Virtual Memory

• Paging
  • Demand paging
  • Page Replacement Algorithms
    • FIFO, Optimal, LRU
    • Stack Algorithms
  • Implementations
    • Approximations
  • Strategies
    • Global vs. local
Thrashing

• If a process does not have “enough” pages, the page-fault rate is very high
  • Page fault to get page
  • Replace existing frame
  • But quickly need replaced frame back
  • This leads to:
    • Low CPU utilization
    • Operating system thinking 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.)

The diagram illustrates the relationship between CPU utilization and the degree of multiprogramming. The graph shows a curve that rises as the degree of multiprogramming increases, reaching a peak before declining, indicating thrashing. The x-axis represents the degree of multiprogramming, while the y-axis shows CPU utilization.
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?
  ∑ size of locality > total memory size
  • Limit effects by using local or priority page replacement
Locality In A Memory-Reference Pattern

Working set at time a
\{18, 19, 20, 21, 22, 23, 24, 29, 30, 33\}

Working set at time b
\{18, 19, 20, 24, 25, 26, 27, 28, 29, 31, 32, 33\}
Working-Set Model

- $\Delta \equiv$ working-set window $\equiv$ a fixed number of page references
  
  Example: 10,000 instructions

- $WSS_i$ (working set of Process $P_i$) =
  
  total number of pages referenced in the most recent $\Delta$ (varies in time)
  
  - if $\Delta$ too small will not encompass entire locality
  
  - if $\Delta$ too large will encompass several localities
  
  - if $\Delta = \infty \Rightarrow$ will encompass entire program

- $D = \sum WSS_i \equiv$ total demand frames
  
  - Approximation of locality

  - if $D > m \Rightarrow$ Thrashing

  - Policy if $D > m$, then suspend or swap out one of the processes

<page reference table>

\[
\ldots\ 2\ 6\ 1\ 5\ 7\ 7\ 7\ 5\ 1\ 6\ 2\ 3\ 4\ 1\ 2\ 3\ 4\ 4\ 3\ 4\ 3\ 4\ 4\ 4\ 1\ 3\ 2\ 3\ 4\ 4\ 3\ 4\ 4\ 4\ 4\ 4\ 4\ 4\ \ldots
\]

$WS(t_1) = \{1,2,5,6,7\}$

$WS(t_2) = \{3,4\}$
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 reference bits to memory
    • Set all reference bits to 0
  • If one of the bits for a page (in memory or reference bit) is 1
    $\Rightarrow$ page in working set
• Why is this not completely accurate?
• Improvement = 10 bits and interrupt every 1000 time units
Page-Fault Frequency

• More direct approach than WSS

• Establish “acceptable” page-fault frequency (PFF) rate and use local replacement policy
  • If actual rate too low, process loses frame
  • If actual rate too high, process gains frame
Working Sets and Page Fault Rates

1. Direct relationship between working set of a process and its page-fault rate
2. Working set changes over time
3. Peaks and valleys over time
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 and speeds file access by driving file I/O through memory rather than `read()` and `write()` system calls.
- Also allows several processes to map the same file allowing the pages in memory to be shared.
- But when does written data make it to disk?
  - Periodically and/or at file `close()` time.
  - For example, when the pager scans for dirty pages.
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
    • I.e. for device I/O
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

• For example, assume 256KB chunk available, kernel requests 21KB
  • Split into $A_L$ and $A_R$ of 128KB each
    • One further divided into $B_L$ and $B_R$ of 64KB
      • One further into $C_L$ and $C_R$ of 32KB each – one used to satisfy request

• Advantage – quickly **coalesce** unused chunks into larger chunk

• Disadvantage - fragmentation
Buddy System Allocator

physically contiguous pages

- 256 KB
- 128 KB \(A_L\)
- 128 KB \(A_R\)
- 64 KB \(B_L\)
- 64 KB \(B_R\)
- 32 KB \(C_L\)
- 32 KB \(C_R\)
Slab Allocator

