Virtual memory algorithms and examples

Stallings, Chapter 8

Design decisions

• Virtual memory or not?
• Paging or segmentation?
• What algorithms?

CONSIDERING ONLY PAGING:

• What page size?
**Page size**

- Reduce internal fragmentation → small
- Reduce page table size → large
- Proportional (1x, 2x, ...) to disk block size
- Optimal value is different for different programs
- Increase TLB hit ratio → large
- Different page size for different applications?
- How does MMU know the page size?

Tbl 8.2 [Stal05]

**Page fault rate**

- More small pages to the same memory space
- References from large pages more probable to go to a page not yet in memory
- References from small pages often to other pages -> often used pages selected to the memory
- Page size closer to process size
- Too small working set size -> larger page fault rate
- Large working set size -> nearly all pages in the memory
Algorithms for memory management

- Fetch policy
  - Demand paging vs prepaging
- Placement policy
- Replacement policy
  - Resident set management, selecting the page to be replaced
  - Methods: OPT, LRU, FIFO, random …
- Cleaning policy
  - Demand cleaning vs precleaning, buffering
- Load control
- Working set size

Resident set mgt

- Number of page frames allocated for a process (resident set size)
  - Fixed: how large, identical for all?
  - Variable: based on what? Working set?
- Replacement policy:
  - Global: over all pages (all processes)
  - Local: just the pages of this process
- Size of the resident set:
  - Too small: higher multiprocessing level, more page faults, trashing
  - Too large: smaller CPU utilisation, wasted memory
### Resident set management

#### Fixed Allocation
- Number of frames allocated to process is fixed.
- Page to be replaced is chosen from among the frames allocated to that process.

#### Variable Allocation
- The number of frames allocated to a process may be changed from time to time, to maintain the working set of the process. (e.g., PFF)
- Page to be replaced is chosen from among the frames allocated to that process.

---

### Replacement policy

- **OPT** – optimal algorithm
  - Nothing is better than this, unrealistic to be implemented,
  - Clairvoyant, knows the future references and choses based on them

- **LRU**, Least recently used
  - High amount of bookkeeping
  - Modification: NRU – Not recently used, easier to implement
    - Only maintain information about recent usage (for example on used bit)

- **FIFO**, First-in-first-out
  - Easy to implement, but not very useful
  - Modification: second chance – do not replace page that was just recently used

- **Clock**

  Metrics? Number of page faults or page fault rate
Clock

- Replace the page where the use bit is zero, while scanning reset the use bit
- MMU sets the use bit to 1 every time a memory reference to that page is made

Figure 8.16 Example of Clock Policy Operation
### Working set

- Set of pages used during last k references (window size)
- Pages not in the working set are candidates to be released or replaced
- Suitable size for the working set? Estimate based on page fault rate

\[
1 \leq |W(t, \Delta)| \leq \min(\Delta, n)
\]
Dynamically adjusting resident set

- PFF – Page Fault Frequency
- VSWS – Variable-interval Sampled Working Set

Swap area, VM backup store

(Fig 4-33 [Tane01])

Fixed allocation –
For the whole process
in advance

Dynamic allocation –
only for the pages that
are swapped out
UNIX / Solaris (+4BSD)
MUISTINHALLINTA

UNIX/Solaris: data structures
Page table - one for each process
- one entry for each logical page

Location in the main memory?

(a) Page table entry

Page frame number | Age | Copy on write | Mod-Ref | Valid | Protect
--- | --- | --- | --- | --- | ---
P & Q: X = 5

P:
Pages of P

Q:
Pages of Q

write

P:
Pages of P

Q:
P's page X = 7
Q's page X = 5

Syksy 2007, Tiina Niklander
**Käyttöjärjestelmät, Luento 9**

---

**UNIX/4BSD: (core map)**

![UNIX/4BSD: (core map)](image1)

**Kaksiviisarinen Clock**

![Kaksiviisarinen Clock](image2)

---

**Syksy 2007, Tiina Niklander**

---

**Kaksi-kiirusoja:**

- **Fronthand:**
  - Set Reference = 0
- **Backhand:**
  - if (Reference == 0)
  - page out

Pages not used during the sweep are assumed not to be in the working sets of the processes.

Suits better for large memories.

Speed of the hands (frame/sec)? Scanrate

Gap between the hands?

Handspread

---

*(Fig 10-16 [Tane01])*

*(Fig 8.23 [Stal05])*
UNIX/Solaris: Kernel memory allocation

- Based on paging
- Locking pages to memory
- Kernel allocates and deallocates all the time (large number of) small areas and buffers
  - For example: proc, vnode, file descriptor
- Kernel need to allocate and deallocate fragments of pages (not only full pages)
  - Dynamic partitioning for allocations within one page
  - => Lazy Buddy System
  - Delay combining until there are 'too many' fragments of certain size

Linux

MUISTINHALLINTA

Ch 8.4 [Stal 05]
**Linux: memory management**

- 32-bit architectures
  - 3 GB reserved for user processes
  - 1 GB for kernel (including the page tables)
    - at the beginning of the physical memory
  - Kernel mode allowed to make references to the full 4 GB
- Kernel's memory allocation based on slabs
  - Only for the kernel, efficient allocation of small memory areas, avoids allocating full pages

