Outline

• Objective for today’s lecture:
  Advanced topics in scheduling

Real Time Schedulers

• Real-time schedulers must support regular, periodic execution of tasks (e.g., continuous media).
  – CPU Reservations
    • “I need to execute for $X$ out of every $Y$ units.”
    • Scheduler exercises admission control at reservation time: application must handle failure of a reservation request.
  – Proportional Share
    • “I need $1/n$ of resources”
  – Time Constraints
    • “Run this before my deadline at time $T$.”

Assumptions

• Tasks are periodic with constant interval between requests, $T_i$ (request rate $1/T_i$)
• Each task must be completed before the next request for it occurs
• Tasks are independent
• Worst-case run-time for each task is constant (max), $C_i$
• Any non-periodic tasks are special
Definitions

- **Deadline** is time of next request
- **Overflow** at time \( t \) if \( t \) is deadline of unfulfilled request
- **Feasible** schedule - for a given set of tasks, a scheduling algorithm produces a schedule so no overflow ever occurs.
- **Critical instant** for a task - time at which a request will have largest response time.
  - Occurs when task is requested simultaneously with all tasks of higher priority
Rate Monotonic

- Assign priorities to tasks according to their request rates, independent of run times.
- Optimal in the sense that no other fixed priority assignment rule can schedule a task set which cannot be scheduled by rate monotonic.
- If feasible (fixed) priority assignment exists for some task set, rate monotonic is feasible for that task set.

Rate Monotonic

\[
\begin{align*}
\tau_1 & \quad C_1 = 1 \\
\tau_2 & \quad T_2 = 2 \\
\end{align*}
\]

\[
\begin{align*}
\tau_1 & \quad C_1 = 1 \\
\tau_2 & \quad T_2 = 5 \\
\end{align*}
\]

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 \( \frac{C_1}{T_1} + \frac{C_2}{T_2} + \ldots + \frac{C_m}{T_m} \leq 1 \)
- If a set of tasks can be scheduled by any algorithm, it can be scheduled by EDF
Proportional Share

- Goals: to integrate real-time and non-real-time tasks, to police ill-behaved tasks, to give every process a well-defined share of the processor.
- Each client, $i$, gets a weight $w_i$
- Instantaneous share $f_i(t) = \frac{w_i}{\sum w_j}$
- Service time ($f_i$ constant in interval)
  $S_i(t_0, t_1) = f_i(t) \Delta t$
- Set of active clients varies $\rightarrow f_i$ varies over time
  $S_i(t_0, t_1) = \int_{t_0}^{t_1} f_i(t) \, dt$

Common Proportional Share Competitors

- Weighted Round Robin – RR with quantum times equal to share

- Fair Share – adjustments to priorities to reflect share allocation (compatible with multilevel feedback algorithms)
Common Proportional Share Competitors

- Weighted Round Robin – RR with quantum times equal to share
  \[ RR: \]
  \[ WRR: \]
- Fair Share – adjustments to priorities to reflect share allocation (compatible with multilevel feedback algorithms)

Lottery Scheduling

- Lottery scheduling [Waldspurger96] is another scheduling technique.
  - Elegant approach to periodic execution, priority, and proportional resource allocation.
  - Give \( W_p \) "lottery tickets" to each process \( p \).
    - GetNextToRun selects "winning ticket" randomly.
      - If \( \sum W_p = N \) then each process gets CPU share \( W_p/N \) probabilistically, and over a sufficiently long time interval.
  - Flexible: tickets are transferable to allow application-level adjustment of CPU shares.
  - Simple, clean, fast.
    - Random choices are often a simple and efficient way to produce the desired overall behavior (probabilistically).
Lottery Scheduling
Waldspurger and Weihl (OSDI 94)

- Goal: responsive control over the relative rates of computation
- Claims:
  - Support for modular resource management
  - Generalizable to diverse resources
  - Efficient implementation of proportional-share resource management: consumption rates of resources by active computations are proportional to relative shares allocated

Basic Idea

- Resource rights are represented by lottery tickets
  - Give \( W_p \) “lottery tickets” to each process \( p \).
  - Abstract, relative (vary dynamically wrt contention), uniform (handle heterogeneity)
  - Responsiveness: adjusting relative # tickets gets immediately reflected in next lottery
- At allocation time: hold a lottery; Resource goes to the computation holding the winning ticket.
  - \( \text{GetNextToRun} \) selects “winning ticket” randomly..

Fairness

- Expected allocation is proportional to # tickets held - actual allocation becomes closer over time.
- Number of lotteries won by client
  \[ E[w] = np \quad \text{where} \quad p = \frac{t}{T} \]
- Response time (# lotteries to wait for first win)
  \[ E[n] = \frac{1}{p} \]

\( w \): # wins
\( \ell \): # tickets
\( T \): total # tickets
\( n \): # lotteries
Example List-based Lottery

\[ T = 20 \]

\[
\begin{array}{cccc}
10 & 2 & 5 & 1 & 2 \\
\text{Summing:} & 10 & 12 & 17 \\
\end{array}
\]

Random(0, 19) = 15

Bells and Whistles

- Ticket transfers - objects that can be explicitly passed in messages
  – Can be used to solve priority inversions
- Ticket inflation
  – Create more - used among mutually trusting clients to dynamically adjust ticket allocations
- Currencies - "local" control, exchange rates
- Compensation tickets - to maintain share
  – use only \( f \) of quantum, ticket inflated by \( 1/f \) in next

