3. MEMORY MANAGEMENT
Copyrights on Figures

• Figures presented in these notes with the following captions are extracted from the textbook and copyrighted by the publisher as indicated

• DISCLAIMER
  – Whilst every attempt is made to maintain the accuracy and correctness of these notes, the author, Dr. Roberto Togneri, makes no warranty or guarantee or promise express or implied concerning the content.
Memory Hierarchy

- **Implications for Memory Management**
  - **Storage Hierarchy** → reduced access time implies increased costs and reduced storage capacity
  - **Locality of Reference** → During course of program execution, memory references by processor, for both instructions and data, tend to cluster
  - **Goal** → reduce memory access time for instructions/data
Cache and Swap Memory

- Memory Management → manage cache, main and swap memory
  - **Cache Memory**: operates in hardware
    - Exploits *locality of reference* idea so that when CPU makes a memory word transfer request the cache hardware will transfer a memory block → next CPU word reference should be in cache → maximise the *cache hit ratio*
  - **Swap Memory**: handled by OS
    - If there are too many processes which cannot all fit in memory at the same time some processes will be kept in the swap memory
      - high-level scheduler will move processes between swap and main memory (*swapping*) → part of memory management system
      - short-term scheduler will schedule processes in main memory to use CPU → memory management responsible for how process memory is accessed and structured
Memory Management Basics

• Functions of Memory Management system
  – Keep track of which parts of memory are in use by which processes
  – Allocate and de-allocate memory for processes in the most efficient manner
    • There are many ways to do this → simple mono-programming to sophisticated virtual memory paging/segmentation systems
  – Manage swapping of process memory from main memory and swap

• Mono-Programming
  – Run only one program at any one time
    • All of available main memory is available to process
    • No multi-programming or time-sharing possible
    • No need for swap memory (but you can “spool” jobs!)

• Figure 4.1, osdi2: Mono-programming examples, variation (c) is used by MS-DOS
Multi-Programming

• Fixed Partitions
  – Main memory is divided into N (possibly unequal) partitions

• Figure 4.2(b), osid2: Fixed memory partitions

  – How to allocate processes to partitions (swapping in or created)?
    • Which partition? Pick the smallest partition which fits (best-fit)
    • Problem of max. partition size → may not be large enough for some processes
    • Problem of internal fragmentation → unused part of partition
      – 128K process in 512K partition → 384K wasted!

  – Swapping processes out of main memory
    • Processes should be swapped out of main memory if they are in the blocked state for a significant amount of time
    • Swapping processes is expensive (disk I/O) → don’t do it too often!
Multi-Programming(*)

- Variable Partitions
  - Process is allocated as much memory as it requires and no more
    - Dynamic allocation as processes are created, terminated and swapped in/out

- How to allocate unused blocks to processes? (placement strategy)
  - More than one block of unused memory can be used, which one?
    - First fit: whichever unused block we examine first is big enough → simple
    - Best fit: examine all unused blocks and find the smallest one that can be used → expensive
    - Also next-fit, quick-fit and worst-fit and the Buddy algorithm

- Merging unused blocks
  - Adjacent unused blocks can be merged into one unused block
    - e.g. if 128K and 256K unused blocks are adjacent → single 384K unused block
    - requires manipulation of data structures used to detect adjacent free blocks
Multi-Programming

- How much memory does a process actually need?
  - Process memory requirement is dynamic as Stack and Data segments change
    - *Data/Heap segment*: `malloc()` and `free()` change data allocation
    - *Stack segment*: subroutine local variables and arguments change stack

- If allocated partition too small → process will run out of memory
  - Process will need to be moved to a larger unused block
  - Process can be swapped out of memory to wait for larger unused block
  - Process can be killed with a memory fault error

- If allocated partition too large → **internal fragmentation**
  - Lots of unused memory which can’t be used!

- Figure 4.4(b), osdi2: *Dynamic memory requirements of a process*
Multi-Programming

- **External Fragmentation**
  - Caused by unused blocks becoming increasingly small due to dynamic allocation of memory
  - Small unused block → unlikely to ever get allocated

- **Compaction** (defragmentation)
  - Relocate used partitions so that there are no unused blocks of memory between them

- **Placement Strategy**
  - Will best-fit reduce external fragmentation compared to first-fit?

