Memory Management Basics

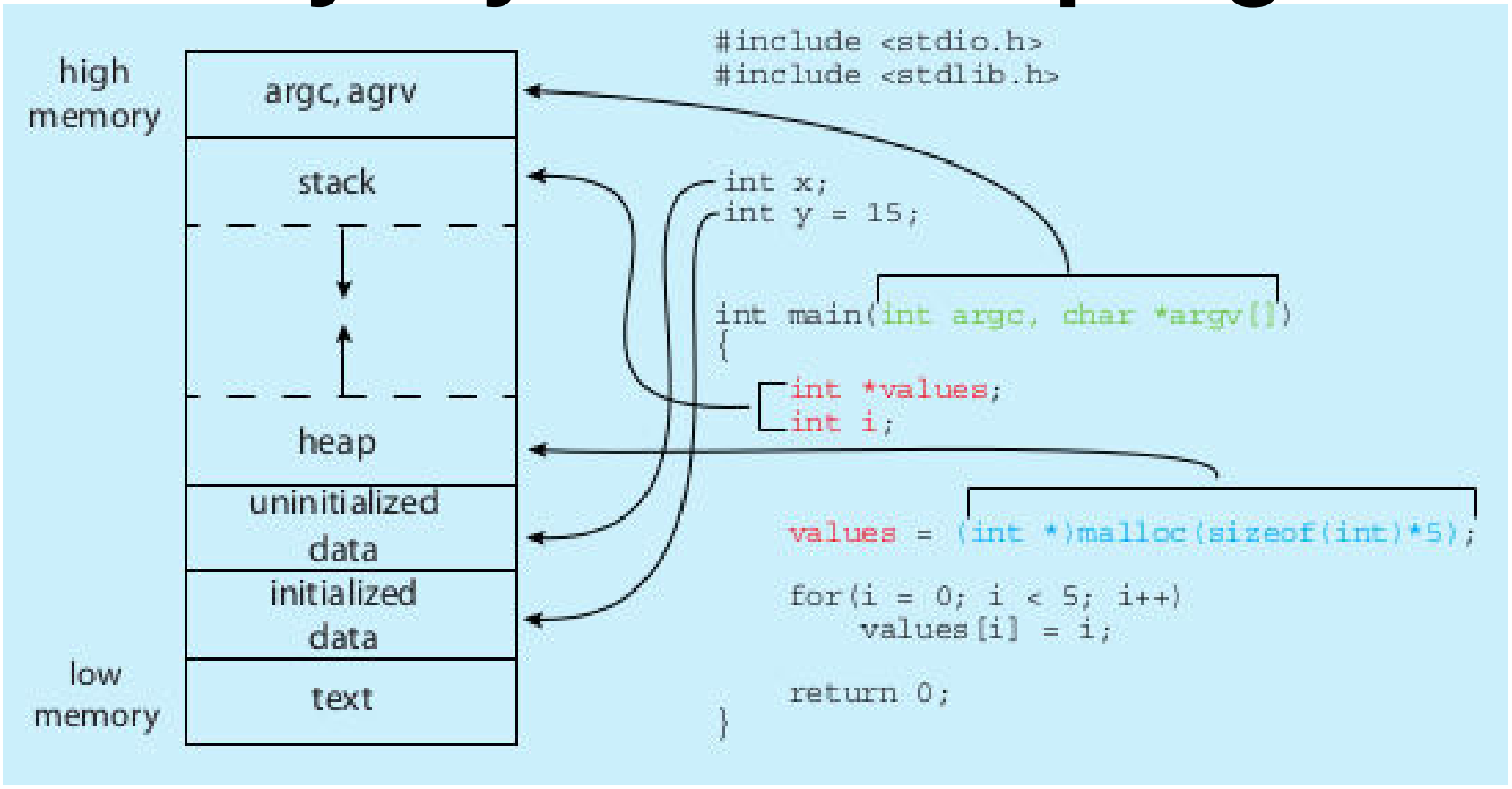
Summary

- Understanding how the processor architecture drives the memory management features of OS and system programs (compilers, linkers)
- Understanding how different hardware designs lead to different memory management schemes by operating systems

Addresses issued by CPU

- During the entire ‘on’ time of the CPU
  - Addresses are “issued” by the CPU on address bus
  - One address to fetch instruction from location specified by PC
  - Zero or more addresses depending on instruction
    - e.g. mov $0x300, r1 # move contents of address 0x300 to r1 --> one extra address issued on address bus

Memory layout of a C program



$ size /bin/ls

text	data	bss	dec	hex	filename
128069	4688	4824	137581	2196d	/bin/ls

Desired from a multi-tasking system

- Multiple processes in RAM at the same time (multi-programming)
- Processes should not be able to see/touch each other’s code, data (globals), stack, heap, etc.
- Further advanced requirements
  - Process could reside anywhere in RAM
  - Process need not be continuous in RAM
  - Parts of process could be moved anywhere in RAM

Different ‘times’

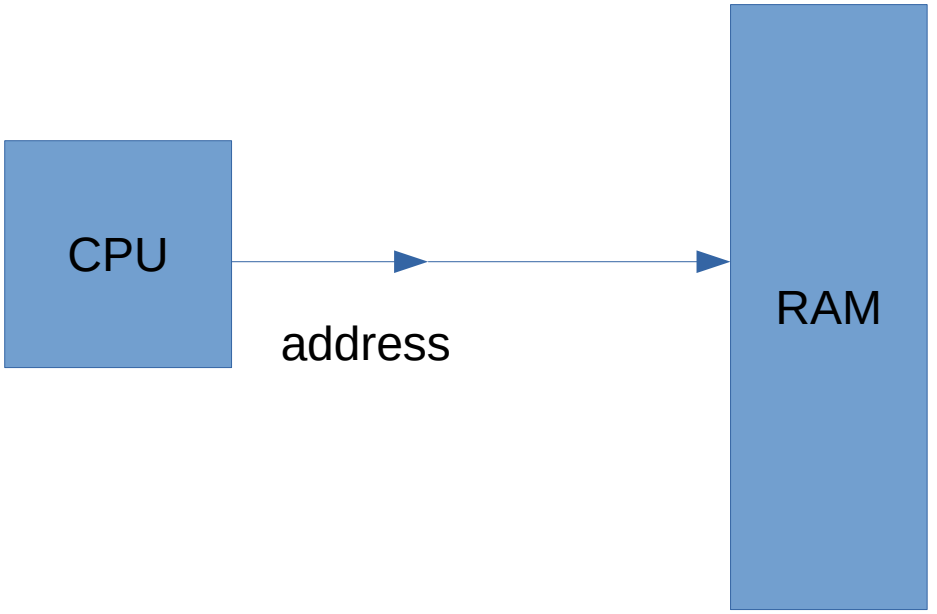
- Different actions related to memory management for a program are taken at different times. So let’s know the different ‘times’
- Compile time
  - When compiler is compiling your C code
- Load time
  - When you execute “./myprogram” and it’s getting loaded in RAM by loader i.e. exec()
- Run time
  - When the process is alive, and getting scheduled by the OS

Different types of Address binding

Which binding is actually used, is mandated by processor features + OS

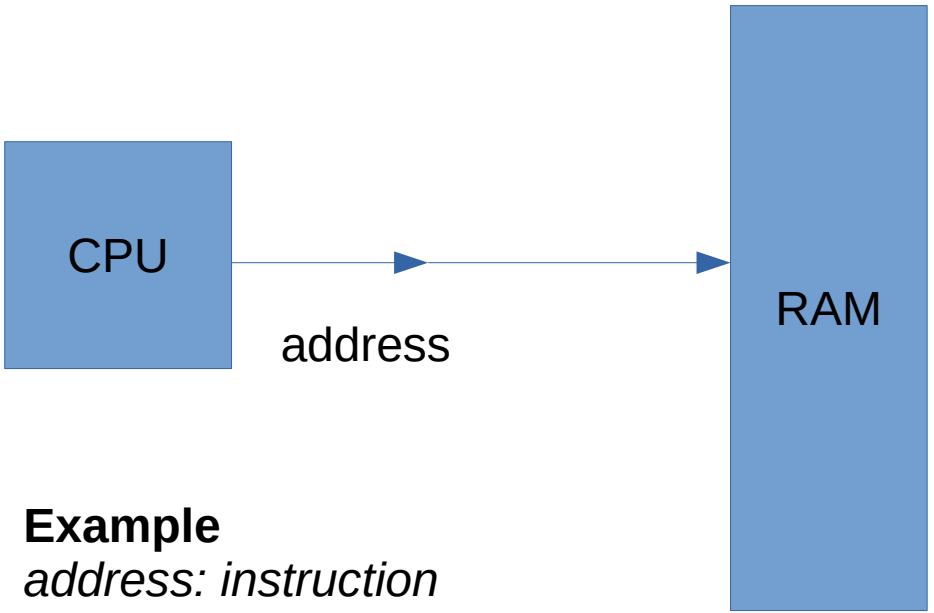
- Compile time address binding
  - Address of code/variables is fixed by compiler
  - Very rigid scheme
  - Location of process in RAM can not be changed ! Non-relocatable code.
- Load time address binding
  - Address of code/variables is fixed by loader
  - Location of process in RAM is decided at load time, but can’t be changed later
  - Flexible scheme, relocatable code
- Run time address binding
  - Address of code/variables is fixed at the time of executing the code
  - Very flexible scheme , highly relocatable code
  - Location of process in RAM is decided at load time, but CAN be changed later also

Simplest case



- Suppose the address issued by CPU reaches the RAM controller directly

Simplest case



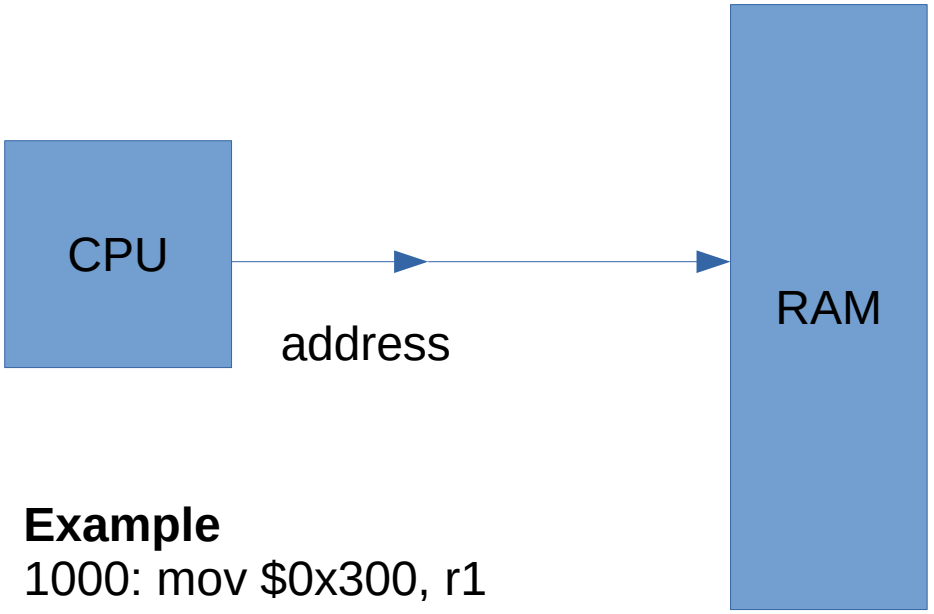
- How does this impact the compiler and OS ?
- When a process is running the addresses issued by it, will reach the RAM directly
- So exact addresses of globals, addresses in “jmp” and “call” must be part the machine instructions generated by compiler
  - How will the compiler know the addresses, at “compile time” ?

Example
address: instruction

1000: mov $0x300, r1
1004: add r1, -3
1008: jnz 1000

Sequence of addressed sent by CPU: 1000, 0x300, 1004, 1008, 1000, 0x300, ...

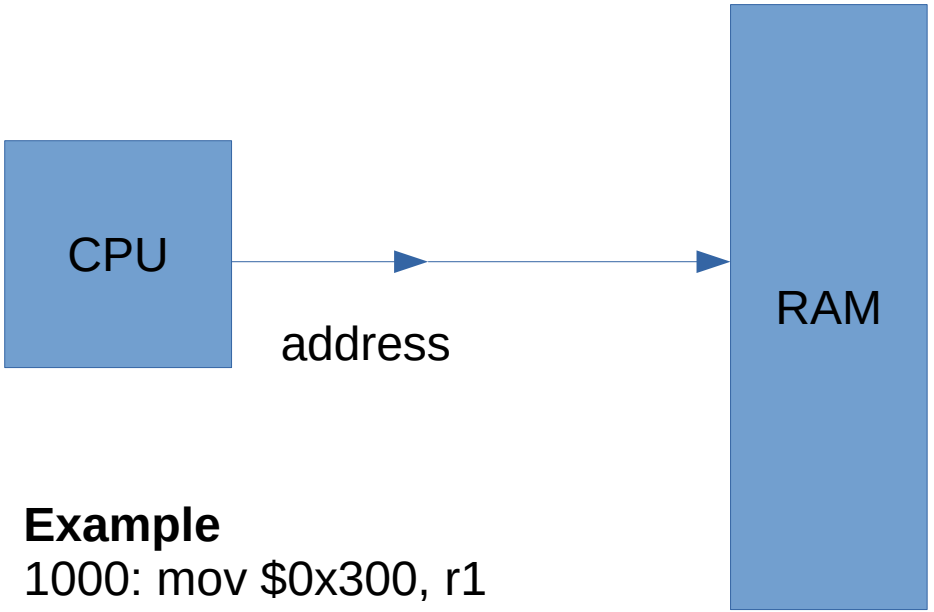
Simplest case



Example
1000: mov $0x300, r1
1004: add r1, -3
1008: jnz 1000

- Solution: compiler assumes some fixed addresses for globals, code, etc.
- OS loads the program exactly at the same addresses specified in the executable file. **Non-relocatable code.**
- Now program can execute properly.

Simplest case



Example
1000: mov $0x300, r1
1004: add r1, -3
1008: jnz 1000

- Problem with this solution
  - Programs once loaded in RAM must stay there, can't be moved
  - What about 2 programs?
    - Compilers being "programs", will make same assumptions and are likely to generate same/overlapping addresses for two different programs
    - Hence only one program can be in memory at a time !
    - No need to check for any memory boundary violations – all memory belongs to one process
- Example: DOS

Base/Relocation + Limit scheme

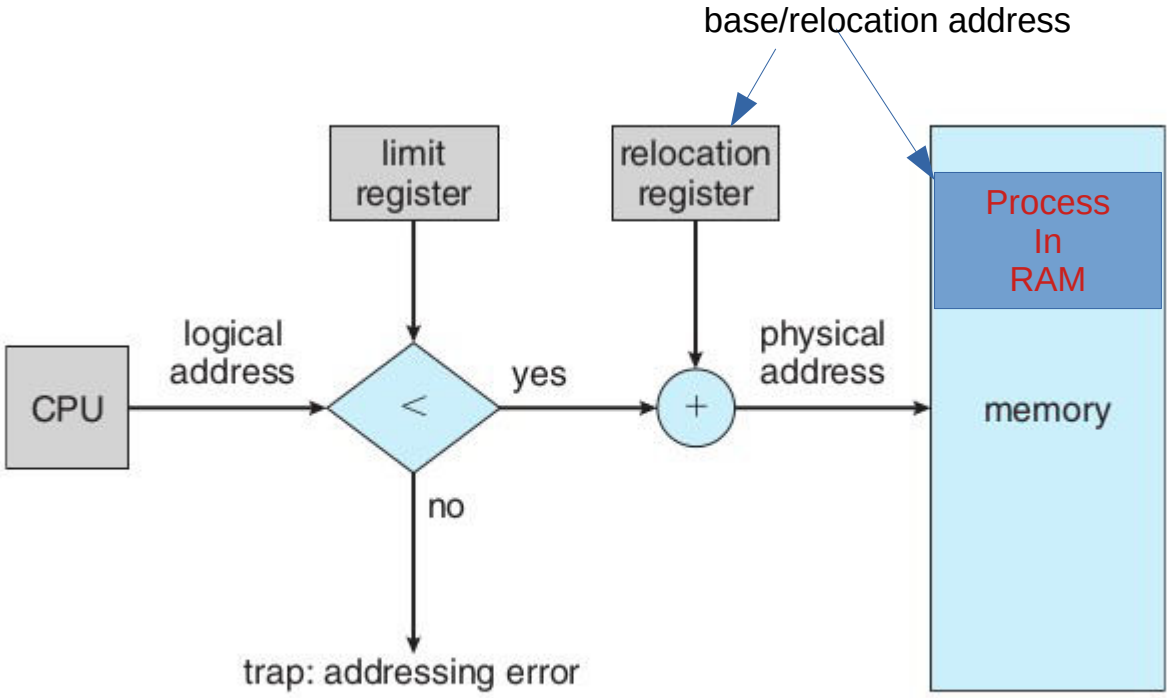


Figure 9.6 Hardware support for relocation and limit registers.

- Base and Limit are two registers inside CPU's Memory Management Unit
- 'base' is added to the address generated by CPU
- The result is compared with base+limit and if less passed to memory, else hardware interrupt is raised

Memory Management Unit (MMU)

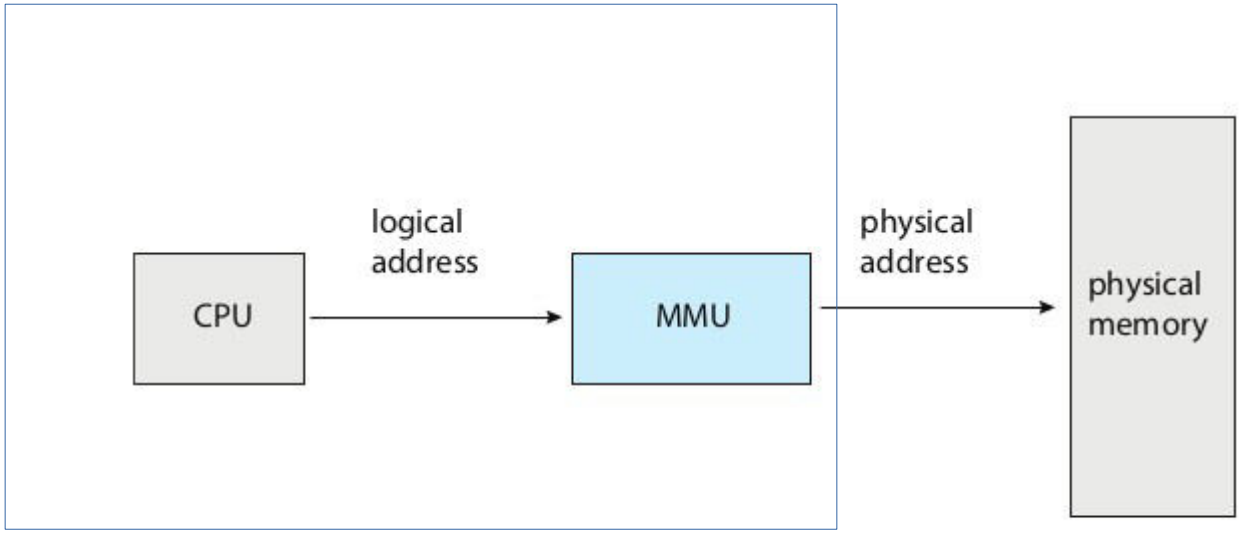
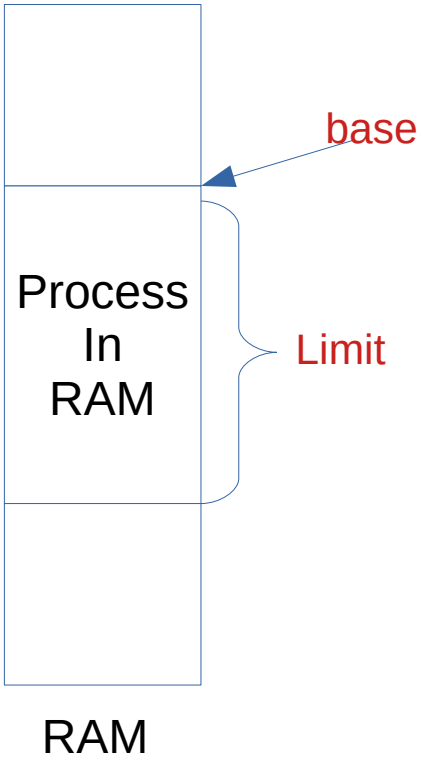
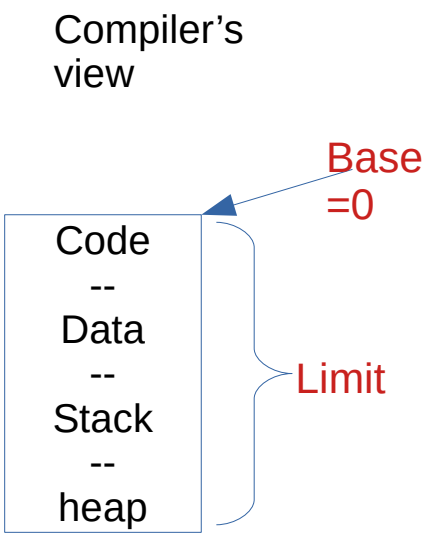


Figure 9.4 Memory management unit (MMU).

- Is part of the CPU chip, acts on every memory address issue by execution unit of the CPU
- In the scheme just discussed, the base, limit calculation parts are part of MMU

Base/Relocation + Limit scheme





- Compiler's work
  - Assume that the process is one continous chunk in memory, with a size limit
  - Assume that the process starts at address zero (!) and calculate addresses for globals, code, etc. And accordingly generate machine code

Base/Relocation + Limit scheme

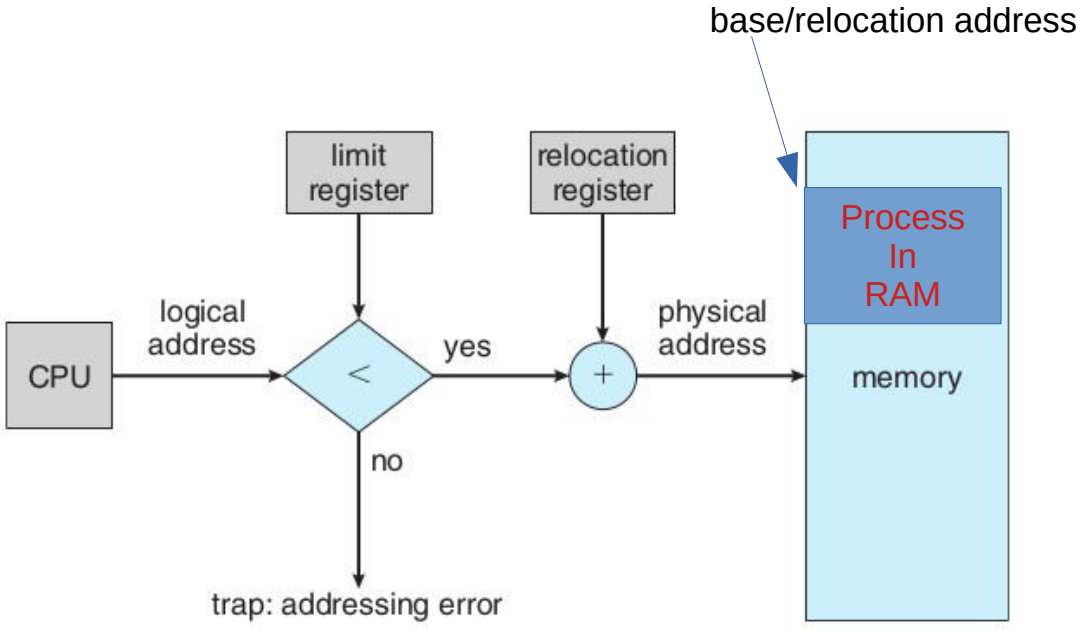


Figure 9.6 Hardware support for relocation and limit registers.

