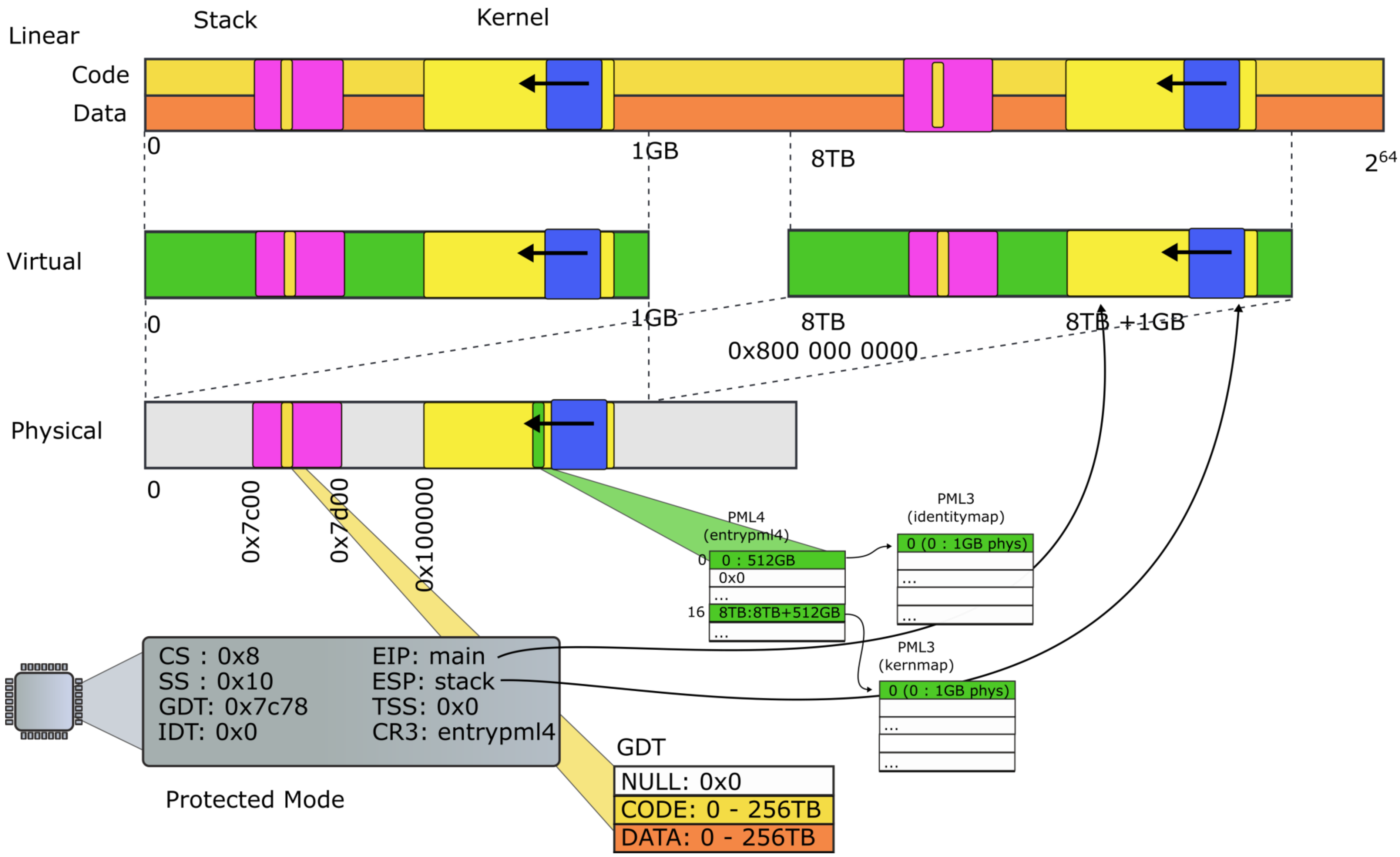
cs5460/6460: Operating Systems

Lecture 08: System Init (Kernel Memory Allocator and Page Table)

Anton Burtsev

February, 2026

State of the system after boot



CS : 0x8
SS : 0x10
GDT: 0x7c78
IDT: 0x0

EIP: main
ESP: stack
TSS: 0x0
CR3: entrypml4

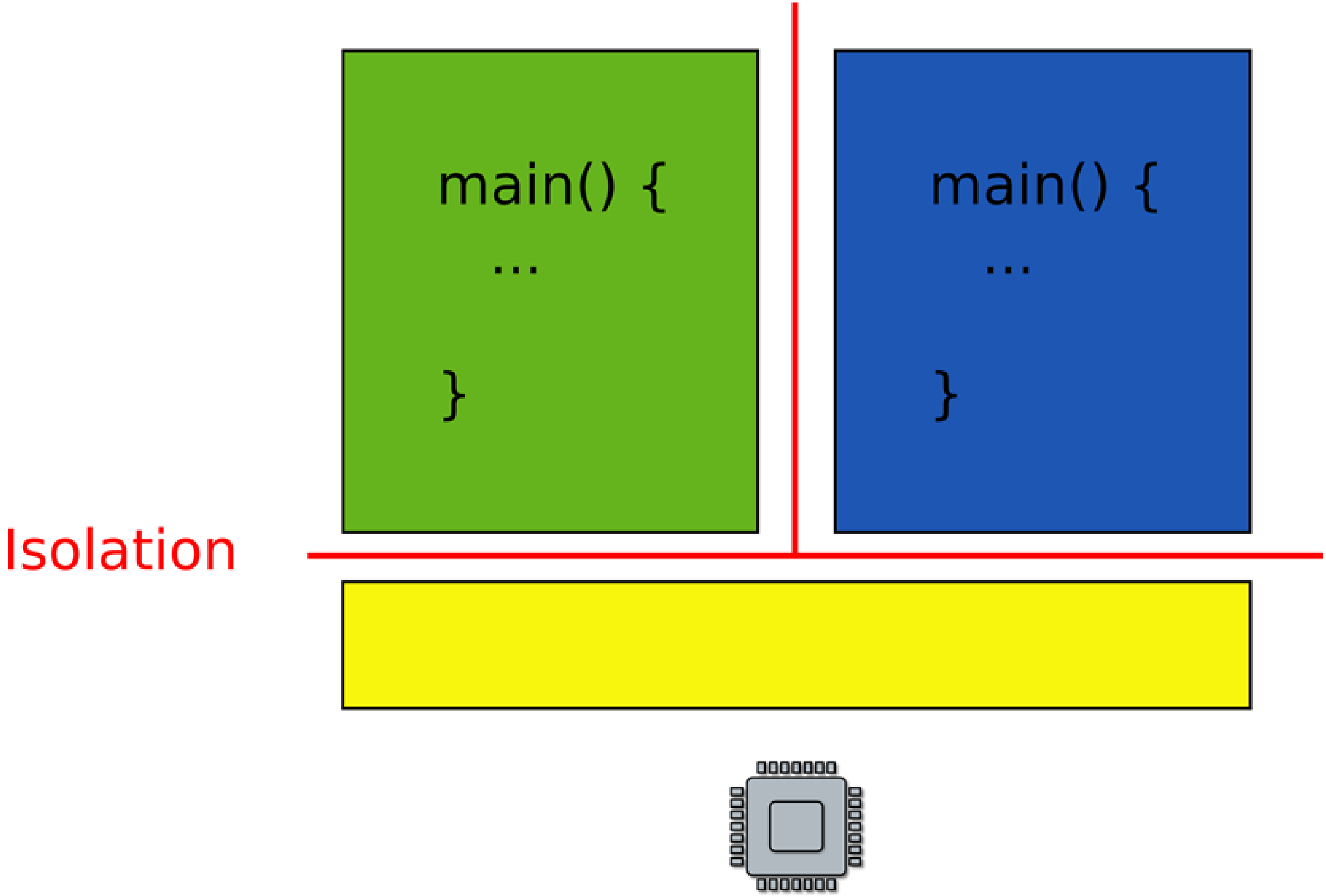
NULL: 0x0
CODE: 0 - 256TB
DATA: 0 - 256TB

Running in main()

```
// Bootstrap processor starts running C code here.
// Allocate a real stack and switch to it, first
// doing some setup required for memory allocator to work.
int
main(void)
{
    kinit1(end, P2V(4*1024*1024)); // phys page allocator
    kvmalloc(); // kernel page table
    mpinit(); // detect other processors
    lapicinit(); // interrupt controller
    seginit(); // segment descriptors
    cprintf("\ncpu%d: starting xv6\n\n", cpunum());
...
}
```

What's next?

We want to run multiple processes



```
main() {
   ...

}
```

```
main() {
   ...

}
```

Isolation

But what is a process?

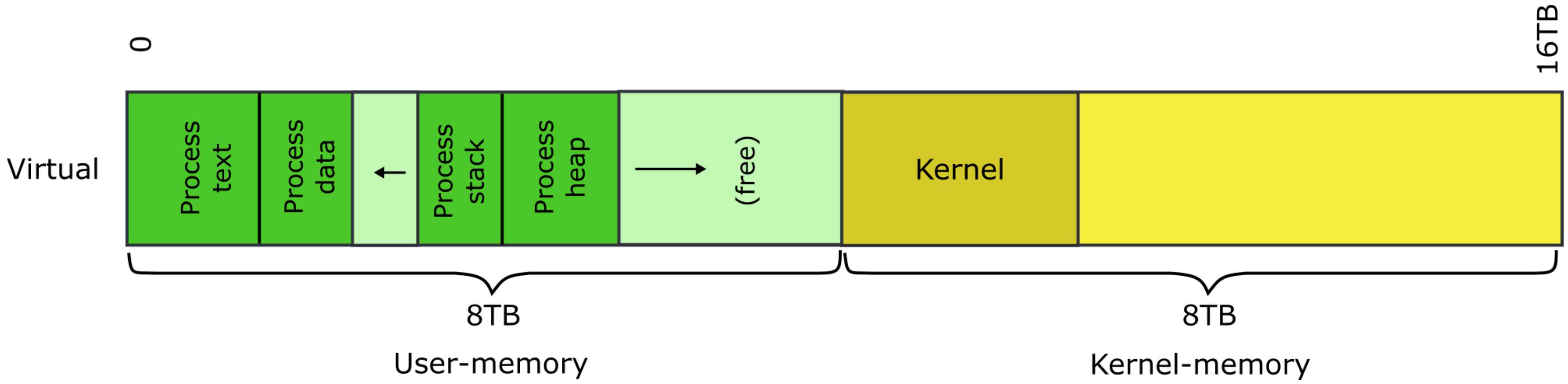
A couple of requirements

- Each process is a collection of resources
- Memory
  - E.g., text, stack, heap
- In-kernel state
  - E.g., open file descriptors, network sockets (connections)

