

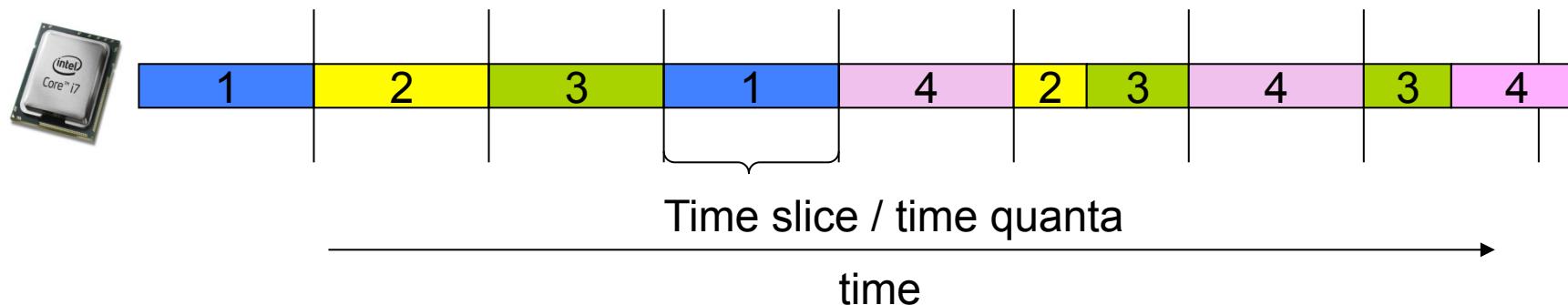
# Memory Management

Chester Rebeiro  
IIT Madras

*CR*

# Multitasking

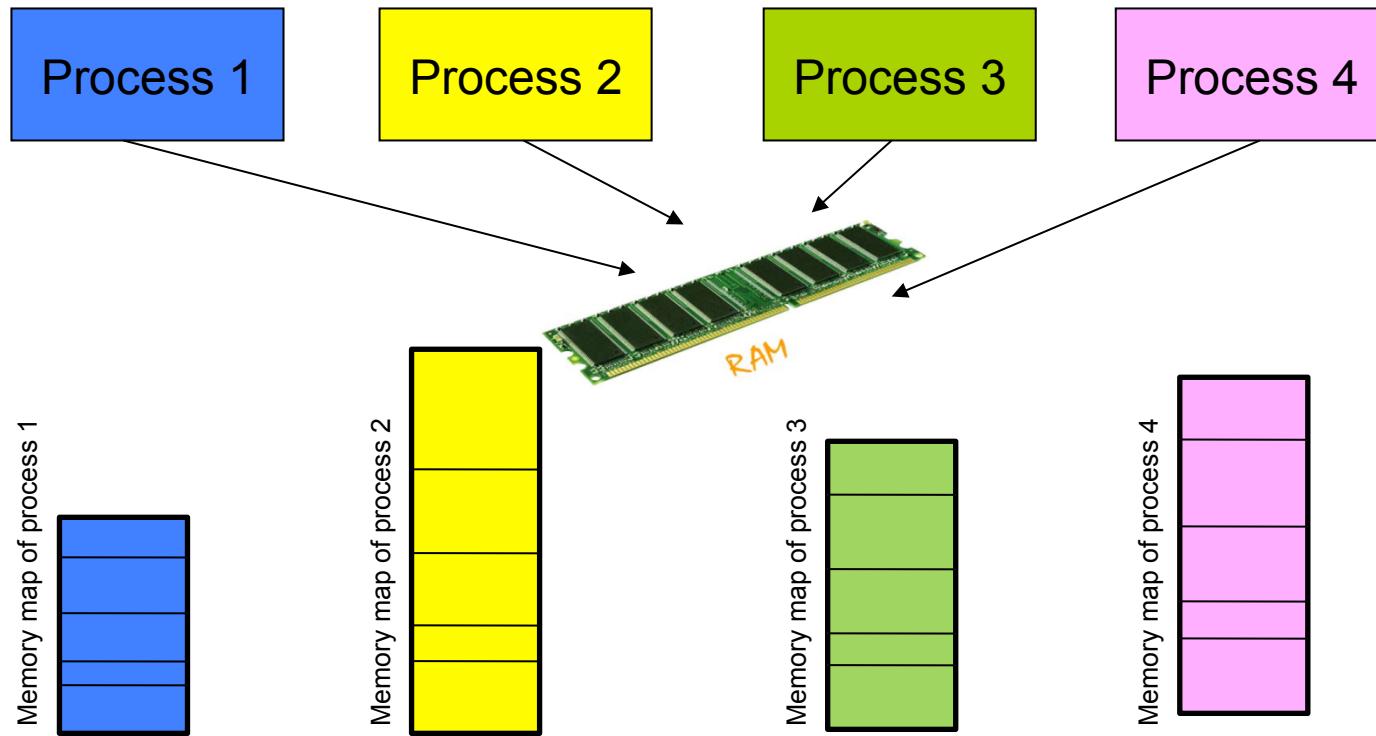
- Multiple programs execute in a time-multiplexed manner



the whole memory cannot simply be segmented and allocated to different processes for these reasons:

1. there is no isolation. there is a straightforward way to access the memory space of one process from another
2. there is a chance that one process' space can leak into another when it grows beyond a point
3. when there are more processes than the available memory, it would run into a crunch
4. it is not convenient to move the process memory addresses easily

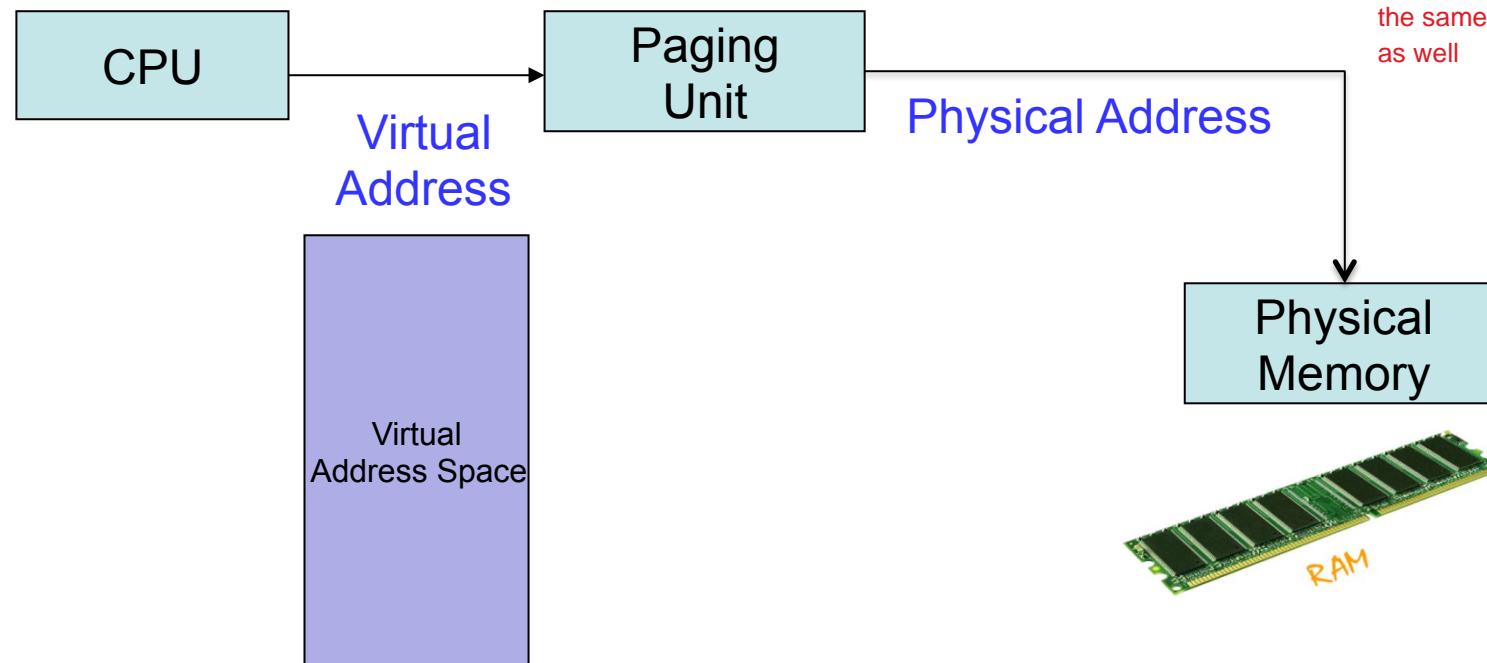
# Sharing RAM



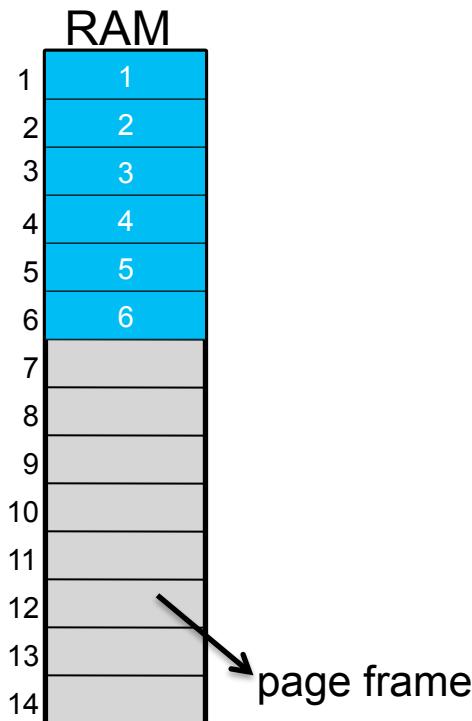
using virtual address solves the above problems:  
each process sees a large memory  
there is a clear isolation between the processes

the penalty paid for this is the overhead of the  
paging and memory management units

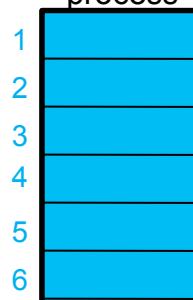
embedded devices sometimes do not prefer this  
mode because of the indeterminism from the  
possibility of having faults - real time systems  
the same goes for caches and are thus disabled  
as well



# Virtual Memory



Virtual address space of process



process page table

block	page frame
1	1
2	2
3	3
4	4
5	5
6	6

Memory in a process divided into fixed size blocks (typically of 4KB)

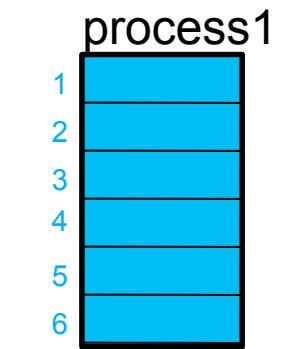
Every time a memory location is accessed, the processor looks into the page table to identify the corresponding page frame number.

Frames do not have to be contiguous

# Virtual Memory

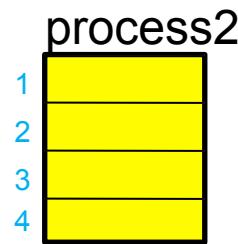


Blocks from  
Several processes  
can share pages in  
RAM  
simultaneously



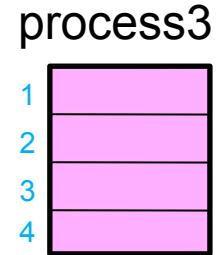
process page table

block	page frame
1	14
2	2
3	13
4	4
5	1
6	8



process page table

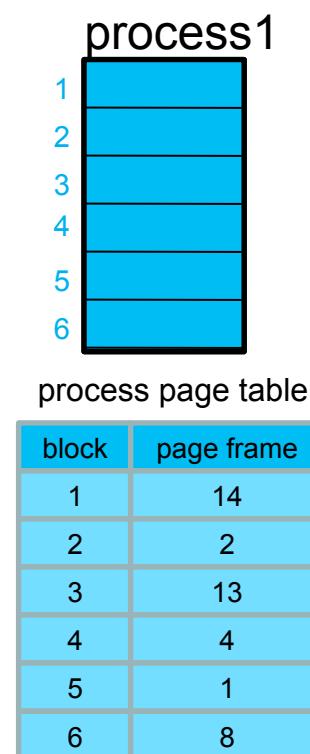
block	page frame
1	10
2	7
3	12
4	9



process page table

block	page frame
1	11
2	6
3	3
4	5

# Virtual Memory



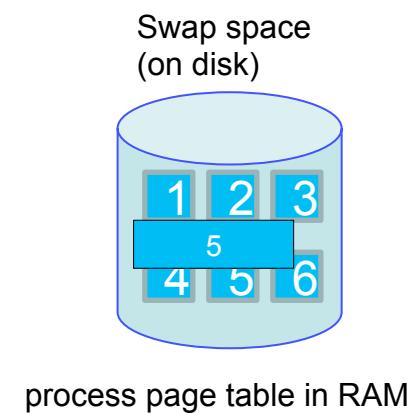
Do we really need to load all blocks into memory before the process starts executing?

No.

Not all parts of the program are accessed simultaneously.  
Infact, some code may not even be executed.

Virtual memory takes advantage of this by using a concept called demand paging.

# Demand Paging



block	page frame	P
1	14	1
2		0
3		0
4		0
5	1	0
6	8	1

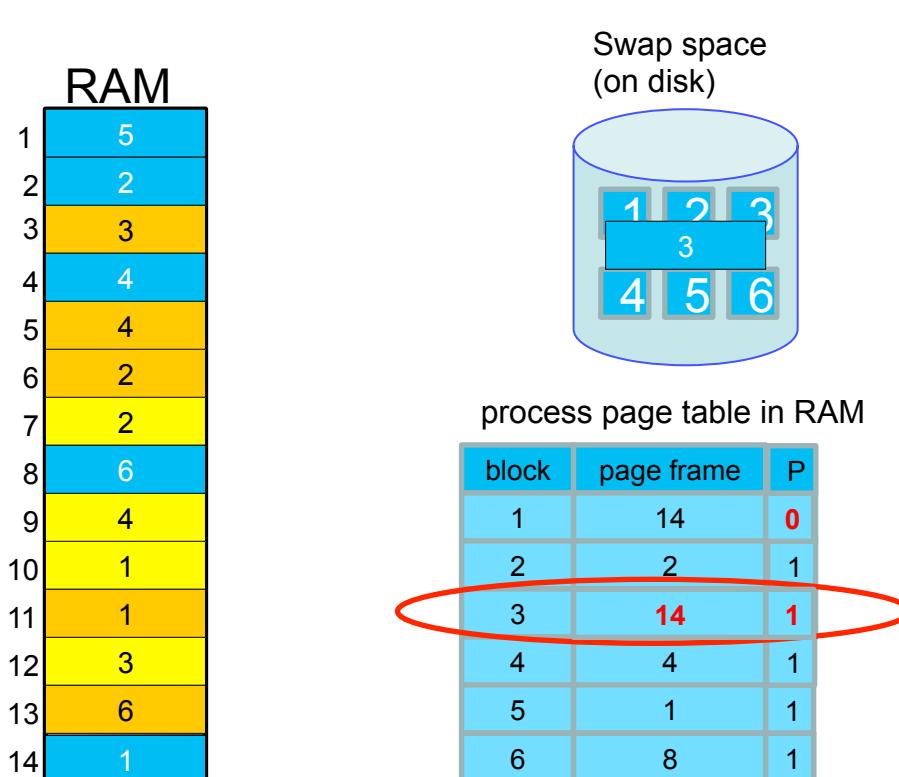
Pages are loaded from disk to RAM, only when needed.

