Paging

Abhilash Jindal

Overview

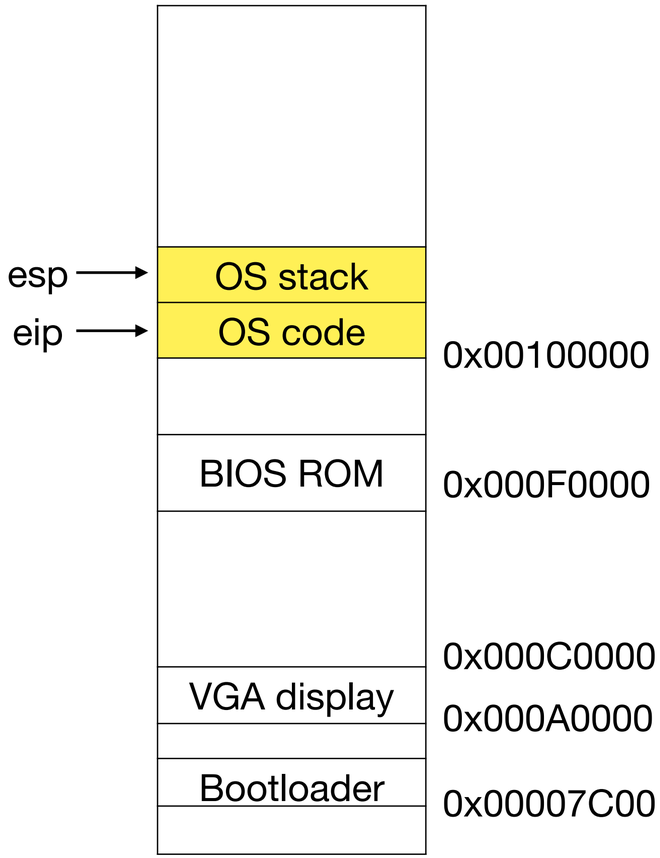
- More flexible address translation with paging (OSTEP Ch 18-20)
  - Paging hardware
- Demand paging: swapping pages to disk when memory becomes full (OSTEP Ch 21-22)
  - Swapping mechanisms
  - Page replacement algorithms
- Paging in action (xv6 book Ch 2, OSTEP Ch 23)
  - Paging on xv6
  - Fork with Copy-on-write, Guard pages

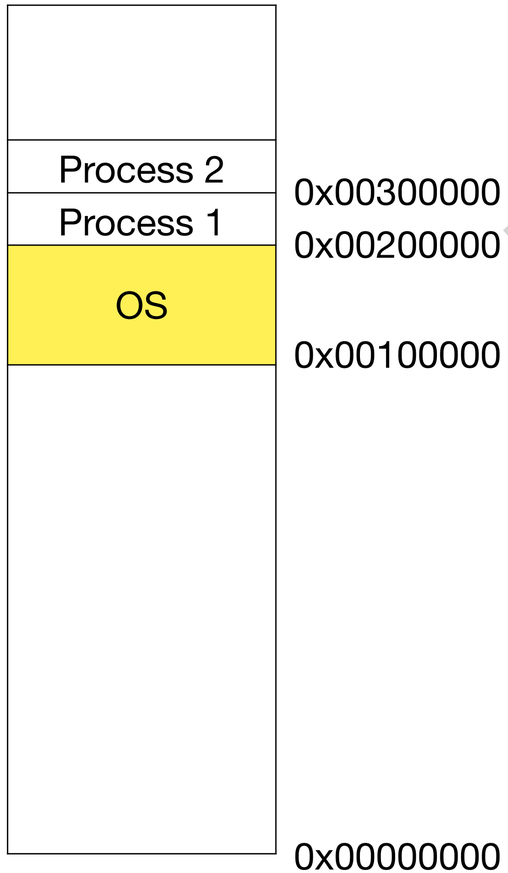
Paging Hardware

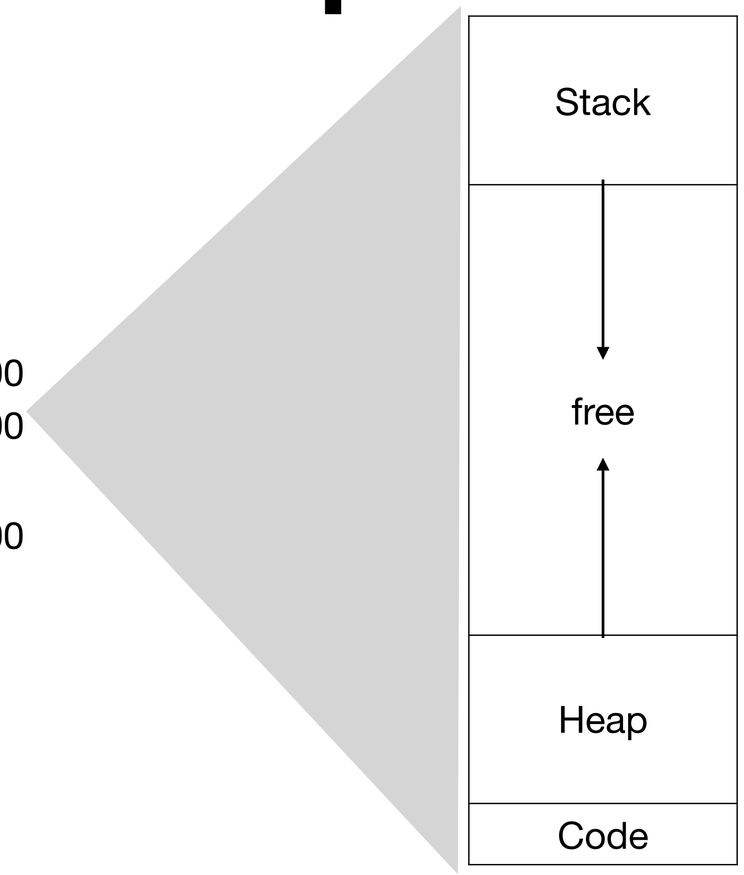
OSTEP Ch 18-20

Intel SDM Volume 3A Ch 4

Memory isolation and address space



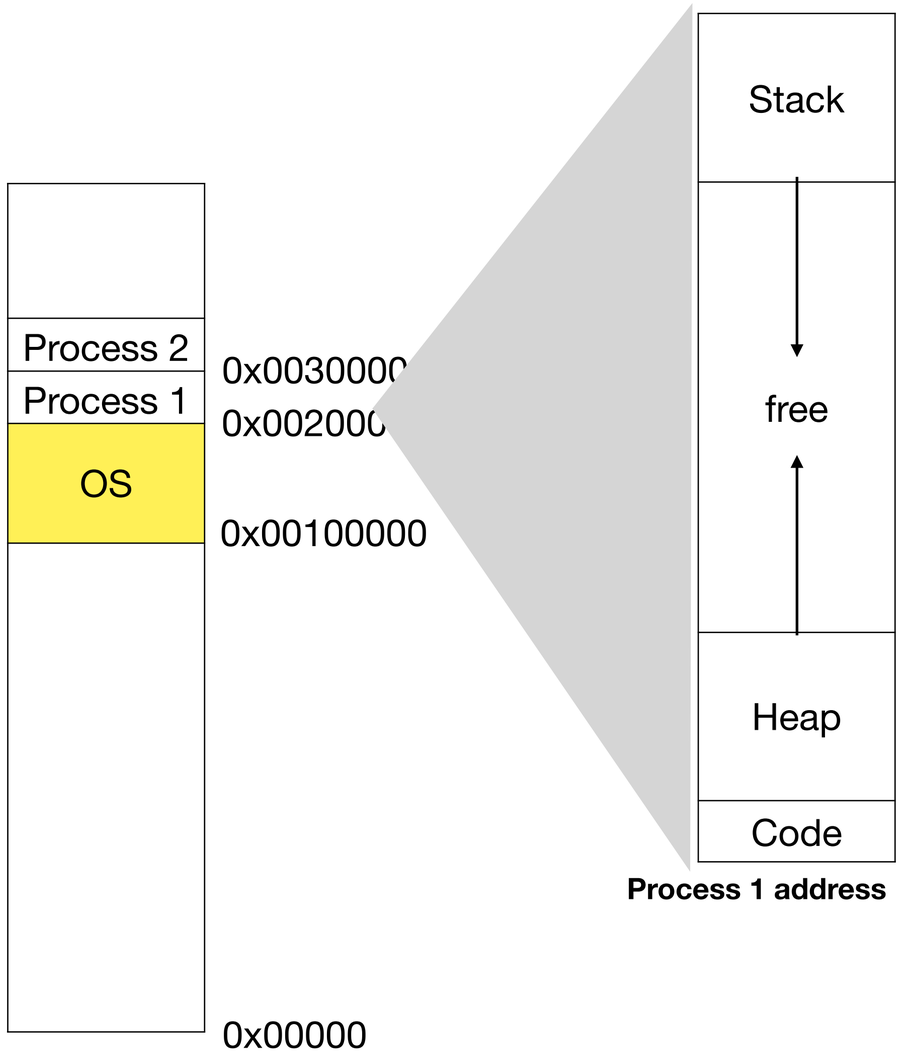




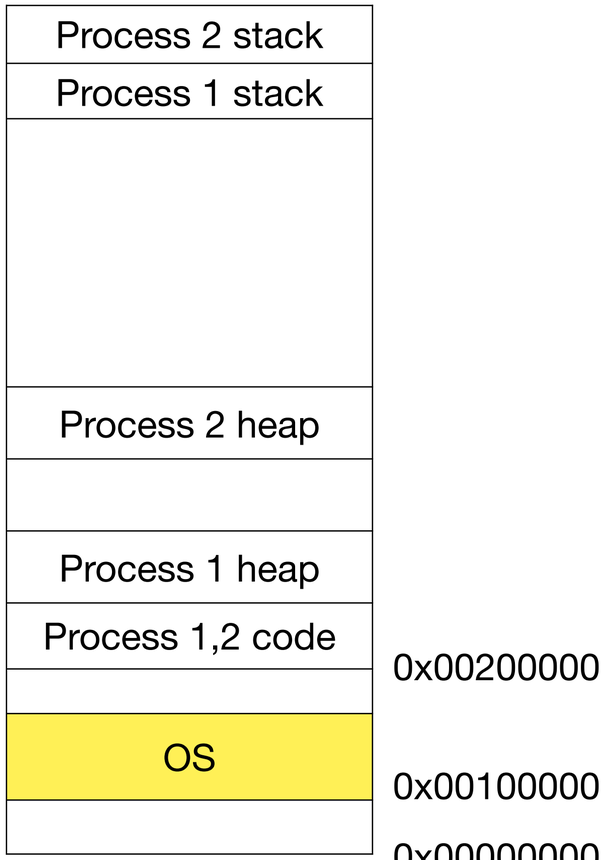
Process 1 address space

Due to address translation, compiler need not worry where the program will be loaded!

Segmentation

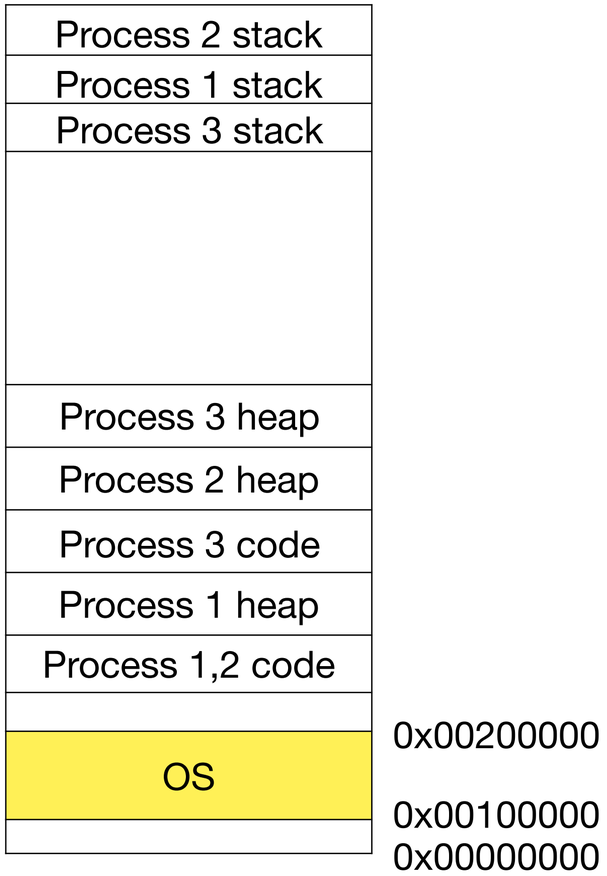


- Mapping large address spaces
  - Place each segment independently to not map free space



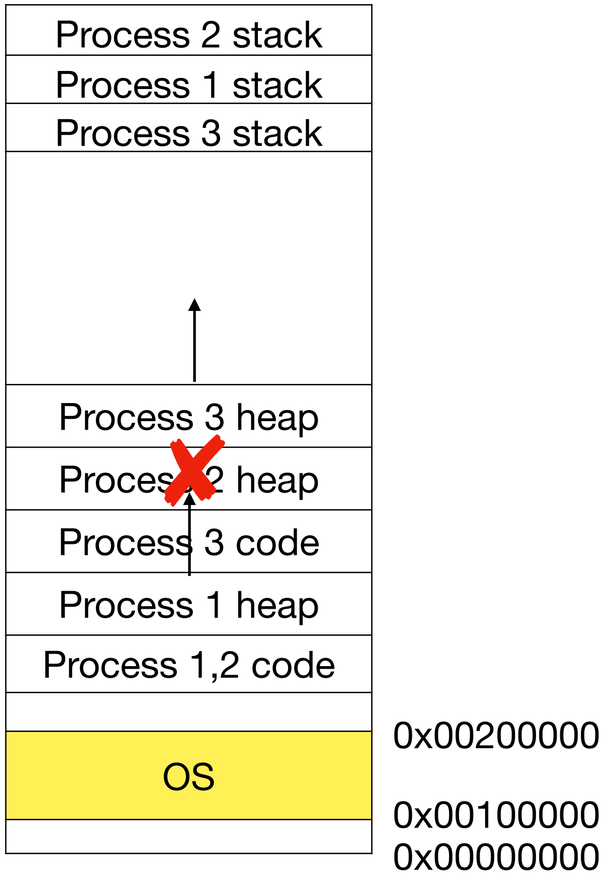
Allocating memory to a new process

- Find free spaces in physical memory
  - Difficult because segments can be of arbitrary sizes



Growing and shrinking address space

- Segments need to be contiguous in memory
- Growing might not succeed if there are other segments next to heap segment



External fragmentation

- After many processes start and exit, memory might become "fragmented" (similar to disk)
  - Example: cannot allocate 20 KB segment
- Compaction: copy all allocated regions contiguously, update segment base and bound registers
  - Copying is expensive
  - Growing heap becomes not possible

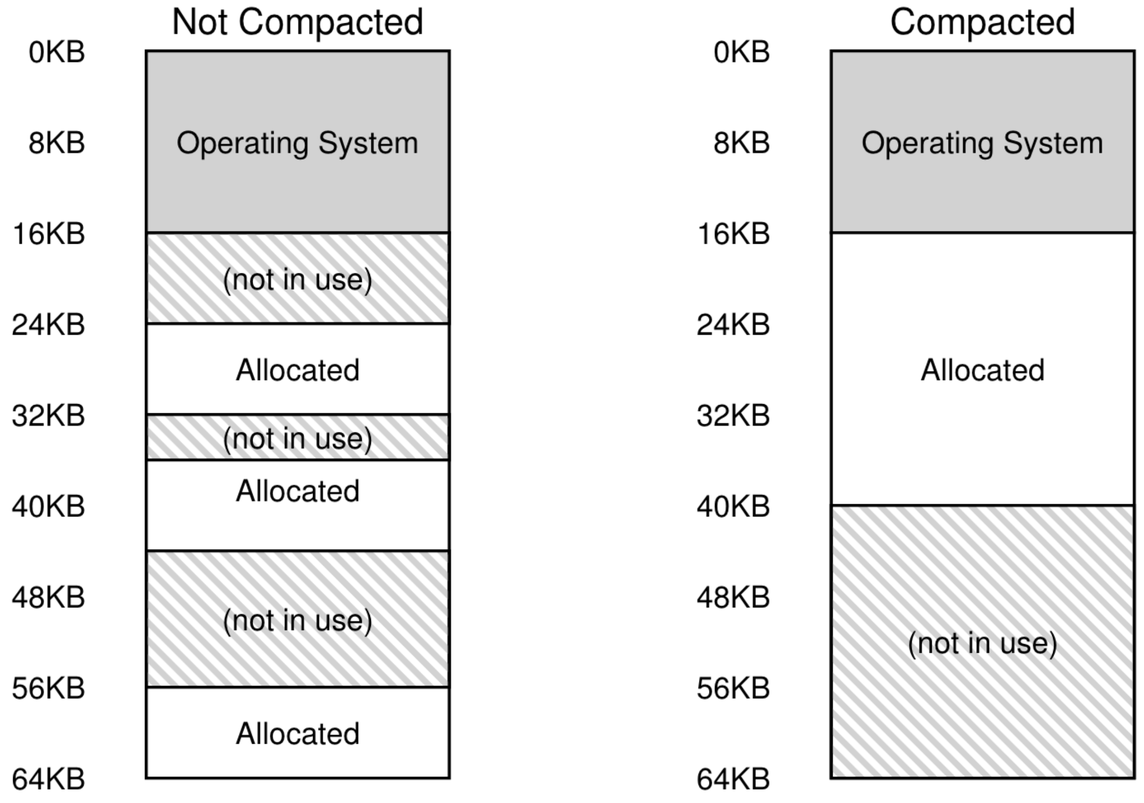


Figure 16.6: **Non-compacted and Compacted Memory**

Limitations of segmentation

Limited flexibility:

- To support large address spaces, burden falls on programmer/compiler to manage multiple segments
- Only an entire segment can be shared. Example: cannot share some part of CS (both processes use the same library)

Different sized segments, segments need to be contiguous in physical memory

- complicates physical memory allocator
- lead to external fragmentation
- growing/shrinking segments is awkward

