

Recitation Week #2

History, C, and /proc FS

Alejandro Cabrera
Operating Systems
COP4610 / CGS5765

Today's Recitation

- Review of Experience Survey Exercise
- History
 - From Unix to Linux, briefly
 - Distributions
- C-Programming
 - Standard Library
 - Input/Output
 - Strings
- /proc File System
- Project 1

Exercise: What's Wrong with this Code?

```
#include <stdio.h>
#include <string.h>

char *str;

int main(int argc, char **argv)
{
    int ii;
    printf("Value of ii: %d\n", ii);
    printf("First character in str: %c\n", str[0]);
    return 0;
}
```

Exercise: What's Wrong with this Code?

```
#include <stdio.h>
#include <string.h>
```

```
char *str;
```

```
int main(int argc, char **argv)
{
    int ii;
    printf("Value of ii: %d\n", ii);
    printf("First character in str: %c\n", str[0]);
    return 0;
}
```

1. `ii` and `str` are not initialized!

Exercise: What's Wrong with this Code?

```
#include <stdio.h>
#include <string.h>

char *str;

int main(int argc, char **argv)
{
    int ii;
    printf("Value of ii: %d\n", ii);
    printf("First character in str: %c\n", str[0]);
    return 0;
}
```

2. Dangling reference:
this will crash the program!

Exercise: One Possible Fix

```
#include <stdio.h>
#include <string.h>

char *str;

int main(int argc, char **argv)
{
    int ii = 10;
    str = "Clean";
    printf("Value of ii: %d\n", ii);
    printf("First character in str: %c\n", str[0]);
    return 0;
}
```

The Beginning: Unix

- First implemented in AT&T Bell Labs. 1969.
- Thompson and Ritchie publish, “The Unix Time-Sharing System”. 1974
- UC @ Berkeley produces BSD, based off of Unix V6. 1975
- Microsoft licenses Unix V7 as Xenix. 1979.
- AT&T releases System V, V1. 1983
- Linus releases Linux 0.0.1. 1991.

Why Was Unix Born?

- AT&T had to make a choice between using third party OS or developing their own.
 - Chose to implement own OS.
- Born from ideas and work performed on MULTICS OS.
- As a result of work on Unix (first implemented in the assembly language), C was born.

Time Line of Feature Introduction

- B-compiler, UNIX v1 – 1971
 - cat, chdir, chmod, chgrp, ed, mkdir, mkfs, mv, rm...
- C-compiler, Pipes, UNIX v3 – 1973.
- UNIX v5, open-sourced – 1974.
- sh, System V v1, UNIX v7. - 1979
- UNIX v10 (last edition) - 1989

- Somewhere between 1979 and 1989...
 - NFS, TCP/IP, STREAMS...

Standardizing UNIX – IEEE and POSIX

- POSIX – *Portable Operating System Interface for Computing Environments*
- What does this mean?
 - You can count on any modern operating system to adhere to this standard.
 - As long as you develop your programs by using functions available in the POSIX standard, “unistd.h”, your program will be portable to POSIX-compliant systems.

What's Included in the Standard?

- 1003.1 – System calls, library routines
- 1003.2 – Shell, basic UNIX (command-line) utilities
- 1003.3 – Test methods to demonstrate conformance
- 1003.4 – Real-time interfaces

Linux – Humble Beginnings

- Shortly after the final version of UNIX was produced, Linus appeared and published the first version of Linux.
- No OS at the time supported the Intel 80386 32-bit processors – Linus wanted to use his PC with that processor.
- It supported only his hardware – AT hard disks, Intel 80386.
- Since he was working on MINIX, some of the design was based off of MINIX.
- Started by porting bash(1.08) and gcc(1.40).
- For more details, refer to wikipedia or the book: *Just for Fun*.

Linux Today

- Current kernel version, as of September 1, 2009, 9:08pm: 2.6.30.5
- Size, bzip2'd, source: 248MB
 - Not all of it is source. Much documentation is included with the kernel – as .txt files.
- Supports pretty much any platform and device the average user will interact with:
 - USB, SSD, Intel, AMD, ARM, ...
- Released to users as *distributions*, of which there are more than a hundred.

