Lecture 9: Memory Layout, Worms, Program Optimization

Computer Architecture and Systems Programming
(252-0061-00)
Timothy Roscoe
Herbstsemester 2012

```
struct rec {
  int i;
  int a[3];
  int *p;
};

void set_p(struct rec *r) {
  r->p = &r->a[r->i];
}
```

Last time: structures

- C Code
- void set_p(struct rec *r) {
  r->p = &r->a[r->i];
}

```
# %edx = r
movl (%edx),%ecx # r->i
leal 0(,%ecx,4),%eax # 4*(r->i)
leal 4(%edx,%eax),%eax # r+4+4*(r->i)
movl %eax,16(%edx) # Update r->p
```

Last time: alignment

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

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

Last time: unions

```
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;
}
```

Last time: FPU stack (x87)

- FPU register format (80 bit extended precision)
  - 8 registers %st(0) - %st(7)
  - Logically form stack
  - Top: %st(0)
  - Bottom disappears (drops out) after too many pushes

```
Last time: SSE3 registers

- All caller saved
- %xmm0 for floating point return value
```

```
128 bit = 2 doubles = 4 singles

<table>
<thead>
<tr>
<th>%xmm0</th>
<th>Argument 81</th>
</tr>
</thead>
<tbody>
<tr>
<td>%xmm1</td>
<td>%xmm9</td>
</tr>
<tr>
<td>%xmm2</td>
<td>%xmm10</td>
</tr>
<tr>
<td>%xmm3</td>
<td>Argument 84</td>
</tr>
<tr>
<td>%xmm4</td>
<td>%xmm11</td>
</tr>
<tr>
<td>%xmm5</td>
<td>%xmm12</td>
</tr>
<tr>
<td>%xmm6</td>
<td>Argument 87</td>
</tr>
<tr>
<td>%xmm7</td>
<td>%xmm14</td>
</tr>
<tr>
<td>%xmm8</td>
<td>%xmm15</td>
</tr>
<tr>
<td>%xmm9</td>
<td>%xmm16</td>
</tr>
<tr>
<td>%xmm10</td>
<td>%xmm17</td>
</tr>
<tr>
<td>%xmm11</td>
<td>%xmm18</td>
</tr>
<tr>
<td>%xmm12</td>
<td>%xmm19</td>
</tr>
</tbody>
</table>
```

"Top"
Summary

- Arrays in C
  - Contiguous allocation of memory
  - Aligned to satisfy every element’s alignment requirement
  - No bounds checking
- Structures
  - Allocate bytes in order declared
  - Pad in middle and at end to satisfy alignment
- Unions
  - Overlay declarations
  - Way to circumvent type system
- Floating point
  - x87: stack machine
  - SSE3: 16 FP registers, mirrors integer architecture

Today

- Memory layout
- Buffer overflow, worms, and viruses
- Program optimization
  - Overview
  - Removing unnecessary procedure calls
  - Code motion/precomputation
  - Strength reduction
  - Sharing of common subexpressions
  - Optimization blocker: Procedure calls

IA32 Linux memory layout

- Stack
  - Runtime stack (8MB limit)
- Heap
  - Dynamically allocated storage
  - When call malloc(), calloc(), new()
- Data
  - Statically allocated data
  - E.g., arrays & strings declared in code
- Text
  - Executable machine instructions
  - Read-only

Memory allocation example

```c
char big_array[1<<24]; /* 16 MB */
char huge_array[1<<28]; /* 256 MB */

int beyond;
char *p1, *p2, *p3, *p4;

int useless() {  return 0; }

int main()
{
  p1 = malloc(1 <<28);  /* 256 MB */
  p2 = malloc(1 << 8);  /* 256 B */
  p3 = malloc(1 <<28);  /* 256 MB */
  p4 = malloc(1 << 8);  /* 256 B */
  /* Some print statements ... */
}
```

Where does everything go?

