COMPILING OBJECTS
AND OTHER LANGUAGE IMPLEMENTATION ISSUES

Credit: Mostly Bryant & O’Hallaron
# Word-Oriented Memory Organization

- **Addresses Specify Byte Locations**
  - Address of first byte in word
  - Addresses of successive words differ by 4 (32-bit) or 8 (64-bit)

<table>
<thead>
<tr>
<th>32-bit Words</th>
<th>64-bit Words</th>
<th>Bytes</th>
<th>Addr.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addr = 0000</td>
<td>Addr = 0000</td>
<td></td>
<td>0000</td>
</tr>
<tr>
<td>Addr = 0004</td>
<td></td>
<td></td>
<td>0001</td>
</tr>
<tr>
<td>Addr = 0008</td>
<td></td>
<td></td>
<td>0002</td>
</tr>
<tr>
<td>Addr = 0012</td>
<td></td>
<td></td>
<td>0003</td>
</tr>
<tr>
<td>Addr = 0016</td>
<td></td>
<td></td>
<td>0004</td>
</tr>
<tr>
<td>Addr = 0020</td>
<td></td>
<td></td>
<td>0005</td>
</tr>
<tr>
<td>Addr = 0024</td>
<td></td>
<td></td>
<td>0006</td>
</tr>
<tr>
<td>Addr = 0028</td>
<td></td>
<td></td>
<td>0007</td>
</tr>
<tr>
<td>Addr = 0032</td>
<td></td>
<td></td>
<td>0008</td>
</tr>
<tr>
<td>Addr = 0036</td>
<td></td>
<td></td>
<td>0009</td>
</tr>
<tr>
<td>Addr = 0040</td>
<td></td>
<td></td>
<td>0010</td>
</tr>
<tr>
<td>Addr = 0044</td>
<td></td>
<td></td>
<td>0011</td>
</tr>
<tr>
<td>Addr = 0048</td>
<td></td>
<td></td>
<td>0012</td>
</tr>
<tr>
<td>Addr = 0052</td>
<td></td>
<td></td>
<td>0013</td>
</tr>
<tr>
<td>Addr = 0056</td>
<td></td>
<td></td>
<td>0014</td>
</tr>
<tr>
<td>Addr = 0060</td>
<td></td>
<td></td>
<td>0015</td>
</tr>
</tbody>
</table>
Where do addresses come from?

- The compilation pipeline

```
prog P
  : foo()
  : end P

P: push ...
  inc SP, x
  jmp _foo
  foo: ...

push ...
inc SP, 4
jmp 75
...
```

```
Library Routines

P: ...
push ...
inc SP, 4
jmp 75
...

jmpl 175
...
```

```
Library Routines

P: ...
push ...
inc SP, 4
jmpl 1175
...
```

Compilation → Assembly → Linking → Loading
Monomorphic & Polymorphic

- Monomorphic swap function
  ```c
  void swap(int& x, int& y){
    int tmp = x;  x = y;  y = tmp;
  }
  ```
- Polymorphic function template
  ```c
  template<class T>
  void swap(T& x, T& y){
    T tmp = x;  x = y;  y = tmp;
  }
  ```
- Call like ordinary function
  ```c
  float a, b;  …  ; swap(a,b);  …
  ```

Credit: John Mitchell
Obtaining Abstraction

- **C:**
  
  ```
  qsort( (void*)v, N, sizeof(v[0]), compare_int );
  ```

- **C++, using raw C arrays:**
  
  ```
  int v[N];
  sort( v, v+N );
  ```

- **C++, using a vector class:**
  
  ```
  vector v(N);
  sort( v.begin(), v.end() );
  ```

Credit: John Mitchell
Language concepts

- Dynamic lookup
  - different code for different object
  - integer “+” different from real “+”

- Encapsulation
  - Implementer of a concept has detailed view
  - User has “abstract” view
  - Encapsulation separates these two views

