Memory Management

- Programmer wants memory to be
  - Indefinitely large
  - Indefinitely fast
  - Non volatile

- Memory hierarchy
  - Small amount of fast, expensive memory - cache
  - Some medium-speed, medium price main memory
  - Gigabytes of slow, cheap disk storage

- Memory manager handles the memory hierarchy

- Requirements for memory management
  - Logical and physical organization
  - Protection and sharing
Memory management

- Programs use logical addresses of their own address space (0..MAX)
- OS kernel usually in a fixed location using physical memory addresses directly
- Rest of the physical memory for user processes and other OS parts
- OS task: memory allocation and process relocation
- Hardware task: address translation (to protect memory)
  - MMU - memory management unit
Basic Memory Management: One program
Monoprogramming without Swapping or Paging

No memory abstraction, no address space, just an operating system with one user process
Multiprogramming with Fixed Partitions

- OS places one process in one partition
- Internal fragmentation (whole partition allocated to a smaller process)
Fixed partitions

- Process queue for partition on disk
  - In shared queue for all partitions or
  - In multiple queues for different sizes,
- No free partition, OS can swap
  - Move one process to disk
  - PCB always in memory
- Program too large for any partition; programmer has to design solution
  - Overlaying: keep just part of the program in memory
  - Write the program to control the part swapping between memory and disk
Relocation and Protection

- Cannot be sure where program will be loaded in memory
  - address locations of variables, code routines cannot be absolute
  - must keep a program out of other processes’ partitions

- Use **base and limit** values
  - address locations added to base value to map to physical addr
  - address locations larger than limit value is an error

- **Address translation by MMU (hardware)**
Sharing

- Access to shared code / data
  - No violation of the protections!
- Shared code
  - must be reentrant, not to change during execution
  - Just one copy of the shared code (e.g. library)
- Shared data
  - Processes co-operate and share data structures
  - E.g. shared buffer of producer and consumer

- Solution: system calls between processes, threads within a process (more solutions in virtual memory)
Swapping (1)

Memory allocation changes as
- processes come into memory
- leave memory

Shaded regions are unused memory
Swapping (2)

- (a) Allocating space for growing data segment
- (b) Allocating space for growing stack & data segment
No fixed predetermined partition sizes

External fragments: \(6M + 6M + 4M = 14M\)

OS could occasionally reorganize the memory \((\text{compaction})\)
Memory Management: bookkeeping allocations and free areas

- Part of memory with 5 processes, 3 holes
  - tick marks show allocation units
  - shaded regions are free
- (b) Corresponding bit map
- (c) Same information as a list
Combining freed areas

Four neighbor combinations for the terminating process X
Allocation

Where to place the new process?

Goal: avoid external fragmentation and compaction

Some alternatives:

- Best-fit
- First-fit
- Next-fit
- Worst-fit
- Quick-fit

Sta Fig 7.5
Virtual memory
(using paging)
Paging

- **OS**: the program split to pages
  - Page location stored in page table

- **Process location**
  - Logical address always the same
  - MMU translates logical address to physical address using page table
  - Each page relocated separately

Sta Fig 7.11
Paging

- Each process has own page table
  - Contains the locations (frame numbers) of allocated frames
  - Page table location stored in PCB, copied to PTR for execution
- OS maintains a table (or list) of page frames, to know which are unallocated
Paging: Address Translation

MMU has one special register, Page Table Register (PTR), for address translation.
Paging: Address Translation

Virtual Address
Page #  Offset

Register
Page Table Ptr

Page Table

Frame #
Offset

Frame #

Page Frame

Program  Paging Mechanism  Main Memory
Page table
Each process has its own page table

Each entry has a present bit, since not all pages need to be in the memory all the time -> page faults

Remember the principle of locality

Logical address space can be much larger than the physical
Page Tables

Internal operation of MMU with 16 4 KB pages
Translation Lookaside Buffer - TLB
(osoitteenmuunnos-puskuri)
### TLB - Translation Lookaside Buffer

<table>
<thead>
<tr>
<th>Valid</th>
<th>Virtual page</th>
<th>Modified</th>
<th>Protection</th>
<th>Page frame</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>140</td>
<td>1</td>
<td>RW</td>
<td>31</td>
</tr>
<tr>
<td>1</td>
<td>20</td>
<td>0</td>
<td>R X</td>
<td>38</td>
</tr>
<tr>
<td>1</td>
<td>130</td>
<td>1</td>
<td>RW</td>
<td>29</td>
</tr>
<tr>
<td>1</td>
<td>129</td>
<td>1</td>
<td>RW</td>
<td>62</td>
</tr>
<tr>
<td>1</td>
<td>19</td>
<td>0</td>
<td>R X</td>
<td>50</td>
</tr>
<tr>
<td>1</td>
<td>21</td>
<td>0</td>
<td>R X</td>
<td>45</td>
</tr>
<tr>
<td>1</td>
<td>860</td>
<td>1</td>
<td>RW</td>
<td>14</td>
</tr>
<tr>
<td>1</td>
<td>861</td>
<td>1</td>
<td>RW</td>
<td>75</td>
</tr>
</tbody>
</table>

