Introduction to Computer Systems
15-213/18-243, spring 2009
10th Lecture, Feb. 12th

Instructors:
Gregory Kesden and Markus Püschel
Last Time

- **Structures**

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

- **Alignment**

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

- **Unions**

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

- **Floating point**
  - x87 (getting obsolete)
  - x86-64 (SSE3 and later)

- Vector mode and scalar mode
  - addps
  - addss
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

Upper 2 hex digits = 8 bits of address
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

address range \(~2^{32}\)

\begin{align*}
\$esp & \quad 0xffffbcd0 \\
p3 & \quad 0x65586008 \\
p1 & \quad 0x55585008 \\
p4 & \quad 0x1904a110 \\
p2 & \quad 0x1904a008 \\
&p2 & \quad 0x18049760 \\
beyond & \quad 0x08049744 \\
big_array & \quad 0x18049780 \\
huge_array & \quad 0x08049760 \\
main() & \quad 0x080483c6 \\
useless() & \quad 0x08049744 \\
final\ malloc() & \quad 0x006be166 \\
\end{align*}

malloc() is dynamically linked
address determined at runtime
## x86-64 Example Addresses

*address range ~2\(^{47}\)*

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$rsp</td>
<td>0x7ffffff8d1f8</td>
</tr>
<tr>
<td>p3</td>
<td>0x2aaabaadd010</td>
</tr>
<tr>
<td>p1</td>
<td>0x2aaaaaad010</td>
</tr>
<tr>
<td>p4</td>
<td>0x000011501120</td>
</tr>
<tr>
<td>p2</td>
<td>0x000011501010</td>
</tr>
<tr>
<td>&amp;p2</td>
<td>0x000010500a60</td>
</tr>
<tr>
<td>beyond</td>
<td>0x0000050a44</td>
</tr>
<tr>
<td>big_array</td>
<td>0x000010500a80</td>
</tr>
<tr>
<td>huge_array</td>
<td>0x0000050a50</td>
</tr>
<tr>
<td>main()</td>
<td>0x00000400510</td>
</tr>
<tr>
<td>useless()</td>
<td>0x00000400500</td>
</tr>
<tr>
<td>final malloc()</td>
<td>0x00386ae6a170</td>
</tr>
</tbody>
</table>

`malloc()` is dynamically linked
address determined at runtime
# C operators

<table>
<thead>
<tr>
<th>Operators</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>()      []        -&gt; .</td>
<td>left to right</td>
</tr>
<tr>
<td>! ~    ++)   )    - + - * &amp; (type) sizeof</td>
<td>right to left</td>
</tr>
<tr>
<td>* / %</td>
<td>left to right</td>
</tr>
<tr>
<td>+ -</td>
<td>left to right</td>
</tr>
<tr>
<td>&lt;&lt; &gt;&gt;</td>
<td>left to right</td>
</tr>
<tr>
<td>&lt;= &gt;=</td>
<td>left to right</td>
</tr>
<tr>
<td>== !=</td>
<td>left to right</td>
</tr>
<tr>
<td>&amp; !=</td>
<td>left to right</td>
</tr>
<tr>
<td>^</td>
<td>left to right</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>left to right</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>?:</td>
<td>right to left</td>
</tr>
<tr>
<td>+= -= *= /= %= &amp; ^= ! = &lt;&lt;= &gt;&gt;=</td>
<td>right to left</td>
</tr>
<tr>
<td>,</td>
<td>left to right</td>
</tr>
</tbody>
</table>

- `->` has very high precedence
- `()` has very high precedence
- `monadic *` just below
C Pointer Declarations: Test Yourself!

\[ \text{int } *p \quad \text{p is a pointer to int} \]

\[ \text{int } *p[13] \]

\[ \text{int } *(p[13]) \]

\[ \text{int } **p \quad \text{p is a pointer to a pointer to an int} \]

\[ \text{int } (*p)[13] \]

\[ \text{int } *f() \quad \text{f is a function returning a pointer to int} \]

\[ \text{int } (*f)() \quad \text{f is a pointer to a function returning int} \]

\[ \text{int } (*(*f())[13])() \]

\[ \text{int } (*(*x[3])())[5] \quad \text{x is an array[3] of pointers to functions returning pointers to array[5] of ints} \]
### C Pointer Declarations (Check out guide)