- Subtyping
  - Relation between interfaces

- Inheritance
  - Relation between implementations

Credit: John Mitchell
Virtual functions (vptr & vtable)

class Base {
    // Inserted by compiler!
    FunctionPointer *__vptr;
public:
    virtual void function1() {}  
    virtual void function2() {}
};

class D1: public Base {
public:
    virtual void function1() {}  
};

class D2: public Base {
public:
    virtual void function2() {}  
};
## Turning C into Object Code

- Code in files `p1.c p2.c`
- Compile with command: `gcc -O1 p1.c p2.c -o p`
  - Use basic optimizations (`-O1`)
  - Put resulting binary in file `p`

<table>
<thead>
<tr>
<th>text</th>
<th>C program (<code>p1.c p2.c</code>)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Compiler (<code>gcc -S</code>)</td>
</tr>
<tr>
<td>text</td>
<td>Asm program (<code>p1.s p2.s</code>)</td>
</tr>
<tr>
<td></td>
<td>Assembler (<code>gcc or as</code>)</td>
</tr>
<tr>
<td>binary</td>
<td>Object program (<code>p1.o p2.o</code>)</td>
</tr>
<tr>
<td></td>
<td>Linker (<code>gcc or ld</code>)</td>
</tr>
<tr>
<td>binary</td>
<td>Executable program (<code>p</code>)</td>
</tr>
</tbody>
</table>

Static libraries (`.a`)
Compiling Into Assembly

C Code

```c
int sum(int x, int y)
{
    int t = x+y;
    return t;
}
```

Generated IA32 Assembly

```
sum:
    pushl %ebp
    movl %esp,%ebp
    movl 12(%ebp),%eax
    addl 8(%ebp),%eax
    popl %ebp
    ret
```

Some compilers use instruction “leave”

Obtain with command

```
/usr/local/bin/gcc -O1 -S code.c
```

Produces file `code.s`
Call Chain Example

Procedure `amI()` is recursive
Stack Frames

- Contents
  - Local variables
  - Return information
  - Temporary space

- Management
  - Space allocated when enter procedure
    - “Set-up” code
  - Deallocated when return
    - “Finish” code
Example

```c
yoo (...) {
    •
    •
    who ();
    •
    •
}
```
Example

```c
yoo(...) {
    who(...) {
        amI();
    }
}
```

Stack

```
%ebp
%esp
yoo
who
amI
amI
amI
```
Example
Example

```
# ... omitted for brevity ...

yoo (...) {
  who (...) {
    amI (...) {
      amI ();
      ...
      ...
    }
  }
}
```

Stack:

```
yoo
who
amI
%ebp
%esp
```
Example

```c
yoo(...) {
    who(...) {
        amI(...) {
            amI(...) {
                amI(...) {
                    amI();
                }
            }
        }
    }
}
```

Stack:

```
%ebp
%esp
amI
amI
amI
amI
yoo
who
```
Example

```
#include <iostream>

using namespace std;

int main() {
    for (int i = 0; i < 10; ++i) {
        cout << i << endl;
    }
    return 0;
}
```
Example

```
yoo(...) {
  who(...) {
    ... amI(); ...
    amI(); ...
  }
}

who(…) {
  ... amI(); ...
  amI(); ...
}

amI(); ...
```

Stack

```
%ebp

yoo

%esp

who
```

Diagram showing the call stack with functions `yoo` and `who`, and variables on the stack with `%ebp` and `%esp` pointers.
Example

```
yoo(...) {
  who(...) {
    amI(...) {
      •
      •
      amI();
      •
      •
    }
  }
}
```

Stack

```
%ebp
%esp
```
Example

```
\texttt{yoo(\ldots)}
\{
\quad \texttt{who(\ldots)}
\{
\quad \texttt{\ldots}
\quad \texttt{amI();}
\quad \texttt{\ldots}
\quad \texttt{amI();}
\quad \texttt{\ldots}
\}
\}
\texttt{\ldots}
```

```
\begin{align*}
\texttt{yoo} & \rightarrow \texttt{who} \\
\texttt{who} & \rightarrow \texttt{amI} & \texttt{amI} \\
\texttt{amI} & \rightarrow \texttt{amI} \\
\texttt{amI} & \rightarrow \texttt{amI}
\end{align*}
```

