CSCE 455/855
Distributed Operating Systems

System Models,
Processor Allocation,
Distributed Scheduling,
and
Fault Tolerance

Steve Goddard
goddard@cse.unl.edu

http://www.cse.unl.edu/~goddard/Courses/CSCE855

System Models
Workstation model

- Each workstation has a processor and an owner
- Disks are optional
  - diskless: no disk
    - low cost, easy system updates, server bottleneck
  - paging files: small, cheap disk for paging
    - reduces network traffic
  - paging files & binary: application binaries are local
    - further reduces network traffic, but updates become more difficult
  - paging files & binaries & caching: also cache file pages
    - less network traffic and load on servers, but cache consistency problems
  - full local file system: each workstation has file system
    - no need for file servers, but transparency can suffer (ala NFS , etc.)
- Local processes take precedence over remote

System Models
Processor pool model

- Pool of idle processors available for everyone
- “Workstations” may not even have a processor

Processor Allocation

- Two basic allocation models:
  - non-migratory: once process is placed, it cannot be moved
  - migratory: process can move in the middle of execution
    - must restore state at new CPU
    - better load balancing, but more complex design

Processor Allocation

- Distributed processor management
  - » if local processor is idle or underutilized, use it
    - ❖ otherwise execute it remotely
  - » resource manager must:
    - ❖ keep track of idle processors
    - ❖ find one when a request is received
    - ❖ send the process to a remote computer
    - ❖ receive results from remote process
  - » first problem: find a CPU

Finding Idle Workstations

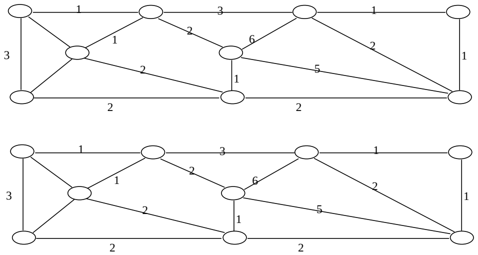
- Server-driven
  - » idle processors announce their availability
  - » processor puts info in a global (replicated) registry
  - » broadcast ‘available’ message
    - ❖ but all processors need to maintain the list
  - » race conditions can occur
    - ❖ more than one client sends work to same idle processor
- Client-driven
  - » client broadcasts need for a processor
    - ❖ heterogeneous environments need info on processing needs
  - » idle processors respond with message to client
    - ❖ servers can delay message in proportion to load
  - » client chooses one

Graph Theory Approach

- Assign process to a CPU
  - to minimize network traffic (interprocess communication)
  - processes and messages make a weighted graph
    - nodes are processes
    - edges are communication paths weighted by messages
  - total network traffic is sum of arcs intersected by partition

Graph Theory Approach

- Assign 9 processes to 3 CPUs
  - two different examples:



Graph Theory Approach

- Essentially looks for tightly clustered processes
  - » place interacting processes on same machine
- Problem with graph approach
  - » assumes pre-knowledge of message traffic
  - » lots of information to process
  - » computationally difficult to achieve

Centralized Load Sharing
Up-Down Algorithm

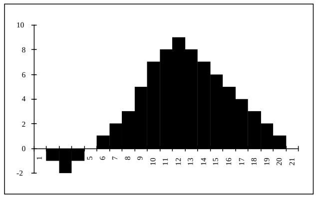
- Objective: fairly divide CPU cycles among users
- When a CPU becomes idle:
  - » if users have processes in their waiting queue...
  - » decide which should run next
  - » decision based on “penalty points”

Centralized Load Sharing
Up-Down Algorithm

- Central coordinator maintains a usage table
  - » for each event, message sent to coordinator
- Table entries for each user process track “penalty points” (for each time unit)
  - » process executing remotely (add points)
  - » pending processes (subtract points)
    - ❖ i.e. processes on ready queue
  - » processor idle (move toward zero)
- Allocate idle CPU based on penalty points
  - » process whose owner has fewest points “wins”
  - » shares computing power equally

