Operating Systems Principles

Design and Implementation
Policies and Mechanisms

Textbook: L. Bic and A. C. Shaw: "Operating Systems Principles"

Other references:
- Other references - see web site: cs.nmt.edu/~cs325
Simple hardware configuration

Typical hardware configuration
This chart plots relative performance as measured by the SPECint benchmarks with base of one being a VAX 11/780.

Memory cost

Prices of DRAMS (from 16K bits to 64M bits) over time in 1977 dollars.
Principle of abstraction

- **Higher-level function:**
  - `Read( file, logical_block, main_memory )`

- **Lower-level functions**
  - Calculate position of `logical_block` on the disk;
  - Move Read/Write head to corresponding track;
  - Check for "seek" errors;
  - Read physical block;
  - Check for "read" errors;
  - Copy block to `main_memory`;
Principle of abstraction

```
read
Calc_pos     Mov_RWhd
ReadPhysBlk  CopyBlkToMem
ChckReadErr  ChckSeekErr
```

driver

HW

"Logical Design"

- Logical -> "virtual" machine
- Real -> "physical" machine

- Design principles -
  - "policy", strategy – plan of action --- design
  - "mechanism", tactics, approach --- implementation

- Example: waiting queue
  - Logical organization: FIFO, priority, some scheduling scheme
  - Physical organization: linked list, doubly linked list, circular list, array, heap, ...
"Timesharing" - time slicing

Real and virtual memory

logical (virtual)  real (physical)

mapping
Real and virtual memory

**logical (virtual)**

The ideal situation: each process (executing program) has its own address space ranging from 0 to a max_size. No access from/to other address spaces is allowed.

Special hardware support (paging, base and limit registers, address mapping hardware (MMU) alleviate the task of the operating system.

**real (physical)**

The real situation: each memory system is composed of a hierarchy of storage components (cache, fast and big memory…). The total storage is divided into segments, pages and access/transfer pieces – proper access and protection is a serious problem.

Virtual, logical und physical devices

![virtual device diagram](virtual_device.png)

- log. device 1
  - Driver_1
  - phys. device 1
- log. device 2
  - Driver_2
  - phys. device 2
**interfaces and Virtual machines**

Operating System

```
kernel
```

virtual machine = logical machine + manager

```
logical machine = physical machine + driver
```

**Virtual Machines**

<table>
<thead>
<tr>
<th>Processes</th>
<th>Processes</th>
<th>Processes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kernel_1</td>
<td>Kernel_2</td>
<td>Kernel_3</td>
</tr>
</tbody>
</table>

Virtual Machine

Hardware
**Example Virtual Machine**

Processes | Processes | Processes
--- | --- | ---
OS_1 | OS_2 | OS_3

Virtual Machine (VM/370)

Hardware

---

**System Configuration**

- Secondary storage
- Printer
- Tape/C
- Terminal
- Controller
- Networking interface
- CPU
- Memory

---
• Monolithic systems: one big collection of routines, all run in privileged mode;
• Structured systems: higher-level functions run in user mode, lower-level functions are combined into a privileged kernel;
• Micro-kernel systems: only the hardware-drivers and a few critical components (thread management, inter-process communication, synchronization) run in kernel mode - everything else runs in user mode.
Software structure

User 1  

(Graphical) User Interface

Applic.-progr. 1  utility-program

Operating System Functions

Operating System Kernel

Hardware

User 2  

Applic.-progr. n

Utility-program

SW-tool

System Calls
POSIX

VM

von Neumann Architecture

Memory contains:
- data and
- programs

Address of next instruction

Control unit

ALU

Data (results)
data

Instructions (commands)

control-signals

This principle was invented by J. P. Eckert and J. W. Mauchly but published first as an abstract concept by John von Neumann. Similar ideas were developed by Konrad Zuse and Schreyer.
Fetch-Execute Cycle

REPEAT
IR := Mp[PC];
PC := PC+1
IF (FC == JpClass) THEN
   PC := Operand
ELSE
   Execute (IR)
END_IF;
UNTIL cpu_HALT;

von Neumann Architecture (SISD)

memory Mp contains:
• data and
• programs

Program Counter PC

instruction register IR

ALU

function code

modifier Address, direct

Operands
Interrupt Mechanism

IF (IRQ_i == 1) THEN
  save PC;  PC := IRLoc_i
ELSE
  IR := Mp[PC]; PC := PC+1
  IF (FC == JpClass) THEN PC := Operand
  ELSE  Execute (IR);
  END_IF
END_IF  // Interrupt

„basic instruction cycle with interrupts“

This transition is triggered by the interrupt nn_i

This location is saved by the interrupt hardware or the handler software.

