• http://en.wikipedia.org/wiki/Dennis_Ritchie
Plans for Monday

• Lab / Lecture
• Lab due: Monday by 11am

• Wednesday: Midterm
• If you don’t have last year’s midterm, get a copy

• Labs graded by Monday
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
Revisiting `swap`

```c
int zip1 = 15213;
int zip2 = 91125;

void call_swap()
{
    swap(&zip1, &zip2);
}
```

Calling `swap` from `call_swap`

```c
void call_swap()
{
    pushl $zip2  # Global Var
    pushl $zip1  # Global Var
    call swap
}
```

Resulting Stack

```
%esp
Rtn adr
&zip1
&zip2
```
Mid-semester feedback

1. What’s been most useful to you (and why)?

2. What could be going better / be more useful (and why)?

3. What could students do to improve the class?

4. What could Matt do to improve the class?
Union Allocation

- Allocate according to largest element
- Can only use ones field at a time

```c
union U1 {
    char c;
    int i[2];
    double v;
} *up;

struct S1 {
    char c;
    int i[2];
    double v;
} *sp;
```
typedef union {
    float f;
    unsigned u;
} bit_float_t;

float bit2float(unsigned u) {
    bit_float_t arg;
    arg.u = u;
    return arg.f;
}

unsigned float2bit(float f) {
    bit_float_t arg;
    arg.f = f;
    return arg.u;
}

Same as (float) u ?

Same as (unsigned) f ?
Alignment

• Aligned Data
  – Primitive data type requires K bytes
  – Address must be multiple of K
  – Required on some machines; advised on IA32
    • treated differently by IA32 Linux, x86-64 Linux, and Windows!

• Motivation for Aligning Data
  – Memory accessed by (aligned) chunks of 4 or 8 bytes (system dependent)
    • Inefficient to load or store datum that spans quad word boundaries
    • Virtual memory very tricky when datum spans 2 pages

• Compiler
  – Inserts gaps in structure to ensure correct alignment of fields
Specific Cases of Alignment (IA32)

• 1 byte: char, ...
  – no restrictions on address

• 2 bytes: short, ...
  – lowest 1 bit of address must be 0_2

• 4 bytes: int, float, char *, ...
  – lowest 2 bits of address must be 00_2

• 8 bytes: double, ...
  – Windows (and most other OS’s & instruction sets):
    • lowest 3 bits of address must be 000_2
  – Linux:
    • lowest 2 bits of address must be 00_2
    • i.e., treated the same as a 4-byte primitive data type

• 12 bytes: long double
  – Windows, Linux:
    • lowest 2 bits of address must be 00_2
    • i.e., treated the same as a 4-byte primitive data type
Different Alignment Conventions

- IA32 Linux
  - $K = 4$; **double** treated like a 4-byte data type

```c
struct S1 {
    char c;
    int i[2];
    double v;
} *p;
```
Arrays of Structures

- Satisfy alignment requirement for every element

```c
struct S2 {
    double v;
    int i[2];
    char c;
} a[10];
```
Accessing Array Elements

- Compute array offset 12i
- Compute offset 8 with structure
- Assembler gives offset a+8
  - Resolved during linking

```
struct S3 {
    short i;
    float v;
    short j;
} a[10];
```

```
short get_j(int idx) {
    return a[idx].j;
}
```

```
# %eax = idx
leal (%eax,%eax,2),%eax  # 3*idx
movswl a+8(%eax,4),%eax
```
char *gets(char *s)
{
    int c;
    char *dest = s;
    int gotchar = 0;
    while (((c = getchar()) != '\n' && c != EOF)) {
        *dest++ = c;
        gotchar = 1;
    }
    *dest++ = '\0';
    if (c == EOF && !gotchar)
        return NULL;
    return s;
}

void echo()
{
    char buf[8];
    gets(buf);
    puts(buf);
}