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

**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

Filled in with actual addresses from L and M

Load-time Code Modification

**Object Code for module C**

- code for procedure s:
  - lbl-a: …
- code for procedure t:
  - lbl-b: …
  - lbl-c: <eol>
  - relative branch via lbl-a
  - lbl-d: <eol>
  - lbl-e: <next = lbl-c>

**Entry table (exports)**

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

**Link Table (imports)**

- link0: q @ lbl-e
- link1: r @ lbl-d
Load-time Code Modification

Object Code for module C

code for procedure s:
   lbl-a: ...

code for procedure t:
   lbl-b: ...
   lbl-c: <eol>
       relative branch via lbl-a
   lbl-d: <eol>
   lbl-e: <next = lbl-c>

Entry table (exports)
   entry0: s @ lbl-a
   entry1: t @ lbl-b

Link Table (imports)
   link0: q @ lbl-e
   link1: r @ lbl-d

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

Load-time 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

Link Table (imports)
   link0: q @ lbl-e
   link1: r @ lbl-d

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

<table>
<thead>
<tr>
<th>Code for procedure s:</th>
<th>Code for procedure t:</th>
</tr>
</thead>
<tbody>
<tr>
<td>lbl-a: ...</td>
<td>lbl-b: ...</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>SVC &lt;q, next=lbl-e&gt;</td>
<td>SVC &lt;r, next=lbl-d&gt;</td>
</tr>
<tr>
<td>relative branch via lbl-a</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>SVC &lt;q, next=lbl-c&gt;</td>
<td></td>
</tr>
</tbody>
</table>

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?