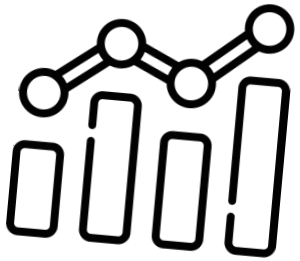
CS 2200
Lab 5

Announcements

- **Homework 4** will be released tonight and is due **Sept 26th @ 11:59 PM**
- **Project 1** demo slots are open!
  - Remember, this is a *multiplier* of your project grade.
  - **Mandatory**
- **Project 2** is out and will be due **Oct 11th @ 11:59 PM**

Today's Topic:
Performance Metrics, C, and GDB



Performance Metrics

*Formula sheet on page **5-13** of the textbook
(or scan the QR code below)

Exercise: Performance Metrics

Tony Stark's program is inefficient and has 2200 instructions of which

- 30% are ADD instructions (5 CPI)
- 50% are LW instructions (4 CPI)
- 20% are SW instructions (4 CPI)

Each clock cycle is 8 nanoseconds. He found a way to improve ADD to 3 CPI.

1. What is the **execution time** before and after?
2. What is the **speedup** and **improvement** in execution time?

Exercise: Amdahl's Law

A processor spends **30%** of its time on ADD instructions. An engineer proposes to improve the add instruction by **5** times. What is the **speedup** achieved because of the modification?

$$T_{\text{after}} = T_{\text{unaffected}} + \frac{T_{\text{affected}}}{x}$$

Rate of improvement!

Static vs. Dynamic

- **Static Frequency:** Number of times the instruction appears in compiled code.
- **Dynamic Frequency:** Number of times the instruction executes.

1. Program A has many ADDI instructions in **loops**, will the static or dynamic frequency of the ADDI instruction be larger?
2. Program B has many ADDI instructions in **conditionals**, will the static or dynamic frequency of the ADDI instruction be larger?

Metric Types

- We’re now going to focus on making our computer better
  - How do we measure this?
- Two main categories of metrics to measure the performance of our processor
  - **Spatial (space) metrics:** how much memory a program uses
  - **Temporal (time) metrics:** how much time a program takes to execute
- Spatial and Temporal metrics are not necessarily linearly related

Metrics Definitions

Execution Time: Total time taken to execute a program

$$\text{Execution time} = (\sum \text{CPI}_j) * \text{clock cycle time, where } 1 \leq j \leq n$$

- CPI is clock cycles per instruction
  - eg: CPI of **add** is 3, CPI of **lw** is 4

$$\text{Execution time} = n * \text{CPI}_{\text{Avg}} * \text{clock cycle time}$$

- CPI_{avg} is average clock cycles per instruction, i.e. $\frac{\Sigma\, CPI_j}{n}$
  - **n** is the number of instructions in the program

Speedup and Improvement in Execution Time

- Speedup

$$Speedup_{AoverB} = \frac{\text{Execution Time on Processor B}}{\text{Execution Time on Processor A}}$$

$$Speedup_{improved} = \frac{\text{Execution Time Before Improvement}}{\text{Execution Time After Improvement}}$$

- Improvement in execution time

$$\text{Improvement in execution time} = \frac{\text{old execution time} - \text{new execution time}}{\text{old execution time}}$$

