Outline for Today

• Advanced topics in scheduling
• Some final examples in concurrent programming
• Next time: Start memory management
• Issues?

Scheduling: Beyond “Ordinary” Uniprocessors

• Multiprocessors
  – Co-scheduling and gang scheduling
  – Hungry puppy task scheduling
  – Load balancing
• Networks of Workstations
  – Harvesting Idle Resources - remote execution and process migration
• Laptops and mobile computers
  – Power management to extend battery life, scaling processor speed/voltage to tasks at hand, sleep and idle modes.
RR and System Throughput

On a *multiprocessor*, RR may improve throughput under light load:

*The scenario:* three salmon steaks must cook for 5 minutes per side, but there’s only room for two steaks on the hibachi.

30 minutes worth of grill time needed: steaks 1, 2, 3 with sides A and B.

**FCFS:** steaks 1 and 2 for 10 minutes, steak 3 for 10 minutes.

Completes in 20 minutes with grill utilization a measly 75%.

RR and System Throughput

**RR:** 1A and 2A...flip...1B and 3A...flip...2B and 3B.

Completes in three quanta (15 minutes) with 100% utilization.

RR may speed up parallel programs if their inherent parallelism is poorly matched to the real parallelism.

E.g., 17 threads execute for \( N \) time units on 16 processors.
Multiprocessor Scheduling

What makes the problem different?

• Workload consists of parallel programs
  – Multiple processes or threads, synchronized and communicating
  – Latency defined as last piece to finish.

• Time-sharing and/or Space-sharing (partitioning up the Mp nodes)
  – Both when and where a process should run

Architectures

Symmetric mp

NUMA

Node 1
Node 2
Node 3
Effect of Workload

Impact of load-balancing on latency
Consider set of processes: 5, 5, 4, 4, 3, 3
3 processors
- If unrelated: (SJF)
  \[
  \text{avg response time} = \frac{3 + 3 + 4 + 7 + 8 + 9}{6} = 5.66
  \]
- If 2 tasks, each 5, 4, 3 (no dependencies)
  \[
  \text{avg response time (SPF)} = \frac{8 + 9}{2} = 8.5
  \]
  \[
  \text{avg response time (LPF)} = \frac{8 + 8}{2} = \text{Red done}
  \]

Affinity Scheduling

- Where (on which node) to run a particular thread during the next time slice?
- Processor’s POV: favor processes which have some residual state locally (e.g. cache)
- What is a useful measure of affinity for deciding this?
  - Least intervening time or intervening activity (number of processes here since “my” last time) *
  - Same place as last time “I” ran.
  - Possible negative effect on load-balance.
Processor Partitioning

- Static or Dynamic
- Process Control (Gupta)
  - Vary number of processors available
  - Match number of processes to processors
  - Adjusts # at runtime.
  - Works with task-queue or threads programming model
  - Impact on “working set”

Process Control Claims
Typical speed-up profile

- Lock contention, memory contention, context switching, cache corruption
- Magic point
- Number of processes per application
Co-Scheduling

John Ousterhout (Medusa OS)
- Time-sharing model
- Schedule related threads simultaneously

Why?
- Local scheduling decisions after some global initialization (Medusa)
- Centralized (SGI IRIX)

Effect of Workload

Impact of communication and cooperation

Issues:
- context switch
+ common state
- lock contention
+ coordination
**CM*’s Version**

- Matrix $S$ (slices) $\times P$ (processors)
- Allocate a new set of processes (task force) to a row with enough empty slots
- Schedule: Round robin through rows of matrix
  - If during a time slice, this processor’s element is empty or not ready, run some other task force’s entry in this column - backward in time (for affinity reasons and purely local “fall-back” decision)

**Design**

- Determining how many processors an application should have
  - Centralized server, with system calls for process status info (user-level) (kernel implementation would be desirable, but…)
- Controlling the number of processes in an application
  - suspend and resume are responsibility of runtime package of application
Networks of Workstations

What makes the problem different?
• Exploiting otherwise “idle” cycles.
• Notion of ownership associated with workstation.
• Global truth is harder to come by in wide area context

Harvesting Idle Cycles

• Remote execution on an idle processor in a NOW (network of workstations)
  – Finding the idle machine and starting execution there. Related to load-balancing work.
• Vacating the remote workstation when its user returns and it is no longer idle
  – Process migration
**Issues**

- Why?
- Which tasks are candidates for remote execution?
- Where to find processing cycles? What does “idle” mean?
- When should a task be moved?
- How?

**Motivation for Cycle Sharing**

- Load imbalances. Parallel program completion time determined by slowest thread. *Speedup* limited.
- Utilization. In trend from shared mainframe to networks of workstations $\rightarrow$ scheduled cycles to statically allocated cycles
  - “Ownership” model
  - Heterogeneity
Which Tasks?

- Explicit submission to a “batch” scheduler (e.g., Condor) or Transparent to user.
- Should be demanding enough to justify overhead of moving elsewhere. Properties?
- Proximity of resources.
  - Example: move query processing to site of database records.
  - Cache affinity

Finding Destination

- Defining “idle” workstations
  - Keyboard/mouse events? CPU load?
- How timely and complete is the load information (given message transit times)?
  - Global view maintained by some central manager with local daemons reporting status.
  - Limited negotiation with a few peers
  - How binding is any offer of free cycles?
- Task requirements must match machine capabilities
When to Move

• At task invocation. Process is created and run at chosen destination.
• Process migration, once task is already running at some node. State must move.
  – For adjusting load balance (generally not done)
  – On arrival of workstation’s owner (vacate, when no longer idle)

How - Negotiation Phase

• Condor example: Central manager with each machine reporting status, properties (e.g. architecture, OS). Regular match of submitted tasks against available resources.
• Decentralized example: select peer and ask if load is below threshold. If agreement to accept work, send task. Otherwise keep asking around (until probe limit reached).
How - Execution Phase

• Issue - Execution environment.
  – File access - possibly without user having account on destination machine or network file system to provide access to user’s files.
  – UIDs?
• Remote System Calls (Condor)
  – On original (submitting) machine, run a “shadow” process (runs as user)
  – All system calls done by task at remote site are “caught” and message sent to shadow.