- OS's work
  - While loading the process in memory – must load as one continous segment
  - Fill in the 'base' register with the actual address of the process in RAM.
  - Setup the limit to be the size of the process as set by compiler in the executable file. *Remember the base+limit in OS's own data structures.*

Base/Relocation + Limit scheme

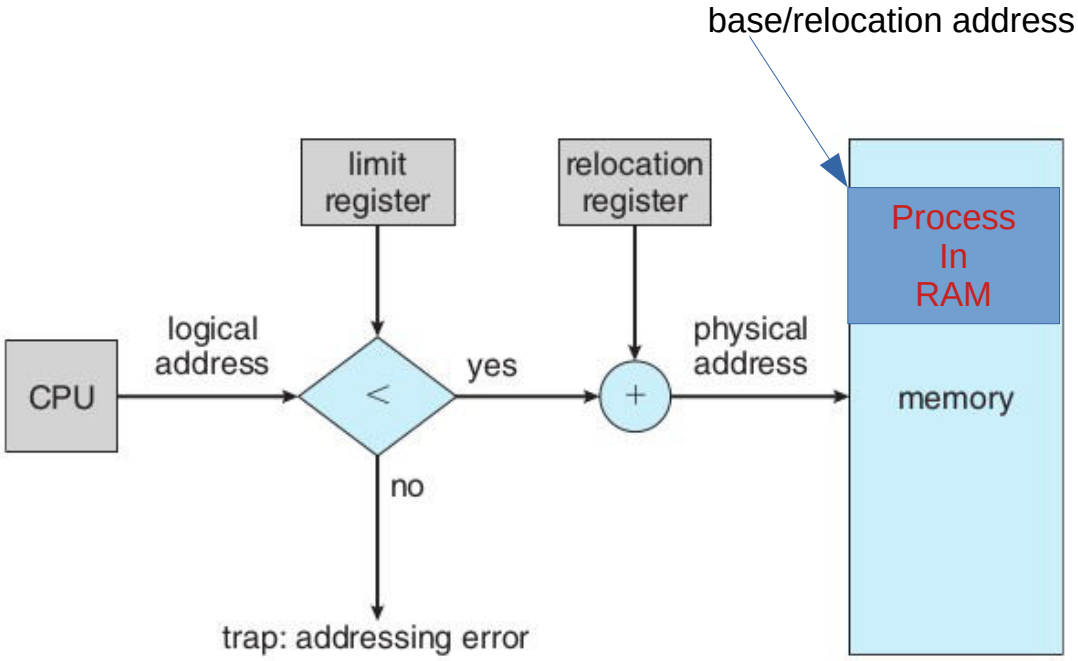
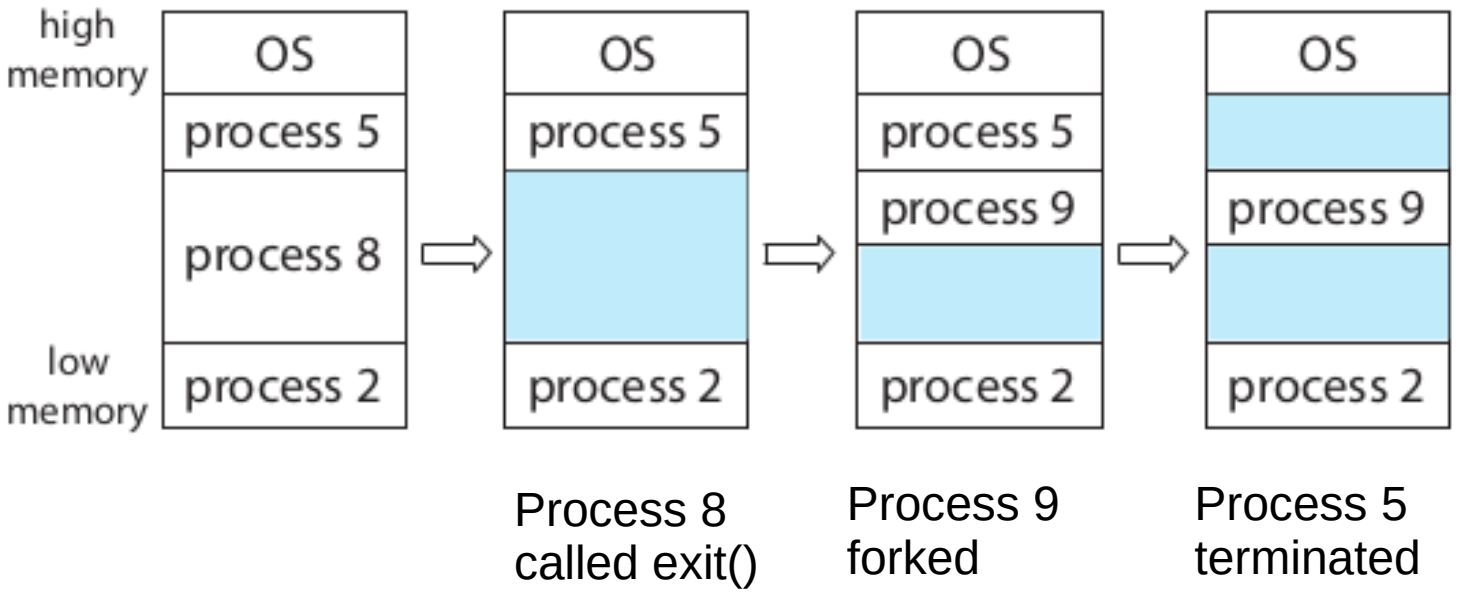


Figure 9.6 Hardware support for relocation and limit registers.

- Combined effect
  - **"Relocatable code"** – the process can go anywhere in RAM at the time of loading
  - Some memory violations can be detected – a memory access beyond base+limit will raise interrupt, thus running OS in turn, which may take action against the process

Example scenario of memory in base+limit scheme



It should be possible to have relocatable code even with “simplest case”

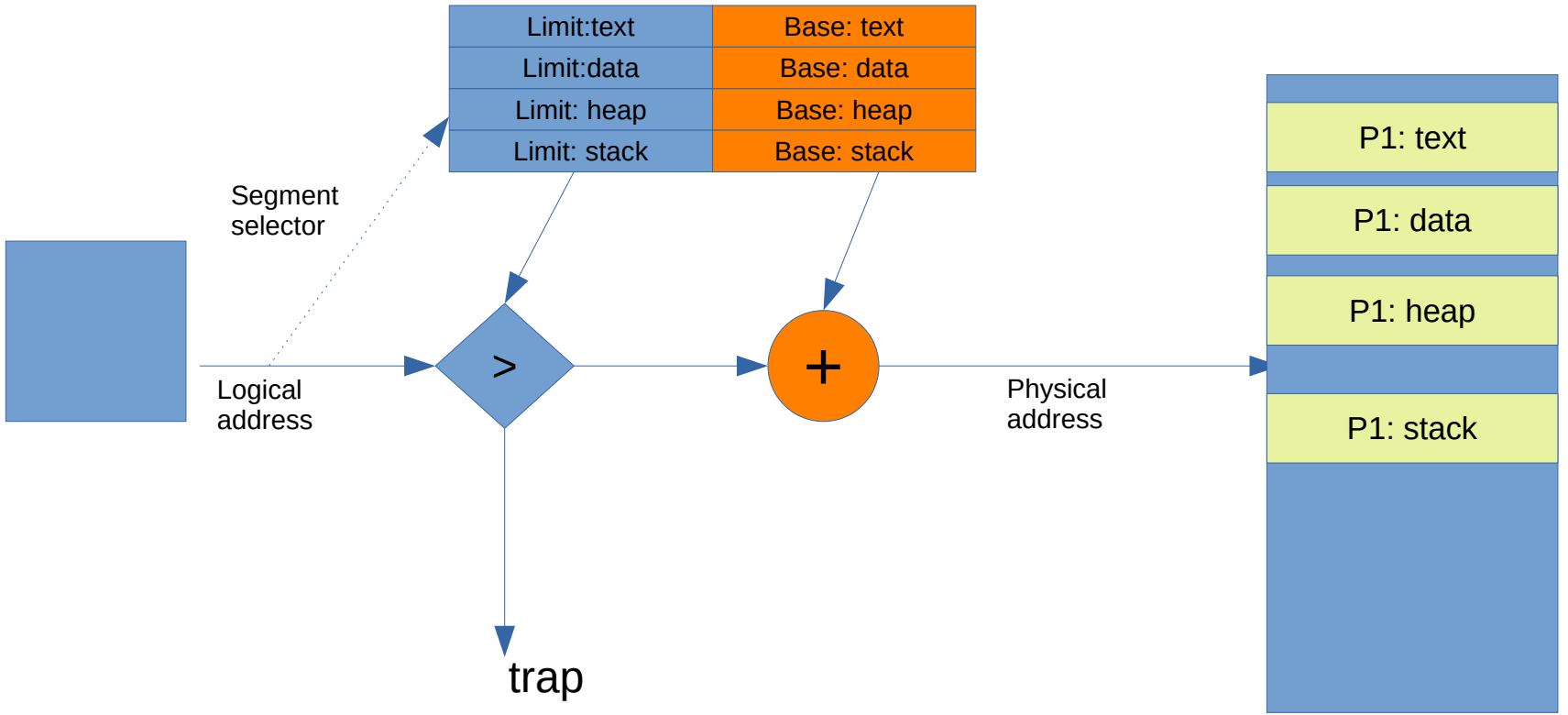
By doing extra work during “loading”.

How?

Next scheme: Segmentation Multiple base +limit pairs

- Multiple sets of base + limit registers
- Whenever an address is issued by execution unit of CPU, it will also include reference to some base register
  - And hence limit register paired to that base register will be used for error checking
- Compiler: can assume a separate chunk of memory for code, data, stack, heap, etc. And accordingly calculate addresses . Each “segment” starting at address 0.
- OS: will load the different ‘sections’ in different memory regions and accordingly set different ‘base’ registers
-

Next scheme:
Multiple base +limit pairs



Next scheme: Multiple base +limit pairs, with further indirection

- Base + limit pairs can also be stored in some memory location (not in registers). Question: how will the cpu know where it's in memory?
  - One CPU register to point to the location of table in memory
- Segment registers still in use, they give an index in this table
- This is x86 segmentation
  - Flexibility to have lot more "base+limits" in the array/table in memory

Next scheme:
Multiple base +limit pairs, with further indirection

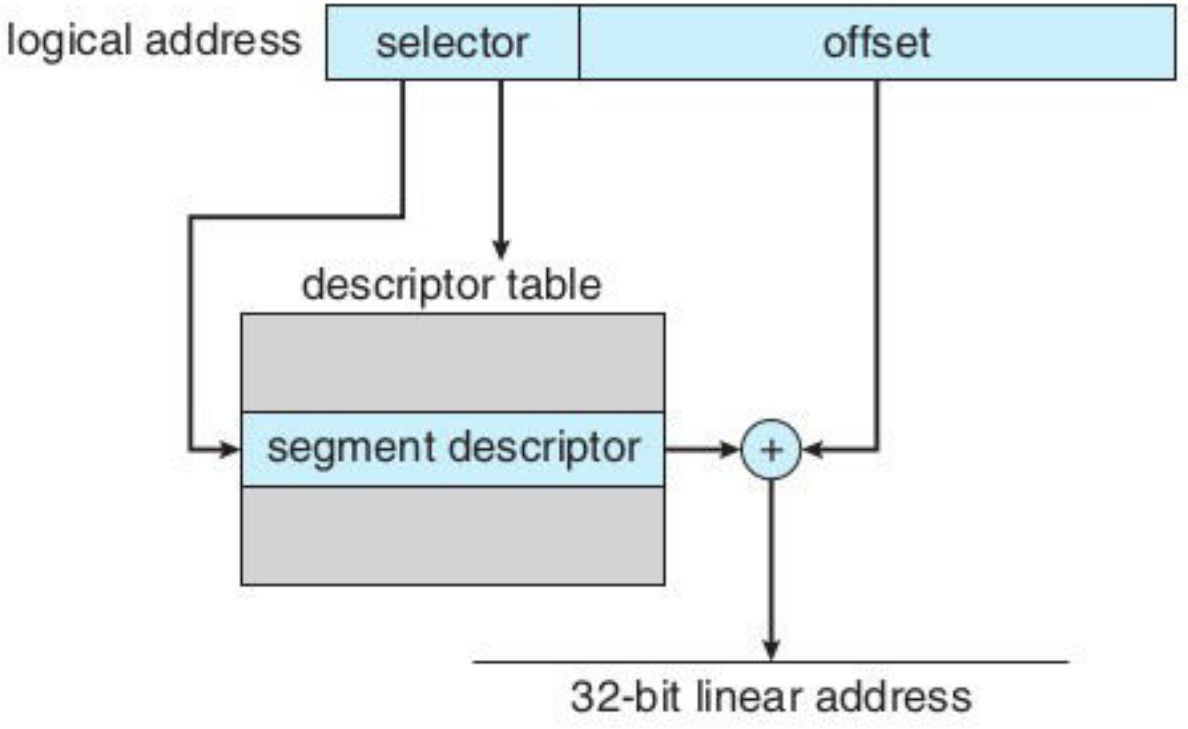


Figure 9.22 IA-32 segmentation.

Problems with segmentation schemes

- OS needs to find a continuous free chunk of memory that fits the size of the “segment”
  - If not available, your exec() can fail due to lack of memory
- Suppose 50k is needed
  - Possible that mong 3 free chunks total 100K may be available, but no single chunk of 50k!
  - **External fragmentation**
- Solution to external fragmentation: **compaction** – move the chunks around and make a continous big chunk available. Time consuming, tricky.

Solving external fragmentation problem

- Process should not be continous in memory!
- Divide the continuous process image in smaller chunks (let’s say 4k each) and locate the chunks anywhere in the physical memory
  - Need a way to map the *logical* memory addresses into *actual physical memory addresses*

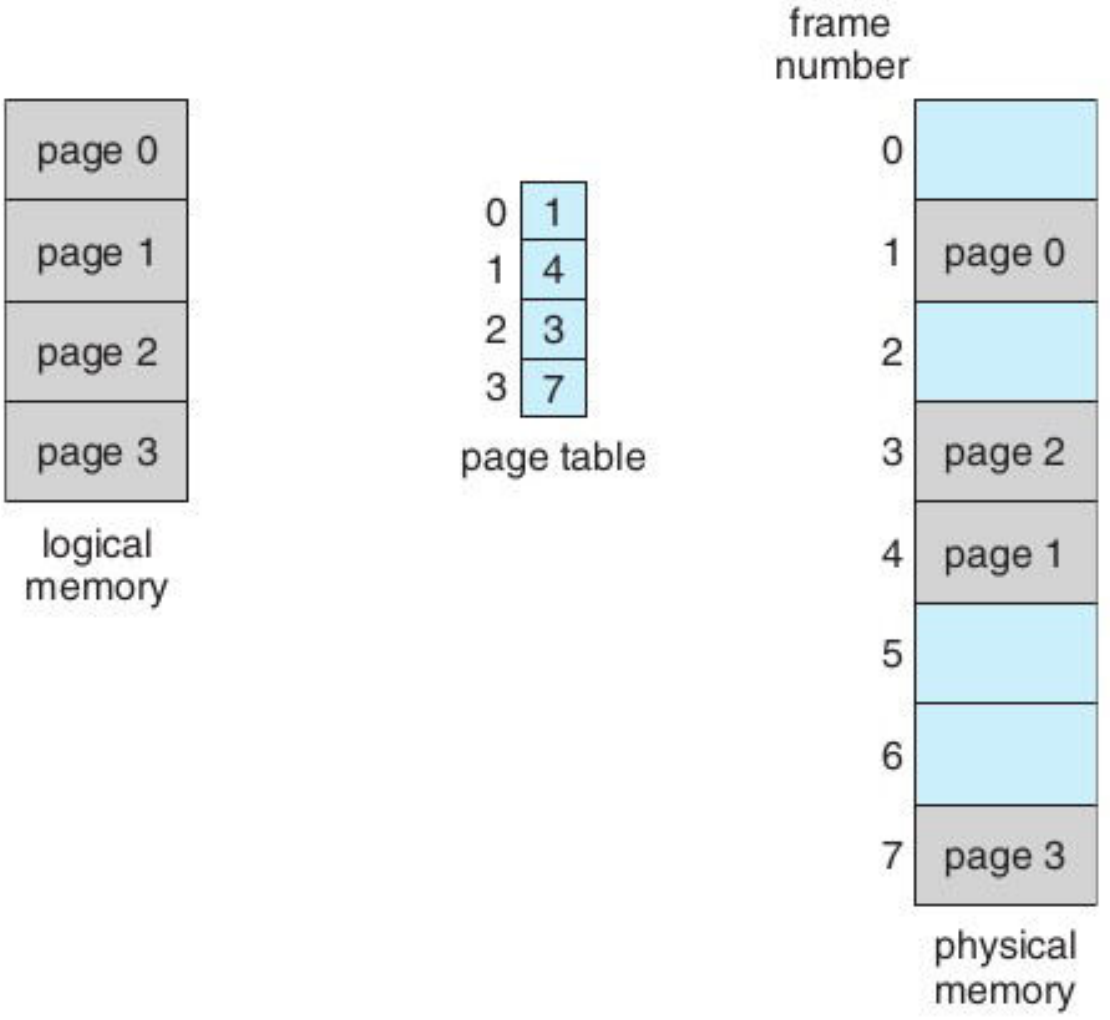


Figure 9.9 Paging model of logical and physical memory.

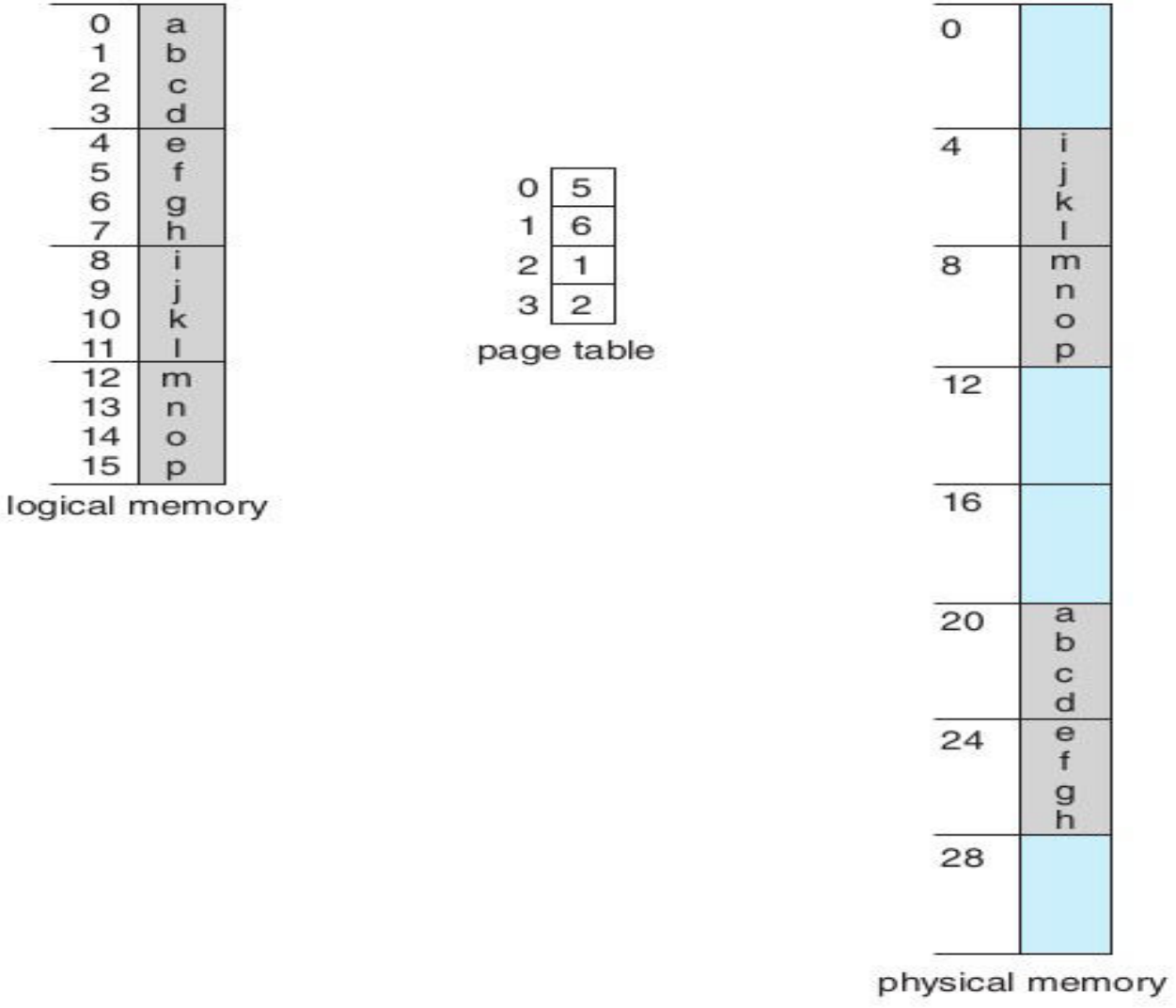


Figure 9.10 Paging example for a 32-byte memory with 4-byte pages.

