TECHNISCHE UNIVERSITÄT DRESDEN

Platform-Specific Optimizations

Increase Performance

Last Lecture

Different Platforms and CPUs

- Get functional correct behavior
  - Cache consistency
  - TLB consistency
  - ...
- With good reuse of generic code

Identified some potential performance-relevant properties

- CPU state (up to 2kB on IA-64)
- TLB
- Caches

Contents

- Generic optimizations
- Platform specific bottlenecks
  - Overview of solutions
- Optimizations in a microkernel
- Practical examples
  - Address-space switching (TLB/Caches)
  - IPC performance
  - Context-switch performance (FPU)
- Conclusion / Evaluation

Generic Optimizations

Optimized data structures and code

- Minimize memory accesses
- Minimize cache and TLB footprint
- Minimize number of instructions for frequently used operations

Optimizations often depend on detailed knowledge of HW

- Cache size / associativity
- TLB size / features (e.g., supported page sizes)
- Available instructions in the ISA

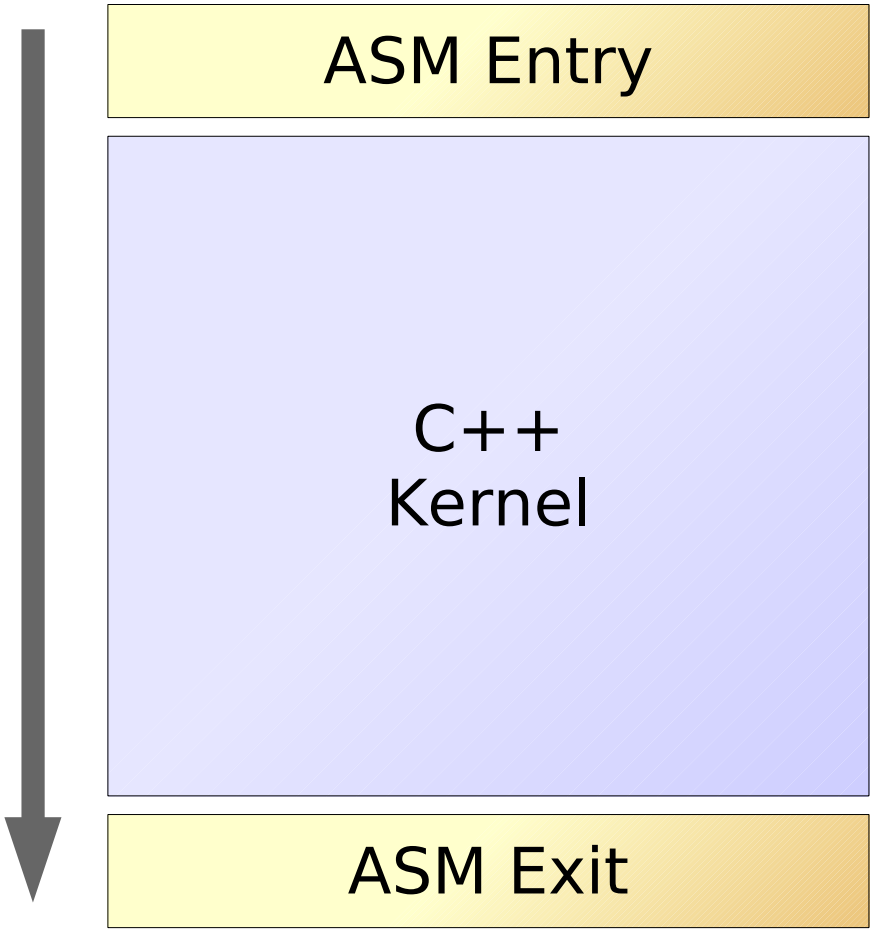
Performance Bottlenecks

- Costs for address-space switches
  - TLB flushes/refills, Cache flushes/refills
- IPC costs
- Long-IPC copy
- CPU/FPU context save and restore
- Kernel entry/exit

What Can We do?

