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

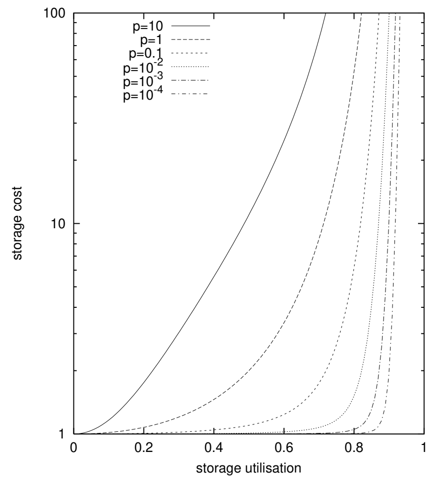
- ➜ Every object is associated with a *bank account*.
- ➜ "Rent" is periodically collected from account for associated objects.
- ➜ Regular "income" is periodically deposited into bank accounts.
- ➜ Overdrawn accounts prevent further creation of persistent objects ⇒ forces users to clean up.
- ➜ "Tax"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

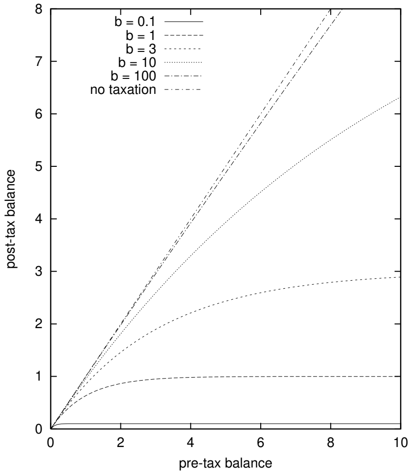
A: Market approach: adjust rent to demand!



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!



Linking in Mungi

Inherent features of a SASOS:

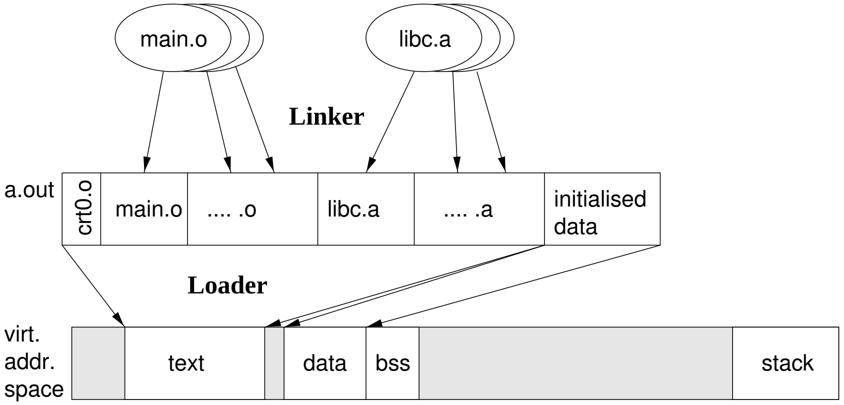
- ➜ data have unique address, *independent of context*,
- ➜ a particular address *always* refers to the same data.

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 `static`s?

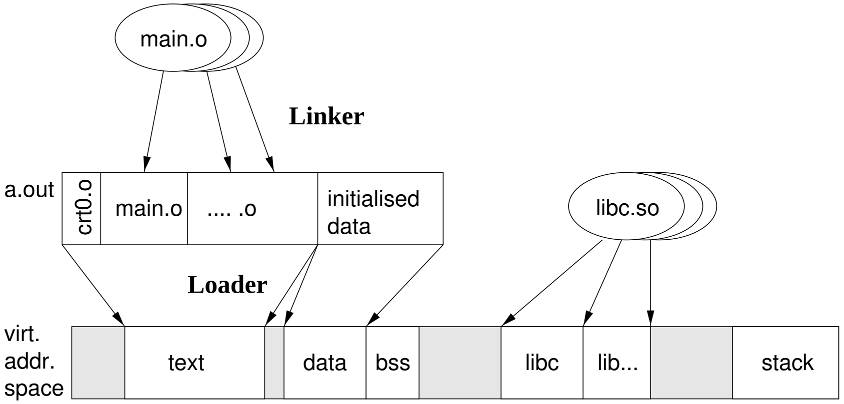
⇒ Need to re-think linking.

