Overview: Virtual Memory

- Background
- Demand Paging
- Page Replacement
- Allocation of Frames
- Thrashing and Data Access Locality
- Process Creation: Copy-on-Write
- I/O Interlock
- Memory-Mapped Files → later

Virtual Memory

- Previously achieved:
  - Separation of user logical memory from physical memory
    - ++ protection, reuse
    - Still, the whole program and data must be loaded in memory

- Virtual memory
  - Idea: Only part of the program needs to be in memory for execution.
    - Throw out pages currently not used (to secondary memory)
    - Logical address space can therefore be much larger than physical address space.
    - Allows address spaces to be shared by several processes.
    - Allows for more efficient process creation.
  - Virtual memory can be implemented via:
    - Demand paging
    - Demand segmentation

Virtual Memory That is Larger Than Physical Memory

- Bring a page into memory only when it is needed
  - Less I/O needed
  - Less memory needed
  - Faster response
  - More users

- Page is needed if referenced (load/store, data/instructions)
  - invalid reference ⇒ abort
  - not-in-memory ⇒ bring to memory
Demand paging ("lazy paging", "lazy swapping"): Transfer of a Paged Memory to Contiguous Disk Space

Rather than swapping entire processes (cf. swapping), we page their pages from/to disk only when first referenced.

Valid-Invalid Bit

- With each page table entry, a valid–invalid bit is associated (1 \( \Rightarrow \) page in memory, 0 \( \Rightarrow \) not-in-memory)
- Initially, valid–invalid bit is set to 0 on all entries
- Example of a page table snapshot:

Valid–Invalid Bit

- During address translation, if valid–invalid bit in page table entry is 0 \( \Rightarrow \) page fault

Steps in Handling a Page Fault (Case: a free frame exists)

- "Busy waiting" for slow disk?
- Better allocate the CPU to some other ready process in the meanwhile…

Page Table When Some Pages Are Not in Main Memory

Steps in Handling a Page Fault (Case: a free frame exists)

- Busy waiting" for slow disk?
- Better allocate the CPU to some other ready process in the meanwhile…

What happens if there is no free frame?

- Page replacement – find some page in memory, but not really in use, swap it out
  - Write-back only necessary if victim page was modified
  - Same page may be brought into memory several times
  - Various algorithms
  - Performance
    - want an algorithm that minimizes number of page faults

Performance of Demand Paging

- Page Fault Rate \( p \) 0 \( \leq \) \( p \) \( \leq \) 1.0
  - If \( p = 0 \): no page faults
  - If \( p = 1 \): every reference is a fault
- Memory access time \( t \)
- Effective Access Time (EAT)

\[
EAT = (1 - p) t + p \cdot ( \text{page fault overhead} + \text{time to swap page out, if modified} + \text{time to swap new page in} + \text{restart overhead} + t )
\]
Demand Paging Example

- **Write-back rate** \( w \) \( 0 \leq w \leq 1 \)
  
  % of page faults where page replacement is needed and the victim page has been modified so it needs to be swapped out

- **Example:**
  
  - Memory access time = 1 microsecond (\( \mu s \))
  - Time for swapping a page = 10 ms = 10,000 \( \mu s \)
  - Write-back rate \( w = 0.5 = 50\% \)
  
  \[ \text{expected swap time per page fault} = (1+w) \times 10,000 \mu s \]

  \[
  \text{EAT} = (1 - p) \cdot 1\mu s + p \cdot 14999 \mu s
  \]

Page Replacement

- Prevent over-allocation of memory by modifying page-fault service routine to include page replacement

- Use modify (dirty) bit to reduce overhead of page transfers – only modified pages are written to disk

- Page replacement completes separation between logical memory and physical memory – large virtual memory can be provided on a smaller physical memory

**Example: Need For Page Replacement**

- Memory access time = 1 microsecond (\( \mu s \))

- Time for swapping a page = 10 ms = 10,000 \( \mu s \)

- Write-back rate \( w = 0.5 = 50\% \)

  \[ \text{expected swap time per page fault} = (1+w) \times 10,000 \mu s \]

  \[
  \text{EAT} = (1 - p) \cdot 1\mu s + p \cdot 14999 \mu s
  \]

Basic Page Replacement

**Extended page-fault service routine:**

- Find the location of the desired page on disk

- Find a free frame:
  - If there is a free frame, use it
  - If there is no free frame, use a page replacement algorithm to select a victim frame and, if dirty, write its page to disk (1,2)

- Read the desired page into the (newly) free frame (3)

- Update the page table (4)

- Restart the process

**How to compare algorithms for page replacement?**

- **Goal:** find algorithm with lowest page-fault rate.

