DM510: Main Memory

Lars Rohwedder

Disclaimer

These slides contain (modified) content and media from the official Operating System Concepts slides: https://www.os-book.com/OS10/slide-dir/index.html

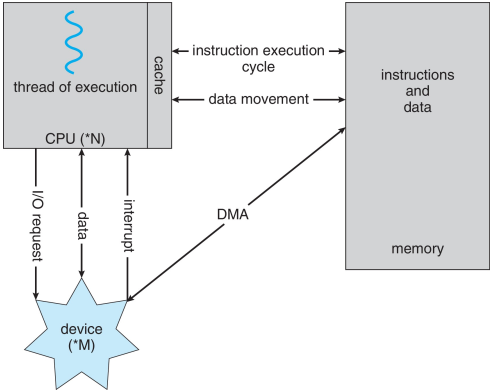
Today’s lecture

- Chapter 9 of course book

Overview

Role of main memory

- Apart from registers, only storage that CPU can directly access

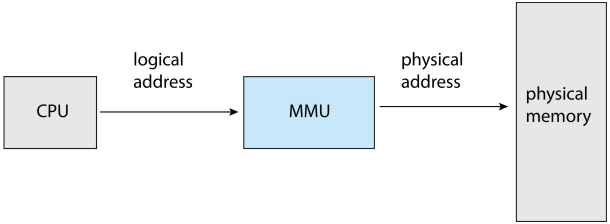


Accessing main memory

- Load or store instruction, for address and possibly data
- Slow compared to instructions only on registers. **Memory stall**: CPU needs to wait for memory access before continuing
- Recently used memory addresses in **cache** for faster access

Logical (virtual) and physical addresses

- CPU’s instructions load and store to **logical** (also known as **virtual**) addresses. They are different from **physical** addresses that memory unit sees
- **Memory-management unit** translates (in hardware) logical to physical addresses
- Many variants of logical-to-physical translation possible



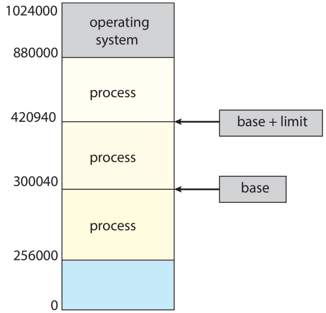
Purpose of logical addresses

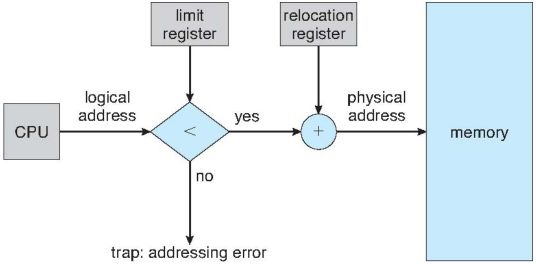
- Protect memory of processes from each other
- Great flexibility for allocating physical memory to processes

Naive Approach: Contiguous Allocation

Contiguous allocation

- Each process receives contiguous section of physical memory addresses
- Before executing user code, kernel sets the following registers (access priviledged):
  relocation register: first physical address (base) for process
  limit register: length of section

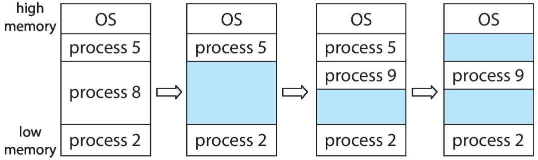




- phyical address = relocation register + logical address

Allocation details

- Need for **variable size** partition of memory: cannot afford to give every process the same (maximum) amount of memory
- Since new processes start and terminate, **holes** of free memory occur



- Which section of memory to allocate to new process?
  - **First-fit:** First hole that is big enough
  - **Best-fit:** Smallest hole that is big enough
  - **Worst-fit:** Biggest hole that is big enough
- Empirically, First-fit and best-fit perform better than worst-fit

Fragmentation

- **External fragmentation:** enough free space, but not contiguous
- **Internal fragmentation:** more space allocated to processes than requested (e.g., rounded up to power of 2)
- Rule of thumb (**50% rule**): for N blocks allocated, $0.5N$ blocks are lost due to fragmentation
- **Compaction:** shuffle around memory to make free memory contiguous (typically slow)

Paging

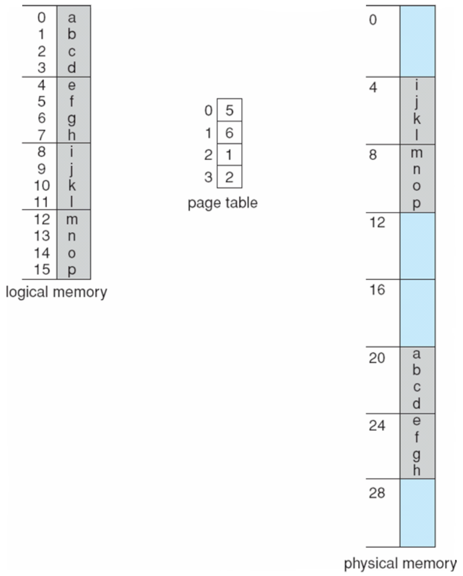
Pages and frames

- Logical memory is split into **pages** of specific size, e.g. 4MB
- Physical memory is split into **frames** of same size as pages
- Page table maps pages to frames
- No external fragmentation

page size

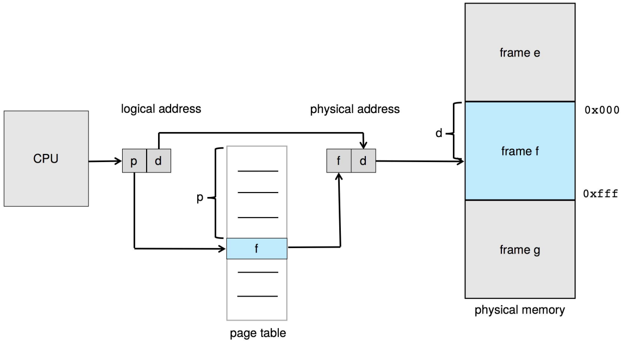
Can only allocate multiples of page size to processes;