Figure 7.4, OS3e: *The effect of dynamic partitioning*
Memory Management Data Structures

• How does OS keep track of unused and used blocks?
  – Bit Maps
    • Memory divided into small allocation units (say 4 bytes)
    • Special bit map, one bit assigned for each unit
      – 1 bit per 4 bytes (32 bits) → bit map occupies (wastes?) 1/33 of total memory
    • Each bit provides the status of the corresponding memory unit
      – 0 if unit is free ; 1 if unit is used
    • Simple data structure to maintain but expensive to use for memory allocation
      – Say, 400 free bytes are needed → search for a run of 100 consecutive 0’s in map!
  – Linked Lists
    • Dynamic data structure with an entry for each memory block
      – P for used partition, H for unused block
      – Start address of partition, Length of partition
      – Pointer to next entry
    • Need to detect and merge adjacent H entries into a single H entry → expensive
    • Best-fit placement requires examining the complete list whereas first-fit only requires finding the first H entry which is big enough → first-fit much quicker
      – Alternative is to keep list sorted in increasing/decreasing partition length → expensive
Memory Management Data Structures

Figure 4.5, osid2: (a) Memory structure, (b) Bit map, (c) Linked list

From Figure 4.6
- (a) requires changing entry 2 from P to H
- (b) requires merging entries 2 and 3 into a single H (list is one entry shorter)
- (c) requires merging entries 1 and 2 into a single H (list is one entry shorter)
- (d) requires merging entries 1, 2 and 3 into a single H (list is two entries shorter)
Relocation and Protection

- Structure of Program executable files
  - Instructions contain references to memory addresses
    - absolute addresses (AA)
      - fixed program memory address range
      - program always occupies the same area of memory
      - only good for mono-programming!
    - relative addresses (RA)
      - necessary for multi-programming
      - all addresses are specified relative to the start address (0000)
      - OS loader needs to translate relative address to an absolute address (relocation problem)
      - OS must not permit a relocation outside the allocated partition (protection problem)

- Solution to Relocation/Protection problem
  - Use special base and limit hardware registers
    - base = start address of process partition ; limit = length of partition
  - Per process base/limit values for each partition
    - Part of PCB which is loaded and saved at each process switch
  - For each (relative) memory reference x:
    If x > limit then protection fault error else AA = (base + x)
  - Address translation performed in hardware for speed
    - If in software the conversion will take longer than the actually memory reference itself!
Linking

- **Program executable (loadable object) file**
  - Final program executable consists of a collection of module object files
    - Each *.o file represents one module
    - Each library routine required represents one or more modules (libm.a  libc.a)
  - Linking merges all modules into the one loadable object
    - Each module has RA relative to start of module
    - Each RA needs to be modified so that it is relative to the start of the object

- **Static Linking**
  - The complete loadable object is linked and loaded into memory

- **Dynamic Linking**
  - Linking of some external modules (usually library routines) is deferred
    - Loadable object contains unresolved references to other programs
    - System library routines can be updated without re-compiling user programs
    - Requires special *shared* (*.so) or *dynamically linked* (*DLL*) libraries
  - Load-time dynamic linking
    - Unresolved references are resolved when the program is first loaded
    - Smaller program executable (code for unresolved references not included)
  - Run-time dynamic linking
    - Unresolved references are resolved during run-time when the unresolved program is called
External Fragmentation Analysis

- Definitions
  - \( m \) = bytes of total memory
  - \( f \) = % of total memory unused
    = (amount of memory used by all unused blocks) / (total memory)
  - \( s \) = average size in bytes of processes
  - \( k \) = average size of unused blocks (holes) as a fraction of \( s \)
    \( ks \) = average size of holes in bytes

- 50% rule
  - Adjacent processes are not merged (into the one process!)
    BUT adjacent holes are merged (into the one larger block)
  - On average if there are \( n \) processes we expect \((n/2)\) holes

- The analysis
  - \( n/2 \) holes must occupy \((m - ns)\) bytes of “unused” memory
    \( \rightarrow (n/2) \times ks = m - ns \rightarrow m = ns(1 + k/2) \)
  - Now: \( f = [(\text{number of holes} = n/2) \times (\text{average hole size} = ks)] / (\text{total memory} = m) \)
    \( \rightarrow f = ((n/2) \times ks) / m = ((n/2) \times ks) / (ns(1 + k/2)) \)
    \( \rightarrow f = k/(k+2) \)

- Some results
  - If holes are half the size of processes \((k = 0.5) \rightarrow f = 20\% \)
  - If holes are a quarter the size of processes \((k = 0.25) \rightarrow f = 11\% \)
  - Smaller hole sizes, more efficient memory utilisation
Overlays

• Does the process have to be in main memory to execute?
  – Locality of Reference implications
    • During execution interval $\Delta t$ only a fraction of the process’s instruction and data is actually needed $\rightarrow$ only load that fraction into memory $\rightarrow$ remainder on swap disk
    • More processes can be executing than can fit completely into main memory! But how can this be implemented?
  – Overlays: manual exploitation of locality of reference (the good old days)
    • Programmer designs program into separately loadable modules which are only loaded into memory as and is as required
      $\rightarrow$ programmer has to expend effort modularising program in an efficient manner
    • 2-pass assembler example
      – 4 modules: Pass 1 (70K) ; Pass 2 (80K) ; Symbol Table (20K) ; Common Code (30K)
      – Process memory is 200K $\rightarrow$ cannot run process with only 150K physical memory
      – Use Overlays (include 10K for overlay driver routine)
        Pass 1 overlay: Symbol Table + Common Code + Pass 1 $\rightarrow$ 120 + 10 = 130K OK!
        Pass 2 overlay: Symbol Table + Common Code + Pass 2 $\rightarrow$ 130 + 10 = 140K OK!