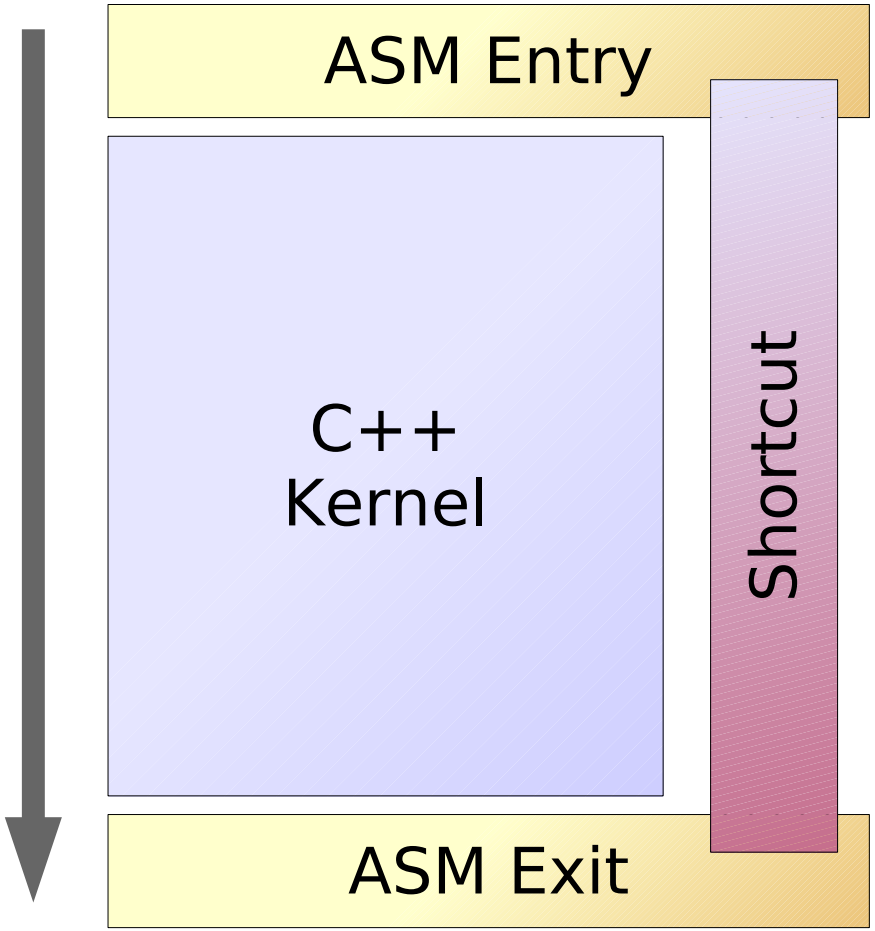
- TLB and cache
  - Use hardware features to minimize costs
- IPC costs
  - CPU specific asm code with minimal overhead
- Long-IPC copy
  - Specially optimized copy functions
- CPU/FPU context switches
  - Use lazy techniques for context save and restore
- Kernel entry/exit
  - Specific instructions for minimized costs

Anchors for Optimizations I



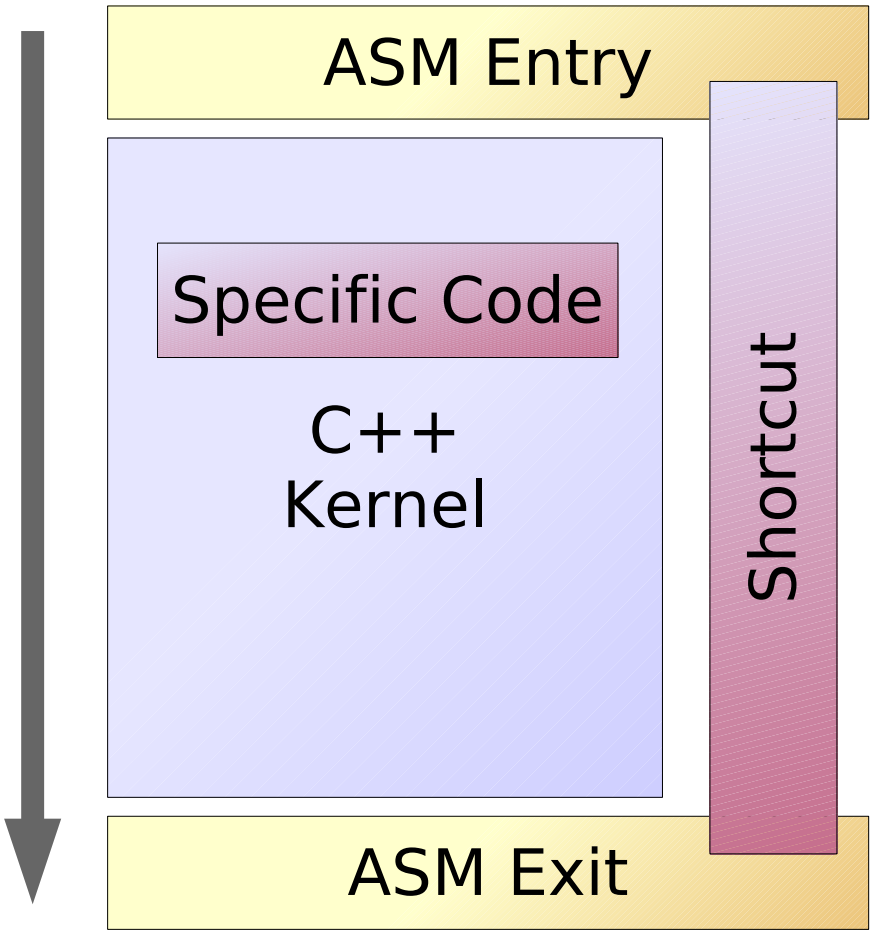
- Optimized ASM entry and exit code
  - Different x86 CPU may have different ASM (`sysenter, sysexit, syscall, sysret ...`)