Static linking in UNIX



☛ all symbols resolved by linker

Dynamic linking in UNIX

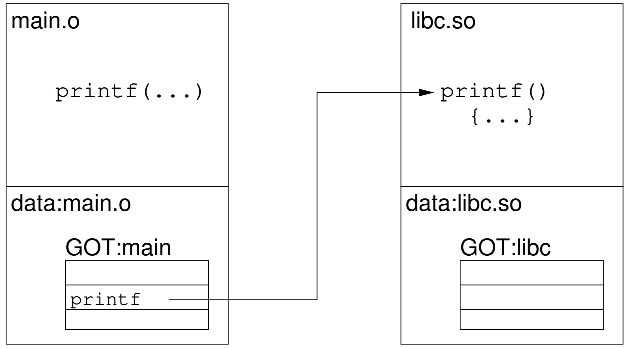


☛ some symbols resolved by loader

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



- ➜ 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.

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).

- ☛ 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.

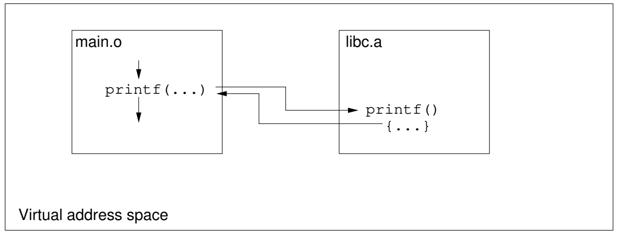
⇒ GOT cannot be allocated at a fixed address.

⇒ 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.
- 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

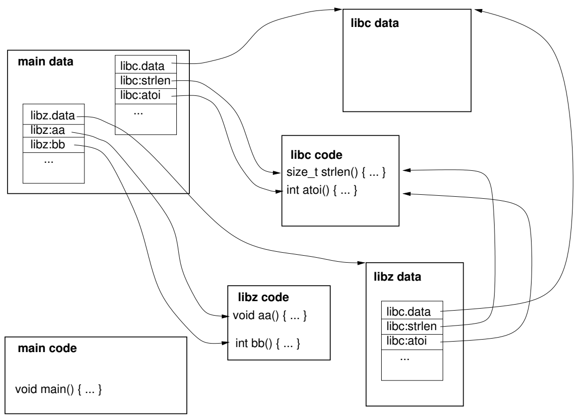
- Same address ⇒ same data in a SASOS

⇒ 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
...
```

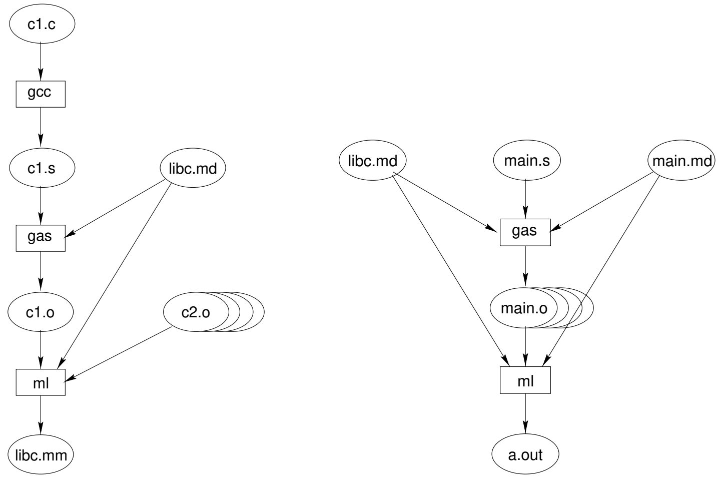
main.mm:

```
[IMPORTS]
libc.mm

[EXPORTS]