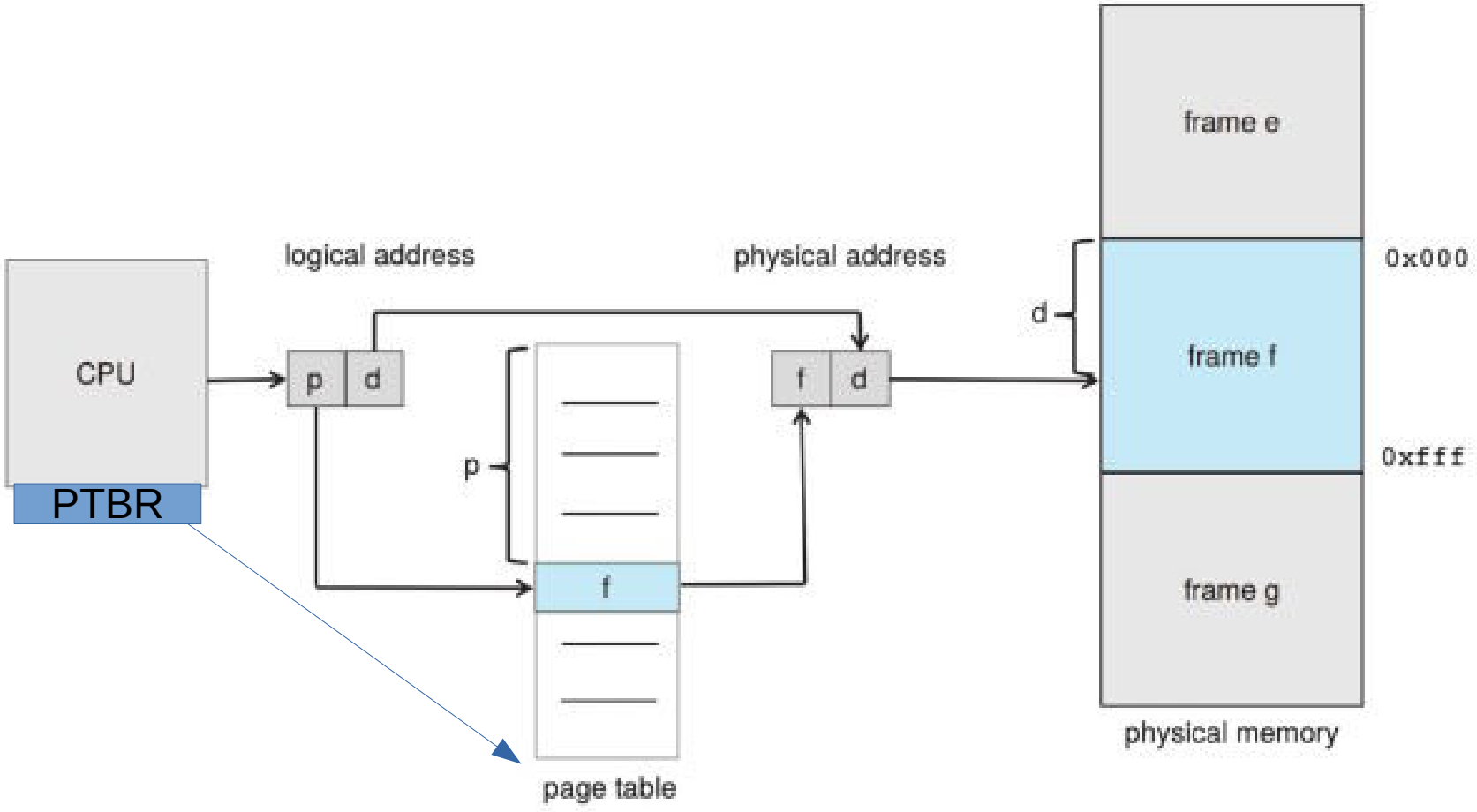


Figure 9.8 Paging hardware.

Paging

- Process is assumed to be composed of equally sized “pages” (e.g. 4k page)
- Actual memory is considered to be divided into page “frames”.
- CPU generated logical address is split into a page number and offset
- A page table base register inside CPU will give location of an in memory table called page table
- Page number used as offset in a table called page table, which gives the physical page frame number
- Frame number + offset are combined to get physical memory address

Paging

- Compiler: assume the process to be one continous chunk of memory (!) . Generate addresses accordingly
- OS: at exec() time – allocate different frames to process, allocate a page table(!), setup the page table to map page numbers with frame numbers, setup the page table base register, start the process
- Now hardware will take care of all translations of logical addresses to physical addresses

X86 memory management

Figure 9.21 Logical to physical address translation in IA-32.

Segmentation in x86

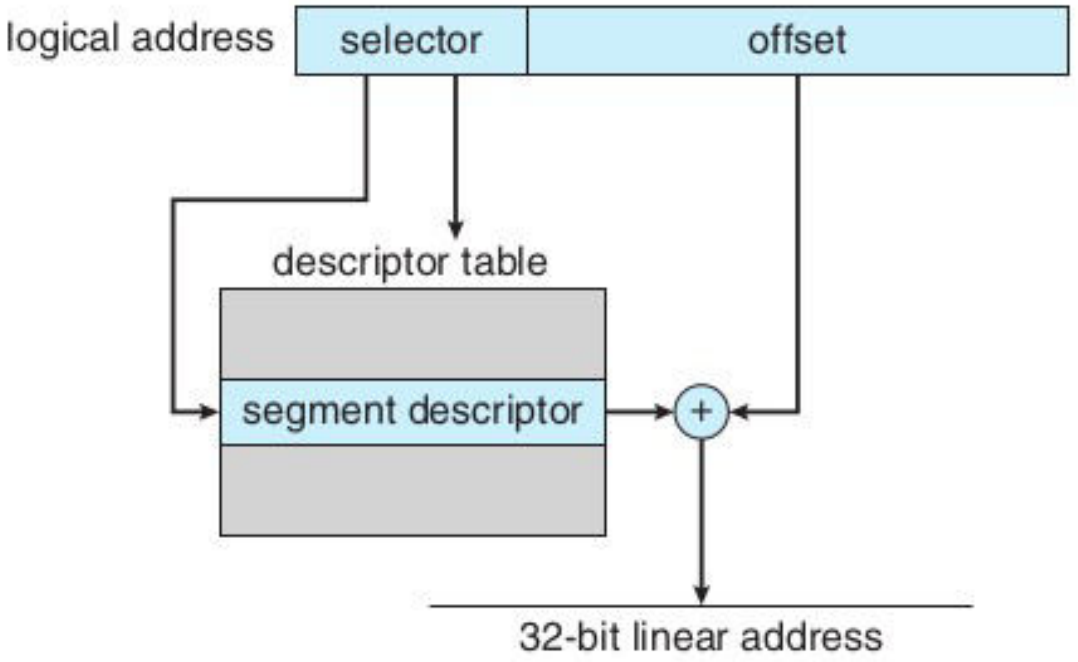


Figure 9.22 IA-32 segmentation.

- The selector is automatically chosen using Code Segment (CS) register, or Data Segment (DS) register depending on which type of memory address is being fetched
- Descriptor table is in memory
- The location of Descriptor table (Global DT- GDT or Local DT – LDT) is given by a GDT-register i.e. GDTR or LDT-register i.e. LDTR
-

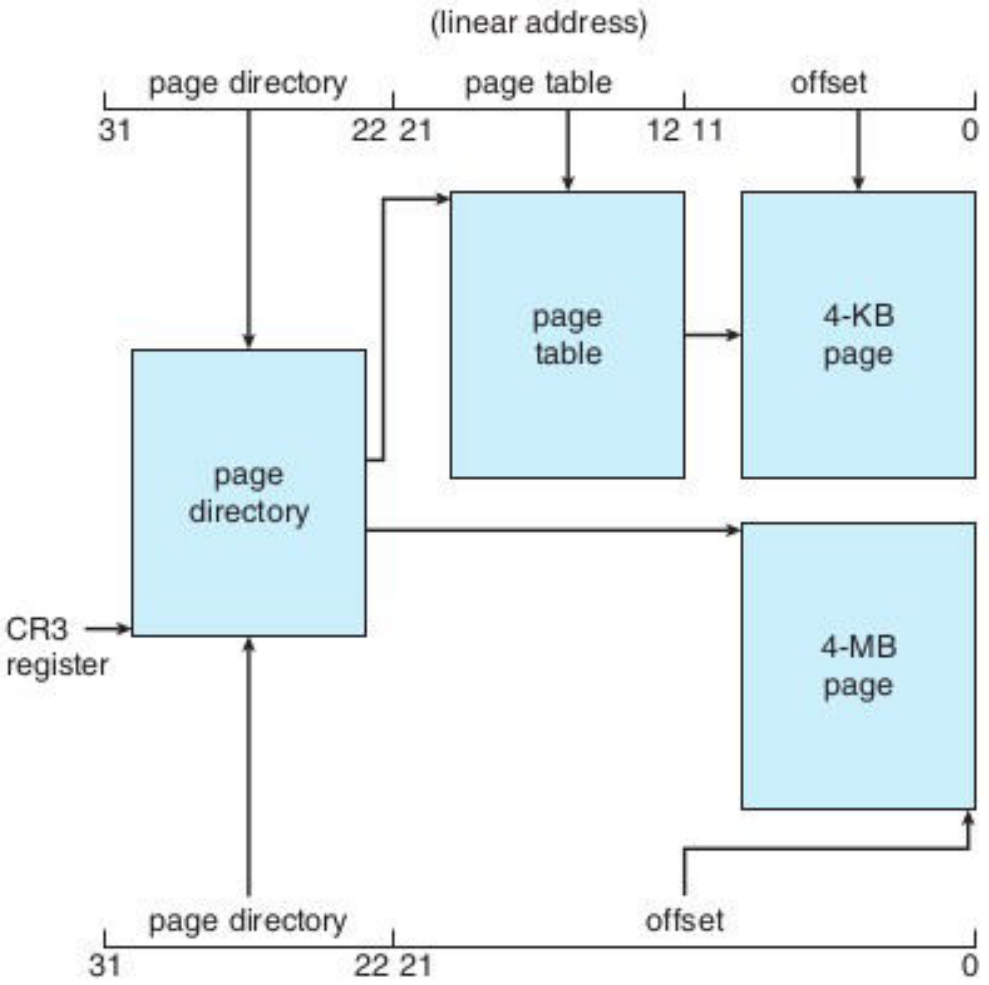


Figure 9.23 Paging in the IA-32 architecture.

Paging in x86

- Depending on a flag setup in CR3 register, either 4 MB or 4 KB pages can be enabled
- Page directory, page table are both in memory

Memory Management – Continued

More on Linking, Loading, Paging

Review of last class

- **MMU : Hardware features for MM**
- **OS: Sets up MMU for a process, then schedules process**
- **Compiler : Generates object code for a particular OS + MMU architecture**
- **MMU: Detects memory violations and raises interrupt --> Effectively passing control to OS**

More on Linking and Loading

- **Static Linking: All object code combined at link time and a big object code file is created**
- **Static Loading: All the code is loaded in memory at the time of exec()**
- **Problem: Big executable files, need to load functions even if they do not execute**
- **Solution: Dynamic Linking and Dynamic Loading**

Dynamic Linking

- **Linker is normally invoked as a part of compilation process**
  - **Links**
    - **function code to function calls**
    - **references to global variables with “extern” declarations**
- **Dynamic Linker**
  - **Does not combine function code with the object code file**
  - **Instead introduces a “stub” code that is indirect reference to actual code**
  - **At the time of “loading” (or executing!) the program in memory, the “link-loader” (part of OS!) will pick up the relevant code from the library machine code file (e.g. libc.so.6)**

Dynamic Linking on Linux

```
#include <stdio.h>
int main() {
int a, b;
scanf("%d%d", &a, &b);
printf("%d %d\n", a, b);
return 0;
}
```

PLT: Procedure Linkage Table
used to call external procedures/functions whose address is to be resolved by the dynamic linker at run time.

Output of objdump -x -D

Disassembly of section .text:

0000000000001189 <main>:

11d4: callq 1080 <printf@plt>

Disassembly of section .plt.got:

0000000000001080 <printf@plt>:

1080: endbr64

1084: bnd jmpq *0x2f3d(%rip) # 3fc8 <printf@GLIBC_2.2.5>

108b: nopl 0x0(%rax,%rax,1)

Dynamic Loading

- **Loader**
  - **Loads the program in memory**
  - **Part of exec() code**
  - **Needs to understand the format of the executable file (e.g. the ELF format)**
- **Dynamic Loading**
  - **Load a part from the ELF file only if needed during execution**
  - **Delayed loading**
  - **Needs a more sophisticated memory management by operating system – to be seen during this series of lectures**

Dynamic Linking, Loading

- **Dynamic linking necessarily demands an advanced type of loader that understands dynamic linking**
  - **Hence called ‘link-loader’**
  - **Static or dynamic loading is still a choice**
- **Question: which of the MMU options will alllow for which type of linking, loading ?**

Continous memory management

What is Continous memory management?

- **Entire process is hosted as one continous chunk in RAM**
- **Memory is typically divided into two partitions**
  - **One for OS and other for processes**
  - **OS most typically located in “high memory” addresses, because interrupt vectors map to that location (Linux, Windows) !**

Hardware support needed: base + limit (or relocation + limit)

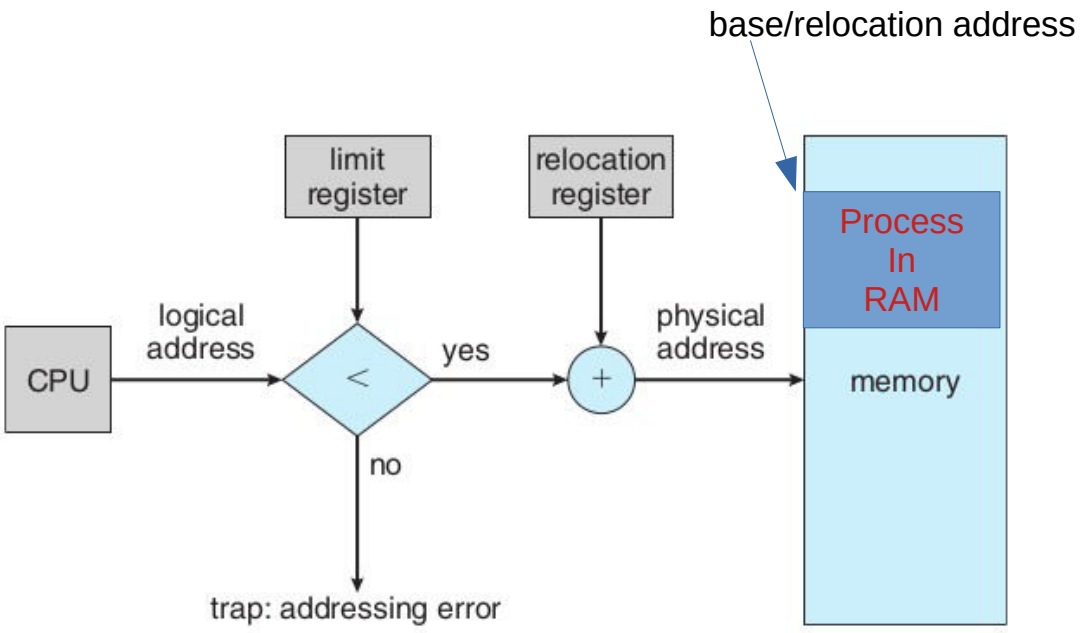


Figure 9.6 Hardware support for relocation and limit registers.

Problems faced by OS

- Find a continuous chunk for the process being forked
- Different processes are of different sizes
  - Allocate a size parameter in the PCB
- After a process is over – free the memory occupied by it
- Maintain a list of free areas, and occupied areas
  - Can be done using an array, or linked list

Variable partition scheme

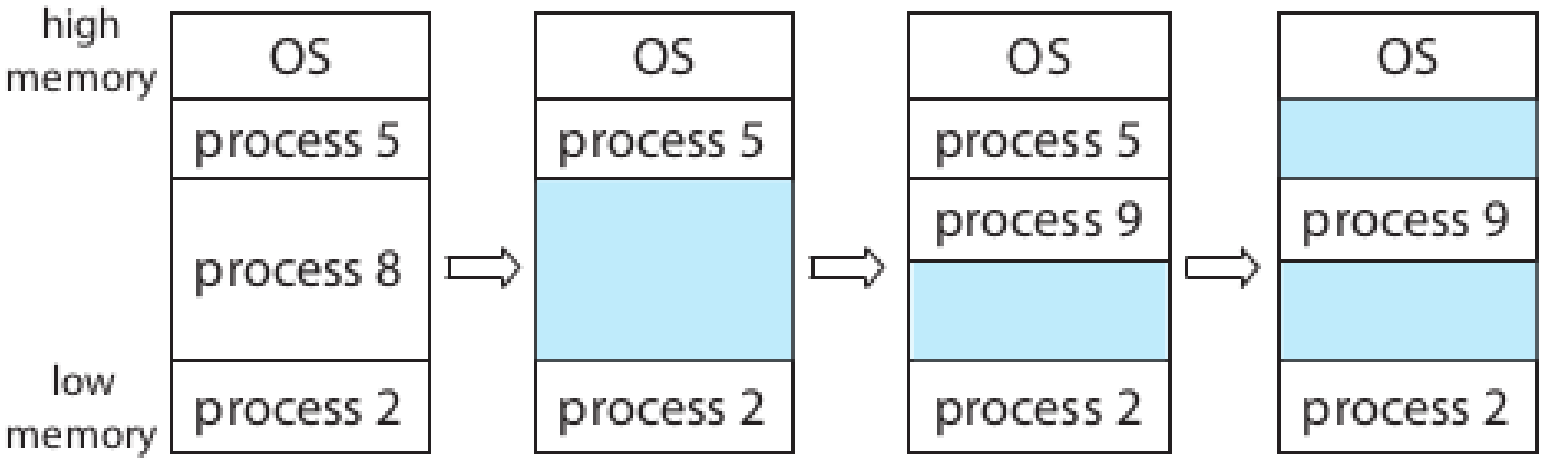


Figure 9.7 Variable partition.

Problem: how to find a “hole” to fit in new process

- **Suppose there are 3 free memory regions of sizes 30k, 40k, 20k**
- **The newly created process (during fork() + exec()) needs 15k**
- **Which region to allocate to it ?**

Strategies for finding a free chunk

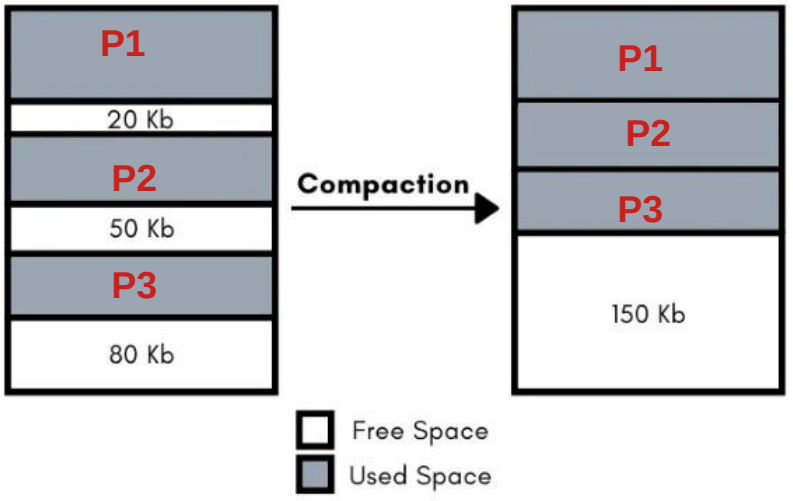
- **6k, 17k, 16k, 40k holes . Need 15k.**
- **Best fit: Find the smallest hole, larger than process. Ans: 16k**
- **Worst fit: Find the largest hole. Ans: 40k**
- **First fit: Find the “first” hole larger than the process. Ans: 17k**

Problem : External fragmentation

- **Free chunks: 30k, 40k, 20k**
- **The newly created process (during fork() + exec()) needs 50k**
- **Total free memory: 30+40+20 = 90k**
  - **But can’t allocate 50k !**

Solution to external fragmentation

- **Compaction !**
- **OS moves the process chunks in memory to make available continous memory region**
  - **Then it must update the memory management information in PCB (e.g. base of the process) of each process**
- **Time consuming**
- **Possible only if the relocation+limit scheme of MMU is available**



Another solution to external fragmentation: Fixed size partitions

- **Fixed partition scheme**
- **Memory is divided by OS into chunks of equal size: e.g., say, 50k**
  - **If total 1M memory, then 20 such chunks**
- **Allocate one or more chunks to a process, such that the total size is >= the size of the process**
  - **E.g.if request is 50k, allocate 1 chunk**
  - **If request is 40k, still allocate 1 chunk**
  - **If request is 60k, then allocate 2 chunks**
- **Leads to internal fragmentation**
  - **space wasted in the case of 40k or 60k requests above**

Fixed partition scheme

- **OS needs to keep track of**
  - **Which partition is free and which is used by which process**
  - **Free partitions can simply be tracked using a bitmap or a list of numbers**
  - **Each process's PCB will contain list of partitions allocated to it**

Solution to internal fragmentation

- **Reduce the size of the fixed sized partition**
- **How small then ?**
  - **Smaller partitions mean more overhead for the operating system in allocating deallocating**

Paging

An extended version of fixed size partitions

- **Partition = page**
  - **Process = logically continuous sequence of bytes, divided in ‘page’ sizes**
  - **Memory divided into equally sized page ‘frames’**
- **Important distinction**
  - **Process need not be continous in RAM**
  - **Different page sized chunks of process can go in any page frame**
  - **Page table to map pages into frames**

Logical address seen as

Paging hardware

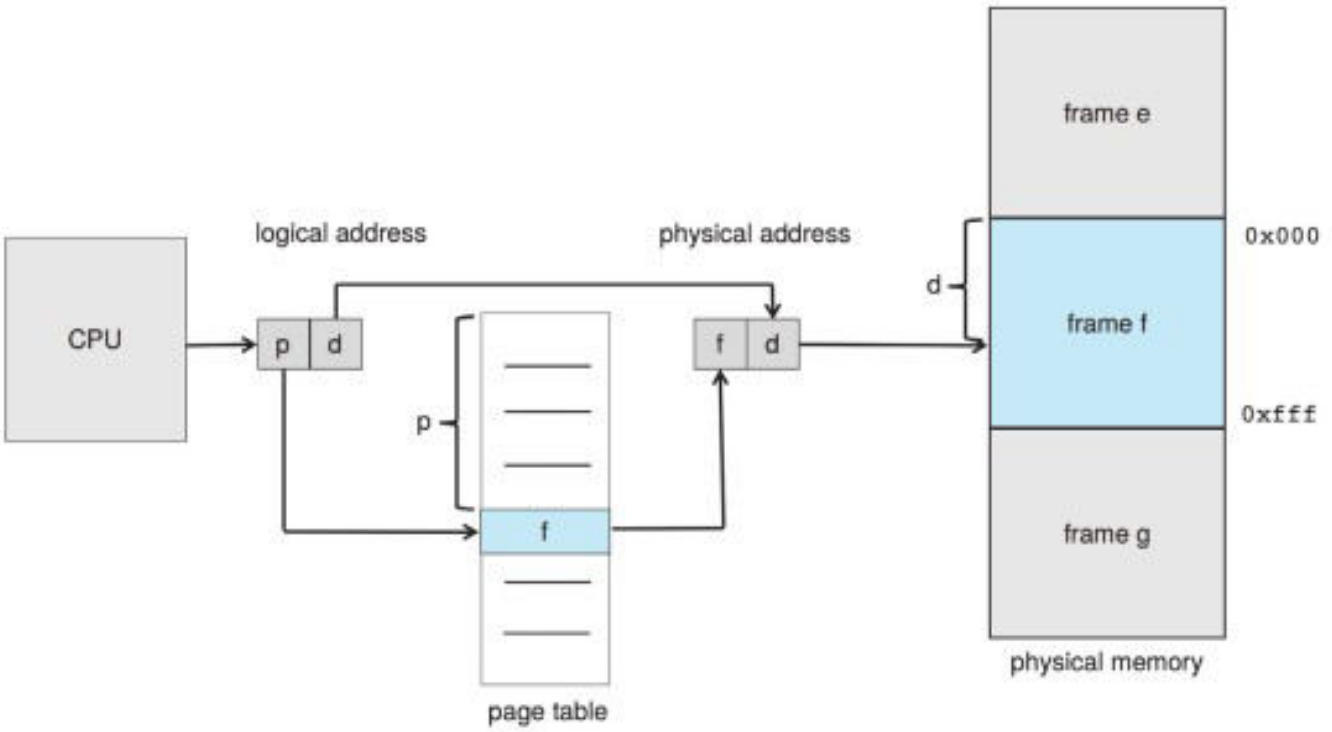


Figure 9.8 Paging hardware.