A ‘present bit’ in the page table indicates if the block is in RAM or not.

If (present bit = 1){ block in RAM}  
else {block not in RAM}

If a page is accessed that is not present in RAM, the processor issues a page fault interrupt, triggering the OS to load the page into RAM and mark the present bit to 1

# Demand Paging



If there are no pages free for a new block to be loaded, the OS makes a decision to remove another block from RAM.

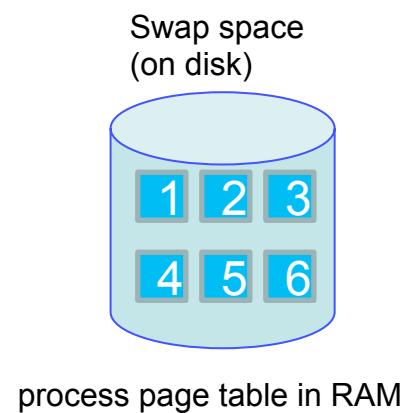
This is based on a replacement policy, implemented in the OS.

Some replacement policies are

- \* First in first out
- \* Least recently used
- \* Least frequently used

The replaced block **may** need to be written back to the swap (swap out)

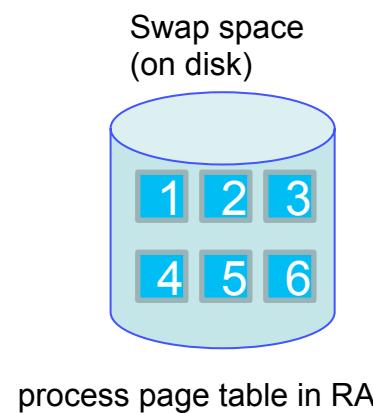
# Demand Paging



The **dirty bit**, in the page table indicates if a page needs to be written back to disk

If the dirty bit is 1, indicates the page needs to be written back to disk.

# Demand Paging



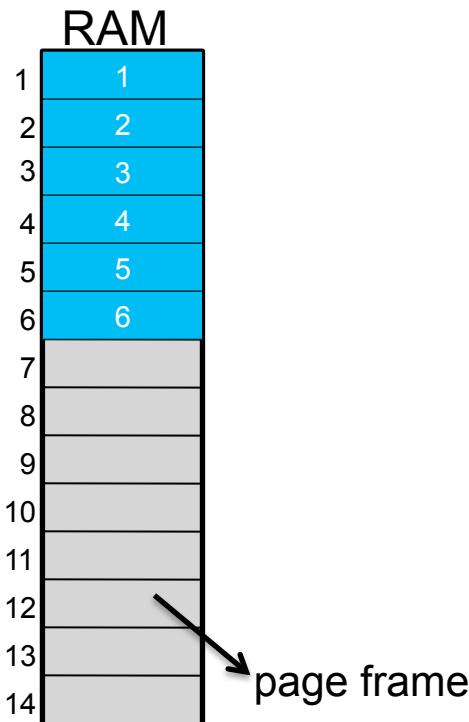
process page table in RAM

block	page frame	P	D	
1	14	0	1	11
2	2	1	1	10
3	14	1	0	00
4	4	1	1	11
5	1	1	0	01
6	8	1	1	10

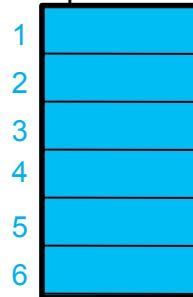
Protection bits, in the page table determine if the page is executable, readonly, and accessible by a user process.

protection bits

# Size of Page Tables



Virtual address space of process



process page table

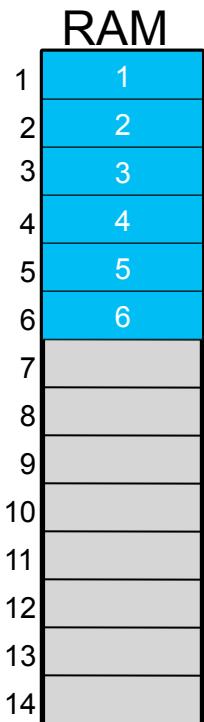
block	page frame
1	1
2	2
3	3
4	4
5	5
6	6

If size of virtual address space is 4 GB ( $2^{32}$ ).

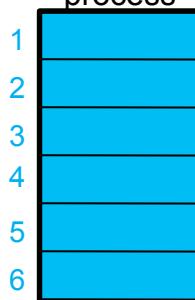
If size of each page frame is 4 KB ( $2^{12}$ )

What is the size of the page table?

# Size of Page Tables



Virtual address space of  
process



process page table

block	page frame
1	1
2	2
3	3
4	4
5	5
6	6

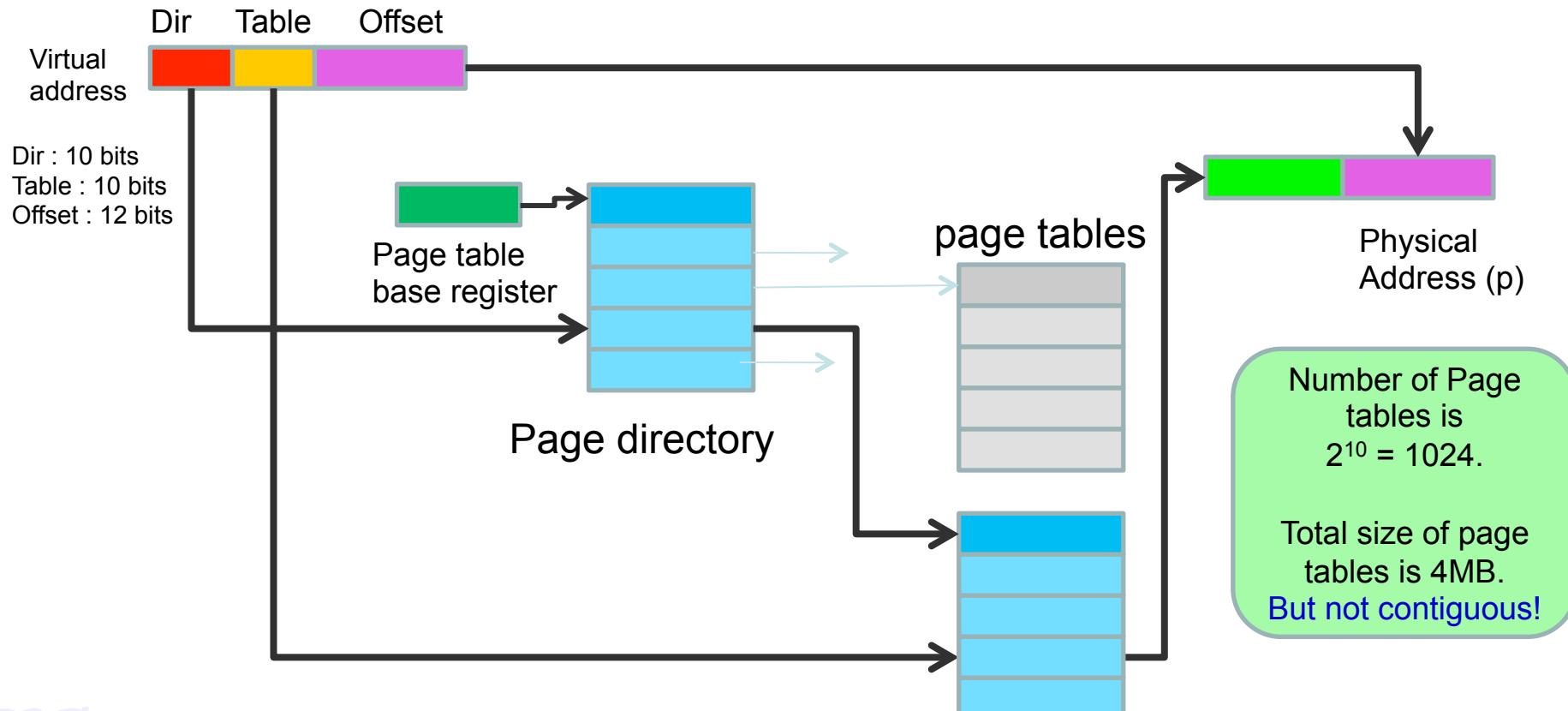
If size of virtual address space is  
4 GB ( $2^{32}$ ).

If size of each page frame is 4 KB ( $2^{12}$ )

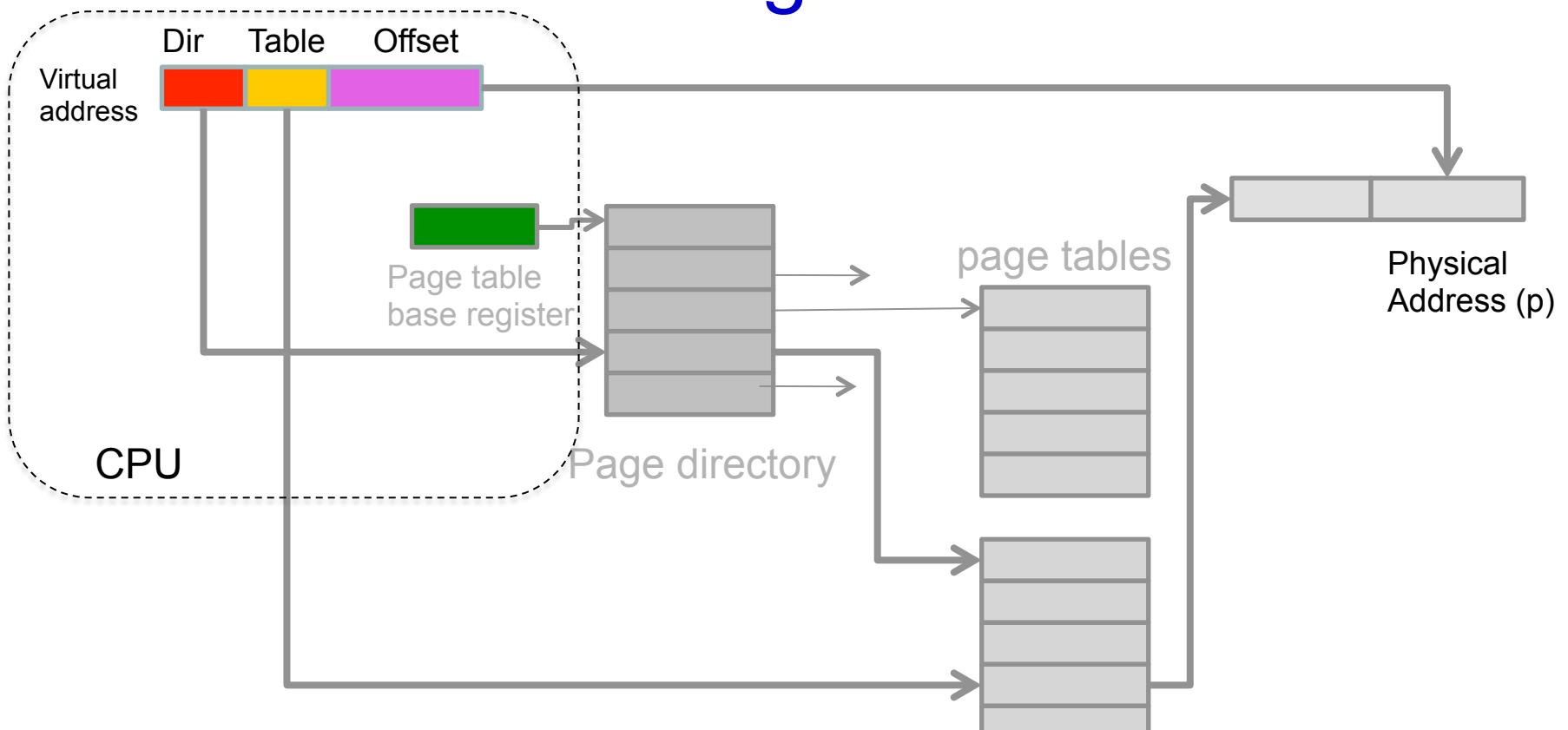
Number of entries in page table?  $2^{20}$

If size of each page entry is 4 bytes,  
4MB of contiguous memory is required.

