09 - Protection in General-Purpose Operating Systems

Acknowledgments to Dr. Ola Flygt of Växjö University, Sweden for the original slides.
History of protection in OSs

- Predecessors of OSs:
  - No system s/w
    - User entered program in binary
    - Via switches or via keyboard
    - Single user had full control of computer
      - Scheduled time for exclusive computer use
      - Prepare before use
        - Load assembler, compiler, shared subroutines, etc.
        - Clean up after use
  - Executive
    - Assist single user with preparation and cleanup
    - Entirely passive:
      - Waited for user's request
      - Provided service on demand
History of protection in OSs (2)

- Monitor
  - Assisted multiple users in multiprogramming systems
  - Actively controlled system resources
    - Provided service if consistent with system policies,
    - denying otherwise
  - Protect one user from interference (malicious or accidental or malicious) by another

- Impact of multiprogramming on security:
  - Before multiprogramming - no need to protect one user from another
  - With multiprogramming - need to
Protected objects in OSs

- Multiprogramming — sharing processor among multiple processes
- Multiprogramming necessitates protecting OS objects:
  - Memory
  - I/O Devices
    - Sharable I/O devices (e.g., disks)
    - Serially reusable I/O devices (e.g., printers)
  - Sharable programs and subroutines
  - Networks
  - Sharable data
- Since OS controls system resources, OS must provide such protection
  - Many protection mechanisms supported by hardware
Security goals of Operating Systems

- Enable multiple users to securely share a computer
  - Separation and sharing of processes, memory, files, devices, etc.

- Ensure secure operation in networked environments
  - Enforce security policies while allowing resource sharing across multiple computers
How to achieve the security goals

- Classic OS protections
  - Memory and address protection
  - Control of access to general objects
  - File protection
  - Authentication

- Additional OS protections
  - Logging & Auditing
  - Intrusion Detection
  - Recovery
  - Many others...
CPU modes

- Processor modes, privileges
- *System mode (privileged mode, master mode, kernel mode)*
- *User mode*

- Transition from user mode to system mode must be done through well defined call gates (system calls)
System calls

- Guarded gates from user mode into kernel mode
  - Transfer control to predefined entry point in more privileged code, using a special CPU instruction (often an interrupt)
  - Allow the more privileged code to specify where it will be entered as well as
    - Save processor state at the time of entry
  - The higher privileged code, by examining processor state set by the less privileged code and/or its stack, determines what is being requested and whether to allow it.
Security methods in OSs

- **Basis of security in OS**: *separation*
  - Keeping one user’s objects secure from interference by other users

- **Kinds of separation**:
  - **Physical separation**
    - Different processes use different physical objects
      - e.g., different printers for different ‘confidentiality levels’ of output
  - **Temporal separation**
    - Processes having different security requirements executed at different times
  - **Logical separation**
    - Illusion that OS executes processes only for single user
  - **Cryptographic separation**
    - Processes conceal their data and computations from other processes
  - Combinations of the above
Security methods in OSs

- Strength of security via separation (least to most secure):
  - Logical separation
  - Temporal separation
  - Physical separation

- Complexity of implementation of separation (least to most complex):
  - Physical separation
  - Temporal separation
  - Logical separation
  - Cryptographic separation

- Resource utilization in different kinds of separation:
  - Poor: physical separation / temporal separation
  - Good: logical separation / cryptographic separation
Levels of protection in OSs

- Absolute separation reduces efficiency
  - need to share some resources for efficiency
- Full sharing-separation spectrum = levels of protection by OS:
  - No protection
    - Caveat emptor (“Let the buyer beware” in Latin)
    - User can still protect self by, e.g., temporal separation
  - Isolation
    - Concurrently running processes hidden from each other
      => unaware of each other
      - Own address space, files, other objects for each process
  - Full sharing or no sharing
    - Object/resource owner declares it as:
      - public (can be shared by all)
    Or
      - private (not shared)
Levels of protection in OSs

- Sharing via access limitation
  - Access to each object by each user determined by access rights
- Sharing by capabilities
  - Extension to “sharing via access limitation”
    - dynamic access rights
    - Can be changed by owner, subject, computation context, object itself
- Limited object use
  - Limits not only object access — limit object use
    - e.g. can view a doc but can’t copy it
    - e.g. can view statistical summary of data but can’t view individual data records (e.g. can see average salary but not John Smith’s salary)
Levels of protection in Oss

- OS can provide different levels of protection for different objects/resources
- Complexity of implementation and fineness of protection:
  - 1) No protection
  - 2) Isolation
  - 3) Full sharing or no sharing
  - 4) Sharing via access limitation
  - 5) Sharing by capabilities
  - 6) Limited object use
Three dimensions of protection in OS

[cf. B. Endicott-Popovsky and D. Frincke]
Granularity of data protection

- Granularity of data protection
  - Applicable only to data
  - Protect by:
    - Bit
    - Byte
    - Element/word
    - Field
    - Record
    - File
    - Volume

| Ease of implementation | Worse (higher granularity) data control (*) |

(*) If no control at proper granularity level, OS must grant access to more data than necessary e.g., if no field-level data control, user must be given whole record.
Memory and Address Protection

- Most obvious protection
  - Protect program memory from being affected by other programs
Memory and Address Protection

- **Fence**
  - Confining users to one side of a boundary
  - e.g., predefined memory address \( n \) between OS and user
    - User program instruction at address \( \leq n \) (OS’s side of the fence) not allowed to execute

![Figure 4-1 Fixed Fence.](image)
Figure 4-2 Variable Fence Register.
Memory and Address Protection

- **Relocation**
  - Programs written as if starting at location 0 in memory
  - Actually, starting at location n — determined by OS
  - Before user instruction executed, each address relocated by adding relocation factor n to it
    - Relocation factor = starting address of program in memory
  - Fence register (h/w register) plays role of relocation register as well
    - Adding n to program address prevents it from accessing any address below n
Figure 4-3 Pair of Base/Bounds Registers.