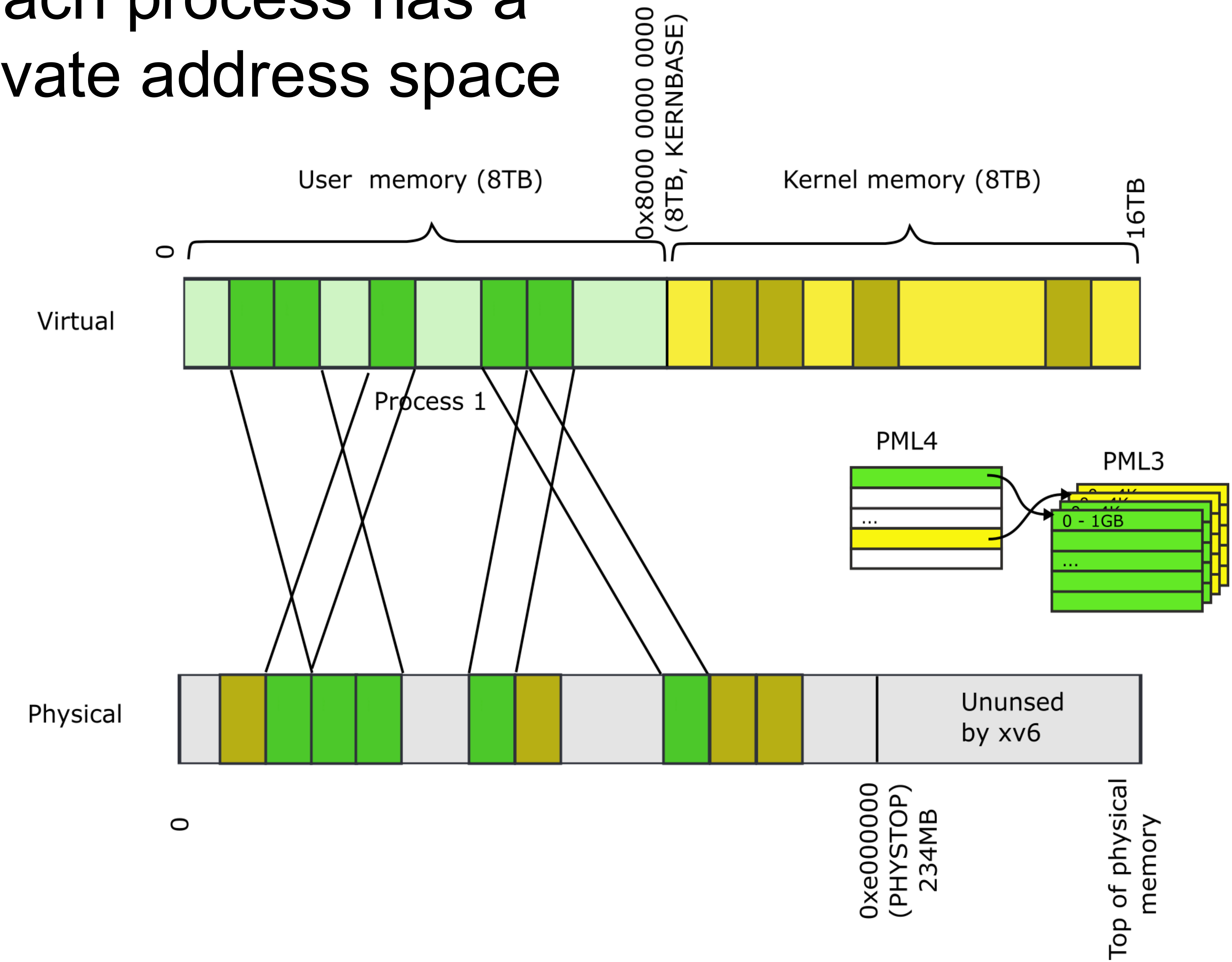
A couple of requirements

- Each process is a collection of resources
- Memory
  - E.g., text, stack, heap
- In-kernel state
  - E.g., open file descriptors, network sockets (connections)
- Processes are isolated from each other
- Processes don't trust each other
  - Individual users, some privileged
- Can't interfere with other processes
- Can't change kernel (to affect other processes)

Each process will have 8TB private address space

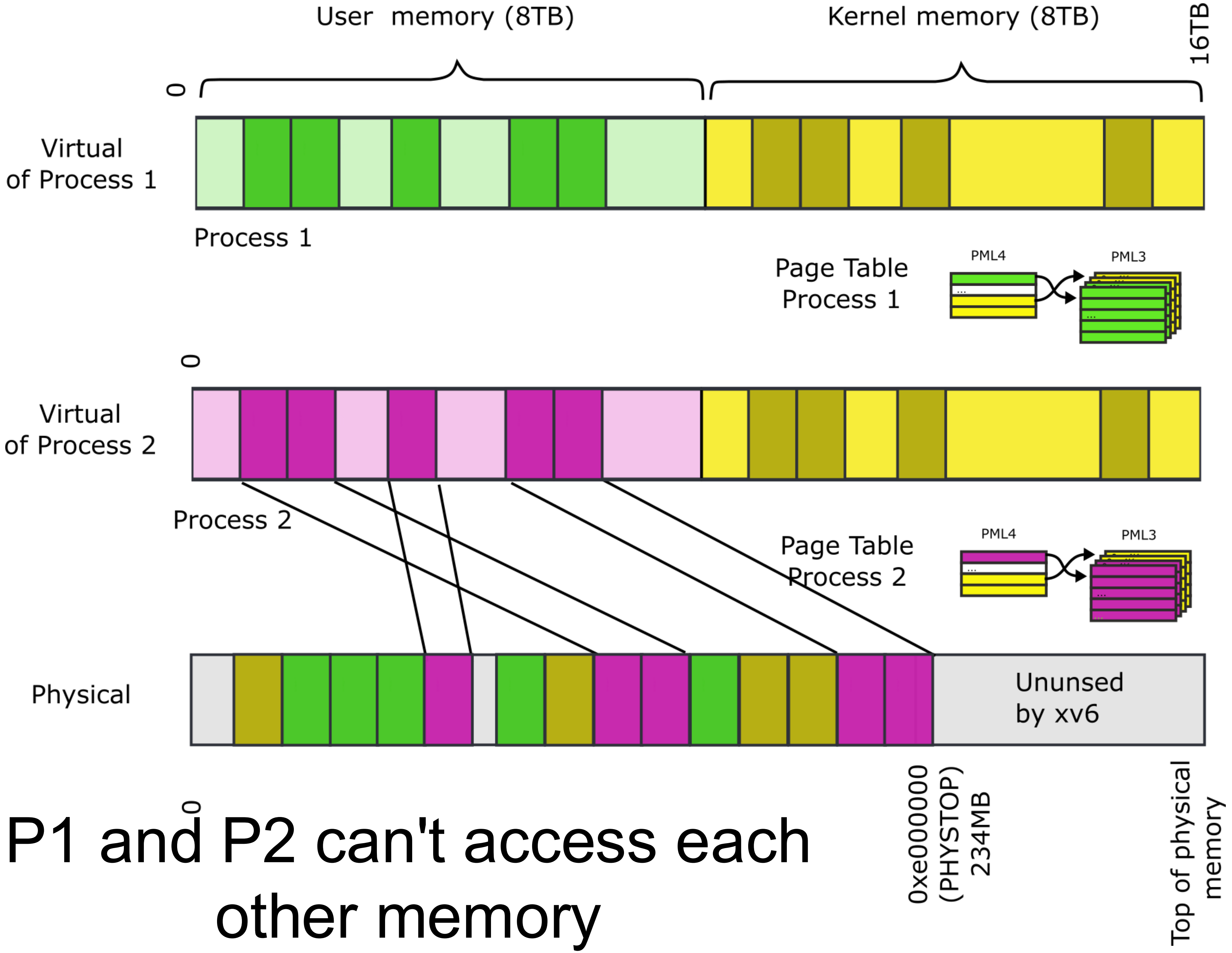


Each process has a private address space



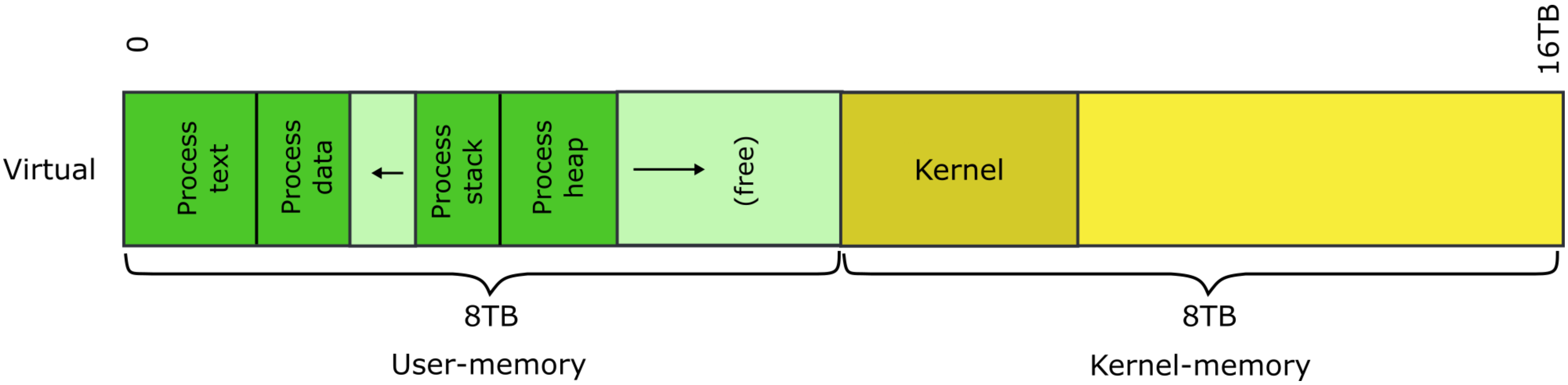
Each process maps the kernel

- It's not strictly required
- But convenient for system calls
- No need to change the page table when process enters the kernel with a system call
- **Things are much faster!**

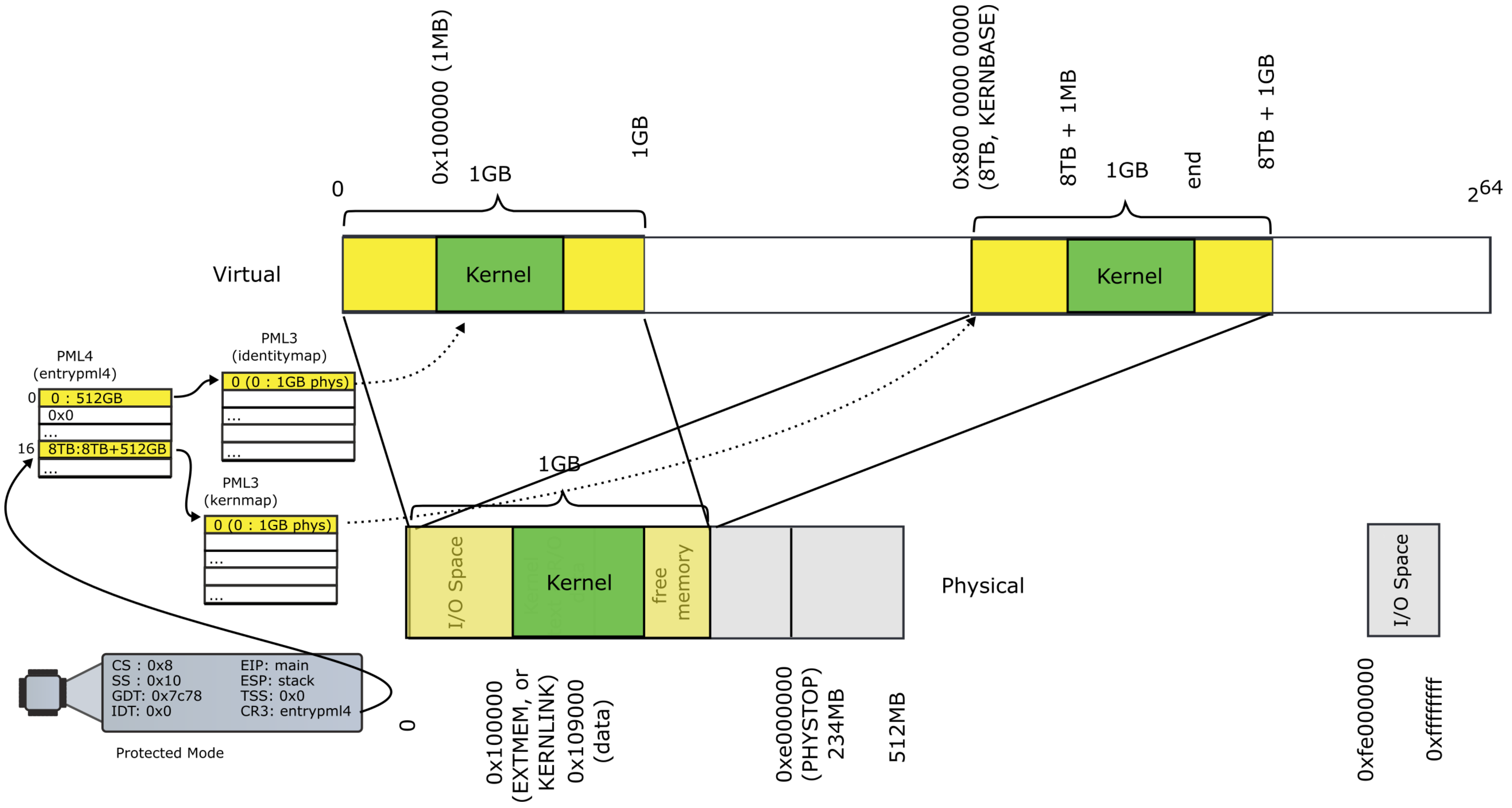


P1 and P2 can't access each other memory

Our goal: split address space



Memory after boot



CS : 0x8
SS : 0x10
GDT: 0x7c78
IDT: 0x0
EIP: main
ESP: stack
TSS: 0x0
CR3: entrypml4

