Resource Management in Mungi

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

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

A: Market approach: adjust rent to demand!
FIRNESS

Q: How stop someone from accumulating large amounts of money enabling them to “buy the whole world”? 
**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
- ???
Linking in Mungi

**Inherent features of a SASOS:**

- data have unique address, *independent of context*,
- a particular address *always* refers to the same data.
Linking in Mungi

**Inherent features of a SASOS:**

- Data have unique address, *independent of context*,
- A particular address *always* refers to the same data.

So, how keep *private data* of separate executions of the same program separate?

- Can allocate stacks at unique addresses.
- How about the data segment, in particular global *statics*?

⇒ Need to re-think linking.
STATIC LINKING IN UNIX

Loader

crt0.o
main.o
..... .o
libc.a
..... .a
initialised
data

Linker

a.out
main.o
..... .o
libc.a
..... .a
initialised
data

text
data
bss
stack

❝ all symbols resolved by linker
DYNAMIC LINKING IN UNIX

some symbols resolved by loader
Properties of Dynamic Linking:

❖ Library code only exists in **single copy** (RAM and disk).
❖ New library versions immediately usable.
❖ Reduced startup latency if library resident.
♣ Execution fails if library removed.
♣ Overhead due to level of indirection.
**Symbol Resolution at Load Time**

- **main.o**
  - `printf(...)`
  - `GOT:main`
  - `printf()`

- **libc.so**
  - `printf()`
  - `GOT:libc`

- **Loader enters external references into global object table (GOT).**
- **Functions invoked by indirect jump via GOT.**
**Lazy Loading**

- Delay library load time until first function invocation.
- GOT initialised to point to stub code.
- Stubs invoke lazy loader.
- Loader fixed up pointers in GOT.
LAZY LOADING

→ Delay library load time until first function invocation.
→ GOT initialised to point to stub code.
→ Stubs invoke lazy loader.
→ Loader fixed up pointers in GOT.

Features:

✌ Reduce startup latency (at expense of later stalls).
✌ Save overhead for libraries not actually invoked.
♣ Delayed failure if library not available.
Dynamic linking problems

Position of library code is *not know at library link time* ⇒ library must use *position-independent code* (PIC).
DYNAMIC LINKING PROBLEMS

Position of library code is not known at library link time
⇒ library must use position-independent code (PIC).

All jumps must be:
- PC-relative,
- off a register, or
- indirect (via GOT).

Every function must first locate GOT (via PC-relative load)
⇒ overhead.
THE GLOBAL OFFSET TABLE IS *private static data*:

- **Global** to library
  - cannot be on stack.
- **Private** to invocation
  - every process needs own copy.

⇒ GOT cannot be allocated at a fixed address.
⇒ GOT must be allocated in a *per-library data segment*
  at known (at link-time) offset from library code
  (to allow PC-relative addressing).

Same holds for any variables declared **static** or **extern** in library
  e.g.: **errno**
LINKING IN A SASOS

- *Static linking* works as in UNIX (with same drawbacks) except for private static data.
**Linking in a SASOS**

- *Static linking* works as in UNIX (with same drawbacks) except for private static data.
- Why copy when everything is in the address space anyway?

Can execute library code *in place*.

Called *global static linking* [CLFL94] (all references resolved at *link* time).
**FEATURES OF GLOBAL STATIC LINKING**

- Code sharing (as for dynamic linking).
- **No** automatic replacement of libraries, re-linking required (as for static linking).
- Removing library results in failure (as for dynamic linking).
- **Cannot have** private static data in library.
- Private static data is a problem for dynamic linking too.
PROBLEMS WITH PRIVATE STATIC DATA

- Same address $\Rightarrow$ same data in a SASOS

$\Rightarrow$ Different processes’ private static data must be allocated at different addresses.

$\Rightarrow$ Need per-invocation, per-library data segments.

- How to locate data segment?
PROBLEMS WITH PRIVATE STATIC DATA

- Same address ⇒ same data in a SASOS
  ⇒ Different processes’ private static data 
  *must* be allocated at different addresses.
  ⇒ Need per-invocation, per-library data segments.
- How to locate data segment?
  ➔ Cannot use PC-relative addressing.
  ➔ Caller must pass address to callee:
    * explicit parameter (Nemesis [Ros95]),
    * *global pointer register* (Mungi [DH99]).
    ☀ No overhead for intra-module calls.
**Dynamic Linking in Mungi**

Module descriptor contains:

- pointer to module’s data segment,
- pointers to imported functions.
Dynamic linking in Mungi...

- Calling sequence differs between local and cross-module invocation
  - supported by modified GNU assembler.
- Cross-module calling overhead is about 3 instructions.
- Module description objects specify module interface.
- Function pointers consist of
  - (entrypoint address, global pointer value).
  - Appropriate compiler modification not done yet.
MODULE DESCRIPTION OBJECT

**libc.mm:**

- [IMPORTS]
- [EXPORTS]
  - strlen
  - bcopy
  - ...
- [OBJECTS]
  - c1.o
  - c2.o
  - ...

**main.mm:**

- [IMPORTS]
- libc.mm
- [EXPORTS]
- [OBJECTS]
  - main.o
  - sub.o
  - ...

