Page replacement algorithms
When a page fault occurs

- OS has to choose a page to evict from memory
- If the page has been modified, the OS has to schedule a disk write of the page
- The page just read overwrites a page in memory (e.g. 4Kbytes)
- Clearly, it’s better not to pick a page at random
- Same problem applies to memory caches
Benchmarking

- Tests are done by generating page references (either from real code or random)
- Sequences of page numbers (no real address, no offset)
- Example:

```
7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1
```
Optimal page replacement

- At the moment of page fault:
  - Label each page in memory is labeled with the number of instructions that will be executed before that page is first referenced
  - Replace the page with the highest number: i.e. postpone as much as possible the next page fault
- Nice, optimal, but unrealizable
  - The OS can’t look into the future to know how long it’ll take to reference every page again
### Example: optimal

<table>
<thead>
<tr>
<th>Sequence</th>
<th>Phys mem</th>
<th>PF</th>
</tr>
</thead>
<tbody>
<tr>
<td>7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1</td>
<td>7 7 7 0 0 0 0 4 4 4 0 0 0 2 2 0 0 0 0 0</td>
<td>* * * * * *</td>
</tr>
<tr>
<td></td>
<td>0 0 1 1 2 2 2 2 2 2 2 2 2 2 0 0 1 1 1 1 1</td>
<td>[6 page faults]</td>
</tr>
<tr>
<td></td>
<td>1 2 2 3 3 3 3 3 3 3 3 3 3 1 1 2 2 7 7 7</td>
<td></td>
</tr>
</tbody>
</table>
Belady’s anomaly

Try this sequence

With 3 page frames

With 4 page frames

With FIFO, with the optimal algorithm, (later) with the LRU
“Not recently used” algorithm

- Use Referenced and Modified bits
- R&M are in hardware, potentially changed at each reference to memory
  - R&M are zero when process is started
- On clock interrupt the R bit is cleared
- On page fault, to decide which page to evict:
  - Classify:
    - Class 0 – R=0, M=0
    - Class 1 – R=0, M=1
    - Class 2 – R=1, M=0
    - Class 3 – R=1, M=1
  - Replace a page at random from the lowest class
FIFO replacement

- FIFO, first in first out for pages
- Clearly not particularly optimal
- It might end up removing a page that is still referenced since it only looks at the page’s age
- Rarely used in pure form...

![Diagram showing FIFO page replacement](image)
Example (FIFO)

<table>
<thead>
<tr>
<th>Sequence</th>
<th>Phys mem</th>
</tr>
</thead>
<tbody>
<tr>
<td>7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1</td>
<td></td>
</tr>
<tr>
<td>7 7 7 0 0 1 2 3 0 4 2 2 2 3 0 0 0 1 2 7</td>
<td></td>
</tr>
<tr>
<td>0 0 1 1 2 3 0 4 2 3 3 3 0 1 1 1 2 7 0</td>
<td></td>
</tr>
<tr>
<td>1 2 2 3 0 4 2 3 0 0 0 1 2 2 2 7 0 2</td>
<td></td>
</tr>
</tbody>
</table>

12 page faults
“Second chance” algorithm

• Like FIFO but...
• Before throwing out a page checks the R bit:
  ○ If 0 remove it
  ○ If 1 clear it and move the page to the end of the list (as it were just been loaded)
  ○ If all pages have R=1, eventually the algorithm degenerates to FIFO (why?)

OS 2009-10
Like “second chance” but...

...implemented differently:
- Check starting from the latest visited page
- More efficient: doesn’t have to move list’s entries all the time
Least recently used (LRU)

- Why: pages recently used tend to be used again soon (on average)
- List of all pages in memory (most recently in the head, least recently used in the tail)
- List update at each memory reference!
- List operations are expensive (e.g. find a page)
- So, difficult...
Least recently used (LRU)

- Idea! Get a counter, maybe a 64bit counter
- The counter is incremented after each instruction and stored into the page entry at each reference
- Store the value of the counter in each entry of the page table (last access time to the page)
- When is time to remove a page, find the lowest counter value (this is the LRU page)

- Nice & good but expensive: it requires dedicated hardware
Example LRU

<table>
<thead>
<tr>
<th>Sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Phys mem</th>
</tr>
</thead>
<tbody>
<tr>
<td>7 7 7 0 1 2 2 3 0 4 2 2 0 3 3 1 2 0 1 7</td>
</tr>
<tr>
<td>0 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0</td>
</tr>
<tr>
<td>1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PF</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
</tr>
</tbody>
</table>

9 page faults
NFU algorithm

- Since LRU is expensive
- NFU: “Not Frequently Used” algorithm
- At each clock interrupt add the R bit to a counter for each page: i.e. count how often a page is referenced
- Remove page with lowest counter value
- Unfortunately, this version tends not to forget anything
Aging (NFU + forgetting)

