# Operating Systems Architecture, Part IIIc

## CS239
## Advanced Operating Systems
## October 14, 1999
## Dr. Richard Guy



## Outline

- Part I: Why study OS?
- Part II: Best way to build OS
- Part III: What's new?
  - Microkernels, revisited
  - Operating System Extensibility
  - **64-bit Architectures**
    - The Challenge of Wide Virtual Addresses
    - Opal
    - Anonymous RPC



## Part IIIc Source Materials

- 64-bit architectures
  - Opal: A single Address Space System for 64-bit Architectures, by Chase, et al, IEEE WWOS92.
  - Sharing and Protection in a Single Address Space OS, by Chase, et al, ACM TOCS 11/94.
  - Anonymous RPC: Low-Latency Protection in a 64-Bit Address Space, by Yarvin, et al, USENIX, 1993.



## Part IIc Sources, continued

- The Expected Lifetime of "Single-Address-Space" Operating Systems, by Kotz and Crow, 1996.



## 64-Bit Address Space Machines

- An entirely different challenge for operating system architecture
- A fundamental assumption about our hardware is in the process of changing
- Should we change the OS to match it?



## The Challenge of Wide Virtual Address Spaces

- When virtual memory was created, 16 or 32 bit addresses spaces were available
- Reasonable size for one process
  - but maybe not for all processes on a machine
  - and certainly not for all processes ever on a machine

## Wide Address Space Architectures

- Computer architects can now give us 64 bit virtual addresses
- A 64 bit address space, consumed at 100 megabytes per second, lasts 5000 years
  - orders of magnitude beyond any process' needs
  - 40 bits can address a terabyte
- Lifetime paper has more careful study



## Do We Care?

- Should OS designers care about wide address spaces?
- Well, what can we do with them?
- One possible answer
  - put all processes in the same address space
  - maybe all processes for all time?



## Traditional Addressing Architectures

- Each process gets its own complete 32-bit address space
- Only that process can correctly "issue" its addresses
- Virtual memory hardware + OS software supports switching of virtual memories for different processes
- IPC must usually go through the OS



## Implications of Single Shared Address Space

- IPC is trivial
  - Everybody agrees on the addresses of everything
  - Shared memory, RPC are thus simple
- Separation of concepts of address space and protection domain
- *Uniform-address space*



## Address Space and Protection Domain

- A process has a protection domain
  - The data that cannot be touched by other processes
- And an address space
  - The addresses it can generate and access
- In standard systems, these concepts are merged



## Separating the Concepts

- These concepts are potentially orthogonal
- Just because you can issue an address doesn't mean you can access it
- (Though clearly to access an address you must be able to issue it)
- Existing hardware can support this separation

## Context-Independent Addressing

- Addresses mean the same thing in any execution context
- So a given address always refers to the same piece of data
- Key concept of uniform-address systems
- Allows many OS optimizations and improvements



## Uniform-Addressing Allows Easy Sharing

- Any process can issue any address
  - So any data can be shared
- All that's required is changing protection to permit desired sharing
- Suggests programming methods that make wider use of sharing



## The Opal System

- New OS using uniform-addressing
- Developed at U. of Washington
- Not intended as slight alteration to existing Unix system
- Other uniform-address operating systems are under development



## Protection Mechanisms for Uniform-Addressing

- Protection domains are assigned portions of the address space
- They can allow other protection domains to access them:
  - Read-only
  - With transferable access permissions
  - With system-enforced page-level locking



## Program Execution in Uniform-Access Memory

- Executing a program creates a new protection domain
- The new domain is assigned an unused portion of the virtual address space
- But it may also get access to used portions
- E.g., a segment containing the required executable image



## Virtual Segments

- Global address space is divided into segments
  - Each composed of variable number of contiguous virtual pages
- Domains can only access segments they *attach* to
- Attempting to access unattached segment causes a segment fault

## Persistent Memory in Opal

- *Persistent* segments exist even when attached to no current domain
- *Recoverable* segments are permanently stored
  - And can thus survive crashes
- All Opal segments can be persistent and recoverable
- Pointers can thus live forever on disk



## Code Modules in Opal

- Executable code stored in *modules*
  - independent of protection domains
- *Pure modules* can be easily shared
  - essentially static
- Can get benefit of dynamic loading without run-time linking



## Address Space Reclamation

- Trivial in non-uniform-address systems
- Tricky in uniform-address systems
- Problem akin to reclaiming inodes in the presence of (multiple) hard links
- But even if segments improperly reclaimed, only trusting domains can be hurt



## Anonymous RPC

- Goal: “Build system that combines the protection of UNIX processes and the speed of threads, to make IPC cheap enough for heavy use.”
- Problems
  - VM to enforce protection is expensive (more state means slower context switches)
  - lightweight threads are fast, not safe or secure



## ARPC main idea

- In very large, sparse address space, VM address mappings can act as capabilities
  - knowledge of address implies permission
  - restricting knowledge provides protection
- Resulting anonymity is basis for general-purpose protection mechanism



## ARPC Implementation

- Need way to assign addresses at load time
  - non-discoverable, non-derivable
- So, select addresses randomly
  - only brute-force search can “break” method
  - but, no truly random number generator
  - use cryptosystem (key: some secure password)
    - plaintext is allocation sequence number
    - resulting encrypted text is securely random

## Is brute force search feasible?

- Method relies on infeasibility of brute force address probing
  - issue addresses and catch seg-faults
- Key parameters
  - time to process segmentation faults
  - size of physical memory
  - size of virtual address space



## "Time to Breach" results

| PhysMem | 1s delay | 1ms delay | 1µs delay |
|---|---|---|---|
| 16MB | 24162 yrs | 24 yrs | 1.25 wks |
| 256MB | 1510 yrs | 1.5 yrs | 13.2 hrs |
| 2GB | 188 yrs | 2.3 mo | 1.7 hrs |
| 16GB | 23 yrs | 8.5da | 12 mins |



## Maintaining Anonymity

- RPC path of control must flow through an intermediary
  - protected
  - aware of RPC binding
  - able to manage control and data flow without revealing either's address to the other
  - kernel is obvious, and slow!



## Anonymity, continued

- Exploit execute-only page protection
  - caller knows address of callee text, but can't damage text or discover data locations
  - use execute-only jump table which contains the actual service entry point as immediate operand
  - use separate stack during server execution
  - cost of anonymity: one additional jump instr + two stack/register switch/save methods
  - if some trust exists, can reduce some/all saves



## Null RPC performance

| Protocol | instructions |
|---|---|
| SRC RPC | 1226 |
| Mach RPC | 950 |
| LRPC | 424 |
| URPC | 251 |
| ARPC | 116 |



## Meta-level Summary Comments

- When exploring a new paradigm, push it to the (believed) limits
- Try separating joint concepts, or splitting a previously "atomic" idea into multiple parts
- Sometimes OS progress pushes HW progress; sometimes, vice-versa