illustrate deadlock & starvation
- 5 philosophers living together
- Their life consists of two states: thinking or eating
- They have a position assigned at a round table on which there are 5 plates of noodles and 5 chopsticks
- A philosopher wishing to eat takes ***both chopsticks*** on the sides of their plate, and eats noodles
- No two philosophers can share a chopstick simultaneously

Philosopher = Process

Chopstick = Shared resource



Dining Philosophers: Getting Around Deadlock

- *Idea:* Make the philosophers release their left chopstick if after having grabbed it they detect that the right chopstick is in use
  - After waiting for some ***fixed time*** the philosopher would try again to grab both chopsticks
- *Issue*: No deadlock, but ***starvation*** is possible
  - If all philosophers start the algorithm simultaneously, no philosopher ever grabs both chopsticks
  - The philosophers are caught in an endless cycle
- *Solution*: Make waiting time ***random***

Dining Philosophers: Other Possible Solutions

Many ad-hoc solutions to deadlock and starvation are possible:

1. *Limit the number of philosophers* at the table
   - Allow at most four philosophers at a time at the table
   - Pass a token around the table so that only the philosopher holding the token can eat (i.e. enforce a sequence)
2. *Asymmetric solution*:
   - Even philosophers try left chopstick first
   - Odd philosophers try right chopstick first
3. Use five *counting semaphores*, each counting the number of available chopsticks per philosopher (careful initialisation needed)
4. etc

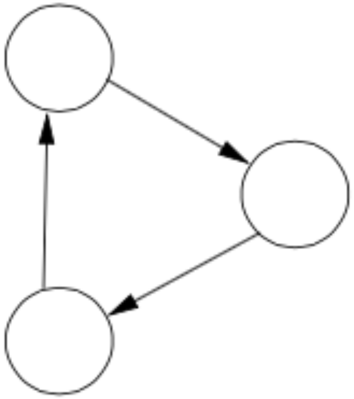
- Formalisation is needed to handle the complexity of the problem

Resource-allocation Graph

- Deadlocks can be described more formally in terms of a directed graph called Resource-Allocation Graph (RAG)

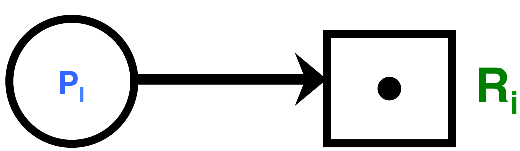
- A RAG completely describes state of a system in terms of
  1. What resources are allocated to what processes
  2. What processes are waiting for what resources

- A usual directed graph is just a set of vertices (i.e., nodes) plus set of directed edges (i.e., arrows) connecting the vertices; example:

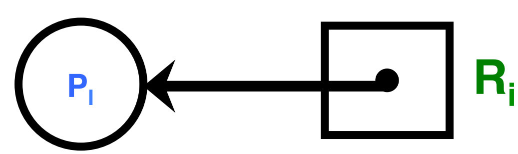


Resource-allocation Graph

- A RAG is a ***special directed graph***, since both vertices and directed edges are partitioned into ***two sets***
- Vertices:
  1. $P = \{P_1, \dots, P_n\}$, all active processes in the system:
  2. $R = \{R_1, \dots, R_m\}$, all resource types in the system:
     - also: number of • inside □ is the number of instances of the resource (two displays, three printers. . .)
- Directed edges:
  1. $P_i \rightarrow R_j$: request edge
  2. $R_j \rightarrow P_i$: assignment edge



Request



Assignment

Example

- Two processes P_1 and P_2, two shared resources R_1 and R_2 (each resource has one instance only)
- Assume the following scenario:

P_1

```
0: ...
while(true) {
   1: request(R1);
   2: ...
   3: request(R2);
   4: ...
      release(R2);
   5: ...
      release(R1);
   0: ...
}
```

P_2

```
0: ...
while(true) {
   1: request(R2);
   2: ...
   3: request(R1);
   4: ...
      release(R1);
   5: ...
      release(R2);
   0: ...
}
```