Centralized Load Sharing
Up-Down Algorithm

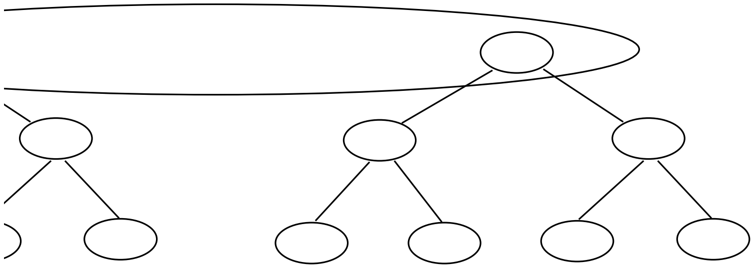
- Awards light process user
  - » user (process) occupying no processor with a pending request



- Can visualize algorithm execution as a scale
- Problems with the up-down algorithm
  - » centralized control
  - » lots of events, messages

Algorithm

- ganization is a hierarchy
  - e “workers”
  - en nodes are “managers”
  - replaced by a “committee”
- shown to work (Micros)



Hierarchical Algorithm (cont.)

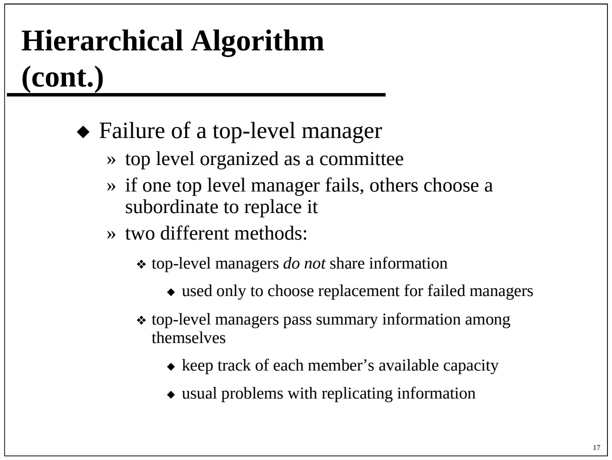
- Communication paths
  - » minimized by each level talking one level up or down the hierarchy
  - » when processor busy, send message to manager
    - ❖ manager can then propagate the message
    - ❖ can also summarize messages
- Manager nodes
  - » manage k processor (“worker”) nodes
    - ❖ or j manager nodes
  - » keep track of idle processors
    - ❖ no attempt at keeping track of down hosts
    - ❖ therefore count is an upper bound

Hierarchical Algorithm (cont.)

- Load sharing
    - jobs can be created at any level of the hierarchy
    - suppose a worker spawns a job needing *n* processes
        - need to find and allocate *n* processes
    - immediate manager notified of the request
        - manager knows of *w* workers available
        - if *w* " *n*, then manager reserves *n* processors
        - otherwise send request to next higher manager

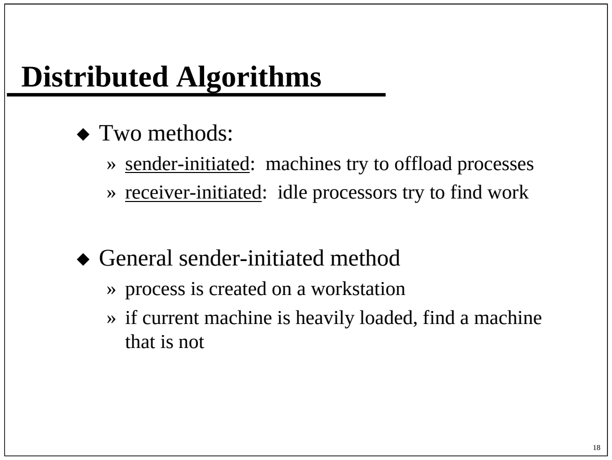
Hierarchical Algorithm (cont.)

- Failure of intermediate managers
    - superior node detects its failure
    - elects a subordinate of the intermediate manager to replace it
        - must get updated information from subordinates



Hierarchical Algorithm (cont.)

