

CS 521: Systems Programming

Beginning C

Lecture 3

Today's Schedule

- Differences: C vs. Other Languages
- Editors
- Our First Program

Today's Schedule

- **Differences: C vs. Other Languages**
- Editors
- Our First Program

Architectural Differences

- C is compiled to *machine code*, unlike Python or Java
 - The compiled *binary executable* contains instructions that your CPU understands
 - There are several compilers on the market today (gcc, clang, msvc) that transform your code into machine code
- Java runs on a virtual machine (JVM)
- Python is interpreted (translated to machine code on the fly)
- We can achieve better performance with C, but are also given more responsibility
 - Memory management is up to us (no automatic garbage collection)



Main Advantages

- C is fairly simple: the language does not have a multitude of features
- But coming from Java, the syntax is still familiar
- It's the *lingua franca* of systems programming
 - When we operate close to the hardware, it can be much easier to implement than the equivalent Java/Python/etc.
 - Want to contribute to the Linux kernel? It's written in C (including the drivers)
- Performance

Main Disadvantages

- Much less functionality is available in the standard library than other languages
 - For example: no built in list, hashmap, tree, etc.
- Memory leaks
- Segmentation faults (invalid memory access)
- No objects – if you're used to object-oriented programming, C will make you rethink your program

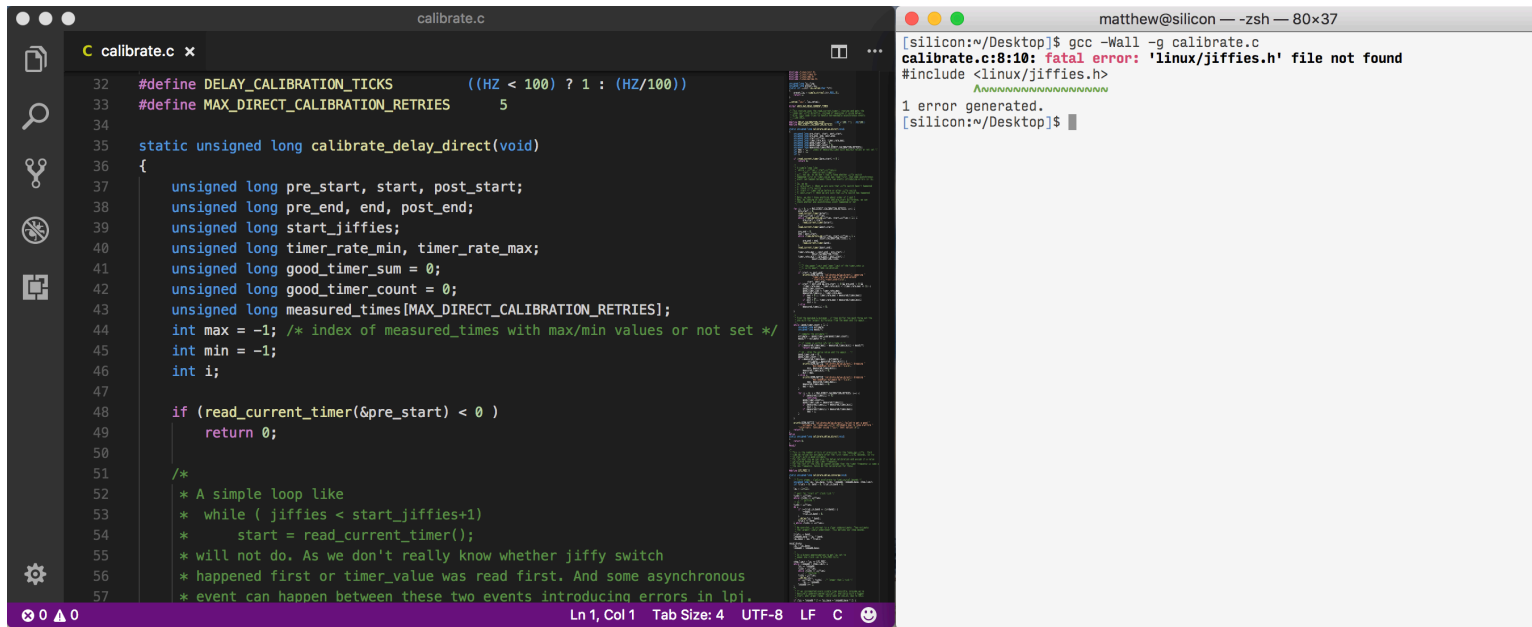
Standardization

- C is not controlled by a single entity; it is a standard
- The standard itself is fairly loose, and allows **undefined behavior** (UB)
 - Basically, the language standard doesn't specify how *everything* should work
 - Compilers can do whatever they want with UB
 - This is why we're making sure we all have the same platform (our VMs) in class  

