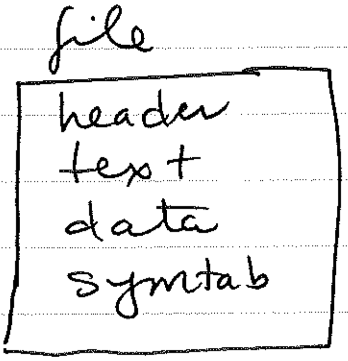
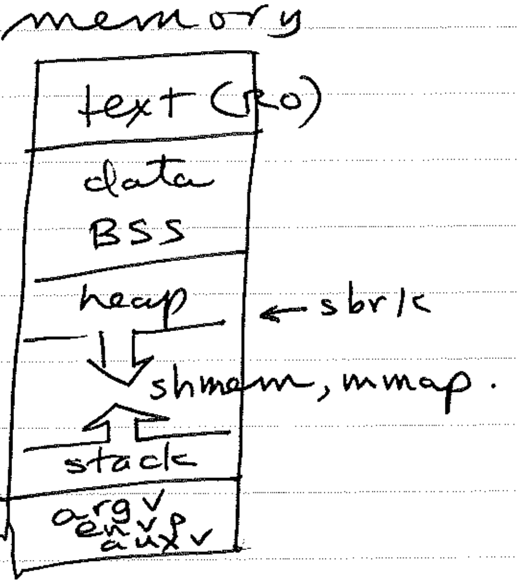
Linking & Loading

Classical unix a.out format

nm
strip





fork() - dups process
memimage
data + BSS+stk = COW

vfork() - suspends parent
until child exec or exit

execve() - relse addr space
- keeps open files
- alloc stk stg
- loads a.out
- jumps to entry address

execlp execl execle
↓ ↓ ↓
execvp → execv → execve

file id by magic#
a.out = assembler output (PDP11 branch insn)
usual * ELF = executable & linkable fmt
COFF = common object file format
DWARF = debugging with attributed record fmt.
jar = java archive (0xcafebabe)
shell = #!/path to interpreter

Link (2)

(ld) linker - combine obj files into exec image

(execve) loader - loads pgm from disk into mem ⇒ process.

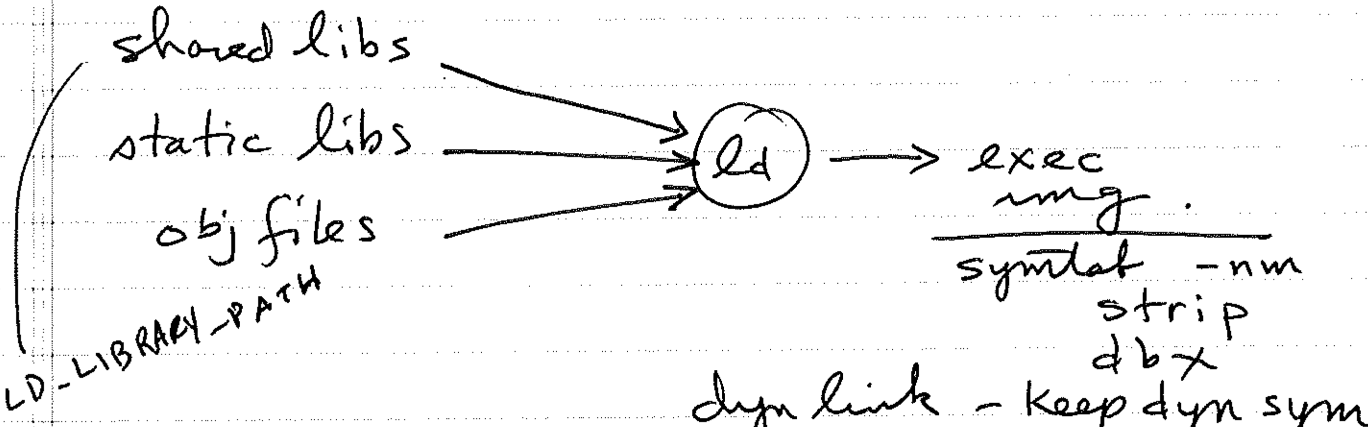
relocation - changing addresses to move to other areas mem.

old - link & load

with VM - new (assign abs V.adr)

symbol resolution -

def one obj; undef others.

