C is the language of OS from late 1970’s onward*. 
(1) OS are coded in C. 
(2) The key OS abstraction, the **process** includes C’s memory and execution model. 

Conclusion: Why must serious OS students know C solidly?

*PL/1 used for Multics was C-like. C was invented to code Unix. Unix started as a simplified Multics.
The beautiful interface

From Tanenbaum, Ch. 1
Interface seen from application software:

- CPU(s). (Instruction Set Architecture from real hardware) limited to user-mode instructions (including traps for system calls)

- plus a Virtual Memory, containing program instructions and data.
C’s memory and execution model fits this well. The OS must implement C’s memory and execution model, so we must understand that model accurately.
i386 Basic Execution Environment, right from Intel 64 and IA-32 Architectures Software Developer’s Manual
How Virtual Memory of One Process is used, typically.

Highest address

Virtual Mem. used by (Linux) kernel, illegal for a user process to access.

command–line arguments, envir. vars.

pointer arrays and C strings.

(The gap. Mostly illegal. Might be used for thread stacks, shared memory or dynamically linked libraries.)

Managed by malloc/free.

uninitialized data, initialized to 0 by exec.

data initialized from executable file

executable code copied from exe. file

stuff used for dynamic linking
The MOST IMPORTANT ABSTRACTION for OS

A PROCESS is (1) One Virtual Memory together with (2) One (or more for multithreading) Virtual CPU(s) that together are CURRENTLY implementing ONE instance of a program in execution and (3) some allocated resources (files, network connections, ... )

In other words, a process is One virtual machine that is currently holding one running instance of one program.
Today’s systems (all but the smallest) provide multiprogramming.

Multiprogramming is many processes in a system at once.

Multiprocessing is multiprogramming with multiple CPUs so several processes can run truly in parallel, i.e., actually at the same time.
Let’s get started ...

1. Form permanent teams.

2. Team Exercise:

   (a) 3 min: Circle and label the word(s) related to EACH feature of C that you identify as important to understand exactly to use the C language in OS studies. For example, functions with file scope are qualified by static and OS kernels are mostly such function, so circle static.

   (b) 5 min: Each (6 person) team choose ONLY the BEST 3 features for me to explain in class.

3. Feature debriefing.

4. Quiz debriefing.

5. Homework for Monday:
Homework for Monday:

- Skim the rest of Chapter 1, find what’s written about `fork` and `exec` (creating and initializing processes) and study that material. Study Chapter 2 up to section 2.1.3, Process Termination.

- Read from Chapter 10, Linux Case Study, Section 10.2, Overview through 10.3.2, Process Management. Lab preparation goals: (1) Use the command line, so try out the examples in Sec. 10.2.3 (unless you’re an expert) and be prepared to do exercises 1-5 or similar ones in a possible quiz. (2) Write, compile, run and observe scenarios with short C programs that demonstrate `fork` and `exec`, so study the examples in which these are used.

- Finally, look closely at the assembly language output file say ASourceFile.s that was written by the command say “`gcc -S ASourceFile.s`”. Next week we’ll introduce the elements needed to understand those codes and work up to assigning homework to analyze exactly what they do. **You will be expected to correlate the C source with the assembly language, especially for loops, stack frames and parameter variables.**

- The first LAB EXERCISE to be assigned Monday will be to experiment with `fork` and `exec` in small programs you write and run on simulated PCs running Linux within the Lab computers.
```c
#include <unistd.h>

static int strlen(char *pch) {
    int len = 0;
    while( *pch != 0 ) {
        len++;
        pch++;
    }
    return len;
}

int main( int argc, char* argv[]) {
    if( argc != 2 ) {
        write(1,"Wrong num. Args\n", 17);
        return 1;
    }
    else {
        write(1, argv[1], strlen(argv[1]));
        write(1, "\n", 1);
        return 0;
    }
}
```
#include <unistd.h>

static int strlen(char *pch)
{
    int len = 0;
    while( *pch != 0 ) {
        len++;
        pch++;
    }
    return len;
}

int main( int argc, char* argv[] ) {
    if( argc != 2 ) {
        write(1,"Wrong num. Args\n", 17);
        return 1;
    } else {
        write(1, argv[1], strlen(argv[1]));
        write(1, "\n", 1);
        return 0;
    }
}
#include <unistd.h>

static int strlen(char *pch)
{
    int len = 0;
    while( *pch != 0 ) {
        len++;
        pch++;
    }
    return len;
}

int main( int argc, char* argv[] ) {
    if( argc != 2 ) {
        write(1,"Wrong num. Args\n", 17);
        return 1;
    } else {
        write(1, argv[1], strlen(argv[1]));
        write(1, "\n", 1);
        return 0;
    }
}
static int strlen(char *pch)
{
    int len = 0;
    while( *pch != 0 ) {
        len++;
        pch++;
    }
    return len;
}

int main( int argc, char* argv[] ) {
    if( argc != 2 ) {
        write(1,"Wrong num. Args\n", 17);
        return 1;
    } else {
        write(1, argv[1], strlen(argv[1]));
        write(1, "\n", 1);
        return 0;
    }
}
#include <unistd.h>

static int strlen(char *pch)
{
    int len = 0;
    while( *pch != 0 ) {
        len++;
        pch++;
    }
    return len;
}

int main( int argc, char* argv[] ) {
    if( argc != 2 ) {
        write(1,"Wrong num. Args\n", 17);
        return 1;
    } else {
        write(1, argv[1], strlen(argv[1]));
        write(1, "\n", 1);
        return 0;
    }
}
#include <unistd.h>

static int strlen(char *pch)
{
    int len = 0;
    while( *pch != 0 ) {
        len++;
        pch++;
    }
    return len;
}

int main( int argc, char* argv[] ) {
    if( argc != 2 ) {
        write(1,"Wrong num. Args\n", 17);
        return 1;
    } else {
        write(1, argv[1], strlen(argv[1]));
        write(1, "\n", 1);
        return 0;
    }
}