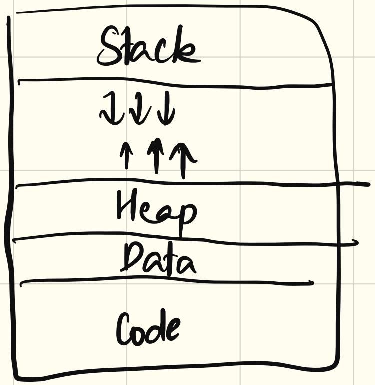
02 Apr 2025 - Operating Systems - II - Week 12

Reference : OSTEP (Three Easy Pieces)

Chapter 13.

→ Recap: Process Address Space

Same layout for all processes



} Virtual view

OS enables virtual view.

→ How do you use this structure and modify this structure

user APIs — by only OS → system calls

Sys calls

brk, mmap, munmap

opp.

allocate deallocate

"may" internally use ⑦⑦

Library API

malloc( )

calloc( )

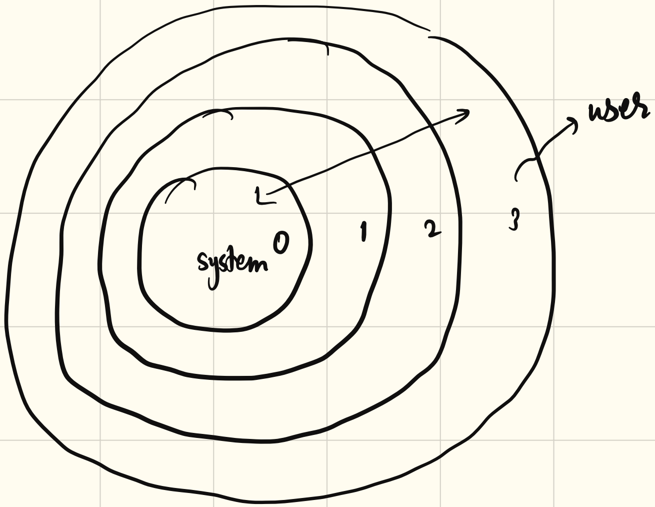
free( )

entry point to the OS

System calls are heavy

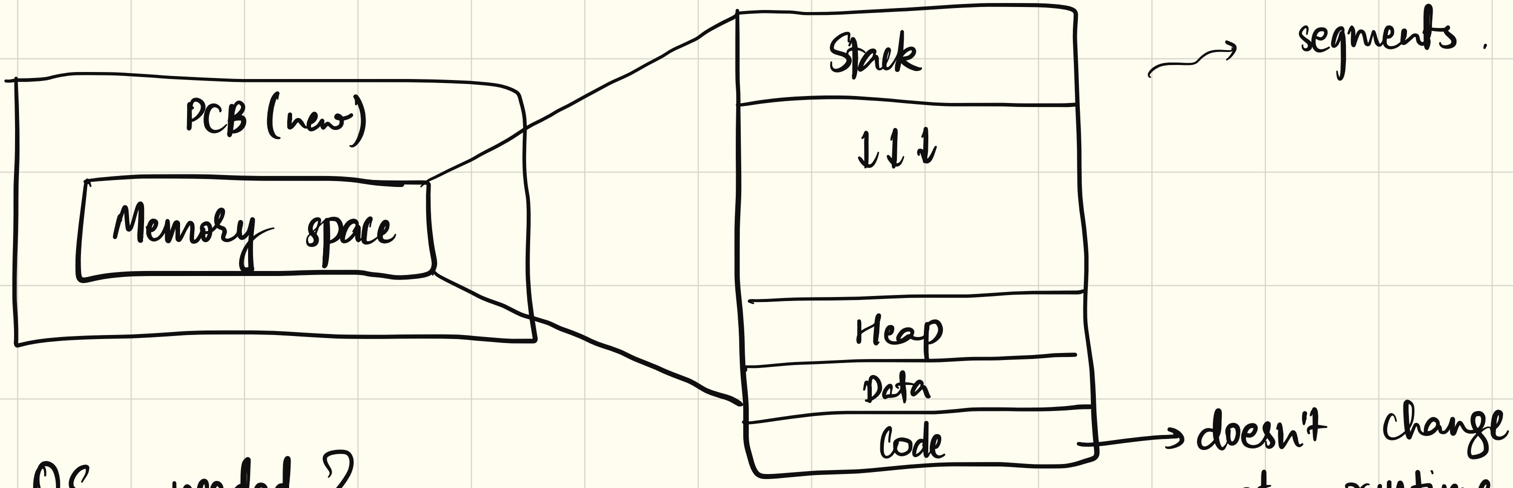
User mode } ~ context switch
Kernel mode

User space should not be allowed to do everything



Why 4?
guest machines,
hypervisors, etc.

Previledge modes
in x86



→ doesn't change at runtime

Why OS needed?

→ There should be corresponding physical address

→ Memory state may be changed by actual memory may not be allocated
available

Lazy allocation

On memory faults,
→ invoke handlers to allocate

Questions:

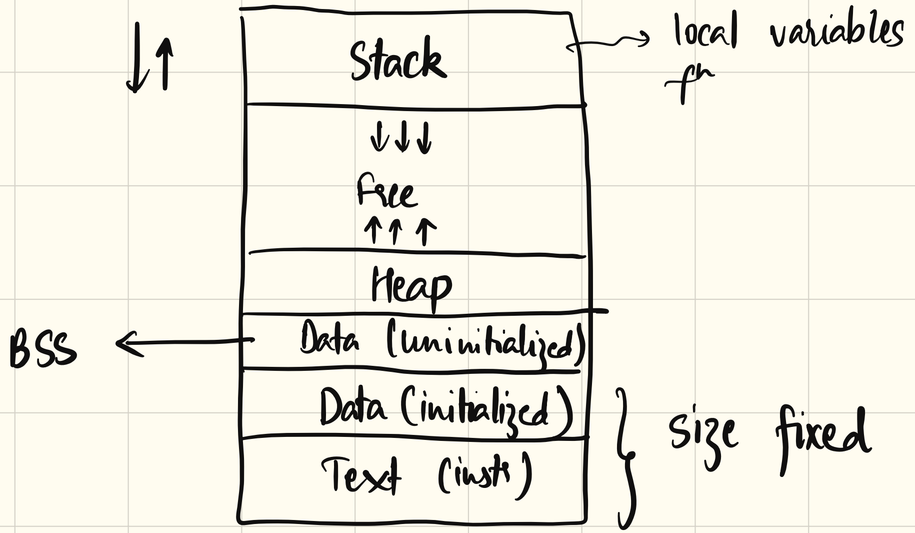
*

*

*

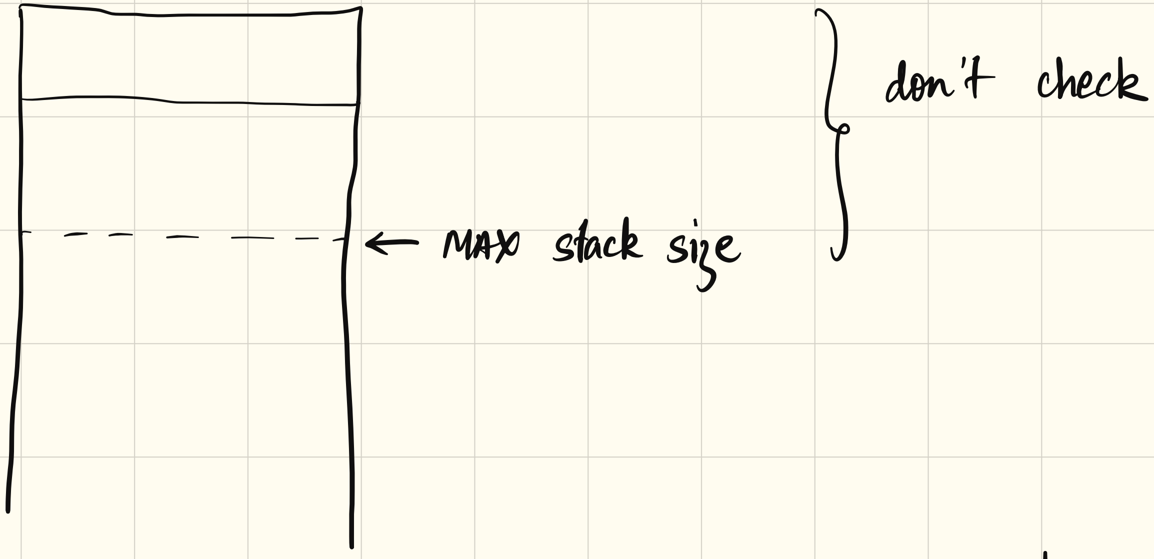
*

Dynamic Sizing the segments (Linux)



→ End of uninitialized data segment can be adjusted dynamically ??

→ Stack and heap overlap

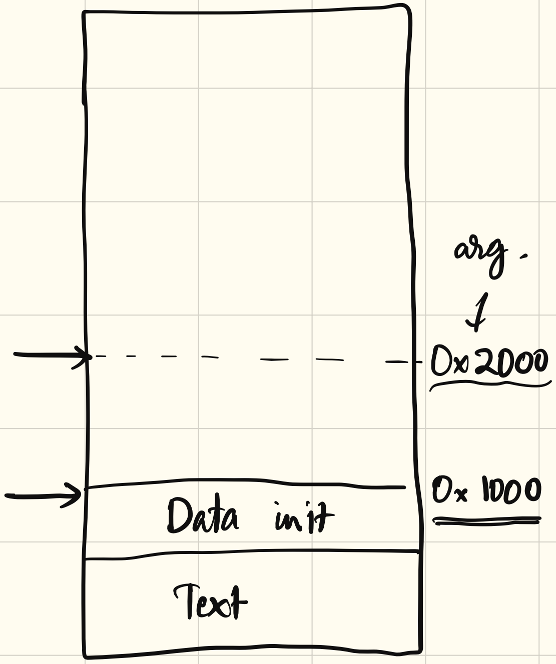


Heap allocation : mmap() → can be used in attacks
(discontinuous)

Sliding the BSS (↑ and ↓) : brk, sbrk

int brk (void* address)

→ Move the end of uninitialized data segment address (if possible)



→

cannot be used directly in attacks

void *sbrk(long size) ⇝ Returns the prev. pointer

→ 0x1000

returns 0x1000

sbrk(0) ⇝ returns the current location of BSS.

Finding the segment

fixed, compile time

etext end of

edata end of initialized data

end end of BSS (uninitialized)

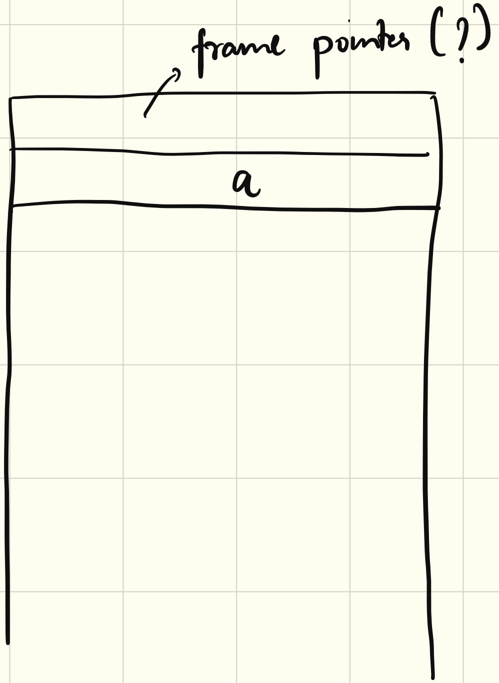
static binding inserted values by compiler

→ cannot be used in run time (static binding)

→ sbrk(0)

→ print

```
int main( ) {
    int a;
    printf ("%~", &a);
}
```



→ Linux provides proc / <pid> / maps.

sbrk , brk → read / write
heap , etc.

How to execute code dynamically ?

Stack is marked non-executable.

Discontiguous Allocation (mmap)

→ multipurpose , powerful
↓
contiguous,
share, etc.

→ brk , sbrk → only scale uninitialized data contiguously

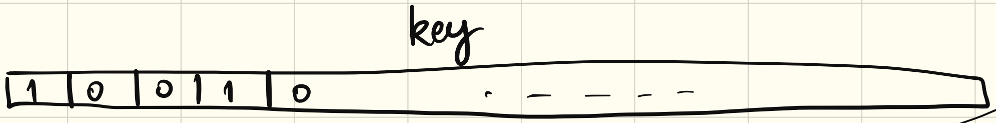
→ Allocate memory at a given memory location.

→ One attack : PGP application

↳ crypographic app

generate keys

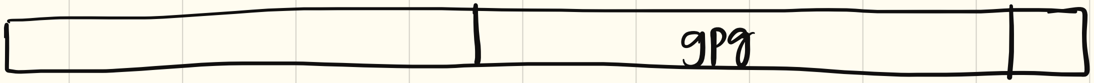
Shared libraries



```
for ( bit in key) {
    if (bit == 1) {
      → multiplication( )
        add ( )
    } else add( ) ;  }
```

Attacker : identified address of multiplication ( )
and add( )

GNU: PG



On frequent use ⤳ L3 cache is done.

x 86 ⤳ flush( )

flush ( ) ⟶ multiplication ( )

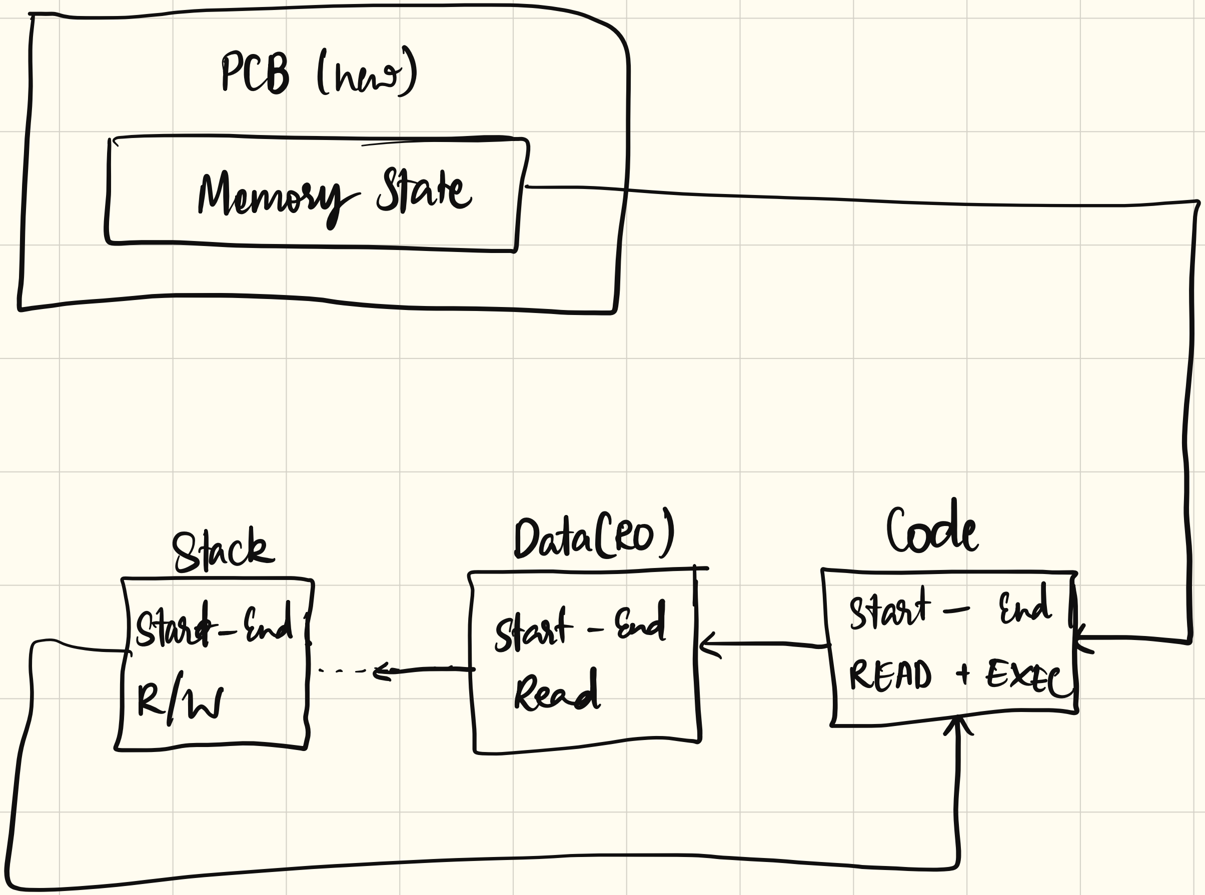
flush ( ) ⟶ add ( )

↘ multiply( )
add( )
} ⟶ If fast, other process has accessed key

→ usage (see slides)

→ manpage.

Memory State of PCB



→ Can merge areas if permission matches (lookup times improve)

→ proc:

```
cat /proc/<pid>/map
```

mem_end.c

```
extern char etext, edata, end;
```

we need to disable some security systems

OS randomizes address.

ASLR

address space layout randomization.

→ disable ASLR

```
cat proc/sys/kernel/randomize_va_space
```

randomize_va_space

- 0 off
- 1 randomize
- 2 fully randomize

```
echo 0 | sudo tee /proc_ _
```

```
gcc -no-pie mem_end.c -o end.
```

-no-pie: do not randomize

./end → prints VA physical address

some sys calls + sudo

ASLR can also be broken
run a script, find offset

brk. c

printf ⤳ does $\overbrace{\text{buffered}}^{\text{heap}}$ operations
uninitialized data

strace ./brk } → print all the system calls invoked by the program

write ( ) → vfs
monitor, file ...

write (arg 1 ,)
↳ which file descripter → file, device, ...
↘ ↓ related call

asm (.... assembly ...)

m fence   flush

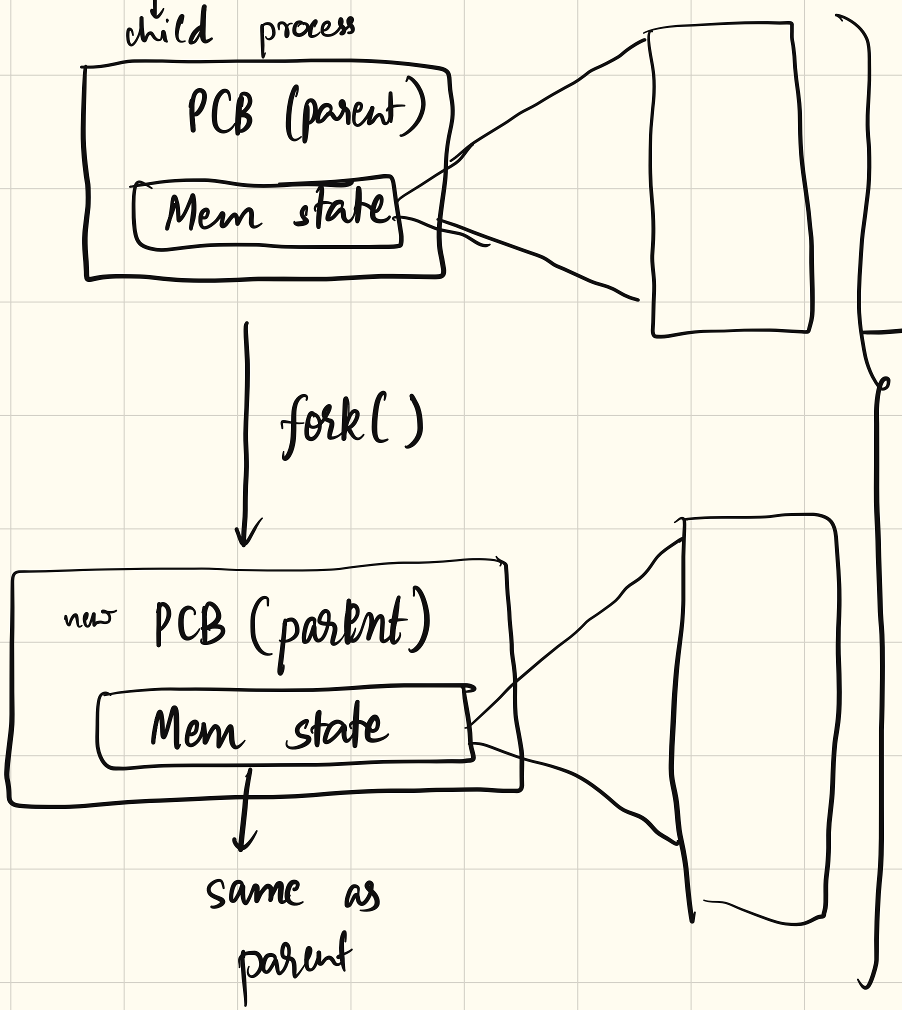
sbrk ⟶ will find out addr
and run
brk

Heap and
unitialized data ~ blurred diff.

WSL 2

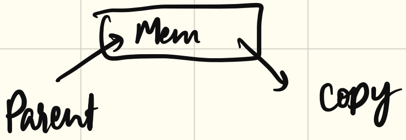
Tuesday → Tutorial

How fork and exec use these system calls?



fork() → copy memory state

all segments that are write, make them read-only



If parent/child someone tries to write,

CPU → error → OS handles using fn → copy and allow write → only relevant page

1GB state for parent

↳ we don't copy entire

→ COW copy-on-write

→ famous bug (9 years, linux)

dirty COW 2009 ~ 2017

---

exec() → execute another binary

→ does not create child process

→ shells → fork()

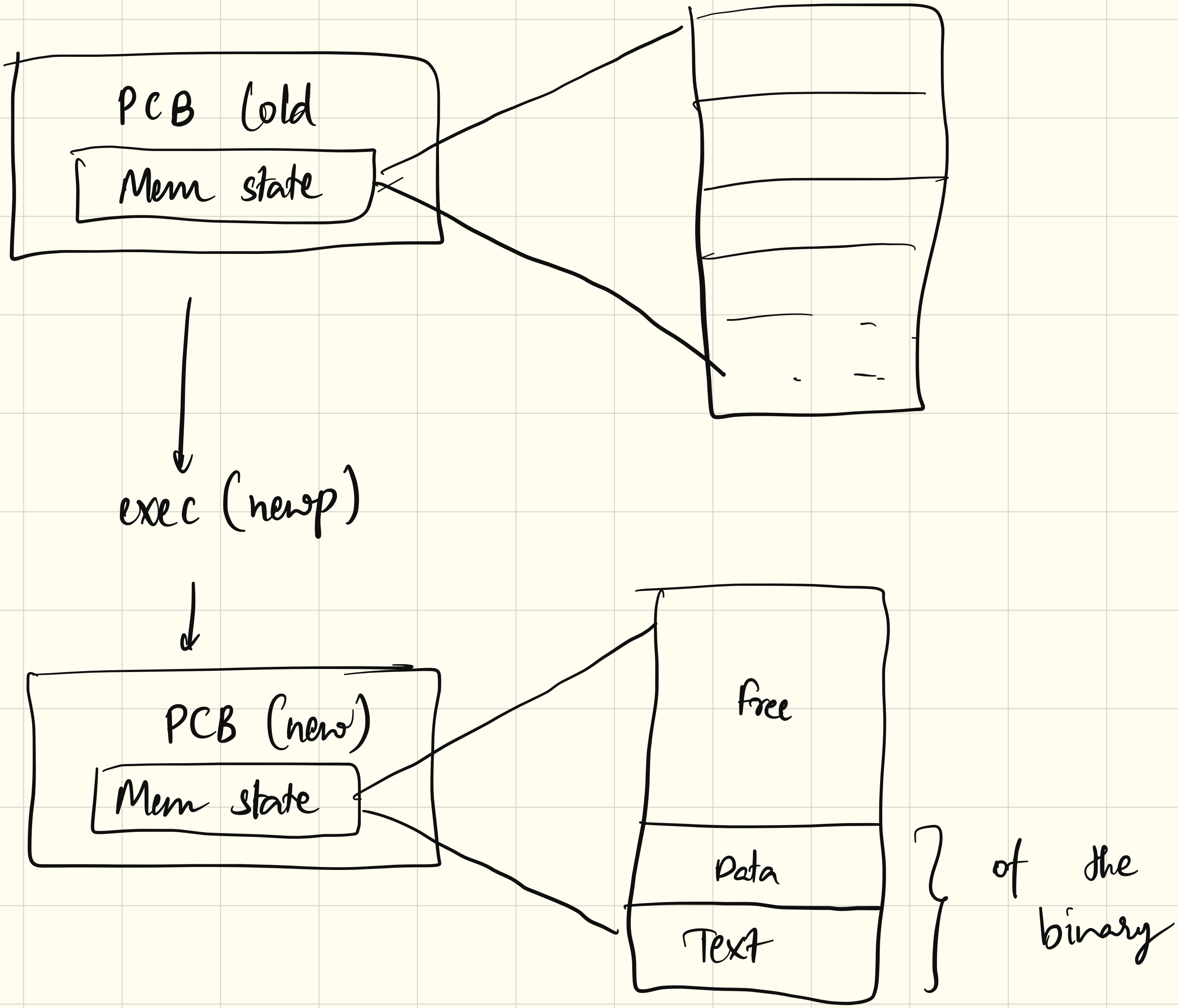
↓

exec()

→ pid does not change

destroy current state

and load new



This is why terminals ~> fork() and exec( )
cPid

```
              → I am child
if (cPid == 0) {
    exec("./~~~"); ──→ destroy
                       this layout
    exit();
                       and . . .
}
```

Address Translation

→ No control on physical memory ← process

First scheme

rbp → base frame pointer → from which location current f^n starts

→ x86 ISA

→ Role of compiler.

→ OS during binary load

03 Apr 2025

→ Process state after exec.

→ base register

PCB

User exec state

PC = 0

SP = 8 KB

T_Base = 20 KB

CPU

PC = 0

SP = 8 KB

Base = 20KB

Ins Fetch (vaddr = 10) → insFetch (paddr = 20KB + 10)

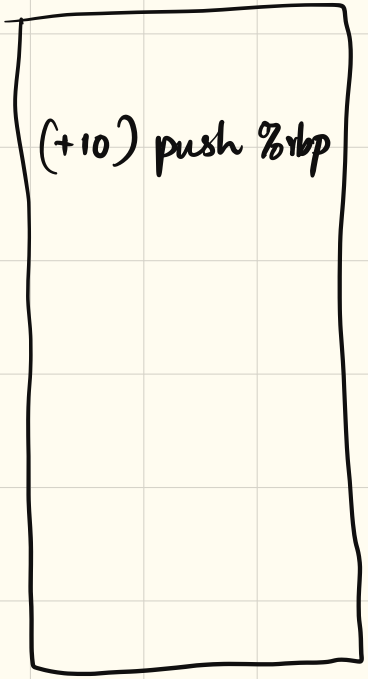
+10 push %rbp

SP → 8 KB

Assuming RSP = 8 KB

store at add^r (8KB − 8)

CPU translated add^r 28 KB − 8



OS sets the base register val depending on the physical location

Memory isolation

How to stop VA = 20 KB access ?

