Memory Management
(Muistinhallinta)

Ch 8.3-8.6 [Sta10]
(Virtual) Memory management
Hardware and software support
Example: Pentium & ARM
Teemu’s Cheesecake

Register, on-chip cache, memory, disk, and tape speeds relative to times locating cheese for the cheese cake you are baking...

- **hand**: 0.5 sec (register)
- **refrigerator**: 1 sec (cache)
- **table**: 10 sec (memory)
- **moon**: 12 days (disk)
- **Europa (Jupiter)**: 4 years (tape)

30 ns? 5 ms?
Virtual Memory (virtuaalimuistit)

- Problem: How can I make my (main) memory as big as my disk drive?
- Answer: Virtual memory
  - keep only most probably referenced data in memory, and rest of it in disk
    - disk is much bigger and slower than memory
    - address in machine instruction may be different than memory address
    - need to have efficient address mapping
    - most of references are for data in memory
  - joint solution with HW & SW
Other Problems Often Solved with VM

- If you want to have many processes in memory at the same time, how do you keep track of memory usage?

- How do you prevent one process from touching another process’ memory areas?

- What if a process needs more memory than physically available?
Memory Management Problem

- How much memory for each process?
  - Is it fixed amount during the process run time or can it vary during the run time?

- Where should that memory be?
  - In a continuous or discontinuous area?
  - Is the location the same during the run time or can it vary dynamically during the run time?

- How is that memory managed?

- How is that memory referenced?
Partitioning

How much physical memory for each process?

Static (fixed) partitioning (*kiinteät partitiot, kiinteää ositus*)
- Amount of physical memory determined at process creation time
- Continuous memory allocation for partition

Dynamic partitioning (*dynaamiset partitiot*)
- Amount of physical memory given to a process varies in time
  - Due to process requirements (of this process)
  - Due to system (i.e., other processes) requirements
Static Partitioning

- **Equal size** - give everybody the same amount
  - Fixed size - big enough for everybody
    - too much for most
  - Need more? Can not run!

- **Unequal size**
  - sizes predetermined
  - Can not combine

- **Variable size**
  - Size determined at process creation time
Dynamic Partitioning

- Process must be able to run with varying amounts of main memory
  - all of memory space is not in physical memory
  - need some minimum amount of physical memory

- New process?
  - If necessary reduce amount of memory for some (lower priority) processes

- Not enough memory for some process?
  - reduce amount of memory for some (lower priority) processes
  - kick (swap) out some (lower priority) process
Fragmentation (*pirstoutuminen*)

- **Internal fragmentation** (*sisäinen pirstoutuminen*)
  - unused memory inside allocated block
  - e.g., equal size fixed memory partitions

- **External fragmentation** (*ulkoinen pirstoutuminen*)
  - enough free memory, but it is splintered as many un-allocatable blocks
  - e.g., unequal size partitions or dynamic fixed size (variable size) memory partitions
Address Mapping (osoitteen muunnos)

Pascal, Java:

while (....)  
  X := Y+Z;

Symbolic Assembly Language:

loop:
  LOAD R1, Y
  ADD R1, Z
  STORE R1, X

(Textual) machine language:

1312: LOAD R1, 2510  
      ADD R1, 2514  
      STORE R1, 2600

(real memory addresses)

program load time, or program run time

Execution time:

101312: LOAD R1,102510  
        ADD R1,102514
        ADD R1,102600

compiler
**Address Mapping**

Textual machine language:

1312:  LOAD  R1, 2510

Execution time:

101312:  LOAD  R1,102510  or
101312:  LOAD  R1, 2510  ??