Today's Schedule

- Differences: C vs. Other Languages
- **Editors**
- Our First Program

Writing C Programs



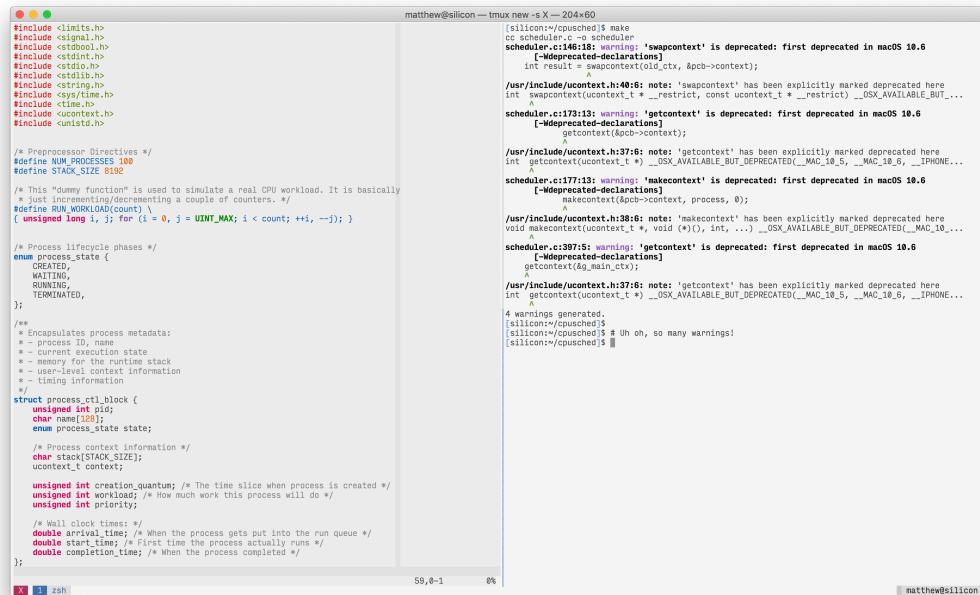
The image shows a code editor window titled 'calibrate.c' and a terminal window. The code editor displays a C program with the following content:

```
32 #define DELAY_CALIBRATION_TICKS ((HZ < 100) ? 1 : (HZ/100))
33 #define MAX_DIRECT_CALIBRATION_RETRIES 5
34
35 static unsigned long calibrate_delay_direct(void)
36 {
37     unsigned long pre_start, start, post_start;
38     unsigned long pre_end, end, post_end;
39     unsigned long start_jiffies;
40     unsigned long timer_rate_min, timer_rate_max;
41     unsigned long good_timer_sum = 0;
42     unsigned long good_timer_count = 0;
43     unsigned long measured_times[MAX_DIRECT_CALIBRATION_RETRIES];
44     int max = -1; /* index of measured_times with max/min values or not set */
45     int min = -1;
46     int i;
47
48     if (read_current_timer(&pre_start) < 0 )
49         return 0;
50
51     /*
52      * A simple loop like
53      * while ( jiffies < start_jiffies+1)
54      *     start = read_current_timer();
55      * will not do. As we don't really know whether jiffy switch
56      * happened first or timer_value was read first. And some asynchronous
57      * event can happen between these two events introducing errors in lpi.
```

The terminal window shows the command `gcc -Wall -g calibrate.c` and the output:

```
calibrate.c:8:10: fatal error: 'linux/jiffies.h' file not found
#include <linux/jiffies.h>
         ^~~~~~
1 error generated.
[silicon:~/Desktop]$
```

