OVERVIEW
talked about in-kernel building blocks:

- threads
- memory
- IPC

drivers will enable access to a wide range of non-kernel resources

need to manage resources
Applications

System Services

Basic Abstractions
<table>
<thead>
<tr>
<th>Memory</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>discrete, limited</td>
<td>continuous, infinite</td>
</tr>
<tr>
<td>hidden in the system</td>
<td>user-perceivable</td>
</tr>
<tr>
<td>managed by pager</td>
<td>managed by scheduler</td>
</tr>
<tr>
<td>page-granular partitions</td>
<td>arbitrary granularity</td>
</tr>
<tr>
<td>all pages are of equal value</td>
<td>value depends on workload</td>
</tr>
<tr>
<td>active policy decisions, passive enforcement</td>
<td>active policy decisions, active enforcement</td>
</tr>
<tr>
<td>hierarchical management</td>
<td>Fiasco: flattened in-kernel view</td>
</tr>
</tbody>
</table>
REAL-TIME
a real-time system denotes a system, whose correctness depends on the timely delivery of results

“it matters, when a result is produced”

real-time denotes a predictable relation between system progress and wall-clock time
EXAMPLES

- engine control in a car
- break-by-wire
- avionics
- railway control

- set-top box media player
- mobile stack in your cell phone

focused catastrophic failures

benign failures complex
① Predictability
② Guarantees
③ Enforcement
- gap between worst and average case
- memory caches, disk caches, TLBs
- "smart" hardware
- system management mode
- disk request reordering
- cross-talk from resource sharing
- servers showing \( O(n) \) behavior
- SMP
- external influences: interrupts
custom RTOS

- small real-time executives tailor-made for specific applications
- fixed workload known a priori
- pre-calculated time-driven schedule
- used on small embedded controllers
- benign hardware
- full Linux kernel and real-time processes run side-by-side
- small real-time executive underneath supports scheduling and IPC
- real-time processes implemented as kernel modules
- all of this runs in kernel mode
- no isolation
- the kernel used in macOS and iOS
- offers a real-time priority band above the priority of kernel threads
- interface: “I need X time with a Y period.”
- threads exceeding their assignment will be demoted
- all drivers need to handle interrupts correctly
- static thread priorities
- \(O(1)\) complexity for most system calls
- fully preemptible in kernel mode
  - bounded interrupt latency
- lock-free synchronization
  - uses atomic operations
- wait-free synchronization
  - locking with helping instead of blocking
- “real-time” architecture for those afraid of touching the OS
- example: Real-Time Java
- a real-time kernel alone is not enough
- microkernel solution: temporal isolation
  - eliminates cross-talk through system calls
  - interrupt handling controlled by scheduler
- user-level servers as resource managers
  - implement real-time views on specific resources

- real-time is not only about CPU
GUARANTEES
worst case execution time (WCET) largely exceeds average case

offering guarantees for the worst case will waste lots of resources

missing some deadlines can be tolerated with the firm and soft real-time flavors
- desktop real-time
- there are no hard real-time applications on desktops
- there is a lot of firm and soft real-time
  - low-latency audio processing
  - smooth video playback
  - desktop effects
  - user interface responsiveness
guarantees even slightly below 100% of WCET can dramatically reduce resource allocation

unused reservations will be used by others at runtime

use probabilistic planning to model the actual execution

quality q: fraction of deadlines to be met
\[
    r'_i = \min\{r \in \mathbb{R} \mid \frac{1}{m_i} \sum_{k=1}^{m_i} P(X_i + k \cdot Y_i \leq r) \geq q_i\}
\]

\[
    r_i = \max(r'_i, w_i) \quad i = 1, \ldots, n
\]

- to fully understand this (or not): see real-time systems lecture
- good for microkernel: reservation can be calculated by a userland service
- kernel just needs to support static priorities
scheduling = admission + enforcement

admission = scheduling analysis
- verifies the feasibility of client requests
- formal task model
- calculates task parameters
- can reject requests

enforcement
- executing the schedule
- preempt when reservation expires
ENFORCEMENT
executed at specific events
enforces task parameters by preemption
e.g. on deadline overrun
picks the next thread
static priorities (e.g. RMS, DMS)
dynamic priorities (e.g. EDF)
seems simple…
- high priority thread calls low priority service, medium priority thread interferes:

![Diagram showing thread priorities and wait conditions]

- Thread 1 waits for Thread 3 ✔
- Thread 3 waits for Thread 2 ✔
- = Thread 1 waits for Thread 2 ✘

**Priority Inversion**
- priority inheritance, priority ceiling
- nice mechanism for this in Fiasco, NOVA: timeslice donation
- split thread control block
  - execution context: holds CPU state
  - scheduling context: time and priority
- on IPC-caused thread switch, only the execution context is switched
- IPC receiver runs on the sender’s scheduling context
- priority inversion problem solved with priority inheritance
- servers run on their clients’ time slice
  - when the server executes on behalf of a client, the client pays with its own time
- this allows for servers with no scheduling context
  - server has no time or priority on its own
  - can only execute on client’s time
  - relieves scheduler from dealing with servers
OPEN ISSUES

- servers could be malicious, so you need timeouts to get your time back
  - now, malicious clients can call the server with a very short timeout
  - on what time will the server do cleanup?
- donation does not work across CPUs
  - would thwart admission; one CPU cannot execute on behalf of another
  - migrate servers or clients?
OPTIMIZATION
- IPC only in the contention case
- optimized for low contention
- bad for producer-consumer problems
- reduce from 2 IPCs to one
- how to protect the short critical section?
- spinlocks suffer lockholder preemption
- allow threads to have short periods where they are never preempted
  - like a low cost global system lock
  - like a userland flavor of disabling interrupts
- **delayed preemption**
- threads set “don’t preempt” flag in UTCB
  - very low cost
  - not a lock, no lockholder preemption
- unbounded delay
  - kernel honors the delayed preemption flag only for a fixed maximum delay
- what delay is useful?
- delay affects all threads
  - effect can be limited to a priority band
  - must be included in real-time analysis
- does not work across multiple CPUs
managing time is necessary
we interact with the system based on time
real-time is a cross-cutting concern
heavy-math admission in userland, simple priorities in the kernel
priority inheritance by timeslice donation
synchronisation, delayed preemption
next week: drivers