IA32 example addresses

```
$esp 0xffffbcd0
p3 0x65586008
p1 0x55585008
p4 0x1004a110
p2 0x1904a008
&p2 0x10049760
beyond 0x08049744
big_array 0x18049780
huge_array 0x08049760
main() 0x080483c6
useless() 0x08049744
final malloc() 0x006be166

malloc() is dynamically linked
address determined at runtime
```

x86-64 example addresses

```
$rsp 0x7ffffff8d1f8
p3 0x2aaabaadd010
p1 0x2aaaaaadc010
p4 0x000011501120
p2 0x000011501010
&p2 0x000010500a60
beyond 0x000000500a44
big_array 0x000010500a80
huge_array 0x000000500a50
main() 0x000000400510
useless() 0x000000400500
final malloc() 0x00386ae6a170

malloc() is dynamically linked
address determined at runtime
```
Today

- Memory layout
- Buffer overflow, worms, and viruses
- Program optimization
  - Overview
  - Removing unnecessary procedure calls
  - Code motion/precomputation
  - Strength reduction
  - Sharing of common subexpressions
  - Optimization blocker: Procedure calls

Worms and Viruses

- Worm: A program that
  - Can run by itself
  - Can propagate a fully working version of itself to other computers
- Virus: Code that
  - Add itself to other programs
  - Cannot run independently
- Both are (usually) designed to spread among computers and to wreak havoc

Early worms

- Term coined in 1975 by John Brunner
  - First "cyberpunk" novel: The Shockwave Rider
- Mid-1970s: research into benign worms at BBN and Xerox PARC
- November 1988: Robert Morris’ Worm
  - First Internet worm, attacked thousands of hosts
  - Morris now professor of Computer Science at MIT
  - Awarded the SIGOPS Mark Weiser award recently ... and the rest is history.

String library code

- Implementation of Unix function `gets()`
  ```c
  /* Get string from stdin */
  char *gets(char *dest)
  {
    int c = getchar();
    char *p = dest;
    while (c != EOF && c != '\n') {
      *p++ = c;
      c = getchar();
    }
    *p = '\0';
    return dest;
  }
  ```
- No way to specify limit on number of characters to read
- Similar problems with other Unix functions
  - `strcpy`: Copies string of arbitrary length
  - `scanf`, `fscanf`, `sscanf`, when given `%s` conversion specification

Vulnerable buffer code

```c
/* Echo line */
void echo()
{
  char buf[4]; /* Way too small! */
  gets(buf);
  puts(buf);
}

int main()
{
  printf("Type a string:");
  echo();
  return 0;
}
```

Buffer overflow disassembly

```
080484b0 <echo>:
080484f0: 55
080484f1: 89 e5
080484f3: 53
080484f4: 8d 5d f8
080484f7: 83 ec 14
080484fa: 89 1c 24
080484fd: e8 ae ff ff ff
08048502: 89 1c 24
08048505: e8 8a fe ff ff
0804850a: 83 c4 14
0804850d: 5b
0804850e: c9
0804850f: c3
08048512: e8 9f fe ff ff
08048517: 85 d0 fc
0804851a: e8 8a fe ff ff
0804851f: c9
08048522: 83 c0
08048523: 89 c3
```

unix> ./bufdemo
Type a string:
1234567
1234567
unix> ./bufdemo
Type a string:
12345678
Segmentation Fault
Exploits based on buffer overflows

- **Buffer overflow bugs allow remote machines to execute arbitrary code on victim machines**
- **Internet worm (one vector)**
  - Early versions of the finger server (fingerd) used `gets()` to read the argument sent by the client:
    - `finger droh@cs.cmu.edu`
  - Worm attacked fingerd server by sending phony argument:
    - `finger "exploit-code padding new-return-address"`
    - Exploit code: executed a root shell on the victim machine with a direct TCP connection to the attacker.

