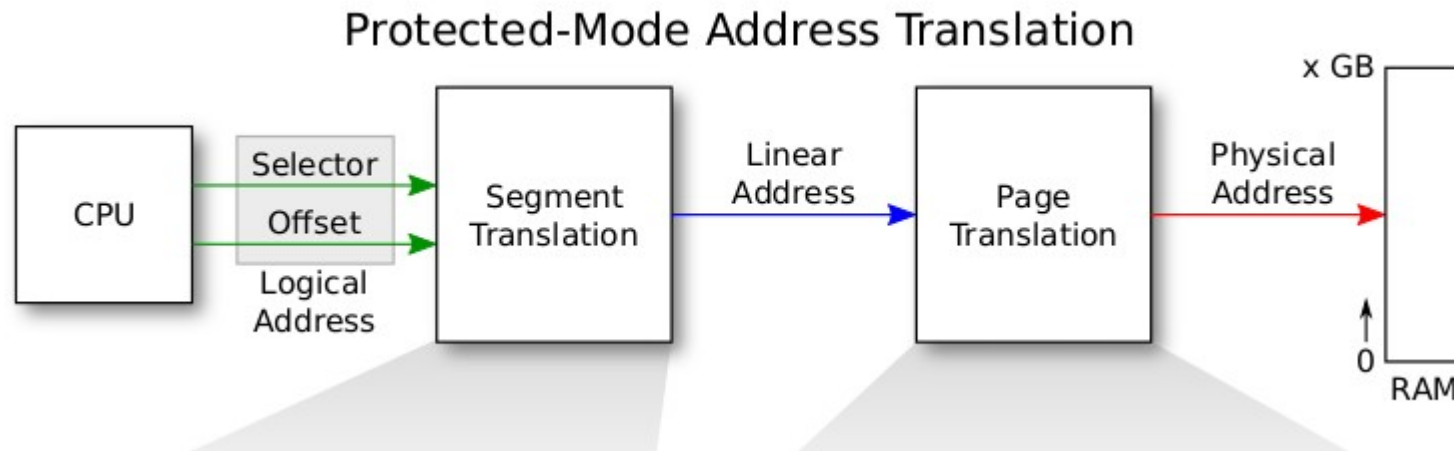# Some numbers and their 'meaning'
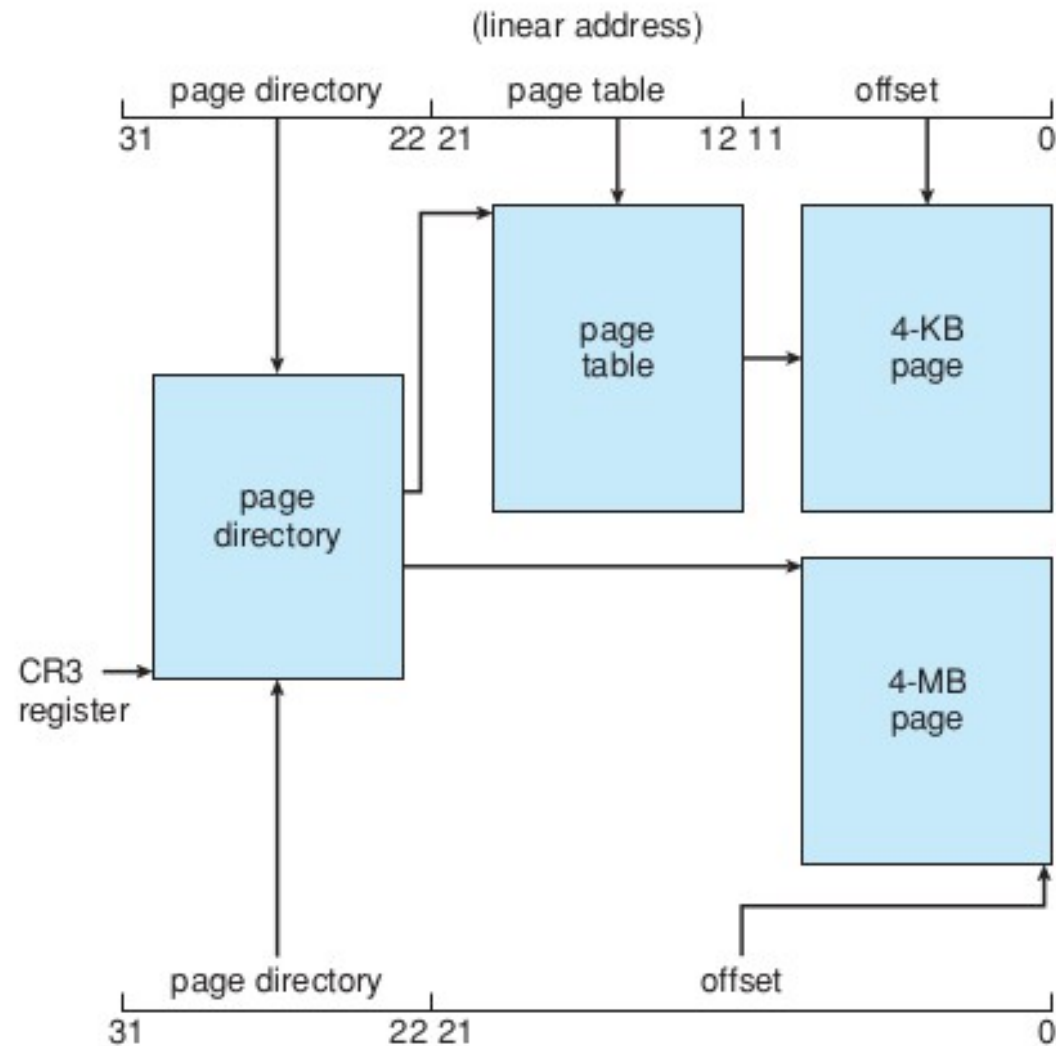
- **These numbers occur very frequently in discussion**

- **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
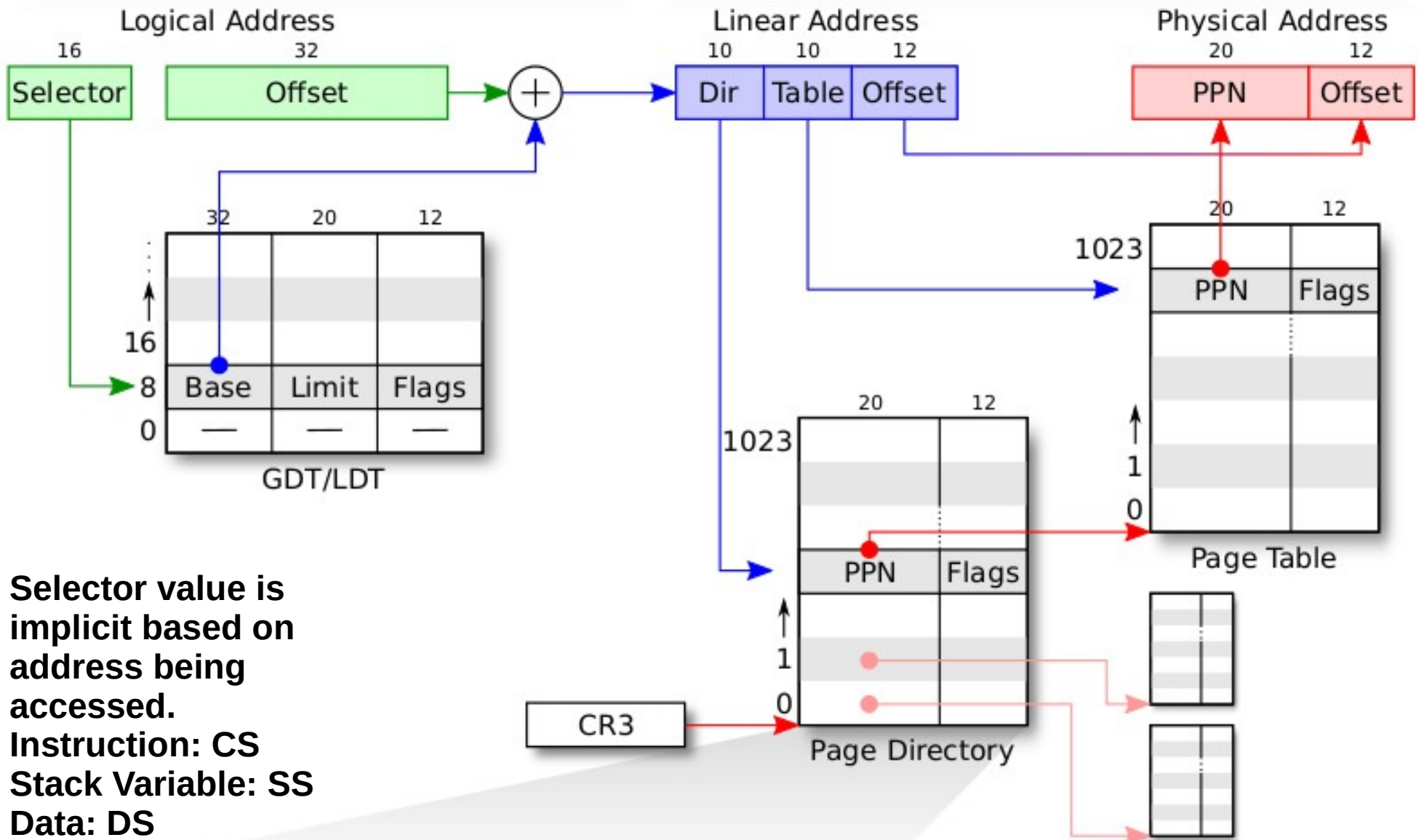

Protected-Mode Address Translation

X86 paging

Figure 8.23 Paging in the IA-32 architecture.

