Memories Are Made of This
Betriebssysteme 2013/2014, P3

Oktober 2013
Outline

1. Design
2. Segmentation
3. Two examples: MULTICS, IA32
4. IA32
So far: Mechanics

- Paging-Principles
- Algorithms
- Modelling

What else do we need for an efficient system?
Process A raises page fault. Which pages to choose from?

<table>
<thead>
<tr>
<th>Page</th>
<th>Age</th>
</tr>
</thead>
<tbody>
<tr>
<td>A0</td>
<td>0A</td>
</tr>
<tr>
<td>A1</td>
<td>07</td>
</tr>
<tr>
<td>A2</td>
<td>05</td>
</tr>
<tr>
<td>A3</td>
<td>04</td>
</tr>
<tr>
<td>A4</td>
<td>06</td>
</tr>
<tr>
<td>A5</td>
<td>03</td>
</tr>
<tr>
<td>B0</td>
<td>09</td>
</tr>
<tr>
<td>B1</td>
<td>04</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Page</th>
<th>Age</th>
</tr>
</thead>
<tbody>
<tr>
<td>B2</td>
<td>06</td>
</tr>
<tr>
<td>B3</td>
<td>02</td>
</tr>
<tr>
<td>B4</td>
<td>05</td>
</tr>
<tr>
<td>B5</td>
<td>06</td>
</tr>
<tr>
<td>B6</td>
<td>1C</td>
</tr>
<tr>
<td>C1</td>
<td>13</td>
</tr>
<tr>
<td>C2</td>
<td>05</td>
</tr>
<tr>
<td>C3</td>
<td>18</td>
</tr>
</tbody>
</table>
**Local**

<table>
<thead>
<tr>
<th>Page</th>
<th>Age</th>
</tr>
</thead>
<tbody>
<tr>
<td>A0</td>
<td>0A</td>
</tr>
<tr>
<td>A1</td>
<td>07</td>
</tr>
<tr>
<td>A2</td>
<td>05</td>
</tr>
<tr>
<td>A3</td>
<td>04</td>
</tr>
<tr>
<td>A4</td>
<td>06</td>
</tr>
<tr>
<td>A5</td>
<td>03</td>
</tr>
<tr>
<td>B0</td>
<td>09</td>
</tr>
<tr>
<td>B1</td>
<td>04</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Page</th>
<th>Age</th>
</tr>
</thead>
<tbody>
<tr>
<td>B2</td>
<td>06</td>
</tr>
<tr>
<td>B3</td>
<td>02</td>
</tr>
<tr>
<td>B4</td>
<td>05</td>
</tr>
<tr>
<td>B5</td>
<td>06</td>
</tr>
<tr>
<td>B6</td>
<td>1C</td>
</tr>
<tr>
<td>C1</td>
<td>13</td>
</tr>
<tr>
<td>C2</td>
<td>05</td>
</tr>
<tr>
<td>C3</td>
<td>18</td>
</tr>
</tbody>
</table>

*Only Pages from the process itself - local*
Local - Global

<table>
<thead>
<tr>
<th>Page</th>
<th>Age</th>
</tr>
</thead>
<tbody>
<tr>
<td>A0</td>
<td>0A</td>
</tr>
<tr>
<td>A1</td>
<td>07</td>
</tr>
<tr>
<td>A2</td>
<td>05</td>
</tr>
<tr>
<td>A3</td>
<td>04</td>
</tr>
<tr>
<td>A4</td>
<td>06</td>
</tr>
<tr>
<td>A5</td>
<td>03</td>
</tr>
<tr>
<td>B0</td>
<td>09</td>
</tr>
<tr>
<td>B1</td>
<td>04</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Page</th>
<th>Age</th>
</tr>
</thead>
<tbody>
<tr>
<td>B2</td>
<td>06</td>
</tr>
<tr>
<td>B3</td>
<td>02</td>
</tr>
<tr>
<td>B4</td>
<td>05</td>
</tr>
<tr>
<td>B5</td>
<td>06</td>
</tr>
<tr>
<td>B6</td>
<td>1C</td>
</tr>
<tr>
<td>C1</td>
<td>13</td>
</tr>
<tr>
<td>C2</td>
<td>05</td>
</tr>
<tr>
<td>C3</td>
<td>18</td>
</tr>
</tbody>
</table>

