Liu Liu

Topic:
  - Assembly Programming
  - Storage
Recap

- Assembly Programming:
  - project2
  - Structure/ Array Representation
  - Structure Alignment
Today’s Topic

- Assembly Programming:
  - Call-chain
  - Factorial

- Storage:
  - RAM
  - Caching
  - Direct - Mapping
IA32/Linux Stack Frame

Current Stack Frame (“Top” to Bottom)
- Parameters for function about to call
  - “Argument build”
- Local variables
  - If can’t keep in registers
- Saved register context
- Old frame pointer

Caller Stack Frame
- Return address
  - Pushed by call instruction
- Arguments for this call
Register Saving Conventions

When procedure \texttt{yoo} calls \texttt{who}:

- \texttt{yoo} is the \textit{caller}, \texttt{who} is the \textit{callee}

Can Register be Used for Temporary Storage?

\texttt{yoo}:

\begin{verbatim}
  . . .
  movl $15213, %edx
  call who
  addl %edx, %eax
  . . .
  ret
\end{verbatim}

\texttt{who}:

\begin{verbatim}
  . . .
  movl 8(%ebp), %edx
  addl $91125, %edx
  . . .
  ret
\end{verbatim}
Register Saving Conventions

When procedure $yoo$ calls $who$:
- $yoo$ is the caller, $who$ is the callee

Can Register be Used for Temporary Storage?

Conventions
- “Caller Save”
  - Caller saves temporary in its frame before calling
- “Callee Save”
  - Callee saves temporary in its frame before using
IA32/Linux Register Usage

Integer Registers

- Two have special uses
  - %ebp, %esp
- Three managed as callee-save
  - %ebx, %esi, %edi
  - Old values saved on stack prior to using
- Three managed as caller-save
  - %eax, %edx, %ecx
  - Do what you please, but expect any callee to do so, as well
- Register %eax also stores returned value
Recursive Factorial

```c
int rfact(int x)
{
    int rval;
    if (x <= 1)
        return 1;
    rval = rfact(x-1);
    return rval * x;
}
```

Registers

- `%eax` used without first saving
- `%ebx` used, but save at beginning & restore at end

```
global rfact
.type
rfact,@function
rfact:
pushl %ebp
    movl %esp,%ebp
    pushl %ebx
    movl 8(%ebp),%ebx
    cmpl $1,%ebx
    jle .L78
    leal -1(%ebx),%eax
    pushl %eax
    call rfact
    imull %ebx,%eax
    jmp .L79
.align 4
.L78:
    movl $1,%eax
.L79:
    movl -4(%ebp),%ebx
    movl %ebp,%esp
    popl %ebp
    ret
```
Rfact Stack Setup

Entering Stack

\[
\begin{align*}
\text{rfact:} & \quad \text{pushl } %ebp \\
& \quad \text{movl } %esp, %ebp \\
& \quad \text{pushl } %ebx
\end{align*}
\]
```c
int rfact(int x) {
    int rval;
    if (x <= 1) {
        return 1;
    }
    rval = rfact(x-1);
    return rval * x;
}
```

Rfact Body

```
movl 8(%ebp),%ebx  # ebx = x
cmpl $1,%ebx      # Compare x : 1
jle .L78          # If <= goto Term
leal -1(%ebx),%eax # eax = x-1
pushl %eax        # Push x-1
call rfact        # rfact(x-1)
imull %ebx,%eax    # rval * x
jmp .L79          # Goto done
.L78:              # Term:
movl $1,%eax      # return val = 1
.L79:              # Done:
```

Registers

- `%ebx` Stored value of x
- `%eax`
  - Temporary value of x-1
  - Returned value from `rfact(x-1)`
  - Returned value from this call
leal -1(%ebx),%eax

```
x
Rtn adr
Old %ebp
Old %ebx

%eax  x-1
%ebx  x

pushl %eax

%eax  x-1
%ebx  x

Rtn adr
%ebp

Old %ebp
Old %ebx

%eax  x-1
%ebx  x

Rtn adr
%ebp

call rfact
```
Assume that \texttt{rfact}(x-1) returns \((x-1)!\) in register \%eax
Rfact Completion

```plaintext
movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
```
Summary