Segmentation + Paging
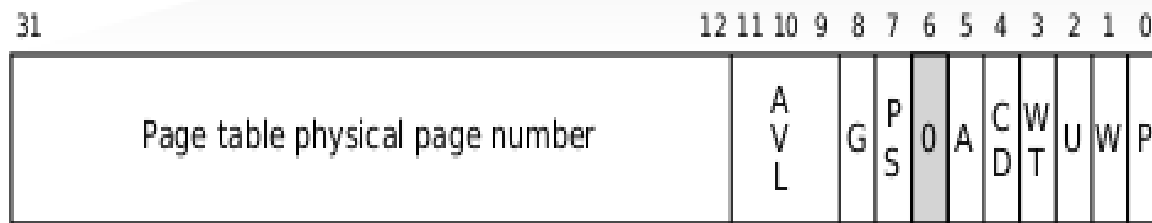

Selector value is implicit based on address being accessed.
Instruction: CS
Stack Variable: SS
Data: DS
etc.

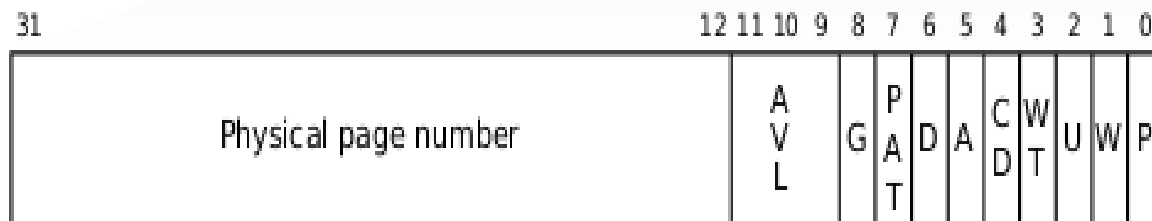# GDT Entry

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)


```
31                              12 11 10 9  8  7  6  5  4  3  2  1  0
┌─────────────────────────────────┬───┬─┬─┬─┬─┬─┬─┬─┬─┬─┐
│                                 │ A │ │P│ │ │C│W│ │ │ │
│  Page table physical page number│ V │G│S│0│A│D│T│U│W│P│
│                                 │ L │ │ │ │ │ │ │ │ │ │
└─────────────────────────────────┴───┴─┴─┴─┴─┴─┴─┴─┴─┴─┘
                    PDE
```

```
31                              12 11 10 9  8  7  6  5  4  3  2  1  0
┌─────────────────────────────────┬───┬─┬─┬─┬─┬─┬─┬─┬─┬─┐
│                                 │ A │ │P│ │ │C│W│ │ │ │
│  Physical page number           │ V │G│A│D│A│D│T│U│W│P│
│                                 │ L │ │T│ │ │ │ │ │ │ │
└─────────────────────────────────┴───┴─┴─┴─┴─┴─┴─┴─┴─┴─┘
                    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

15		3	2	1	0
CS/DS/ES FS/GS/SS	Selector index		TI	RPL	

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

EFLAGS register

31	22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
EFLAGS	ID VIP VIF AC VM RF 0 NT IOPL OF DF IF TF SF ZF 0 AF 0 PF 1 CF

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

31	30	29	28	...	19	18	17	16	15	...	6	5	4	3	2	1	0
PG	CD	NW			AM			WP			NE	ET	TS	EM	MP	PE

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

31		0
	Page fault virtual address	

CR2

CR3

31		12 11		5 4	3	2	0

CR3   Page-Directory-Table Base Address   P C D   P W T

PWT   Page-level writes transparent   PCT   Page-level cache disable

CR4

31		11 10 9 8 7 6 5 4 3 2 1 0

CR4   OSXMM   OSFX   PCE   PGE   MCE   PAE   PSE   DE   TSD   PVI   VME

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

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

```c
void

freerange(void *vstart, void *vend)

{

  char *p;

  p = (char*)PGROUNDUP((uint)vstart);

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

    kfree(p);

}
```

```c
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); }
```
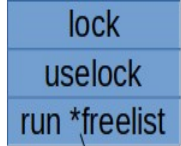
Free List in XV6 Obtained after main() -> kinit1()

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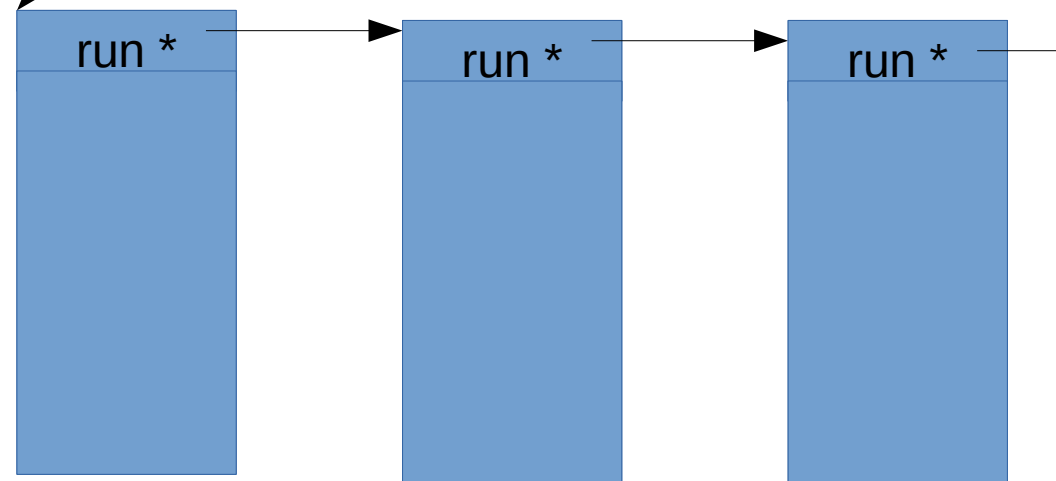**Remember**
Right now Logical = Physical address.
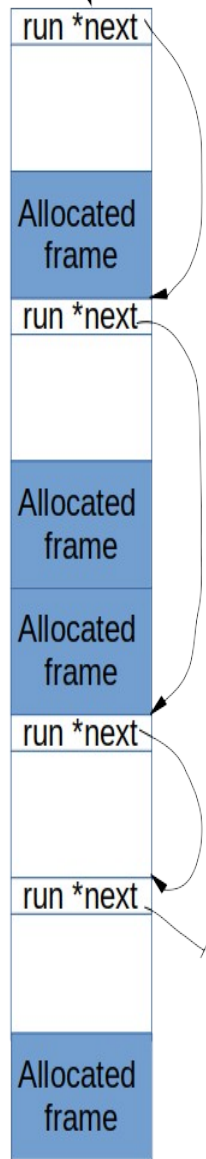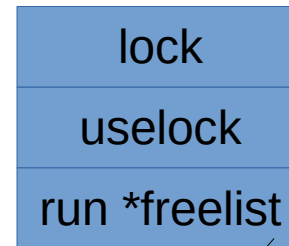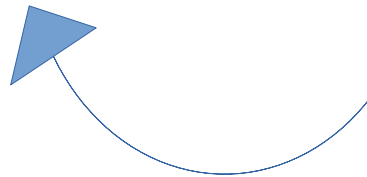
lock
uselock
run *freelist

run *next

Allocated frame

run *next

Allocated frame

Allocated frame

run *next

run *next

Allocated frame

RAM – divided into frames

Actually like this in memory

lock

uselock

run *freelist

kmem

Seen independently

run *

run *

run *

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();
}
```

```c
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;
}
```

```c
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
};
```
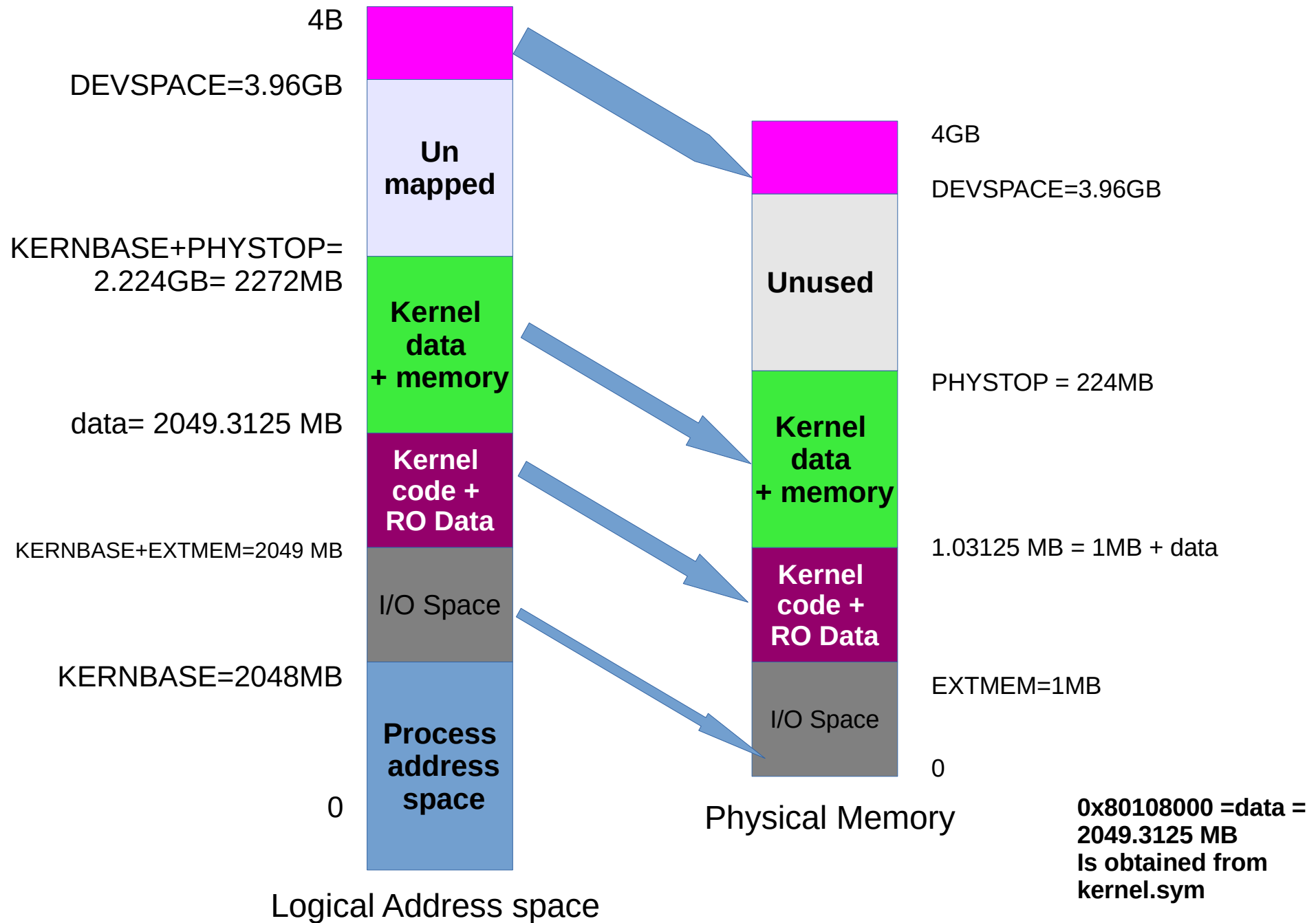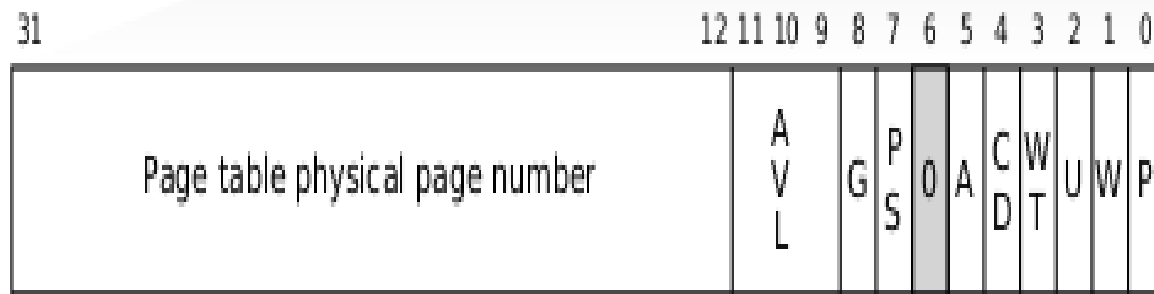
kmap[] mappings done in kvmalloc(). This shows segmentwise, entries are done in page directory and page table for corresponding VA-> PA mappings
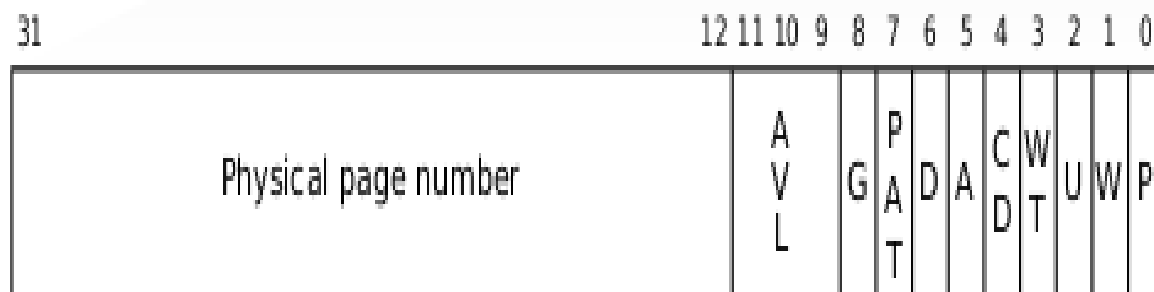
4B

DEVSPACE=3.96GB

KERNBASE+PHYSTOP=
2.224GB= 2272MB

data= 2049.3125 MB

KERNBASE+EXTMEM=2049 MB

KERNBASE=2048MB

0

| Un mapped |
| Kernel data + memory |
| Kernel code + RO Data |
| I/O Space |
| Process address space |

Logical Address space

4GB

DEVSPACE=3.96GB

| Unused |
| PHYSTOP = 224MB |
| Kernel data + memory |
| 1.03125 MB = 1MB + data |
| Kernel code + RO Data |
| EXTMEM=1MB |
| I/O Space |

0

Physical Memory

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

Remidner: PDE and PTE entries

PDE

31		12	11 10 9	8	7	6	5	4	3	2	1	0
Page table physical page number			AVL	G	PS	0	A	CD	WT	U	W	P

PTE

31		12	11 10 9	8	7	6	5	4	3	2	1	0
Physical page number			AVL	G	PAT	D	A	CD	WT	U	W	P

