Virtual memory
Operations and policies

Chapters 3.4. – 3.6

Lecture 4: Thu 16.9.2010 Tiina Niklander
Policies and methods

- Fetch policy (N outopolitiikka)
  - 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 (Käyttöjoukkoe)
  - 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
    - Some time mistakes happen

See Fig 8.1 [Stal05]
Page Replacement Algorithms

- Page fault forces choice
  - which page must be removed
  - make room for incoming page

- Modified page must first be saved
  - unmodified just overwritten

- Better not to choose an often used page
  - will probably need to be brought back in soon
Page address stream

<table>
<thead>
<tr>
<th></th>
<th>2</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>5</th>
<th>2</th>
<th>4</th>
<th>5</th>
<th>3</th>
<th>2</th>
<th>5</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>OPT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>5</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>LRU</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>5</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>FIFO</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>CLOCK</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>5</td>
<td>F</td>
<td>F</td>
</tr>
</tbody>
</table>

F = page fault occurring after the frame allocation is initially filled

Figure 8.15 Behavior of Four Page-Replacement Algorithms
Optimal Page Replacement Algorithm

- Replace page needed at the farthest point in future
  - Optimal but unrealizable
  - Clairvoyant (not possible!)
  - can be found afterwards from trace
- Used as a reference point for other algorithms
  - Nothing is better that optimal!
- Estimate by …
  - logging page use on previous runs of process
  - although this is impractical
FIFO (First-In First-Out) Page Replacement Algorithm

- Replace the page that has been in the memory longest time
- Page at beginning of list replaced
- Maintain a linked list of all pages
  - In order they came into memory
  - Alternatively: store the load time on each page
- Disadvantage
  - Page in memory the longest may be often used and should not be evicted
Second Chance

- Pages sorted in FIFO order
- Use reference bit R and modified bit M
- If the page to be evicted has been recently used (R=1) give it second chance; move to end of queue and set R=0
  - Will be evicted, if all other pages are also given second chance

(Fig 4-16 [Tane01])
Not Recently Used
Page Replacement Algorithm

- Each page has Reference bit, Modified bit
  - bits are set when page is referenced, modified
  - R cleared periodically, M cleared at disk writes
- Pages are classified
  1. not referenced, not modified
  2. not referenced, modified
  3. referenced, not modified
  4. referenced, modified
- NRU removes page at random from lowest numbered non-empty class
The Clock Page Replacement Alg.

- Go through all pages in circular fashion
- Try to locate unused page with the NRU classification, used page gets second chance

When a page fault occurs, the page the hand is pointing to is inspected. The action taken depends on the R bit:
- R = 0: Evict the page
- R = 1: Clear R and advance hand
Least Recently Used (LRU)

- Recently used pages assumed to be used again soon
- Evict page that has been unused for longest time
  - How to find?
- Heavy bookkeeping
  - Update each reference time to the pages
  - Maintain pages in the referenced order (linked list)
- Infeasible if followed strict
**Least Recently Used (LRU)**

- **Strict alternatives:**
  - Keep a linked list of pages in reference order, update this list on every memory reference !!
  - Keep counter in each page table entry
    - Use *special hardware* to increment the counter
    - choose page with lowest value counter
    - periodically zero the counter

- **Simulating algorithms: NFU, Aging**
  - NFU - Not Frequently used
  - Aging
    - In the adding: shift counter right, add R bit to leftmost
    - Forgets the history
LRU hardware using matrix

On reference: set row all 1s and column all 0s

At any time the row with lowest binary value is LFU page
### Aging algorithm simulates LRU

On each clock time process the counters:
- Shift right one bit
- Set left-most bit to one on pages referenced after previous tick

<table>
<thead>
<tr>
<th>Page</th>
<th>(a)</th>
<th>(b)</th>
<th>(c)</th>
<th>(d)</th>
<th>(e)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>10000000</td>
<td>11000000</td>
<td>11100000</td>
<td>11110000</td>
<td>01111000</td>
</tr>
<tr>
<td>1</td>
<td>00000000</td>
<td>10000000</td>
<td>11000000</td>
<td>01100000</td>
<td>10110000</td>
</tr>
<tr>
<td>2</td>
<td>10000000</td>
<td>01000000</td>
<td>00100000</td>
<td>00100000</td>
<td>10001000</td>
</tr>
<tr>
<td>3</td>
<td>00000000</td>
<td>00000000</td>
<td>10000000</td>
<td>01000000</td>
<td>00100000</td>
</tr>
<tr>
<td>4</td>
<td>10000000</td>
<td>11000000</td>
<td>01100000</td>
<td>10110000</td>
<td>01011000</td>
</tr>
<tr>
<td>5</td>
<td>10000000</td>
<td>01000000</td>
<td>10100000</td>
<td>01010000</td>
<td>00101000</td>
</tr>
</tbody>
</table>
Working set model
Resident Set vs. Working Set

