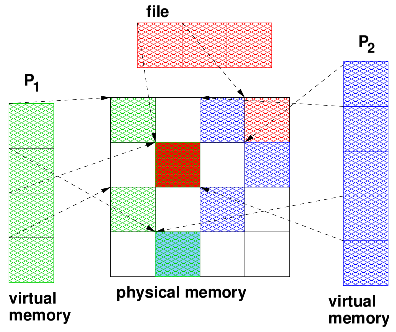
Single-Address-Space Operating Systems

- New paradigm for OS design
- Enabled by 64-bit hardware
- Motivation: use H/W features to:
  - ➜ improve overall performance,
  - ➜ simplify applications.

Address Spaces

Traditional OS use a separate address space for each process.



Multiple address spaces:

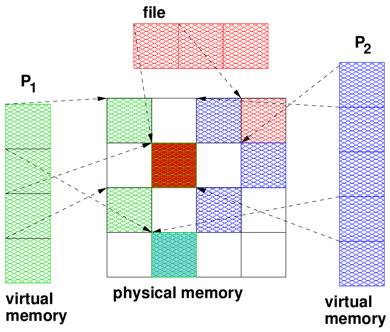
- Each address space has own virtual→physical mapping.
- Advantages:
  - ➜ Maximises available address space
  - ➜ Isolates processes (provide protection)
- Drawbacks:
  - ➜ Meaning of virtual address depends on process context
  - ➜ Isolation inhibits sharing

How do processes share data?

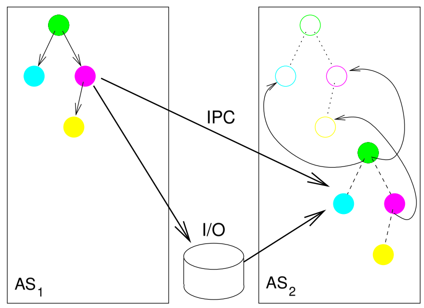
- Via files:
  - ➜ One process writes data to a file, another reads file
  - ➜ Similarly pipes, sockets, ...
- Via message passing (IPC):
  - ➜ One process sends message, another receives
- Via shared memory:
  - ➜ both establish shared memory arena (`mmap()`)
  - ➜ shared buffers are mapped to the same physical memory locations
  - ➜ both can access the same data directly

All require OS intervention.

Sharing between address spaces



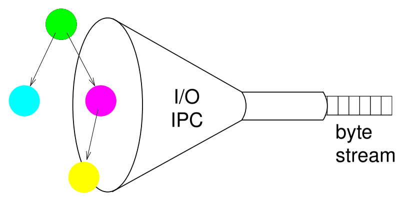
Problems with sharing: Pointers!



- ➜ pointers are bound to an address space
- ➜ they are meaningless outside

Sharing across address spaces

... requires copying and conversions



Data structure

- ➜ implies loss of typing
- ➜ increases code complexity (order of 30% of app code!)
- ➜ increases run-time overhead

Other problems with address spaces

memory data:	file data:

`item_t a, *x;`	`item_t a;`
	`int x;`
	`FILE *f;`

`...`
`a = *x;`
	`...`
	`f = fopen("f","r");`
	`fseek (f, x, SEEK_SET);`
	`fread (*a, sizeof(item_t), 1, f);`

address is `*x`	address is `("f",*x)`

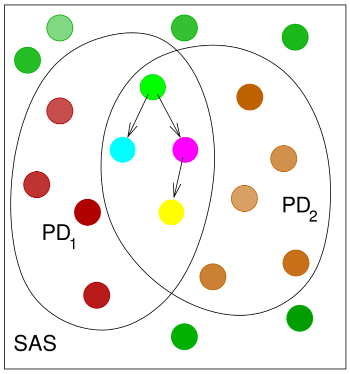
Inconsistent naming of persistent and volatile data

WHY DO WE HAVE PROBLEMS WITH SHARING?