Writing C Programs



```
matthew@silicon ~$ cc scheduler.c -o scheduler
scheduler.c:146:18: warning: 'swapcontext' is deprecated: first deprecated in macOS 10.6
[-Wdeprecated-declarations]
int result = swapcontext(old_ctx, &pcb->context);
               ^
/usr/include/ucontext.h:48:6: note: 'swapcontext' has been explicitly marked deprecated here
int swapcontext(ucontext_t * __restrict, const ucontext_t * __restrict) __OSX_AVAILABLE_BUT_DEPRECATED(
      ^
scheduler.c:173:13: warning: 'getcontext' is deprecated: first deprecated in macOS 10.6
[-Wdeprecated-declarations]
    getcontext(&pcb->context);
    ^
/usr/include/ucontext.h:37:6: note: 'getcontext' has been explicitly marked deprecated here
int getcontext(ucontext_t *) __OSX_AVAILABLE_BUT_DEPRECATED(__MAC_10_5, __MAC_10_6, __IPHONE_4_0)
      ^
scheduler.c:177:13: warning: 'makecontext' is deprecated: first deprecated in macOS 10.6
[-Wdeprecated-declarations]
    makecontext(&pcb->context, process, 0);
    ^
/usr/include/ucontext.h:38:6: note: 'makecontext' has been explicitly marked deprecated here
void makecontext(ucontext_t *, void (*)(void), int, ...) __OSX_AVAILABLE_BUT_DEPRECATED(__MAC_10_5, __MAC_10_6, __IPHONE_4_0)
      ^
scheduler.c:397:5: warning: 'getcontext' is deprecated: first deprecated in macOS 10.6
[-Wdeprecated-declarations]
    getcontext(&g_main_ctx);
    ^
/usr/include/ucontext.h:37:6: note: 'getcontext' has been explicitly marked deprecated here
int getcontext(ucontext_t *) __OSX_AVAILABLE_BUT_DEPRECATED(__MAC_10_5, __MAC_10_6, __IPHONE_4_0)
      ^
4 warnings generated.
matthew@silicon ~$ ./scheduler
[Silicon]~/cpu-sched$
[Silicon]~/cpu-sched$
[Silicon]~/cpu-sched$ Uh oh, so many warnings!
[Silicon]~/cpu-sched$
matthew@silicon ~$ zsh
```

Systems Culture

- There is a somewhat different culture in the systems world
- Using an IDE (like Eclipse, IntelliJ, etc) is less common
 - The Unix command line provides many of the usual IDE features
- Many developers prefer to use a text editor and a terminal to write their programs
 - Text editor: edit, save
 - Terminal: compile, run

Recommendation

- Use whatever is comfortable for you
- If you get a chance, try to learn the basics of a terminal editor (even `nano` counts!)
 - Or `vim`, `emacs`, `micro`
- (maybe at least know how to quit vim and emacs...!)
 - By the way, what's the universal "quit" key combination in the terminal?

Today's Schedule

- Differences: C vs. Other Languages
- Editors
- **Our First Program**

Hello World

```
#include <stdio.h>

int main(void)
{
    printf("Hello world!\n");
    return 0;
}
```

...and to run it:

```
cc hello.c -o hello
./hello
```

Slightly More Advanced

```
#include <stdio.h>

void say_hello(int times);

int main(void) {
    say_hello(6);
    return 0;
}

void say_hello(int times) {
    int i;
    for (i = 1; i <= times; ++i) {
        printf("Hello world! (%d)\n", i);
    }
}
```

Differences from Java/Python

- Including libraries looks a bit different
- No public/private etc. access modifiers
- Forward declarations (prototypes)
- No objects
- No exceptions
- A **huge** difference: what return types are used for
 - Often error checking!
- But, there are a lot of similarities...

Similarities to Java/Python

- Arithmetic is mostly the same
- We use `&&`, `||`, and `!=` instead of `and`, `or` and `not`
- `if`, `then`, `else`
- Loops
- Switches

Some Advice

- The similarity between C and Java can be deceiving
- In these small programs, there's hardly a difference!
- However, you will soon see that the structure of larger programs ends up being quite different
 - Since there are no classes, the focus shifts to writing functions
 - Organization might seem a bit less natural, but you can still break your functions up into *modules*