

Memory Management

Management of a **limited resource**:

(Memory hunger of applications increases with capacity!)

⇒ **Sophisticated algorithms needed**, together with support from HW and from compiler and loader.

Key point: program's view memory is set of memory cells starting at address 0x0 and finishing at some value (**logical address**)

Hardware: have set of memory cells starting at address 0x0 and finishing at some value (**physical address**)

Want to be able to store memory of several programs in main memory at the same time

Swapping

If **memory demand is too high**, memory of some processes is **transferred to disk**

Usually **combined with scheduling**: low priority processes are swapped out

Problems:

- **Big transfer time**
- What to do with **pending I/O?**

First point reason why **swapping is not principal memory management technique**

exception: systems without hardware support by memory management unit (some embedded systems)

⇒ **user decides** which process is swapped out

possible only at **few pre-defined moments**

⇒ **multi-processing severely limited**

need suitable **mapping from logical addresses to physical addresses**:

- at **compile time**: **absolute references** are generated (eg MS-DOS .com-files)
- at **load time**: can be done by **special program**
- at **execution time**: **needs HW support**

Address mapping can be taken one step further:

dynamic linking: use only **one copy of system library**

⇒ OS has to help: same code accessible to more than one process

Fragmentation

Swapping raises two problems:

- over time, many **small holes** appear in memory (**external fragmentation**)
- programs only a little smaller than hole ⇒ **leftover too small to qualify as hole** (**internal fragmentation**)

Strategies for choosing holes:

- **First-fit**: Start from beginning and use first available hole
- **Rotating first fit**: start after last assigned part of memory
- **Best fit**: find smallest usable space
- **Buddy system**: Free holes are administered according to tree structure; smallest possible chunk used

Paging

Alternative approach: Assign **memory of a fixed size (page)**

⇒ avoids **external fragmentation**

Translation of logical address to physical address **done via page table**

Hardware support mandatory for paging:

If page table **small**, use **fast registers**

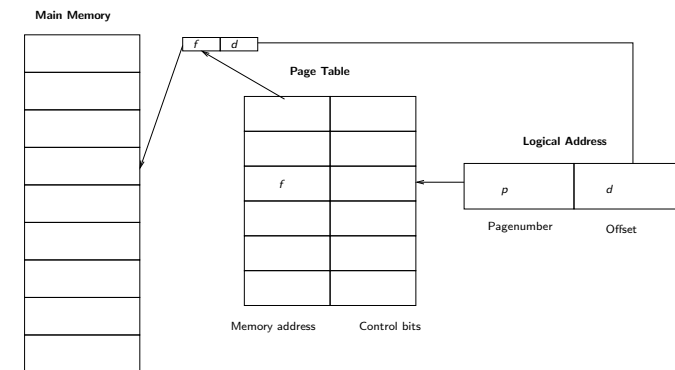
Store large page tables in main memory, but **cache most recently used entries**

Instance of a general principle:

Whenever **large lookup** tables are required, use **cache (small but fast storage)** to store most recently used entries

Memory protection easily added to paging:

protection information **stored in page table**



Segmentation

Idea: **Divide memory according to its usage** by programs:

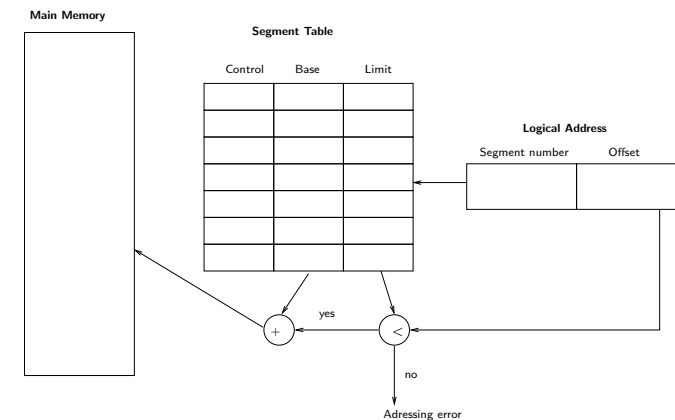
- **Data**: **mutable**, **different** for each instance
- **Program Code**: **immutable**, **same** for each instance
- **Symbol Table**: **immutable**, **same** for each instance, **only necessary for debugging**