## 2 Level Page Translation

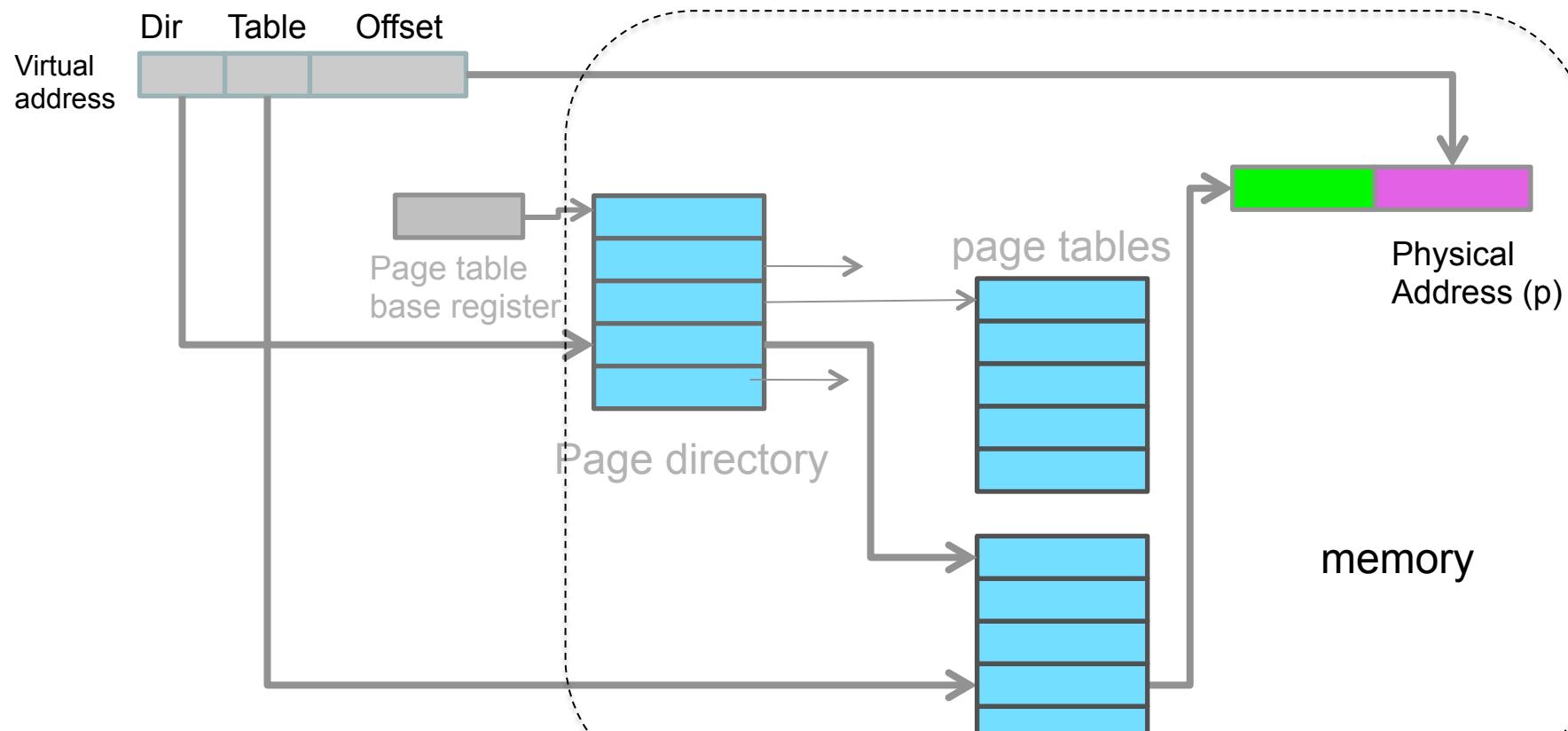


## 2 Level Page Translation

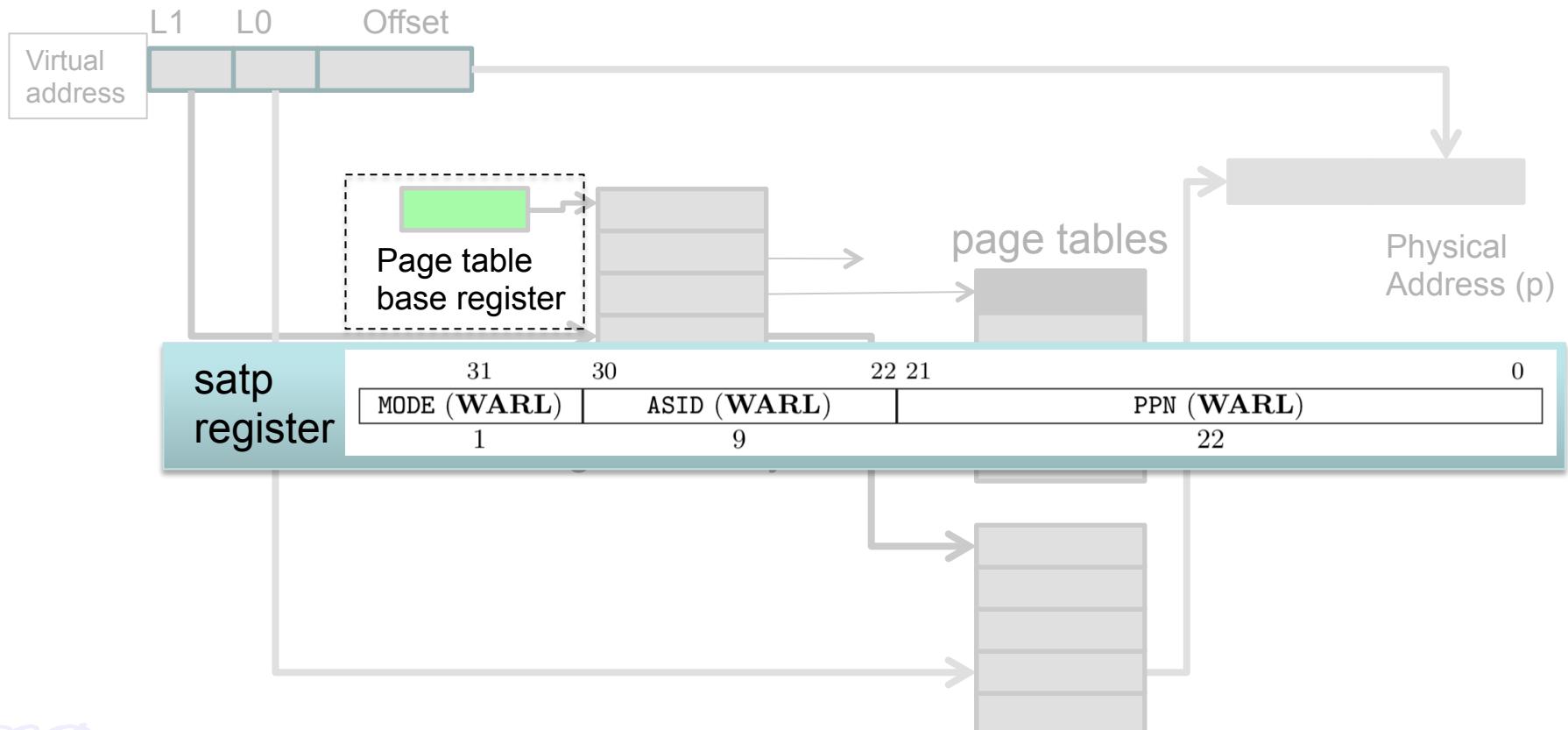


CR

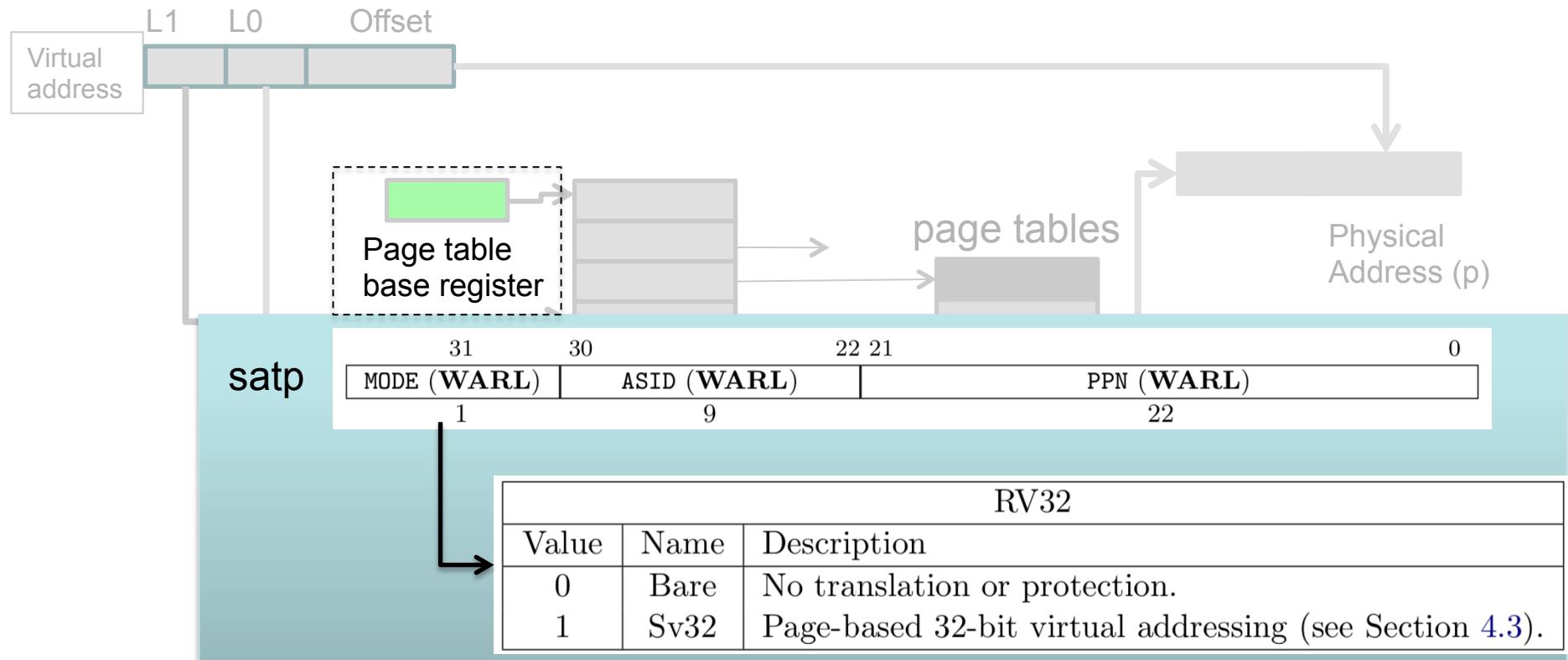
## 2 Level Page Translation



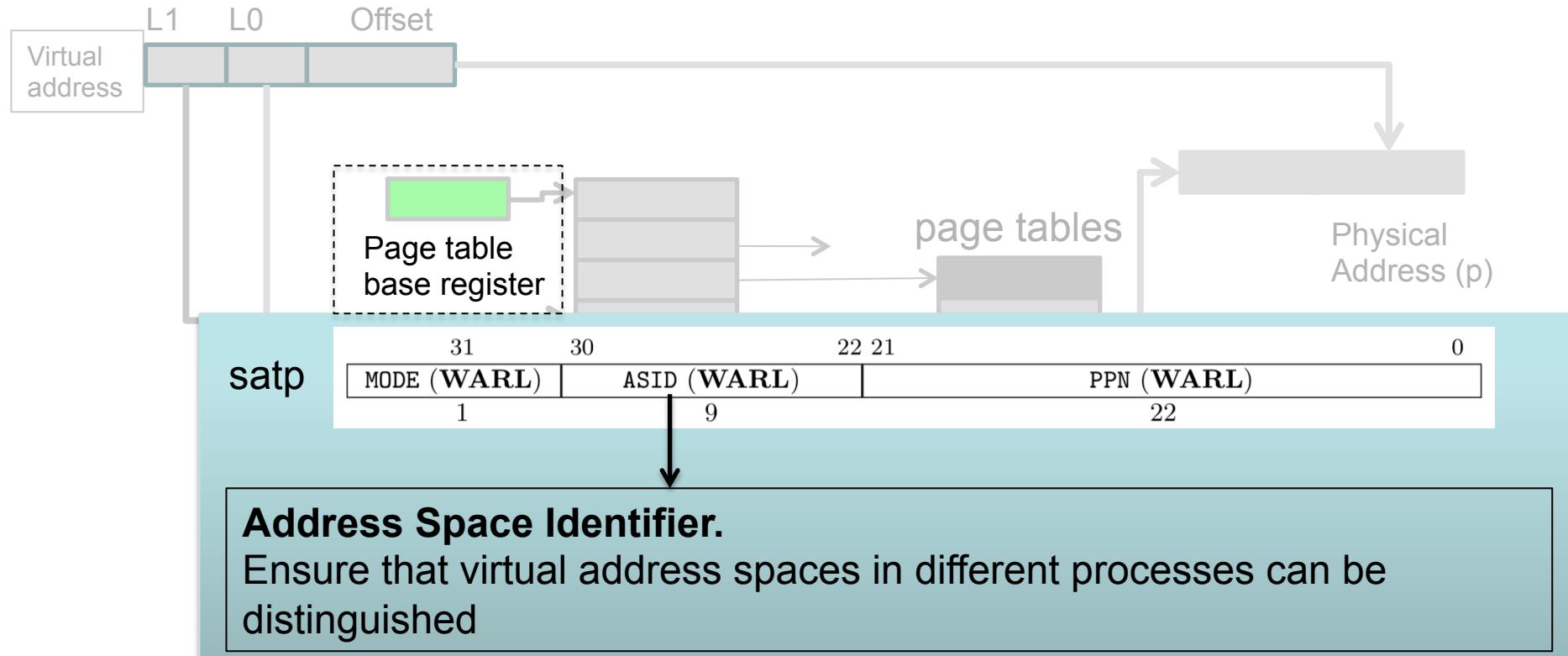
# 32-bit address translation in RISCV



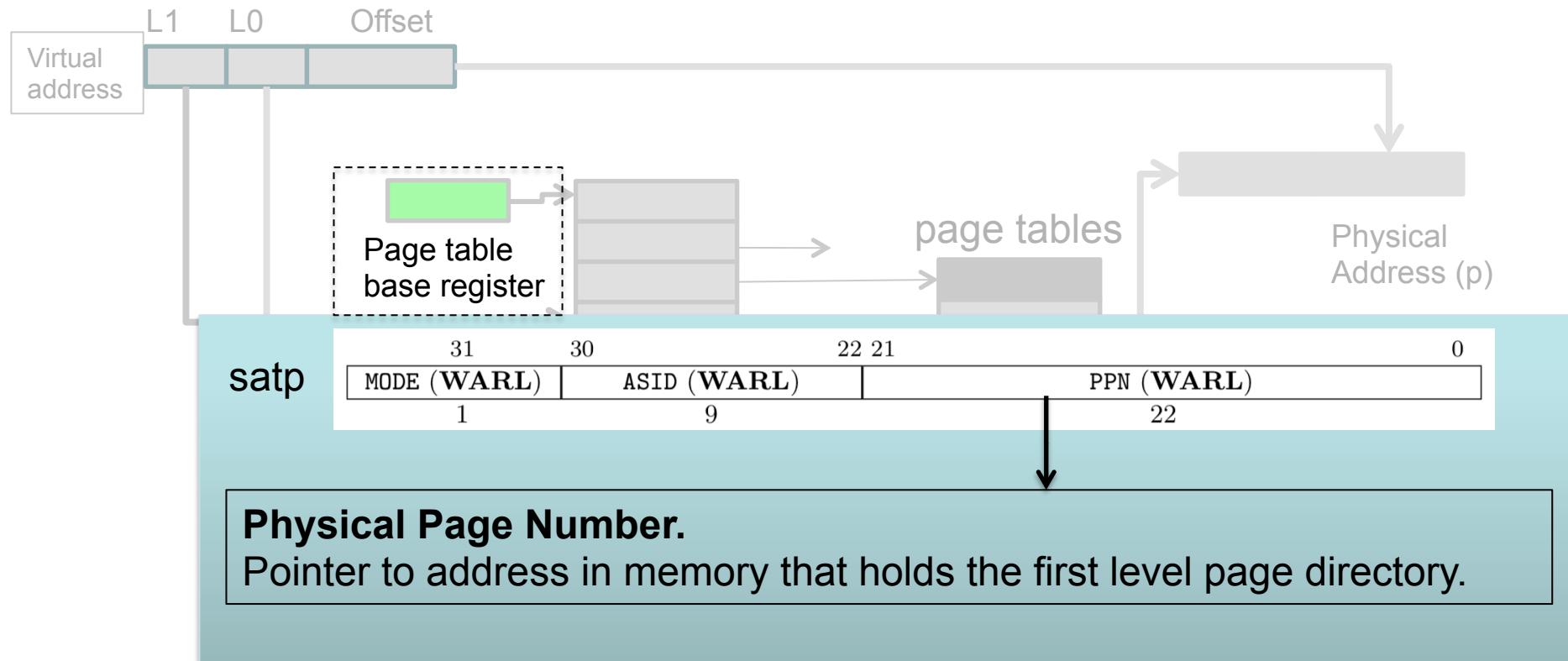
# 32-bit address translation in RISCV



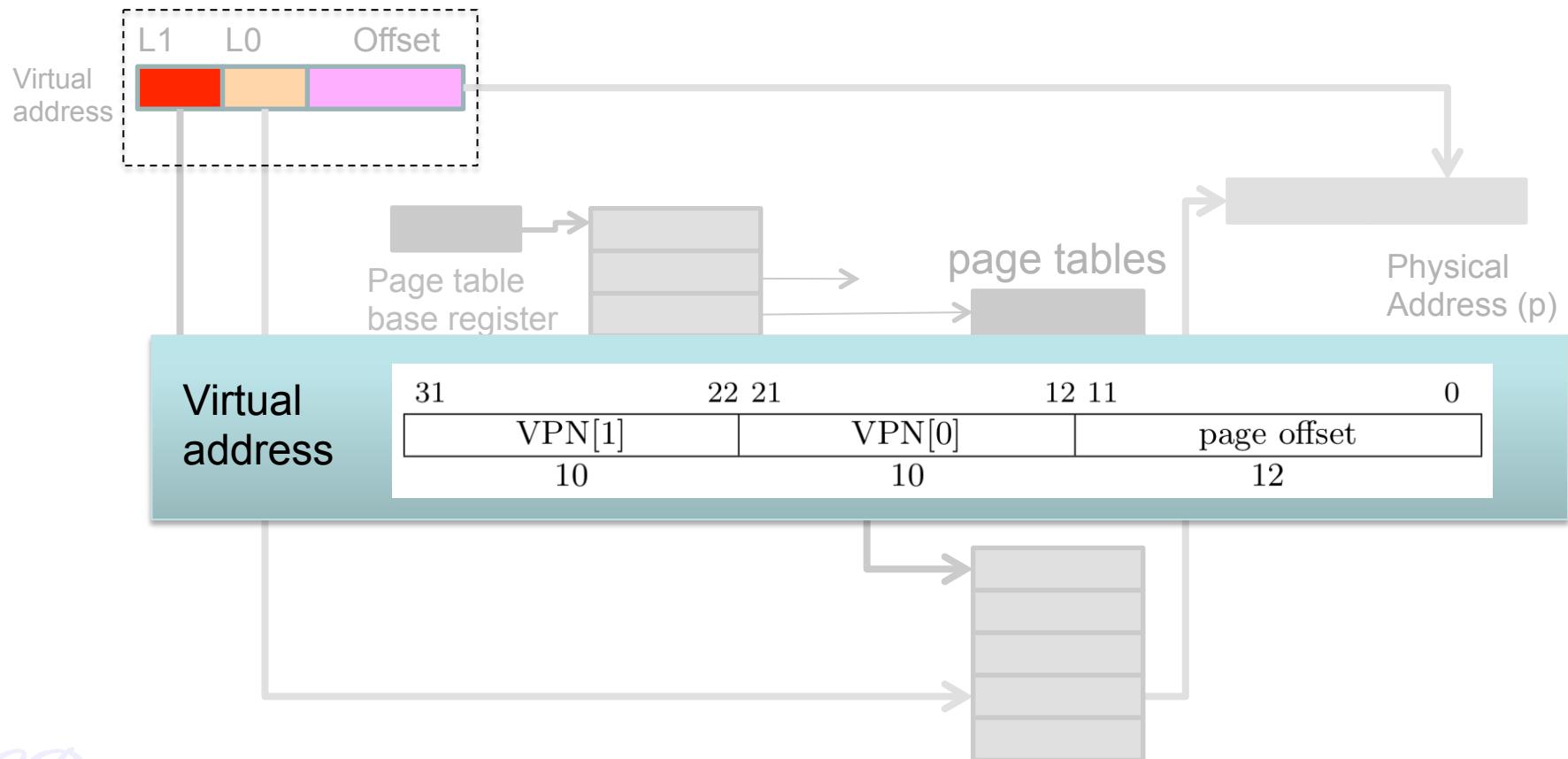
# 32-bit address translation in RISCV



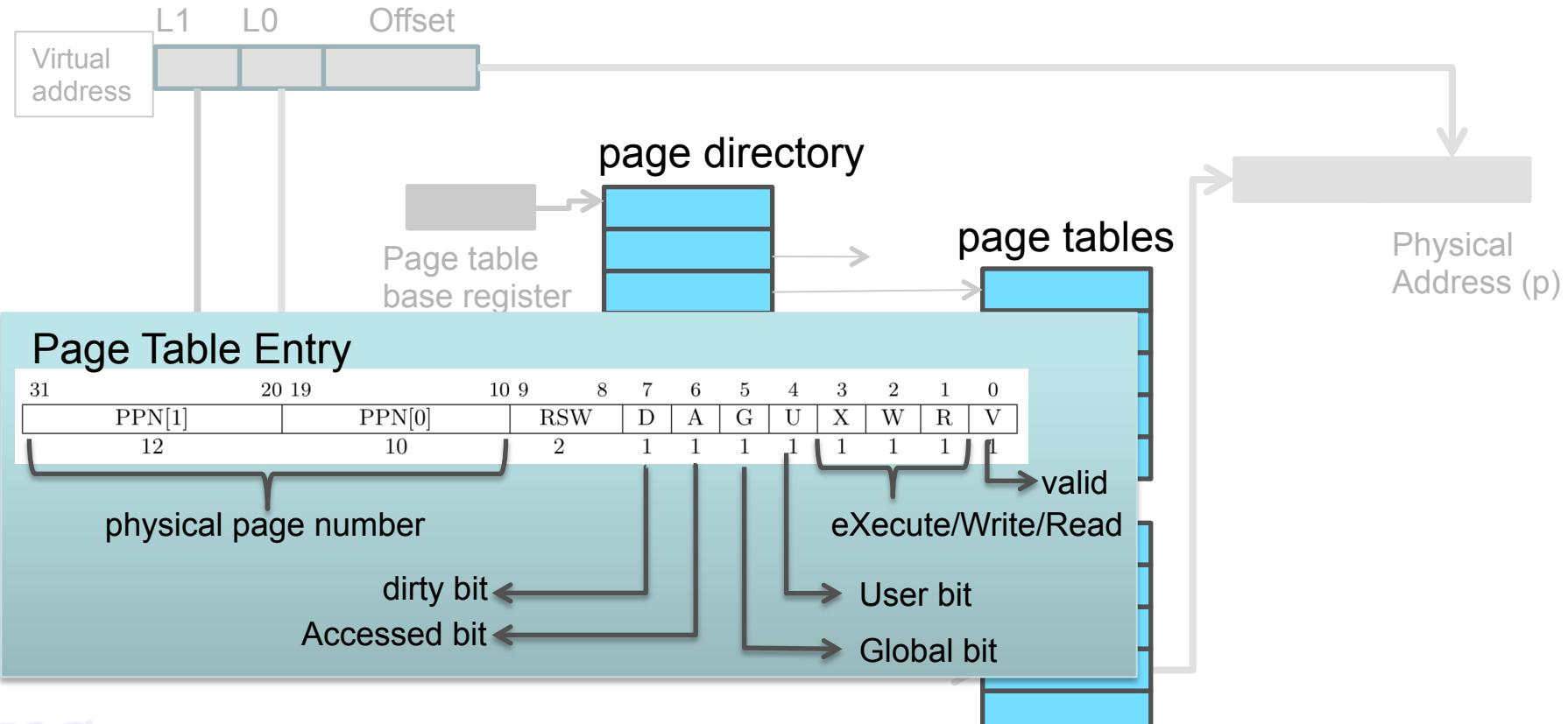