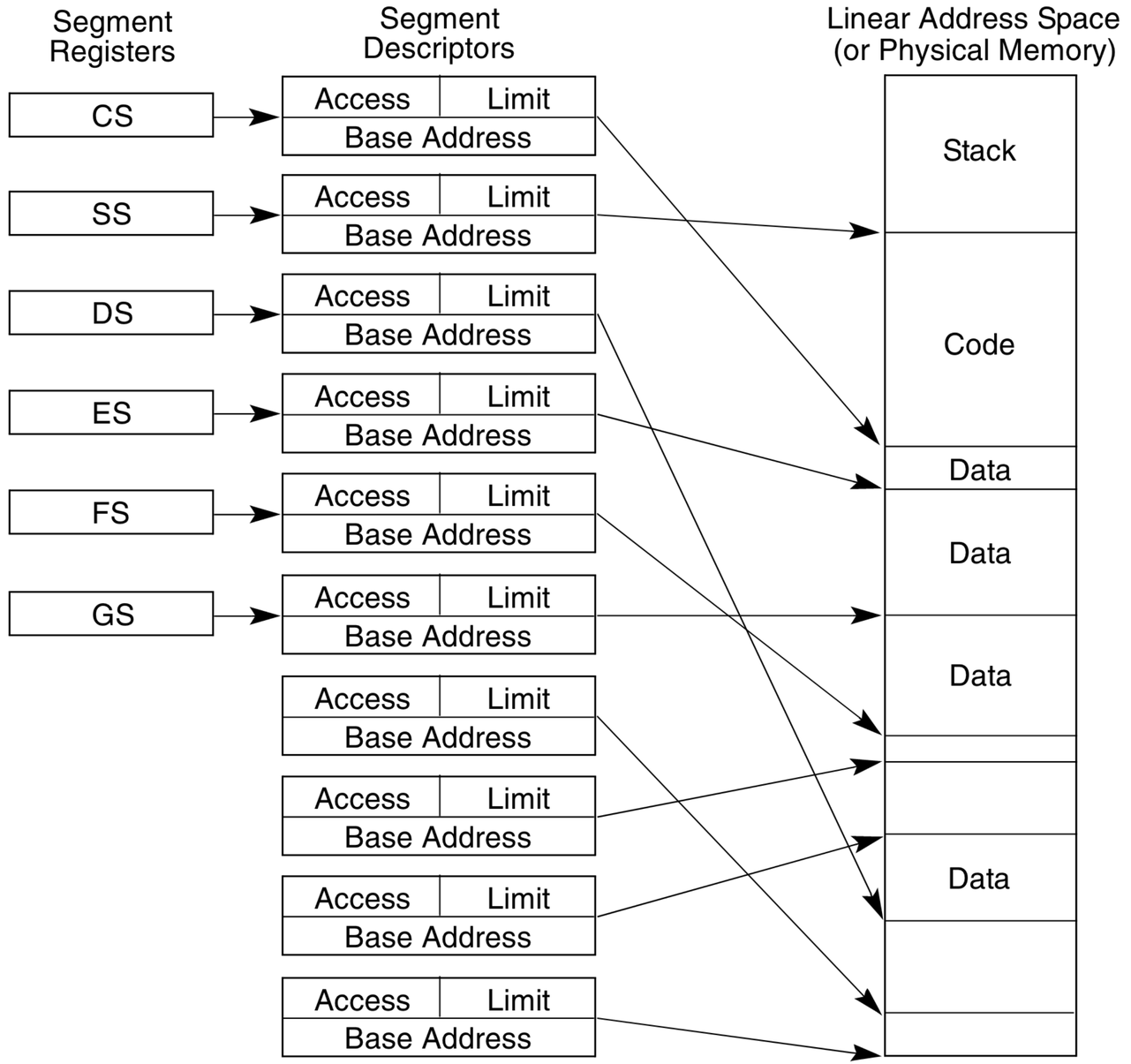


Figure 3-4. Multi-Segment Model

Paging

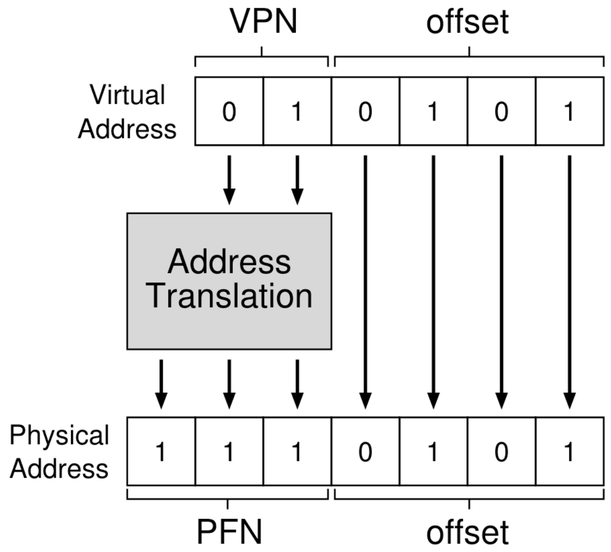
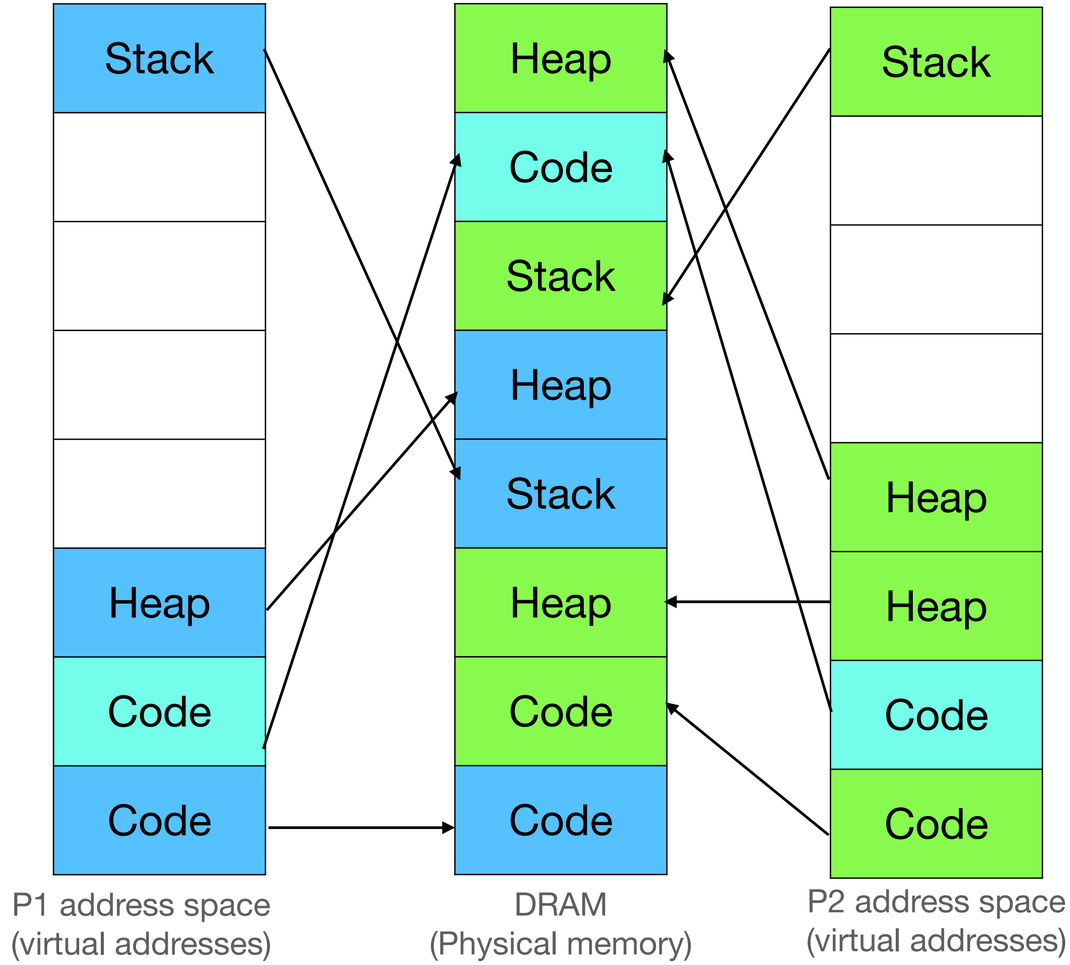


Figure 18.3: **The Address Translation Process**



Notebook analogy

Segmentation

OS

0	1	2	3	4	5
xv6	is	an	O**S**	for	x86

DB

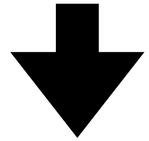
0	1	2	3
Write	an	SQL	query

Notebook

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
OS:1 DB:7	xv6	is	an	O**S**	for	x86	Write	an	SQL	query				

Preparing for OS exam:

- Read second letter from 3rd page



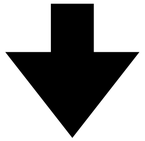
- Read second letter from 4th page

Notebook analogy

Paging

Preparing for OS exam:

- Read second letter from 3rd page



- Read second letter from 8th page

OS

0	1	2	3	4	5
xv6	is	an	O**S**	for	x86

DB

0	1	2	3
Write	an	SQL	query

Notebook

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
OS:1 DB:5	0:3,1:4, 2:2,**3:8**, 4:6,5:10	an	xv6	is	0:9,1:2, 2:7,3:12	for	SQL	O**S**	Write	x86		query		

Paging

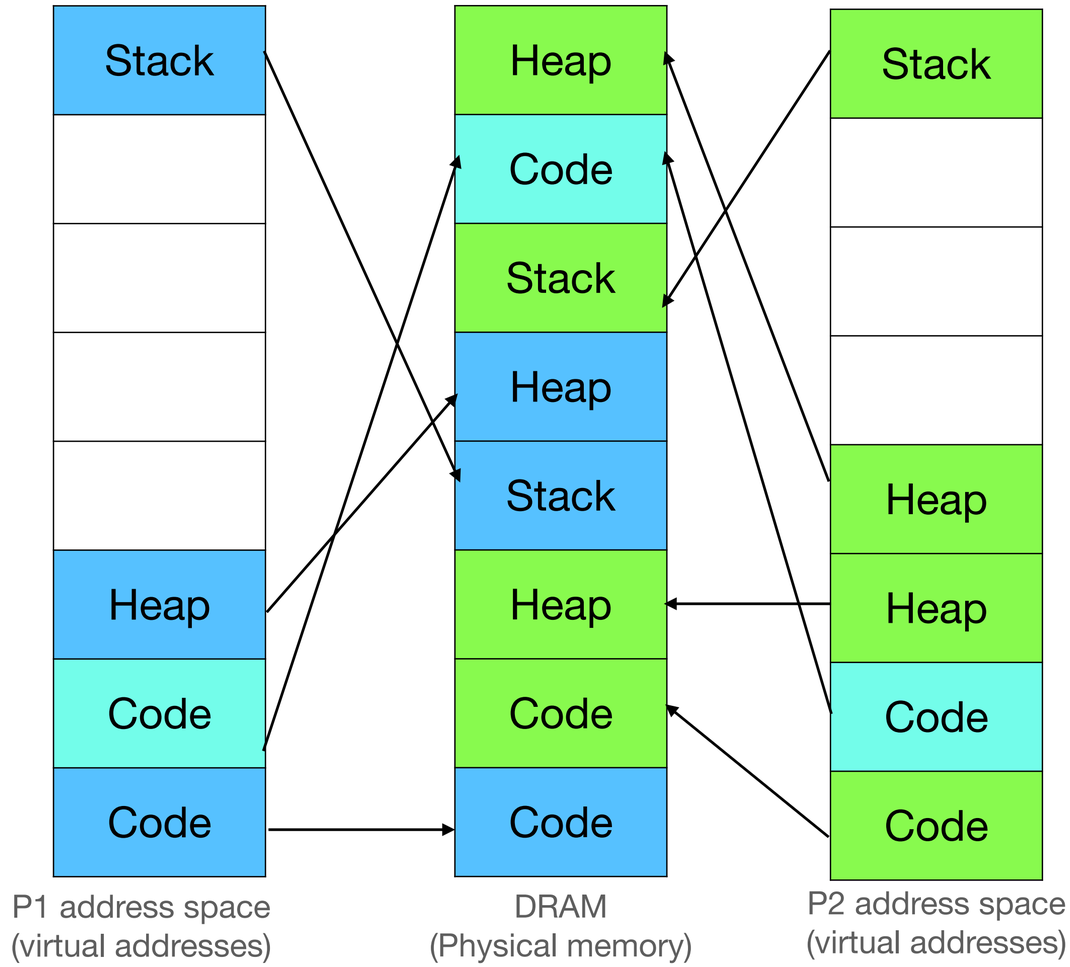
Segmentation	**Paging**
Large address spaces need multi segment model. Burden falls on programmer/compiler to manage multiple segments	Transparently support large address spaces. Programmer/compiler work with a flat virtual address space
Different sized segments lead to external fragmentation	Fixed sized pages (4KB, 4MB)
Different sized segments complicate physical memory allocator	
Full segment needs to be contiguous in physical memory	Page is contiguous. Neighbouring addresses (in different pages) may not be contiguous
Growing/shrinking segments is awkward	Simple: allocate another page, free a page
Address translation hardware has an adder (va + base) and a comparator (va < limit)	Address translation hardware is much more complicated

How to do page-based address translation?

Page table: Maintain a lookup table for each process

Virtual page number	Physical page number
7	3
6	x
5	x
4	x
3	x
2	4
1	6
0	0

Virtual page number	Physical page number
7	5
6	x
5	x
4	x
3	7
2	3
1	6
0	1

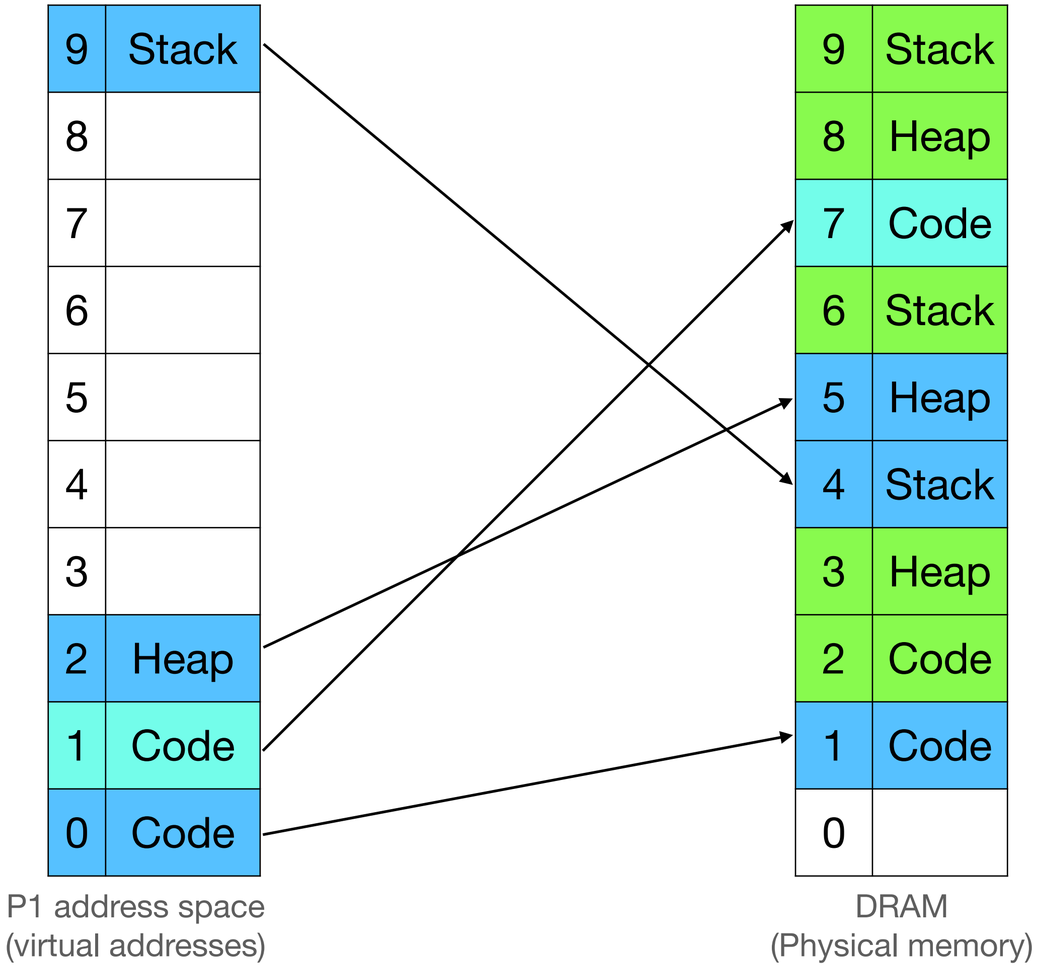


How to do page-based address translation? (2)

Page table bits

- Present bit: valid mapping
- Permission bits: read only, writeable, executable

Virtual page number	Physical page number	Present	Permission
9	4	Y	rw
8	x	N	
7	x	N	
6	x	N	
5	x	N	
4	x	N	
3	x	N	
2	5	Y	rw
1	7	Y	rx
0	1	Y	rx



Page table size

- Virtual addresses: 2^{32}
- Size of page = (4KB) = 2^{12}
- Number of page table entries = 2^{20}
- Number of pages in a 4GB DRAM = $2^{32}/2^{12} = 2^{20}$
- Size of each page table entry = 20 bits ~ 3 bytes
- Size of page table = $3*2^{20}$ = 3MB!
- 1000 processes => ~3GB memory!

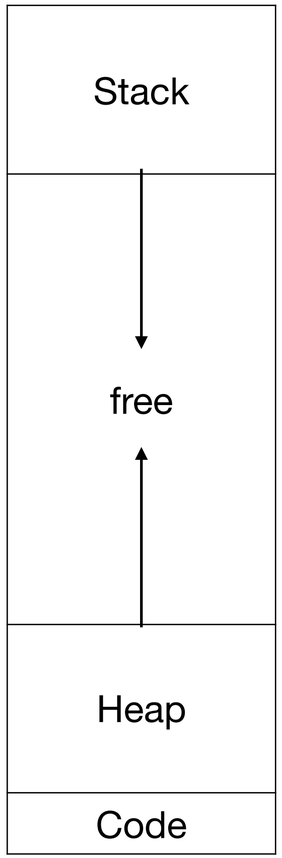
Virtual page number	Physical page number
7	3
6	x
5	x
4	x
3	x
2	4
1	6
0	0

Reducing page table size

- Bigger pages
  - Virtual addresses: 2^{32}
  - Size of page = (**4MB**) = 2^{22}
  - Number of page table entries = 2^{10}
  - Number of pages in a 4GB DRAM = $2^{32}/2^{22} = 2^{10}$
  - Size of each page table entry = 10 bits ~ 2 bytes
  - Size of page table = $2*2^{10}$ = 2KB!
- Bigger pages increase internal fragmentation

Observation

- Lot of page table entries are invalid



Process 1 address space

Virtual page number	Physical page number
7	3
6	x
5	x
4	x
3	x
2	4
1	6
0	0

Multi-level page table

Virtual page number	Physical page number
7	3
6	x
5	x
4	x
3	x
2	4
1	6
0	0

PPN: 8

Virtual page number	Physical page number
6,7	9
4,5	x
2,3	11
0,1	10

PPN: 9

Virtual page number	Physical page number
7	3
6	x

PPN: 11

Virtual page number	Physical page number
3	x
2	4

PPN: 10

Virtual page number	Physical page number
1	6
0	0

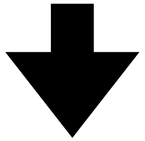
- Page directory entries point to page table pages
- Unused portions of virtual address space is skipped!

Notebook analogy

Paging

Preparing for OS exam:

- Read second letter from 3rd page



- Read second letter from 8th page

OS

0	1	2	3	4	5
xv6	is	an	OS	for	x86

DB

0	1	2	3
Write	an	SQL	query

Notebook

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
OS:1 DB:5	0:3,1:4, 2:2,**3:8**, 4:6,5:10	an	xv6	is	0:9,1:2, 2:7,3:12	for	SQL	OS	Write	x86		query		

Notebook analogy

Page directories: call 4 pages a “section”

OS

0	1	2	3	0	1	2	3
xv6	is	an	O**S**	for	x86		
Section 0				Section 1			

DB

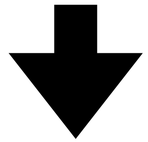
0	1	2	3
Write	an	SQL	query
Section 0			

Notebook

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
OS:1 DB:5	**0:11**, 1:14	an	xv6	is	0:13	for	SQL	O**S**	Write	x86	0:3,1:4, 2:2,**3:8**	query	0:9,1:2, 2:7,3:12	0:6, 1:10

Preparing for OS exam:

- Read second letter from 3rd page in Section 0



- Read second letter from 8th page

Address translation

Simple address space

- 16KB address space has 2^{14} addresses
- Each page has 64 (= 2^6) bytes
- Number of pages = 2^8
- First 8 bits are page number, last 6 bits are offset within the page

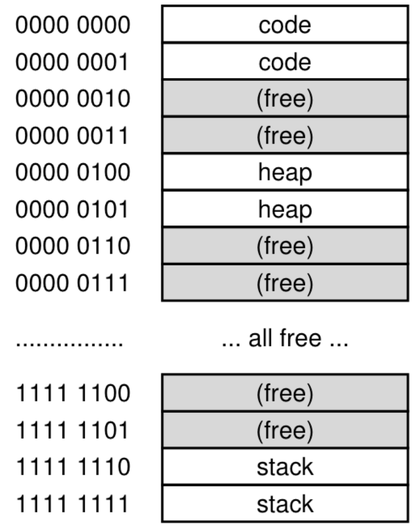
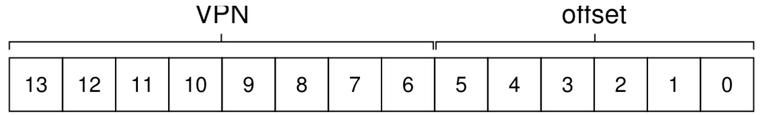
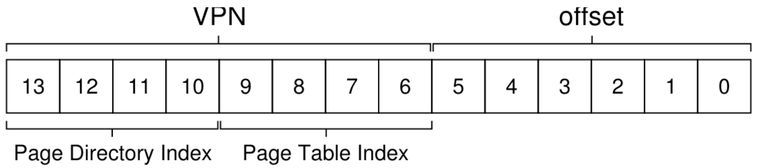


Figure 20.4: **A 16KB Address Space With 64-byte Pages**



Address translation

Simple address space



- Example: 0x3F81 (VA) => 0x0DC1 (PA)

Page dir idx 15				Page table idx 14				Offset					
1	1	1	1	1	1	1	0	0	0	0	0	0	1
3				F				8				1	

Physical page number 55								Offset					
0	0	1	1	0	1	1	1	0	0	0	0	0	1
0				D				C				1	

Page Directory		Page of PT (@PFN:100)			Page of PT (@PFN:101)		
PFN	valid?	PFN	valid	prot	PFN	valid	prot
100	1	10	1	r-x	—	0	—
—	0	23	1	r-x	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	80	1	rw-	—	0	—
—	0	59	1	rw-	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	55	1	rw-
101	1	—	0	—	45	1	rw-

Figure 20.5: **A Page Directory, And Pieces Of Page Table**

x86 segmentation and paging

- Segmentation:
  - Virtual address (logical address) => “linear address”
- Paging:
  - Linear address => Physical address