Avoiding overflow vulnerability

```c
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    fgets(buf, 4, stdin);
    puts(buf);
}
```

- Use library routines that limit string lengths
  - `fgets` instead of `gets`
  - `strncpy` instead of `strcpy`
- Don’t use `scanf` with `%s` conversion specification
  - Use `fgets` to read the string
  - Or use `%ns` where `n` is a suitable integer

System-level protections

- **Randomized stack offsets**
  - At start of program, allocate random amount of space on stack
  - Makes it difficult for hacker to predict beginning of inserted code
- **Nonexecutable code segments**
  - In traditional x86, can mark region of memory as either “read-only” or “writeable”
    - Can execute anything readable
    - Add explicit “execute” permission

Today

- **Memory layout**
- **Buffer overflow, worms, and viruses**
- **Program optimization**
  - **Overview**
    - Removing unnecessary procedure calls
    - Code motion/precomputation
    - Strength reduction
    - Sharing of common subexpressions
    - Optimization blocker: Procedure calls

Example matrix multiplication

- Standard desktop computer, compiler, using optimization flags
- Both implementations have exactly the same operations count ($2n^3$)
- **What is going on?**

Matrix-Matrix Multiplication (MMM) on 2x Core 2 Duo 3 GHz
Gflop/s (giga floating point operations per second)

MMM plot: analysis

- Reason for 20x: Blocking or tiling, loop unrolling, array scalarization, instruction scheduling, search to find best choice
- Effect: more instruction level parallelism, better register use, less L1/L2 cache misses, less TLB misses
Harsh reality

- There’s more to runtime performance than asymptotic complexity
- One can easily loose 10x, 100x in runtime or even more
- What matters:
  - Constants (100n and 5n is both O(n), but …)
  - Coding style (unnecessary proc. calls, unrolling, reordering, …)
  - Algorithm structure (locality, instruction level parallelism, …)
  - Data representation (complicated structs or simple arrays)

Optimizing compilers

- Use optimization flags, default can be no optimization [-O0]!
- Good choices for gcc: -O2, -O3, -march=×××, -m64
- Try different flags and maybe different compilers
  - icc is usually faster than gcc

Example

```
void mmm(double *a, double *b, double *c, int n) {
    /* Multiply 4 x 4 matrices a and b */
    double c[4][4]; /* set to zero */
    double b[4][4];
    double a[4][4];

    for (i = 0; i < 4; i++)
        for (j = 0; j < 4; j++)
            for (k = 0; k < 4; k++)
                c[i][j] += a[i][k] * b[k][j];
}
```

- Compiled without flags:
  - ~1300 cycles
- Compiled with -O3 -m64 -march=××× -fno-vectorize
  - ~150 cycles
- Core 2 Duo, 2.66 GHz

Limitations of optimizing compilers

- If in doubt, the compiler is conservative
- Operate under fundamental constraints
  - Must not change program behavior under any possible condition
  - Often prevents it from making optimizations when would only affect behavior under pathological conditions.
- Behavior that may be obvious to the programmer can be obfuscated by languages and coding styles
  - e.g., data ranges may be more limited than variable types suggest
- Most analysis is performed only within procedures
  - Whole-program analysis is too expensive in most cases
- Most analysis is based only on static information
  - Compiler has difficulty anticipating run-time inputs
Today

- Memory layout
- Buffer overflow, worms, and viruses
- Program optimization
  - Overview
  - Removing unnecessary procedure calls
  - Code motion/precomputation
  - Strength reduction
  - Sharing of common subexpressions
  - Optimization blocker: Procedure calls
  - Optimization blocker: Memory aliasing

Example: Data type for vectors

```c
/* data structure for vectors */
typedef struct{
    int len;
    double *data;
} vec;

/* retrieve vector element and store at val */
int get_vec_element(vec *v, int idx, double *val)
{
    if (idx < 0 || idx >= v->len)
        return 0;
    *val = v->data[idx];
    return 1;
}
```