Anchors for Optimizations II



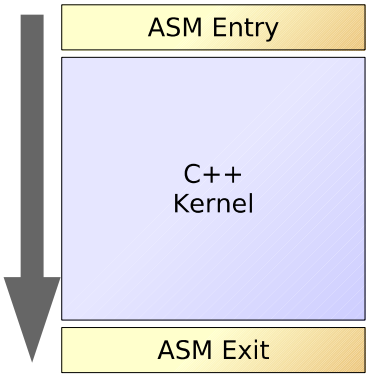
- Optimized ASM entry and exit code
- Direct shortcuts

Anchors for Optimizations III



- Optimized ASM entry and exit code
- Direct shortcuts
- Replace functions inside the kernel
  - Specific page-table code
  - Specific copy routines
  - ...

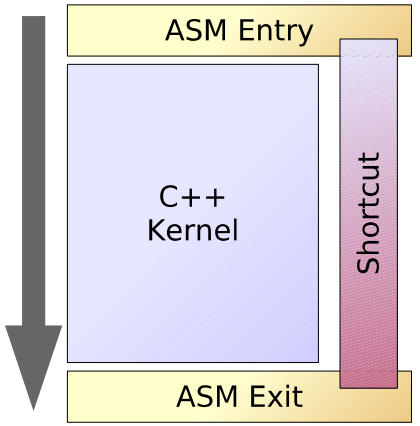
Properties of Approach I



Optimized ASM entry/exit code

- CPU specific code in any case
- Install the fastest version of at kernel boot time (using special instructions)
- Often significant performance gain

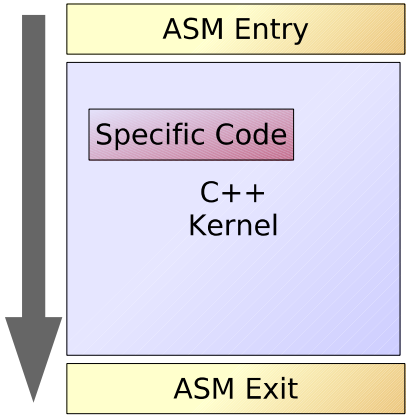
Properties of Approach II



Shortcuts

- Duplicated logic for generic functionality (duplicated code)
- Expensive to maintain (often ASM code)
- Often significant improvement of common cases (e.g., IPC call)

Properties of Approach III



CPU Optimized Code in the kernel

- Most flexible usage of special hardware features
- Interfaces between the optimized and the generic code are hard to define

Practical Examples

Address-Space Switch
IPC Performance
Context / Thread-Switch (FPU)

Address-Space Costs (most important on µKernels)

- Address-space switches are frequent
- Two kinds of costs
  - Direct: Page-table reload, TLB flush, Cache flush
  - Indirect: TLB refill, Cache refill

- Caches are problematic if virtually tagged/indexed
  - Cache contents are associated with virtual addresses
  - Flush costs in the order of milliseconds on ARM926

Address-Space IDs

Solutions:

- Address-space IDs (ASID)
  - Add ASID to TLB/Cache tag (hardware feature)