- Take NFU but...
- At each clock interrupt:
  - Right shift the counters (divide by 2)
  - Add the R bit to the left (MSB)
- As for NFU remove pages with lowest counter
- Note: this is different from LRU since the time granularity is a clock tick and not every memory reference!
# NFU+ageing

<table>
<thead>
<tr>
<th>Page</th>
<th>R bits for pages 0-5, clock tick 0</th>
<th>R bits for pages 0-5, clock tick 1</th>
<th>R bits for pages 0-5, clock tick 2</th>
<th>R bits for pages 0-5, clock tick 3</th>
<th>R bits for pages 0-5, clock tick 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0101011</td>
<td>1100110</td>
<td>1101010</td>
<td>1000100</td>
<td>0111000</td>
</tr>
<tr>
<td>1</td>
<td>1000000</td>
<td>1100000</td>
<td>1100000</td>
<td>1000000</td>
<td>1011000</td>
</tr>
<tr>
<td>2</td>
<td>1000000</td>
<td>0100000</td>
<td>0010000</td>
<td>0010000</td>
<td>1000100</td>
</tr>
<tr>
<td>3</td>
<td>0000000</td>
<td>0000000</td>
<td>1000000</td>
<td>0100000</td>
<td>0010000</td>
</tr>
<tr>
<td>4</td>
<td>1000000</td>
<td>1100000</td>
<td>0110000</td>
<td>1011000</td>
<td>0101100</td>
</tr>
<tr>
<td>5</td>
<td>1000000</td>
<td>0100000</td>
<td>1010000</td>
<td>0101000</td>
<td>0010100</td>
</tr>
</tbody>
</table>

(a) (b) (c) (d) (e)
Process’ behavior

- Locality of reference: most of the time the last $k$ references are within a finite set of pages < a large address space
- The set of pages a process is currently using is called the *working set* (WS) of the process
- Knowing the working set of processes we can do very sophisticated things (e.g. pre-paging)
- Pre-paging: load pages before letting the process to run so that the page-fault rate is low, also, if I know the WS when I swap the process then I can expect it to be the same in the future time when I reload the process in memory
Working set

Working set

k-most-recent memory references
WS based algorithm

- Store execution time information in the table entries (storing reference is too expensive!)
- At clock interrupt handle R bits as usual (clear them)
- At page fault, scan entries:
  - If R=1 just store current time in the entry
  - If R=0 compute “current-last time page was referenced” and if > threshold the page can be removed since it’s no longer in the working set (not used for threshold time)
- **Note:** we’re using time rather than actual memory references
e.g.: WS based algorithm

- Information about one page
- Time of last use
- Page referenced during this tick
- Page not referenced during this tick

Scan all pages examining R bit:
- if (R == 1)
  - set time of last use to current virtual time
- if (R == 0 and age > τ)
  - remove this page
- if (R == 0 and age ≤ τ)
  - remember the smallest time

Current virtual time: 2204
WSClock algorithm

- Use the circular structure (as seen earlier)
- At page fault examine the page pointed by the handle
- R=1, page in the WS – don’t remove it (set R to zero)
- R=0, M=0 no problem (as before, check age, page clean and decide depending on age)
- If M=1, schedule disk write appropriately to procrastinate as long as possible a process switch
  - If return to starting point, then one page will eventually be clean (maybe after a context switch)
  - Scheduling multiple disk write can be efficient in efficient systems (with disk scheduling and multiple disks)
  - No write is schedulable (R=1 always), just choose a clean page
  - No clean page, use the current page under the handle
WSClock

R=1

R=0

OS 2009-10
## Summary

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optimal</td>
<td>Not implementable, useful for benchmarking</td>
</tr>
<tr>
<td>NRU (Not recently used)</td>
<td>Very crude</td>
</tr>
<tr>
<td>FIFO</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 (better implementation)</td>
</tr>
<tr>
<td>LRU (Least Recently Used)</td>
<td>Excellent but difficult to implement</td>
</tr>
<tr>
<td>NFU</td>
<td>Crude approx to LRU</td>
</tr>
<tr>
<td>Aging</td>
<td>Efficient in approximating LRU</td>
</tr>
<tr>
<td>Working set</td>
<td>Expensive to implement</td>
</tr>
<tr>
<td>WSClock</td>
<td>Good and efficient</td>
</tr>
</tbody>
</table>
Design issues
Design issues

- Local vs. global allocation policy
  - When a page fault occurs, whose page should the OS evict?
  - Which process should get more or less pages?
    - Monitor the number of page faults for every process (PFF – page fault frequency)
    - For many page replacement algorithms, the more pages the less page faults
Page fault behavior

- Thrashing
- Optimal (fair to others)
- Too many pages

Page faults/sec vs. Number of page frames assigned

OS 2009-10
Load control

- If the WS of all processes > memory, there’s *thrashing*
- E.g. the PFF says a process requires more memory but none require less
- Solution: swapping – swap a process out of memory and re-assign its pages to others
Page size