- large page size ⇒ high internal fragmentation
- small page size ⇒ large page table



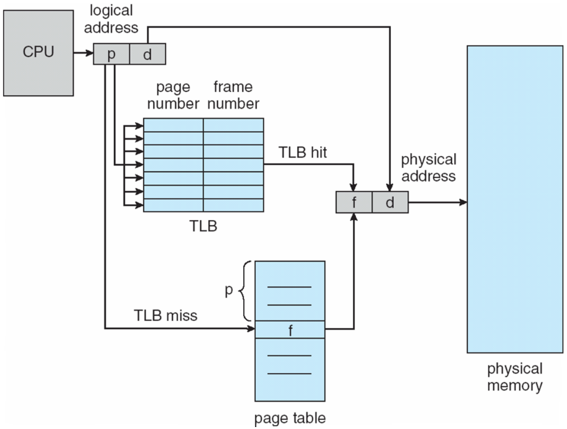
Address translation

- High order bits: page/frame number
- Low order bits: offset within page/frame
- One page table for each process stored in memory $\Rightarrow$ two memory accesses per memory instruction



Translation look-aside buffer

- Small lookup table in hardware (TLB) stores recently used page numbers and their corresponding frame numbers



- Page in TLB: very fast access
- Page not in TLB: need to lookup table in main memory (slow), add page/frame combination to TLB
- Size of TLB is limited

Effective access time (TLB)

How long does a logical memory access take on average? (effective access time)

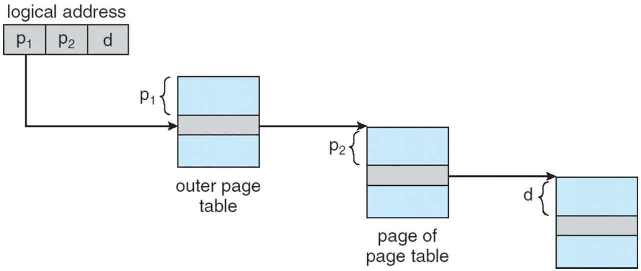
- Suppose we need 10 nanoseconds for physical memory access
- Further, in 80% of the accesses we find page in TLB (**hit ratio**)
- Thus, in 20% we need a second memory access
- EAT $= 0.8 \cdot 10 + 0.2 \cdot (10 + 10) = 12$ nanoseconds
- If hit ratio was 99%, then EAT $= 0.99 \cdot 10 + 0.01 \cdot (10 + 10) = 10.1$ nanoseconds
  $\Rightarrow$ 1% slowdown

Hierarchical page table

Motivation

If address space is very large (e.g. 64bit), then naive approach leads to either too large page sizes or too large page table

- Use first bit section index outer page table, which contains pointer to inner page table indexed by second bit section
- Tradeoff: The more nested tables the more physical memory accesses per logical memory access

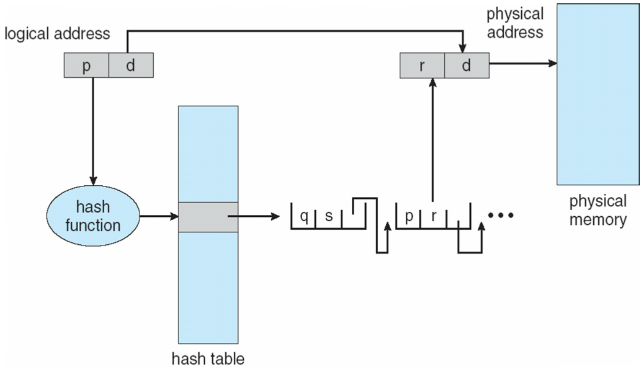


Hashed page table

Motivation

If address space is very large (e.g. 64bit), then naive approach leads to either too large page sizes or too large page table

- Use hash table to map logical to physical addresses

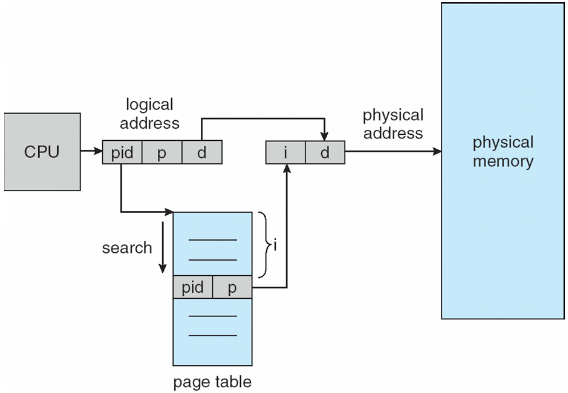


Inverted page table

Motivation

If address space is very large (e.g. 64bit), then naive approach leads to either too large page sizes or too large page table

- Observation: number of frames often much lower than addressable pages
- Use table with one entry per frame
- Disadvantage: requires searching through table, since we cannot index it based on logical address



Midterm evaluation

https://etc.ch/arjN