# 32-bit address translation in RISCV



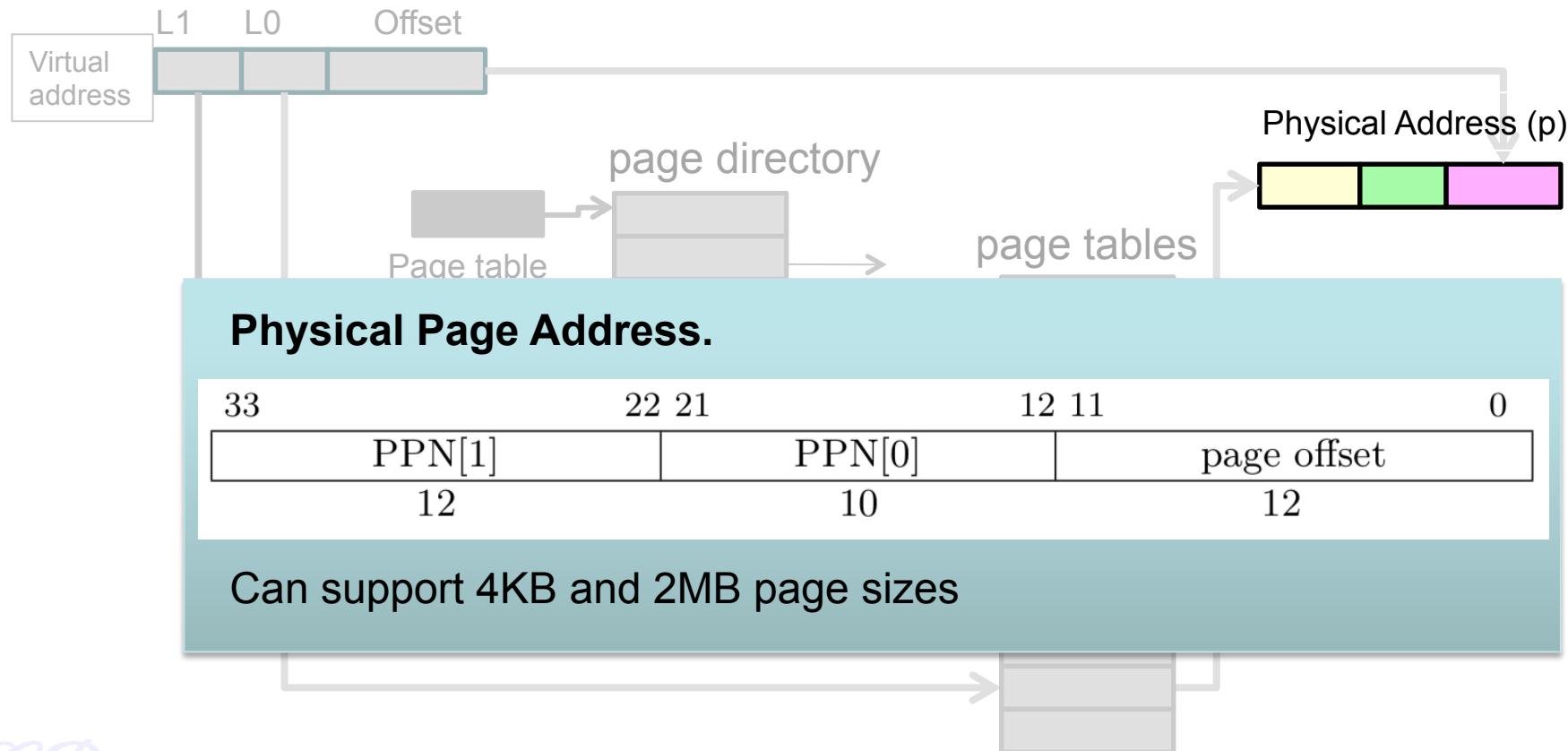
# 32-bit address translation in RISCV



# 32-bit address translation in RISCV

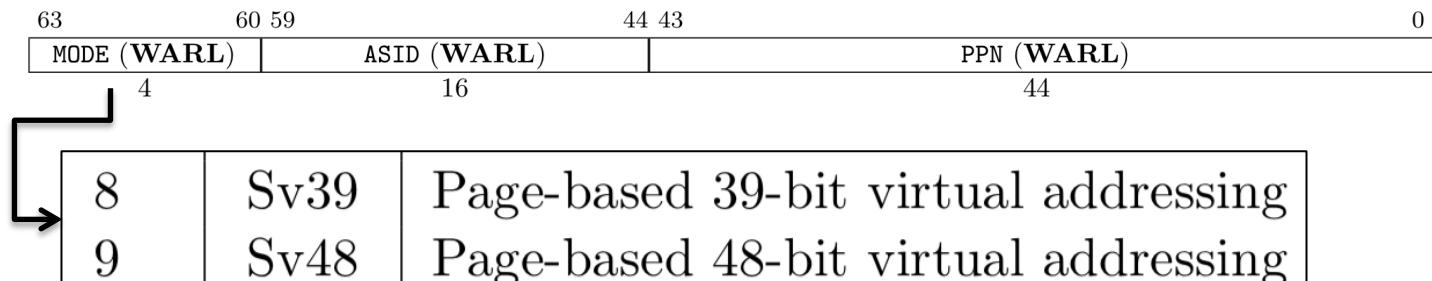


# 32-bit address translation in RISCV



# RISC V 64

- Supports 39 bit and 48 bit addressing

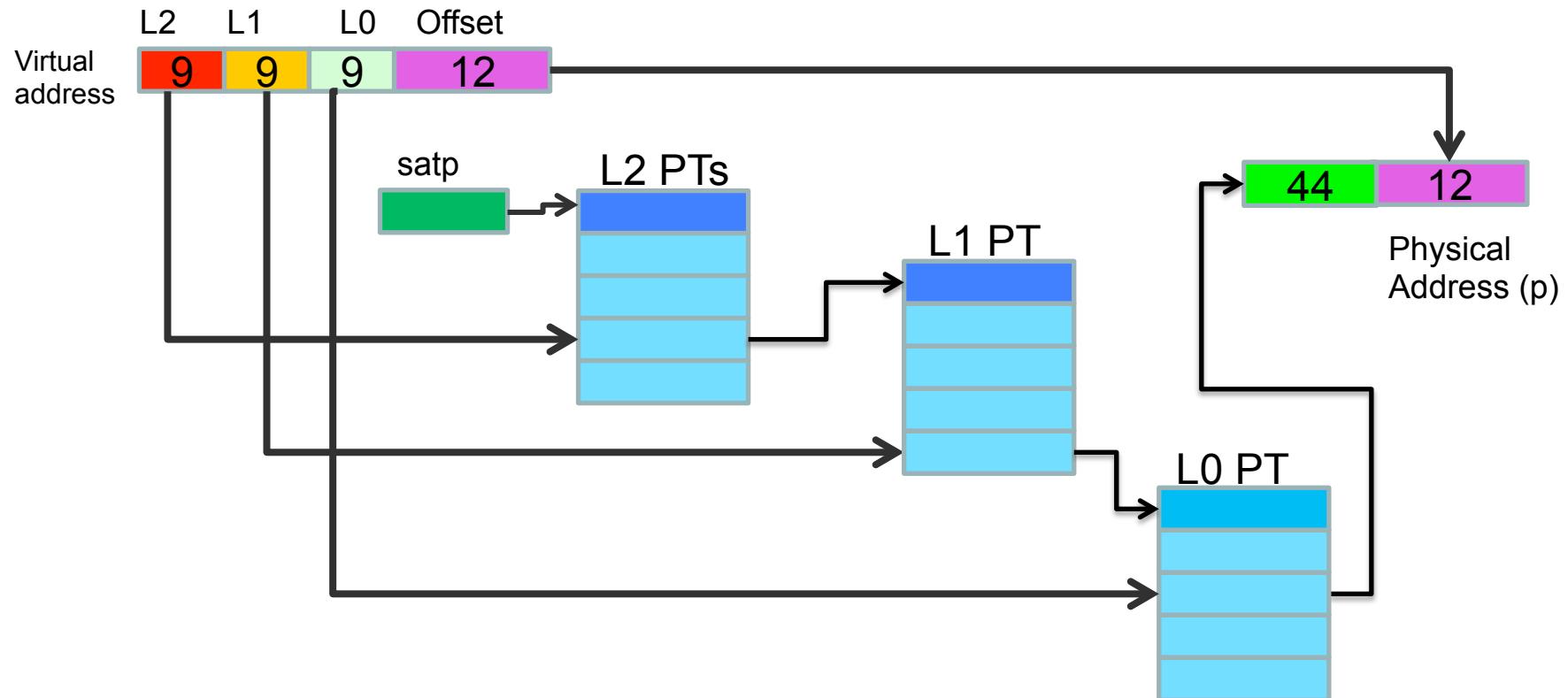


satp register

xv6 uses 39-bit addressing (riscv.h)

```
184 // use riscv's sv39 page table scheme.  
185 #define SATP_SV39 (8L << 60)  
186  
187 #define MAKE_SATP(pagetable) (SATP_SV39 | (((uint64)pagetable) >> 12))
```

