Memory Protection

Philip W. L. Fong

Department of Computer Science
University of Calgary
Calgary, Alberta, Canada

CPSC 525/625 (Winter 2018)
Multiprogramming

- Remember my Apple computer: single user
- Multiprogrammed OS
  - multiple users sharing resources
  - processes and threads
    - “Processes have different resources, implying controlled access”
    - “Threads share resources with less access control”
Enter the Notion of Processes

Before:

After:
Memory Protection

- Preventing memory interference
Memory Protection

- Preventing memory interference
- Traditionally implemented by hardware
Outline

1. Memory Fence
2. Virtual Memory
3. Reflections
Outline

1. Memory Fence
2. Virtual Memory
3. Reflections
Fixed Fence

- **Problem:** User code may read/write memory belonging to the OS.
**Problem:** User code may read/write memory belonging to the OS.

**Solution:**
- OS code resides in address 0 to $n$
- User code resides in address $n + 1$ onward
- Memory fault if user code attempts to access address within range 0–$n$. 
Fence Register

- **Problem:** Fixed address layout, inflexible — OS size may change over time.
Fence Register

- **Problem:** Fixed address layout, inflexible — OS size may change over time.
- **Solution:**
  - Store end of OS code in a fence register
  - Reject instruction during user mode if addressing OS code in fence
Problem: Only protects kernel from users. What about process against process?

Solution:
Base register: all addresses in a program is offsetted by the base register address.
Bounds register: all offsetted address is then compared to the bounds register, which delimits the upper bound of a legitimate address for the program.

OS changes the contents of base and bounds registers when context-switching to a different process.
Base/Bounds Register

- **Problem**: Only protects kernel from users. What about process against process?
- **Solution**:
  - **Base register**: all addresses in a program is offsetted by the base register
    
    \[ address + base \]
Base/Bounds Register

- **Problem:** Only protects kernel from users. What about process against process?
- **Solution:**
  - **Base register:** all addresses in a program is offsetted by the base register
    \[ address + base \]
  - **Bounds register:** all offsetted address is then compared to the bounds register, which delimits the upper bound of a legitimate address for the program
    \[ address + base < bounds \]
**Problem:** Only protects kernel from users. What about process against process?

**Solution:**

- **Base register:** all addresses in a program is offsetted by the base register
  
  \[ address + base \]

- **Bounds register:** all offsetted address is then compared to the bounds register, which delimits the upper bound of a legitimate address for the program
  
  \[ address + base < bounds \]

- OS changes the contents of base and bounds registers when context-switching to a different process.
Problem: Within a process, still possible to overwrite code.
Code vs Data

**Problem:** Within a process, still possible to overwrite code.

**Solution:** Separating the data space and code space:

- Data base
- Data bounds
- Program base
- Program bounds
Code vs Data

- **Problem:** Within a process, still possible to overwrite code.
- **Solution:** Separating the data space and code space:
  - Data base
  - Data bounds
  - Program base
  - Program bounds

Each of data and program space can be loaded separately into different parts of the physical memory space.
**Problem:** Protects only contiguous memory blocks. All-or-nothing protection. What if:
- mutual suspicion among code units within the same process
- change of accessibility over time (e.g., writable only during program initialization)
Problem: Protects only contiguous memory blocks. All-or-nothing protection. What if:
- mutual suspicion among code units within the same process
- change of accessibility over time (e.g., writable only during program initialization)

Solution: Tagged Architecture
- Protecting each memory word separately
- Each memory word is associated with a tag (a few bits) specifying accessibility (e.g., read-only, read/write, execute-only)
Example: Burroughs B6500-7500

- 3 tag bits to differentiate
  - data words
  - pointers
  - control words (i.e., stack pointers, etc)
Example: BiiN

- one tag for a block of size 128 or 256 bytes
- less costly
Not Popular

While the tagged architecture is very attractive by design, stock OSes (e.g., Windows, MacOS) are designed for conventional architecture (e.g., Intel).
Outline

1. Memory Fence
2. Virtual Memory
3. Reflections
Segmentation

- Divide a program into many small pieces (segment)
  - a segment may correspond to a procedure or an array
- Address: \( \langle \text{segment}, \text{offset} \rangle \)
- OS maintains a table (per process) mapping segment id to physical address
- Need mechanisms to check for access beyond end of a segment
General Advantages of Segmentation

1. Segments can be dynamically relocated at run time
2. Segments can be swapped out of physical memory into secondary storage
3. Every memory reference is mediated by the operating system, providing opportunities for protection
Security Advantages of Segmentation

