VM and ARM MMU Overview

Mattias Holm & Kristian Rietveld

Universiteit Leiden
The Netherlands

Outline

- Memory management overview.
- ARM MMU Specifics.
- SMACK VM.

Memory Management Overview

Memory Management

- Needed to protect applications from each other.
- Necessary if an application requests more memory than is physically available.
- Every process can behave as if it is the only process running on the machine.

Memory Management (cont.)

- Two major technologies:
  - Segmentation
  - Paging

Segmentation

- Memory segments are defined by a start address and a certain length.
- Each segment has access attributes:
  - Read, write, execute, etc.
- Violation of attributes results in a segment violation or segfault (UNIX SIGSEGV).

Paging

- Introduced to allow memory to be swapped out to disk.
- Memory divided into pages of fixed size (usually 4 KB).
- Pages have access attributes like segments.
- Has mostly replaced segmentation: we will focus on paging.

Address spaces

- Every process gets its own address space.
- Loads/stores occur within this address space, using “virtual addresses”.
- To be able to access physical memory, the virtual address must be translated to a physical address.

OS & CPU

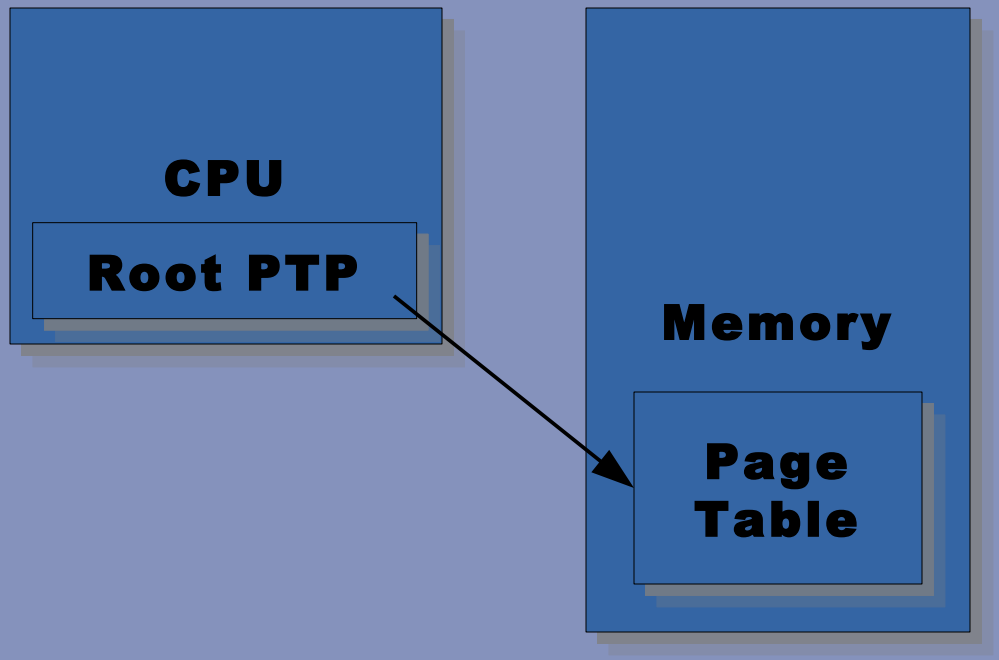
- Address translation and enforcement of page access attributes is in general performed by the CPU.
- Addresses spaces are created and managed by the operating system.
- So, we need to give the CPU the necessary information to perform translation & protection.

Page tables

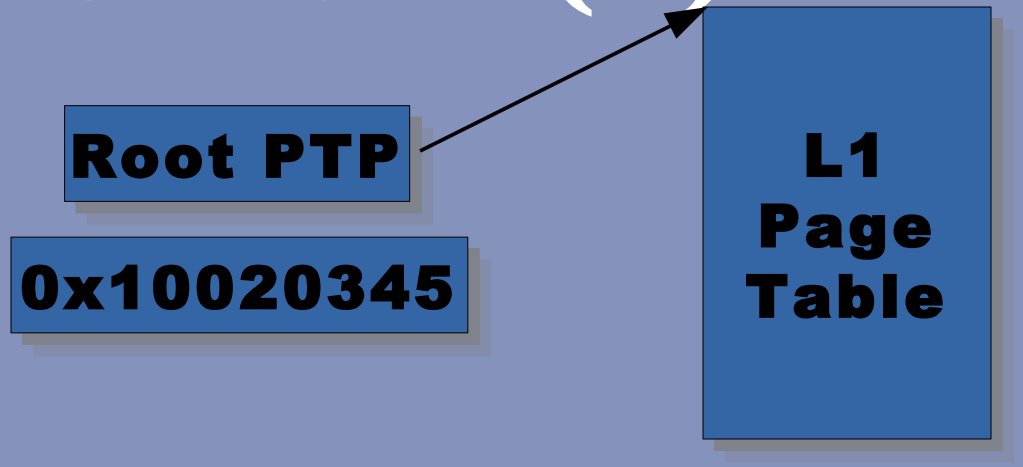
- Page Table Pointers (PTPs) identify paging tables.
- Page Table Entries (PTEs) map virtual to physical addresses and track page attributes.
- Translation Lookaside Buffer (TLB) caches PTEs for quick access.
- If an entry is not in the TLB, memory system will do a page walk.

