Operating System Resource Management

Burton Smith
Technical Fellow
Microsoft Corporation

Background

- Resource Management (RM) is a primary operating system responsibility
  - It lets competing applications share a system
- Client RM in particular faces new challenges
  - Increasing numbers of cores (hardware threads)
  - Emergence of parallel applications
  - Quality of Service (QoS) requirements
  - The need to manage power and energy

Tweaking current practice is clearly not enough

Conventional OS Thread Scheduling

- The kernel maintains queues of runnable threads
  - One queue per priority per core, for example
- A core chooses a thread from the head of its nonempty queue of highest priority and runs it
- The thread runs for a “time quantum” unless it blocks or a higher priority thread becomes runnable
- Thread priority can change at scheduling boundaries
- The new priority is based on what just happened:
  - Unblocked from I/O (UI, storage , network)
  - Preempted by a higher priority thread
  - Quantum expired
  - New thread creation
  - *etc…*

Shortcomings

- Kernel thread blocking is expensive
  - It incurs a needless change in protection
  - User-level thread blocking is much cheaper
- Kernel thread progress is unpredictable
  - This has made non-blocking synchronization popular
- Processes have little to say about core allocations
  - but processes play a big role in memory management
- Service Level Agreements are difficult to ensure
  - Priority is not a reliable determiner of performance
- Power and energy are not connected with priority

Current practice can’t address the new challenges

A Way Forward

- Resources should be allocated to processes
  - Cores of various types
  - Memory (working sets)
  - Bandwidths, *e.g.* to shared caches, memory, storage and interconnection networks
- The OS should:
  - Optimize the responsiveness of the system
  - Respond to changes in user expectations
  - Respond to changes in process requirements
  - Maintain resource, power and energy constraints

What follows is a scheme to realize this plan

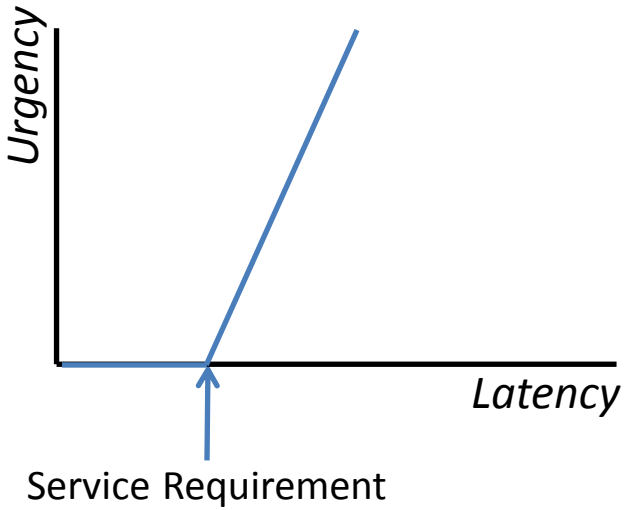
Latency

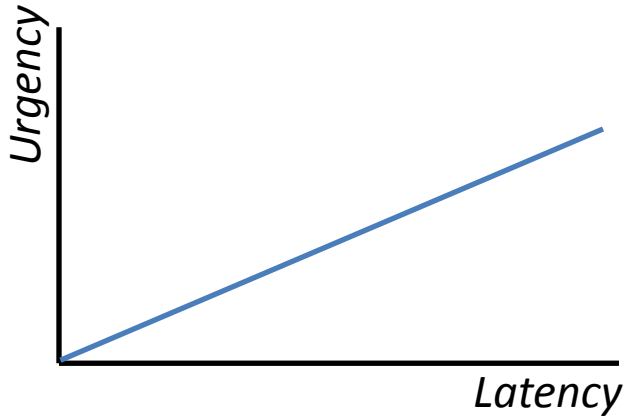
- Latency determines process responsiveness
  - The time from a mouse click to its result
  - The time from a service request to its response
  - The time from job launch to job completion
  - The time to execute a specified amount of work
- The relationship is usually a nonlinear one
  - Achievable latencies may be needlessly fast
  - There is usually a threshold of acceptability
- Latency depends on the allocated resources
  - Some resources will have more effect than others
  - Effects will often vary with computational phase

