



COMP9242
Advanced Operating Systems
S2/2011 Week 9:
Microkernel Design



NICTA Funding and Supporting Members and Partners



Copyright Notice



These slides are distributed under the Creative Commons Attribution 3.0 License

- You are free:
 - to share—to copy, distribute and transmit the work
 - to remix—to adapt the work
- under the following conditions:
 - Attribution:** You must attribute the work (but not in any way that suggests that the author endorses you or your use of the work) as follows:
 - “Courtesy of Gernot Heiser, [Institution]”, where [Institution] is one of “UNSW” or “NICTA”

The complete license text can be found at <http://creativecommons.org/licenses/by/3.0/legalcode>



Microkernel Principles: Minimality



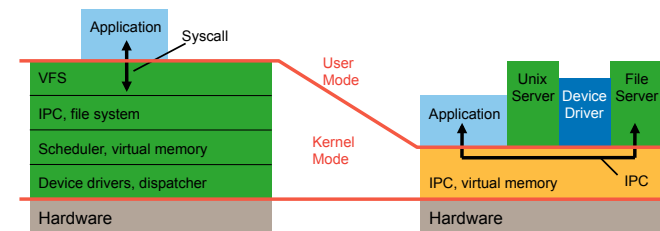
Said Liedtke [SOSP'95]:

A concept is tolerated inside the microkernel only if moving it outside the kernel, i.e. permitting competing implementations, would prevent the implementation of the system's required functionality.

- Strict adherence leads to a very small kernel
- Advantages of small kernel
 - Easy to implement, port?
 - in practice limited architecture-specific micro-optimization
 - Easier to optimise
 - Hopefully enables a minimal *trusted computing base* (TCB)
 - small attack surface, fewer failure modes
 - Easier debug, maybe even *prove* correct?
- Challenges:
 - API design: generality with small code base
 - Kernel design and implementation for high performance
 - ... and correctness!



Consequence of Minimality: User-level Services



- Kernel provides no services, only mechanisms
- Strongly dependent on fast IPC and exception handling



Microkernel Principles: Policy Freedom



- Consequence of generality and minimality requirements:
 - *A true microkernel must be free of policy*
 - Policies limit
 - May be good for many cases, but always bad for some
 - Example: disk pre-fetching
 - Attempts to make policies general lead to bloat
 - Implementing combination of policies
 - Try to determine most appropriate one at run-time



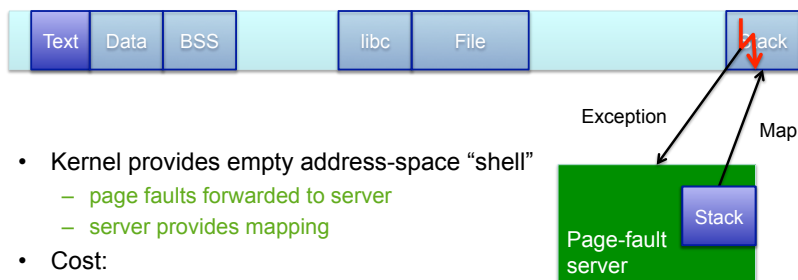
Policy Example: Address-Space Layout



- Kernel determines layout, knows executable format, allocates stack
 - limits ability to import from other OSES
 - cannot change layout
 - small non-overlapping address spaces beneficial on some archs
 - kernel loads apps, sets up mappings, allocates stack
 - requires file system in kernel or interfaced to kernel
 - bookkeeping for revocation & resource management
 - heavyweight processes
 - memory-mapped file API



Policy-Free Address-Space Management



- Kernel provides empty address-space “shell”
 - page faults forwarded to server
 - server provides mapping
- Cost:
 - 1 round-trip IPC, plus mapping operation
 - mapping may be side effect of IPC
 - kernel may expose data structure
 - kernel mechanism for forwarding page-fault exception
- “External pagers” first appeared in Mach [Rashid et al, '88]
 - ... but were optional



What Mechanisms?



- Fundamentally, the microkernel must abstract
 - Physical memory
 - CPU
 - Interrupts/Exceptions
- Unfettered access to any of these bypasses security
 - No further abstraction needed for devices
 - memory-mapping device registers and interrupt abstraction suffices
 - ...but some generalised memory abstraction needed for I/O space
- Above isolates execution units, hence microkernel must also provide
 - Communication (traditionally referred to as IPC)
 - Synchronization



What Mechanisms?