  – Virtual Memory: automatic exploitation of locality of reference (today)
    • Uses logical-to-physical address mapping, paging and/or segmentation to automatically assign physical memory as and when as required
    • Complex system which forms the basis of all modern memory management systems
Problems with Multi-Programming

• Problems with Multi-programming (variable partition)
  – External Fragmentation
    • Compaction → expensive
    • Placement strategies do not provide a real solution
      → first-fit is best because it’s the quickest!
  – All of process memory must be loaded into main memory
    • Must use overlays → onus (not bonus) on the programmer
      (as if (s)he didn’t have enough to worry about!)
  – Dynamic memory allocation (internal fragmentation)
    • Allocate more memory than is initially required → internal fragmentation
    • Allocate just enough memory
      → process swapped or terminated when data/stack has to grow
      → if larger partition allocated address relocation different for either data/stack
Problems with Multi-Programming

- Can’t handle multiple independent dynamic memory modules
  - *Compiler modules*: source text (grows), symbol table (grows), constants table (grows), parse tree (grows), stack (grows/shrinks)
  - More complex than simple dynamic memory allocation

![Memory structure for typical compiler](image)

*Figure 4.19, osdi2: Memory structure for typical compiler*
Virtual Memory (VM) Concepts: Paging

- **Physical Address (PA)**
  - the address seen by the physical memory
  - address range limited by the amount of physical memory that is available
    - 4MB RAM → 22-bit physical address
  - address range divided into small uniform chunks called **page frames**
    - typically 4K to 32K

- **Virtual Address (VA)**
  - the address used by the process
  - address range limited by the CPU address range
    - 32-bit CPU → 32-bit virtual address → 4 GB of virtual memory
  - address range divided into small uniform chunks called **pages**
    - page size must be the same as the page frame size

- **VA to PA mapping**
  - Process memory consists of a collection of contiguous pages which map to possibly non-contiguous page frames
  - Process making a VA reference must be mapped to a PA reference

- **Paging not swapping**
  - Instead of swapping whole process to disk, process pages are paged to disk
VM Solutions to MM Problems

• VM solution to external fragmentation
  – Non-contiguous page frames can span multiple unused blocks
    • Current Memory State: [ A | 100K hole | B | 32K hole]
      With 4K pages we have 100K = 25 page frames and 32K = 8 page frames
      Process C requires 120K (30 pages) of memory → can’t fit unused blocks
      : Map first 25 pages of process C to the 25 page frames from the 100K hole
      and the remaining 5 pages to the first 5 page frames from the 32K hole

• VM solution to loading all of process memory
  – With paging only those page frames currently referenced need be kept
    • 2-pass assembler requires a total of 200K but in pass 1 only references 120K of this
      memory → with 4K pages → 30 page frames are kept in main memory and
      remaining 20 page frames are kept on disk until they are needed

• Dynamic memory allocation
  – Assign a (ridiculously?) large virtual address range for the process and allow
    pages to be allocated / de-allocated as required
    • With a 32-bit CPU allocate 4GB virtual memory per process(!) as follows
      0 [ Code (fixed size) | Data → … ← Stack] 4GB
      Data and stack can never meet / collide unless there really is 4GB of RAM!
  – Relies on only used pages being allocated and mapped accordingly
    • e.g. {0M … 2M (code+data)} and {4095M … 4096M (stack)} need be allocated
VM Problems

- Possible internal fragmentation
  - Process memory usually not an integer number of pages
    - Last page is only partially used → wasted

- Overheads in VA to PA mapping
  - VA to PA address translation needs to be fast
    - Requires special on-chip CPU Memory Management Unit (MMU) hardware
  - Size of data structure (page table) used to map VA to PA is large
    - For 4GB VA range → typically need 1MB overhead per process!

- When to page in and page out pages?
  - More complex than swapping
    - There are more pages than processes to swap → too much time can be wasted
  - When and which should pages be paged out and paged into memory
    - Page Replacement Strategies

- VM with paging does not handle independent dynamic modules
  - Paging provides a linear address range for the same process
    - Makes it difficult to relocate dynamic modules which grow independently
  - Use VM with pure segmentation or segmentation with paging
VA to PA mapping: Page Tables