- The problems are with pointers
  - ➜ pointer problems result from per-address-space mappings
  - ➜ result from the desire to maximise the available address space
  - ➜ results from limitations on address bits
- But we have 64-bit architectures now!
- Why not abolish private mappings????
  - ➜ all address spaces are merged into one
  - ➜ each process has same virtual→physical mapping
  - ➜ all memory objects (text, data, stack, libraries) are allocated at unique addresses
  - ➜ 2^{64} is big enough to include "files" as memory objects

⇒ *single-address-space system*

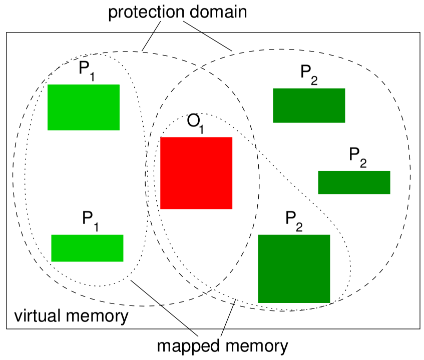
SINGLE-ADDRESS-SPACE OPERATING SYSTEMS



SASOS CHARACTERISTICS:

- Unique addresses for all data items
  - ➜ threads always agree about the address of data
- Sharing by reference
  - – simply pass pointer
- no marshalling or conversion of data formats required
  - ➜ on-disk format same as in-memory format

PROTECTION IN A SASOS



PROTECTION:

- Everything is *visible*
- *Protection domain* defines what is *accessible*
- Access requires mapping virtual to physical addresses
- Mapping established by system

⇒ System controls access by establishing *partial view* of the single address space

- Can implement usual protection models (ACLs, capabilities)

SINGLE ADDRESS SPACE ADVANTAGES

APPLICATION VIEW

- Simple naming mechanism – 64 bit address – supported by "conventional" hardware.
- User data structures can contain embedded references to other data.
- Eliminates excessive copying of data and software pointer translation.

SASOS ADVANTAGES: SYSTEM VIEW

- Simplifies data migration
- Simplifies process migration
- Orthogonality of translation and protection
- No need for file system — all disk I/O is paging
- RAM is cache for VM — unified buffer & disk cache management
- Easy to implement zero-copy operations
- In-place execution — no need for position-independent code

⇒ Simplified system implementation and increased performance

SASOS ADVANTAGES: HARDWARE VIEW

- Virtual caches are no problem
  virtual address maps uniquely to physical address
- Hardware separating translation from protection could increase performance due to increased TLB coverage
  (e.g. IA-64 *protection keys*)

Single-Address-Space Operating Systems

IBM System/38 [Ber80] and successor **AS/400** [Sol96] (1978)

- high-level object-oriented architecture built on single-level store
- geared towards data-intensive commercial applications
- protection based on tagged capabilities

Drawbacks:
- totally different environment
- requires hardware support
- performance...

Angel [MSS+93] (City University, London, 1992–5)

- runs on standard hardware
- microkernel architecture with lightweight RPC
- protection server for flexible protection model

Drawbacks:
- prototype is 32-bit only
- performance?

Opal [CLFL94] (U of Washington, 1992–4)

- runs on standard hardware
- protection domains as 1st class objects
- password capabilities
- implemented on top of Mach

Drawbacks:
- applications must handle capabilities (e.g. on RPC)
- no fast rights amplification
- performance!

Sombrero [SMF96] (Arizona State U, 1994–now)

- designed (not implemented) special protection hardware
- simulated on Alpha
- established some software engineering advantages of SASOS

Drawbacks:
- special hardware!

Mungi [HEV+98] (UNSW, 1994–now)

- "pure" SASOS (no message-passing IPC)
- standard 64-bit hardware
- implemented on L4 (MIPS, Alpha, Itanium, Power, x86)
- discretionary and mandatory access control
- user-level device drivers and system extensions
- POSIX emulation
- fastest SASOS to date

SASOS Issues

- Protection model
- System extensibility
- POSIX compatibility
- Resource Management
- Linking
- Persistence
- Performance

Discussed in context of Mungi

Mungi Security

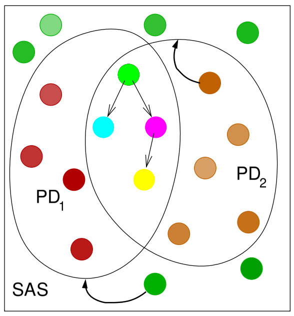
Two basic kinds of mechanisms:

- Discretionary access control
  - ➜ *user-oriented* mechanism
  - ➜ users determine which of their data should be accessible to others
  - ➜ essential for *privacy*
  - ➜ two basic models: *access control lists* and *capabilities*
- Mandatory access control
  - ➜ *system-oriented* mechanism
  - ➜ system-wide *security policy* limits data flow
  - ➜ essential for use of *untrusted extensions*
  - ➜ range of models: Denning, Bell-LaPadula, Chinese Wall, role-based....

Mungi has both

Discretionary Access Control in Mungi

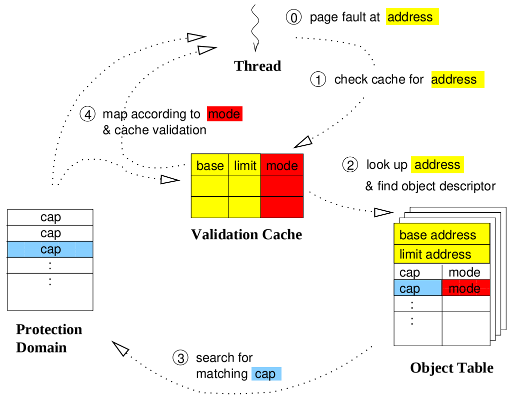
- Threads execute inside a *protection domain* (PD)
- A protection domain is defined as a set of *capabilities*
- Capabilities and protection domains are user-level objects
- Thread may or may not have control over its PD
  - ➜ supports user-controlled confinement



Main Mungi abstractions:

- Unit of protection is the *memory object*
  - ➜ contiguous page range
  - ➜ associated with a set of *password capabilities*
- Unit of execution is the *thread*
  - ➜ kernel-scheduled
  - ➜ execute in an *active protection domain* (APD)
  - ➜ associated with a (user-level) *TCB object* (UTCB)
  - ➜ thread control is via access to UTCB
- An APD consists of (caps for) an array of *Clists*
  - ➜ A Clist is an object consisting of an array of caps
  - ➜ APD itself is in kernel space
- Caps confer sets of rights, combination of:
  - ➜ `read`, `write`, `execute`, `delete`, `enquire`, `PDX`

Access validation:



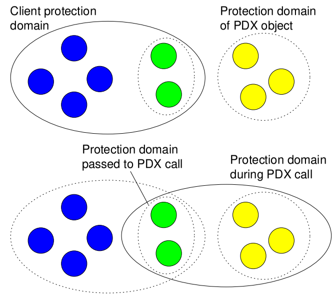
Note: All capability presentation is *implicit*

Threads and protection domains

- A thread can be started in an existing APD or a new one
- New APD is instantiated from a template
  - ➜ called the *protection domain object* (PDO)
  - ➜ system-defined structure
  - ➜ consists of an array of *clist* capabilities,
  - ➜ access restricted to trusted management code
  - ➜ PDO creation requires special privileges
- Thread can also change APD temporarily
  - ➜ called *protection-domain extension*, PDX
  - ➜ requires PDX cap
  - ➜ serves as protected-procedure call mechanism

Protected Procedure Calls

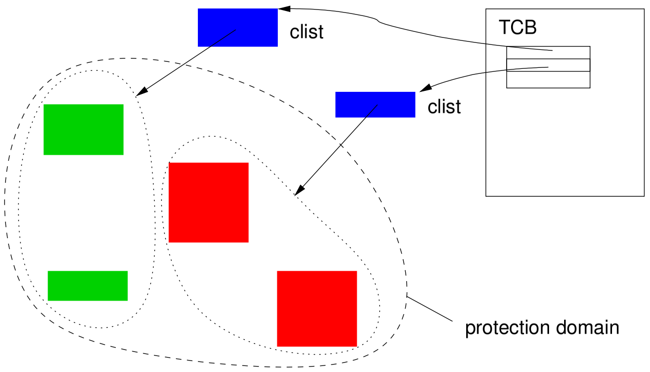
- Object can have (PDX) type:
  - ➜ has *PDX capabilities*,
  - ➜ registered set of *entry points*,
  - ➜ an associated *PDX clist*.
- Owner's APD changes *for the duration of the call*
- Allows secure invocation of an object in a PD different from caller's
- *Discretionary* access control validates entry points and invocation right