<table>
<thead>
<tr>
<th>Declaration</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>int *p</code></td>
<td><code>p</code> is a pointer to <code>int</code></td>
</tr>
<tr>
<td><code>int *(p[13])</code></td>
<td><code>p</code> is an array[13] of pointer to <code>int</code></td>
</tr>
<tr>
<td><code>int **p</code></td>
<td><code>p</code> is a pointer to a pointer to an <code>int</code></td>
</tr>
<tr>
<td><code>int (*p)[13]</code></td>
<td><code>p</code> is a pointer to an array[13] of <code>int</code></td>
</tr>
<tr>
<td><code>int *f()</code></td>
<td><code>f</code> is a function returning a pointer to <code>int</code></td>
</tr>
<tr>
<td><code>int (*f)()</code></td>
<td><code>f</code> is a pointer to a function returning <code>int</code></td>
</tr>
<tr>
<td><code>int *((*f())[13])()</code></td>
<td><code>f</code> is a function returning ptr to an array[13] of pointers to functions returning <code>int</code></td>
</tr>
<tr>
<td><code>int *((*x[3])() [5])</code></td>
<td><code>x</code> is an array[3] of pointers to functions returning pointers to array[5] of <code>int</code>s</td>
</tr>
</tbody>
</table>
Avoiding Complex Declarations

- Use `typedef` to build up the declaration

- Instead of `int (**x[3])(())[5]`:
  ```
  typedef int fiveints[5];
  typedef fiveints* p5i;
  typedef p5i (*f_of_p5is)();
  f_of_p5is x[3];
  ```

- `x` is an array of 3 elements, each of which is a pointer to a function returning an array of 5 ints
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
Internet Worm and IM War

- November, 1988
  - Internet Worm attacks thousands of Internet hosts.
  - How did it happen?
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;
}
```

```
unix> ./bufdemo
Type a string:1234567
1234567

unix> ./bufdemo
Type a string:12345678
Segmentation Fault

unix> ./bufdemo
Type a string:123456789ABC
Segmentation Fault
```
## Buffer Overflow Disassembly

### 080484f0 <echo>:

<table>
<thead>
<tr>
<th>Address</th>
<th>Operands</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>080484f0</td>
<td>55</td>
<td>push %ebp</td>
</tr>
<tr>
<td>080484f1</td>
<td>89 e5</td>
<td>mov %esp,%ebp</td>
</tr>
<tr>
<td>080484f3</td>
<td>53</td>
<td>push %ebx</td>
</tr>
<tr>
<td>080484f4</td>
<td>8d 5d f8</td>
<td>lea 0xfffffffffffffff8(%ebp),%ebx</td>
</tr>
<tr>
<td>080484f7</td>
<td>83 ec 14</td>
<td>sub $0x14,%esp</td>
</tr>
<tr>
<td>080484fa</td>
<td>89 1c 24</td>
<td>mov %ebx,(%esp)</td>
</tr>
<tr>
<td>08048502</td>
<td>89 1c 24</td>
<td>mov %ebx,(%esp)</td>
</tr>
<tr>
<td>08048505</td>
<td>e8 8a fe ff ff ff</td>
<td>call 80484b0 &lt;gets&gt;</td>
</tr>
<tr>
<td>0804850a</td>
<td>83 c4 14</td>
<td>add $0x14,%esp</td>
</tr>
<tr>
<td>0804850b</td>
<td>5b</td>
<td>pop %ebx</td>
</tr>
<tr>
<td>0804850e</td>
<td>c9</td>
<td>leave</td>
</tr>
<tr>
<td>0804850f</td>
<td>c3</td>
<td>ret</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Address</th>
<th>Operands</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0804852</td>
<td>e8 f9 fe ff ff ff</td>
<td>call 80484f0 &lt;echo&gt;</td>
</tr>
<tr>
<td>0804857</td>
<td>8b 5d fc</td>
<td>mov 0xfffffffffffffffcc(%ebp),%ebx</td>
</tr>
<tr>
<td>080485a</td>
<td>c9</td>
<td>leave</td>
</tr>
<tr>
<td>080485b</td>
<td>31 c0</td>
<td>xor %eax,%eax</td>
</tr>
<tr>
<td>080485d</td>
<td>c3</td>
<td>ret</td>
</tr>
</tbody>
</table>
Buffer Overflow Stack

Before call to gets

Stack Frame for main

Return Address
Saved %ebp

[3][2][1][0]

Stack Frame for echo

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

echo:
    pushl %ebp          # Save %ebp on stack
    movl %esp, %ebp
    pushl %ebx          # Save %ebx
    leal -8(%ebp),%ebx  # Compute buf as %ebp-8
    subl $20, %esp      # Allocate stack space
    movl %ebx, (%esp)   # Push buf on stack
    call gets          # Call gets
    ...
Buffer Overflow
Stack Example

Before call to gets

Stack Frame for main
  Return Address
  Saved %ebp
  [3][2][1][0]

Stack Frame for echo

Before call to gets

Stack Frame for main

Stack Frame for echo

80485f2: call 80484f0 <echo>
80485f7: mov 0xfffffffffc(%ebp),%ebx # Return Point
Buffer Overflow Example #1

Before call to gets

Overflow buf, but no problem

Input 1234567
Buffer Overflow Example #2

Before call to gets

Stack Frame for main

0xfffffc658

0xfffffc638

buf

Stack Frame for echo

Input 12345678

Stack Frame for main

0xfffffc658

0xfffffc638

buf

Stack Frame for echo

Base pointer corrupted

804850a:  83 c4 14    add    $0x14,%esp    # deallocate space
804850d:  5b        pop    %ebx          # restore %ebx
804850e:  c9        leave              # movl %ebp, %esp; popl %ebp
804850f:  c3        ret                  # Return
Buffer Overflow Example #3

Before call to gets

Stack Frame for main

08 04 85 f7
ff ff c6 58
xx xx xx xx

Stack Frame for echo

Input 12345678

Stack Frame for main

08 04 85 00
43 42 41 39
38 37 36 35
34 33 32 31

buf

Stack Frame for echo

buf

Return address corrupted

80485f2: call 80484f0 <echo>
80485f7: mov 0xffffffffc(%ebp),%ebx  # Return Point
Malicious Use of Buffer Overflow

- Input string contains byte representation of executable code
- Overwrite return address with address of buffer
- When `bar()` executes `ret`, will jump to exploit code
Exploits Based on Buffer Overflows

- **Buffer overflow bugs allow remote machines to execute arbitrary code on victim machines**

- **Internet worm**
  - 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

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

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