Remote System Calls

<table>
<thead>
<tr>
<th>Submitting machine</th>
<th>Executing machine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shadow</td>
<td>Remote Job</td>
</tr>
<tr>
<td>Remote syscall code</td>
<td>User code</td>
</tr>
<tr>
<td>Regular syscall stubs</td>
<td>Remote syscall stubs</td>
</tr>
<tr>
<td>OS Kernel</td>
<td>OS Kernel</td>
</tr>
</tbody>
</table>
How - Process Migration

Checkpointing current execution state (both for recovery and for migration)

- Generic representation for heterogeneity?
- Condor has a checkpoint file containing register state, memory image, open file descriptors, etc. Checkpoint can be returned to Condor job queue.
- Mach - package up processor state, let memory working set be demand paged into new site.
- Messages in-flight?

Idleness is Powerful

Transition diagram:
- High power cost × time (Busy)
- ? (transition)
- ? (transition)
- Low power × time cost (Idle)
Dynamic Voltage Scaling

The question: at what clock rate/voltage should the CPU run in the next scheduling interval?

- Voltage scalable processors
  - StrongARM SA-2 (500mW at 600MHz; 40mW at 150MHz)
  - Speedstep Pentium III
  - AMD Mobile K6 Plus
  - Transmeta
- Power is proportional to $V^2 \times F$
- Energy will be affected
  (+) by lower power,
  (-) by increased time

Interval Scheduling
(adjust clock based on past window, no process reordering involved)

- Weiser et. al.
- Algorithms (when):
  - Past
  - AVG
- Stepping (how much)
  - One
  - Double
  - Peg – min or max
- Based on unfinished work during previous interval
Implementation of Voltage Scheduling Algorithms

Issues:
- Capturing *utilization* measure
  - Start with no a priori information about applications and need to dynamically infer / predict behavior (patterns / “deadlines” / constraints?)
  - Idle process or “real” process – *usually* each quantum is either 100% idle or busy
  - $\text{AVG}_N$: weighted utilization at time $t$
    $$W_t = \frac{NW_{t-1} + U_{t-1}}{N+1}$$
- Adjusting the clock speed
  - Idea is to set the clock speed sufficiently high to meet deadlines (but deadlines are not explicit in algorithm)

Based on Earliest Deadline First

- Dynamic algorithm
- Priorities are assigned to tasks according to the deadlines of their current request
- With EDF there is no idle time prior to an overflow
- For a given set of $m$ tasks, EDF is feasible iff
  $$C_1/T_1 + C_2/T_2 + \ldots + C_m/T_m \leq 1$$
- If a set of tasks can be scheduled by any algorithm, it can be scheduled by EDF
lpARM System

- Speed-control register
- Processor cycle ctrs
- System sleep control

Figure 2: lpARM System Block Diagram

Intuition

\[ C_1 = 1 \]

\[ T_1 \]

\[ T_i \]

\[ C_2 = 1 \]

\[ T_2 \]

\[ \tau_1 \]

\[ \tau_2 \]

time
Intuition

EventBarrier

- EventBarrier has a binary “memory”.
- It has a “broadcast” to notify all waiting threads of an event.
- The broadcast primitive waits until the event is handled.

EventBarrier::Wait()

If the EventBarrier is not in the signaled state, wait for it.

EventBarrier::Signal()

Signal the event, and wait for all waiters/arrivals to respond.

EventBarrier::Complete()

Notify EventBarrier that caller’s response to the event is complete. Block until all threads have responded to the event.
EventBarrier Example

EventBarrier channel;

void OutputThread { 
    while (TRUE) { 
        ComputeDataToSend();
        channel.Wait();
        SendData();
        channel.Complete();
    }
}

void ChannelScheduler() { 
    while (TRUE) { 
        WaitUntilTimeToOpenChannel();
        channel.Signal();  /* open floodgate for burst of outgoing data */
        /* channel is closed */
    }
}

**Invariants:**
1. Output thread never blocks in Wait() if the channel is already open.
2. Channel never closes while a thread is sending data.
3. Each thread sends at most once each time the channel opens.
**Highway 110 with EventBarrier**

EventBarrier north;
EventBarrier south;

void HeadingNorth {
    north.Wait();
    go across one-lane bridge;
    north.Complete();
}

void HeadingSouth() {
    south.Wait();
    go across one-lane bridge;
    south.Complete();
}

void BridgeScheduler() {
    while (TRUE) {
        north.Signal();
        south.Signal();
        ....
    }
}

**Highway 110 with Locks and Condition Variables**

OneVehicle(int direc) //direc is either 0 or 1; giving the direction in which the car is to cross

{ 
    ArriveBridge(direc);
    CrossBridge(direc);
    ExitBridge(direc);
}

ArriveBridge(dir)
{
    bridgeLock->Acquire();
    while (num_on_bridge !=0 &
          dir != direction)
        OKtogo[dir]->Wait();
    direction = dir;
    num_on_bridge++;
    bridgeLock->Release();
}

ExitBridge(dir)
{
    bridgeLock->Acquire();
    num_on_bridge--;
    if (num_on_bridge == 0) {
        direction = !dir;
        Oktogo[direction]->Broadcast();
        bridgeLock->Release();
    }
}

Variation of this problem has “load limit” restriction of 3 cars at a time

Fairness?