Operating Systems

Christopher Kruegel
Department of Computer Science
UC Santa Barbara
http://www.cs.ucsb.edu/~chris/
Virtual Memory and Paging

- What if a program is too big to be loaded in memory
- What if a higher degree of multiprogramming is desirable
- Physical memory is split in page frames
- Virtual memory is split in pages
- OS (with help from the hardware) manages the mapping between pages and page frames
Mapping Pages to Page Frames

- Virtual memory: 64KB
- Physical memory: 32KB
- Page size: 4KB
- Virtual memory pages: 16
- Physical memory pages: 8
Memory Management Unit

- Automatically performs the mapping from virtual addresses into physical addresses
Memory Management Unit

- Addresses are split into a page number and an offset
- Page numbers are used to look up a table in the MMU with as many entries as the number of virtual pages
- Each entry in the table contains a bit that states if the virtual page is actually mapped to a physical one
- If it is so, the entry contains the number of physical page used
- If not, a page fault is generated and the OS has to deal with it
Page Tables

- Page tables contain an entry for each virtual page

- If virtual memory is big (e.g., 32 bit and 64 bit addresses) the table can become of unmanageable size

- Solution: instead of keeping them in the MMU move them to main memory

- Problem: page tables are used each time an access to memory is performed. Adding a level of indirection, may kill performance

- Another solution: multilevel tables
Multilevel Page Tables

- Two types of tables
  - Top-level (in MMU)
  - Second-level (in a page frame)
- Virtual address split in three parts
  - Top-level page index
  - Second-level page index
  - Offset

- Each entry in a table contains
  - Page frame number
  - Present/absent bit
  - Protection bits (RWX)
  - Modified bit (or “dirty bit”)
  - Referenced bit
  - Caching disabled bit (for memory mapped I/O)
Multilevel Page Tables

- 32 bit virtual address
- PT1: Top-level index, 10 bits
- PT2: Second-level index, 10 bits
- Offset: 12 bits
- Page size: 4KB
- Second-level maps 4MB (1024 entries of 4KB)
- Top-level maps 4GB (1024 entries of 4MB)
- Process is 12 MB
Locality of Reference

- Multilevel tables can hold many pages but they still require multiple index lookups for each memory access

- Most programs use a subset of their memory pages (loops, sequential executions, updates to the same data structures, etc) and the set changes slowly
  - Locality of reference
  - Working set
Translation Look-aside Buffers

- Translation Look-aside Buffer (TLB)
  - hardware device that allows fast access without using the page table

- Small number of entries (e.g., 8) accessible as an associative memory
- Checked before doing a page table mapping
- If lookup succeeds (hit), the page is accessed directly
- If TLB lookup fails (miss), the page table is used and the corresponding entry in TLB is added
- When an entry is taken out of the TLB, the modified bit is updated in the corresponding entry of the page table
- TLB management can be done both in hardware (MMU) or in software (by the OS)
Translation Look-aside Buffers

<table>
<thead>
<tr>
<th>Valid</th>
<th>Virtual page</th>
<th>Modified</th>
<th>Protection</th>
<th>Page frame</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>140</td>
<td>1</td>
<td>RW</td>
<td>31</td>
</tr>
<tr>
<td>1</td>
<td>20</td>
<td>0</td>
<td>R X</td>
<td>38</td>
</tr>
<tr>
<td>1</td>
<td>130</td>
<td>1</td>
<td>RW</td>
<td>29</td>
</tr>
<tr>
<td>1</td>
<td>129</td>
<td>1</td>
<td>RW</td>
<td>62</td>
</tr>
<tr>
<td>1</td>
<td>19</td>
<td>0</td>
<td>R X</td>
<td>50</td>
</tr>
<tr>
<td>1</td>
<td>21</td>
<td>0</td>
<td>R X</td>
<td>45</td>
</tr>
<tr>
<td>1</td>
<td>860</td>
<td>1</td>
<td>RW</td>
<td>14</td>
</tr>
<tr>
<td>1</td>
<td>861</td>
<td>1</td>
<td>RW</td>
<td>75</td>
</tr>
</tbody>
</table>
Inverted Page Table

• When virtual pages are too many, maintaining a page table is not feasible

• Solution: Inverted Page Table
  – One entry per physical page frame
  – Each entry contains a pair <process, virtual page>

• Address cannot be resolved simply by looking for an index in a table
• When process n accesses page p, the table must be scanned for an entry <n, p>
• Solution
  – TLB should catch most of the accesses
  – Table hashed on virtual addressed to resolve the mapping
Inverted Page Table

Traditional page table with an entry for each of the $2^{52}$ pages.