- Want:  R1 ← Mem[102510]  or  Mem[2510] ?
- Who makes the mapping?  When?
Address Mapping, address translation

- At program load time
  - Loader (*lataaja*)
  - Static address binding (*staattinen osoitteiden sidonta*)

- At program execution time
  - CPU
  - With every instruction
  - Dynamic address binding (*dynaaminen osoitt. sidonta*)
  - Swapping (*heittovaihto*)
  - Virtual memory
Virtual Memory Implementation

Methods

- Base and limit registers (*kanta- ja rajarekisterit*)
- Segmentation (*segmentointi*)
- **Paging** (*sivutus*)
- Segmented paging, multilevel paging

Hardware support

- MMU - Memory Management Unit
  - Part of processor
  - Varies with different methods
- Sets limits on what types of virtual memory (methods) can be implemented using this HW
Base and Limit Registers

- Continuous memory partitions
  - One or more (4?) per process
  - May have separate base and limit registers
    - Code, data, shared data, etc
    - By default, or given explicitly in each mem. ref.

- BASE and LIMIT registers in MMU
  - All addresses logical in machine instructions
  - Exec. time address mapping for address (x):
    - Check: $0 \leq x < \text{LIMIT}$
    - Physical address: $\text{BASE} + x$

see Comp. Org I
Virtual memory

- Only needed reserved areas (chunks) in the memory, no need to be contiguous
  - Demand paging (*tarvenouto*)

- Chunk size?
  - Fixed size = Paging
  - Variable size = Segmentation
  - Combined = Paged segments, multilevel paging

- OS bookkeeping (*KJ kirjanpito*)
  - **Page frame table** (*sivutilataulu, sivukehystaulu*)
    - Which physical page frames are free, which are occupied
  - Each process has its own **page table** (*sivutaulu*)
    - Is this page in memory or on disk? -- ”presence-bit”
    - In memory, which physical frame contains this logical page?
    - Other control? Bits: Modified, Referenced

OS course content

Paging “most common” ⇔ here only paging
Virtual Memory: Paging (sivutus)

Load A

Program: pages
Memory: frames

OS loads process A from disk

(Sta10 Fig 8.15)
Address Mapping with Paging VM
(Sivuttavan virtuaalimuistin osoitteenmuunnos)

(Sta10 Fig 8.16)
Paged Address Translation

Virtual address: 1 30
Access type: r

Page table:
0: 0 rwx 65
1: 1 rw 14
2: 1 rw 55
...

Check for valid entry
Valid entry: 14 30
Access rights: r \in \{rw\}
Page frame
Physical address

Check access rights:
(virt. mem. used to solve memory protection problem)
Page Fault

Page fault interrupt

Stop execution
Initiate reading page 1 from disk
Schedule next process to run
I/O interrupt
Page 1 read, update page table
Make orig. process ready-to-run

Virtual address

Access type

r

Check for valid entry:
not valid!

Page table

Check access rights
r ∈ {rw}

Physical address

Schedule orig. process again, at the same instruction

Page table

0: 0 rwx 65
1: 1 rw 14
2: 1 rw 55

...
Virtual memory: Translation Lookaside Buffer (TLB) *(osoitteenmuunnospuskuri)*

- Address translation for each memory reference, at least once for each instruction

- Page table elements in memory = extra memory access?
  - Too slow!

- Solution
  - Principle of locality! Page table element referenced soon again
  - Store recently used page table elements (of this process) in TLB on CPU’s memory management unit

- TLB, translation lookaside buffer
  - Just like cache
  - Fast set of registers (Pentium: 32 registers)
  - Associative search, but could use direct mapping
  - Hit ratio (*Osumatodennäköisyys*) 99.9% ? (Almost always!)
Example: Direct-Mapped 16-entry TLB

Physical address

0x00B6C8E6 046

ReadW I2, 0xAB00C7DA 046

Match

Correct address mapping found

? = Match

Physical address

Tag 28

Index 4

Page offset

0000:

....

0111:

....

1000:

....

1001:

....

1010:

....

AB00C7D 00B6C8E6

Page frame 32
Translation Lookaside Buffer (TLB)

“Hit” on TLB?
- address translation is in TLB - real fast

“Miss” on TLB?
- must read page table entry from memory
- takes time – not much, just a memory reference
  - Entry might be in cache!
- cpu waits idle until it is done

Just like normal cache, but for address mapping
- implemented just like cache
- instead of cache line data has physical address
- split TLB? 1 or 2 levels?