Latency and Throughput

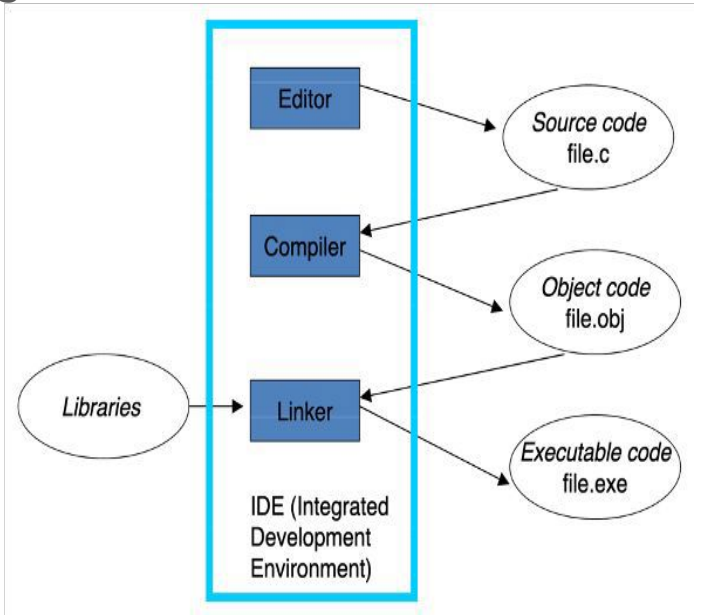
- **Latency**
  - How many units of time it takes an instruction to execute from start to finish
  - Improved by increasing clock speed or reducing CPI
- **Throughput**
  - Number of instructions executed by the processor per unit of time
  - Improved through pipelining

C and GDB

*Here is a good external resource for learning C.

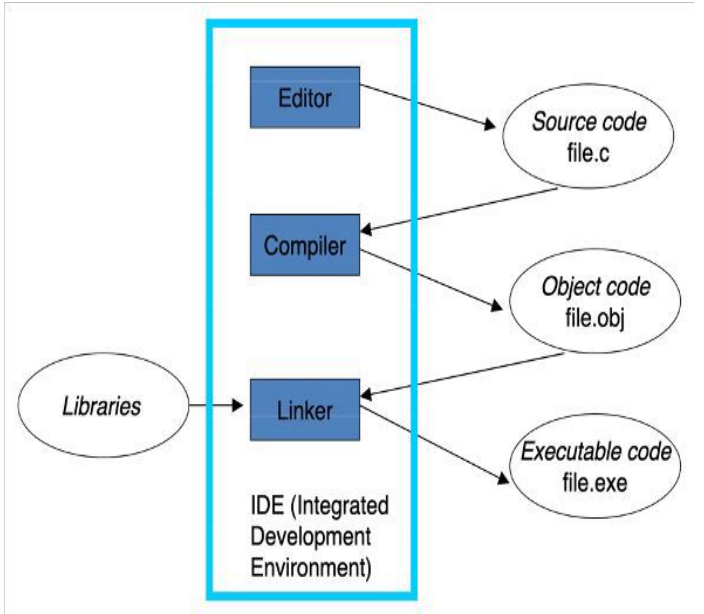
Writing and Compiling

- Compiler **analyzes the source file and converts it into machine language** which creates an executable file.
- Most commonly used compiler is **gcc!**
- **What does the following command do?**
  - `gcc -o program program.c`
- What happens when we need to link executables together?
  - We'll use a **Makefile** to make life easier.



Writing and Compiling

- `gcc -o program program.c`
  - Writes build into named output file
- **Makefile**
  - Builds & compiles program
  - Checks dependencies
  - Determines if recompilation is necessary
  - Other commands such as clean, debug, etc.



Questions: Preprocessor Directives

- What does `#include` do?
- What does `#define` do?
- What is an example use of `#define`?

Questions: Datatypes and Structs

- C is **strictly typed** and **compiled**. What does this mean?
- What built-in C function can we use to find the number of bytes of a data type?
- What is a struct?
- How do we access members of a struct?
- What does an arrow operator (`->`) do when accessing a member of a struct?