Example: System States

State	Situation of P_1
0	Holds no resources
1	Holds none, requests R_1
2	Holds R_1
3	Holds R_1, requests R_2
4	Holds R_1 and R_2
5	Holds R_1, R_2 released

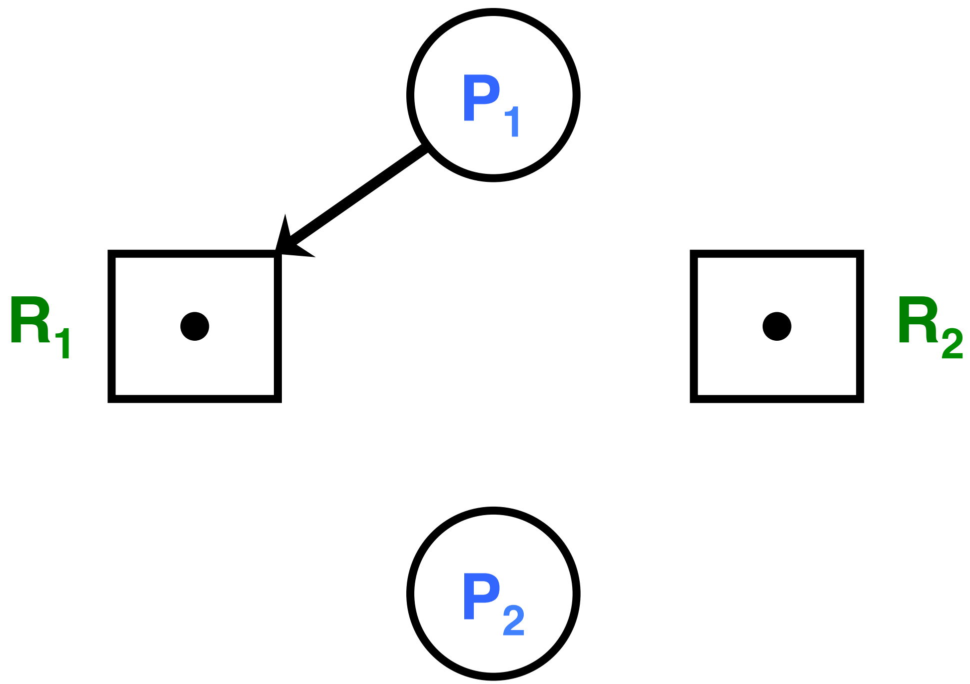
State	Situation of P_2
0	Holds no resources
1	Holds none, requests R_2
2	Holds R_2
3	Holds R_2, requests R_1
4	Holds R_2 and R_1
5	Holds R_2, R_1 released

- Each pair of possible states corresponds to one RAG
- Note: some states such as (4,4) are impossible in this example

