CS342 Computer Security **Handout # 5**
Prof. Lyn Turbak **Monday, Oct. 04, 2010**
Wellesley College

Lab 4: Introduction to x86 Assembly

Reading:
Hacking, 0x250, 0x270

Overview

Today, we continue to cover low-level programming details that are essential for understanding software vulnerabilities like buffer overflow attacks and format string exploits. You will get exposure to the following:

- Understanding conventions used by compiler to translate high-level programs to low-level assembly code (in our case, using Gnu C Compiler (gcc) to compile C programs).
- The ability to read low-level assembly code (in our case, Intel x86).
- Understanding how assembly code instructions are represented as machine code.
- Being able to use `gdb` (the Gnu Debugger) to read the low-level code produced by `gcc` and understand its execution.

In tutorials based on this handout, we will learn about all of the above in the context of some simple examples.

Intel x86 Assembly Language

Since Intel x86 processors are ubiquitous, it is helpful to know how to read assembly code for these processors.

We will use the following terms: *byte* refers to 8-bit quantities; *short word* refers to 16-bit quantities; *word* refers to 32-bit quantities; and *long word* refers to 64-bit quantities.

There are many registers, but we mostly care about the following:

- EAX, EBX, ECX, EDX are 32-bit registers used for general storage.
- ESI and EDI are 32-bit indexing registers that are sometimes used for general storage.
- ESP is the 32-bit register for the *stack pointer*, which holds the address of the element currently at the top of the stack. The stack grows "up" from high addresses to low addresses. So pushing an element on the stack decrements the stack pointer, and popping an element increments the stack pointer.
- EBP is the 32-bit register for the *base pointer*, which is the address of the current activation frame on the stack (more on this below).
- EIP is the 32-bit register for the *instruction pointer*, which holds the address of the next instruction to execute.

At the end of this handout is a two-page "Code Table" summarizing Intel x86 instructions. The Code Table uses the standard Intel conventions for writing instructions. But the GNU assembler in Linux uses the so-called AT&T conventions, which are different. Some examples:

AT&T Format	**Intel Format**	**Meaning**
`movl $4, %eax`	`movl eax, 4`	Load 4 into EAX.
`addl %ebx, %eax`	`addl eax, ebx`	Put sum of EAX and EBX into EAX.
`pushl $X`	`pushl [X]`	Push the contents of memory location named X onto the stack.
`popl %ebp`	`popl ebp`	Pop the top element off the stack and put it in EBP.
`movl %ecx, -4(%esp)`	`movl [esp - 4] ecx`	Store contents of ECX into memory at an address that is 4 less than the contents of ESP.
`leal 12(%ebp), %eax`	`leal eax [ebp + 12]`	Load into EAX the address that is 12 more than the contents of EBP.
`movl (%ebx,%esi,4), %eax`	`movl eax [ebx + 4*esi]`	Load into EAX the contents of the memory location whose address is the sum of the contents of EBX and four times the contents of ESI.
`cmpl $0, 8(%ebp)`	`cmpl [ebp + 8] 0`	Compare the contents of memory at an address 8 more than the contents of EBP with 0. (This comparison sets flags in the machine that can be tested by later instructions.)
`jg L1`	`jg L1`	Jump to label `L1` if last comparison indicated “greater than”.
`jmp L2`	`jmp L2`	Unconditional jump to label `L2`.
`call printf`	`call printf`	Call the `printf` subroutine.

We will focus on instructions that operate on 32-bit words (which have the `l` suffix), but there are ways to manipulate quantities of other sizes (the `b` suffix operates indicates byte operations and the `w` suffix indicates 16-bit-word operations).

Typical Calling Conventions for Compiled C Code

The stack is typically organized into a list of activation frames. Each frame has a base pointer that points to highest address in the frame; since stacks grow from high to low, this is at the bottom of the frame:[1]

```
                    <local vars for F>
                    ....
                    <local vars for F>
  <base pointer>:   <old base pointer (of previous frame)>
                     # ----Bottom of frame for F----
                     <return address for call to F>
                     <arg 1 for F>
                     <arg 2 for F>
                     ...
                     <arg n for F>
                     <local vars for caller of F>
                     ...
                     <local vars for caller of F>
<old base pointer>: <older base pointer>
                     # ----Bottom of frame for caller of F----
```

To maintain this layout, the calling convention is as follows:

1. The caller pushes the subroutine arguments on the stack from last to first.

2. The caller uses the `call` instruction to call the subroutine. This pushes the return address (address of the instruction after the `call` instruction) on the stack and jumps to the entry point of the called subroutine.