Indexed by virtual page.

256-GB physical memory has $2^{16}$ 4-KB page frames.

Hash table

Indexed by hash on virtual page

Virtual page

Page frame
Page Replacement Algorithms

• Page fault forces choice
  – Which page must be removed to make room for an incoming page?

• Criteria:
  – Modified page must first be saved
  – Unmodified just overwritten
  – Better not to choose an often used page that will probably need to be brought back in soon
R and M Bits

- Each page has a Reference (R) bit and a Modified (M) bit

- R and M bits can be provided by the hardware or can be managed in software
  - Initial process with no pages in memory
  - When page is loaded
    - R bit is set
    - READONLY mode is set
  - When modification is attempted a fault is generated
    - M bit is set
    - READ/WRITE mode is set
Optimal Page Replacement Algorithm

- Determine how far in the execution of the program a page will be hit

- Replace page needed at the farthest point in the future
  - Optimal but unrealizable

- Useful to compare with other algorithms
  - Log page use on first execution of the program
  - Develop a scheduling for following executions (with same inputs)
Not Recently Used Page Replacement Algorithm

• When process starts, R and M bit are set to 0
• Periodically R bit is cleared to take into account for pages that have not been referenced recently
• Pages are classified according to R and M
  – Class 0: not referenced, not modified
  – Class 1: not referenced, modified (R bit has been cleared!)
  – Class 2: referenced, not modified
  – Class 3: referenced, modified
• NRU removes page at random from lowest numbered non empty class
• Easy to implement but not terribly effective
FIFO Page Replacement Algorithm

- Maintain a linked list of all pages
- List is ordered by loading time
- The page at the beginning is replaced
  - Oldest one

- Advantage
  - Easy to manage

- Disadvantage
  - Page in memory the longest may be often used
Second Chance Page Replacement Algorithm

- Pages in list are sorted in FIFO order
- R bits are cleared regularly
- If the R bit of the oldest page is set it is put at the end of the list
- If all the pages in the list have been referenced the page that was “recycled” will reappear with the R bit cleared and will be thrown away
Second Chance
Page Replacement Algorithm

- Page list if fault occurs at time 20, A has R bit set

Page loaded first

Most recently loaded page

A is treated like a newly loaded page
Clock
Page Replacement Algorithm

- Same concept as Second Chance, different implementation
- Hand points to oldest page
- When a page fault occurs
  - If R=0: evict the page
  - If R=1: clear R and advance hand
Least Recently Used (LRU) Page Replacement Algorithm

• Assumption: pages used recently will be used again soon
  – Throw out page that has been unused for longest time

• Very expensive: Must keep a linked list of pages
  – Most recently used at front, least at rear
  – Update this list every memory reference !!

• Alternative
  – Maintain global counter that is incremented at each instruction execution
  – Maintain similar counter in each page table entry
  – If page is referenced copy global counter in page counter
  – Choose page with lowest value counter
Another alternative:

- Maintain matrix with $n \times n$ bits, where $n$ is the number of page frames

- If page $j$ is accessed
  - set to 1 all the bits in the corresponding row
  - set to 0 all the bits in the corresponding column

- At any moment the page with the row containing the lowest value is the least recently used
### Least Recently Used (LRU) Page Replacement Algorithm

<table>
<thead>
<tr>
<th>Page 0</th>
<th>Page 1</th>
<th>Page 2</th>
<th>Page 3</th>
<th>Page 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0 1 1 1</td>
<td>0 0 1 1</td>
<td>0 0 0 1</td>
<td>0 0 0 0</td>
</tr>
<tr>
<td>1</td>
<td>0 0 0 0</td>
<td>1 0 1 1</td>
<td>1 0 0 1</td>
<td>1 0 0 0</td>
</tr>
<tr>
<td>2</td>
<td>0 0 0 0</td>
<td>0 0 0 0</td>
<td>1 1 0 1</td>
<td>1 1 0 0</td>
</tr>
<tr>
<td>3</td>
<td>0 0 0 0</td>
<td>0 0 0 0</td>
<td>0 0 0 0</td>
<td>1 1 1 0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Page 0</th>
<th>Page 1</th>
<th>Page 3</th>
<th>Page 2</th>
<th>Page 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0</td>
<td>0 1 1 1</td>
<td>0 1 1 0</td>
<td>0 1 0 0</td>
<td>0 1 0 0</td>
</tr>
<tr>
<td>1 0 1 1</td>
<td>0 0 1 1</td>
<td>0 0 1 0</td>
<td>0 0 0 0</td>
<td>0 0 0 0</td>
</tr>
<tr>
<td>1 0 0 1</td>
<td>0 0 0 1</td>
<td>0 0 0 0</td>
<td>1 1 0 1</td>
<td>1 1 0 0</td>
</tr>
<tr>
<td>1 0 0 0</td>
<td>0 0 0 0</td>
<td>1 1 1 0</td>
<td>1 1 0 0</td>
<td>1 1 1 0</td>
</tr>
</tbody>
</table>
Not Frequently Used (NFU) Page Replacement Algorithm