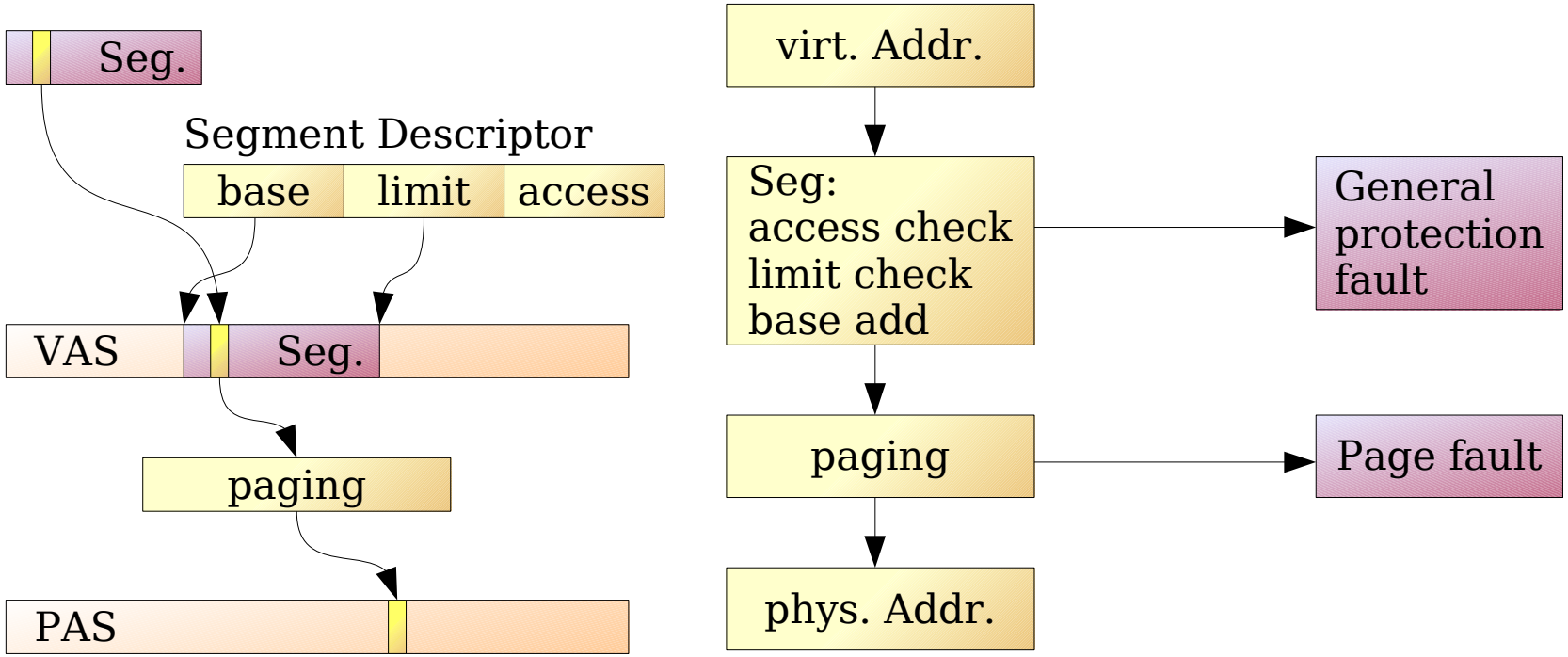
ASID	Virt. Page No.	Phys. Frame No.
A	0x4000	0x100
B	0x4000	0x300
B	0x4200	0x100
A	0x4010	0x110
	...	
A	0x4300	0x250

CPUs & ASIDs

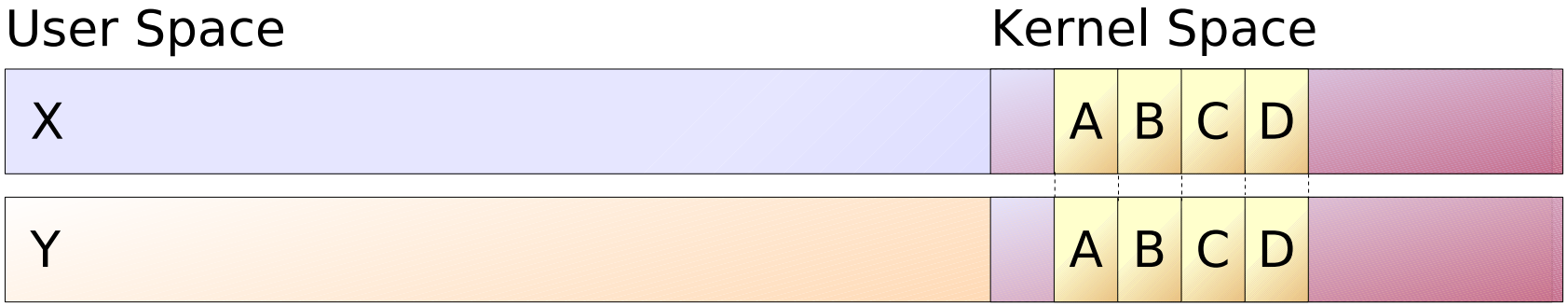
- IA-64: phys. tagged caches
  - supports explicit ASID in TLB: ASID management in s/w
- AMD-64: phys. tagged caches
  - Opteron supports implicit TLB tags (flush filter): transparent for OS
- IA-32: phys. tagged caches
  - No ASIDs in TLB, but Segmentation
- ARMv5: virt. tagged caches
  - No ASID in TLB/Cache, but Protection Domains and Fast Context-Switch Extension (FCSE)
- ARMv6: phys. tagged/index caches, ASIDs (8bit)