Outline

- Create the kernel address space
- Create kernel memory allocator
- Allocate memory for page tables
  - Page table directory and page table

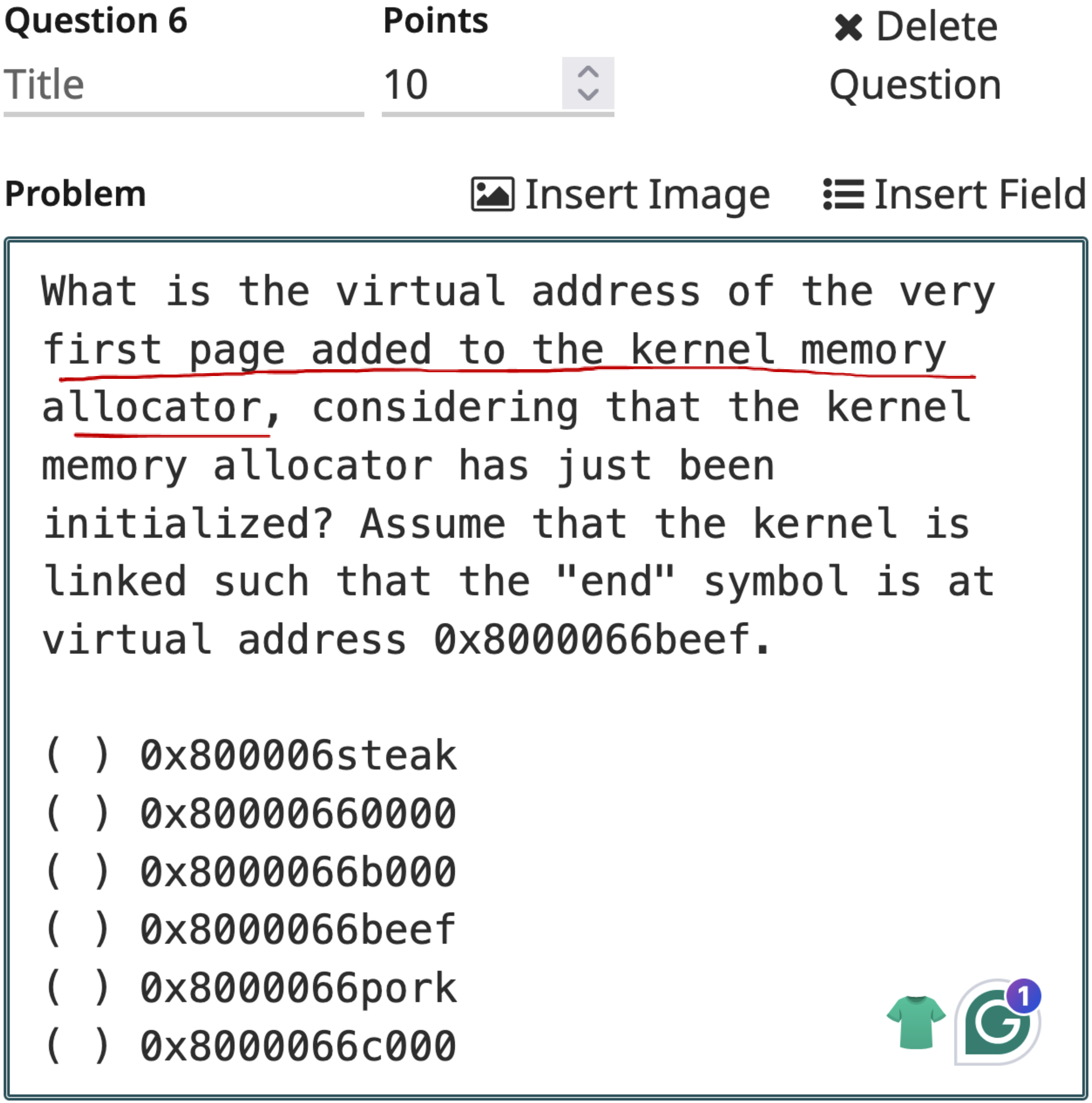
Kernel memory allocator

- Kernel needs normal 4-level, 4KB page table
- Right now we have:
  - One (statically allocated) page table
  - That has only two 1GB entries
- 4KB page table is a better choice
  - Xv6 processes are small
  - Wasting 1GB or 2MB on a program that fits into 1KB is absurd

- But to create page tables we need memory
- Where can it come from?

Simple memory allocator

- Goal:
  - alloc() and free()
  - To allocate page tables, stacks, data structures, etc.



Question 6

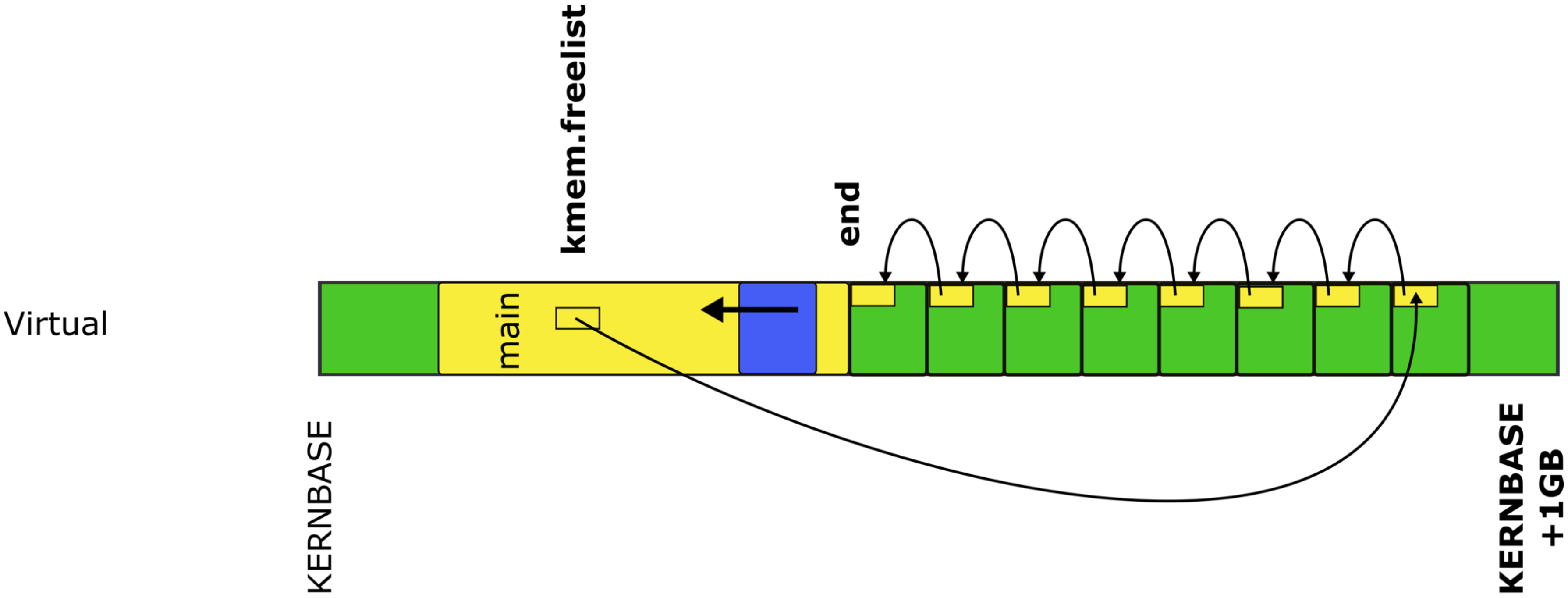
Points 10

Problem

What is the virtual address of the very first page added to the kernel memory allocator, considering that the kernel memory allocator has just been initialized? Assume that the kernel is linked such that the "end" symbol is at virtual address 0x8000066beef.

( ) 0x800006steak
( ) 0x80000660000
( ) 0x8000066b000
( ) 0x8000066beef
( ) 0x8000066pork
( ) 0x8000066c000

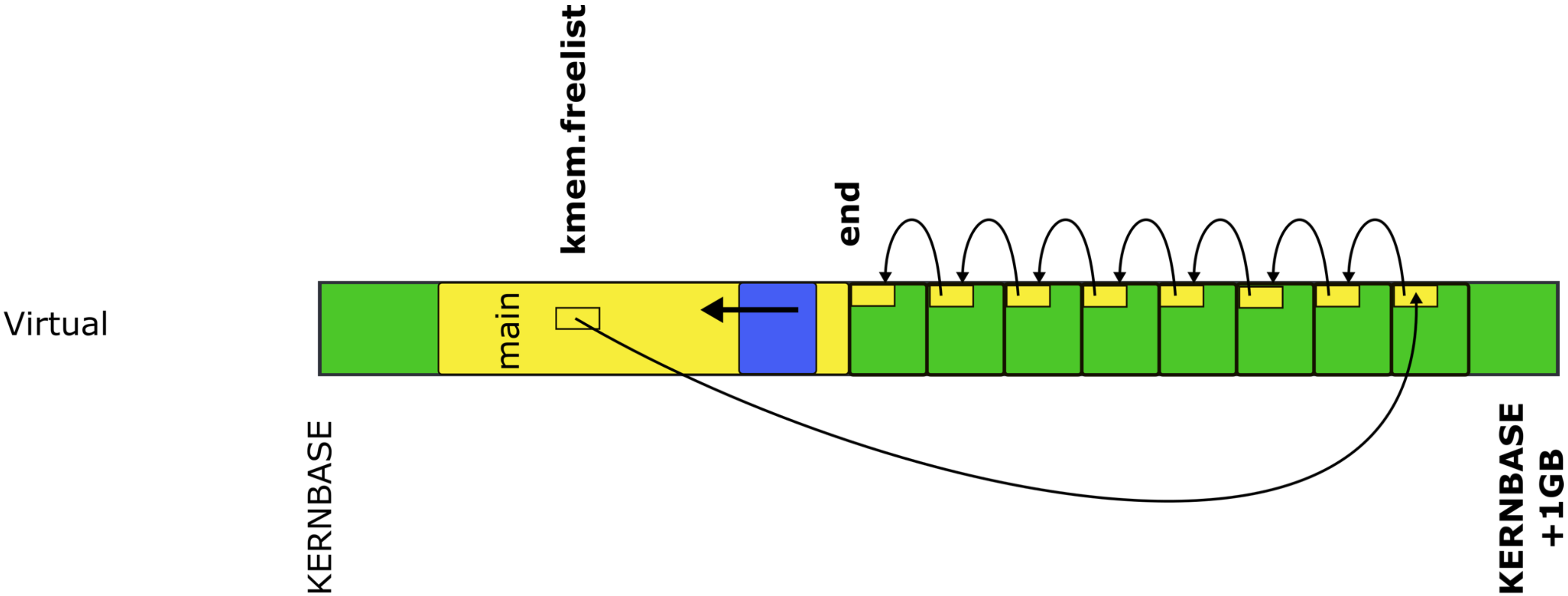
Page allocator



CS : 0x8 EIP: main
SS : 0x10 ESP: stack
GDT: 0x7c78 TSS: 0x0
IDT: 0x0 CR3: entrypml4

