Virtual Memory Management

B.Ramamurthy

Introduction

- Memory refers to storage needed by the kernel, the other components of the operating system and the user programs.
- In a multi-processing, multi-user system, the structure of the memory is quite complex.
- Efficient memory management is very critical for good performance of the entire system.
- In this discussion we will study memory management policies, techniques and their implementations.

Topics for discussion

- Memory management requirements
- Memory management techniques
- Memory operation of relocation
- Virtual memory
- Principle of locality
- Demand Paging
- Page replacement policies

Memory management requirements

- **Relocation**: Branch addresses and data references within a program memory space (user address space) have to be translated into references in the memory range a program is loaded into.
- **Protection**: Each process should be protected against unwanted (unauthorized) interference by other processes, whether accidental or intentional. Fortunately, mechanisms that support relocation also form the base for satisfying protection requirements.

Memory management requirements (contd.)

- **Sharing** : Allow several processes to access the same portion of main memory : very common in many applications. Ex. many server-threads executing the same service routine.
- **Logical organization :** allow separate compilation and run-time resolution of references. To provide different access privileges (RWX). To allow sharing. Ex: segmentation.

...requirements(contd.)

- **Physical organization:** Memory hierarchy or level of memory. Organization of each of these levels and movement and address translation among the various levels.
- **Overhead** : should be low. System should be spending not much time compared execution time, on the memory management techniques.

Memory management techniques (Covered last class)

- **Fixed partitioning**: Main memory statically divided into fixed-sized partitions: could be equal-sized or unequal-sized. Simple to implement. Inefficient use of memory and results in internal-fragmentation.
- **Dynamic partitioning :** Partitions are dynamically created. Compaction needed to counter external fragmentation. Inefficient use of processor.
- **Simple paging:** Both main memory and process space are divided into number of equal-sized frames. A process may in non-contiguous main memory pages.

Memory management techniques (contd.)

- **Simple segmentation :** To accommodate dynamically growing partitions: Compiler tables, for example. No fragmentation, but needs compaction.
- **Virtual memory with paging:** Same as simple paging but the pages currently needed are in the main memory. Known as demand paging.
- **Virtual memory with segmentation:** Same as simple segmentation but only those segments needed are in the main memory.
- **Segmented-paged virtual memory**

Basic memory operation: Relocation

- A process in the memory includes instructions plus data. Instruction contain memory references: Addresses of data items, addresses of instructions.
- These are *logical addresses*: relative addresses are examples of this. These are addresses which are expressed with reference to some known point, usually the beginning of the program.
- *Physical addresses* are absolute addresses in the memory.
- Relative addressing or position independence helps easy relocation of programs.

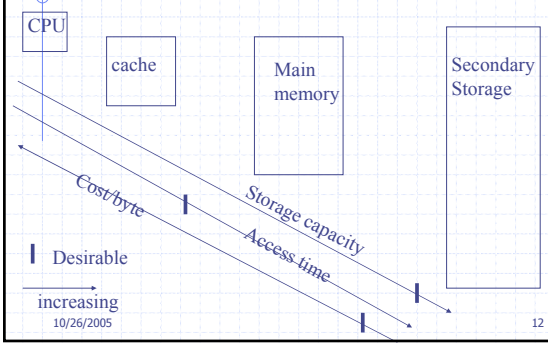
Demand Paging and Virtual Memory

- Consider a typical, large program you have written:
  - There are many components that are mutually exclusive. Example: A unique function selected dependent on user choice.
  - Error routines and exception handlers are very rarely used.
  - Most programs exhibit a slowly changing **locality of reference**. There are two types of locality: **spatial and temporal**.

Locality

- **Temporal locality**: Addresses that are referenced at some time Ts will be accessed in the near future (Ts + delta_time) with high probability. Example : Execution in a loop.
- **Spatial locality**: Items whose addresses are near one another tend to be referenced close together in time. Example: Accessing array elements.
- How can we exploit this characteristics of programs? Keep only the current locality in the main memory. Need not keep the entire program in the main memory. (Virtual Memory concept)

Desirable memory characteristics



Demand paging

- Main memory (physical address space) as well as user address space (virtual address space) are logically partitioned into equal chunks known as pages. Main memory pages (sometimes known as frames) and virtual memory pages are of the same size.
- Virtual address (VA) is viewed as a pair (virtual page number, offset within the page). Example: Consider a virtual space of 16K , with 2K page size and an address 3045. What the virtual page number and offset corresponding to this VA?