Accessing members of a struct

```
typedef struct s
{
      char c;
      int x;
      int* pointer;
} s_t;
```

```
s_t my_struct;              // Declare struct
my_struct.c = 'a';          // Set c to “a”
my_struct.x = 5;            // Set x to 5
int temp = 20;              // Create a new int
s_t.pointer = &temp;        // Set pointer to address of int
```

Pointers

- Refers to a location in memory
- Pointers are defined using the asterisk `*`.
  - `int *a_ptr;` is a pointer to an `int`
- Also, use an asterisk to dereference a pointer (find the value at that address).
  - `int b = *b_ptr;`
- Use `&` to get the address of a variable.
  - `int *c_ptr = &c;`
- Careful with pointer arithmetic! When you add a value to a pointer, you will increase the address by `value * sizeof(datatype)`
  - `uint32_t *t_ptr = 0x300;`
  - `t_ptr + 1 -> 0x304`

Exercise: Pointers

```
// sizeof(short) == 2
// memory address of x is 0x5000
short x = 5;
short* p = &x;
p = p + 6;
```

What will `p` equal after this code runs?

Structs and pointers

- Sometimes, we hold a *pointer* to a struct in a variable.
  - Thus, we hold the memory address of the **start** of that struct in a variable.
- How do we access the struct?
  - `s_t* my_struct = (s_t*) malloc(sizeof(s_t));`
  - `malloc` returns a void pointer, **always** need to cast!
  - `my_struct->c = 'a';`
  - `(*mystruct).c = 'b';`

```
typedef struct s
{
    char c;
    int x;
    int* pointer;
} s_t;
```

How do we print in C?

- `printf("Hello World\n");`
  - `\n` is newline character
- What if we want to print `x` where `int x = 8;`?
  - `printf("x is: %d\n", x);`

Exercise: Functions

- What is the difference between **pass by value** and **pass by reference?**
- Which of these is C?
- How would we write a function to swap in C?

Memory Allocation

- `malloc` is used to dynamically allocate a single block of memory with the required size from the heap.
  - `mp = (cast_type*) malloc(byte_size);`
- `malloc()`: allocates a fixed memory size from the heap and returns a pointer to it.
- `calloc()`: same as malloc but zeros out the memory.
- `realloc()`: takes in some malloced data, allocates some new memory on the heap and copies the data across.
- `free()`: frees some memory pointed to by a pointer.

GDB

- Requires the compiler flag `-g` to make use of GDB effectively.
- Here are a few important commands
  - `r` runs your program until it hits a breakpoint, segfaults, or terminates.
  - `c` continues execution until next bp, segfault, or termination
  - `n` goes to next instruction (step-over function calls)
  - `s` steps forward (step-into function calls)
  - `b file.c:n` sets a breakpoint at line `n` in `file.c`
  - `p var` prints the value stored in variable `var`
  - `x/nfu <memory location>` examines a large area of memory.
  - `bt` prints a backtrace which allows you to discover the context in which an error was thrown.

Exercise: GDB

*You will be asked to **upload your debugged factorial.c file** for the lab attendance quiz! Use Docker terminal!*

Let's debug the given file, **factorial.c (download Lab 5 folder from Canvas)**

```
#include <stdio.h>
int main()
{
     int i, num, j;
     printf("Enter the number: ");
     scanf("%d", &num);
     for (i = 1; i < num; i++) {
          j = j * i;
     }
     printf("The factorial of %d is %d\n", num, j);
}
```

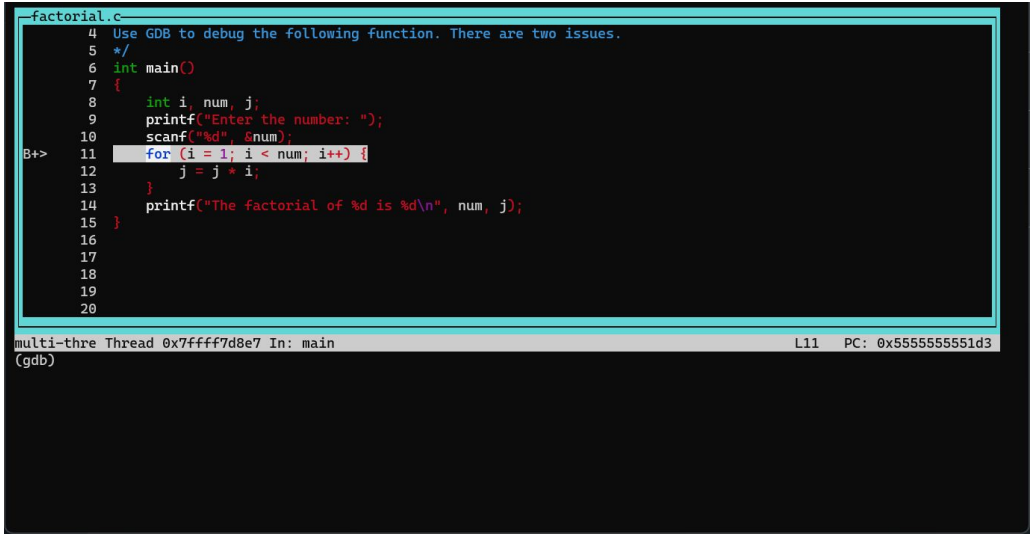
```
gcc -g factorial.c
gdb a.out (or lldb a.out)
```

What can we do to debug?

- `b 10`
- `r`
- `p i`
- `p j`
- `p num`

GDB has a TUI 😮

- **T**ext **U**ser **I**nterface (TUI)
- Here's a good external source that goes over the basics.
- Let's say you've set a breakpoint using `b` and ran the program to that point. Now, type `tui enable` and we get:

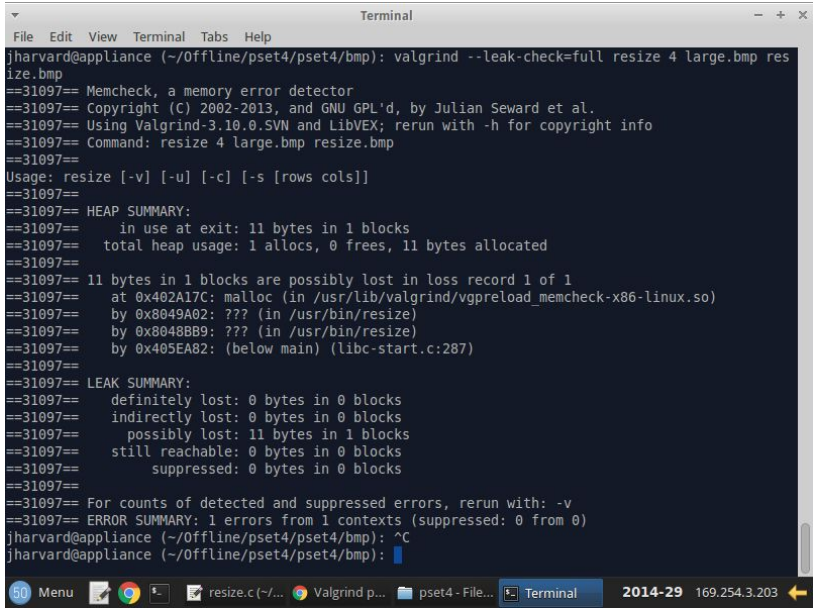


Valgrind: Don’t Leak Memory!

- Valgrind is tool to check C/C++ programs for memory leaks
- Also has support to detect race conditions and deadlocks with Helgrind
- Pay attention to the leak summary at the end!
- If compiled with “-g”, provides individual line numbers where memory leaks may be occurring

Install Valgrind in your docker container:

```
apt install valgrind
```



Exercise: Valgrind

Let's debug the given file with the following header, **leaktest.c**

```
#ifndef LEAKTEST_H
#define LEAKTEST_H

void simpleLeak();
void complexLeak();

int freeingIsMyJob();
int* freeingIsYourJob();

typedef struct Node {
    int data;
    struct Node* next;
} Node;

Node *createNode(int value);
Node *addNode(Node *head, Node *node);
void *deleteNode(Node *head, int value);

#endif
```

```
make leaktest
./leaktest
```

```
make memtest
```

View valgrind-out.txt to find the memory leaks!

16578

Number of the day

Thanks for attending!