• A counter is associated with each page

• At each clock interval, the counter is incremented if the page has been referenced (R=1)

• The page with the lowest counter is removed

• Problem: pages that have been heavily used in the past will always maintain high counter values

• Need for an aging mechanism
  – First shift the counter
  – Then set bit in most significant position if referenced
### Not Frequently Used (NFU) Page Replacement Algorithm

<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>00000000</td>
<td>11000000</td>
<td>11100000</td>
<td>11110000</td>
<td>01110000</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>10000000</td>
<td>01000000</td>
<td>00100000</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
Page Replacement Algorithm

- Locality of reference: Most programs use a subset of their memory pages (loops, sequential executions, updates to the same data structures, etc) and the set changes slowly.

- The working set is the set of pages used by the k most recent memory references.

- For a reasonable value of k, the number of page faults is reduced and the process does not *thrash*.

- If the working set can be determined it can be preloaded at context switch to minimize the initial demand of pages.
• $w(k,t)$ is the size of the working set at time, $t$
Working Set Page Replacement Algorithm

- Algorithm:
  when a page has to be evicted, find one that is not in the working set

- Use a shift register of size k

- At every reference
  - Right-shift the register
  - Insert page in left most position

- At replacement time
  - Remove duplicates and obtain working set
  - Remove page not in working set

- Problem: too heavy to maintain
Working Set Page Replacement Algorithm

- Use execution time instead of references

- Working set composed of pages referenced in the last $t$ msec of execution

- Each entry contains
  - The time the page was last used
  - The reference bit, R
• At every page fault
  – If R=1 the current time is written in the page entry
  – If R=0
    • If the “age” (current time - time of last reference) is smaller than t, the page is spared (but the page with the highest age/smallest time of last usage in the working set is recorded)
    • If the “age” is greater than t, the page is a candidate
      – If there is one or more candidates, the candidate with highest age is evicted
      – If there are no candidates the oldest page in the working set is evicted

• Problem: whole page table must be scanned
Working Set
Page Replacement Algorithm

Current virtual time: 2204

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

Page table:

- Information about one page
  - 2084
  - 2003
- Time of last use
  - 1980
- Page referenced during this tick
  - 1213
  - 2014
  - 2020
- Page not referenced during this tick
  - 2032
  - 1620
WSClock
Page Replacement Algorithm

• Every time a page is loaded is added to a circular list
• Each page is marked with the time of last use
• At page fault:
  – Examine page pointed by hand
    • If R=1: R is cleared, time is updated and hand advanced
    • If R=0:
      – If age is greater than t
        » If page is clean (M=0) then evict
        » If page is dirty (M=1) page is scheduled for writing to disk and hand advanced
      – If age is less than t the hand is advanced
  – If hand returns to initial position
    • If no writes are scheduled choose a random clean page
    • If writes have been scheduled continue until a clean, old page is found
WSClock
Page Replacement Algorithm
WSClock
Page Replacement Algorithm

[Diagram of page replacement algorithm with page numbers and timestamps]
# 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>
Modeling Page Replacement Algorithms

- Program memory access is characterized as a string of referenced page numbers, called *reference string*

- Memory has $n$ virtual pages and $m < n$ page frames

- Memory is modeled as an array $M$ divided in two portions:
  - Top $m$ rows are the actual mapping
  - Bottom $n - m$ rows represent swapped pages

- As the reference string is examined
  - the top portion is checked to see if the reference page is present
    - If not, a page fault is generated
  - In any case the chosen algorithm algorithm is used to determine the configuration of the next column
Example with LRU