This transition is executed by the interrupt handler (driver) command “RTI”
**Software Structure**

- User 1
- User 2
- User n

- (Graphical) User Interface – "shell"

- System Calls (SVC)

- Operating System
  - Functions and Services
  - Operating System kernel

- Hardware

**OS Functions**

- Process management
- Memory management
- File management
- Input/Output
- Networking
- Utility services
**Simple batch system - memory allocation**

Free space

- data

User program

- code

Operating system

**Bootstrap Loader**

Task: load a small initial program into memory and start it.

This process is also called „deadstart“, „coldstart“, „bootstrap“; it must be supported by a hardware “initial load” function.

```
Press_Load:
    FOR i := 0 TO 79 DO
        Read(M[i]);
    END_FOR;
    GOTO (M[0]);
```
Bootstrap

In a second/third phase the minimal program can read and subsequently start a bigger program - e.g. a more powerful loader or the kernel of an executive system (Exec, Monitor program, OS).

<table>
<thead>
<tr>
<th>LA</th>
<th>n</th>
<th>Absolute Code (Program/Data)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>79</td>
<td>Absolute Code (Program/Data)</td>
</tr>
</tbody>
</table>

```
LA := M[r];
n := M[r+1];
IF n==0 THEN GOTO LA;
FOR i := 0 TO n-1 DO
   M[LA+i] := M[r+2+i] END_FOR;
END_LOOP;
```

Entry_Address = start of execution ( PC := E_A; go! )
Bootstrap Loader

3 loading stages

1. Bootblock loads Boot program

2. Boot program loads Master boot record & partition table

3. Boot program loads operating system

Bootstrap Loader II

Initially a HW program (ROM) reads the first sector of the boot disk with the "Partition Table"; then it reads the bootblock of the active partition.

All other steps are identical to the simple boot process.
Early Programming

- Use of the "bare machine"
- Certain instruction sequences which appear frequently are arranged into subroutines
- Subroutines which are commonly used are organized into a library
  for application programming a symbolic notation is introduced -
  > Assembler FC „64“ = „jmp“
- Address management is alleviated by introducing symbolic names and automatic relocation at translation and loading time

Early Programming

- Binding (linkage edit) and Loading are delegated to special programs. (linkage editor, loader)
- Execution is supervised and controlled by a "Monitor" program (Exec).
- Additional service programs (utilities) are provided: editor, compiler, SortMerge programs
- Control cards and a "Job Control Language" (JCL) permit user control of "job" execution
- All systems and utility programs comprise the operating software; the machine-oriented programs lower layers form the operating system proper.
Betriebssysteme

**Betriebssysteme ws2000/2001**

Prof. H. D. Clausen

---

**Multiprogramming systeme - memory allocation**

```
<table>
<thead>
<tr>
<th>user job_n</th>
</tr>
</thead>
<tbody>
<tr>
<td>User job_2</td>
</tr>
<tr>
<td>user job_1</td>
</tr>
<tr>
<td>operating system</td>
</tr>
</tbody>
</table>
```

Jan 26
### CPU and I/O Activities

<table>
<thead>
<tr>
<th>Zeit</th>
<th>Aktivität</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CPU</td>
</tr>
<tr>
<td></td>
<td>I/O</td>
</tr>
<tr>
<td></td>
<td>Interrupt</td>
</tr>
<tr>
<td></td>
<td>Interrupt</td>
</tr>
</tbody>
</table>

### I/O Overlap (Multiprogramming / Multitasking)

- **Process 1**
- **Process 2**
**Runtime Organization - Service Invocation**

- **Subroutine/Coroutine**
  - Single thread of execution

- **Message Passing**
  - Multiple threads
  - Logical / virtual
  - Physical / real

---

**Memory Protection**

- **job_1**
  - Basisregister

- **job_2**
  - Limitregister

- **job_n**

- **Operating system**
Protection - Kernel/User Mode

- \texttt{job_1}
- \texttt{job_2}
- \texttt{job_n}

Types of Operating Systems

- simple Batch Systems
- Timesharing Systems
- Workstation / PC Systems
- Multiprocessor-/Parallel-systems
- Real Time Systems
- Process Control Systems
- Embedded Systems
- Signal processing systems
- Distributed Systems
- Networked Systems
**Monolithic System**

- **User programs**: $P_1 \rightarrow \rightarrow P_n$
- **Operating System**
- **HW System**

**Process-Oriented System**

- **User process**: $u_i$
- **System process**: $s_j$
- **Operating System Kernel**
- **HW System**

- **Processes**:
  - Call; Branch; Return;
  - Proc-Call; SVC; Send/Receive;

- **Communication**

- **Interrupts**: 
  - From User to Kernel
  - From System to Kernel

Prof. H. D. Clausen
**Object-Oriented System**

- User object: $o_i$
- System object: $o_j$
- Capability
- Kernel Call: Return;
- Interrupt:

**Operating System Kernel**

**Future?**

- “current efforts to secure the PC’s traditionally open architecture will either lead
  - to an unwieldy system that tries to employ impenetrable membranes, encrypted buses and tamper resistant memory, or
  - To an inferior solution that fails to thwart dishonest users and limits backup and interoperability “
- “investing in proven architectural improvements such as guarded pointers and data tags (capabilities) is a more cost effective and long overdue alternative”
Yet another quote…

“The world of software production has dramatically changed during the last decades from pure assembler programming to procedural programming to object-oriented programming. Each step raised the level of abstraction and increased programmer productivity. Operating systems, on the other hand, remained largely unaffected by this process.

