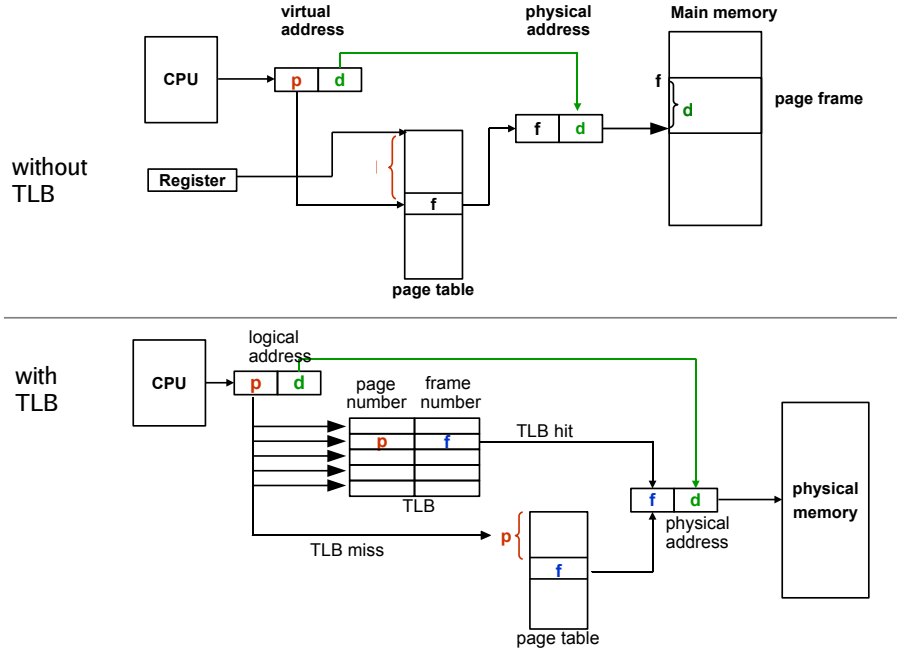
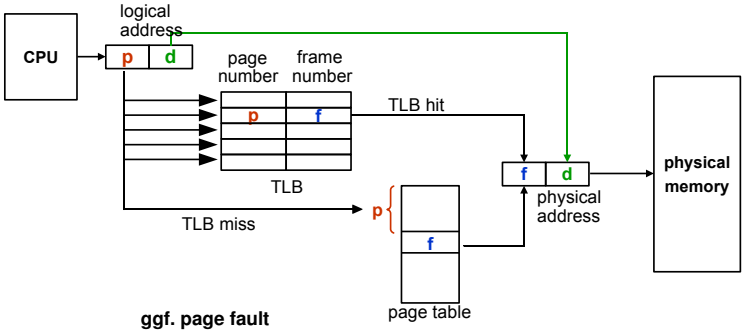
Memory Management (3)



Translation Look-Aside Buffer (1)

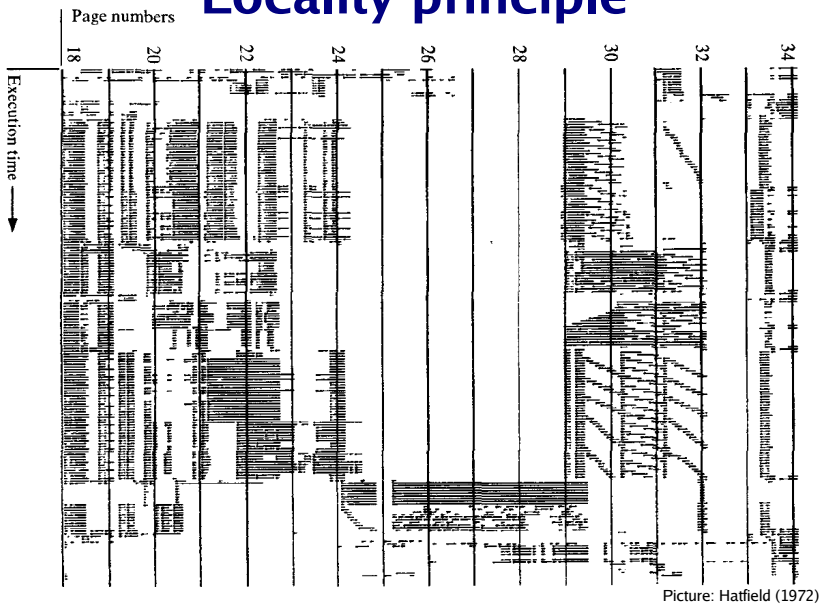
- Translation Lookaside Buffer (TLB): fast hardware cache, holds the most recently used page table entries
- Associative memory: when translating an address the page number is compared with all TLB entries in parallel.



Translation Look-Aside Buffer (2)

- TLB hit
  -> accessing the page table is not necessary
- TLB miss
  -> access the page table
  replace an old TLB entry with the new info
- hit ratio influences average time needed for an address translation.
- Lokality principle: Programs typically access neighboring addresses
  -> even with small TLBs high hit ratios (typically: 80-98%).

