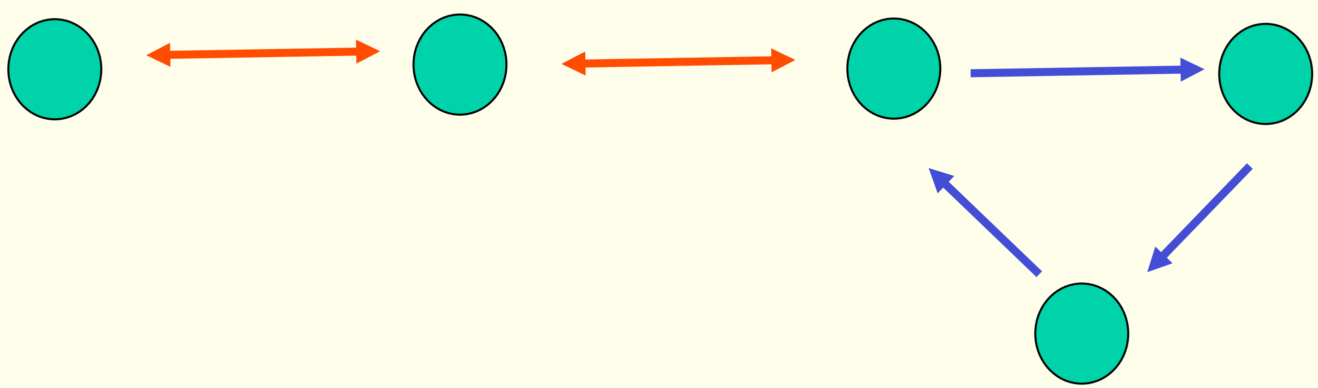
Concurrent & Distributed Systems

distributed co-ordination 2

distributed coordination

- **mutual exclusion**
- ***deadlocks***
- **election algorithms**

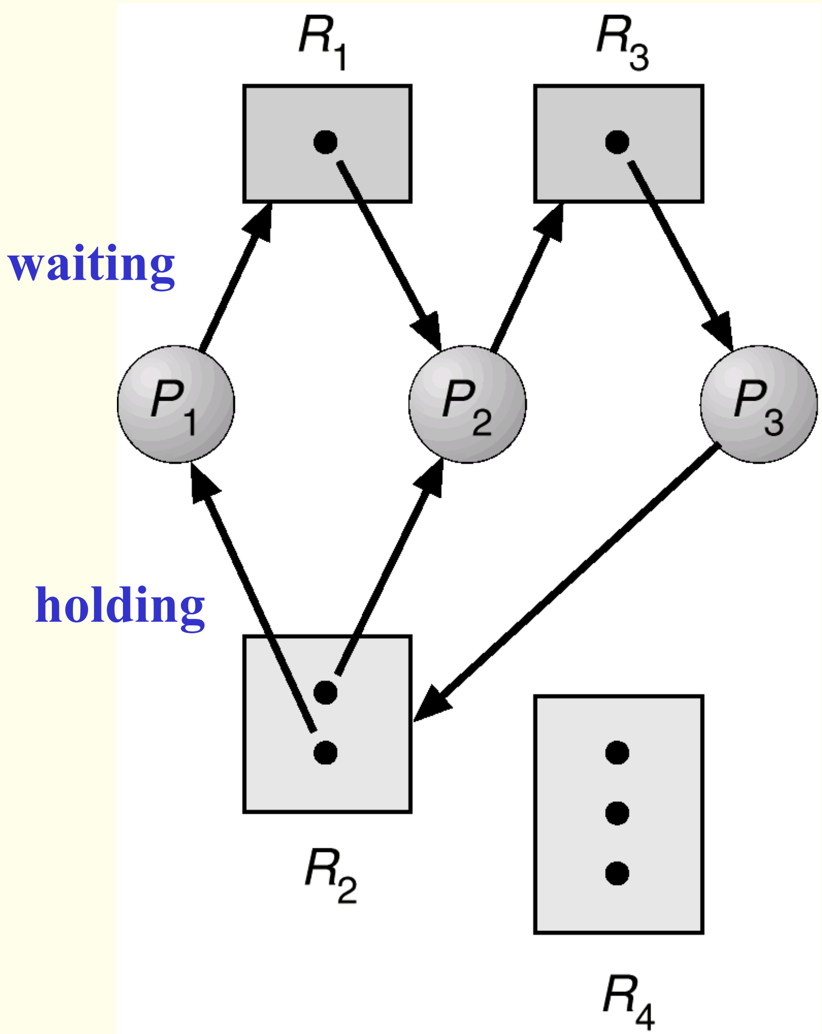


deadlock - a reminder

- ***all 4 of the following conditions are necessary conditions for deadlock:***

- ***mutual exclusion***
  - **at least 1 held resource must be non-sharable**
- ***hold and wait***
  - **at least 1 process is holding a resource, and waiting for another**
- ***no preemption***
  - **a process holding a resource cannot be pre-empted**
- ***circular wait***
  - **must be a loop with a process waiting on a resource, held by another process, which in turn is waiting on a resource, held by yet another process …**
  - **if this forms a loop, we have fulfilled 1 condition for deadlock.**