Stack

```
\begin{align*}
\texttt{yoo} \rightarrow \texttt{who} \\
\texttt{who} \rightarrow \texttt{amI} & \rightarrow \texttt{amI} \\
\texttt{amI} \rightarrow \texttt{amI} & \rightarrow \texttt{amI}
\end{align*}
```

%ebp → %esp
Example

```
yoo (...) {
    ...
    who ();
    ...
}
```

Stack

```
yoo

%ebp ➔
%esp ➔
```

```
yoo

who

amI

amI

amI
```
IA32/Linux Stack Frame

- **Current Stack Frame** (“Top” to Bottom)
  - “Argument build:” Parameters for function about to call
  - 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
Program to Process

- We write a program in e.g., C.
- A compiler turns that program into an instruction list.
- The CPU interprets the instruction list (which is more a graph of basic blocks).

```c
void X (int b) {
    if(b == 1) {
        ...
    }
    int main() {
        int a = 2;
        X(a);
    }
```
Process in Memory

- Program to process.
  - What you wrote

```c
void X (int b) {
    if(b == 1) {
        ...
    int main() {
        int a = 2;
        X(a);
    }
}
```

- What is in memory.

<table>
<thead>
<tr>
<th>Stack</th>
<th>Heap</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>main; a = 2</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>X; b = 2</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>void X (int b) {</code></td>
<td><code>void X (int b) {</code></td>
<td><code>void X (int b) {</code></td>
</tr>
<tr>
<td><code>    if(b == 1) {</code></td>
<td><code>    if(b == 1) {</code></td>
<td><code>    if(b == 1) {</code></td>
</tr>
<tr>
<td><code>        </code></td>
<td><code>        </code></td>
<td><code>        </code></td>
</tr>
<tr>
<td><code>    ...</code></td>
<td><code>    ...</code></td>
<td><code>    ...</code></td>
</tr>
<tr>
<td><code>    int main() {</code></td>
<td><code>    int main() {</code></td>
<td><code>    int main() {</code></td>
</tr>
<tr>
<td><code>        int a = 2;</code></td>
<td><code>        int a = 2;</code></td>
<td><code>        int a = 2;</code></td>
</tr>
<tr>
<td><code>        X(a);</code></td>
<td><code>        X(a);</code></td>
<td><code>        X(a);</code></td>
</tr>
<tr>
<td><code>    }</code></td>
<td><code>    }</code></td>
<td><code>    }</code></td>
</tr>
</tbody>
</table>
Processes and Process Management

• A program consists of code and data
• On running a program, the OS loader:
  • Reads and interprets the executable file
  • Sets up the process’s memory to contain the code & data from executable
  • Pushes argc, argv on the stack
  • Sets the CPU registers & calls _start()
• Program starts executing at _start()
  
  _start(args) {
      initialize_language_runtime();
      ret = main(args);
      exit(ret)
  }

  Process is now running from program file
• When main() returns, runtime calls exit system call which destroys the process and returns all resources
A shell forks and then execs a calculator

```c
int pid = fork();
if (pid == 0) {
    close(".history");
    exec("/bin/calc");
} else {
    wait(pid);
}

int pid = fork();
if (pid == 0) {
    close(".history");
    exec("/bin/calc");
} else {
    wait(pid);
}
```

**Process Control Blocks (PCBs)**
A shell forks and then execs a calculator

```c
int shell_main() {
    int a = 2;
    ...
}
```

```c
int calc_main() {
    int q = 7;
    ...
}
```

pid = 127
open files = ".history"
last_cpu = 0

pid = 128
open files = ".history"
last_cpu = 0
Anatomy of an address space

- Header
- Code
- Initialized data

Executable File

Process’s address space

- mapped segments
  - DLL’s
  - Stack
  - Heap
  - Initialized data
  - Code
  - Inaccessible
Linker Symbols

• Global symbols
  • Symbols defined by module $m$ that can be referenced by other modules.
  • E.g.: non-\texttt{static} C functions and non-\texttt{static} global variables.

• External symbols
  • Global symbols that are referenced by module $m$ but defined by some other module.