Distributions

- Ubuntu, Fedora, Slackware, SUSE, Red Hat, Debian, Gentoo – all of these are distributions.
- Differences between distributions:
 - Package manager: aptitude, yum, portage, etc.
 - Used to install programs, libraries, documentation.
 - Kernel version: most are behind a few cycles
 - Ubuntu 9.04: kernel 2.6.28.15
 - Windowing Interface: Gnome, KDE, etc.
 - Target audience: power-user, newbie, enterprise, etc.
 - Community

Which Distribution (Distro) to Use?

- The best advice I can give here is to use what you feel most comfortable using.
- If you haven't installed Linux on your computer before, maybe this class is the best time to give it a try!
- Other reasoning to choose one distribution over another:
 - Local standard - Colleagues/coworkers all use same distribution.

Additional References

- <http://www.lwn.net/>
 - Linux news site. Covers distros, conferences, and recent kernel development. Includes many links to free books, documentation, and the like.
- <http://www.kernel.org/>
 - Here's where you can obtain the latest Linux kernel, if you want to get your hands dirty.

Programming with the C Standard Library

- Standardized ways to:
 - Open/close files
 - Read input
 - Write output
 - Manipulate/compare strings
 - Convert strings into numbers and vice-versa
 - Allocate/deallocate memory
 - Sort input (quickly) using a predicate function
 - Search through input (quickly) using a predicate function
 - Much, much more...

Opening and Closing Files

```
FILE *fopen(const char *file_name, const char *flags)
void fclose(FILE *file)
```

- By using FILE pointers, you can access the contents of a file.
- Flags may be:
 - “r” - Read. It's an error if the file doesn't exist.
 - “w” - Write. Replaces existing file or creates a new file.
 - “a” - Append. Adds data to existing file or creates it.
 - “w+” - Equivalent to “r” and “w”.
 - “a+” - Appending and reading.

Format Specifiers:

- Nearly all input/output functions in the C Standard Library derive their flexibility and power from the use of format specifiers and flags.
- Common specifiers:
 - %d – integer value
 - %u – unsigned integer value
 - %f – float value
 - %X – hexadecimal value
 - %s – string
- For a complete reference, refer to:
- <http://www.cplusplus.com/reference/clibrary/cstdio/printf/>

Null-terminated String

- All printing functions in the C standard depend on the presence of a null-character to indicate the end of a string.
- This is why appending a '\0' is so important.
- To illustrate, an implementation of printf looks roughly like this:

```
void printf(const char *format, ...)  
{  
    while(format != '\0'){  
        ...  
        ++format;  
    }  
}
```

Reading Input

```
int scanf(const char *format, ...)  
int sscanf(const char *buffer, const char *format, ...)  
int fscanf(FILE *stream, const char *format, ...)  
int fgets(char *buffer, int num, FILE* stream)
```

- The scanf family uses format specifiers to parse the input.
 - Returns number of **items** read.
- fgets reads into the buffer from the file a given number characters.
 - Very useful for ignoring lines of text or reading an entire (small) file into memory!
 - Returns number of **characters** read.

Writing Output

```
int printf(const char *format, ...)  
int sprintf(const char *buffer, const char  
    *format, ...)  
int fprintf(FILE *stream, const char *format, ...)  
int fputs(char *str, FILE* stream)
```

- The printf family uses format specifiers to format the output.
- fputs places the string *str* into the file.
- All return the number of characters written. In the case of sprintf, an implied null-character is written to the string. This character is not counted.

Comparing Strings

```
int strncmp(const char *str1, const char *str2, size_t num)
int strcmp(const char *str1, const char *str2)
```

- Never compare two char* like this:
 - if(str1 == str2)
- This is a pointer comparison – not what you intend!
 - Use str[n]cmp.
 - Returns +# if first non-matching character is bigger in str1.
 - Returns -# if first non-matching character is bigger in str2.
 - Returns 0 if strings are equal.