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
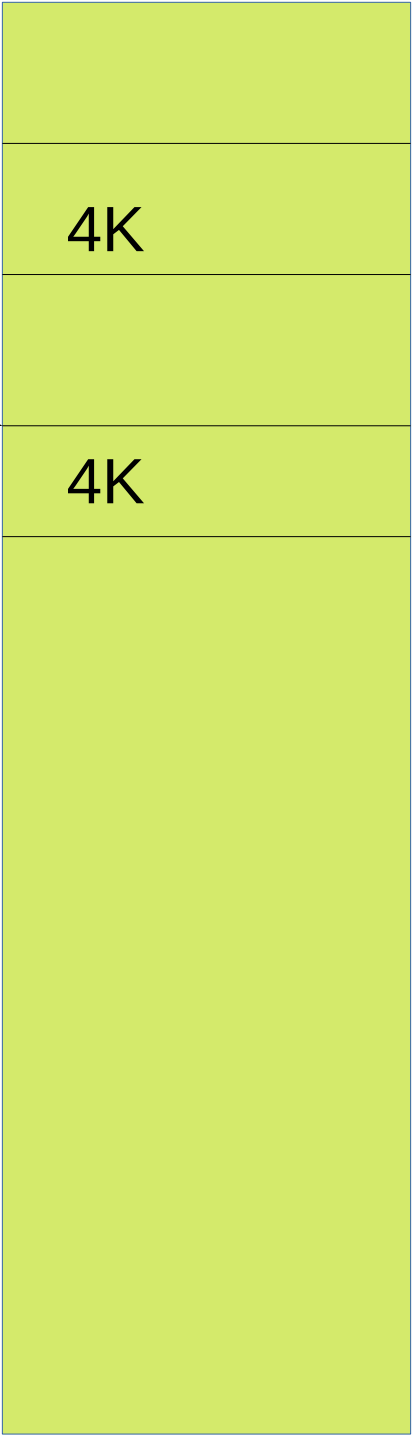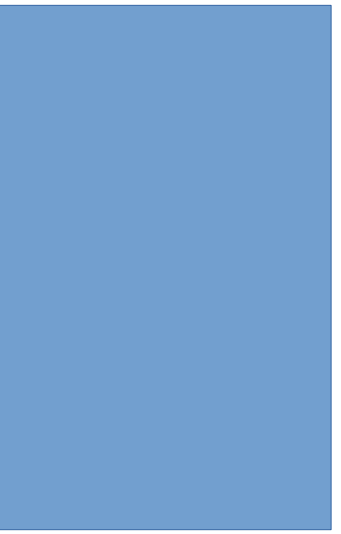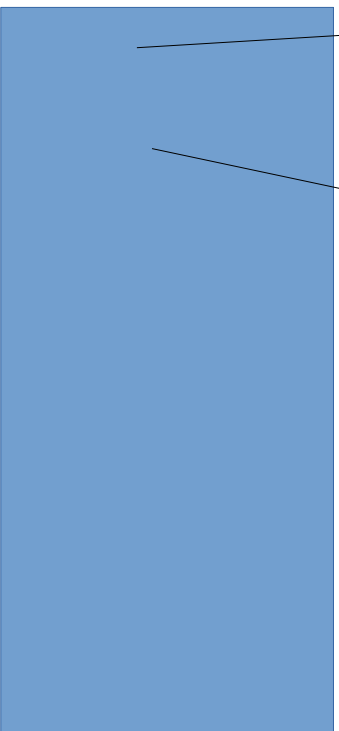G   Global page
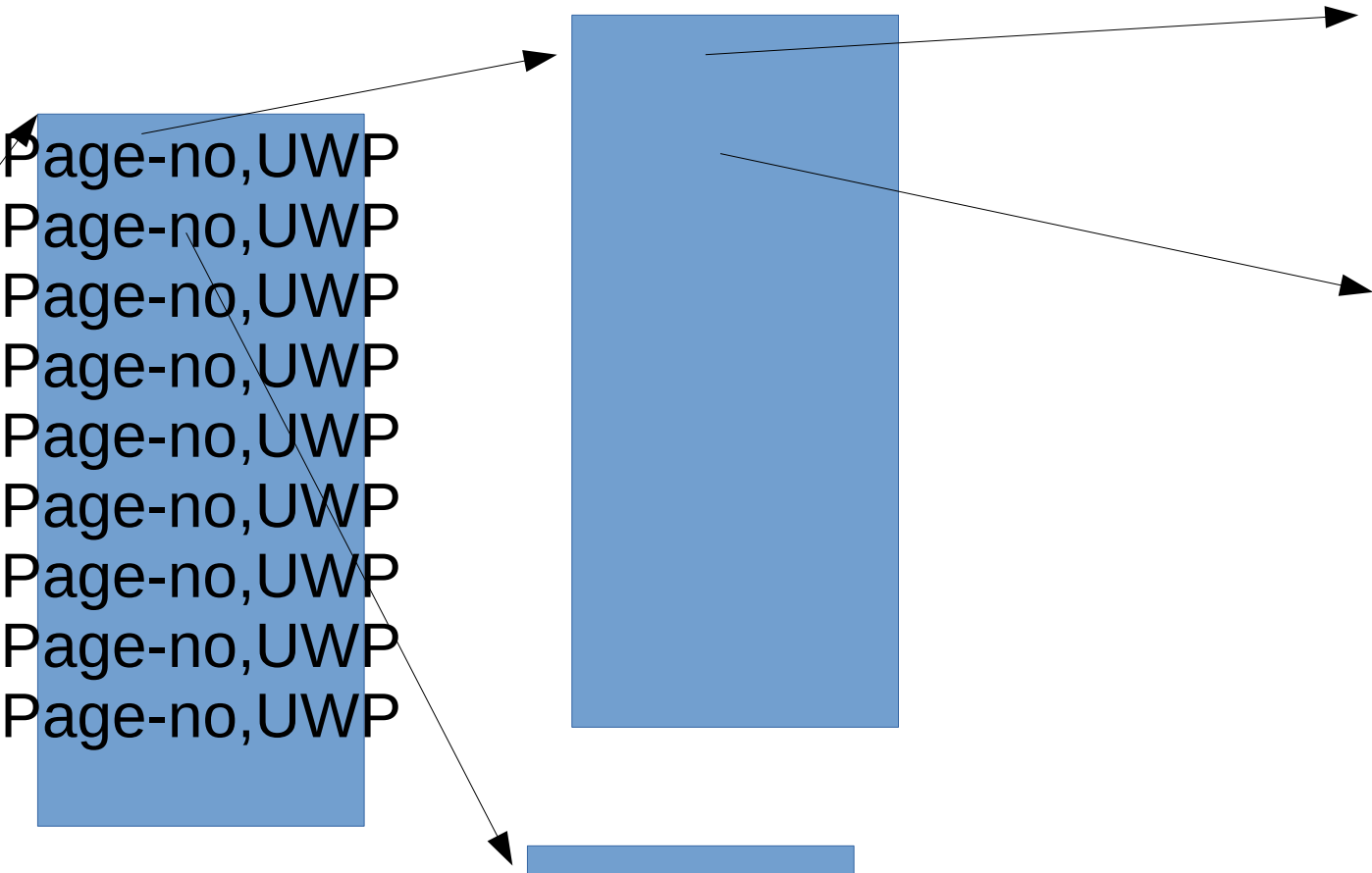AVL Available for system use

Before mappages()
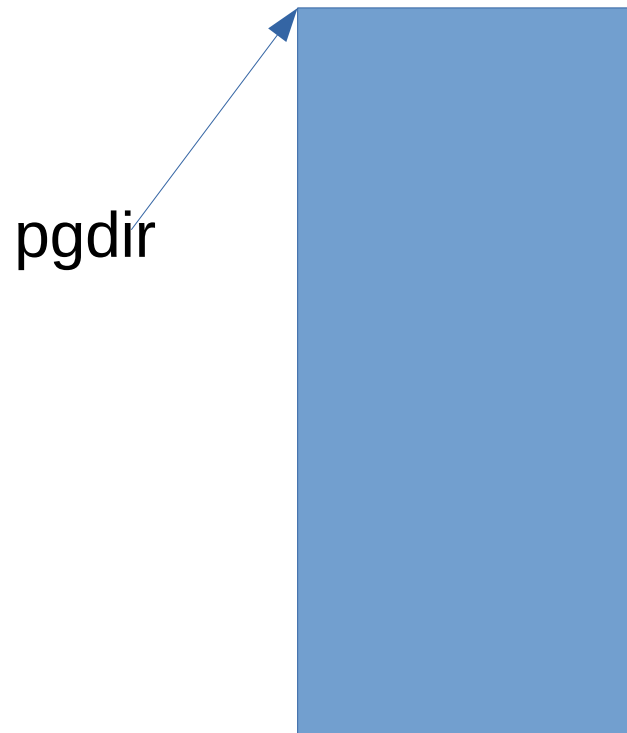
pgdir

pgdir

Page-no,UWP
Page-no,UWP
Page-no,UWP
Page-no,UWP
Page-no,UWP
Page-no,UWP
Page-no,UWP
Page-no,UWP
Page-no,UWP

4K

4K

**What mappages()
Does**

After kvmalloc() in main()

Logical Address = offset

Linear Address

RAM

Physical Addr

Dir | pg | Offset

4MB

0

CS, SS, etc.

