CSE 451: Operating Systems
Spring 2003

Lecture 1
C Hacking

Steve Gribble
gribble@cs.washington.edu
323B Sieg Hall

Today's agenda

- Administrivia
- Crash course on C programming
  - Wednesday: debugging
  - Friday: performance analysis

Course overview

- everything you need to know will be on the course web page:

  http://www.cs.washington.edu/451

- your instructor: Brian Bershad
  - your 1st week C drill sergeant: Steve Gribble
- your TAs:
  - Valentin Razmov
  - Doug Buxton
- please come to your lab sections this week

C hacking

- why bother with C anymore?
  - most operating systems are written in C
    - for performance and control
  - C forces you to understand "the layer below"
    - yields a better mental model of how software works
    - understanding at this level will help you with higher-level languages too!
  - C demonstrates why "better" languages are important
    - Java: type safety, object oriented-ness, garbage collection
    - C: none of the above

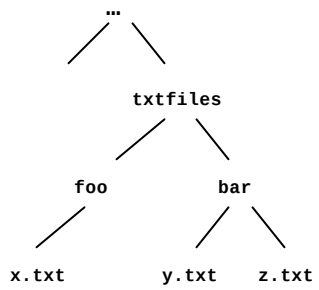
C is a level below Java

- the good news
  - you have very precise control over resources
  - you can carefully optimize your code
  - you are not at the mercy of a garbage collector, a type system, or a JIT compiler
- the bad news
  - like all power tools, if misused C can badly hurt you
    - memory leaks
    - corrupt data structures
    - pointer errors and ensuing crashes
  - there isn't a powerful class library
    - STL is the closest (for C++), but it's pretty scruffy

Your assignment for this week

- to build a simple search engine in C
  - crawl a subtree of a Unix filesystem *(not the web)*
  - parse .txt files extracting out words *(not HTML)*
  - build up an "inverted index" in memory *(not over a cluster)*
    - a mapping from words to documents
  - prompt the user for words and print results *(not a website)*
- what you'll learn how to do:
  - build simple data structures in C
    - master memory allocation and pointers
  - learn about Unix system calls
  - do a little performance analysis

Crawling…

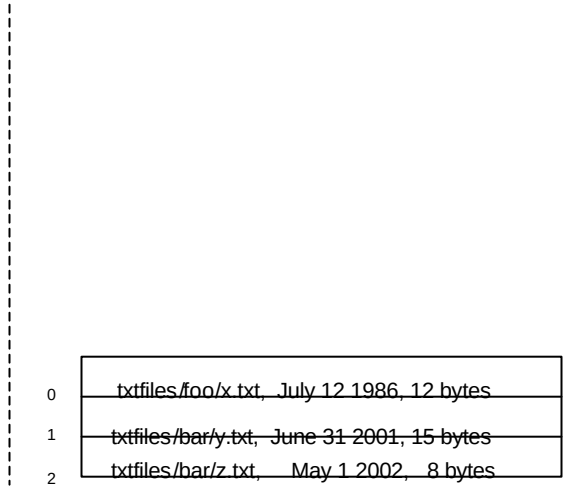


x.txt: “Test file.\r\n”
y.txt: “Another test file.\r\n”
z.txt: “Another.”

- recurse through a subtree of the filesystem
- avoid falling into “symlink” traps
- open, parse, and index all of the .txt files

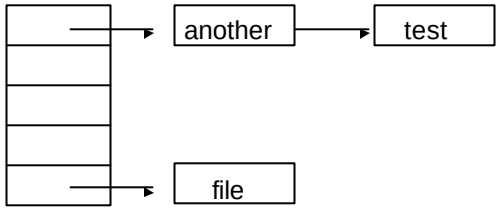
Inverted indexing

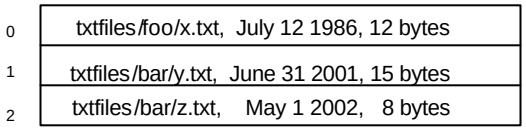
x.txt: “Test file.\r\n”
y.txt: “Another file.\r\n”
z.txt: “Another.”



Inverted indexing

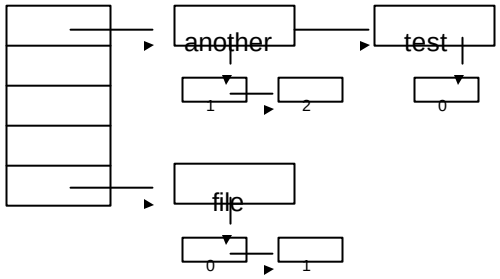
x.txt: “Test file.\r\n”
y.txt: “Another file.\r\n”
z.txt: “Another.”

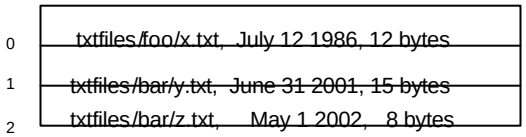




Inverted indexing

x.txt: “Test file.\r\n”
y.txt: “Another file.\r\n”
z.txt: “Another.”





Some hints

- learn how to use “man”
  - man man
  - man uname
  - man fopen
  - man fgets
  - man malloc
  - man free
  - man scanf
  - man printf

C programming: a crash course

(a complicated) hello world in C

- hello_world.c

```
#include <stdlib.h>
#include "hello.h"

void main(void) {
   hello(4);
   exit(0);
}
```

- hello.c

```
#include <stdio.h>

void hello(int num_times) {
   int  i;
   for (i=0; i<num_times; i++){
      printf("Hello %d!\n", i);
   }
}
```

- hello.h

```
void hello(int num_times);
```


compiling, linking, running

```
[fiji]$ gcc –o hello.o –c hello.c
[fiji]$ gcc –o hello_world.o –c hello_world.c
[fiji]$
[fiji]$ gcc –o hello hello_world.o hello.o
[fiji]$
[fiji]$ ./hello
Hello 0!
Hello 1!
Hello 2!
Hello 3!
[fiji]$
```

- C source code → object files → executable

compile *link*

Data types in C

- the usual assortment of data types
  - int — signed integers, usually 32 bits long
  - short — signed integers, usually 16 bits long
  - long — signed integers, either 32 or 64 bits long
  - char — one byte of memory (8 bits long)
    - can add “unsigned” in front, e.g., unsigned short
  - float — signed floating point number
  - double — signed, double-precision floating point

- arrays

```
int i[5];           // declares i to be an array of size 5
i[0] = 12;          // assigns value “12” to element 0 of i
i[10] = 14;         // what happens here?
i[4] = 0x1FFFFFFF;  // largest int value [2s complement]
i[4] = 0xFFFFFFFF;  // -1  [2s complement]
```


Advanced data types

- you can define your own datatypes with “typedef”

```
typedef unsigned char  gribble_int;
   ...
gribble_int   gi = 12;
```

- complex structures using “struct”

```
typedef struct foo_struct {
   int                x;
   char               y;
} foo;


void bar(void) {
  struct foo_struct    fs;
  foo                  fs2;

  fs.x = 12;  fs.y = ‘z’;  fs2.x = 10; fs2.y = ‘x’;
}
```

Typecasting

- C lets you change the type of an expression, using type-casting (or “casting” for short)

```
typedef long gribble_long;

gribble_long  gl;
long          l;

l = (long) gl;
gl = (gribble_long) 12;
```

- Careful! C assumes you know what you’re doing…
  - potential trouble if amount of memory backing 2 types differs

```
int x = 0x87654321;
char y;

y = (char) x;    printf(“%d\n”, (int) y);
```


Memory management in Java

- the Java runtime manages memory on your behalf
  - you never allocate memory directly
    - instead, you instantiate objects using “new”

```
String x = new String(“hello world”);
```

  - the garbage collector frees memory for you
    - figures out when an object can be reclaimed (no more references exist to that object)

Memory management in C

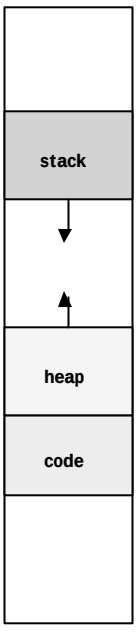
- some memory is managed on your behalf
  - the instructions which implement your functions
    - compiler, linker, and OS collude to allocate this
  - the memory that backs global and “static” variables
    - compiler, linker, and OS collude to allocate this
  - the memory that backs local variables within functions
    - compiler allocates this out of the “stack” when function is called
    - compiler frees this from the stack when function exits

Memory management in C

- you need to manage some memory on your own
  - you need to allocate memory to hold your data structures
    - hash tables, arrays, linked lists, sets, …etc.
    - allocated out of the “heap”
  - you must free this memory when you are done with it!
    - this is hard: elaborate bookkeeping to keep track of what memory you have allocated and when it is safe to free

```
char *x;     // a “pointer” – we’ll cover this soon

x = (char *) malloc(12);  // allocate 12 bytes
free(x);                  // free memory
```



