```
#include <stdlib.h>
#include <string.h>
#include <ctype.h>

#define MAXPAROLA 30
#define MAXRIGA 80

int main(int argc, char *argv[])
{
  int freq[MAXPAROLA] ; /* vettore di contatori
  delle frequenze delle lunghezze delle parole */
  char riga[MAXRIGA] ;
  int i, inizio, lunghezza ;
  FILE * f ;

  for(i=0; i<MAXPAROLA; i++)
    freq[i]=0 ;

  if(argc != 2)
  {
    fprintf(stderr, "ERRORE, serve un parametro con il nome del file\n");
    exit(1);
  }
  f = fopen(argv[1], "rt") ;
  if(f==NULL)
  {
    fprintf(stderr, "ERRORE, impossibile aprire il file %s\n", argv[1]);
    exit(1);
  }

  while( fgets( riga, MAXRIGA, f ) != NULL )
```

# Deadlock

## Definition and modeling

Stefano Quer, Pietro Laface, and Stefano Scanzio
Dipartimento di Automatica e Informatica
Politecnico di Torino
skenz.it/os stefano.scanzio@polito.it

# Deadlock

- Condition for **deadlock**
  - A P/T requires an unavailable resource, it enters a waiting state, and it waits forever
- Deadlock consists in
  - A set of P/T all awaiting the occurrence of an event that can only be caused by another process in the same set
- Deadlock **implies** starvation, **not** the opposite
  - The starvation of a P/T implies that this P/T waits indefinitely, but the other P/T can proceed in the usual way (without being in deadlock)
  - All P/T in deadlock are in starvation

# The Deadlock Problem

- A set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set.
  - Example: $P_1$ and $P_2$
    - each of them holds a pen drive and
    - needs another one.
  - Solution with 2 semaphores A and B, initialized to 1

| $P_1$ | $P_2$ |
|---|---|
| wait (A) | wait(B) |
| wait (B) | wait(A) |

# Necessary conditions for occurrence of a deadlock

| **Conditions** | **Description** |
|---|---|
| Mutual exclusion | Only one process at a time can use a **not sharable** resource |
| Hold and wait | A process **holding** at least one resource is allowed to **wait** for acquiring additional resources held by other processes |
| No preemption | A resource can be released only voluntarily by the process holding it, **cannot** be **preempted** by the system. |
| Circular wait | A set of waiting processes $\{P_1, P_2, \ldots, P_n\}$ such that<br>$P_1$ is **waiting** for a resource that is held by $P_2$,<br>$P_2$ is **waiting** for a resource that is held by $P_3$,<br>…,<br>and $P_n$ is **waiting** for a resource that is held by $P_1$ |

All must occur simultaneously to have a deadlock

Necessary but not sufficient conditions.
They are distinct but not independent (e.g., 4→2)

# Summary

- Deadlock modeling
- Management strategies
  - Ignore
  - A posteriori
    - Detect
    - Recovery
  - A priori
    - Prevent
    - Avoidance

**Ignore the problem assuming the probability of a deadlock in the system is very low**

- Method used by many operating systems, including Windows and Unix
- Less appropriate if concurrency and complexity of the system increase

This section 01

In case of deadlock

Section 02

Section 03

In case of possibility of deadlock

# Deadlock modeling

- **Resource allocation graph** G = (V, E)
  - Allows deadlock description and analysis
- The set of vertices V is composed of processes and resources
  - Process set P = $\{P_1, P_2, \ldots, P_n\}$
    - Processes are indistinguishable and in an indefinite number
    - Each process accesses a resource via a standard protocol consisting of
      - Request
      - Utilization
      - Release

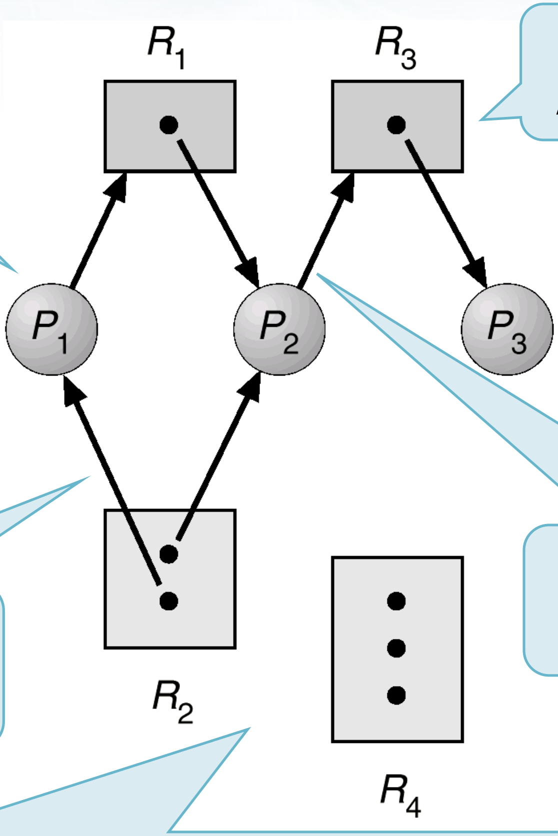
# Modeling

- System resource set R = $\{R_1, R_2, \ldots, R_m\}$
  - The resources are divided into classes (types)
  - Each resource type $R_j$ has $W_i$ instances
  - All instances of a class are **identical**: any instance satisfies a demand for that type of resource

