CS 354 - Last Lecture

①

Review

Intro to OS.

- CPU virtualization
- Memory virtualization

many pgms run, how to run them all at once?

1 CPU

Mechanisms vs. Policies

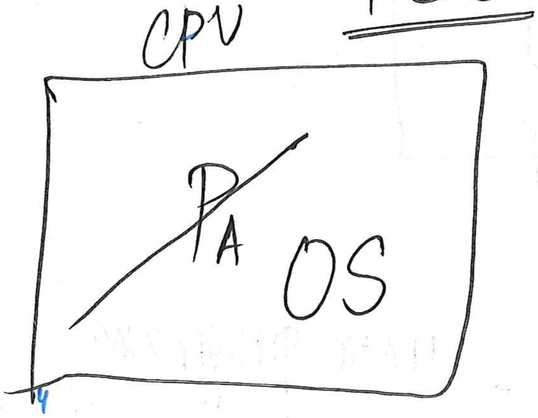
Policies: FIFO, RR

Mechanisms

"Process" state

support from h/w

PCB - Process Control Block. "C struct"



"scheduler"

Ready

P_B, P_c, P_D

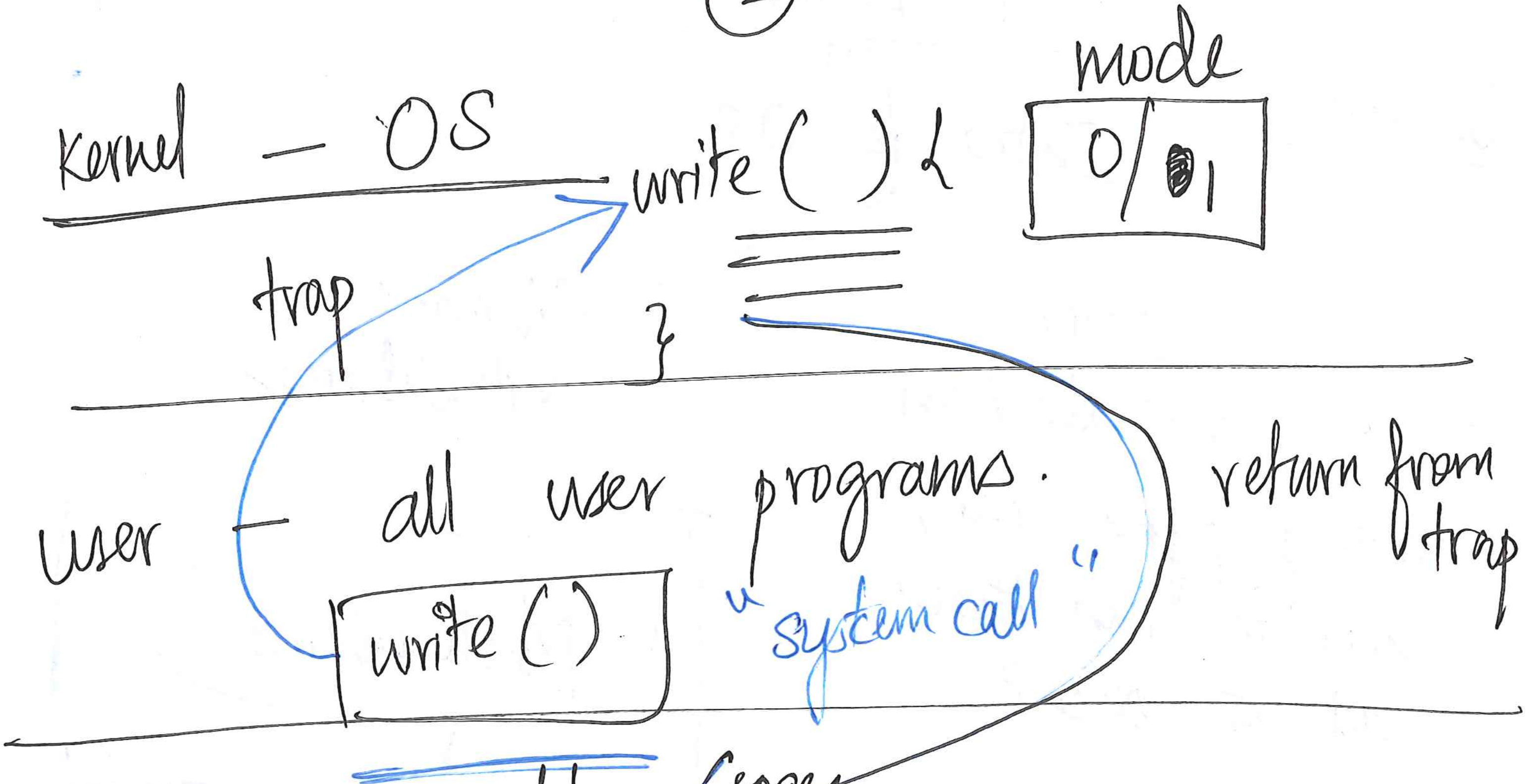
"timer interrupts"

↓

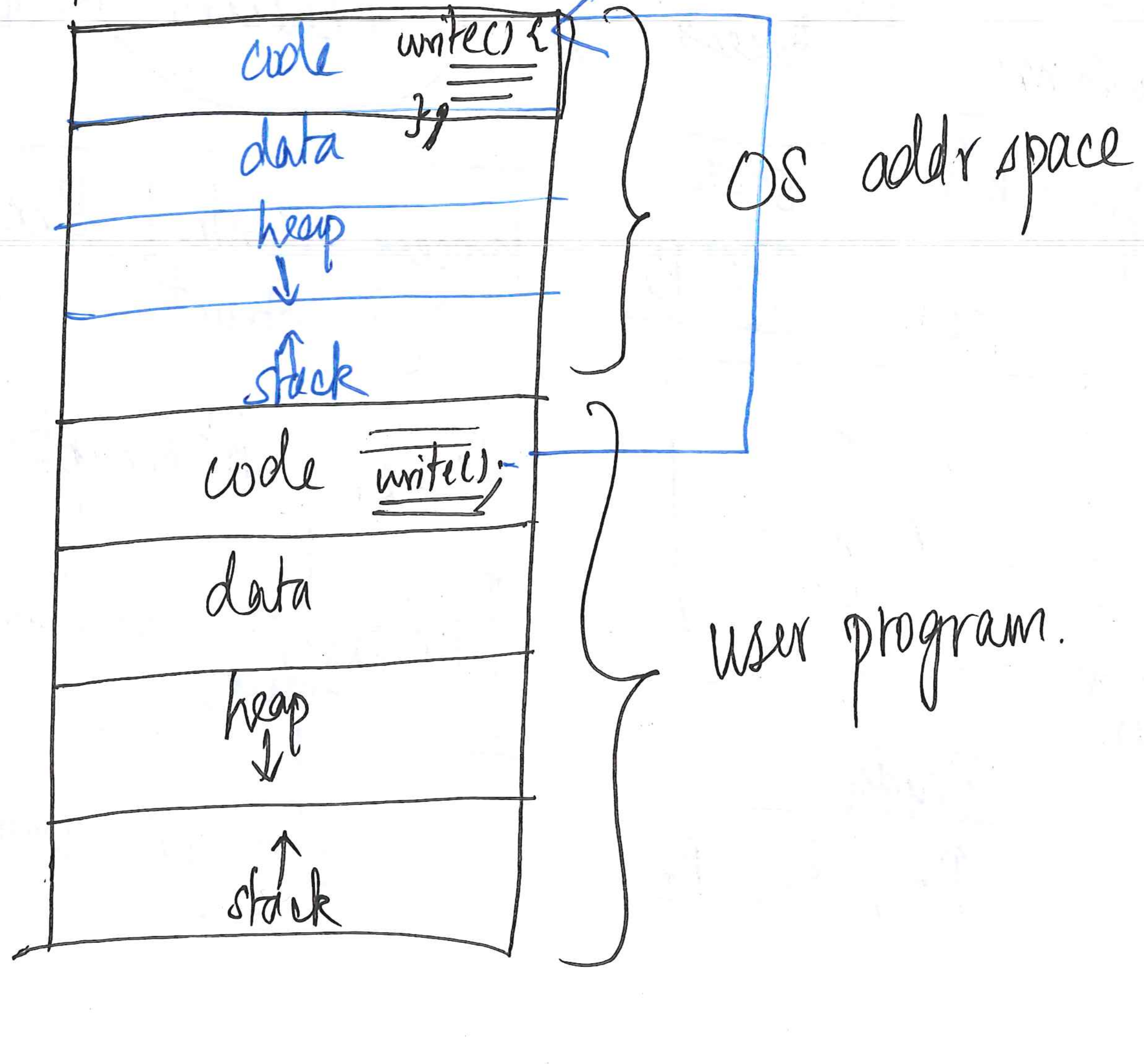
interrupt handler. (timer).

↓

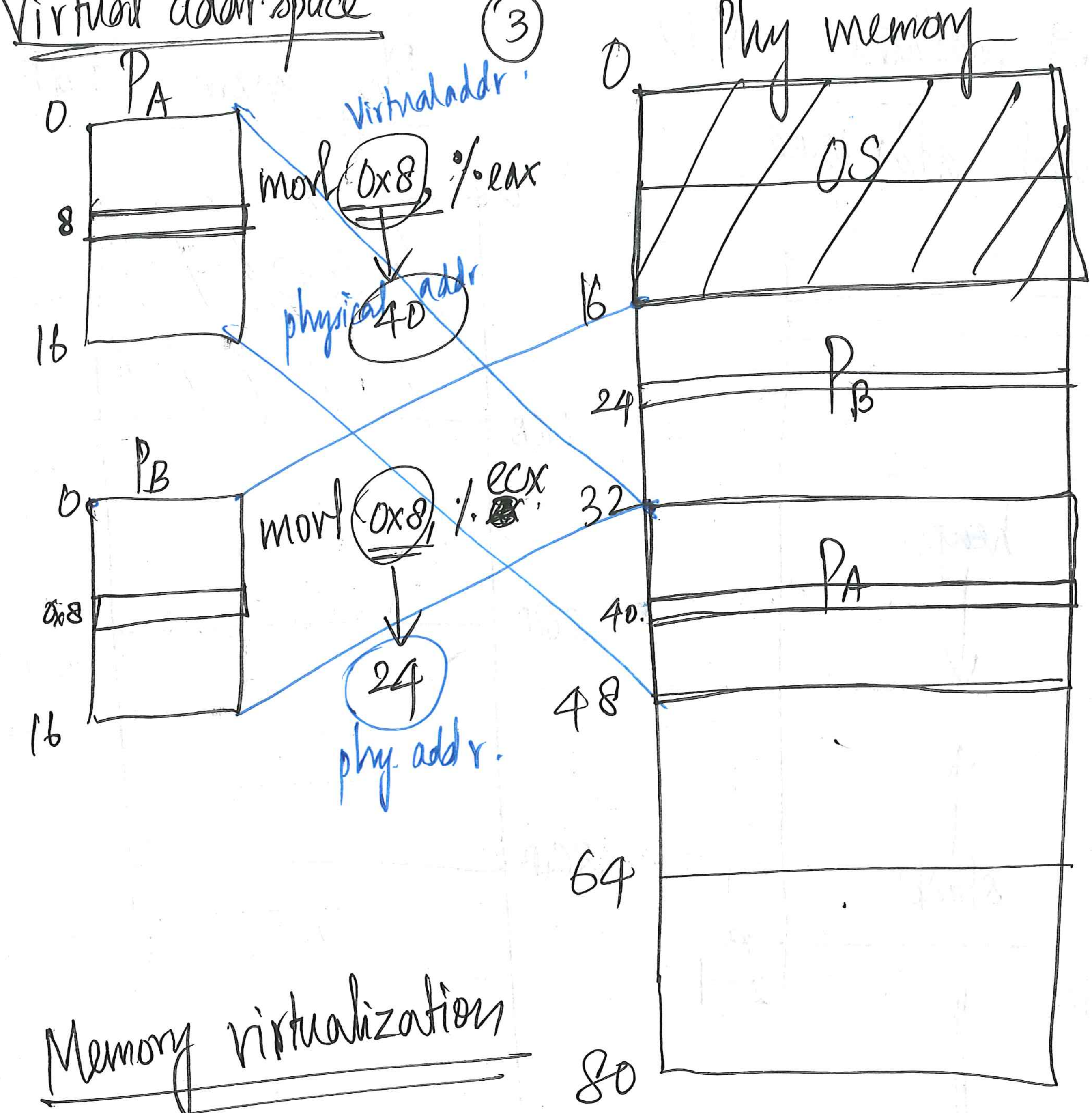
OS gains control.



one process — address ~~space~~.