- \( VA = [ p \mid d ] \) and \( PA = [ f \mid d ] \)
  - \( PA = f \)-bit frame number + d-bit offset
    - \( 2^{f+d} \) bit PA
  - \( VA = p \)-bit page table index + d-bit page offset
    - \( 2^{p+d} \) bit VA
  - Usually \( p >> f \) since \( VA \) range >> \( PA \) range
  - \( p \)-bits used to retrieve corresponding \( f \)-bit frame from per process page table
    - One page table per process
    - Address of page table entry = \( Page \ Table \ Ptr. + p \)
    - \( 2^p \) entries in page table
    - Typical page table entry: \( [P \mid M \mid R \mid rwx \ access \mid f \)-bit frame number] \n      \( P = 1 \)-bit Present bit (1 if page in memory, 0 if page not in memory)
      \( M = 1 \)-bit Modified bit (1 if page has been modified (“dirty”), 0 if not)
      \( R = 1 \)-bit Referenced bit (1 if page has been referenced recently, 0 if not)
  - If \( P = 0 \) → page not in memory (on swap disk) → page fault
    - OS enters page fault handling routine
      - is there a free page? If not → page out a page according to page replacement strategy
        (R and M bits used to determine best page to evict → R=0 and M=0 is a good candidate)
      - read in required page from swap disk (page in) and modify page table accordingly
      - restart instruction
VA to PA mapping: Example (*)

Figure 4.9, osid2: Example of VA to PA mapping
VA to PA address translation system

- Figure 8.3, OS3e: Address translation in a paging system
Problem #1: VA to PA mapping is too slow!

- Page table is kept in main memory
  - Page table access requires an extra memory read
    → virtual memory references take twice as long as normal memory references!
    - “Cache” the page table → use hardware or software TLB

- Translation Look-aside Buffer (TLB)
  - Special on-chip MMU associative memory hardware
    - parallel lookup rather than indexing → fast but expensive!
    - **Hardware management:** MMU handles TLB miss events
    - **Software management:** OS handles TLB miss events
      - **TLB miss actions:** find entry in page table, load new entry in TLB, restart instruction
        - With software management on-chip MMU has space for extra cache and other features to
          (more than) compensate for software memory management overheads

- TLB is reloaded per process context switch

<table>
<thead>
<tr>
<th>Valid</th>
<th>VA page</th>
<th>Modified</th>
<th>Protection</th>
<th>PA 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>

- Figure 4.12, osdi2: **Typical TLB entries**
VA to PA address translation system (take #2)

- Figure 8.5, OS3e: *Incorporating the TLB in the address translation system*
Problem #2: Page Table too big!

- Process is assumed to have access to complete VA range
  - Page table is too big!
    - 32-bit CPU and 4K pages $\rightarrow p = 20 \rightarrow 2^{20} = 1$ million entry page table
    $\rightarrow$ entries at least 1 byte $\rightarrow$ 1 MB per page table per process!

- Multi-level Page Tables
  - 2-level page tables
    - $VA = [ p \mid q \mid d ]$ and $PA = [ f \mid d ]$
      - p-bits to index top-level page table $\rightarrow$ returns pointer to 2\textsuperscript{nd}-level page table
      - q-bits to index 2\textsuperscript{nd}-level page table $\rightarrow$ returns f-bit page frame
    - $2^p$ entries in top-level page table $\rightarrow$ each entry spans $2^{q+d}$ of VA memory
      - top-level page table entry: $[ \text{Pointer to 2}^{\text{nd}}\text{-level page table} ]$
    - $2^q$ entries in 2\textsuperscript{nd}-level page table $\rightarrow$ each entry spans $2^d$ of VA memory (1 page)
      - 2\textsuperscript{nd}-level page table entry: $[P \mid M \mid R \mid f\text{-bit frame number}]$
      - there are $2^p \times (2^{\text{nd}}\text{-level page tables})! \rightarrow$ BUT not all of these are needed
  - Good: Only keep those lower-level page tables that are needed
  - Bad: $n$-level page tables require $n$ memory references to perform translation!
    - TLB is mandatory!
  - Ugly: multi-level page table, TLB and page fault handling $\rightarrow$ what a mess!
Multi-Level Page Tables (*)

Figure 4.10, osdi2: 2-level page table example
Inverted Page Table

- **Problems with page tables**
  - Need a page table (or multi-level page tables) per process
  - TLB needed to minimise direct memory read access to page tables
  - 64-bit CPU now common
    - VA range is now 4GB of 4GB addresses (16 “something real big” B)
    - multi-multi-multi-level page tables?

