Chapter 6, The Operating System Machine Level

Level 5  Problem-oriented language level
         Translation (compiler)

Level 4  Assembly language level
         Translation (assembler)

Level 3  Operating system machine level
         Partial interpretation (operating system)

Level 2  Instruction set architecture level
         Interpretation (microprogram) or direct execution

Level 1  Microarchitecture level
         Hardware

Level 0  Digital logic level

6.1 Virtual Memory
6.2 Virtual I/O Instructions
6.3 Virtual Instructions For Parallel Processing
6.4 Example Operating Systems
6.5 Summary
The Operating System Machine Level

The operating system, from the programmer’s point of view, adds a variety of new instructions and features, above and beyond what the ISA level provides.

In many systems, the OS can be “defeated” and these operations may be “programmed” at the machine level (in hardware or ISA software), but it isn’t usually a good idea … leave it to the experts … that is, become an expert and/or only do it when there is no other way!

AST defines the OSM level as the “complete set” of instructions available to the application programmer.

It contains nearly all the ISA level instructions and new instructions for “system calls”.

A system call invokes a predefined operating system service. Typical systems calls include file read and write operations. The OSM level provides interpreted instructions for execution on the underlying machine.

The important concepts that will be covered are:

- Virtual Memory
- File I/O
- Parallel Processing

Figure 6-1. Positioning of the operating system machine level.
Virtual Memory

Another aspect of the memory subsystem in a computer

H&J slide 7-4

Memory System Design

- The memory hierarchy: from fast and expensive to slow and cheap
  - Example: Registers → Cache → Main Memory → Disk
  - At first, consider just two adjacent levels in the hierarchy
  - The cache: High speed and expensive
    - Kinds: Direct mapped, associative, set associative
  - Virtual memory—makes the hierarchy transparent
    - Translate the address from CPU’s logical address to the physical address where the information is actually stored
    - Memory management—how to move information back and forth
    - Multiprogramming—what to do while we wait
    - The “TLB” helps in speeding the address translation process
- Overall consideration of the memory as a subsystem
**Historical developments:**

1961 Manchester, England research proposed a method for *automatic overlay processing*. This effectively *separated the address space from the absolute memory location*.

Originally memory was severely limited:
Code was written “to fit” memory instead of being “fast” or “complete”.

If only the code that was really required could be loaded into the main memory, programs could execute more efficiently … but those code segments may not be contiguous.

Therefore, *divide the code into pieces and only load the pieces that are needed* … but what about addressing …

If the programs “*virtual address*” could be translated to the memory systems “*physical address*” everything could work out!

What is going to manage this “virtual-physical” translation … the operating system!

---

**Virtual Memory**

The memory management unit, MMU, is responsible for mapping logical addresses issued by the CPU to physical addresses that are presented to the cache and main memory.

![Virtual Memory Diagram]

**A word about addresses:**

- **Effective address**—an address computed by the processor while executing a program. Synonymous with *logical address*.
  - The term effective address is often used when referring to activity inside the CPU. Logical address is most often used when referring to addresses when viewed from outside the CPU.

- **Virtual address**—the address generated from the logical address by the memory management unit, MMU.

- **Physical address**—the address presented to the memory unit.

(Note: *Every* address reference must be translated.)
Concepts:

- Only load memory with what is currently needed.
- Divided the program into pieces, called overlays, that fit in memory.
- Use the overlays one at a time until done … the concept of a working set of memory.
- Allow programmers to manage the process (now done by OS software)

Virtual Addresses—Why

The logical address provided by the CPU is translated to a virtual address by the MMU. Often the virtual address space is larger than the logical address, allowing program units to be mapped to a much larger virtual address space.