resource allocation graph with a deadlock



deadlock *prevention*

- prevent any of the four conditions for deadlock occurring. Techniques seen before earlier in the module can be applied, although they may need to be altered to give a “distributed form”.

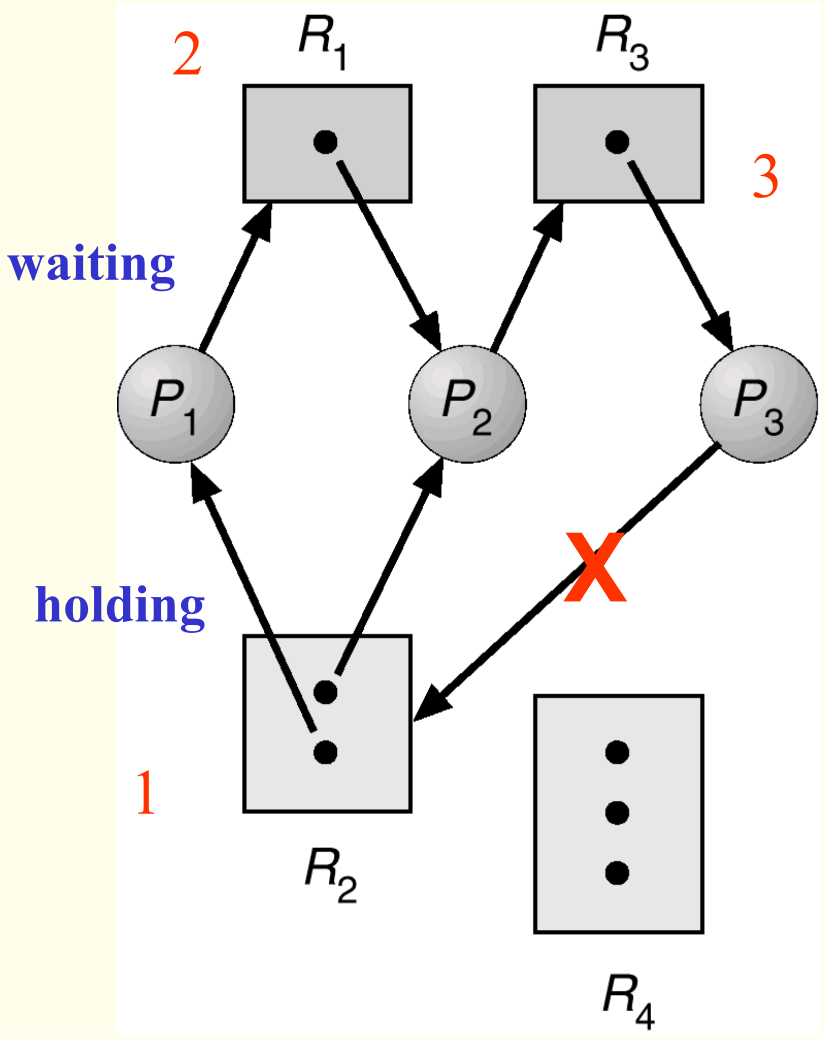
1. sharable resources remove the *mutual exclusion* condition
   – this is often not possible
2. resource allocation can be used to break the *hold and wait* condition
   – e.g. must be able to obtain all resources at once
   – or at least enough to allow some processing to take place

deadlock ***prevention***

3. implicit preemption can be used to break the *no preemption* condition
   - e.g. if a process has 1 resource but still awaits another – then preempt
   - not useful for, e.g. printers or objects

4. resource-ordering deadlock-prevention breaks the *circular wait* condition
   - define a *global* ordering among the system resources.
   - assign a unique number to all system resources.
   - a process may request a resource with unique number *i* , iff it is not holding a resource with a unique number greater than *i*.
   - simple to implement, and requires little overhead.

resource allocation graph with a deadlock



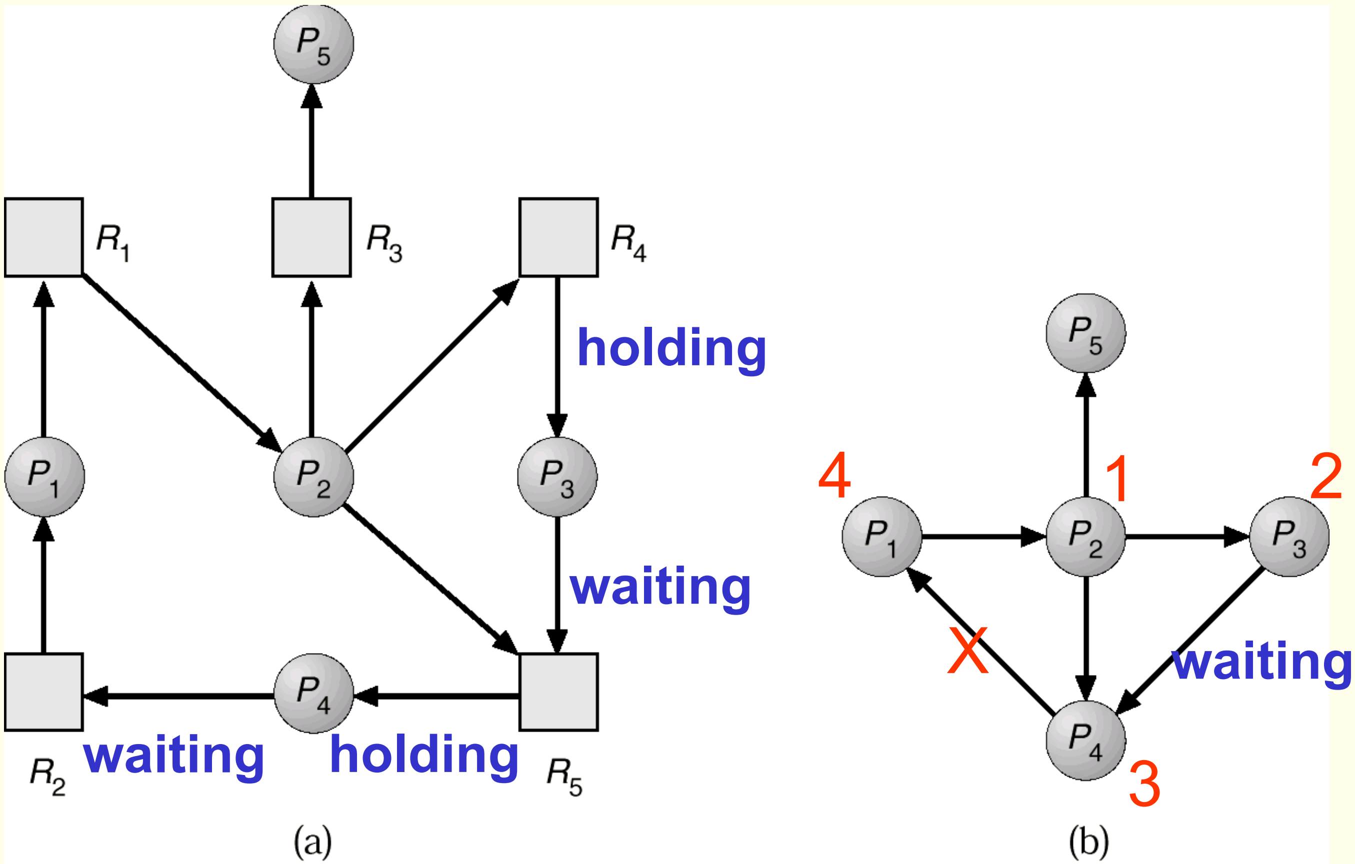
but there are other approaches for the distributed case ....

timestamped deadlock-prevention scheme

- the scheme prevents deadlocks.

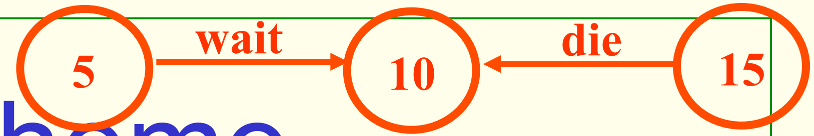
- here we only allow a process to wait for a **resource *under certain conditions***
- ***non*-preemptive**
- each process P_i is assigned a unique priority number
- priority numbers are used to decide whether a process P_i should wait for a process P_j; otherwise P_i is rolled back
- for every edge $P_i \rightarrow P_j$ in the wait-for graph, P_i has a higher priority than P_j
- thus a cycle cannot exist

resource allocation graph & wait-for graph



Resource-Allocation Graph **Corresponding wait-for graph**

Thus a cycle as shown on the right hand graph could not occur!

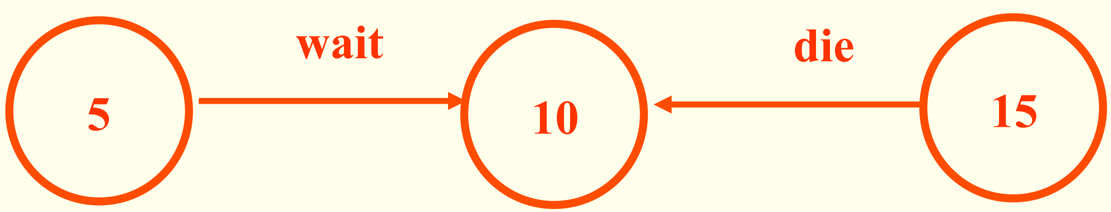


wait-die scheme

- however, processes with low priorities may suffer from *starvation* better to assign a **unique timestamp** when process is *created.* Same principle, but
  - non-preemptive
  - control of wait
- If P_i requests a resource currently held by P_j , P_i is **allowed to wait only if it has a <u>smaller</u> timestamp** than P_j (P_i is <u>**older**</u> than P_j). Otherwise, P_i is rolled back (dies).
- *Example*: Suppose that processes P_1 , P_2 , and P_3 have timestamps 5, 10, and 15 respectively.
  - if P_1 requests a resource held by P_2, then P_1 will wait.
  - If P_3 requests a resource held by P_2, then P_3 will be rolled back.

naming … an aside

	Wait-die	Wound-wait
timestamp	Lower … higher	Lower … higher
Process age	Older … younger	Older … younger



wound-wait scheme

- Based on a ***preemptive*** technique; counterpart to the wait-die system.
- If P_i requests a resource currently held by P_j , P_i is allowed to wait only **if it has a <u>larger</u> timestamp than does P_j** (P_i is **<u>younger</u>** than P_j). Otherwise P_j is rolled back (P_j is wounded by P_i).
- Example: Suppose that processes P_1 , P_2, and P_3 have timestamps 5, 10, and 15 respectively.
  - If P_1 requests a resource held by P_2, then the resource will be *preempted* from P_2 and P_2 will be rolled back.
  - If P_3 requests a resource held by P_2, then P_3 will wait.