Protected Mode

Page allocator



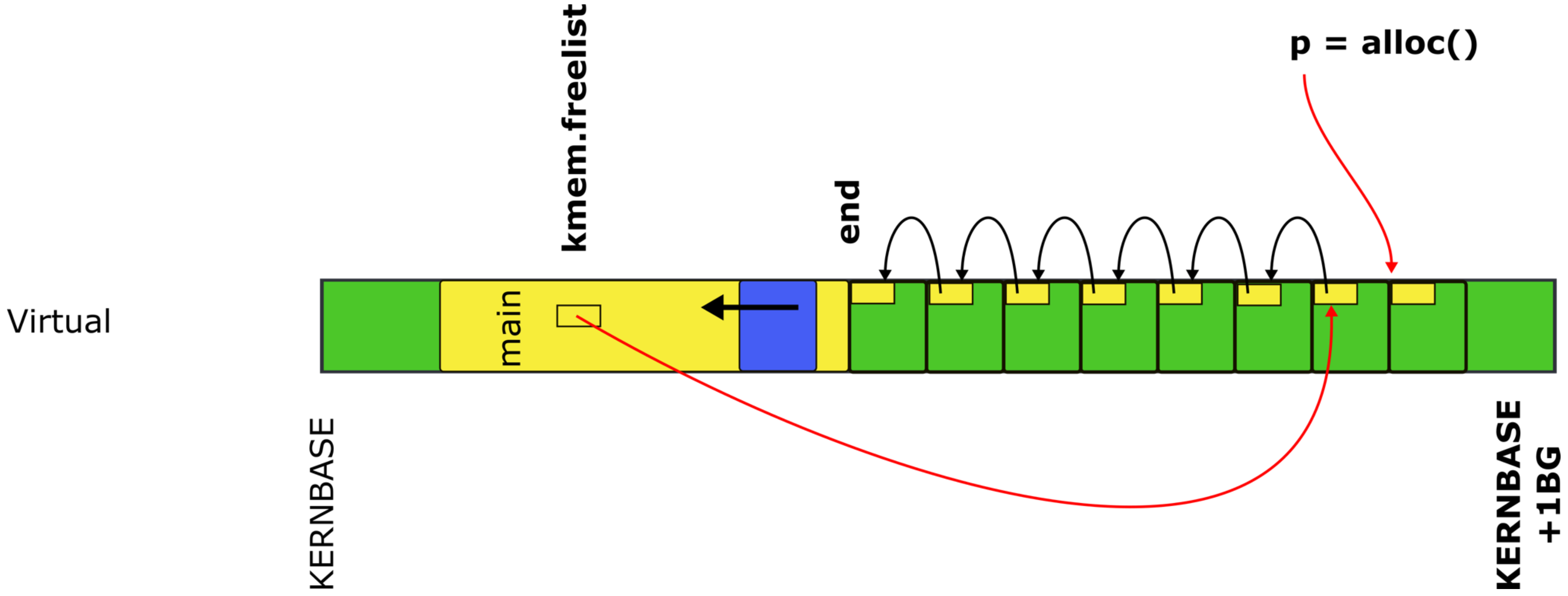
CS : 0x8
SS : 0x10
GDT: 0x7c78
IDT: 0x0

EIP: main
ESP: stack
TSS: 0x0
CR3: entrypml4

Protected Mode

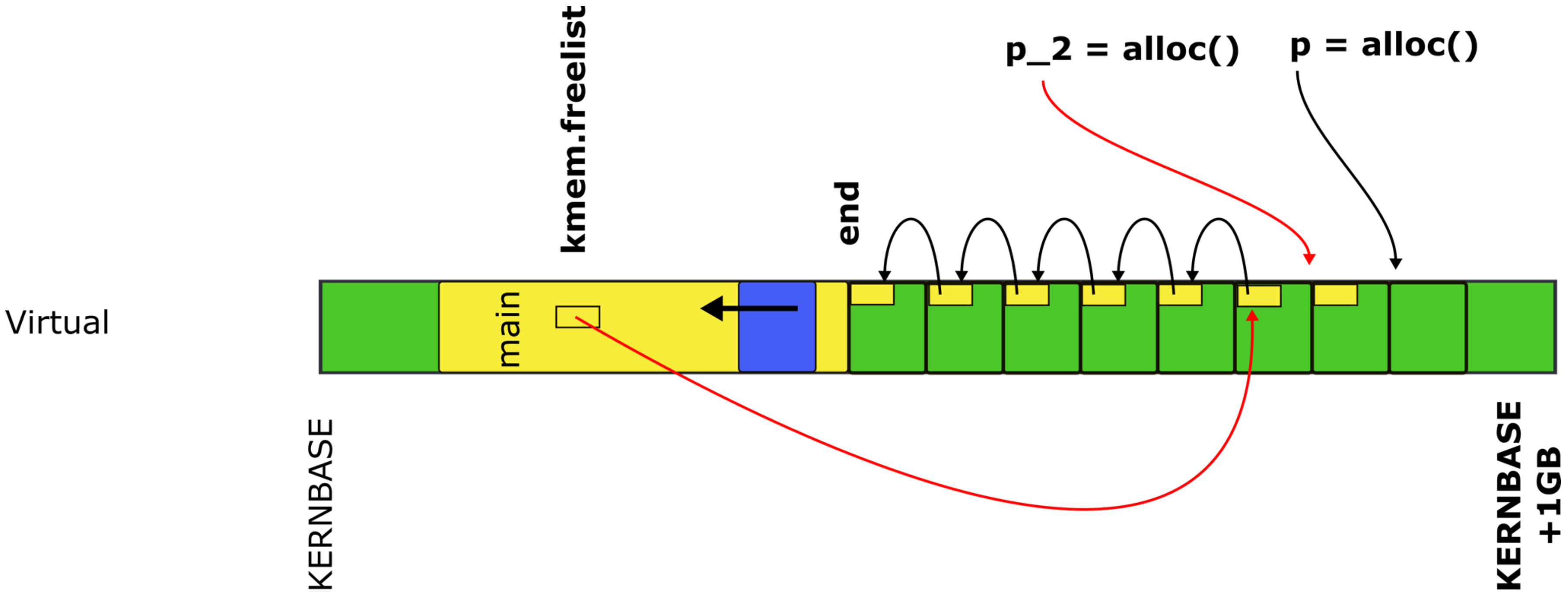
```
struct run {
  struct run *next;
};
```

Page allocator



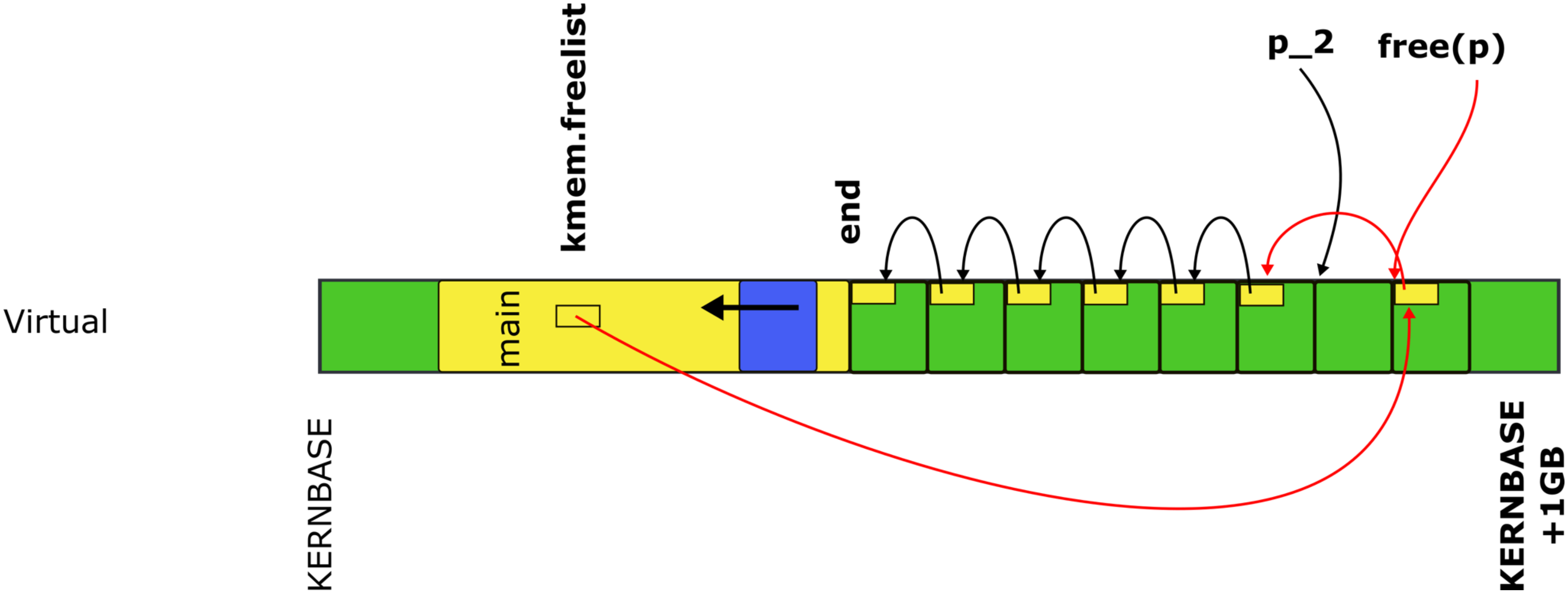
```
3014 struct run {
3015   struct run *next;
3016 };
```

Page allocator



```
3014 struct run {
3015   struct run *next;
3016 };
```

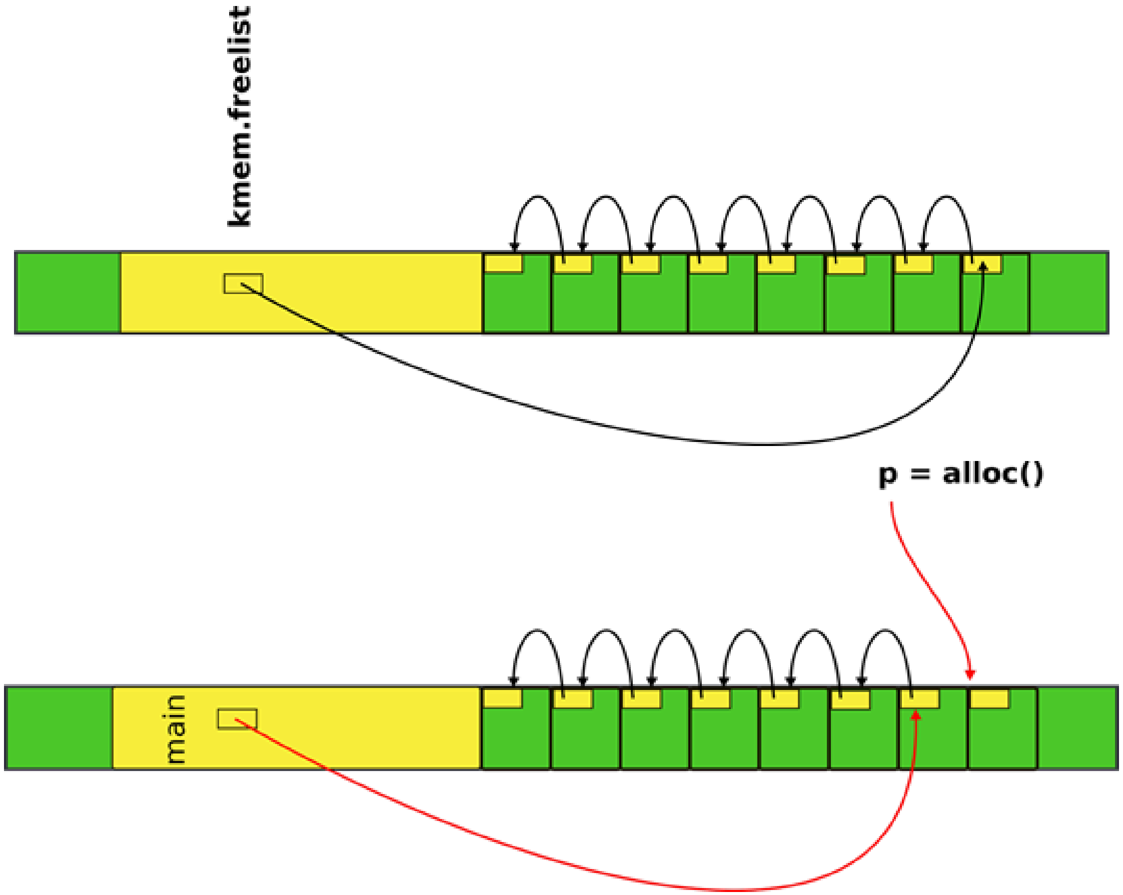
Page allocator

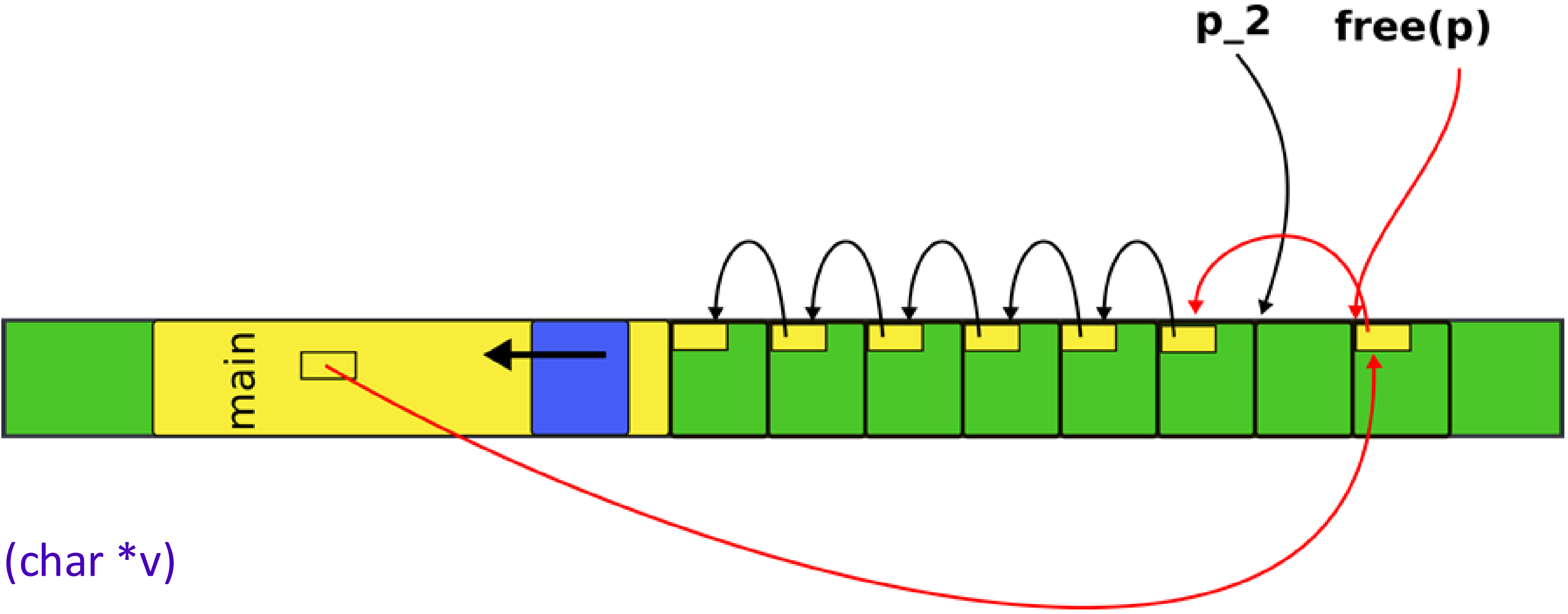


```
3014 struct run {
3015   struct run *next;
3016 };
```

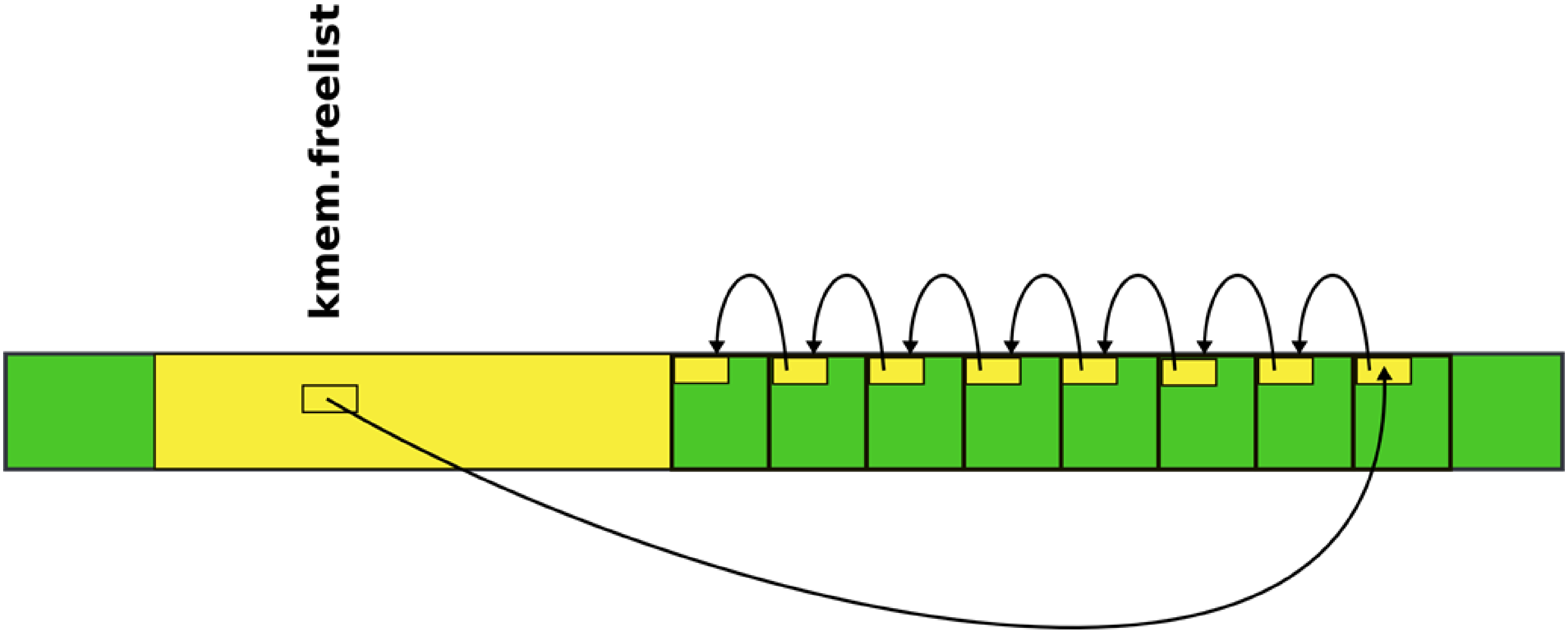
kalloc() - kernel allocator

```
char*
kalloc(void)
{
  struct run *r;
…
  r = kmem.freelist;
  if(r)
    kmem.freelist = r−>next;
…
  return (char*)r;
}
```



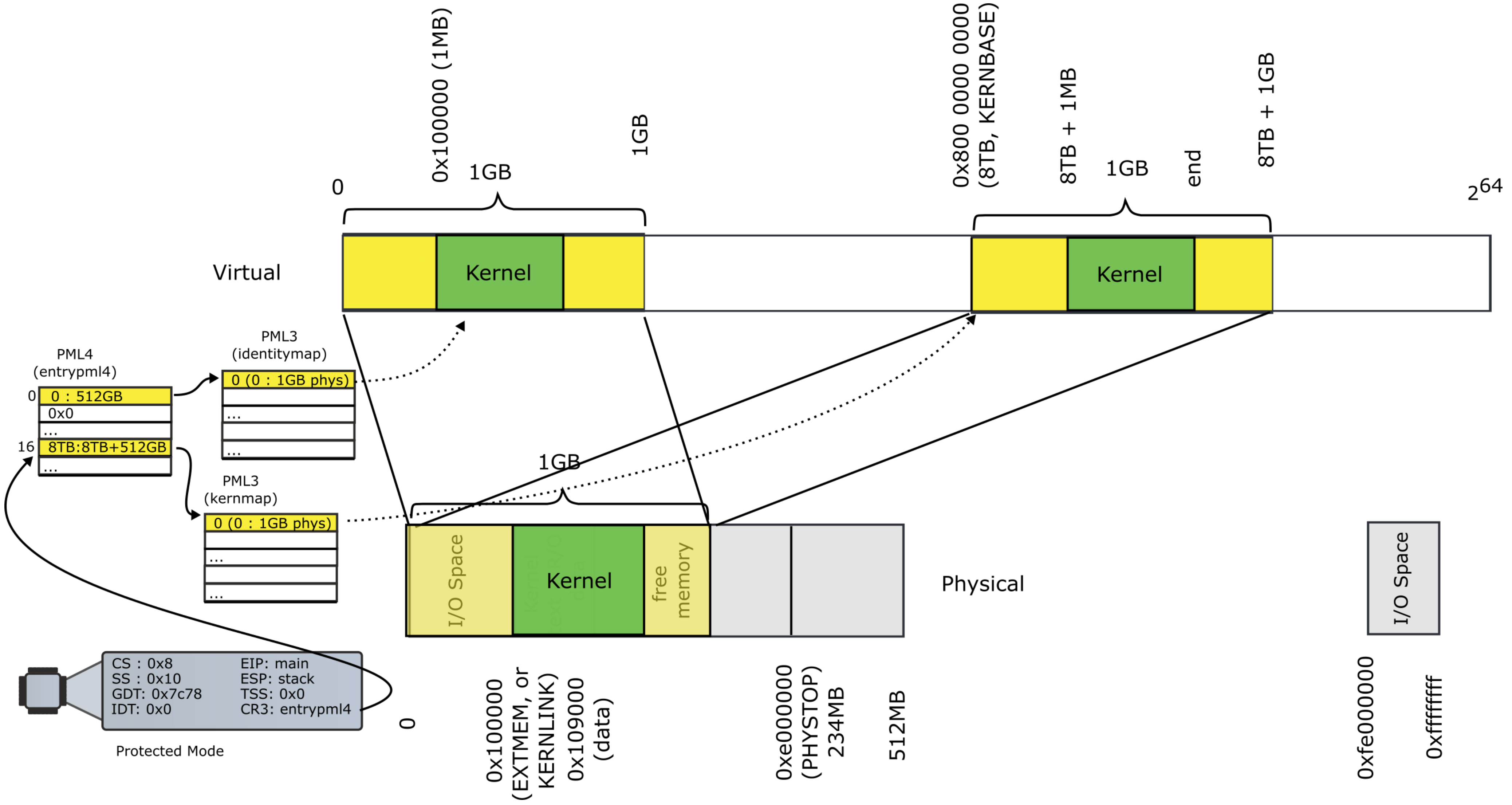


```
kfree(char *v)
{
  struct run *r;
...
  r = (struct run*)v;
  r−>next = kmem.freelist;
  kmem.freelist = r;
...
}
```



- Where can we get memory to keep the list itself?
- Note, the list is maintained within each page
- It has to write each page though to update the "next" pointer

There is free memory in the 1GB page we've mapped



0

0x100000 (1MB)

1GB

1GB

0x800 0000 0000 (8TB, KERNBASE)

8TB + 1MB

1GB

end

8TB + 1GB

2^{64}

Virtual

Kernel

Kernel

PML4 (entrypml4)

0 | 0 : 512GB

0x0

...

16 | 8TB:8TB+512GB

...

PML3 (identitymap)

0 (0 : 1GB phys)

...

...

PML3 (kernmap)

0 (0 : 1GB phys)

...

...

1GB

I/O Space

Kernel

free memory

Physical

I/O Space

CS : 0x8
SS : 0x10
GDT: 0x7c78
IDT: 0x0

EIP: main
ESP: stack
TSS: 0x0
CR3: entrypml4

Protected Mode

0

0x100000 (EXTMEM, or KERNLINK)

0x109000 (data)