Selector

Base	Limit	Permissions
3		
2 0	4GB	Write
1 0	4GB	Read, Execute
0 0	0	0

GDT

DS

SS

CS

Page Table

CR3

kpgdir

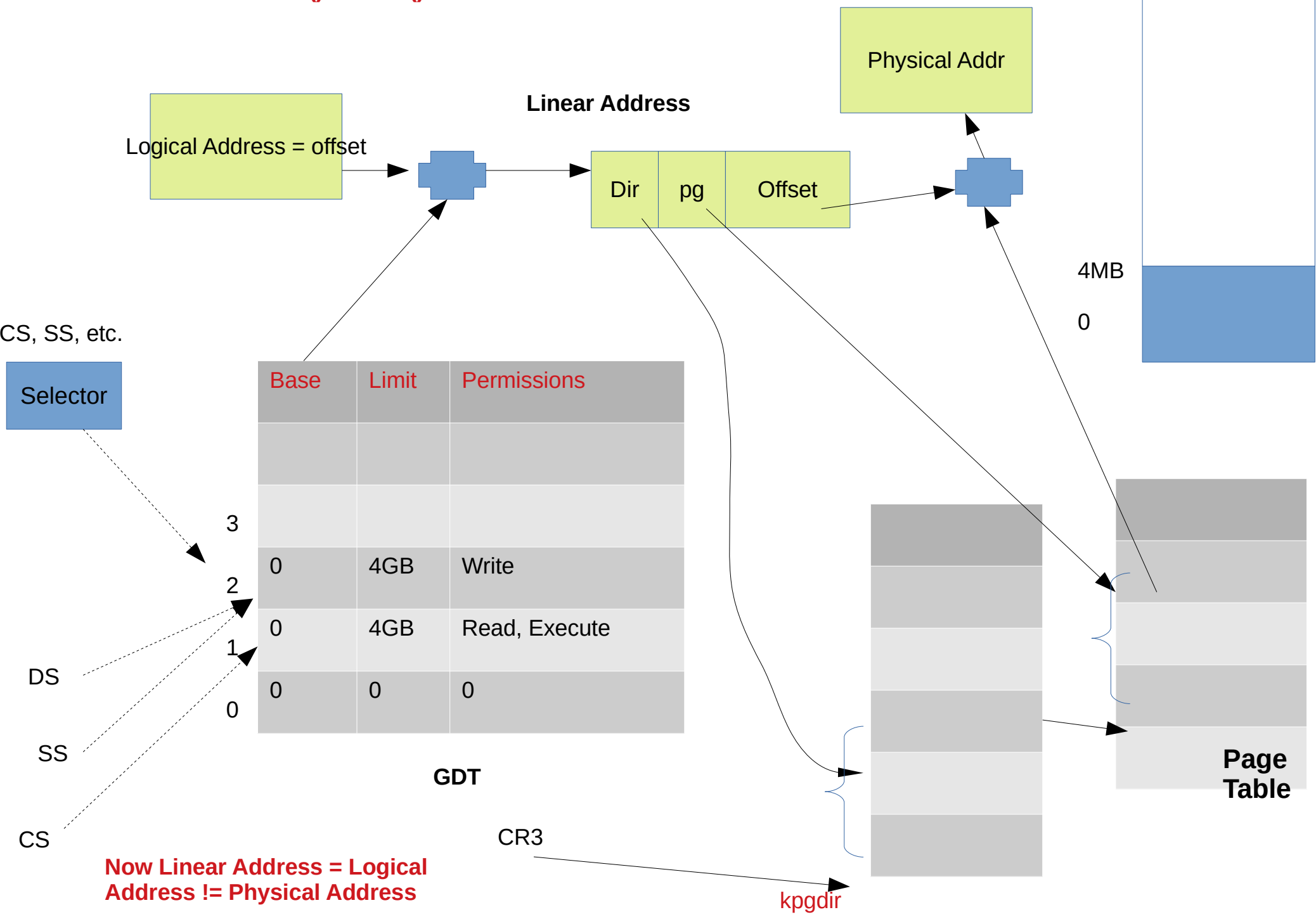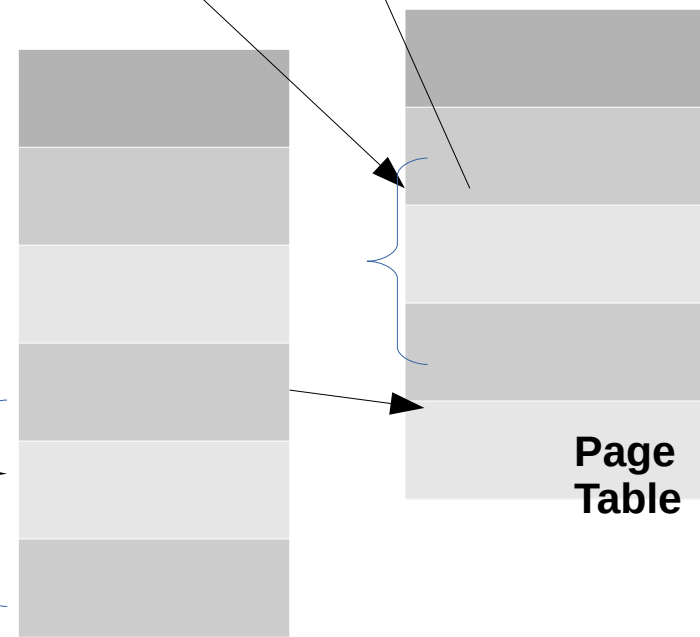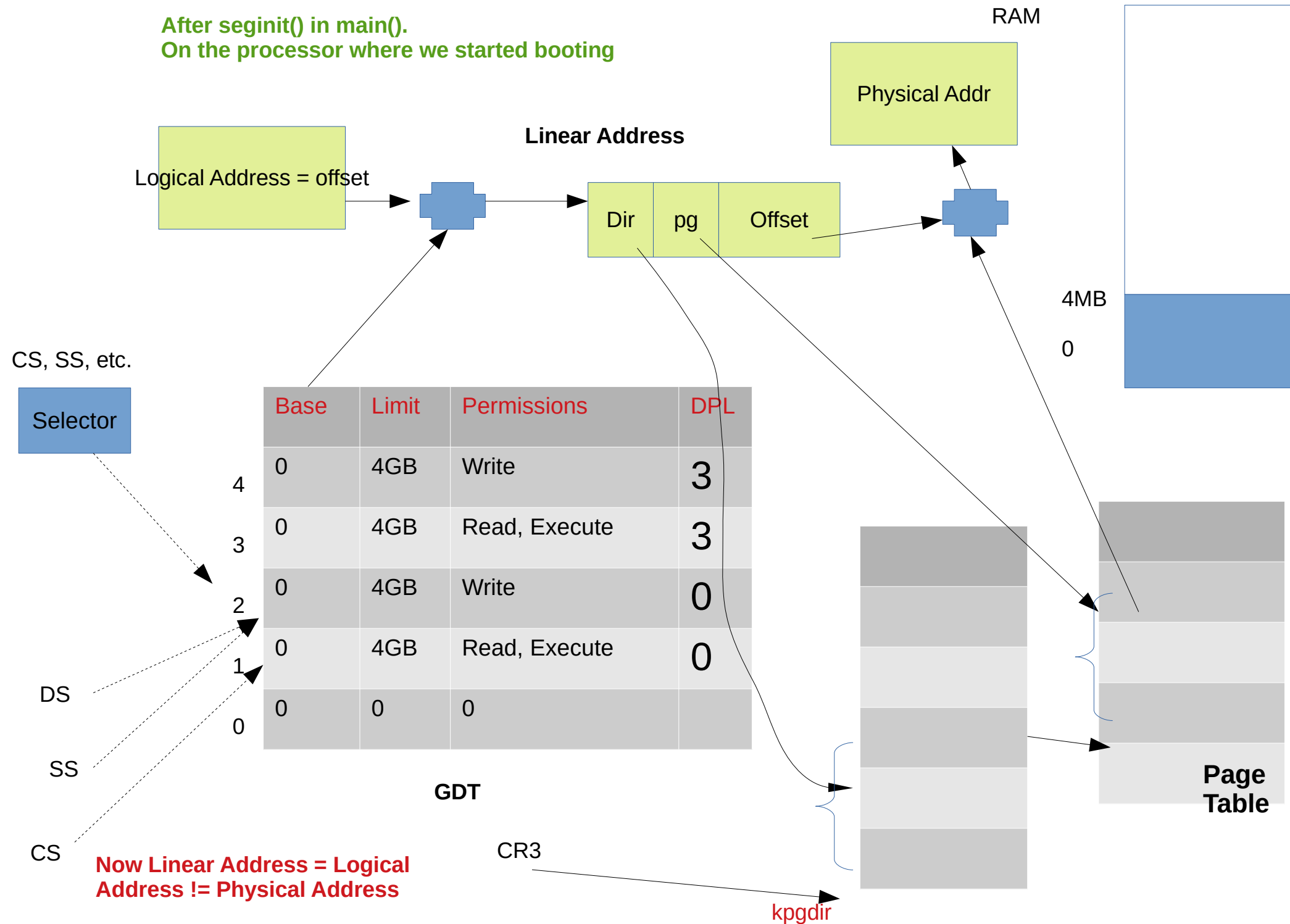**Now Linear Address = Logical Address != Physical Address**

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

Logical Address = offset

Linear Address

Physical Addr

RAM

Dir | pg | Offset

CS, SS, etc.

Selector

Base	Limit	Permissions	DPL
0	4GB	Write	3
0	4GB	Read, Execute	3
0	4GB	Write	0
0	4GB	Read, Execute	0
0	0	0	

4
3
2
1
0

DS

SS

CS

GDT

CR3

4MB

0

Page
Table

kpgdir

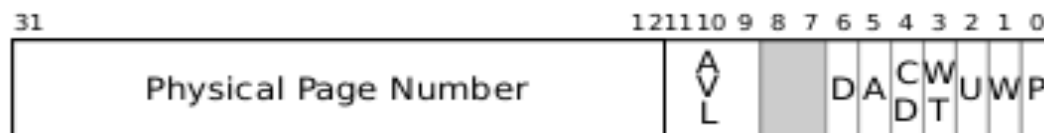**Now Linear Address = Logical**
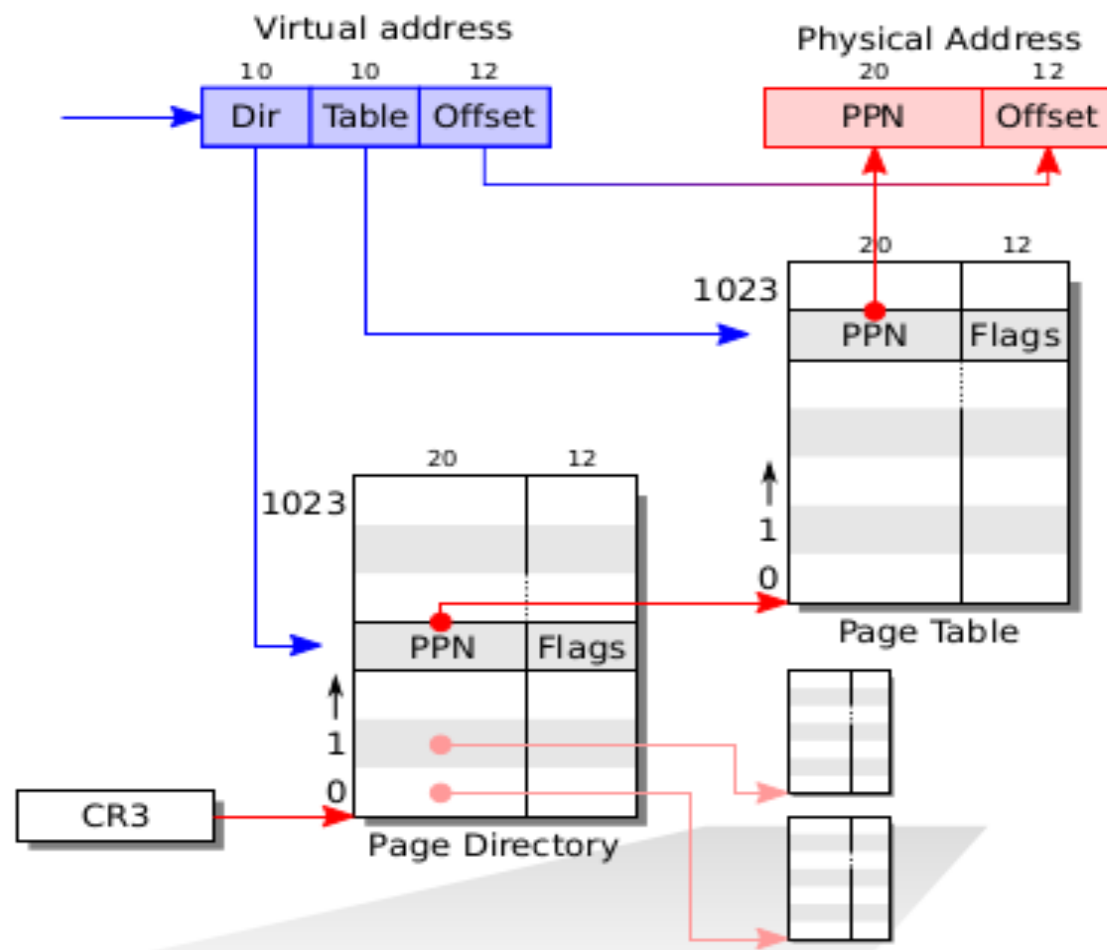**Address != Physical Address**