MMU’s job

To translate a logical address generated by the CPU to a physical address:

1. Extract the page number p and use it as an index into the page table.

(Page table location is stored in a hardware register

Also stored in PCB of the process, so that it can be used to load the hardware register on a context switch)

2. Extract the corresponding frame number f from the page table.

3. Replace the page number p in the logical address with the frame number f .

Job of OS

- **Allocate a page table for the process, at time of fork()/exec()**
  - **Allocate frames to process**
  - **Fill in page table entries**
- **In PCB of each process, maintain**
  - **Page table location (address)**
  - **List of pages frames allocated to this process**
- **During context switch of the process, load the PTBR using the PCB**

Job of OS

- **Maintain a list of all page frames**
  - **Allocated frames**
  - **Free Frames (called frame table)**
  - **Can be done using simple linked list**
  - **Innovative data structures can also be used to maintain free and allocated frames list (e.g. xv6 code)**
  -

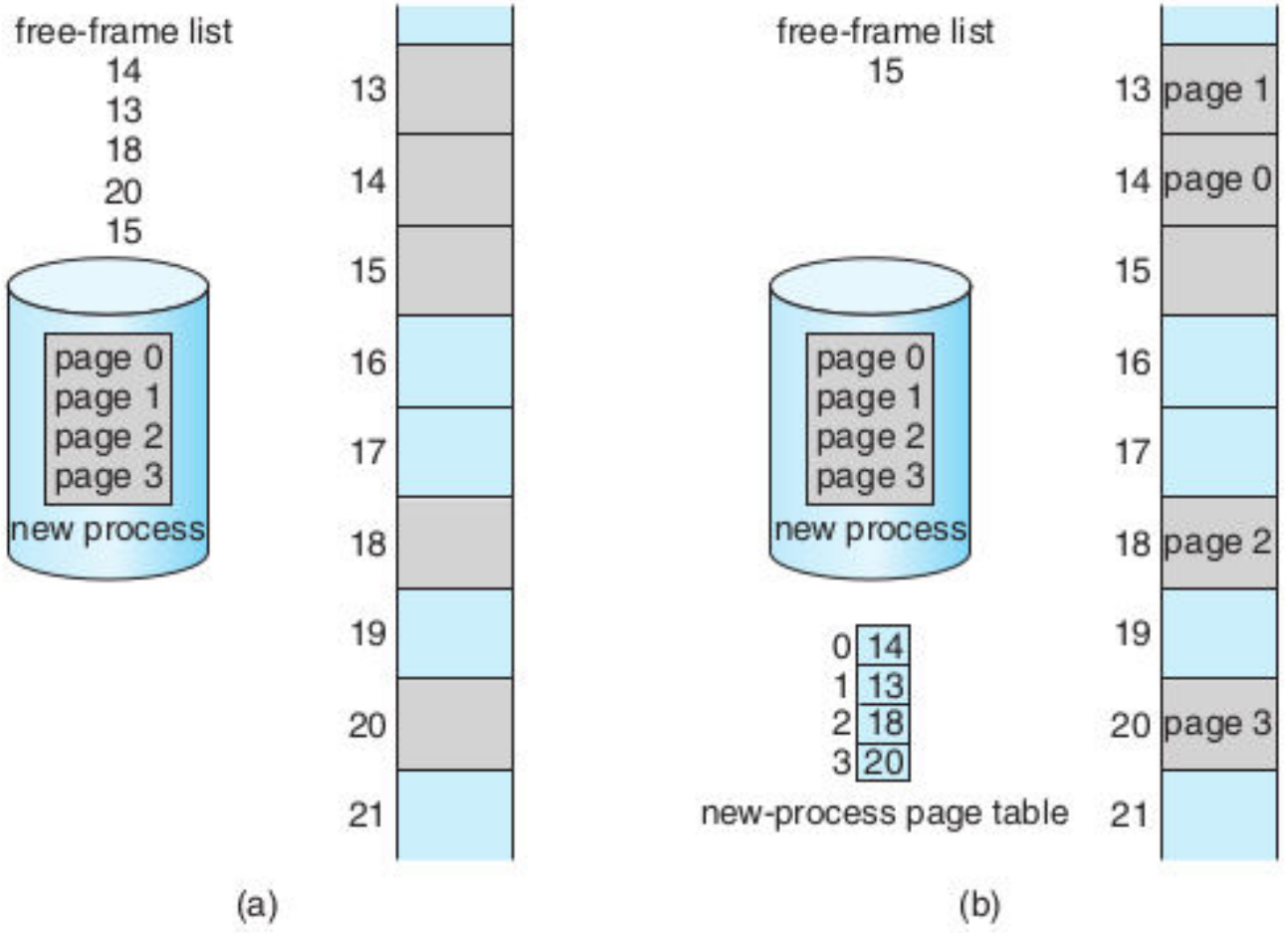


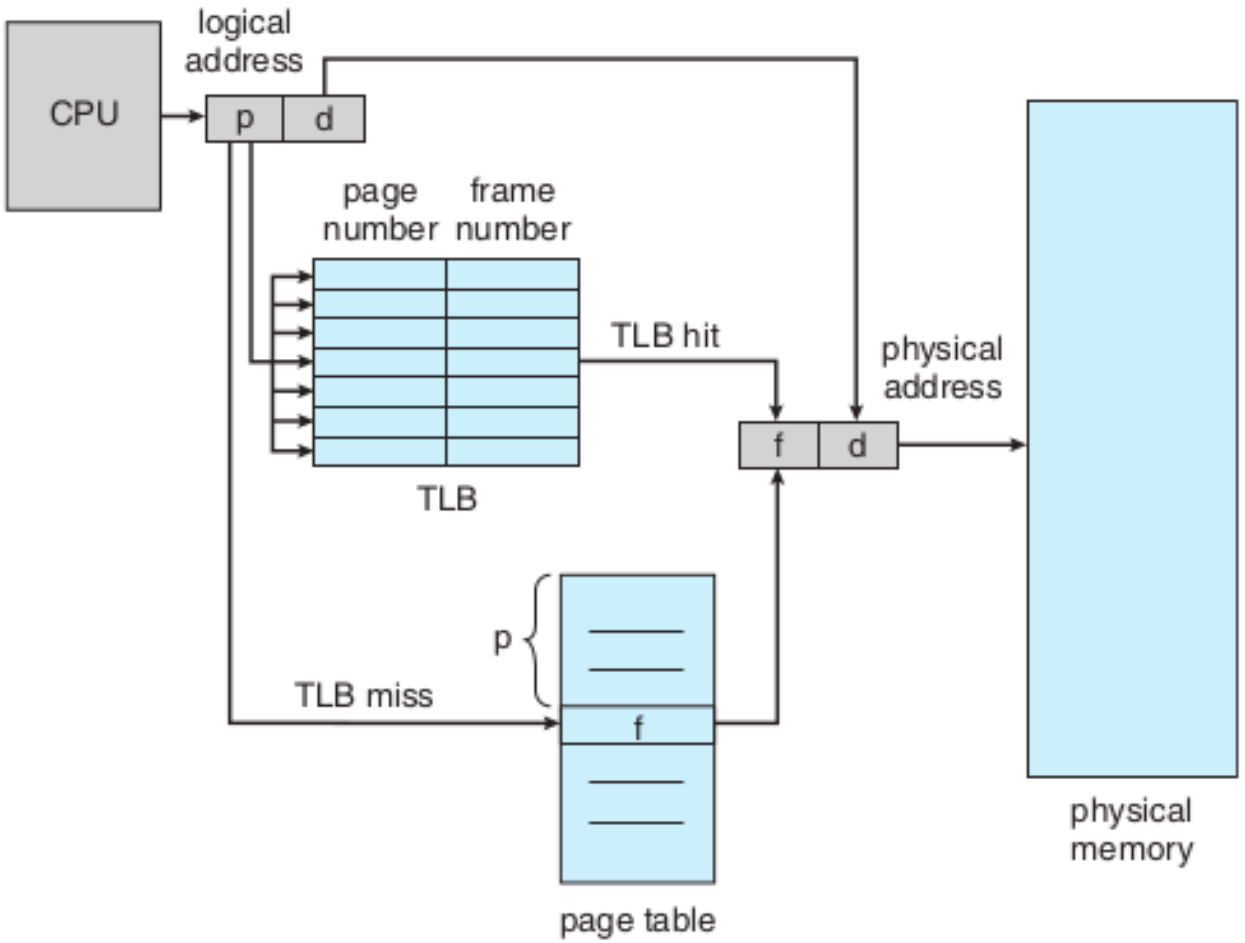
Figure 9.11 Free frames (a) before allocation and (b) after allocation.

Disadvantage of Paging

- **Each memory access results in two memory accesses!**
  - **One for page table, and one for the actual memory location !**
  - **Done as part of execution of instruction in hardware (not by OS!)**
  - **Slow down by 50%**

Speeding up paging

- **Translation Lookaside Buffer (TLB)**
- **Part of CPU hardware**
- **A cache of Page table entries**
- **Searched in parallel for a page number**



Speedup due to TLB

- **Hit ratio**
- **Effective memory access time**

  = Hit ratio * 1 memory access time + miss ratio * 2 memory access time
- **Example: memory access time 10ns, hit ratio = 0.8, then**

  effective access time = 0.80 × 10 + 0.20 × 20

  = 12 nanoseconds

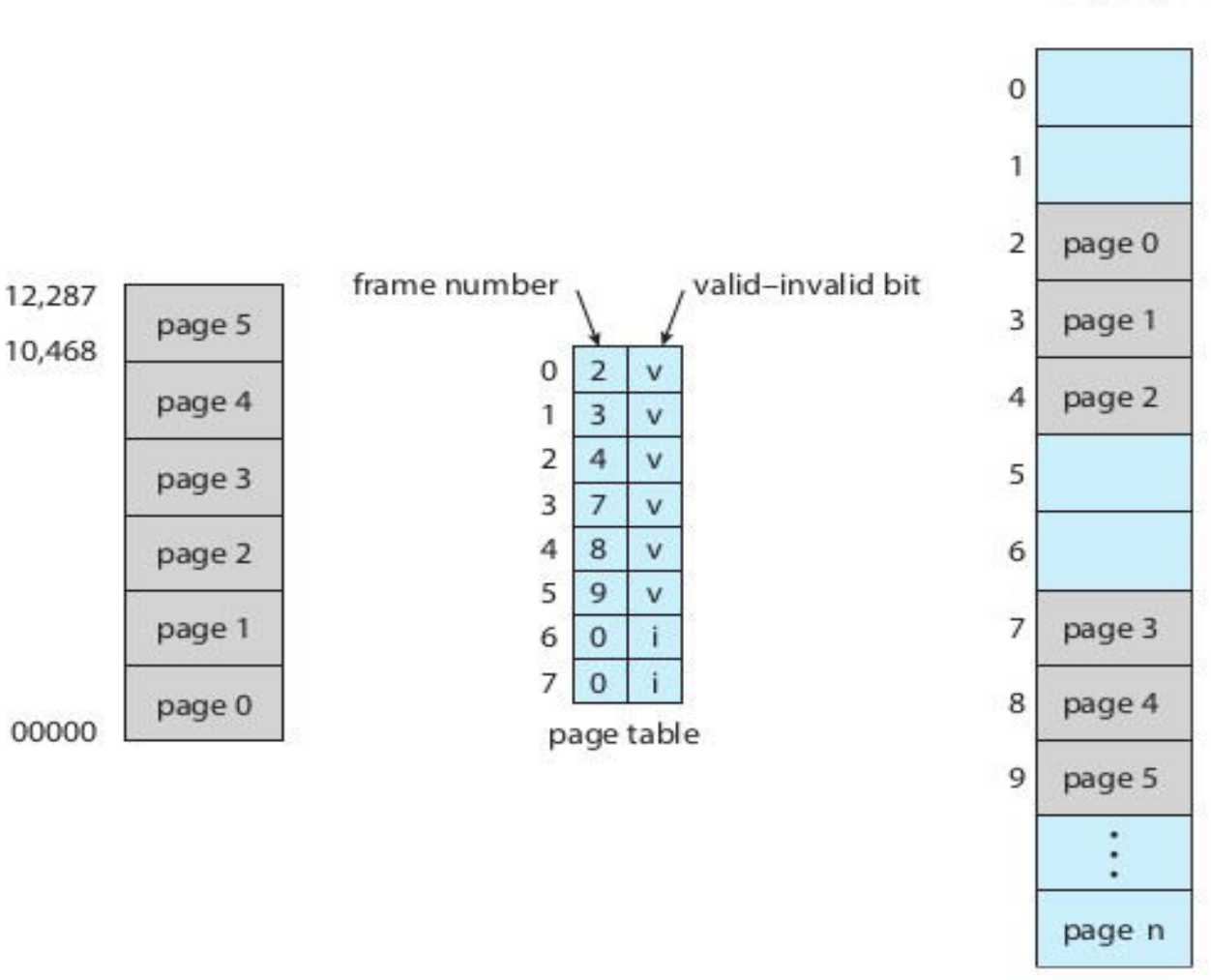
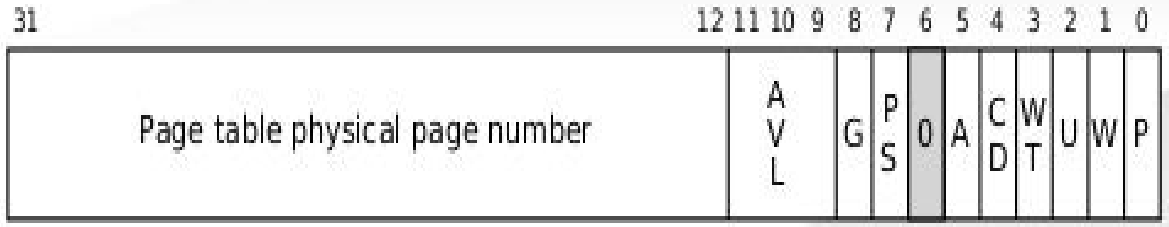


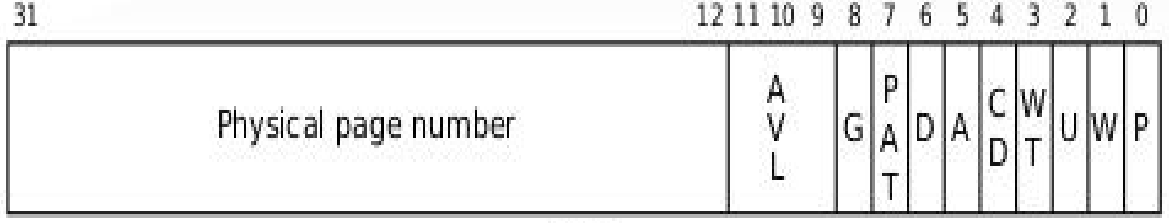
Figure 9.13 Valid (v) or invalid (i) bit in a page table.

Memory protection with paging

X86 PDE and PTE



PDE



PTE

P	Present
W	Writable
U	User
WT	1=Write-through, 0=Write-back
CD	Cache disabled
A	Accessed
D	Dirty
PS	Page size (0=4KB, 1=4MB)
PAT	Page table attribute index
G	Global page
AVL	Available for system use

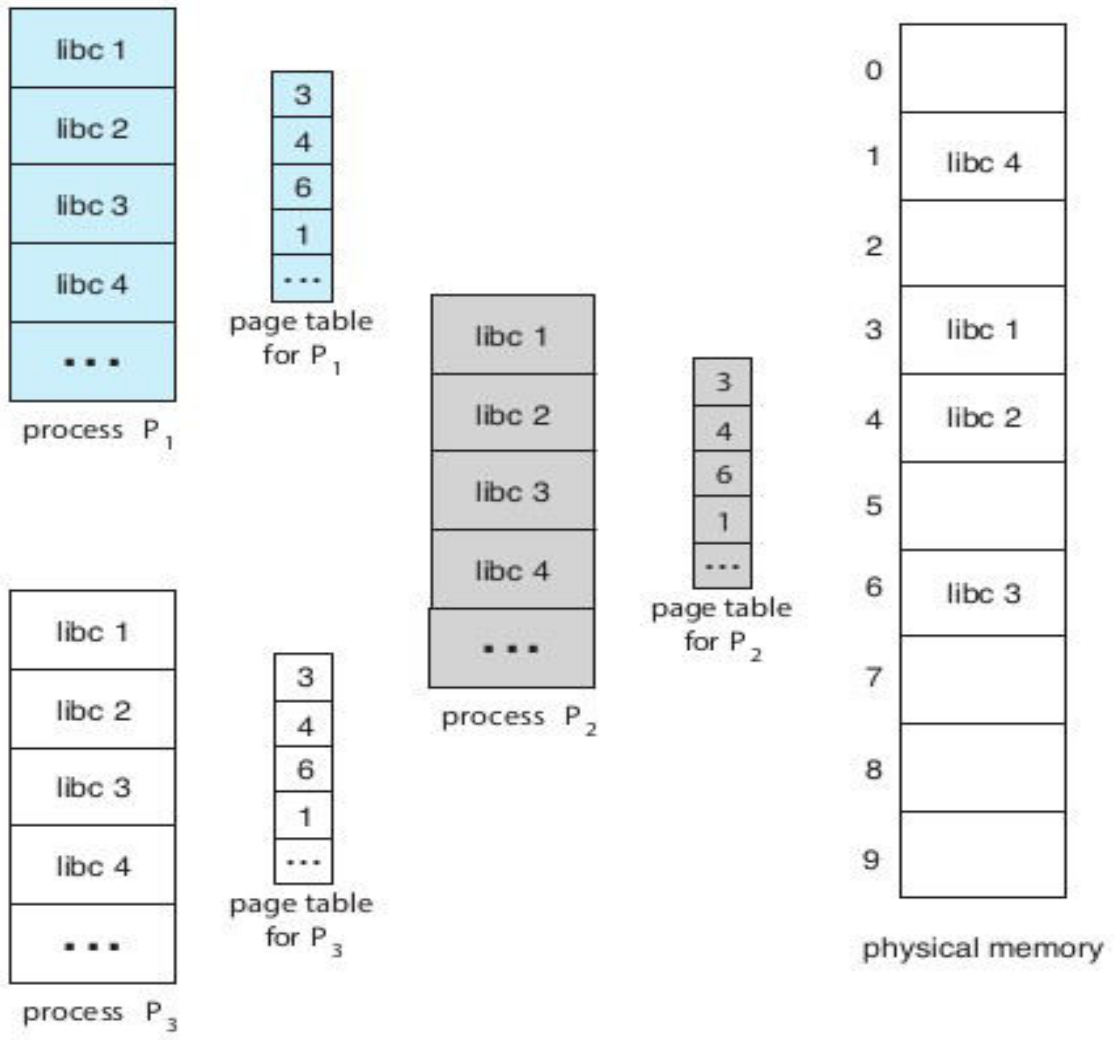


Figure 9.14 Sharing of standard C library in a paging environment.

Shared pages (e.g. library) with paging

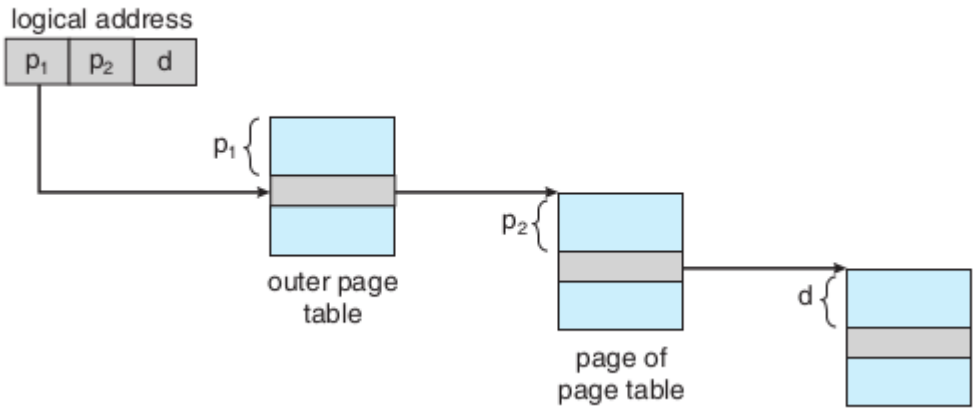
Paging: problem of large PT

- **64 bit address**
- **Suppose 20 bit offset**
  - **That means 2^20 = 1 MB pages**
  - **44 bit page number: 2^44 that is trillion sized page table!**
  - **Can’t have that big continuous page table!**

Paging: problem of large PT

- **32 bit address**
- **Suppose 12 bit offset**
  - **That means 2^12 = 4 KB pages**
  - **20 bit page number: 2^20 that is a million entries**
  - **Can’t always have that big continuous page table as well, for each process!**

Hierarchical paging



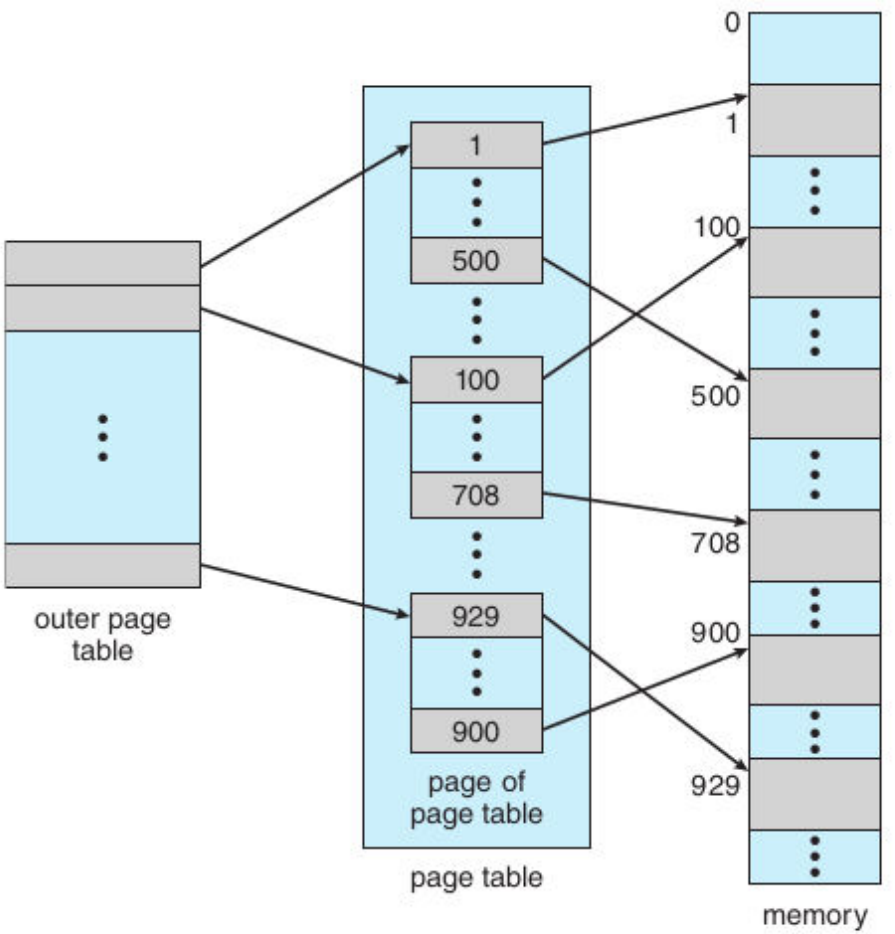
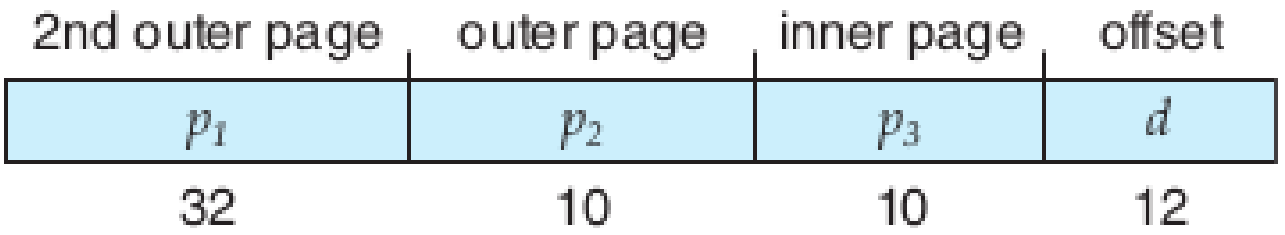


