Classifying Load Distribution Algorithms (cont.)

- How is system state (load on each processor) used?

  - Static / deterministic
    - Does not consider system state; uses static information about average behavior
    - Load distribution decisions are hard-wired into the algorithm
    - Little run-time overhead

  - Dynamic
    - Takes current system state into account
    - Has the potential to outperform static load distribution because it can exploit short-term fluctuations in system state
    - Has some overhead for state monitoring

  - Adaptive
    - Subclass of dynamic
    - Modify the algorithm based on the state
    - For example, use different load distribution policies based on load thresholds

Classifying Load Distribution Algorithms (cont.)

- Can a task be transferred to another processor once it starts executing?

  - Preemptive / migratory transfers
    - Can transfer a task that has partially executed
    - Have to transfer entire state of the task
      - Address space: code, data, stack, heap
      - Process state: register contents, scheduling info, memory tables, I/O states, process ID, file info, etc.
      - Environment variables, working directory, inherited privileges, etc.
    - Expensive!!

  - Non-preemptive / non-migratory transfers
    - Can only transfer tasks that have not yet begun execution
    - Still have to transfer environment info
      - Program: code
      - Environment variables, working directory, inherited privileges, etc.
    - But no process state or dynamic address space data to transfer

Classifying Load Distribution Algorithms (cont.)

- Is the algorithm stable?

  - Queuing-theoretic approach
    - When the long-term arrival rate of work to a system is greater than its capacity to perform work, the system is *unstable*
      - Overhead due to load distribution can itself cause *instability*
        - » Exchanging state, transfer tasks, etc.
    - Even if an algorithm is stable, it may cause the system to perform worse than if the algorithm were not used at all — if so, we say the algorithm is *ineffective*
    - We want an algorithm that is both effective and stable

  - Algorithmic perspective
    - If an algorithm performs fruitless actions indefinitely with finite probability, it is *unstable* (e.g., processor thrashing)
      - Transfer task from P1 to P2, P2 exceeds threshold, transfers to P1, P1 exceeds…

Components of a Load Distribution Algorithm

- Transfer policy

  - Determines if a processor is in a suitable state to participate in a task transfer

- Location policy

  - Finds suitable processors (senders or receivers) to share load

- Selection policy

  - Selects a task for transfer, once the transfer policy decides that the processor is a sender

- Information policy

  - Decides:
    - When information about the state of other processors should be collected
    - Where it should be collected from
    - What information should be collected

Components of a
Load Distribution Algorithm

- ■ Transfer policy

  - ● Determines whether or not a processor is a sender or a receiver
    - ■ Sender — overloaded processor
    - ■ Receiver — underloaded processor

  - ● Threshold-based transfer
    - ■ Establish a *threshold*, expressed in units of load (however load is measured)
    - ■ When a new task originates on a processor, if the load on that processor exceeds the threshold, the transfer policy decides that that processor is a sender
    - ■ When the load at a processor falls below the threshold, the transfer policy decides that the processor can be a receiver

  - ● Single threshold
    - ■ Simple, maybe too many transfers

  - ● Double thresholds — high and low
    - ■ Guarantees a certain performance level

  - ● Imbalance detected by information policy

Components of a
Load Distribution Algorithm (cont.)

- ■ Location policy

  - ● Once the transfer policy designates a processor a sender, finds a receiver
    - ■ Or, once the transfer policy designates a processor a receiver, finds a sender

  - ● Polling — one processor polls another processor to find out if it is a suitable processor for load distribution, selecting the processor to poll either:
    - ■ Randomly
    - ■ On a nearest-neighbor basis
    - ■ Based on information collected in previous polls

  - ● Can poll processors either serially or in parallel (e.g., multicast)
    - ■ Usually some limit on number of polls, and if that number is exceeded, the load distribution is not done

  - ● Can also just broadcast a query to find a node who wants to be involved

Components of a
Load Distribution Algorithm (cont.)

- ■ Selection policy

  - ● Selects a task for transfer, once the transfer policy decides that a particular machine is a sender

  - ● Non-preemptive
    - ■ Select the new tasks that caused the processor to become a sender (by increasing its load above the threshold)

  - ● Preemptive
    - ■ Transfer long tasks
      - – Overhead in task transfer should be less than reduction in response time caused by the task
      - – Have to predict execution time
    - ■ Transfer tasks whose response time will be improved after the transfer

  - ● Other factors to consider
    - ■ Minimize overhead in transfer (small tasks)
    - ■ Location-dependent system calls (use resources that are only on one processor)

Components of a
Load Distribution Algorithm (cont.)

- ■ Information policy

  - ● Decides:
    - ■ When information about the state of other processors should be collected
    - ■ Where it should be collected from
    - ■ What information should be collected

  - ● Demand-driven
    - ■ A processor collect the state of the other processors only when it becomes either a sender or a receiver (based on transfer and selection policies)
    - ■ Dynamic — driven by system state
      - – Sender-initiated — senders look for receivers to transfer load onto
      - – Receiver-initiated — receivers solicit load from senders
      - – Symmetrically-initiated — combination where load sharing is triggered by the demand for extra processing power or extra work

Components of a
Load Distribution Algorithm (cont.)

