Design and Implementation of an

Operating System

to Support

Distributed Multimedia Applications

Context

- we are in the year 1996
- multimedia was promised, but not delivered (think Windows 95)
- traditional operating systems are not well-suited for this, so redesign from scratch
- „Nemesis“

Assumptions

- General purpose platforms will **process** continuous media.
- Users will run many apps which manipulate media simultaneously.
- Such apps will make varying demands on resources during their execution.
- Application mix and load will be dynamic.

What‘s wrong?

- application QoS crosstalk
  - by sharing the same physical processor without decent mechanisms to control the resulting interference
  - contention when services multiplex low-level resources
- real-time threads do not help

Design Principle

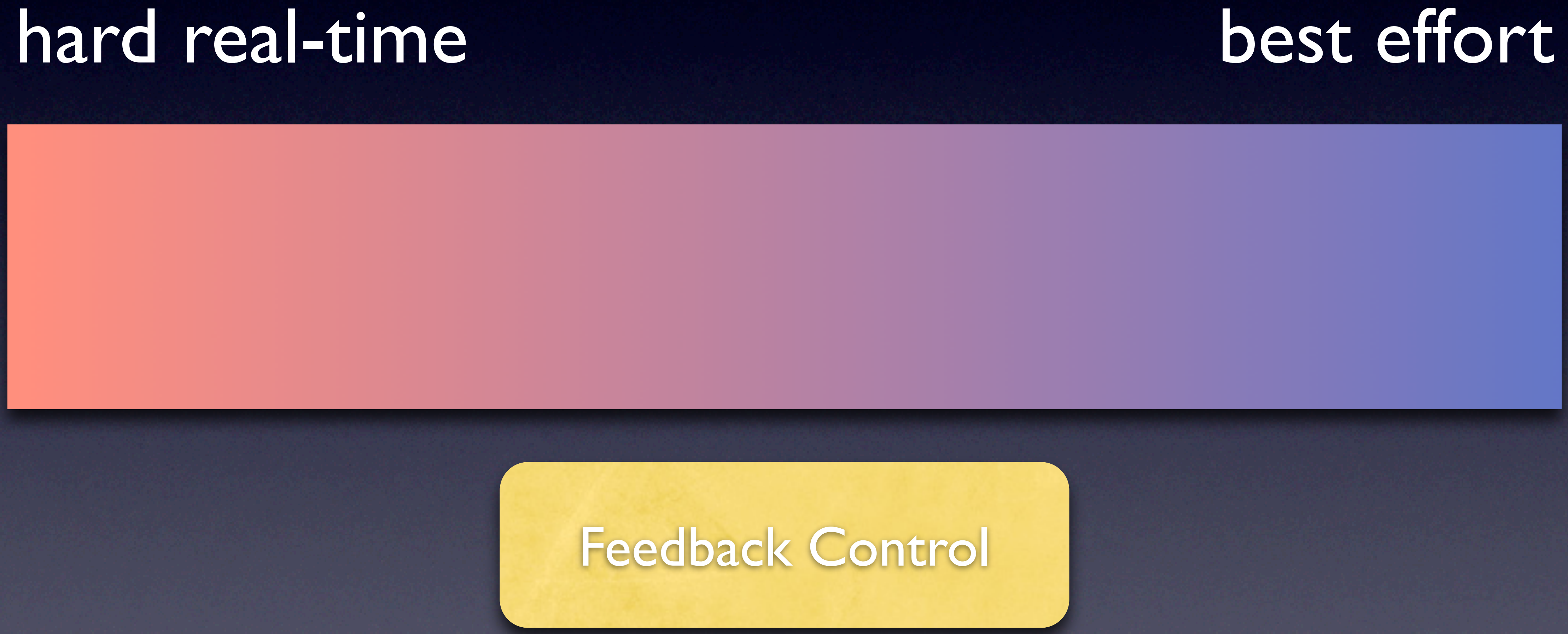
- execute as much functionality as possible in the application domain
- services are provided as shared libraries (Welcome to Exokernels?)
- use a **single address space** and protect with page table access control fields
- multiplexing system resources at the lowest level possible

Multimedia

two important properties of continuous media streams:

- temporal property
- informational property

QoS Model

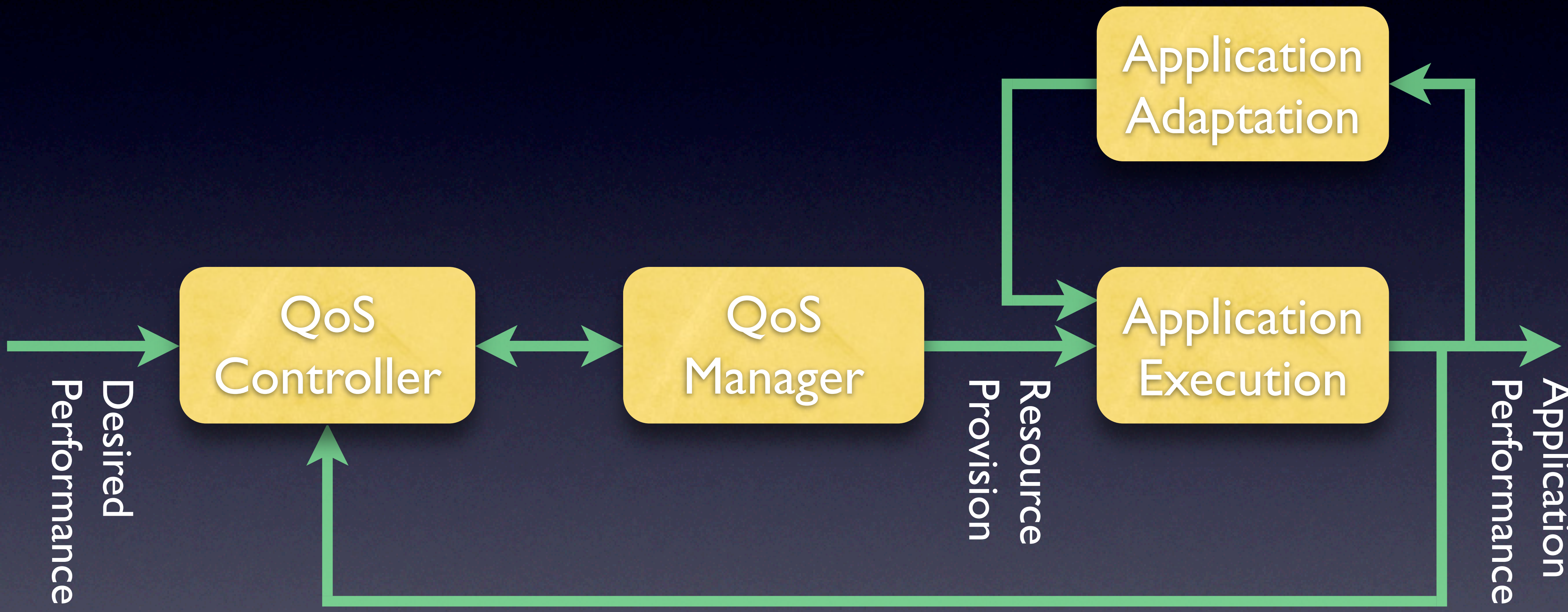


hard real-time

best effort

Feedback Control

QoS Feedback



Implications

- applications **do not need** to know their resource requirements in advance
- applications **need** to adapt
- servers introduce virtual resources that must be allocated consistently
- resource usage must be properly accounted

Resource Accounting

- migrating threads
  - threads cross protection domains
  - must be scheduled by the kernel
  - application no longer in control
- server has its own time
  - resource accounting must be communicated
  - crosstalk still occurs
  - nesting calls?