Figure 9.15 A two-level page-table scheme.

outer page	inner page	offset
p_1	p_2	d
42	10	12

More hierarchy



Problems with hierarchical paging

- **More number of memory accesses with each level !**
  - **Too slow !**
- **OS data structures also needed in that proportion**

Hashed page table

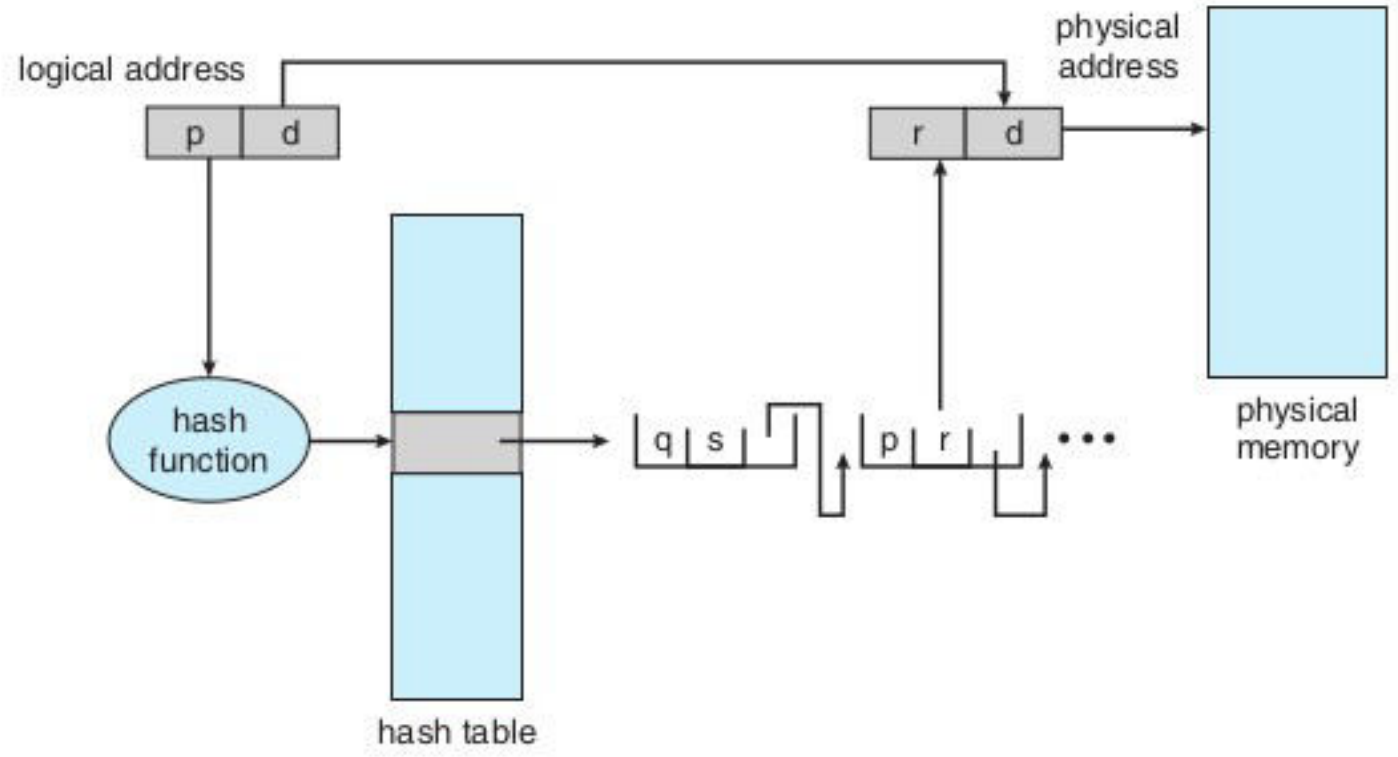


Figure 9.17 Hashed page table.

Inverted page table

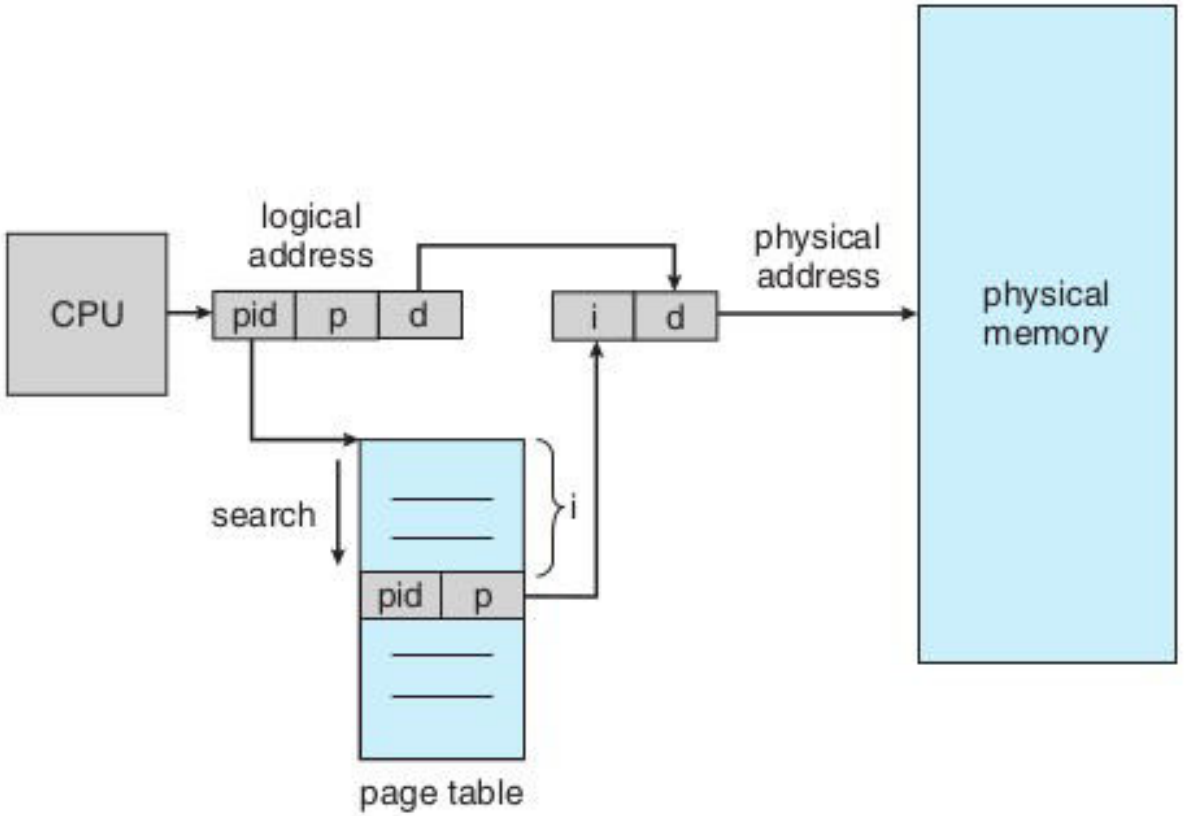


Figure 9.18 Inverted page table.

Normal page table – one per process --> Too much memory consumed

Inverted page table : global table – only one
Needs to store PID in the table entry

Examples of systems using inverted page
tables include the 64-bit Ultra SPARC and Power PC

virtual address
consists of a triple:
<process-id, page-number, offset>

Case Study: Oracle SPARC Solaris

- 64 bit SPARC processor , 64 bit Solaris OS
- Uses Hashed page tables
  - one for the kernel and one for all user processes.
  - Each hash-table entry : base + span (#pages)
    - Reduces number of entries required

Case Study: Oracle SPARC Solaris

- **Cashing levels: TLB (on CPU), TSB(in Memory), Page Tables (in Memory)**
  - **CPU implements a TLB that holds translation table entries (TTE s) for fast hardware lookups.**
  - **A cache of these TTEs resides in a in-memory translation storage buffer (TSB), which includes an entry per recently accessed page**
  - **When a virtual address reference occurs, the hardware searches the TLB for a translation.**
  - **If none is found, the hardware walks through the in memory TSB looking for the TTE that corresponds to the virtual address that caused the lookup**

Case Study: Oracle SPARC Solaris

- If a match is found in the TSB , the CPU copies the TSB entry into the TLB , and the memory translation completes.
- If no match is found in the TSB , the kernel is interrupted to search the hash table.
- The kernel then creates a TTE from the appropriate hash table and stores it in the TSB for automatic loading into the TLB by the CPU memory-management unit.
- Finally, the interrupt handler returns control to the MMU , which completes the address translation and retrieves the requested byte or word from main memory.

Swapping

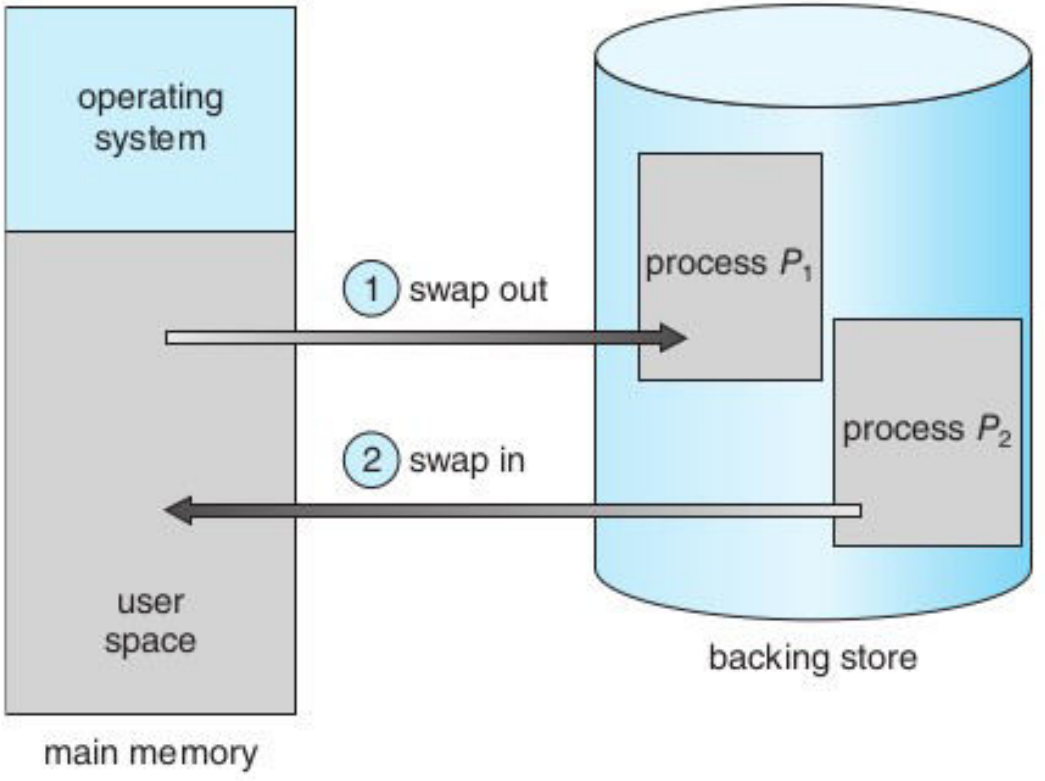


Figure 9.19 Standard swapping of two processes using a disk as a backing store.

Swapping

- **Standard swapping**
  - **Entire process swapped in or swapped out**
  - **With continous memory management**
- **Swapping with paging**
  - **Some pages are “paged out” and some “paged in”**
  - **Term “paging” refers to paging with swapping now**

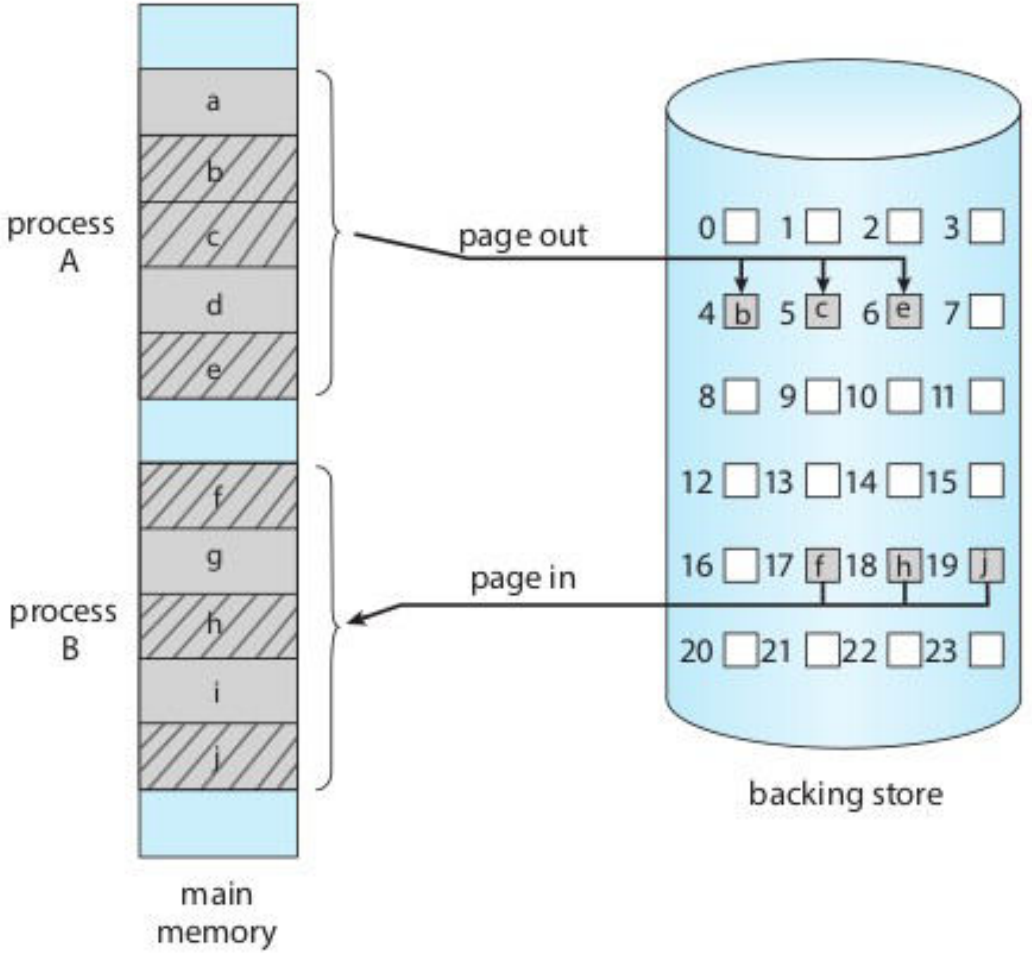


Figure 9.20 Swapping with paging.

Words of caution about ‘paging’

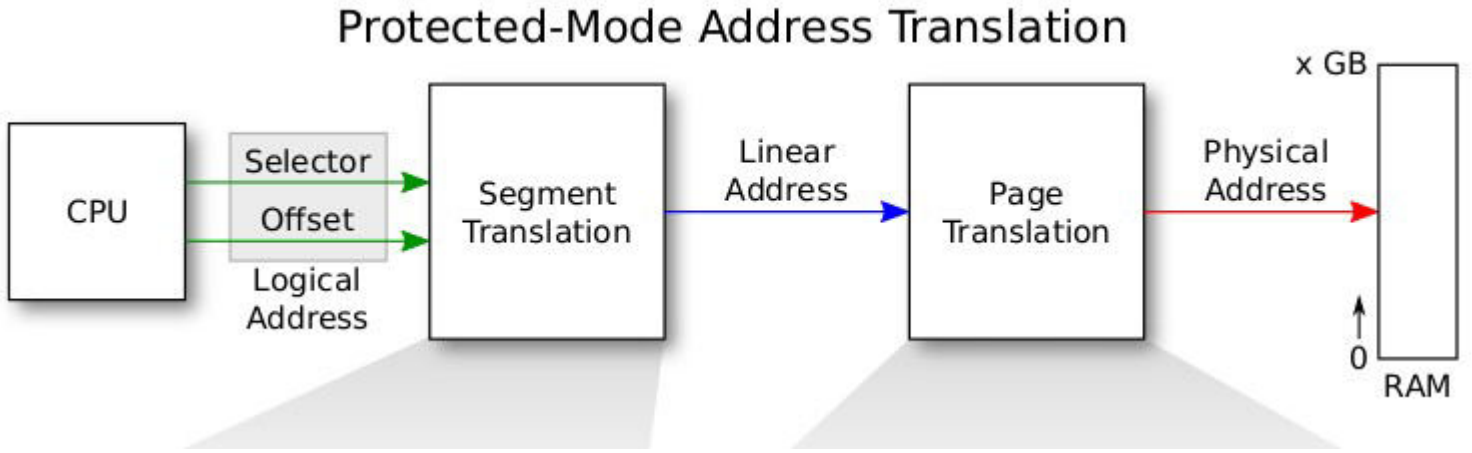
- **Not as simple as it sounds when it comes to implementation**
  - **Writing OS code for this is challenging**

Some numbers and their ‘meaning’

- These numbers occur very frequently in discussion
- 0x 80000000 = 2 GB = KERNBASE
- 0x 100000 = 1 MB = EXTMEM
- 0x 80100000 = 2GB + 1MB = KERNLINK
- 0x E000000 = 224 MB = PHYSTOP
- 0x FE000000 = 3.96 GB = 4064 MB = DEVSPACE
  - 4096 – 4064 = 32 MB left on top

X86 Memory Management

X86 address : protected mode address translation



X86 paging

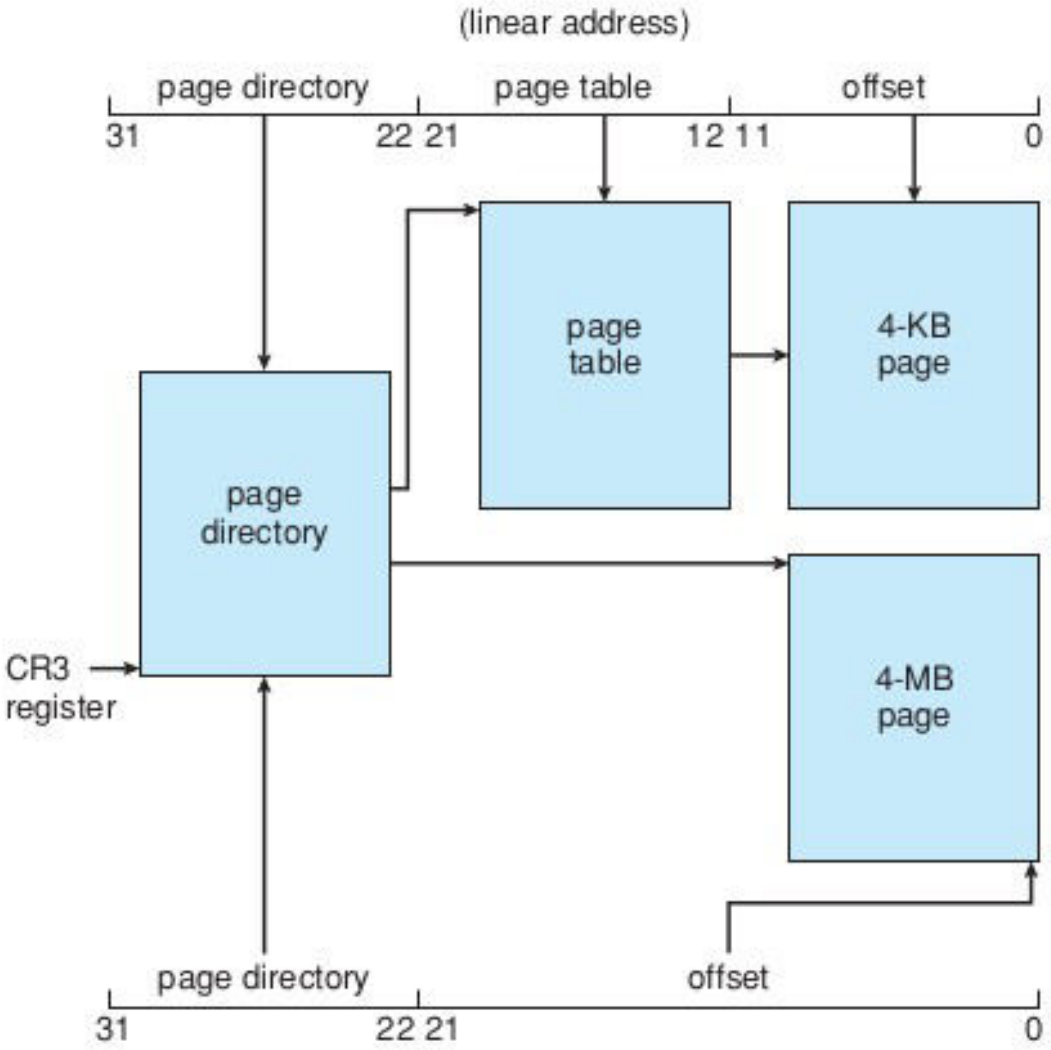
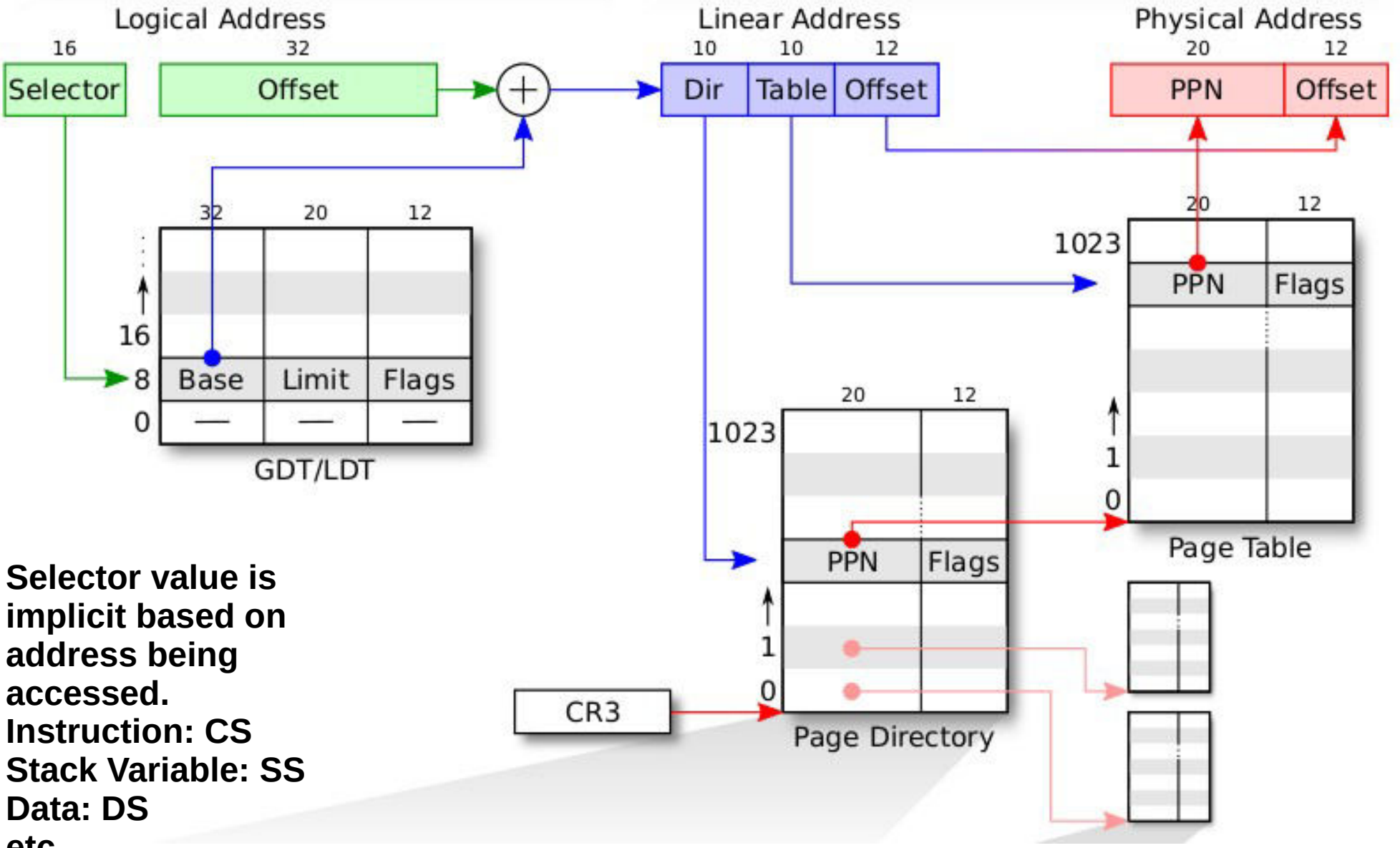