---
cse/UNSW

---

COMP9242 2002/S2 W13 P20
**CODE PREPARATION PROCESS**

```
C

c1.s
gas
gcc
c1.o
ml
libc.mm
c1.c

c2.o
main.o
gas
main.md
libc.md

a.out
```

- c1.c
- gcc
- c1.s
- gas
- c1.o
- ml
- libc.mm
- c1.c
- gcc
- c2.o
- ml
- libc.mm
- main.o
- gas
- libc.md
- main.s
- main.md
- ml
- main.s
- main.md
- ml
- a.out

__COMP9242 2002/S2 W13 P21__

cse/UNSW
PROBLEMS: EXPORTED VARIABLES

Cannot do: extern int errno;
PROBLEMS: EXPORTED VARIABLES

Cannot do:  extern int errno;

🔹 Bad practice anyway, changed to:

extern int *__errno();
#define errno (*(__errno()))

for multi-threading support in ➔ Solaris,
➔ Digital Unix,
➔ Irix,
➔ Linux, ...
## Performance

<table>
<thead>
<tr>
<th></th>
<th>Irix/32-bit/SGI-cc</th>
<th></th>
<th>Mungi/64-bit/GCC</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>static</td>
<td>dynamic</td>
<td>dyn/stat</td>
<td>static</td>
</tr>
<tr>
<td>lookup</td>
<td>7.26(3)</td>
<td>8.02(3)</td>
<td>1.104(10)</td>
<td>7.568(6)</td>
</tr>
<tr>
<td>f. trav.</td>
<td>4.77(3)</td>
<td>5.17(4)</td>
<td>1.084(15)</td>
<td>6.013(6)</td>
</tr>
<tr>
<td>b. trav.</td>
<td>5.13(2)</td>
<td>5.68(4)</td>
<td>1.107(12)</td>
<td>6.976(4)</td>
</tr>
<tr>
<td>insert</td>
<td>4.61(2)</td>
<td>5.02(2)</td>
<td>1.087(10)</td>
<td>4.528(4)</td>
</tr>
<tr>
<td>total</td>
<td>21.7(1)</td>
<td>23.9(1)</td>
<td>1.097(12)</td>
<td>25.08(1)</td>
</tr>
</tbody>
</table>
## PERFORMANCE

<table>
<thead>
<tr>
<th></th>
<th>Irix/32-bit/SGI-cc</th>
<th></th>
<th>Mungi/64-bit/GCC</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>static</td>
<td>dynamic</td>
<td>dyn/stat</td>
<td>static</td>
</tr>
<tr>
<td>lookup</td>
<td>7.26(3)</td>
<td>8.02(3)</td>
<td>1.104(10)</td>
<td>7.568(6)</td>
</tr>
<tr>
<td>f. trav.</td>
<td>4.77(3)</td>
<td>5.17(4)</td>
<td>1.084(15)</td>
<td>6.013(6)</td>
</tr>
<tr>
<td>b. trav.</td>
<td>5.13(2)</td>
<td>5.68(4)</td>
<td>1.107(12)</td>
<td>6.976(4)</td>
</tr>
<tr>
<td>insert</td>
<td>4.61(2)</td>
<td>5.02(2)</td>
<td>1.087(10)</td>
<td>4.528(4)</td>
</tr>
<tr>
<td>total</td>
<td>21.7(1)</td>
<td>23.9(1)</td>
<td>1.097(12)</td>
<td>25.08(1)</td>
</tr>
</tbody>
</table>

## PERFORMANCE: APPLES VS APPLES (I.E., 64-BIT STATIC)

<table>
<thead>
<tr>
<th></th>
<th>Irix</th>
<th></th>
<th>Mungi</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>good</td>
<td>bad</td>
<td>good</td>
<td>bad</td>
</tr>
<tr>
<td>lookup</td>
<td>7.367</td>
<td>7.169</td>
<td>7.452</td>
<td>0.973</td>
</tr>
<tr>
<td>forward traverse</td>
<td>5.904</td>
<td>6.085</td>
<td>6.079</td>
<td>1.031</td>
</tr>
<tr>
<td>backward traverse</td>
<td>6.796</td>
<td>6.992</td>
<td>6.991</td>
<td>1.029</td>
</tr>
<tr>
<td>insert</td>
<td>4.755</td>
<td>4.724</td>
<td>4.801</td>
<td>0.993</td>
</tr>
<tr>
<td>total</td>
<td>24.822</td>
<td>24.970</td>
<td>25.323</td>
<td>1.006</td>
</tr>
</tbody>
</table>
CONCLUSIONS

 mogul’s dynamic linking overhead is significantly lower than Irix’s
 mogul’s higher overhead is inherent in multi-address-space approach:
   ➔ Irix cannot guarantee that library can always be mapped to same address.
   ➔ Irix cannot execute “in-place”.
   ➔ Irix cannot avoid using position-independent code.

• Compare quickstart facility in Digital Unix [DEC94]:
  allocate libraries at fixed addresses.

 mogul’s Single-address-space system reduce costs.
Mungi: Recent Achievements

- Source release
- Component system for extensibility
- Posix emulation
- Driver framework
  - user-level drivers (serial, display, IDE, Ethernet on Alpha)
- New thread model
  - integrated with normal Mungi access control for memory
- Ported to portable L4 kernel (Pistachio)
  - runs on Itanium, Alpha, MIPS
- Persistence
Present Mungi-related activities:

- Fast I/O system for Mungi (based on LFS idea)
- Self-hosting
- User/login model
- SMP
- Distribution of address space
Present Mungi-related activities:

- Fast I/O system for Mungi (based on LFS idea)
- Self-hosting
- User/login model
- SMP
- Distribution of address space
- Run on my desktop/laptop
Other OS Research at UNSW

- Linux/IA-64
- efficient sharing
- SASOS
- Mungi
- SMP scheduling
- 64-bit
- page tables
- high-performance
- log-structured
- swap file system
- L4 nucleus
- MIPS
- Alpha
- StrongARM
- SPARC
- L4 Linux
- embedded systems
- CPU simulator
- Sulima
- PLEB
- simulator
- 64-bit
- page tables
- L4 Linux
- SMP scheduling
References