```shell
unix> gdb bufdemo
(gdb) break echo
(gdb) run
(gdb) print /x $ebp
$1 = 0xffffc638
(gdb) run
(gdb) print /x $ebp
$2 = 0xffffbb08
(gdb) run
(gdb) print /x $ebp
$3 = 0xffffc6a8
```
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
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

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

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

Best code

This code is not obviously stupid

160x

Triple loop
Matrix-Matrix Multiplication (MMM) on 2 x Core 2 Duo 3 GHz

Gflop/s

Matrix size

- **Multiple threads**: 4x *(towards end of course)*
- **Vector instructions**: 4x *(not in this course)*
- **Memory hierarchy and other optimizations**: 20x

- **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 procedure calls, unrolling, reordering, ...)
  - Algorithm structure (locality, instruction level parallelism, ...)
  - Data representation (complicated structs or simple arrays)
Harsh Reality

- **Must optimize at multiple levels:**
  - Algorithm
  - Data representations
  - Procedures
  - Loops

- **Must understand system to optimize performance**
  - How programs are compiled and executed
    - Execution units, memory hierarchy
  - How to measure program performance and identify bottlenecks
  - How to improve performance without destroying code modularity and generality
Optimizing Compilers

- Use optimization flags, **default is no optimization** (-O0)!
- Good choices for gcc: -O2, -O3, -march=xxx, -m64
- Try different flags and maybe different compilers
Example

```c
double a[4][4];
double b[4][4];
double c[4][4]; // set to zero

/* Multiply 4 x 4 matrices a and b */
void mmm(double *a, double *b, double *c, int n) {
    int i, j, k;
    for (i = 0; i < 4; i++)
        for (j = 0; j < 4; j++)
            for (k = 0; k < 4; k++)
                c[i * 4 + j] += a[i * 4 + k] * b[k * 4 + j];
}
```

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

- **Compilers are good at:** mapping program to machine
  - register allocation
  - code selection and ordering (scheduling)
  - dead code elimination
  - eliminating minor inefficiencies

- **Compilers are not good at:** improving asymptotic efficiency
  - up to programmer to select best overall algorithm
  - big-O savings are (often) more important than constant factors
    - but constant factors also matter

- **Compilers are not good at:** overcoming “optimization blockers”
  - potential memory aliasing
  - potential procedure side-effects
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
/* data structure for vectors */
typedef struct{
    int len;
    double *data;
} vec;

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

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

/* 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;
}

Bound check unnecessary in sum_elements
Why?

Overhead for every fp +:
• One fct call
• One <
• One >=
• One ||
• One memory variable access

Slowdown: probably 10x or more
/ * 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 i, long n)
{
    long j;
    int ni = n*i;
    for (j = 0; j < n; j++)
        a[n*i+j] = b[j];
}
```
void set_row(double *a, double *b, long i, long n) {
    long j;
    for (j = 0; j < n; j++)
        a[n*i+j] = b[j];
}