Figure 4-4 Two Pairs of Base/Bounds Registers.
Memory and Address Protection

- **Base/Bounds Register**
  - Base register = variable fence register
    - Determines starting address, i.e. lower limit, for user program addresses
  - Bounds register
    - Determines upper limit for user program addresses
  - Each program address forced to be above base address
    - Base register contents added to it & each program address checked to be below bounds address

- **To protect user’s instructions from user’s own data address errors – use two pairs of registers:**
  - Register pair for data
  - Register pair for instructions
Memory and Address Protection

**Tagged Architecture**

- Gives low granularity of access rights
- Every word of machine memory has ≥1 tag bits defining access rights to this word (a h/w solution!)
- Access bits set by OS
- Tested every time instruction accesses its location

![Table Example of Tagged Architecture](image)

<table>
<thead>
<tr>
<th>Tag</th>
<th>Memory Word</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td>0001</td>
</tr>
<tr>
<td>RW</td>
<td>0137</td>
</tr>
<tr>
<td>R</td>
<td>0099</td>
</tr>
<tr>
<td>X</td>
<td>R</td>
</tr>
<tr>
<td>X</td>
<td>W</td>
</tr>
<tr>
<td>X</td>
<td>R</td>
</tr>
<tr>
<td>X</td>
<td>W</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>R</td>
<td>4091</td>
</tr>
<tr>
<td>RW</td>
<td>0002</td>
</tr>
</tbody>
</table>

Code:  
R = Read-only  
RW = Read/Write  
X = Execute-only
Memory and Address Protection

Segmentation

- Benefits addressing and enhances memory protection for free
- Effect of an unbounded number of base/bounds registers
- Program segmentation:
  - Program divided into logical pieces (called segments)
    - e.g. Pieces are: code for single procedure / data of an array / collection of local data values
  - Consecutive program segments can be easily stored in nonconsecutive memory locations
Figure 4-6 Logical and Physical Representation of Segments.
## Memory and Address Protection

- **Addressing with segmentation**
  - Data item D addressed as:
    - *(segment_name_of_D, offset_of_D_within_segment)*
  - Instructions addressed analogously
  - For each process, OS keeps a separate Segment Translation Table (STT)
    - Rows in STT: *(segment_name, segment_offset)*
      - *segment_name* – name of segment containing data item
      - *segment_offset* – starting location for named segment
    - OS translates each data or instruction address using STT

- Two processes can share a segment S by having the same *segment_name* and *segment_offset* value in their STTs
Figure 4-7 Translation of Segment Address.
Memory and Address Protection

- Security-related benefits of segmentation
  - Strong segment protection
    - STT under exclusive OS control
      - each address requires STT access to get segment_offset for segment S
      - OS checks that address translates into S’s memory space (not beyond its end)
  - Different protection levels for different segments (approximates tagging at higher granularity)
    - e.g. segments with: R-only data / X-only code / W data
  - Different protection levels for different processes accessing the same segment
Memory and Address Protection

- Problems with segmentation
  - Programmer must be aware of segmentation
  - Efficiency
    - OS lookup of STT is slow
  - Fragmentation of main memory (by variable-sized holes left after “old” segments)
Memory and Address Protection (11)

Paging

- Principles:
  - Programs divided into equal-sized pages
  - Memory divided into same-sized page frames
  - Size is usually $2^n$, from 512 B to 4096 B
  - Address format for item (data or instruction) I:
    (page_nr_of_I, offset_of_I_within_page)
  - OS maintains Page Translation Table (PTT)
    — maps pages into page frames
- Address translation similar as for segmentation
  - But guaranteed that offset falls within page limit
    - e.g., for page size of 1024 = $2^{10}$,
      10 bits are allocated for page_offset
Figure 4-8 Page Address Translation.
Memory and Address Protection

- **Benefits of paging**
  - Programmer can be oblivious to page boundaries (automatic)
    - Paging completely hidden from programmer
  - No fragmentation of main memory

- **Problem w/ paging**
  - Can’t associate access rights with pages
    - Pages are random collections of items that require different protection level in general
    - Pages are not ‘access rights’ units (logical units) to be protected at the same level
Memory and Address Protection

Combined paging with segmentation

- **Principle:**
  - Paging offers efficiency
    - Hiding from programmer
    - No fragmentation
  - Segmentation offers ‘logical protection’
    - Grouping items w/ similar protection needs within the same segment

- **Paged segmentation:**
  - Programmer defines segments
  - Segments broken into pages automatically

- **Benefits of paging and segmentation but extra layer of address translation**
  - Additional h/w deals with this overhead
Figure 4-9 Paged Segmentation.