From all processes - global
Local - Global

- **Local**: Fixed resident size, static distribution
- **Global**: Variable resident set size, dynamic distribution
- **Generally**: Global strategies better, reflect dynamic change of working set
- **Otherwise**:
  - if WS-size rises: thrashing even if pages available „elsewhere“
  - if WS-size shrinks: page frames wasted
Page Replacement

- Some usable for both: FIFO, LRI
- others not: Workingset, WSClock
Global strategies always decide about the number of page frames assigned

- Aging - maybe to coarse
- fairly share memory between processes, adapt regularly (what if processes have different sizes?)
- In relation to process size
  - minimum required (a single instruction may need up to 6 pages)
  - use as starting value, adapt at runtime
Page Fault Frequency Algorithm
Decides on the size of the resident set of a process
based on the fact that page fault rate indirectly proportional to resident set size
- page fault rate too high: assign more memory
- page fault rate too low: reduce memory
Thrashing

- Thrashing: system deals more with page faults and swapping than with work
- WS of all processes > size of RAM: always too many page faults
- PFF can show this: several processes too much RAM; none to much
- solution: reduce degree of multiprogramming (i.e. remove (swap out) processes) until improvement reached
Multiple Address Spaces

- So far: virtual memory one-dimensional, all addresses between 0 and max
- One address space: Needs organisation
  - code, heap, stack
  - multiple stacks, growing heap
- Often separation into multiple address-spaces advantageous
Multiple Address Spaces

- So far: virtual memory one-dimensional, all addresses between 0 and max
- One address space: Needs organisation
  - code, heap, stack
  - multiple stacks, growing heap
- Often separation into multiple address-spaces advantageous
e.g. Compiler

- requires several data areas:
  - source code
  - symbol table
  - constants table
  - structural tree
  - stack

- all of them may grow continuously
e.g. Compiler
Compiler
What could we do if such a case occurs

- terminate compilation - not elegant
- reallocation - requires effort
- multiple address spaces - use segments
Outline

1. Design
2. Segmentation
3. Two examples: MULTICS, IA32
4. IA32
Segmentation

- linear address space (0..max)
- may have different size
- size may change independently to other segments
Segmentation

- Address now has two parts: segment number, address within segment
- Segment is logical unit, programmer defines and uses them
- Protection per segment possible
- Shared libraries: can be put in their own segment - one library, one segment
### Segmentation / Paging

<table>
<thead>
<tr>
<th>Consideration</th>
<th>Paging</th>
<th>Segm.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Programmer needs to be aware of technique</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Number of linear address spaces</td>
<td>1</td>
<td>Many</td>
</tr>
<tr>
<td>Can total address size exceed size of physical memory</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Can procedures and data be distinguished and separately protected</td>
<td>Somewhat</td>
<td>Yes</td>
</tr>
<tr>
<td>Can dynamic tables be accommodated easily</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Is sharing of procedures between users facilitated</td>
<td>Not directly</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Why were these techniques invented

Paging
To get a large linear address space without need to buy more hardware
Why were these techniques invented

**Paging**
To get a large linear address space without need to by more Hardware

**Segmentation**
To allow programs to be broken up into logically independent address spaces and to aid sharing and protection
Implementing Segmentation

- **Important difference:**
  - pages have fixed size
  - segments do not
Segmentation + Paging

- Combine Methods to enjoy both advantages
- First used in MULTICS
- Details see later
Shared Memory

- Multiple processes use same memory
  - because they are running the same code
  - because they want to share some data efficiently
Shared Memory

- Multiple processes use same memory
  - because they are running the same code
  - because they want to share some data efficiently
Alternatively

![Diagram](image.png)
Shared Memory

- Operating System obviously must know about shared memory
- Swap-out of a full process must consider shared memory
- Termination of a process: share memory may still be needed
Copy on Write

- Special kind of shared memory
- After fork
  - two processes, both running the same program and almost same data
  - makes share not to copy everything
- Sharing code: no problem
- Sharing data: is a problem
- How do we know what is data and what is program code?
Copy on Write

- Special kind of shared memory
- After fork
  - two processes, both running the same program and almost same data
  - makes share not to copy everything
- Sharing code: no problem
- Sharing data: is a problem
- How do we know what is data and what is program code?
- We could, from the header files. Does it help?
Copy on Write

- Special kind of shared memory
- After fork
  - two processes, both running the same program and almost same data
  - makes share not to copy everything
- Sharing code: no problem
- Sharing data: is a problem
- How do we know what is data and what is program code?
- We could, from the header files. Does it help?
- Not much.
we just need to be made aware if a process writes to a page and changes the content
reading does not present any problems
how do we know when a process writes to a page?
we just need to be made aware if a process writes to a page and changes the content

reading does not present any problems

how do we know when a process writes to a page? Normally we do not.