Where are the FP operations?

set_row:
    xorl   %r8d, %r8d       # j = 0
    cmpq   %rcx, %r8        # j:n
    jge    .L7              # if >= goto done
    movq   %rcx, %rax       # n
    imulq  %rdx, %rax       # n*i outside of inner loop
    leaq   (%rdi,%rax,8), %rdx # rowp = A + n*i*8
    .L5:
        movq   (%rsi,%r8,8), %rax # t = b[j]
        incq   %r8              # j++
        movq   %rax, (%rdx)     # *rowp = t
        addq   $8, %rdx         # rowp++
        cmpq   %rcx, %r8        # j:n
        jl     .L5              # if < goto loop
    .L7:
        rep ; ret              # done:
                                # return
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
Strength Reduction

- Replace costly operation with simpler one
- Example: Shift/add instead of multiply or divide
  \[16 \times x \rightarrow x \ll 4\]
  - 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

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

```c
int ni = 0;
for (i = 0; i < n; i++) {
  for (j = 0; j < n; j++)
    a[ni + j] = b[j];
  ni += n;
}
```
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
Share Common Subexpressions

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

### 3 mults: $i*n$, $(i-1)*n$, $(i+1)*n$

```c
/* Sum neighbors of $i,j$ */
up = val[(i-1)*n + j ];
down = val[(i+1)*n + j ];
left = val[i*n + j-1 ];
right = val[i*n + j+1 ];
sum = up + down + left + right;
```

### 1 mult: $i*n$

```c
int inj = i*n + j;
up = val[inj - n];
down = val[inj + n];
left = val[inj - 1 ];
right = val[inj + 1 ];
sum = up + down + left + right;
```

```assembly
leaq 1(%rsi), %rax  # i+1
leaq -1(%rsi), %r8  # i-1
imulq %rcx, %rsi    # i*n
imulq %rcx, %rax    # (i+1)*n
imulq %rcx, %r8     # (i-1)*n
addq %rdx, %rsi     # i*n+j
addq %rdx, %rax     # (i+1)*n+j
addq %rdx, %r8      # (i-1)*n+j
```

```assembly
imulq %rcx, %rsi    # i*n
addq %rdx, %rsi     # i*n+j
movq %rsi, %rax     # i*n+j
subq %rcx, %rax     # i*n+j-n
leaq (%rsi,%rcx), %rcx # i*n+j+n
```
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');
}
```

*Extracted from 213 lab submissions, Fall 1998*
Performance

- Time quadruples when double string length
- Quadratic performance

![Graph showing CPU seconds vs string length]
Why is That?

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

```c
/* My version of strlen */
size_t strlen(const char *s) {
    size_t length = 0;
    while (*s != '\0') {
        s++;
        length++;
    }
    return length;
}
```
void lower(char *s)
{
    int i;
    for (i = 0; i < strlen(s); i++)
        if (s[i] >= 'A' && s[i] <= 'Z')
            s[i] -= ('A' - 'a');
}

void lower(char *s)
{
    int i;
    int len = strlen(s);
    for (i = 0; i < len; 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

<table>
<thead>
<tr>
<th>String Length</th>
<th>CPU Seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>256</td>
<td>0.000001</td>
</tr>
<tr>
<td>512</td>
<td>0.00001</td>
</tr>
<tr>
<td>1k</td>
<td>0.0001</td>
</tr>
<tr>
<td>2k</td>
<td>0.001</td>
</tr>
<tr>
<td>4k</td>
<td>0.01</td>
</tr>
<tr>
<td>8k</td>
<td>0.1</td>
</tr>
<tr>
<td>16k</td>
<td>1</td>
</tr>
<tr>
<td>32k</td>
<td>10</td>
</tr>
<tr>
<td>64k</td>
<td>100</td>
</tr>
<tr>
<td>128k</td>
<td>1000</td>
</tr>
<tr>
<td>256k</td>
<td>100000</td>
</tr>
</tbody>
</table>

Graph showing CPU time in seconds for different string lengths, with two performance categories labeled: lower1 and lower2.
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;
}
```