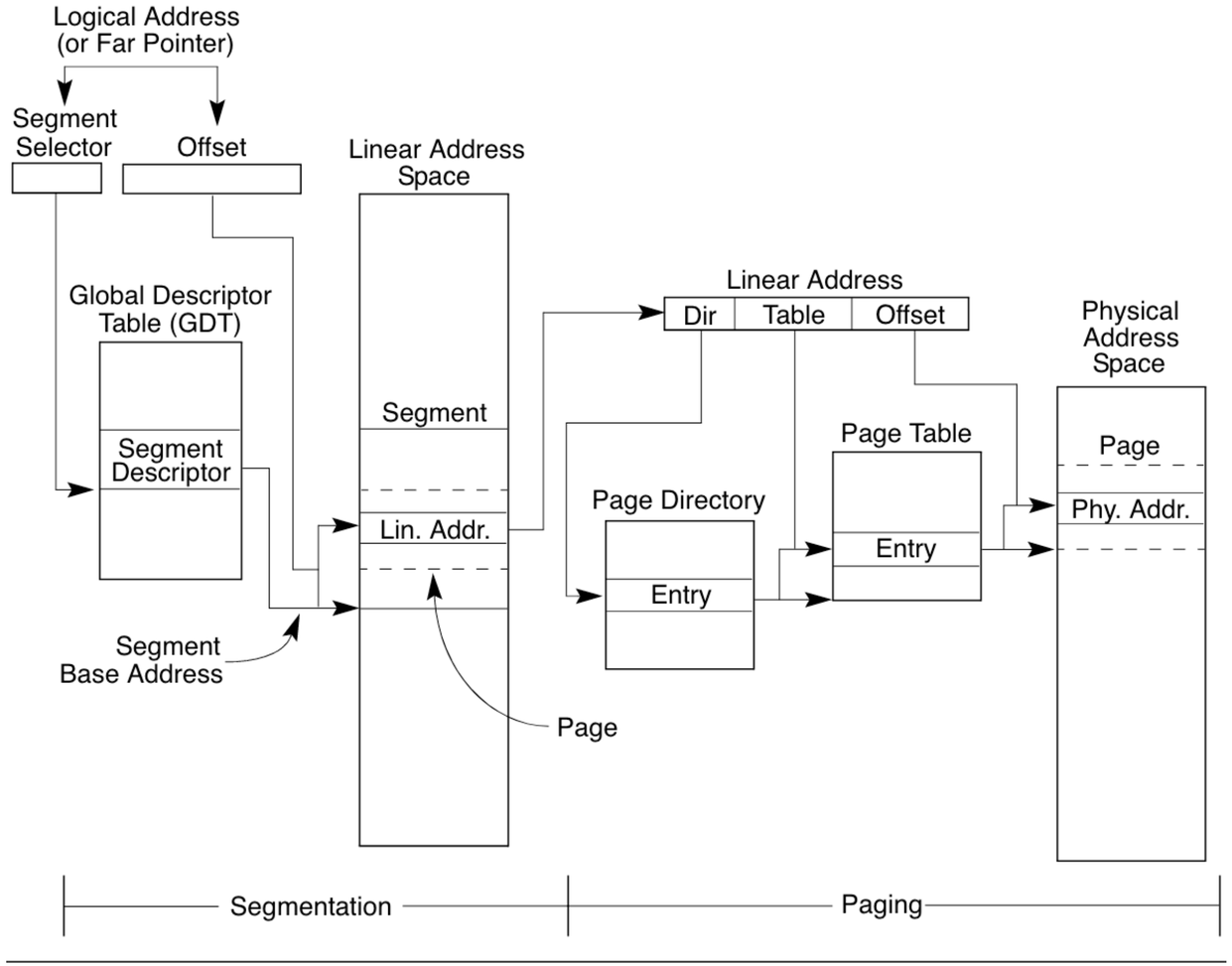


Figure 3-1. Segmentation and Paging

Address translation with paging on x86

- CR3 contains page directory base

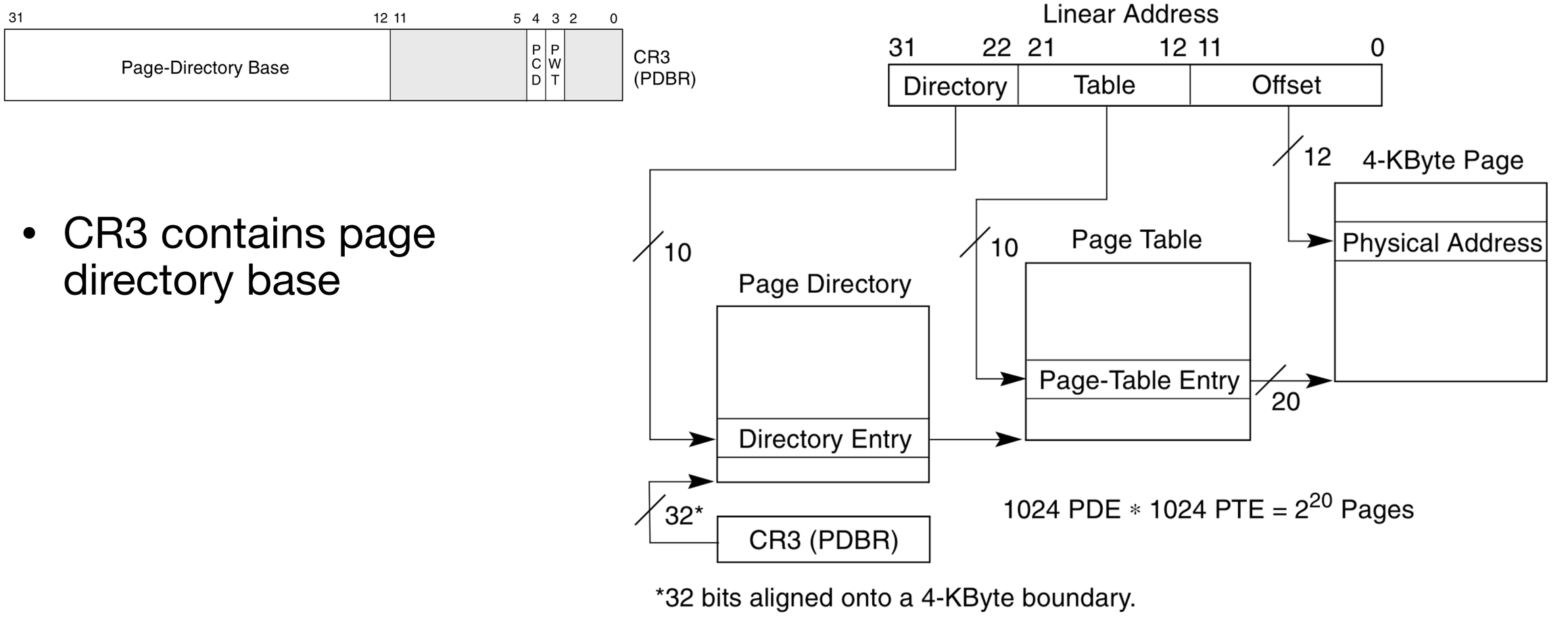
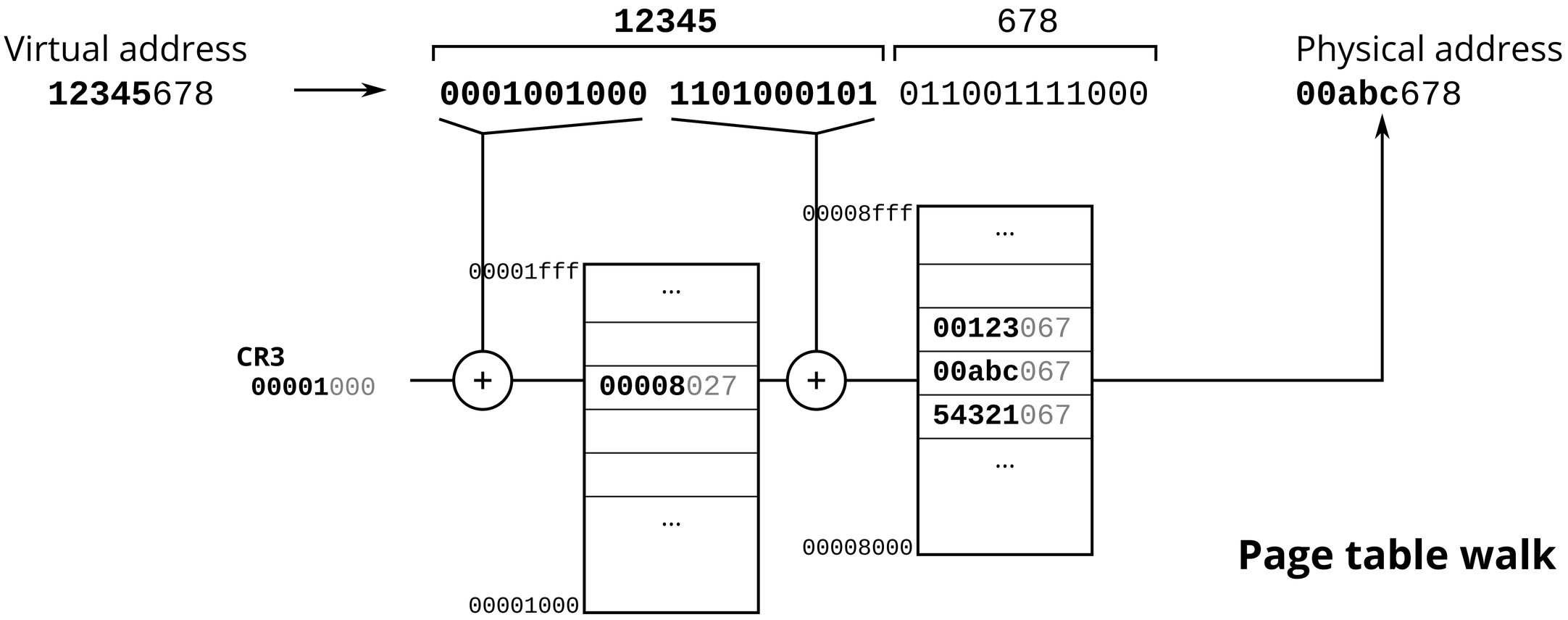


Figure 3-12. Linear Address Translation (4-KByte Pages)

Address translation with paging on x86



x86 PTEs, PDEs

- 2^{20} 4KB pages in a 4GB DRAM
  - Page base address, page table base address are 20 bits
- Bits set by OS, used by hardware:
  - Present: It is a valid entry
  - Read/write: Can write if 1
  - User/supervisor: CPL=3 can access if 1
- Bits set by hardware, used/cleared by OS:
  - Accessed: Hardware accessed this page
  - Dirty: Hardware wrote to this page

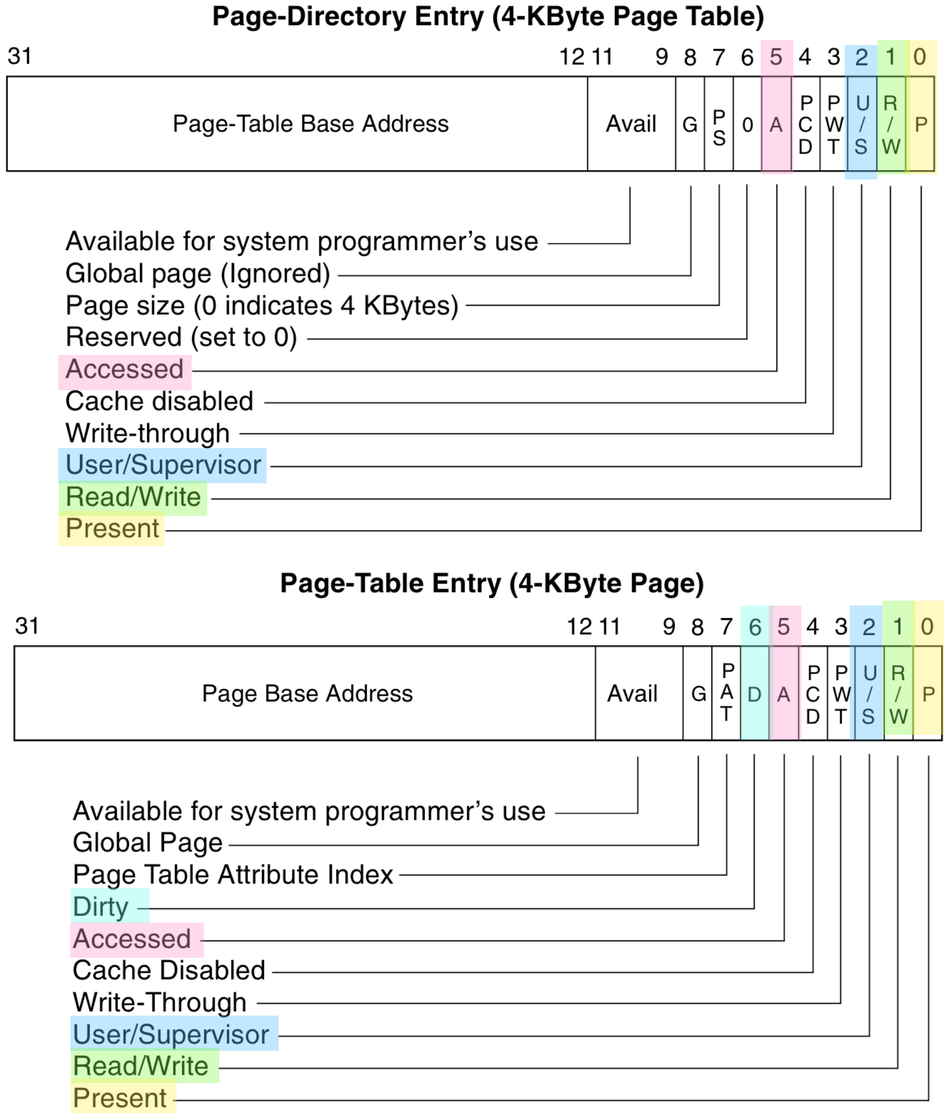
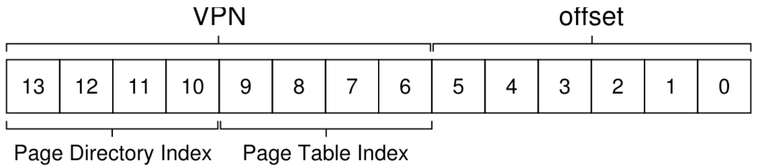


Figure 3-14. Format of Page-Directory and Page-Table Entries for 4-KByte Pages and 32-Bit Physical Addresses

Performance degradation!



- Accessing 1 memory location requires accessing 3 memory locations!

Page dir idx 15				Page table idx 14				Offset					
1	1	1	1	1	1	1	0	0	0	0	0	0	1
3				F				8				1	

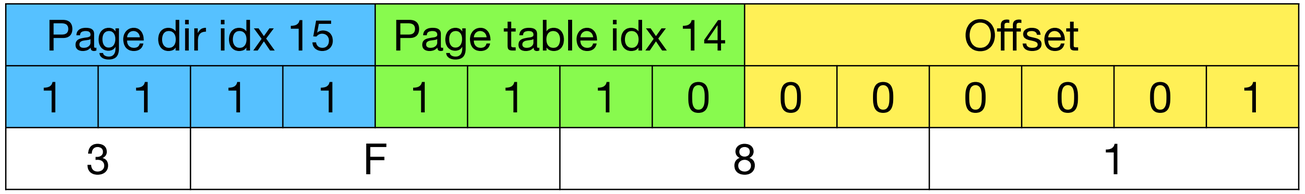
Physical page number 55								Offset					
0	0	1	1	0	1	1	1	0	0	0	0	0	1
0				D				C				1	

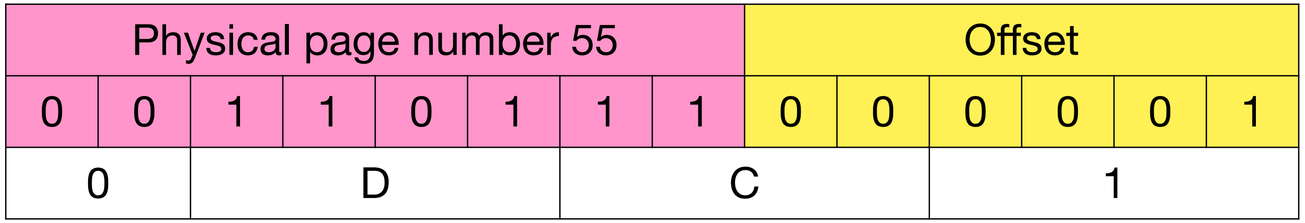
Page Directory		Page of PT (@PFN:100)			Page of PT (@PFN:101)		
PFN	valid?	PFN	valid	prot	PFN	valid	prot
100	1	10	1	r-x	—	0	—
—	0	23	1	r-x	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	80	1	rw-	—	0	—
—	0	59	1	rw-	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	55	1	rw-
101	1	—	0	—	45	1	rw-

Figure 20.5: **A Page Directory, And Pieces Of Page Table**

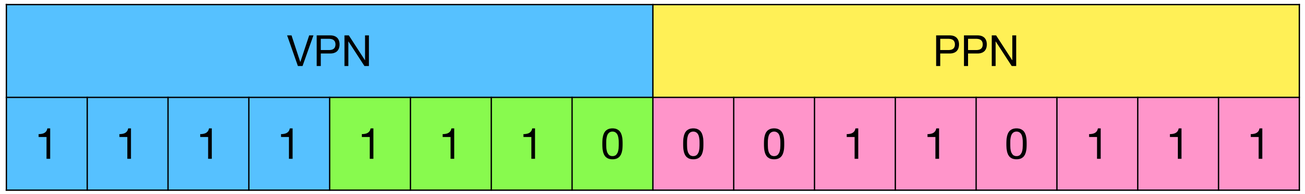
Translation-lookaside buffer (TLB)

- First check page translation in TLB before walking the page table
- TLB hit: ~0.5-1 cycle
- TLB miss: ~10-100 cycles





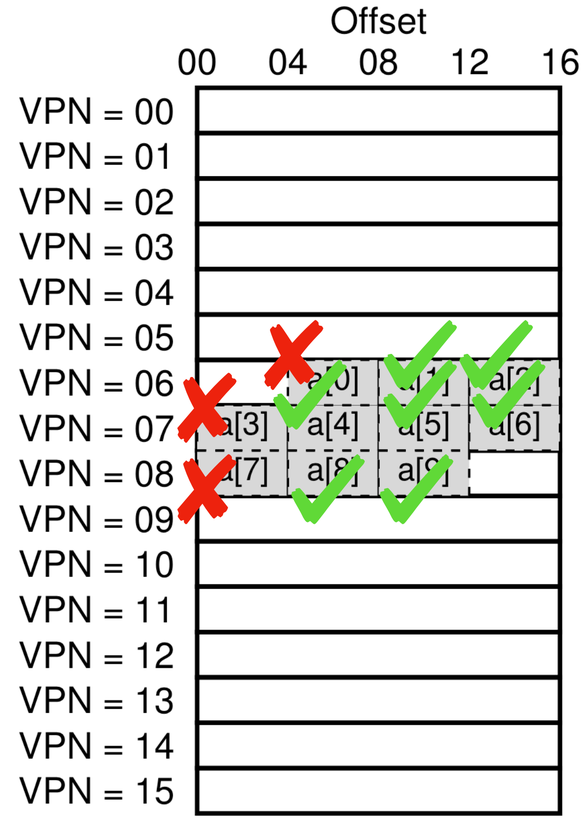
TLB



Which programs will run faster?

Which programs will have lesser TLB misses?

```
int sum = 0;
for (i = 0; i < 10; i++) {
    sum += a[i];
}
```



- High **spatial locality**: after the program accessed a memory location, it will access a nearby memory location
- High **temporal locality**: after the program accessed a memory location, it will soon access it again

Example bad programs

- Low spatial and temporal locality: most accesses lead to TLB miss
- Large hash table with random access

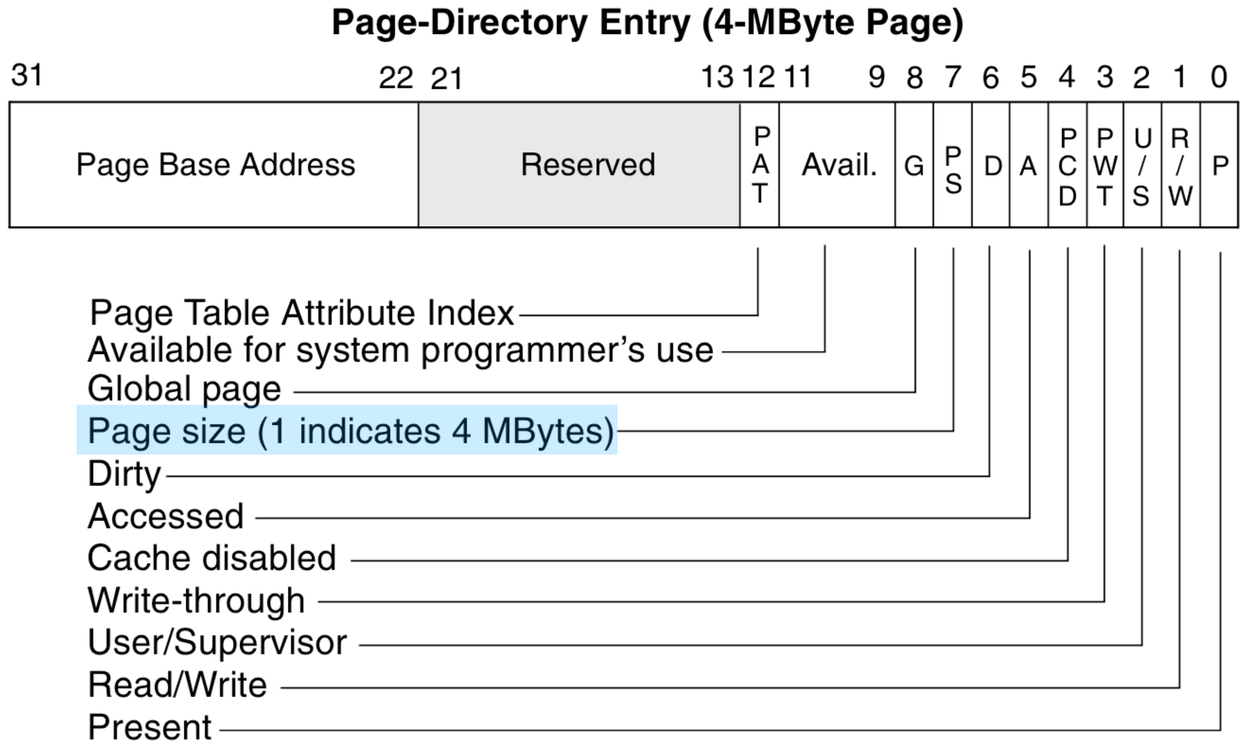
```
int get(int I)

  return a[I];
```

VPN=1	a[0]	a[1]	a[2]	a[3]
VPN=2	a[4]	a[5]	a[6]	a[7]
VPN=3	a[8]	a[9]	a[10]	a[11]
VPN=4	a[12]	a[13]	a[14]	a[15]
VPN=5	a[16]	a[17]	a[18]	a[19]

- Traversing large graphs: two neighbours can be on different pages
- Ok for small hash tables, small graphs.
  - Working set of the program: Amount of memory that it touches
- TLB reach = (number of TLB entries) * (page size)
- Working set > TLB reach

Increasing TLB reach

- Larger TLBs
  - 32 -> 64 entries.
  - Larger caches => slower hits
- Larger pages
  - 1 TLB entry for 4MB of addresses (with a 4MB page)
  - 1024 TLB entries for 4MB of addresses (with 4KB pages)
- Allocating large pages on Linux:




```
posix_memalign(void **memptr, size_t alignment, size_t size);
int madvise(void addr, size_t length, int advice= MADV_HUGEPAGE);
```

TLB on my x86-64 machine

```
$ cpuid
  ..
  (simple synth)  = Intel Core (unknown type) (Kaby Lake / Coffee Lake) {Skylake}, 14nm
  ..
   cache and TLB information (2):
      0x63: data TLB: 2M/4M pages, 4-way, 32 entries
            data TLB: 1G pages, 4-way, 4 entries
      0x03: data TLB: 4K pages, 4-way, 64 entries
      0x76: instruction TLB: 2M/4M pages, fully, 8 entries
      0xb5: instruction TLB: 4K, 8-way, 64 entries
      0xc3: L2 TLB: 4K/2M pages, 6-way, 1536 entries
 ..
```

Separate TLBs for different page sizes

Separate TLBs for instruction and data

Much larger (slower) L2 TLB

Data TLB reach:

- 4KB * 64 = 256KB
- 4MB * 32 = 128MB
- 1GB * 4 = 4GB

Instruction TLB reach:

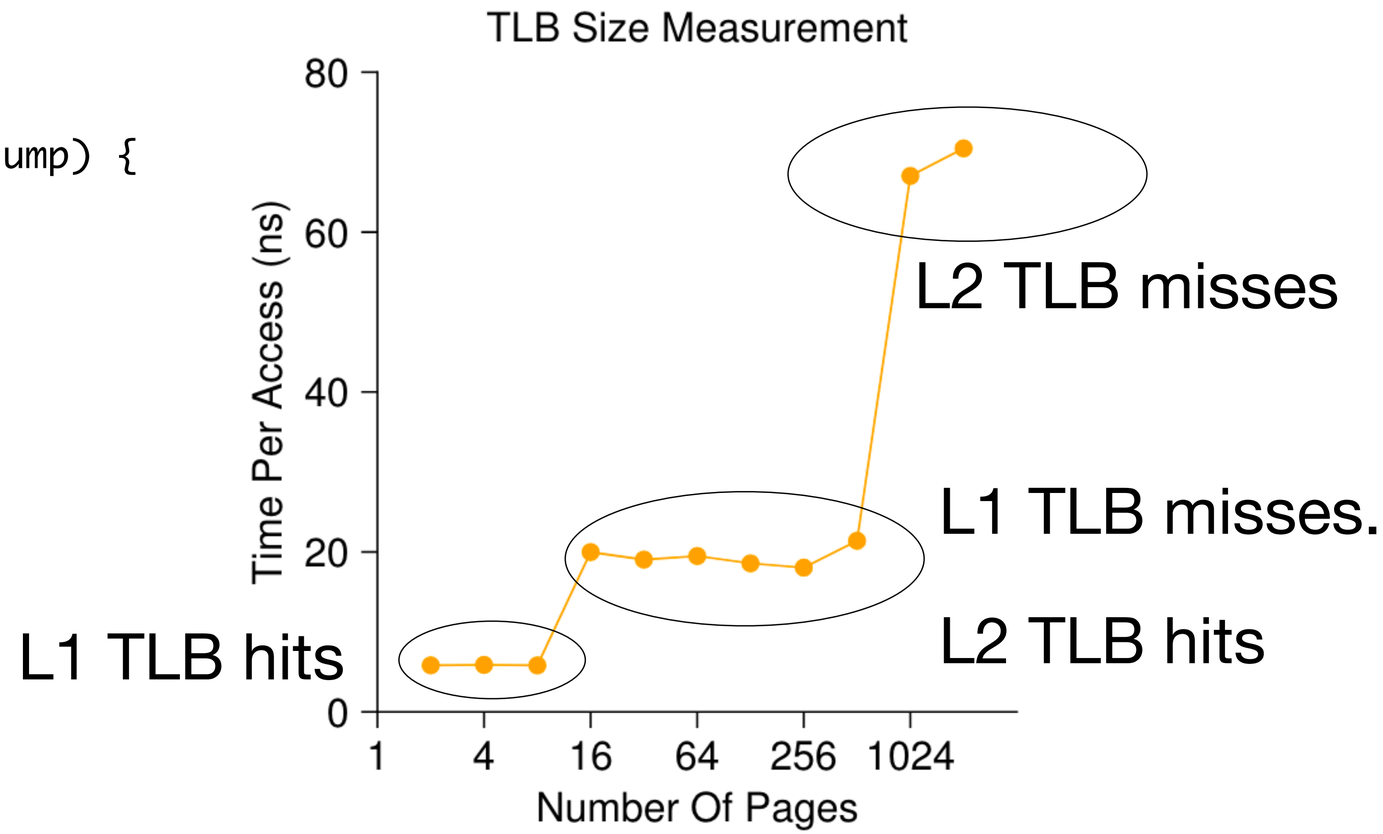
- 4KB * 64 = 256KB
- 4MB * 8 = 32MB

L2 TLB reach:

- 4KB * 1536 = 6MB
- 2MB * 1536 = 3GB

TLB size, multiple TLBs example

```
int jump = PAGESIZE / sizeof(int);
for (i = 0; i < NUMPAGES * jump; i += jump) {
      a[i] += 1;
}
```



TLB replacement policies

- Need to replace an entry after TLB is full. Which entry to replace to minimise TLB miss rate?
- Least recently used (LRU):
  - If an entry hasn’t been used recently, it is unlikely to be used soon => assumes spatial and temporal locality of access.
  - Corner case behaviours: Program cycling over N+1 pages.
- Hardware typically implements something simple
  - FIFO
  - Random replacement: Just pick an entry randomly

Context switching

- During context switch, OS changes CR3 register to change page table

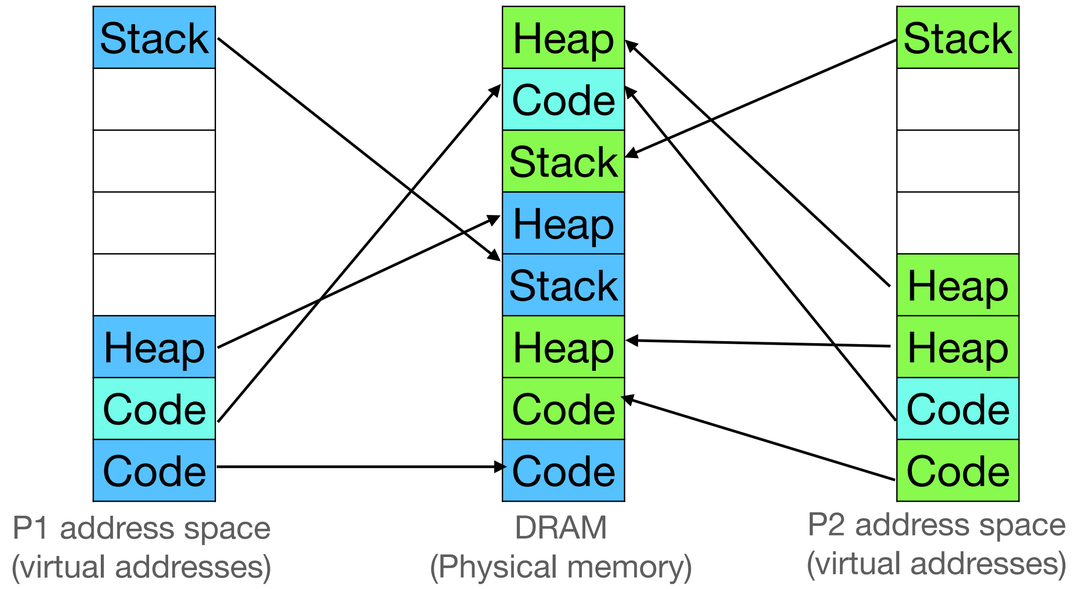
```
movl %eax, %cr3
```

- Privileged operation
- Marks each TLB entry invalid. Every memory access after context switch causes TLB miss!

Virtual page number	Physical page number
~~0~~	~~0~~
~~2~~	~~4~~
0	1

Virtual page number	Physical page number
7	3
6	x
5	x
4	x
3	x
2	4
1	6
0	0

Virtual page number	Physical page number
7	5
6	x
5	x
4	x
3	7
2	3
1	6
0	1



INVLPG instruction

- TLB is neither write-back, nor write-through cache
  - Need to run INVLPG <virtual page number> when a page table entry is modified
- Similar to how OS needs to run LGDT when GDT entries are changed

Tagged TLBs

- TLB entries are “tagged” with a *process context identifier*
- Additional bits in CR3 register tells hardware about the current PCID
- Upon context switch:
  - OS changes CR3: PCID, page directory base
  - Hardware *need not* invalidate TLB entries. ~5us
  - When process gets back control, some of its TLB entries might still be present!

TLB

Virtual page number	Physical page number	Process context identifier
0	0	P1
2	4	P1
0	1	P2

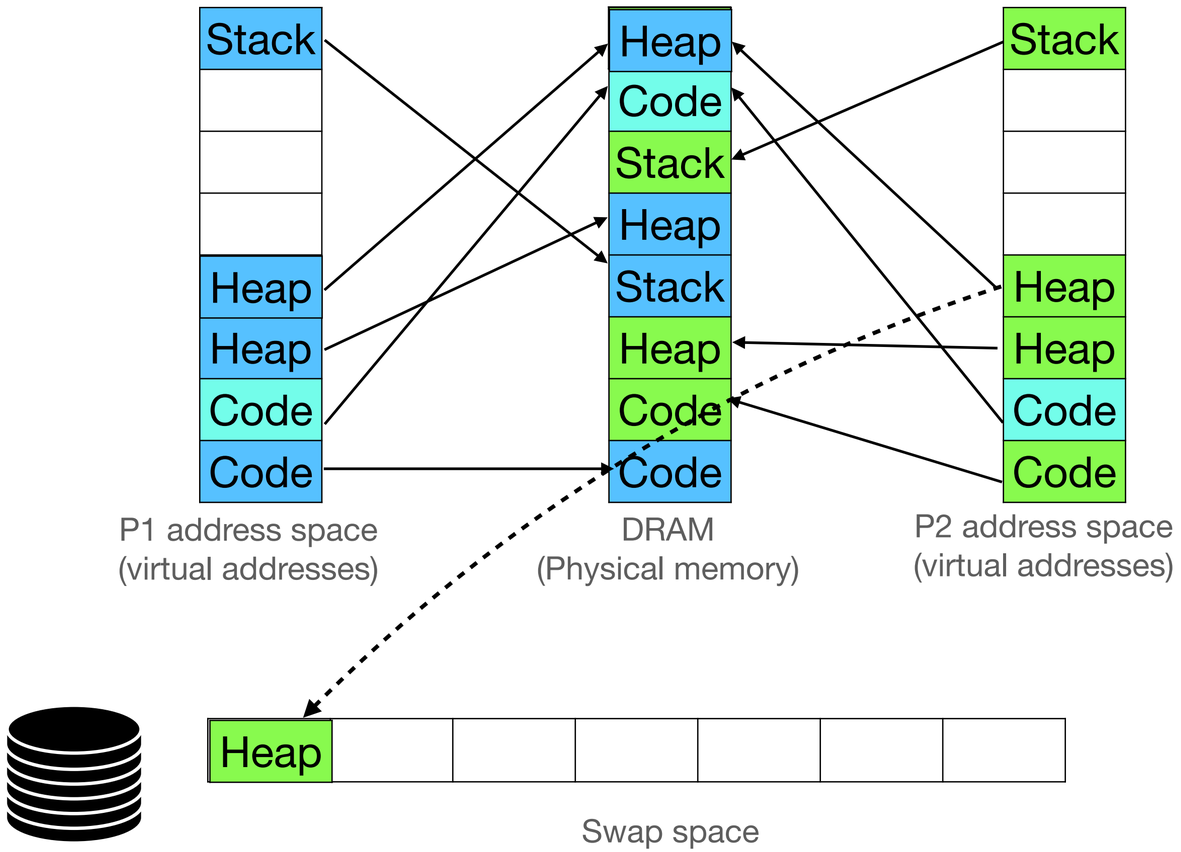
Demand Paging

OSTEP Ch 21-22

Demand paging

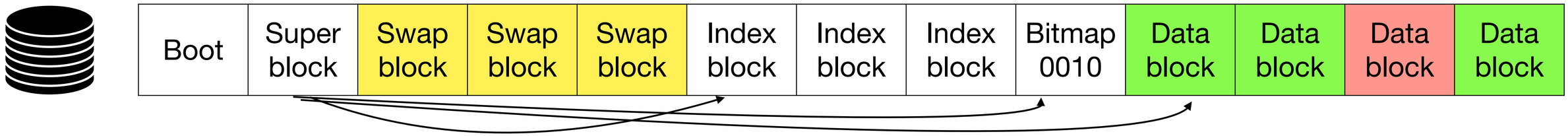
Providing illusion of large virtual address spaces

- Swap out a page to disk when physical memory is about to run out of space:
  - Mark page as not present in page table
  - Remember where page is swapped out on disk
- Add swapped out physical page to free list



Disk layout with swap space

- Reserved swap blocks, not touched by file system
- Swap space does not require crash consistency: anyways garbage after restart (all processes are dead)



Swapping out a page

- Find a free swap block on disk
- Copy page to the free block
- Run INVLPG instruction to remove page from TLB
- Mark not present, remember swap block number in PTE
- Add page to free list

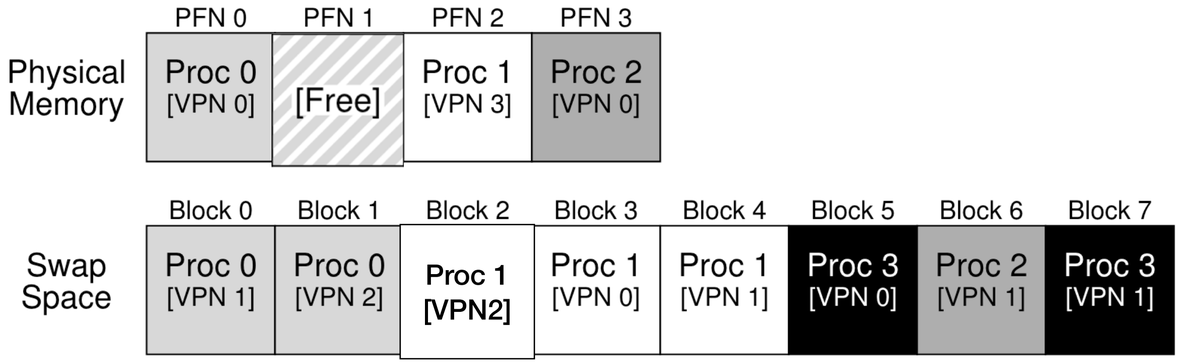
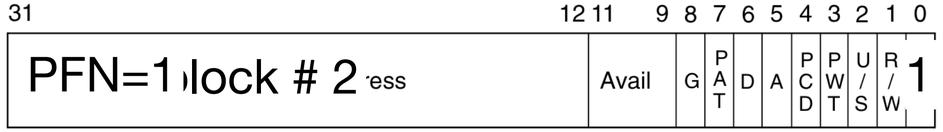


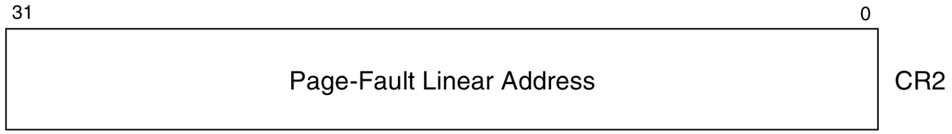
Figure 21.1: **Physical Memory and Swap Space**

Page table entry for Proc 1 [VPN2]



Swapping in a page

- Hardware does page table walk to find that the page is not present. It raise page fault.
- OS handles page fault:
  - Copies page to a free physical page
  - Updates page table entry
- Hardware retries instruction. This time finds the page, adds entry to TLB, continues as normal



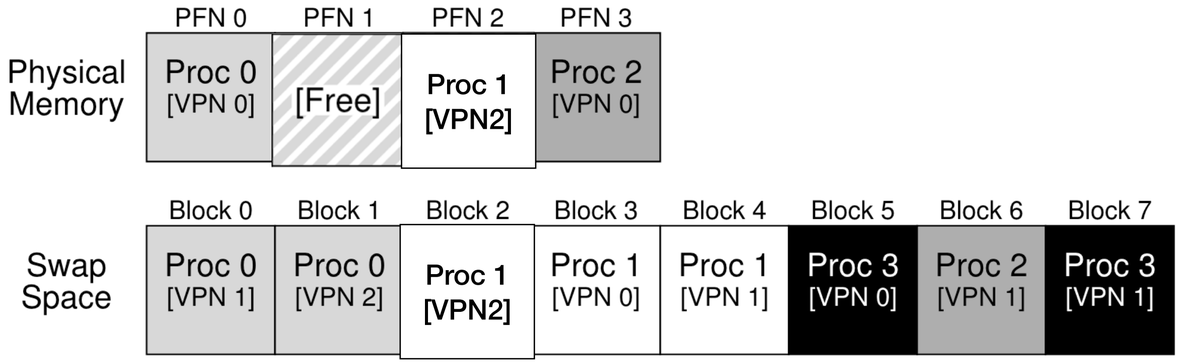
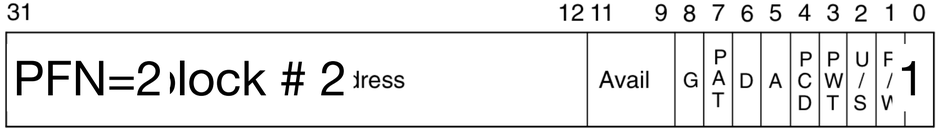


Figure 21.1: **Physical Memory and Swap Space**

Page table entry for Proc 1 [VPN2]



Page replacement policies

- When to evict pages?
- How many pages to evict?
- Which page to evict?

When to evict pages? How many pages to evict?

- Swap out one page when we completely run out of physical memory
  - What if OS itself needed a new page?
- Start swapping out before we completely run out
  - When there are less than N free pages left
- Swap out multiple pages in one shot until we have M (> N) free pages left
  - Sends multiple disk writes in one shot reduces seek delay

Which page to evict?

- Goal: minimize number of swap ins/outs
- Belady's algorithm for optimal page replacement
  - Evict the page required furthest in the future
  - Optimal because all other pages will be required sooner
- Future is unknown!

Memory access sequence:

0, 1, 2, 0, 1, 3, 0, 3, 1, 2, 1

Access	Hit/Miss?	Evict	Resulting Cache State
0	Miss		0
1	Miss		0, 1
2	Miss		0, 1, 2
0	Hit		0, 1, 2
1	Hit		0, 1, 2
3	Miss	2	0, 1, 3
0	Hit		0, 1, 3
3	Hit		0, 1, 3
1	Hit		0, 1, 3
2	Miss	3	0, 1, 2
1	Hit		0, 1, 2

Figure 22.1: **Tracing The Optimal Policy**

FIFO

- Evict the page that came first to the cache
- OS appends the page to a queue when it swaps in a page (or when it allocates a new page)

Memory access sequence:

0, 1, 2, 0, 1, 3, 0, 3, 1, 2, 1

Access	Hit/Miss?	Evict	Resulting Cache State
0	Miss		First-in→ 0
1	Miss		First-in→ 0, 1
2	Miss		First-in→ 0, 1, 2
0	Hit		First-in→ 0, 1, 2
1	Hit		First-in→ 0, 1, 2
3	Miss	0	First-in→ 1, 2, 3
0	Miss	1	First-in→ 2, 3, 0
3	Hit		First-in→ 2, 3, 0
1	Miss	2	First-in→ 3, 0, 1
2	Miss	3	First-in→ 0, 1, 2
1	Hit		First-in→ 0, 1, 2

Figure 22.2: **Tracing The FIFO Policy**

Belady's anomaly

Bigger caches can have lower hit rates!

Access	Hit/miss	Resulting cache	Access	Hit/miss	Resulting cache
1	Miss	1	**1**	Miss	1
2	Miss	1, 2	**2**	Miss	1, 2
3	Miss	1, 2, 3	**3**	Miss	1, 2, 3
4	Miss	2, 3, 4	**4**	Miss	1, 2, 3, 4
1	Miss	3, 4, 1	**1**	**Hit**	1, 2, 3, 4
2	Miss	4, 1, 2	**2**	**Hit**	1, 2, 3, 4
5	Miss	1, 2, 5	**5**	Miss	2, 3, 4, 5
1	**Hit**	1, 2, 5	**1**	Miss	3, 4, 5, 1
2	**Hit**	1, 2, 5	**2**	Miss	4, 5, 1, 2
3	Miss	2, 5, 3	**3**	Miss	5, 1, 2, 3
4	Miss	5, 3, 4	**4**	Miss	1, 2, 3, 4
5	**Hit**	5, 3, 4	**5**	Miss	2, 3, 4, 5

FIFO does not follow "stack property". Cache of size 4 may not contain elements in cache of size 3.