Memory

- memory is an array of bytes
  - potential addresses from 0 to 2^N-1
  - for Intel x86, N=32 (32-bit architecture)
- each program makes use of three "zones"
  - the heap
    - things you allocate with malloc
  - the stack
    - local variables within functions
    - allocated for you
  - the 'text segment'
    - code instructions, global and static variables
    - OS allocates for you when program is launched
    - "linker" provides a recipe to fill in values

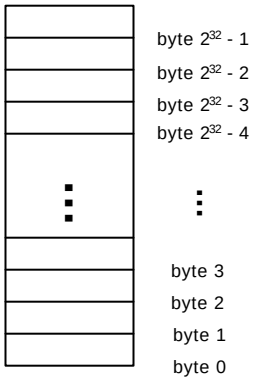
The trickiest data types you'll ever see

- "pointer" data types
  - a pointer contains memory address
    - "points" to a location in memory

```
unsigned char *p;
unsigned char  y = 0x4E;

p = (unsigned char *) 0x00000002;
*p = 0x05;
*(p + 1) = 0x11;
*(p - 1) = 0x3F;
*(p - 2) = y;

printf("%d\n", (int) (*p));
```



Strings: arrays of characters

- Strings in C are just NULL-terminated arrays of chars

```
char *my_string = “Hi!”;
char another_string[4] = {‘H’, ‘i’, ‘!’, ‘\0’};
char *final_string;

final_string = (char *) malloc(4*sizeof(char));
if (final_string == NULL)  exit(-1);
final_string[0] = ‘H’;
final_string[1] = ‘i’;
final_string[2] = ‘!’;
final_string[3] = ‘\0’;

printf(“%s %s %s\n”, my_string, another_string,
       final_string);
```


Pointers and addresses

- & = “ address of ”

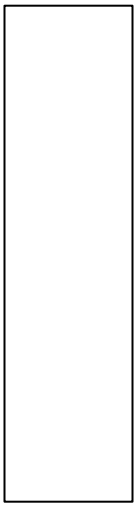
```
int main(void) {
  int x=1, *z;

  z = &x;
  printf(“%d %08x\n”, *z, z);

  z = (int *)
     malloc(2 * sizeof(int));

  *z = y;
  *(z+1) = 100;
  *(z+2) = 121;   // whoops!

  return 0;       // same as exit(0)
}
```



Putting it all together: a linked list

linked_list.h

```
typedef enum {LL_FALSE, LL_TRUE} ll_bool;
typedef struct ll_struct {
   char        id;
   int         value;
   struct ll_struct  *next;
} linked_list;

ll_bool ll_add(linked_list **l, char id, int value);
ll_bool ll_remove(linked_list **l, char id);
ll_bool ll_find(linked_list **l, char id, int *value);
```

linked list: schematic

Using linked_lists

main.c

```
#include <stdio.h>
#include “linked_list.h”

linked_list *l = NULL;   // start out with an empty list

int main(void) {
   int val;

   if (ll_add(&l, ‘a’, 12) == LL_FALSE)
      exit(-1);

   if (ll_find(&l, ‘a’, &val) == LL_FALSE)
      exit(-1);

   if (ll_remove(&l, ‘a’) == LL_FALSE)
      exit(-1);

   return 0;
}
```

Putting it all together: a linked list

linked_list.c

```
#include <stdlib.h>
#include “linked_list.h”

ll_bool ll_add(linked_list **ll, char id, int value) {
  linked_list *ptr, *tmp = *ll;

  // allocate new linked list element
  ptr = (linked_list *) malloc(sizeof(linked_list));
  if (ptr == NULL) return LL_FALSE;
  ptr->next = NULL;
  ptr->id = id;   ptr->value = value;

  if (*ll == NULL) {  *ll = ptr;  return LL_TRUE;  }

  while (tmp->next != NULL) {  tmp = tmp->next; }
  tmp->next = ptr;

  return LL_TRUE;
}
```

Putting it all together: a linked list

```
ll_bool ll_remove(linked_list **ll, char id) {
     linked_list *prev, *tmp;

     if (*ll == NULL) return LL_FALSE;   // empty list
     tmp = *ll;

     if (tmp->id == id) {          // special case:  is the first
        (*ll) = tmp->next;
        free(tmp);
        return LL_TRUE;
     }

     while (tmp->next != NULL) {   // general case:  not the first
        prev = tmp;
        tmp = tmp->next;
        if (tmp->id == id) {
          prev->next = tmp->next;
          free(tmp);
          return LL_TRUE;
        }
     }
     return LL_FALSE;
 }
```