REAL-TIME VS. SYSTEMS

MICHAEL ROITZSCH
system whose quality depends on the **functional correctness** of computations and the **time** those results are produced
real-time school
mathematically sound
simplifying assumptions
random task sets
strong contract
theory

systems school
practical systems
baroque details
actual applications
usable interface
implementation

- scheduling was part of plant operation
- work items were scheduled to machines
- algorithms were known, but not by today’s names

- periodic task model
- first description of RMS and EDF
- including the utilization bounds
- with proof

- introduced the idea of tracking actual execution time to police overruns
- the reservation concept was born
- implemented in RT-Mach

- time as a first-class, globally managed shared resource
- based on the periodic task model
- extended with inheritance, reservations
DISASTER STRIKES

- the 90’s: multimedia driving practical real-time work
- this could have been a wonderful world, but then...

... CPUs got fast.

- “server” as a real-time concept
- allocation of time with period and budget
- jobs enqueued to be run by the server
- deadlines postponed on budget overrun
- illusion of a dedicated, slower processor
- virtual fluid-flow of CPU time
- individual job deadlines hard to manage

- "slack time" ≈ nothing urgent to do
- redirect slack to help out on overruns

- adapt server budget to varying application demand
- control loop observes tasks, QoS manager tunes parameters
4.2. Control architecture

As we take measurements upon the termination of each job, we do not assume any fixed sampling period. In fact, our feedback scheme is based on a discrete event model. Moreover, a system-wide notion of 'sampling' is entirely missing, as the sampling events are asynchronous for the different tasks. These considerations dictate a decentralized control scheme (Figure 2), where to each task is attached a dedicated task controller that is responsible for maintaining the task QoS level within specified bounds with the minimum CPU utilization. Still, the total bandwidth requests from the different task controllers are allowed to exceed the bound in Equation (1). To handle this situation (henceforth referred to as overload), a supervisor component is used to compute the actual bandwidth allocations that allow one to respect the QoS requirements of the various classes of tasks.

One final requirement is that the workload due to the control components themselves needs to be very low. This rules out such design approaches as dynamic programming or model predictive control, which feature excellent performance at the price of unacceptable computation resources.

4.3. Task controller

At a task controller comprise so softw oc compone nts: a feedback controller and a predictor. At the termination of each job $J_k$, sensors located inside the scheduler provide it with time $c_k$ and the experienced scheduling error $ε_{k+1}$. The former may be used by the predictor in the estimation of a range $C_{k+1}$ expected to contain the next job computation time. This information, along with the measured $ε_{k+1}$ value, is used by the feedback controller to fulfill the task design goals.

4.3.1. Feedback controller

In the following, we assume that the feedback controller operates with perfect predictions ($c_k ∈ C_k$). It operates evaluating the worst-case effects (with respect to the design goals) caused by the uncertainty of $c_k$. As discussed above, the reachability of the equilibrium makes the scheme resilient to occasional errors.

Robust-controlled invariance: As $ε_k$ evolves in the lattice $E$ introduced in Section 4.1, the RCIS sets of interest are of the form $I = [−\hat{e}, E] = [−\hat{e}P, \hat{E}P]$, with $\hat{e} ∈ N \cap [0, L]$ and $\hat{E} ∈ N$ (we chose...
SUMMARY

adaptation mechanisms

fluid flow abstraction

periodic tasks

- ✔ serve jobs by demand
- ✗ complex infrastructure
- ✔ flexible on overruns
- ✗ job behavior less clear
- ✔ predictable job behavior
- ✗ inflexible
criticality

scheduling algorithm

priority

static priority dispatch

- criticality describes which tasks should be preferred under resource overload
- overload is allowed because of tiered assurance of timing parameters
real-time school systems school
real-time folks building systems systems folks designing time contracts

- each thread carries a virtual timestamp
- increases when thread runs, inversely proportional to its weight
- thread with smallest timestamp runs
- this is also the principle behind the Completely Fair Scheduler (CFS) in Linux
- **warp time** controls dispatch latency
- effective virtual time = actual virtual time - warp time
- effective time is used for scheduling
- warping describes short-term urgency

- using fair-share schedulers for real time
- careful policing of sharing weights

- integrated management of CPU, memory and disk to improve responsiveness
- external specification files
application
class
budget
period
flags
working-set keep-alive
I/O priority
(a) Impact of a 50 process fork bomb on mplayer

Frames per Second (fps)

Elapsed Time (Seconds)
adaptation mechanisms

fair sharing

time budget policing

SUMMARY

✔ serve jobs by demand
✘ complex enforcement

✔ timing guarantees
✘ only long-term control

✔ intuitive and simple
✘ no timing control
real-time work tends to start with theory and strong enforcement

- practical use often is only an afterthought

- systems work tends to start with best effort systems

- strong guarantees are just an afterthought

- golden middle ground still not found