Traditional hypervisor vs microkernel abstractions

Resource	Hypervisor	Microkernel
Memory	Virtual MMU (vMMU)	Address space
CPU	Virtual CPU (vCPU)	Thread or scheduler activation
Interrupt	Virtual IRQ (vIRQ)	IPC message or signal
Communication	Virtual NIC	Message-passing IPC
Synchronization	Virtual IRQ	IPC message

Abstracting Memory: Address Spaces

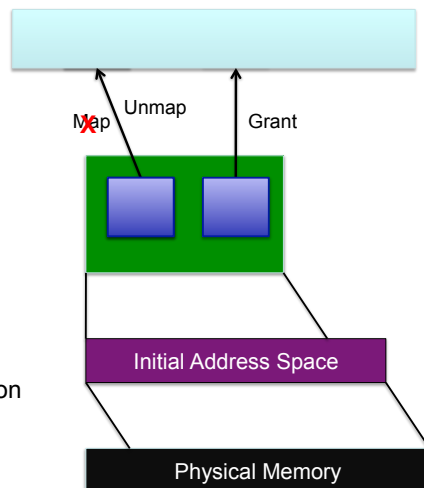


- Minimum address-space abstraction: empty slots for page mappings
 - paging server can fill with mappings
 - virtual address → physical address + permissions
- Can be
 - page-table-like: array under full user control
 - TLB-like: cache for mappings which may vanish
- Main design decision: is source of a mapping a page or a frame?
 - Frame: hardware-like
 - Page: recursive address spaces (original L4 model)

Traditional L4: Recursive Address Spaces



- Mappings are page → page
- Magic initial address space to anchor recursion



Abstracting Interrupts and Exceptions



- Can abstract as:
 - Upcall to interrupt/exception handler
 - hardware-like diversion of execution
 - need to save enough state to continue interrupted execution
 - IPC message to handler from magic “hardware thread”
 - OS-like
 - needs separate handler thread ready to receive

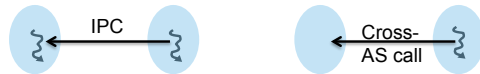


- Page fault tends to be special-cased
 - Tends to require handling external to faulter
 - IPC message to page-fault server
 - But also “self-paging” as in Nemesis [Hand '99] or Barrelfish

Abstracting Execution



- Can abstract as:
 - kernel-scheduled threads
 - Forces (scheduling) policy into the kernel
 - vCPUs or scheduler activations
 - This essentially virtualizes the timer interrupt through upcall
 - Scheduler activations also upcall for exceptions etc
 - Multiple vCPU only for real multiprocessing
- Threads can be tied to address space or “migrating”



- Tight integration/interdependence with IPC model!

Communication Abstraction (IPC)



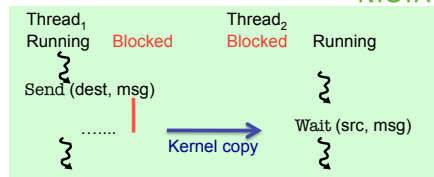
Sender: send (dest, msg)
Receiver: receive (src, msg)

- Seems simple, but requires several major design decisions
 - Does the sender block if the receiver isn't ready?
 - Does the receiver block if there is no message?
 - Is the message format/size fixed or variable?
 - Do “dest”, “src” refer to active (thread) or passive (mailbox) entities?

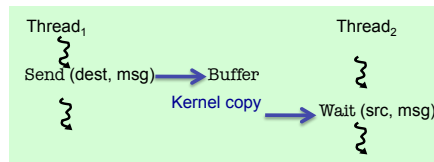
Blocking vs Non-Blocking IPC



- Blocking send:
 - Forces synchronization (rendez vous) with receiver
 - Doubles as synchronization primitive
 - Requires kernel threads
 - ... else block whole app



- Non-blocking send:
 - Requires buffering
 - Data copied twice
 - Can buffer at receiver, but then can only have single message in transit
- Non-blocking receive requires polling or asynchronous upcall
 - Polling is inefficient, upcall forces concurrency on apps
 - Usually have at least an option to block



Message Size and Location



Fixed- vs variable-size messages:

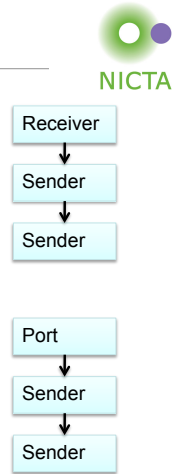
- Fixed simplifies buffering and book-keeping
- Variable requires receiver to provide big enough buffer
 - Only an issue if messages are very long

Dedicated message buffer vs arbitrary pointer to data:

- (Small) dedicated message buffer may be pinned (virtual registers)
- Arbitrary data strings may cause page faults
 - abort IPC?
 - handle fault by invoking pager?

