Gate 96)

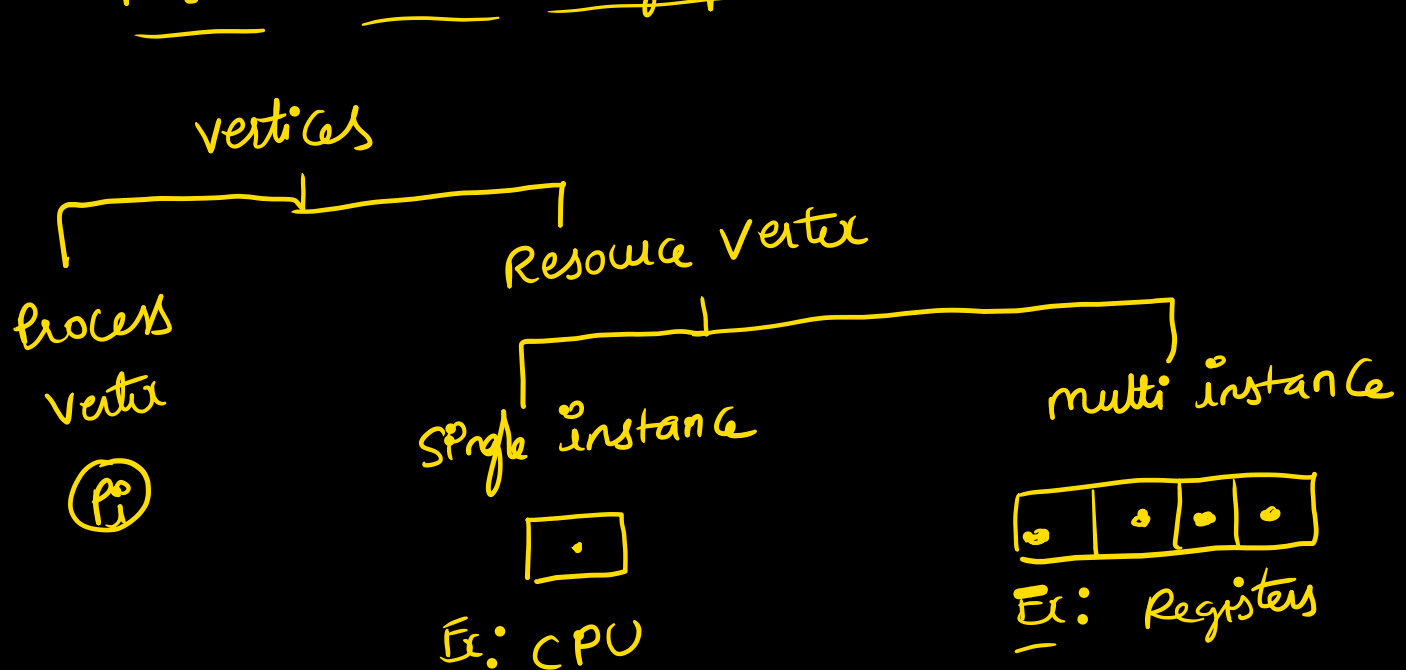
	Current alloc			max need			future need		
	R_0	R_1	R_2	R_0	R_1	R_2	R_0	R_1	R_2
P_0	1	0	2	4	1	2	3	1	0
P_1	0	3	1	1	5	1	1	2	0
P_2	1	0	2	1	2	3	0	2	1

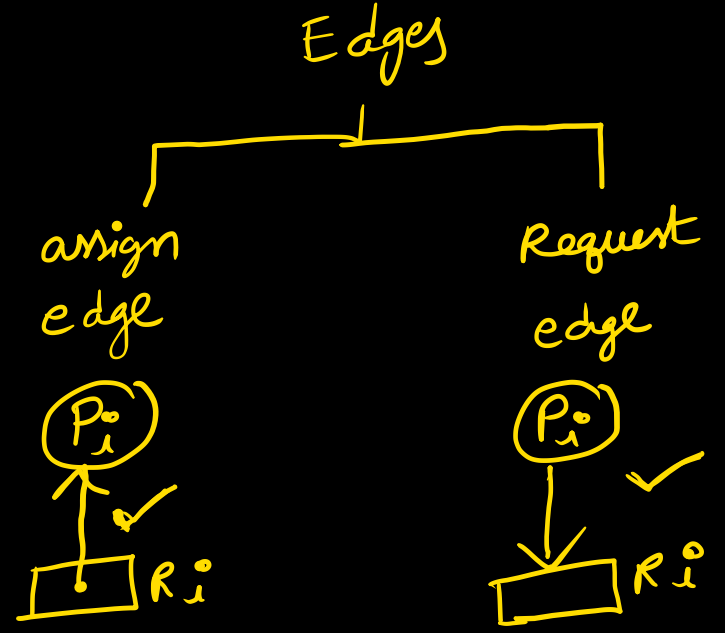
avail = (2 2 0)

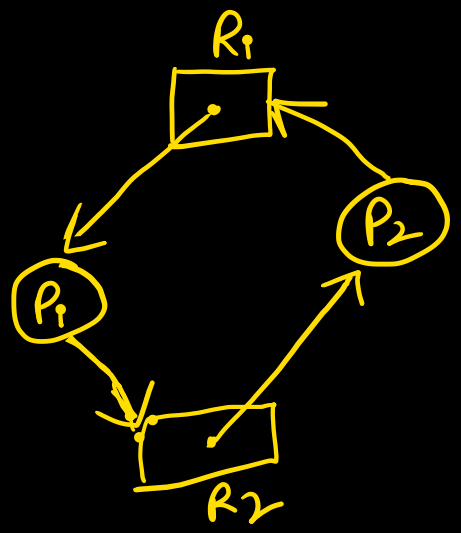
Req = P_0 (0 1 0)

- State tables X
  Resource allocation graph ⟶ Safe or not ?

Resource allocation graph:







Single instance resource type Graph, if there is a cycle then will be deadlock surely. (necessary and sufficient Condition.

Proof :→ Take table

alloc		
	R_1	R_2
P_1	1	0
P_2	0	1

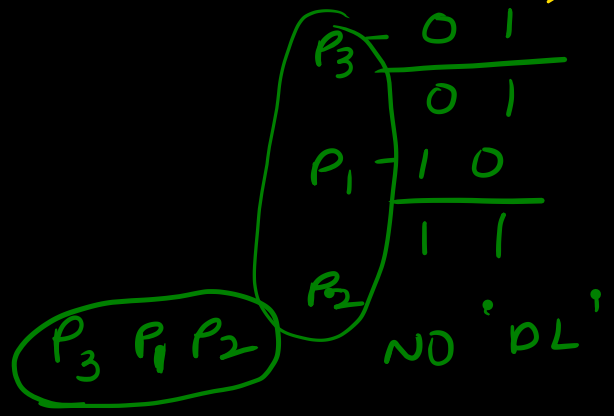
future need	
R_1	R_2
0	1
1	0

avail = (0 0)

MI RT → Cycle is just a necessary condition. Not sufficient. So cycle may or may not be deadlock.

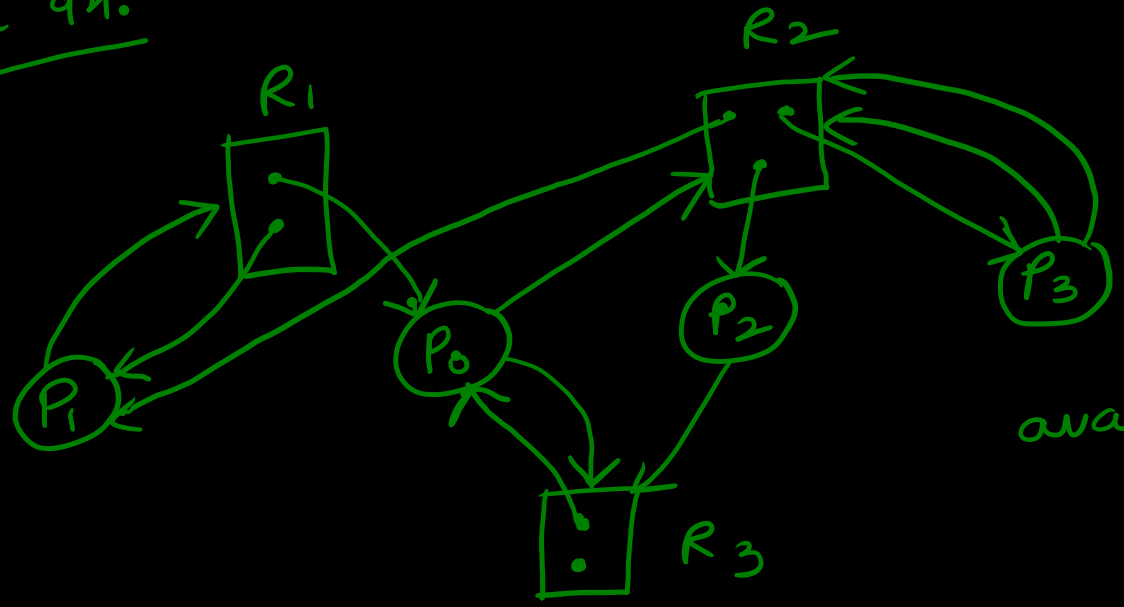
	alloc R_1	alloc R_2	request R_1	request R_2
P_1	1	0	0	1
P_2	0	1	1	0
P_3	0	1	0	0

avail = (0, 0)



NO 'DL'

Gate 94:



alloc

	R_1	R_2	R_3
P_0	1	0	1
P_1	1	1	0
P_2	0	1	0
P_3	0	1	0

req

R_1	R_2	R_3
0	1	1
1	0	0
0	0	1
0	2	0

avail: 0 0 1

P_2	0 1 0	
	0 1 1	
a P_0	1 0 1	
	1 1 2	
P_1	1 1 0	
	2 2 2	
P_3	0 1 0	
	2 3 2	

P_2 P_0 P_1 P_3 → "Safe"

Deadlock handling:

- DL ignore → ignored ✓
- DL Prevention → 4 condition ✓
- DL avoidance → Bankers algo ✓
- DL Detection & Recovery ✓

{ 4 Conditions / 4 Handling

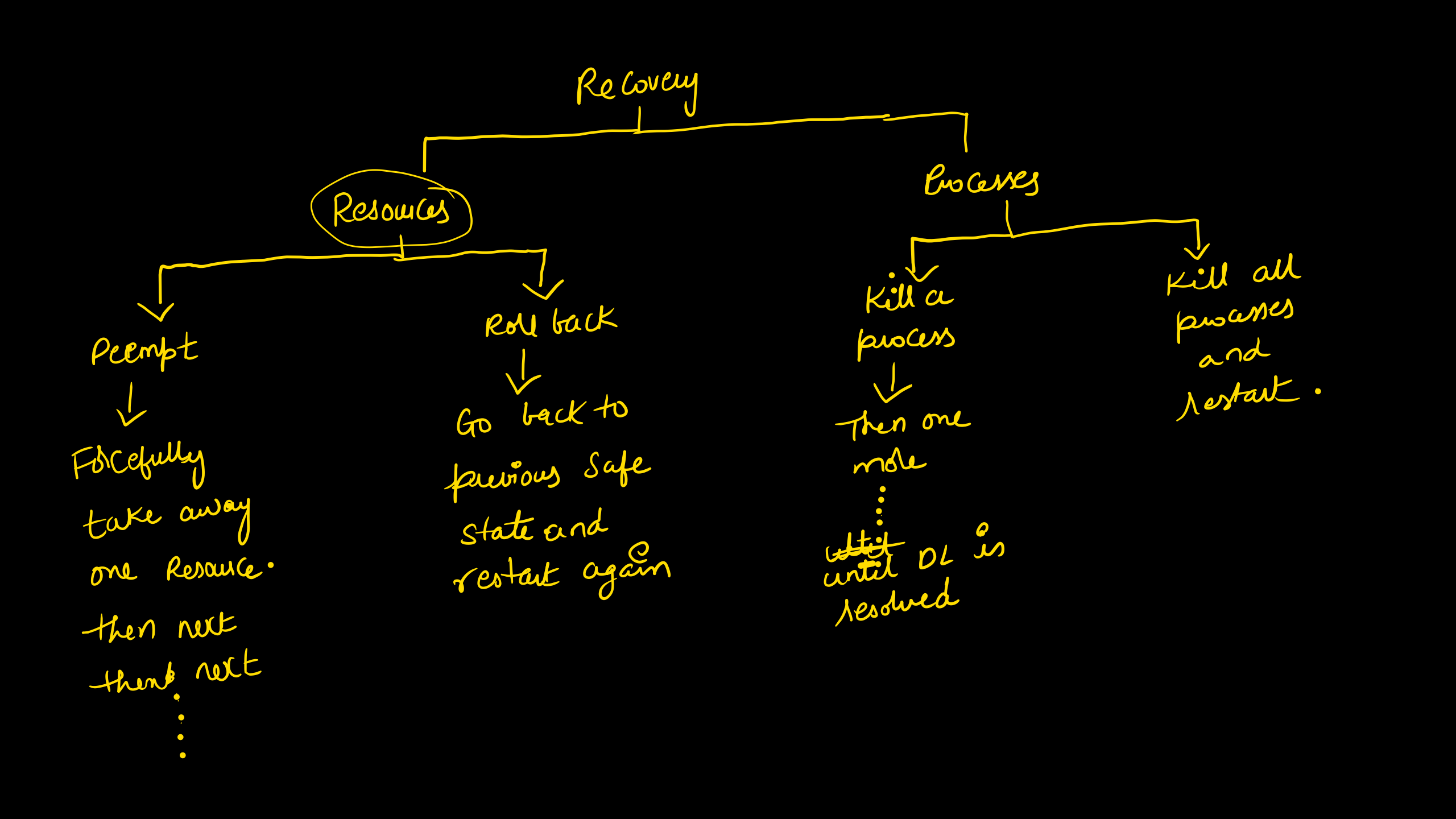
Deadlock detection and Recovery:

Detection:

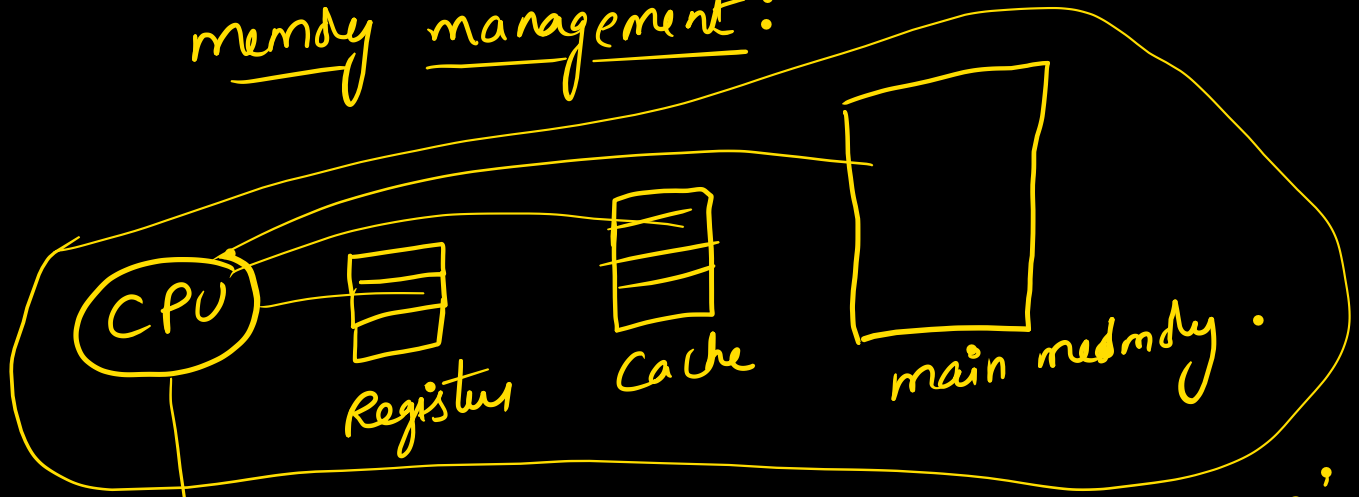
- Single Instance resource type
  - Cycle: (Nec & Suff) ✓
- multi instance resource type
  - Bankers algo:

what happens if there is DL?

- Recovery


Recovery
Resources
Processes
Preempt
Forcefully take away one Resource. then next then next
Roll back
Go back to previous safe state and restart again
Kill a process
Then one more
until DL is resolved
Kill all processes and restart.

memory management:



$\eta \uparrow$ if more processes are in 'mm'.

$\Rightarrow \eta \uparrow$ if mm $\uparrow$ → it is costly.

with (

$\eta \uparrow \rightarrow$ 100% $\rightarrow$ mm $\rightarrow \infty$

Ex:

mm - 4 MB, Process size is 4 MB.

1-process

assume process needs 80% I/O time.

then CPU utilization is 20%

---

mm 16 MB 4 MB PS #proc = 4

p is prob of I/O of one process.

CPU util = $(1-p^4)$

$p = 80\%$ $\eta = 60\%$

p of I/O for 4 process (p^4)

$(1-p^4)$

---

mm - 32 MB PS 4 MB

process = 8

CPU = $(1-p^8)$ $p = 0.8$

$\eta = 83\%$.

---

mm 48 MB PS - 4 MB

process = 12 $p = 0.8$

CPU util = $(1-p^{12}) \cong 93\%$.

As degree of multiprogramming is increasing, the $\eta \uparrow$.

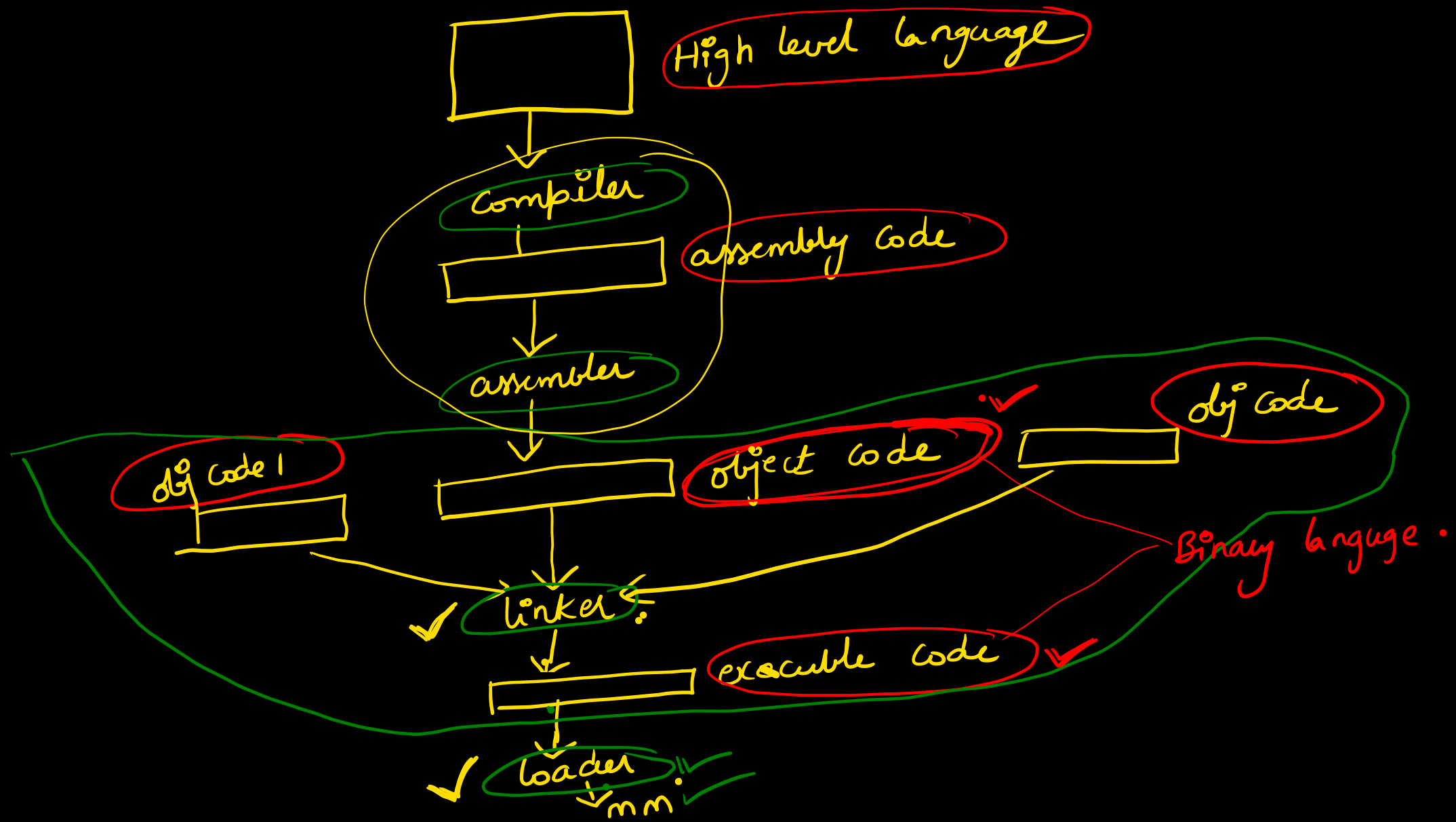
As $mm \to \infty$, $\eta \to 100\%$

↓

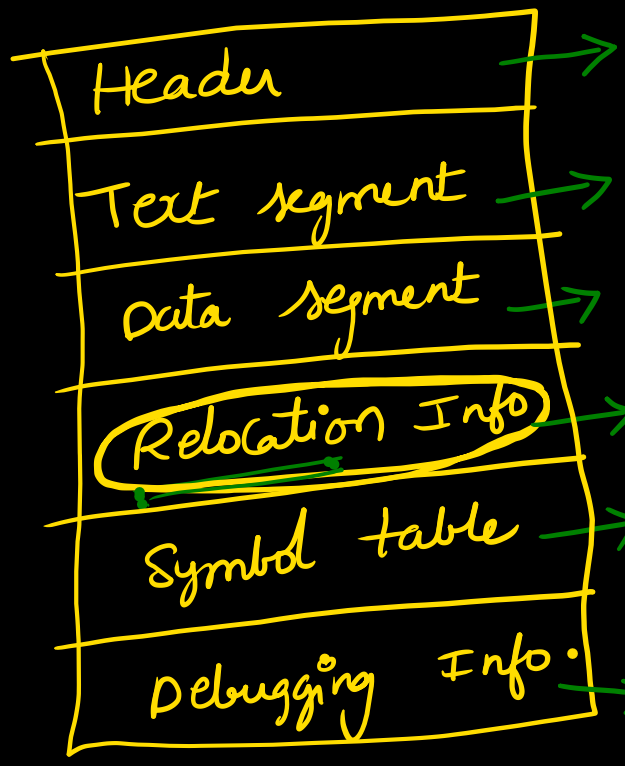
not practical

↓

The main goal of memory management is to efficiently use 'mm' to increase the utilization of CPU.


High level language
compiler
assembly code
assembler
obj code 1
object code
obj code
Binary language.
linker
executable code
loader
mm

Object code:



- Header → like index saying what all sections are present
- Text segment → code
- Data segment → Data item (static and global variables)
- Relocation Info → useful for loader. (for converting relative add to absolute addrss)
- Symbol table → useful for linker (contains all symbols in the program like variables, functions)
  - Need to be resolved
- Debugging Info: → not needed.

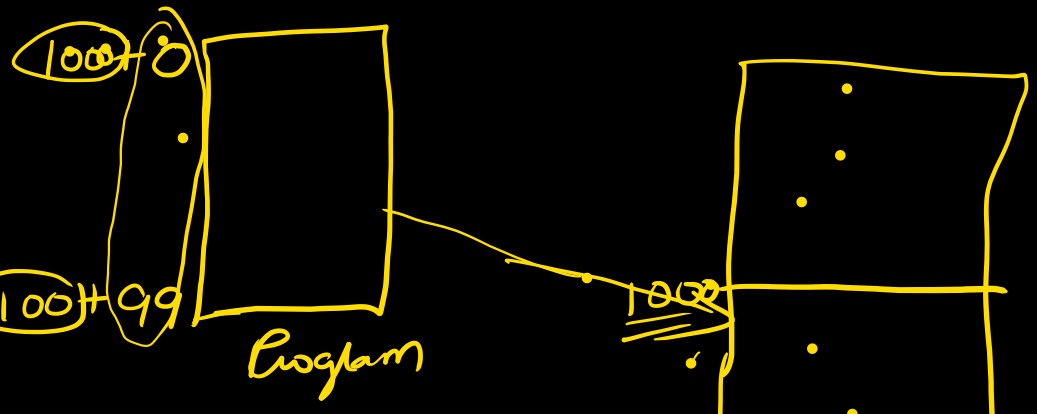
Loading (loader):

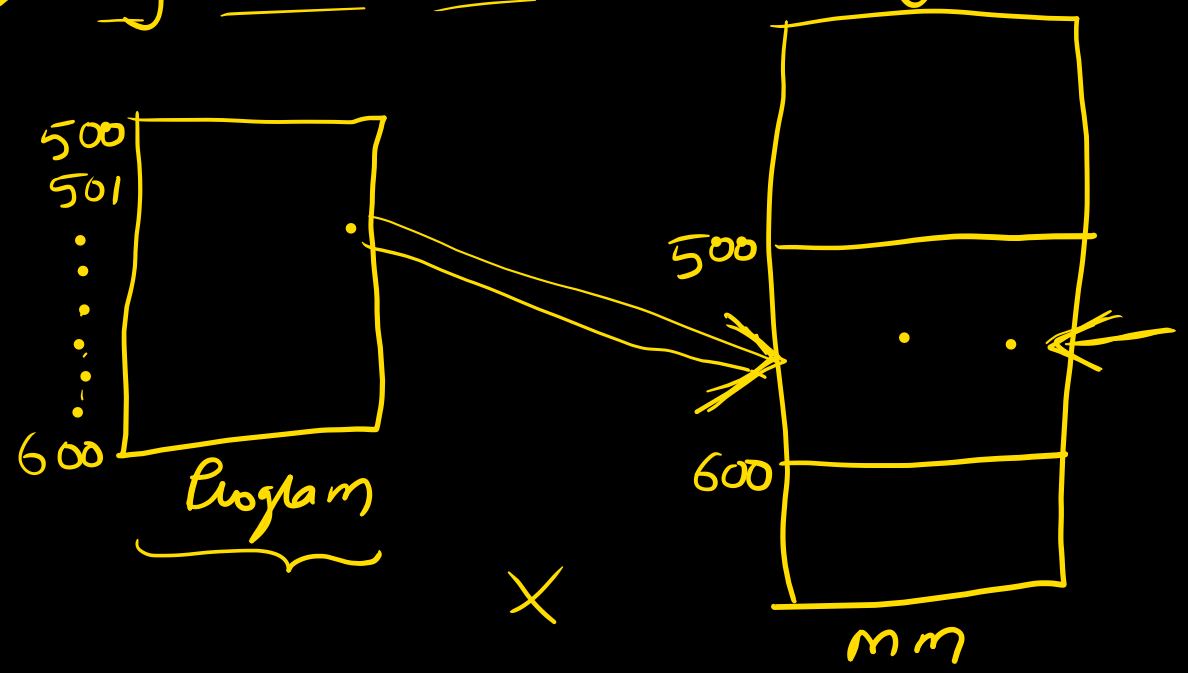
3 types

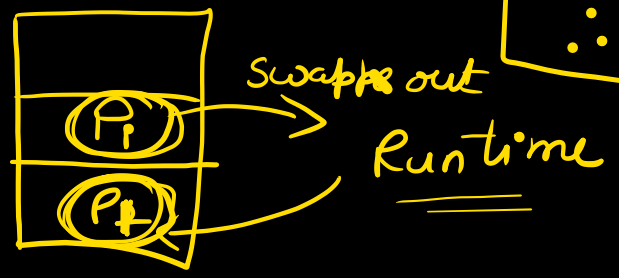
→ absolute loading

→ Relocatable loading

→ Dynamic run time loading.

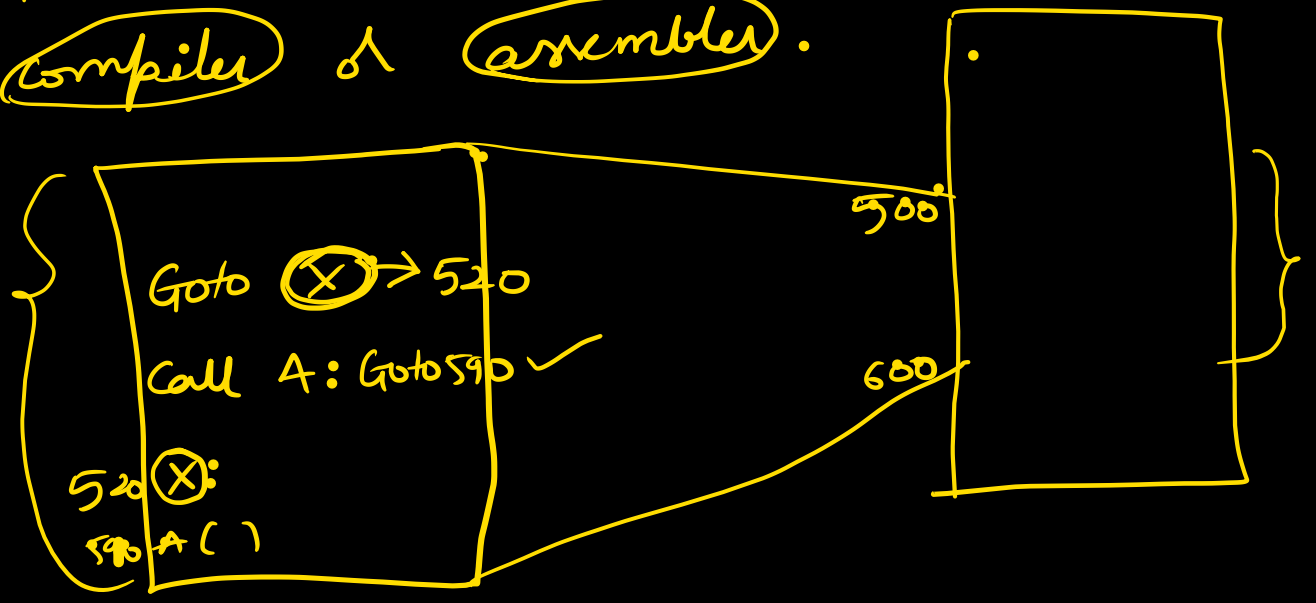






absolute loading:

It requires that a given program is always loaded into same location in main memory. Thus the program presented to loader has specific & absolute main memory addresses. The assignment of specific address values to memory references within the program can be done either by programmer & compiler & assembler.



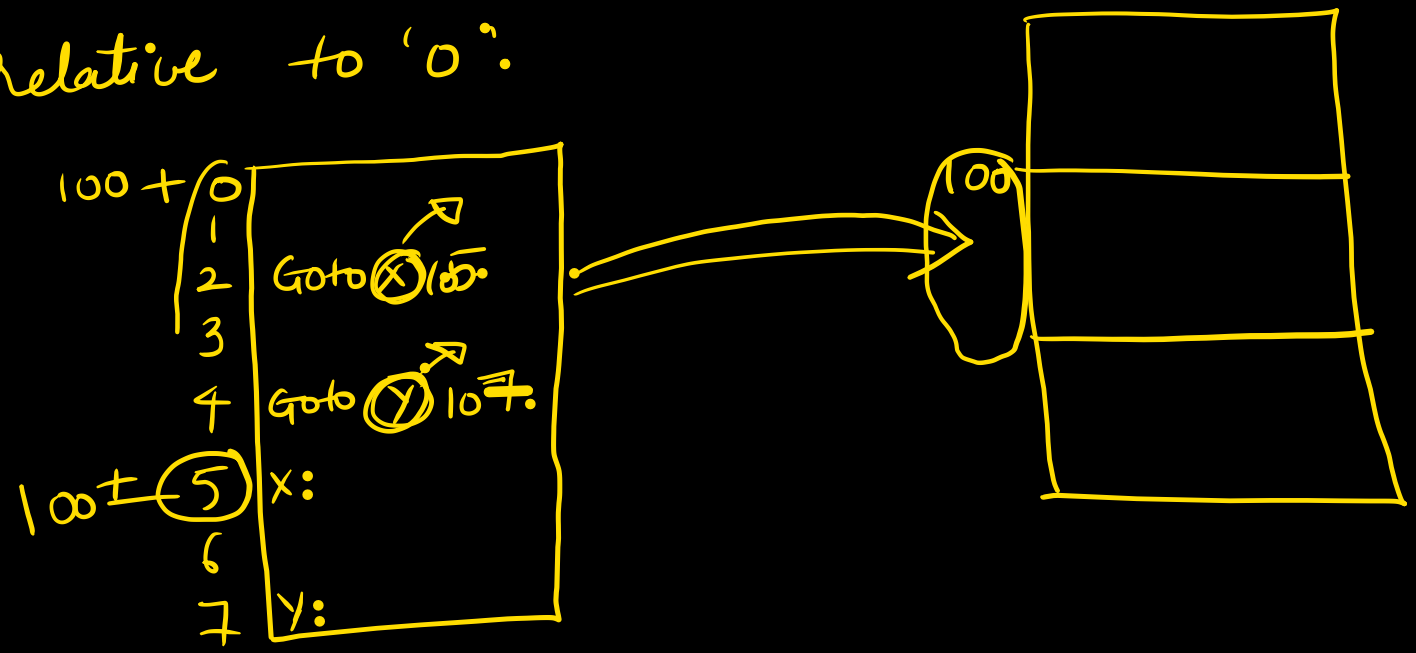
Disadv: we Cant know in advance, where a program can be loaded.

what if we want to modify the program.

Relocatable loading:

The decision of where to load a program is taken during load time. ∴ The program should be in a way that it can be loaded anywhere. The assembler or compiler produces not actual main memory addresses but addresses that are relative to '0'.

- relative → absolute
  - ↓
  - Relocation.



loader must simply add the beginning address to each memory reference as it loads the module into memory.

loader will use relocation bits or relocation info of object code file for relocatable loading

Dynamic Run time loading:

When we need to swap in and swap out a process multiple times into diff locations we cannot bare on relocatable loading. we need to calculate addresses dynamically at runtime. we use special processor H/W not S/W for this. we will see this later.

absolute loading → Compile time

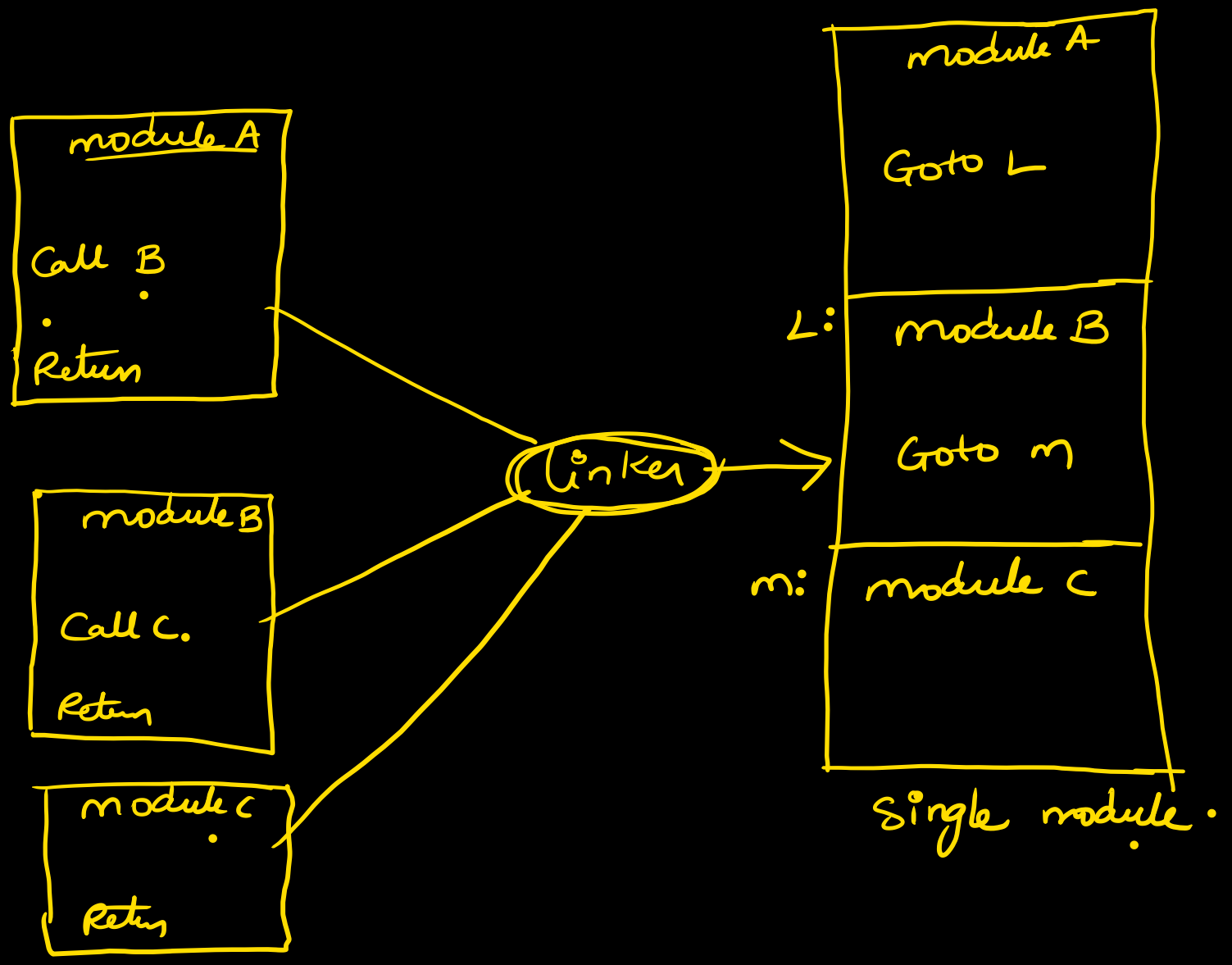
~~Relat~~ Relocatable loading → load time

Dynamic → Run time

} binding times

loader should always be in 'mm'.

Linking :


module A
Call B
Return
module B
Call C.
Return
module C
Return
Linker
module A
Goto L
L:
module B
Goto m
m:
module C
single module.

The function of a linker is to take as i/p a collection of object modules and produce a single module consisting of integrated set of program and data modules to be passed to loader.