- **Inverted Page Table**
  - Page table of main memory page frames rather than per process pages
    - Only one table needed for whole system
    - Size depends on physical memory not CPU
      - 128 MB RAM $\rightarrow$ 17-bit PA ($\ll 64$-bit VA!)
    - PA = [f | d], page table with $2^f$ entries
      - $f$th entry = [ProcessID | p] maps to $f$th page frame
  - Too good to be true!
    - VA to PA mapping expensive since this requires a “search”
    - Process (PID) makes a VA = [ p | d ]
      - page table searched for matching [PID | p] $\rightarrow f$th entry matches $\rightarrow$ PA = [ f | d ]
    - By comparison: normal page tables are indexed, or an associative lookup made with TLB
    - Solution? inverted page table stored as an associative memory in hardware (like TLB but bigger!)
VM with Segmentation

• Two views of memory
  – Paging: one-dimensional single linear address range
    • process sees 000…00H to FFF … FFH → single VA address
  – Segmentation: two-dimensional collection of independent linear address ranges
    • process sees separate {000…00H to FFF…FFH} ranges in each segment
    • multiple VA addresses → Segment Address (SA) = \[ s \mid d = VA \]
      s → segment number
      d → offset within segment = VA

• Advantages with Segmentation
  – Solves multiple independent dynamic memory module problem
    • one logical segment = one functional module!
    • *Compiler segments*: (s = 0) source text, (s = 1) symbol table,
      (s = 2) constants table, (s = 3) parse tree, (s = 4) stack
  – Separate program routine modules (e.g. library routines) can be kept separate
    • With paging all modules have to be linked into a single contiguous linear address
      – if one module is modified all modules have to be recombined again
      – dynamic linking requires independent region in VA for loading external modules
    • Segmentation with one module per segment
      – re-compile and insert modified module (no need to keep other modules around)
      – facilitates load-time and run-time dynamic linking
VM with Segmentation

- **Sharing** of process data
  - Code (text) can be shared by different processes arising from the same program
    - (N users running N processes of single program /bin/tcsh)
  - Easy with segmentation, more complicated with paging
    - Paging → need to identify which page frames represent the code area, and then manipulate the individual process page table entries → **shared pages**
    - Segmentation → allocate one segment for code, make it a shared segment and allow other processes to specify this same segment in the SA

- **Protection** of different logical segments
  - Different access rights for the different logical segments to protect program developer and debug programs!
    - code segment → execute only (detects accidental modification of code data)
    - data segment → read/write only (detects accidentally jumping into an array data area)

<table>
<thead>
<tr>
<th>Consideration</th>
<th>Paging</th>
<th>Segmentation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Need the programmer be aware of this technique?</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>How many linear address ranges?</td>
<td>1</td>
<td>Many</td>
</tr>
<tr>
<td>Can total address space exceed physical memory size?</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Can data and procedures be separately identified and protected?</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Can tables (lists, stack, etc.) with fluctuating size requirements be easily accommodated?</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Is sharing of procedures between users facilitated?</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

* Figure 4.21, osdi2: *Comparison of paging and segmentation*
Pure Segmentation (*)

- **SA = [ s | d ] and PA = [ Base + d ]**
  - s-bit used to reference entry in **segment table**
    - 2^s entries in segment table; Segment table entry = [ length | base ]
    - Length (limit): specifies the maximum size of the segment
      Base: start address of segment in main memory
  - If (d > length) then “segmentation fault” else PA = (Base + d)

- Figure 8.10, OS3e: Address translation in a segmentation system
Segmentation with Paging

- Similar problems to multi-programming
  - External Fragmentation between segments & Internal Fragmentation within segment
  - All of segment memory must be loaded or none
- Segmentation with Paging → the best of both worlds
  - Segment table contains pointer to page table for data within segment
    - Length of segment must be in integer number of pages
    - No need for length specification → use page fault mechanism

- Figure 8.11, OS3e: Address translation in a segmentation/paging system

3/02/00
COS214, Roberto Togneri, E&E Eng, Univ. of Western Australia
Segmentation with Paging: Intel Pentium

- Each SA reference is converted to a segment selector
  - Pentium has six segment registers
    - CS register holds the selector for the code segment
    - DS register holds the selector for the data segment
  - Segment selector format
    - [13-bit segment table index | 1-bit type | 2-bit privilege level]
      - type = 0 → Global Descriptor Table (GDT) → global segment table accessible by all processes, usually holds OS kernel related segments
      - type = 1 → Local Descriptor Table (LDT) → process’s own segment table privilege level → 0-3 levels, 0 being the most privileged
    - Both LDT and GDT are restricted to 8K (13-bit index) entries
  - Segment table (descriptor table) entry format

- Figure 4.28, osdi2: Pentium code segment descriptor. Data segments differ slightly
Segmentation with Paging: Intel Pentium

- Segment Descriptor details
  - Page Size of 4KB
  - 32-bit Base address → 32-bit Pentium addressing
    - Base is broken up into 3 pieces → compatibility with Intel 286
  - 20-bit Limit address
    → 1MB segment size (if G=0, limit is in bytes)
    → 4GB segment size (if G=1, limit is in pages)

- Pure segmentation
  - \( SA = [ s \mid d ] \)
    - \( s \) = segment selector → returns (Base, Limit) information
    - If (\( d > \text{Limit} \)) then “segment fault” else \( PA = \text{Base} + d \)