Fairness in page replacement

- Someone had lots of pages. I had very little. My page was evicted
- OS maintains “resident size” per process: 1, 7, 9
- First select a victim process with highest resident size, remove its pages

Least Recently Used (LRU)

- Most programs exhibit temporal locality:
  - If a page was accessed recently, it shall be accessed soon
  - Keep list according to recency

Access	Hit/Miss?	Evict	Resulting Cache State
0	Miss		LRU→ 0
1	Miss		LRU→ 0, 1
2	Miss		LRU→ 0, 1, 2
0	Hit		LRU→ 1, 2, 0
1	Hit		LRU→ 2, 0, 1
3	Miss	2	LRU→ 0, 1, 3
0	Hit		LRU→ 1, 3, 0
3	Hit		LRU→ 1, 0, 3
1	Hit		LRU→ 0, 3, 1
2	Miss	0	LRU→ 3, 1, 2
1	Hit		LRU→ 3, 2, 1

Figure 22.5: **Tracing The LRU Policy**

Difficulty in implementing LRU

- In FIFO, list is updated by the OS when a new page is allocated, or when a page is swapped out
- In LRU, list needs to be updated at every access
  - OS is not running during page accesses :-/

Access	Hit/Miss?	Evict	Resulting Cache State
0	Miss		LRU→ 0
1	Miss		LRU→ 0, 1
2	Miss		LRU→ 0, 1, 2
0	Hit		LRU→ 1, 2, 0
1	Hit		LRU→ 2, 0, 1
3	Miss	2	LRU→ 0, 1, 3
0	Hit		LRU→ 1, 3, 0
3	Hit		LRU→ 1, 0, 3
1	Hit		LRU→ 0, 3, 1
2	Miss	0	LRU→ 3, 1, 2
1	Hit		LRU→ 3, 2, 1

Figure 22.5: **Tracing The LRU Policy**

Implementation options

- Option 1: Hardware maintains the LRU list
  - 4GB / 4KB ~ 2^{20} pages
  - List size: 20 bits * 2^{20} pages ~ 3MB
  - List cannot be in CPU => must be in memory
  - Each memory access causes another set of memory accesses to update list

Implementation options

- Option 2: Hardware updates timestamp in page table entries. OS scans PTEs to find page with oldest timestamp
- PTEs live in memory. Updating timestamp again touches memory at every access. Defeats TLB.
- Lazily update timestamp:
  - when mapping is brought to TLB
  - when mapping is evicted from TLB
  - Once in a while
- Need many more bits in PTE to store timestamp
- Victim process has highest resident size. Scanning (worst case 2^{20}) PTEs will be slow

Page table

Physical page number	Access Timestamp
10	8:11am
11	7:05am
15	7:00am
12	8:21am

TLB

Virtual page number	Physical page number
0	10
1	11
2	15
3	12

Approximating LRU

- Give up on finding *least* recently used. OK to evict a less recently used
- Hardware just lazily sets 1 access bit

Page table

Physical page number	Access Timestamp
10	8:11am
11	7:05am
15	7:00am
12	8:21am

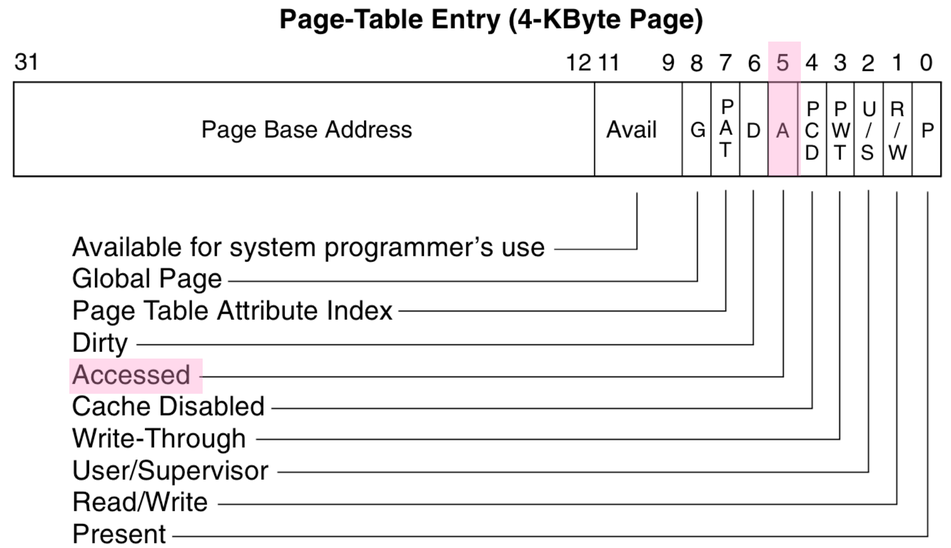
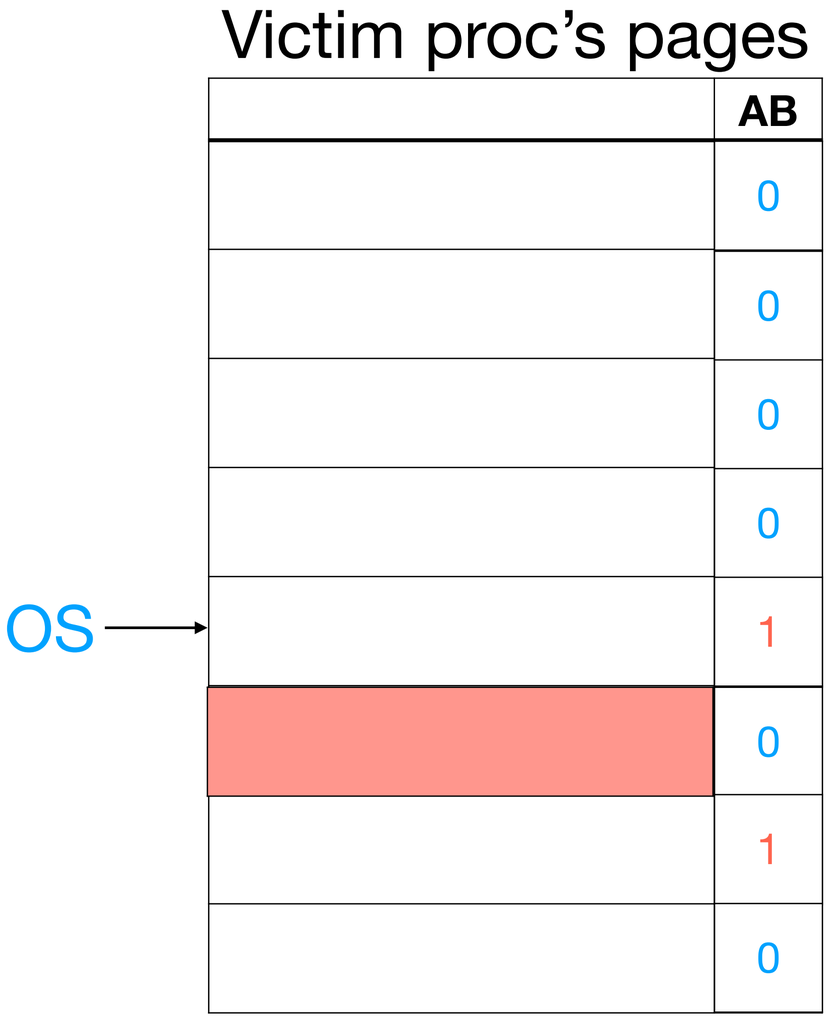


Figure 3-14. Format of Page-Directory and Page-Table Entries for 4-KByte Pages and 32-Bit Physical Addresses

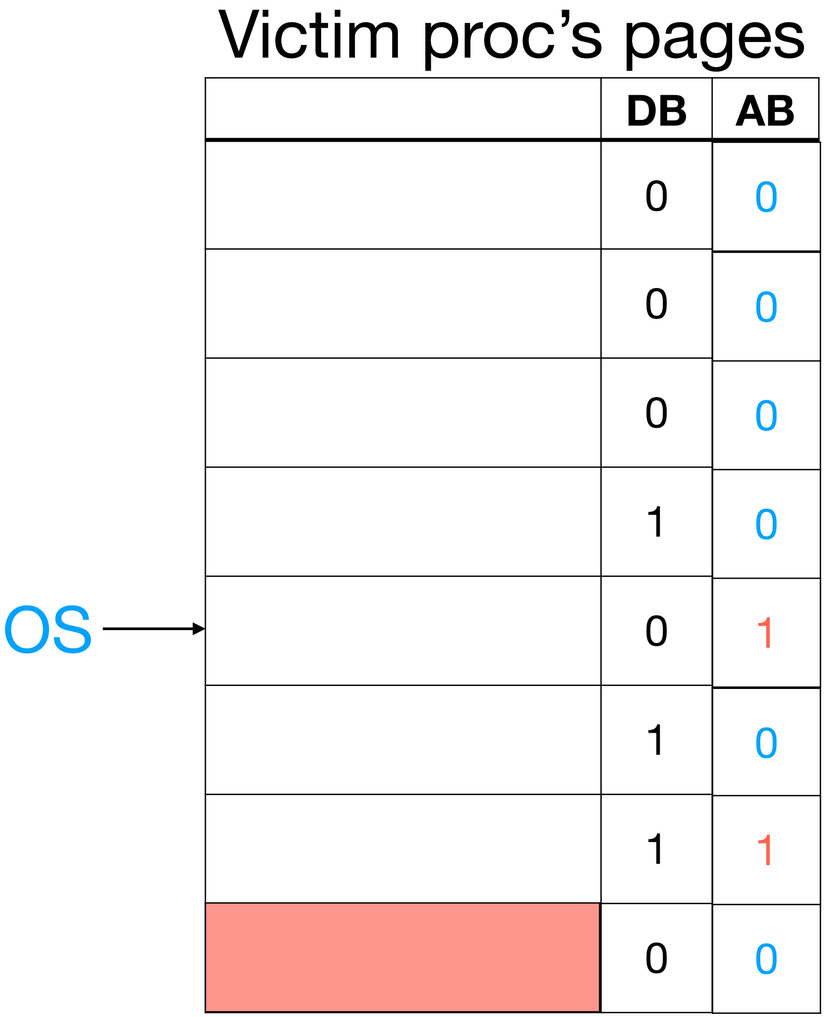
Clock algorithm

- OS clears access bit. Evicts page with access bit = 0
- Hardware sets access bit to 1
- Evicted page was “not recently used”

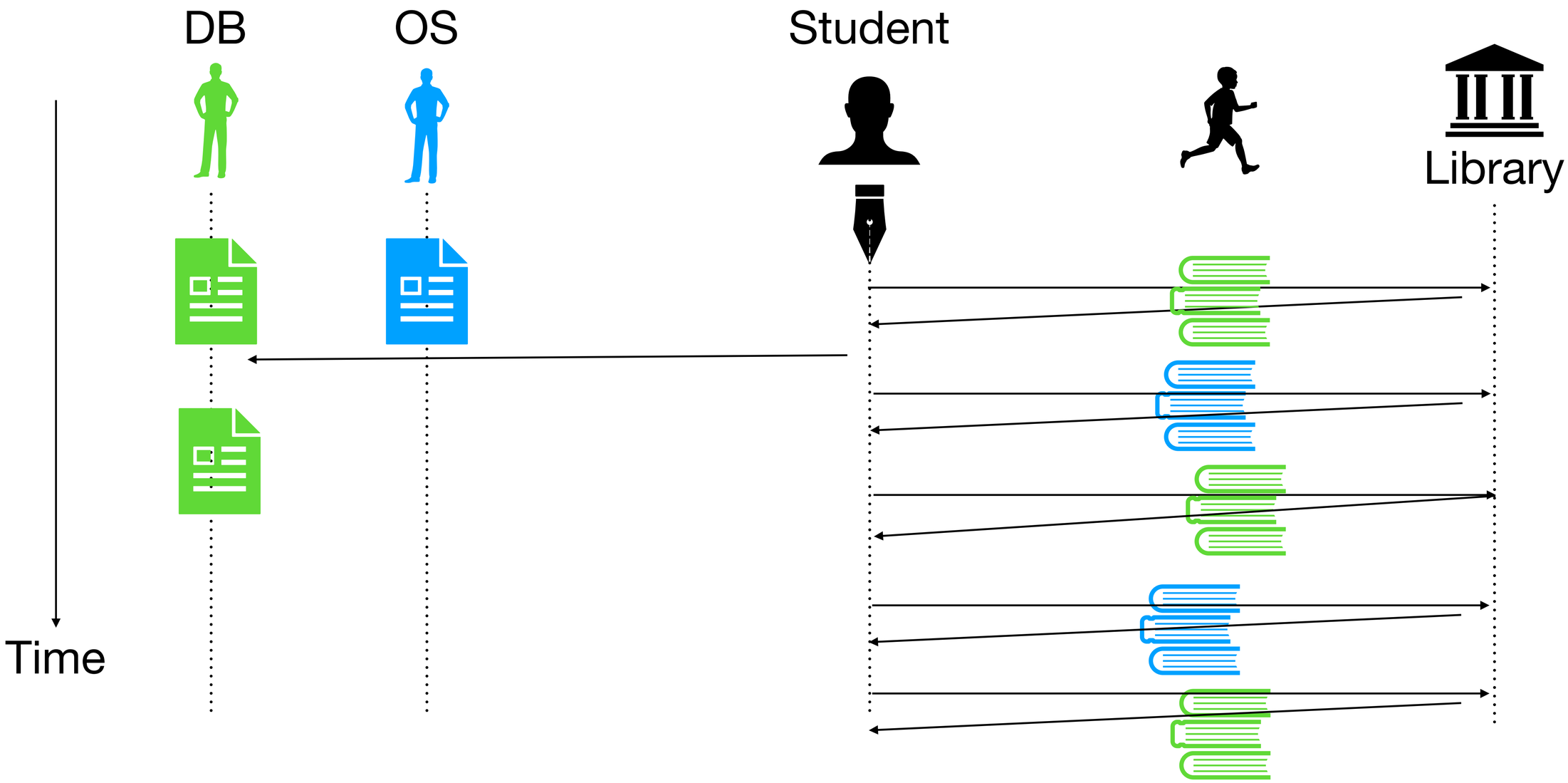


Clock algorithm (2)

- Optimisation:
  - We should prefer evicting page that has not changed since it was brought back from disk
  - Such pages can just be deleted without doing a copy
- OS clears dirty bit when it brings a page to memory from disk
- Evict dirty pages iff not able to find a clean page



Thrashing: Library analogy



Thrashing: Library analogy (2)

- Problem: Library only allows one book to be checked out. Student is constantly running to/from the library. Not able to do work on any assignment.
- Solution:
  - Reduce “working set”
    - I will work on OS assignment completely before worrying about DB assignment
    - I will not work on DB assignment
  - Buy a book to avoid going to library

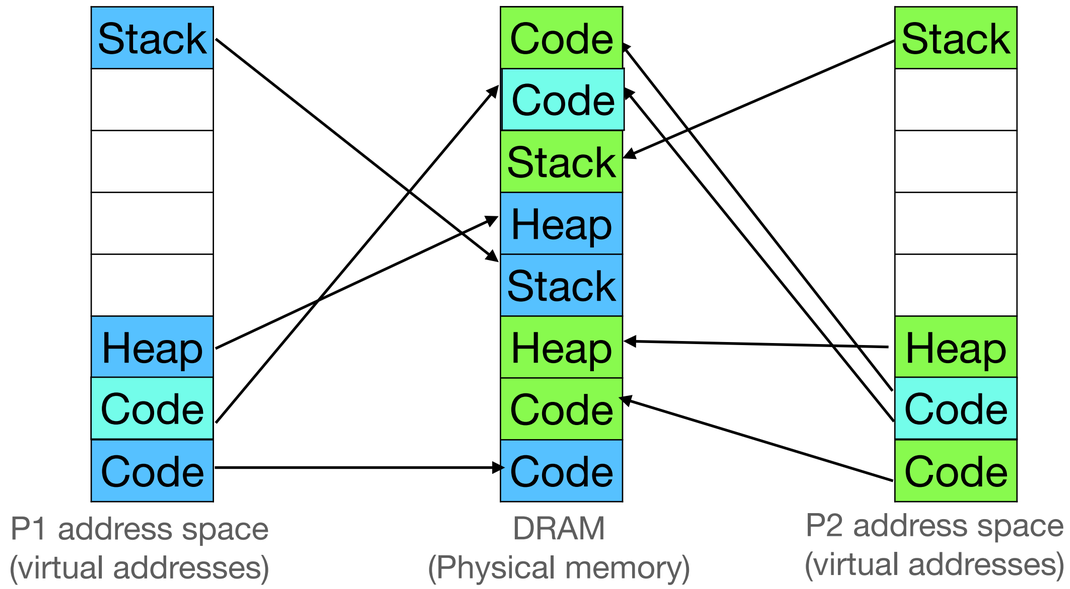
Thrashing

- Total working set of running processes is larger than physical memory. Constantly swapping in and out pages to/from disk. Not able to work.
- Solution:
  - Reduce working set
    - Admission control: run some processes for some time and then some other
    - Out-of-memory killer: Linux kills most memory intensive process
  - Buy more memory to avoid copying to/from disk

Reducing memory pressure

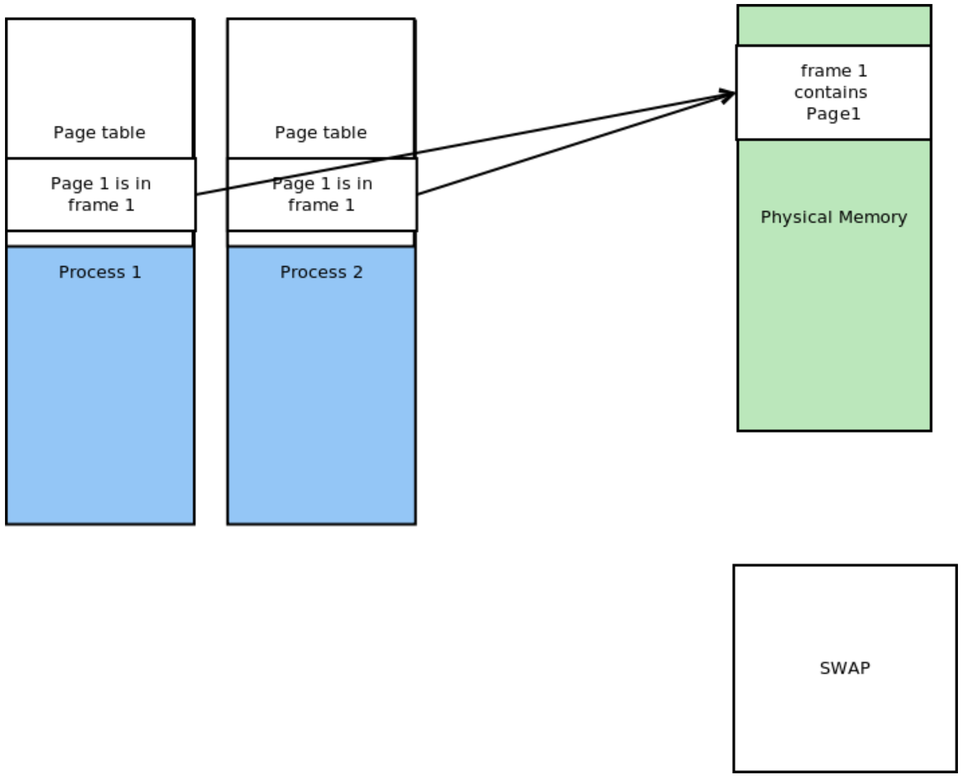
Kernel same page merging (KSM)

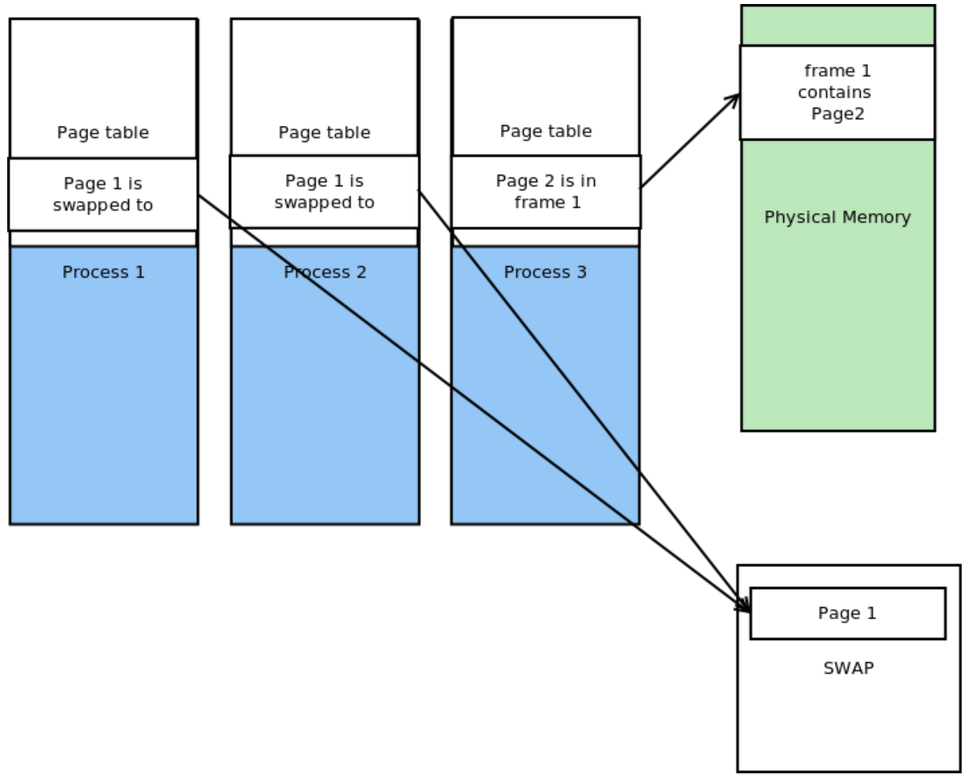
- OS periodically scans a few pages and computes their hash
- If two page hashes are same, deduplicate
  - Change PTE of one process to point to the common page
  - Add duplicate page to free list



Shared pages and demand paging

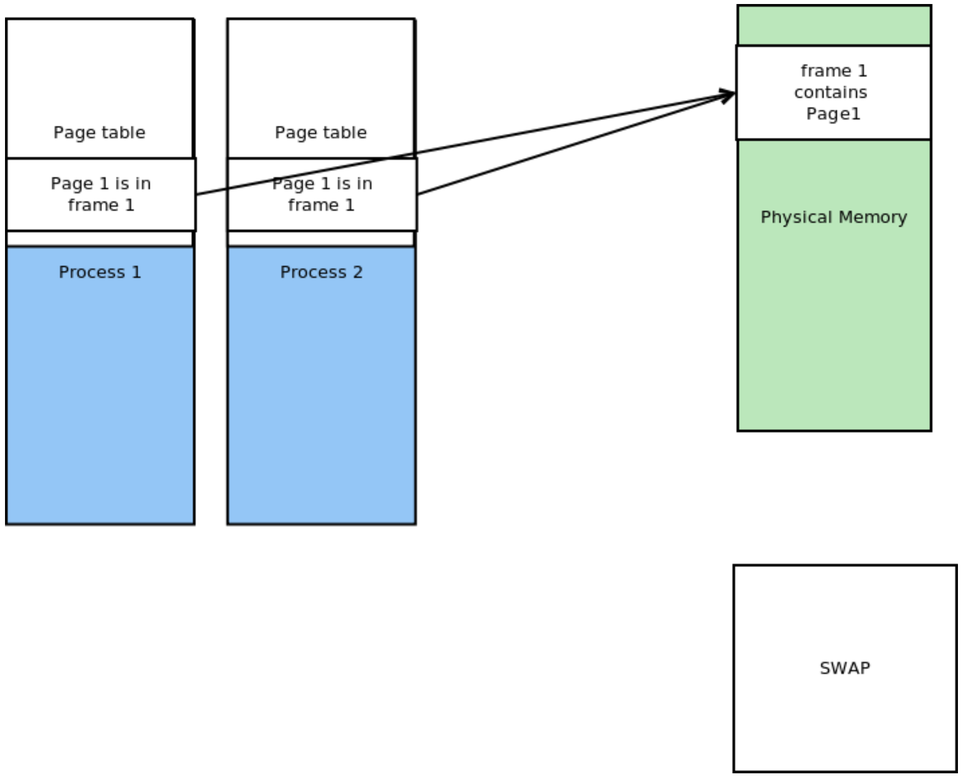
- Need to update multiple PTEs when pages are swapped out/in