IA-32 Segmentation

- Memory protection & relocation independent from paging



IA-32 Small Address Spaces



A, B, C, and D: user apps in small address space

- Switches among A, B, C, D, and current space
  - Modify descriptor table (setup right segments)
  - Avoid TLB flush (no page-table reload)

IA-32 SMAS in Fiasco

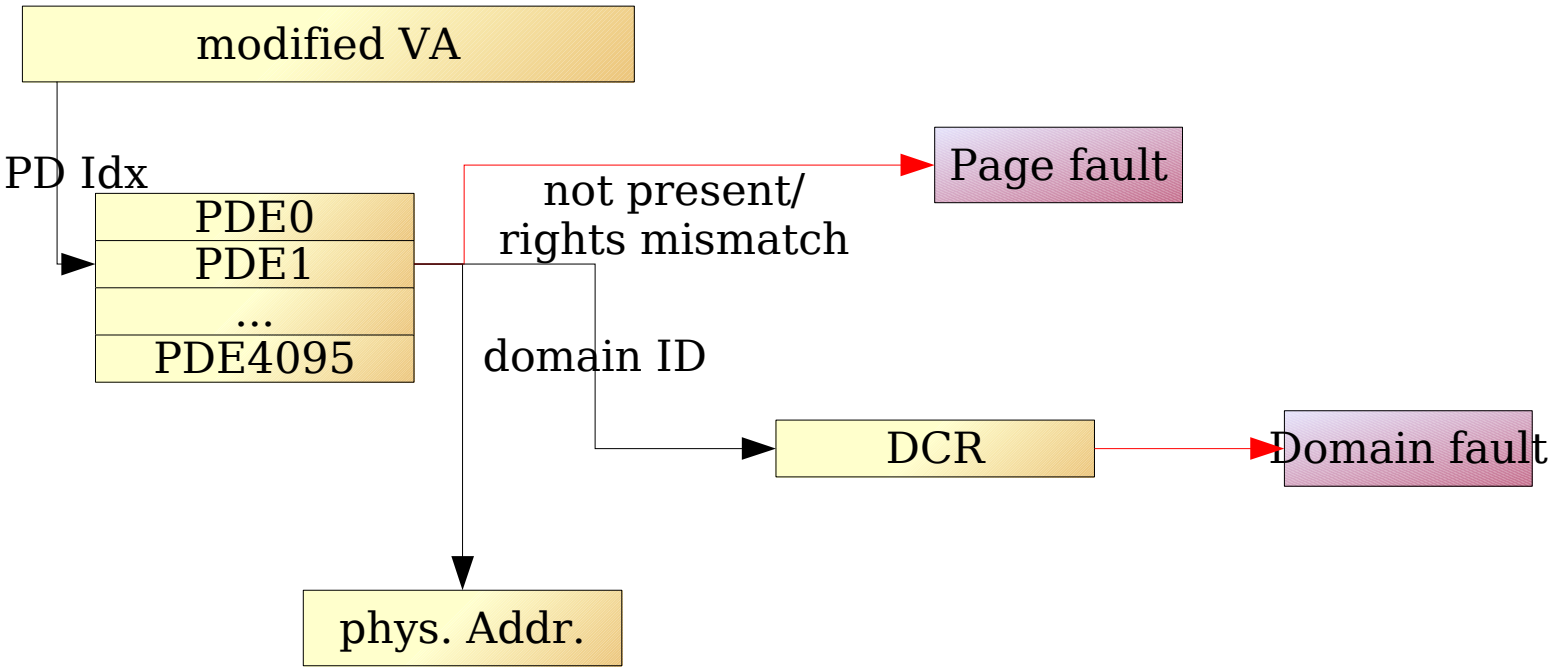
- IA-32 Segmentation for multiplexing address spaces (ASID emulation)
  - Reduce TLB flushes/refills

- Implemented with special in-kernel functions
  - Small space has normal page table: lazy copy technique to copy entries to current page table
  - Special copy to/from user functions for long IPC
  - Modify descriptor table on address-space switches
- Special code in ASM entry/exit code
- Special code in ASM IPC Shortcut

ARM Protection Domains

- 1st level page table entry:
  - Describes 1 Mbyte virtual memory
  - 4 Bit Protection Domain ID (16 values)
- Domain Control Register (DCR)
  - 16 x 2 Bit for current domain access rights (no access, client, manager)
- 4 Bit Protection Domain ID is part of TLB and Cache TAG