Example: Summing vector elements

```c
/* sum elements of vector */
double sum_elements(vec *v, double *res)
{
    int i;
    n = vec_length(v);
    *res = 0.0;
    double *data = get_vec_start(v);
    for (i = 0; i < n; i++)
    {
        *res += data[i];
    }
    return *res;
}
```

Example: Removing procedure call

```c
/* sum elements of vector */
double sum_elements(vec *v, double *res)
{
    int i;
    n = vec_length(v);
    *res = 0.0;
    double val;
    for (i = 0; i < n; i++)
    {
        get_vec_element(v, i, &val);
        *res += val;
    }
    return *res;
}
```

Removing procedure calls

- Procedure calls can be very expensive
- Bound checking can be very expensive
- Abstract data types can easily lead to inefficiencies
  - Usually avoided for in superfast numerical library functions

- Watch your innermost loop!
- Get a feel for overhead versus actual computation being performed

Today

- Memory layout
- Buffer overflow, worms, and viruses
- Program optimization
  - Overview
  - Removing unnecessary procedure calls
  - Code motion/precomputation
  - Strength reduction
  - Sharing of common subexpressions
  - Optimization blocker: Procedure calls
  - Optimization blocker: Memory aliasing
Code motion

- Reduce frequency with which computation is performed
  - If it will always produce same result
  - Especially moving code out of loop

- Sometimes also called precomputation

```c
void set_row(double *a, double *b, long 1, long n)
{
    long j;
    for (j = 0; j < n; j++)
        a[n*i+j] = b[j];
}
```

Compiler-generated code motion

```c
void set_row(double *a, double *b, long 1, long n)
{
    long j;
    for (j = 0; j < n; j++)
        *rowp++ = b[j];
}
```

Where are the FP operations?

- Replace costly operation with simpler one
- Example: Shift/add instead of multiply or divide

```
int inj = i*n + j;
for (j = 0; j < n; j++)
    a[n*i + j] = b[j];
```

Strength Reduction

- Replace costly operation with simpler one
- Example: Shift/add instead of multiply or divide

```
16*x      \rightarrow x << 4
```

- Utility machine dependent
- Depends on cost of multiply or divide instruction
- On Pentium IV, integer multiply requires 10 CPU cycles

- Example: Recognize sequence of products

```
for (i = 0; i < n: i++)
    for (j = 0; j < n: j++)
        a[n*i + j] = b[j];
```

Share common subexpressions

- Reuse portions of expressions
- Compilers often not very sophisticated in exploiting arithmetic properties

```
/* run neighborhood of i,j */
up = val[i+1] + 1;
down = val[i-1] + 1;
left = val[i] + 1;
right = val[i] + 1;
sum = up + down + left + right;
```

Today

- Memory layout
- Buffer overflow, worms, and viruses
- Program optimization
  - Overview
  - Removing unnecessary procedure calls
  - Code motion/precomputation
  - Strength reduction
    - Sharing of common subexpressions
    - Optimization blocker: Procedure calls
    - Optimization blocker: Memory aliasing

```
int ni = 0;
for (j = 0; j < n; j++)
    a[n*i + j] = b[j];
```

```
int inj = i*n + j;
for (j = 0; j < n; j++)
    a[n*i + j] = b[j];
```

```
int inj = i*n + j;
for (j = 0; j < n; j++)
    a[n*i + j] = b[j];
```

```
int inj = i*n + j;
for (j = 0; j < n; j++)
    a[n*i + j] = b[j];
```
Today

- Memory layout
- Buffer overflow, worms, and viruses
- Program optimization
  - Overview
  - Removing unnecessary procedure calls
  - Code motion/precomputation
  - Strength reduction
  - Sharing of common subexpressions
  - Optimization blocker: Procedure calls
  - Optimization blocker: Memory aliasing

Optimization blocker #1: procedure calls

- Procedure to convert string to lower case

```c
void lower(char *s)
{
    int i;
    for (i = 0; i < strlen(s); i++)
        if (s[i] >= 'A' && s[i] <= 'Z')
            s[i] -= ('A' - 'a');
}
```

Why is that?

```c
void lower(char *s)
{
    int i;
    for (i = 0; i < strlen(s); i++)
        if (s[i] >= 'A' && s[i] <= 'Z')
            s[i] -= ('A' - 'a');
}
```

Why is that?