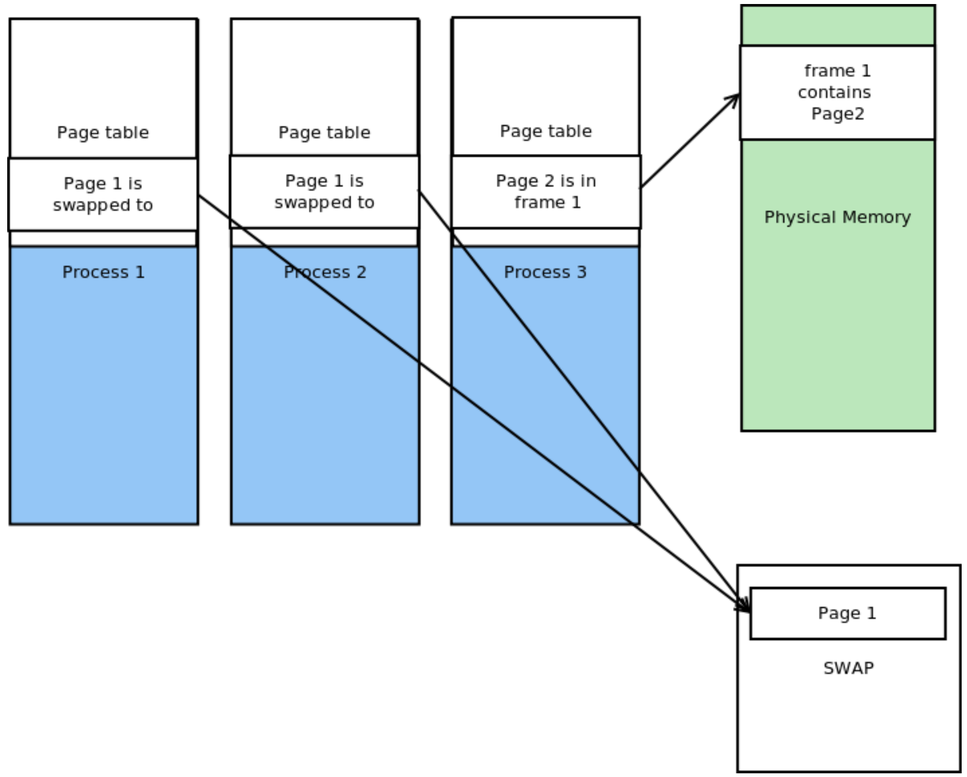




Reverse maps

- When swapping in/out a shared page, all PTEs must be updated.
- rmap: PPN -> list[*PTE]

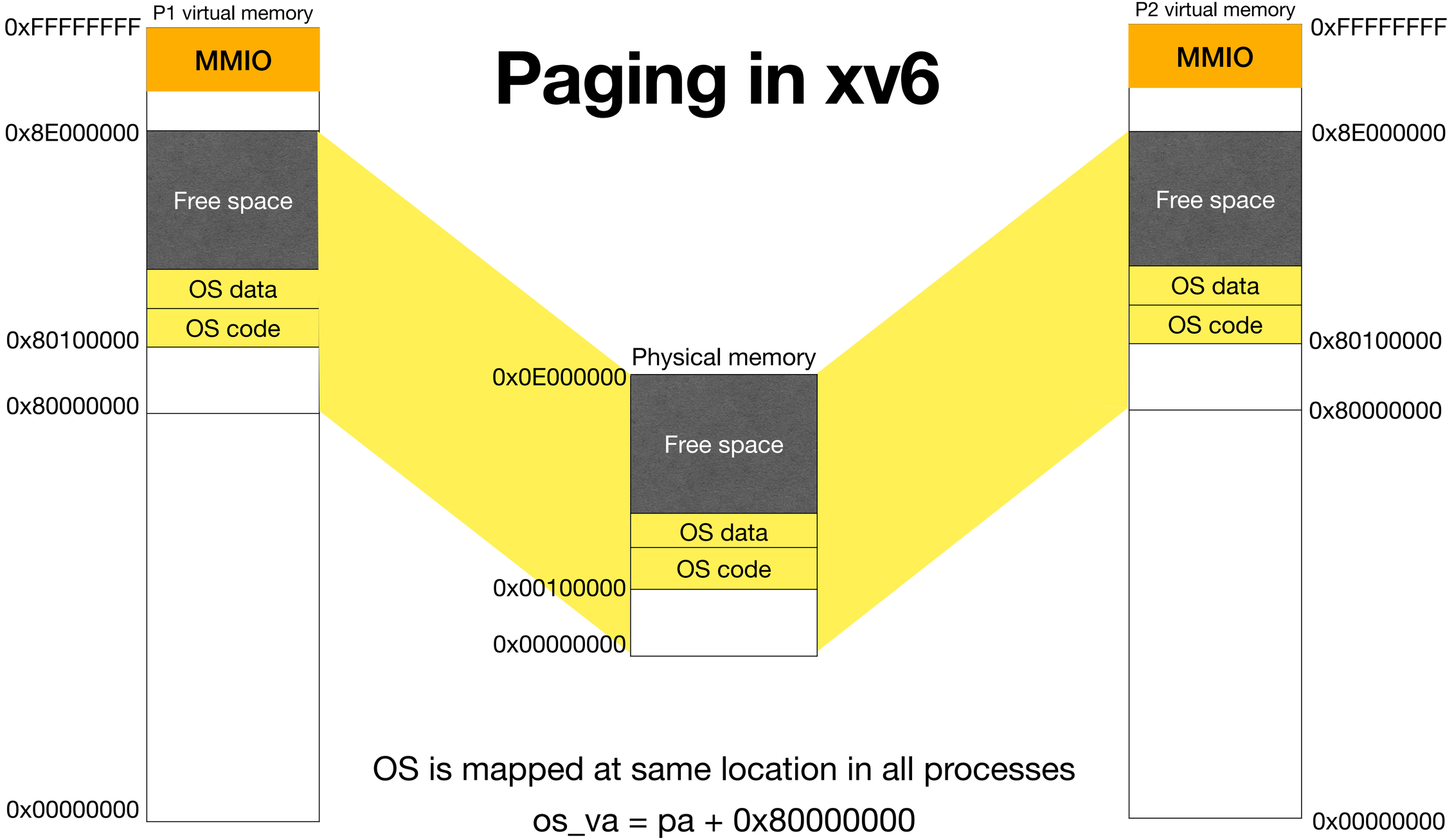




Paging in action in xv6

xv6 book Ch 2

Paging in xv6



OS is mapped at same location in all processes

os_va = pa + 0x80000000

Mapping OS into process address space

- User/supervisor bit is not set for OS pages

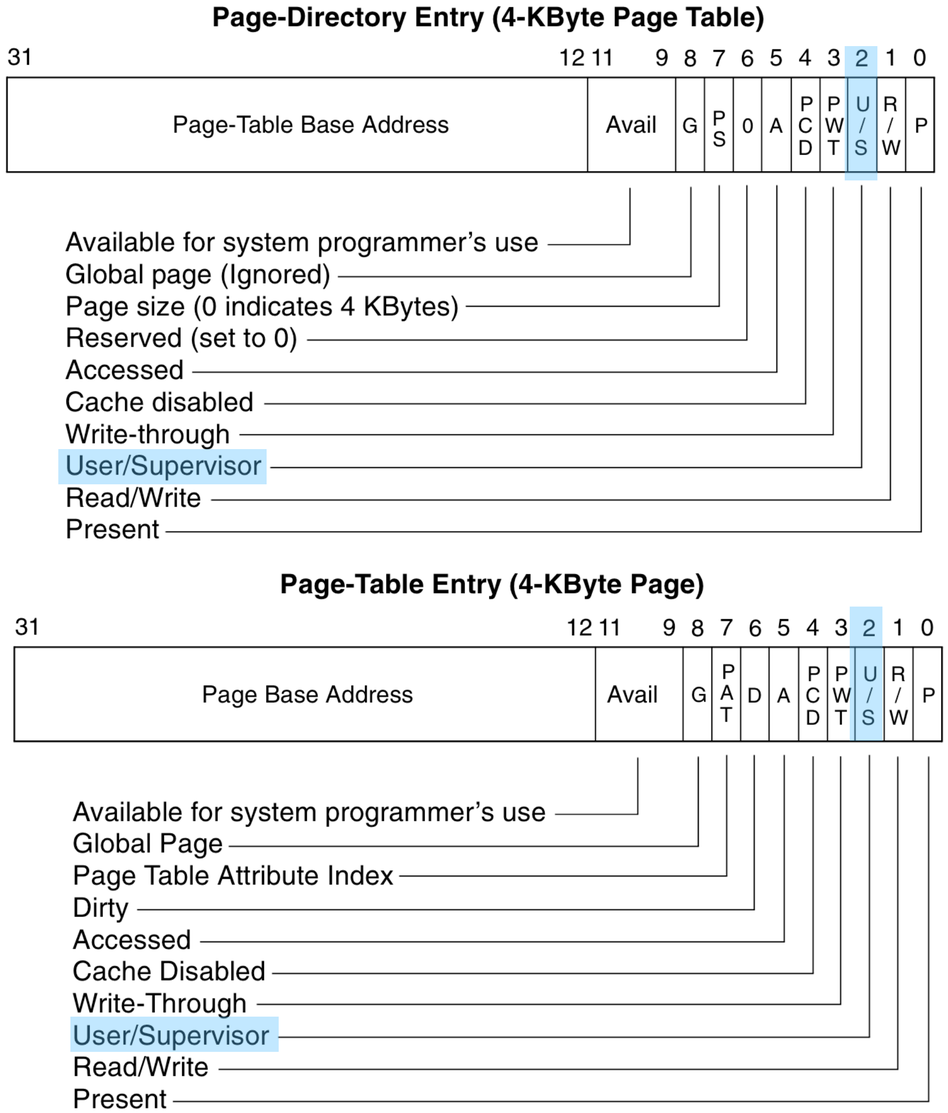
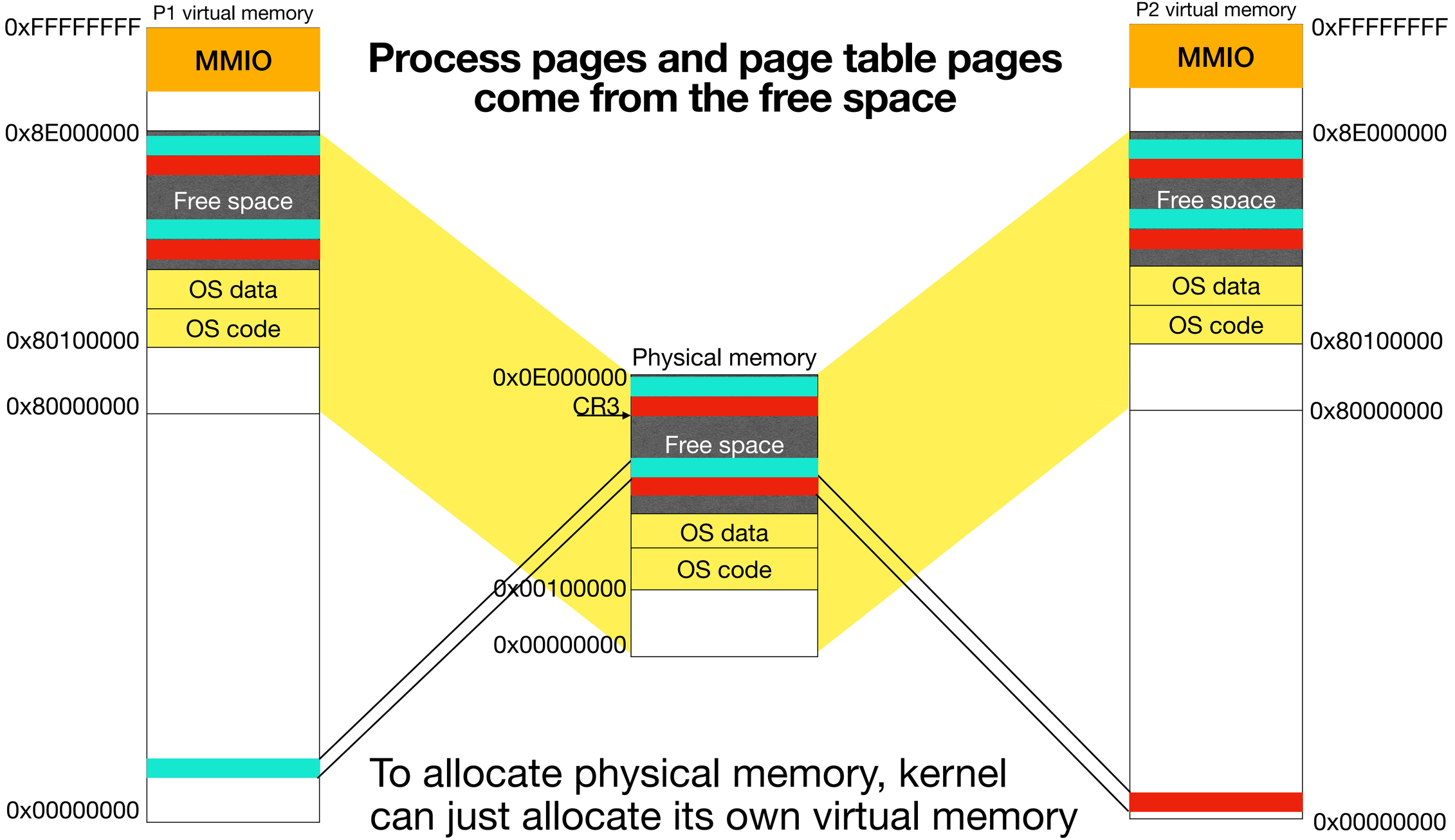


Figure 3-14. Format of Page-Directory and Page-Table Entries for 4-KByte Pages and 32-Bit Physical Addresses

Process pages and page table pages come from the free space

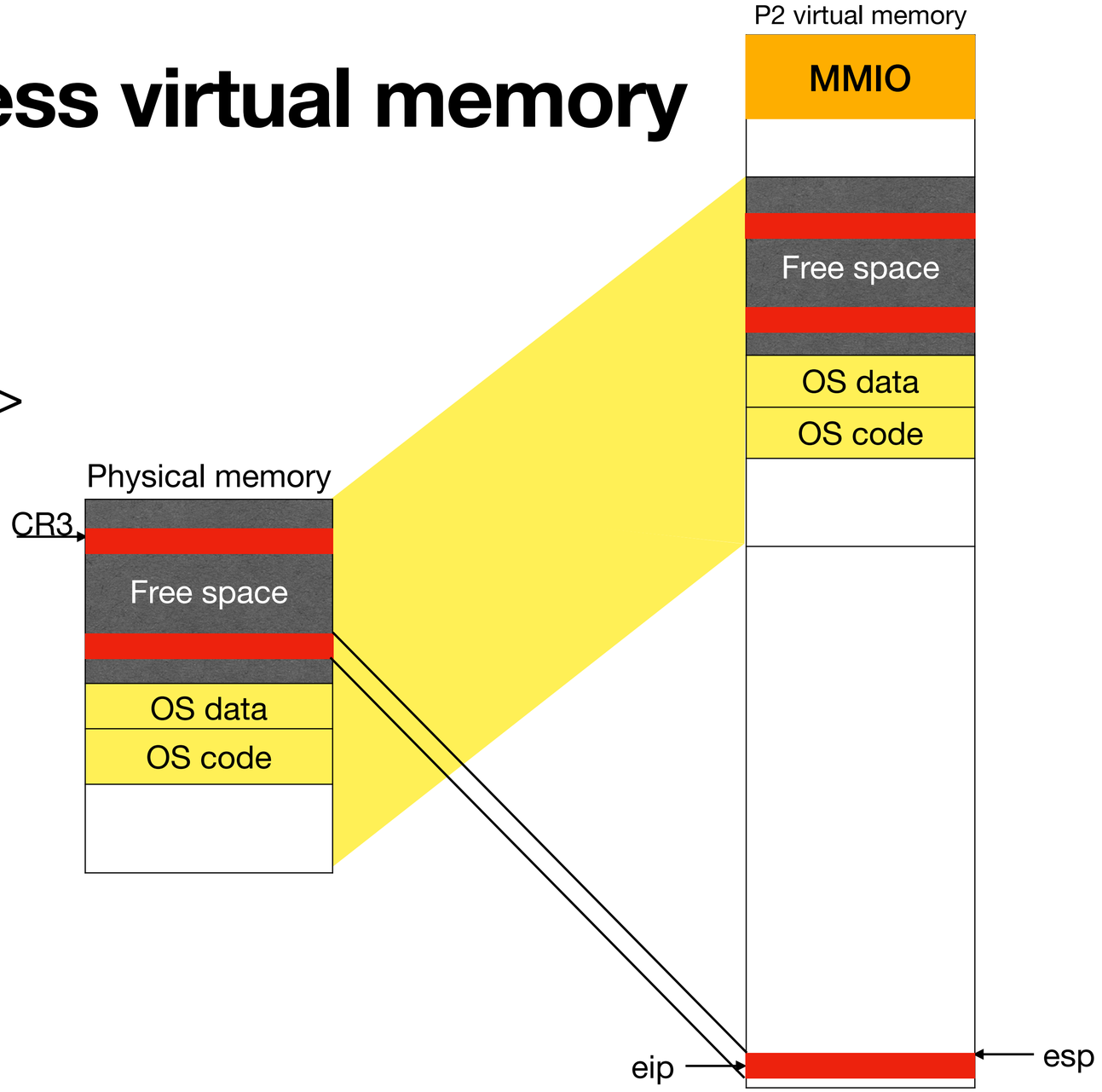


To allocate physical memory, kernel can just allocate its own virtual memory

Mapping OS into process virtual memory

Trap handling

- Kernel stack and IDT are in the address space of the process. => Hardware need not switch page tables for handling interrupts

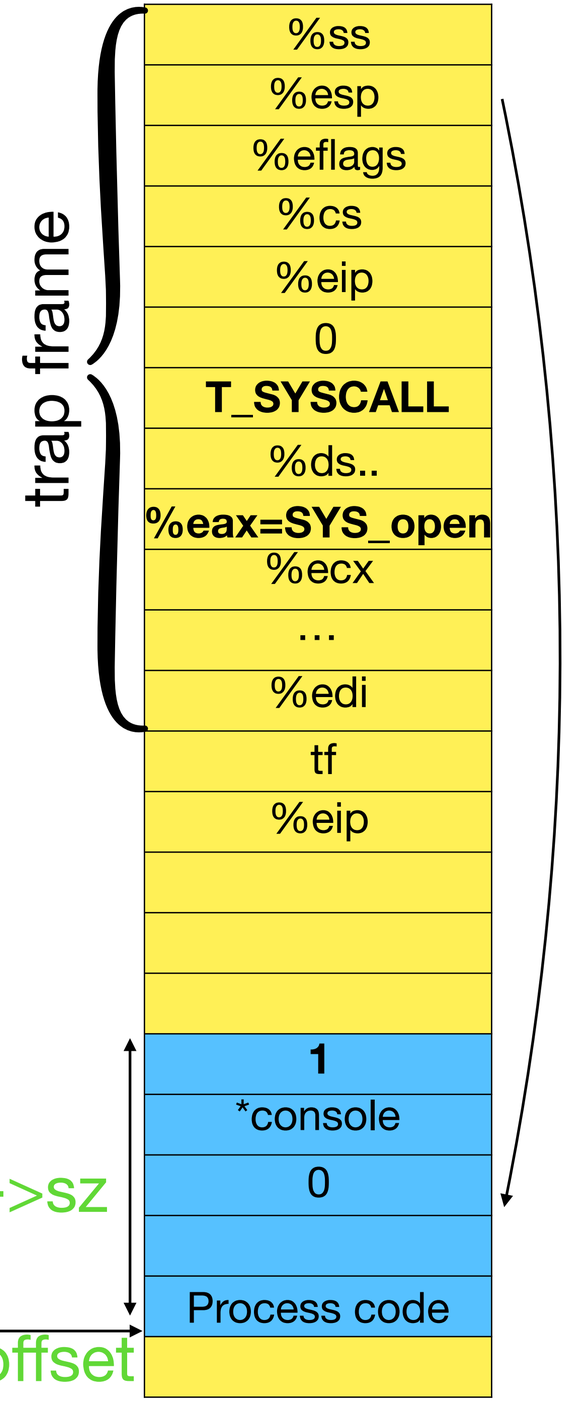


Visualising syscall handling p19-syscall

```
# sys_open("console", O_WRONLY)
          pushl $1
          pushl $console
          pushl $0
          movl $SYS_open, %eax
          int $T_SYSCALL
          pushl %eax
```

```
int sys_open(void) {
  int fd, omode;
  if(argint(1, &omode) < 0) {
    return -1;
  }
  ..
  return fd;
}
```

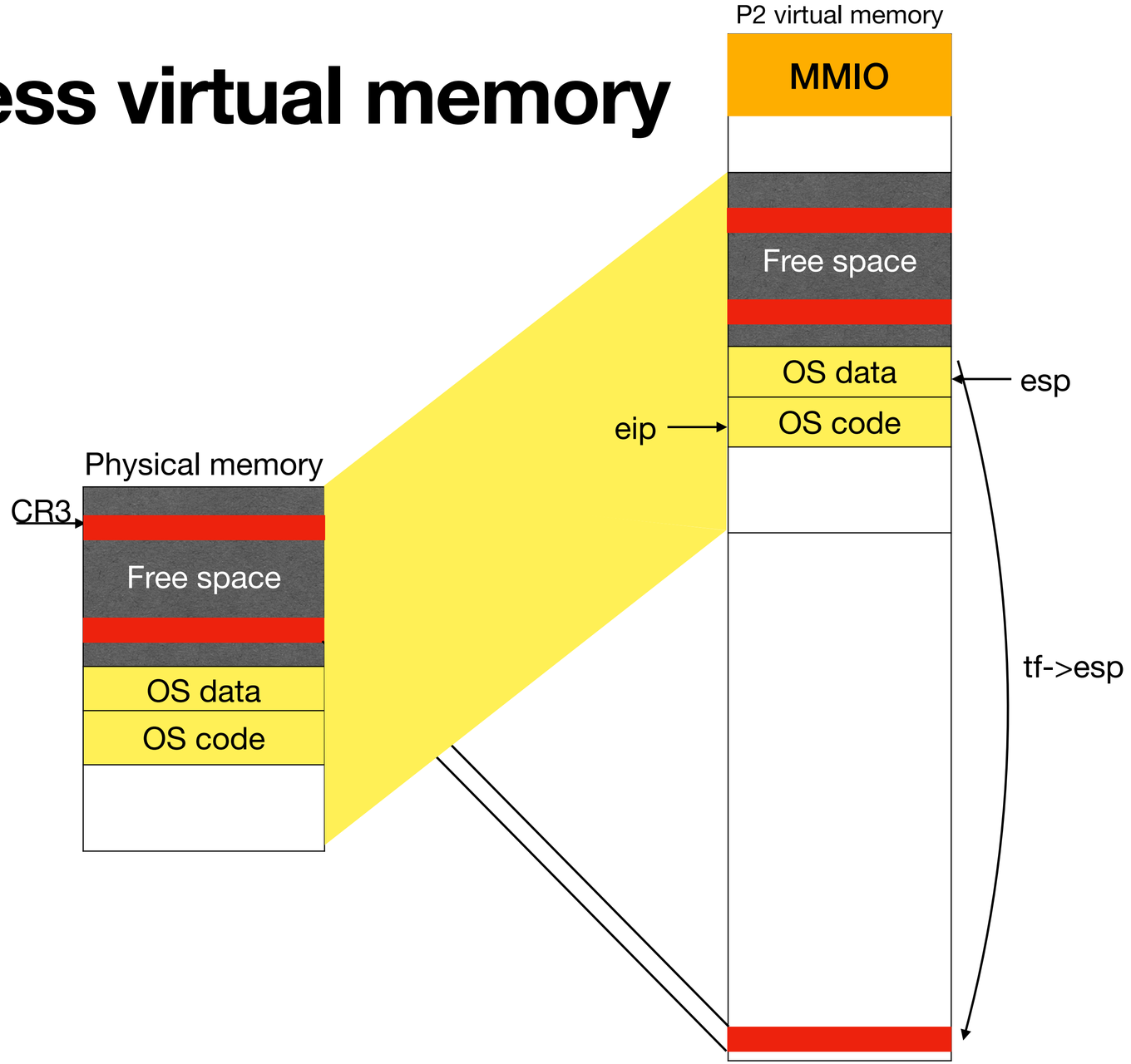
```
int fetchint(uint addr, int *ip) {
  if(addr >= p->sz || addr+4 > p->sz)
    return -1;
  *ip = *(int*)(addr + p->offset);
}

int argint(int n, int *ip) {
  return fetchint((myproc()->tf->esp)
                  + 4 + 4*n, ip);
}
void syscall(void) {
  int num = curproc->tf->eax;
  curproc->tf->eax = syscalls[num]();
}
```