The Stack Makes Recursion Work

- Private storage for each *instance* of procedure call
  - Instantiations don’t clobber each other
  - Addressing of locals + arguments can be relative to stack positions
- Can be managed by stack discipline
  - Procedures return in inverse order of calls

IA32 Procedures Combination of Instructions + Conventions

- Call / Ret instructions
- Register usage conventions
  - Caller / Callee save
  - %ebp and %esp
- Stack frame organization conventions
Storage in Computer
Memory

Thus far, we have used a very simple model of memory

- Main Memory is a linear array of bytes that can be accessed given a memory address
- Also used registers to store values

Reality is more complex. There is an entire memory system.

- Different memories exist at different levels of the computer
- Each vary in their speed, size, and cost
Random-Access Memory (RAM)

Key features

- RAM is packaged as a chip.
- What does Random refer to?
  - Basic storage unit is a cell (one bit per cell).
  - Multiple RAM chips form a memory.

Static RAM (SRAM)

- Each cell stores bit with a six-transistor circuit.
- Retains value indefinitely, as long as it is kept powered.
- Relatively insensitive to disturbances such as electrical noise.
- Faster and more expensive than DRAM.

Dynamic RAM (DRAM)

- Each cell stores bit with a capacitor and transistor.
- Value must be refreshed every 10-100 ms.
- Sensitive to disturbances.
- Slower and cheaper than SRAM.
Read-Only Memory

• DRAM and SRAM are volatile memories
  – Lose information if powered off.

• Nonvolatile memories retain value even if powered off.
  – Generic name is read-only memory (ROM).
  – Misleading because some ROMs can be read and modified.

• Types of ROMs
  – Programmable ROM (PROM)
  – Erasable programmable ROM (EPROM)
  – Electrically eraseable PROM (EEPROM)
  – Flash memory

• Firmware
  – Program stored in a ROM
  • Boot time code, BIOS (basic input/output system)
  • Graphics cards, disk controllers.
Conventional DRAM Organization

d \times w\text{ DRAM:}

- dw total bits organized as d supercells of size w bits

16(sc) \times 8(\text{bit/sc}) \text{ DRAM chip}

![Diagram showing the organization of a DRAM chip with supercells and internal row buffer.](image)
Reading DRAM Supercell (2,1)

Step 1(a): Row access strobe (RAS) selects row 2
Step 1(b): Row 2 copied from DRAM array to row buffer
Reading DRAM Supercell (2,1)

Step 2(a): Column access strobe (CAS) selects column 1

Step 2(b): Supercell (2,1) copied from buffer to data lines, and eventually back to the CPU
Memory Modules

We’ve been discussing single DRAM Chips

Several DRAM chips are bundled into Memory Modules
  - SIMMS - Single Inline Memory Module
  - DIMMS - Dual Inline Memory Module

Source for Pictures: http://en.kioskea.net/contents(pc/ram.php3
Memory Module

addr (row = i, col = j)

- supercell (i,j)
- 64 MB memory module consisting of eight 8Mx8 DRAMs

DRAM 0
DRAM 7
bits 56-63
bits 48-55
bits 40-47
bits 32-39
bits 24-31
bits 16-23
bits 8-15
bits 0-7

64-bit doubleword at main memory address A

Memory controller

64-bit doubleword
Memory speeds

Processor Speeds: 1 GHz processor speed is 1 nsec cycle time.

Memory Speeds (50 nsec)

<table>
<thead>
<tr>
<th>DIMM Module</th>
<th>Clock Speed[MHz]</th>
<th>Bus Speed[ MHz]</th>
<th>Transfer Rate [MB/s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC1600 DDR200</td>
<td>100</td>
<td>200</td>
<td>1,600</td>
</tr>
<tr>
<td>PC2100 DDR266</td>
<td>133</td>
<td>266</td>
<td>2,133</td>
</tr>
<tr>
<td>PC2400 DDR300</td>
<td>150</td>
<td>300</td>
<td>2,400</td>
</tr>
</tbody>
</table>

Access Speed gap

- Instructions that store or load from memory
Machine Architecture
System/Memory Bus

A bus is a collection of parallel wires that carry address, data, and control signals.