Discussion?
TLB Miss and Page Fault

(TLB huti ja Sivunpuutos-keskeytys)

Process switch (minimum 2)

(Sta10 Fig 8.18)
Hardware support for virtual memory

- MMU and its special registers
  - PTR (page table register)
    - Physical start address of process page table
      (copied from PCB – process control block)
  - TLB (translation lookaside buffer)
    - Caches page table entries from earlier address mappings
  - “Page fault” – interrupt
  - Updating reference and modified bits

- Process switch \((prosessin vaihto)\)
  - PTR register \(\leftarrow\) Physical start address of process page table
  - Invalidate old TLB content (it is usually process specific)
    - Each location has valid bit
  - Copy dirty cache lines back to memory
Memory/Disk Organisation

CPU
- instr
- regs
- TLB
- cache
- mem.addr
- data

Bus

Memory
- page table
- page

Disk
- page table
- page
- page
- page
- page
TLB and Cache

Page table entry can be found from cache!

(Sta10 Fig 8.19)
TLB vs. Cache

**TLB Miss**
- CPU waits idling
- HW implementation
- Invisible to process
- Data is copied from memory to TLB
  - from page table data
  - from cache?
- Delay 4 (or 2 or 8?) clock cycles

**Cache Miss**
- CPU waits idling
- HW implementation
- Invisible to process
- Data is copied from memory to cache
  - from page data
- Delay 4 (or 2 or 8?) clock cycles
TLB Misses vs. Page Faults

**TLB Miss**
- CPU waits idling
- HW implementation
- Data is copied from memory to TLB (or from cache)
- Delay 1-8 (?) clock cycles

**Page Fault**
- Process is suspended and cpu executes some other processes
- SW implementation
- Data is copied from disk to memory
- Delay 10-30 ms(?)
Inverted page table
(käänteinen sivutaulu)

- Just one shared inverted page table
- MMU: PTR (page table reg), PidR (process id register), TLB

(Synopsis Fig 8.17)
Hierarchical page table
(\textit{monitasoimen sivutaululu})

- Several systems allow large virtual address space
  - Page table split to pages, some of it on the disk
  - Top level of page table fits to one page, always in memory

\begin{itemize}
  \item 4-kbyte root page table (page table directory)
  \item 4-Mbyte user page table (in virtual memory)
  \item 4-Gbyte user address space
\end{itemize}

\begin{itemize}
  \item 1 K items \((= 1024 = 2^{10})\)
  \item 1K * 1K = 1M items
\end{itemize}

\begin{itemize}
  \item 32b osoite
    \begin{tabular}{c c c}
      Dir & Page & Offset \\
      10 & 10 & 12
    \end{tabular}
\end{itemize}

\(1K \times 1K = 1M \text{ items}\)
Virtual Memory Policies

- Fetch policy (*noutopolitiikka*)
  - demand paging: fetch page only when needed 1st time
  - working set: keep all needed pages in memory
  - prefetch: guess and start fetch early

- Placement policy (*sijoituspolitiikka*)
  - any frame for paged VM

- Replacement policy (*poistopolitiikka*)
  - local, consider pages just for this process for replacement
  - global, consider also pages for all other processes
  - dirty pages must be written to disk (*likaiset, muutetut sivut*)

More details in an Operating Systems course
Virtual Memory Example
Pentium (IA-32)
Pentium support for memory management

- Unsegmented unpaged, max $2^{32} = 4$ GB
  - Virtual address = physical address
  - Efficient $\Rightarrow$ feasible in real-time systems

- Unsegmented paged (Sivuttava), max 4 GB
  - Linear address space (lineaarinen osoiteavaruus)
  - Page and frame size: 4KB or 4MB
  - Memory protection is page frame based

- Segmented unpaged (Segmentoiva), max $2^{48} = 64$ TB
  - Several segments $\Rightarrow$ several linear memory spaces
  - Memory protection is segment based

- Segmented paged (Sivuttava segmentointi), max 64 TB
  - Memory management using pages and page frames
  - Memory protection is segment based
• If Paging=Enabled, use page tables
  else linear address = physical address (e.g., OS Devide drivers?)
• Control registers (see further in the text book)
Pentium: Segment Descriptor (segmenttikuvaaja)

Segment Descriptor (Segment Table Entry)

Base
Defines the starting address of the segment within the 4-GByte linear address space.