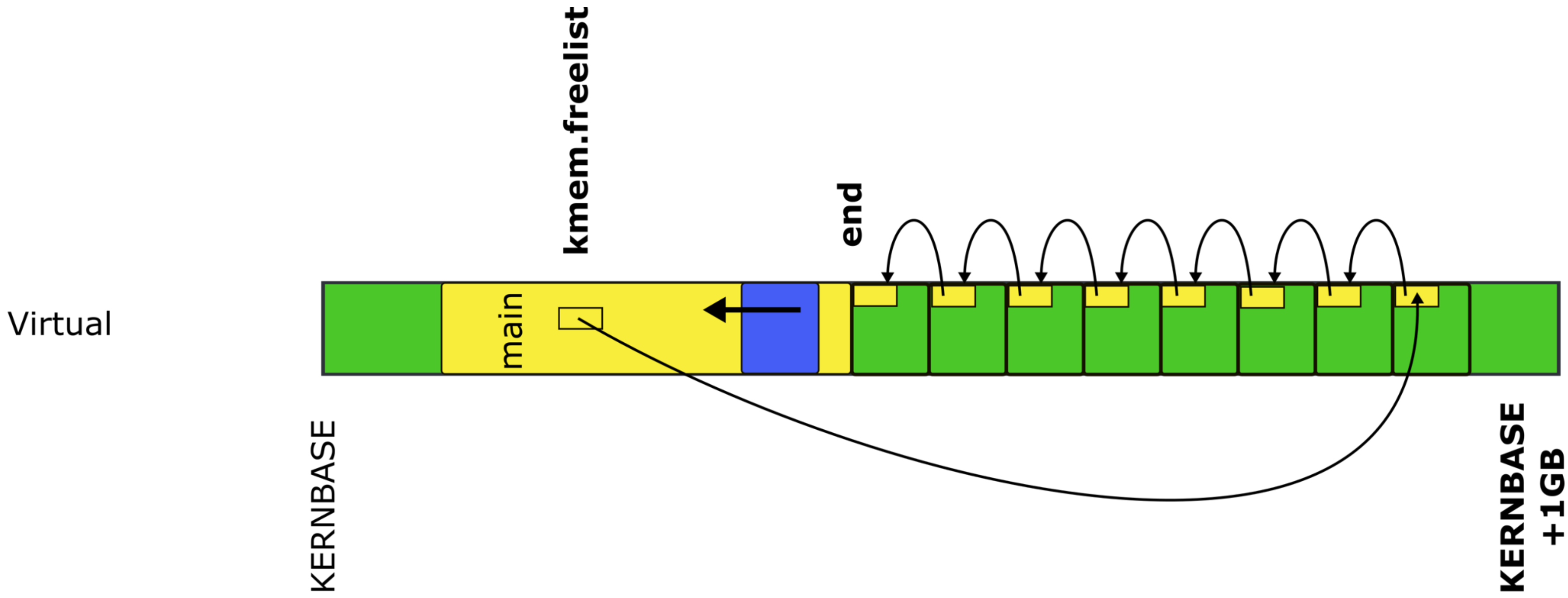
0xe000000 (PHYSTOP) 234MB

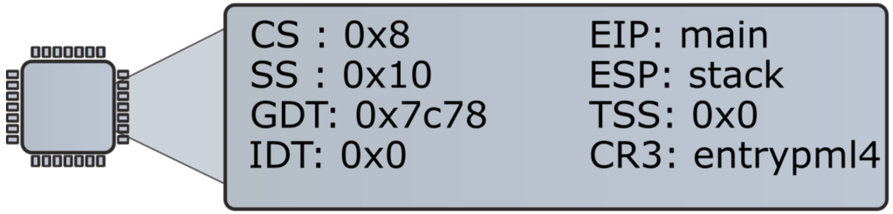
512MB

0xfe000000

0xffffffff

Donate this free memory to the allocator





CS : 0x8
SS : 0x10
GDT: 0x7c78
IDT: 0x0
EIP: main
ESP: stack
TSS: 0x0
CR3: entrypml4

Protected Mode

- Take memory from the end of the kernel binary
- To the end of the 4MB page

kinit1(): initialize the allocator with free memory

```
int
main(void)
{
  void *kinit1_end = (1024*1024*1024 > PHYSTOP) ? P2V(PHYSTOP)
                                                 : P2V(1024*1024*1024);
  kinit1(end, kinit1_end); // phys page allocator
  kvmalloc(); // kernel page table
  mpinit(); // detect other processors
  lapicinit(); // interrupt controller
  seginit(); // segment descriptors
  picinit(); // disable pic
  ioapicinit(); // another interrupt controller
    ...
```

Freerange()

```
kinit1(void *vstart, void *vend)
{
…
  freerange(vstart, vend);
}
```

- Free range of memory from vstart to vend giving it to the allocator
- i.e., adding pages to the list

freerange()

```
freerange(void *vstart, void *vend)
{
  char *p;
  p = (char*)PGROUNDUP((uint)vstart);
  for(; p + PGSIZE <= (char*)vend; p += PGSIZE)
    kfree(p);
}
```

- freerange() internally simply frees the pages from vstart to vend
- kfree() adds them to the allocator list

Where do we start?

```
int
main(void)
{
  void *kinit1_end = (1024*1024*1024 > PHYSTOP) ? P2V(PHYSTOP)
                                    : P2V(1024*1024*1024);
  kinit1(end, kinit1_end); // phys page allocator
```

- What is this **end**?

```
extern char end[];
```

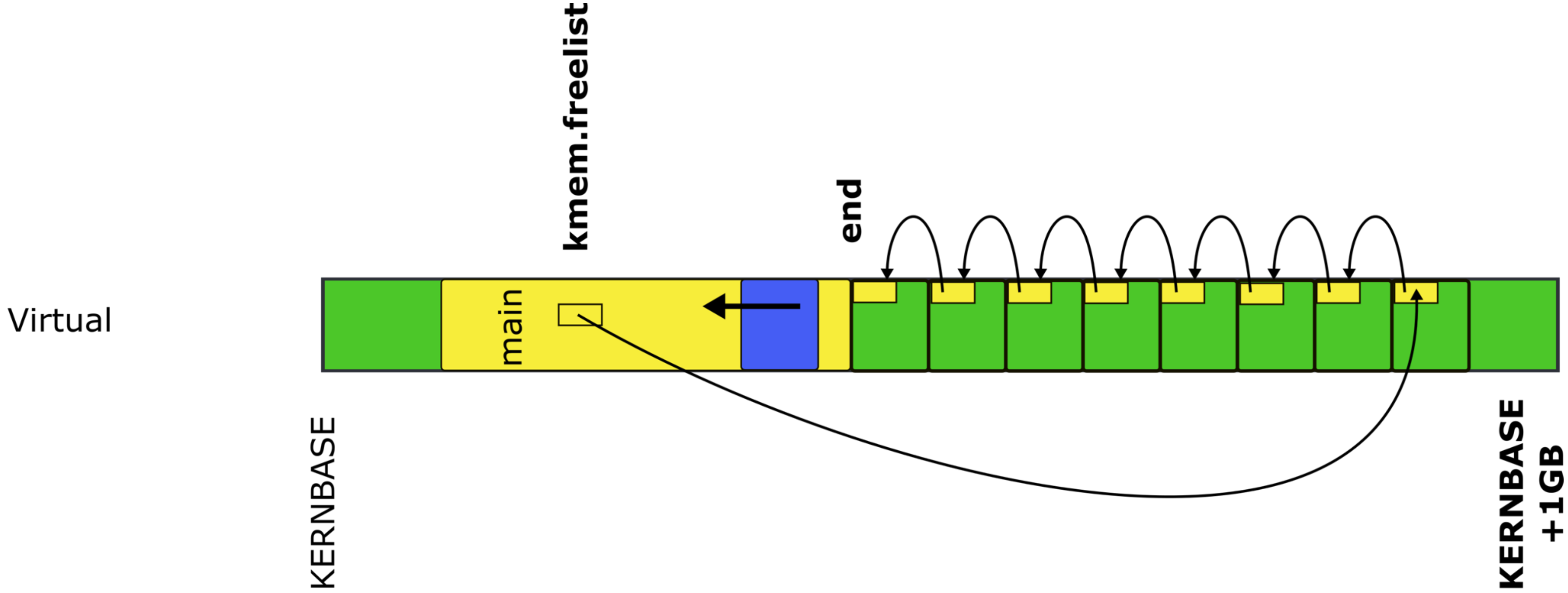
Where do we start?

```
int
main(void)
{
  void *kinit1_end = (1024*1024*1024 > PHYSTOP) ? P2V(PHYSTOP)
                                                : P2V(1024*1024*1024);
  kinit1(end, kinit1_end); // phys page allocator
```

- What is this **end**?

```
extern char end[]; // first address after
                kernel loaded from ELF file
```

Donate this free memory to the allocator



CS : 0x8 EIP: main
SS : 0x10 ESP: stack
GDT: 0x7c78 TSS: 0x0
IDT: 0x0 CR3: entrypml4

Protected Mode

Recap

- Kernel has a memory allocator
- It allocates memory in chunks of 4KB
- Good enough to maintain kernel data structures

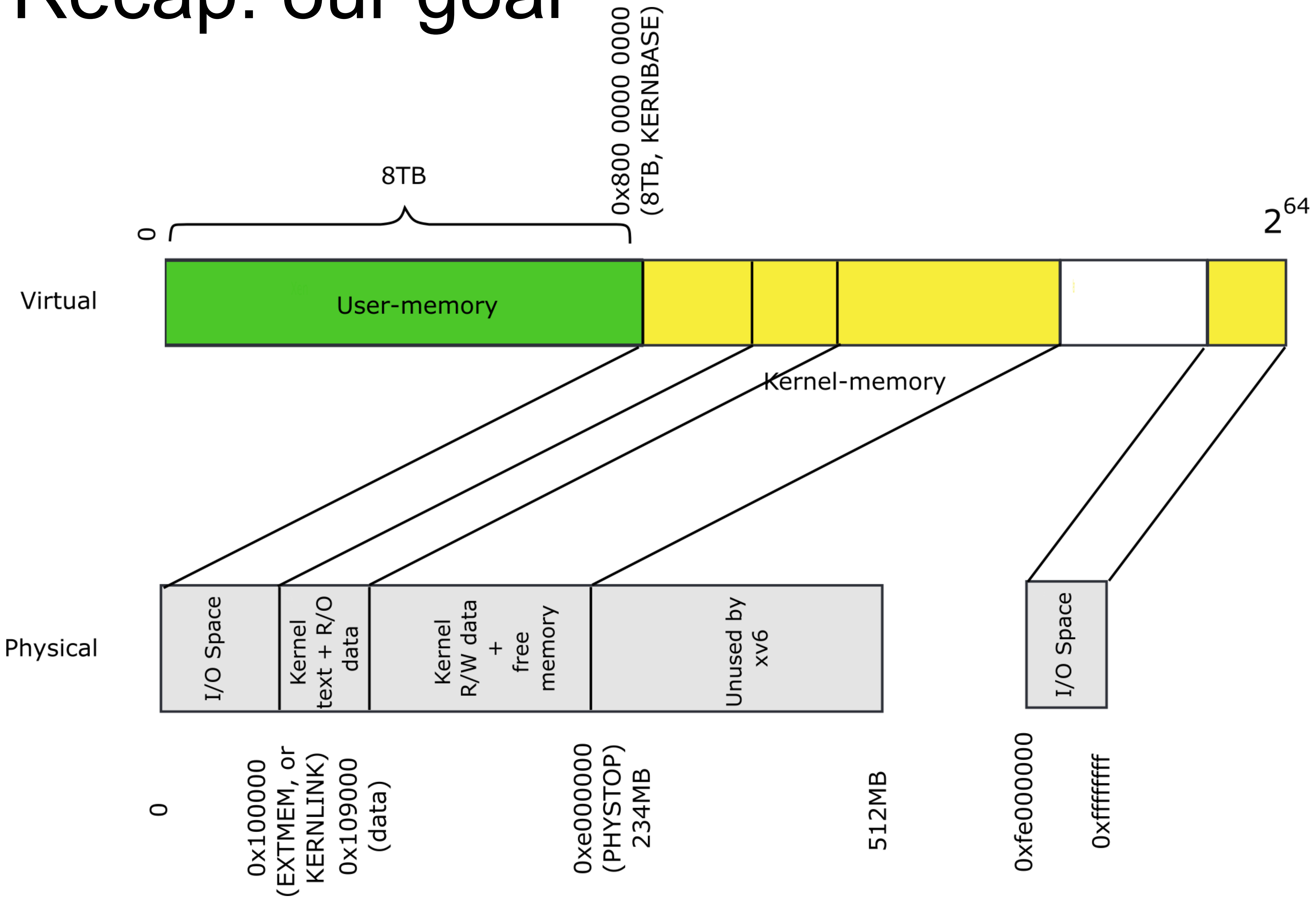
Kernel page table
(for 4KB page tables)

Back to main(): Kernel address space

```
int
main(void)
{
  void *kinit1_end = (1024*1024*1024 > PHYSTOP) ? P2V(PHYSTOP)
                                                : P2V(1024*1024*1024);
  kinit1(end, kinit1_end); // phys page allocator
  kvmalloc(); // kernel page table
  mpinit(); // detect other processors
  lapicinit(); // interrupt controller
  seginit(); // segment descriptors
  picinit(); // disable pic
  ioapicinit(); // another interrupt controller
…
```

- What do you think has to happen?
  - i.e., how do we construct a kernel page table?

