Chapter 9: Virtual Memory

Background
- Virtual memory – separation of user logical memory from physical memory.
  - Only part of the program needs to be in memory for execution
  - 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 (skip)

Virtual Memory That is Larger Than Physical Memory
- This is the new part beyond last time. Pages may reside on disk in addition to physical memory.

Virtual-address Space
- We still program with this model of process organization.
- The paging and VM layer support this abstraction.

Demand Paging
- Bring a page into memory only when it is needed
  - Less I/O needed
  - Less memory needed
  - Faster response
  - More users
- Page is needed → reference to it
  - invalid reference → abort
  - not-in-memory → bring to memory
- Lazy scheduler – never swaps a page into memory unless page will be needed
  - Swapper that deals with pages is a pager

Process swapper v. page swapper
- Lazy page swapper only brings pages that are needed from disk.
  - Page not referenced ⇒ page isn’t swapped from disk to memory.
- The scheduler discussed in chapter 8 is the process swapper. It is not lazy – entire processes are swapped in and out of memory, regardless of whether this includes unreferenced pages or not.
- The term pager is used to distinguish page-level swappers from whole process swappers.
Transfer of a Paged Memory to Contiguous Disk Space

Swap space organized as a set of pages.

Page Fault Interrupts

Page fault interrupt: interrupt to support page swapping transparently!!! (Add to our list of key interrupts: I/O interrupt, timer interrupt, (program exception, memory, hardware fault))

When a process references a page that is not in main memory, the HW generates a page fault interrupt which...

- Causes PC of interrupts process to be save to its PCB
- Causes PC to be loaded with address of the page fault interrupt handler
- Control passes seamlessly from running process to kernel PF interrupt handler which will begin I/O to swap in the needed page
- Scheduler puts faulting process in blocked queue; selects next process to run, and context switches to it.

Valid-Invalid Bit

- With each page table entry a valid-invalid bit is associated
  - v ⇒ in-memory, i ⇒ not-in-memory
- Initially valid-invalid bit is set to i on all entries
- Example of a page table snapshot:

<table>
<thead>
<tr>
<th>Frame #</th>
<th>Frame 1</th>
<th>Frame 2</th>
<th>Frame 3</th>
<th>Frame 4</th>
<th>Frame 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>valid-invalid bit</td>
<td>v</td>
<td>v</td>
<td>i</td>
<td>v</td>
<td>v</td>
</tr>
</tbody>
</table>

- During address translation, if valid-invalid bit in page table entry is i => page fault

Page Table When Some Pages Are Not in Main Memory

Steps in Handling a Page Fault

- If there is a reference to a page, first reference to that page will cause HW interrupt to operating system’s:
  - page fault interrupt handler
- Operating system looks at bit in PMT table to decide:
  - Invalid reference ⇒ abort
  - Just not in memory ⇒ swap in
- Get empty frame
- Swap page into frame
- Reset tables
- Set validation bit ⇒ v
- Restart the instruction that caused the page fault
Cost of demand paging

- Clearly this is not going to be totally transparent.
  - Disk is slow
    - Although we may get lucky if the disk has a fast cache.
  - When we looked at pages, we saw that the access time could vary depending on if the page table entry of an access was in the TLB versus main memory.
    - Two memory access costs versus one
  - We can quantify the performance impact of demand paging in a similar way.

Performance of Demand Paging

- Page Fault Rate 0 ≤ p ≤ 1.0
  - If p = 0 no page faults
  - If p = 1, every reference is a fault
- Effective Access Time (EAT)
  \[
  EAT = (1 - p) \times \text{memory access} + p \times (\text{page fault overhead} + \text{swap page out} + \text{swap page in} + \text{restart overhead})
  \]

Demand Paging Effective Access Time Example

- Memory access time = 200 nanoseconds
- Average page fault service time = 8 milliseconds
- \[
  EAT = (1 - p) \times 200 + p \times 8,000,000
  = 200 + p \times 7,999,800
  \]
- If one access out of 1,000 causes a page fault, then
  \[
  EAT = 200 + 7,999,800 = 8.2 \text{ microseconds}
  \]
  This is a slowdown by a factor of 40!!

EAT and demand paging

- Takeaway message
  - Keep page faults rare
  - This isn’t so hard most of the time.
    - Most of our processes fit in memory
      - As a result, most of the time processes page fault very rarely and don’t we notice the overhead.

You likely have suffered from page faults

- Quite likely we have experienced this and may not have known what the underlying cause was.
  - Running a program that consumes most of your memory, and suddenly the entire system grinds to an unusable speed.
  - Resuming a program that was in the background for a while, and experiencing unbearable speed problems while it comes back to life.
  - Running too many programs at once, all of them run slowly.
  - Each of these are cases where the VM system is doing its job
    - Pro: Programs executing fine (if slowly) even though consuming more than physical memory can support. Abstraction works!
    - Con: Disk speed is orders of magnitude slower than physical RAM. Usability of interactive apps is gone.

Out of frames?