- **Working set** = Set of pages the process is currently using
  - $w(k,t)$ is the size of the working set at time, $t$

- **Resident set** = Page frames currently allocated for process

- **Number of allocated page frames** (sivukehysten lkm)
  - Too small
    - more processes in the memory, more page faults
      - Too many? Trashing? (ruuhkautuminen)
  - Too large
    - occupy space that beneficial for other processes
    - CPU utilisation (käyttöaste) too low
    - "wasted space"
**Working Set Strategy**

- Pages used by the most recent $k$ memory references (window size)
- Common approximation $W(t, \Delta)$
  - Pages used during the recent $t$ msec of current virtual time, time that the process has been actually executing
- Pages not in the working set, can be evicted
- For page replacement:
  - Hardware set the R and M bits
  - Each page entry has ‘time of last use’
The Working Set Algorithm

In case all pages in working set, evicts the oldest unreferenced and/or unmodified page
The WSClock
Page Replacement Algorithm

- As clock, but only residence set
- Evict unmodified page not in working set and
- Schedule for disk write modified page not in WS
- If no candidate page found during one round, wait for writes or evict any clean (=unmodified) WS page
## Review of Page Replacement Algorithms

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optimal</td>
<td>Not implementable, but useful as a benchmark</td>
</tr>
<tr>
<td>NRU (Not Recently Used)</td>
<td>Very crude</td>
</tr>
<tr>
<td>FIFO (First-In, First-Out)</td>
<td>Might throw out important pages</td>
</tr>
<tr>
<td>Second chance</td>
<td>Big improvement over FIFO</td>
</tr>
<tr>
<td>Clock</td>
<td>Realistic</td>
</tr>
<tr>
<td>LRU (Least Recently Used)</td>
<td>Excellent, but difficult to implement exactly</td>
</tr>
<tr>
<td>NFU (Not Frequently Used)</td>
<td>Fairly crude approximation to LRU</td>
</tr>
<tr>
<td>Aging</td>
<td>Efficient algorithm that approximates LRU well</td>
</tr>
<tr>
<td>Working set</td>
<td>Somewhat expensive to implement</td>
</tr>
<tr>
<td>WSClock</td>
<td>Good efficient algorithm</td>
</tr>
</tbody>
</table>
Cleaning Policy

- **Need for a background process, paging daemon**
  - periodically inspects state of memory

- **When too few frames are free**
  - selects pages to evict using a replacement algorithm

- **It can use same circular list (clock)**
  - as regular page replacement algorithm but with different pointer
When to clean the page? (write to disk)

- Only when changed
- Demand cleaning (tarvetalletus)
  - Write to disk only when the page frame is needed
  - Long delay in freeing, crash?
- Precleaning (ennaltatalletus)
  - Write to disk periodically in bigger groups
- Cleaning the freed page frame
  - Zeroing might be wasted work
  - What if the page is needed soon again?
    - unmodified free page might be reused
- What if the disk utilization is low (no work on disk)?
Page buffering

- Maintain certain number of free page frames
- ~ page frame cache (of freed pages frames)
  - Fast recovery from page frame buffer, if the page on freed frame is referenced
- Page marked to be cleaned
  - Add to free page frame list, if not modified
  - Add to write-to-disk list, if modified
- Page still on the same allocated page frame
  - Remove the reference from page table
- Allocate page frames from the free frames list using FIFO order
UNIX: Block cache

- Page buffers, block caches - the content in separate area
- The data structure, hash table and linked list, only has management information and link to the actual location
- Hash key: device#, block#
- Linux has just one buffer for process pages and file blocks
Separation of Policy and Mechanism

Advantages:
- Way to manage complexity
- More modular code

Disadvantages:
- Split between user and kernel spaces
- Extra overhead
- More mode switches
- More message passing

- External pager in user mode
  - Page mapping, swap area
- Page fault handler in kernel
  - Ask for page from external pager
Design Issues
Page Size - Design Issues

- 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]
### Table 8.2 Example Page Sizes

<table>
<thead>
<tr>
<th>Computer</th>
<th>Page Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Atlas</td>
<td>512 48-bit words</td>
</tr>
<tr>
<td>Honeywell-Multics</td>
<td>1024 36-bit word</td>
</tr>
<tr>
<td>IBM 370/XA and 370/ESA</td>
<td>4 Kbytes</td>
</tr>
<tr>
<td>VAX family</td>
<td>512 bytes</td>
</tr>
<tr>
<td>IBM AS/400</td>
<td>512 bytes</td>
</tr>
<tr>
<td>DEC Alpha</td>
<td>8 Kbytes</td>
</tr>
<tr>
<td>MIPS</td>
<td>4 kbytes to 16 Mbytes</td>
</tr>
<tr>
<td>UltraSPARC</td>
<td>8 Kbytes to 4 Mbytes</td>
</tr>
<tr>
<td>Pentium</td>
<td>4 Kbytes or 4 Mbytes</td>
</tr>
<tr>
<td>PowerPc</td>
<td>4 Kbytes</td>
</tr>
<tr>
<td>Itanium</td>
<td>4 Kbytes to 256 Mbytes</td>
</tr>
</tbody>
</table>
Page Size