- **Method:** Simulation.
  
  - Assume initially empty page table
  - Run algorithm on a particular string of memory references (reference string – page numbers only)
  - Count number of page faults on that string.

  - In all our examples, the reference string is 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5.

**First-In-First-Out (FIFO) Algorithm**

- Use a time stamp or a queue

- **Victim** is the “oldest page”

- Assume table size = 3 frames / process

- (3 pages can be in memory at a time per process) and reference string:

  \[
  \begin{array}{cccccccc}
  1 & 2 & 3 & 4 & 1 & 2 & 5 & 1 \\
  2 & 3 & 1 & 2 & 3 & 4 & 5 & 3 \\
  3 & 1 & 2 & 3 & 4 & 5 & 2 & 4 \\
  \end{array}
  \]

- A total of 9 page faults

  - After page 3 is loaded, page 1 is the oldest of them all (gray box)
  
  - The fact that we re-use an existing page does not alter who has been in there the longest.
Expected Graph of Page Faults Versus Number of Frames

Generally, more frames => Less page faults

Same FIFO: More frames = better?

• 4 frames/process
  (4 pages can be in memory at a time per process)
• Reference string:
  1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

<table>
<thead>
<tr>
<th>Frame</th>
<th>Reference String</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1 1 1 1 1 5 5 5 4 4</td>
</tr>
<tr>
<td>2</td>
<td>2 2 2 2 2 1 1 1 2 2</td>
</tr>
<tr>
<td>3</td>
<td>3 3 3 3 3 2 2 2</td>
</tr>
<tr>
<td>4</td>
<td>4 4 4 4 4 3 3</td>
</tr>
</tbody>
</table>

FIFO Replacement – Belady’s Anomaly

• more frames with more page faults – possible!

An Optimal Algorithm

• "optimal":
  • has the lowest possible page-fault rate (NB: still ignoring dirty-ness)
  • does not suffer from Belady’s anomaly

Belady’s Algorithm:

Farthest-First, MIN, OPT

• Replace page that will not be used for the longest period of time....
• How do you know this?

Remark: Belady’s algorithm is only optimal if there are no dirty write-backs. Otherwise, it is a heuristic algorithm.

Least Recently Used (LRU) Algorithm

• Optimal algorithm not feasible?
  .... try using recent history as approximation of the future!

Algorithm:

• Replace the page that has not been used for the longest period of time

Example:

1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

<table>
<thead>
<tr>
<th>Frame</th>
<th>Reference String</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1 1 1 1 1 5 5 5 4 4</td>
</tr>
<tr>
<td>2</td>
<td>2 2 2 2 2 1 1 1 2 2</td>
</tr>
<tr>
<td>3</td>
<td>3 3 3 3 3 2 2 2</td>
</tr>
<tr>
<td>4</td>
<td>4 4 4 4 4 3 3</td>
</tr>
</tbody>
</table>

A total of 8 page faults

LRU Algorithm Implementations

• Timestamp implementation
  • The process maintains a logical clock (counter for number of memory accesses made)
    • Every page entry has a timestamp field
    • Every time a page is referenced, copy the logical clock into the timestamp field
    • When a page needs to be replaced, search for oldest timestamp

• Stack implementation
  • keep a stack of page numbers in doubly linked format:
    • page referenced:
      • move it to the top
      • requires 6 pointers to be changed
    • No search for replacement
  • stack before a
  • stack after

Remark: Linear search, stack and vector names
LRU Approximation Algorithms

- Reference bit for each page
  - Initially = 0 (by OS)
  - When page is referenced, bit set to 1
  - Replace a page whose bit is 0 (if one exists).
  - We do not know the access order, however.
  - May improve precision by using several bits.

- Second chance algorithm
  - Clockwise replacement
  - If page to be replaced (in clock order) has reference bit = 1 then:
    - set reference bit 0
    - leave page in memory
    - replace next page (in clock order), subject to same rules

Counting Algorithms

- Keep a counter of the number of references that have been made to each page
- LFU Algorithm: replaces page with smallest count
- MFU Algorithm: based on the argument that the page with the smallest count was probably just brought in and has yet to be used
- Not common in today's operating systems

Allocation of Frames

- Each process needs a minimum number of pages
  - Example: IBM 370 – 6 pages to handle SS MOVE instruction:
    - instruction is 6 bytes, might span 2 pages
    - 2 pages to handle from
    - 2 pages to handle to

  - Two major allocation schemes
    - fixed allocation
    - priority allocation

Fixed Allocation

- Equal allocation
  - Example: if there are 100 frames and 5 processes, give each process 20 frames.