Getting Specific: The PowerPC 601

- The PowerPC 601 CPU generates 32-bit logical addresses.
- The MMU translates these to 52-bit virtual addresses before the final translation to physical addresses.
- Thus while each process is limited to 32 bits, the main memory can contain many of these processes.
- Other members of the PPC family will have different logical and virtual address spaces, to fit the needs of various members of the processor family.
Virtual Addressing—Advantages

- **Simplified addressing.** Each program unit can be compiled into its own memory space, beginning at address 0 and potentially extending far beyond the amount of physical memory present in the system.
  - No address relocation required at load time.
  - No need to fragment the program to accommodate memory limitations.
- **Cost effective use of physical memory.**
  - Less expensive secondary (disk) storage can replace primary storage. (The MMU will bring portions of the program into physical memory as required)
- **Access control.** As each memory reference is translated, it can be simultaneously checked for read, write, and execute privileges.
  - This allows access/security control at the most fundamental levels.
  - Can be used to prevent buggy programs and intruders from causing damage to other users or the system.

This is the origin of those “bus error” and “segmentation fault” messages.
Question: How could we execute a program requiring 64k memory on a processor with only a 4k memory

Execute various program overlays into the main memory. When they are done, load the next overlay and continue.

Whenever an address is out-of-range, execute a trap (fault):

(1) Error Trap – out-of-range access and give up (programmer’s mistake)

(2) Page Trap – page faults

- Save contents of memory
- locate appropriate overlay
- load the overlay
- “map” the addresses
- continue execution

Note: Memory can either be moved as pages (defined size) or segments (entire defined region)

CPU Design Changes:

Software execute “virtually addressed” code

Memory Management translates “virtual address” to “physical address” for hardware
A mapping process is needed between virtual and physical addresses

- must keep track of where virtual program is located
- must “catch” the virtual address and determine how best to proceed
- must map virtual to physical address when data are in the main memory
- must halt, fetch required memory, update the memory map, and continue when the data is not in main memory

The virtual memory operations must be transparent to the programmer/user!!

H&J Slides Virtual Memory Recap … (7-45 and 7-42, Previously shown 7-61, 7-62, & 7-63)

**Virtual Memory**

A virtual memory is a memory hierarchy, usually consisting of at least main memory and disk, in which the processor issues all memory references as effective addresses in a flat address space. All translations to primary and secondary addresses are handled transparently to the process making the address reference, thus providing the illusion of a flat address space.

Recall that disk accesses may require 100,000 clock cycles to complete, due to the slow access time of the disk subsystem. Once the processor has, through mediation of the operating system, made the proper request to the disk subsystem, it is available for other tasks.

Multiprogramming shares the processor among independent programs that are resident in main memory and thus available for execution.
Virtual Memory General Types:

Paged Virtual Memory
- A program is treated as a sequential set of code addressed from 0 to N-1.
- Memory is defined as fixed length pages for mappings.
- Typical page sizes are from 512 Bytes to 64 kBytes, with 2 or 4 kBytes popular.
- The last page is not “full”.

Segmented Virtual Memory
- A program is treated as multiple code segments (objects), each addressed from 0.
- Code segments may be any size.
- Code segments are loaded into main memory when needed.
- Reorganizing memory allocation in the main memory may be required occasionally.
Implementation of Paging (back to AST)

1. The original/reference program resides on a hard disk

2. Physical memory holds copies of the reference program and data
   - If and when changes are made they must be stored (overwrite original data or be placed in a scratch space)
   - This process is similar to a cache … but memory pages are used instead of cache lines

Definitions used:
- Virtual Memory Program → Pages
- Physical Memory → Page Frames (frames holding the program memory page)

![Diagram](image)

Figure 6-4. Formation of a main memory address from a virtual address.
Lookup Table Size:

- **32-bit** processor address → 4 GB
- **128 MBytes** RAM PC → 27 bit total
- **Page Size 32 kBytes** → 15 bits

Therefore:

- \(32 - 15 = 17\)-bit virtual page TAG
- \(27 - 15 = 12\)-bit physical page TAG

**Build a page table of \(2^{17}\) locations with 12-bit physical page address and status.**

**Note:** the page table may be located in the main memory used by the OS to hold mapping information.

---

**Figure 6-5.** A possible mapping of the first 16 virtual pages onto a main memory with eight page frames.
More advanced circuitry: H&J 7-71 and 7-68

**Fig 7.41**
**Memory Management by Paging**

- This figure shows the mapping between virtual memory pages, physical memory pages, and pages in secondary memory. Page n - 1 is not present in physical memory, but only in secondary memory.
- The MMU manages this mapping.
Fig 7.42
Virtual Address Translation in a Paged MMU

- 1 table per user per program unit
- One translation per memory access
- Potentially large page table

A page fault will result in 100,000 or more cycles passing before the page has been brought from secondary storage to MM.
Fast Address Translation: Regaining Lost Ground

- The concept of virtual memory is very attractive, but leads to considerable overhead:
  - There must be a translation for every memory reference.
  - There must be two memory references for every program reference:
    - One to retrieve the page table entry,
    - one to retrieve the value.
  - Most caches are addressed by physical address, so there must be a virtual to physical translation before the cache can be accessed.

The answer: a small cache in the processor that retains the last few virtual to physical translations: a Translation Lookaside Buffer, TLB.

The TLB contains not only the virtual to physical translations, but also the valid, dirty, and protection bits, so a TLB hit allows the processor to access physical memory directly.

The TLB is usually implemented as a fully associative cache:

TLBs have appeared in Intel and SPARC block diagrams previously shown.

TLBs are limited in size … 64 entries for example.
Fig 7.43 Translation Lookaside Buffer Structure and Operation

Main memory or cache

Virtual address from CPU

Desired word

Physical address

Page number  Word

Physical page  Word

TLB

TLB hit, Page is in primary memory.

Hit  N

TLB miss. Look for physical page in page table.

To page table

Associative lookup of virtual page number in TLB

Virtual page number

Access control bits: presence bit, dirty bit, valid bit, usage bits

Physical page number

**Definitions:**

Pages in main memory form the **working set**.

**Demand Paging** is when pages are only loaded when there is a demand for the addresses 
(demand paging builds the working set without loading “active working sets”)

A **Page Replacement Policy** is required (the same ones defined for cache replacement)

- Least Recently Used
- FIFO
- Etc.

**Page Size Considerations**

- Each program has multiple pages
- The last page is not full – **internal fragmentation**
- Data array organization may be a problem (m x n, complex data, etc.)
A problem with paged memory operations:

When multiple tables are placed in a single virtual address space,
- When tables grow, they may start to overlap
- Example of code compilation: symbol table, source text, constant table, parse tree for syntactic analysis, and stack.

An alternate approach, what if every table had its own virtual memory space that was allowed to grow without bounds … provide multiple memory segments for each one ….
**Segmented Virtual Memory**

Programs are divided into multiple segments that are individually grouped and addressed.

![Segmented Virtual Memory Diagram](image)

Allows simple alignment of multiple regions.

**Example: Compiler Operations**

Symbol table, source text, constant table, parsing tree, call stack

Note: segments are allowed to grow without “interfering” with other segments

If procedures are segmented, a change in procedure size will not affect existing code or procedures.

<table>
<thead>
<tr>
<th>Consideration</th>
<th>Paging</th>
<th>Segmentation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Need the programmer be aware of it?</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>How many linear addresses spaces are there?</td>
<td>1</td>
<td>Many</td>
</tr>
<tr>
<td>Can virtual address space exceed memory size?</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Can variable-sized tables be handled easily?</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Why was the technique invented?</td>
<td>To simulate large memories</td>
<td>To provide multiple address spaces</td>
</tr>
</tbody>
</table>

**Figure 6-9.** Comparison of paging and segmentation.
Fig 7.38
Memory Management by Segmentation

- Notice that each segment’s virtual address starts at 0, different from its physical address.
- Repeated movement of segments into and out of physical memory will result in gaps between segments. This is called external fragmentation.
- Compaction routines must be occasionally run to remove these fragments.
Fig 7.39
Segmentation Mechanism