Page Table Walk

- Processor performs a load/store to an address that is not in the TLB.
- Assume address 0x10020345
- Processor uses the page table pointer (stored in special register) to find the page table.

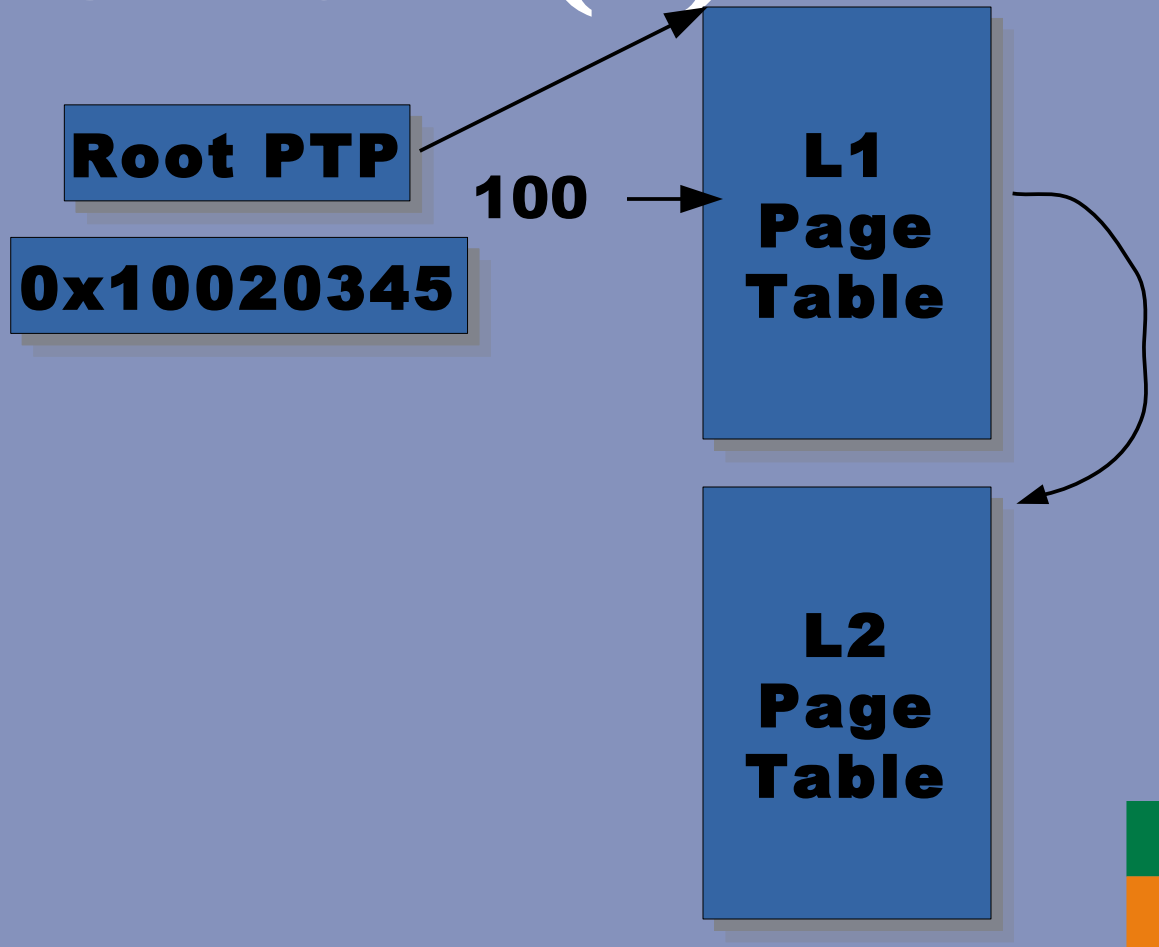


Page Table Walk (1)