Buses are typically shared by multiple devices.

Information passed through transactions.

Diagram showing:
- CPU chip
- ALU
- Register file
- System bus
- Memory bus
- Bus interface
- I/O bridge
- Main memory
Memory Read Transaction (1)

CPU places address A on the memory bus.

Load operation: `movl A, %eax`
Memory Read Transaction (2)

Main memory reads A from the memory bus, retrieves word x, and places it on the bus.

Load operation: movl A, %eax
Memory Read Transaction (3)

CPU read word x from the bus and copies it into register %eax.

Load operation: \texttt{movl A, %eax}
Memory Write Transaction (1)

CPU places address A on bus. Main memory reads it and waits for the corresponding data word to arrive.

Store operation: \texttt{movl \%eax, A}
CPU places data word y on the bus.

Store operation: \texttt{movl \%eax, A}
Memory Write Transaction (3)

Main memory read data word \( y \) from the bus and stores it at address \( A \).

\[
\text{Store operation: } \text{movl} \ %\text{eax}, \ A
\]
Memory Hierarchy (Review)

L0: registers

L1: on-chip L1 cache (SRAM)

L2: off-chip L2 cache (SRAM)

L3: main memory (DRAM)

L4: local secondary storage (local disks)

L5: remote secondary storage (distributed file systems, Web servers)

CPU registers hold words retrieved from L1 cache.

L1 cache holds cache lines retrieved from the L2 cache memory.

L2 cache holds cache lines retrieved from main memory.

Main memory holds disk blocks retrieved from local disks.

*Local disks hold files retrieved from disks on remote network servers.

Smaller, faster, and costlier (per byte) storage devices

Larger, slower, and cheaper (per byte) storage devices
Cache Memories

Cache memories are small, fast SRAM-based memories managed automatically in hardware

- Hold frequently accessed blocks of main memory

CPU looks first for data in L1, then in L2, then in main memory
Locality

Memory references are bunched together

- A small portion of address space is accessed at any given time

Cache this portion of the address space in faster memories
Types of Locality

Temporal locality
- Recently accessed locations will likely be accessed again in near future

Spatial locality
- Will likely access locations close to ones recently accessed in near future
Sources of Locality

Temporal locality
- Code within a loop
- Same instructions fetched repeatedly

Spatial locality
- Data arrays
- Local variables in stack
- Data allocated in chunks (contiguous bytes)
Why Is Locality Good?

Address the gap between CPU speed and RAM speed

Spatial and temporal locality implies a portion of overall address space can fit in high speed memory

CPU can access instructions and data from this high speed memory

Small high speed memory can make computer faster and cheaper

This is caching
Caching in a Memory Hierarchy

Smaller, faster, more expensive device at level k caches a subset of the blocks from level k+1.

Data is copied between levels in block-sized transfer units.

Larger, slower, cheaper storage device at level k+1 is partitioned into blocks.
General Caching Concepts

Program needs object d, which is stored in some block b.

Cache hit
- Program finds b in the cache at level k. E.g., block 14.

Cache miss
- b is not at level k, so level k cache must fetch it from level k+1. E.g., block 12.
- If level k cache is full, then some current block must be replaced (evicted). Which one is the “victim”?
  - Placement policy: where can the new block go? E.g., b mod 4
  - Replacement policy: which block should be evicted? E.g., LRU
Cache Miss

Cold (compulsary) miss
- Cold miss occurs when a memory location is accessed for the 1st time

Conflict miss
- Most caches limit blocks at level k+1 to a small subset of the block positions at level k.
  - E.g., Block i at level k+1 must be placed in block (i mod 4) at level k
- Conflict misses occur when the level k cache is large enough, but multiple data items all map to the same level k block.
  - E.g., Referencing blocks 0, 8, 0, 8, 0, 8, ... would miss every time

Capacity miss
- Occurs when the set of active data blocks (working set) is larger than the cache
Remember:

- Each level is a cache for previous
- Each level stores different sizes of information
The transfer unit between the CPU register file and the cache is a 4-byte block.