Direct vs Indirect IPC Addressing

- Direct: Queue senders/messages at receiver
 - Need unique thread IDs
 - Kernel guarantees identity of sender
 - useful for authentication
 - Can't have multiple receivers wait for message
 - eg pools of worker threads
- Indirect: Mailbox/port object
 - Just a user-level handle for the kernel-level queue
 - Extra object type – extra weight?
 - Communication partners are anonymous
 - Need separate mechanism for authentication



Typical Approaches

- Asynchronous send plus synchronous receive
 - most convenient for programmers
 - minimises explicit concurrency control at user level
 - generally possible to get away with single-threaded processes
 - main drawback is need for kernel to buffer
 - violates minimality, adds complexity
 - typical for 1st generation microkernels
- Traditional L4 model is totally synchronous
 - Allows very tight implementation
 - Not suitable for manycores
 - Requires (kernel-scheduled) multi-threaded apps!
 - Kernel policy on intra-process scheduling!
- OKL4 microvisor IPC is totally asynchronous
 - ... but forces one partner to supply buffer
 - synchronization via virtual IRQs

Brief History of Microkernels

1st Generation: mid-1980 (Mach, Chorus etc)

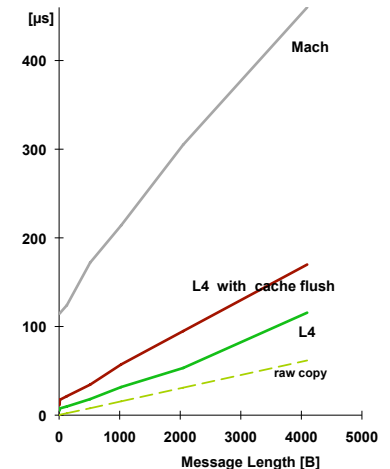
- Stripped-down monolithic OSes
- Lots of functionality and policy
 - device drivers, low-level file systems, swapping
 - very general, rich and complex IPC
- Big
 - Mach had about 300 kernel APIs, 100s kLOC C
- Slow: 100 μ s IPC
 - cache footprint shown a major factor in poor performance [Liedtke 95]
 - consequence of IPC complexity, poor design and implementation
 - stripping out stuff from a big blob doesn't produce a good microblob!



Brief History of Microkernels

2nd Generation: mid-1990s – L4

- “Radical” approach [Liedtke '93, Liedtke '95]:
 - Strict minimality
 - From-scratch design and implementation
- Fast!



Brief History of Microkernels



2nd Generation: mid-1990s – L4

- “Radical” approach [Liedtke'93, Liedtke '95]:
 - strict minimality, designed and implemented from scratch
- Fast
- Minimal and orthogonal mechanisms
 - L4 V2 API: 7 system calls plus a few kernel protocols
 - reduced IPC complexity
 - 15 kLOC(?) x86 assembler
- Multiple implementations
 - L4 MIPS, L4 Alpha (UNSW 95–97), 10 kLOC (half C, half assembler)
 - Fiasco (Dresden 97–now), x86 (later ARM), 30 kLOC (C++)
- Portable kernel: Pistachio (Karlsruhe/UNSW, 2002–06)
 - V4: x86, PPC, ARM, MIPS, Alpha, Itanium
 - L4-embedded (NICTA), morphed into commercially-deployed OKL4
 - first capability-based L4 kernel

Brief History of Microkernels



3rd Generation: seL4 [Elphinstone et al 2007, Klein et al 2009]

- Security-oriented design
 - capability-based access control
 - strong isolation
- Hardware resources subject to user-defined policies
 - including kernel memory (no kernel heap)
 - except time ☺
- Designed for *formal verification*