If not, it would be necessary to reformulate the division into classes

- The set of edges E is composed of
  - Request edges
    - $P_i \rightarrow R_j$, i.e., from a process to a resource type
  - Assignment edge
    - $R_j \rightarrow P_i$, i.e., from a resource to a process

# Modeling



Vertices: Processes
$P_1$, $P_2$, $P_3$

Vertices: Resources
An instance of $R_1$ and $R_3$

$P_1$ holds $R_2$ and is waiting for $R_1$

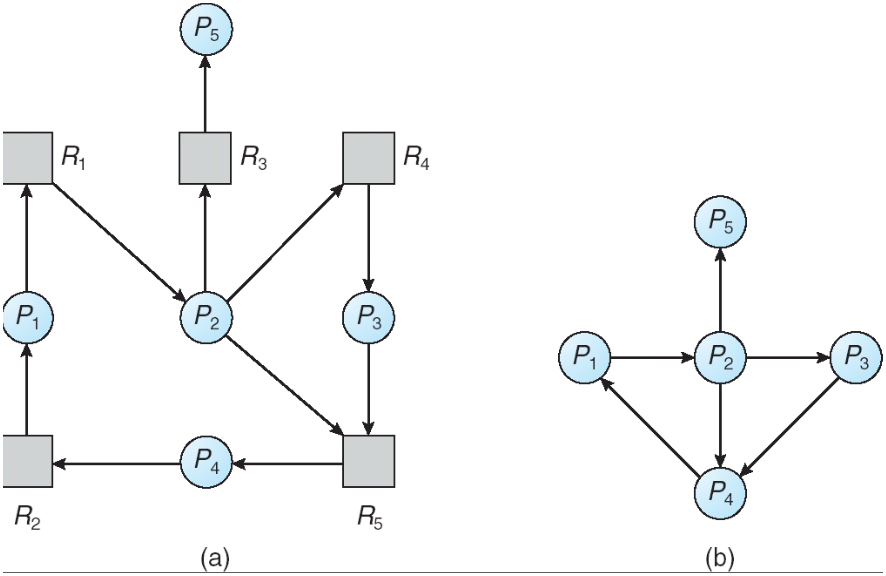
Request edge:
$P_2$ requests for a $R_3$ type resource

Assignment edge:
$P_1$ holds $R_2$

Vertices: Resources
$R_2$ and $R_4$ with 2 and 3 instances, respectively

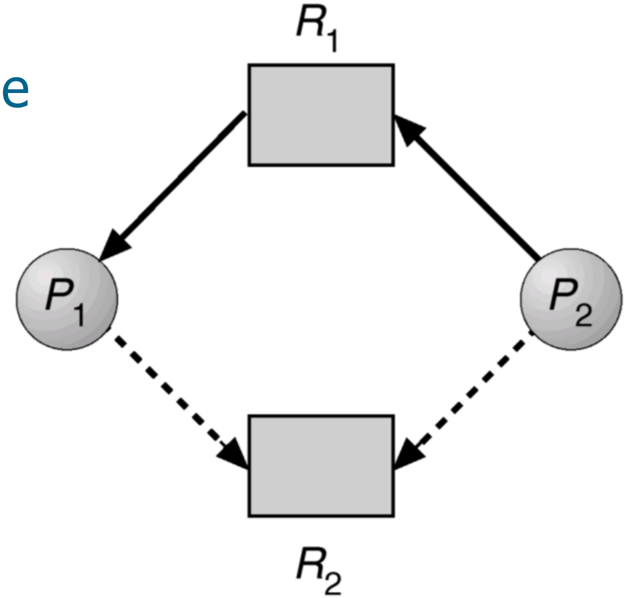
# Modeling

- A **resource allocation graph** can be sometime simplified in a **wait-for graph** by
  - deleting the resource vertices
  - creating the edges between the remaining vertices
- Use and consideration similar to the resource allocation graph



# Modeling

- Sometimes it is useful to extend the resource-allocation graph to a **claim graph** by
  - adding a claim edge: $P_i \dashrightarrow R_j$, indicates that process $P_j$ **can ask resource $R_j$ in the future**
  - A claim edge is represented by dashed line