3. In order to create a new frame, the callee pushes the old base pointer and remembers the current stack address as the new base pointer via the following instructions:

```
    pushl %ebp              # \ Standard callee entrance
    movl %esp, %ebp         # /
```

4. The callee then allocates local variables and performs its computation.

When the callee is done, it does the following to return:

1. It stores the return value in the EAX register.

2. It pops the current activation frame off the stack via:

```
movl %ebp, %esp
popl %ebp
```

   This pair of instructions is often written as the `leave` pseudo-instruction.

3. It returns control to the caller via the `ret` instruction, which pops the return address off the stack and jumps there.

4. The caller is responsible for removing arguments to the call from the stack.

[1] We will follow the convention of displaying memory on the page increasing from low to high addresses.

Writing Assembly Code by Hand for the SOS Program

Following the above conventions, we can write assembly code by hand for the sum-of-squares program we studied last time:

```
/* Contents of the file sos.c */

#include <stdio.h>

/* Calculates the square of integer x */
int sq (int x) {
  return x*x;
}

/* Calculates the sum of squares of a integers y and z */
int sos (int y, int z) {
  return sq(y) + sq(z);
}

/* Reads two integer inputs from command line
   and displays result of SOS program */
int main (int argn, char** argv) {
  int a = atoi(argv[1]);
  int b = atoi(argv[2]);
  printf("sos(%i,%i)=%i\n", a, b, sos(a,b));
}
```

```
# HANDWRITTEN ASSEMBLY CODE FOR THE SOS PROGRAM (in the file sos.s)

        .section .rodata        # Begin read-only data segment
        .align 32               # Address of following label will be a multiple of 32
.fmt:                           # Label of SOS format string
        .string "sos(%i,%i)=%i\n" # SOS format string
        .text                   # Begin text segment (where code is stored)
        .align 4                # Address of following label will be a multiple of 4
sq:                             # Label for sq() function
        pushl   %ebp            # \ Standard callee entrance
        movl    %esp, %ebp      # /
        movl    8(%ebp), %eax   # result <- x
        imull   8(%ebp), %eax   # result <- x*result
        leave                   # \ Standard callee exit
        ret                     # /
       .align 4                 # Address of following label will be a multiple of 4
sos:                            # Label for sos() function
        pushl   %ebp            # \ Standard callee entrance
        movl    %esp, %ebp      # /
        pushl   8(%ebp)         # push y as arg to sq()
        call    sq              # %eax <- sq(y)
        movl    %eax, %ebx      # save sq(y) in %ebx
        addl    $4, %esp        # pop y off stack (not really necessary)
        pushl   12(%ebp)        # push z as arg to sq()
        call    sq              # %eax <- sq(z)
        addl    $4, %esp        # pop z off stack (not really necessary)
        addl    %ebx, %eax      # %eax <- %eax + %ebx
        leave                   # \ Standard callee exit
        ret                     # /
        .align 4                # Address of following label will be a multiple of 4
```

```
.globl main                     # Main entry point is visible to outside world
main:                           # Label for main() function
        pushl   %ebp            # \ Standard callee entrance
        movl    %esp, %ebp      # /

        # int a = atoi(argv[1])
        subl    $8, %esp        # Allocate space for local variables a and b
        movl    12(%ebp), %eax  # %eax <- argv pointer
        addl    $4, %eax        # %eax <- pointer to argv[1]
        pushl   (%eax)          # push string pointer in argv[1] as arg to atoi()
        call    atoi            # %eax <- atoi(argv[1])
        movl    %eax, -4(%ebp)  # a <- %eax
        addl    $4, %esp        # pop arg to atoi off stack

        # int b = atoi(argv[2])
        movl    12(%ebp), %eax  # %eax <- argv pointer
        addl    $8, %eax        # %eax <- pointer to argv[2]
        pushl   (%eax)          # push string pointer in argv[2] as arg to atoi()
        call    atoi            # %eax <- atoi(argv[2])
        movl    %eax, -8(%ebp)  # b <- %eax
        addl    $4, %esp        # pop arg to atoi off stack

        # printf("sos(%i,%i)=%d\n", a, b, sos(a,b))#
        # First calculate sos(a,b) and push it on stack
        pushl   -8(%ebp)        # push b
        pushl   -4(%ebp)        # push a
        call    sos             # %eax <- sos(a,b)
        addl    $8, %esp        # pop args to sos off stack
        pushl   %eax            # push sos(a,b)
        # Push remaining args to printf
        pushl   -8(%ebp)        # push b
        pushl   -4(%ebp)        # push a
        pushl   $.fmt           # push format string for printf
        # Now call printf
        call    printf
        addl    $16, %esp       # pop args to printf off stack (not really necessary)
        leave                   # \ Standard callee exit
        ret                     # /
# END OF ASSEMBLY CODE FILE
```

Here's how to compile and run our hand-written code:

```
[cs342@localhost assembly-intro] gcc -o sos-by-hand sos-by-hand.s
[cs342@localhost assembly-intro] sos-by-hand 3 4
sos(3,4)=25
[cs342@localhost assembly-intro] sos-by-hand 10 5
sos(10,5)=125
```

Compiling `sos.c` to Assembly Code

Writing assembly code by hand is tedious and error prone. This is why compilers were invented! They automatically translate code that's written at a higher level than assembly[2] into assembly instructions. These instructions can be assembled into even lower level machine code – the bits that can actually be executed on a processor like an x86.

We can use `gcc` to compile `sos.c` into assembly code as follows:[3]

```
    [cs342@localhost assembly-intro] gcc -S sos.c
```

This creates the file `sos.s` shown below. Note that the code is a bit different than what we generated by hand.

```
# Contents of the assembly file sos.s created by gcc -S sos.c
        .file    "sos.c"
        .text
.globl sq
        .type    sq, @function
sq:
        pushl    %ebp
        movl     %esp, %ebp
        movl     8(%ebp), %eax
        imull    8(%ebp), %eax
        popl     %ebp
        ret
        .size    sq, .-sq
.globl sos
        .type    sos, @function
sos:
        pushl    %ebp
        movl     %esp, %ebp
        pushl    %ebx
        subl     $4, %esp
        movl     8(%ebp), %eax
        movl     %eax, (%esp)
        call     sq
        movl     %eax, %ebx
        movl     12(%ebp), %eax
        movl     %eax, (%esp)
        call     sq
        leal     (%ebx,%eax), %eax
        addl     $4, %esp
        popl     %ebx
        popl     %ebp
        ret
        .size     sos, .-sos
        .section .rodata
.LC0:
        .string  "sos(%i,%i)=%d\n"
        .text
```

[2] Of course, we know that C is not at *that* much higher a level than assembly, but I digress ...

[3] These are the results we get if we compile the code on a 32-bit machine like those in the Linux microfocus cluster. We get very different results if we compile the code on a 64-bit machine like puma.

```
.globl main
        .type    main, @function
main:
        pushl    %ebp
        movl     %esp, %ebp
        andl     $-16, %esp
        subl     $32, %esp
        movl     12(%ebp), %eax
        addl     $4, %eax
        movl     (%eax), %eax
        movl     %eax, (%esp)
        call     atoi
        movl     %eax, 24(%esp)
        movl     12(%ebp), %eax
        addl     $8, %eax
        movl     (%eax), %eax
        movl     %eax, (%esp)
        call     atoi
        movl     %eax, 28(%esp)
        movl     28(%esp), %eax
        movl     %eax, 4(%esp)
        movl     24(%esp), %eax
        movl     %eax, (%esp)
        call     sos
        movl     $.LC0, %edx
        movl     %eax, 12(%esp)
        movl     28(%esp), %eax
        movl     %eax, 8(%esp)
        movl     24(%esp), %eax
        movl     %eax, 4(%esp)
        movl     %edx, (%esp)
        call     printf
        leave
        ret
        .size    main, .-main
        .ident   "GCC: (GNU) 4.4.1 20090725 (Red Hat 4.4.1-2)"
        .section .note.GNU-stack,"",@progbits
```

Even though the code looks different, it behaves the same way, as demonstrated by compiling it to machine code:

```
[cs342@localhost assembly-intro] gcc -o sos-from-assembly sos.s
[cs342@localhost assembly-intro] sos-from-assembly 3 4
sos(3,4)=25
```

Optimizing `sos.c`

Invoking `gcc` with an optimization flag (`-O1, -O2, -O3`) can create more compact code by using clever optimizations.

```
    [cs342@localhost assembly-intro] gcc -S -O3 -o sos_O3.s sos.c
```

```
# Part of the contents of sos_O3.s created by gcc -S -O3 -o sos_O3.s sos.c
.globl sq
        .type    sq, @function
sq:
        pushl    %ebp
        movl     %esp, %ebp
        movl     8(%ebp), %eax
        popl     %ebp
        imull    %eax, %eax
        ret
        .size    sq, .-sq
        .p2align 4,,15
.globl sos
        .type    sos, @function
sos:
        pushl    %ebp
        movl     %esp, %ebp
        movl     8(%ebp), %eax
        movl     12(%ebp), %edx
        popl     %ebp
        imull    %eax, %eax
        imull    %edx, %edx
        leal     (%edx,%eax), %eax
        ret
```

Using GDB to Disassemble Code

What if we don't have the source code to generate assembly code, but only the binary code? Then we can use the GNU Debugger (gdb) to disassemble the binary, as shown below:

```
[cs342@localhost assembly-intro] gdb sos-from-assembly
GNU gdb (GDB) Fedora (6.8.50.20090302-40.fc11)
Copyright (C) 2009 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "i586-redhat-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...

(gdb) disassemble sq
Dump of assembler code for function sq:
0x080483f4 <sq+0>: push   %ebp
0x080483f5 <sq+1>: mov    %esp,%ebp
0x080483f7 <sq+3>: mov    0x8(%ebp),%eax
0x080483fa <sq+6>: imul   0x8(%ebp),%eax
0x080483fe <sq+10>: pop    %ebp
0x080483ff <sq+11>: ret
End of assembler dump.

(gdb) disassemble 0x080483f4
Dump of assembler code for function sq:
0x080483f4 <sq+0>: push   %ebp
0x080483f5 <sq+1>: mov    %esp,%ebp
0x080483f7 <sq+3>: mov    0x8(%ebp),%eax
0x080483fa <sq+6>: imul   0x8(%ebp),%eax
0x080483fe <sq+10>: pop    %ebp
0x080483ff <sq+11>: ret
End of assembler dump.

(gdb) disassemble sos
Dump of assembler code for function sos:
0x08048400 <sos+0>:  push   %ebp
0x08048401 <sos+1>:  mov    %esp,%ebp
0x08048403 <sos+3>:  push   %ebx
0x08048404 <sos+4>:  sub    $0x4,%esp
0x08048407 <sos+7>:  mov    0x8(%ebp),%eax
0x0804840a <sos+10>: mov    %eax,(%esp)
0x0804840d <sos+13>: call   0x80483f4 <sq>
0x08048412 <sos+18>: mov    %eax,%ebx
0x08048414 <sos+20>: mov    0xc(%ebp),%eax
0x08048417 <sos+23>: mov    %eax,(%esp)
0x0804841a <sos+26>: call   0x80483f4 <sq>
0x0804841f <sos+31>: lea    (%ebx,%eax,1),%eax
0x08048422 <sos+34>: add    $0x4,%esp
0x08048425 <sos+37>: pop    %ebx
0x08048426 <sos+38>: pop    %ebp
0x08048427 <sos+39>: ret
End of assembler dump.
(gdb)
```

A Recursive Factorial Program

Below is a C program for recursively calculating factorials.

```
/* This is the contents of the file fact.c */
int fact (int n) {
  if (n <= 0) {
    return 1;
  } else {
    return n*fact(n-1);
  }
}

int main (int argn, char** argv) {
  int x = atoi(argv[1]);
  printf("fact(%i)=%i\n", x, fact(x));
}
```

Let's compile it and take it for a spin!

```
[cs342@localhost assembly-intro] gcc -o fact fact.c
[cs342@localhost assembly-intro] fact 3
fact(3)=6
[cs342@localhost assembly-intro] fact 4
fact(4)=24
```

Hand-written x86 Assembly for Recursive Factorial Program

Below is the result of hand-compiling the factorial program using the calling conventions studied earlier:

```
# This is the contents of the file fact-by-hand.s

        .section        .rodata # Begin read-only data segment
        .align 32               # Address of following label will be a multiple of 32
.fmt:                           # Label of fact program format string
        .string "fact(%i)=%i\n" # fact program format string
.text                           # Begin text segment (where code is stored)
        .align 4                # Address of following label will be a multiple of 4
fact:                           # Label for factorial function
        pushl %ebp              # \ Standard callee entrance
        movl %esp, %ebp         # /
        cmpl $0, 8(%ebp)        # Compare n and 0
        jg factGenCase          # Jump if greater to general case
#       call print_stack        # Base case: show the stack state using Lyn's stack walker
        movl $1, %eax           # result <- 1
        jmp factRet             # Jump to shared return code
        .align 4                # Address of following label will be a multiple of 4
factGenCase:                    # Label for general case
        movl 8(%ebp), %eax      # %eax <- n
        subl $1, %eax           # %eax <- (n-1)
        pushl %eax              # push (n-1) for recursive call to factorial
        call fact               # call fact(n-1)
        imull 8(%ebp), %eax     # result <- n*result
        .align 4                # Address of following label will be a multiple of 4
factRet:                        # Shared return code for factorial
        leave                   # \ Standard callee exit
        ret                     # /
        .align 4                # Address of following label will be a multiple of 4
.globl main                      # Main entry point is visible to outside world
main:                           # Label for main() function
        pushl %ebp              # \ Standard callee entrance
        movl %esp, %ebp         # /
        subl $4, %esp           # Allocate space for local variable x
        movl 12(%ebp), %eax     # %eax <- argv pointer
        addl $4, %eax           # %eax <- pointer to argv[1]
        pushl (%eax)            # push string pointer in argv[1] as arg to atoi()
        call atoi               # %eax <- atoi(argv[1])
        movl %eax, -4(%ebp)     # Save x for later printf
        pushl %eax              # Push x for fact call
        call fact               # Call fact(x)
        pushl %eax              # Push result of fact(x) for printf
        pushl -4(%ebp)          # push x for printf
        pushl $.fmt             # push format string for printf
        call printf             # Call printf("fact(%i)=%i\n", n, fact(n))
        leave                   # \ Standard callee exit
        ret                     # /
```

We can compile and run this as follows:

```
[cs342@localhost assembly-intro] gcc -o fact-by-hand fact-by-hand.s
[cs342@localhost assembly-intro] fact-by-hand 5
fact(5)=120
```

Using GDB again

Suppose we uncomment the line in `fact-by-hand.s` containing `call print_stack` and recompile as follows:

```
    [cs342@localhost assembly-intro] gcc -c print_stack.c
    [cs342@localhost assembly-intro] gcc -o fact-by-hand fact-by-hand.s
```

Here, the `-c` option creates a `.o` object file for the function `print_stack` defined in `print_stack.c` (not shown here). This function displays a representation of the stack when invoked.

Let's use `gbd` to disassemble `fact-by-hand`:

```
[cs342@localhost assembly-intro] gdb fact-by-hand
GNU gdb (GDB) Fedora (6.8.50.20090302-40.fc11)
Copyright (C) 2009 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "i586-redhat-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...

(gdb) disassemble main
Dump of assembler code for function main:
0x08048480 <main+0>: push   %ebp
0x08048481 <main+1>: mov    %esp,%ebp
0x08048483 <main+3>: sub    $0x4,%esp
0x08048486 <main+6>: mov    0xc(%ebp),%eax
0x08048489 <main+9>: add    $0x4,%eax
0x0804848c <main+12>: pushl  (%eax)
0x0804848e <main+14>: call   0x8048374 <atoi@plt>
0x08048493 <main+19>: mov    %eax,-0x4(%ebp)
0x08048496 <main+22>: push   %eax
0x08048497 <main+23>: call   0x8048454 <fact>
0x0804849c <main+28>: push   %eax
0x0804849d <main+29>: pushl  -0x4(%ebp)
0x080484a0 <main+32>: push   $0x8048a00
0x080484a5 <main+37>: call   0x8048364 <printf@plt>
0x080484aa <main+42>: leave
0x080484ab <main+43>: ret
End of assembler dump.

(gdb) disassemble fact
Dump of assembler code for function fact:
0x08048454 <fact+0>: push   %ebp
0x08048455 <fact+1>: mov    %esp,%ebp
0x08048457 <fact+3>: cmpl   $0x0,0x8(%ebp)
0x0804845b <fact+7>: jg     0x804846c <factGenCase>
0x0804845d <fact+9>: call   0x80486fa <print_stack>
0x08048462 <fact+14>: mov    $0x1,%eax
0x08048467 <fact+19>: jmp    0x804847c <factRet>
0x08048469 <fact+21>: lea    0x0(%esi),%esi
End of assembler dump.
(gdb)
```

Displaying the Stack

The hand-compiled factorial program uses a stack display program named `print_stack` that displays the state of the stack when it's called. Let's see what it does in the case of invoking the factorial program on 3:[4]

```
----------------------------------TOP-OF-STACK---------------------------------
bfc2e688: bfc2e690
bfc2e68c: 08048462
bfc2e690: bfc2e69c
------------------
bfc2e694: 08048478
bfc2e698: 00000000
bfc2e69c: bfc2e6a8
------------------
bfc2e6a0: 08048478
bfc2e6a4: 00000001
bfc2e6a8: bfc2e6b4
------------------
bfc2e6ac: 08048478
bfc2e6b0: 00000002
bfc2e6b4: bfc2e6c8
------------------
bfc2e6b8: 0804849c
bfc2e6bc: 00000003
bfc2e6c0: bfc2f647 ->3
bfc2e6c4: 00000003
bfc2e6c8: bfc2e748 ->
------------------
bfc2e6cc: 0014da86
bfc2e6d0: 00000002
bfc2e6d4: bfc2e774
bfc2e6d8: bfc2e780
bfc2e6dc: 0045b000
bfc2e6e0: 00000000
bfc2e6e4: ffffffff
bfc2e6e8: 00133fc4
bfc2e6ec: 0804826e
bfc2e6f0: 00000001
bfc2e6f4: bfc2e730 ->
bfc2e6f8: 00122de6
bfc2e6fc: 00134818
bfc2e700: 0045b2d8
bfc2e704: 002a2ff4
bfc2e708: 00000000
bfc2e710: bfc2e748 ->
bfc2e714: 58fc02d6
bfc2e718: f48535a9
bfc2e71c: 00000000
bfc2e728: 00000002
bfc2e72c: 080483a0
bfc2e730: 00000000
bfc2e734: 00128fd0
bfc2e738: 0014d9ab
bfc2e73c: 00133fc4
bfc2e740: 00000002
bfc2e744: 080483a0
bfc2e748: 00000000
------------------
bfc2e74c: 080483c1
bfc2e750: 08048480
```