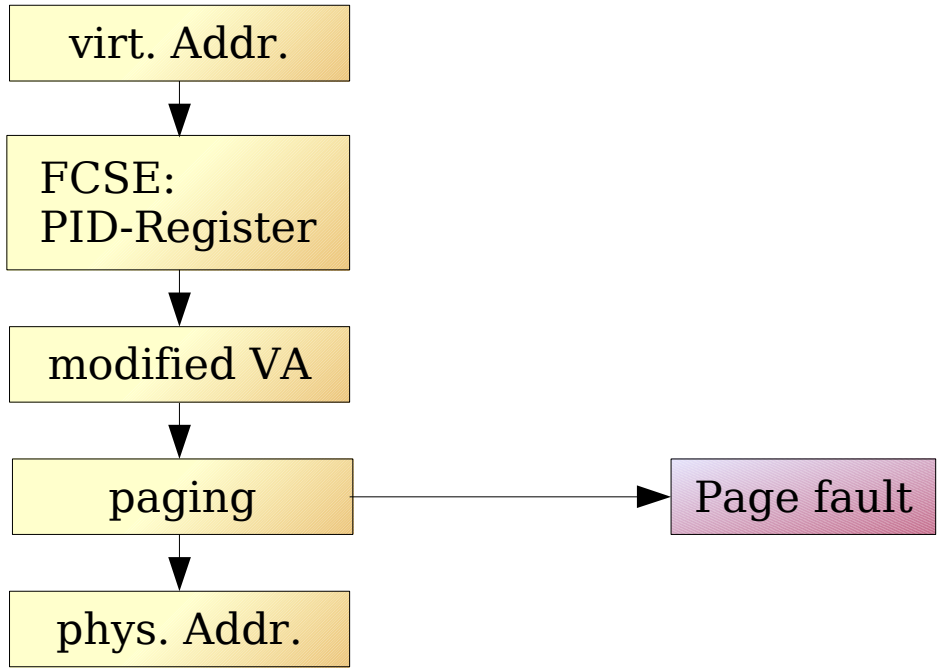
ARM Protection Domains (2)



ARM FCSE

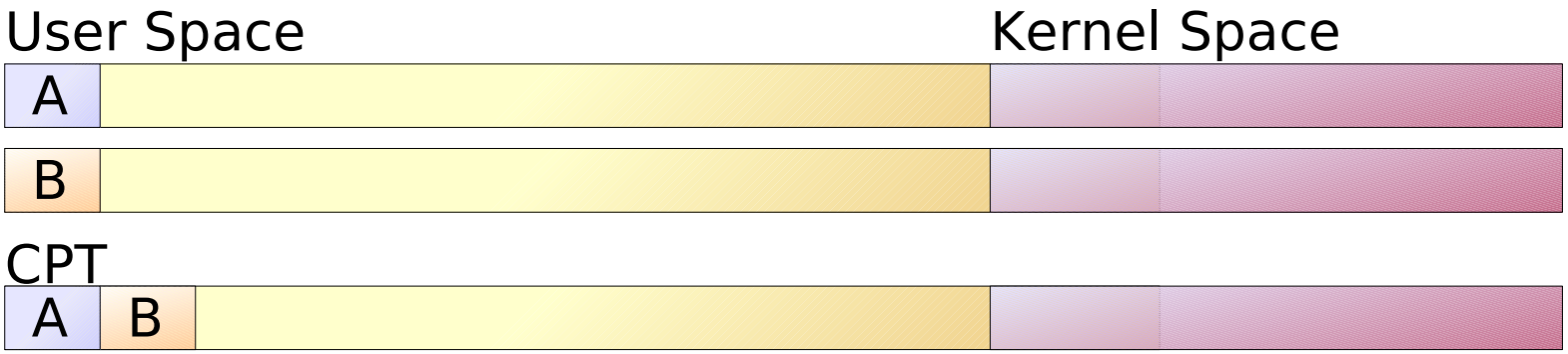
- FCSE (aka WinCE extension)
- PID Register based memory relocation
- No Protection

$$MVA = \begin{cases} VA + 32\text{MB} \cdot PID & \text{if } VA < 32\text{MB} \\ VA & \text{if } VA \geq 32\text{MB} \end{cases}$$



ARM SMAS 1

- Need caching page table (CPT)
- Isolation → Protection Domains
- Relocation → FCSE (aka WinCE extension)



- Never switch PT, only CPT used for HW-translation

ARM SMAS 2

- Domain ID emulates ASID

Address-Space Switch:

- Reload DCR (access permissions)
- Reload PID Register (relocation)

Caveats:

- Domain replacement if more than 16 address spaces
- Effective address space limited to 32 MB (FCSE)
- Not yet implemented in Fiasco