- String length is called in every iteration!
  - And strlen is O(n), so lower is O(n^2)

```c
void lower(char *s)
{
    int i;
    for (i = 0; i < strlen(s); i++)
        if (s[i] >= 'A' && s[i] <= 'Z')
            s[i] -= ('A' - 'a');
}
```

Performance

- Time quadruples when string length doubles
- Quadratic performance

```plaintext
CPU Seconds

<table>
<thead>
<tr>
<th>String Length</th>
<th>CPU Seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>256</td>
<td>10</td>
</tr>
<tr>
<td>512</td>
<td>20</td>
</tr>
<tr>
<td>1k</td>
<td>40</td>
</tr>
<tr>
<td>2k</td>
<td>80</td>
</tr>
<tr>
<td>4k</td>
<td>160</td>
</tr>
<tr>
<td>8k</td>
<td>320</td>
</tr>
<tr>
<td>16k</td>
<td>640</td>
</tr>
<tr>
<td>32k</td>
<td>1280</td>
</tr>
<tr>
<td>64k</td>
<td>2560</td>
</tr>
<tr>
<td>128k</td>
<td>5120</td>
</tr>
<tr>
<td>256k</td>
<td>10240</td>
</tr>
</tbody>
</table>
```

Improving performance

```c
void lower(char *s)
{
    int i;
    for (i = 0; i < strlen(s); i++)
        if (s[i] >= 'A' && s[i] <= 'Z')
            s[i] -= ('A' - 'a');
}
```

- Move call to strlen outside of loop
- Since result does not change from one iteration to another
- Form of code motion/precomputation

Performance

- Lower2: Time doubles when double string length
- Linear performance

```plaintext
CPU Seconds

<table>
<thead>
<tr>
<th>String Length</th>
<th>CPU Seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>256</td>
<td>0.001</td>
</tr>
<tr>
<td>512</td>
<td>0.002</td>
</tr>
<tr>
<td>1k</td>
<td>0.004</td>
</tr>
<tr>
<td>2k</td>
<td>0.008</td>
</tr>
<tr>
<td>4k</td>
<td>0.016</td>
</tr>
<tr>
<td>8k</td>
<td>0.032</td>
</tr>
<tr>
<td>16k</td>
<td>0.064</td>
</tr>
<tr>
<td>32k</td>
<td>0.128</td>
</tr>
<tr>
<td>64k</td>
<td>0.256</td>
</tr>
<tr>
<td>128k</td>
<td>0.512</td>
</tr>
<tr>
<td>256k</td>
<td>1.024</td>
</tr>
</tbody>
</table>
```
Optimization blocker:
Procedure calls

• Why couldn't compiler move `strlen` out of inner loop?
  – Procedure may have side effects
  – Function may not return same value for given arguments
  – Could depend on other parts of global state
  – Procedure lower could interact with `strlen`
• Compiler usually treats procedure call as a black box that cannot be analyzed
  – Consequence: conservative in optimizations
• Remedies:
  – Inline the function if possible
  – Do your own code motion

```c
int lencnt = 0;
size_t strlen(const char *s)
{
    size_t length = 0;
    while (*s != '\0') {
        s++; length++;
    }
    lencnt += length;
    return length;
}
```

Moral:
collaborate with the compiler!

• Turn on optimization!
• Let the compiler do what it’s good at.
• Remove obstacles to optimizer
• Do it yourself if necessary
• Sometimes at odds with abstraction and encapsulation
  – That’s the tradeoff…

Next time: Advanced C

• Operators
• Function pointers
• Typedefs and structures
• `goto`
• Assertions
• Are arrays the same as pointers?
• `setjmp()`/`longjmp()`
• Coroutines