- Proportional allocation
  - Allocate according to the size of process
    - \[ a_i = \frac{s_i}{S} \times m \]
    - \[ a_1 = \frac{10}{137} \times 64 = 5 \]
    - \[ a_2 = \frac{127}{137} \times 64 = 59 \]

Priority Allocation

- Use a proportional allocation scheme using priorities rather than size

- If process \( P_i \) generates a page fault,
  - select for replacement one of its frames
  - select for replacement a frame from a process with lower priority

Global vs. Local Allocation

- Global replacement
  - process selects a replacement frame from the set of all frames;
  - one process can take a frame from another

- Local replacement
  - each process selects from only its own set of allocated frames
If a process does not have "enough" pages, the page-fault rate is very high. This leads to:
- low CPU utilization
- operating system thinks that it needs to increase the degree of multiprogramming
- another process added to the system

Thrashing = a process is busy swapping pages in and out

Demand Paging and Thrashing

Why does demand paging work?

Locality model
- Locality = set of pages that are actively used together
- Process migrates from one locality to another
- Localities may overlap

Why does thrashing occur?
Σ sizes of localities > total memory size

Program Structure Matters!

Example: Matrix initialization
- int data[128][128];
- Each row is stored in one page
- Assume page size = 128 ints
- Assume < 128 frames available, LRU repl.

Program 1:
for (j = 0; j < 128; j++)
for (i = 0; i < 128; i++)
// traverse column-wise
data[i][j] = 0;
-> 128 x 128 = 16,384 page faults

Program 2:
for (i = 0; i < 128; i++)
for (j = 0; j < 128; j++)
// traverse row-wise
data[i][j] = 0;
-> 128 page faults

Copy-on-Write

Copy-on-Write (COW) allows both parent and child processes to initially share the same pages in memory
- If either process modifies a shared page, only then the page is copied
- Free pages are allocated from a pool of zeroed-out pages
- More efficient process creation as only modified pages are copied
- No improvement if child process does exec (overwrites all)

Alternatively: vfork() in Unix, Solaris, Linux:
- Light-weight fork variant: sleeping parent and immediate exec() by child process
I/O interlock

- I/O Interlock
  - Pages must sometimes be locked in memory.

- Consider I/O.
  Pages that are used for copying a file from a device must be locked from being selected for eviction by a page replacement algorithm.

Memory-Mapped Files

- Memory-mapped file I/O allows file I/O to be treated as routine memory access by mapping a disk block to a page in memory.

- A file is initially read using demand paging. A page-sized portion of the file is read from the file system into a physical page. Subsequent reads/writes to/from the file are treated as ordinary memory accesses.

- Simplifies file access by treating file I/O through memory rather than read() / write() system calls.

- Also allows several processes to map the same file, allowing the pages in memory to be shared.

Data used by Page Replacement Algorithms

- All algorithms need extra data in the page table, e.g., one or several of the following:
  - A reference bit to mark if the page has been used.
  - A modify (dirty) bit to mark if a page has been written to (changed) since last fetched from SM.
  - Additional mark bits.
  - Counters or time stamps (used for theoretical algorithms – must often be converted into use of mark bits).
  - A queue or stack of page numbers...

Page-Fault Frequency Scheme

- Establish “acceptable” page-fault rate.
  - If actual rate too low, process loses frame.
  - If actual rate too high, process gains frame.

Other Issues – Preparing

- Preparing
  - To reduce the large number of page faults (kernel entry overhead) that occurs at process startup.
  - Preparing all or some of the pages a process will need, before they are referenced.
  - But if prepared pages are unused, I/O and memory was wasted.
  - Assume s pages are prepared and fraction α of the pages is used.
    - Is cost of s * α saved page faults > or < than the cost of preparing s * (1 – α) unnecessary pages?
    - α near zero ⇒ preparing loses.
Other Issues – Page Size

- Page size selection must take into consideration:
  - fragmentation
  - table size
  - I/O overhead
  - locality

Other Issues – TLB Reach

- TLB Reach = the amount of memory accessible from the TLB
  - TLB Reach = (TLB Size) ÷ (Page Size)
- Ideally, the working set of each process is stored in the TLB. Otherwise there is a high degree of page faults.
- Increase the page size. This may lead to an increase in fragmentation as not all applications require a large page size
- Provide Multiple Page Sizes. This allows applications that require larger page sizes the opportunity to use them without an increase in fragmentation.

References