Protection Domain Manipulation:

- All capability presentation is *implicit* (via clists).
- A thread can manipulate its protection domain:
  - ➜ by modifying its clists,
  - ➜ **provided that the APD contains the clists**.
- A thread can be set up so that it's APD:
  - – does not contain the clists defining it,
  - – does not contain write access to any "public" objects.
- Such a thread is *confined*.

Discretionary Confinement in Mungi



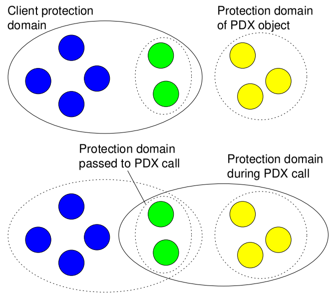
Mandatory Access Control in Mungi

- Using *domain and type enforcement* (DTE) model [EH01a]:
  - – Each object has a *type* label
  - – Each APD has a *domain* label
  - – Each thread has:
    - ➜ a type label (because it's an object)
    - ➜ a domain label (because it belongs to an APD)
  - – a PDX object has:
    - ➜ a type label (because it's an object)
    - ➜ a domain label (because it has an associated PD)
- System-wide security policy is a relation on types and domains

Mandatory access control operation

- MAC policy relation is represented in (user-level) *policy object*
- Kernel consults on each access validation:
  - – Object access: domain has access to type
  - – APD creation / PDX call:
    - ➜ thread has access to invoked object
    - ➜ caller APD has right to transfer to target APD
- Policy object consists of a number of (mostly simple) validation functions
  - ➜ invoked via PDX ⇒ also subject to DAC and MAC!
  - ➜ MAC validations are cached in separate validation cache

PDX AGAIN...



- ➜ *discretionary* access control validates entry points and invocation right
- ➜ *mandatory* access control validates right to use target PD
- ➜ *discretionary* and *mandatory* access control validate data access

- Can use this as the basis for secure system extensions!
  - ➜ Component model based on PDX for extending system

OS EXTENSIBILITY

- Linux loadable kernel modules:
  - ➜ Run as part of the kernel ⇒ no protection.
  - ➜ Unsuitable for OS extension/customisation by users.
- User-level servers (Mach, Windows-NT):
  - ➜ based on message-based communication with servers,
  - ➜ performance problems ⇒ migrate extensions into kernel.
  - ➜ newer systems try to do better (e.g. SawMill)

EXISTING APPROACHES TO OS EXTENSIBILITY (CONT'D)

- Safe kernel extensions by *trusted code* (e.g. SPIN [BSP+95]):
  - ➜ extensions must be programmed in *type-safe* language (Modula-3),
  - ➜ restrictive programming model,
  - ➜ large trusted computing base,
  - ➜ unconvincing performance.
- Safety by *sandboxing* kernel extensions (e.g. Vino [SESS96]):
  - ➜ poor performance.

WHAT'S WRONG?

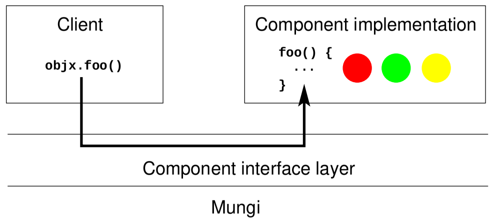
- Kernel extensions create huge *security problems*.
  - ➜ Kernel code is inherently unrestricted.
  - ➜ Imposition of restrictions results in *cost* and *complexity*.
- User-level extensions can be secure but:
  - ➜ have potential *performance problems*, and
  - ➜ need to be supported by an appropriate *framework*.

WHAT'S NEEDED?