- Failure of a top-level manager
  - » top level organized as a committee
  - » if one top level manager fails, others choose a subordinate to replace it
  - » two different methods:
    - ❖ top-level managers *do not* share information
      - used only to choose replacement for failed managers
    - ❖ top-level managers pass summary information among themselves
      - keep track of each member’s available capacity
      - usual problems with replicating information



Distributed Algorithms

- Two methods:
  - » sender-initiated: machines try to offload processes
  - » receiver-initiated: idle processors try to find work

- General sender-initiated method
  - » process is created on a workstation
  - » if current machine is heavily loaded, find a machine that is not

Distributed Algorithms (cont.)

- Three variants of the general sender-initiated method (Eager et al.)
  - random migration
    - pick a machine at random
    - send process to that machine
    - repeat procedure (i.e. if that machine is loaded, send to another randomly chosen machine)
  - random probes
    - pick a machine at random
    - send probes until suitable machine found
    - try a max of *N* probes
  - probing *k* machines
    - probe *k* machines to get their exact load
    - send process to least loaded in the set

Distributed Algorithms (cont.)

- Analysis of the three algorithms
  - third algorithm (choose best of *k*) performs best
    - i.e. load balancing , fewest process migrations, etc.
    - but not the best *overall*
    - must factor in overhead from probes
    - gain from the algorithm too small to offset additional *k* probes
  - moral of the story: simple algorithms are preferred
    - overhead from complex algorithms often erase gains

Distributed Algorithms (cont.)

- Problems with sender-initiated distributed algorithms
  - incomplete information
    - A sends to B, thinking B has a light load
    - B sends to A, because in reality A's load is lighter
    - A sends to B...
  - heavily loaded system - all machines overloaded
    - therefore all machines will try to offload processes
    - probing won't accomplish anything (can't find a lightly loaded machine)
    - but additional overhead is incurred (when the system can least afford it)

Distributed Algorithms (cont.)

- Receiver-initiated distributed algorithm
  - when a process terminates, check load
    - if load is light, look for work
    - send probe to a machine, or *k* machines
    - stop if no work found after *N* probes
  - doesn't create traffic when system is overloaded
    - generates traffic when machines are lightly loaded
    - but what else do the machines have to do anyway?

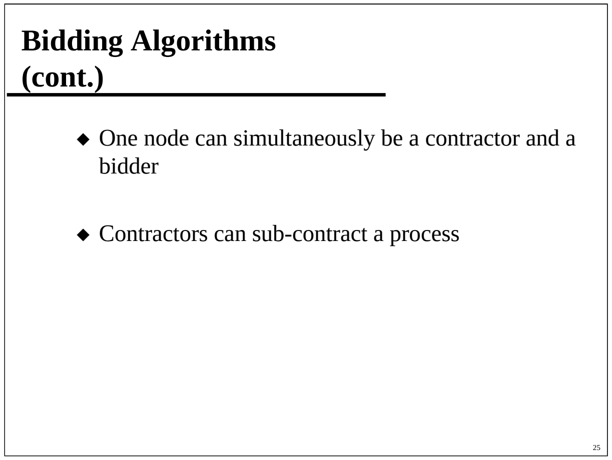
Distributed Algorithms (cont.)

- Combining approaches
  - » look for work when lightly loaded, offload work when heavily loaded
    - ❖ unclear what the dynamics would be
  - » keep histories of chronically under-loaded or over-loaded machines