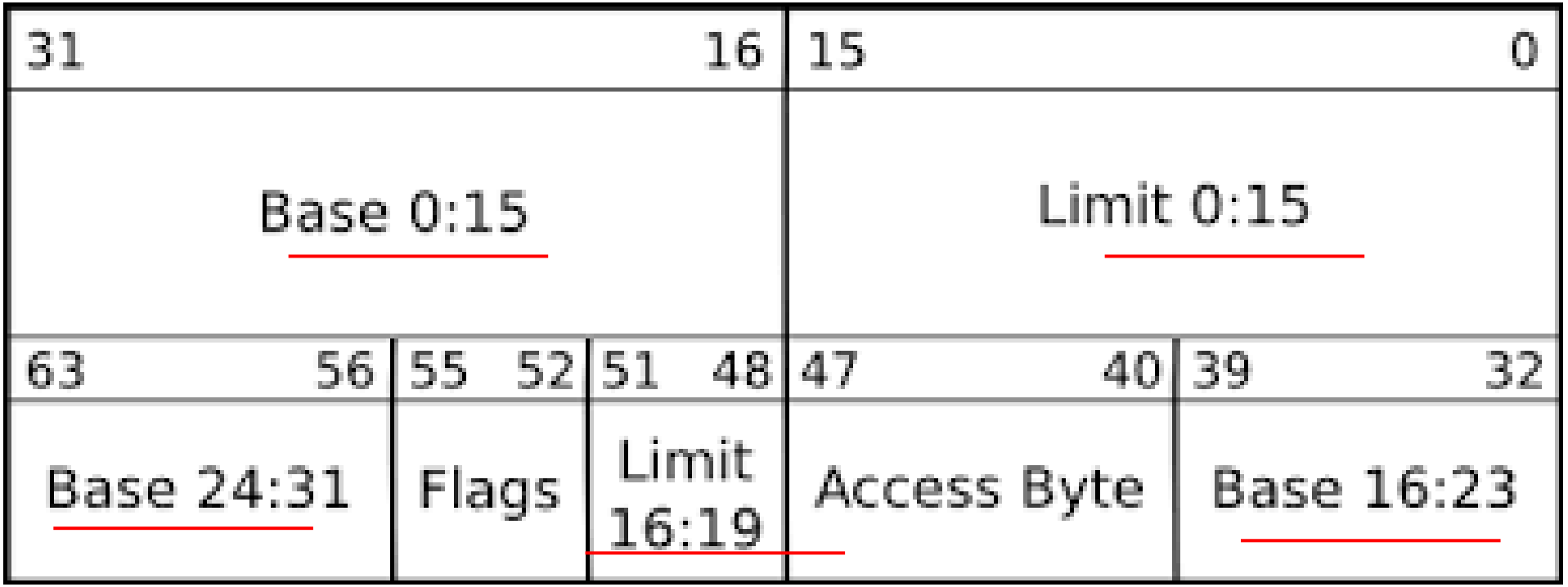
Figure 8.23 Paging in the IA-32 architecture.

Segmentation + Paging

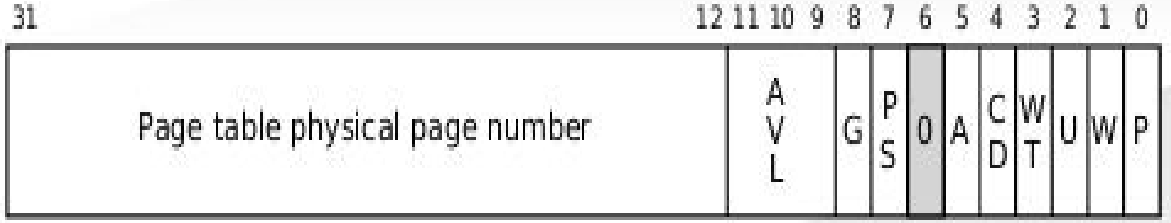


Selector value is implicit based on address being accessed.
Instruction: CS
Stack Variable: SS
Data: DS
etc.

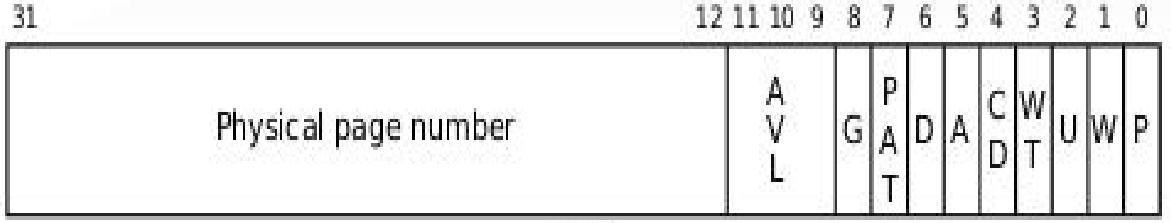
GDT Entry


31
16
15
0
Base 0:15
Limit 0:15
63
56
55
52
51
48
47
40
39
32
Base 24:31
Flags
Limit 16:19
Access Byte
Base 16:23

Page Directory Entry (PDE)
Page Table Entry (PTE)



PDE



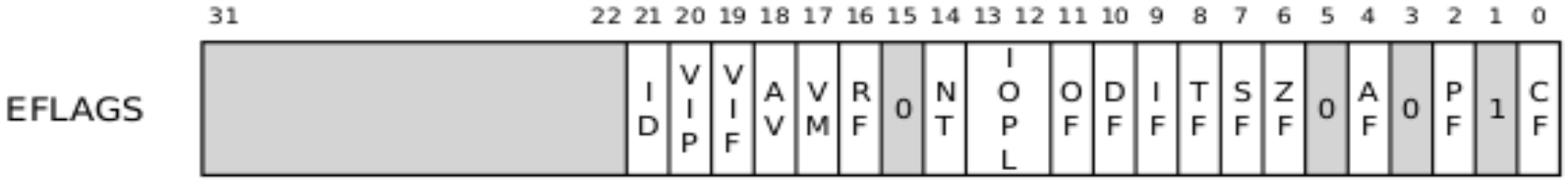
PTE

P	Present
W	Writable
U	User
WT	1=Write-through, 0=Write-back
CD	Cache disabled
A	Accessed
D	Dirty
PS	Page size (0=4KB, 1=4MB)
PAT	Page table attribute index
G	Global page
AVL	Available for system use

Segment selector

TI Table index (0=GDT, 1=LDT)
RPL Requester privilege level

EFLAGS register



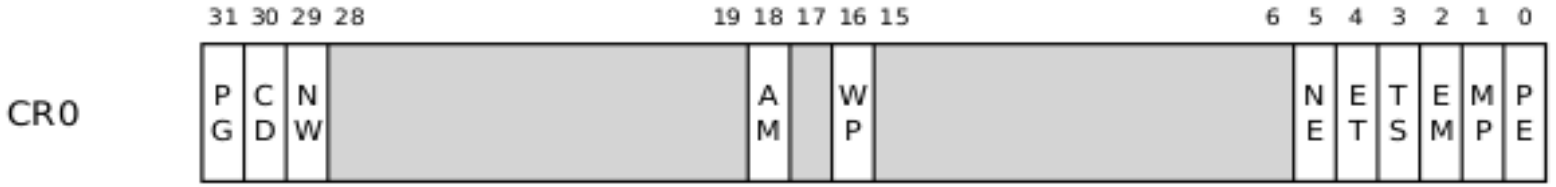
Status flags
- CF Carry flag
- PF Parity flag
- AF Auxiliary carry flag
- ZF Zero flag
- SF Sign flag
- OF Overflow flag

Control flags
- DF Direction flag

System flags
- TF Trap flag
- IF Interrupt enable flag
- IOPL I/O privilege level
- NT Nested task

- RF Resume flag
- VM Virtual-8086 mode
- AC Alignment check
- VIF Virtual intr. flag
- VIP Virtual intr. pending
- ID ID flag

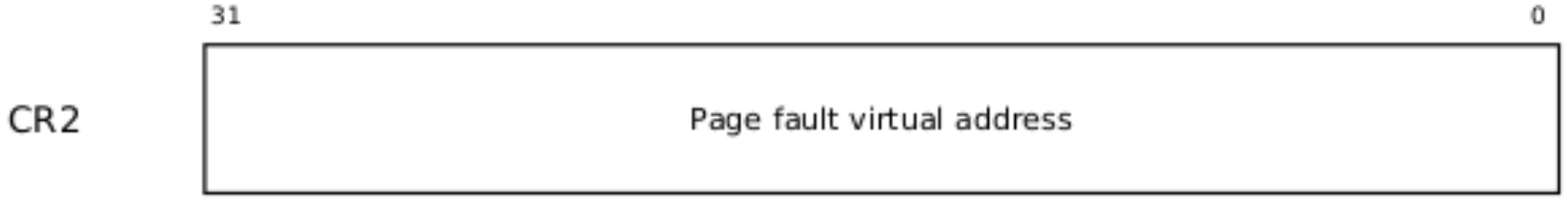
CR0



PE	Protection enabled	ET	Extension type	NW	Not write-through
MP	Monitor coprocessor	NE	Numeric error	CD	Cache disable
EM	Emulation	WP	Write protect	PG	Paging
TS	Task switched	AM	Alignment mask		

PG: Paging enabled or not **WP: Write protecion on/off**
PE: Protection Enabled --> protected mode.

CR2



CR3

PWT Page-level writes transparent

PCT Page-level cache disable

CR4

VME	Virtual-8086 mode extensions
PVI	Protected-mode virtual interrupts
TSD	Time stamp disable
DE	Debugging extensions
PSE	Page size extensions
PAE	Physical-address extension
MCE	Machine check enable
PGE	Page-global enable
PCE	Performance counter enable
OSFXSR	OS FXSAVE/FXRSTOR support
OSXMM-EXCPT	OS unmasked exception support

mmu.h : paging related macros

```
#define PTXSHIFT        12      // offset of PTX in a linear address
#define PDXSHIFT        22      // offset of PDX in a linear address
#define PDX(va)   (((uint)(va) >> PDXSHIFT) & 0x3FF)// page directory index
#define PTX(va)   (((uint)(va) >> PTXSHIFT) & 0x3FF)// page table index
// construct virtual address from indexes and offset
#define PGADDR(d, t, o) ((uint)((d) << PDXSHIFT | (t) << PTXSHIFT |
(o)))
// +--------10------+-------10-------+---------12----------+
// | Page Directory |   Page Table    | Offset within Page  |
// |      Index     |      Index      |                     |
// +----------------+-----------------+---------------------+
//  \--- PDX(va) --/ \--- PTX(va) --/
```

mmu.h : paging related macros

```
// Page directory and page table constants.
#define NPDENTRIES      1024    // # directory entries per page directory
#define NPTENTRIES      1024    // # PTEs per page table
#define PGSIZE          4096    // bytes mapped by a page
#define PGROUNDUP(sz)  (((sz)+PGSIZE-1) & ~(PGSIZE-1))
#define PGROUNDDOWN(a) (((a)) & ~(PGSIZE-1))
```

mmu.h : paging related macros

```
// Page table/directory entry flags.
#define PTE_P           0x001   // Present
#define PTE_W           0x002   // Writeable
#define PTE_U           0x004   // User
#define PTE_PS          0x080   // Page Size


// Address in page table or page directory entry
#define PTE_ADDR(pte)   ((uint)(pte) & ~0xFFF) // get
all but last 12 bits
#define PTE_FLAGS(pte)  ((uint)(pte) &  0xFFF) // get
last 12 bits
```

mmu.h : Segmentation related macros

```
// various segment selectors.

#define SEG_KCODE 1  // kernel code

#define SEG_KDATA 2  // kernel data+stack

#define SEG_UCODE 3  // user code

#define SEG_UDATA 4  // user data+stack

#define SEG_TSS   5  // this process's task state
```

mmu.h : Segmentation related macros

```
// various segment selectors.
#define SEG_KCODE 1  // kernel code
#define SEG_KDATA 2  // kernel data+stack
#define SEG_UCODE 3  // user code
#define SEG_UDATA 4  // user data+stack
#define SEG_TSS   5  // this process's task state
#define NSEGS     6
```

mmu.h : Segmentation related macros

```
struct segdesc { // 64 bit in size
  uint lim_15_0 : 16;  // Low bits of segment limit
  uint base_15_0 : 16; // Low bits of segment base address
  uint base_23_16 : 8; // Middle bits of segment base address
  uint type : 4;       // Segment type (see STS_ constants)
  uint s : 1;          // 0 = system, 1 = application
  uint dpl : 2;        // Descriptor Privilege Level
  uint p : 1;          // Present
  uint lim_19_16 : 4;  // High bits of segment limit
  uint avl : 1;        // Unused (available for software use)
  uint rsv1 : 1;       // Reserved
  uint db : 1;         // 0 = 16-bit segment, 1 = 32-bit segment
  uint g : 1;          // Granularity: limit scaled by 4K when set
  uint base_31_24 : 8; // High bits of segment base address
};
```

mmu.h : Segmentation related code

```
// Application segment type bits
#define STA_X       0x8     // Executable segment
#define STA_W       0x2     // Writeable (non-executable segments)
#define STA_R       0x2     // Readable (executable segments)

// System segment type bits
#define STS_T32A    0x9     // Available 32-bit TSS
#define STS_IG32    0xE     // 32-bit Interrupt Gate
#define STS_TG32    0xF     // 32-bit Trap Gate
```

Code from bootasm.S bootmain.c is over!
Kernel is loaded.
Now kernel is going to prepare itself

main() in main.c

- Initializes “free list” of page frames
  - In 2 steps. Why?
- Sets up page table for kernel
- Detects configuration of all processors
- Starts all processors
  - Just like the first processor
- Creates the first process!

- Initializes
  - LAPIC on each processor, IOAPIC
  - Disables PIC
  - “Console” hardware (the standard I/O)
  - Serial Port
  - Interrupt Descriptor Table
  - Buffer Cache
  - Files Table
  - Hard Disk (IDE)

main() in main.c

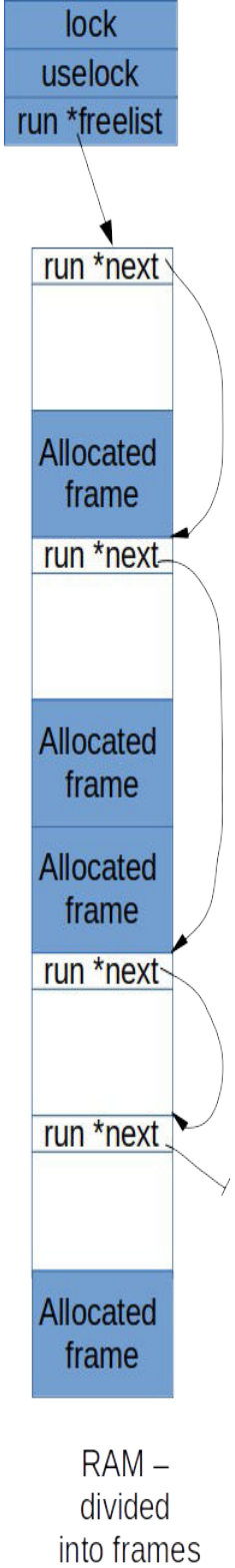
```
int

main(void) {

  kinit1(end,
P2V(4*1024*1024)); // phys
page allocator

  kvmalloc();      // kernel page
table
```

```
void

kinit1(void *vstart, void
*vend) {

  initlock(&kmem.lock,
"kmem");

  kmem.use_lock = 0;

  freerange(vstart, vend);

}
```

main() in main.c

```
void

freerange(void *vstart, void
*vend)

{

  char *p;

  p =
(char*)PGROUNDUP((uint)vst
art);

  for(; p + PGSIZE <=
(char*)vend; p += PGSIZE)

    kfree(p);

}
```

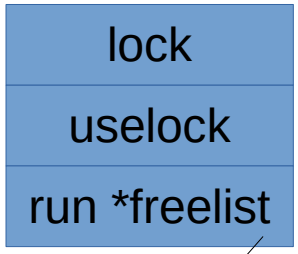
```
kfree(char *v) {
  struct run *r;
  if((uint)v % PGSIZE || v <
end || V2P(v) >= PHYSTOP)
    panic("kfree");
  // Fill with junk to catch
dangling refs.
  memset(v, 1, PGSIZE);
  if(kmem.use_lock)
    acquire(&kmem.lock);
  r = (struct run*)v;
  r->next = kmem.freelist;
  kmem.freelist = r;
  if(kmem.use_lock)
    release(&kmem.lock); }
```



Free List in XV6 Obtained after main() -> kinit1()

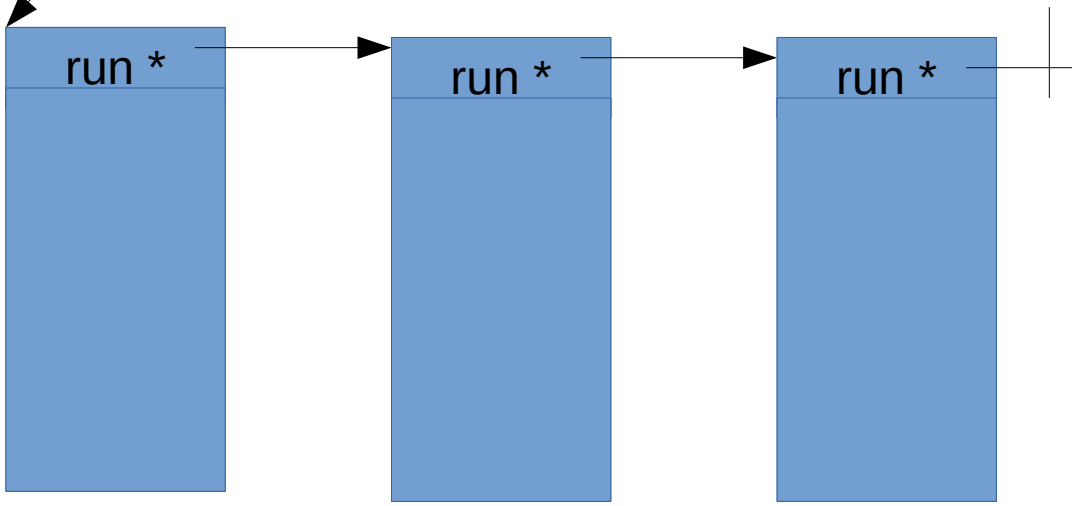
Pages obtained Between
end = 801154a8 = 2049 MB to P2V(4MB) = 2052 MB
Remember
Right now Logical = Physical address.

Actually like this in memory



kmem

Seen independently



Back to main()

```
int

main(void) {

  kinit1(end, P2V(4*1024*1024)); // phys page allocator

  kvmalloc();      // kernel page table
```

```
// Allocate one page table for the machine for the kernel address

// space for scheduler processes.

void

kvmalloc(void)

{

  kpgdir = setupkvm();

  switchkvm();

}
```

Back to main()

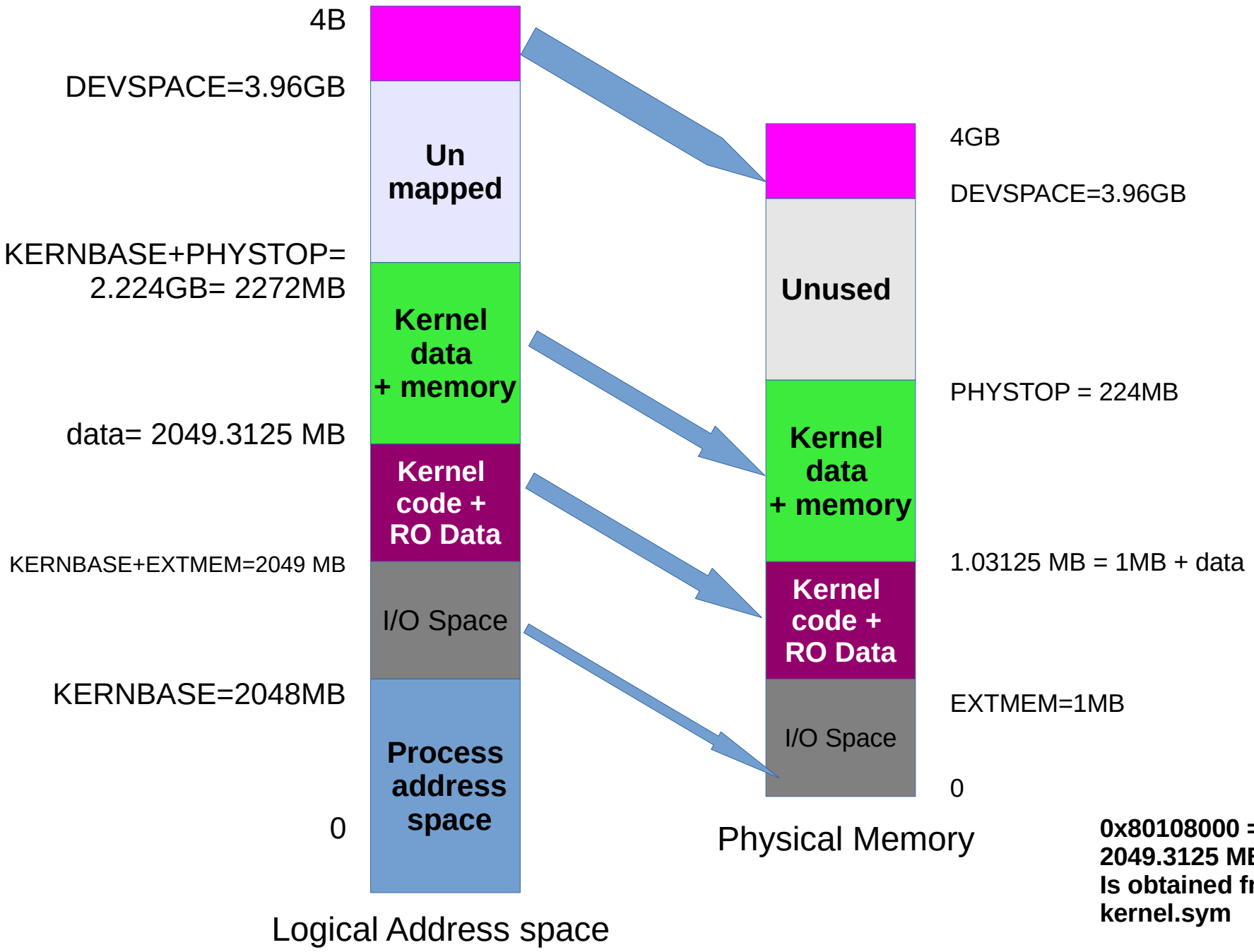
```
int
main(void) {
  kinit1(end, P2V(4*1024*1024)); // phys page allocator
  kvmalloc();      // kernel page table
```

```
// Allocate one page table for the machine for the kernel address
// space for scheduler processes.
void
kvmalloc(void)
{
  kpgdir = setupkvm(); // global var kpgdir
  switchkvm();
}
```

```
pde_t*
setupkvm(void)
{
  pde_t *pgdir;
  struct kmap *k;
  if((pgdir = (pde_t*)kalloc()) == 0)
    return 0;
  memset(pgdir, 0, PGSIZE);
  if (P2V(PHYSTOP) > (void*)DEVSPACE)
    panic("PHYSTOP too high");
 for(k = kmap; k < &kmap[NELEM(kmap)]; k++)
    if(mappages(pgdir, k->virt, k->phys_end - k->phys_start,
                (uint)k->phys_start, k->perm) < 0) {
      freevm(pgdir);
      return 0;
    }
  return pgdir;
}
```

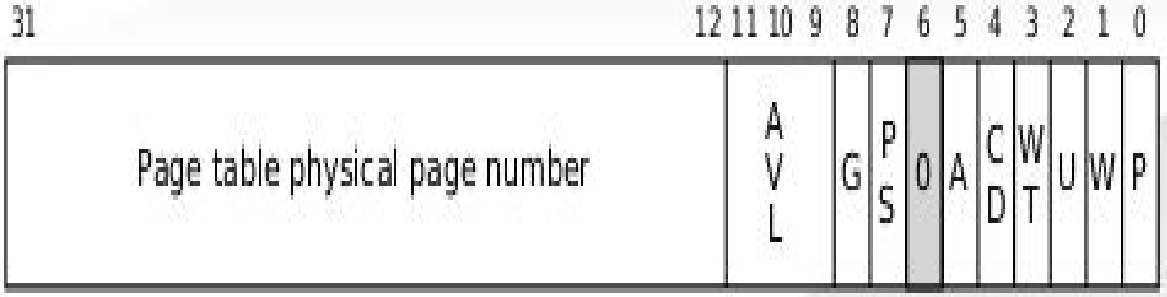
```
static struct kmap {
  void *virt;
  uint phys_start;
  uint phys_end;
  int perm;
} kmap[] = {
 { (void*)KERNBASE, 0,             EXTMEM,    PTE_W}, // I/O space
 { (void*)KERNLINK, V2P(KERNLINK), V2P(data), 0},     // kern text+rodata
 { (void*)data,     V2P(data),     PHYSTOP,   PTE_W}, // kern data+memory
 { (void*)DEVSPACE, DEVSPACE,      0,         PTE_W}, // more devices
};
```

kmap[] mappings done in kvmalloc(). This shows segmentwise, entries are done in page directory and page table for corresponding VA-> PA mappings

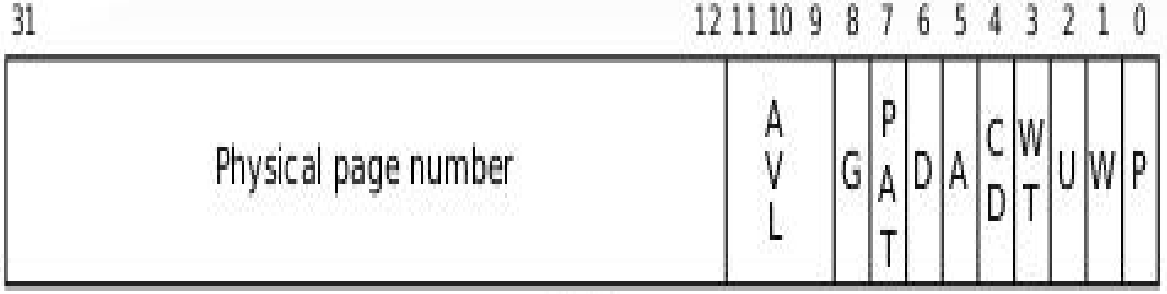


0x80108000 =data = 2049.3125 MB Is obtained from kernel.sym

Remidner: PDE and PTE entries

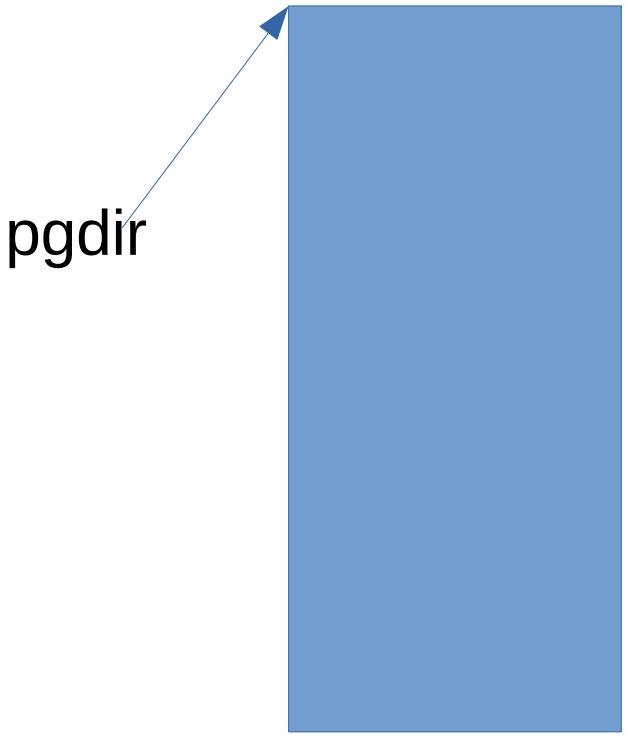


PDE

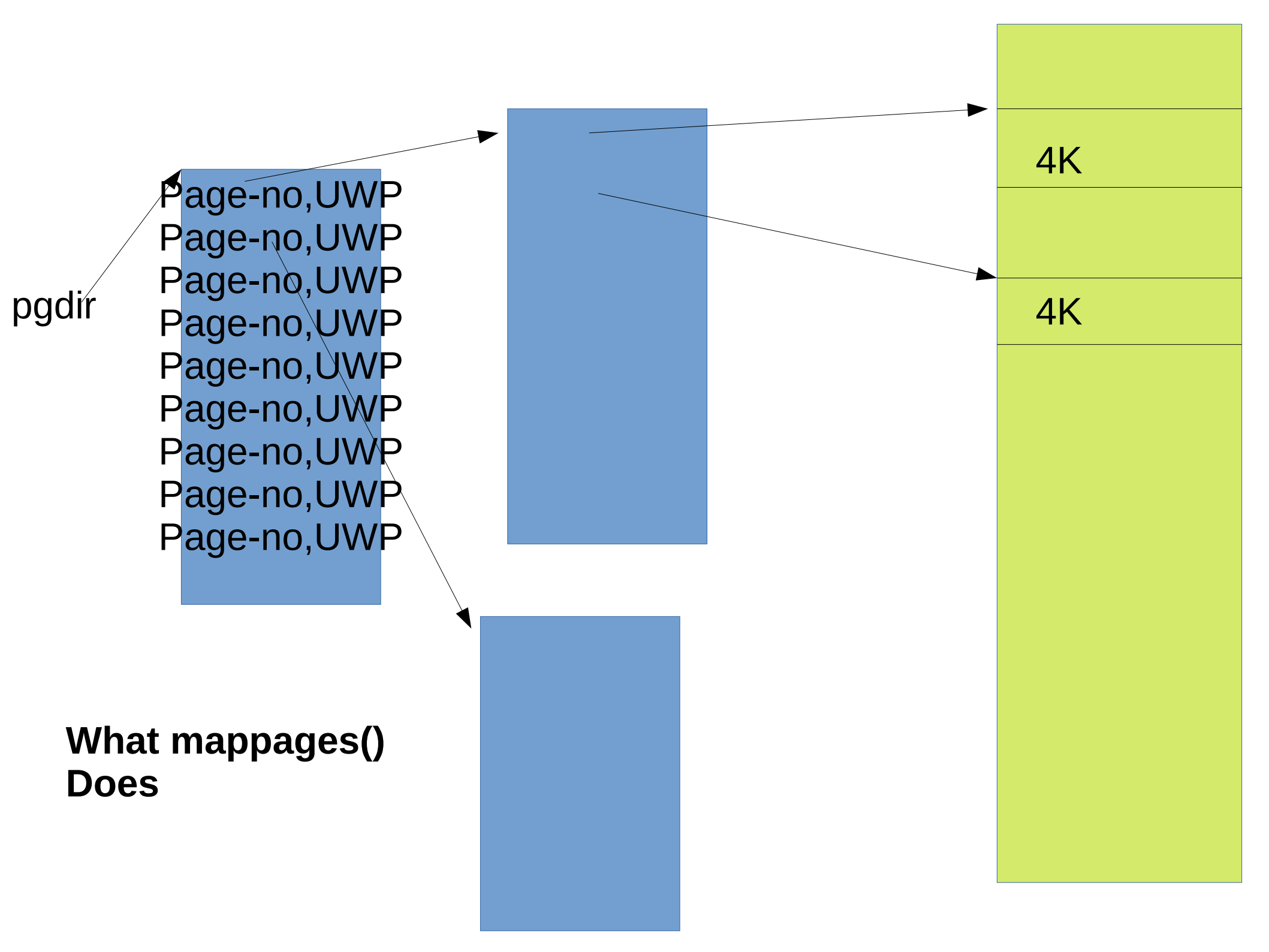


PTE

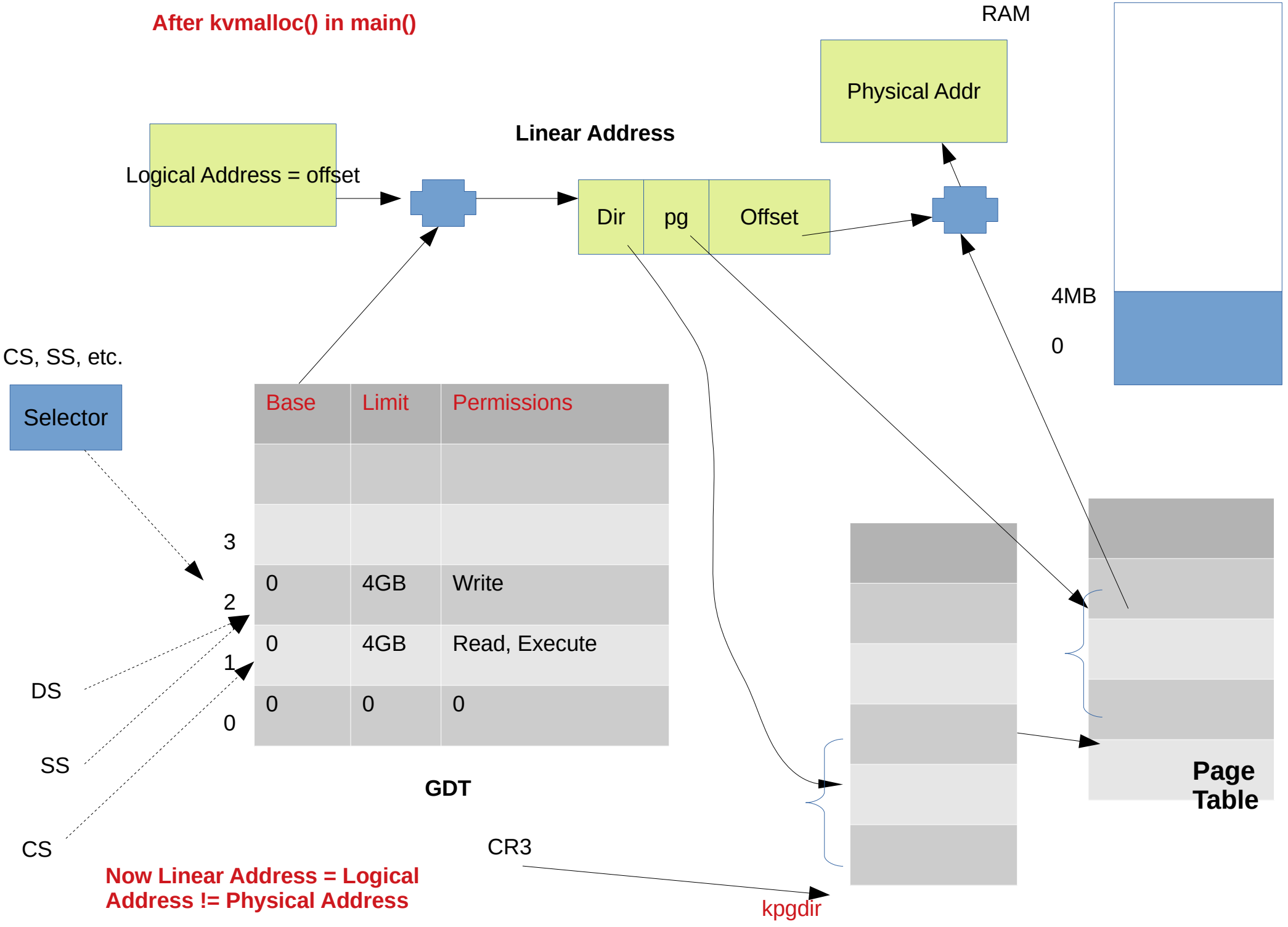
P	Present
W	Writable
U	User
WT	1=Write-through, 0=Write-back
CD	Cache disabled
A	Accessed
D	Dirty
PS	Page size (0=4KB, 1=4MB)
PAT	Page table attribute index
G	Global page
AVL	Available for system use



Before mappages()

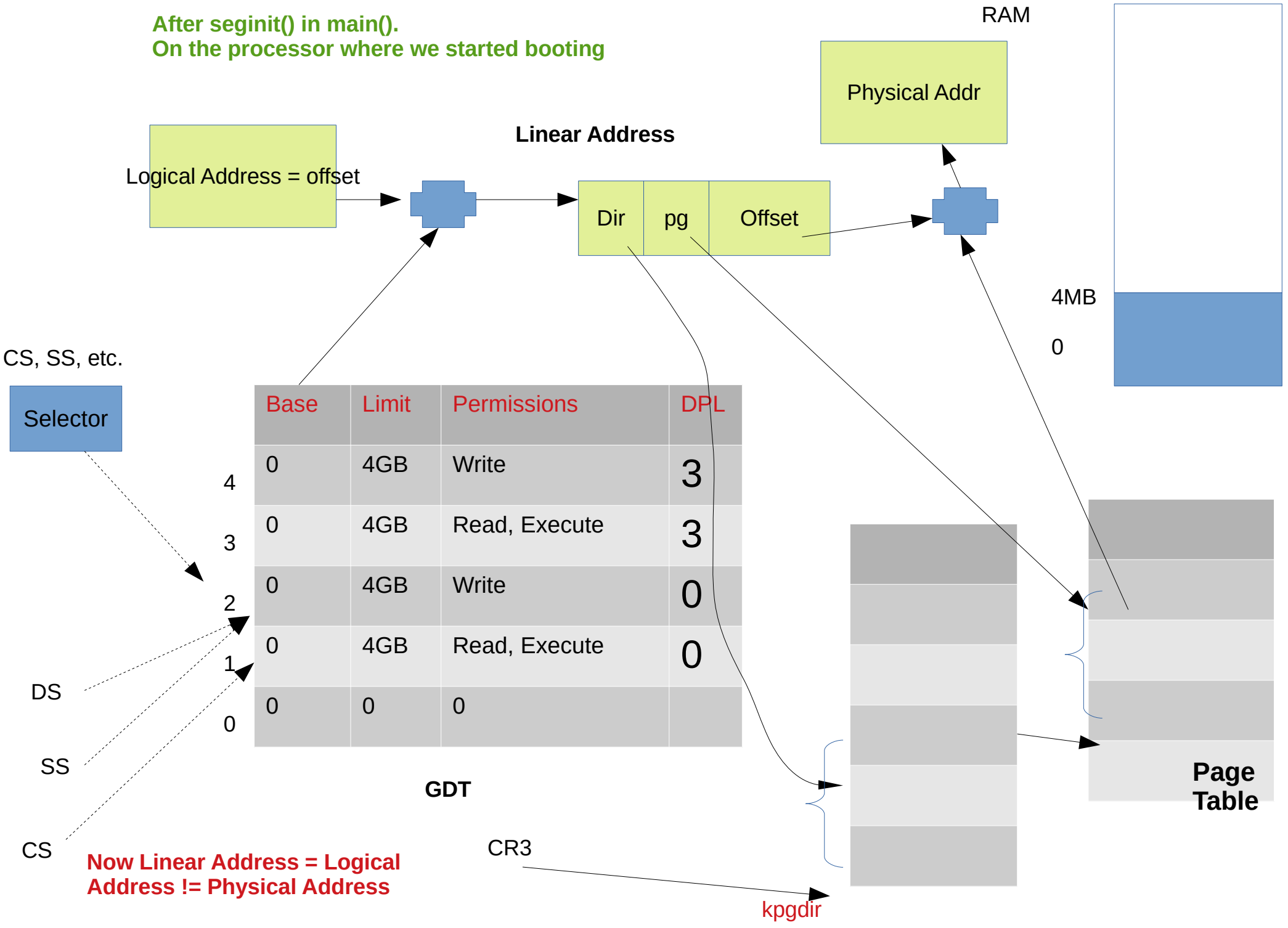


What mappages()
Does


After kvmalloc() in main()
RAM
Physical Addr
Linear Address
Logical Address = offset
Dir
pg
Offset
4MB
0
CS, SS, etc.
Selector
Base
Limit
Permissions
3
0
4GB
Write
2
0
4GB
Read, Execute
1
0
0
0
DS
SS
CS
GDT
Page Table
CR3
kpgdir
Now Linear Address = Logical Address != Physical Address

main()->seginit()

- Re-initialize GDT
- Once and forever now
- Just set 4 entries
  - All spanning 4 GB
  - Differing only in permissions and privilege level

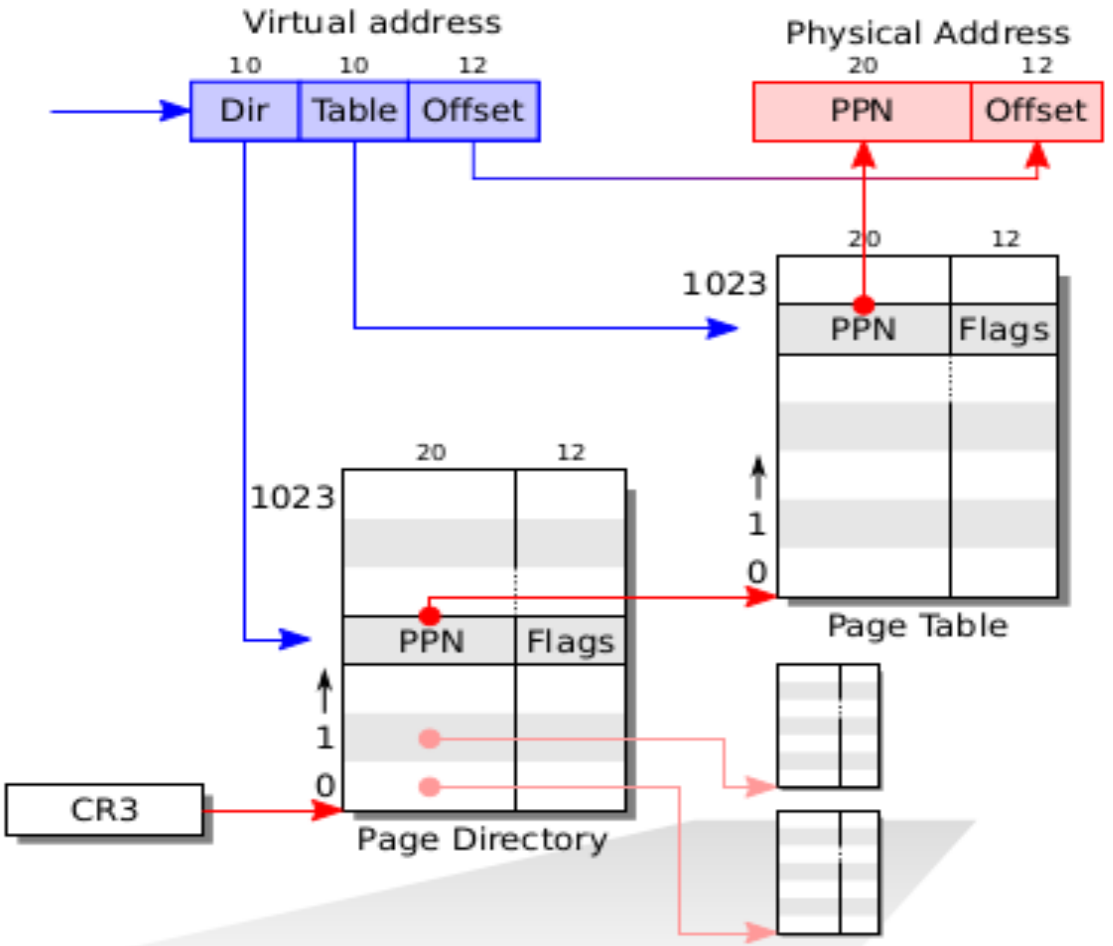

After seginit() in main().
On the processor where we started booting
RAM
Physical Addr
Linear Address
Logical Address = offset
Dir
pg
Offset
4MB
0
CS, SS, etc.
Selector
Base
Limit
Permissions
DPL
4
0
4GB
Write
3
3
0
4GB
Read, Execute
3
2
0
4GB
Write
0
1
0
4GB
Read, Execute
0
0
0
0
0
DS
SS
CS
GDT
Page Table
CR3
kpgdir
Now Linear Address = Logical Address != Physical Address

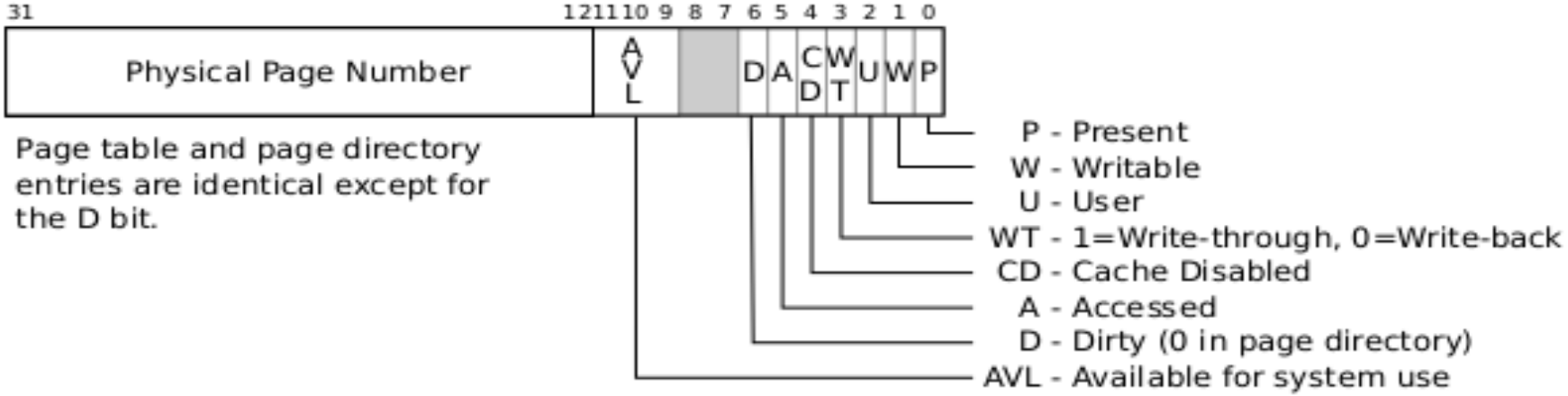
After seginit()

- While running kernel code, necessary to switch CS, DS, SS to index 1,2,2 in GDT
- While running user code, necessary to switch CS, DS, SS to index 3,4,4 in GDT
- This happens automatically as part of “trap” handling (covered separately)

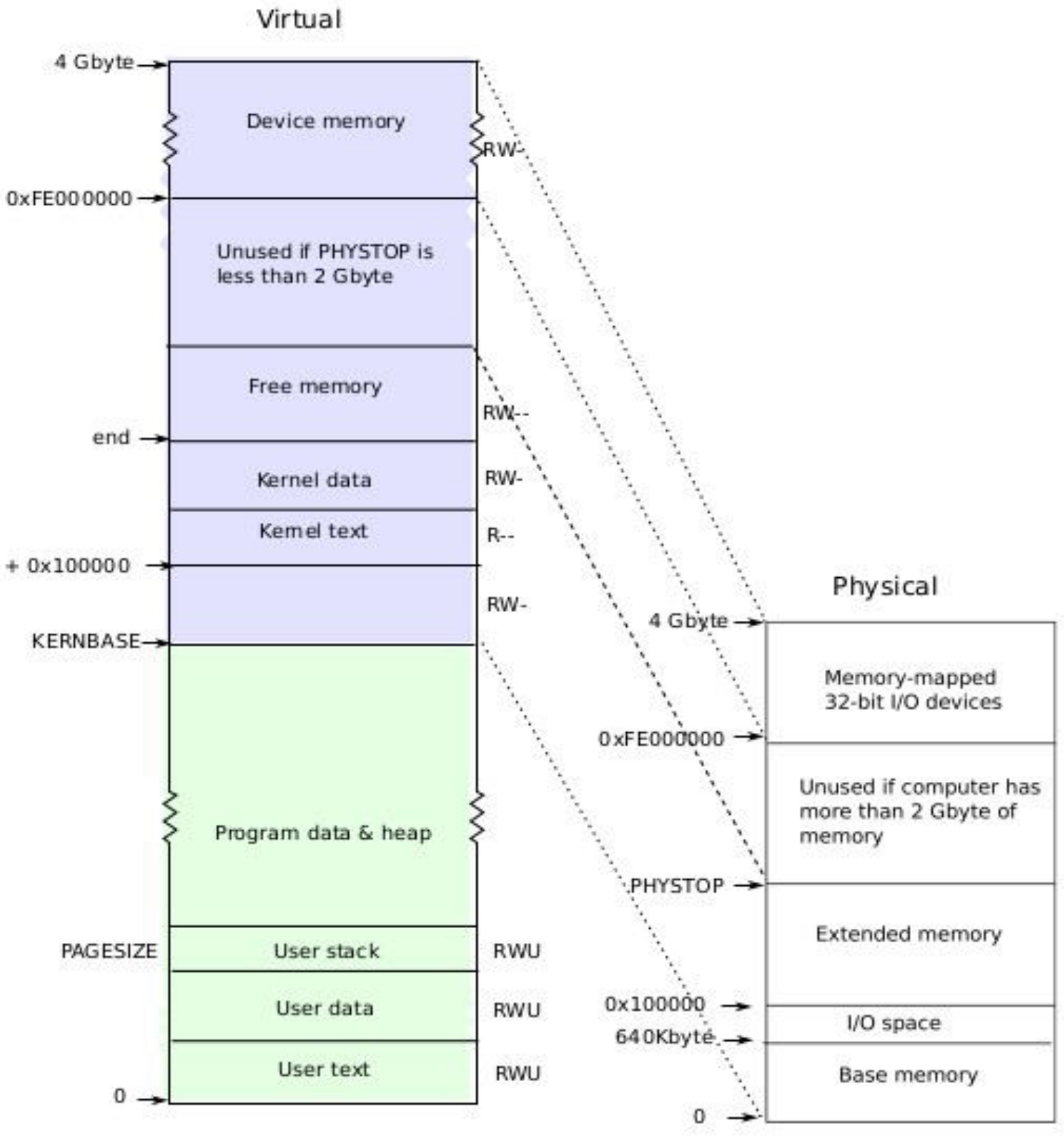
Memory Management

X86 page table hardware

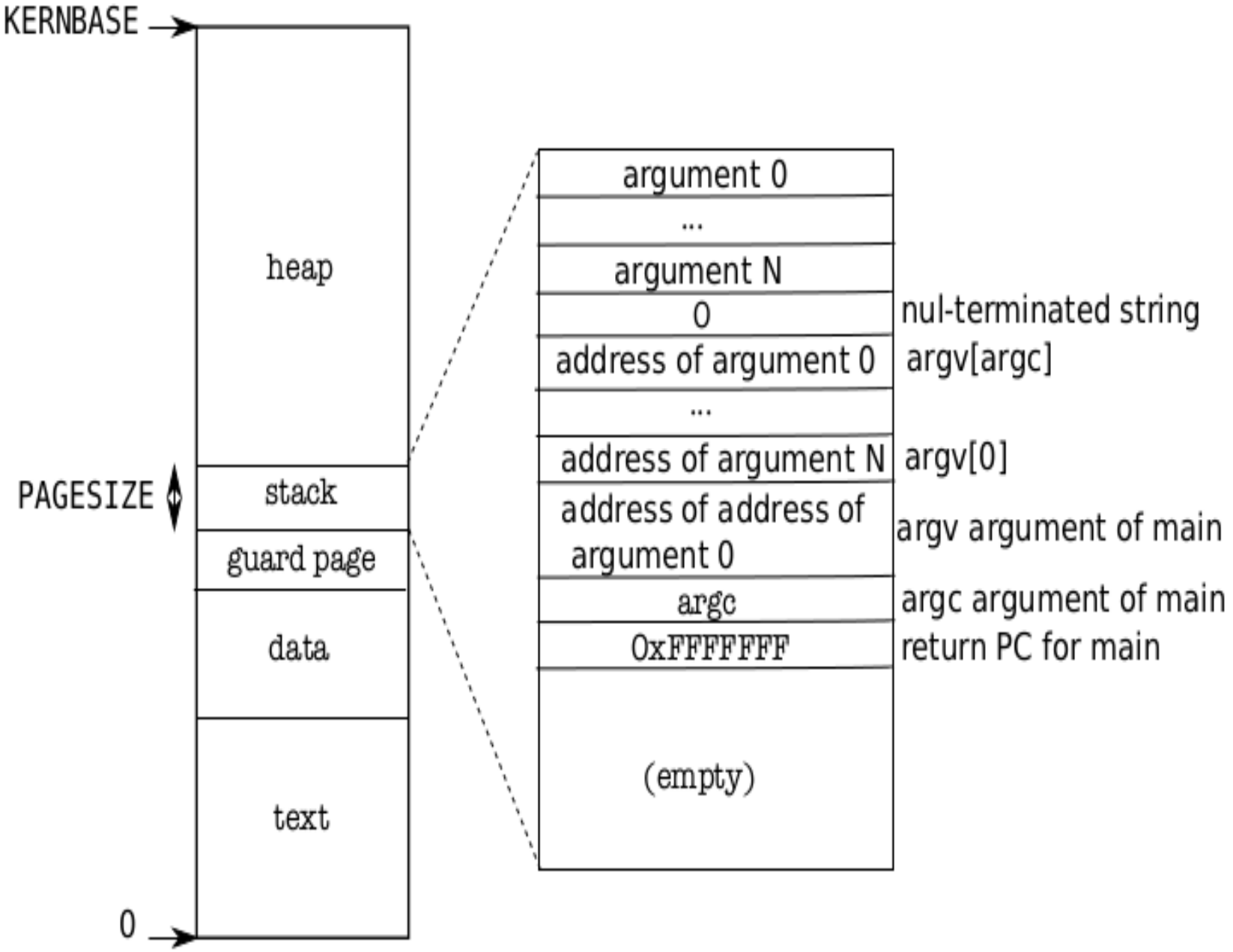




Page table and page directory entries are identical except for the D bit.



Layout of process's VA space

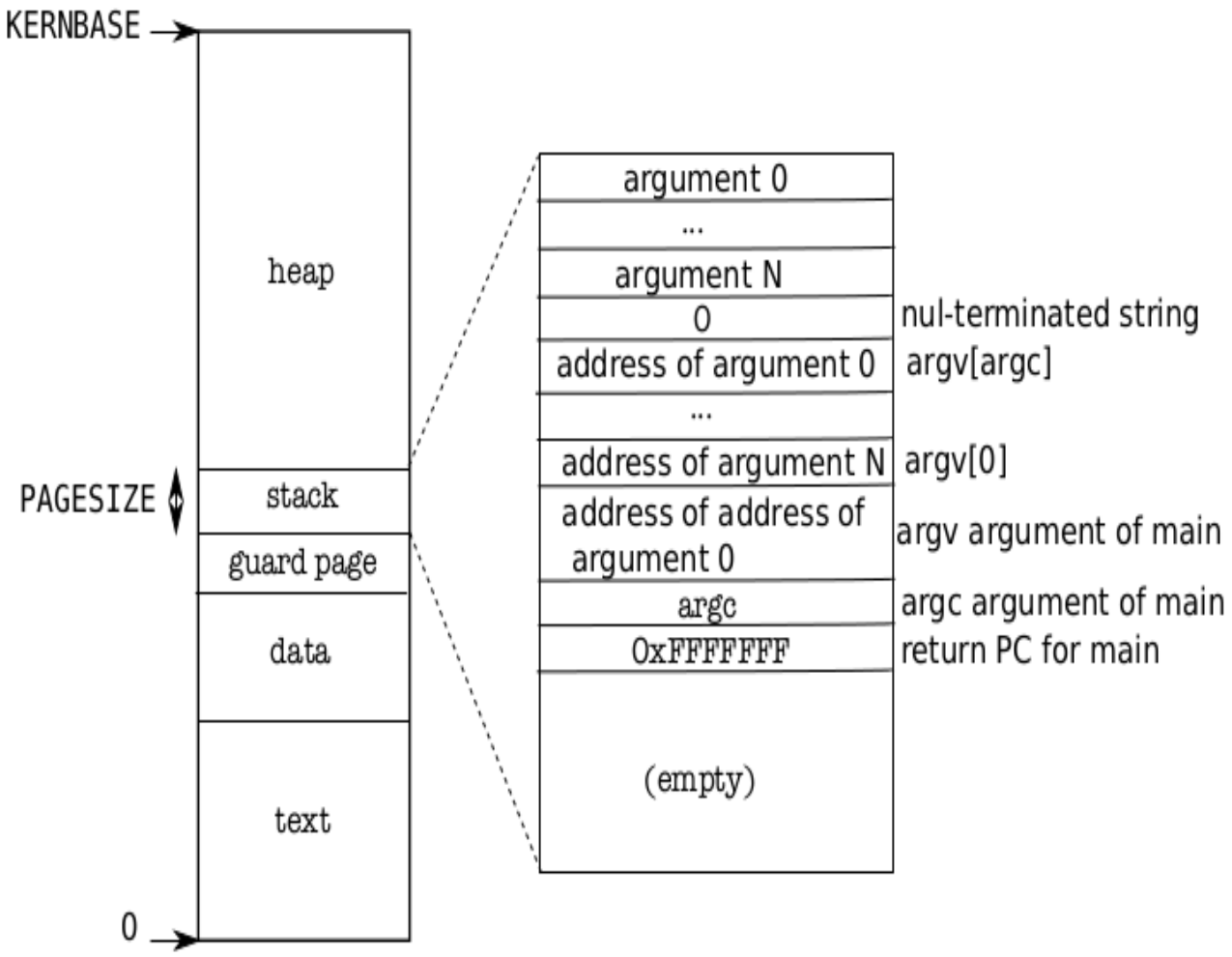


Memory Layout of a user process

After exec()

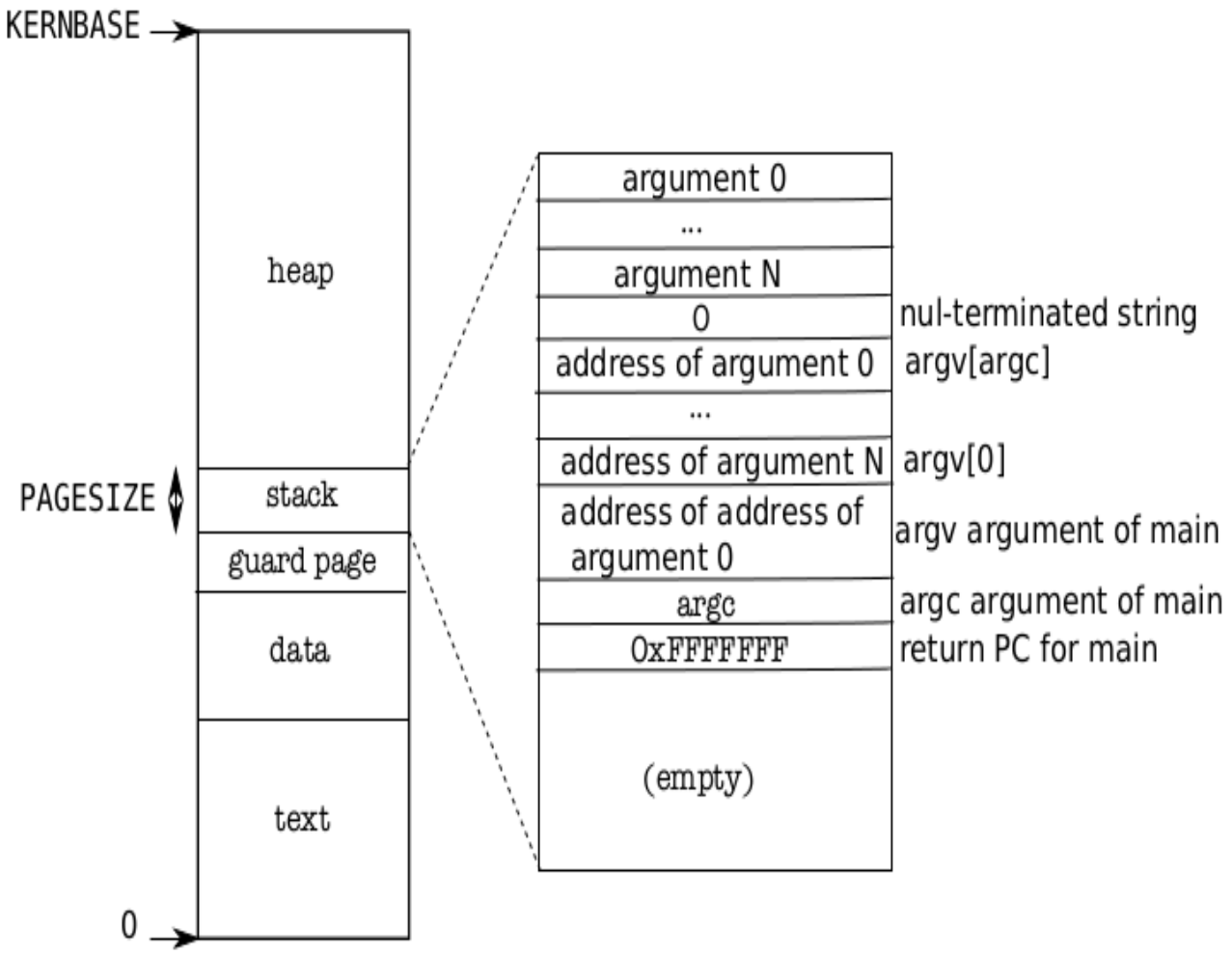
Note the argc, argv on stack

Memory Layout of a user process



The “guard page” is just a mapping in page table. No frame allocated. It’s marked as invalid. So if stack grows (due to many function calls), then OS will detect it with an exception

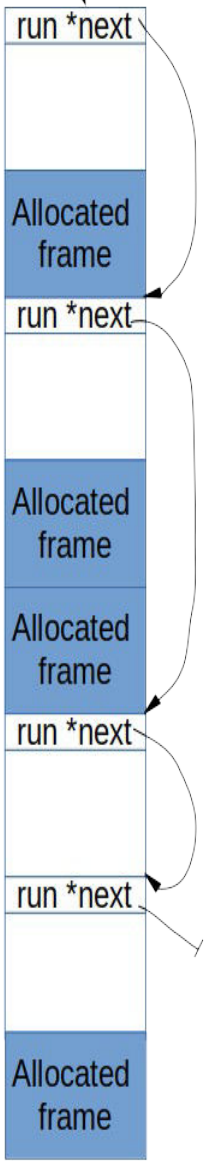
Memory Layout of a user process



On sbrk()
The system call to grow process's address space.
Calls growproc()

growproc()
Allocate a frame, Add an entry in page table at the top (above proc->sz)
//This entry can't go beyond KERNBASE
Calls switchuvm()

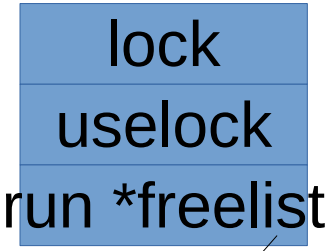
Switchuvm()
Ultimately loads CR3, invalidating cache



RAM – divided into frames

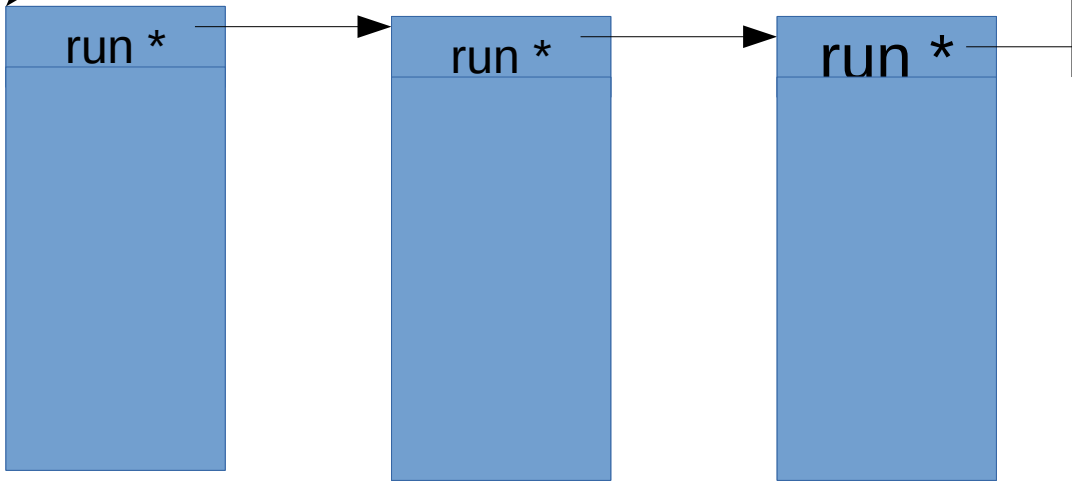
Free List in XV6

Actually like this in memory



kmem

Seen independently



exec()

- **sys_exec()**

  exec(path, argv)

- **exec(parth, argv)**

```
ip = namei(path))
readi(ip, (char*)&elf, 0, sizeof(elf)) != sizeof(elf)
for(i=0, off=elf.phoff; i<elf.phnum; i++, off+=sizeof(ph)){
    if(readi(ip, (char*)&ph, off, sizeof(ph)) != sizeof(ph))
    if((sz = allocuvm(pgdir, sz, ph.vaddr + ph.memsz)) == 0)
    if(loaduvm(pgdir, (char*)ph.vaddr, ip, ph.off, ph.filesz) < 0)
}
```

exec()

- **exec(parth, argv)**

```
// Allocate two pages at the next page boundary.
 // Make the first inaccessible.  Use the second as the user stack.
 sz = PGROUNDUP(sz);
 if((sz = allocuvm(pgdir, sz, sz + 2*PGSIZE)) == 0)
// Push argument strings, prepare rest of stack in ustack.
  for(argc = 0; argv[argc]; argc++) {
    sp = (sp - (strlen(argv[argc]) + 1)) & ~3;
    if(copyout(pgdir, sp, argv[argc], strlen(argv[argc]) + 1) < 0)
```