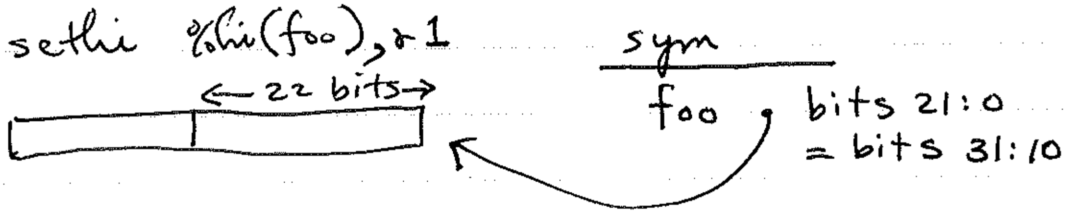


dyn link - keep dyn sym.

Reloc & code mod

- asm generates unreloc addrs (rel start of segment)
- also reloc dict

ex sethi %hi(foo),r1



(CH-2)

Architecture   Link(3)

need ABI spec.

ch 3 - machine interface

API

struct layout

- data rep
- big/little endian/align req.
- registers, stack frame.
- call/return conventions
- dyn stack?

ch5 - prog load & dyn link

addresses: absolute
PC-relative
reg ± offset

virtual mem?
- if not loader must adjust addresses
- or must run in fixed loc.

mapped files
- exec faults in pages from exec
- demand zero for BSS
- fault from swap

shared libs

PIC = position independent code.
- must run any location
- use relative jumps
-

CH -3

Link (4)

Object Files

Header - magic #, list of segments & sizes.

Text - object code.

Data - RO data, RW data, ZW data (BSS)

relocation dict

global symbols (def & undef)

-g debugging info - local vars
source files & lines

Formats

a.out

text ← page boundary

data ← no boundary

BSS

header: magic :
textsiz :
datsiz :
bsssiz :
symsiz .
entryadr .
txtreloc size .
datreloc size .

history

PDP-11

magic # 0407 (octal)

br pc+7words

— newly alloc mem auto init = 0

— ∴ array of ∅'s in bss.

int a [] = {0,0,0,0}

int a [4]; — better.

text pages $\xrightarrow[\text{RO map}]{}$

data pages $\xrightarrow[\text{COW map.}]{}$

relocatable a.out

- linkable files
- relocation table
  - entry: address - within module.
    - offset from start of module.
    - bit field spec.
    - is it external def?
    - PC rel?
    - name index
  - move module → adjust all addrs.
- symbol table
  - arb long strings
  - rep by symtab offset.

a.out → COFF → ELF

■ Unix ELF
- allows dyn linking
- allows C++
- debug fmt = DWARF

3 Kinds:
- relocatable - created by compilers & as.
- executable - cre by linkers.
- shared = library files

ELF = set of logical sections
each section ⇒ segment

Link (6)

ELF
extends T+D+B.

magic # [4] = "\177 ELF" = "~ELF"

177
01111111

reloc file = collection of sections
exec file = collection of segments.

sections: text (PROG)
data
rodata = read only
bss
rel.text
rel.data
rel.rodata
} reloc info for sections.
init
fini
} startup & finishup code for C++ classes.
symtab
dynsym
} symtab
strtab
dynstr
} strings
got globaloffset
plt proc linkage
.debug syms
.line = line #s to source.
.interp = path to #!