User-level extensibility can be made to work if [EH01b]:

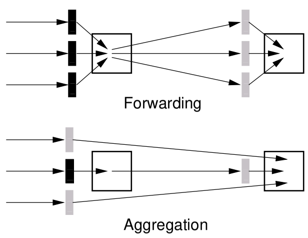
- Performance can be ensured.
  - ➜ Requires fast inter-process communication.
  - ➜ Has been demonstrated (L4, Pebble, Mungi).
- Security can be guaranteed.
  - ➜ Extensions operate within "normal" OS protection system.
  - ➜ Will work if OS protection is *strong and flexible* enough.
- A framework for extensions is provided which supports:
  - ➜ transparent invocation of extended services,
  - ➜ low overhead extension and customisation of extensions,
  - ➜ software technology to minimise complexity.

MUNGI COMPONENT MODEL



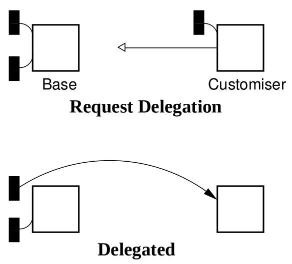
- ➜ Component implementation is in different PD from caller
  - ➜ Can use for invoking protected subsystems
- ➜ PDX is used for invocation
- ➜ Component data is created *inside* the component PD
- ➜ Client and component are mutually protected
- ➜ Mandatory security policy limits data propagation
- ➜ Single address space ⇒ *no need to marshal arguments!*

EXTENDING EXTENSIONS



- ➜ Components export *interfaces*.
- ➜ Component instances can invoke interfaces of other instances (and thus extend them): *forwarding*.
- ➜ *Aggregation* allows direct invocation of extended interface.

CUSTOMISATION



- ➜ *Delegation* is a dynamic form of aggregation that allows an invocation of a base component to be transparently handled by another component.
- ➜ Avoids the semantic nightmares of *virtual inheritance*.

Overhead of Mandatory Access Control

Benchmark	no MAC ms	with MAC ms	O/H %
OO1	187.8	187.8	0.0
Jigsaw$_{56\times56}$	374	375	0.3
Andrew	672	674	0.3

Extension System Performance: Microbenchmarks

Broken PostScript Picture — sorry

Extension System Performance: Macrobenchmarks

Environment	Time
Linux (RAM disk)	283 ms
Mungi (statically linked)	146 ms
Mungi (extension)	247 ms

Resource Management in Mungi

A major design aspect of Mungi is to make the system as *unintrusive as possible*.

This means:

- ➜ no restrictions whatsoever on pointer/capability use,
- ➜ presentation of a valid capability at any time should guarantee access,
- ➜ object persistence is under full control of users (as traditional files)

The system should also *not rely on "sensible" users*, like asking users to register/de-register "interest" in an object.

- ➜ How do we deal with garbage?

Resource Management ...

Automatic garbage collection is impossible because:

- ➜ reference counting is impossible as system cannot track references,
- ➜ scanning schemes cannot work as system cannot find pointers.

Quota:

- ➜ require checking whenever an object is allocated ⇒ overhead,
- ➜ cannot distinguish between used and unused space.

What else?

Resource Management Using Bank Accounts

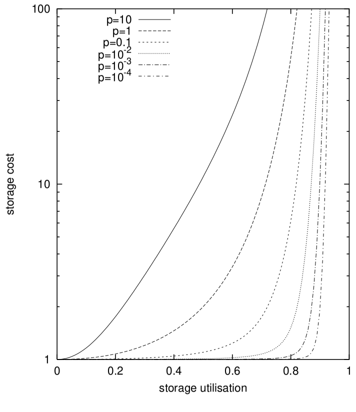
- ➜ Every object is associated with a *bank account*.
- ➜ "Rent" is periodically collected from account for associated objects.
- ➜ Regular "income" is periodically deposited into bank accounts.
- ➜ Overdrawn accounts prevent further creation of persistent objects ⇒ forces users to clean up.
- ➜ "Tax" on high balances prevents excessive accumulation of funds.

Based on similar ideas in Amoeba [MT86] and the Monash Password Capability System [APW86].

Graceful Degradation

Q: How stop system from brickwalling when disk is full?