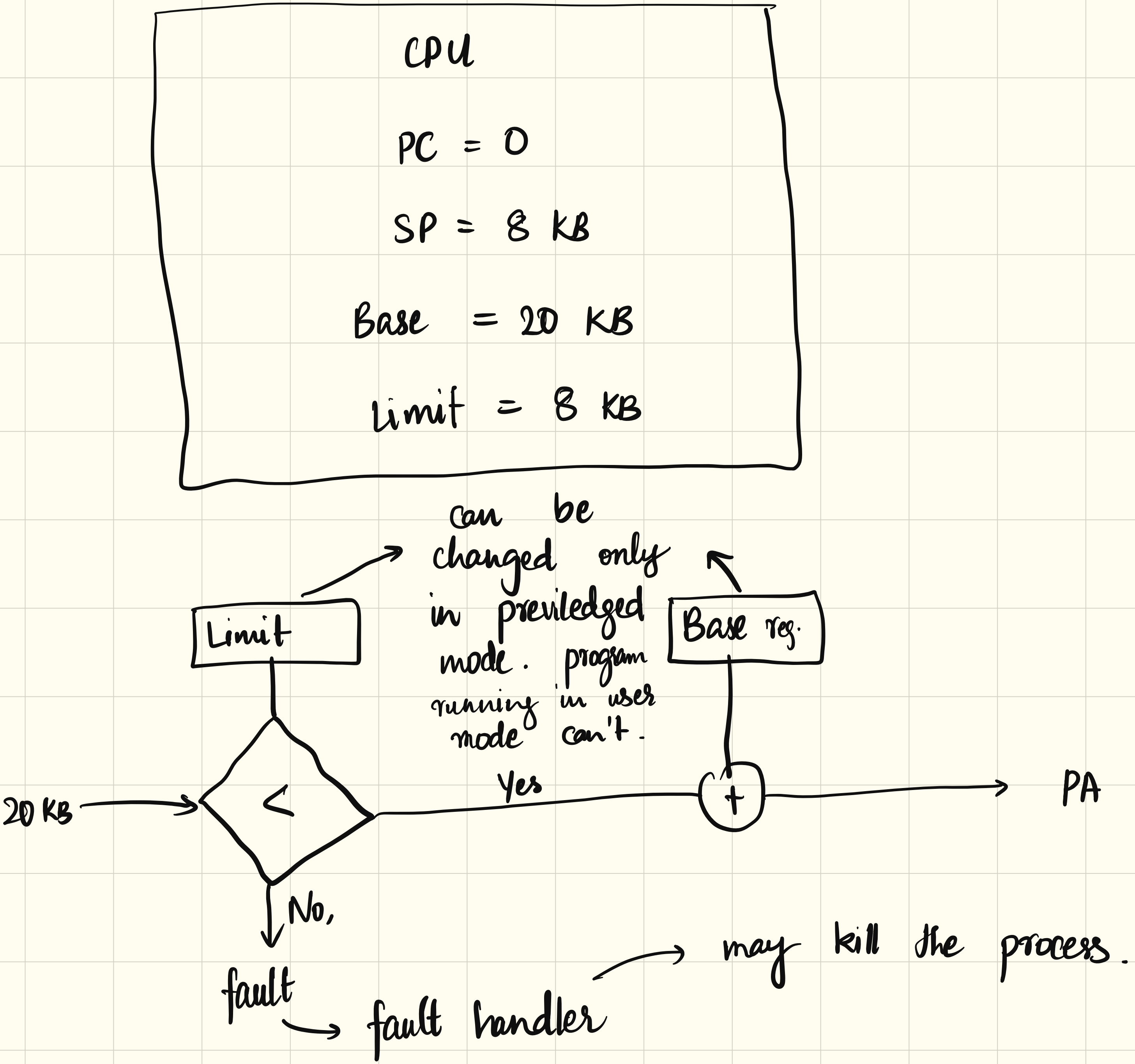
Earlier

Linker and loader
↓
converts relocatable code addresses
↓
no translation
direct CPU acces

→ easy attacks

∴ VA

∴ Limit register



CPU

PC = 0

SP = 8 KB

Base = 20 KB

limit = 8 KB

can be changed only in previledged mode. program running in user mode can't.

may kill the process.

Context switch → save and restore.

Issues

→ Contiguous physical memory

→ Physical memory must be greater than address space size.

→ Small address space size → Unhappy users

→ Memory inefficient

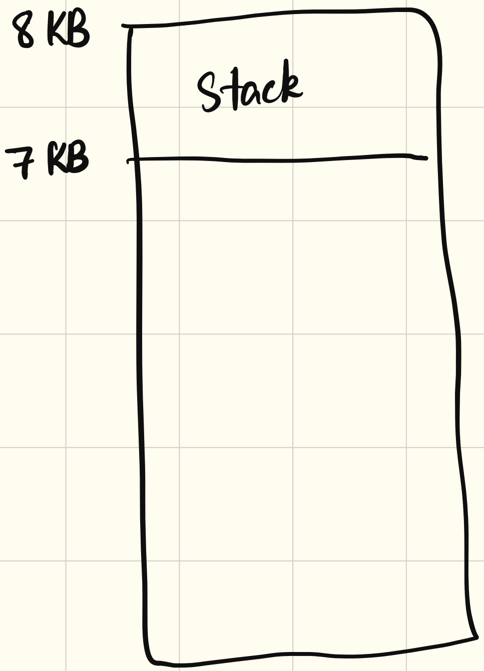
→ Degree of multiprogramming → very less

→ might be good for embedded system. batch processes

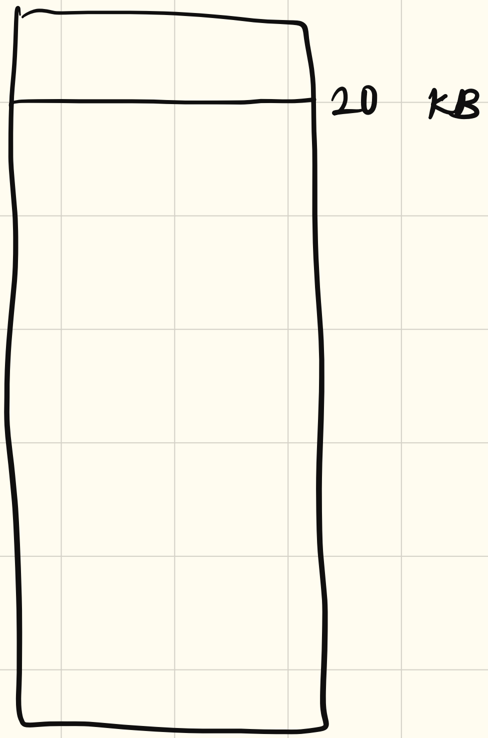
Segmentation

→ Pair of base and limit register for each segment.

→ Generally 3-4 segments, which is why stack and heap are sometimes a single segment



Stack
Base = 21 KB
Limit = 1 KB



Suppose heap :

3000

3000 + 32 KB ✗

offset :

3000 − 1024 + 32 KB ✓

How does CPU decide which segment to use?

→ ① Explicit add^{r}

Stack growth in opp. direction?

Context switch?

Advantages and Disadvantages?

Explicit Addressing:

Use past of the code to identify the segment

e.g:

→ VA = 8KB, address length = 13 bits, 3 segments

→ 2 MSB used to specify segment:

00 → code

01 → data

11 → stack

→ Max size of each segment ⟶ $\overbrace{2^{11}}^{11\ \text{bits}} \rightarrow \boxed{2\ \text{KB}}$

IO → automatically handled

Issues

Advantage : we need to allocate memory only when needed,

(code + data) → load

Implicit Addressing

CPU automatically decides based on instruction:

→ Code segment for inst^r access

→ Fetch, jump, call } → access code segment

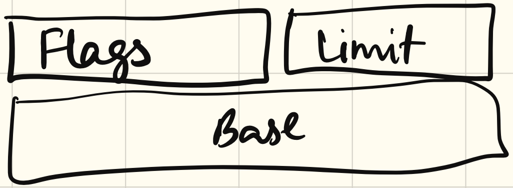
→ State segment for stack operations

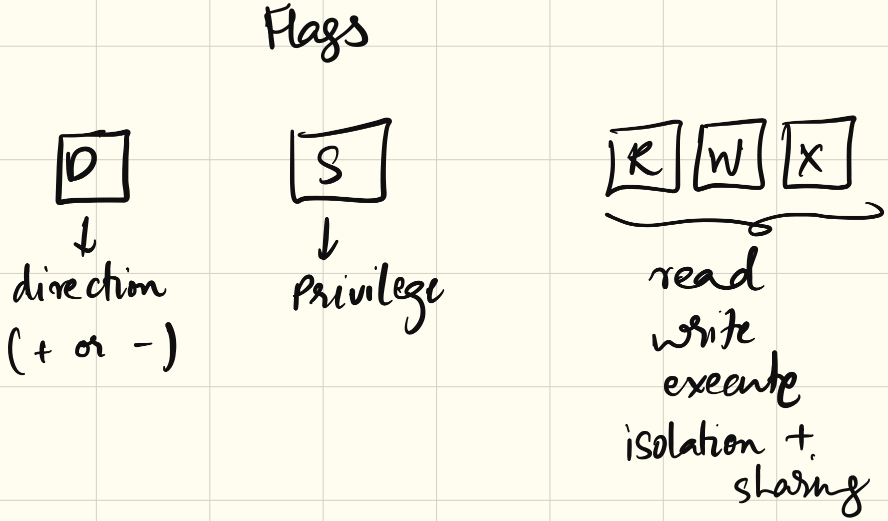
→ Push, pop ⇒ stack

indirect addressing
with SP, BP, (rbp, sbp)

→ Data segment for other addresses.

Stack growth

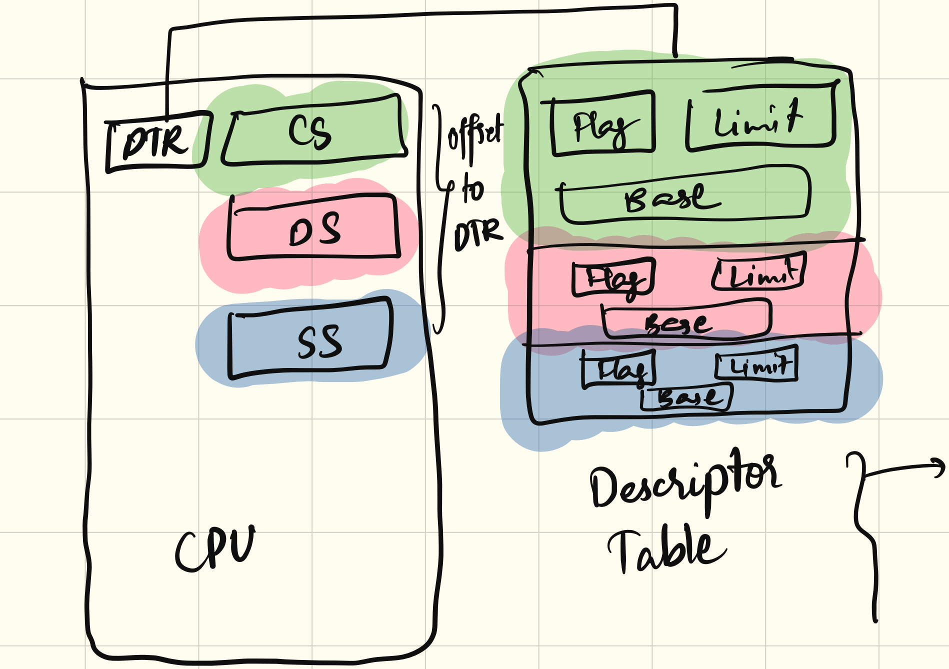




0 → +ve direction
1 → -ve direction
} May change depending on arch.

In reality

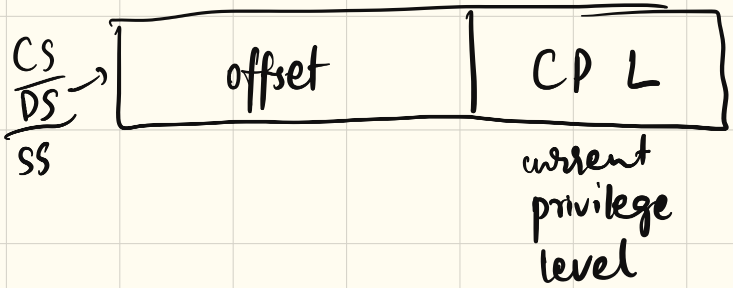
DTR : descriptor table register



Descriptor Table } → may be for each register
or a single table → for all processes

Multiple code/stack/data segments possible

e.g.: insts: for OS seperate



descriptors depend on the architecture.

limit

Separate descriptors for user and kernel mode for the same process.

Today →

Used only for privilege

Address translation → paging

requires HW support

Advantages:

- → Easy to implement, more flexibility, enforce permission
- → Save memory wastage for unused addresses.

Disadvantages

- → External fragmentation.
- → Cannot support discontiguous sparse mapping.

Disk defragmentation → halts other things

Next class: ~~Paging~~ Tutorial Session