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**

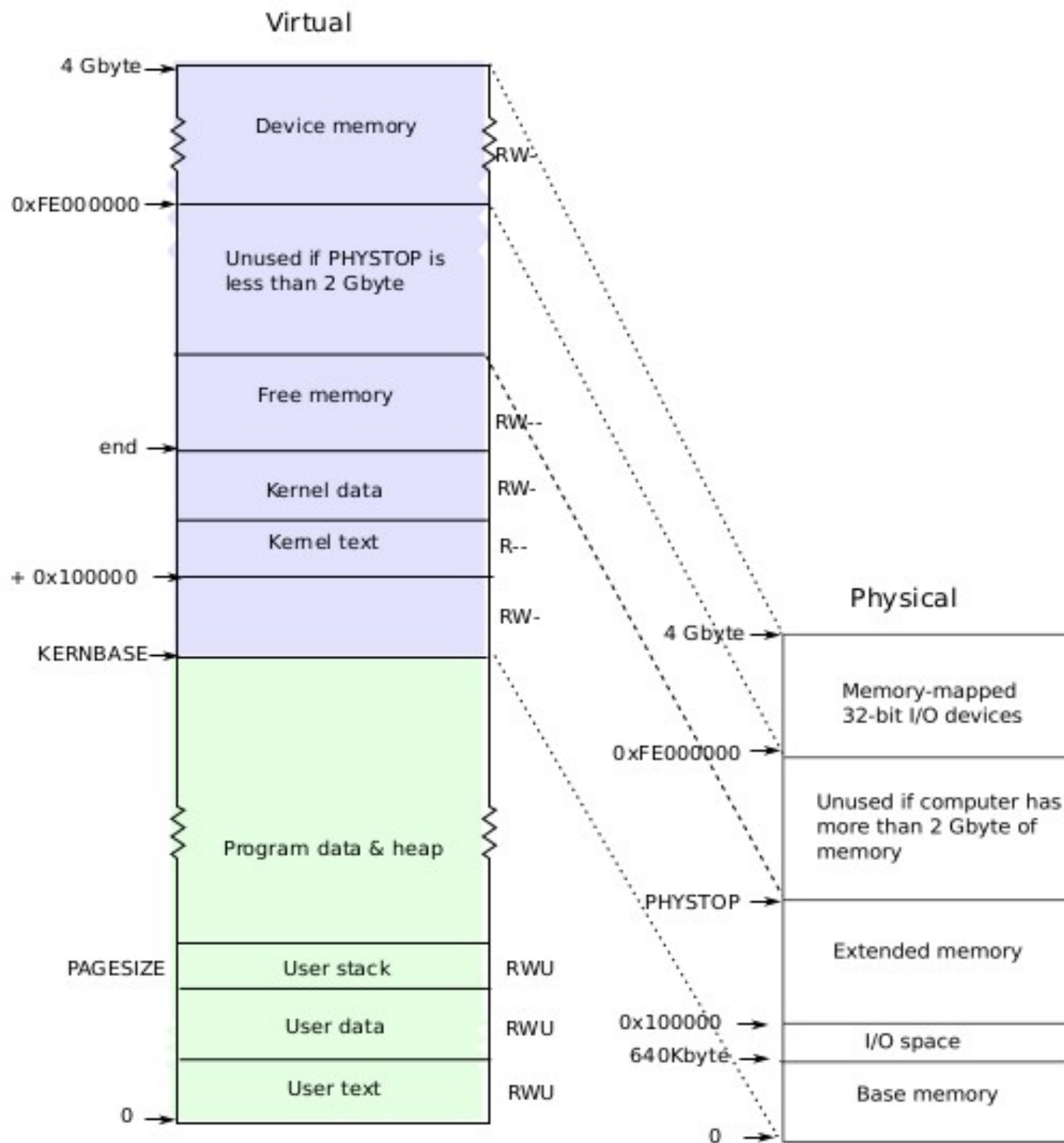- **This happens automatically as part of "trap" handling (covered separately)**
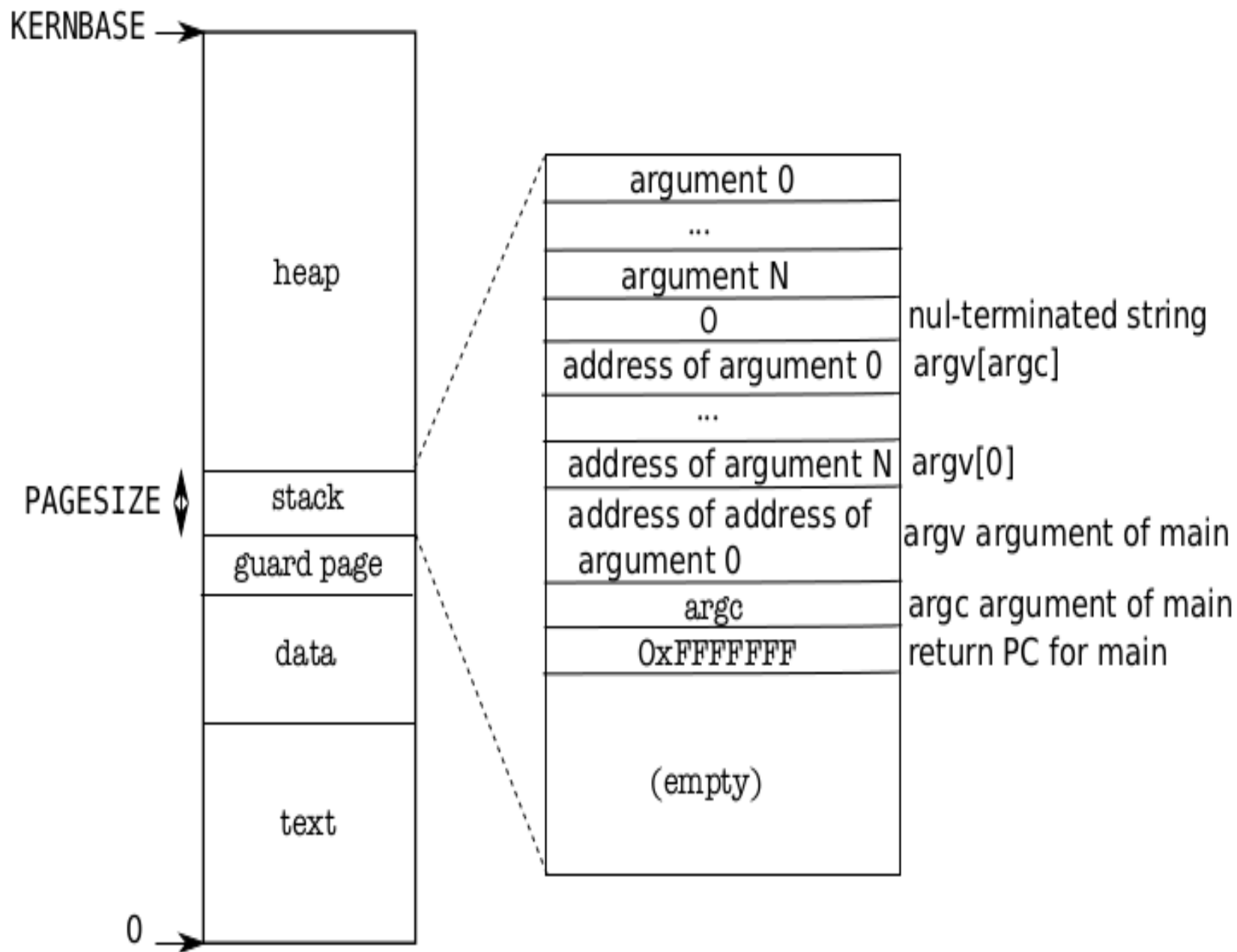
Memory Management

X86 page table hardware

Virtual address

10	10	12
Dir	Table	Offset

Physical Address

20	12
PPN	Offset

1023

20	12
PPN	Flags

Page Table

1023

20	12
PPN	Flags

CR3

Page Directory

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

31	1211	10	9	8	7	6	5	4	3	2	1	0
Physical Page Number	A V L				D	A	C D	W T	U	W	P

P - Present
W - Writable
U - User
WT - 1=Write-through, 0=Write-back
CD - Cache Disabled
A - Accessed
D - Dirty (0 in page directory)
AVL - Available for system use

Layout of process's VA space

KERNBASE →

heap

PAGESIZE ↕   stack

guard page

data

text

0 →

argument 0

...

argument N

0   nul-terminated string

address of argument 0   argv[argc]

...

address of argument N   argv[0]