Locality principle

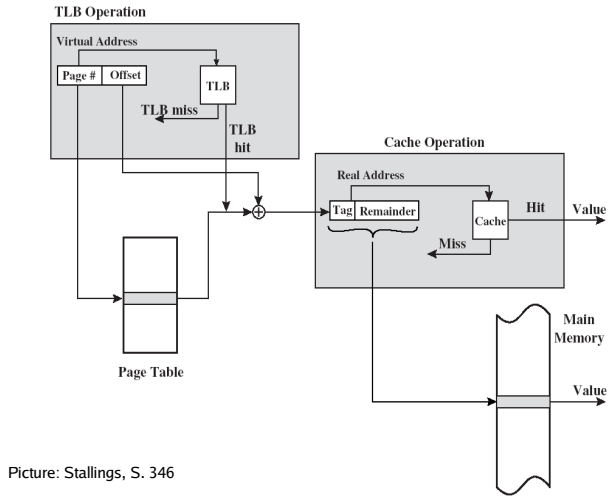


Picture: Hatfield (1972)

Translation Look-Aside Buffer (3)

- Contents of TLB are process-specific! Two possibilities:
  - Each TLB entry has a „valid bit“. At context switch the whole contents of the TLB are invalidated.
  - Each TLB entry contains process identification (PID) that is compared with the PID of the accessing process.
- Examples for TLB sizes:
  - Intel 80486: 32 entries
  - Pentium-4, PowerPC-604: 128 entries each for code and data

Translation Look-Aside Buffer (4) – with memory cache



Picture: Stallings, S. 346

Translation Look-Aside Buffer (5)

What are the operating system's tasks?

- load page table register
- In case of page fault: retrieve missing page from the swap and refresh page table
- possibly before that: page replacement – what page should be removed from main memory? (-> later)

Everything else: hardware

- TLB access and possibly page table access
- When page is in memory: calculation of the phys. address
- read contents from cache or perhaps main memory

Software-TLB

- Alternative to CPU/MMU handling the TLB
- If a page isn't found in the TLB, the MMU generates a TLB fault
- operating system runs an error handler for the TLB fault:
  - search for page
  - pick a TLB entry that can be replaced
  - overwrite TLB entry with the new page/page frame translation
  - continue execution (where the TLB miss occurred)

Inverted page tables (1)

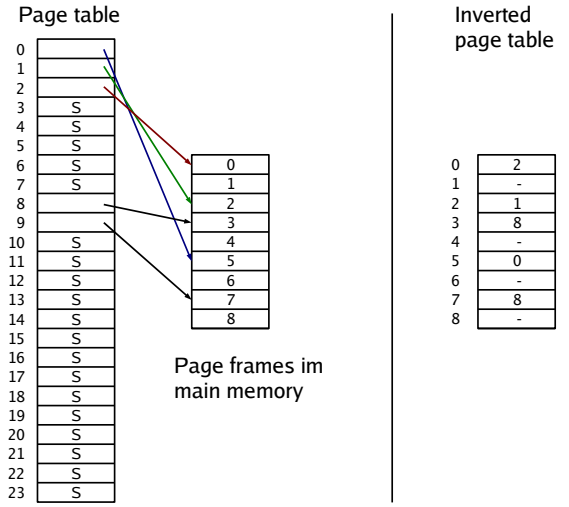
- When there is huge virtual memory, a lot of page table entries are needed,
  e.g. 2^{32} Byte address space, 4 KByte/page
  -> >1 million page entries,
  so: page table size is > 4 MByte (per process!)
- Save space using inverted page tables:
  - normal: one entry per (virtual) page with reference to page frame (in main memory)
  - inverted: one entry per page frame with reference to tupel (process ID, virtual page)

Inverted page tables (2)

- Problem: Given process *p* and its page *n* find the entry (*p*,*n*) in the inverted table -> lots of lookups
- Use TLB here as well, in order to find „most used“ pages quickly
- In case of TLB miss there is no way out: search

- Different solution for problem of huge page tables: Multi-layer paging

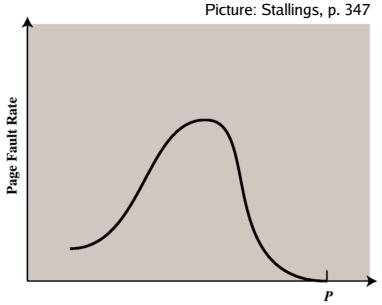
Inverted page tables (3)



Effects of page size

- Internal fragmentation: the smaller the pages, the less fragmentation
- Small pages -> big tables
  possible swapping of parts of the table
  -> double Page Faults when accessing a page, whose table entry is swapped out
- Locality principle:
  small pages: local, few faults.
  bigger pages, poor use of locality
  no more faults as page size approaches size *P* of process memory

Picture: Stallings, p. 347



(a) Page Size

Multi-Layer Paging (1)

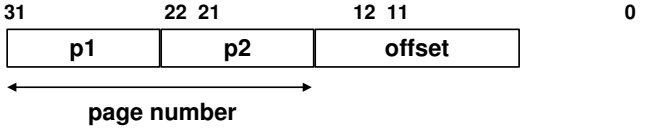
Page table can become quite large.