**Linux: paging**

- Architecture independent
  - Alpha: 64b addresses, full support for 3 levels,
    - Page size 8KB, offset 13 bits
  - x86: 32b address, only 2-levels,
    - Page size 4KB, offset 12 bits
- 3-level page table
  - page directory (PD), 1 page
  - page middle directory (PMD), multiple pages
  - page table (PT), multiple pages
- Page directory always in memory
  - This the page table address given to MMU at process switch
  - Other pages can be on disk

(Fig 8.25 [Stal05])
**Linux: address translation**

- Split address to 4 elements
  - index to page directory (PD-offset)
  - index to page middle directory (PMD-offset)
  - index to page table (PT-offset)
  - offset

\[
\begin{align*}
\text{PMD} & \_\text{base} = \text{PD} \_\text{base} + \text{PD}[\text{PD}\_\text{offset}] \\
\text{PT} & \_\text{base} = \text{PMD} \_\text{base} + \text{PMD}[\text{PMD}\_\text{offset}] \\
\text{Page} & \_\text{base} = \text{PT} \_\text{base} + \text{PT}[\text{PT}\_\text{offset}] \\
\text{fyys.os} & = \text{Page}\_\text{base} + \text{offset}
\end{align*}
\]

- X86-trick
  - Page middle directory (PMD) reduced to one item, which compiler optimizes away.
  - MMU assumes that the page directory refers directly to the page table (PT)
  - Suitable to all architectures that support only two-level paging

---

**Linux: placement (allocation)**

- Reserves a consecutive area (region) for consecutive pages
  - More efficient fetching and storing with disks
  - Optimize the disk utilisation, with the cost of memory allocation

- Buddy System: 1, 2, 4, 8, 16 or 32 page frames
  - Allocates pages in large groups, increases internal fragmentation
  - Implementation: table with list of different sizes unallocated page regions
**Linux: replacement**

- Globaali Clock using M-bit + sort of LRU
  - Counter of dynamically allocated page frames
- 8 bit counter of age
  - MMU increases the counter at each page reference
  - Background process (kswapd) reduces the counter each second
    - if too few page frames free, deallocated unused frames
- Uses **pagebuffer**
  - To store deallocated pages for a short duration

---

**Diagram:**

- active_list: age > 0, in use
- inactive_dirty_list: age = 0, not used, clean or dirty
- inactive_clean_list: age = 0, not used, clean

It does not seem possible to understand the VM system in a modular way. The interfaces between, say, the zone allocator and the swap policy are many and varied... What a nightmare” (Joe Knapka)
**Linux: kernel memory allocation**

- Kernel memory locked to the memory
- Dynamic allocation based on pages
- Dynamically loadable modules fully to the kernel area
  - buddy system on the pages
- Additional need for small, short term memory allocations -> use slab
  - One page 'cut' to smaller areas (slabs)
  - buddy system within each page
    - x86: 32, 64, 128, 252, 508, 2040, 4080 tavua (page size 4KB)

**Slab allocation**

Fig 9.27 – Silberschatz, Galvin & Gagne: Operating system concepts with Java
**Windows 2000**

**MUISTINHALLINTA**

Ch 8.5 [Stal 05]
Ch 11.5 [Tane 01]

---

W2K

- 32-bit addresses
  => 4 GB virt. os.
- page size 4K - 64K
  - Pentium: 4KB
  - Alpha: 8K

(IFG 8.26 [Stal05])

"0x0000????:" 64-Kbyte region for NULL-pointer assignments (inaccessible)

"-1" 64-Kbyte region for bad pointer assignments (inaccessible)

"0xFFFF????:" 2-Gbyte region for the operating system (inaccessible)

2-Gbyte user address space (unreserved, usable)

(Fig 11.23 [Tane 01])

"0xFFFF????:"
W2K: Paging

- No segmentation
- Variable resident set size per process
  - Start with a fixed set of frames
  - If large number of free frames, process can get more frames
    - A greedy one is not allowed to allocate last 512 frames
  - If the free memory reduces, the resident set of processes can be decreased
- Demand fetch (no prefetching)
- Disk I/O using larger blocks
  - 1-8 pages at each time

W2K: paging

(Fig 11.26 [Tane01])

Bits 20

Page frame

G: Page is global to all processes
I: Large (4-MB) page
D: Page is dirty
A: Page has been accessed
C: Caching enabled/disabled

Wt: Write through (no caching)
U: Page is accessible in user mode
W: Writing to the page permitted
V: Valid page table entry

Pentium

Fig 11.24 [Tane01]
(seur. kalvo)
W2K address space

[W2K address space diagram]

Tane01

Fig. 11-24. Mapped regions with their shadow pages on disk. The lib.dll file is mapped into two address spaces at the same time.

W2K: replacement

- Balance set manager - daemon
  - Each second 1 sec check if enough free frames

- Working set manager
  - Running when needed
  - Dynamic working set size
  - Released page frames of processes
  - Starts with large idle processes

- Swapper-daemon
  - every 4 sec: search for processes, whose threads (all of them) have been passive for long duration

- Parts of the OS and buffer pool locked to memory
- Caching of released pages
W2K: page frames

![Diagram of page frames]

*Fig 11.27 [Tane01]*

"The code is so complex that the designers loathe to touch parts of it for fear of breaking something somewhere"

Andrew Tanenbaum [Tane01, p. 823]