- Segmentation with Paging
  - \( SA = [ s \mid d ] \)
    - \( s \) = segment selector → returns (Base, Limit) information
    - \( VA = \text{Base} + d = [ 10\text{-bit Dir} \mid 10\text{-bit Page} \mid 12\text{-bit Offset}] \)
      Dir = Page directory (top-level page table) → 32-bit pointer to 2nd-level Page table
      Page = 2nd-level Page table → returns 20-bit \( f \) (other 12-bits used for P,M,R, etc.)
    - \( PA = [ f \mid \text{Offset} ] \)
      Global register contains the address of Page Directory table for the process
Segmentation with Paging: Intel Pentium

• Pure Paging
  – Segmentation with Paging using Base = 0, G = 1 and Limit = 4GB
    • Effectively uses one segment which spans full 32-bit CPU address range

• Other Pentium features
  – Small on-chip TLB storing most recent entries
    • TLB entry: [ Valid | Dir | Page | Page frame | access and M bits ]
  – Privilege
    • Each process context contains a 0-3 privilege level indication in the PSW register
    • If process with privilege L(Pr) attempts to access a segment with a lower selector privilege L(Seg) < L(Pr) then a protection trap is triggered
    • Typical uses of levels
      – Level 0 → Kernel privileges
      – Level 1 → System Calls
      – Level 2 → Shared libraries
      – Level 3 → User programs
Page Fault Handling (*)

- If page table or TLB has P = 0 then a page fault occurs
  - Requested page must be fetched from the swap disk (or created if new)
  - A free page frame in physical memory is needed or an existing page needs to be evicted

  - **Page Replacement Policy:** Which page should be evicted?

![Page Fault Handling Diagram](image)

*Figure 8.6, OS3e: Page Fault Handling*
Page Replacement Policy

• Why a policy?
  – If no policy, then a page at random can be selected (Random Policy)
    • If it is a frequently accessed page, then another page fault will soon occur for this particular page → too many page faults in so short a time
  – If policy, then a close to optimal selection can be made (Optimal Policy)
    • Select the page which will have its next reference farthest in the future relative to all other pages → can’t predict the future → this policy cannot be implemented

• First-In, First-Out (FIFO)
  – Keep a linked list for all page frames sorted in order of use and select oldest page in system for replacement
    • new page (created or paged in) is added to head of list
    • page at tail of list (oldest in system) is selected for replacement
  – Problem? Oldest page may be old because its heavily used (i.e. not yet retired)!
Page Replacement Policy: LRU

• Least Recently Used (LRU)
  – Each page frame has the time of last access recorded and the page frame with the oldest last access (least recently used) is selected for replacement
  • Recording the access time for each page is the tricky implementation bit!
    – Updating a time-of-last-use field with the current timestamp involves a system call for every memory access instruction!
    – Keeping a stack (linked list) which moves the currently referenced page to the top of the list involves manipulating a linked structure for every memory access instruction!
• Requires special hardware assistance or a hybrid software/hardware approximation
  – Better than FIFO but worse than Optimal
    • Examples: Optimal (9 page faults), FIFO(15 page faults), LRU(12 page faults)
    • If it can be implemented efficiently this is the winning entry!
• LRU approximations
  • Second-chance FIFO
  • Clock
  • Enhanced second-chance → NRU (Not Recently Used)
  • Not Frequently Used (NFU) or Least Frequently Used (LFU)
Page Replacement Structures

• The R and M bits
  – Special hardware for LRU is usually not available → use the R and M bits
    • R = page referenced recently ; M = page has been modified in main memory
    • R and M bits are usually set by the CPU hardware
      – If page is referenced then R = 1 and if page is modified then M = 1
      – The R is (re)set to 0 periodically (i.e. with each context switch)
    • R and M bits can also be set by the OS software
      – When page is first paged in, set R = 1, M = 0 and set page access to read-only
      – Reset R = 0 periodically and set R = 1 for each page reference
      – When an attempt is made to modify page → protection fault signals OS that the page is being modified → set M = 1 and set page access to read-write

• Linked Lists
  – A linked list of used page frames is maintained and searched or manipulated to find the page frame to replace
    • A pointer is used to travel from the head to the tail of the queue
    • The tail usually points back to the head → circular queue!
    • Double-linked list allows pointer to also move from tail to head
Second-Chance FIFO Policy

- **Second-Chance FIFO**
  - Assume a double-linked list sorted by page arrival
    - Newly arrived page frames are placed at the head of the queue → queue automatically sorted with the newest page at the head and oldest at the tail
    - There is one item in the list for each page frame
  - Check the tail of the queue (oldest page) until page is found
    - If \( R = 1 \) then reset \( R = 0 \), move page to head of queue, and check the next item
    - If \( R = 0 \) then select page frame for replacement
      - Resetting \( R \) supplements (and may replace) the periodic resetting to 0
      - If older pages are heavily used then \( R = 1 \) and these pages will not be selected
      - Selected page frame is placed at the head of the queue with \( R = 1 \)

