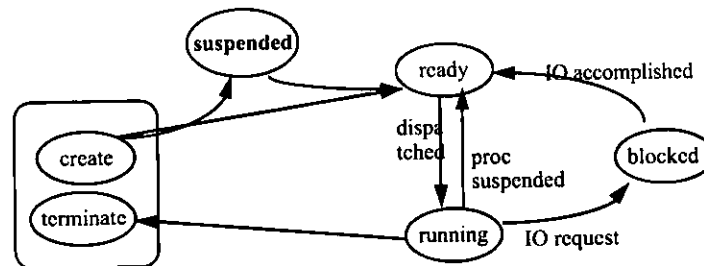


Process Management

- An issue in distributed scheduling is **load balancing**, which tries to distribute the tasks to be executed among the resources of the system. (Resources include CPU, disks, etc.)
- Load balancing can be achieved either locally (if there are multiple resources in a node) or in a distributed fashion
- Distributing tasks across a communication medium is sometimes referred to as the **resource allocation problem**
- Resource allocation actually refers to scheduling multiple resources, but it is a complex (NP) problem, so do CPU only (NP also?)
- Process **migration** is a facility to dynamically relocate a process after it started executing (similar to preemption to another CPU)
- Therefore, the process state (entire address state?) has to be transferred from the source processor to the destination processor
- The state of the process has to be frozen before being transferred, accounting for insufficiency in migration. Hard problem.

Load Balancing



- At creation time, the stack, PCB, etc are setup
- If the system is too full, then the task goes to the **suspended** state
- Load balancing algorithms will *prefer* to take tasks from the system suspended state, rather than the ready state
- If the task is in the ready state, chances are it ran for a little while and it just waiting for CPU. This is migration and need to transfer resources.

Load Balancing

- The first issue is how to measure the load. Some measure it by the size of the ready queue. Simple and cheap approach.
- Some say that there is little correlation between queue length and the actual load in the processor. So, use the utilization
- The problem is how to measure utilization:
 - background process continuously monitoring (V system): overhead
 - idle thread/process (Mach system): imprecise
 - accounting info over period of time: what interval?
- The second issue is the difference between **load balancing (LB)** and **load sharing (LS)**: balancing implies all processors should have approximately the same load. sharing allows a high and a low load processor to share their load.
- the problem with LB is the overhead involved in achieving it
- the problem with LS is that it may have peaks of load that are harder to deal with in a dynamic environment

Types of LB Algorithms

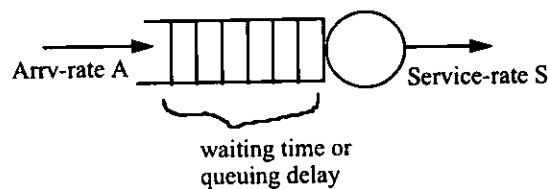
- If a processor has a high load, it will try to share it with another processor (*sender initiated*)
- If it has a low load, it should offer its services to the distributed system (*receiver initiated*)
- If it does both, it is called a *hybrid system*
- LB Algorithms are based on the info provided by the system, the tasks, the users.
 - *static*: pre-determines the LB, not taking into account current state. Example: unixd systems. log on to unixd[1234] in RR fashion
 - *dynamic*: takes the timely information and uses it in the LBAlg. Exm: If a load is very high in a node, will not send it there. More overhead, better performance (if not counting overhead)
 - *adaptive*: special case of dynamic, in which the algorithm may change as a response to the state of the system. Exm: If the system load is very high, will not even attempt to collect state from others. Best performance, but complex to figure out

Components of LB

- *Transfer policy*: decides **whether** node is eligible, usually based on a threshold (Q size, load, etc)
- *Selection policy*: among the eligible tasks, **which** one is to be transferred . based on remaining execution time, waiting time, decrease in response time, decrease in resp time - comm time, overhead of transfer, size, location, interaction with others, etc
- *Location policy*: **to which** node to transfer. need also to locate the eligible receivers (through polling, broadcast, guessing, randomly, etc)
- *Information policy*: **when** (or **how often**) to trigger the collection of system state and **where** to collect from
 - *on demand* : whenever beyond a threshold (sender/receiver/hybrid)
 - *periodic* : periodically, so that there is no overhead if load is high, but predictable overhead.
 - *state-change-driven*: when a node changes (dramatically) its state, it will advertise to others (master or peers)

Stability Conditions

- Queuing theory: there is a queue of jobs waiting for service. How long will it be before job i is service?
- If I have a single processor, and many queues, how should the jobs in different queues be services
- If I have several processors (servers), how many queues should there be?
- all these, and then some, questions are studied through queuing theory...