# Detection and recovery techniques

- The system is allowed to enter in a deadlock state, to then intervene.
- Algorithm in two steps
  - **Deadlock detection**
    - The system performs a deadlock detection algorithm
  - **Recovery from deadlock**
    - If deadlock has been detected, a recovery action is performed

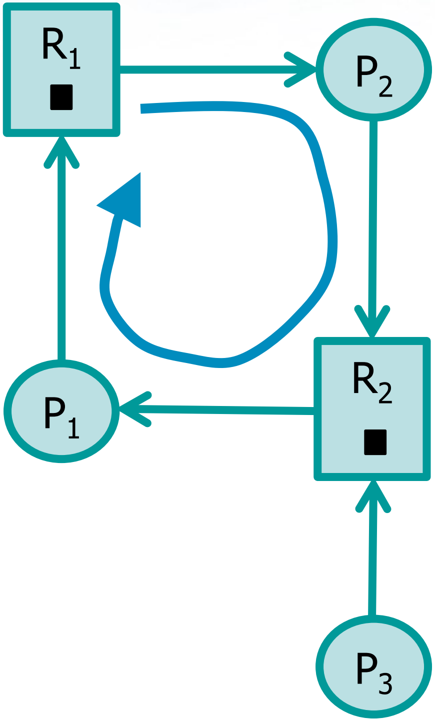
# Detection: strategies

- Given an allocation graph, deadlock can be detected by checking for cycles
  - If the graph contains no cycles, then there is no deadlock
  - If the graph contains one or more cycles then
    - Deadlock exist if each type of resource has a **single instance**
    - Deadlock is possible if the are **several instances** per resource type
      - The presence of cycles is necessary but not sufficient condition in the case of multiple instances per resource type

For multiple instances see the Banker's Algorithm

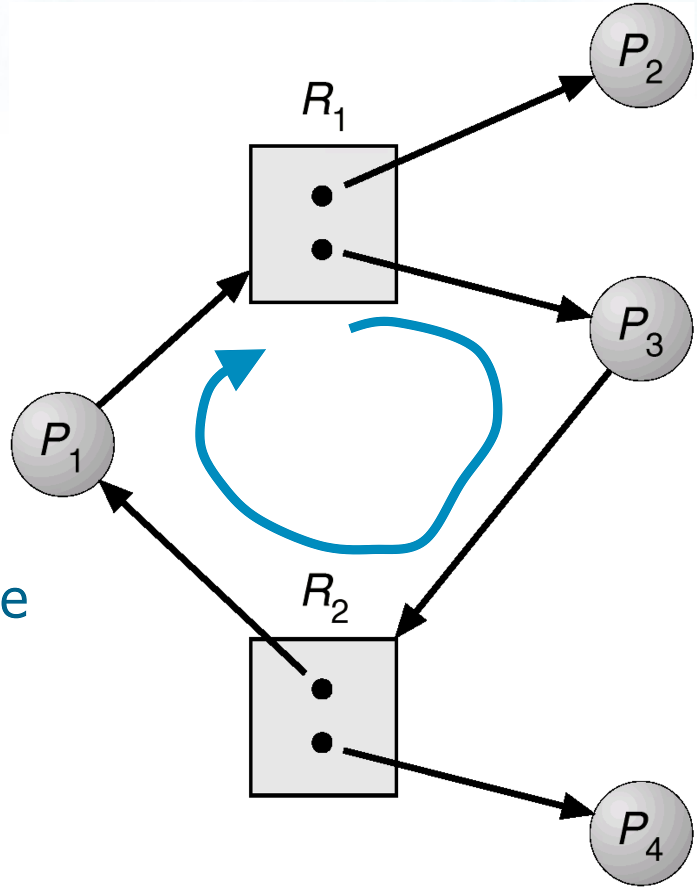
# Example

- Processes
  - $P_1$, $P_2$, $P_3$
- Resources
  - $R_1$ and $R_2$ with a single instance
- A cycle exists
- Deadlock
  - $P_1$ waits for $P_2$
  - $P_2$ waits for $P_1$