But we can prevent it:
Copy on Write

- Set all pages to r/o
- read access: no change

write access:
- page fault
- copy page
- set page to r/w

No need to copy pages that are never changed!
Paging Daemon

- Paging works best when there nearly always are sufficiently many pages free
- Nasty if there is none when I need one
  - have to wait until page is written - takes time
- Preclude it: Paging Daemon
  - mostly inactive
  - wakes up regularly
  - checks state of memory
  - not enough free page frames: start swapping
- Content stays in RAM for the time being - so if page is reaccessed, nothing happened
- nearly always one page free - performance increase
Page Fault Handling

- PF raised by HW
- IP -> Stack
- Jumps into kernel
- Check the faulting address
Page Fault Handling

- Check validity of address
- Look for free page frame (or make one available)
- Save page frame if dirty
- Load page from file
- Adapt page table
- Continue with interrupted machine instruction (when process eventually is rescheduled again)
continue with interrupted machine instruction

- sounds simpler than it might be
- instruction using two addresses

\[
\text{MOV.L } \#6(\text{A1}), \#2(\text{A0})
\]
- loads data of type LONG from \(\text{M[A1+6]}\) auf \(\text{M[A0+2]}\)
- Command needs 6 bytes
- when can a page fault happen here?

---

<table>
<thead>
<tr>
<th>Address</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>1000</td>
</tr>
<tr>
<td>0002</td>
<td>0006</td>
</tr>
<tr>
<td>0004</td>
<td>0002</td>
</tr>
</tbody>
</table>

\[\text{Opcode: } \text{MOVE}\]
\[\text{First operand: } 6\]
\[\text{Second operand: } 2\]
continue with interrupted machine instruction

- when can a page fault happen here?
  - Load Instruction
  - Load first operand
  - Load second operand
  - Read data from source address
  - Write data to destination address

- So we have five possible places for a page fault
- How many page faults could we have in worst case?
continue with interrupted machine instruction

- when can a page fault happen here?
  - Load Instruction
  - Load first operand
  - Load second operand
  - Read data from source address
  - Write data to destination address

- So we have five possible places for a page fault

- How many page faults could we have in worst case? 4. If we assume that no page gets swapped out in the meantime.
continue with interrupted machine instruction

- Now where do we continue?
- Luckily we don’t need to know. We just could repeat the whole process.
- Motorola 68000 and DEC VAX had registers with autoincrement
  - after computing effective address the register was autoincremented
  - good for implementing loops
  - but bad if a page fault occurs after autoincrementing

\texttt{MOVM.L D0-D3/D6/D7,A0-A5,-(SP)}
- This does: D0-> M[SP-4], D1-> M[SP-8], D2-> M[SP-12]....
- Solution: save the processor’s internal state on a page fault and continue with execution of interrupted instruction
Pinning

- a process requests I/O (e.g. `read(FD, bufferm, nrBytes)`)
- process is blocked
- other processes raise page faults
- danger, that the read-destination-page is replaced
- DMA-transfer to wrong location
- Page therefore must be locked for replacement (pinning)
- or transfered via kernel buffers
Swapping

- Swap out - great! But where to?
- Special Swap-area
  - special partition on harddisk
  - special file in filesystem
- Two options
Swapping

- static assignment; simple but restricted
Swapping

- dynamic assignment, more administrative overhead
Outline

1. Design
2. Segmentation
3. Two examples: MULTICS, IA32
4. IA32
MULTICS Memory Structure

- Each program consists of up to $2^{18}$ Segments of 64KW (36-bit-words)
  - Every segment was virtual memory
- Segmenttable had 250,000 entries and was itself a segment (and could therefore be swapped)
- Segmentdescriptor shows if a segment was swapped
- Part of a segment in memory: segment shown as not swapped
- Descriptor contains pointer to segment’s pagetable
MULTICS VM

36 bits

- Segment 6 descriptor
- Segment 5 descriptor
- Segment 4 descriptor
- Segment 3 descriptor
- Segment 2 descriptor
- Segment 1 descriptor
- Segment 0 descriptor