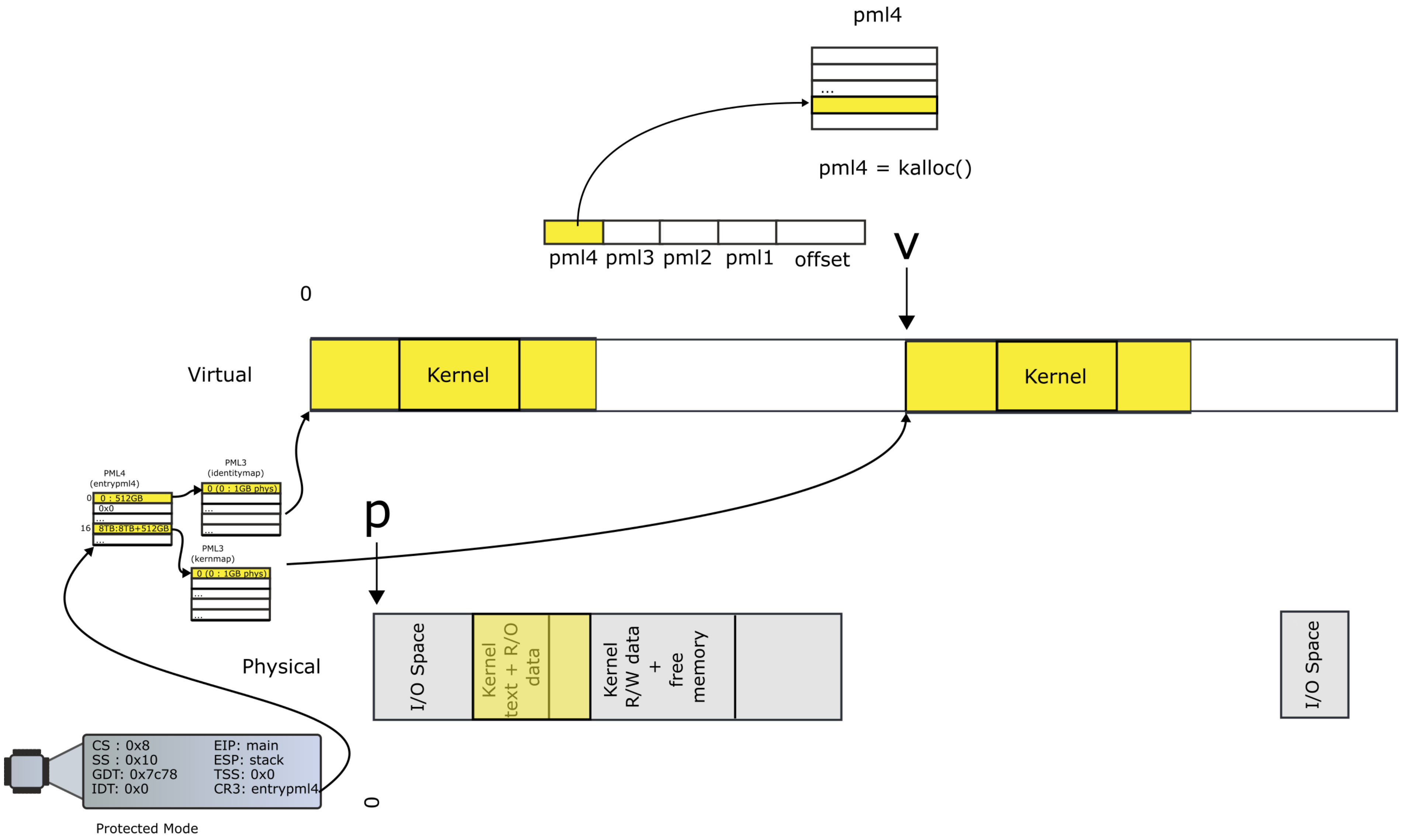
Recap: our goal



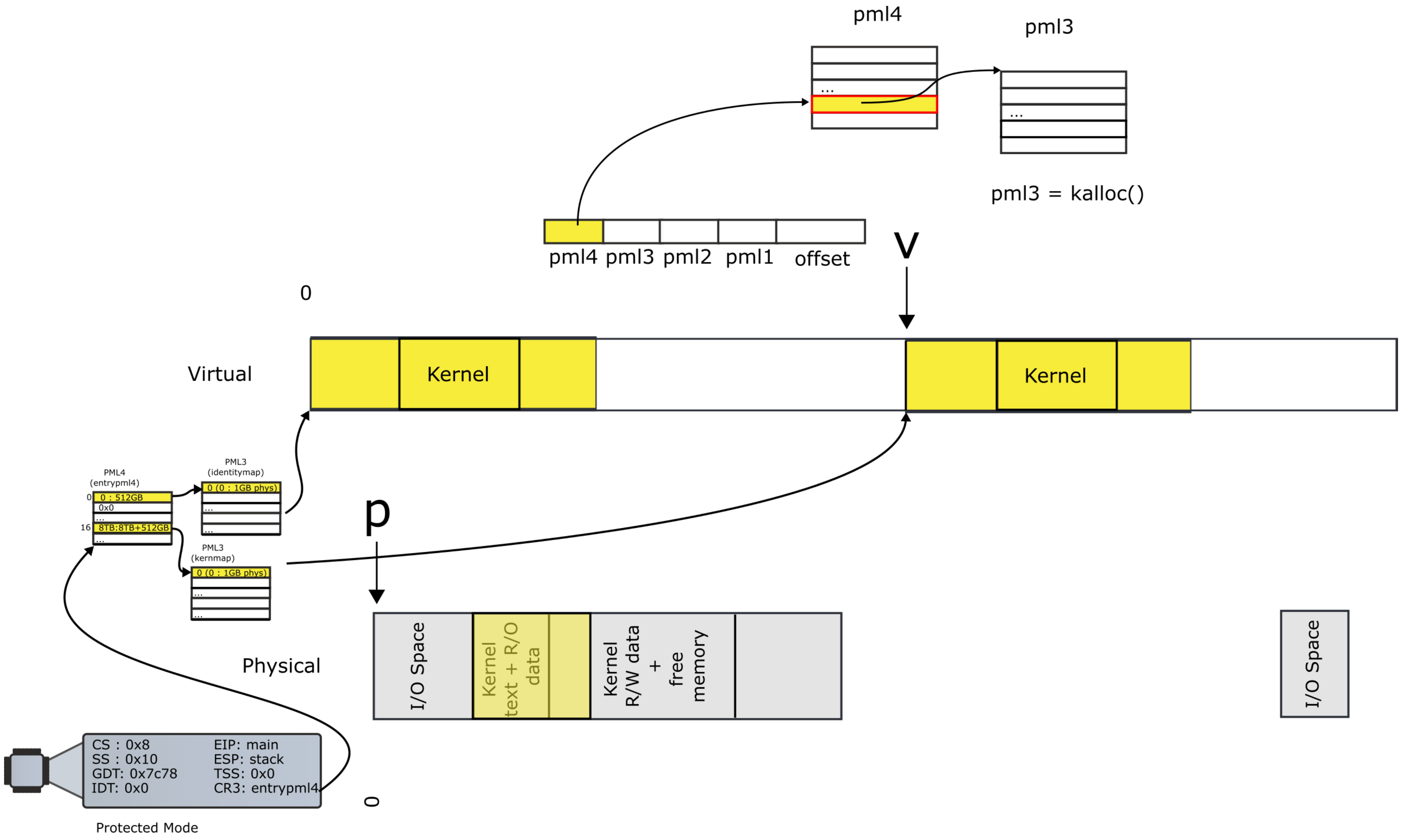
Outline

- Map a region of virtual memory into page tables
  - Start from 2GBs
  - Iterate memory page by page
  - Allocate page table directory and page tables as we go
  - Fill in page table entries with proper physical addresses