**Goal is to speed up paging**

**TLB is a cache in MMU for page table entries**

*(hardware issue)*
TLB - translation lookaside buffer

- Part of memory management unit (MMU)
  - Cache for used page table entries to avoid extra memory access during address translation
- Associative search
  - Compare with all elements at the same time (fast)
- Each TLB element contains: page number, page table entry, validity bit
- Each process uses the same page numbers 0, 1, 2, …, stored on different page frames
  - TLB must be cleared during process switch
  - At least clear the validity bits (this is fast)
Operation of Paging and TLB

Sta Fig 8.8.
Fig. 5.47 from Hennessy-Patterson, Computer Architecture

DEC Alpha AXP 21064 memory hierarchy

- Fully assoc, 32 entry data TLB
  - 8 KB, direct mapped, 256 line (each 32B)
- Fully assoc, 12 entry instruction TLB
  - 8 KB, direct mapped, 256 line (each 32B)
- Data cache
  - 2 MB, 64K line (each 32B)
- Instruction cache
  - 2 MB, 64K line (each 32B)
- L2 cache
  - Fully assoc, direct mapped, unified, write-back
- Main memory
- Paging disk (DMA)
TLB and cache

Sta Fig. 8.10
Multilevel and Inverted Page Tables
Multilevel page table

- Large virtual address space
  - Logical address could be 32- or 64-bits
- Each process has a large page table
  - Using 32-bit address and frame size 4KB (12 bit offset), means \(2^{20} = 1\text{M}\) of page tables entries for a single process
  - Each entry requires several bytes (let's say 4 bytes), so the final size of page table could be for example 4 MB
- Thus the page table is divided to several pages also and part of it can be on the disk
  - Only the part of the page table that covers the pages used currently in the execution of the process is in memory
Two-level hierarchical page table

- Top-most level in one page and always in the memory

1 K entries (= 1024 = 2^{10})

1K * 1K = 1M entries

Figure 8.4 A Two-Level Hierarchical Page Table
Address translation with two levels

(Fig 4-12 [Tane01])
Address translation with two levels

Sta Fig 8.5
Inverted Page Tables

Comparison of a traditional page table with an inverted page table

- Traditional page table with an entry for each of the $2^{52}$ pages
- 256-MB physical memory has $2^{16}$ 4-KB page frames
- Indexed by virtual page
- Indexed by hash on virtual page
- Virtual page
- Page frame
Inverted page table

- Physical memory often smaller than the virtual address space of processes
- Invert booking: Store for each page frame what page (of which process) is stored there
  - Only one global table (inverted page table), one entry for each page frame.
- Search for the page based on the content of the table
  - Inefficient, if done sequentially,
  - Use hash to calculate the location, start search from there
- If page not found, page fault
- Useful, only if TLB is large
Inverted page table

- Frame number
- Index of the table
- Not stored in the entry

Sta Fig 8.6

Figure 8.6 Inverted Page Table Structure
Virtual memory
Operations and policies

Chapters 3.4. – 3.7
Policies and methods

- **Fetch policy (Noutopoliikka)**
  - When to load page to memory?

- **Placement policy (Sijoituspolitiikka)**
  - Where to place the new page?

- **Replacement policy (Korvaus/poistopolitiikka)**
  - Which page to be evicted and replaced

- **Cleaning policy (Levylle kirjoitus-politiikka)**
  - When to evict a modified page?

- **Multiprogramming, load control (Moniajoaste)**
  - How many processes in the system at the same time?
  - How much memory, how many pages per process?

- **Working set (Käyttöjoukko)**
  - How many page frames allocated for one process? (resident set)
  - Which areas have been referenced recently?
Perform page replacement
Page fault (*Sivun puutos*)

- If the requested page is not in memory, MMU causes interrupt
- OS process the interrupt and notice a page fault
  - Move the process to blocked state
  - Allocate page frame for the new page
  - Start page load from disk using driver to command controller
  - Switch another process for execution
- After the device interrupt
  - OS notices the termination of page transfer
  - Move process to ready state
  - Continue the process now or later (scheduling decision)
Locking pages

- To avoid page fault, page can be locked to memory
  - Kernel pages
  - Key data structures of OS
  - I/O buffers (at least during the transfer)
  - Process pages (at least in real time systems)

- How?
  - Page frame table has a lock bit on the frame
    - Per frame
  - Or page table entry has the lock bit
    - Per process
Page replacement

- **When?**
  - Page fault
  - Memory too full, no free frames (usually some kept free!)
  - Not enough free unallocated frames (limit value)

- **Which page to replace?**
  - One, that is not needed in the (near) future
    - Predict based on the past behaviour
    - Locality principle helps: no more references to the page, process passed that phase
    - Sometimes mistakes happen

See Fig 8.1 [Stal05]