Virtual addr. space



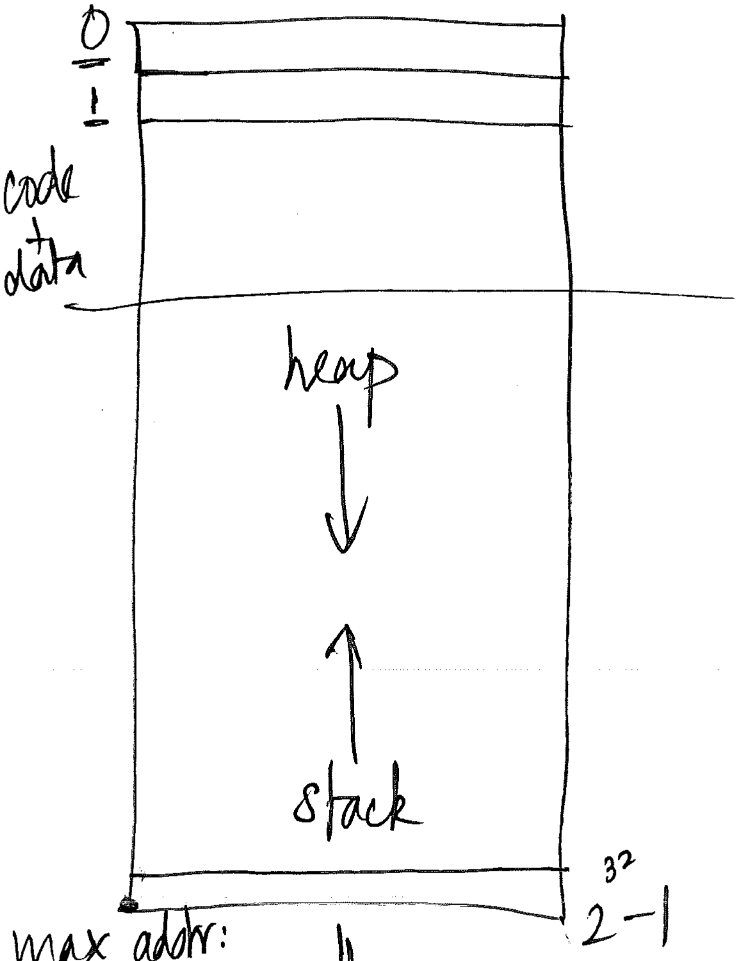
Memory virtualization

Virtual addr → Phy. addr

hardware support.

32-bit machine

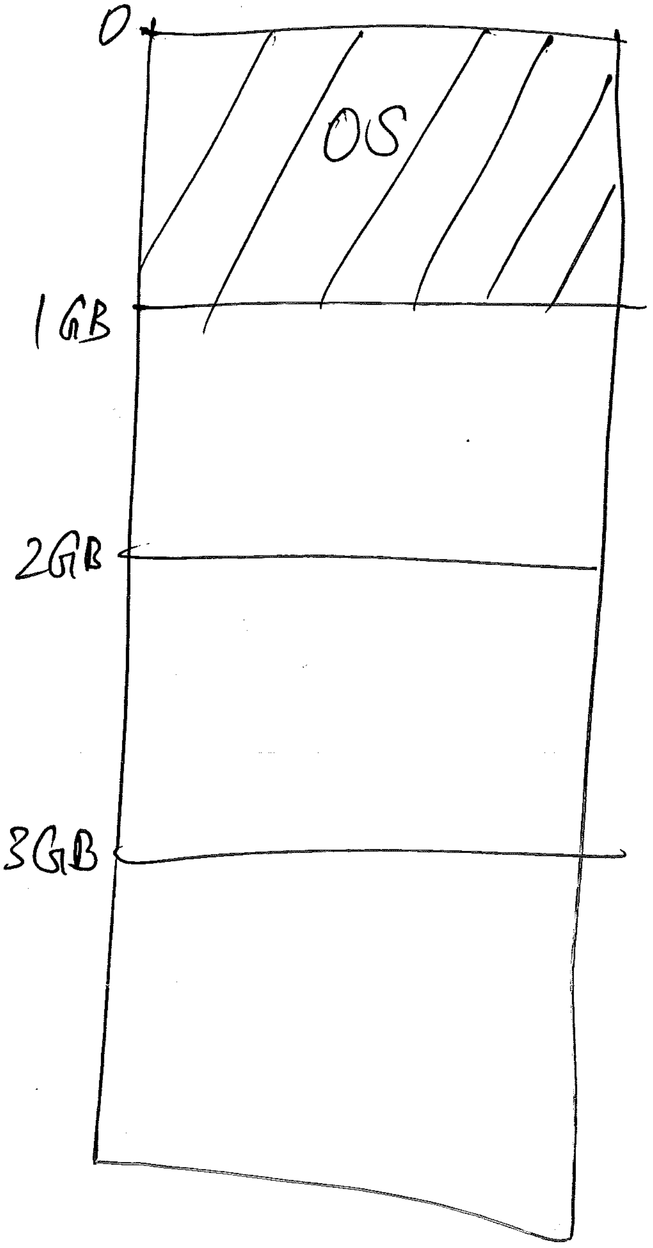
Virtual addr space



$$2^{32} = 2^2 \times \boxed{2^{30}}$$

$$= 4 \times 1GB = \underline{4GB}$$

Phy mem. (4GB)

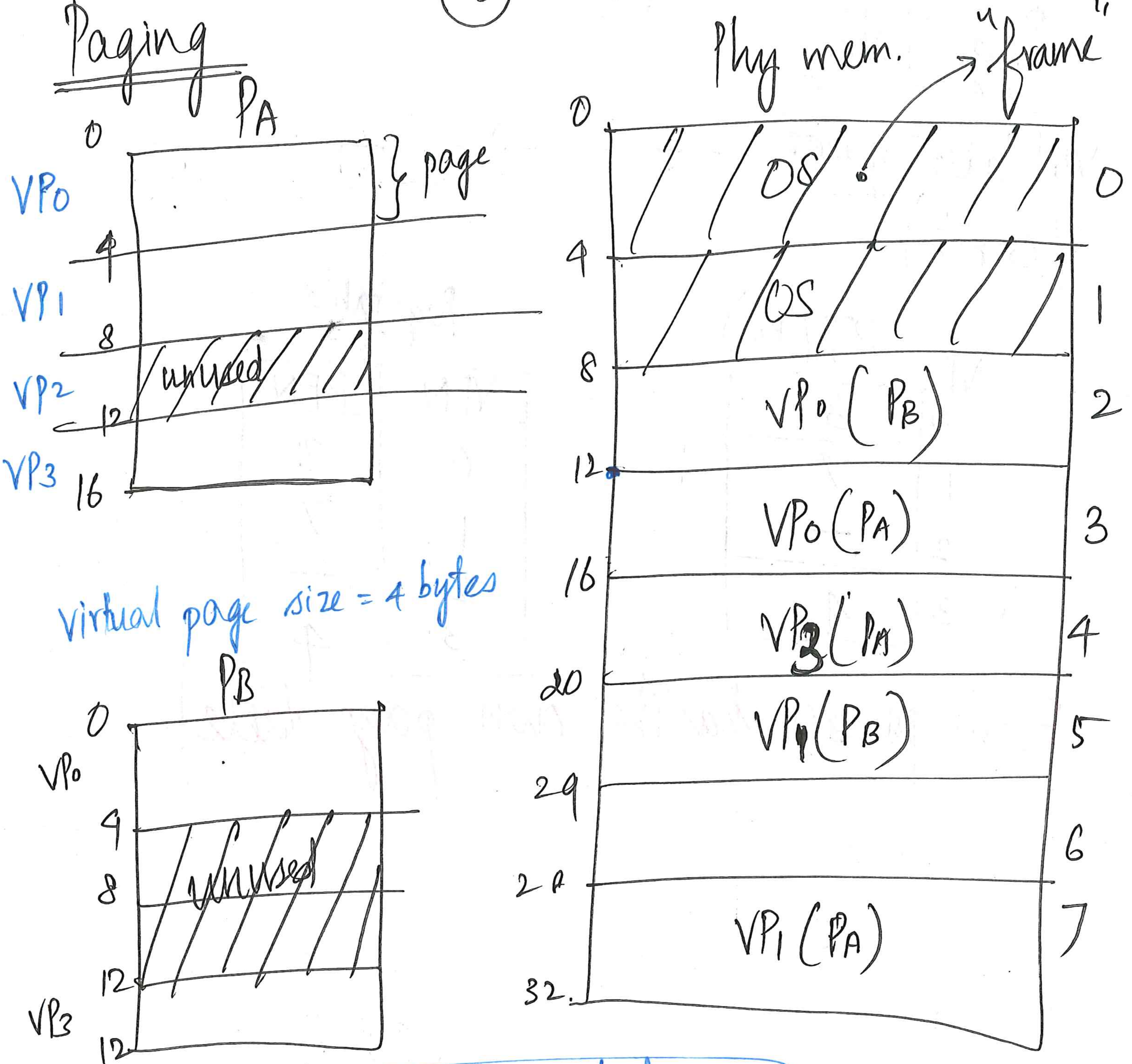


$2^{10} = 1$KB

$2^{20} = 1$MB

$2^{30} = 1$GB

Paging



page size = 4 bytes

frame size = 4 bytes

Page Table

Virt addr. space: — — — — — —

(6 bits)

PT(P_A)

VP	
0	3
1	7
2	—
3	4

Pagetable

VPN	PFN
0	3
1	7
2	—
3	4

Each process has its own page table!

Virt. addr space: 6 bits

_ _ | _ _ _ _

VPN Virtual Page Number | VPO Virtual Page offset.

page size = 16 bytes.

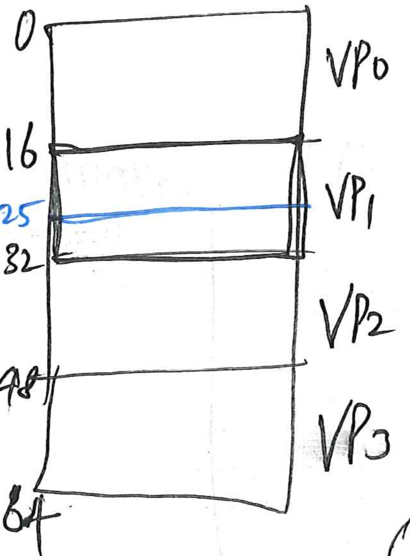
$$\frac{2^6}{2^4} = \boxed{2^2} = 4 \text{ virtual pages.}$$

8 physical frames.

$$\frac{128}{16} = \frac{2^7}{2^4} = 2^3 = \boxed{8}$$

addr: $\boxed{\overset{16}{0}\ 1}\ \boxed{\overset{8}{1}\ \overset{4}{0}\ \overset{2}{0}\ \overset{1}{1}}$

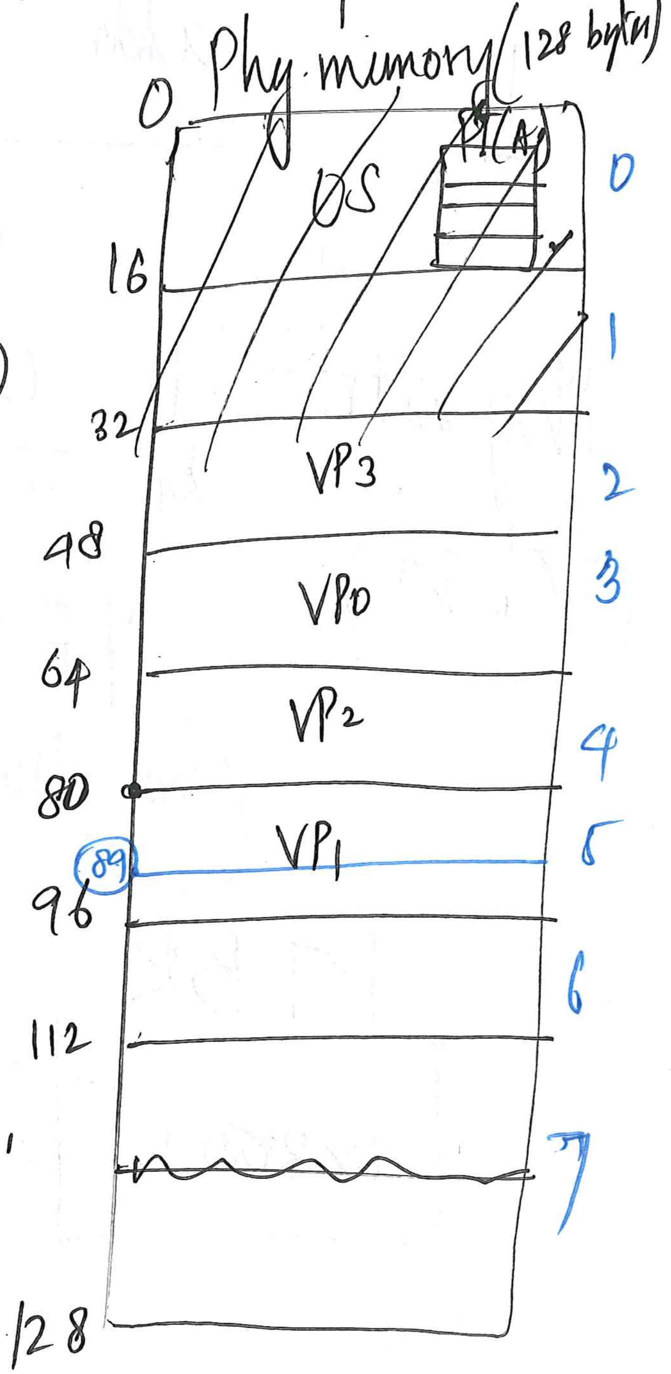
VPN = 1 VPO = 9

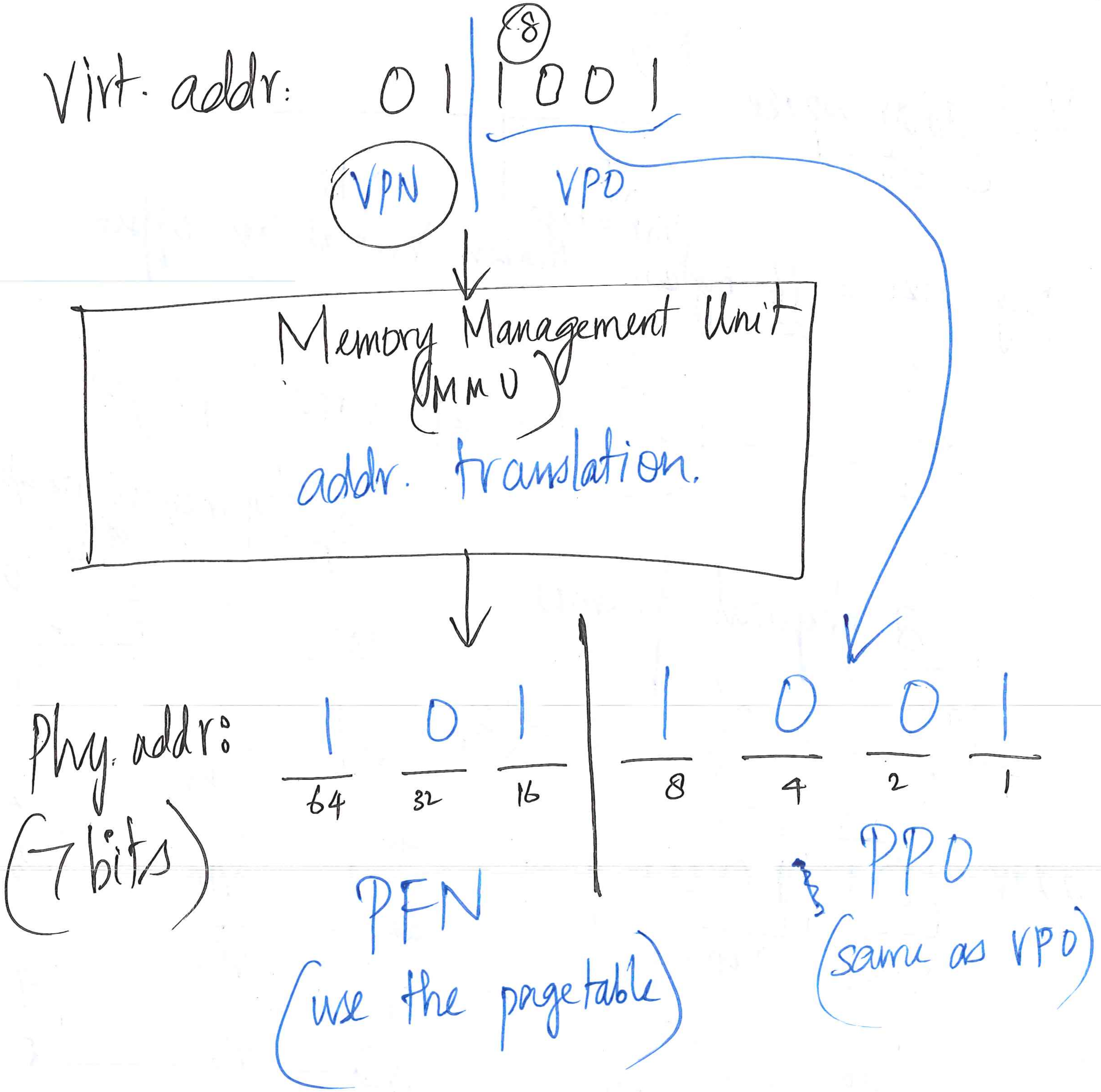


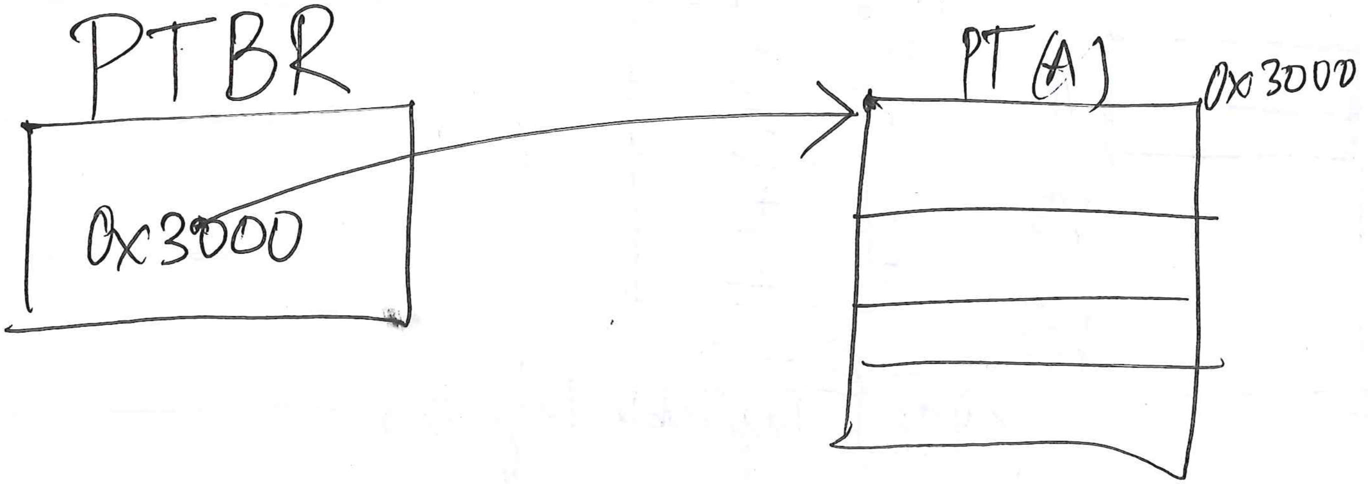
PT

VP	PFN
0	3
1	5
2	4
3	2

PTE - PageTable Entry




8
Virt. addr: 0 1 | 1 0 0 1
VPN
VPO
Memory Management Unit
(MMU)
addr. translation.
Phy. addr:
(7 bits)
1 0 1 | 1 0 0 1
64 32 16 8 4 2 1
PFN
(use the pagetable)
PPO
(same as VPO)


PTBR
0x3000
PT(A)
0x3000

Problems

1. for each memory ref ⟹ need 2 mem ref

1. PTE
2. Actual phy. addr.

CPU

TLB

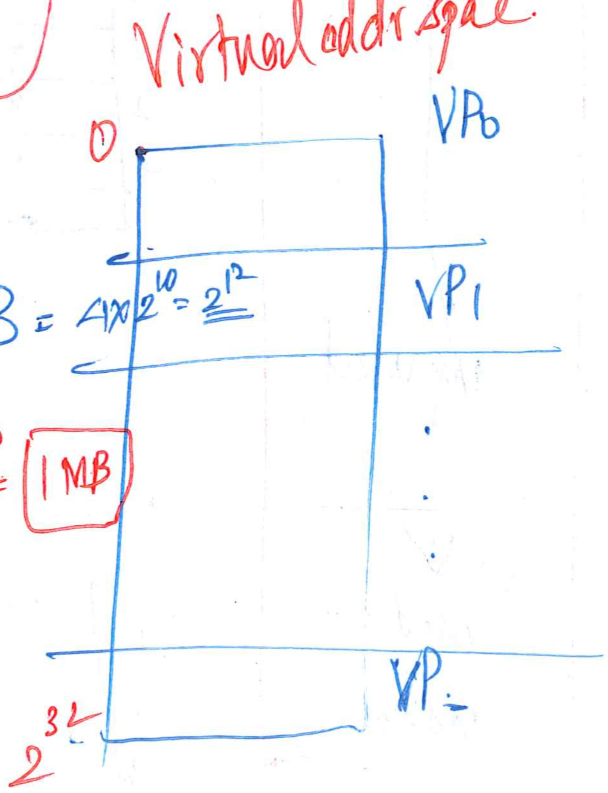
VPN	PFN
0	5
1	7
2	3
3	1

special cache.

2. 32 bit addr space

page size = 4 KB = $4 \times 2^{10} = 2^{12}$

$\# \text{ of } V.P.s = \dfrac{2^{32}}{2^{12}} = 2^{20} = \boxed{1\text{MB}}$



1 PTE = 4 bytes

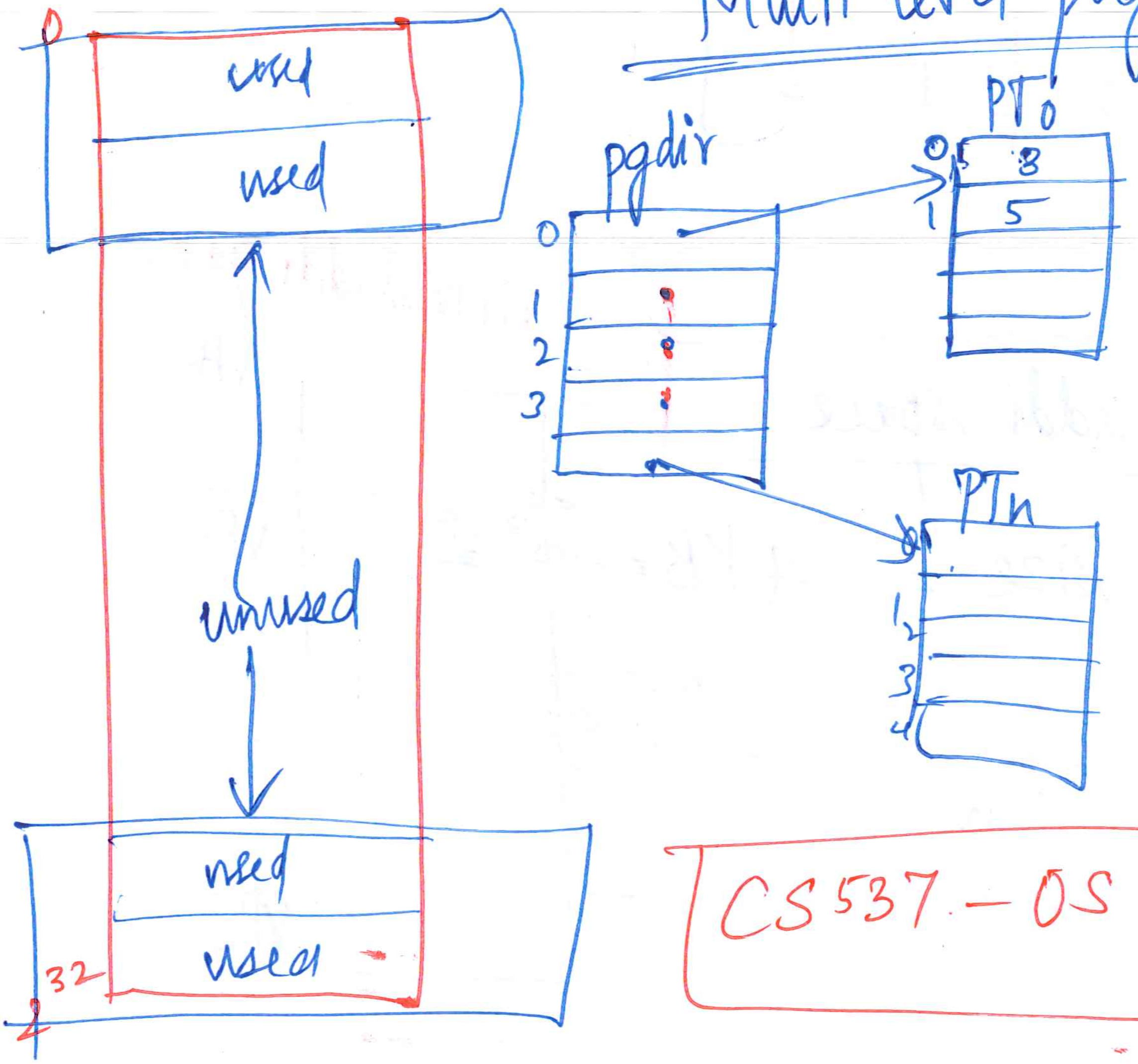
1M entries in the PT

⟹ 4 x 1 MB = 4MB for one PT.

100 processes run ⟹ 400 MB

1000 " " ⟹ ~4GB

Multi-level page tables

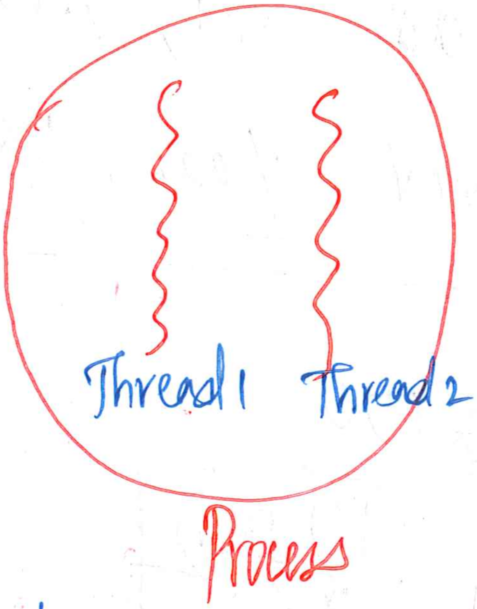


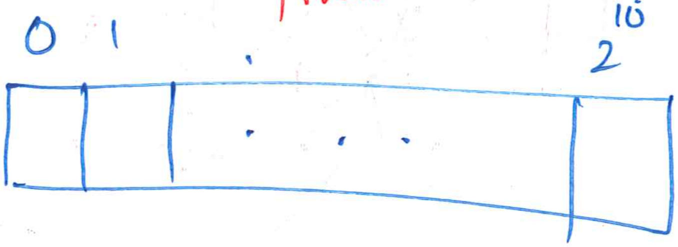
CS537 – OS

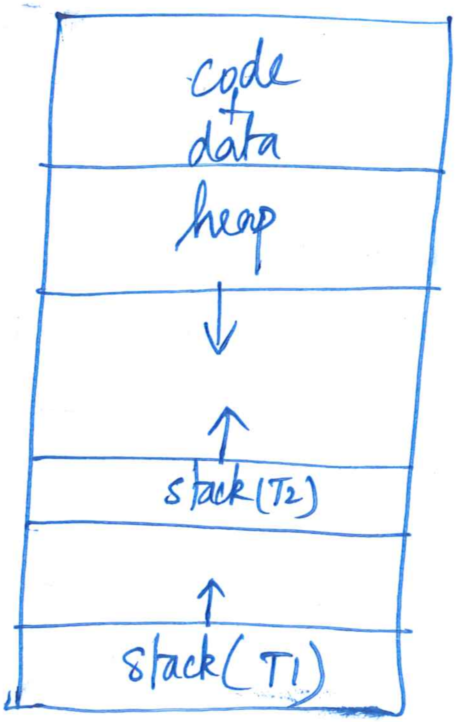
Concurrency

```
main() {

}
fn1() {

}
fn2() {

}
```

"Threads"





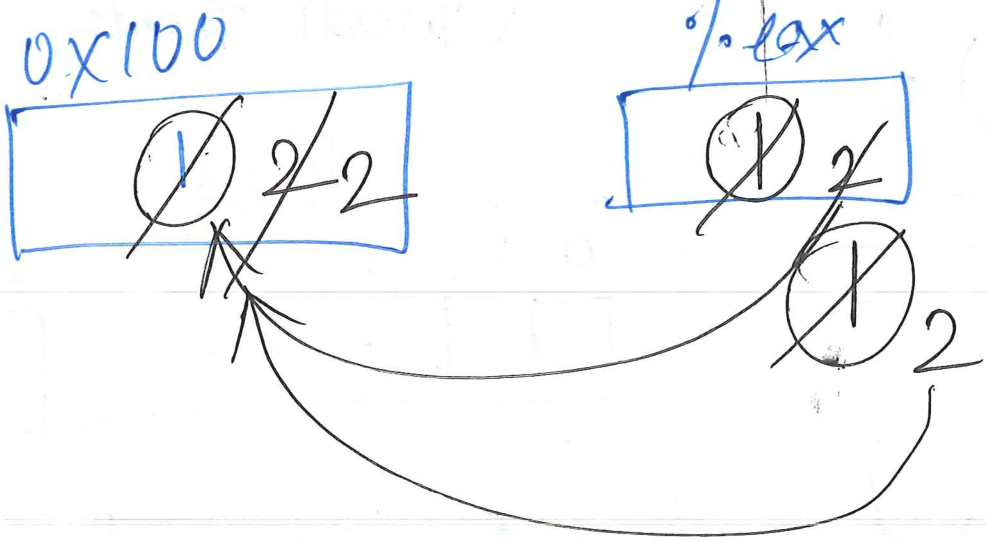


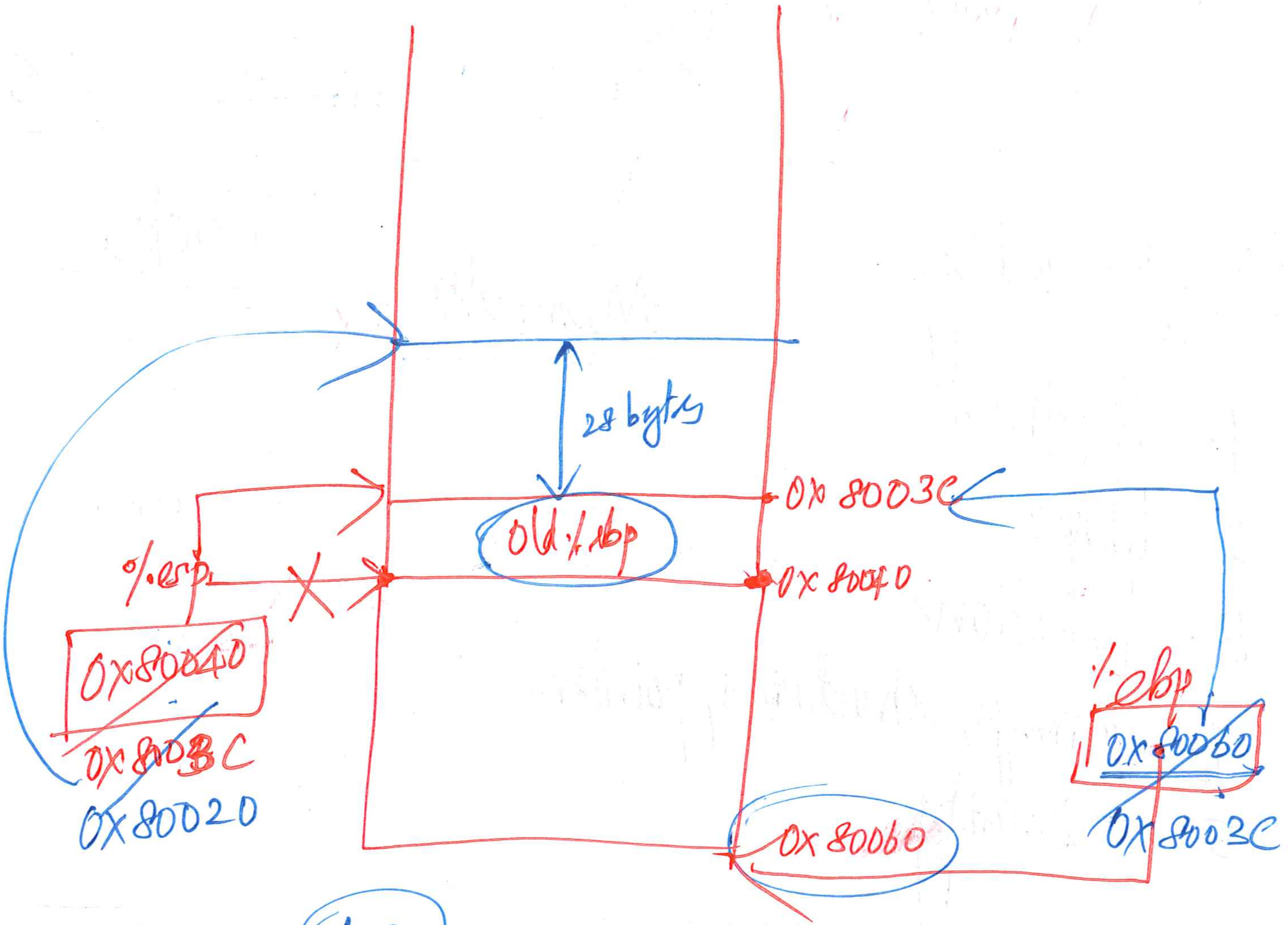
T1

```
① mov ~~$~~0x100, %eax
→ add $1, %eax
  mov %eax, 0x100
```