Kernel Objects

\[
\begin{array}{c}
\text{1000 base} \\
\text{ticket} \\
\end{array}
\]

\[
\begin{array}{c}
\text{amount} \\
\text{currency} \\
\end{array}
\]

\[
\begin{array}{c}
\text{C_name} \\
\end{array}
\]

\[
\begin{array}{c}
\text{300} \\
\end{array}
\]

\[
\begin{array}{c}
\text{Issued tickets} \\
\end{array}
\]

\[
\begin{array}{c}
\text{Backing tickets} \\
\text{Currency name} \\
\text{Active amount} \\
\end{array}
\]
Exchange rate:
1 bob = 20 base

Example List-based Lottery

T = 3000 base

\begin{center}
\begin{tabular}{c|c|c|c}
 & 10 & 2bob & 5 task3 \\
\hline
 Task2 & & & \\
\hline
 Task3 & & & \\
\hline
 Base & 2bob & & \\
\end{tabular}
\end{center}

Random(0, 2999) = 1500

Compensation

- A holds 400 base, B holds 400 base
- A runs full 100msec quantum, B yields at 20msec
- B uses 1/5 allotted time
  Gets compensation ticket valued at
  \( \frac{400}{1/5} = 2000 \) base at next lottery
Ticket Transfer

- Synchronous RPC between client and server
- Create ticket in client’s currency and send to server to fund its currency
- On reply, the transfer ticket is destroyed

Control Scenarios

- Dynamic Control
  Conditionally and dynamically grant tickets
  Adaptability
- Resource abstraction barriers supported by currencies. Insulate tasks.

Other Kinds of Resources

Control relative waiting times for mutex locks.
- Mutex currency funded out of currencies of waiting threads
- Holder gets inheritance ticket in addition to its own funding, passed on to next holder (resulting from lottery) on release.
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 79%.

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 8 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

Effect of Workload

Impact of load-balancing on latency
Consider set of processes: 5, 5, 4, 4, 3, 3
3 processors
• If unrelated: (SJF)
  avg response time = (3 + 3 + 4 + 7 + 8 + 9)/6 = 5.66
• If 2 tasks, each 5, 4, 3 (no dependencies)
  avg response time (SPF) = (8 + 9)/2 = 8.5
  avg response time (LPF) = (8 + 8)/2 = 8
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) x 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 → 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

Submitting machine
- Shadow
  - Remote syscall code
  - Regular syscall stubs
- OS Kernel

Executing machine
- Remote Job
  - User code
  - Remote syscall stubs
- OS Kernel

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?
Migration based on Virtual Machines

What is a virtual machine monitor (VMM) – example: VMware

- VMware Workstation acts as a “true” virtual monitor
- Provides direct access to processor and memory
- Fast execution of instructions
- Not processor emulation
- Shares the processor with the host (host does scheduling)

Isolation Between V.M.s

State that must be migrated is well-defined

Encapsulation

Encapsulation means portability. Because the virtual machine is basically just a file, it is easily transportable between machines using VMware Workstation (assuming they meet our HW requirements)

- Portability across different HW systems
- Standard virtual hardware environment
- Undeletable and non-persistent disk mode
Processor Scheduling & Energy Management

- Processor vendors keep making chips faster (and more power hungry)
- Premise: sometimes we need that peak performance but for lots of applications, we don’t
- Solution: sell the chip based on peak MHz and then let it run at slower/lower power design point

Exploiting Idleness

<table>
<thead>
<tr>
<th>State</th>
<th>High power cost</th>
<th>Low power cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Busy</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>Idle</td>
<td>transition</td>
<td>transition</td>
</tr>
</tbody>
</table>

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: (600mW 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
IpARM System

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

![IpARM System Block Diagram](image)

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
  - \( \text{AVG}_t = \frac{(N*W_t - U_{t-1})}{(N-1)} \)
- Adjusting the clock speed
  - Idea is to set the clock speed sufficiently high to meet deadlines (but deadlines are not explicit in algorithm)

Interval Scheduling (Speed Setting)

- Adjust clock based on past window, no process reordering involved
- Based on unfinished work during previous interval
- Per-task basis may be more successful

- Weiser et. al.
- Algorithms (when):
  - Past
  - AVG
  - Stepping (how much)
    - One
    - Double
    - Peg — min or max

- Per-task basis may be more successful
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
  \[ \frac{C_1}{T_1} + \frac{C_2}{T_2} + \ldots + \frac{C_m}{T_m} \leq 1 \]
- If a set of tasks can be scheduled by any algorithm, it can be scheduled by EDF

Intuition

\[
\begin{align*}
\tau_1 & \quad C_1 = 1 \\
T_1 & \quad T_1 \\
\tau_2 & \quad T_2 \quad C_2 = 1 \\
\end{align*}
\]

Intuition

\[
\begin{align*}
\tau_1 & \quad C_1 = 1 \\
T_1 & \quad T_1 \\
\tau_2 & \quad T_2 \quad C_2 = 1 \\
\end{align*}
\]
**Recent Advances**

- Automatically determining "deadlines" — inferring periodic behavior from User Interaction events and IPC (Vertigo)

---

**Intuition**

\[
C_1 = 1, C_2 = 1 = 10 \times 1/7's \\
\frac{1}{2} + \frac{1}{5} = \frac{7}{10} \text{ of max speed}
\]

---

**Intuition**

\[
C_1 = C_2 = 1 = 10 \times 1/7's \\
\text{at } \frac{7}{10} \text{ speed} \\
\frac{1}{2} + \frac{1}{5} = \frac{7}{10} \text{ of max speed}
\]

---