comparison of the two schemes

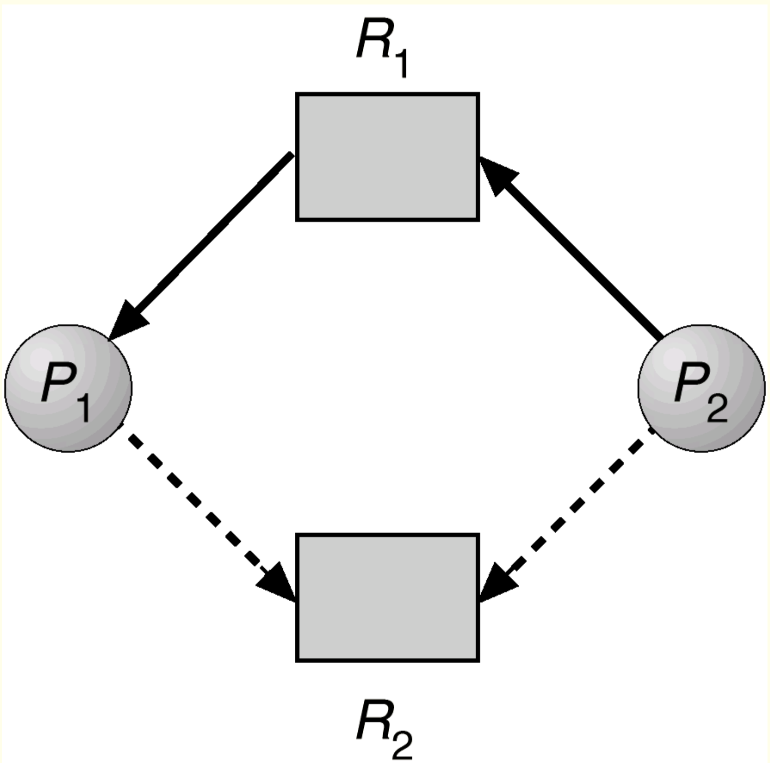
- both schemes avoid ***starvation*** provided rolled back processes keep their original timestamp - eventually a process will be the oldest
- however they still are not “ideal”….

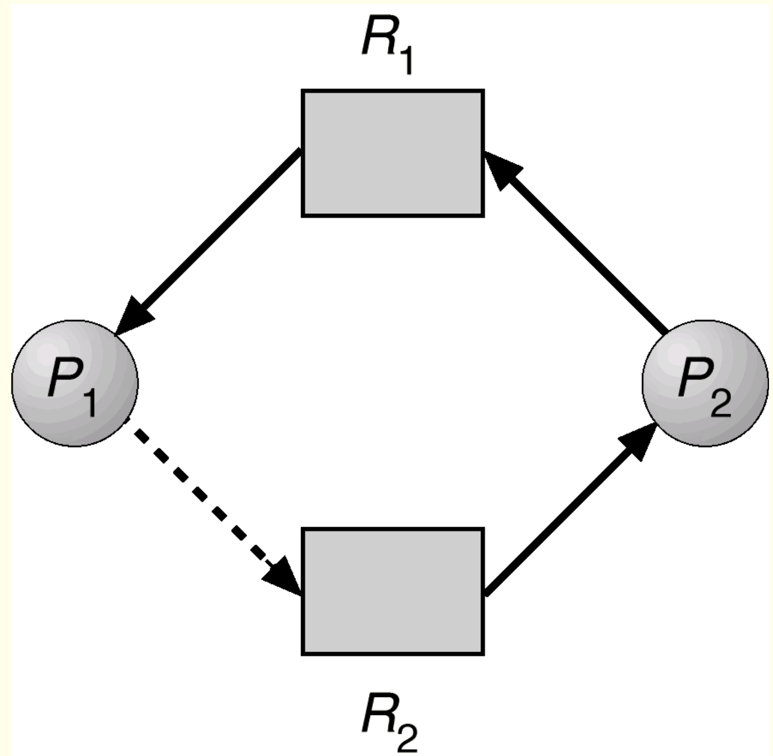
comparison of the two schemes

	Wait die	Wound-wait
Older process	must wait for younger one	never waits
Younger process	Get rolled back	must wait for older processes and can be wounded