MIPS
short data, long data
short bss, long bss

pseudo
UNDEF
COMMON = common blocks
ABS = abs syms

ELF exec

= collection of segments
for mmap

= similar set of names.

each seg is concat of input sections

mapped text seg = header + text + rodata

data seg = rw data

ext by BSS via demandzero

ext to heap by sbrk()

~~init & fini calls~~

init & fini in text seg

-called befor/after main

ordering can be critical

init - C++ static ctors

fini - C++ static dtors

ocaml - ordered by module
all global code.

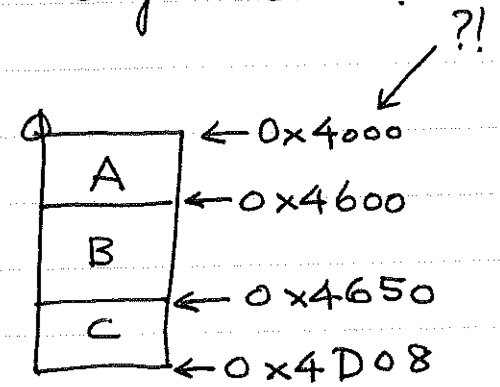
CH-4

Storage Allocation

- static.

set of like sections → segment.

ex: A size 0x600
B size 0x4F
C size 0x701 } →



align each section to max hardware align SPARC = 8 even if not need. Pentium: align load is faster

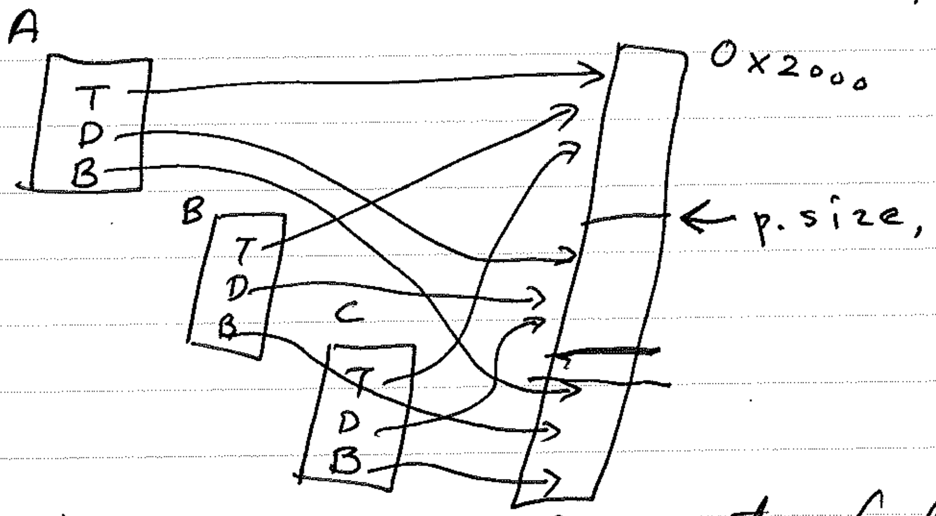
not page bndy

reloc as concat them together.

Fortran COMMON blocks overlaid - sized by largest.

Collect all text seg together
" " data " "
" " bss "

bss not in reloc or exec, but alloc demand zero.



text seg round up to full page
data+bss start next page bndy
& round up. pg size.

Fortran IV — no dyn stg alloc.

- had to statically alloc all data
  ∴ tweak arrays & recompile
- share via common
- modules declare common arrays dim 1
- BLOCKDATA section actually alloc common.
  just recompile this

COMMON → BSS

C++ duplicates

- virtual fn tables
- templates ⇒ macros

~~recompile template ∀~~

∀ file ∀ class ⇒ compile VFT

∀ file ∀ type ⇒ compile template and VFT.

/SUNWspro/CC
→ uses subdir to cache compilations

Initializers & Finalizers

C: atexit(...)