Requires again HW support

can use same **principle as for paging**, but have to do overflow check

Paging motivated by ease of allocation, segmentation by use of memory

⇒ **combination of both works well** (eg 80386)



Virtual memory

Idea: complete **separation of logical and physical memory**

⇒ Program can have extremely large amount of virtual memory

Generalisation of paging and segmentation

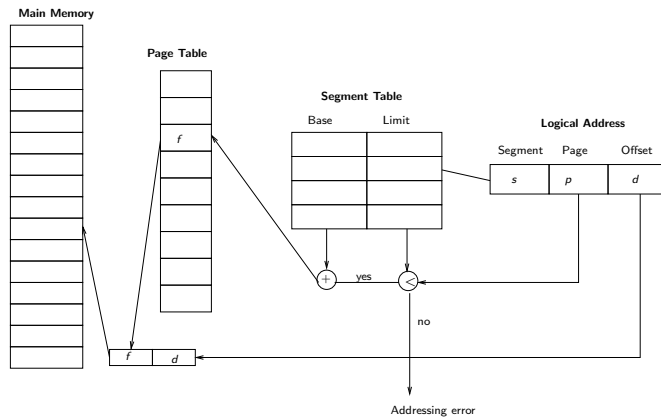
works because most programs use only **small fraction of memory intensively**.

Efficient implementation tricky

Reason: Enormous difference between

- memory access speed (ca. 60ns)
- disk access speed (ca. 6ms)

Factor 100,000 !!



Demand Paging

Virtual memory implemented as demand paging:

memory divided into **units of same length (pages)**, together with valid/invalid bit

Two strategic decisions to be made:

- Which process to **"swap out"** (move whole memory to disk and block process): done by swapper
- **which pages to move to disk** when additional page is required: done by pager

Minimisation of rate of page faults (page has to be fetched from memory) **crucial**

If we want 10% slowdown due to page fault, require fault rate $p < 10^{-6}!!$

Page replacement algorithms

1.) **FIFO:**

easy to implement, but **does not take locality into account**

Further problem: Increase in number of frames can cause increase in number of page faults (Belady's anomaly)

2.) **Optimal algorithm:**

select page which will be re-used at the latest time (or not at all)

⇒ **not implementable**, but **good for comparisons**

3.) **Least-recently used:**
use past as guide for future and replace page which has been unused for the longest time

Problem: **Requires a lot of HW support**

Possibilities:

-Stack in microcode

-Approximation using reference bit: HW sets bit to 1 when page is referenced.

Now use FIFO algorithm, but skip pages with reference bit 1, resetting it to 0

⇒ **Second-chance algorithm**

2.) **Page-Fault Frequency:**

takes direct approach:

- give process additional frames if page frequency rate high
- remove frame from process if page fault rate low

Thrashing

If process lacks frames it uses constantly, page-fault rate very high.

⇒ CPU-throughput decreases dramatically.

⇒ Disastrous effect on performance.

Two solutions:

1.) **Working-set model** (based on locality):

Define working set as set of pages used in the most recent Δ page references

keep only **working set in main memory**

⇒ Achieves high CPU-utilisation and prevents thrashing

Difficulty: **Determine the working set!**

Approximation: use reference bits; copy them each 10,000

references and define working set as pages with reference bit set.

Memory Management in the Linux Kernel

Have only four segments in total:

- Kernel Code
- Kernel Data
- User Code
- User Data

Paging used as described earlier

Have elaborate permission system for pages

Kernel memory and user memory

Memory usage in linux influenced by HW limitations of earlier architecture

Result: When using 32 bit architecture, division of logical address space in

- 3GB for user processes (so-called high memory)
- 1GB for kernel (so-called low memory)

If kernel requires more memory, pages in high memory may be temporarily mapped into low memory

Division not present in 64 bit architecture

Page caches

Experience shows: have repeated cycles of allocation and freeing same kind of objects (eg inodes, dentries)

can have pool of pages used as cache for these objects (so-called slab cache)

cache maintained by application (eg file system)

`kmalloc` uses slab caches for commonly used sizes