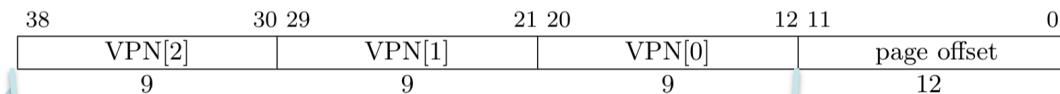
# 3 Level Page Translation



# 39-bit addressing in RISC V 64

- Three levels of page tables (can support 4KB, 2MB, 1GB pages)

## Virtual address



64 bit virtual address  
(most significant 25 bits  
unused)

$2^{27}$  page table  
entries

# 39-bit addressing in RISC V 64

- Three levels of page tables (can support 4KB, 2MB, 1GB pages)

## Virtual address

38	30 29	21 20	12 11	0
VPN[2]	VPN[1]	VPN[0]	page offset	
9	9	9	12	

## Page table entry

63	54 53	28 27	19 18	10 9	8	7	6	5	4	3	2	1	0
Reserved	PPN[2]	PPN[1]	PPN[0]	RSW	D	A	G	U	X	W	R	V	
10	26	9	9	2	1	1	1	1	1	1	1	1	

44-bit physical  
address

# PTE in xv6

## Page Table Entry

63	54 53	28 27	19 18	10 9	8	7	6	5	4	3	2	1	0
<i>Reserved</i>	PPN[2]	PPN[1]	PPN[0]	RSW	D	A	G	U	X	W	R	V	
10	26	9	9	2	1	1	1	1	1	1	1	1	

riscv.h

```
330 #define PTE_V (1L << 0) // valid
331 #define PTE_R (1L << 1)
332 #define PTE_W (1L << 2)
333 #define PTE_X (1L << 3)
334 #define PTE_U (1L << 4) // 1 -> user can access
335
336 // shift a physical address to the right place for a PTE.
337 #define PA2PTE(pa) (((uint64)pa) >> 12) << 10)
```

# 39-bit addressing in RISC V 64

- Three levels of page tables (can support 4KB, 2MB, 1GB pages)

## Virtual address

38 VPN[2]	30 29 9	21 20 9	12 11 9	0 page offset 12
--------------	------------	------------	------------	------------------------

## Page table entry

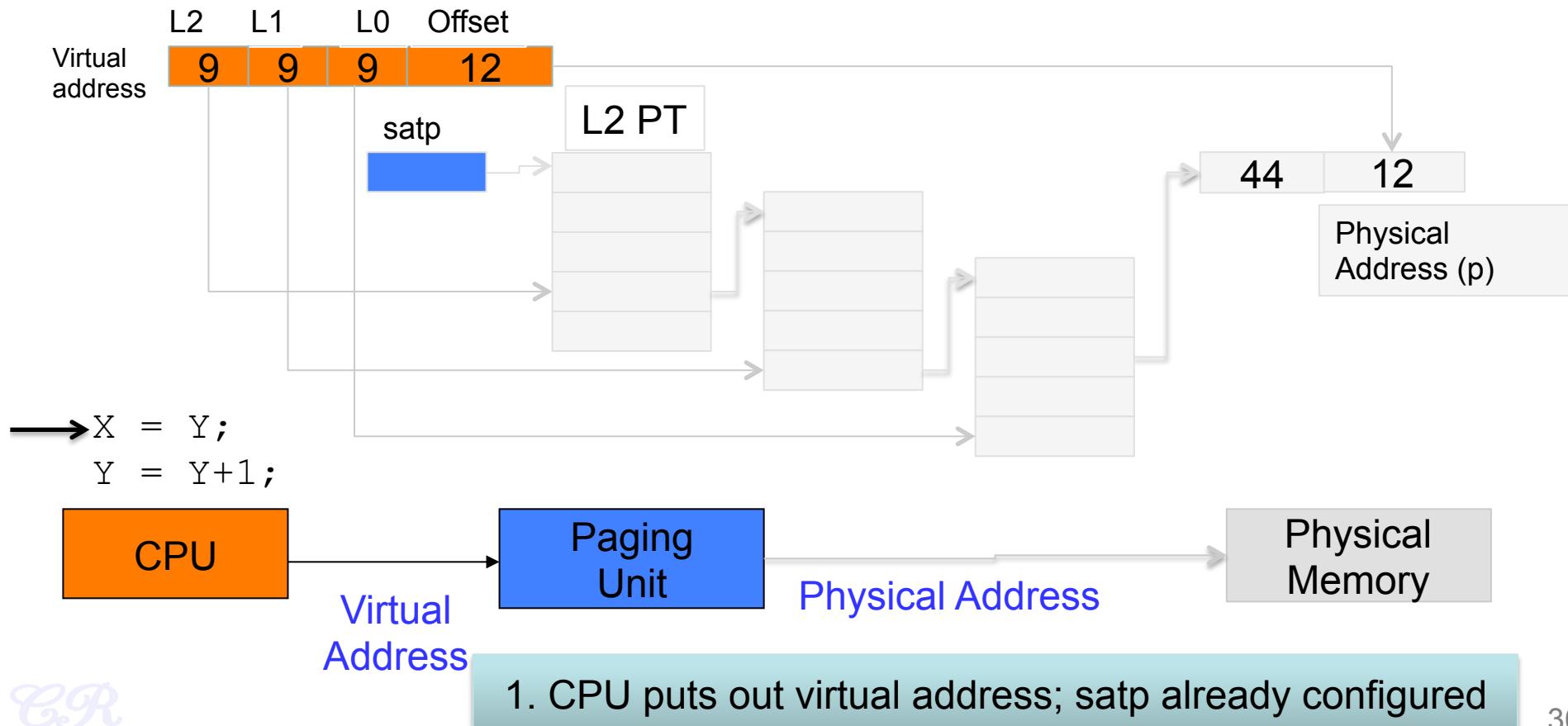
63 Reserved 10	54 53 PPN[2] 26	28 27 PPN[1] 9	19 18 PPN[0] 9	5 G 1	4 U 1	3 X 1	2 W 1	1 R 1	0 V 1
----------------------	-----------------------	----------------------	----------------------	-------------	-------------	-------------	-------------	-------------	-------------

44-bit physical address from PTE + 12 bits offset

## Physical address

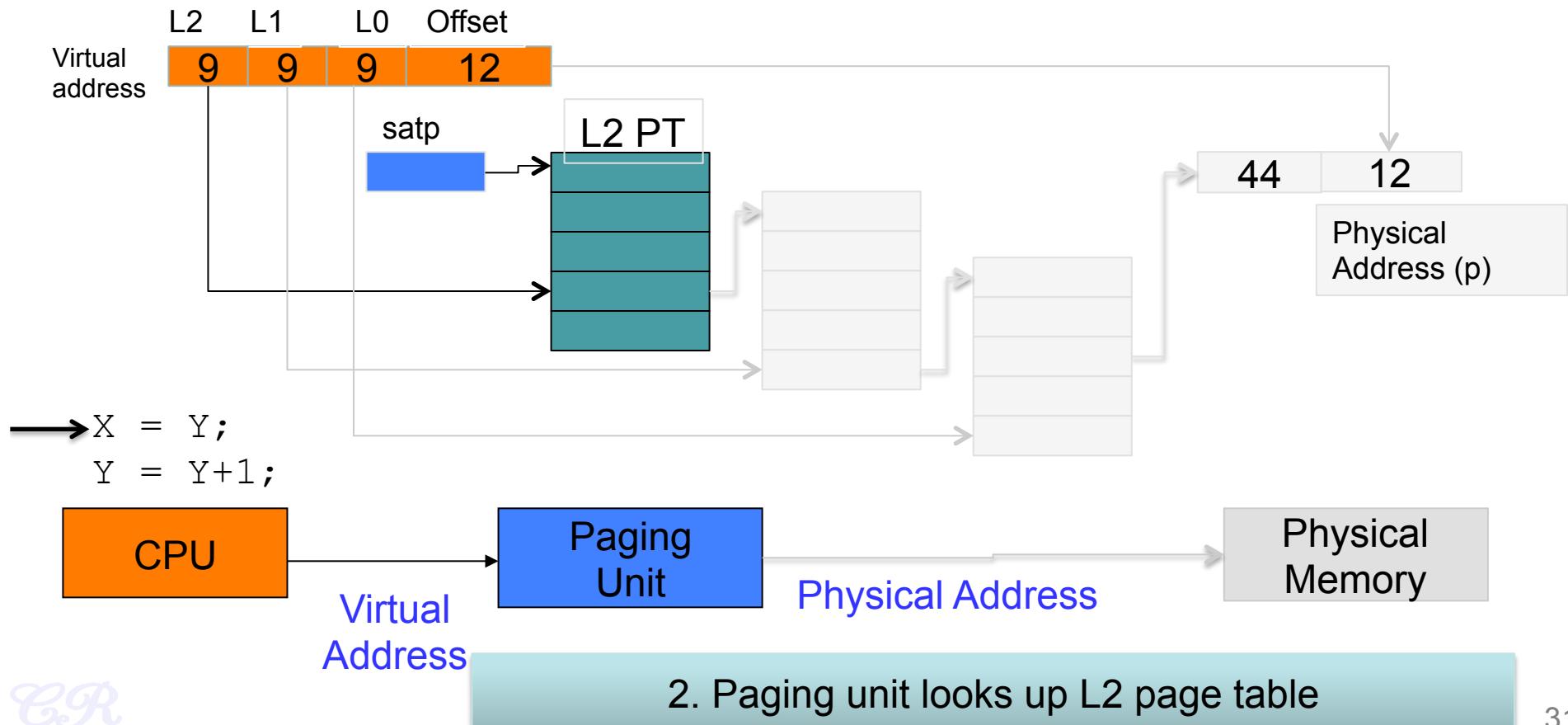
55 PPN[2] 26	30 29 9	21 20 9	12 11 9	0 page offset 12
--------------------	------------	------------	------------	------------------------