C++: static vars' ctors/dtors.
- init seg = list of ptrs to startup.
- fini seg =

ordering problem in C++

ocaml → do ordering by linker order
can't have topological dependency cycle

CH-5 Symbol Mgmt

Names

- global syms def
- global sym ref undef
- seg names
- local syms for debuggers
- line # info

symtab

- like compiler but flat
- name max len?
- store hash in table
    - do strcmp *only* when hash matches

```
struct sym { char *name;
             int hashvalue;
             struct sym *link
           }
```

global sym table

each sym — def *exactly* once from one module
— ref *.

symbol resolution
- 2nd pass

refs
- data ptr
- in insn
- multiple insns

```
ex.   sethi %hi(X), g1
      or    g1, %lo(x), g1
        22 bits     10 bits
```

__Special syms__

etext - end of text
edata - end of data
end - end of BSS
__CTOR_LIST__

__Name Mangling__

- avoid collisions : user ids __vs__ gensyms

Cstyle :

old _main new main
gensym raw .gensym

C++ encoding : type & scope

V vs C::V

f(int) vs f(float) oper>>

nm

externals: foo → foo

func (float, int, unsgn char)

→ func__FfiUc

classnames: #&name. ex: 4Pair

First::Second::Third → Q35First6Second5Third

cl::fn(void) → fn__2clFv

* → __ml

|= → __aor

⇒ link time type checking

Debugging info

- line # info.
- types, locations of vars
- each fun local var map.

- duplicate header files

#include <iostream> X 20 copies

strip.

CH-6

Libraries

Link(14)

library files: archive formats

symtab → SEQ of obj files

- saves space on disk
- each fn sep obj

'.a' suffix

'ar' cmd

linkers make pass over libs.

-lm ⇒ /usr/lib/libm.a
/usr/lib/libm.so

option must be last.

w

CH=7 Relocation

1^{st} pass - lay out position of all sections in segment
arranges segment addresses

2^{nd} pass - fix up storage addrs of all global syms.
data adrs = absolute
code = absolute | PC relative.

load (exec) relocation
- only if sys has no Virt mem.

- call X → needs rel diff X.
- handle partial addresses

sethi_{22} lo_{10} — hi-ord 30 bits.

C~~H=8~~ ~~Loadi~~

CH=8

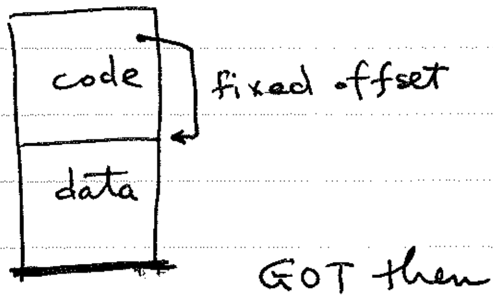
Loading & Overlays

load :- read header
- allocate addr space
- mmap program
- zero out bss
- create stack.

overlays : used for old (non-VM) systems.
- let pager/swapper handle.

PIC = position independent code
- within module all code refs relative
- data at fixed offset from code

linker creates GOT = global offset table.



```
     call L
L:   add  $GOT, ra
           ↗
        offset.
```

GOT then points at data

CH=9

Shared Libraries

Link (17)

- static lyink → put in exec
- shared → loaded @ exectime
  or dyn.
- exec → lib
- at load time
  Startup code maps them
  into address space.
- static linked shared

- reserve address space.
  ex: VAX 0x00— to 0x7F— user
  0x80— to 0xFF— shared.

startup code

- finds libs & maps them
  into address space
- shared or COW
- may be in: OS
  executable
  dyn linker.

- must be avail when pgm run
- if changed?
- link time: sym bound to adresses
- can't change entry addresses.

shared static lib

- header
- init routine
- JUMP TABLE
- code
- global data
- private data

2 libs – shared lib itself
– stub lib linked into exec.

```
JUMP TABLE:   read:  jmp read'
              write: jmp write'
```

change code doesn't affect jmp table
can't rearrange jmp table.