Example: RAG

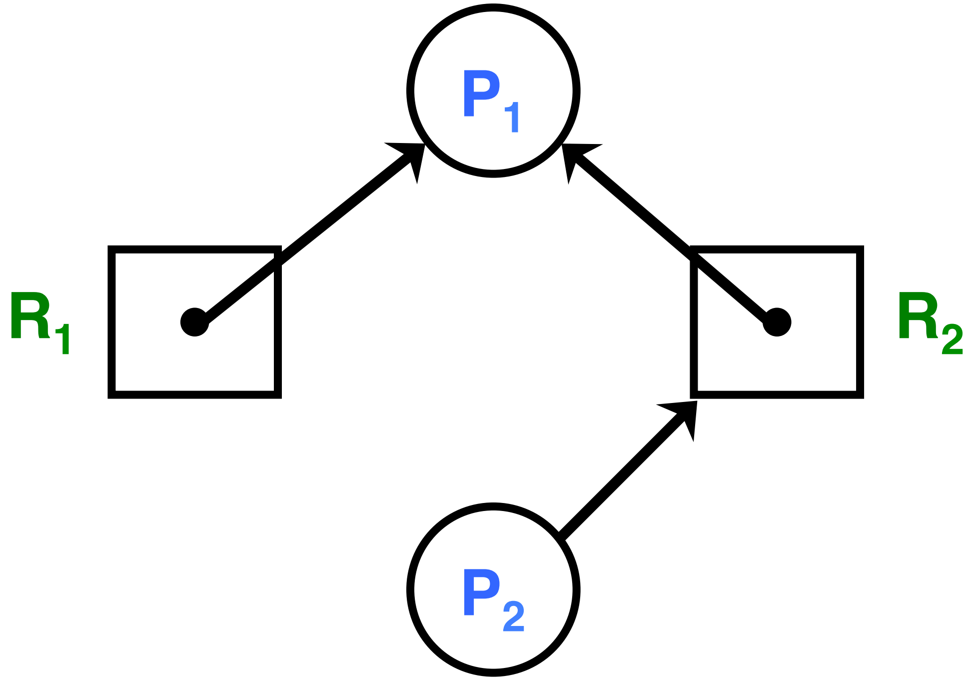
(State P_1, State P_2)

- (1,0): P_1 asks for R_1 which is free



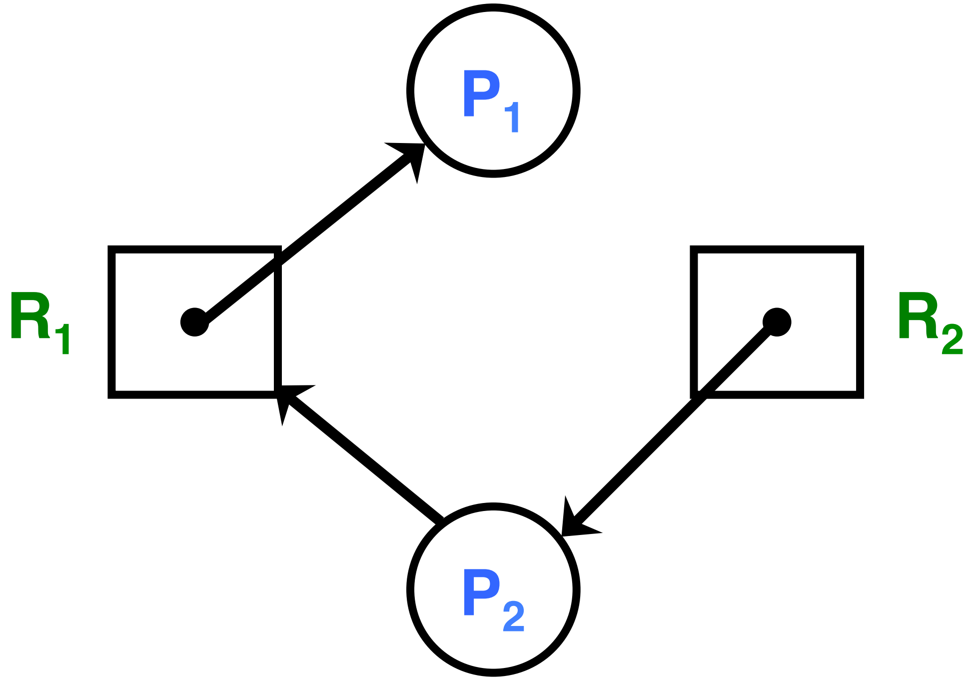
Example: RAG (2)

- (State P_1, State P_2)
- (4,1): P_2 blocked (waiting for P_1 to release R_2)