Urgency

- The *urgency function* of a process defines how latency translates into responsiveness
  - Its shape expresses the nonlinearity of the relationship
  - The shape will depend on the application and on the current user interface state (*e.g.* minimized)
- We let *total urgency* be the instantaneous sum of the current urgencies of the running processes
  - Resources determine latencies determine urgencies
- Assigning resources to processes to minimize total urgency maximizes system responsiveness

Urgency Function Examples



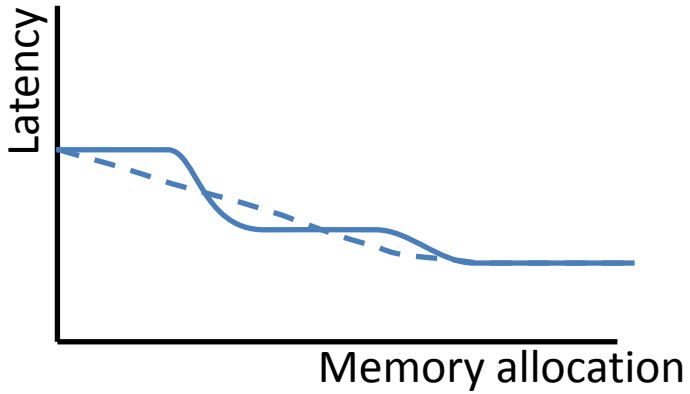


Manipulating Urgency Functions

- Urgency functions are like priorities, except:
  - They apply to processes, not kernel threads
  - They are explicit functions of process latency
- The User Interface can adjust their slopes
  - Up or down based on user behavior or preference
  - The deadlines can probably be left alone
- Total urgency is easy to compute in the OS given the process latencies
  - Its objective is to minimize it

Latency Functions

- Latency will generally decrease with resources
  - Latency increase as cores are added can be avoided by fixed-overhead parallel decomposition
  - Second derivatives will typically be non-negative
  - Unfortunately, sometimes we have “plateaus”:



- We will assume any “plateaus” are ignorable

Determining Latency Functions

- Latency depends on the allocated resources
  - It also depends on internal application state
- Unlike utility, latency must be measured
  - By the OS, by a user-level runtime, or both
  - The user-level runtime can suggest resource changes based on dynamic application data
  - Either could predict latency based on history

Corporate Resource Management

- The CEO owns the resources: people, space, …
  - Activities are expected to meet performance targets
  - Targets may change based on customer demand
  - Just-In-Time Agreements also constrain performance
  - The CEO optimizes total return across activities
- The activities ask for and compete for the resources
  - Their needs may change as their work progresses
- The total available resources are bounded
  - Surplus can be laid off/leased out, helping cash flow
- Cash on hand must not fall too low
  - If it does, some activities might need to be put on hold

Computer Resource Management

- The **OS** owns the resources: **cores**, **memory**, …
  - **Processes** are expected to meet performance targets
  - Targets may change based on customer demand
  - **Service Level** Agreements also constrain performance
  - The **OS** optimizes total **urgency** across **processes**
- The **processes** ask for and compete for the resources
  - Their needs may change as their work progresses
- The total quantity of available resources is bounded
  - Surplus can be **powered off**, helping **power consumption**
- **Battery energy** must not fall too low
  - If it does, **some processes** might need to be put on hold

RM As An Optimization Problem

Continuously minimize $\sum_{p \in P} U_p(L_p(a_{p,0}, \ldots a_{p,n-1})$ with respect to the resource allocations $a_{p,r}$, where

- P, U_p, L_p, and the $a_{p,r}$ are all time-varying;
- P is the index set of runnable processes;
- The urgency U_p depends on the latency L_p;
- L_p depends in turn on the allocated resources $a_{p,\mathrm{r}}$;
- $a_{p,r} \geq 0$ is the allocation of resource r to process p;
- $\sum_{p \in P} a_{p,r} = A_r$, the available quantity of resource r.
  - All slack resources are allocated to process 0