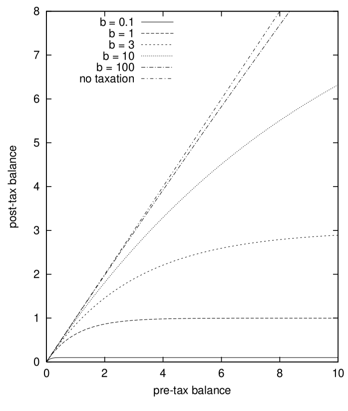
A: Market approach: adjust rent to demand!



Fairness

Q: How stop someone from accumulating large amounts of money enabling them to "buy the whole world"?

A: Taxation: limit balance by imposing a progressive tax!



Resource management issues:

- Secondary memory — solved
- Primary memory — have a model, work to be done
- Kernel memory (TCBs) — to be done
- CPU time, scheduling — to be done
  ➜ Lottery scheduling [WW94] worth looking at
- Network bandwidth — to be done
- ???

References

[APW86] Mark Anderson, Ronald Pose, and Chris S. Wallace. A password-capability system. *The Comp. J.*, 29:1–8, 1986.

[Ber80] Viktors Berstis. Security and protection in the IBM System/38. In *Proc. 7th Symp. Comp. Arch.*, pages 245–250. ACM/IEEE, May 1980.

[BSP+95] Brian N. Bershad, Stefan Savage, Przemysław Pardyak, Emin Gün Sirer, Marc E. Fiuczynski, David Becker, Craig Chambers, and Susan Eggers. Extensibility, safety and performance in the SPIN operating system. In *Proc. 15th ACM SOSP*, pages 267–284, Copper Mountain, CO, USA, Dec 1995.

[CLFL94] Jeffrey S. Chase, Henry M. Levy, Michael J. Feeley, and Edward D. Lazowska. Sharing and protection in a single-address-space operating system. *ACM Trans. Comp. Syst.*, 12:271–307, 1994.

[EH01a] Antony Edwards and Gernot Heiser. A component architecture for system extensibility. Technical Report UNSW-CSE-TR-0103, School Comp. Sci. & Engin., University NSW, Sydney 2052, Australia, Mar 2001. URL `ftp://ftp.cse.unsw.edu.au/pub/doc/papers/UNSW/0103.ps.Z`.

[EH01b] Antony Edwards and Gernot Heiser. Components + Security = OS Extensibility. In *Proc. 6th ACSAC*, pages 27–34, Gold Coast, Australia, Jan 2001. IEEE CS Press.

[HEV+98] Gernot Heiser, Kevin Elphinstone, Jerry Vochteloo, Stephen Russell, and Jochen Liedtke. The Mungi single-address-space operating system. *Softw.: Pract. & Exp.*, 28(9):901–928, Jul 1998.

[MSS+93] Kevin Murray, Ashley Saulsbury, Tom Stiemerling, Tim Wilkinson, Paul Kelly, and Peter Osmon. Design and implementation of an object-orientated 64-bit single address space microkernel. In *Proc. 2nd USENIX WS. Microkernels & other Kernel Arch.*, pages 31–43, Sep 1993.

[MT86] Sape J. Mullender and Andrew S. Tanenbaum. The design of a capability-based distributed operating system. *The Comp. J.*, 29:289–299, 1986.

[SESS96] M.I. Seltzer, Y. Endo, C. Small, and K.A. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In *Proc. 2nd USENIX OSDI*, pages 213–228, Nov 1996.

[SMF96] Alan C. Skousen, Donald S. Miller, and Ronald G. Feigen. The Sombrero operating system: An operating system for a distributed single very large address space — general introduction. Technical Report TR-96-005, Computer Science and Engineering Department, Arizona State University, Tempe, AZ 85287-5406, Apr 1996.

[Sol96] Frank G. Soltis. *Inside the AS/400*. Duke Press, Loveland, CO, USA, 1996.

[WW94] Carl A. Waldspurger and William E. Weihl. Lottery scheduling: Flexible proportional-share resource management. In *Proc. 1st USENIX OSDI*, pages 1–11, Monterey, CA, USA, Nov 1994.