Four necessary conditions for deadlock: mutual exclusion, hold and wait, no preemption, circular wait

for every reusable R_i
$t_i = |R_i|$ = # of units of resource of type i
≥ 0
for every consumable R_i
there exists subset R C P
producers of R_i
A **bi-partite graph** (bigraph) is a graph whose nodes can be divided into two disjoint sets such that two adjacent nodes cannot come from the same set.
for every reusable resource R_i:

- # of assignment edges $\#(R_i, *) \le t_i$
- $a_i = t_i - \#(R_i, *)$
- a process cannot request more than the total number of units of a resusable resource
  $\#(P_j, R_i) \le t_i - \#(R_i, P_j)$

for every consumable resource R_i

- (R_i, P_j) exists iff P_j is a producer of R_i
- $a_i \ge 0$

if $\#(P_i, R_j) > a_j \Rightarrow p_i$ blocked (more requests than available resources)
If a graph can be reduced by all the processes, then there is **no** deadlock.
If a graph cannot be reduced by all the processes, the irreducible processes constitute the set of deadlock processes in the graph
Definition
A **knot** K in a graph is a nonempty set of nodes such that for every node x in K, all nodes in K and only the nodes in K are reachable from x.
(all x, all y ∈ K => x → y) AND (all x ∈ K, some z, such that x → z => z ∈ K)

A cycle is a necessary condition for a deadlock.
If the graph is expedient, then a knot is a sufficient condition for deadlock.
Systems with Single-unit Requests
A process can only request one resource unit at a time.
Theorem: A knot is a necessary and sufficient condition for deadlock.
Systems with Single-unit Resources
There is only one unit of every resource.
Theorem: A cycle in an *expedient graph* is a necessary and sufficient condition for deadlock.
System with only consumable resources
such as interrupts, signals, messages, and data in I/O buffers
deadlock may occur if a **receive** operation is blocking

Definition: **claim-limited graph** of a consumable resource system:
each resource has 0 available units
it has a request edge (p_i, R_j) iff P_i is a consumer of R_j

- **Basic Client Server Model**
- Characteristics:
  - There are processes offering services (servers)
  - There are processes that use services (clients)
  - Clients and servers can be on different machines
  - Clients follow request/reply model when using services
  - Synchronous communication: request-reply protocol
  - In LANs, often implemented with a connectionless protocol (unreliable)
  - In WANs, communication is typically connection-oriented TCP/IP (reliable)
  - - High likelihood of communication failures

Decentralized Architectures

- In the last couple of years we have been seeing a tremendous growth in peer-to-peer systems.
- Structured P2P: nodes are organized following a specific distributed data structure
- Unstructured P2P: nodes have randomly selected neighbors
- Hybrid P2P: some nodes are appointed special functions in a well-organized fashio

Processor: Provides a set of instructions along with the capability of
automatically executing a series of those instructions.
Thread: A minimal software processor in whose context a series of

- instructions can be executed. Saving a thread context implies stopping

- the current execution and saving all the data needed to continue the

execution at a later stage.
Process: A software processor in whose context one or more threads may be

- executed. Executing a thread, means executing a series of instructions

in the context of that thread.
Context Switching:
User-space solution
All operations can be completely handled within a single process =>
implementations can be extremely efficient.
All services provided by the kernel are done **on behalf of the process in which a thread resides** => if the kernel decides to block a thread, the entire process will be blocked.
Threads are used when there are lots of external events: **threads block on a per-event basis** => if the kernel cannot distinguish threads, how can it support signaling events to them?

Kernel solution
The whole idea is to have the kernel contain the implementation of a thread package. This means that all operations return as system calls
Operations that block a thread are no longer a problem: the **kernel schedules another available thread** within the same process.
Handling external events is simple: the **kernel** (which catches all events) **schedules the thread associated with the event**.
The big problem is the **loss of efficiency** due to the fact that each thread operation requires a trap to the kernel.
Virtualization is becoming increasingly important:
Hardware changes faster than software
Ease of portability and code migration
Isolation of failing or attacked components

Basic model
A server is a process that waits for incoming service requests at a specific transport address. In practice, there is a one-to-one mapping between a port and a service.