… there is a growing divergence between application programming and operating system programming. To close this semantic gap between the applications and the OS interface a large market of middleware systems has emerged over the last years.

While these systems hide the ancient nature of operating systems, they introduce many layers of indirection with several performance problems.

While previous object-oriented operating systems demonstrated that it is possible and beneficial to use object-orientation, they also made it apparent that it is a problem when implementation technology (object orientation) and protection mechanism (address spaces) mismatch.”

Not a pretty picture ...

Diagram:
- Application program
- Application program
- API
- Middleware
- Operating System
- Hardware
**CPU und I/O activity - (a) instruction waits**

![Diagram showing CPU and I/O activity with instruction waits]

**I/O and CPU Overlapping**

![Diagram showing I/O and CPU overlapping with command and interrupt]

---

**Prof. H. D. Clausen**

---

**Betriebssysteme**
**CPU und I/O activity - (b) parallel I/O with polling**

![Diagram of parallel I/O with polling](image)

**CPU und I/O activity - (c) device interrupts**

![Diagram of device interrupts](image)
The basic structure of protection rings (Multics)

XINU

Is Not Unix!

Figure 4.1. The layering of components in Xina

Douglas Comer: Operating System Design – the XINU Approach (LSI-11, Pentium)
Design Principles

- Functionality vs. form
- components - composition - aggregation
- Concentrate on basic design elements and functions (the logical level, not implementation aspects ??)
- Follow the KISS principle: (keep it simple, stupid!)
- Aside: implementation principles:
  - Use HL implementation languages!
  - Compilers need to efficiently exploit the HW (JIT)
  - Exploit efficient basic methods, e.g.
    * Fast context switching (address space)
    * Highly efficient IPC
    * Efficient process/thread switching and RPC

Layered Structure
Layered Systems

- Layer - a collection of functions which are closely coupled and at the same hierarchical level
- hierarchal level - corresponds to a certain level of abstraction
- abstraction - extraction of the essential (logical) characteristic features of a real (physical) machine and definition of an abstract (logical) machine
- The services are made available by “system calls” (SVCs) in form of standardized functions at APIs
  [API - application program interface, SAP - service access point]

“virtualization”

Layer 1
- e.g. Pentium

Layer 2
- Java-VM

Layer 3
- Java

Processing - step

Processing - step

Processing - step

time
HAL and virtual machines

- A Hardware Abstraction Layer (HAL) is an attempt to map the physical characteristics of hardware onto a more abstract and general logical machine.
- A virtual machine offers to the user a certain interface which generates the illusion of a "real" machine.
- The different layers of an operating system can be seen as virtual machines of increasingly higher abstraction.
- Different virtual machines can coexist in one real system at the same level and in multiprogramming mode.
- The "Java VM" supports specifically the concepts and structure of the high-level programming system Java; this (should) guarantee the portability of applications (and late binding/dynamic loading as well as JIT compiling).

Goals

- Design goals - "elegant and clean"
- Implementation goals - "efficient and compact"

- 2 competing views:
  - Top-down – goal oriented (deductive)
  - Bottom-up – tool oriented (inductive)
- Common goal: both approaches concentrate on the kernel
  - Micro – kernel
  - Exo – kernel
- *** the kernel design and implementation is the *** real problem
The common elements are:

- Improved protection mechanisms to support “trusted systems”
- Highly efficient set of basic elements (functions, primitives, objects)
- Extensibility i.e. the capability to dynamically load additional OS components such as e.g. drivers in a “safe and secure” mode

Mach

- Micro-kernel: Scheduler, Interprocess Communic. Basic I/O
- Hardware
- Memory Management
- Terminal I/O
- File System

User 1

User 2

Interface to system calls and functions

User mode

Kernel mode
Windows NT Struktur

Logon proc. security Subsyst.

OS/2 applic. OS/2 Subsyst.

Win16 applic. Win16 VDM

WIN32 applic. MSDOS applic. MSDOS VDM

POSIX applic. POSIX Subsyst.

Win32 Subsyst.

Executive - Systemdienste

I/O Manager

Object Manager Process Manager Memory Manager Security Manager Local Proc. Calls

(Mikro) Kernel

Hardware Abstraction Layer

Hardware

I/O Manager

user mode

kernel mode