Stability condition

- If $A/S < 1$, then the system is stable, otherwise, the system is unstable. If the system is unstable, the queue will grow infinitely
- In load balancing, A is not the only factor to consider: if $A+LB < S$, then the system is stable, where LB is the overhead/load due to the load balancing algorithm
- Even when stable, LBAIgs can still perform worse than a system without LB . This is usually due to poor LB techniques AND the overhead of collecting info and transferring tasks.
- Another form of instability is when a task arrives at a node (node 1) and increases the load above the threshold, so it gets transferred to node 2. When it arrives at a node 2, it increases the load above the threshold, so it gets transferred to node 3. And it keeps skipping from node to node, ad infinitum.

LB: Sender-Initiated Algorithms

- **stability:** In sender-initiated algorithms, if the load is light, the sender node will eventually find a receiver node, thus being allowed to load shed. If an on-demand **information policy** is used, as the load rises, the polling of other nodes becomes a burden to the system, eventually outweighing the benefits of LB . If periodic **information policy** is used, with high loads the same phenomenon occurs.
- **transfer policy (TP):** use a threshold policy on the readyQ length for the transfer policy: a task t arrives and makes $|Q| > T$, the node becomes a sender.
- **selection policy (SP):** consider *only* newly arrived tasks
- **location policy (LP):** different algorithms (see next slide)
- **information policy (IP):** demand-driven

Sender-Initiated Algorithms

- **location policy (LP):** different algorithms:
- *Random LP:* choose a remote site randomly. No overhead of collecting information, but no knowledge may make tasks be transferred among nodes with $|Q| > T$. Still, better than no LB.
- *Threshold LP:* select remote sites randomly, but before sending tasks, poll Q length. If $|Q| > T$, don't send task, choose another node. More overhead, but better performance.
- *ShortestQ LP:* select k nodes at random, poll them, send to the node with smallest $|Q|$ and $|Q| < T$. More overhead (k times) than threshold LP, but only marginal improvement

Receiver-Initiated Algorithms

- **stability:** In receiver-initiated algorithms, if the load is high, the receiver node will eventually find a sender node to load shed. This means that the polls will be well used. In light loads, the polls will be wasted, but since the load is low, no instability.
- **transfer policy (TP):** use a threshold policy on the readyQ length for the transfer policy: a task t departs and makes $|Q| < T$, the node becomes a receiver.
- **selection policy (SP):** consider newly arrived tasks, tasks whose transfer overhead is small wrt the transfer delay, and other criteria presented earlier.
- **location policy (LP):** select a node at random, poll it. If the node is a sender, receive the task. If not, select another node. If a node limit was reached, wait for some time, start again.
when is the transfer done? is it always task migration? why? is there a solution for this?
- **information policy (IP):** demand-driven since polling only starts when $|Q| < T$

Symmetrically-Initiated Algorithms

- **stability:** In hybrid algorithms, since it has a sender-initiated component, if the load is high, polling could make the system unstable. But the receiver-initiated part will find a suitable task
- **transfer policy (TP):** use a double threshold $\langle T, T' \rangle$ policy on the readyQ length for the transfer policy: a task t arrives and makes $|Q| > T$, the node becomes a sender. If a task finished, making $|Q| < T'$, the node becomes a receiver
- **selection policy (SP):** consider *any* of the approaches
- **location policy (LP): Sender Initiated**
 - 1- A node with a high load (sender) broadcasts a *task_offered* message, and waits for an *accept* message from some receiver.
 - 2- On receipt of a *task_offered* message, a node ignores it or increases its estimate of load (**why??**) and sends an *accept*.
 - 3- If an *accept* was NOT received within a timeout interval, the node broadcasts a message that the system load is high
 - 4- If an *accept* was received, the node decides whether to send.

Sym-init Algorithm

- **location policy (LP): Receiver Initiated**
 - 1- A node with a low load (receiver) broadcasts a *task_wanted* message, and waits for an *task* from a sender.
 - 2- If a *task* was NOT received within a timeout interval, the node broadcasts a message that the system load is low
- If an *task* is received, the node increases its estimate of the load
- **information policy (IP):** demand-driven since polling only starts when conditions arise. NOTE that there is little information dissemination among the nodes. Most of the information is *inferred* from the messages (*task_wanted*, *task_offered*) received, or from lack of messages (timeouts). Thus not increasing the system load significantly
- Note also that the advantages AND disadvantages of both SI and RI algorithms are present. SI component is bad in high loads, RI component does not help during low loads. Average loads are the most benefitted systems by this algorithm