---

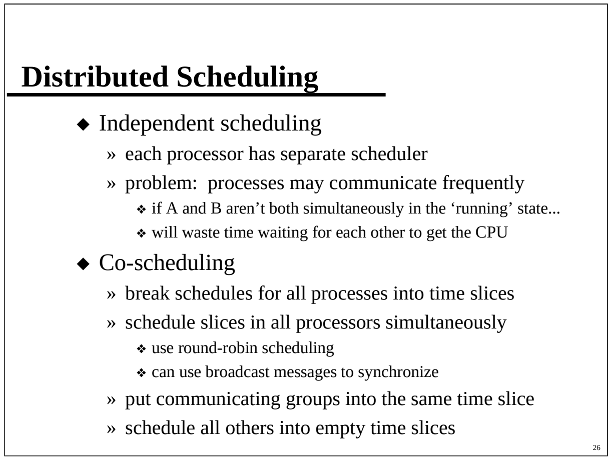
Bidding Algorithms

- Simulate contract bidding
  - » processes buy CPU time
  - » processors give cycles to highest bidder
- A three-step process:
  - » a new process needing CPU time is created
  - » a bid is constructed for the process
    - ❖ detailing the computational environment needed
    - ❖ CPU loading, queue & stack sizes, special I/O needs, floating point hardware, etc.
  - » processors receive bids, chooses highest set of bidders it can comfortably accommodate
    - ❖ must multi-cast contract to <u>all</u> bidders
    - ❖ Why?



Bidding Algorithms (cont.)

- One node can simultaneously be a contractor and a bidder
- Contractors can sub-contract a process



Distributed Scheduling

- Independent scheduling
  - each processor has separate scheduler
  - problem: processes may communicate frequently
    - if A and B aren’t both simultaneously in the ‘running’ state...
    - will waste time waiting for each other to get the CPU
- Co-scheduling
  - break schedules for all processes into time slices
  - schedule slices in all processors simultaneously
    - use round-robin scheduling
    - can use broadcast messages to synchronize
  - put communicating groups into the same time slice
  - schedule all others into empty time slices

Fault Tolerance

- Overall message
  - although fault tolerance is one of the reasons cited in favor of distributed systems
  - it’s *really*, *really*, hard to achieve
  - and not much research has been done!!
- Types of faults
  - fail-silent (or fail-stop)
    - processor just stops or machine crashes
    - easy to detect
  - Byzantine
    - processor/process continues to run, giving incorrect data
    - much harder to analyze and correct

Fault Tolerance

- Fault tolerance in distributed systems
  - transaction processing
    - Already discussed aborting transactions and two-phase commit
  - replication techniques
    - TMR (Triple Modular Redundancy)
      - A triplicated voter follows each stage in the circuit
      - Majority rules
    - Active replication (state machine approach)
      - extension of TMR

Replication Techniques

- Active replication
  - provide *n* processors and vote on output
  - for example: if 2 out of 3 are the same, use that result
  - problem: it takes lots of processors to achieve this
    - voters must also be treated as suspect
  - <u>k fault tolerant</u>: can survive k faulty components
    - for fail safe: k + 1 - just use the other processor
    - Byzantine failures: 2k + 1
  - another problem: all requests must be serviced in the same order (in voters and nodes)
    - <u>atomic broadcast problem</u>
    - note that only write operations in a file server need to be ordered

Replication Techniques (cont.)

- Primary backup
  - primary does the work...
    - sends work to backup for synchronization
  - general structure: if primary fails, use the backup
    - no ordering needed
    - requires fewer processors
  - but difficult to agree on when backup takes over in Byzantine failures

Two-Army Problem

- General problem
  - » two units of army (or file servers) need to coordinate a strike on the enemy
  - » communicate by messenger, but messenger may be captured (message lost)
  - » no matter how many acknowledgments, can never be sure that the last message was received

---

Byzantine Generals Problem

- General problem
  - » *n* generals (or file servers) need to coordinate a strike on the enemy
  - » communicate by telephone on perfect lines (reliable communication)
  - » *m* generals are traitors (faulty processors)
    - ❖ give incorrect and contradictory information
  - » generals must exchange troop strengths
    - ❖ each general will end up with a vector of length *n* containing troop strengths of each army
    - ❖ if general_i is loyal, element_i is his troop strength

Byzantine Generals Problem

- One (limited) solution
  - » each general sends a reliable message to all other generals, giving troop strength

1) Each general sends their troop strength
2) Each general collects numbers in a vector
3) Each general sends vector to all other generals

Byzantine Generals Example

- If any value has a majority, that is a true result, otherwise the value is unknown
- In the face of m faulty processors
  - » can be achieved only if 2m + 1 correctly functioning processors are present
  - » meaning you need 3m + 1 processors total