Distributed Processing

Three directions that traditional operating systems have gone in terms of processing:

1. support for threads
2. distributed processing
3. multiprocessor

Threads

Have talked about threads in OS, but review a little.

Also called *lightweight processes*. Contain an execution state within a shared address space.

Threads are natural to use for a server handling requests. Each request can be handled by a thread. Can also have multi-threaded clients for handling user interaction along with network and file I/O.

Threads vs. Processes

- threads are cheaper to create
- switching between threads in same process is much cheaper
- easier sharing of resources between threads
- lack of protection between threads within a process

Terminology among different systems:

Distributed OS kernel	Thread name	Exec. Env. Name
Amoeba	Thread	Process
Chorus	Thread	Actor
Mach	Thread	Task
V System	Process	Team
Unix	–	Process

System Models for Distributed Computing

Processor Pool Model

Amoeba. Best argument for using this approach comes from queueing theory. "Replacing n small resources by one big one that is n times more powerful, reduces the average response time n-fold.

Dedicated processors. Do not support user interaction. For example: Beowulf cluster of processors running Linux.

Increased use of *blades* that are plugged into a rack of machines.

Workstation Model

Every user has their own computer. Can share computing resources.

Can show both theoretically and practically that many idle nodes exist at any one time. Look at Fig 11.1 and 11.2 from Singhal.

Would like to use these idle nodes. What is idle?

- idle process
- no user logged on
- threshold on system parameters
- screen saver

What is performance we are trying to optimize?

- response time
- throughput (jobs through system)

Web Computing Cluster

A related model with some of the same issues is a cluster handling requests where a front-end *switch* is directing requests to a particular server.

Scheduling Issues

- load measure—resource queue lengths. Most common is CPU queue length. V-System uses the CPU utilization. Other measures—memory, file, display.
- Static/Dynamic, Adaptive then changes it how operates (may quit collecting status, or collect it less often).
- Load Balancing/Load Sharing. Higher overhead for load balancing because it tries to keep the loads equal (as opposed to just keeping processors busy).
- Preemptive/Non-preemptive transfers. Migration during execution implies passing state.
- Level of computation sharing. What is the type of remote service? A specific service or code execution.
- Data location. Where is the data located relative to the client and remote execution node? If there is a lot of data, it may make sense to move computation to the data if possible.

Components

- transfer policy—should a task be transferred? Usually based on a threshold.
- selection policy—which task to transfer. Next task, tasks that will take a long time (to justify migration costs).
- location policy—which node to transfer to. Use polling, multicasting or broadcasting.
- information policy—when to collect system state information
  - demand-driven—only collect information when needed.
  - periodic—non-adaptive
  - state-change-driven—only when state has changed

Examples

- Sender-Initiated
  - Look as needed—ELZ is a good example.
  - Centralized. Zhou. A central server keeps track of idle and busy nodes. All changes are reported to this node.
  - Buddy System. Shin and Chang. 10-20 workstations keep complete information amongst themselves.
- Receiver-Initiated—Idle nodes search out for tasks, leads to preemptive transfers since tasks often have received some amount of service.
- Symmetrically-Initiated—do both at once. Can have an upper and lower threshold. Each node tries to keep its load within an acceptable range. Does adjust its threshold.
- Adaptive Algorithms—can change thresholds, may want to do for sender-initiated at high loads. Keep track of responses from the probes so a cache of receivers, senders, and oks are kept at each node.

Co-Scheduling

co-scheduling or gang scheduling to get processes that are cooperating to run at the same time.

Load Sharing

Public Resource Computing

Receiver-initiated approach for Internet-wide scale.

- SETI—analyze data for signs of intelligence. Download a client.
- distributed.net—find encryption keys. Also download a client.
- distriblets project. Project at WPI where chunks of computations are downloaded as Java applets.

ELZ Paper

"Adaptive Load Sharing in Homogeneous Distributed Systems" by Eager, Lazowska and Zahorjan

Sender-initiated policies.

It is adaptive in that policies react to system state. Basic idea is to understand how relatively simple load sharing policies work. Problems with complex policies:

- complexity causes more overhead (hence increasing response time)
- can make poorer decisions if information is out of date.
- instability if decisions are too fine.

Identified a transfer policy (when to transfer) and a location policy (where to transfer).

Assume a system model of 20 homogeneous node with some processor cost assigned to a task transfer.

Exponentially distributed arrival rates and service times. Used an analytical study backed up by a simulation.

Policies: all try to transfer a new task if the queue length is greater than or equal to a threshold T.

- Random: pick another node and transfer task to that node. Have a transfer limit L_t.
- Threshold: probe nodes until a satisfactory one (using T) is found. Have a probe limit (3 found to be best) beyond which the task is kept locally.
- Shortest: probe a fixed number of nodes (probe limit(and select the best.

Look at primary results. Basically show that simple is good.

Wills and Finkel Work

"Load Sharing Using Multicasting"

Buddy policy by Shin and Chang is to broadcast state changes to buddy set of nodes

Node states: underloaded (eligible to receive tasks), medium, full-loaded (try to transfer tasks)

Policies:

- Multithresh: maintain a multicast group for lightly-loaded nodes. If a node is lightly loaded then join group. If overloaded then send to group and use first response. Initiator can get swamped with replieds.
- Multileader: Two multicast addresses—group and leader. On state changes, send message to group. If overloaded, send message to leader who returns a lightly loaded node. Must deal with movement and loss of leader.

Look at results. Can also look at scaling.

Implemented as MQP on WPI's Beowulf cluster. System is called PANTS (PANTS Application Node Transparency System).