Vertical Integration

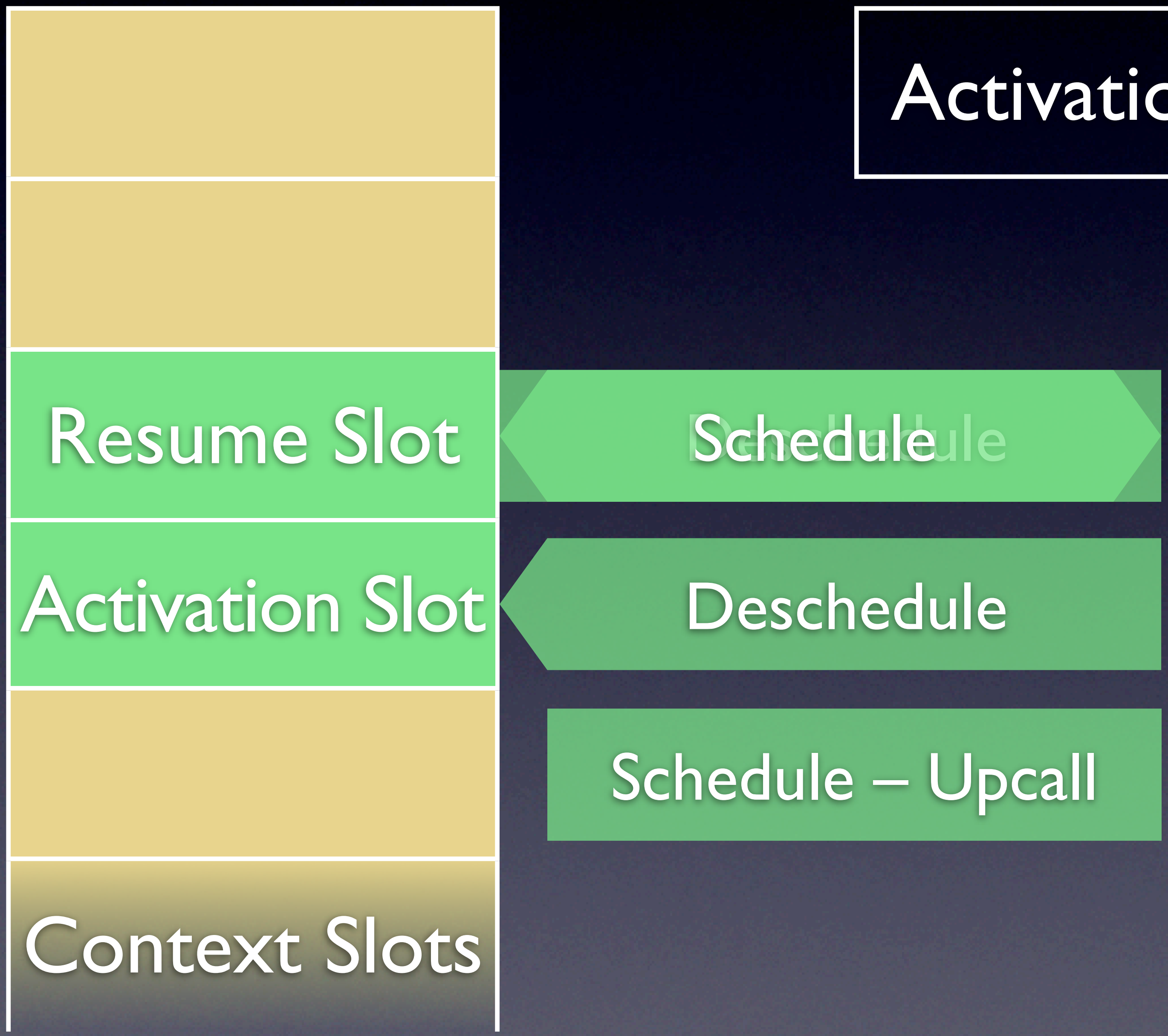
- minimal use of shared servers
- server performs only privileged operations
- everything else is done by the application
- controlled exposure of internal server state to clients
- single virtual address space eases sharing

Virtual Processor Interface

- tells the application when and why it is being scheduled
- supports user-level multiplexing of the CPU
- key concepts: activations, events
- provides time independent of scheduler clock

Activations

Activation Bit = 0



Resume Slot

Activation Slot

Context Slots

Schedule

Deschedule

Schedule – Upcall

Events

- monotonically increasing integer
- read and modified atomically by the sending domain
- recipient holds readonly copy updated by the kernel
- these channels are initiated by the Binder
- IDC and interrupt dispatch build on events

Kernel Structure

- no kernel threads, only interrupt and trap handlers
- interrupts are relayed to device driver domains
- this way, devices with a high interrupt rate do not interfere with the scheduling

Scheduling

- scheduling domains receive shares of the processor over short time frame
- scheduling domains are sets of domains
- scheduling domains can be
  - under a QoS contract
  - best-effort
- scheduling algorithm is open to choice

Synchronization

read(e)	returns the value of event e
await(e,v)	blocks the caller until e ≥ v
await_until(e,v,t)	await(e,v) with timeout
advance(e,n)	e += n
read(s)	returns the value of sequencer s
ticket(s)	returns and advances s

Higher Level Constructs

- interface reference vs. invocation reference
- the latter can be a pointer to the actual code or to a **surrogate**
- binding can be implicit or explicit
- interfaces use an IDL compiler (MIDDL)
- no global symbols

Opinion

- traditional vs. multimedia
- current multimedia apps **are** traditional
- single address space?
- is communication that expensive?
- real-time too troublesome?
- we have probabilistic scheduling with proven QoS properties