# 3 Level Page Translation



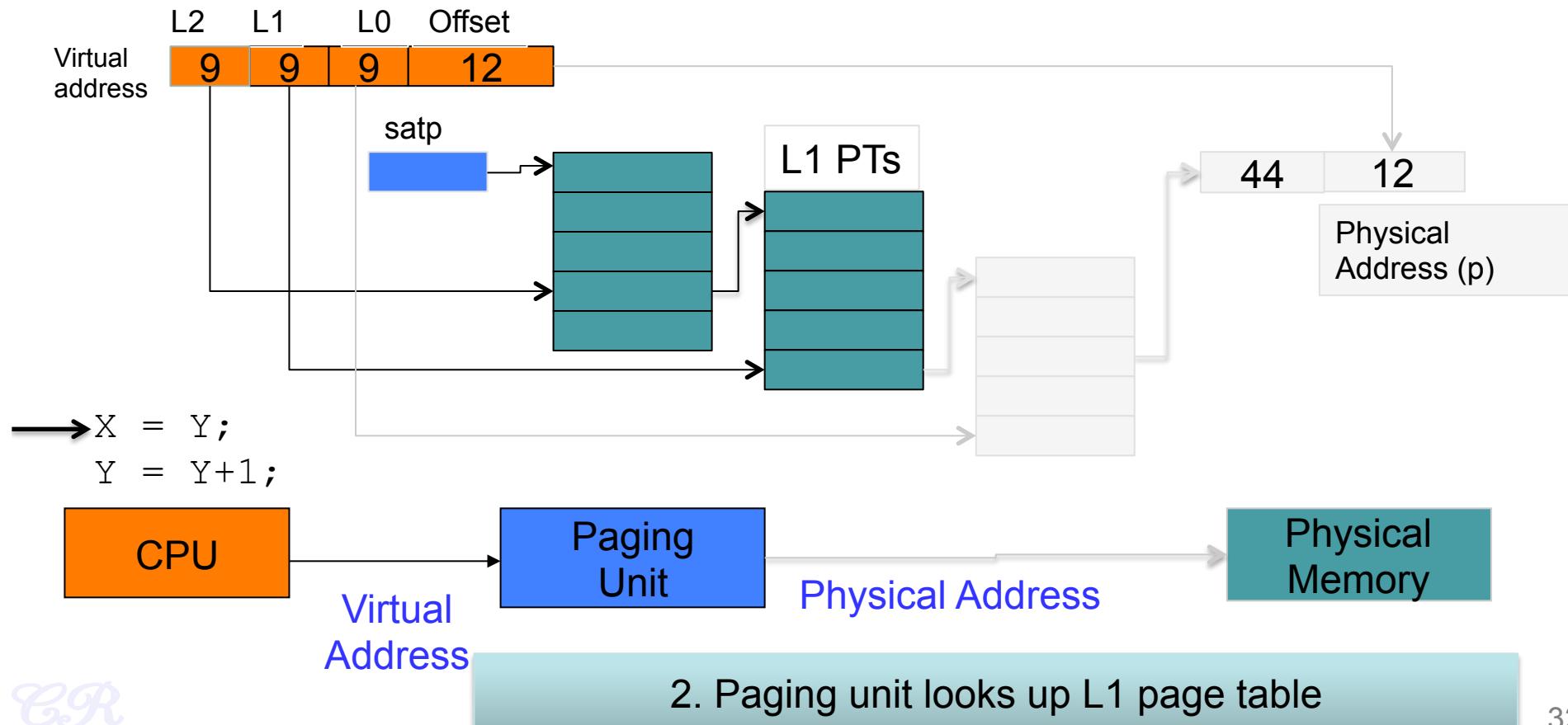
CR

# 3 Level Page Translation

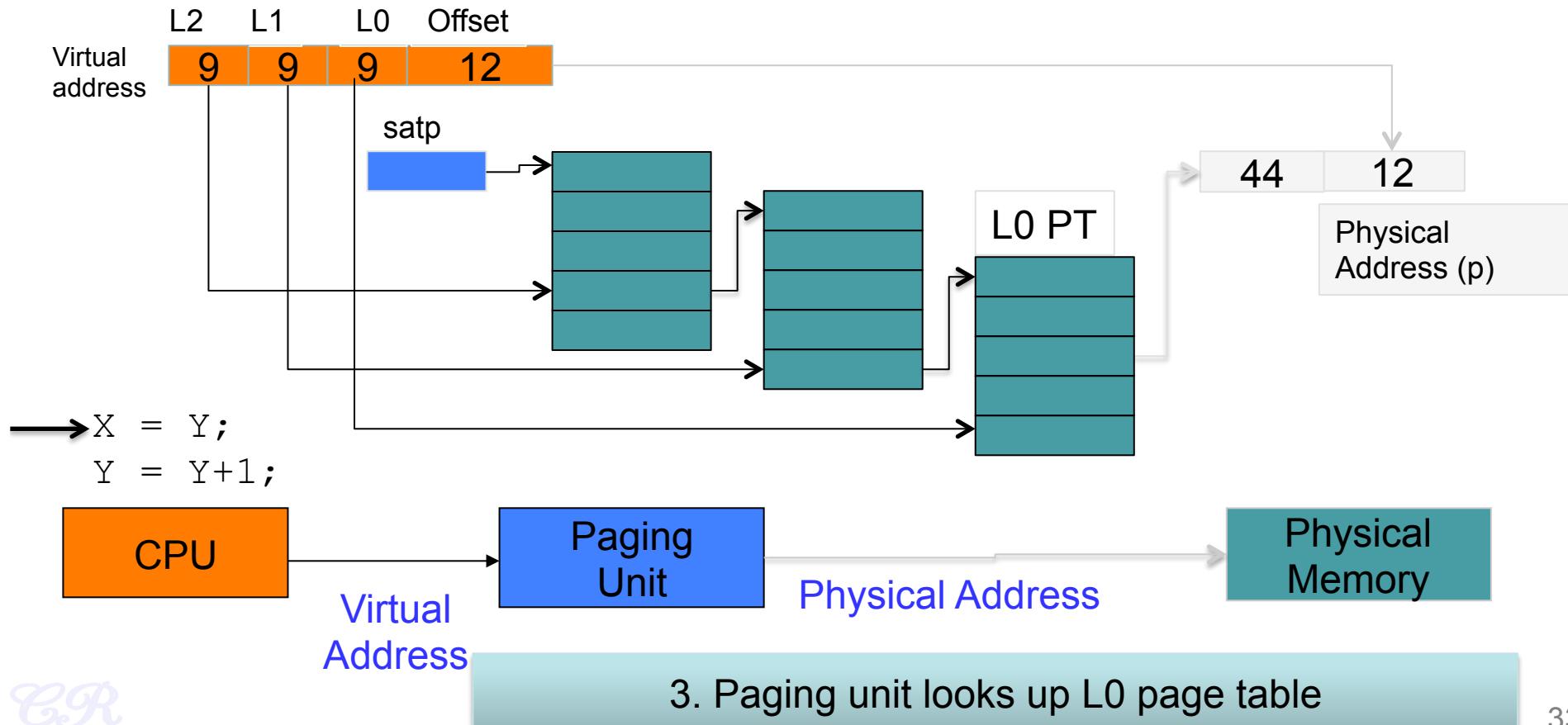


CR

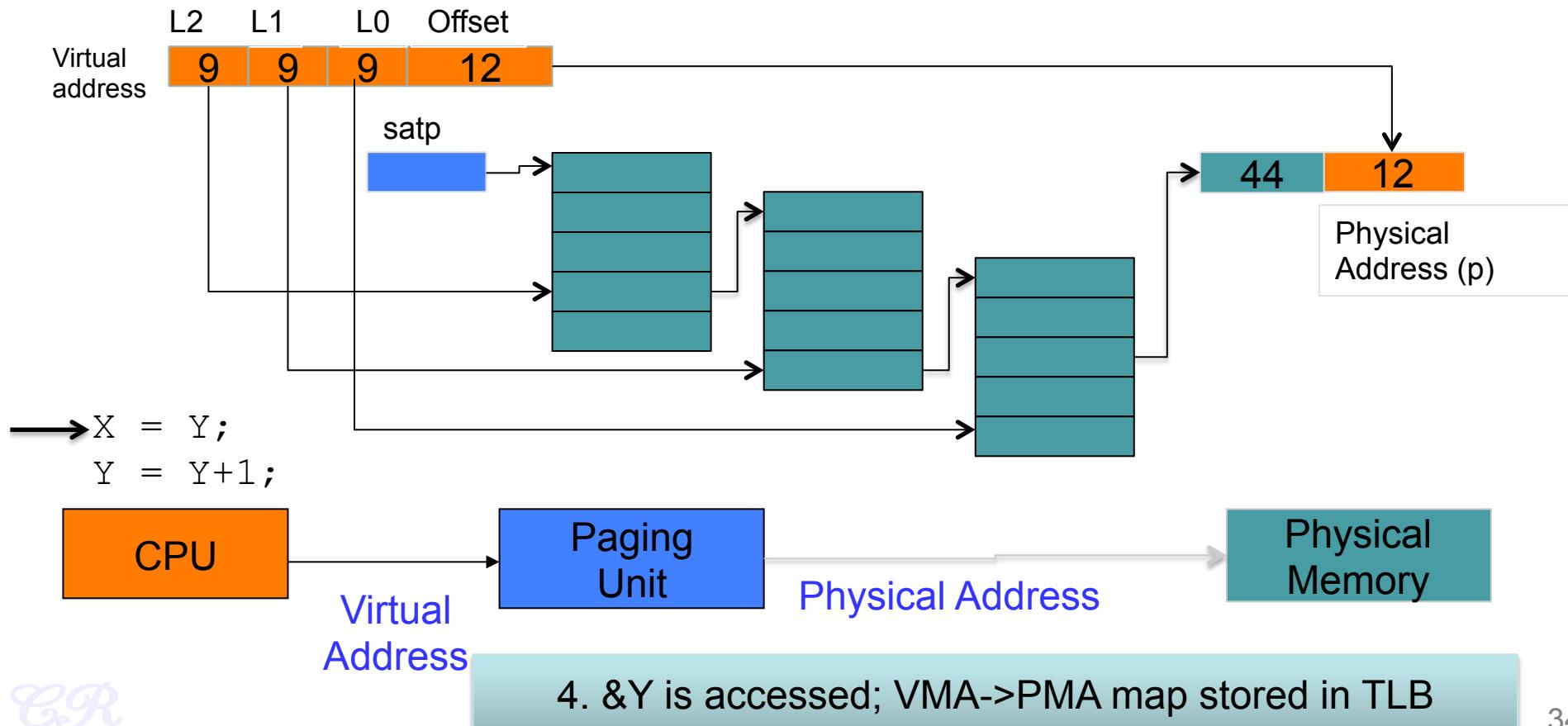
# 3 Level Page Translation



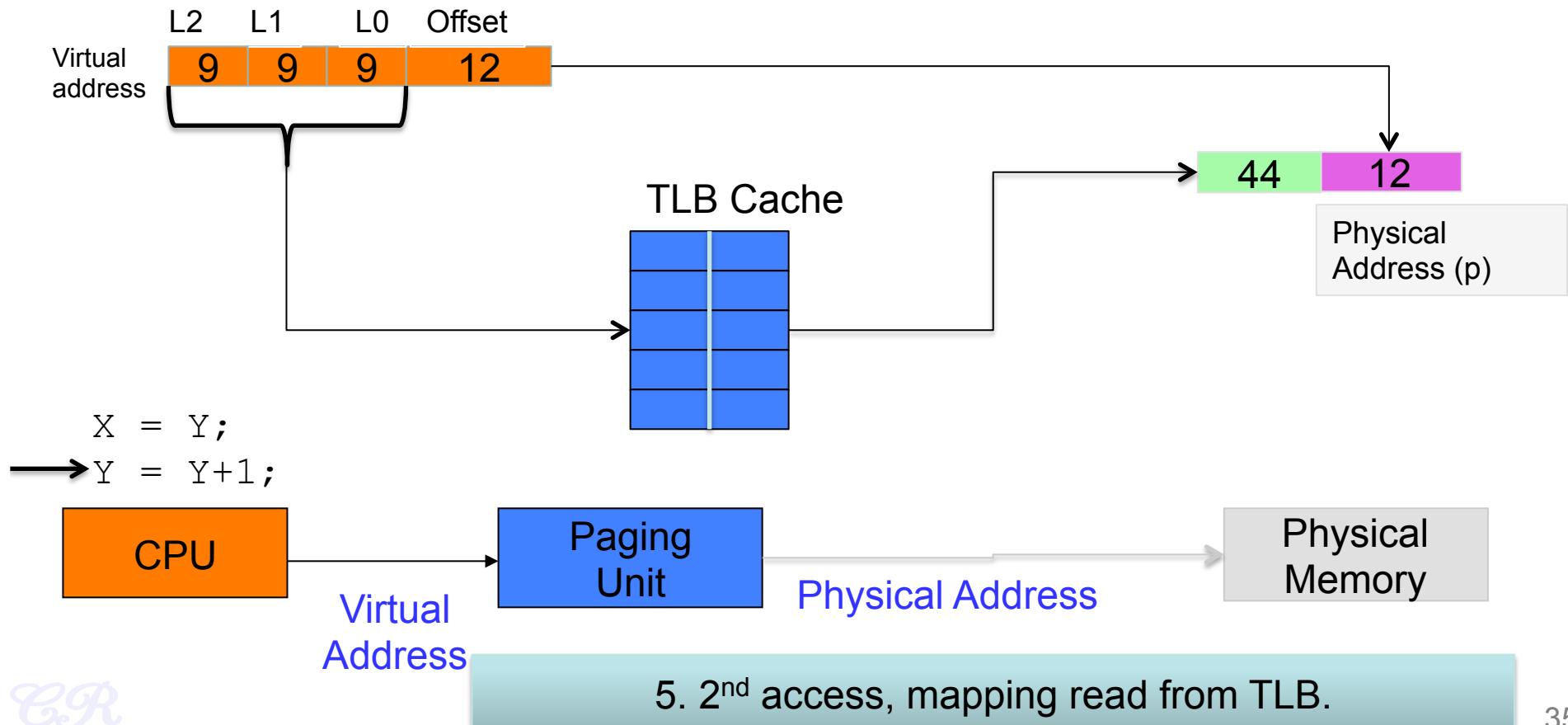
# 3 Level Page Translation



# 3 Level Page Translation

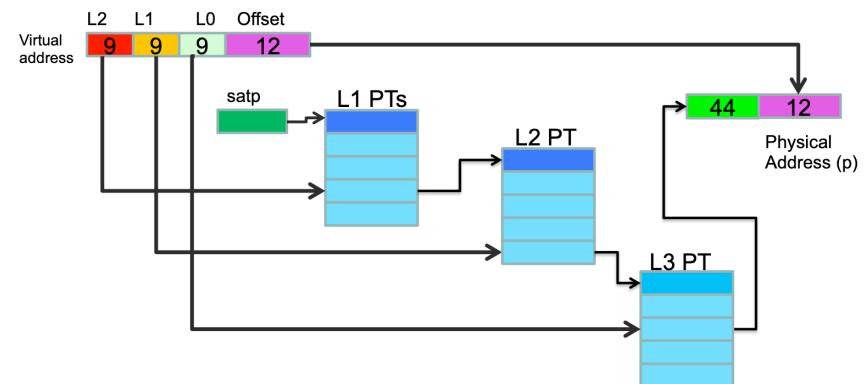
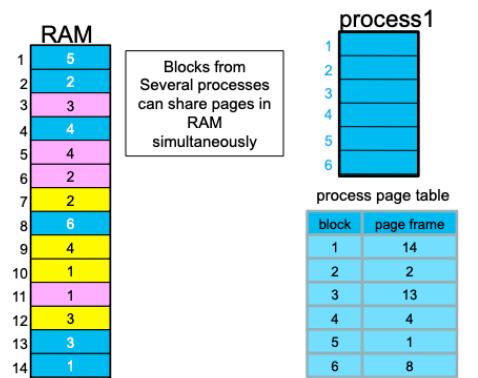


# 3 Level Page Translation



# Ponder About

Every process has its own Virtual Address Space.  
Every Virtual Address Space can have  $2^{27}$  page table entries  
Every Page table entry is 8 bytes. Total space  $2^{29}$  !!!



Do we need so much memory just for page tables?

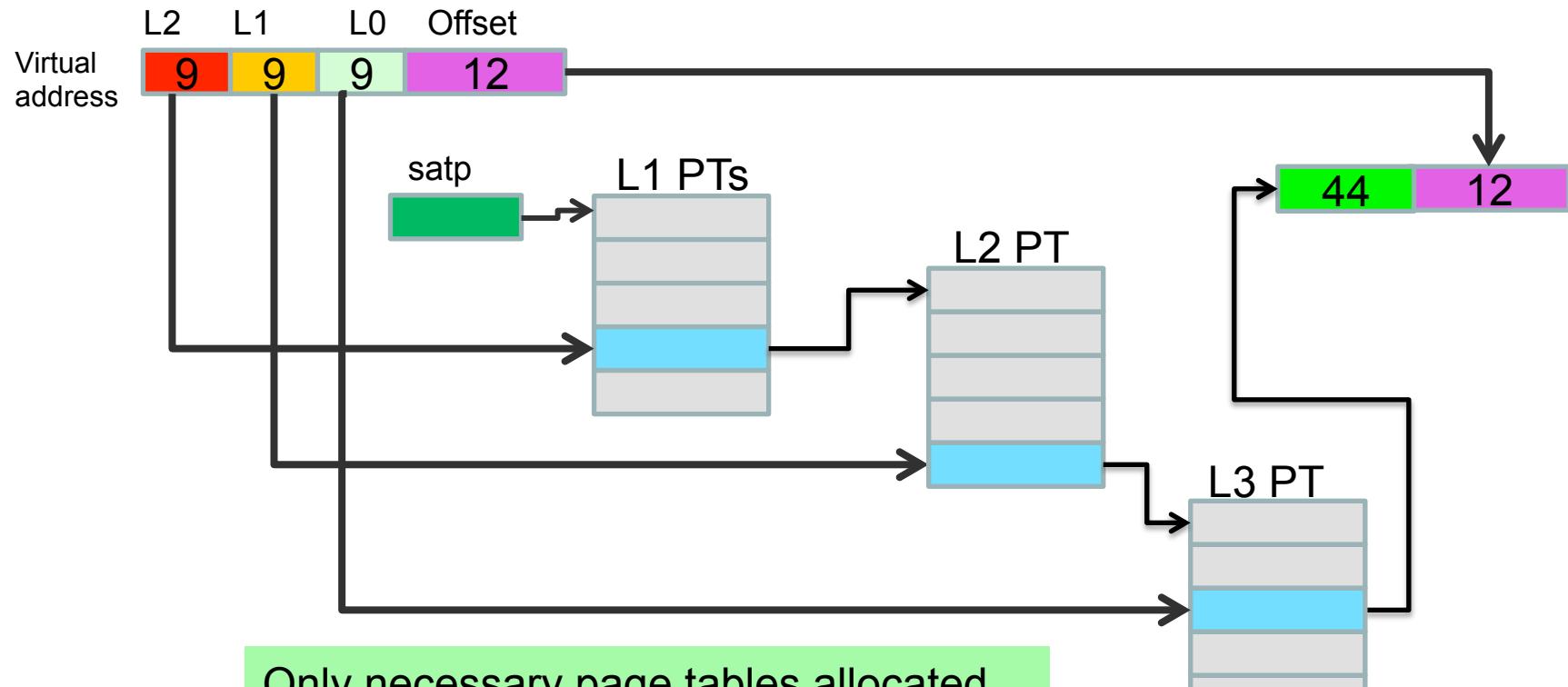
CR

# Not entire virtual address space used



CR

# On Demand Page Table Entries



# Creating Page Entries in xv6

vm.c

```
73 static pte_t *
74 walk(pagetable_t pagetable, uint64 va, int alloc)
75 {
76     if(va >= MAXVA)
77         panic("walk");
78
79     for(int level = 2; level > 0; level--) {
80         pte_t *pte = &pagetable[PX(level, va)];
81         if(*pte & PTE_V) {
82             pagetable = (pagetable_t)PTE2PA(*pte);
83         } else {
84             if(!alloc || (pagetable = (pde_t*)kalloc()) == 0)
85                 return 0;
86             memset(pagetable, 0, PGSIZE);
87             *pte = PA2PTE(pagetable) | PTE_V;
88         }
89     }
90     return &pagetable[PX(0, va)];
91 }
```