Mapping OS into process virtual memory

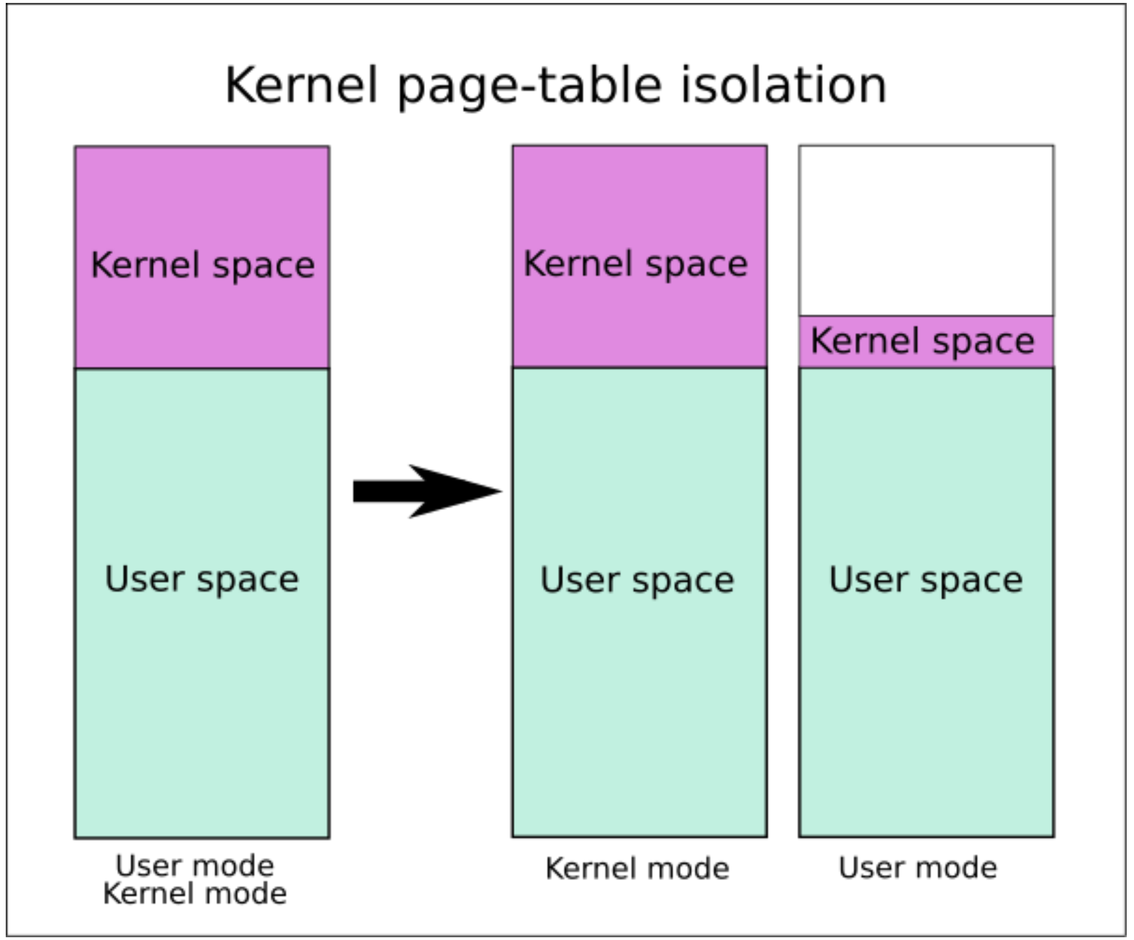
Reading sys call parameters

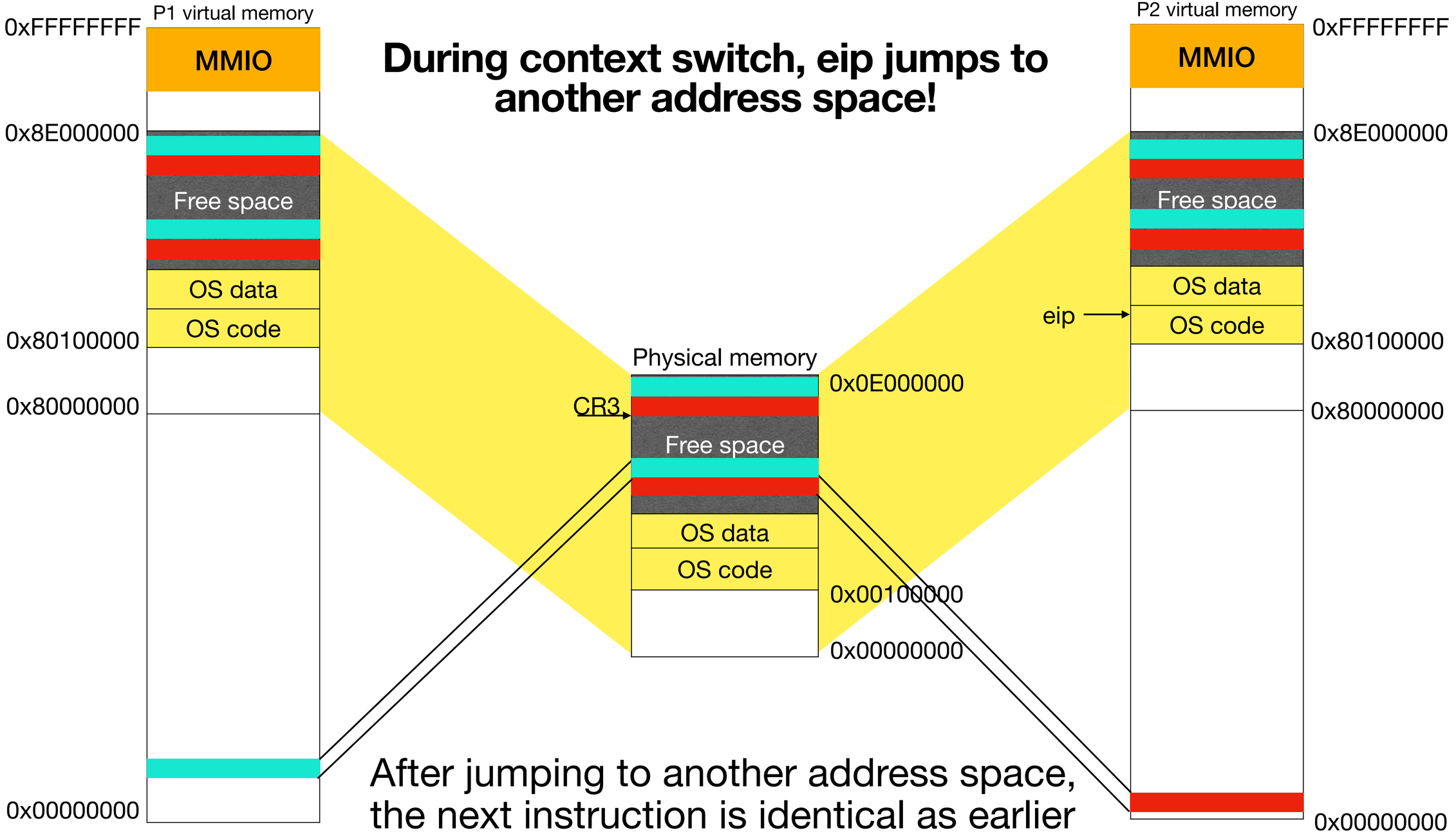
```
int fetchint(uint addr, int *ip) {
  if(addr >= p->sz || addr+4 > p->sz)
    return -1;
  *ip = *(int*)(addr + p->offset);
}

int argint(int n, int *ip) {
  return fetchint((myproc()->tf->esp)
                  + 4 + 4*n, ip);
}
```



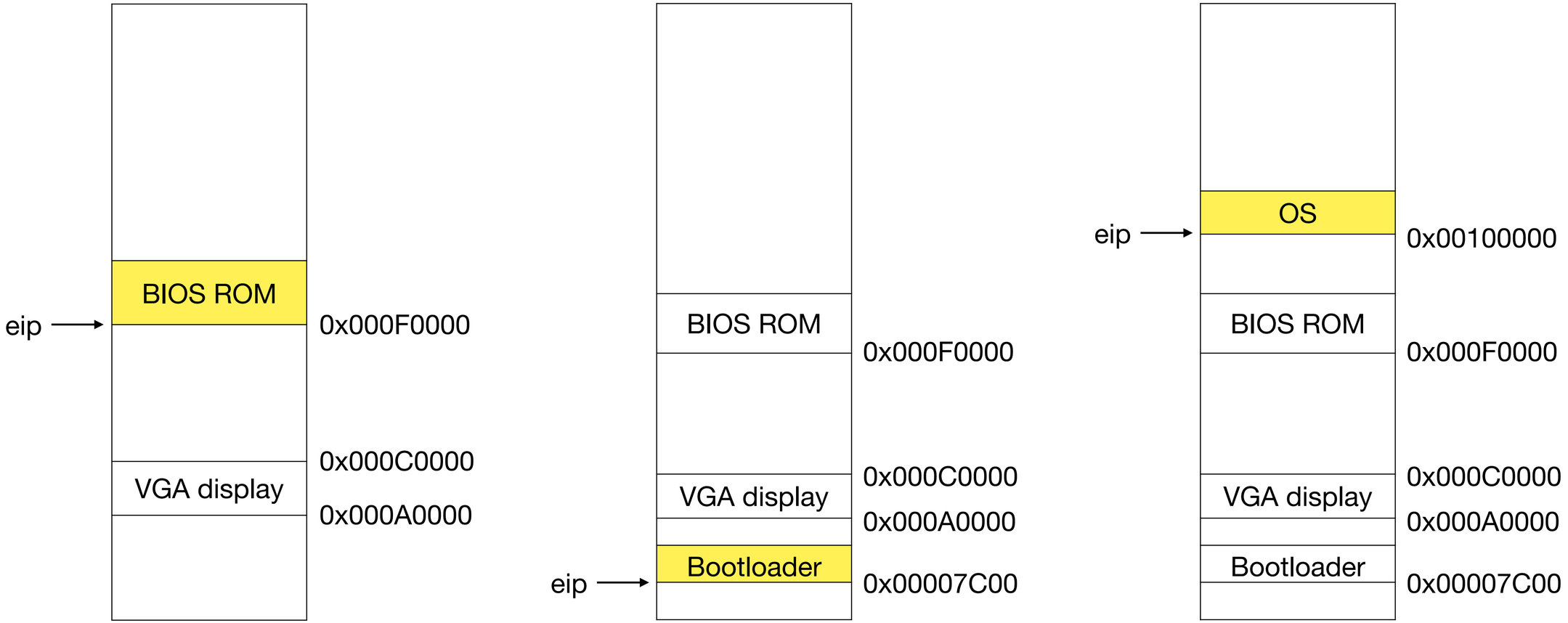
Kernel page table isolation (KPTI)

- Need to switch page tables to handle interrupts/syscalls
  - Syscall and interrupt heavy workloads like Postgres see 7-17% overheads (16-23% without tagged TLB)




During context switch, eip jumps to another address space!
P1 virtual memory
0xFFFFFFFF
MMIO
0x8E000000
Free space
OS data
OS code
0x80100000
0x80000000
0x00000000
Physical memory
0x0E000000
CR3
Free space
OS data
OS code
0x00100000
0x00000000
P2 virtual memory
0xFFFFFFFF
MMIO
0x8E000000
Free space
OS data
eip
OS code
0x80100000
0x80000000
0x00000000
After jumping to another address space,
the next instruction is identical as earlier

Boot up sequence: BIOS to bootloader to OS



Kernel has different physical and virtual addresses

- kernel.ld declares virtual address 0x80100000, physical address 0x100000
- kernel.ld marks _start as entry point. _start is V2P_WO(entry) i.e, (0x8010000c - 0x80000000)
- Running readelf -l kernel shows

```
$ readelf -l kernel

Elf file type is EXEC (Executable file)
Entry point 0x10000c
There are 3 program headers, starting at offset 52

Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  LOAD           0x001000 0x80100000 0x00100000 0x07aab 0x07aab R E 0x1000
  LOAD           0x009000 0x80108000 0x00108000 0x02516 0x0d4a8 RW  0x1000
  GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RWE 0x10

Section to Segment mapping:
  Segment Sections...
   00     .text .rodata
   01     .data .bss
   02
```

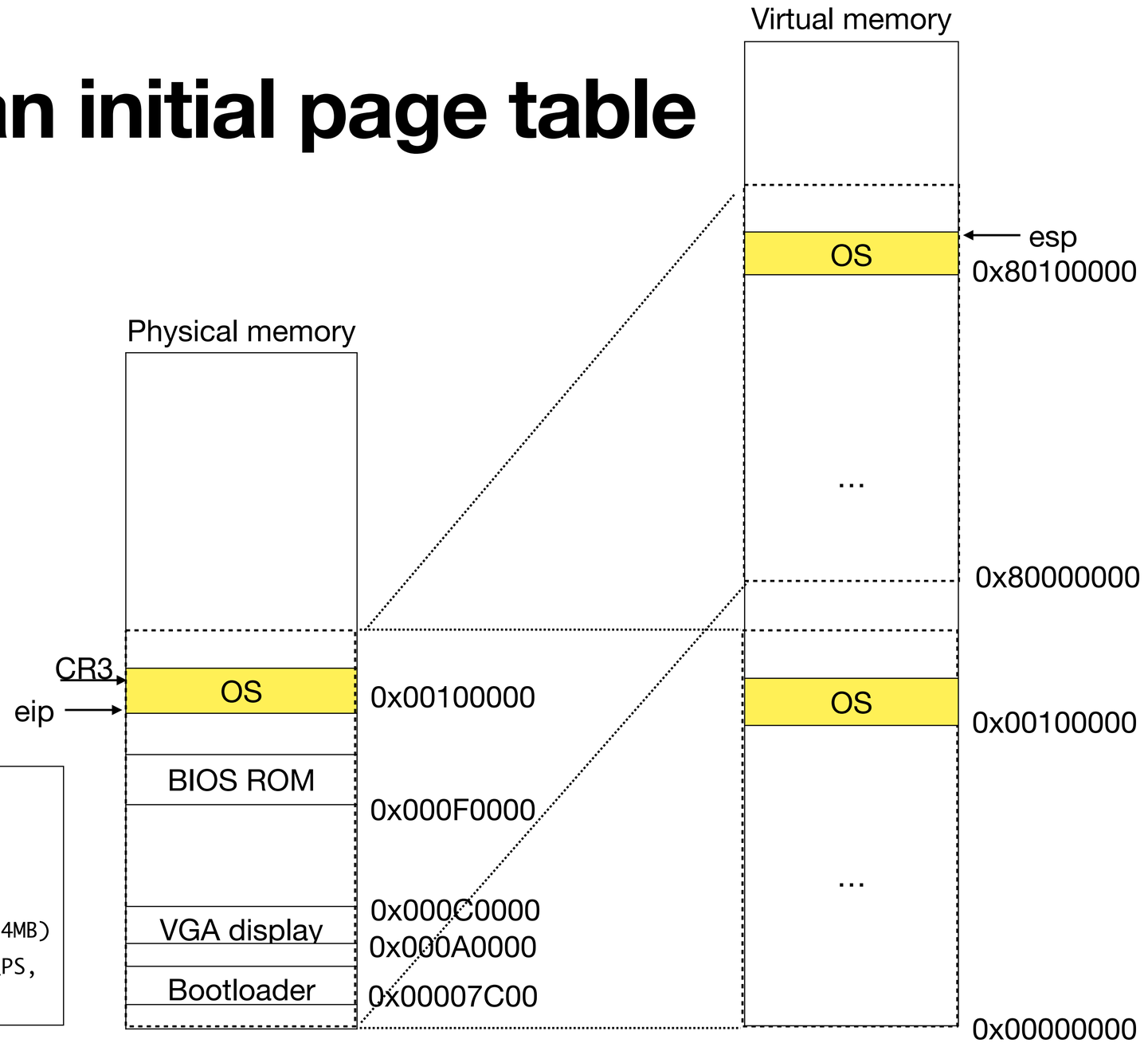
entry.S sets up an initial page table

```
entry:
  # Set page directory
  movl    $(V2P_WO(entrypgdir)), %eax
  movl    %eax, %cr3
  # Turn on paging.
  movl    %cr0, %eax
  orl     $(CR0_PG|CR0_WP), %eax
  movl    %eax, %cr0

  movl $(stack + KSTACKSIZE), %esp
  mov $main, %eax
  jmp *%eax

int main (void) {
  kinit1(end, P2V(4*1024*1024));
```

eip →

```
__attribute__((__aligned__(PGSIZE)))
pde_t entrypgdir[NPDENTRIES] = {
  // Map VA's [0, 4MB) to PA's [0, 4MB)
  [0] = (0) | PTE_P | PTE_W | PTE_PS,
  // Map VA's [KERNBASE, KERNBASE+4MB) to PA's [0, 4MB)
  [KERNBASE>>PDXSHIFT] = (0) | PTE_P | PTE_W | PTE_PS,
};
```

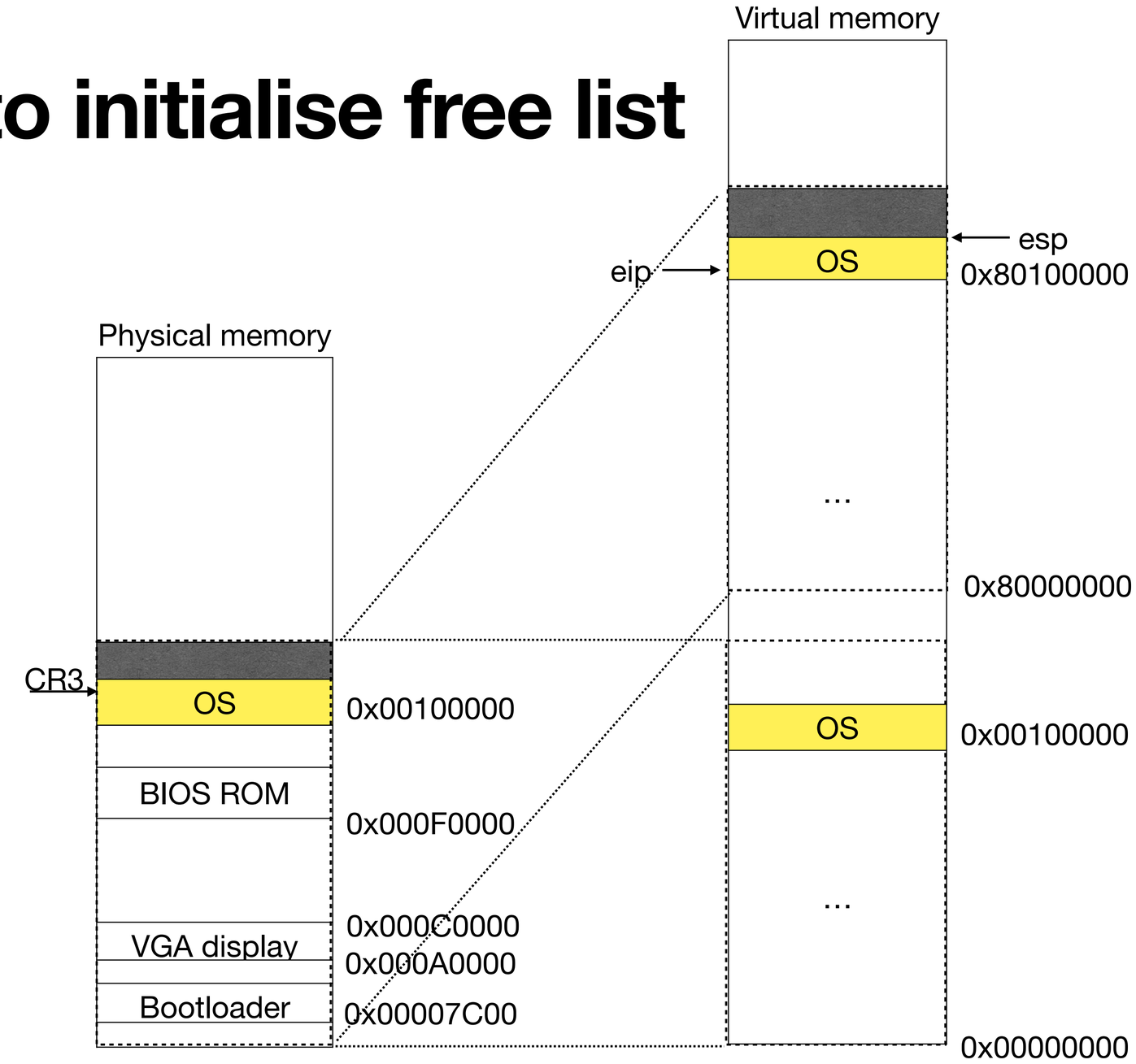
CR3 →



main calls kinit1 to initialise free list

```
int main (void) {
  kinit1(end, P2V(4*1024*1024));
  …
}
```

Now pages can be allocated from the free list!