- Page size $p$, $n$ pages of memory
- Average process size $s$, in pages $s/p$
- Each entry in the page table requires $e$ bytes
- On average $p/2$ is lost (fragmentation)
- Internal fragmentation: how much memory is not used within pages
- Wasted memory: $p/2 + se/p$
- Minimizing it yields the optimal page size (under simplifying assumptions)
Two memories

- Separate data and program address spaces
- Two independent spaces, two paging systems
- The linker must know about the two address spaces
Other issues

- **Shared pages, handle shared pages (e.g. program code)**
  - Sharing data (e.g. shared memory)
- **Cleaning policy**
  - Paging algorithms work better if there are a lot of free pages available
  - Pages need to be swapped out to disk
  - Paging daemon (write pages to disk during spare time and evict pages if there are too few)
Page fault handling

1. Page fault, the HW traps to the kernel
   1. Perhaps registers are saved (e.g. stack)
2. Save general purpose microprocessor information (registers, PC, PSW, etc.)
3. The OS looks for which page caused the fault (sometimes this information is already somewhere within the MMU)
4. The system checks whether the process has access to the page (otherwise a protection fault is generated, and the process killed)
5. The OS looks for a free page frame, if none is found then the replacement algorithm is run
6. If the selected page is dirty ($M=1$) a disk write is scheduled (suspending the calling process)
7. When the page frame is clean, the OS schedules another transfer to read in the required page from disk
8. When the load is completed, the page table is updated consequently
9. The faulting instruction is backed up, the situation before the fault is restored, the process resumes execution
Segmentation
Why?

- Many separate address spaces (segments) (e.g. data, stack, code, and many others if needed)
- Each segment is separate (e.g. addresses from 0 to some MAX)
- Segments might have different lengths
- Segment number + address within segment
- Linking is simplified (libraries within different segments can assume addresses starting from 0) – e.g. if a part of the libraries is recompiled the remainder of the code is unaffected
- Shared library (DLL’s) implementation is simpler (the sharing is simpler)
### Comparing paging and segmentation

<table>
<thead>
<tr>
<th>Consideration</th>
<th>Paging</th>
<th>Segmentation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Need the programmer be aware that this technique is being used?</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>How many linear address spaces are there?</td>
<td>1</td>
<td>Many</td>
</tr>
<tr>
<td>Can the total address space exceed the size of physical memory</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Can procedures and data be distinguished and separately protected?</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Can tables whose size fluctuate be accommodated easily?</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Is sharing of procedures between users facilitated?</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Why was this technique invented?</td>
<td>To get a large linear address space without having to buy more physical memory</td>
<td>To allow programs and data to be broken up into logically independent address spaces and to aid sharing and protection</td>
</tr>
</tbody>
</table>
Pure segmentations

Operating system

A

B

C

D

Operating system

A

B

C

Operating system

A

B

Operating system

OS 2009-10
Fragmentation

- External fragmentation:
  - Memory fragments not used (we’ve already seen this)
  - Memory wasted in unused holes
Segmentation + paging (Pentium)

- 16K segments
- 1G 32bit words (DoubleWords)
- Two tables: LDT, GDT – Local (to the process) and global (to the processor) descriptor table
- To work with a segment the machine loads the segment number into a special register (CS, DS, etc.) – CS, DS are 16 bit registers
- The descriptor of the segment (see next slide)
The segment descriptor

- This is used by the microcode within the Pentium to work with segments

<table>
<thead>
<tr>
<th>Base 24-31</th>
<th>G</th>
<th>D</th>
<th>0</th>
<th>Limit 16-19</th>
<th>P</th>
<th>DPL</th>
<th>S</th>
<th>Type</th>
<th>Base 16-23</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base 0-15</td>
<td></td>
<td></td>
<td></td>
<td>Limit 0-15</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Limit in pages/bytes
- 16/32 bit segment
- Privilege level
- Segment type protection
- Segment present in memory
- System/application
- Limit (20 bits)

Page size is 4K

CS/DS → Selector

Index | G/L | Privilege
Getting the address

- Selector
- Offset
- Base address
  - Limit
  - Other fields
- Descriptor
- 32-bit linear address

OS 2009-10
Paging on the Pentium

- 2-level page table in memory

<table>
<thead>
<tr>
<th>Dir</th>
<th>Page</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>10 bits</td>
<td>10 bits</td>
<td>12 bits</td>
</tr>
</tbody>
</table>

Address to the page

Each points to 4Mbytes of pages
More on the Pentiums

- TLB, to avoid repeated accesses to memory
- The whole thing can be used with just a single segment to obtain a linear 32bit address space
- Set base and limit appropriately
- Protection (a few bits)