T2

```
② mov $100, %eax
  add $1, %eax
  mov %eax, 0x100
```





28 → 0x1C

0x8003C
− 0x 1C

0x80020

Final

OS

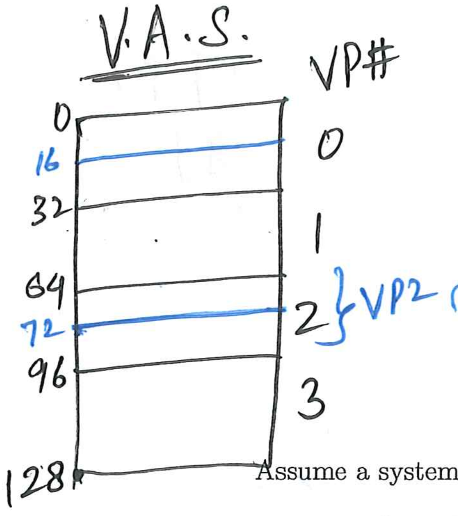
Cache

Mem. alloc

X. Assembly

1. if/then
2. loops
3. functions
4. arrays, structures, pointers
5. Security

aefis. wisc. edu



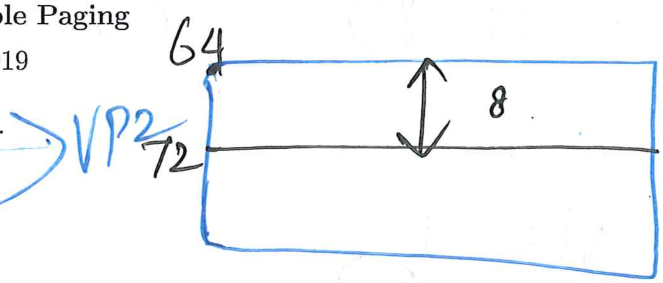
$\frac{72}{32} = \boxed{2}$

$16/32 = 0.$

CS 354-Intro to Computer Systems

Worksheet - Simple Paging

April 22, 2019



Assume a system with a simple linear page table.

Parameters:

- virtual address space size = 128 bytes
- page size = 32 bytes
- physical memory size = 256 bytes
- Size of one Page Table Entry (PTE) = 1 byte
- Value of Page Table Base Register (PTBR) = 8

The **left most bit (MSB)** in a PTE is the **valid** bit and it determines if the virtual page is valid or not. The **3 right most bits (LSBs)** in a PTE contains the **Physical Frame Number (PFN)**. You may assume that all the other bits are unused. The contents of the Page Table are shown below.

Page Table (from entry 0 down to the max size)

V.A: _ _ | _ _ _ _ _

VPN VPO

0	0x81
1	0x00
2	0x87
3	0x03

V PFN

0x81 → $\boxed{1}$000 0$\boxed{001}$ PFN=1

0x87 → 1000 0$\boxed{111}$ (PFN = 7)

The instruction below loads a **single byte** from virtual address 72 into the register %eax. This instruction resides at virtual address 16 within the address space of the process.

16: mov 72, %eax

In the diagram of physical memory shown on the next page:

1. Put a **BOX** around the **page table** and label it.
2. Put a **BOX** around each **valid virtual page** (and label them).
3. **CIRCLE** the memory addresses that get referenced during the execution of the instruction, including both instruction fetch and data access (assume there is no TLB).
4. **LABEL** the memory addresses (that you circled) with a **NUMBER** that indicates the **ORDER** in which various physical addresses get referenced.

Physical Memory

PTBR	0 1	1	2 3	3	4	5	6	7	PFN ~~PN~~
PT	8	9	10	11	12	13	14	15	0
	16	17	18	19	20	21	22	23	
	24	25	26	27	28	29	30	31	
	32	33	34	35	36	37	38	39	
VP0	40	41	42	43	44	45	46	47	1
	48 2	49	50	51	52	53	54	55	
	56	57	58	59	60	61	62	63	
	64	65	66	67	68	69	70	71	
	72	73	74	75	76	77	78	79	
	80	81	82	83	84	85	86	87	2
	88	89	90	91	92	93	94	95	
	96	97	98	99	100	101	102	103	
	104	105	106	107	108	109	110	111	
	112	113	114	115	116	117	118	119	3
	120	121	122	123	124	125	126	127	
	128	129	130	131	132	133	134	135	
	136	137	138	139	140	141	142	143	
	144	145	146	147	148	149	150	151	4
	152	153	154	155	156	157	158	159	
	160	161	162	163	164	165	166	167	
	168	169	170	171	172	173	174	175	
	176	177	178	179	180	181	182	183	5
	184	185	186	187	188	189	190	191	
	192	193	194	195	196	197	198	199	
	200	201	202	203	204	205	206	207	
	208	209	210	211	212	213	214	215	6
	216	217	218	219	220	221	222	223	
	224	225	226	227	228	229	230	231	
	232 4	233	234	235	236	237	238	239	7
VP3	240	241	242	243	244	245	246	247	
	248	249	250	251	252	253	254	255	

9. Functions in Assembly

Consider the given C function and its corresponding assembly code.

C Function

```
int proc(void)
{
    int x,y;
    scanf("%x %x", &y, &x);
    return x-y;
}
```

Assembly Routine

```
proc:
1     pushl    %ebp
2     movl     %esp, %ebp
3     subl     $24, %esp
4     subl     $4, %esp
5     leal     -12(%ebp), %eax
6     pushl    %eax
7     leal     -16(%ebp), %eax
8     pushl    %eax
# .LC0 is pointer to string "%x %x"
9     pushl    $.LC0
10    call     scanf
11    addl     $16, %esp
12    movl     -12(%ebp), %edx
13    movl     -16(%ebp), %eax
14    subl     %eax, %edx
15    movl     %edx, %eax
16    leave
17    ret
```

Assume the procedure proc starts executing with the following register values. i.e., these are the register values **before** line 1 in proc is executed.

Register	Value
%esp	0x80040
%ebp	0x80060

Suppose proc calls scanf (line 10), and that scanf reads values 0x46 and 0x53 from the standard input. Assume that the string "%x %x" is stored at memory location 0x300070. Write all values in **hexadecimal**.

(a) What value does %ebp get set to on line 2? 0x8003C

(b) What value does %esp get set to on line 4? 0X80020

(c) At what address is local variable x stored? -12(%ebp) = 0x8003C − 0x C = 0x80030

(d) At what address is local variable y stored? -16(%ebp) = 0x8003C − 0x 10 = 0x8002C

(e) What is the value of register %ebp after leave (line 16) is executed? 0x80060

movl %ebp, %esp
pop %ebp