Lessons Learned from 2nd Generation



Micro-optimisation: core feature of L4

- Programming languages:
 - original i496 kerne ['95]: all assembler
 - UNSW MIPS and Alpha kernels ['96,'98]: half assembler, half C
 - Fiasco [TUD '98], Pistachio ['02]: C++ with assembler “fast path”
 - seL4 ['07], OKL4 ['09]: all C
- Lessons:
 - C++ sux: code bloat, no real benefit
 - Changing calling conventions not worthwhile
 - Conversion cost in library stubs and when entering C in kernel
 - Reduced compiler optimization
 - Assembler unnecessary for performance
 - Can write C which leads to near-optimal code
 - C entry from assembler cheap if calling conventions maintained
 - seL4 performance with C-only pastpath as good as other L4 kernels

Lessons Learned from 2nd Generation



Micro-optimisation: core feature of L4

- Liedtke: process-oriented kernel for simplicity and efficiency
 - Per-thread kernel stack, co-located with TCB
 - reduced TLB footprint (i486 had no large pages!)
 - easier to deal with blocking in kernel
 - Cost: high memory overhead
 - about 1/4–1/2 of kernel memory
 - Effectively needed continuations anyway for nested faults
 - page-fault during “long” IPC
 - No performance benefit on modern hardware [Warton, BE UNSW'05]
- Liedtke: virtual TCB array for fast lookup from thread ID
 - Cost: large VM consumption, increased TLB pressure
 - No performance benefit on modern hardware [Nourai, BE UNSW'05]

Lessons Learned from 2nd Generation



API complexity still too high

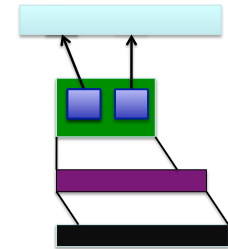
- IPC semantics:
 - In-register, in-line and by-reference message
 - Timeouts on each IPC
 - Mappings created as a side-effect of IPC
- Timeouts: need way to avoid DOS-attacks by blocking partner
 - Timeouts too general: no systematic approach to determine them
 - Significant source of kernel complexity
 - In practice only use zero and infinity
 - Replaced (in NICTA version) by fail-if-not-ready flag
- Various “long” message forms: complex and rarely used
 - Require handling of in-kernel page faults (during copying)
 - massive source of kernel complexity
 - Replaced (by Pistachio) by pinned message buffers (“virtual registers”)
 - essentially retained by seL4

Lessons Learned from 2nd Generation



API complexity: Recursive address-space model

- Conceptually elegant
 - trivially supports virtualization
- Drawback: Complex mapping database
 - Kernel needs to track mapping relationship
 - Tear down dependent mappings on unmap
 - Mapping database problems:
 - accounts for 1/4–1/2 of kernel memory use
 - SMP coherence is performance bottleneck
- NICTA’s L4-embedded, OKL4 removed MDB
 - Map frames rather than pages
 - need separate abstraction for frames / physical memory
 - subsystems no longer virtualizable (even in OKL4 cap model)
- Properly addressed by seL4’s capability-based model



Lessons Learned from 2nd Generation



Blocking IPC is not sufficient in practice

- Does not map well to hardware-generated events (interrupts)
 - Many real-world systems are event-driven (especially RT)
 - Mapping to synchronous IPC model requires proliferation of threads
 - Forces explicit concurrency control on user code
 - Made worse by IPC being too expensive for synchronization
- Attempt by Liedtke to address with “user-level” IPC [Liedtke '01]
 - intra-address-space only
 - thread manipulates partner’s TCB
 - part of thread state kept in user-level TCB (UCTB)
 - caller executes kernel IPC code in user mode
 - inconsistencies fixed up on next kernel entry
 - too messy & limiting in practice
- Introduction of asynchronous notify (L4-embedded, NICTA'04)
 - much closer to hardware interrupts
 - OKL4 Microvisor completely discarded synchronous IPC

Lessons Learned from 2nd Generation



Access control, naming and resource management

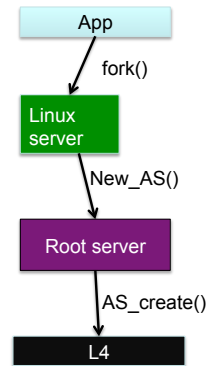
- L4 used global thread IDs to address IPC
 - fast as it avoids indirection via ports or mailboxes
 - inflexible, as server threads need to be externalised (thread pools!)
 - various hacks around this were tried, none convinced
 - expensive to virtualize (intermediate must relay each message)
 - cannot enforce access monitor
 - “clans and chiefs” hack doubles message, too expensive in practice
 - global names are a covert channel [Shapiro '03]
- Need anonymising intermediate message target (endpoints)

Lessons Learned from 2nd Generation



Access control, naming and resource management

- L4 had no proper model for *rights delegation*
 - Partially due to ad-hoc resource protection approach
- Subsystem could DOS kernel
 - Create mappings until kernel out of memory
 - In V4 addressed by restricting resource management to root server
 - Requires subsystem asking root server to perform operations
 - expensive!
 - Result: performance vs security tradeoff!
- Properly addressed by seL4's caps and resource-management



Lessons Learned from 2nd Generation



Suitability for real-time systems

- Basic idea was there: hard-prio round-robin scheduling, but...
 - Temporary priority inversion from implementation tricks
 - Lazy scheduling: IPC leaves blocked threads in ready queue
 - scheduler cleans up next time it runs
 - often thread runs again before scheduler invocation
 - » scheduler run can be expensive
 - Direct context switch: reply_and_wait without scheduler
 - after sending reply, check receive queue for next sender
 - FIFO-ordered IPC receive queues
 - Fixed by OKL4 (and Fiasco?)
 - Lazy dispatch: leave thread in wait queue until preempted
 - Make direct context switch observe prios
- Still issue of fixed-prio policy not suitable for all RT uses

Lessons Learned from 2nd Generation



Suitability for real-time systems

- Kernel runs with interrupts disabled
 - Better for average-case performance
 - Much simpler kernel by avoiding concurrency
- Long-running system calls cause problems for RT
 - Addressed by V2 kernels via preemption points
 - Briefly turn on interrupts at safe locations
 - Gets (limited) concurrency back into kernel
 - Addressed by Fiasco by making kernel fully preemptible
 - Massive increase in kernel complexity, very hard to verify!
 - Better solution (OKL4, seL4):
 - Keep system calls short (eg no “long” IPC)
 - Explicit preemption points where long ops are inevitable
 - dismantling resource derivation trees
 - cost of ability to delegate rights!

seL4 Design Principles



- Fully delegatable access control
- All resource management is subject to user-defined policies
 - Applies to kernel resources too!
- Suitable for *formal verification*
 - Requires small size, avoid complex constructs
- Performance on par with best-performing L4 kernels
 - Prerequisite for real-world deployment!
- Suitability for real-time use
 - Only partially achieved to date ☹
 - on-going work...

(Informal) Requirements for Formal Verification

- Verification scales poorly \Rightarrow small size (LOC and API)
- Conceptual complexity hurts \Rightarrow KISS
- Global invariants are expensive \Rightarrow KISS
- Concurrency difficult to reason about \Rightarrow single-threaded kernel

Largely in line with traditional L4 approach!



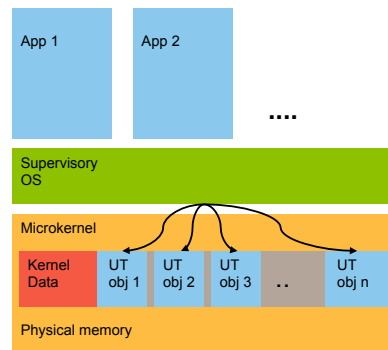
Fundamental Abstractions

- Capabilities as opaque names and access tokens
 - All kernel operations are cap invocations (except `Yield()`)
- IPC:
 - Synchronous (blocking) message passing plus asynchronous notification
 - Endpoint objects implemented as message queues
 - Send: get receiver TCB from endpoint or enqueue self
 - Receive: obtain sender's TCB from endpoint or enqueue self
- Other APIs:
 - `Send()/Receive()` to/from virtual kernel endpoint
 - Can interpose operations by substituting actual endpoint
- Fully user-controlled memory management
 - The *real* conceptual novelty of seL4



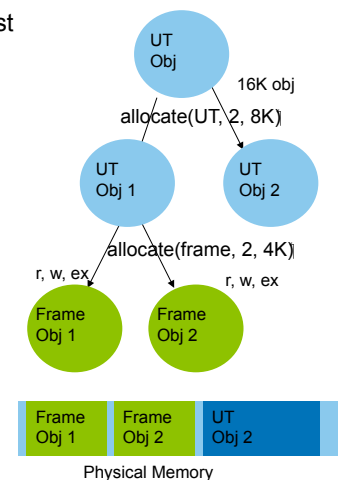
seL4 Physical Memory Management

- Some kernel memory is *statically* allocated at boot time
- Remainder is divided into *untyped* (UT) objects
 - 2^n region of physical memory
 - size-aligned
- Initial task gets caps for all untyped memory
- Kernel never dynamically allocates memory after boot
 - apps must provide memory
 - by re-typing untyped