- 64 bit pointer to L2 page table
- PTE for this virtual address
- If alloc=1 and then on demand allocate
- If va is above  $2^{39}-1$  then something is wrong, stop kernel

# On Demand Page Entries in xv6

VM.C

```

73 static pte_t *
74 walk(pagetable_t pageta) {
75 {
76     if(va >= MAXVA)
77         panic("walk");
78
79     for(int level = 2; level > 0; level--) {
80         pte_t *pte = &pagetable[PX(level, va)];
81         if(*pte & PTE_V) {
82             pagetable = (pagetable_t)PTE2PA(*pte);
83         } else {
84             if(!alloc || (pagetable = (pde_t*)kalloc()) == 0)
85                 return 0;
86             memset(pagetable, 0, PGSIZE);
87             *pte = PA2PTE(pagetable) | PTE_V;
88         }
89     }
90     return &pagetable[PX(0, va)];
91 }
```

Virtual address

	PPN[2]	PPN[1]	PPN[0]	page offset
30 29	26	9	9	12

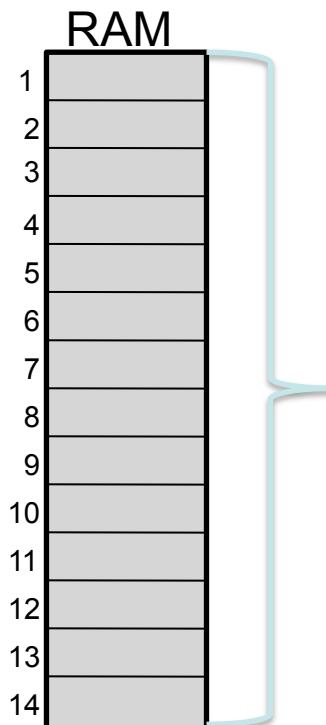
If valid bit is set, then extract physical address from PTE

If alloc=1, create a new page table with kalloc and set all bits to 0

Fill in the page table entry and set the valid bit.

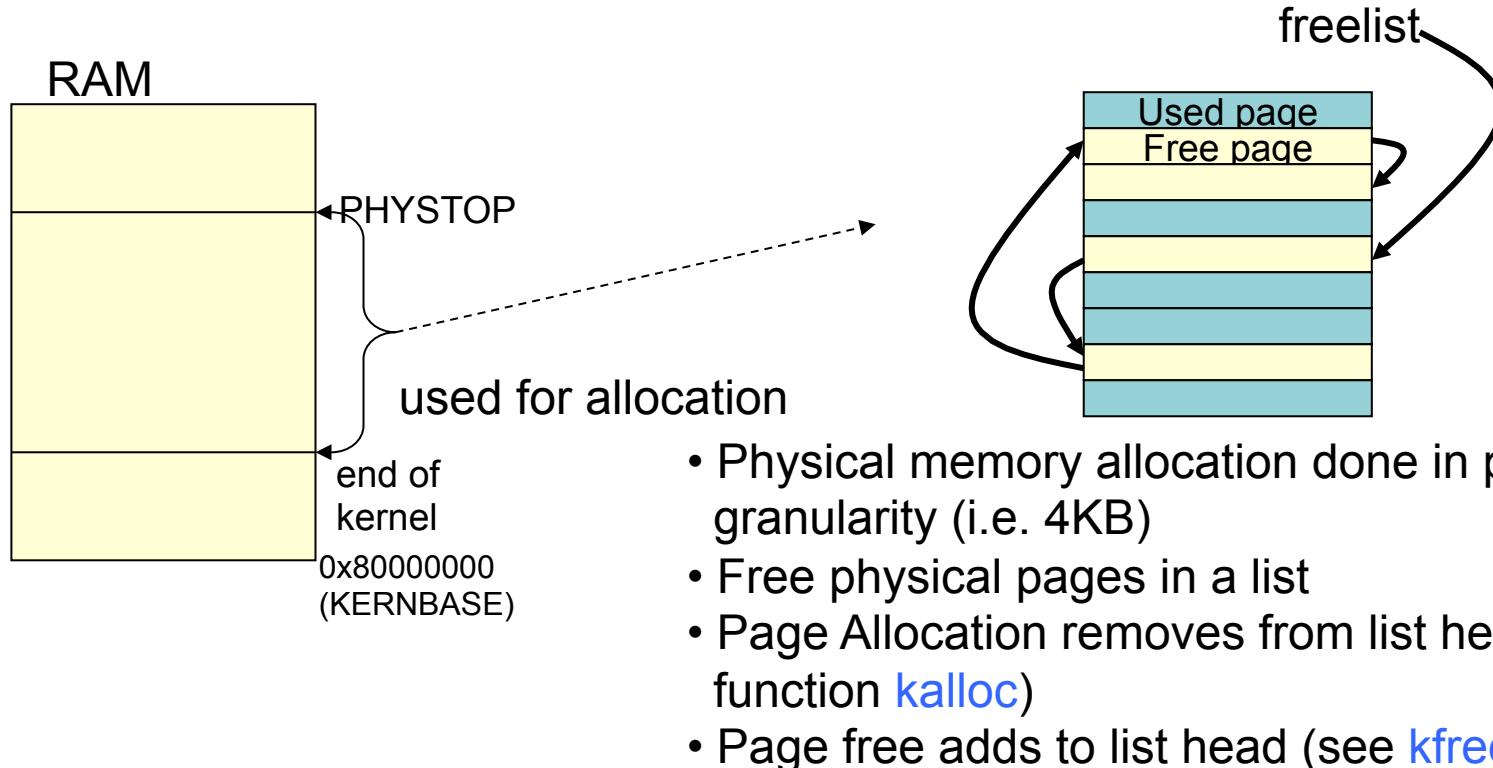
CR

# Managing Page frames in RAM



OS should be able to  
manage the free page frames.

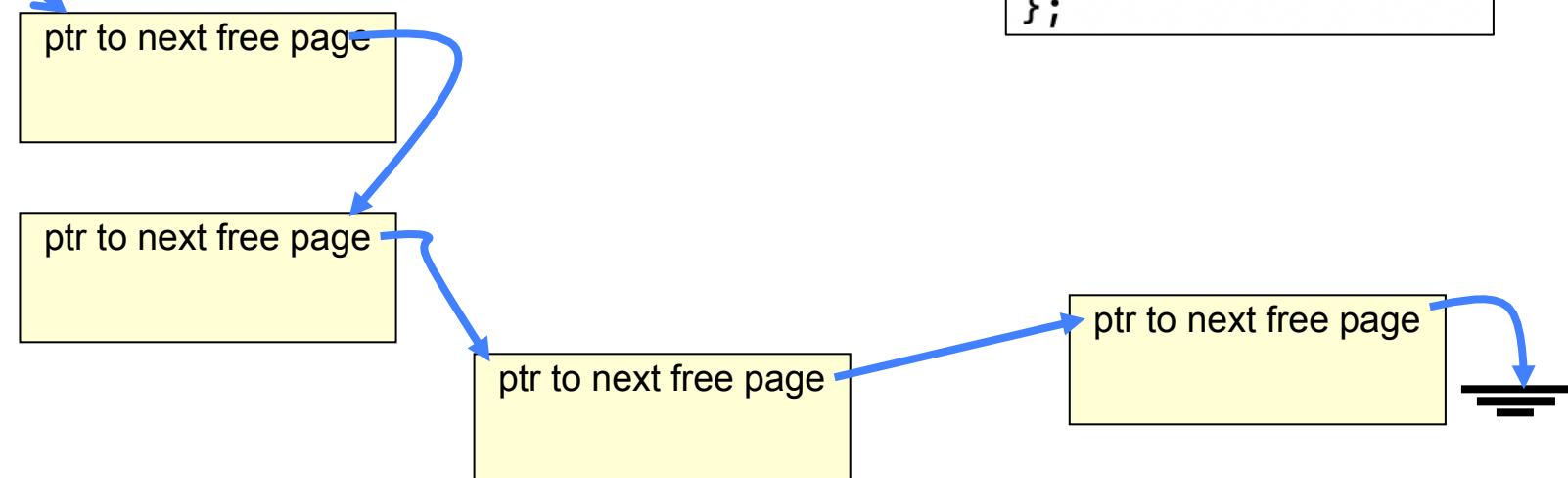
# Physical Page Allocation (kinit)



# Freelist Implementation

- How is the freelist implemented?
  - No exclusive memory to store links
  - Uses two functions: *kfree* and *kalloc*

freelist



# kinit

## kalloc.c

```
26 void
27 kinit() —————→ Called from main
28 {
29     initlock(&kmem.lock, "kmem");
30     freerange(end, (void*)PHYSTOP);
31 } —————→ end of RAM
32 —————→ end of kernel
33 void
34 freerange(void *pa_start, void *pa_end)
35 {
36     char *p;
37     p = (char*)PGROUNDUP((uint64)pa_start);
38     for(; p + PGSIZE <= (char*)pa_end; p += PGSIZE)
39         kfree(p);
40 }
```

Add each chunk of 4KB physical memory to the freelist

# kfree

kalloc.c

```
46 void
47 kfree(void *pa)
48 {
49     struct run *r;
50
51     if(((uint64)pa % PGSIZE) != 0) || ((char*)pa < end || (uint64)pa >= PHYSTOP)
52         panic("kfree");
53
54     // Fill with junk to catch dangling refs.
55     memset(pa, 1, PGSIZE);
56
57     r = (struct run*)pa;
58
59     acquire(&kmem.lock);-----
60     r->next = kmem.freelist;
61     kmem.freelist = r;
62     release(&kmem.lock);-----
63 }
```

kalloc and free only work at page level granularity

Only valid physical memory can be added to the freelist

Add page to the list

```
struct {
    struct spinlock lock;
    struct run *freelist;
} kmem;
```

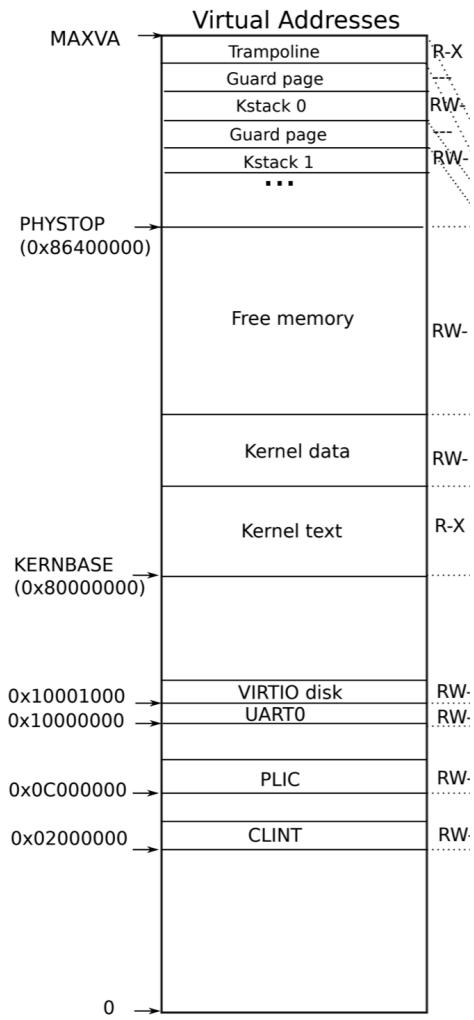
# kalloc

## kalloc.c

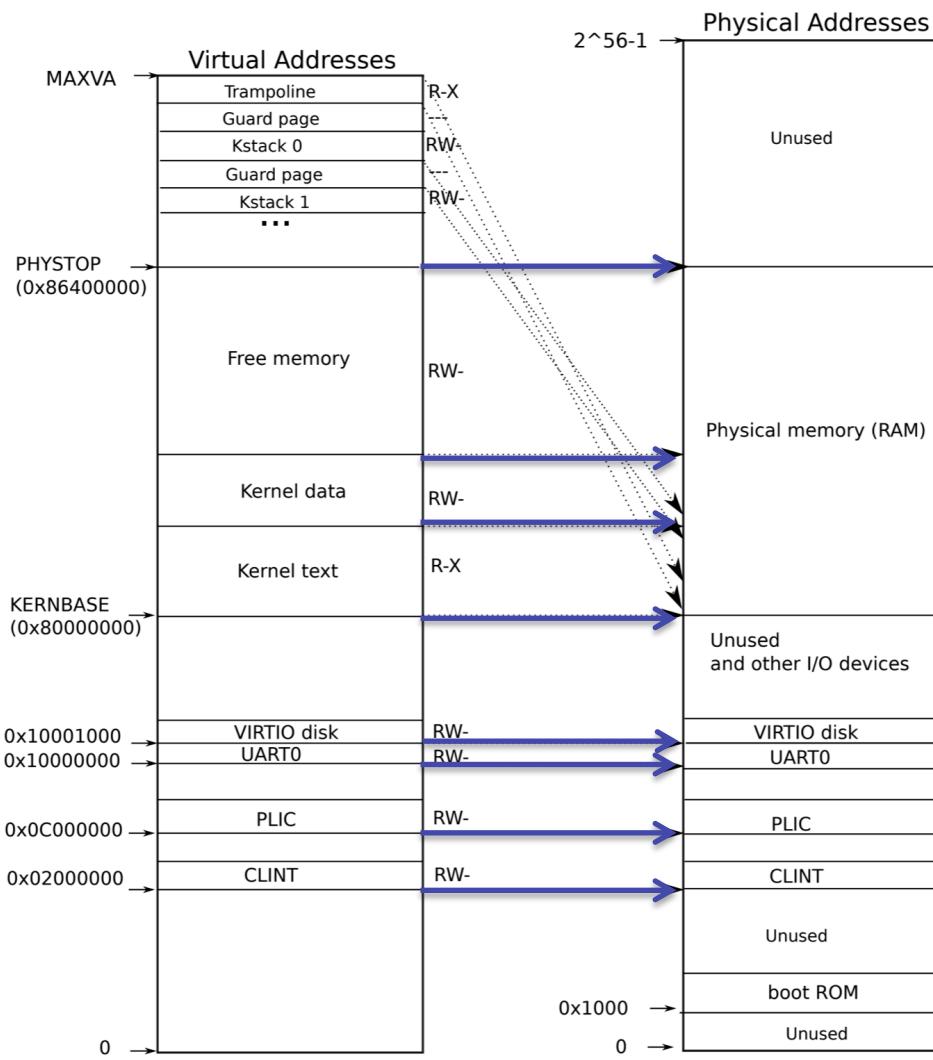
```
68 void *  
69 kalloc(void)  
70 {  
71     struct run *r;  
72  
73     acquire(&kmem.lock);-----  
74     r = kmem.freelist;  
75     if(r)  
76         kmem.freelist = r->next;  
77     release(&kmem.lock);-----  
78  
79     if(r)  
80         memset((char*)r, 5, PGSIZE); // fill with junk  
81     return (void*)r;  
82 }
```

Remove head of freelist

# Kernel Address Space

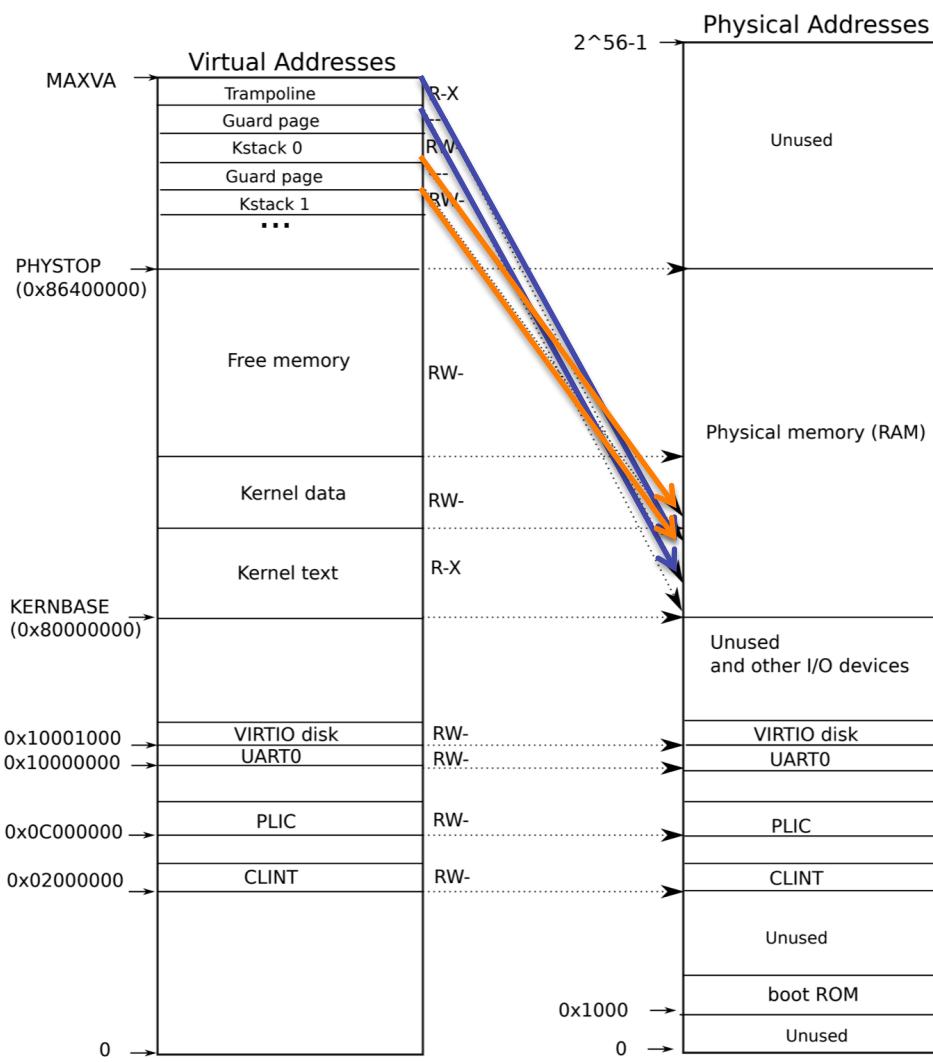


# Kernel Address Space



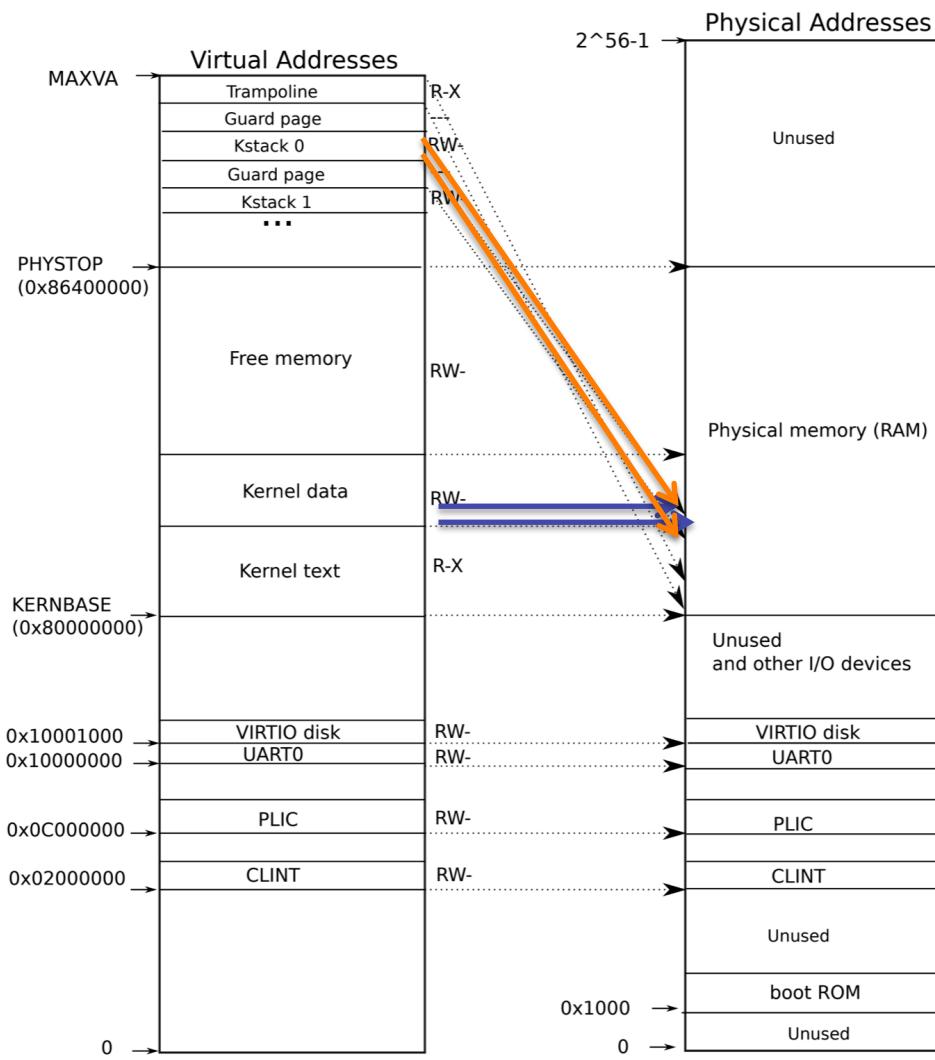
-1- the direct map for many of the devices and the RAM

# Kernel Address Space



-2- some regions don't have a direct map

# Kernel Address Space



-3- some regions (like the stack can be accessed from 2 virtual addresses)

One direct and the other indirect

# kvminit (vm.c)

```
-- ..
23 void
24 kvminit()
25 {
26     kernel_pagetable = (pagetable_t) kalloc();
27     memset(kernel_pagetable, 0, PGSIZE);
28
29     // uart registers
30     kvmmap(UART0, UART0, PGSIZE, PTE_R | PTE_W);
31
32     // virtio mmio disk interface
33     kvmmap(VIRTIO0, VIRTIO0, PGSIZE, PTE_R | PTE_W);
34
35     // CLINT
36     kvmmap(CLINT, CLINT, 0x10000, PTE_R | PTE_W);
37
38     // PLIC
39     kvmmap(PLIC, PLIC, 0x400000, PTE_R | PTE_W);
40
41     // map kernel text executable and read-only.
42     kvmmap(KERNBASE, KERNBASE, (uint64)etext-KERNBASE, PTE_R | PTE_X);
43
44     // map kernel data and the physical RAM we'll make use of.
45     kvmmap((uint64)etext, (uint64)etext, PHYSTOP-(uint64)etext, PTE_R | PTE_W);
46
47     // map the trampoline for trap entry/exit to
48     // the highest virtual address in the kernel.
49     kvmmap(TRAMPOLINE, (uint64)trampoline, PGSIZE, PTE_R | PTE_X);
50 }
```

This is going to form the page directory L1 page table. kernel\_pagetable is to be loaded in satp.

Map various devices to virtual memory  
kvmmap allocates L2 and L3 pages whenever required.

Map kernel space.  
Note code regions are Executable and read-only

# kvmmap

```
119 void  
120 kvmmap(uint64 va, uint64 pa, uint64 sz, int perm)  
121 {  
122     if(mappages(kernel_pagetable, va, sz, pa, perm) != 0)  
123         panic("kvmmap");  
124 }
```

```
150 int  
151 mappages(pagetable_t pagetable, uint64 va, uint64 size, uint64 pa, int perm)  
152 {  
153     uint64 a, last;  
154     pte_t *pte;  
155  
156     a = PGROUNDDOWN(va);  
157     last = PGROUNDDOWN(va + size - 1);  
158     for(;;){  
159         if((pte = walk(pagetable, a, 1)) == 0)  
160             return -1;  
161         if(*pte & PTE_V)  
162             panic("remap");  
163         *pte = PA2PTE(pa) | perm | PTE_V;  
164         if(a == last)  
165             break;  
166         a += PGSIZE;  
167         pa += PGSIZE;  
168     }  
169     return 0;  
170 }
```

Add page table entries for each page

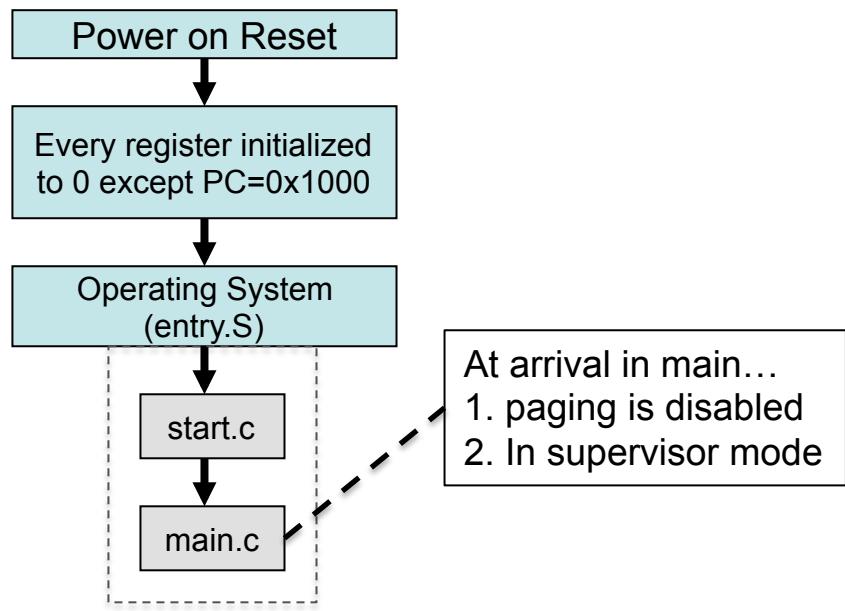
Create the page table entry (note the 1 in the 3<sup>rd</sup> argument)

Set the permission bits in PTE accordingly

CR

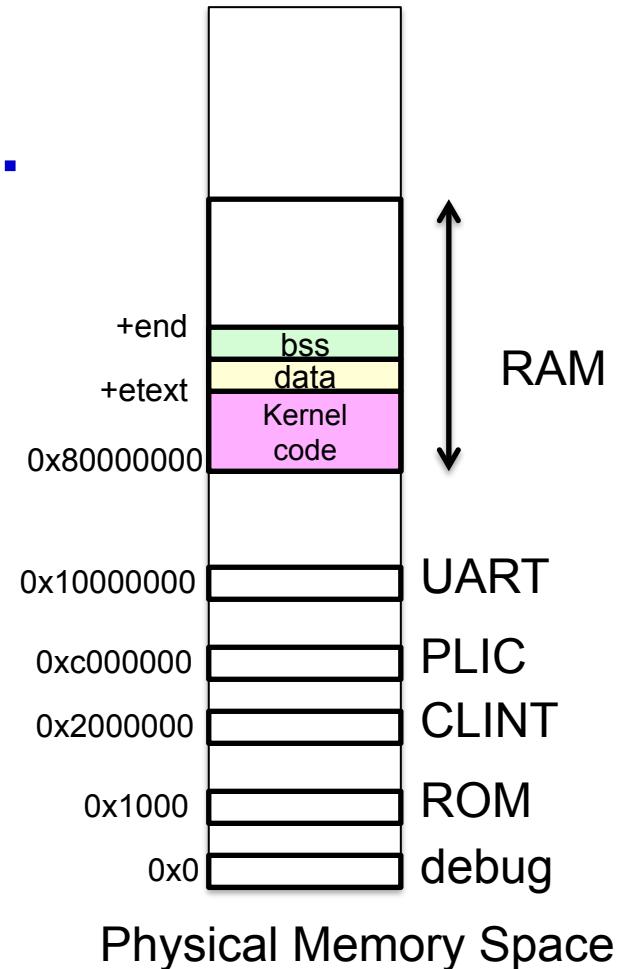
back to booting...

# so far...



Next to do: turn on paging

CR



```

10 void
11 main()
12 {
13     if(cpuid() == 0){
14         consoleinit();
15         printinit();
16         printf("\n");
17         printf("xv6 kernel is booting\n");
18         printf("\n");
19         kinit();           // physical page allocator
20         kvminit();        // create kernel page table
21         kvminithart();    // turn on paging
22         procinit();       // process table
23         trapinit();       // trap vectors
24         trapinit hart(); // install kernel trap vector
25         plicinit();       // set up interrupt controller
26         plicinit hart(); // ask PLIC for device interrupts
27         binit();          // buffer cache
28         iinit();          // inode cache
29         fileinit();       // file table
30         virtio_disk_init(); // emulated hard disk
31         userinit();       // first user process
32         __sync_synchronize();
33         started = 1;
34     } else {
35         while(started == 0)
36             ;
37         __sync_synchronize();
38         printf("hart %d starting\n", cpuid());
39         kvminithart();    // turn on paging
40         trapinit hart(); // install kernel trap vector
41         plicinit hart(); // ask PLIC for device interrupts
42     }

```



# Memory initialization

Before turning on paging, we need to ensure that we have

- (1) Page frame creation (kinit)
- (2) create page tables for kernel (kvminit)
- (3) Only then can we turn on paging (kvminithart)

# Turn on Paging

VM.C

```
54 void  
55 kvminithart()  
56 {  
57     w_satp(MAKE_SATP(kernel_pagetable));  
58     sfence_vma();  
59 }
```

Kernel\_pagetable is a global variable of type pagetable\_t (ie uint64 \*)

w\_satp → sets the page table bits in the satp register

sfence\_vma → fence to guarantee any stores in the instruction stream should complete before subsequent instructions after sfence\_vma

```
static inline void  
w_satp(uint64 x)  
{  
    asm volatile("csrw satp, %0" : : "r" (x));  
}
```