- Different levels of accessibility for different segments
- Different users can share a segment, with different access rights
- Physical addresses cannot be forged
Paging

- Divide program into equal-sized pages
Paging

- Divide program into equal-sized **pages**
- Divide physical memory space into equal-sized **page frames**
Paging

- Divide program into equal-sized pages
- Divide physical memory space into equal-sized page frames
- Address: \( \langle page, offset \rangle \)
Paging

- Divide program into equal-sized pages
- Divide physical memory space into equal-sized page frames
- Address: \langle page, offset \rangle
- OS maintains a table (per process) mapping page numbers to frame numbers
Paging

- Divide program into equal-sized pages
- Divide physical memory space into equal-sized page frames
- Address: \( \langle \text{page}, \text{offset} \rangle \)
- OS maintains a table (per process) mapping page numbers to frame numbers
- Since all frames have same size, checking upper bound of offset is straightforward.
Paging

- Divide program into equal-sized pages
- Divide physical memory space into equal-sized page frames
- Address: \( \langle \text{page}, \text{offset} \rangle \)
- OS maintains a table (per process) mapping page numbers to frame numbers
- Since all frames have same size, checking upper bound of offset is straightforward.
- Coarser grained than segmentation: same accessibility level within a page
Paging + Segmentation

- Paging: implementation efficiency
- Segmentation: logical protection
- IBM 390 mainframe: paged segmentation
Memory protection can be achieved by software technology.
Memory protection can be achieved by software technology.

**Example:** Omniware
- OmniVM bytecode: RISC architecture
- Segments
- **Software-based Fault Isolation (SFI)** is employed to rewrite unsafe code into safe code, using one of two techniques:
  - **Segment matching:** guard code is injected before the instruction to check that the referenced segment id matches the allowed segment
  - **Sandboxing:** the segment id of the target address is dynamically overwritten by the allowed segment id
Memory protection can be achieved by software technology.

**Example:** Omniware

- OmniVM bytecode: RISC architecture
- Segments
- **Software-based Fault Isolation (SFI)** is employed to rewrite unsafe code into safe code, using one of two techniques:
  - **Segment matching:** guard code is injected before the instruction to check that the referenced segment id matches the allowed segment
  - **Sandboxing:** the segment id of the target address is dynamically overwritten by the allowed segment id

Software-based protection is potentially finer grained: beyond segments
Discussion: Protection within a Process

- Other than anticipating programming errors, what are security considerations that motivate memory protection (or in general, mutual suspicion) among code units within a process?
Other than anticipating programming errors, what are security considerations that motivate memory protection (or in general, mutual suspicion) among code units within a process?

- Extensible systems
- Browser plug-ins (via dynamic loading/linking)
Discussion: Protection within a Process

Other than anticipating programming errors, what are security considerations that motivate memory protection (or in general, mutual suspicion) among code units within a process?

- Extensible systems
  - Browser plug-ins (via dynamic loading/linking)
- Systems with scripting capabilities
  - Visual Basic in Excel
Discussion: Protection within a Process

Other than anticipating programming errors, what are security considerations that motivate memory protection (or in general, mutual suspicion) among code units within a process?

- Extensible systems
  - Browser plug-ins (via dynamic loading/linking)
- Systems with scripting capabilities
  - Visual Basic in Excel
- Mobile code
  - Java applets
Discussion: Protection within a Process

Other than anticipating programming errors, what are security considerations that motivate memory protection (or in general, mutual suspicion) among code units within a process?

- Extensible systems
  - Browser plug-ins (via dynamic loading/linking)
- Systems with scripting capabilities
  - Visual Basic in Excel
- Mobile code
  - Java applets
- Systems that interpret code
  - Database query evaluation (code injection)
The traditional world view: Processes as units of protection.

Diagram:
- User 1
- Process 1
- Process 2
- User 2
- User 3
- Kernel
A Change of World View

- The traditional world view: Processes as units of protection.

- With the emergence of dynamic loaded code, scripting, mobile code, and multi-language development, the traditional world view simply is no longer valid.
The traditional world view: Processes as units of protection.

With the emergence of dynamic loaded code, scripting, mobile code, and multi-language development, the traditional world view simply is no longer valid.

As we shall see, intra-process memory protection is typically achieved by software means.
Bibliographic Notes

- Most of the materials in these slides are based on Section 5.1 of [Pfleeger et al.].
- OmniWare and SFI: