Multics: Dynamic Linking

Arvind Krishnamurthy
Spring 2004

Multics

- Today: look at Mutics VM
- CTSS: probably the first time-sharing system (1959-1965)
  - "Compatible Interactive Time Sharing System"
  - Introduced:
    - working online
    - storing information online
  - No protection, off-the-shelf hardware
  - 4 consoles running at 110 baud attached to an IBM machine
    - Two tape drives/user, swapped programs and data
  - Introduced interactive debugging, editors, command-line processors
  - Simple, few key ideas, throw out irrelevant, highly successful

Multics (MIT/Bell/GE)

- Multics: “second system effect”
  - Huge, complicated, tough to debug, terrible performance
- Designed around 1965
  - New hardware, new OS, new programming language
  - Multiple processes, separate address spaces, segmentation with paging
  - Take an interesting idea to the extreme (good research direction!)
    - Extreme sharing
    - Support sharing & dynamic linking
- Unix: third system
  - Understand the limits from the second system, step back, choose with taste, pick some key ideas

Key Ideas in Multics VM

- Combine virtual memory & file systems
  - Two ways to refer to data: (segment number, offset) and (file name, offset); segment is stored on disk or memory
  - Kind of like “mmap” for all data
- Fine-grain sharing
  - Multics took sharing to the extreme
  - Sharing at the level of segments
  - Process = many segments (data or code)
  - Individual library packages are shared; different subsets of processes share different libraries
- Dynamic linking
  - Segments can be “made known” at runtime
  - Share information and upgrade incrementally
- Autonomy (independent address space per. process)
  - Two different libraries might be at different addresses on different processes
  - Need that if we have to support fine-grain sharing

Static Linking Review

```
int y;
extern int z;

int foo() {
        y = 1;
        z = 2;
}
```

[foo.c]

```
000: move xxx, r1
004: store 1, (r1)
008: move xxx, r2
00C: store 2, (r2)
010: ret

RELOCATION TABLE:
(remember what addresses
  need to be changed)
y: 000
z: 008
```

[foo.s]

Static Linking Example (contd.)

```
int z;
extern int y;

int main() {
        y = 11;
        z = 12;
        bar();
}
```

[bar.c]

```
000: move xxx, r1
004: store 11, (r1)
008: move xxx, r2
00C: store 12, (r2)
010: jsr xxx
014: ret

RELOCATION TABLE:
y: 000
z: 008
bar: 010
```

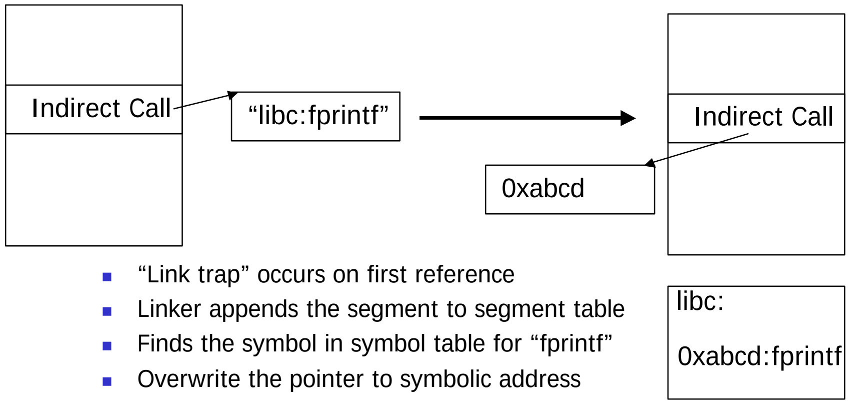
[bar.s]

Combined Result

```
000: move 100, r1
004: store 11, (r1)
008: move 104, r2
00C: store 12, (r2)
010: jsr 018
014: ret
018: move 100, r1
01C: store 1, (r1)
020: move 104, r2
024: store 2, (r2)
028: ret


100: (space for y)
104: (space for z)
```

Dynamic Linking: Step 1

- Resolving external references at runtime
- Use a level of indirection:
    - Initially symbolic references, later become memory references



- "Link trap" occurs on first reference
- Linker appends the segment to segment table
- Finds the symbol in symbol table for "fprintf"
- Overwrite the pointer to symbolic address
- Return back and retry the instruction