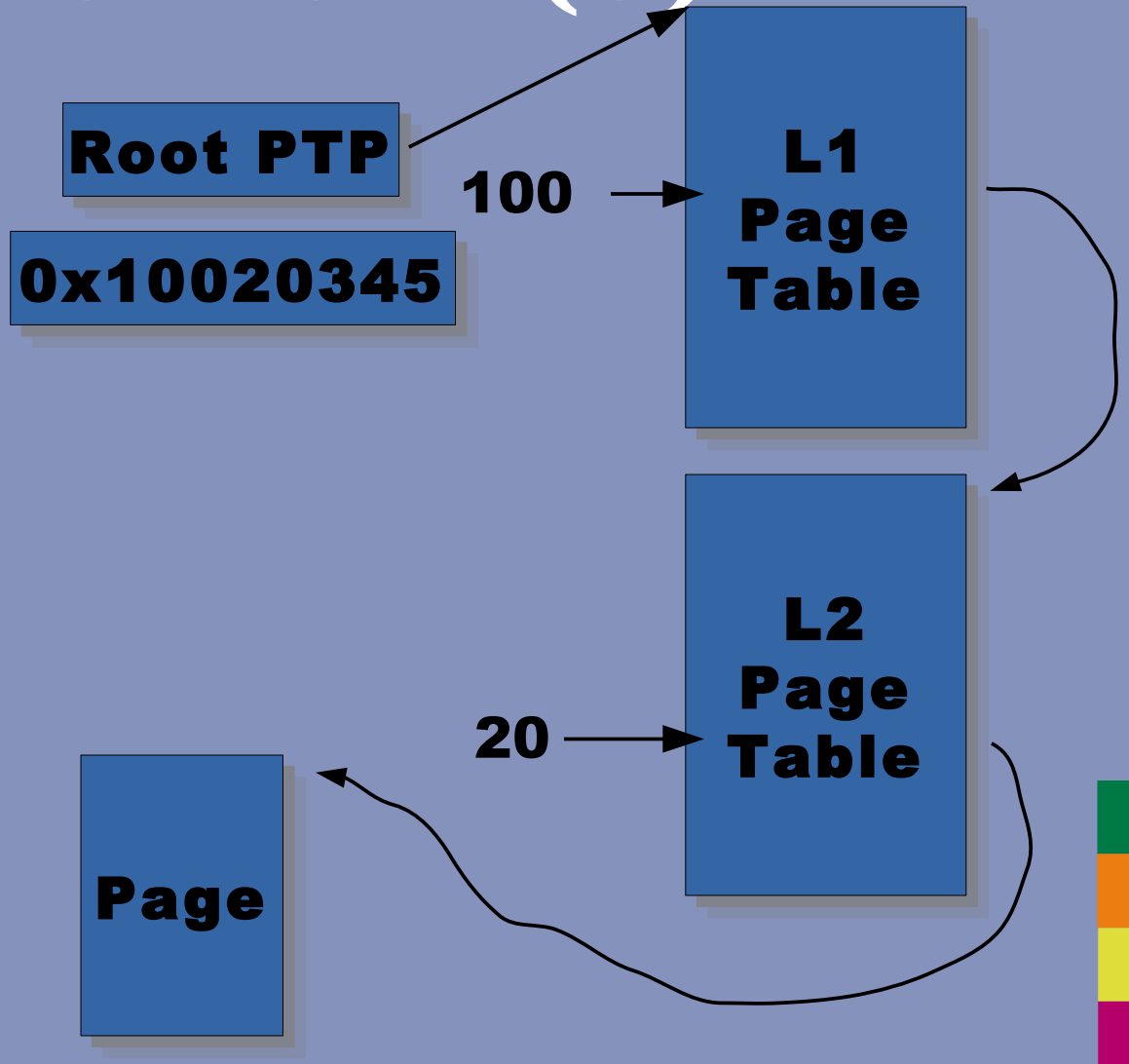
Page Table Walk (2)

- Processor extracts high bits of the virtual address and loads PTP from L1 table.