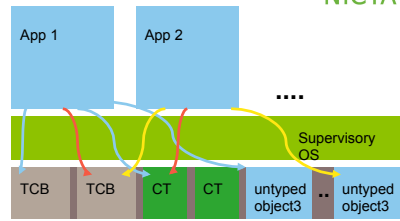
Kernel Object Allocation...

- No implicit allocation in kernel!
- Object allocation upon explicit request
 - ... by entity holding caps to untyped
- Untyped can be retyped into:
 - frames (for application use)
 - TCBS
 - Cspaces (cap tabled)
 - Address spaces (page tables)
 - Endpoints
 - I/O page tables (x86)
- Consequence:
 - kernel data as strongly partitioned as user data



Object Management

- Delegate authority
 - Delegate part or all of authority over objects
 - Allow others to obtain services directly
 - Requires “grant” right
- Delegation of untyped objects
 - Subsystems can manage resources
 - according to their own policy
 - within limits of what was delegated
 - Hierarchical resource management
 - Objects manipulated by authorised users
 - no need for porixying



Example:
add/remove capability mappings to/from Cspace



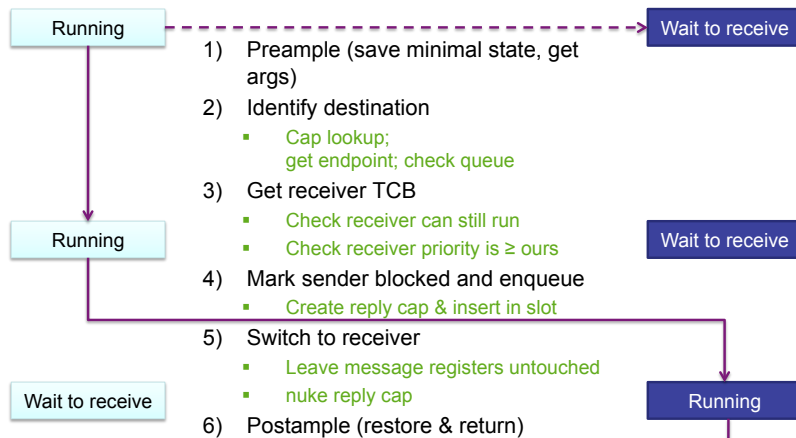
Event Kernel and Preemption Points

- Event kernel saves massive memory due to shared kernel stack
 - Also more cache-friendly
 - Actual trade-offs complex, evaluated for Fluke [Ford et al '99]
- Works well with preemption points
 - Do not enable interrupts, just check for pending
 - if so, back out to kernel boundary, leaving syscall args in tact
 - let interrupt happen
 - when handled, syscall is automatically restarted
 - Similar to EROS [Shapiro et al '99]
 - Has side-effect of re-establishing all invariants
 - simplifies verification



Synchronous IPC Implementation

Simple send (e.g. as part of RPC-like “call”):



Approx 200 cycles on ARM11!



Fastpath Coding Tricks

```
slow = cap_get_capType(en_o) != cap_endpoint_cap ||
       !cap_endpoint_cap_get_capCanSend(en_o);
if (slow) enter_slow_path();
```

Common case: 0

Common case: 1

- Reduces branch-prediction footprint
- Avoids mispredicts, stalls & flushes
- Uses ARM instruction predication
- But: increases slow-path latency
 - should be minimal compared to basic slow path cost



Other implementation tricks



- Cache-friendly data structure layout, especially TCBS
 - data likely used together is on same cache line
 - helps best-case and worst-case performance
- Kernel mappings locked in TLB (using superpages)
 - helps worst-case performance
 - helps establish invariants: page table never walked when in kernel
- Lazy FPU switch
 - FPU context tends to be heavyweight
 - Only few apps use FPU (and those don't do many syscalls)
 - On context switch, simply disable FPU
 - On access by app, exception is triggered
 - now switch FPU context and enable