- What happens when we run out of free frames in memory?
  - Page = virtual chunk of address space
    - frame = physical region of address space
  - Page replacement algorithms come into play here
    - Finding pages that are not in use
      - Swap them to disk, reuse their frame for page that is about to be used.
    - Algorithm for page replacement is important. We want to avoid swapping out pages actively in use.
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
  - In other words, if a page is going to be replaced and was never written to while in physical memory, we may be able to overwrite it without swapping to disk if the original version is still out in swap space.
- Page replacement completes separation between logical memory and physical memory – large virtual memory can be provided on a smaller physical memory

Need For Page Replacement

Where does B go in physical memory?

Basic Page Replacement

- 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
- Bring the desired page into the (newly) free frame; update the page and frame tables
- Restart the process

Page Replacement Algorithms

- Want lowest page-fault rate
- Evaluate algorithm by running it on a particular string of memory references (reference string) and computing the 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

Graph of Page Faults Versus The Number of Frames

Adding RAM corresponds to adding frames. Ever wonder why adding RAM to a machine can make it appear to go faster? This graph is why!
First-In-First-Out (FIFO) Algorithm
- Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- 3 frames (3 pages can be in memory at a time per process)

<table>
<thead>
<tr>
<th>Frame</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4, 5</td>
</tr>
<tr>
<td>2</td>
<td>1, 3</td>
</tr>
<tr>
<td>3</td>
<td>2, 4</td>
</tr>
</tbody>
</table>

- 4 frames
<table>
<thead>
<tr>
<th>Frame</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1, 5</td>
</tr>
<tr>
<td>2</td>
<td>2, 5</td>
</tr>
<tr>
<td>3</td>
<td>3, 2</td>
</tr>
<tr>
<td>4</td>
<td>4, 3</td>
</tr>
</tbody>
</table>

Belady's Anomaly: more frames → more page faults

FIFO Page Replacement
- Reference string: 7, 0, 1, 2, 0, 0, 4, 2, 3, 0, 2, 1, 0, 7, 0, 1

<table>
<thead>
<tr>
<th>Reference String</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
</tr>
</tbody>
</table>

Page Frames
- 7
- 10
- 12
- 14
- 16
- 18

FIFO Illustrating Belady's Anomaly

Optimal page replacement
- FIFO wasn't ideal. What is?
  - To start, we can perform a thorough experiment.
    - What would be the optimal page replacement algorithm.
  - Like SJF for scheduling, let's design an algorithm that can see into the future.
    - Obviously this won't be implementable.
    - **BUT**, it will give us a metric to evaluate real algorithms by.

Optimal Algorithm
- Replace page that will not be used for longest period of time
- 4 frames example
  - 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

<table>
<thead>
<tr>
<th>Frame</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>4, 5</td>
</tr>
</tbody>
</table>

- How do you know this?
- Used for measuring how well your algorithm performs

Optimal Page Replacement
- Reference string: 7, 0, 1, 2, 0, 0, 4, 2, 3, 0, 2, 1, 0, 7, 0, 1

<table>
<thead>
<tr>
<th>Reference String</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
</tr>
</tbody>
</table>

Page Frames
- 7
- 10
- 12
- 14
- 16
- 18
Realistic algorithms

- Can’t look into the future.
- But we can use the past to predict what we expect to happen based on what we already saw.
- Look for page that hasn’t been used for the longest period of time.
  - Likely it’s still not going to be used in the near future.
- We can do this.

Least Recently Used (LRU) Algorithm

- Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

  
  Reference string:
  
<table>
<thead>
<tr>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>5</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

- Counter implementation
  - Every page entry has a counter; every time page is referenced through this entry, copy the clock into the counter.
  - When a page needs to be changed, look at the counters to determine which are to change.

LRU Page Replacement

Reference string:

\[
\begin{array}{cccccccccccc}
7 & 0 & 1 & 2 & 3 & 0 & 4 & 2 & 3 & 2 & 1 & 2 & 0 & 1 & 7 & 0 & 1 \\
\end{array}
\]

page frames

LRU Algorithm (Cont.)

- Stack implementation – keep a stack of page numbers in a double link form:
  - Page referenced:
    - move it to the top
    - requires 6 pointers to be changed
  - No search for replacement

Use Of A Stack to Record The Most Recent Page References

Reference string:

\[
\begin{array}{cccccccccccc}
4 & 7 & 0 & 7 & 1 & 0 & 1 & 2 & 1 & 2 & 7 & 1 & 2 \\
\end{array}
\]

Stack before a

\[
\begin{array}{cccccccccccc}
2 & 1 & 0 & 7 & 4 \\
\end{array}
\]

Stack after b

LRU

- We have a problem.
  - Both techniques would require clock fields or stack to be updated on every reference.
  - That would be expensive.