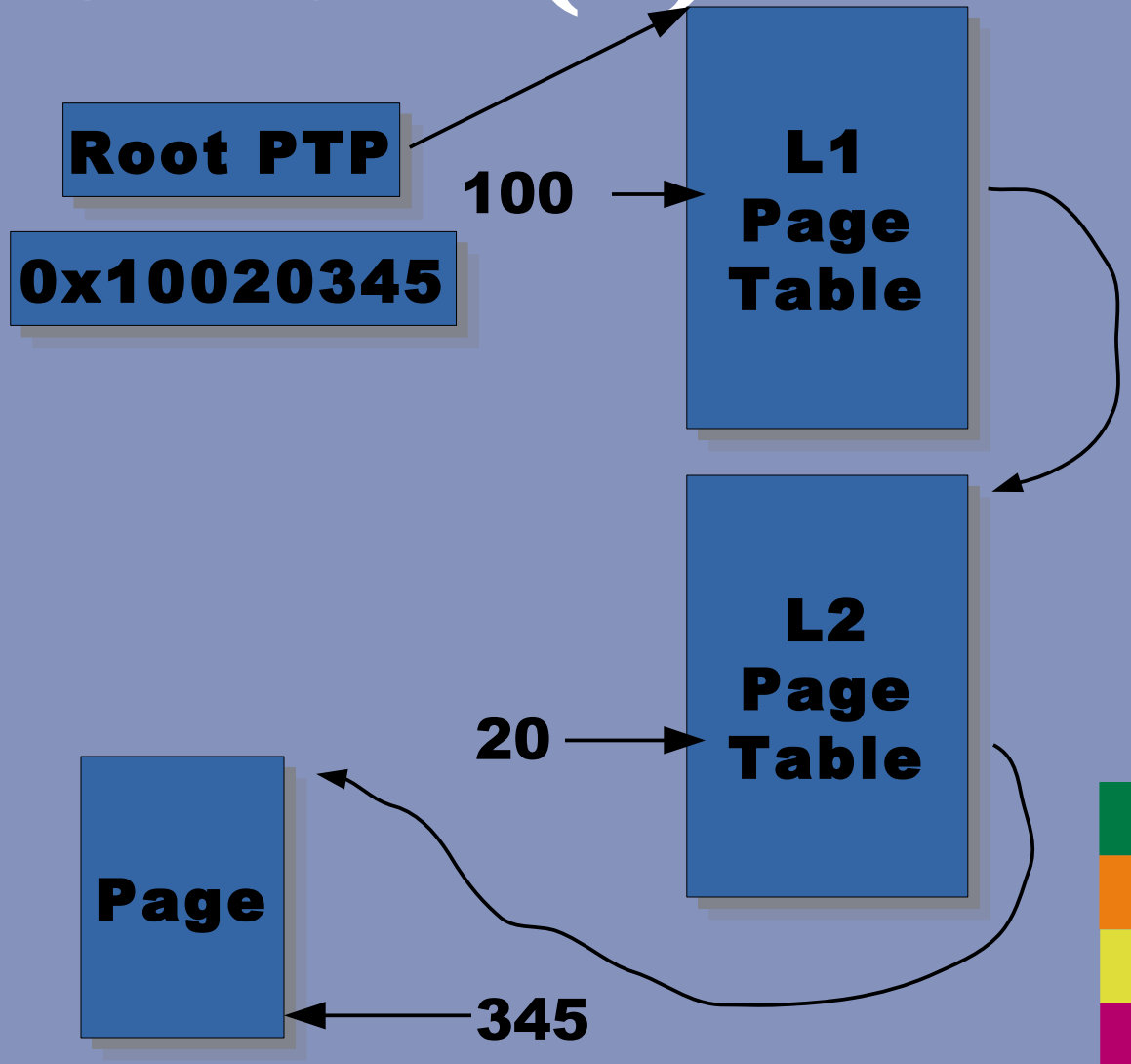
Page Table Walk (3)

- Processor extracts high bits of the virtual address and loads PTP from L1 table.
- Processes uses mid bits to load L2 PTE.



Page Table Walk (4)

- Processor extracts high bits of the virtual address and loads PTP from L1 table.
- Processes uses mid bits to load L2 PTE.
- Use the PTE and lower bits to compute the physical address.



ARM MMU

ARMv7

- ARMv7 comes in 3 variants:
  - ARMv7-A with MMU (paging).
  - ARMv7-R for hard realtime applications with MPU (segmentation, no virtual addressing).
  - ARMv7-M micro-controller version, no memory protection.