Super Secure String Comparison

```
int safe_strncmp(const char *str1, const char *str2)
{
    int len1 = strlen(str1);
    int len2 = strlen(str2);
    if(len1 != len2)
        return ((len1 > len2) ? 1 : -1);
    return strncmp(str1, str2, len1);
}
```

- Pros:
 - **Safe**: Never compares out of bounds.
 - **Behaves like strcmp**: returns correct string order.
- Cons:
 - **Slow**: Always needs to calculate length.
 - Assumes both str1 and str2 are null-terminated.

Copying Strings

```
char *strncpy(char *dest, const char *src, size_t num)
```

```
char *strcpy(char *dest, const char *src)
```

- Both versions copy the source string into the destination.
- Advice:
 - Be sure you have enough space in dest to handle src!
 - Same advice given for strcmp applies here.
 - Can also create a safe_strcpy function.
- Returns pointer to dest string.

Searching Strings

```
char *strstr(const char *pattern, const char *string)
char *strchr(const char character, const char *string)
```

- `strstr()` searches for the first occurrence of `pattern` in `string`.
- `strchr()` searches for first occurrence of `character` in `string`.
- Both return start address of target item in string.
- In both cases, if target is not found, null pointer is returned.

Memory Allocation

```
void *malloc(const size_t num_bytes)
void *calloc(const size_t num_objs, const size_t obj_size)
void free(void *obj)
```

- Dynamic memory allocation is necessary when you don't know the size ahead of time of something in your program.
 - How many characters does str1 need to be able to store to safely perform strcpy(str1, str2)?
- malloc returns a void* that you can cast to the type you need.
- calloc returns a 0-initialized void*.
- Be sure to call free on any memory you dynamically allocate!

Dangling References, Memory Leaks

Memory Leak: Forgetting to free allocated memory.

```
void alloc(int size)
{
    /* Never freed! */
    int *leak = (int *) malloc (10 * sizeof(int));
}
```

Dangling Reference: Trying to use freed or uninitialized memory.

```
void dang_ref(int size)
{
    int *ref = (int *) malloc (10 * sizeof(int));
    free(ref);

    /* Already freed! Cannot be accessed! */
    printf("%d\n", ref[0]);
}
```

Auxiliary Functions

```
void perror(const char *str)
```

```
void qsort(void *base, size_t num, size_t size, int  
(*comparator) (const void *, const void *))
```

```
void bsearch(void *base, size_t size, int  
(*comparator) (const void*, const void*))
```

- Examples of using above functions are provided on the web site.
 - _ perror – Reads global error variable and prints useful information.
 - _ qsort – Uses quick sort to sort a group of objects.
 - _ bsearch – Uses binary search to find an object in a group.
- qsort and bsearch use function pointers. It looks intimidating, but this makes qsort and bsearch **generic and powerful**.

Linux /proc File System

- Contains information about nearly every aspect of the system:
 - CPU(s): model, make, number, features...
 - Memory: How much? Page faults, etc.
 - Hard disks and storage
 - Processes running: parents, children, etc.
- Easy way to access it:
 - `$> cat /proc/cpuinfo`
- Looking up details:
 - `$> man proc`

Linux /proc File System

- Additional references:
 - Advanced Unix Programming, Chp. 7
 - man proc (extremely useful, if terse)
 - Google.

Project 1: Using /proc FS

- Due: September 14, 2009 – The Monday after!
- Project 1 will have you scanning the /proc FS for various pieces of information about the system.
 - Date
 - Up Time
 - Idle Time
 - CPU Info
 - Memory Info
 - Kernel Version
 - Terminal Process Details
- This will involve much I/O and string manipulation.
 - A good chance to put to use all those previous slides!

Project 1: Advice

- Most of the information you need is in the previous slides.
- Research here involves running:
 - `$> cat /proc/<something>`
- ... and seeing what's in a given entry.
- Review the programming examples given in these slides and on the recitation site.
- Start early! You have two days less than two weeks.
 - It can be a short project if you plan your code carefully.

Any questions?