Type of servers:
Iterative servers: handles a request itself; can handle only one client at a time,
Concurrent servers: Does not handle a request itself; pass it to a separate thread or another process
Superservers: Servers that listen to several ports, i.e., provide several independent services. In practice, when a service request comes in, they start a subprocess to handle the request (UNIX inetd)

Stateless servers
Never keep accurate information about the status of a client after having handled a request:
Don't record whether a file has been opened (simply close it again after access)
Don't promise to invalidate a client's cache
Don't keep track of your clients
Consequences
Clients and servers are completely independent
State inconsistencies due to client or server crashes are reduced
Possible loss of performance because, e.g., a server cannot anticipate client behavior (think of prefetching file blocks)

Software Agents
What is an Agent?
Definition: An autonomous process capable of reacting to, and initiating changes in its environment, possibly in collaboration with users and other agents
collaborative agent: collaborate with others in a multiagent system
mobile agent: can move between machines
interface agent: assist users at user-interface level
information agent: manage information from physically different sources
Transport Layer
Important
The transport layer provides the actual communication facilities for

most distributed systems.

Standard Internet Procotols:

- TCP: connection-oriented, reliable, stream-oriented communication
- UDP: unreliable (best-effort) datagram communication

Basic RPC operation

Observations

Application developers are familiar with simple procedure model
Well-engineered procedures operate in isolation (black box)
There is no fundamental reason not to execute procedures on separate machine

Basic principle

Every machine has a timer that generates an interrupt H times per second.
There is a clock in machine p that ticks on each timer interrupt.
Denote the value of that clock by $C_p(t)$, where t is UTC time.
Ideally, we have that for each machine p, $C_p(t) = t$, or, in other words, $dC/dt = 1$.

Real distance is

$$d_i = c\Delta_i - c\Delta_r = \sqrt{(x_i - x_r)^2 + (y_i - y_r)^2 + (z_i - z_r)^2}$$

Lamport's Logical Clock

Condition requirements:

1. for any two events **a** and **b** in a process P_i,
   if a occurs before b, then
2. $C_i(a) < C_i(b)$
3. if **a** is the event of sending a message **m** in P_i
   and **b** is the event of receiving the same message **m**
   at process P_j, then
   $C_i(a) < C_j(b)$

Implementation rules:

1. two successive events in P_i
2. $C_i = C_i + d$ ($d > 0$)
3. if a and b are two successive events in P_i and $a \rightarrow b$ then
4. $C_i(b) = C_i(a) + d$ ($d > 0$)
5. event a: sending of message m by process P_i,
   timestamp of message m : $t_m = C_i(a)$ then
6. $C_j = \max (C_j, t_m + d)$ $d > 0$

Vector Clocks

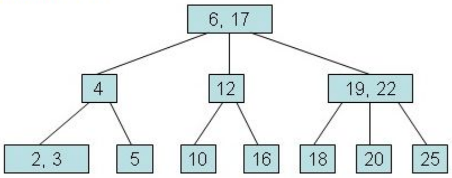
Implementation Rules:

1. two successive events a, b in process P_i:
2. $C_i(b)[i] = C_i(a)[i] + d$ ($d > 0$)
3. event a at P_i sending message m to process P_j with receiving event b; vector timestamp $\mathbf{t_m} = \mathbf{C_i(a)}$ is assigned to m; on receiving m, P_j updates C_j as follows:
4. all k, $C_j(b)[k] = \max(C_j(b)[k], t_m[k])$

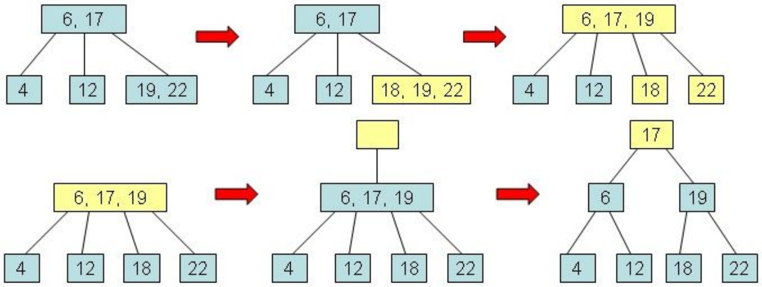
Assertion.