ARMv7-A/R

- VMSA – Virtual Memory System Architecture (paging).
- PMSA – Protected Memory System Architecture (segmentation).
- Our kernel runs on ARMv7-A (Cortex-A8).

ARMv7-A MMU

- Control using coprocessor 15 and mrc + mcr instructions.
- 2 level page tables.
- Page sizes: 4KB, 64KB, 1MB, 16MB.
- Permissions for supervisor and user (read/write).
- No-Execute (NX) bit.

Level 1 Table Entries

	31 … 24	23 … 20	19	18	17	16	15	14 … 12	11	10	9	8 … 5	4	3	2	1	0
Fault	IGNORE															0	0
Page table	Page table base address, bits [31:10]										IMP	Domain	SBZ	NS	SBZ	0	1
Section	Section base address, PA[31:20]		NS	0	nG	S	AP[2]	TEX [2:0]	AP [1:0]		IMP	Domain	XN	C	B	1	0
Supersection	Supersection base address PA[31:24]	Extended base address PA[35:32]	NS	1	nG	S	AP[2]	TEX [2:0]	AP [1:0]		IMP	Extended base address PA[39:36]	XN	C	B	1	0
Reserved	Reserved															1	1

Level 2 Table Entries

	31 … 16	15	14 … 12	11	10	9	8 … 7	6 … 5	4 … 3	2	1	0
Fault	IGNORE										0	0
Large page	Large page base address, PA[31:16]	XN	TEX [2:0]	nG	S	AP [2]	SBZ	AP [1:0]		C B	0	1
Small page	Small page base address, PA[31:12]			nG	S	AP [2]	TEX [2:0]	AP [1:0]		C B	1	XN

Page attributes

- Global
- Memory region attributes:
  - Cacheable, Bufferable, TEX.
- Shareable.
- Domain.

Memory Access Attributes

TEX[2:0]	C	B	Description
000	0	0	Strongly ordered
000	0	1	Shareable device
000	1	0	Outer and inner write-through, no write-allocate
000	1	1	Outer and inner write-back, no write-allocate
001	0	0	Outer and inner non-cacheable
001	0	1	Reserved
001	1	0	IMPLEMENTATION DEFINED
001	1	1	Outer and inner write-back, write-allocate
010	0	0	Non-shareable device
010	0	1	Reserved
010	1	-	Reserved
011	-	-	Reserved
1BB	A	A	Cacheable memory; outer = AA, inner = BB

Outer/inner attributes

AA/BB	Attribute
00	Non-cacheable
01	Write-back, write-allocate
10	Write-through, no write-allocate
11	Write-back, no write-allocate

Large Pages & Sections

- Large pages do not decrease table size.
- Sections and super-sections reduce the need for L2 table blocks and the penalty for walking the full table.
- Large pages, sections and super-sections increase the memory covered by the TLB.

Root Table Pointers

- There are two root table pointers, with configurable address coverage:
  - TTBR0: Recommended for user applications (non-global).
  - TTBR1: Recommended for system (global).

TLB & ASID

- The TLB caches the PTEs.
- PTE is in TLB if it is non-global, bound to an ASID which must be synched with the root PTP.
  - No TLB flush necessary on context switch as ASID will change.

For the assignment, you don't have to bother with any of this.

Implementation of Virtual Memory in SMACK

Allocating Virtual Addresses

- Each process has its own VM table.
- Process associated with a sorted linked list that tracks allocated VM blocks.
- Can place allocated page list in a balancing tree for faster search (not done at the moment).

VM allocation

- Operating system needs to:
  - Manage virtual memory (namespace vm_) for every process.
  - Manage physical memory (namespace pmap_) globally.
  - Create page tables (namespace hw_).

