Distributed Processing

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

1. support for threads (light-weight processes)
2. distributed processing
3. multiprocessor

Threads

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

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:

<table>
<thead>
<tr>
<th>Distributed OS kernel</th>
<th>Thread name</th>
<th>Exec. Env. Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Amoeba</td>
<td>Thread</td>
<td>Process</td>
</tr>
<tr>
<td>Chorus</td>
<td>Thread</td>
<td>Actor</td>
</tr>
<tr>
<td>Mach</td>
<td>Thread</td>
<td>Task</td>
</tr>
<tr>
<td>V System</td>
<td>Process</td>
<td>Team</td>
</tr>
<tr>
<td>Unix</td>
<td>–</td>
<td>Process</td>
</tr>
<tr>
<td>Linux (clone())</td>
<td>Shared Memory Task</td>
<td>Task</td>
</tr>
</tbody>
</table>

Can have either user-level or kernel-level threads.
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 runnin Linux.

Workstation Model

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

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

What is performance we are trying to optimize?

- response time
- throughput (jobs through system)
**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 ok's are kept at each node.

Co-Scheduling

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

Requirements for Load Sharing

• scalability—how large can the mechanism scale?

• location transparency—hide distributed nature

• determinism—works the same in all locations

• preemption—be able to move away from workstations that are no longer idle.

• heterogeneity—should know particular hardware of machines.

Task Migration

State transfer and then unfreeze

Load Sharing

Examine ELZ paper. Wills/Finkel work.
Memory Management

Look at memory management on a multiprocessor and a distributed system.

Multiprocessor

Two models

1. Shared memory, uniform access
   Look at cache affinity paper for processor scheduling.

2. Non-uniform memory.

Hyper-threading

Intel Pentium 4 architecture feature. One CPU, but this CPU can be shared amongst multiple threads in some cases where it would otherwise be idle.

Processor might be stalled due to cache miss, branch misprediction, etc.

Appears as logical multiprocessor to the Operating System, although the two processors are clearly not equal as one gets opportunity to run only when the other is idle.

Server Scheduling

“Using Cohort Scheduling to Enhance Server Performance.” Slides:
http://www.cs.wpi.edu/~cs535/f05/larus:usenix02/

Effect of Clock Resolution on Scheduling

“Using Cohort Scheduling to Enhance Server Performance.” Slides:
http://www.cs.wpi.edu/~cs535/f05/etsion:sigmetrics03.pdf