Convex Optimization

- A convex optimization problem has the form:

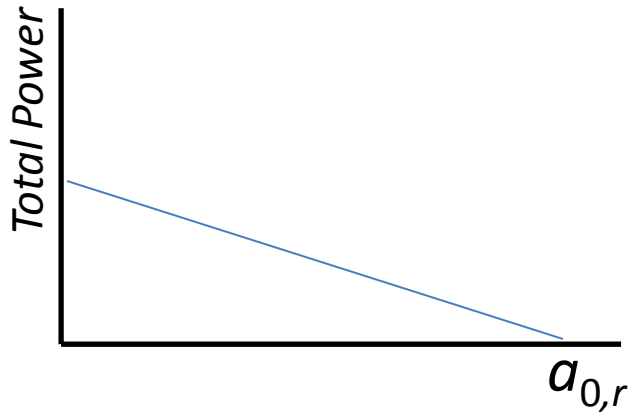
  Minimize $f_0(x_1, \ldots x_m)$

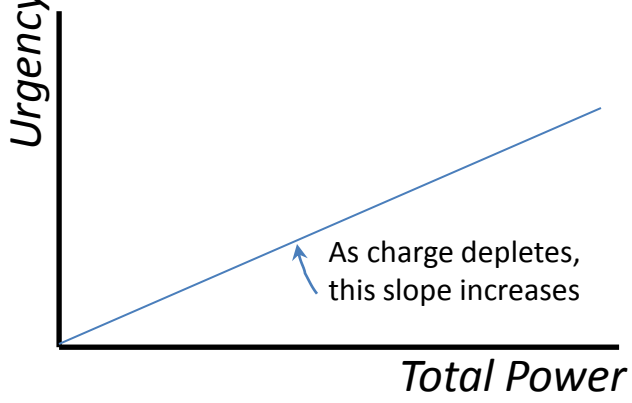
  subject to $f_i(x_1, \ldots x_m) \leq 0$, $i = 1, \ldots k$

  where the functions $f_i : \mathrm{R}^m \rightarrow \mathrm{R}$ are all convex
- Convex optimization has several virtues
  - It guarantees a single global extremum
  - It is not much slower than linear programming
- RM is a convex optimization problem

Managing Power and Energy

- System power W can be limited by an affine constraint $\sum_{p \neq 0} \sum_r w_r \cdot a_{p,r} \leq W$
- Energy can be limited using U_0 and L_0
  - Assume all slack resources $a_{0,r}$ are powered off
  - L_0 is defined to be the *total system power*
    - It will be convex in each of the slack resources $a_{0,r}$
  - U_0 has a slope that depends on the battery charge
    - Low-urgency work loses to P_0 when the battery is depleted





Obtaining Derivatives

- The gradient of the objective function tells us "which way is down", thus enabling descent
- Recall the chain rule: $\partial U/\partial a_r = \partial U/\partial L \cdot \partial L/\partial a_r$
- The urgency functions are no problem, but the latency functions are another matter
  - The user runtime can suggest estimates
  - The OS might try to add or remove a small δa_r
  - Historical data can be used if the process has the same characteristics (*e.g.* is in the same "phase")
  - For this last idea machine learning might help

An Example

Prototype Schedules

- The OS can maintain a “prototype” schedule
  - As events occur, it can be perturbed
  - It forms a good initial feasible solution
- Processes with SRs can be left alone so long as their urgency when invoked remains low
  - There is usually an associated fixed frame rate
  - The controlling urgency functions have two states
- Resources can be held in reserve if necessary
  - To avoid the overhead of repurposing them
  - They can be parked in an idle process (*e.g.* 0) with an urgency function that tends to keep them there

Conclusions

- RM faces new challenges, especially on clients
- RM can be cast as convex optimization to help address these challenges
- This idea is usable at multiple levels:
  - Between an OS and its processes
  - Between a hypervisor and its guest OSes
  - Between a process and its subtasks
- Estimating latency as a function of resources becomes an important part of the story