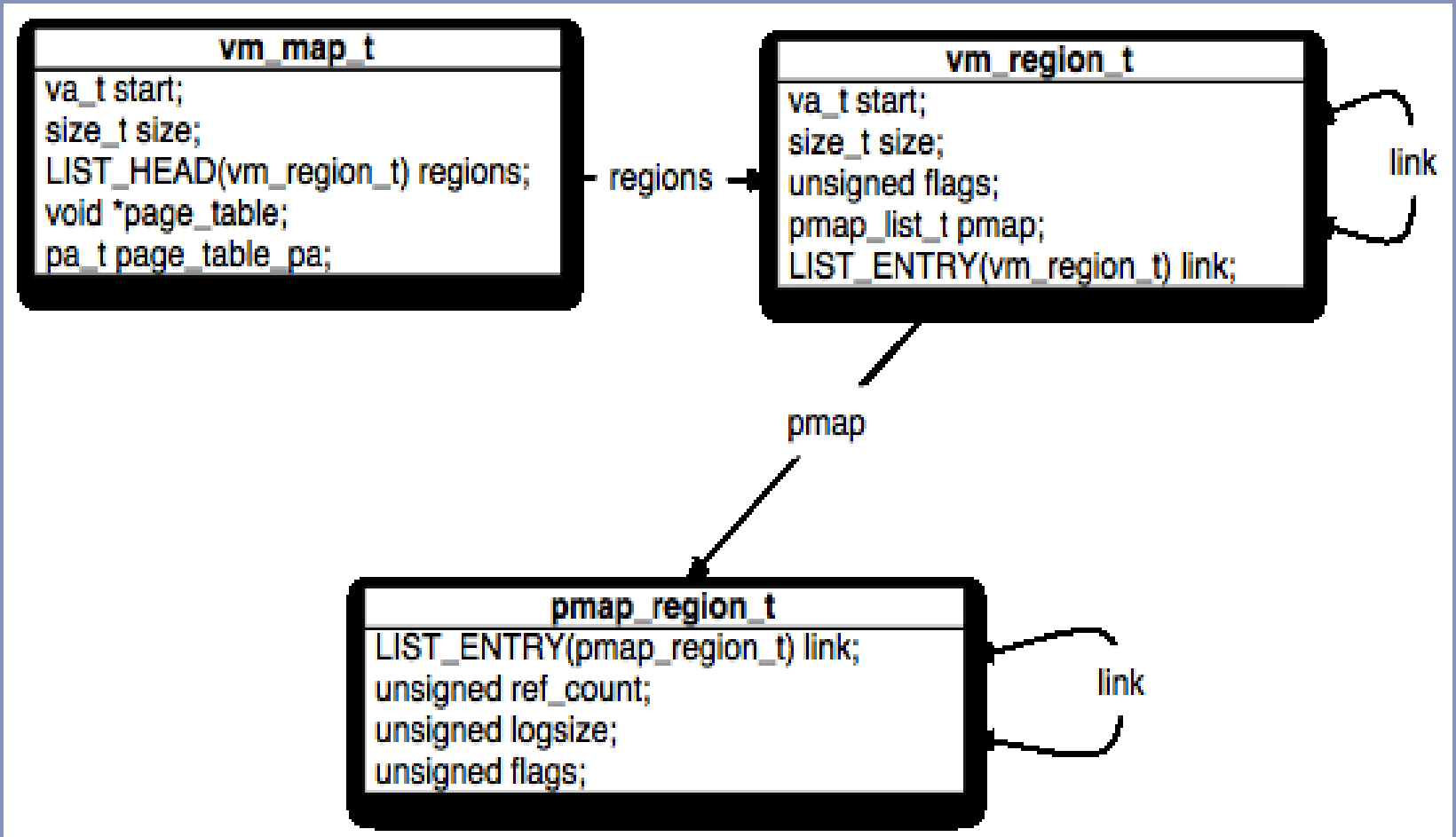
Important Functions

- vm_map()
- vm_unmap()
- vm_map_physical()
- pmap_alloc()
- hw_map()

Important Types

- vm_map_t: a valid range of virtual memory addresses that can be mapped and the mapped vm regions (within that range).
- vm_region_t: a mapped region of memory (i.e. The region has backing physical memory).
- pmap_region_t: physically mapped set of consecutive pages.

