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.
Traditional OS use a separate address space for each process.
M U L T I P L E  A D D R E S S  S P A C E S:

- Each address space has its own virtual→physical mapping.
MULTIPLE ADDRESS SPACES:

• Each address space has own virtual→physical mapping.

• Advantages:
  ➔ Maximises available address space
  ➔ Isolates processes (provide protection)
MULTIPLE ADDRESS SPACES:

- Each address space has its own virtual→physical mapping.

- Advantages:
  - Maximises available address space
  - Isolates processes (provides 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, ...
**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
**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 \( \textit{mmap()} \)
  - shared buffers are mapped to the same physical memory locations
  - both can access the same data directly
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

P1

virtual
memory

file

physical memory

P2

virtual
memory

file

physical memory

virtual
memory
PROBLEMS WITH SHARING: POINTERS!
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
SHARING ACROSS ADDRESS SPACES

... requires copying and conversions

$\rightarrow$ implies loss of typing
$\rightarrow$ increases code complexity (order of 30% of app code!)
$\rightarrow$ increases run-time overhead
OTHER PROBLEMS WITH ADDRESS SPACES

memory data: file data:

item_t a, *x;
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
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
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
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 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????
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
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
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 domain

Virtual memory

Mapped memory

P1
O1
P2
P1
P2
Protection:

- Everything is visible

- Protection domain defines what is accessible
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
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
**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
**ANGEL** [MSS\textsuperscript{+}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 1\textsuperscript{st} class objects
- password capabilities
- implemented on top of Mach
**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
**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
SASOS Issues

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

Discussed in context of Mungi
TWO BASIC KINDS OF MECHANISMS:

- Discretionary access control
- Mandatory access control
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
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....
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
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

![Diagram showing protection domains PD1 and PD2 with overlapping resources and connections between them, labeled SAS (Software Assurance System) and PD1, PD2]
Main Mungi Abstractions:

- Unit of protection is the *memory object*

- Unit of execution is the *thread*

- An APD consists of (caps for) an array of *Clists*

- Caps confer sets of rights
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*

• An APD consists of (caps for) an array of *Clists*

• Caps confer sets of rights
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*

- Caps confer sets of rights
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
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:

0. page fault at address

Thread

1. check cache for address

Validation Cache

2. look up address & find object descriptor

3. search for matching cap

Object Table

4. map according to mode & cache validation

Protection Domain

Object Table

cap        mode
   :
   :
cap        mode
base address
limit address
base limit mode
Validation Cache
**ACCESS VALIDATION:**

0. page fault at address

1. check cache for address & find object descriptor

2. look up address & find object descriptor

3. search for matching cap

4. map according to mode & cache validation

**Protection Domain**

<table>
<thead>
<tr>
<th>cap</th>
<th>cap</th>
<th>cap</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Object Table**

<table>
<thead>
<tr>
<th>base address</th>
<th>limit address</th>
</tr>
</thead>
<tbody>
<tr>
<td>cap</td>
<td>mode</td>
</tr>
<tr>
<td>cap</td>
<td>mode</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Validation Cache**

<table>
<thead>
<tr>
<th>base</th>
<th>limit</th>
<th>mode</th>
</tr>
</thead>
</table>

Note: All capability presentation is *implicit*
THREADES 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
THREADES 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*
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
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.
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.
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

clist
clist
TCB

protection domain
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
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)
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)
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
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
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
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.
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.
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 \[\text{EH01b}\]:

- Performance can be ensured.
  - Requires fast inter-process communication.
  - Has been demonstrated (L4, Pebble, Mungi).
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.
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.
Component implementation is in different PD from caller
- Can use for invoking protected subsystems

foo() {
...
}
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
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!
Components export *interfaces*.
- Component instances can invoke interfaces of other instances (and thus extend them): *forwarding*.
- *Aggregation* allows direct invocation of extended interface.
Request Delegation

→ *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

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>no MAC ms</th>
<th>with MAC ms</th>
<th>O/H %</th>
</tr>
</thead>
<tbody>
<tr>
<td>OO1</td>
<td>187.8</td>
<td>187.8</td>
<td>0.0</td>
</tr>
<tr>
<td>Jigsaw(_{56 \times 56})</td>
<td>374</td>
<td>375</td>
<td>0.3</td>
</tr>
<tr>
<td>Andrew</td>
<td>672</td>
<td>674</td>
<td>0.3</td>
</tr>
</tbody>
</table>
EXTENSION SYSTEM PERFORMANCE: MICROBENCHMARKS

<table>
<thead>
<tr>
<th></th>
<th>Mungi</th>
<th>SPIN</th>
<th>VINO</th>
<th>COM</th>
<th>omniORB</th>
<th>ORBacus</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create</td>
<td>118</td>
<td></td>
<td></td>
<td></td>
<td>5622</td>
<td>4240</td>
</tr>
<tr>
<td>Invoke</td>
<td>100</td>
<td>101</td>
<td>885</td>
<td>1993</td>
<td>768</td>
<td>9319</td>
</tr>
</tbody>
</table>
## Extension System Performance: Macrobenchmarks

<table>
<thead>
<tr>
<th>Environment</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux (RAM disk)</td>
<td>283 ms</td>
</tr>
<tr>
<td>Mungi (statically linked)</td>
<td>146 ms</td>
</tr>
<tr>
<td>Mungi (extension)</td>
<td>247 ms</td>
</tr>
</tbody>
</table>
A major design aspect of Mungi is to make the system as *unintrusive as possible*. 
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.
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?
Automatic garbage collection is impossible because:

- reference counting is impossible as system cannot track references,
- scanning schemes cannot work as system cannot find pointers.
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?
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 \( \Rightarrow \) 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?
**GRACEFUL DEGRADATION**

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

**A:** Market approach: adjust rent to demand!
Q: How stop someone from accumulating large amounts of money enabling them to “buy the whole world”?
**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
- ???