D/B bit
In a code segment, this is the D bit and indicates whether operands and addressing modes are 16 or 32 bits.

Descriptor Privilege Level (DPL)
Specifies the privilege level of the segment referred to by this segment descriptor.

Granularity bit (G)
Indicates whether the Limit field is to be interpreted in units by one byte or 4 KBytes.

Limit
Defines the size of the segment. The processor interprets the limit field in one of two ways, depending on the granularity bit: in units of one byte, up to a segment size limit of 1 MByte, or in units of 4 KBytes, up to a segment size limit of 4 GBytes.

S bit
Determines whether a given segment is a system segment or a code or data segment.

Segment Present bit (P)
Used for nonpaged systems. It indicates whether the segment is present in main memory. For paged systems, this bit is always set to 1.

Type
Distinguishes between various kinds of segments and indicates the access attributes.

(Sta10 Table 8.5)
### Page Directory Entry and Page Table Entry

**Accessed bit (A)**
This bit is set to 1 by the processor in both levels of page tables when a read or write operation to the corresponding page occurs.

**Dirty bit (D)**
This bit is set to 1 by the processor when a write operation to the corresponding page occurs.

**Page Frame Address**
Provides the physical address of the page in memory if the present bit is set. Since page frames are aligned on 4K boundaries, the bottom 12 bits are 0, and only the top 20 bits are included in the entry. In a page directory, the address is that of a page table.

**Page Cache Disable bit (PCD)**
Indicates whether data from page may be cached.

**Page Size bit (PS)**
Indicates whether page size is 4 KByte or 4 MByte.

**Page Write Through bit (PWT)**
Indicates whether write-through or write-back caching policy will be used for data in the corresponding page.

**Present bit (P)**
Indicates whether the page table or page is in main memory.

**Read/Write bit (RW)**
For user-level pages, indicates whether the page is read-only access or read/write access for user-level programs.

**User/Supervisor bit (US)**
Indicates whether the page is available only to the operating system (supervisor level) or is available to both operating system and applications (user level).
Virtual Memory Example

ARM
ARM Memory System Overview

Memory Management Unit (MMU)

Access control hardware

Access bits, domain

Abort

TLB

Virtual address

Virtual memory translation hardware

Physical address

Main memory

ARM core

Control bits

Cache and write buffer

Virtual address

Physical address

Cache line fetch hardware

(Sta10 Fig 8.22)
ARM Virtual Memory Address Translation for Small Pages - Diagram

- Single L1 page table
  - 4K 32-bit entries
  - Each L1 entry points to L2 page table
- Each L2 page table
  - 256 32-bit entries
  - Each L2 entry points to page in main memory
- 32-bit virtual address
  - 12 bit - L1
  - 8 bit - L2
  - 12 bit - offset (=page index)

(Sta10 Fig 8.23)
ARMv6 Memory Management Formats

Second level page table:
- Large page entry replicated 16 times
- Mix of small and large pages allowed

(Sta10 Fig 8.24)
Hennessy-Patterson: Computer Architecture, Fig 5.47 Alpha AXP

Instr. TLB
fully assoc,
12 entries

Instr. CACHE
direct mapped,
8 KB,
256 lines (a’32B)

Data TLB
fully assoc,
32 entries

Data CACHE
direct mapped,
8 KB,
256 lines

Unified Level 2 CACHE
2 MB, 64K lines (a’32B)
direct mapped,
write-back

Main Memory

Disk
Virtual Memory Summary

- How to partition physical memory for processes?
  - Fixed partitions (various methods)
  - Dynamic partitions: segments, pages

- Paged virtual memory
  - Multilevel page tables

- How to translate addresses?
  - TBL, multi-level TLB

- How does TBL work with cache?

- Examples: Intel & ARM
Review Questions

- What hardware support is needed for virtual memory implementation?
- Differences of paging and segmentation?
- Why to combine paging and segmentation?
- Relationship of TLB and cache?
  - similarities, differences?