	Wait die	Wound-wait
timestamp	Low … high	Low … high
Process age	Older … younger	Older … younger

deadlock ***avoidance***





- **use approach used in concurrent systems - *a priori claims***
- **direct all request through a central coordinating host**
- **coordinating host maintains resource allocation graph**
- **simple - but has a bottleneck**

deadlock *detection* – centralised approach

- each site keeps **a *local* wait-for graph**.
- the **nodes** of the graph correspond to all the **processes** (local & remote) that are currently holding *or* requesting any ***local* resources**
- **a global wait-for** graph is maintained in a ***single* coordination process**; this graph is the union of all local wait-for graphs.
- as with concurrent systems - look for cycles in the graph
- there **are three different points in time** when the wait-for graph may be constructed - the three options are :
  1. a new edge is inserted or removed in any local wait-for graph
  2. at periodic intervals
  3. the coordinator invokes the cycle-detection algorithm
- unnecessary rollbacks may occur
  - "race-conditions" can cause *false cycles* to be detected
  - "race conditions" fail to show that *true cycles* have self-corrected .

detection algorithm based on option 3

INTUITIVE IDEA

- this approach removes “race condition” problems
- append unique identifiers (timestamps) to requests from different sites.
- when process P_i , at site *A*, requests a resource from process P_j , at site *B*, a request message with timestamp *T* is sent.
- The edge $P_i \rightarrow P_j$ with the label *T* is inserted in the local wait-for graph of *A*.
- the edge is inserted in the local wait-for graph of *B* only if *B* has received the request message and cannot immediately grant the requested resource.

the algorithm

1. The controller sends an initiating message to each site in the system.
2. On receiving this message, a site sends its local wait-for graph to the coordinator.
3. When the controller has received a reply from each site, it constructs a graph as follows:
   (a) The constructed graph contains a vertex (node) for every process in the system.
   (b) The graph has an edge $P_i \rightarrow P_j$ iff
   (1) there is an edge $P_i \rightarrow P_j$ in one of the wait-for graphs,
   or
   (2) there is an edge $P_i \rightarrow P_j$ with some label T appearing in more than one wait-for graph.

If the constructed graph contains a cycle $\Rightarrow$ deadlock.

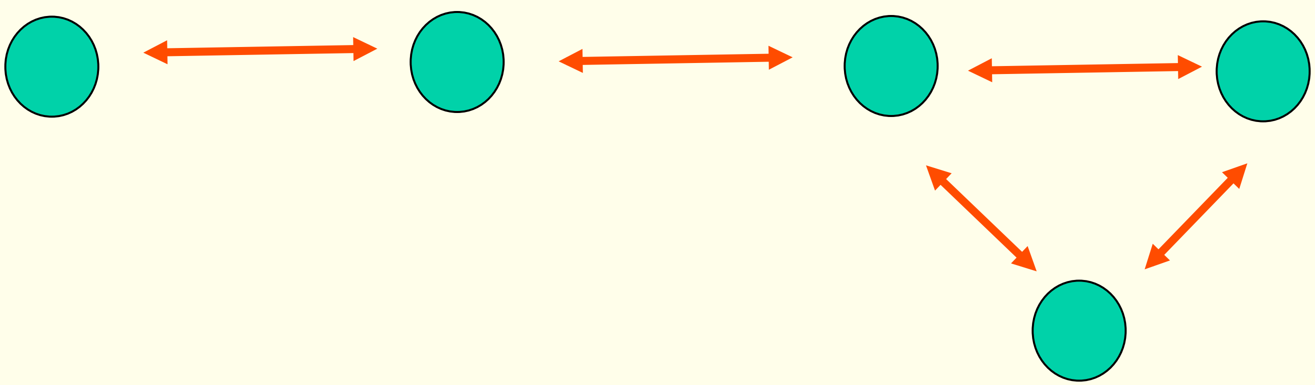
fully distributed approach

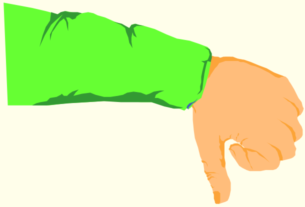
INTUITIVE IDEA

- **all controllers share equally the responsibility** for detecting deadlock.
- every site constructs a wait-for graph that represents a part of the total graph.
- we add one additional node P_{ex} to each local wait-for graph.
- if a local wait-for graph contains a cycle that does not involve node P_{ex} , then the system is in a deadlock state.
- a cycle involving P_{ex} implies the *possibility* of a deadlock.
- to ascertain whether a deadlock does exist, a distributed deadlock-detection algorithm must be invoked to form a "union" of graphs between sites and checks for cycles.