Virtual Page Number and Offset

3045 / 2048 = 1

3045 % 2048 = 3045 - 2048 = 997

VP# = 1

Offset within page = 997

Page Size is always a power of 2? Why?

Page Size Criteria

Consider the binary value of address 3045 :

1011 1110 0101

for 16K address space the address will be 14 bits. Rewrite:

00 1011 1110 0101

A 2K address space will have offset range 0 - 2047 (11 bits)

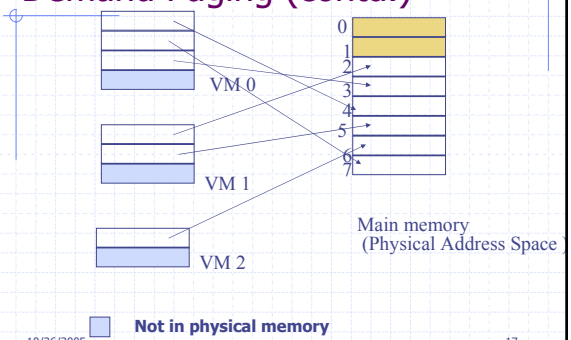
00 1 | 011 1110 0101

Page# Offset within page

Demand paging (contd.)

- There is only one physical address space but as many virtual address spaces as the number of processes in the system. At any time physical memory may contain pages from many process address space.
- Pages are brought into the main memory when needed and "rolled out" depending on a page replacement policy.
- Consider a 8K main (physical) memory and three virtual address spaces of 2K, 3K and 4K each. Page size of 1K. The status of the memory mapping at some time is as shown.

Demand Paging (contd.)



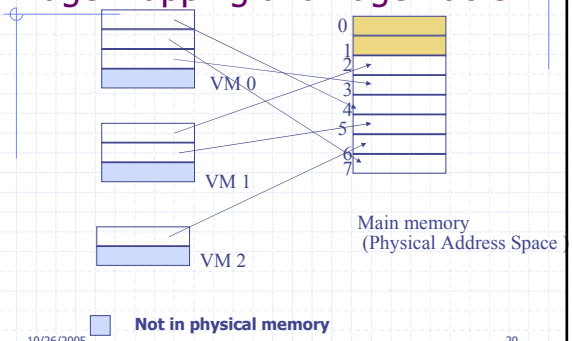
Issues in demand paging

- How to keep track of which logical page goes where in the main memory? More specifically, what are the data structures needed?
  - Page table, one per logical address space.
- How to translate logical address into physical address and when?
  - Address translation algorithm applied every time a memory reference is needed.
- How to avoid repeated translations?
  - After all most programs exhibit good locality. "cache recent translations"

Issues in demand paging (contd.)

- What if main memory is full and your process demands a new page? What is the policy for page replacement? LRU, MRU, FIFO, random?
- Do we need to roll out every page that goes into main memory? No, only the ones that are modified. How to keep track of this info and such other memory management information? In the page table as special bits.

Page mapping and Page Table



Page table

- One page table per logical address space.
- There is one entry per logical page. Logical page number is used as the index to access the corresponding page table entry.
- Page table entry format:

Presentbit, Modify bit, Other control bits, Physical page number

- Look at *TranslationEntry* class in *translate.h* in *machine* directory of *nachos*
- Look at *translate.cc* for code for *address translation* that we discuss next.

Address translation

- **Goal: To translate a logical address LA to physical address PA.**

1. LA = (Logical Page Number, Offset within page)
   Logical Page number LPN = LA DIV pagesize
   Offset = LA MOD pagesize
2. If Pagetable(LPN).Present step 3
   else **PageFault** to Operating system.
3. Obtain Physical Page Number (PPN)
   PPN = Pagetable(LPN).Physical page number.
4. Compute Physical address:
   PA = PPN *Pagesize + Offset.

