CS4043

Date

# Advanced Programming

* Doxygen tool code to doc.

Performance 2-3 weeks

- Why backward compatibility? There are still hardwares present that are still being used. Link old hardwares with new softwares. ~~[illegible]~~. Link new HW with old SW.

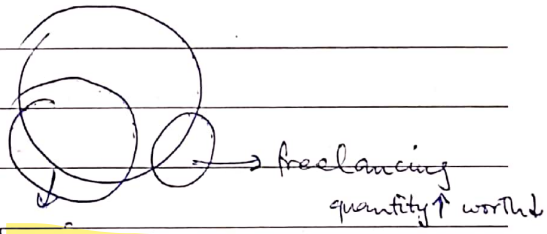
- C/C++ is the standard, new languages follow it

compiled lang. C → C++ → Java → C# ← object oriented approach

JS, perl, android, python, ruby ... ] interpreted

legacy = familiar with providers old SW/lang./HW

- TIOBE index analyses software and company pays you money accordingly




- languages, technologies, API, toolchains

- business process operations
  - quantity↓ worth↑
  - show pakistan export bureau certificate to strengthen profile

## Source code:

- source code → (obj files) → compile → Link → ... → load

comments, .h, .docx, precompilation → #include #define #pragma
conditional compiling: #ifdef #ifndef #elifdef

if you want to generate documentation from your code auto. then write comments according to standards.

doxygene tool

// / * / ** ] ISO standard

- #if 0 ... #endif ] won't be visible to documentation tool cuz 0, if 1 then visible

decision before compiling:
- * new code, old HW = backward compatible
- * old code, new HW = forward compatible
- may have to write 2 types of code for back + forward compatibility

## Compilation:

- proprietary SW should give flavors for different processors. Maybe company compiled SW on some old processor and you downloaded the compiled code, it will not run on your new processor.

- ~~Date~~ Libraries are not in compilation stage
- If code is more↑, compilation time ↑
- Hard coding increases compiling time

Example:

```
for ( i )
    for ( j )
        a[j][i] = cos ( some complex trignometric op.)
```

hard code → cout << "a[" . . .

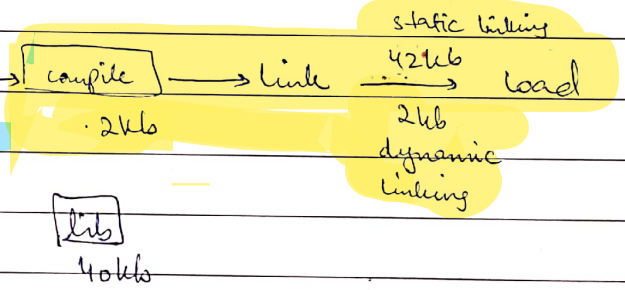
- use algo minimum code, use tools, parameter finetuning = optimize compilation
  - -O
- arithmatic op. takes longest time to compile, Use better parameters
  - -m arch  - -m tune ← to change architecture

Solutions

- ① you compile once, other people access it from a cache
  ② distributed compilation (distcc tool), compile code in parts on different places

Linking:

- static ~~library~~ linking → library is mixed with link file
  - takes more storage
- dynamic linking
  - library is loaded with obj file on runtime
  - takes less storage




- if library is built static then it can't be used as static library or static linking

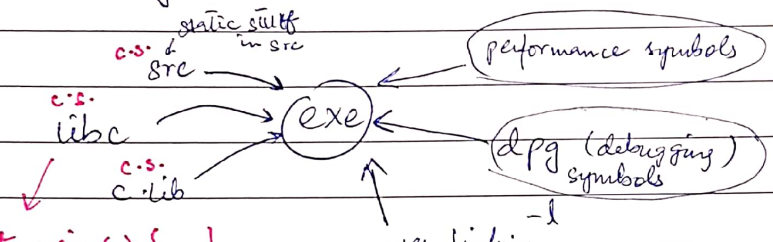
DIY

* make a static and a dynamic library + compile with your code

Date 24th February, 2022. Advanced Programming

# Build Process

- gcc ← for C, popular in open source, bloated file size
  lib code + your code is compiled
- clang ← for c-based support, open source




int main() { }
nothing else in code,
still libc will be imported

- Microsoft makes bloated code
- dpg symbols + c.s. code symbols used for analysis
- performance symbols check performance of code.

start → code → time → end

start byte

| ELF64 |
| --- |
| compiled code .text |

end byte

executable + linkable format = ELF

- gcc -c hello.c
  ↓ compilation only
- gcc -s hello.c
  ↓ assembling of code

| | hello.c | 49 |
| --- | --- | --- |
| hello.s | disassemble -d | 388B |
| hello.o | compile | 1.2KB |
| hello.t | executable | 16K |

after linking

objdump -d hello.o
disassembly of section .text
00 ... 00 <main>
0: 55 (hexa values)
1: 48 89 e5
push %rbp

- If we compile c.s. we will get compiled code in .text after linking we get ELF.

Date

- objdump -d hello.f
  disassembly of section .fini
  <main> section is pushed forward. (1119)
  in .text there is <start> which occurs before <main>.
  in it libraries talks to OS and makes it ready.

- executable is compiler's work.
  OS
  STD

## Compilations:

ELF 64 → ELF Header

ELF file types : • Relocatable — compiled file can be relocated after linking → compiled

compiler adds many things to our code to make it exec.

• Executable → executable

library

double relocation,
① by compiler level
② on loading by OS level

<main> is at 1119, this shows maybe OS is very small and is at 0 – 1118 address, (not possible)
so it means it has gone thru 2 relocations.
Originally it'd be at much higher address.

C → obj.o (1.2KB) → ELF64 obj (16KB)

S → obj.o (560B) → hello.oassembly (16KB)

- ELF file has ELF header (-h), program header (-l) and section header (-S)

- hello.c — relocatable file
  hello.f — exec file

to check headers → readelf -l hello.o
program in file

Date readelf -h hello.c

readelf -h hello.c ... hello.f

| | Compile Relocatable | Executable |
|---|---|---|
| ELF header | ✓ | ✓ |
| prog. header | 0 | 56 → 13 prog. headers |
| sec. header | 64 ↓ divided b/w 12 sections | 64 ↓ div. b/w 27 sections and libraries too |

ELF header :

Magic :

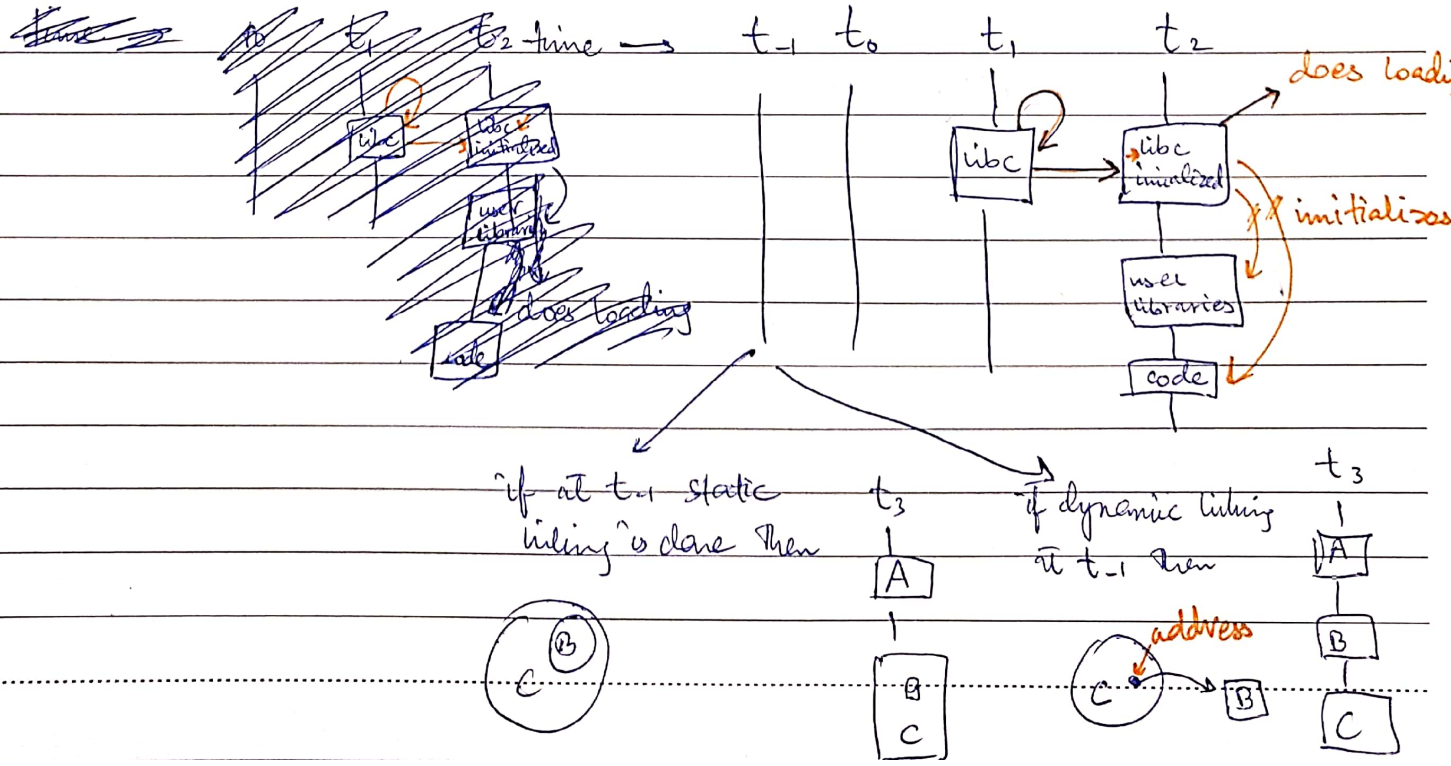
- OS decides what section to make executable = some level of security
- viruses intend to trick ELF, take code to assembly and make changes that virus wants.
- Not easy to bypass OS.
- Compiler puts stuff in sec. head after linking stuff is placed in prog. header

March 03, 2022. Advanced Programming

Build Processes: Linking : an .exe is produced

ldd a.out → to see libraries dependencies if any.

libc → runs + Tells how to run stuff

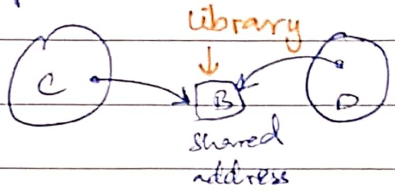


Date

→ so, not preferred

— dll in windows

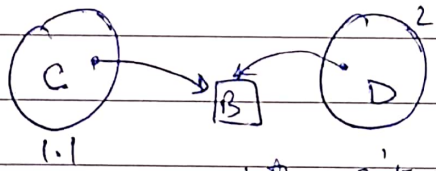
- In static, if multiple programs use B, B will be made copies. If dynamic :




- dll is vulnerable
- reverse engineering can be done on ELF.
- linux favors dynamic linking
- versioning in dynamic linking

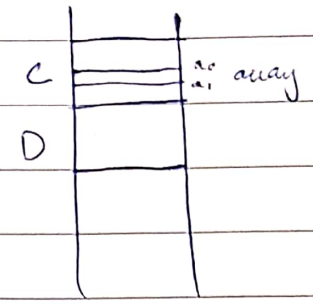
1.1 - r1
1.2 - r2
1.3 - r3
1.4 - r4
major version, minor version, patches

2.0 difficulty in difference in major versions.

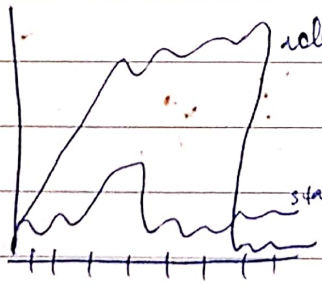



both can't use B one different versions

RAM




gives segmentation fault when trying to access illegal memory location.
outside of $a_0 + a_1$ will also give segmentation fault




GPL systems
↓
linux gives guarantees that these will work

Date

getIS.o

- library code → compile → ELF relocatable code → linking → ELF executable
  user code → compile → ELF relocatable code → linking → ELF executable

```
int get() {
    return 15;
}
```

```
int main() {
    return getIS();
}
```

gives error ; getIS don't know

Put it in redundancy

solution

① #include header file
containing getIS()

```
int main() {
    return getIS();
}
```

② int getIS();

```
int main() {
    return getIS();
}
```

- If want to generate static lib : at compilation
  : gcc -c getIS.c → getIS.o
  : gcc -c getIS.c -o libgetIS.a → libgetIS.o
  - getIS.o will be in current directory
  - libgetIS.a can be sent + sold as package
- when want to make general file use lib filename.a

/usr/lib → where libraries are stored

A
B
C
D
E
F
↳ pool of libs

/usr/lib64 →

at compilation

" gcc getIS.c -l getIS → will check in default location → then will check with lib getIS.a → then linking done

Date

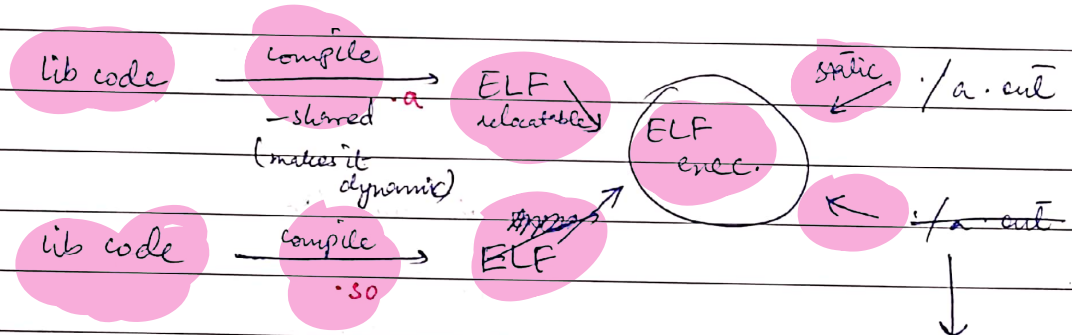
- gcc main.c -L <alternate path> -lget15 → first check default loc. Then check alternate path

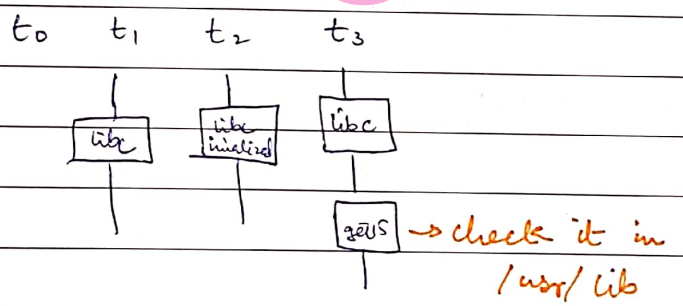
gcc main.c -lm
↓
math library argument

- -l , -lm options are used for both static + dynamic OS doesn't care if S or D. It will be compiled as it is. but -shared keyword makes it dynamic while compiling library

file get15.c
gcc -shared get15.c -o libget15.so

- .so for dynamic
- .a for static




LD_LIBRARY_PATH: $LD_LIBRARY_PATH:
(appending it, adding new in existing path)
then do ./a.out

when we don't specify location, it creates linking problems in installation

- export LD_LIBRARY_path = /alt/etc
(life : till Terminal is open)

LD_library_path = /alt/etc ./a.out
(life : will run only once)

if you want it gets runs everytime you open terminal : put it in bashrc

Date

- gcc main.c -l1 -l2 -l3 — can't compile cuz all contain same foo()

-l1 -l2 -l3 — no compilation error
foo() -faa() -fee()

-l1 -l2 -l3 — when l1 is first linked, error cuz it doesn't know fee().
foo(){ fee() } fee(){ faa() } faa(){ foo() }

t0 t1 t2 t3
like libc libc libc
l1 l1
l2

-l1 -l2 -l3 → -l2 -l1 -l3
foo(){ fee() } fee() faa() → fee() foo(){ fee() } faa()

solution: shuffle around
but possible only in static linking
don't use dynamic linking if circle in compilation of libraries

- readelf -S a.out
.dynamic DYNAMIC
↓
it contains shared lib libgetIS.so
shared lib libc.so.6
cuz Sir specified -lgetIS -lc while compiling

- use symbolic links when dealing w/ multiple versions

- want to install 2 libs in one OS? -libmona1.1 → libmona.so.1.1
like -libmona1.3 → libmona.so.1.3
do -wl soname:libmona.so.1 (points to)
but can't do -wl soname:libmona.so.4 (too much gap) errors.

Date March 8th, 2022.

# Debugging for Memory Problems using Valgrind

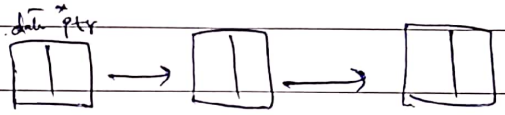
- uninitialized variables
- memory leaks
- read / write to unallocated memory (maybe segmentation fault)
- deleting or freeing dynamically created memory twice

valgrind : catches all the memory problems

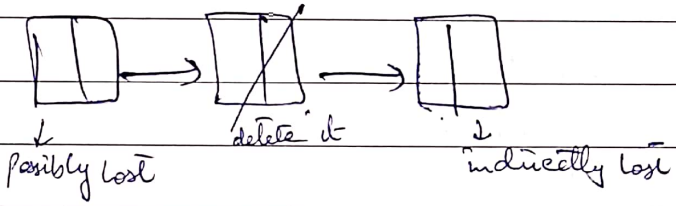
definitely lost → `int *x = new int (10);`

~~delete x~~ ← not done

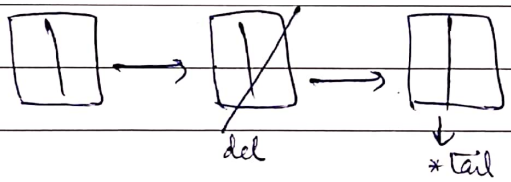
indirectly lost →




possibly lost →




still reachable →




*tail reachable only by *tail otherwise lost due to 2nd one's deletion

- valgrind : Tells memory behaviour over execution period of code
  massif

- memory problems in code

usage:

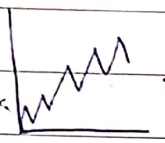
valgrind --tool=massif --time-unit=B ./a.out

Date

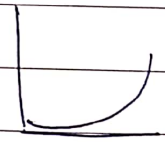
Solution

- improve or change algo
- improve code

ubuntu kills such programs which take up too much memory



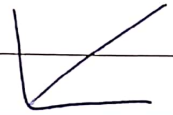
→ spikes mean you're not doing deallocation



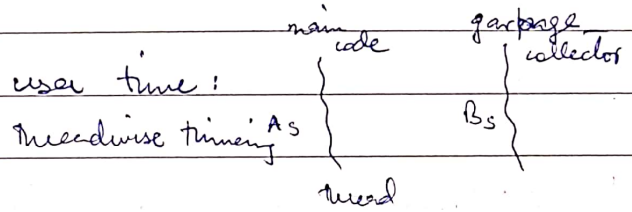
exponential - bad

$O(n^2)$

memory ↑ time



$O(n)$




$A_s + B_s$ (time) = user time

sys time =

real time = overall pic of time

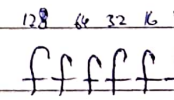
## Performance Measurement

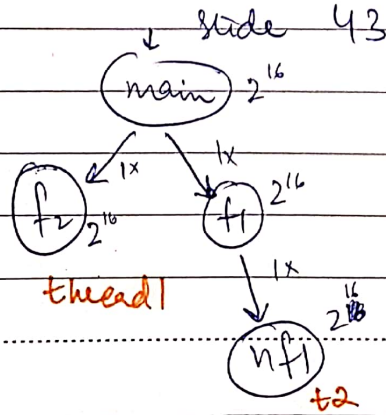
### Code Profilers

```
: code
struct = func (start)   ← OS measures time
: code
struct2 = func (end)
: code
```

elapsed = end - start

symbols



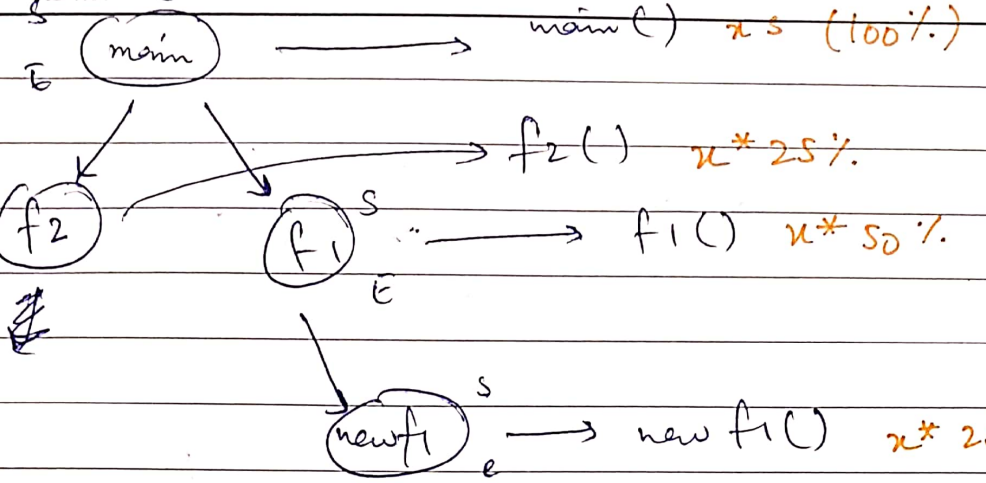

real time — x s

user time — x s

sys time — ~0 s (w2 real + user time same

Date ↓

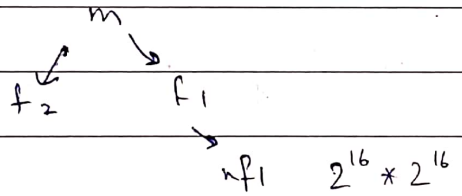
time inside functions :




accumulative time

O(n x4) → when for ( ) ; m
nf1() ; f2 f1 $2^{16} + 2^{16}$
nf1

$O(n^2 + 2n)$ → when for ( ) {
nf1();
}




- gprof : charts + diagrams

stock count : gives price of your code.

Date March 10th, 2022.

# Automated Builds

① profit
② sharing

Liscensing → MIT
GPL-2
GPL-3
EULA

configure.in → autoconf → configure.ac

makefile.in → automake → makefile.ac

makefile.ac + configure.ac → makefile → make install

(close) binaries/source (open)
↳ installers
big files
small packages
↳ manual
↳ build tools ./configure, make, make install

process

Auto tools
↳ automake
↳ autoconf

[ ] ← parameters

target = pre-requisite
step1
step2
step3
:

target → here pre-requisites are checked
step1
step2

Reverse Engineering

make (all)
make (target)
make all (target) → builds 1st target

make doc → Documentation creation

make install → copy
step1: cp (copy)
step2: mv (move)

make clear → clean from current directory

make distclean → super user required to clean from the system → need superuser "sudo"

make.amam → set target + steps + mention everything in order of software. But these (mentioned above) common names are used mostly

→ We don't know other until read documentation.

# Sandbox environment for testing

Date

private / personal repo : gitolite

- safe to run make files for testing in sandbox. If no errors then make it permanent in own system.

target 1 ↑ pre req ↑ — SHORTCUTS hai ye

all : aprogram → go to aprogram & run all steps

target 2 aprogram : foo.o bar.o

gcc -o aprogram foo.o bar.o

prereq can be .c file or .o file

target 3 foo.o : foo.c

pre req for t2

gcc -c foo.c

target 4 bar.o : bar.c

pre req for t2

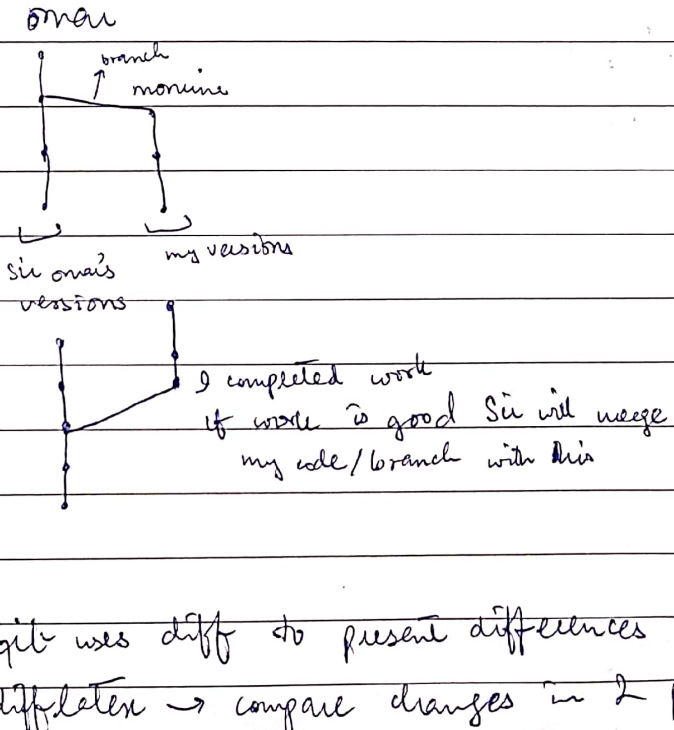
gcc -c bar.c

step 1
step 2
⋮
step n
halt

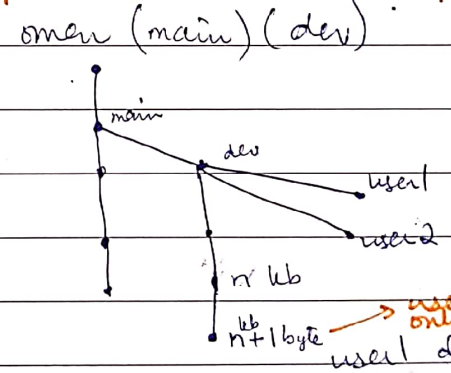
makefile checks these that they are ~~done~~ already done so it doesn't run them. It directly goes to step n

## Version Control

- keeps record of changes from time to time




in same folder




user added a file only change is stored

user1 did pull req & is working on his PC the doesn't know about user2 & can't see other code. when everybody uploads code, then changes will be visible

- Git uses diff to present differences done in code
- diffutils → compare changes in 2 pdfs.

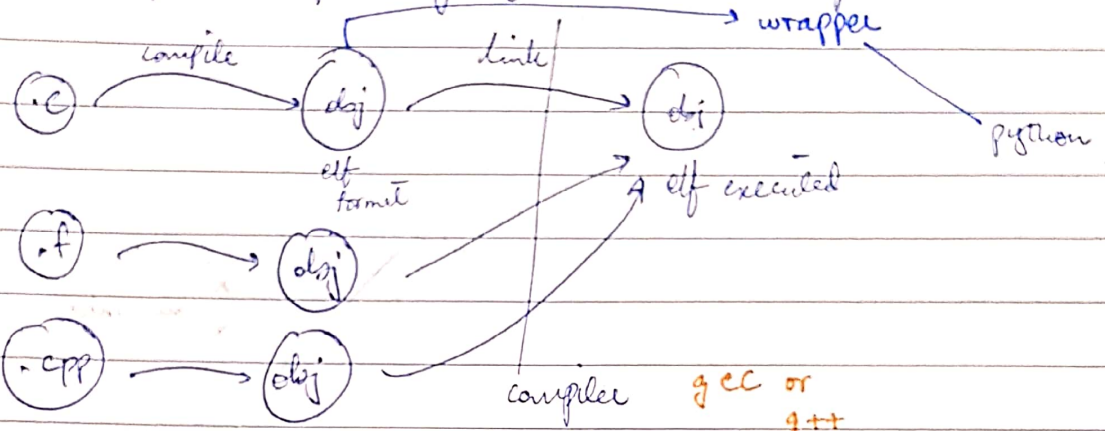
Date March 15th, 2022.

Slide214 to
- preproc
- compilation args elf 64
- linking -s/d
- automated builds - makefile (src code given, make makefile)
- dbg debugging - memory problems
- ~~code profilers~~

## Cross Compilation:

- cross - platform
- cross - architecture → works on phones + PCs
- cross - lang → python + C++ etc

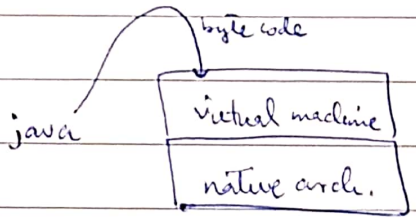
- support for multiple languages → cross compilation.

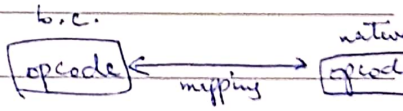



- should work w/ multiple langs.

native : HW relate
base architectu

python
java
interpreted lang.



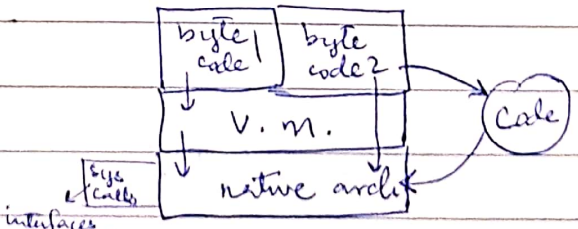




- wrapper libraries in slides

java running on V.M.
opcode of byte code is being mapped on native

system calls:
- input/output
- exec, fork

- you have to import libs to talk to native ↓ wrapper libs




- can have code which bypasses V.M. This makes code fast.

~~fast~~ use shared libs for fast code

DIY
wrapper
cross compil
input as
shared lib

Date

Slide 50 [example]

int ret = fputs (s, fp) → will be compiled as dll (dynamic linking)

myCFunctions . myfputs ( b"Hello", b "write . txt")

↓ to let C lang. know it's a whole string + not just 'H'

- example will run on native, will pass v.m.

- gcc (static .o, static2 .o) all → single binary machine code

- .py → .pyc = compiled code
  ↕ can't mix
  .c → .a (elf) = compiled code

use lib to communicate .pyc with .a .

- z[ array ] = x[ n ] * a

  *x
  *z
  int n
  int alpha

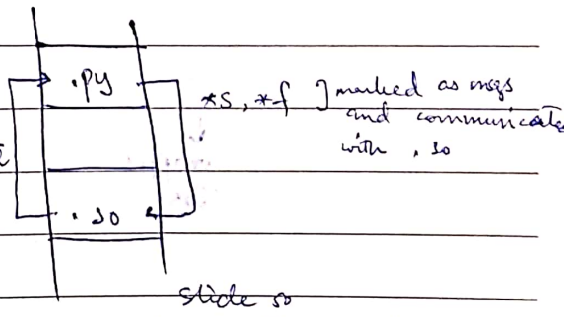
z = mul (x, z, n, alpha)

matlab .c → DIY: how can we do stuff in lib & return in some other thing?

for loop {

z[i] = x[i] * a

- pass addresses to library in python DIY




Marshalling: data stored in internal obj and transferred (.py to .so) & sent back same way (.so to