Descriptor segment

Page table for segment 3

- Page 2 entry
- Page 1 entry
- Page 0 entry

Page table for segment 1

- Page 2 entry
- Page 1 entry
- Page 0 entry

Main memory address of the page table

Segment length (in pages)

Page size:
0 = 1024 words
1 = 64 words

0 = segment is paged
1 = segment is not paged

Miscellaneous bits

Protection bits
MULTICS Addresses

Segment number

18

Page number

6

Offset within the page

10

Address within the segment
MULTICS Address Translation

- Load Segment Descriptor
- Pagetable in memory
  - no: segment fault, load page table
- Inspect page table entry
  - eventually raise page fault
- add offset
- access memory
MULTICS Address Translation
for the reading impaired
### Speedup

**TLB with 16 entries**

<table>
<thead>
<tr>
<th>Comparison field</th>
<th>Segment number</th>
<th>Virtual page</th>
<th>Page frame</th>
<th>Protection</th>
<th>Age</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>4</td>
<td>1</td>
<td>7</td>
<td>Read/write</td>
<td>13</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>0</td>
<td>2</td>
<td>Read only</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>3</td>
<td>1</td>
<td>Read/write</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>Execute only</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>2</td>
<td>12</td>
<td>Execute only</td>
<td>9</td>
</tr>
</tbody>
</table>

**Is this entry used?**

0: No, 1: Yes
MULTICS

- developed almost 50 years ago
- many groundbreaking principles
- last system shutdown in 2000
- www.multicians.org
Outline

1. Design

2. Segmentation

3. Two examples: MULTICS, IA32

4. IA32
Features similar to MULTICS
Segmentation and paging
16k segments, each 4GB
Less segments. Size more important than number. Often large segments, rarely very many
LDT - GDT

- Local Descriptor Table LDT
  - for each process
  - local segments (Code, Data, Stack)

- Global Descriptor Table GDT
  - for system segments
  - also kernel
Segment Registers

- 6 segment registers
  - CS: Selector for Code Segment
  - DS: Selector for Data Segment
  - ES: Selector for Data Segment
  - FS: Selector for Data Segment
  - GS: Selector for Data Segment
  - SS: Selector for Stack Segment
Segment Selector

- **RPL**: Requested Privilege Level
- Index 0 forbidden, means: not available
- when a segment register contains a selector, the corresponding descriptor is stored in an internal CPU register
Segment Descriptor

- 0: 16-Bit segment
- 1: 32-Bit segment

- 0: Li is in bytes
- 1: Li is in pages

- 0: Segment is absent from memory
- 1: Segment is present in memory

- Privilege level (0-3)
  - 0: System
  - 1: Application

- Segment type and protection

- Base 24-31
- G D 0
- Limit 16-19
- P DPL S Type
- Base 16-23

- Base 0-15
- Limit 0-15

- 32 Bits
- Relative address
CPL, DPL, RPL

- CPL: current privilege level - in which ring are we running
- DPL: privilege level of a segment or gate
  - indicates the highest level the a program can have to be allowed to access the page
  - generally: $\text{DPL} = 1 \Rightarrow$ only code with $\text{CPL} = 0$ or $\text{CPL} = 1$ may access the page
- RPL: override privilege, if RPL is greater than CPL, RPL is used
  - i.e. $\text{DPL} = 1 \Rightarrow$ only code with $\max(\text{RPL}, \text{CPL}) = 0$ or $\max(\text{RPL}, \text{CPL}) = 1$ may access the page
  - can ensure that privileged code cannot access a segment on behalf of an application program unless that program itself would have access
Address translation

- we start with (selector, offset)
- microprogram looks for correct descriptor in internal registers
- selector 0 or segment swapped out: interrupt
- offset exceeds segment size: interrupt
- add base field to offset
  - check limits of course
- result: linear address
- paging turned off: linear address is physical address
Address translation
for the reading impaired

Selector

Descriptor

Base address

Limit

Other fields

Offset

32-Bit linear address

+
Address translation

- Paging enabled: linear address is virtual address
- map to physical page using well known paging mechanism
- fixed page size 4KB
- two level paging
- page directory has 1024 entries (PDE)
- TLB exists
- one selector in all segment registers: normal paging
  - the case in all common OSs
  - OS/2 was different. R.I.P.

Memories Are Made of This  
Oktober 2013