Rewrite Symbolic references

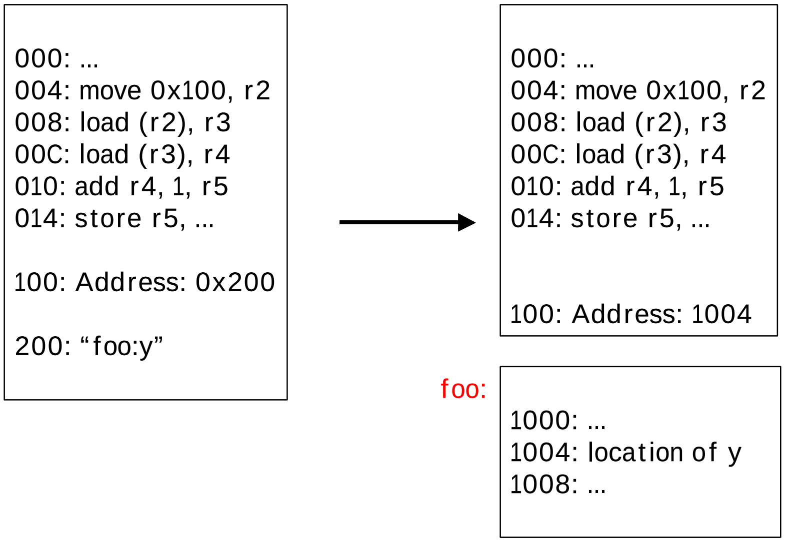
```
extern int y; /* “foo” */
int z;

int main() {
          ...
          z = y + 1;
          ...
}
```

```
000: ...
004: move 0x100, r2
008: load (r2), r3
00C: load (r3), r4
010: add r4, 1, r5
014: store r5, ...

100: Address: 0x200

200: “foo:y”
```

Has “symbolic ref” bit set to cause a link trap

- Add a level of indirection:
  - To prevent modifying code
  - To share pointer across many references

Rewrite Symbolic references



```
000: ...
004: move 0x100, r2
008: load (r2), r3
00C: load (r3), r4
010: add r4, 1, r5
014: store r5, ...

100: Address: 0x200

200: “foo:y”
```

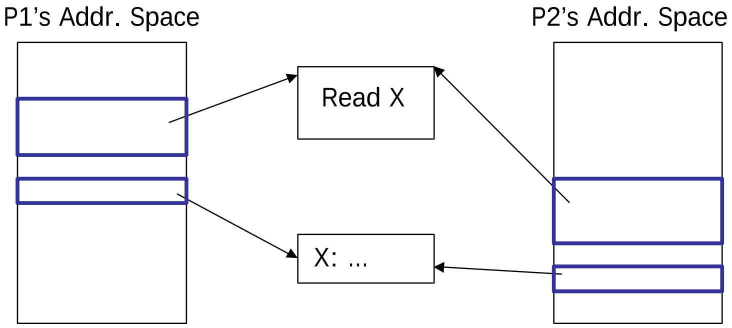
```
000: ...
004: move 0x100, r2
008: load (r2), r3
00C: load (r3), r4
010: add r4, 1, r5
014: store r5, ...


100: Address: 1004
```

foo:

```
1000: ...
1004: location of y
1008: ...
```

Sharing: Step 2

- Cannot modify code shared by different processes


P1’s Addr. Space

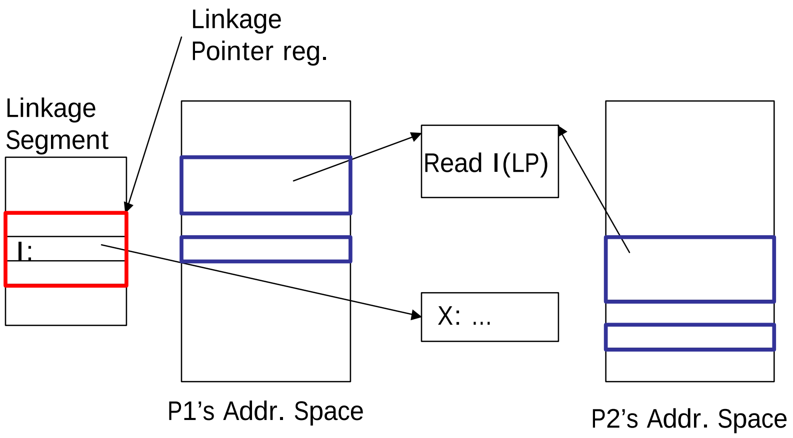
P2’s Addr. Space

Read X

X: ...

- Need per-process table for links
- Linkage section: all imports for a given segment, for a given process
- Linkage segment: collection of all linkage sections for a given process

Linkage Section


Linkage Pointer reg.

Linkage Segment

I:

Read I(LP)

X: ...