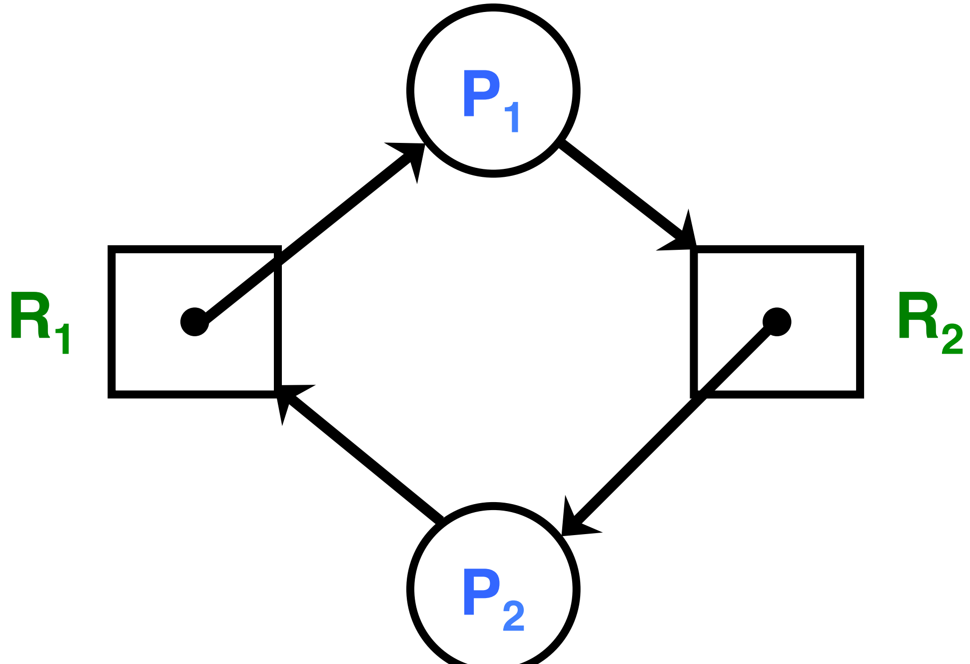
Example: RAG (3)

- (State P_1, State P_2)
- (2,3): Technically not a deadlock, but next instruction in P_1 (in the example) leads inevitably to a deadlock



Example: RAG (4)

- (State P_1, State P_2)
- (3,3): deadlock



Four Necessary Conditions for Deadlock

1. **Mutual exclusion** (limited access)
   - At least one resource may be acquired exclusively by only one process at a time
2. **Hold-and-wait** (wait-for)
   - Processes may ask for resources while holding other resources
3. **No preemption**
   - Once allocated, resources are released only voluntarily by the process holding the resource, after process is finished with it
4. **Circular chain of request** (circular-wait)
   - Two or more processes locked in a circular chain in which each process is waiting for one or more resources that the next process in the chain is holding
   - Equivalent to a cycle in the RAG

Note: ***These are not sufficient conditions***

Cycles in RAG and Deadlock

Possibilities:

1. There are no cycles in the RAG: then there is no deadlock
   - A cycle is ***a necessary condition*** for a deadlock

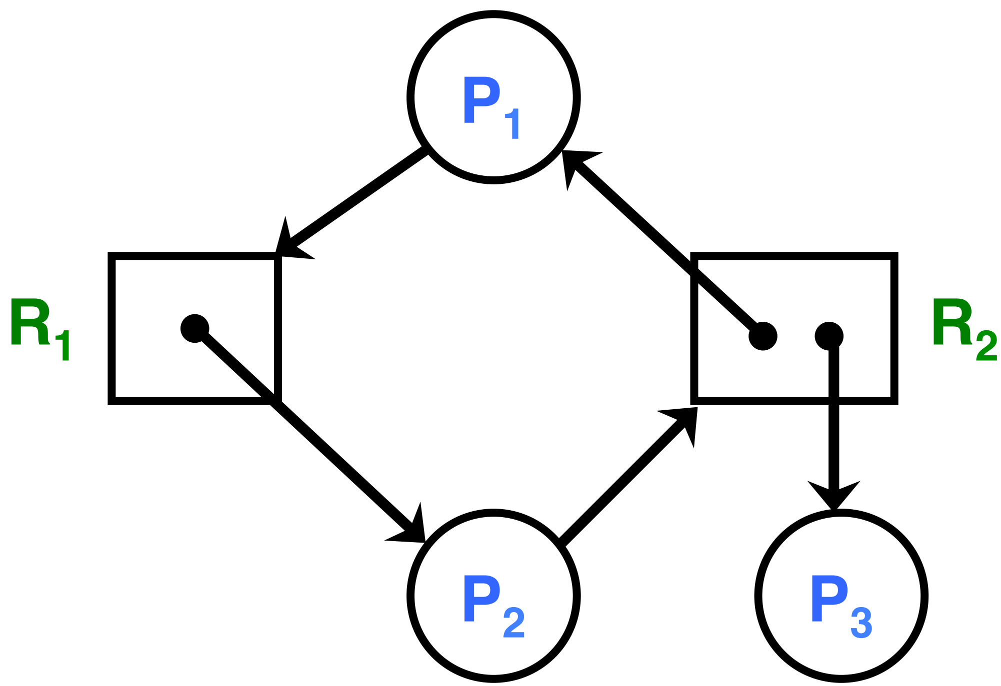
2. There is a cycle in the RAG and
   A. There is ***a single instance*** of each resource in the cycle: then there is a deadlock
   B. There are ***several instances*** of at least one resource in the cycle: then there may or there may not be a deadlock

- In general, a cycle is ***not a sufficient condition*** for a deadlock
- An exception is case A) above

Examples: RAG Cycles and Multiples Resource Instances

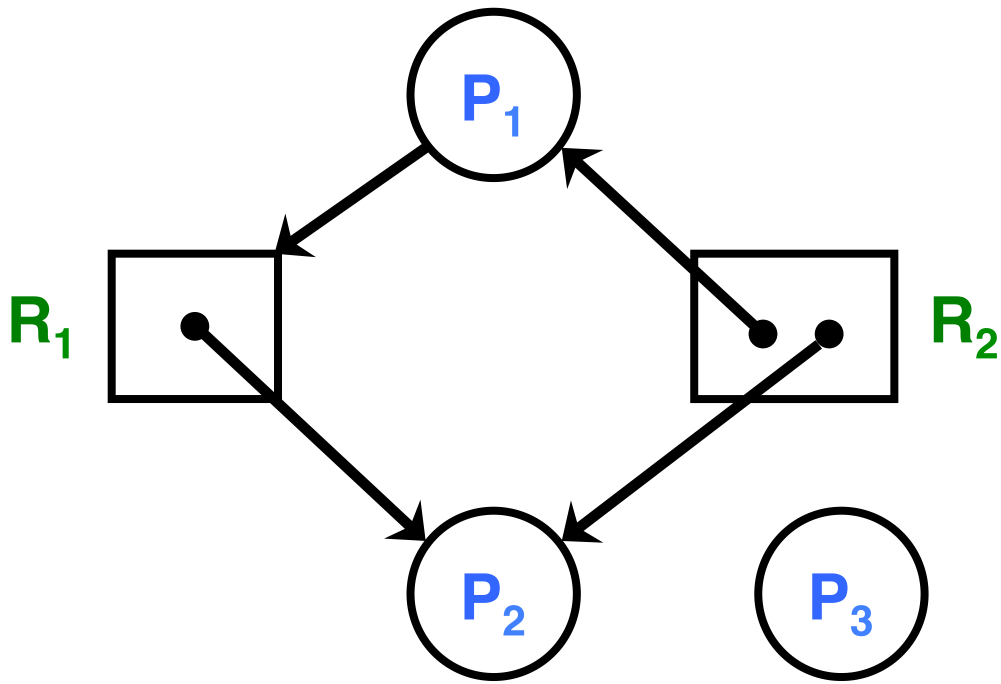
- **Note:** There are two instances of R_2
- Cycle but no deadlock

$$P_1 \rightarrow R_1 \rightarrow P_2 \rightarrow R_2 \rightarrow P_1$$



Examples: RAG Cycles and Multiple Resource Instances (2)