• Local symbols
  • Symbols that are defined and referenced exclusively by module $m$.
  • E.g.: C functions and variables defined with the \texttt{static} attribute.
  • \textbf{Local linker symbols are not local program variables}
Resolving Symbols

```c
int buf[2] = {1, 2};

int main()
{
    swap();
    return 0;
}

extern int buf[];

int *bufp0 = &buf[0];
static int *bufp1;

void swap()
{
    int temp;
    bufp1 = &buf[1];
    temp = *bufp0;
    *bufp0 = *bufp1;
    *bufp1 = temp;
}
```

Global
- int buf[2] = {1, 2};
- extern int buf[];

External
- int main();
- void swap();

Local
- int temp;

Global
- bufp0, bufp1

Linker knows nothing of temp
Relocating Code and Data

Relocatable Object Files

- System code
  - main()
  - int buf[2]={1,2}

- System data

Executable Object File

- .text
- .data
- .bss

headers

System code

main()

System data

int buf[2]={1,2}

More system code

System data

int buf[2]={1,2}

int *bufp0=&buf[0]

int *bufp1

.static int *bufp1

.bss

Even though private to swap, requires allocation in .bss
Strong and Weak Symbols

- Program symbols are either strong or weak
  - **Strong**: procedures and initialized globals
  - **Weak**: uninitialized globals

```
int foo=5;
p1() {
}

int foo;
p2() {
}
```

```
strong

int foo=5;
p1() {
}

int foo;
p2() {
}
```
Linker’s Symbol Rules

- Rule 1: Multiple strong symbols are not allowed
  - Each item can be defined only once
  - Otherwise: Linker error

- Rule 2: Given a strong symbol and multiple weak symbol, choose the strong symbol
  - References to the weak symbol resolve to the strong symbol

- Rule 3: If there are multiple weak symbols, pick an arbitrary one
  - Can override this with `gcc -fno-common`
Linker Puzzles

Link time error: two strong symbols (p1)

References to x will refer to the same uninitialized int. Is this what you really want?

writes to x in p2 might overwrite y! Evil!

writes to x in p2 will overwrite y! Nasty!

References to x will refer to the same initialized variable.

Nightmare scenario: two identical weak structs, compiled by different compilers with different alignment rules.
Using Static Libraries

- Linker’s algorithm for resolving external references:
  - Scan `.o` files and `.a` files in the command line order.
  - During scan, keep a list of the current unresolved references.
  - As each new `.o` or `.a` file, `obj`, is encountered, try to resolve each unresolved reference against the symbols defined in `obj`.
  - If any entries in the unresolved list at end of scan, then error.

- Problem:
  - Command line order matters!
  - Moral: put libraries at the end of the command line.

```
unix> gcc -L. libtest.o -lmine
unix> gcc -L. -lmine libtest.o
libtest.o: In function `main':
  libtest.o(.text+0x4): undefined reference to `libfun'
```
Loading Executable Object Files

Executable Object File

- ELF header
- Program header table (required for executables)
  - .init section
  - .text section
  - .rodata section
  - .data section
  - .bss section
  - .symtab
  - .debug
  - .line
  - .strtab
- Section header table (required for relocatables)

Kernel virtual memory

- User stack (created at runtime)
- Memory-mapped region for shared libraries
- Run-time heap (created by malloc)
- Read/write segment (.data, .bss)
- Read-only segment (.init, .text, .rodata)
- Unused

Memory outside 32-bit address space

- %esp (stack pointer)
- brk (Loaded from the executable file)

Memory - mapped region for shared libraries

- User stack (created at runtime)
- Run-time heap (created by malloc)
- Read/write segment (.data, .bss)
- Read-only segment (.init, .text, .rodata)
- Unused

0x08048000

0x100000000

0xf7e9ddc0
Definitions

• **Architecture**: (also instruction set architecture: ISA) The parts of a processor design that one needs to understand to write assembly code.
  • Examples: instruction set specification, registers.

• **Microarchitecture**: Implementation of the architecture.
  • Examples: cache sizes and core frequency.

• Example ISAs (Intel): x86, ARM