P1’s Addr. Space

P2’s Addr. Space

- Linkage section: links for all external references
- Layout of linkage section same across all processes
    - This is the reason why “I” is process-independent

Linkage Section Example

```
extern int foo::y;
extern int bar::z;

int progtest() {
        ...
        z = y;
        ...
}
```

[prog.c]

```
000: ...
004: load 0(LP), r2
008: load (r2), r3
00C: load 4(LP), r4
010: store r3, (r4)
```

```
Linkage section for "prog"
    Has 2 entries:

000: Address 100
004: Address 200

100: "foo::y"
200: "bar::z"
```

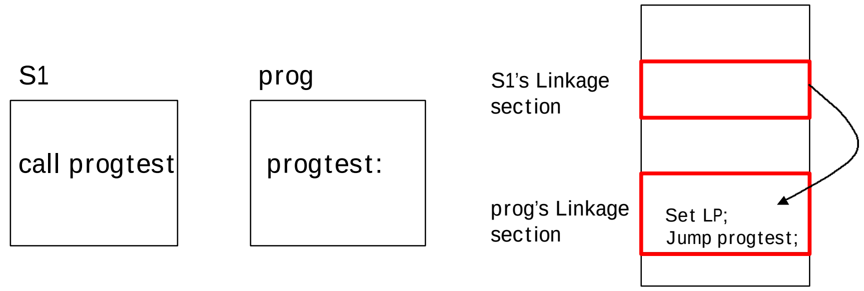
Process so far...

- When process refers to the “prog” segment
  - “link trap” happens
  - Make code segment “prog” known
  - Instantiate linkage section for “prog” in the linkage segment
    - Use symbol table, cross-reference list from the object file
- When the code segment refers to the data “foo::y”
  - “link trap” happens
  - foo’s segment is loaded and foo’s linkage section is instantiated
  - Modify address in linkage section for “prog” to point to “foo::y”
- Only problem left: how do you get the linkage pointer register point to the right place?

Step 3: Procedure call

- When PC is in segment, LP points to the segment's linkage section
- At every procedure call, change LP
- How to do this?

Procedure Call



S1

call progtest

prog

progtest:

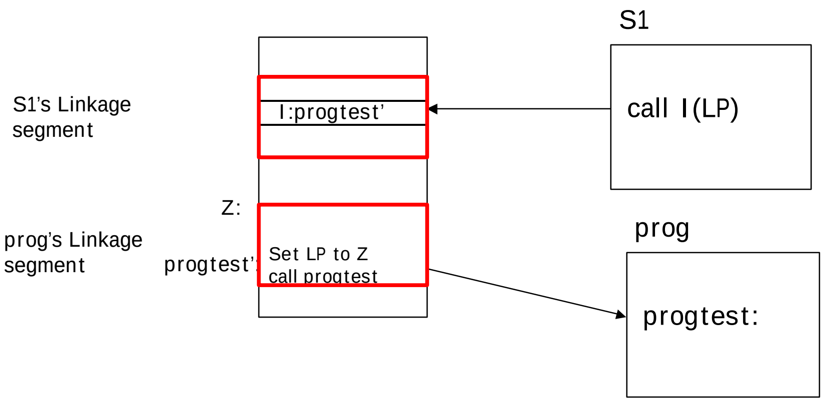
S1's Linkage section

prog's Linkage section

Set LP;
Jump progtest;

- When S1 calls prog::progtest
  - Change LP to point to prog's linkage section
  - Then, jump to progtest
  - Now progtest's references will go through prog's linkage section (for the current process only)

Procedure Call (contd.)



S1

S1's Linkage segment

l:progtest'

call l(LP)

Z:

prog's Linkage segment

progtest':

Set LP to Z
call progtest

prog

progtest:

- Note that location l in job1's linkage segment is initially symbolic
- Map code segment of prog
- Instantiate linkage section for prog with 2 instructions per exported procedure

Questions

- How many link traps does the following code generate:

```
S1::foo() {
    for (j=0; j<10; j++)
        call prog::progtest();
}
```

- How about the following code?

```
S1::foo() {
        call prog::progtest();
        …
        call prog::progtest();
}
```

- What happens when there is another segment R that calls progtest also?

Postscript

- Why so complicated?
  - Fine-grained sharing
  - Dynamic linking
  - Independent address spaces
- For the next 20 years, no one attempted dynamic linking and sharing at the same time
- Until MIT takes revenge:
  - MIT X-windows: megabytes of X toolkits
  - Need shared libraries
- Similar mechanisms are now standard in all major operating systems