distributed coordination

- **mutual exclusion**
- **deadlocks**
- ***election algorithms***





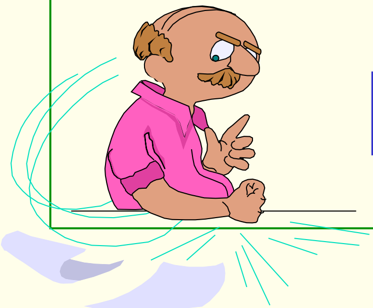
election algorithms

- many algorithms we have seen have **coordinators.**
- following an event (like failure) - use **election algorithms** to determine where a new copy of the coordinator should be restarted.

- **assume** that a **unique priority number** is associated with each active process in the system, e.g.the priority number of process P_i is i.

- the coordinator is **always** the process with **the largest priority number**.

- when a coordinator fails, the algorithm must elect that active process with the largest priority number.
- two algorithms, the ***bully algorithm*** and the ***ring algorithm***, can be used to elect a new coordinator following a failure.



Bully Algorithm (1 of 3)

- applicable to systems where **every process can send a message to every other process** in the system.
- if process P_i sends a request that is not answered by the coordinator within a **time interval *T***, assume that the coordinator has **failed.**
- Then, P_i **tries to elect itself** as the new coordinator.
- P_i sends an election message to every process **with a higher priority** number, P_i then waits for any of these processes **to answer within time *T*.**

Bully Algorithm (2 of 3)

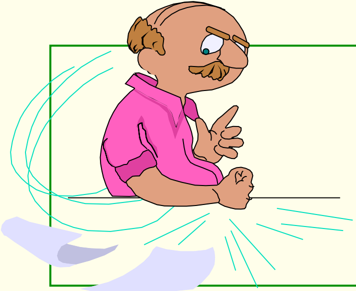
- if **no response within *T***, assume that all processes with numbers greater than i have failed:

  P_i elects itself the **new coordinator**.

- if answer is received, P_i begins time interval T', **waiting to receive a message** that a process with a higher priority number has been elected.

- **if no message is sent within T'**, assume the process with a higher number has failed:

  P_i should **restart the algorithm**



Bully Algorithm (3 of 3)

- if **P_i is not the coordinator,** at any time during execution it may receive one of the following two messages from process P_j.
  - P_j is **the new coordinator** ($j > i$). P_i records this information.
  - P_j **started an election** ($j < i$). P_i sends a response to P_j and **begins its own election algorithm**, provided that P_i has not already initiated such an election.
- after **a failed process P_i recovers**, it immediately begins the execution of the same algorithm.
- if there are no active processes with higher numbers, the recovered process forces all processes with lower number to let it become the coordinator process, even if there is a currently active coordinator with a lower number.

Ring Algorithm (1 of 2)

- applicable to systems organized as a ring (logically or physically).
- assumes that the links are **unidirectional**, and that processes send their messages **to their right neighbours**.
- each process maintains an ***active list***, consisting of all the priority numbers of all active processes in the system when the algorithm ends.
- if process P_i **detects a coordinator failure**, it creates a new active list that is initially empty. It then sends a message *elect(i)* to its right neighbour, and adds the number i to its active list.
- assume that communication works even if leader and other services mail fail

Ring Algorithm (2 of 2)

- if ***P_i* receives** a message *elect(j)* from the process on the left, it must respond in one of three ways:

  1. if this is **the first *elect* message** it has seen or sent, P_i creates **a new active list** with the numbers *i* and *j*. It then sends the message *elect (i)* , followed by the message *elect (j)*

  otherwise

  2. if $i \neq j$, ***P_i* adds *j* to its active list**, & forward the message to the right.
  3. if $i = j$, then **the active list for *P_i*** now contains the numbers of all the active processes in the system. P_i can **now determine the largest number** in the active list to identify the **new coordinator** process.