At any instant
all i, all j : $C_i[i] \geq C_j[i]$

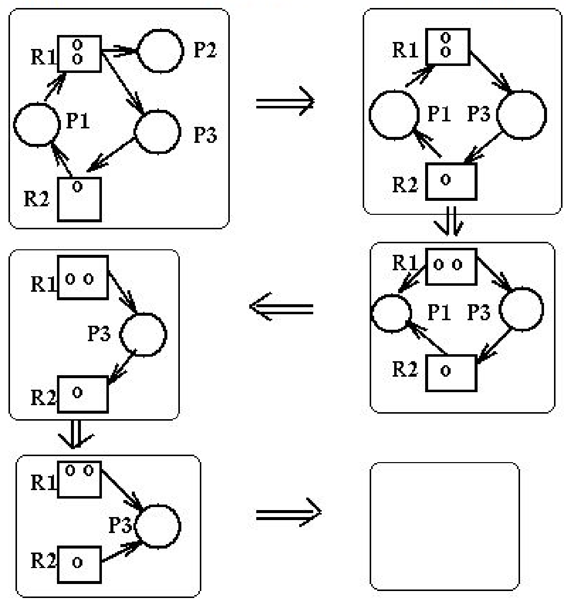
Minimum is 1



Insert 18



An example of reduction of an RAG (Resource Allocation Graph)

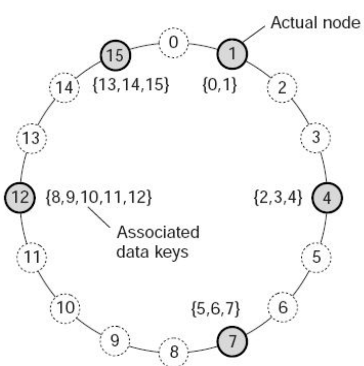


Overlay Network

Nodes are formed by the processes of the network.

Structured P2P Systems
Basic idea

- Organize the nodes in a structured overlay network such as a logical ring, and make **specific nodes responsible for services** based only on their ID.
- A common approach is to use a distributed hash table (DHT) to organize the nodes.
- Traditional hash functions convert a key to a hash value, which can be used as an index into a hash table:
  - Keys are unique -- each represents an object to store in the table;
  - The hash function value is used to insert an object in the hash table and to retrieve it.
- In a DHT, data objects and nodes are each assigned a key which hashes to a random number from a very large identifier space (to ensure uniqueness).
- A mapping function assigns objects to nodes, based on the hash function value.
- A lookup, also based on hash function value, returns the network address of the node that stores the requested object.

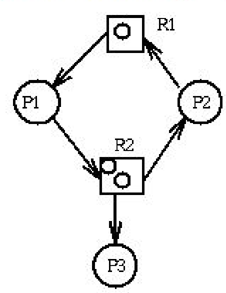



```
Note
The system provides an operation
LOOKUP(key) that will efficiently
route the lookup request to the
associated node.

Achieved by organizing processes through a
distributed hash table (DHT)

e.g. a data item with key k is mapped to the node with the
smallest idenifier id ≥ k. This node
is referred to as the successor of the datum with
key k, and is denoted as succ(k).
```

Example of RAG with loop but no deadlock



Example of Resource Allocation Graph with loop but no deadlock

Other example
Organize nodes in a d-dimensional space and let every node take the responsibility for data in a specific region. When a node joins => split a region. (e.g. d = 2 in the following figure.)