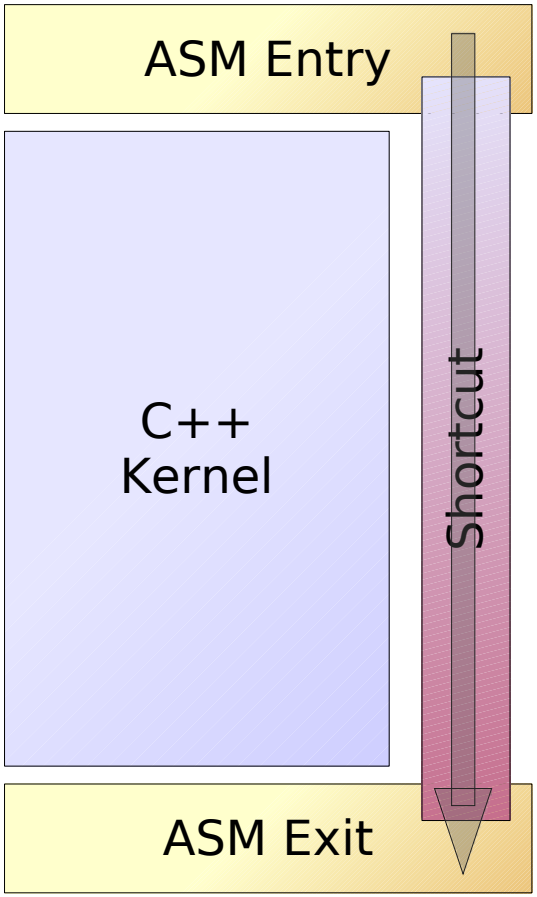
IPC Performance

- Fiasco IPC path is completely preemptible
  - Complex state machine for synchronization
- Must support page faults on TCBs
- IPC timeouts (other than 0 or infinite)
- Real-time extensions
- C++ Compiler allocates Registers

➔ ASM IPC Shortcut

ASM IPC Shortcut 1

- Small and fast shortcuts in assembly language
- IA-32 IPC shortcut:
  - Uninterruptible (low upper bound for runtime allows diabled IRQs): simple logic, no IPC state machine
  - Optimized register allocation: parts of the message may stay in CPU registers
  - Optimized for branch prediction of CPU



ASM IPC Shortcut 2

- Small ASM code may reduce cache and TLB pollution for the common case

- HOWEVER -

- Increase cache and TLB pollution if not taken
- Maintenance
  - Every feature that affects IPC must also be implemented in ASM (e.g. RT Scheduling, lazy FPU, SMAS, UTCBs ...)
  - Bugfixes in logic must be ported into ASM shortcuts
- Unimplemented for ARM

Fast Long-IPC Copy

- Some CPUs support fast mem-to-mem copy
- Cache-line prefetch
- Cache bypassing
- UX IPC-window bypassing

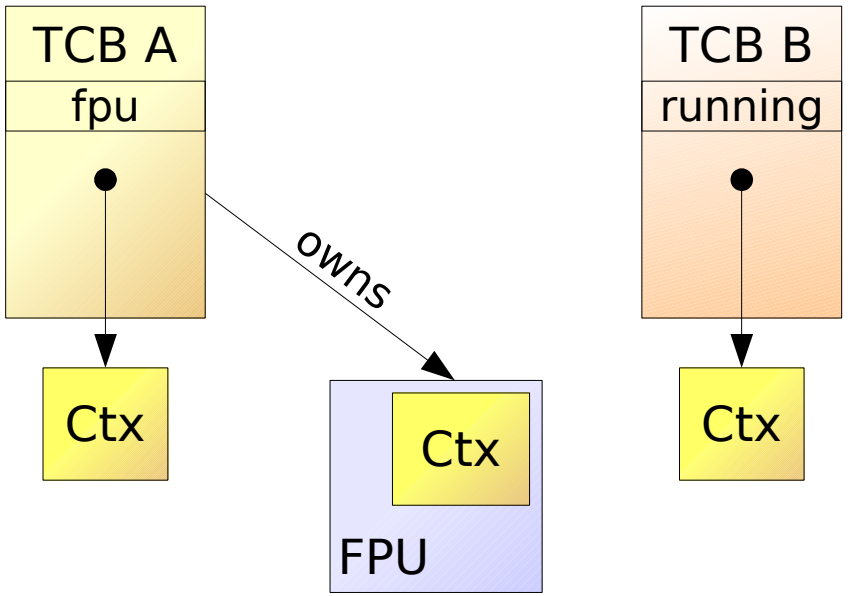
FPU Context Switches

- IA-32/IA-64 have FPU context 0.5KB/2KB
  - Save/restore on each context switch is extremely costly
  - FPU is used infrequently

