Lecture 17

Andrea LaPaugh

April 9, 1996

Loading

Actual addresses in a program depend on starting address

Ex (Lecture 15 example revisited):

```
        LI     R0, 1               B001      00
        LI     R1, LENGTH          B112      01
        SUB    R1, R1, R0          2110      02
        LI     R2, TABLE           B20E*     03
        LI     R3, 0               B300      04
LOOP:   JLT    R1, DONE            610B*     05
        LOAD   R4, (R2+0)          9420      06
        ADD    R3, R3, R4          1334      07
        ADD    R2, R2, R0          1220      08
        SUB    R1, R1, R0          2110      09
        JMP    LOOP                5005*     0A
DONE:   LI     R1, SUM             B120*     0B
        STORE R3, (R1+0)           A310      0C
        HALT                       0000      0D
LENGTH: EQ  12                               --
TABLE:  SPACE LENGTH                         0E
                                             ...
                                             1F
SUM:    SPACE 1                              20
```

Operating System loads program where it can

Compiler can't know where prog. will be loaded

Object module produced by compiler marks addresses that need to be fixed

Separate Compliation

compile pieces of a program separately

bind them together to execute

Two advantages:

- don't have to recompile infequently changed
  parts of very large programs
- can share libraries of compiled code

declaration vs definition

Unix pragmatic strategy:

- declare each external object only once
  (in header file)
- ``include`` header file in every module
  that refers to the object
- one module defines each object

Example

Suppose have implementations for stack and queue

Interfaces (declarations)

```
stack.h
queue.h
```

Implementations

```
stack.c  (#include "stack.h")
queue.c  (#include "queue.h")
```

Have program `main.c` that uses
a stack and a queue

```
#include "stack.h"
#include "queue.h"
```

```
lcc -c stack.c
lcc -c queue.c
lcc -c main.c
lcc main.o stack.o queue.o
```

Compiling

Compiler produces OBJECT MODULE

* code
* addresses associated with functions
  and external variables defined within
* places where external variables are
  referenced

Compile each program as though it started at 0

Linking

LINK EDITOR

a program that resolves external references

combines object modules into executable code

(see "man lcc" and "man ld")

Concatenate object modules

gives start address for each

RELOCATE: add start address to each

instruction address field

RESOLVE: find actual addresses for

each external variable, fill in

at each place marked by the compiler

Problem: a.out still starts at 0

Linker can't know where program is to

be loaded when executed.

Still need loader

Multiprogramming

Operating system "kernel"

  keeps track of several programs

INTERRUPT:

  stop

  save PC somewhere "special"

  change PC

Interrupts are

  necessary to manage input-output devices

  useful for multiprogramming

OS allows several programs to "share" CPU by

  keeping table of "current" PC's for programs

  setting clock to interrupt periodically

Memory Limitations

Problem 1:

program needs more memory than machine has

direct solution: ``overlays``

Problem 2:

several programs need to share same memory

direct solution: divide up the memory

``Better`` solution to both problems:

all programs assume access to whole memory

each program actually uses a small portion

Virtual Memory

VIRTUAL MACHINES

simulate multiple copies of a machine on itself

each program thinks it has the whole machine

physical address space

how much memory is there?

limitation: $/bit cost

virtual address space

maximum amount of memory

an instruction could directly reference

limitation: address size (bits/instruction)

Paging

Widely-used method for implementing VM

Design hardware to ``trap'' all addresses

Keep virtual memory for each program on disk

Slow brute-force solution:

go to the disk for each memory reference

Divide into PAGES

Keep table with

* flag indicating if page is in memory
* relative position of page in memory

Make page size a power of two,

use leading bits of address for page name

Fetching from memory

Each memory reference

check if page is in memory

get it from disk if not

use page table to reset upper address bits

Each page brought in has to REPLACE another

``page replacement strategies``

still being studied, invented

Virtual Memory Example

32-bit adresses

2^{12}-byte pages

2^{24} bytes (16Megabytes) of physical memory

How many bytes for page number?

How many pages can be in physical memory?

Example continued

Suppose the following is a fragment
of the page table (addresses in hex):

page no.	in memory	physical location
9F9FA	yes	AAA
9F9FB	no	disk location
9F9FC	no	disk location
9F9FD	yes	999

What happens for each of these memory accesses?

9F9FA001

9F9FB002

9F9FC003

9F9FB005

Size of Virtual Memory

16 bits is not enough

24 bits is not enough

32 bits is not enough

Is 64 bits enough?

18446744073709551616 $>$ 10^{19} different addresses

Need more sophisticated paging strategy,

page table would be too big

(How big is page table in example?)

multilevel

associative

Some big numbers

10^{20}: number of grains of sand on a beach

10^{27}: number of oxygen atoms in a thimble

2^{256}: number of electrons in the universe

Systems Programming Summary

Building layers of abstraction

- simplify programming process
- facilitate "sharing" of
  - scarce resources
  - other programs

General approach used

- does not vary much across systems
- developed as consensus of many
- embodies significant intellectual
  achievements
- seems simpler in hindsight than
  while it was being developed
- has had major impact