- **Note:** There are two instances of R_2
- Cycle broken if P_3 releases R_2

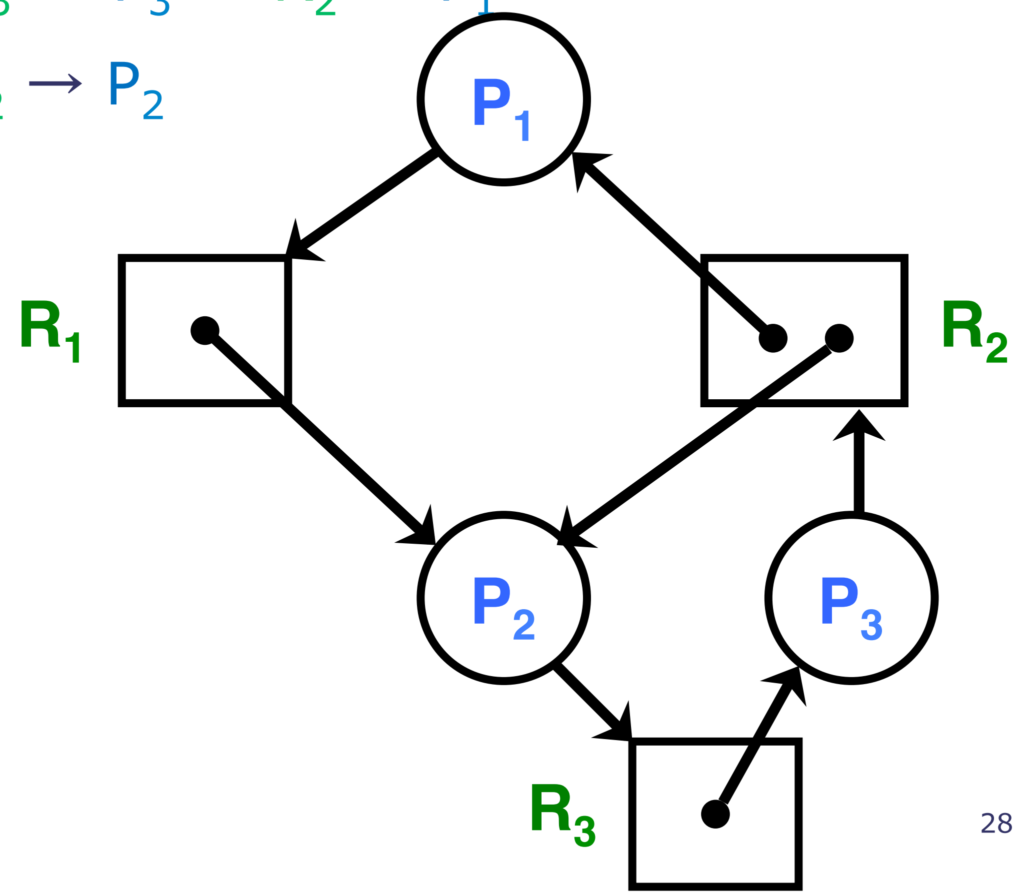


Examples: RAG Cycles and multiple Resource Instances (3)

Cycle(s) & deadlock:

$P_1 \rightarrow R_1 \rightarrow P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$

$P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_2$



Conclusion

- Starvation vs. Deadlock
  - Starvation: thread/process waits indefinitely
  - Deadlock: circular waiting for resources

- Four conditions for deadlocks
  - **Mutual exclusion**
    - Only one thread/process at a time can use a resource
  - **Hold and wait**
    - Thread/process holding at least one resource is waiting to acquire additional resources held by other threads/processes
  - **No preemption**
    - Resources are released only voluntarily by the threads/processes
  - **Circular wait**
    - $\exists$ set $\{T_1, \ldots, T_n\}$ of threads/processes with a cyclic waiting pattern