

# Dynamic Linking of Software Components

Michael Franz  
For CMSC 838Y  
Spring 2003

## Five implementation methods

- **Runtime table lookup**
  - External references indirected through a table
- **Load-time table modification**
  - External references rewritten at load-time
- **Runtime code modification**
  - External references rewritten while running
- **Load-time code generation**
  - Intermediate code compiled to machine code at load time
- **Full load-time compilation**
  - Source code compiled at runtime

## Example

### Module C

```
extern void q(...);
extern void r(...);
void s() {...}
void t() {
  q(...);
  s(...);
  r(...);
  q(...);
}
```

### Module L

```
void p() {...}
void q() {...}
```

### Module M

```
void r() {...}
```

## Runtime Table Lookup

### Object Code for module C

```
code for procedure s:
lbl-a: ...
code for procedure t:
lbl-b: ...
lbl-c: indirect call via link0
relative branch via lbl-a
lbl-d: indirect call via link1
lbl-e: indirect call via link0
```

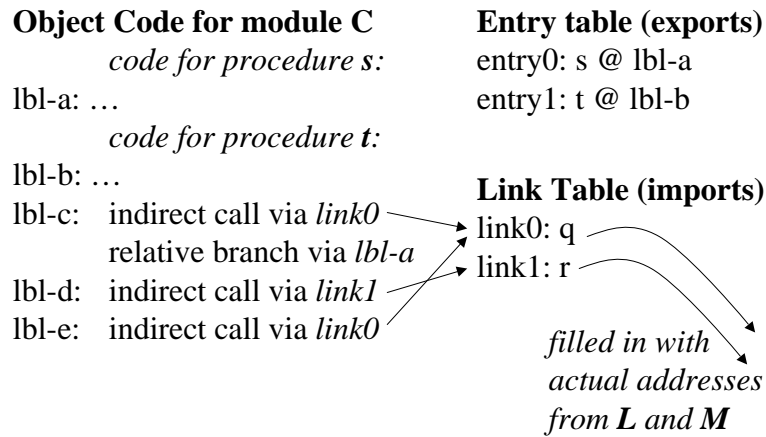
### Entry table (exports)

```
entry0: s @ lbl-a
entry1: t @ lbl-b
```

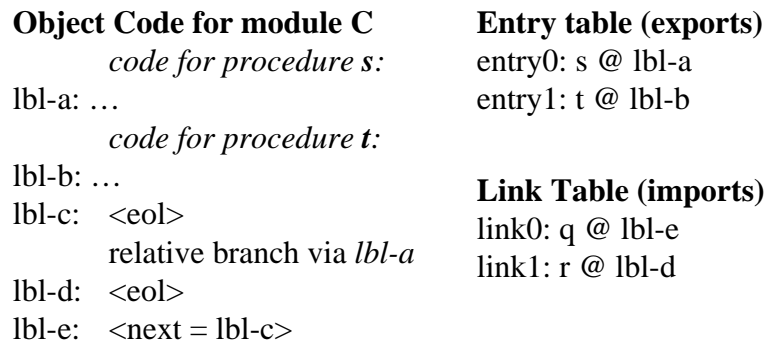
### Link Table (imports)

```
link0: q
link1: r
```

## Runtime Table Lookup



## Load-time Code Modification



## Load-time Code Modification

<b>Object Code for module C</b>	<b>Entry table (exports)</b>
code for procedure <i>s</i> :	entry0: <i>s</i> @ lbl-a
lbl-a: ...	entry1: <i>t</i> @ lbl-b
code for procedure <i>t</i> :	
lbl-b: ...	<b>Link Table (imports)</b>
lbl-c: <eol>	link0: <i>q</i> @ lbl-e
relative branch via <i>lbl-a</i>	link1: <i>r</i> @ lbl-d
lbl-d: <eol>	
lbl-e: <next = lbl-c>	