Example: - 32 bit addresses,
- 4 KByte page size,
- 4 Byte per entry

Page table:
>1 million entries,
4 MByte table size
(per process!)

Multi-Layer Paging (2)

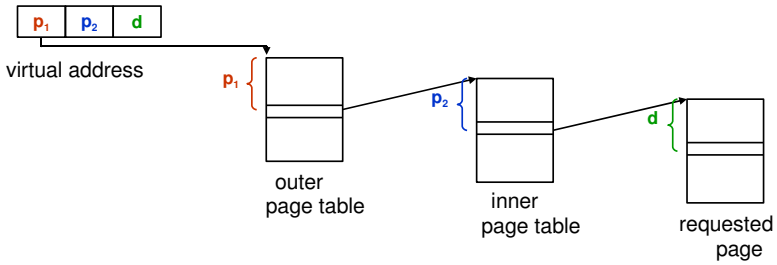
- Double-layer Paging:
  - partition page number further, e.g.:



  - p_1: Index into outer page table, each of whose entries points to an inner page table zeigen
  - p_2: Index into one of the inner page tables, whose entries point to page frames in main memory
  - The inner page tables need not be memory-resident.
- Similar: implement three-layer paging etc.

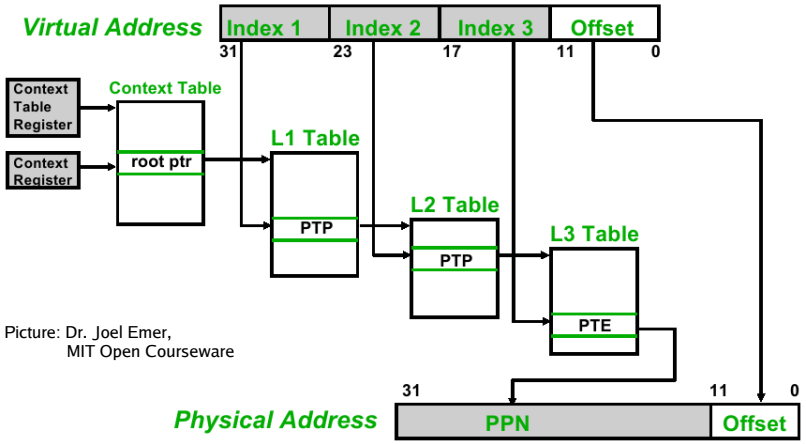
Address translation in Multi-Layer Paging (1)

Double-layer Paging:



Address translation in Multi-Layer Paging (2)

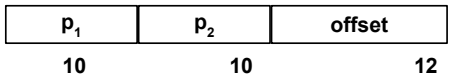
Three-layer Paging with SPARC CPUs:



Multi-Layer Paging (4)

- Size of page tables:

  Example:



  - Outer page table has 1024 entries, pointing to (potentially) 1024 inner page tables, each of which holds 1024 entries.
  - With a length of 4 Byte per page table entry each page table has the size of a 4 KByte page.
  - System uses only as many inner tables as is necessary.

Multi-Layer Paging (5)

- For each address translation even further memory accesses are needed, thus using TLBs is even more important.
- The key for the TLB is the combination of all parts of the page number: (p_1,p_2,...).

Memory Protection with Paging (1)

- Protection against access by other processes:
  - Since every process has its own page table, accessing memory areas of other processes is impossible. (On the other hand, this makes implementation of shared memory harder.)
- Protection against (e.g.) illegal write access:
  - Page table entries contain an additional protection code, which declares whether a page can be read / written etc. (This may also depend on whether access is attempted from user or kernel mode.)

Memory Protection with Paging (2)

- Page allocation is transparent for the programmer !
- Creation of protection codes by compiler and / or linker:
  - Program is partitioned into sections whose sizes are multiples of the page size.
  - For every section a protection code (for all pages of this section) is generated and written to the head of the program (binary) file.
  - Program loader sets the protection codes in the page table entries.

Page Sharing with Paging (1)

- *In theory* entries of different page tables could point to the same page frame.

  Problems:
  - How to find out, whether a page is already used by a different process and in which page frame it is located?
  - When changes happen (e.g. of the assigned page frame) many page tables would need to be updated.

Page Sharing with Paging (2)

- *In practice* the parts of memory that shall be shared between several processes are
  - either implemented as a shared segment with its own page table (combination of segmentation and paging, e.g. Unix) or
  - there is a pseudo process address space for which the operating system keeps a separate (global) page table (e.g. Windows).