Paging & Segmentation

Frédéric Haziza <daz@it.uu.se>

Department of Computer Systems
Uppsala University

Spring 2007

Outline

Definition

Paging

Memory-management technique that permits the physical address space of a process to be non-contiguous

Avoids external fragmentation

Basics

New setup:

- Physical memory is divided into frames
- Logical memory is divided into pages

A frame

- has the same size as a page
- is a place where a (logical) page can be (physically) placed

Logical to Physical

Translation

Logical address space → page table per process → physical address space

Page 0
Page 1
Page 2
Page 3

Logical memory

0	1
1	4
2	3
3	7

Page table

↓ Frame number

0	
1	Page 0
2	
3	Page 2
4	Page 1
5	
6	
7	Page 3

Physical memory

Note: Similar to *relocation register per page*

How does it work?

See black board...

...take your notes...

Page size

Page size is defined by the hardware.

Often of the form 2^n, between 512 bytes and 16 MB, typically 4-8 KB

page number	page offset
p	d
$m - n$	n

Must be carefully chosen: too large means more internal fragmentation, too small means too much overhead in paging management and processing.

If page table entry = 4 byte (= 32 bits), 2^{32} page frames.

If page frame = 4 KB $\Rightarrow$ can address 2^{44} bytes (=16 TB) of physycial memory.

Characteristics

The current page table (address) is saved and restored when doing a context switch.

The OS also has a frame table containing information about all frames, e.g. whether they are free or which process(es) is/are using it

Characteristics

- No external fragmentation

  all frames (physical memory) can be used by processes
- Possible internal fragmentation

  on average 1/2 page per process (the last page)
- The physical memory used by a process is no longer contiguous
- The logical memory of a process is still contiguous
- The logical and physical addresses are separated

  the process does not see the translation or the difference to having physical memory

Implementation

Implementation must be done in hardware for efficiency.

All process address references go through address translation: memory not allocated to the process can not be touched by the processes.

Small page tables

Example (PDP-11)

16-bit addresses with 8 KB page size
$\Rightarrow$ 8 pages

kept in dedicated registers, updated (like all registers) at context switch.

Larger page tables

Modern systems can have millions of pages.
⇒Move the page table to main memory, addressed by Page Table Base Register (PTBR) which is updated at context switch.

Problem: Each logical memory reference generates two physical references (one for page table, one for real access).

Solution: Can we remember one maybe?
Translation Look-aside Buffer (TLB): fast cache with associative memory. (Keeps only recently used Page Table entries)

Associative memory looks up the key (page number) in parallel with cache - very fast, very expensive.

Very large page tables

two-level page tables

Split the page number part in two:
outer and inner page number/tables.

Each page table will now be 1K (easier to handle)

Other solutions:

- 64-bit architectures require more complexity: multi-level page tables or hashed page tables.
- Inverted page tables

(See section 8.5 page 297)

Protection

Simply add a protection bits to the page table entry.
Typically: Valid/Invalid and Read/Write/Execute

Lookup in parallel: Neglectable overhead

Note that we created a problem due to internal fragmentation

Shared pages

Example

- 40 users, each of whom executes a text editor
- 150 KB of code
- 50 KB of data

$\Rightarrow$ We need 8 000 MB

If one copy of the text editor: Need 2 150 KB
If code is reentrant, it can be shared.

- Non self-modifying code
- code state (variables, temp results) should be local to each process

$\Rightarrow$ shared code pages should not be writable.

Shared pages

Example

Compiler, Window System, run-time libraries, database systems,. . .

Used for shared data (Cf Process Communication)

Segmentation

User preference

View memory as a collection of variable-sized segments, rather than a linear array of bytes

Separate segments for different types of memory content:

- main program
- program libraries
- constants
- stack
- objects
- symbol table
- . . .

and each segment can have its own protection, grow independently, etc...

Segmentation

User preference

Elements are identified within a segment by their offset from the beginning of the segment

Example

- the 1^{st} statement of the program
- the 7^{th} entry of the stack
- the 5^{th} instruction of function foo()
- ...

Segmentation

Segmentation

Memory-management scheme that supports this user view of memory

Logical address space is a collection of segments.

- name
- length

Logical address

Pair: <segment-number, offset>

Note: paging needs one address, implicitly split by hardware

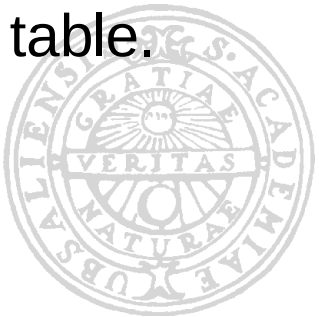
Address translation

User can now refer to objects in the program by a two-dimenseional address

But the actual physical memory is still a one-dimensional sequence of bytes

Must find a mapping from logical to physical

Address translation is done similar to the old memory partitions, but with *base* and *limit* per segment, stored in a segment table.



Characteristics

Segmentation

"paging with variable page size"

Advantages:

- memory protection added to segment table like paging
- sharing of memory similar to paging (but per area rather than per page)

Drawbacks:

- allocation algorithms as for memory partitions
- external fragmentation, back to compaction problem...

Solution: combine segmentation and paging!
Read section 8.7 on Linux & Intel Pentium systems