- Likely not to have hardware assistance for real LRU.
- So, we try to approximate it with what we DO have hardware assistance for.
  - “Reference bits” – hardware sets this bit to 1 when a page is referenced.
  - Set to zero when page is brought in.
LRU Approximation Algorithms

- Reference bit
  - With each page associate a bit, initially = 0
  - When page is referenced bit set to 1
  - Replace the one which is 0 (if one exists)
    - We do not know the order, however

- Second chance
  - Need reference bit
  - Clock 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

Second-Chance (clock) Page-Replacement Algorithm

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

Allocation of Frames

- Each process needs 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 – For example, if there are 100 frames and 5 processes, give each process 20 frames.

- Proportional allocation – Allocate according to the size of process
  
  - \( S = \sum P_i \)
  
  - \( m = \text{total number of frames} \)
  
  - \( a_i = \text{allocation for } P_i = \frac{S_i}{S} \times m \)

  \[ m = 64 \]
  
  \[ S_1 = 10 \]
  
  \[ S_2 = 127 \]
  
  \[ a_1 = \frac{10}{137} \times 64 = 5 \]
  
  \[ a_2 = \frac{127}{137} \times 64 = 61 \]

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 number
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

Thrashing

- 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

Thrashing (Cont.)

Demand Paging and Thrashing

- Why does demand paging work?
  - Locality model
    - Process migrates from one locality to another
    - Localities may overlap
- Why does thrashing occur?
  - Σ size of locality > total memory size

Locality In A Memory-Reference Pattern

Working-Set Model

- Δ = working-set window = a fixed number of page references
  - Example: 10,000 instruction
- WSS (working set of Process P) = total number of pages referenced in the most recent Δ (varies in time)
  - If Δ too small will not encompass entire locality
  - If Δ too large will encompass several localities
  - If Δ = ∞ will encompass entire program
- D = Σ WSS = total demand frames
- If D > m → Thrashing
- Policy if D > m, then suspend one of the processes
Keeping Track of the Working Set

- Approximate with interval timer + a reference bit
- Example: $\Delta = 10,000$
  - Timer interrupts after every $5000$ time units
  - Keep in memory 2 bits for each page
  - Whenever a timer interrupts copy and sets the values of all reference bits to 0
  - If one of the bits in memory $= 1$ ⇒ page in working set
- Why is this not completely accurate?
- Improvement = 10 bits and interrupt every 1000 time units

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

Benefits of VM and Demand Paging

- Greater level of concurrency
- Better utilization of main memory
- Sharing of pages
- Copy-on-Write
- Memory-Mapped Files (later)
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 is the page copied.
- COW allows more efficient process creation as only modified pages are copied.
- Free pages are allocated from a pool of zeroed-out pages.

Before Process 1 Modifies Page C

- 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.

Memory-Mapped Shared Memory in Windows
Allocating Kernel Memory

- Treated differently from user memory
- Often allocated from a free-memory pool
- Kernel requests memory for structures of varying sizes
- Some kernel memory needs to be contiguous

Buddy System

- Allocates memory from fixed-size segment consisting of physically-contiguous pages
- Memory allocated using power-of-2 allocator
  - Satisfies requests in units sized as power of 2
  - Request rounded up to next highest power of 2
  - When smaller allocation needed than is available, current chunk split into two buddies of next-lower power of 2
    - Continue until appropriate sized chunk available

Buddy System Allocator

Slab Allocator

Slab Allocation

Other Issues -- Prepaging

- Prepaging
  - To reduce the large number of page faults that occurs at process startup
  - Prepare 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 p pages are prepared and α of the pages is used
    - Is cost of α * p = save pages faults > or < than the cost of preparing α * (1-α) unnecessary pages?
    - α near zero => prepaging 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) X (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

Other Issues – Program Structure

- Program structure
  - Int[128,128] data;
  - Each row is stored in one page
  - Program 1
    ```c
    for (j = 0; j < 128; j++)
      for (i = 0; i < 128; i++)
        data[i,j] = 0;
    ```
    - 128 x 128 = 16,384 page faults
  - Program 2
    ```c
    for (i = 0; i < 128; i++)
      for (j = 0; j < 128; j++)
        data[i,j] = 0;
    ```
    - 128 page faults

Reason Why Frames Used For I/O Must Be In Memory

Operating System Examples

- Windows XP
- Solaris
Windows XP

- Uses demand paging with clustering. Clustering brings in pages surrounding the faulting page.
- Processes are assigned working set minimum and working set maximum.
- Working set minimum is the minimum number of pages the process is guaranteed to have in memory.
- A process may be assigned as many pages up to its working set maximum.
- When the amount of free memory in the system falls below a threshold, automatic working set trimming is performed to restore the amount of free memory.
- Working set trimming removes pages from processes that have pages in excess of their working set minimum.

Solaris

- Maintains a list of free pages to assign faulting processes.
- Lotfree – threshold parameter (amount of free memory) to begin paging.
- Desfree – threshold parameter to increasing paging.
- Minfree – threshold parameter to being swapping.
- Paging is performed by pageout process.
- Pageout scans pages using modified clock algorithm.
- Scanrate is the rate at which pages are scanned. This ranges from slowscan to fastscan.
- Pageout is called more frequently depending upon the amount of free memory available.

Solaris 2 Page Scanner

End of Chapter 9