- Overhead due to page table and internal fragmentation

\[ \text{overhead} = \frac{s \cdot e}{p} + \frac{p}{2} \]

- Where
  - \( s \) = average process size in bytes
  - \( p \) = page size in bytes
  - \( e \) = page entry

Optimized when

\[ p = \sqrt{2se} \]

Example cases:

- \( s = 1 \) MB, \( e = 8B \) → \( p_{\text{opt}} = 4 \) KB
- \( s = 100 \) MB, \( e = 8B \) → \( p_{\text{opt}} = 40 \) KB
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

- Too small working set size -> larger page fault rate
- Large working set size -> nearly all pages in the memory
Working set size

- 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

\[ 1 \leq |W(t, \Delta)| \leq \min(\Delta, n) \]
Page Fault Rate

- Page fault rate as a function of the number of page frames assigned
- Page Fault Frequency algorithm (PFF):
  - Keep page fault rate between A and B for each process
  - Increase residence set size at A and reduce at B
Load Control

- Despite good designs, system may still thrash

- When Page Fault Frequency (PFF) algorithm indicates
  - some processes need more memory
  - but no processes need less

- Solution:
  Reduce number of processes competing for memory
  - swap one or more to disk, divide up pages they held
  - reconsider degree of multiprogramming
### Allocation and replacement

<table>
<thead>
<tr>
<th>Fixed Allocation</th>
<th><strong>Local Replacement</strong></th>
<th><strong>Global Replacement</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>• Number of frames allocated to process is fixed.</td>
<td>• Not possible.</td>
</tr>
<tr>
<td></td>
<td>• Page to be replaced is chosen from among the frames allocated to that process.</td>
<td></td>
</tr>
<tr>
<td><strong>Variable Allocation</strong></td>
<td>• The number of frames allocated to a process may be changed from time to time, to maintain the working set of the process.</td>
<td>• Page to be replaced is chosen from all available frames in main memory; this causes the size of the resident set of processes to vary.</td>
</tr>
<tr>
<td></td>
<td>• Page to be replaced is chosen from among the frames allocated to that process.</td>
<td></td>
</tr>
</tbody>
</table>

Tbl 8.4 [Stal05]
Local versus Global Allocation Policies

- (a) Original configuration
- (b) Local page replacement
- (c) Global page replacement
Shared pages and libraries

- Several processes use, just one copy in memory
  - Shared pages read-only
  - Same page frame pointed by multiple page tables
- Code must be position-independent
- Code must be re-entrant, does not change during execution
- A special case of memory-mapped files
Sharing libraries

- Old style: Link the library with program statically
- New style: Link the library at run time dynamically
- Windows: DLL – dynamic link library
- Save space
  - Just one copy in the memory
- Removing bugs
  - Just make one correction in the shared library
- Save recompilation time
  - Library change do not require new linking
Sharing: example editor

Logical address space of process $P_1$

page table for $P_1$

ed 1
3
4
ed 2
6
1
data 1

Logical address space of process $P_2$

page table for $P_2$

ed 1
3
4
ed 2
6
ed 3
7
data 2

Logical address space of process $P_3$

page table for $P_3$

ed 1
3
4
ed 2
6
ed 3
2
data 3
Sharing a memory-mapped file

- Mapped file is an alternative of I/O, file is accessed as a big character array in memory
- Can be used as shared memory between processes

Tan08 Fig 10-3
Swap area, VM backup store

(Heittovaihtoalue)

**Static**
- Reserved in advance
- Space for whole process
  - Copy the code/ text at start or
  - Reserve space, but swap gradually based on need
- PCB has swap location info

**Dynamic**
- Reserved space for each page when needed
- Page table has swap block number for the page
  - OR separate disk map to store page locations on swap
- No space reservation for pages that are never stored on swap

Swap location:
Windows: pagefile.sys, win386.swp
Linux: swap partition
(a) Paging to static swap area
(b) Backing up pages dynamically
Page Fault Handling - more details

1. Hardware traps to kernel
2. General registers saved
3. OS determines which virtual page needed
4. OS checks validity of address, seeks page frame
5. If selected frame is dirty, write it to disk
6. OS schedules disk operation to bring new page in from disk
7. Page tables updated
8. Faulting instruction backed up to when it began
9. Faulting process scheduled
10. Registers restored
11. Program continues
Operating System Involvement with Paging

- **Process creation**
  - determine program size
  - create page table

- **Process execution**
  - MMU reset for new process
  - TLB flushed

- **Page fault time**
  - determine virtual address causing fault
  - swap target page out, needed page in

- **Process termination time**
  - release page table, pages