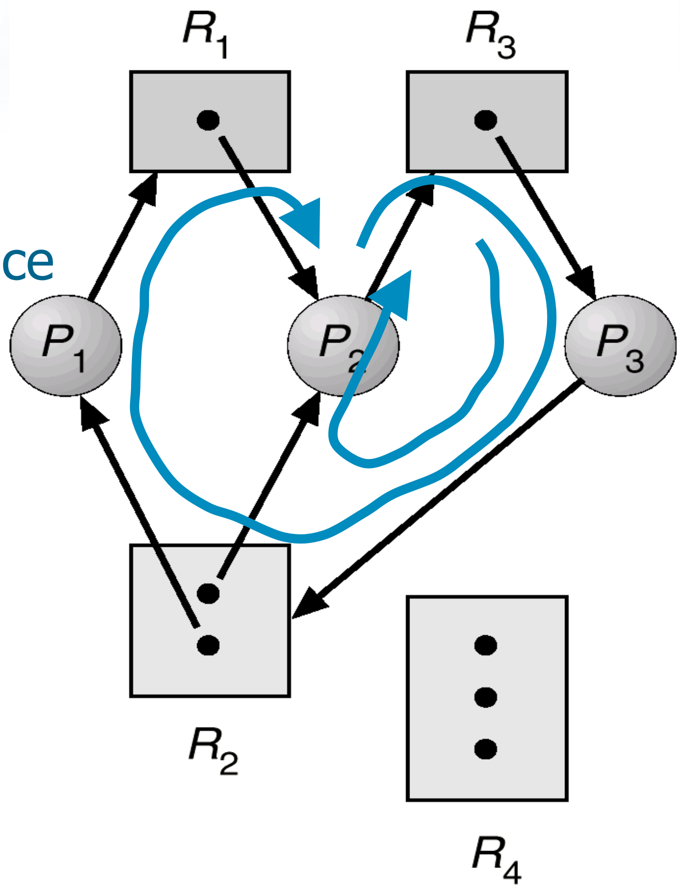
# Example

- Processes
  - $P_1$, $P_2$, $P_3$, $P_4$
- Resources
  - $R_1$ and $R_2$ with two instances
- A cycle exists
- No deadlock
  - $P_2$ and $P_4$ can terminate
  - $P_1$ can acquire $R_1$ and terminate
  - $P_3$ can acquire $R_2$ and terminate



# Example

- Processes
  - $P_1$, $P_2$, $P_3$
- Resources
  - $R_1$ and $R_3$ with an instance
  - $R_2$ with two instances
  - $R_4$ with three instances
- Two cycles exist
- Deadlock
  - $P_1$ waits for $R_1$
  - $P_2$ waits for $R_3$
  - $P_3$ waits for $R_2$



# Detection: costs

- The detection phase has the high computational cost
  - An algorithm to detect a cycle in a graph is required
    - The presence of cycles can be verified by a visit in depth
    - A graph is acyclic if a visit in depth does not meet arcs labeled "backward" directed to gray vertices
      - If you reach a gray vertex, i.e., you cross a backward arc, you have a cycle
    - The computational cost of this operation is equal to
      - $\Theta(|V|+|E|)$ for representations with adjacency list
      - $\Theta(|V|^2)$ for representations with adjacency matrix

# Detection: costs

- When detection is performed?
  - Every time a process makes a request not immediately satisfied
  - At fixed time intervals, e.g., every 30 minutes
  - At variable intervals of time, e.g., when the CPU usage falls below a given threshold

# Recovery

- Different strategies are possible for deadlock recovery
  - Terminate all **processes** in deadlock
  - Terminate a **process** at a time, among the ones in deadlock
    - Select a victim process, re-check the deadlock condition, and possibly iterate
  - Select a deadlocked process and
    - preempt the (some) **resources** it holds, resource allocation graph imposing a rollback, re-check the deadlock condition, and possibly iterate
  - Remove specific **arcs** from the resource allocation graph to eliminate cycles
    - **Holding arcs** or **waiting arcs**

# Recovery

| **Strategy** | **Description** |
|---|---|
| Terminate all deadlocked processes | **Complexity**: **low**, but easy to cause inconsistencies on databases<br>**Cost**: **much higher than it might be** strictly **necessary** |
| Terminate a process at a time among the ones in deadlock | **Complexity**: **high**, since it is necessary to select the victims with objective criteria (priority, current and future execution time, number of held resources, etc.)<br>**Cost**: **high**, after each termination must re-check the deadlock condition |
| Preempt the resources of a deadlocked process at a time | **Complexity**: rollback is necessary to return the selected process to a safe state<br>**Cost**: the victim process selection must aim at minimizing the preemption cost |

# Recovery

| Strategy | Description |
|---|---|
| Remove **holding** arcs (i.e., specific resources) | • **Complexity**: rollback is necessary to return the selected process to a safe state. The arc must be properly selected.<br>• **Cost**: the victim process selection must aim at minimizing the preemption cost<br>• Same as previous strategy |
| Remove **waiting** arcs | • **Complexity**: The arc must be properly selected.<br>• **Cost**: the victim must manage only the failure of a resource request (e.g., a malloc that returns with an error message). |

Best strategy

# Conclusions

- Detection and recovery operations are
  - logically complex
  - computationally expensive
- In any case, if a process requires many resources, starvation may occur
  - The same process is repeatedly chosen as the victim, incurring repeated rollbacks
    - To avoid starvation the victim selection algorithm should take into account the number of a process rollbacks