Example

- Page size : 1024 bytes.
- Page table

Virtual_page#	Valid bit	Physical_Page#
0	1	4
1	1	7
2	0	-
3	1	2
4	0	-
5	1	0

- PA needed for 1052, 2221, 5499

Page fault handler

- When the requested page is not in the main memory a page fault occurs.
- This is an interrupt to the OS.
- Page fault handler:

1. If there is empty page in the main memory , roll in the required logical page, update page table. Return to address translation step #3.
2. Else, apply a replacement policy to choose a main memory page to roll out. Roll out the page, if modified, else overwrite the page with new page. Update page table, return to address translation step #3.

Page Fault Handling (1)

- Hardware traps to kernel
- General registers saved
- OS determines which virtual page needed
- OS checks validity of address, seeks page frame
- If selected frame is dirty, write it to disk

Page Fault Handling (2)

- OS brings schedules new page in from disk
- Page tables updated
- Faulting instruction backed up to when it began
- Faulting process scheduled
- Registers restored
- Faulted process is resumed

Translation look-aside buffer

- A special cache for page table (translation) entries.
- Cache functions the same way as main memory cache. Contains those entries that have been recently accessed.
- When an address translation is needed lookup TLB. If there is a miss then do the complete translation, update TLB, and use the translated address.
- If there is a hit in TLB, then use the readily available translation. No need to spend time on translation.

TLBs – Translation Lookaside Buffers

Valid	Virtual page	Modified	Protection	Page frame
1	140	1	RW	31
1	20	0	R X	38
1	130	1	RW	29
1	129	1	RW	62
1	19	0	R X	50
1	21	0	R X	45
1	860	1	RW	14
1	861	1	RW	75

A TLB to speed up paging

Page Size (1)

Small page size

- Advantages
  - less internal fragmentation
  - better fit for various data structures, code sections
  - less unused program in memory
- Disadvantages
  - programs need many pages, larger page tables

Page Size (2)

- Overhead due to page table and internal fragmentation

page table space

$$overhead = \frac{s \cdot e}{p} + \frac{p}{2}$$

internal fragmentation

- Where
  - s = average process size in bytes
  - p = page size in bytes
  - e = page entry

Optimized when $p = \sqrt{2se}$

Resident Set Management

- Usually an allocation policy gives a process certain number of main memory pages within which to execute.
- The number of pages allocated is also known as the resident set (of pages).
- Two policies for resident set allocation: fixed and variable.
- When a new process is loaded into the memory, allocate a certain number of page frames on the basis of application type, or other criteria.
- When a page fault occurs select a page for replacement.

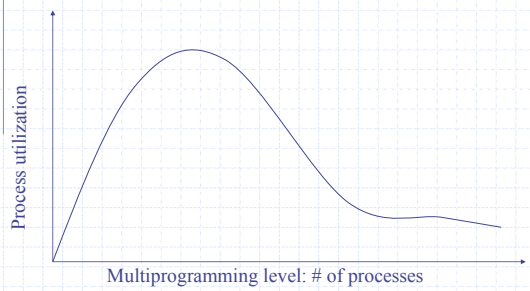
Resident Set Management (contd.)

- **Replacement Scope**: In selecting a page to replace,
  - a **local replacement policy** chooses among only the resident pages of the process that generated the page fault.
  - a **global replacement policy** considers all pages in the main memory to be candidates for replacement.
- In case of variable allocation, from time to time evaluate the allocation provided to a process, increase or decrease to improve overall performance.

Load control

- Multiprogramming level is determined by the number of processes resident in main memory.
- Load control policy is critical in effective memory management.
  - Too few may result in inefficient resource use,
  - Too many may result in inadequate resident set size resulting in frequent faulting.
  - Spending more time servicing page faults than actual processing is called "thrashing"

Load Control Graph



Load control (contd.)

- Processor utilization increases with the level of multiprogramming up to to a certain level beyond which system starts "thrashing".
- When this happens, only those processes whose resident set are large enough are allowed to execute.
- You may need to suspend certain processes to accomplish this.