Important Types



Allocating Physical Addresses

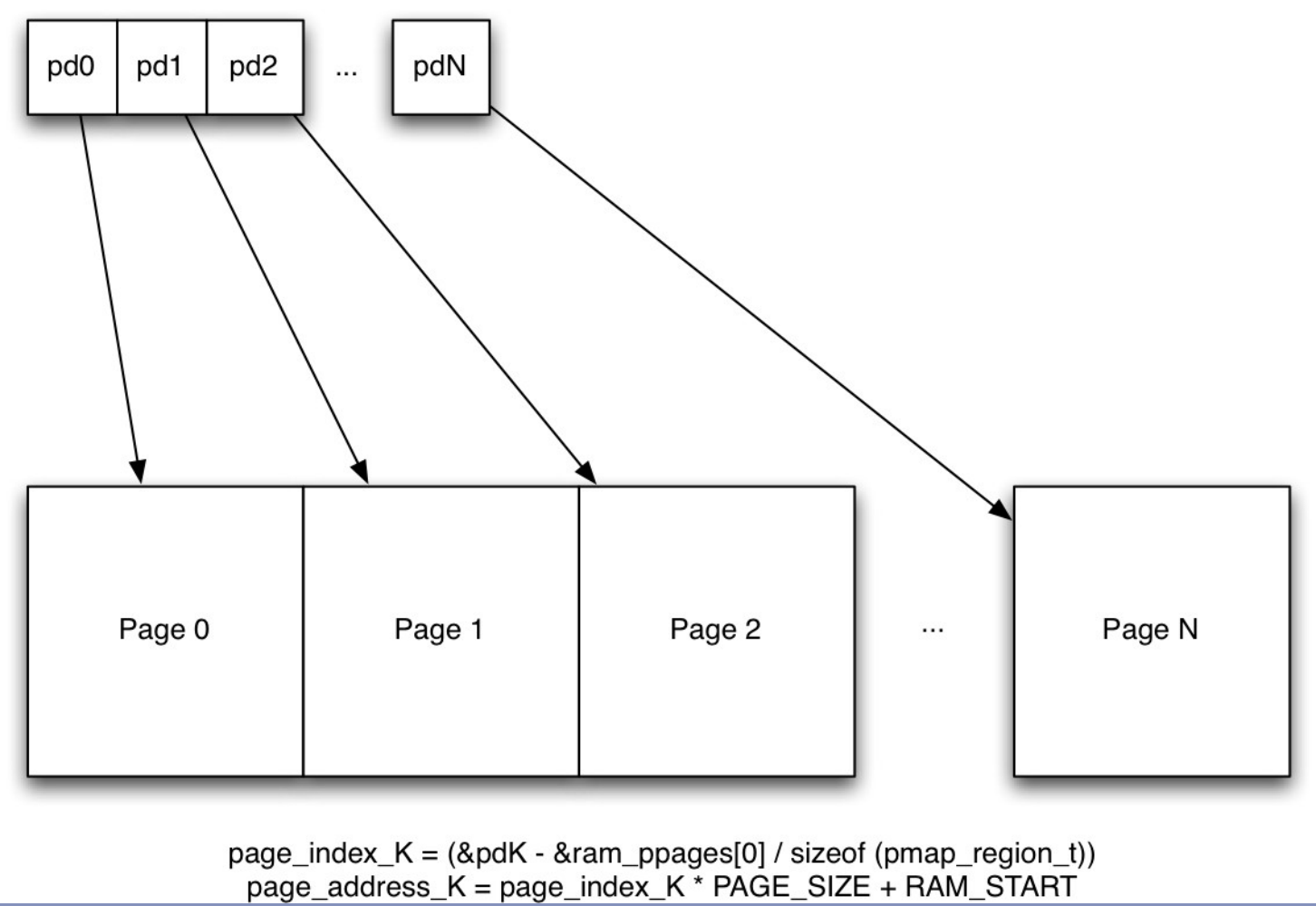
- Kernel needs to track which physical pages have been allocated.
- Need a list of free pages.

- Naive solution: linked list of free pages.
- Better solution: to be implemented by you.

Naive solution

- One global array of pmap_region_t objects. One object describes on page of physical memory.
- Page region object pointers can be directly mapped to the page's physical address.
- List of free and used regions is maintained.
- Goal: get rid of this array, allocate pmap_region_t on demand.