/usr/lib/libc.a

Malloc hack – your own version
~~you have v~~ – but libs still call original

∴
```
extern void *(*mallocptr)(size_t);
#define malloc(s) ((*mallocptr)(s))
```

in lib init
```
    #undef malloc
    mallocptr = &malloc
```

stub in app, not lib, so: Link (19)

if app has malloc, it's used
else lib's malloc is used

Linking and Loading

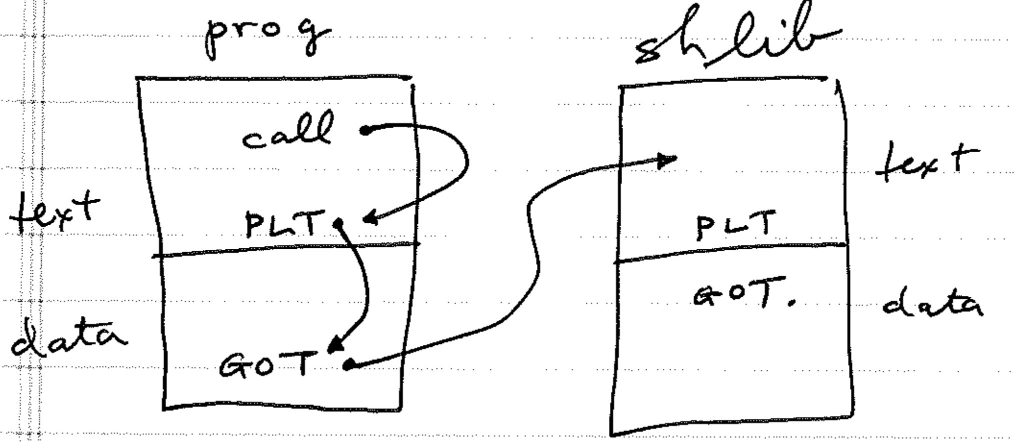
Dynamic Linking & Loading

fetches libs from std /usr/lib
or compiled in -L libs
also: setenv LD_LIBRARY_PATH.

advantage: easier create than static shared
easier update
semantics more like unshared.

disadv: slower: link done every run.
larger - need symtab
libs must be backward compat

uses PIC
needs GOT = global offset table
PLT = procedure linkage table
- "lazy" evaluation



PLT

in exec

```
PLTØ :  unimp
        unimp
        unimp
PLT1 :  unimp
        unimp
        unimp
          :
PLT101  sethi (.-PLTØ),g1
        ba,a  PLTØ
        nop
PLT102  sethi (.-PLTØ),g1
        ba,a  PLTØ
        nop
```

in mem

```
PLTØ :  save %sp,-64,%sp
        call dynlink
        nop
PLT1    .word identification
        unimp
        unimp

PLT101 : sethi (.-PLTO),g1
         sethi %hi(printf),g1
         jmpl g1+%lo(printf),g0
PLT102 : sethi (.-PLTO),g1
         sethi %hi(fopen),g1
         jmpl g1+%lo(fopen),g0
```

1. at load time, dyn linker alters PLTØ and PLT1 to xfer ctl to it
"identific" word is a message to itself.

2. prog calls "printf", which is really "PLT101"
sethi puts dist to PLTØ into $g1_{[31:10]}$
$g1 = (101 * 12) \ll 10$
branches to PLTØ calls dynlink with new frame

3. using ident, it finds data structs.
$(g1 \gg 10)/12$ gives index where "printf" info is

4. dynlink alloc space for "printf" text & data, loads it, unwinds stack, fixes PLT 101, jumps to "printf"

FIX NOTE: fix PLT 101

- must update re-entrant in case interrupt / signal

atomic update: 3rd word.

① ⟹ sethi
ba,a
nop → jmpl } annuls delay insn.

② ⟹ sethi
sethi
jmpl } 1st sethi irrelevant
jmpl delay slot uses next PLT sethi
→ OK because jmpl addr computed

/usr/lib/ld.so.1
/usr/lib/libc.so.1

libs