- Information policy (cont.)
  - Periodic
    - Processors exchange load information at periodic intervals
    - Based on information collected, transfer policy on a processor may decide to transfer tasks
    - Does not adapt to system state — collects same information (overhead) at high system load as at low system load
  - State-change-driven
    - Processors disseminate state information whenever their state changes by a certain degree
    - Differs from demand-driven in that a processor <u>disseminates</u> information about its state, rather than <u>collecting</u> information about the state of other processors
    - May send to central collection point, may send to their peers

3 Sender-Initiated Algorithms
(Eager, Lazowska, Zahorjan, 1986)

- Transfer Policy   (who will participate?)
  - Based on load & threshold(s), processors decide if they are a sender or a receiver
  - Triggered by new task (on a sender)

- Selection Policy   (transfer which task?)
  - New tasks only (non-preemptive)

- Location Policy   (where to transfer?)
  1. Random
     - Doesn't use remote state information
     - Transfers task to a processor selected at random (which may have to transfer it yet again to some other processor)
     - Problem — system will eventually spend all its time transferring tasks
       - Solution — limit number of transfers
     - Provides substantial performance improvement over no load sharing

3 Sender-Initiated Algorithms
(Eager, Lazowska, Zahorjan) (cont.)

- Location Policy (cont.)
  2. Threshold
     - Poll a processor at random
       - If it's a receiver, transfer the task to it
       - Otherwise, poll another processor
     - Limit the number of polls to keep the overhead down
       - If can't find anyone to take the task, the sender has to keep it
     - Avoids useless transfers, so provides substantial performance improvement over the random location policy
  3. Shortest
     - Poll a random set of processors (less than some limit) to find their queue lengths
     - Select processor with shortest queue length, and select it to receive the task, unless its queue length > threshold
     - Provides only marginal performance improvement over the threshold location policy (extra information didn't really help)

3 Sender-Initiated Algorithms
(Eager, Lazowska, Zahorjan) (cont.)

- Information Policy   (collect state?)
  - Random location policy
    - No state collected
  - Threshold / shortest location policy
    - Demand-driven — polling happens when transfer policy identifies a processor as a sender

- Stability
  - Location policy is <u>not effective</u> at high system loads, and causes <u>instability</u> by failing to adapt to the system state
    - No processor is likely to be lightly loaded
    - Polling activity increases as the rate at which work arrives in the system increases
      - Eventually reaches a point where the cost of load sharing is greater than the benefit
        » Most of effort is wasted in polling and responding to polls
      - Work exceeds capacity $\Rightarrow$ instability

Receiver-Initiated Algorithms
(Shivaratri and Krueger, 1990)

- ■ Transfer Policy   (who will participate?)
  - ● Based on load & threshold(s), processors decide if they are a sender or a receiver
  - ● Triggered by termination of a task (on a receiver)

- ■ Selection Policy   (get which task?)
  - ● Non-preemptive
    - ■ May not be a new task ready for transfer
  - ● Preemptive
    - ■ Long tasks
    - ■ Tasks whose performance will increase

- ■ Location Policy   (get from where?)
  - ● Threshold
    - ■ Poll a processor at random
      - – If it's a sender, transfer a task from it
      - – Otherwise, poll another processor

Receiver-Initiated Algorithms
(Shivaratri and Krueger) (cont.)

- ■ Location Policy (cont.)
  - ● Threshold (cont.)
    - ■ Limit the number of polls to keep the overhead down
      - – If can't find anyone to get a task from, receiver must wait until another task completes, or some timeout occurs

- ■ Information Policy   (collect state?)
  - ● Demand-driven — polling happens when transfer policy identifies a processor as a receiver

- ■ Stability
  - ● At high system load, there is a high probability that a receiver will find a suitable sender to share the load within a few polls $\Rightarrow$ stable and effective
  - ● At low loads, polls more, but not so much as to cause instability

Symmetrically-Initiated Algorithms

- ■ At same time (use previous algorithms):
  - ● Senders are searching for receivers
  - ● Receivers are searching for senders

- ■ Get advantages of both algorithms:
  - ● At low system loads, the senders are successful at finding underloaded receivers
  - ● At high system loads, the receivers are successful at finding overloaded senders

- ■ Get disadvantages of both algorithms:
  - ● At high system loads, the senders can cause instability
  - ● The receivers usually require expensive preemptive task transfers

Adaptive
Symmetrically-Initiated Algorithms

- ■ Threshold Policy uses two thresholds:
  - ● If queue > upper thresh, proc. is a sender
  - ● If queue < lower thresh, proc. is a receiver
  - ● Otherwise, processor is OK

- ■ Still symmetrically-initiated, but tries to use information from previous polls
  - ● Start out assuming everyone is a receiver, gradually learn everyone's status, update due to later polls

- ■ Evaluation:
  - ● At high system loads, senders avoid indiscriminate polling, so do not cause instability
  - ● The receivers still usually require expensive preemptive task transfers