Lazy Context Save/Restore

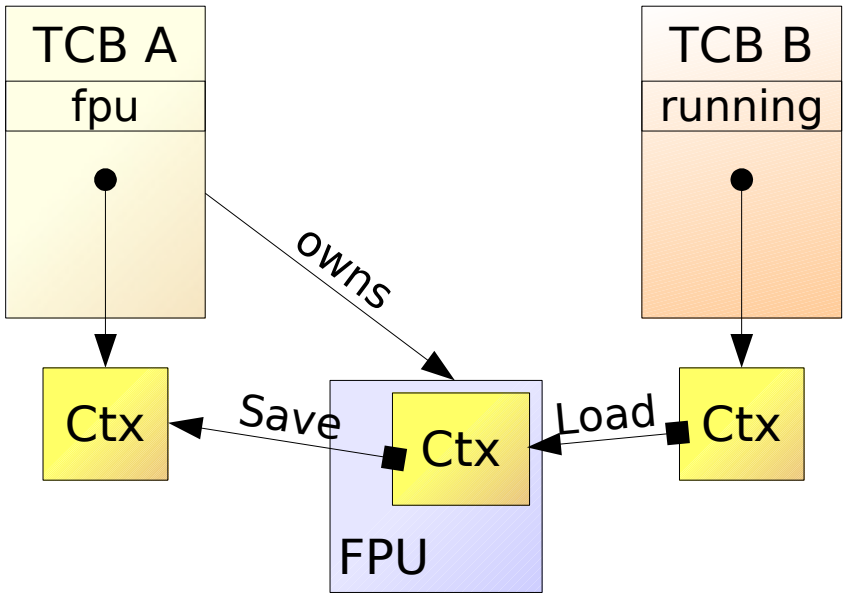
- IA-32, IA-64 support FPU protection/disabling
  - Huge FPU state is saved and restored lazily
    - If only one thread uses FPU state is never saved/restored
    - If some threads use FPU state saving/restoring is minimized
  - Avoid write/read of 0.5kB to 2kB to/from memory
    - Avoid cache/TLB pollution

Lazy FPU Handling



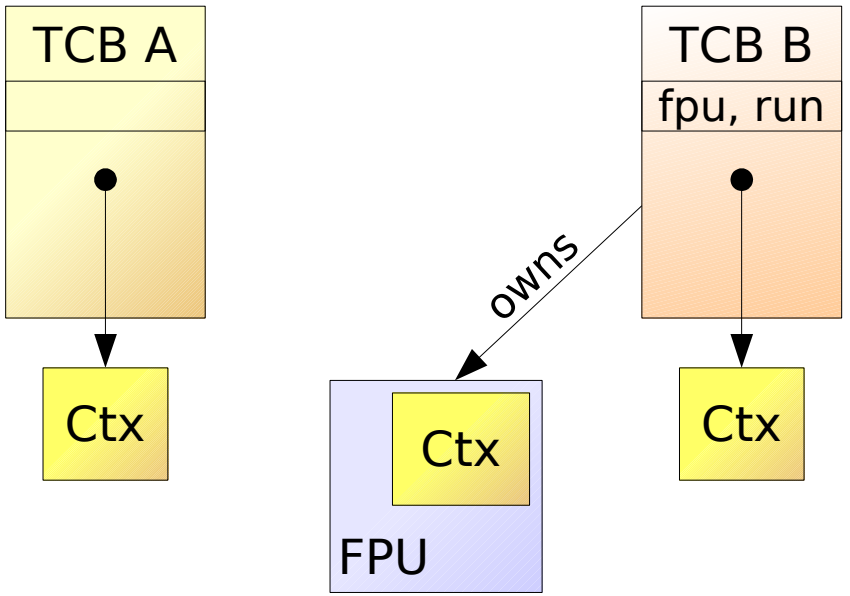
- B is running, A owns FPU
  - FPU is protected

Lazy FPU Handling



- B is running, A owns FPU
  - FPU is protected
- B accesses FPU
  - Save FPU state to A
  - Load FPU state from B

Lazy FPU Handling



- B is running, A owns FPU
- B accesses FPU
  - Save FPU state to A
  - Load FPU state from B
  - B becomes FPU owner
  - FPU is un-protected
- B runs and can use the FPU

Evaluation

- IPC Shortcut:

Ping-Pong IPC benchmark (in cycles)

	Shortcut	Slow Path
Intra AS	775	2183
Inter AS	2142	4209

Intel Pentium 4 (Northwood) 1796MHz

- IA-32 Small Address Spaces:
  - Often 5-10 percent, best case up to 70 percent performance improvement
  - ASID preemption issue