CS 355
Operating Systems

Paging Algorithms and Segmentation

The Working Set

- Processes exhibit locality of memory reference
  - During any phase of execution, the process references only a relatively small fraction of its pages.
- The set of pages a process is currently using is called its working set
- Demand paging
  - A process starts up with no pages in memory and a bunch of page faults until the working set is in memory
- If memory is too small to hold the entire working set, it will lead to trashing
- What happens during a context switch?

Working Set Model

- Designed to greatly reduce the page fault rate
- Pages are loaded before letting a process run
  - this is also known as pre-paging
- At any instance of time \( t \), there exists a set \( w(k, t) \) consisting of all pages used by the \( k \) most recent memory references.
- The working set changes over time
- References tend to cluster on a small number of pages: data, instruction and stack

The Working Set Page Replacement Algorithm

- Keeping track of all pages used in the last \( k \) memory references is expensive
- Instead, record pages used in the last time interval, say 100 msec.
- Virtual time (kept per process)
- The working set is the set of pages referenced during the past \( k \) seconds of virtual time

The Working Set

- Algorithm: replace the page not in working set
- Page table: clock tick clears R bit periodically
- On page fault, scan all R bits.
  - If 1, the current virtual time is written into tolu (time of last use) field.
  - If 0, compute age of page with virtual-tolu.
- If age is smaller than time interval \( k \)?
The Working Set Page Replacement Algorithm

Scan all pages examining R list
if (R = 1)
    read time of last use to current virtual time
if (R = 0 and age = 1)
    remove this page
if (R = 0 and age < 1)
    remember the smallest time

• The more page frames there are, the fewer pages faults a program will get?

Belady’s Anomaly

- FIFO with 3 page frames
- FIFO with 4 page frames
- P’s show which page references show page faults

Review of Page Replacement Algorithms

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Comment</th>
<th>Algorithm</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optimal</td>
<td>Not implementable, but useful as a benchmark</td>
<td>Belady’s Anomaly</td>
<td></td>
</tr>
<tr>
<td>NRU (Not Recently Used)</td>
<td>Very crude</td>
<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>
<td>LRU (Least Recently Used)</td>
<td>Excellent, but difficult to implement exactly</td>
</tr>
<tr>
<td>Clock</td>
<td>Realistic</td>
<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>
<td>Working set</td>
<td>Somewhat expensive to implement</td>
</tr>
<tr>
<td>WSClock</td>
<td>Good efficient algorithm</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Modeling Page Replacement Algorithms

- Memory references is an ordered list of page numbers called the reference string
- Paging system (with one process)
  - reference string
  - paging algorithm
  - number of page frames
- An n-row array to track memory
  - top: m pages that are currently in memory
  - bottom: n-m pages that have been referenced but paged out
  - m = # of frames; n = # of virtual pages
Memory Array with LRU

Reference string: 0 2 1 3 5 4 3 7 4 7 3 3 5 3 1 1 7 1 3 4 1

Page faults: P P P P P P P P P P

Distance string: = = = = = = = = 4 = 4 2 3 1 5 1 2 6 1 1 4 2 3 5 3

\[ M(m, r) \subseteq M(m+1, r) \]

The Distance String

Probability density functions for two hypothetical distance strings

Predicting Page Fault Rates

- The distance string can also be used to predict the number of page faults that will occur with memories of different size
- A page fault occurs any time a distance string is \( > m \)
- Start by scanning the distance string, page by page.
- Keep track of the number of times each distance occurs in vector \( C \)
- Compute vector \( F \) from \( C \): \n
\[ F_m = \sum_{k=m+1}^{n} C_k + C_{\infty} \]

Predicting Page Fault from Distance String

\[ C_1 = 4 \quad F_1 = 20 \]
\[ C_2 = 3 \quad F_2 = 17 \]
\[ C_3 = 3 \quad F_3 = 14 \]
\[ C_4 = 3 \quad F_4 = 11 \]
\[ C_5 = 2 \quad F_5 = 9 \]
\[ C_6 = 1 \quad F_6 = 8 \]
\[ C_7 = 0 \quad F_7 = 8 \]
\[ C_\infty = 8 \quad F_\infty = 8 \]

Segmentation

- So far virtual addresses go from 0 to max
- Consider programs with many growing memory chunks
- Shuffling – taking free space from other tables
- Lots of work, similar to managing overlays
- The solution is to have more than one virtual addressing spaces, so that each may grow and shrink at will, called segments
- Each segment consist of linear addresses going from 0 to some max
- An address in segmented memory has two parts – segment number and address within the segment

Paging

- Suffers from internal fragmentation
- NO user view
- Page size fixed
- In paging, user specifies only a single address
- Single address is partitioned by HW into page no. and offset
One Dimensional Address Space

- One-dimensional address space with growing tables
- One table may bump into another

Segmenation

- Allows each table to grow or shrink, independently

Advantage of Segmentation

- Allows data structures to grow independently
- Simplifies the linking process for compilation
  - Each procedure/function in separate segment
  - All separately compiled procedures start at segment address 0, respectively
  - A procedure call to a procedure in segment \( n \) will be linked as a two part address \((n, 0)\)
  - If the same procedure is subsequently modified and recompiled, no other procedures need be changed, as their starting addresses have not been modified
- Shared libraries are shared by putting it in a segment shared by multiple processes

Segmentation vs Paging

<table>
<thead>
<tr>
<th>Consideration</th>
<th>Paging</th>
<th>Segmentation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Need the program be executable that the technique being used?</td>
<td>Yes</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>No</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 fluctuates be accommodated?</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Is sharing of procedures between users feasible?</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Why use this technique overlapped?</td>
<td>To get large linear address space without tying to use 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>

Implementation of Pure Segmentation

- Development of checkerboarding
- Removal of the checkerboarding by compaction

Segmentation with Paging: MULTICS

- Descriptor segment points to page tables
- Segment descriptor – numbers are field lengths
MULTICS Virtual Address

A 34-bit MULTICS virtual address

MULTICS Address Translation

Conversion of a 2-part MULTICS address into a main memory address

MULTICS TLB

Segmentation with Paging: Pentium

A Pentium selector

Pentium Segment Descriptors

Pentium: Linear Address

Conversion of a (selector, offset) pair to a linear address

- Pentium code segment descriptor
- Data segments differ slightly
Pentium: 2-level Paging

Mapping of a linear address onto a physical address

Segmentation and Paging

- Paging is transparent to the programmer
- Paging eliminates external fragmentation
- Pieces are to be moved in and out – paging helps!

- Segmentation is visible to the programmer
- Segmentation allows for growing data structures, modularity, and support for sharing and protection
- Each segment is broken into fixed-size pages

Segmentation and Paging

- Combine paging & segmentation -- best of both worlds
- What about overheads in terms of HW support?
- Is it worth?

Pentium: Protection

- As long as a program restricts itself to using segments at its own level, everything works fine.
- Attempts to access data at a higher level are permitted, at a lower level are illegal and causes traps.
- Attempts to call procedures at a different level (higher or lower) are allowed but controlled.
  - To make an inter-level call, the CALL instruction must contain a selector instead of an address.
  - This selector designates a descriptor called a callgate, which gives the address of the procedure to be called.