[4] A problem in the `print_stack` function prevents it from printing the whole stack and returning the value. But you get the idea ...

```
bfc2e754: 00000002
bfc2e758: bfc2e774
bfc2e75c: 08048930
bfc2e760: 08048920
bfc2e764: 001237e0
bfc2e768: bfc2e76c
bfc2e76c: 00134660
bfc2e770: 00000002
bfc2e774: bfc2f63a ->fact-by-hand
bfc2e778: bfc2f647 ->3
bfc2e77c: 00000000
bfc2e780: bfc2f649 ->BIBINPUTS=:/home/fturbak/church/lib/bibtex
bfc2e784: bfc2f674 ->DVIPSHEADERS=.:/usr/share/texmf/dvips//:/home/fturbak/lib/tex/psfonts/cmpsfont/pfb:/home/fturbak/l
bfc2e788: bfc2f709 ->TWHOMEDIR=/home/cs307/public_html/tw
bfc2e78c: bfc2f72e ->HOSTNAME=localhost.localdomain
bfc2e790: bfc2f74d ->BSTINPUTS=:/home/fturbak/church/lib/bibtex:/home/fturbak/lib/tex/jfp
bfc2e794: bfc2f792 ->SHELL=/bin/bash
bfc2e798: bfc2f7a2 ->TERM=dumb
bfc2e79c: bfc2f7ac ->CATALINA_HOME=/home/tomcat
bfc2e7a0: bfc2f7c7 ->HISTSIZE=1000
bfc2e7a4: bfc2f7d5 ->SSH_CLIENT=149.130.163.181 4858 22
bfc2e7a8: bfc2f7f8 ->OLDPWD=/home/cs342/download/assembly-intro
bfc2e7ac: bfc2f823 ->QTDIR=/usr/lib/qt-3.3
bfc2e7b0: bfc2f839 ->QTINC=/usr/lib/qt-3.3/include
bfc2e7b4: bfc2f857 ->SSH_TTY=/dev/pts/1
bfc2e7b8: bfc2f86a ->USER=cs342
bfc2e7bc: bfc2f875 ->EMACS=t
bfc2e7c0: bfc2f87d ->LS_COLORS=
bfc2e7c4: bfc2f888 ->TERMCAP=
bfc2e7c8: bfc2f891 ->COLUMNS=80
bfc2e7cc: bfc2f89c ->MAIL=/var/spool/mail/cs342
bfc2e7d0: bfc2f8b7 ->PATH=/usr/java/sdk/bin:/usr/network/bin:/usr/local/smlnj/bin:/usr/lib/qt-3.3/bin:/usr/kerberos/sbi
bfc2e7d4: bfc2f98c ->PWD=/home/cs342/download/assembly-intro
bfc2e7d8: bfc2f9b4 ->LANG=en_US.UTF-8
bfc2e7dc: bfc2f9c5 ->SSH_ASKPASS=/usr/libexec/openssh/gnome-ssh-askpass
bfc2e7e0: bfc2f9f8 ->TEXINPUTS=:/home/cs230/lib/tex:/home/cs342/lib/tex:/home/fturbak/lib/tex:/home/cs230/lib/tex:/home
bfc2e7e4: bfc2fcd3 ->SHLVL=2
bfc2e7e8: bfc2fcdb ->HOME=/home/cs342
bfc2e7ec: bfc2fcec ->LOGNAME=cs342
bfc2e7f0: bfc2fcfa ->PRINTER=minil
bfc2e7f4: bfc2fd08 ->QTLIB=/usr/lib/qt-3.3/lib
bfc2e7f8: bfc2fd22 ->CVS_RSH=ssh
bfc2e7fc: bfc2fd2e ->CLASSPATH=:/home/cs230/download/HiLo:/home/cs230/download/TextFun:/home/cs230/download/TextStats:/
bfc2e800: bfc2fe53 ->SSH_CONNECTION=149.130.163.181 4858 149.130.136.42 22
bfc2e804: bfc2fe89 ->NPX_PLUGIN_PATH=/usr/java/j2sdk1.4.0/jre/plugin/i386/ns4
bfc2e808: bfc2fec2 ->LESSOPEN=|/usr/bin/lesspipe.sh %s
bfc2e80c: bfc2fee4 ->TWLOADPATH=.:/home/cs307/public_html/tw/textures:/home/cs307/public_html/tw/objects:/home/cs307/pu
bfc2e810: bfc2ff95 ->DISPLAY=localhost:11.0
bfc2e814: bfc2ffac ->INSIDE_EMACS=23.1.1,comint
bfc2e818: bfc2ffc7 ->G_BROKEN_FILENAMES=1
bfc2e81c: bfc2ffdc ->_=./fact-by-hand
bfc2e820: 00000000
bfc2e824: 00000020 [^@^@^@ ]
bfc2e828: 003b9414
bfc2e82c: 00000021 [^@^@^@!]
bfc2e830: 003b9000
bfc2e834: 00000010
Segmentation fault
```