*actual call-sites  
filled in with *addrs*  
from **L** and **M***

## Load-time Code Modification

<b>Object Code for module C</b>	<b>Entry table (exports)</b>
code for procedure <i>s</i> :	entry0: <i>s</i> @ lbl-a
lbl-a: ...	entry1: <i>t</i> @ lbl-b
code for procedure <i>t</i> :	
lbl-b: ...	<b>Link Table (imports)</b>
lbl-c: relative branch via <i>q</i>	link0: <i>q</i> @ lbl-e
relative branch via <i>lbl-a</i>	link1: <i>r</i> @ lbl-d
lbl-d: relative branch via <i>r</i>	
lbl-e: relative branch via <i>q</i>	

*actual call-sites  
filled in with *addrs*  
from **L** and **M***

## Comparison

- Runtime table lookup
  - Code is position independent
    - Can be relocated at runtime; only requires changing the link table
    - Facilitates shared libraries (sharing code among separate processes)
- Load-time code modification
  - Eliminates extra indirection

## Runtime Code Modification

### Object Code for module C

*code for procedure s:*

lbl-a: ...

*code for procedure t:*

lbl-b: ...

lbl-c: SVC <q, next=lbl-e>  
relative branch via *lbl-a*

lbl-d: SVC <r,next=lbl-d>

lbl-e: SVC <q,next = lbl-c>

### Entry table (exports)

entry0: s @ lbl-a

entry1: t @ lbl-b

*SVC is a special trap instruction that goes to a separate linking routine*

## Runtime Code Modification

### Object Code for module C

*code for procedure s:*

lbl-a: ...

*code for procedure t:*

lbl-b: ...

lbl-c: relative branch via q

relative branch via *lbl-a*

lbl-d: SVC <r,next=lbl-d>

lbl-e: relative branch via q

### Entry table (exports)

entry0: s @ lbl-a

entry1: t @ lbl-b

*Linking q*

## Runtime Code Modification

### Object Code for module C

*code for procedure s:*

lbl-a: ...

*code for procedure t:*

lbl-b: ...

lbl-c: relative branch via q

relative branch via *lbl-a*

lbl-d: relative branch via r

lbl-e: relative branch via q

### Entry table (exports)

entry0: s @ lbl-a

entry1: t @ lbl-b

*Linking p*

## Benefit

- Amortizes load-time cost over course of running program.
  - Useful when linking very large libraries from which only a few symbols are required.
  - But (one-time) operations expensive
    - requires special trap
    - needs instruction cache flush
- Can also be implemented using a procedure linkage table (as in ELF)
  - requires extra space in object file, but no special instruction

## Load-time code generation

- Compile to “intermediate representation” rather than to machine code
  - more compact than machine code
  - platform-independent
- Generate code at load-time
  - Or more generally at runtime, on demand
- Essentially what’s happening with JVM JIT compilers today

## Analysis

- Benefit
  - Smaller size → less I/O → faster start time
  - Platform independence → wider availability
- Drawback
  - Fast code generation → lower performance
  - Shared libraries → less I/O benefit
  - Code generator increases the TCB

## Java

- Essentially a combination of “lazy linking” (runtime code modification) and runtime code generation.
- But
  - bytecodes not that compact (see Pugh `99 *Compressing Java Class files*)
  - Bytecodes not very source language independent
    - Not good for functional or imperative source languages
    - Witness: easy to decompile bytecode back to Java source
- Is .NET the solution?

## Full load-time Compilation

- Just distribute source and recompile each time before you run (!)
- Not too practical
  - Compilation too slow
  - Doesn't protect intellectual property
    - But Java class files really don't either

## Questions

- How have things changed since 1997? How is linking relevant today?
- Where are things going?
- What are the issues enabling or preventing different technologies?

This document was created with Win2PDF available at <http://www.daneprairie.com>.  
The unregistered version of Win2PDF is for evaluation or non-commercial use only.