- State of memory array, M, after each item in reference string is processed

| Reference string | 0 | 2 | 1 | 3 | 5 | 4 | 6 | 3 | 7 | 4 | 7 | 3 | 3 | 5 | 5 | 3 | 1 | 1 | 1 | 7 | 1 | 3 | 4 | 1 |
|                  | 0 | 2 | 1 | 3 | 5 | 4 | 6 | 3 | 7 | 4 | 7 | 3 | 3 | 5 | 5 | 3 | 1 | 1 | 1 | 7 | 1 | 3 | 4 | 1 |
|                  | 0 | 2 | 1 | 3 | 5 | 4 | 6 | 3 | 7 | 4 | 7 | 3 | 3 | 5 | 3 | 3 | 3 | 1 | 7 | 1 | 3 | 4 | 1 |
|                  | 0 | 2 | 1 | 3 | 5 | 4 | 6 | 3 | 3 | 4 | 4 | 7 | 7 | 7 | 5 | 5 | 5 | 3 | 3 | 7 | 1 | 3 | 1 | 3 |
|                  | 0 | 2 | 1 | 3 | 5 | 4 | 6 | 6 | 6 | 6 | 6 | 4 | 4 | 4 | 7 | 7 | 7 | 5 | 5 | 5 | 7 | 7 | 1 | 3 |
|                  | 0 | 2 | 1 | 1 | 5 | 5 | 5 | 5 | 5 | 6 | 6 | 6 | 4 | 4 | 4 | 4 | 4 | 4 | 5 | 5 | 5 | 5 | 5 | 5 |
|                  | 0 | 2 | 2 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 |
|                  | 0 | 0 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 |

UC Santa Barbara
## Belady's Anomaly

<table>
<thead>
<tr>
<th>Youngest page</th>
<th>Oldest page</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 0 1 4 4 4 2 3 3</td>
<td>0 1 2 3 0 1 1 1 4 2 2</td>
</tr>
<tr>
<td>0 1 2 3 0 0 0 1 4 4</td>
<td>9 Page faults</td>
</tr>
</tbody>
</table>

All pages frames initially empty

<table>
<thead>
<tr>
<th>Youngest page</th>
<th>Oldest page</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 3 3 3 4 0 1 2 3 4</td>
<td>0 1 2 2 2 3 4 0 1 2 3</td>
</tr>
<tr>
<td>0 1 1 1 2 3 4 0 1 2</td>
<td>10 Page faults</td>
</tr>
<tr>
<td>0 0 0 1 2 3 4 0 1</td>
<td>P P P P P P P P P P</td>
</tr>
</tbody>
</table>
Stack Algorithms

- Algorithms that satisfy
  \( M(m,r) \) is in \( M(m+1,r) \)
  are called stack algorithms

- Stack algorithms do not suffer from the Belady’s anomaly

- This means: if the same reference string is run on two memories with frame pages \( m \) and \( m+1 \) respectively, the set of pages loaded in memory in corresponding points in the reference string are one a subset of the other

- Violated at the seventh reference in previous example
Design Issues

- Local vs. global allocation policies
- Load control
- Page size and internal fragmentation
- Sharing pages
- Locking pages
- Separating policy and mechanism
Local versus Global Allocation Policies

- (a) Original configuration
- (b) Local page replacement
- (c) Global page replacement
Load Control

• Despite good designs, system may still thrash

• The Page Fault Frequency algorithm uses page faults frequency to determine
  – Which processes need more memory
  – Which processes need less

• Reduce number of processes competing for memory
  – Swap one or more to disk, divide up pages they held
  – Reconsider degree of multiprogramming
Page Size

Small page size

• Advantages
  – Less internal fragmentation
  – Better fit for various data structures, code sections
  – Less unused program in memory

• Disadvantages
  – Programs need many pages, larger page tables
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

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

- Two processes sharing same program, sharing its page table
Locking Pages in Memory

• Virtual memory and I/O occasionally interact

• Process issues call for read from device into buffer
  – While waiting for I/O, another processes starts up
  – New process has a page fault
  – Buffer for the first process may be chosen to be paged out

• Need to specify some pages locked
  – Exempted from being target pages
Separation of Policy and Mechanism

- Page fault handling with an external pager
Operating System Involvement

- Process creation
  - Determine program size
  - Create page table
  - Allocate swap
- Process execution
  - MMU reset for new process
  - TLB flushed
  - Make page table current
- Page fault time
  - Determine virtual address causing fault
  - Swap target page out, needed page in
- Process termination time
  - Release page table, pages