• Alternate strategy
• **Slab** is one or more physically contiguous pages
• **Cache** consists of one or more slabs
  • (this is not the hardware memory cache of the cpu)
• Single cache for each unique kernel data structure
  • Each cache filled with **objects** – instantiations of the data structure
  • For example – Cache representing semaphores stores instances of semaphore objects
• When cache created, filled with objects marked as **free**
• When structures stored, objects marked as **used**
• If slab is full of used objects, next object allocated from empty slab
  • If no empty slabs, new slab allocated
• Benefits include no fragmentation, fast memory request satisfaction
Slab Allocation

- **3-KB objects**
- **7-KB objects**
- **kernel objects**
- **caches**
- **slabs**

Physically contiguous pages
Slab Allocator in Linux

• For example process descriptor is of type `struct task_struct`
• Approx 1.7KB of memory
• New task -> allocate new struct from cache
  • Will use existing free `struct task_struct`
• Slab can be in three possible states
  1. Full – all used
  2. Empty – all free
  3. Partial – mix of free and used
• Upon request, slab allocator
  1. Uses free struct in partial slab
  2. If none, takes one from empty slab
  3. If no empty slab, create new empty
Slab Allocator in Linux (Cont.)

• Slab started in Solaris, now wide-spread for both kernel mode and user memory in various OSes

• Linux 2.2 had SLAB, now has both SLOB and SLUB allocators
  • SLOB for systems with limited memory
    • Simple List of Blocks – maintains 3 list objects for small, medium, large objects
  • SLUB is performance-optimized SLAB removes per-CPU queues, metadata stored in page structure
Other Considerations -- Prepaging

• Prepaging
  • To reduce the large number of page faults that occurs at process startup
  • Prepage all or some of the pages a process will need, before they are referenced
  • But if prepaged pages are unused, I/O and memory was wasted
  • Assume $s$ pages are prepaged and $\alpha$ fraction of the pages is used
    • Is cost of saved pages faults $>$ or $<$ than the cost of prepaging unnecessary pages?
Other Issues – Page Size

• Sometimes OS designers have a choice
  • Especially if running on custom-built CPU

• Page size selection must take into consideration:
  • Fragmentation
  • Page table size
  • **Resolution**
  • I/O overhead
  • Number of page faults
  • Locality
  • TLB size and effectiveness

• Always power of 2, usually in the range $2^{12}$ (4,096 bytes) to $2^{22}$ (4,194,304 bytes)

• On average, growing over time
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

    ```
    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

    ```
    for (i = 0; i < 128; i++)
        for (j = 0; j < 128; j++)
            data[i, j] = 0;
    ```

    128 page faults
Other Issues – I/O interlock

• **I/O Interlock** – Pages must sometimes be locked into 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

• **Pinning** of pages to lock into memory
Operating System Examples

• Linux

• Windows

• Solaris
Linux

• Kernel memory – slab allocation

• Rest – uses demand paging
  • Global page replacement policy
    • LRU approximation clock algorithm
  • Maintains two lists
    • Active List – pages considered to be in use
    • Inactive List – pages that have not recently been referenced and are eligible to be reclaimed
• Accessed bit
  • Set when first allocated
  • Added to the rear of the active_list
  • When referenced access bit set and moved to the rear of the Active_list
• Periodically access bits for pages in the active_list are reset
• Pages from the front of the Active_list may move to the rear of the Inactive_list
• If a page on inactive_list is referenced it is moved to active_list
• Free page List
  • Paging daemon – *kswapd*
  • Awakens and if the size of Free page list below a threshold
    • Reclaims the pages from the front of Inactive_list
Windows

• 32 bit architecture
  • Virtual address space 2 GB – can be extended to 3 GB
  • Physical memory – up to 4 GB

• 64 bit architecture
  • 128 TB Virtual address space
  • Up to 24 TB physical memory (server version supports 128 TB )
Windows

• 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

- Assigns page from the list of free pages
- Maintains a list of free pages to assign faulting processes
- **Lotsfree** – 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
- **Priority paging** gives priority to process code pages
Solaris 2 Page Scanner

- **fastscan**
- **slowscan**

- **minfree**
- **desfree**
- **lotsfree**

**scan rate**

**amount of free memory**

8192