- The computation of physical address from virtual address requires an integer addition for each memory reference, and a comparison if segment limits are checked.
- Q: How does the MMU switch references from one segment to another?
Fig 7.40  The Intel 8086 Segmentation Scheme

The first popular 16-bit processor, the Intel 8086 had a primitive segmentation scheme to “stretch” its 16-bit logical address to a 20-bit physical address:

The CPU allows 4 simultaneously active segments, CODE, DATA, STACK, and EXTRA. There are 4 16-bit segment base registers.
A problem with segmented virtual memory

![Diagram of memory segments](image)

**Figure 6-10.** (a)–(d) Development of external fragmentation
(c) Removal of the external fragmentation by compaction.

**External fragmentation** or checkerboarding

Attempts to reduce memory compaction:
- Best fit algorithm – smallest hole that will work
- First fit algorithm – first hole in list big enough to fit
Segmented-Paged Virtual Memory

**Figure 6-11.** Conversion of a two-part MULTICS address into a main memory address.

MULTICS: Multiplexed Information and Computing Service

Descriptor segment for each process, defines segments to be used.

Descriptor points to a page table allowing the segments to be paged.
Intel® 64 and IA-32 Architectures Software Developer’s Manual
Volume 3 (3A, 3B & 3C): System Programming Guide

See Chapter 3: PROTECTED-MODE MEMORY MANAGEMENT

The following information and section of the textbook is based on the Intel manual ….
Supports: demand paging, pure segmentation and segmentation with paging!

GDT: Global Descriptor Table
system level segment identification (only one)

LDT: Local Descriptor Table
program level segment identification (one for each program)

Descriptors allow Privilege Level operation and identification of further descriptors to be used. With 13-bits, 8k segment descriptors can be identified. Index 0 is invalid and will cause a trap.

![Diagram of selector](image)

**Figure 6-12.** A Pentium II selector.

During execution, when a selector is loaded, the corresponding segment descriptor is fetched and loaded to descriptor registers resident in the MMU functional area.

![Diagram of segment descriptor](image)

**Figure 6-13.** A Pentium II code segment descriptor. Data segments differ slightly.
The application of the selector and descriptor to create an address is shown below.

![Diagram](image)

**Figure 6-14.** Conversion of a (selector, offset) pair to a linear address.

If paging is enabled, the linear address is interpreted as a virtual address and mapped onto the physical address space using page tables.

For a 32-bit address and using a 4 kB page, a segment might contain 1 million pages …. Therefore, a two level mapping may be used as

![Diagram](image)

**Figure 6-15.** Mapping of a linear address onto a physical address.
Intel IA-32 protection related to virtual memory.

The segments belong to specific levels … as long as the level is maintained, everything is fine. Attempts to access data at higher levels is permitted, while lower level are illegal and cause traps to be executed (protection faults).

![Diagram](image-url)

**Figure 6-16.** Protection on the Pentium II.
Virtual Memory on the UltraSPARC II/III (SPARC V9)

Maximum physical memory is 41 bits … 4 page sizes are supported as shown …

64-bit virtual space with 44-bits supported.
- One 43-bit segment at the lowest addresses of the virtual memory and
- One 43-bit segment at the highest addresses of the virtual memory

Figure 6-17. Virtual to physical mappings on the UltraSPARC.

Figure 6-18. Data structures used in translating virtual addresses on the UltraSPARC. (a) TLB. (b) TSB. (c) Translation table.

TSB is a translation storage buffer: a software cache of heavily used TLB entries.

Power PC Virtual to Physical Address Translation

Fig 7.45
PowerPC 601 MMU Operation

"Segments" are actually more akin to large (256 MB) blocks.
Memory Hierarchy Accessing Summary

Fig 7.44  Operation of the Memory Hierarchy