Page Replacement Algorithms

- Page fault forces choice
  - which page must be removed
  - make room for incoming page
- Modified page must first be saved
  - unmodified just overwritten
- Better not to choose an often used page
  - will probably need to be brought back in soon

Optimal Page Replacement Algorithm

- Replace page needed at the farthest point in future
  - Optimal but unrealizable
- Estimate by …
  - logging page use on previous runs of process
  - although this is impractical

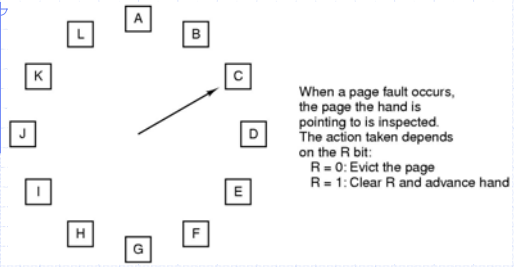
Not Recently Used Page Replacement Algorithm

- Each page has Reference bit, Modified bit
  - bits are set when page is referenced, modified
- Pages are classified
  1. not referenced, not modified
  2. not referenced, modified
  3. referenced, not modified
  4. referenced, modified
- NRU removes page at random
  - from lowest numbered non empty class

FIFO Page Replacement Algorithm

- Maintain a linked list of all pages
  - in order they came into memory
- Page at beginning of list replaced
- Disadvantage
  - page in memory the longest may be often used

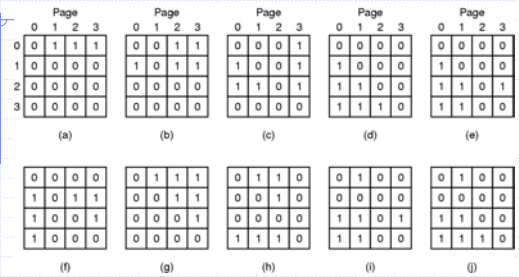
The Clock Page Replacement Algorithm



Least Recently Used (LRU)

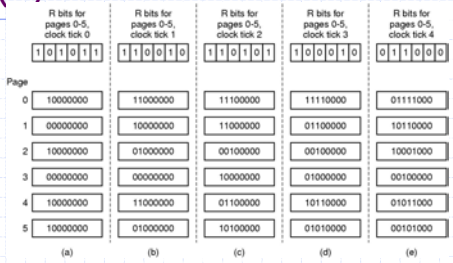
- Assume pages used recently will used again soon
  - throw out page that has been unused for longest time
- Must keep a linked list of pages
  - most recently used at front, least at rear
  - update this list every memory reference !!
- Alternatively keep counter in each page table entry
  - choose page with lowest value counter
  - periodically zero the counter

Simulating LRU in Software (1)



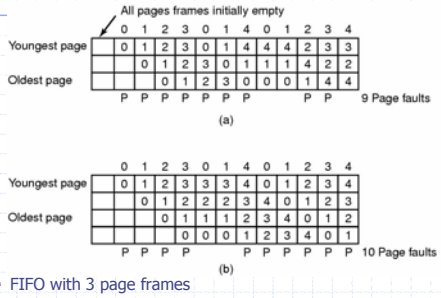
LRU using a matrix – pages referenced in order 0,1,2,3,2,1,0,3,2,3

Simulating LRU in Software (2)



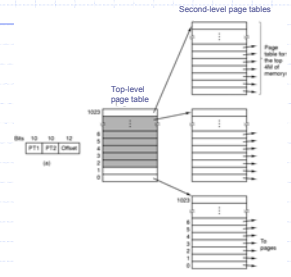
- The aging algorithm simulates LRU in software
- Note 6 pages for 5 clock ticks, (a) – (e)

Modeling Page Replacement Algorithms Belady's Anomaly



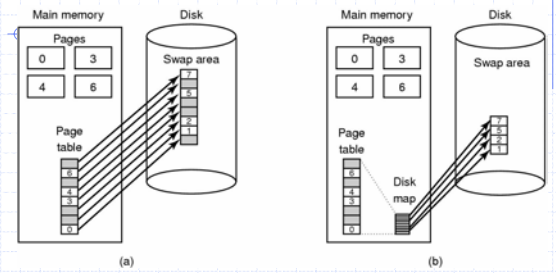
- FIFO with 3 page frames
- FIFO with 4 page frames
- *P*s show which page references show page faults

Two Level Page Tables



- 32 bit address with 2 page table fields
- Two-level page tables

Backing Store



(a) Paging to static swap area
(b) Backing up pages dynamically