- Figure 4.13, osid2: Operation of second chance (b) oldest page A has \( R=1 \) → moved to the head and \( R \) reset to 0
Clock Policy

- **Clock**
  - Assume a circular queue sorted by page arrival
    - Pointer is used to indicate where the next newly arrived page frame is inserted and is then incremented to the next entry
    - The circular queue has one entry for each page frame
  - Use the pointer to search for the next page to replace
    - If \( R = 1 \) then reset \( R = 0 \) and increment pointer
    - else if \( R = 0 \) then select page for replacement
- Simpler implementation of second-chance FIFO, but same idea

![Figure 8.14, OS3e: Clock-policy operation](image)
NRU and NFU Policy

• Not Recently Used (NRU)
  – Use both the R and M bits and define 4 classes
    • Class 1: \( R = 0, \ M = 0 \) (best page to replace)
    • Class 2: \( R = 0, \ M = 1 \)
    • Class 3: \( R = 1, \ M = 0 \)
    • Class 4: \( R = 1, \ M = 1 \) (worst page to replace)
  – Search linked-list or circular queue for the first page that is in the lowest class and select that page for replacement
    • May require a complete pass through list as the first page in each class is searched from class 1 to class 4.
    • Checking for \( M=0 \) in addition to \( R=0 \) ensures pages which do not need to be saved to swap are selected in preference
    • As with Clock policy the R can be (re)set to 0 for each frame that is checked

• Not Frequently Used (NFU) / Least Frequently Used (LFU)
  – Counter of references is kept for each page
    • At each context switch OS scans each page and adds value of R to counter
  – Search for counter with smallest value and select that page for replacement
    • Page which has been referenced the least is selected (and counter reset to 0)
      – Older pages with initially high counts will not be selected!
        (e.g. pass 1 assembler pages will have high counts and will not be selected for replacement when page 2 assembler data is referenced)
    • NFU is indeed the LFU algorithm for page replacement !!!!
Thrashing

- High page fault rate → Thrashing
  - If \( \frac{\text{number of page faults}}{\text{total number of page references}} \) is too high
    - System spends most of its time with (page in) and (page out) operations which require disk I/O read and writes
    - \textit{CPU utilisation is low but interactive response is slow and disk I/O activity is high}
      - “My mouse is not moving, the keyboard is dead but the disk is grinding” (Anon user)
  - Causes?
    - Too many processes requiring more memory “now” than is physically available
      - Reduce number of processes or buy more physical memory
    - Not enough pages in memory for active process → too many page faults
      - \textbf{Pre-page} or load all pages forming the process’s \textbf{Working Set Size} (WSS)
    - Too many pages in memory for process → too many page faults by other processes
      - \textbf{Page release} any pages not forming part of WSS
Working Set Size

- **Working Set Model**
  - Due to locality of reference only a small fraction of process memory (the WSS) is referenced recently
    - This “fraction” changes with time as process executes different functions
  - If all pages in the current WSS are in memory there are no page faults
    - Obvious by definition of the WSS
    - Page faults occur when the WSS changes
  - Need to identify which pages belong to the WSS
    - Page replacement policies which make use of the R bit
  - Need to pre-page and ensure there are enough WSS pages in main memory for all processes
    - Fetch and Allocation Policy

![Diagram of WSS and Time](image)
Fetch and Allocation Policies

• Fetch Policy
  – **Demand Paging**: Pages are allocated only when a page fault occurs
    • Normal mode of operation
  – Pre-Paging: Pages are allocated in anticipation of a reference
    • If pages allocated before reference then no page fault
    • If incorrectly anticipated then more page faults
    • OS can’t anticipate pages but program developer can

• Allocation Policy
  – Local vs. Global Scope
    • *Local Scope*: only pages belonging to process causing page fault are candidates for replacement
    • *Global Scope*: all pages in system are candidates for replacement
  – Fixed vs. Variable Allocation
    • *Fixed Allocation*: Process has a fixed number of pages it can use
      – Implies local scope since a page fault requires replacing an existing page
    • *Variable Allocation*: Process is allowed to vary the number of pages in can have
  – Best Policy: Variable allocation, Global Scope
    • *Variable Allocation*: Allows process to keep the current WSS in memory
    • *Global Scope*: Allows system to replace the least used page in system
PFF Allocation and Page Release

- **Page Fault Frequency (PFF) Algorithm**
  - More pages allocated → lower page fault rate
    - Figure 4.18, osdi2: *Page Fault Rate as function of allocated page frames*
    - If Process A fault rate < lower threshold B
      - Process A has too many frames allocated → use replacement policy to select pages to save to disk (**page release**) → add such pages to system list of free pages
    - If Process A fault rate > upper threshold A
      - Process A does not have enough frames allocated → allow process to allocate more free pages

- **Page Release and Free Pages**
  - System periodically scans system for least recently / oldest pages and saves these to the swap disk → added to *pool of free pages*
Locked Pages and Optimal Page Size

- **I/O pages cannot be paged out of main memory**
  - I/O operation may require device to send/read data directly from/to memory
    - DMA transfers
    - Reading from and writing to streaming tape device
  - Pages associated with I/O buffer must be kept in memory
    - Device is given the physical address for the location of buffer memory
    - If pages are paged out and then paged in when needed the *physical location in memory may change!*
  - Pages are **locked** → cannot be subject to replacement
    - Usually kernel buffer pages (not allowed by user processes, for obvious reasons!)