The transfer unit between the cache and main memory is a 4-word block (16 bytes).

The big slow main memory has room for many 4-word blocks.

The small fast L1 cache has room for two 4-word blocks.

The tiny, very fast CPU register file has room for four 4-byte words.
Cache Content

Cache: A smaller, faster storage device that acts as a staging area for a subset of the data in a larger, slower device.

You are essentially allowing a smaller region of memory to hold data from a larger region. Not a 1-1 mapping.

What kind of information do we need to keep:

- The actual data
- Where the data actually comes from
- If data is even considered valid
Cache Mapping

Multiple locations in memory map to same location in cache

In addition to content, cache must keep which entry it is actually caching
Finding data in cache

Part of memory address applied to cache

Remaining is stored as tag in cache

If tag matches, hit, use data

No match, miss, fetch data from memory
General Org of a Cache Memory

Cache is an array of sets.
Each set contains one or more lines.
Each line holds a block of data.

$S = 2^s$ sets
$s$: index bits

1 valid bit per line
$t$ tag bits per line
$B = 2^b$ bytes per cache block

$E$ lines per set

Cache size: $C = B \times E \times S$ data bytes
Addressing Caches

The word at address A is in the cache if the tag bits in one of the <valid> lines in set <set index> match <tag>.

The word contents begin at offset <block offset> bytes from the beginning of the block.
Direct-Mapped Cache

Simplest kind of cache

Characterized by exactly one line per set.

```
set 0: valid  tag  cache block
set 1: valid  tag  cache block
       ...        
set S-1: valid  tag  cache block
```

$E=1$ lines per set
Accessing Direct-Mapped Caches

Set selection

- Use the set index bits to determine the set of interest.
Accessing Direct-Mapped Caches

Line matching and word selection

- **Line matching**: Find a valid line in the selected set with a matching tag
- **Word selection**: Then extract the word

=1? (1) The valid bit must be set

(2) The tag bits in the cache line must match the tag bits in the address

(3) If (1) and (2), then cache hit, and block offset selects starting byte.
Why Use Middle Bits as Index?

**High-Order Bit Indexing**
- Adjacent memory lines would map to same cache entry
- Poor use of spatial locality

**Middle-Order Bit Indexing**
- Consecutive memory lines map to different cache lines
- Can hold C-byte region of address space in cache at one time
Direct-Mapped Cache Simulation

M=4 bit addresses
B=2 bytes/block, S=4 sets, E=1 entry/set

Address trace (reads):
0 [0000₂], 1 [0001₂], 13 [1101₂], 8 [1000₂], 0 [0000₂]

<table>
<thead>
<tr>
<th>t=1</th>
<th>s=2</th>
<th>b=1</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>xx</td>
<td>x</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Address</th>
<th>v</th>
<th>tag</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 [0000₂] (miss)</td>
<td>1</td>
<td>0</td>
<td>M[0-1]</td>
</tr>
<tr>
<td>8 [1000₂] (miss)</td>
<td>1</td>
<td>1</td>
<td>M[8-9]</td>
</tr>
<tr>
<td>13 [1101₂] (miss)</td>
<td>1</td>
<td>1</td>
<td>M[12-13]</td>
</tr>
<tr>
<td>0 [0000₂] (miss)</td>
<td>1</td>
<td>0</td>
<td>M[0-1]</td>
</tr>
<tr>
<td>1 [0001₂] (miss)</td>
<td>1</td>
<td>1</td>
<td>M[12-13]</td>
</tr>
</tbody>
</table>
Example: Direct mapped cache

Given: 32 bit address, 64KB cache, 32 byte block, byte-addressable memory

Q: How many sets, how many bits for the tag, how many bits for the offset?

<table>
<thead>
<tr>
<th>Set</th>
<th>Valid</th>
<th>Tag</th>
<th>Cache Block</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set 0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Set 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Set n-1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>