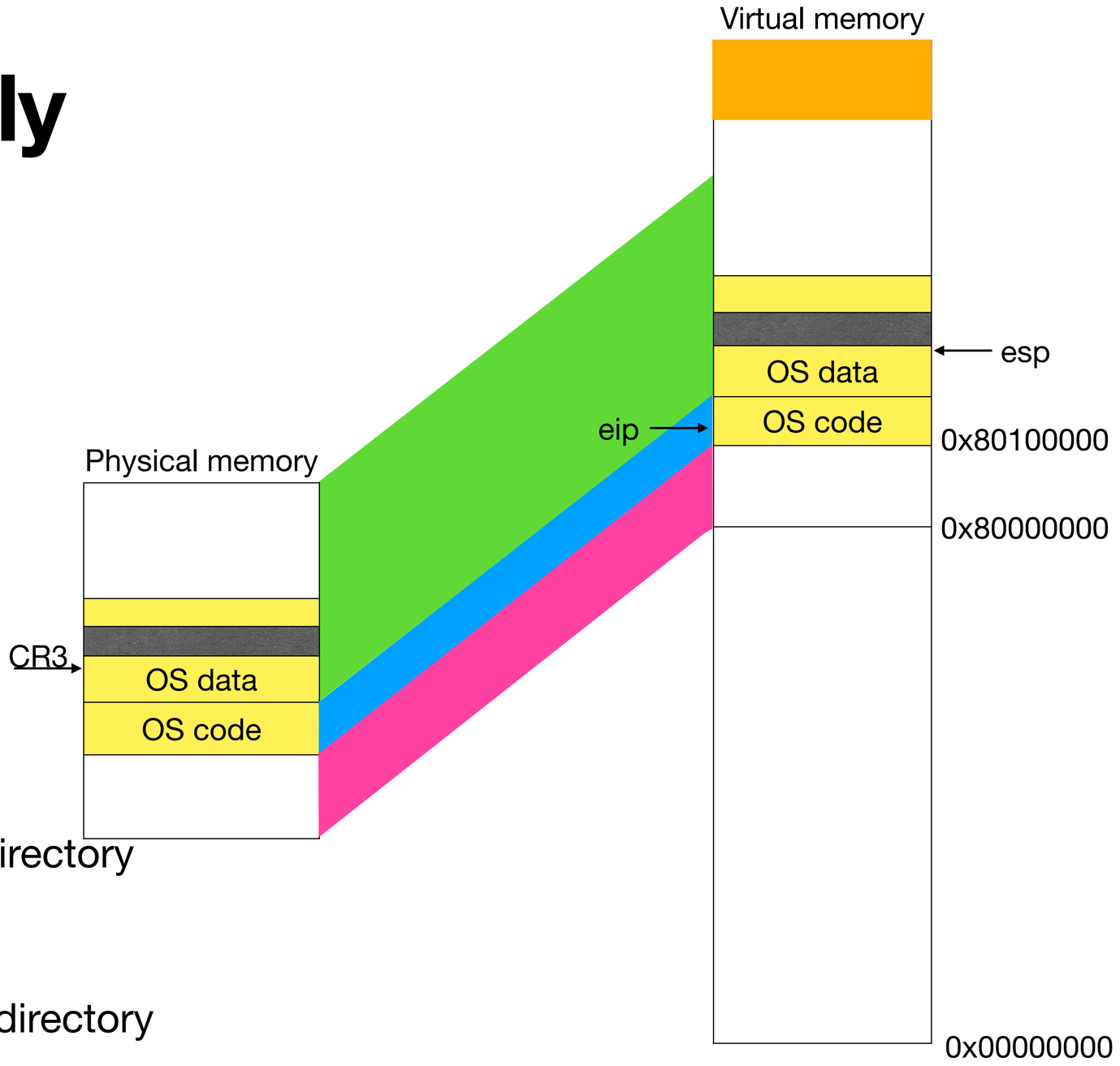
Mark code read-only

Remove identity mapping

```
static struct kmap {
  void *virt;
  uint phys_start;
  uint phys_end;
  int perm;
} kmap[] = {
{ (void*)KERNBASE, 0,              EXTMEM,    PTE_W},
{ (void*)KERNLINK, V2P(KERNLINK), V2P(data), 0},
{ (void*)data,     V2P(data),     PHYSTOP,   PTE_W},
{ (void*)DEVSPACE, DEVSPACE,      0,         PTE_W},
};
```

main.c calls kvmalloc in vm.c

- setupkvm allocates a new page for page directory
- mappages adds PTEs to map four areas
- switchkvm changes CR3 to the new page directory



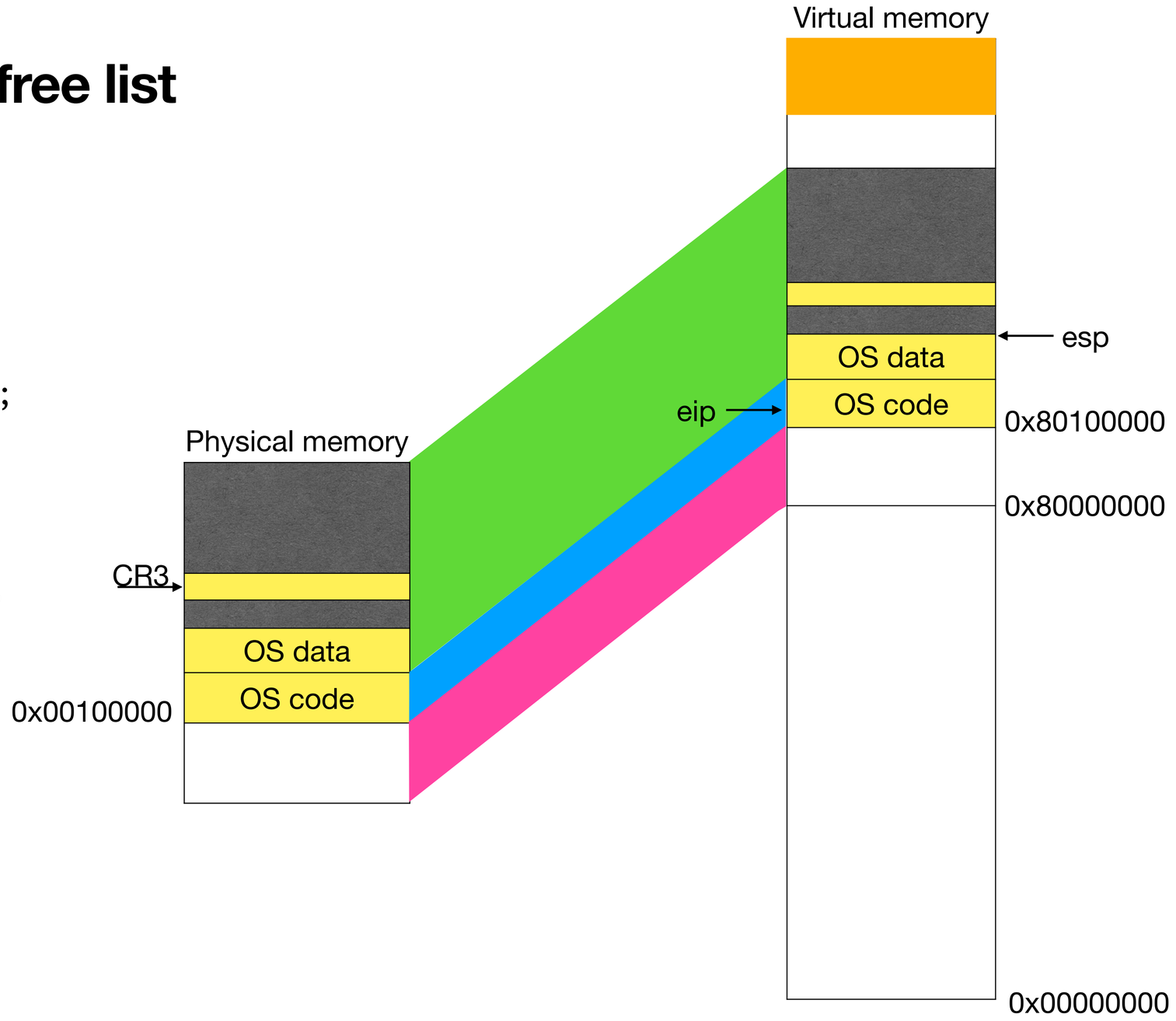
Mapping pages

- mappages takes page directory, virtual address, size, physical address, and permissions
  - It calls walkpgdir with alloc=1 (to allocate page table pages if they do not exist) to find the page table entry
  - It puts physical address in the PTE, marks it as present, and puts other permissions on PTE
- walkpgdir:
  - mimics hardware's page table walk. It takes first 10 bits to index into page directory to find page table page. It takes next 10 bits to index into page table page. It returns page table entry.
  - If page table page does not exist, it allocates a new page and adds it to page directory

main calls kinit2 to expand free list

```
int main (void) {
  kinit1(end, P2V(4*1024*1024));
  kvmalloc();
  …
  kinit2(P2V(4*1024*1024), P2V(PHYSTOP);
}
```

Rest of the physical memory is made available to allocator

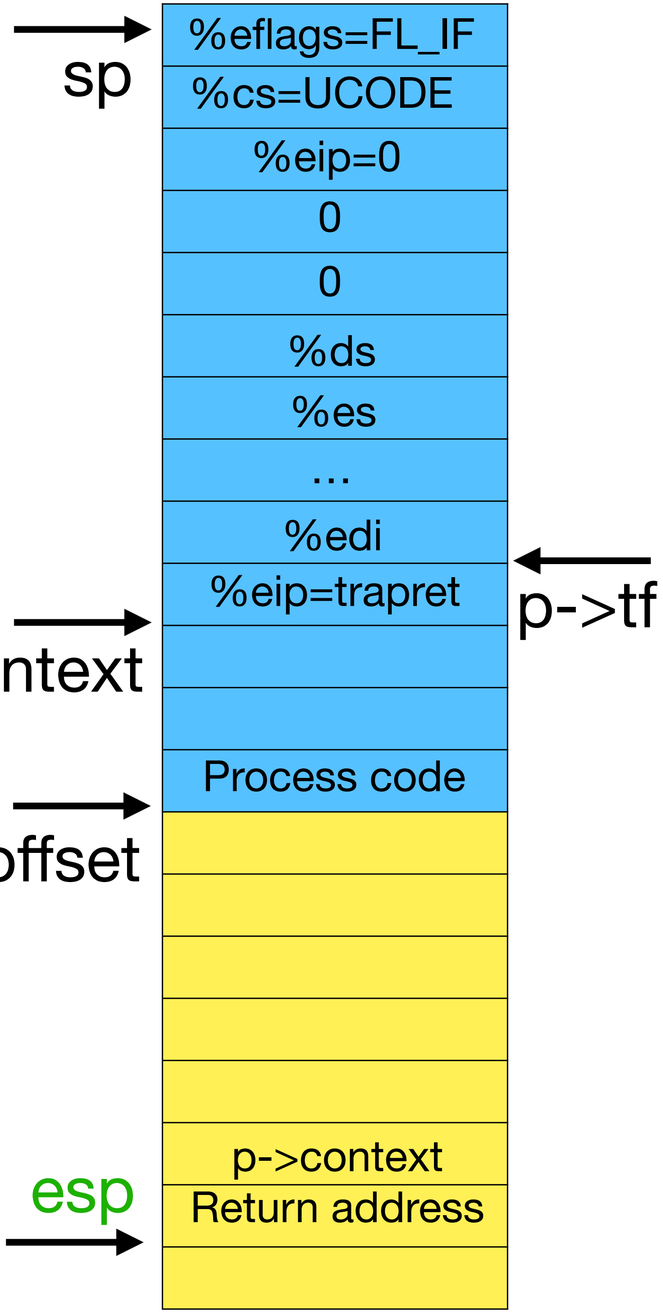


Setting up new process

```
pinit(){
  p = allocproc();
  memmove(p->offset, _binary_initcode_start,);
  p->tf->ds,es,ss = (SEG_UDATA<<3) | DPL_USR;
  p->tf->cs = (SEG_UCODE<<3) | DPL_USR;
  p->tf->eflags = FL_IF;
  p->tf->esp = PGSIZE;
  p->tf->eip = 0;
}
allocproc() {
  sp = (char*)(STARTPROC + (PROCSIZE<<12));
  sp -= sizeof *p->tf;
  p->tf = (struct trapframe*)sp;
  sp -= sizeof *p->context;
  p->context = (struct context*)sp;
  p->context->eip = (uint)trapret;
  return p;
}
```

eip

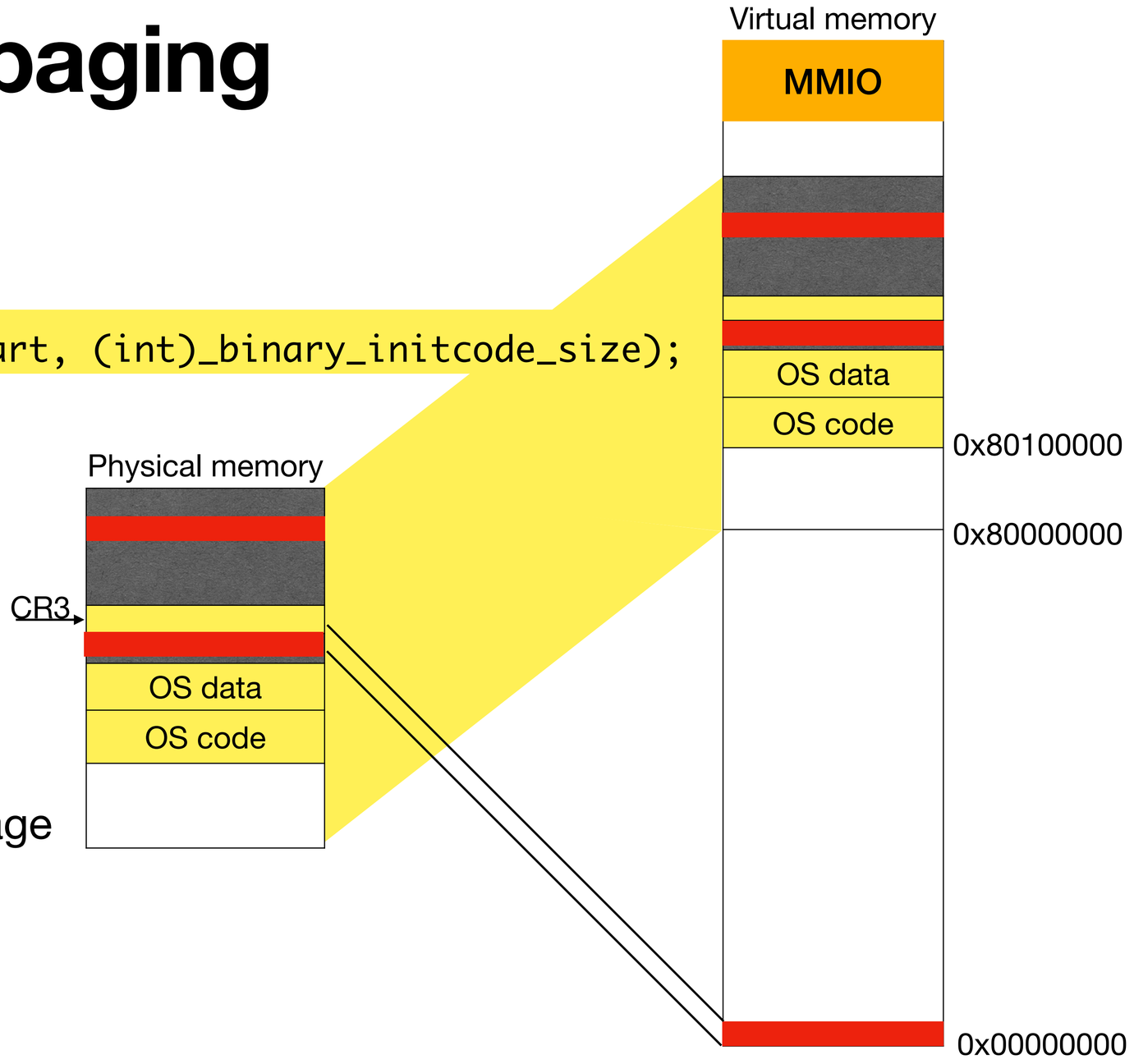
```
scheduler() {
  …
  swtch(p->context);
}
swtch:
  movl 4(%esp), %eax
  movl %eax, %esp
  movl $0, %eax
  ret

.globl trapret
trapret:
  popal
  popl %gs
  popl %fs
  popl %es
  popl %ds
  addl $0x8, %esp
  iret
```



Key changes from paging

```
pinit(){
  p = allocproc();
  p->pgdir = setupkvm();
  inituvm(p->pgdir, _binary_initcode_start, (int)_binary_initcode_size);
  p->sz = PGSIZE;
 …
}
```

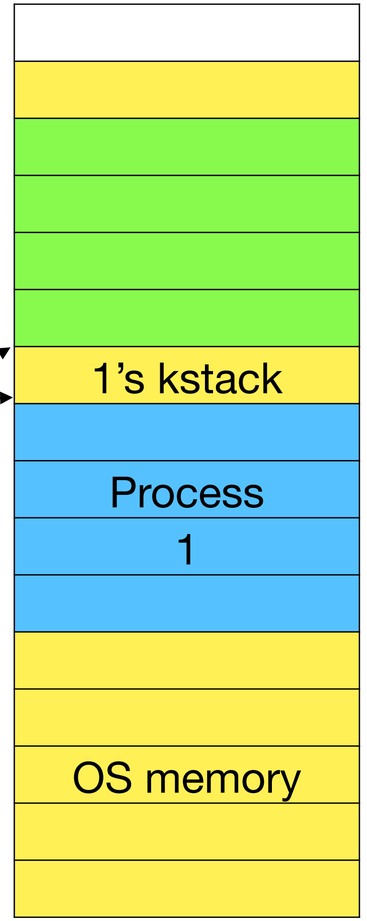
- inituvm:
  - allocates a page, clears it
  - adds the page to va=0 in process' page table. Notice that the user bit is set
  - copies code in the page



Setting up kernel stack

```
void seginit(void) {
  c->gdt[SEG_UCODE] = SEG(.., STARTPROC, (PROCSIZE-1) << 12, DPL_USER);
  c->gdt[SEG_UDATA] = SEG(.., STARTPROC, (PROCSIZE-1) << 12, DPL_USER);
}

static struct proc* allocproc(void) {
  sp = (char*)(STARTPROC + (PROCSIZE>>12));
  p->kstack = sp - KSTACKSIZE;
}

void switchuvm(struct proc *p) {
  mycpu()->gdt[SEG_TSS] = SEG16(STS_T32A, &mycpu()->ts,
                                sizeof(mycpu()->ts)-1, 0);
  mycpu()->ts.ss0 = SEG_KDATA << 3;
  mycpu()->ts.esp0 = (uint)p->kstack + KSTACKSIZE;
  ltr(SEG_TSS << 3);
}
```

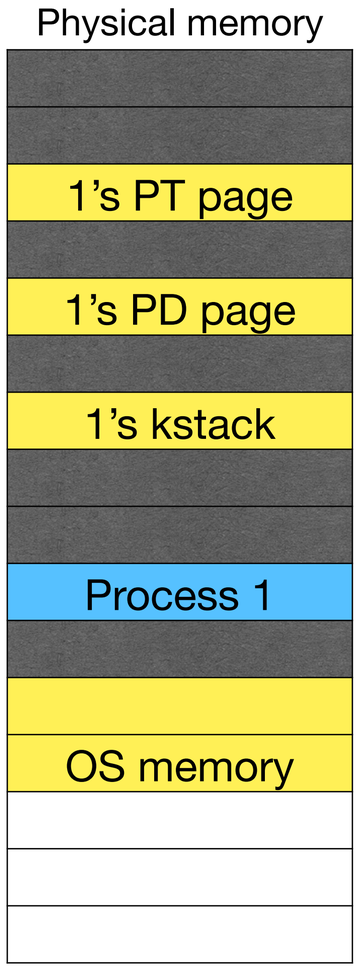
```
void scheduler(void) {
 // pick RUNNABLE process p
 switchuvm(p);
 swtch(p->context);
}
```



Key changes from paging

```
void seginit(void) {
  c->gdt[SEG_UCODE] = SEG(.., 0, 0xffffffff, DPL_USER);
  c->gdt[SEG_UDATA] = SEG(.., 0, 0xffffffff, DPL_USER);
}

static struct proc* allocproc(void) {
  p->kstack = kalloc();
  sp = p->kstack + KSTACKSIZE;
  …
}
void switchuvm(struct proc *p) {
  mycpu()->gdt[SEG_TSS] = SEG16(STS_T32A, &mycpu()->ts,
                                sizeof(mycpu()->ts)-1, 0);
  mycpu()->ts.ss0 = SEG_KDATA << 3;
  mycpu()->ts.esp0 = (uint)p->kstack + KSTACKSIZE;
  ltr(SEG_TSS << 3);
  lcr3(V2P(p->pgdir));  // switch to process address space
}
```

```
void scheduler(void) {
 // pick RUNNABLE process p
 switchuvm(p);
 swtch(p->context);
}
```



- User segments map to entire memory since protection is done via paging.
- kstack, process memory need not be contiguous
- switchuvm changes page tables

sbrk system call

- sys_sbrk calls growproc(n)
- growproc(n) calls (de)allocuvm
- allocuvm checks that process is not trying to grow into OS area, maps pages in page table with writeable, user-accessible bits
- deallocuvm deallocates pages one by one from newsz to oldsz. If page table page is not found, we move directly to next pde. If PTE is found and present, we free the physical page and change pte to zero.