- **Optimum Page Size**
  - Smaller page sizes
    - Reduces internal fragmentation → less wastage
    - BUT more pages needed → larger page tables
  - Analysis
    - Average process size = s, Page Size = p, Size of Page Table entry = e
      - Number of pages needed per process = (s/p)
      - Size of per process page table = (s/p).e
      - Average internal fragmentation in last process page = (p/2)
    - Overhead (Ov) = (se/p) + (p/2) → Min. Overhead when d(Ov)/dp = 0
      - $d(Ov)/dp = -se/p^2 + 1/2 = 0 \rightarrow p = \sqrt{2se}$ → *large process requires larger page size*
    - $s = 2$MB, $e = 8 \rightarrow$ Optimal $p = 5793$ bytes ≈ 4K or 8K pages
Case Study: UNIX System V (*)

- **Paged Virtual Memory Data Structure Formats**
  - **Page Table**
    - One page table per process
  - **Disk Block Descriptor**
    - One entry for each process page describing where on disk swap the page copy is kept
    - Each process needs to keep track of the copies on swap for (page in) and (page out)
  - **Page Frame Data Table**
    - Describes each frame of real memory and indexed by frame number
    - Used to keep track of free pages and identify I/O (locked) pages
      - kernel forces page release in order to keep a pool of free pages
  - **Swap-use Table**
    - One table for each swap device, with one entry for each page on the device

- **Figure 8.20, OS3e: UNIX SVR4 Memory Management Formats**
Case Study: UNIX System V

• Memory Management Processes
  – Swapper (PID = 0)
    • Wakes up every 4 seconds or so to check which processes to swap in / out
      – Swap out candidates
        • process idling a long time
        • process in main memory a long time
      – Swap in candidates
        • swapped out a long time
        • process small compared to available system memory
  – Init Daemon (PID = 1)
    • Spawns all other processes
  – Page Daemon (PID = 2)
    • demand paging
    • global Clock replacement policy (modified)
    • automatic page release mechanism
      – if number of free page frames falls below a minimum threshold
        then page daemon wakes up and evicts pages from main memory

• System Calls
  • \( p = \text{malloc}(\text{int size}) \) allocates size bytes of data memory which can be accessed
    sequentially by the pointer \( p \) → allows run-time allocation of variables
  • \( \text{free}(p) \) deallocates the data memory which was previously allocated to \( p \)
  • \( \text{brk}(\text{end_data_segment}) \) increases the data segment allocated to the process
    → called by \( \text{malloc}(\ ) \) when data space runs out
Case Study: Windows NT

- Special Features
  - Shared Memory
    - Allows different process page table mappings to refer to the same physical region
    - Protection by Copy-on-Write page table entry flag
  - Protection
    - Two modes of process operation: user and kernel mode
    - Read-only and Read/Write
    - Execute-only (cannot read nor modify page but can jump to it)
      - these pages are usually shared between process invocations of the same program
    - Guard-page
      - used to demarcate data structures, generates an exception when accessed but continues
    - No-access
      - used to prevent access to unallocated page address, generates exception and creates a trap
    - Copy-on-write
      - if process attempts to write to a shared page, a copy of it is made for the process
        → allows other processes to still use the original unmodified shared page
  - Address Space
    - 4GB VA → lower 2GB for User memory + upper 2GB for System memory
      - System memory: directly mapped addresses, locked system memory, etc.
    - 2-level page table (for compatibility with Intel)
Case Study: Windows NT

• Data Structures
  – Page table
    • one page table per process to represent VA to PA mapping for process
  – Page Frame Database
    • used by VM manager to record the state of physical page frames
    • page frames can be in one of 6 states
      – valid: in use by process, contains pointer to page table that references it
      – zeroed: free and contents initialised to zero
      – free: free but un-initialised
      – standby: was in use by process, but is no longer in process’ working set
        modified: same as standby except page has been modified and not yet written to disk
      – bad: hardware error generated and this page in memory is defective

• Policies and Working Set
  – Demand page fetch policy
  – Local FIFO replacement page policy
    • each process has a minimum and maximum working set limit
      – VM manager will lower the minimum if process does not generate enough faults
    • if process has reached its maximum working set → local replacement used
  – Automatic working-set trimming
    • if process memory almost full VM manager will evict pages so processes only use their minimum working set number