[OBJECTS]
main.o
sub.o
...
```


Code preparation process



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

	Irix/32-bit/SGI-cc			*Mungi/64-bit/GCC*		
	static	*dynamic*	*dyn/stat*	*static*	*dynamic*	*dyn/stat*
lookup	7.26(3)	8.02(3)	1.104(10)	7.568(6)	8.199(4)	1.083(3)
f. trav.	4.77(3)	5.17(4)	1.084(15)	6.013(6)	6.040(3)	1.004(6)
b. trav.	5.13(2)	5.68(4)	1.107(12)	6.976(4)	7.011(4)	1.005(1)
insert	4.61(2)	5.02(2)	1.087(10)	4.528(4)	4.755(3)	1.051(1)
total	21.7(1)	23.9(1)	1.097(12)	25.08(1)	26.00(1)	1.037(1)

Performance: Apples vs Apples (i.e., 64-bit static)

	Irix	*Mungi*		*Mungi/Irix*	
		good	*bad*	*good*	*bad*
lookup	7.367	7.169	7.452	0.973	1.012
forward traverse	5.904	6.085	6.079	1.031	1.030
backward traverse	6.796	6.992	6.991	1.029	1.029
insert	4.755	4.724	4.801	0.993	1.010
total	24.822	24.970	25.323	1.006	1.020

Conclusions

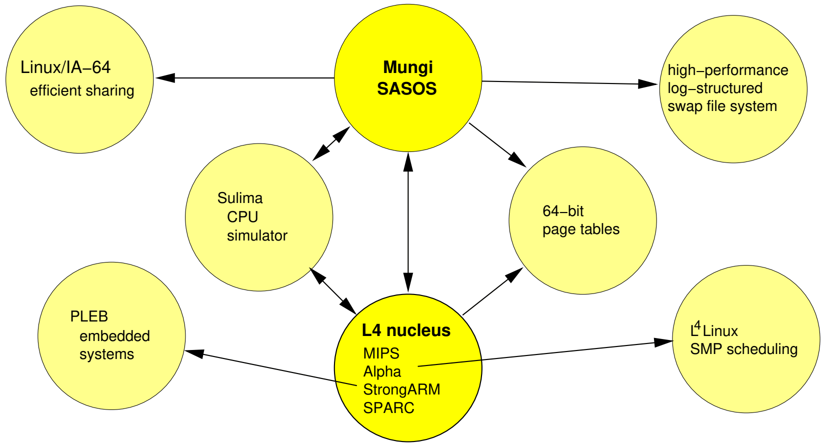
- Mungi's dynamic linking overhead is significantly lower than Irix'
- Irix's higher overhead is *inherent* in multi-address-space approach:
  - ➜ Irix cannot guarantee that library can always be mapped to same address.
  - ⇒ Irix cannot execute “in-place”.
  - ⇒ Irix cannot avoid using position-independent code.
- Compare *quickstart* facility in Digital Unix [DEC94]: allocate libraries at fixed addresses.
- Single-address-space system reduce costs.

Present Mungi-related activities:

- Fast I/O system for Mungi (based on LFS idea).
- Driver framework.
- Persistence.
- Distribution of address space.
- SASOS-techniques for dynamic linking in IA-64 Linux.
- SMP and high-performance server features.
- Component object system and secure OS extensibility.

... and a source release (finally!)

Other OS Research at UNSW



References

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

[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, November 1994.

[DEC94] Digital Equipment Corp. *DEC OSF/1 Programmer's Guide*, 1994. Order No AA-PS30C-TE.

[DH99] Luke Deller and Gernot Heiser. Linking programs in a single address space. In *Proc. 1999 USENIX Technical Conf.*, pages 283–294, Monterey, Ca, USA, June 1999.

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

29:289–299, 1986.

[Ros95] Timothy Roscoe. *The Structure of a Multi-Service Operating System*. PhD thesis, University of Cambridge Computer Laboratory, April 1995. TR-376, URL http://www.cl.cam.ac.uk/-ftp/papers/reports/TR376-tr-multi-service-os.ps.gz.