A lot to go over today...

- TAs
- Survey
- Recitation and blackboard websites
- Syllabus
- History of Linux
- Project 1
- C and /proc
- Useful tools
The TAs

• Sarah Diesburg
  – Lead TA, lecturer, grader
  – Office hours W 12:30-1:30pm, F 3:30-4:30pm
  – Office location LOV 105-A

• Alejandro Cabrera
  – Grader
  – Office hours M W 9:00am-10:00am
  – Office location LOV 105-E

• Email us at cop4610t@cs.fsu.edu
Experience Survey

• Please fill out and turn in today or Monday
  – I’ll better know how to assist you
Websites

• http://www.cs.fsu.edu/~cop4610t/
  – Also linked off the main class website
• Blackboard website on http://campus.fsu.edu
  – Discussion board
Syllabus Review
Projects: Partners

- Projects will be completed in pairs
- Choose a partner and send me an email (cop4610t@cs.fsu.edu) with
  - The name and email address of your partner
  - Your FSU card # and made up 4 digit pin (for door access to OS computer lab)
- You need only submit one project per group
Projects: Deliverables

• Source code (*.c, *.h files)
• README
• Makefile
• Project Report
Projects: Grading

• 70% coding, 30% documentation
• Even if you can't get a project to compile, be sure to provide good documentation. You can still get 30% of the points!
Projects: Error Policy

• Document known errors/bugs.
  – There's a bigger penalty if I find an error in your program that you did not.

• Document what you did not complete in a program, if a required feature is missing.
  – It'll be easier to grade.
Projects: Slack Policy

• Each student has three days to use to extend project deadlines.
  – Example: A project is due Monday. You turn it in Wednesday. Two slack days are used, no penalty is incurred. On the next project, it's due on a Tuesday, you turn it in Wednesday - you're now out of slack days.

• Once you're out of slack days, you lose 10 points per day a project is late.

• A project that is more than three days late gets a 0.
A Note on Projects

• Start projects when they're assigned.
  – They're often trickier than they look.
  – Might be a bonus for turning a project in early!

• Ask questions early.
  – If you're asking questions, be it to yourself or to others, you're thinking about the project. This will make it easier to complete them correctly and on time.

• Write small programs to test your program or C language features you don't understand.
  – What does execv("ls", "-l") do?
Good Coding Style

• You'll often have to go back to your program, read through your logic, and update it to fix errors.
• This is MUCH easier if you incorporate good coding style and practices from the start of a project.
• Compare:

```c
int check(char *p) {
    if(p == NULL) return TRUE;
    return FALSE;
}

int isNull(const char *ptr) {
    if(ptr == NULL) {
        return TRUE;
    }
    return FALSE;
}
```
History of Linux
The Beginning: Unix

- First implemented in AT&T Bell Labs, 1969.
- 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...
- 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 2.6.35.4 (as of today’s lecture)
• Supports pretty much any platform and device the average user will interact with. 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.
Project 1

Implementing a Shell
Project 1

• Implement a shell interface that behaves like the shell bash.
• Due in 3 weeks
  – September 17, 2010, 11:59:59pm
• Project specification is on website, as well as test suite and grading sheet
Shell: What is it?

- The Shell is an interpreter for a simple programming language.
- If you type a command it recognizes, it performs that command.
  - How does a Shell recognize a command?
- A Shell may also be run via a file.
  - If the Shell can understand this file, a Shell script, it will execute all the commands in that file.
Shells You May Have Seen Before

- **sh** – The first Shell produced. Came with the first Unix.
- **csh** – The C-shell,
- **ksh** – The Korn shell
- **tcsh** – The Tenex C-shell. (used on linprog)
- **bash** – The Bourne Again Shell (most Linux)
- **DOS/cmd** – The Windows Shell.
Shell Preparation

• I’ll help prepare you first by reviewing basic C functions and coding tools
  – Get very familiar with these, pick a partner, set up project environment, try to start project

• Next week I will go over shell basics, step-by-step
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 use 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.

```c
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 of 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.
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

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

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.
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
Useful Tools
manpages

• Extensive documentation that come with almost all Unix-like systems
• For documentation on C functions or packages
• Examples
  – $> \text{man bash}$
  – $> \text{man strncpy}$
• Sometimes multiple definitions, so use man section numbers
  – ‘man 1 printf’ shows bash printf
  – ‘man 3 printf’ shows C printf
• For more information on sections, see ‘man man’
tar: Tape ARchiver

- **tar**: general purpose archive utility (not just for tapes)
  - Usage: `tar [options] [files]`
  - Originally designed for maintaining an archive of files on a magnetic tape.
  - Now often used for packaging files for distribution
  - If any files are subdirectories, **tar** acts on the entire subtree.
**tar**

- To archive a project
  - `>$ tar cvf <tarfile-name.tar> <directory holding things to tar>`

- To extract a tarball
  - `>$ tar xvf <tarfile-name.tar>`

- To extract a gzipped tarball
  - `>$ tar xvfz <tarfile-name.tar.gz>`
Editors -- Vim

• The vi editor was created by Bill Joy, the founder of Sun Microsystems when he was a graduate student

• The vim editor, vi improved, is the Linux version of the vi editor
  – multiple windows, highlighting text, and command history

• http://www.vim.org/
Editors -- Emacs

• GNU Emacs is an extensible, customizable text editor
  – Content-sensitive editing modes, including syntax coloring, for a variety of file types including plain text, source code, and HTML
• http://www.gnu.org/software/emacs/
Editors -- Others

• Nano and/or pico are also available on most Linux systems
  – Extremely basic
Make

- **make**: A program for building and maintaining computer programs
  - developed at Bell Labs around 1978 by S. Feldman (now at Google)

- Instructions stored in a special format file called a “**makefile**”.

- Look at makefile example on website
Debuggers

• Debuggers let you examine the internal workings of your code while the program runs.
  – Debuggers allow you to set *breakpoints* to stop the program's execution at a particular point of interest and examine variables.
  – To work with a debugger, you first have to recompile the program with the proper debugging options.
  – Use the `–g` command line parameter to *cc, gcc, or g++*
    • Example: `gcc –g –c foo.c`
GDB, the GNU Debugger

- Text-based, invoked with:
  
  ```bash
  gdb [<programfile> <corefile>|<pid>]
  ```

- Issue ‘man gdb’ for more info
GDB Quick-Start

$> ./my.x
$> Segmentation fault
$> gdb ./my.x

(gdb) run

... Segmentation fault

0x08048384 in main() at my.c:4

4 __________________________ *s = 'H';

(gdb) bt

#0 0x08048384 in main() at my.c:4
GNU DDD

```c
(list *) 0x804df80

value = 85
self = 0x804df80
next = 0x804df90

value = 86
self = 0x804df90
next = 0x804dfa0

list->next = new List(a_global + start++);
list->next->next = new List(a_global + start++);
list->next->next->next = list;

(void) list:
    // Display this
    delete list;
delete list->next;
delete list;

// Test
void list()
{
    list;
}

// void ref;
{
    date;
delete date;
}

(gdb) graph display *(list->next->next->self) dependent on 4
(gdb)
```