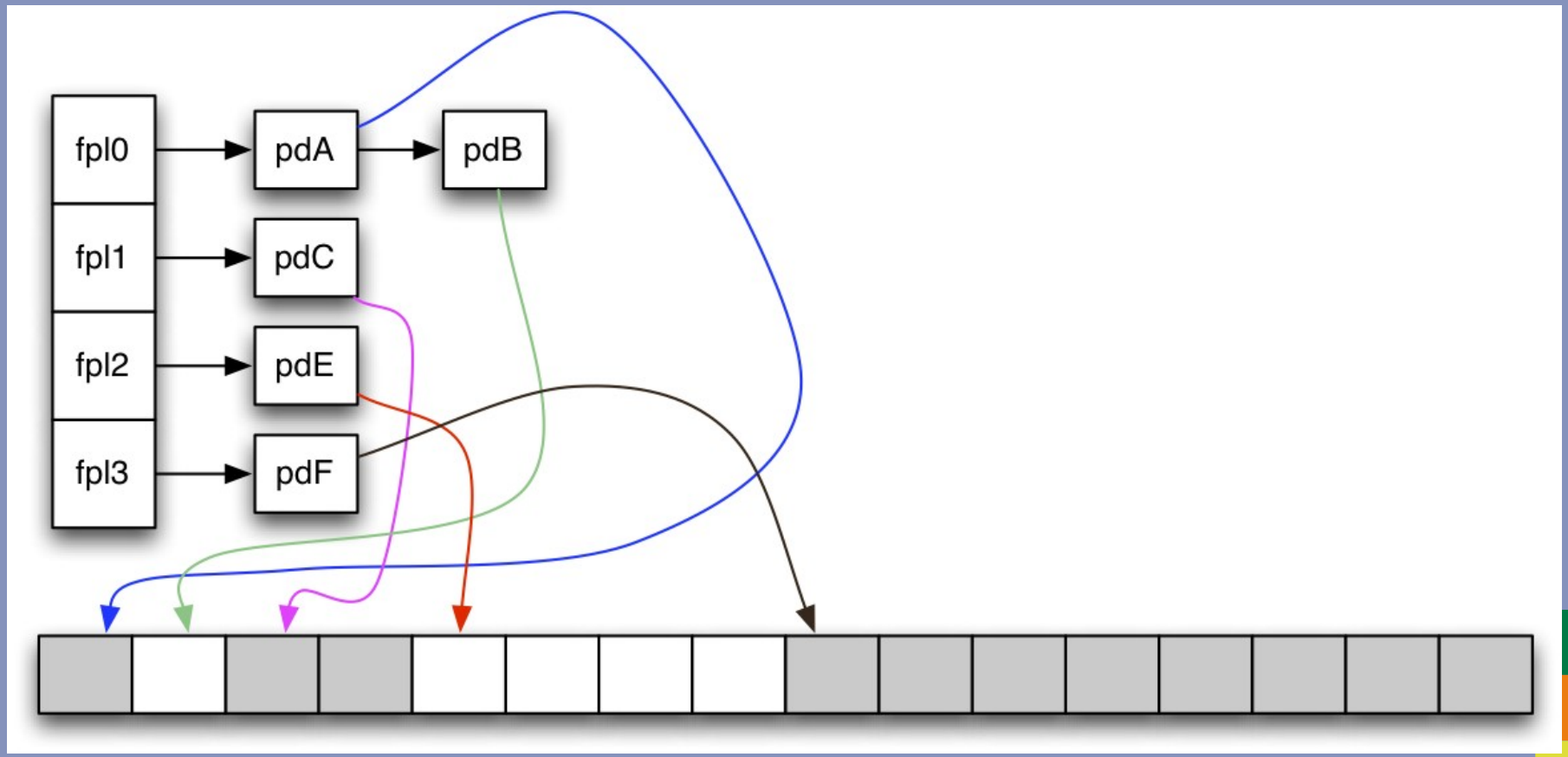
Naive solution



Better solution

- Inspired by Linux.
- A descriptor points at a 2^n sized block of **consecutive** pages.
- For each *n*, maintain a list of free blocks.
- If a block of power *n* is requested, but does not exist, split block of power *n+1*.
- Much less descriptors necessary.
- We maintain blocks of consecutive pages.

Better solution



VM structure allocation

- The kernel needs to allocate VM structures somehow.
- You cannot use generic code in the kernel to allocate space for these structures.
- Why not?

VM structure allocation

- The generic code will again call vm_map(), pmap_alloc(), etc. to allocate new pages to store data on.
- Infinite recursion!
- To break this chain of recursion, you will implement “page stealing”.
- Steal one physical page and use this for storing VM structures and to describe itself.

Stealing pages

- A list of free vm_region_t objects is used.
- Steal first free physical page, map it to first free virtual address.
- Initialize newly mapped page as an array of region descriptors and place these in the list of free vm_region_t objects.
- Use one of the descriptors to describe the stolen page itself.

Working towards a solution

- Start today, not two weeks before the deadline!
- Phase 1: study assignment text and starting point. E.g. how does pmap_init() work and how is physical page allocation carried out?
- Thoroughly understand this before starting implementation!
- 2 weeks studying / 2 weeks coding.

Practicum

- Practicum in zaal 411.