address of address of
argument 0   argv argument of main

argc   argc argument of main

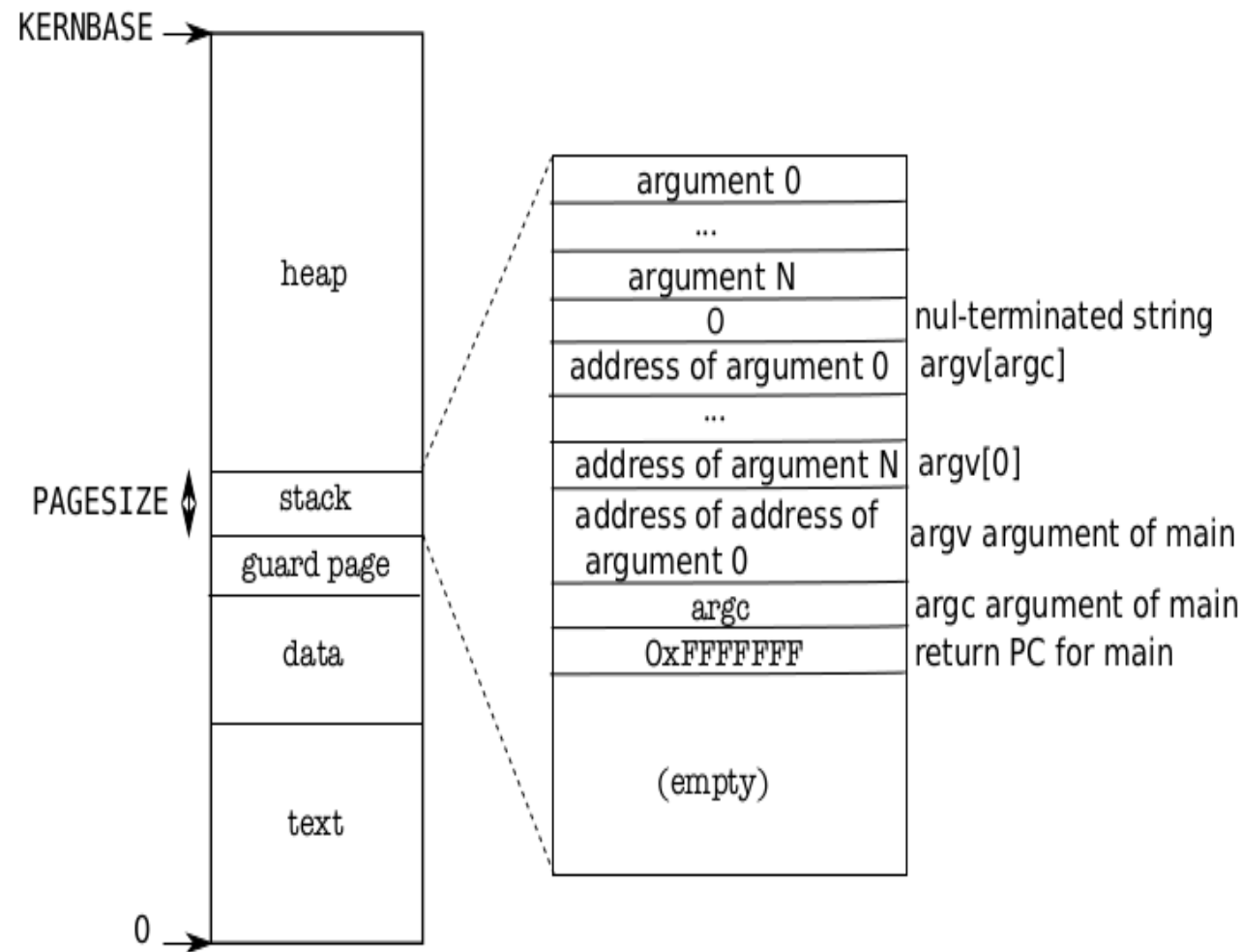0xFFFFFFF   return PC for main

(empty)
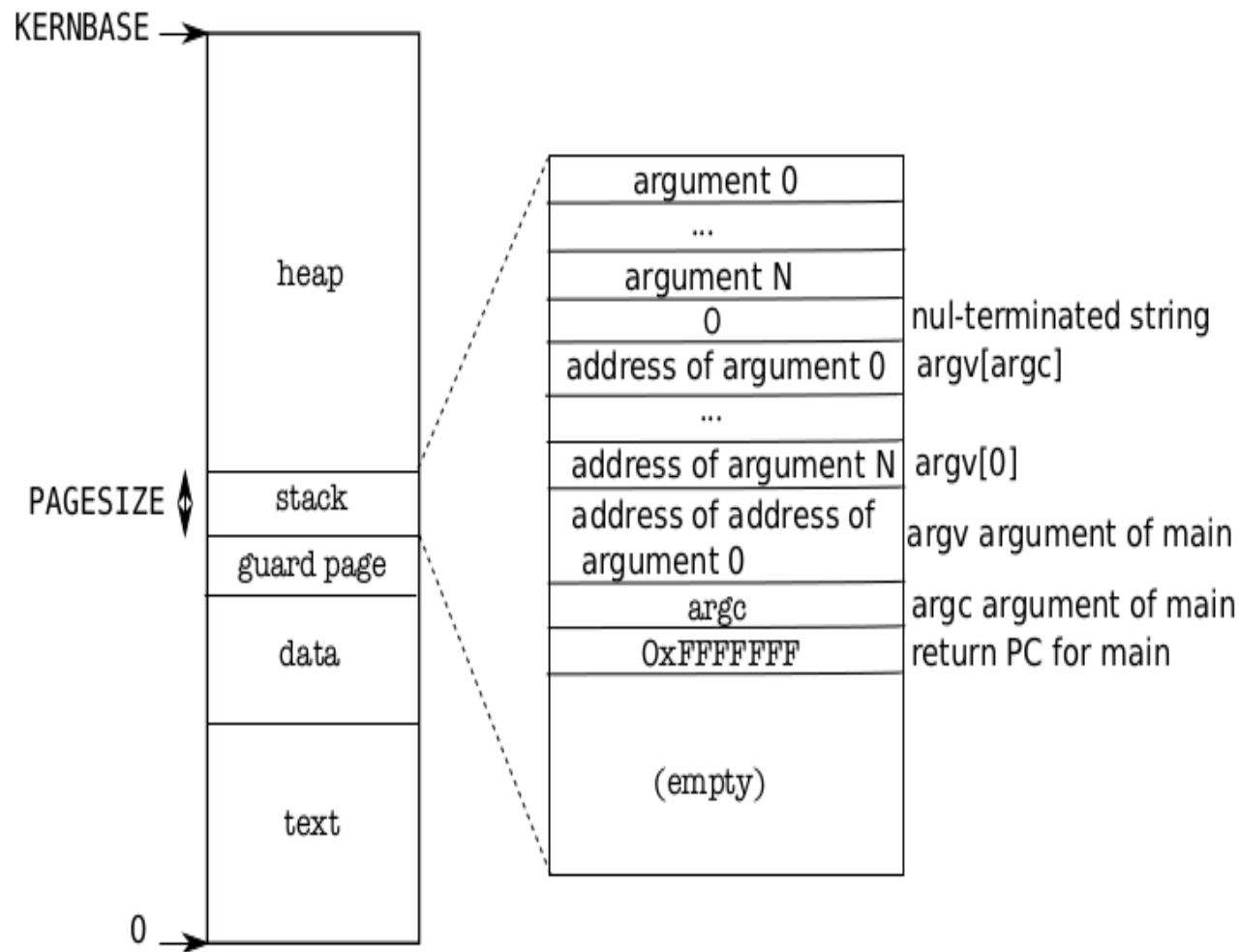
**Memory
Layout
of a
user
process**

**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

KERNBASE →

heap

PAGESIZE ↕ stack

guard page

data

text

0 →

argument 0	
...	
argument N	
0	nul-terminated string
address of argument 0	argv[argc]
...	
address of argument N	argv[0]
address of address of argument 0	argv argument of main
argc	argc argument of main
0xFFFFFFFF	return PC for main
(empty)	

Memory Layout of a user process

KERNBASE →

heap

PAGESIZE ↕ stack

guard page

data

text

0 →

argument 0	
...	
argument N	
0	nul-terminated string
address of argument 0	argv[argc]
...	
address of argument N	argv[0]
address of address of argument 0	argv argument of main
argc	argc argument of main
0xFFFFFFFF	return PC for main
(empty)	

lock

uselock

run *freelist

run *next

Allocated
frame

run *next

Allocated
frame

Allocated
frame

run *next

run *next

Allocated
frame

RAM –
divided
into frames
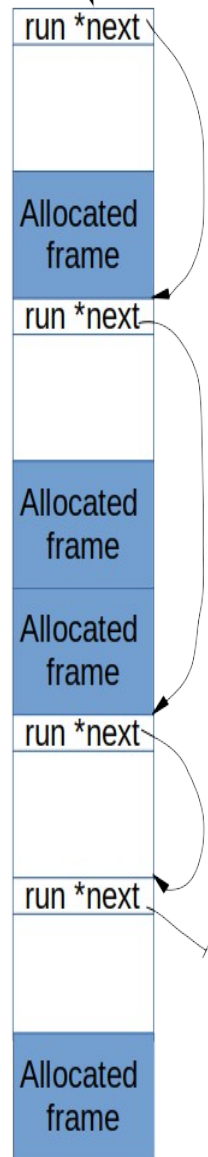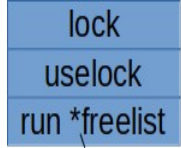
Actually like
this in memory

**Free List in
XV6**

lock

uselock

run *freelist

kmem

**Seen
independent
ly**

run *

run *

run *
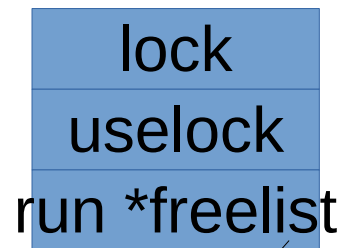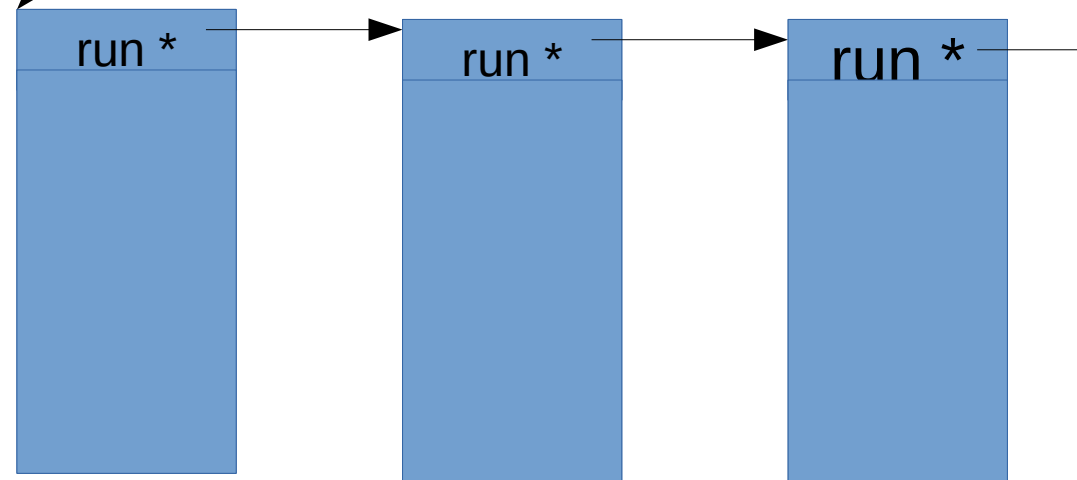
exec()

- **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)