Allocate page table directory entry



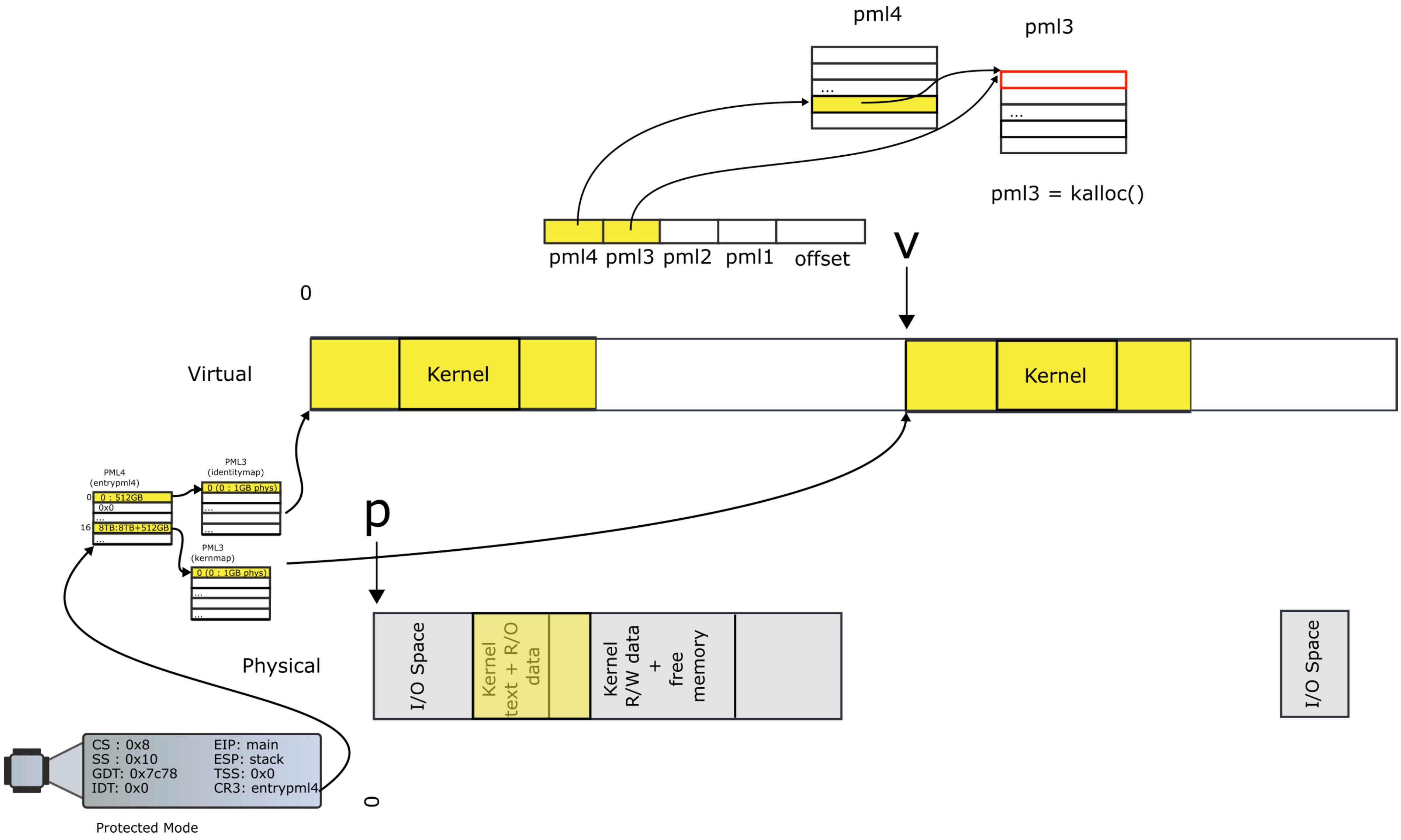
Allocate next level page table



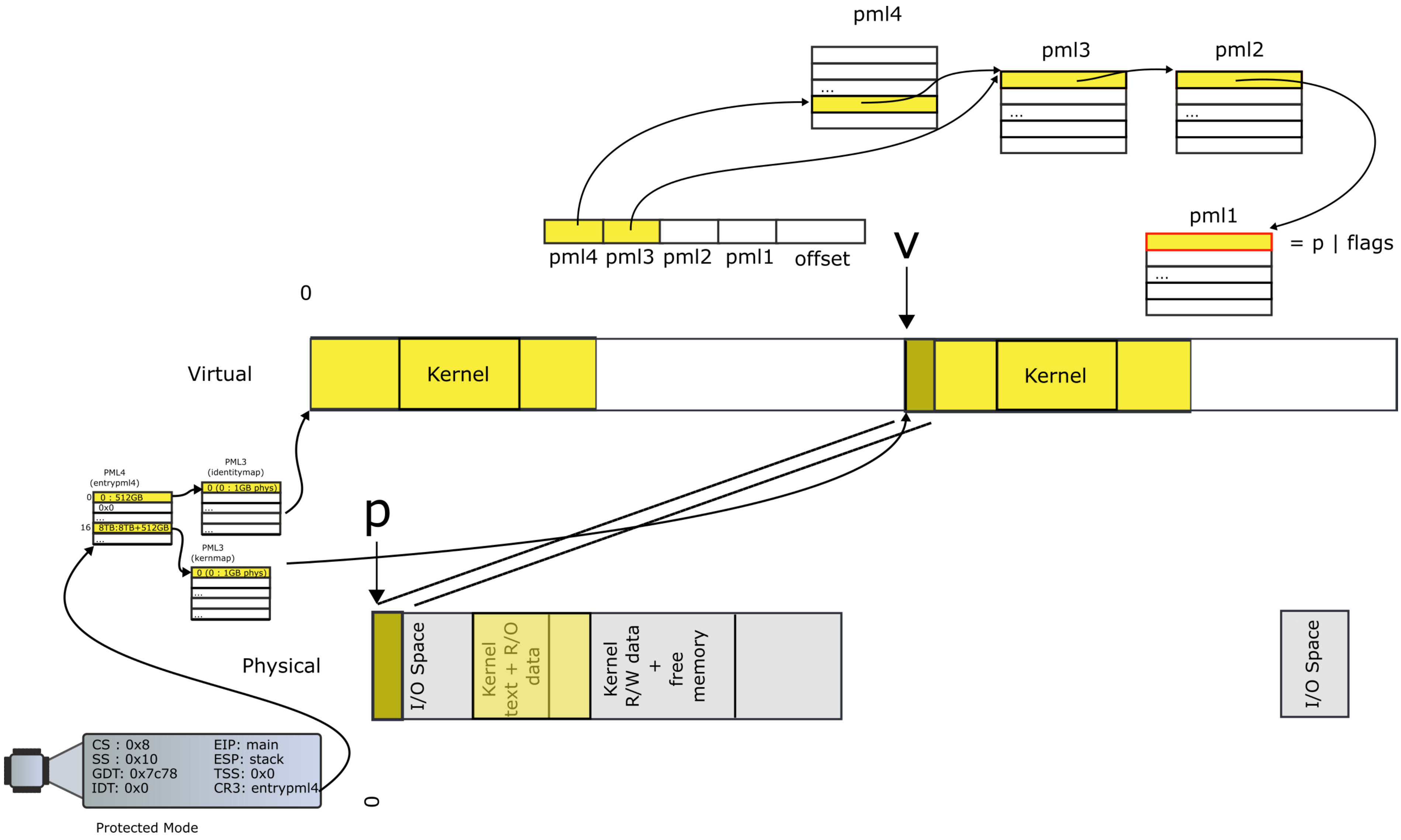
pml3 = kalloc()

CS : 0x8
SS : 0x10
GDT: 0x7c78
IDT: 0x0
EIP: main
ESP: stack
TSS: 0x0
CR3: entrypml4

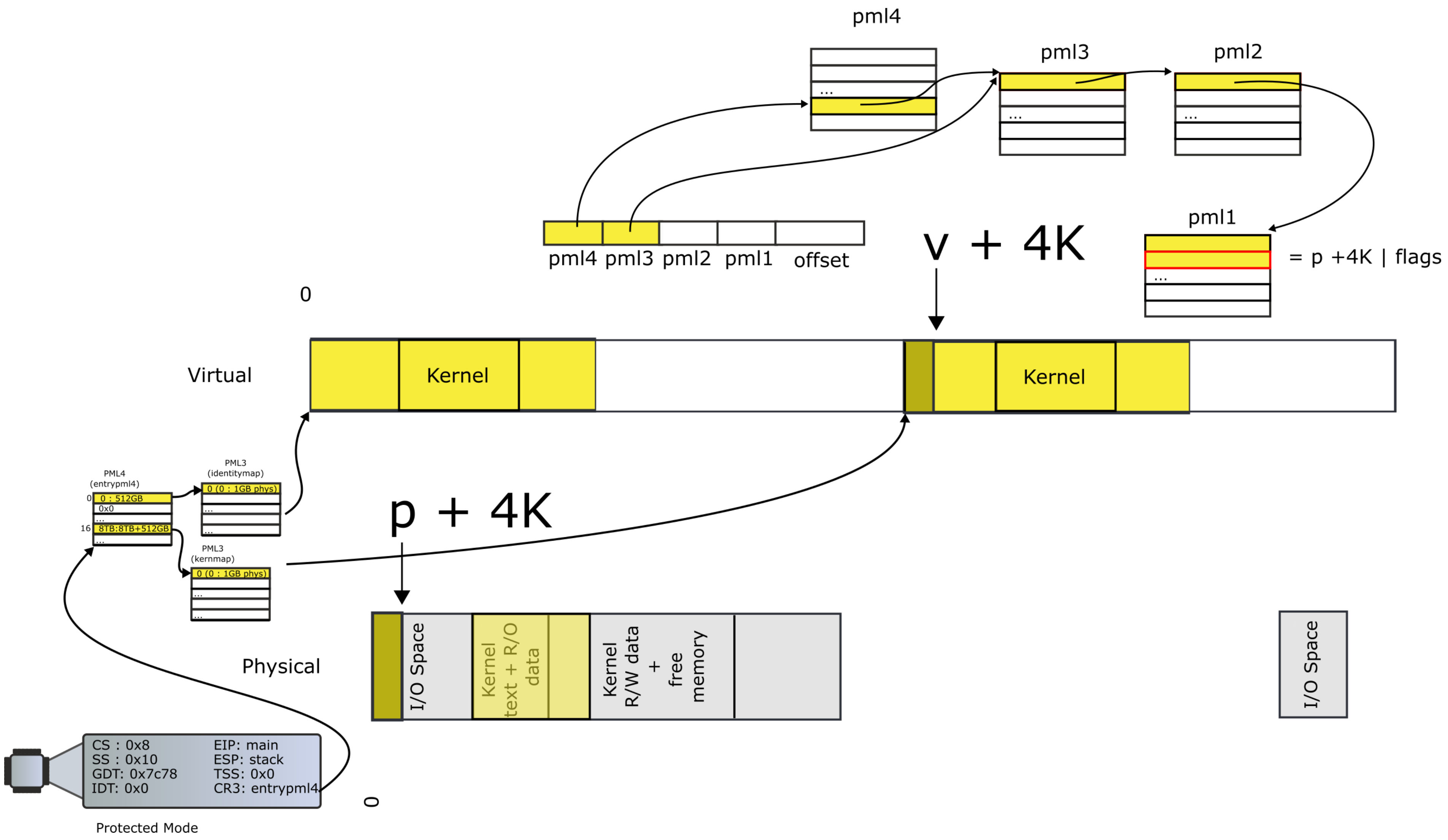
Locate PTE entry



Update mapping with physical addr



Move to next page



CS : 0x8
SS : 0x10
GDT: 0x7c78
IDT: 0x0

EIP: main
ESP: stack
TSS: 0x0
CR3: entrypml4

Protected Mode

This is exactly what kernel is doing

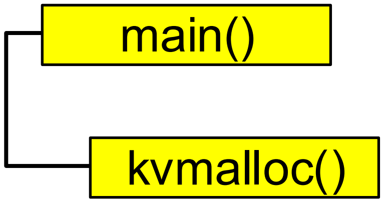
(let’s read the source code)

```
// Bootstrap processor starts running C code here.
// Allocate a real stack and switch to it, first
// doing some setup required for memory allocator to work.
int
main(void)
{
  void *kinit1_end = (1024*1024*1024 > PHYSTOP) ? P2V(PHYSTOP)
                                                 : P2V(1024*1024*1024);
  kinit1(end, kinit1_end); // phys page allocator
  kvmalloc(); // kernel page table
  mpinit(); // detect other processors
  lapicinit(); // interrupt controller
  seginit(); // segment descriptors
  picinit(); // disable pic
  ioapicinit(); // another interrupt controller
  consoleinit(); // console hardware
...
}
```

Allocate page tables

kvmalloc()

```
// Allocate one page table for the machine for the kernel address
// space for scheduler processes.
void
kvmalloc(void)
{
  kpml4 = setupkvm();
  switchkvm();
}
```

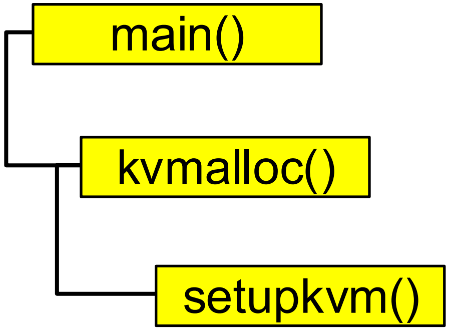


main()

kvmalloc()

Allocate page table directory

```
setupkvm(void)
{
  pml4e_t *pml4;
  struct kmap *k;

  if((pml4 = (pml4e_t*)kalloc()) == 0)
    return 0;
  memset(pml4, 0, PGSIZE);
  if (P2V(PHYSTOP) > DEV_P2V(DEVSPACE))
    panic("PHYSTOP too high");
  for(k = kmap; k < &kmap[NELEM(kmap)]; k++)
    if(mappages(pml4, k->virt, k->phys_end - k->phys_start,
                (uint64)k->phys_start, k->perm) < 0) {
      freevm(pml4);
      return 0;
    }
  return pml4;
}
```



main()

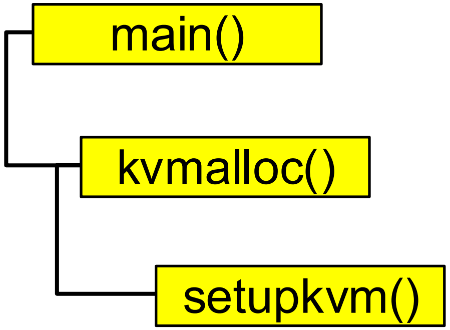
kvmalloc()

setupkvm()

e table directory

What is the address of this page?

```
setupkvm(void)
{
  pml4e_t *pml4;
  struct kmap *k;

  if((pml4 = (pml4e_t*)kalloc()) == 0)
    return 0;
  memset(pml4, 0, PGSIZE);
  if (P2V(PHYSTOP) > DEV_P2V(DEVSPACE))
    panic("PHYSTOP too high");
  for(k = kmap; k < &kmap[NELEM(kmap)]; k++)
    if(mappages(pml4, k->virt, k->phys_end - k->phys_start,
                (uint64)k->phys_start, k->perm) < 0) {
      freevm(pml4);
      return 0;
    }
  return pml4;
}
```

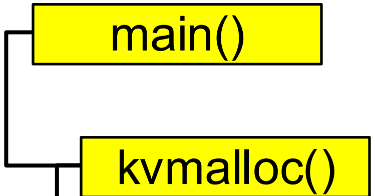


https://pollev.com/cs5460

```
setupkvm(void)
{
  pml4e_t *pml4;
  struct kmap *k;

  if((pml4 = (pml4e_t*)kalloc()) == 0)
    return 0;
  memset(pml4, 0, PGSIZE);
  if (P2V(PHYSTOI
    panic("PHYSTO
  for(k = kmap; k
    if(mappages(p
             (uint64)k
      freevm(pml4)
      return 0;
    }
  return pml4;
}
```

What is the address of this page?

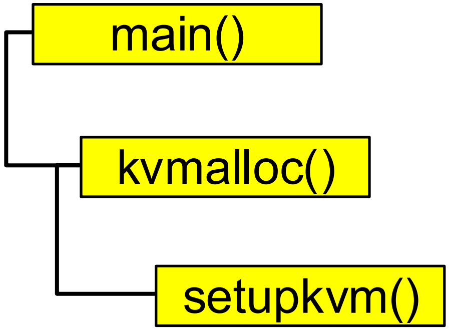
e table directory



```
setupkvm(void)
{
  pml4e_t *pml4;
  struct kmap *k;

  if((pml4 = (pml4e_t*)kalloc()) == 0)
    return 0;
  memset(pml4, 0, PGSIZE);
  if (P2V(PHYSTOP) > DEV_P2V(DEVSPACE))
    panic("PHYSTOP too high");
  for(k = kmap; k < &kmap[NELEM(kmap)]; k++)
    if(mappages(pml4, k->virt, k->phys_end - k->phys_start,
                (uint64)k->phys_start, k->perm) < 0) {
      freevm(pml4);
      return 0;
    }
  return pml4;
}
```

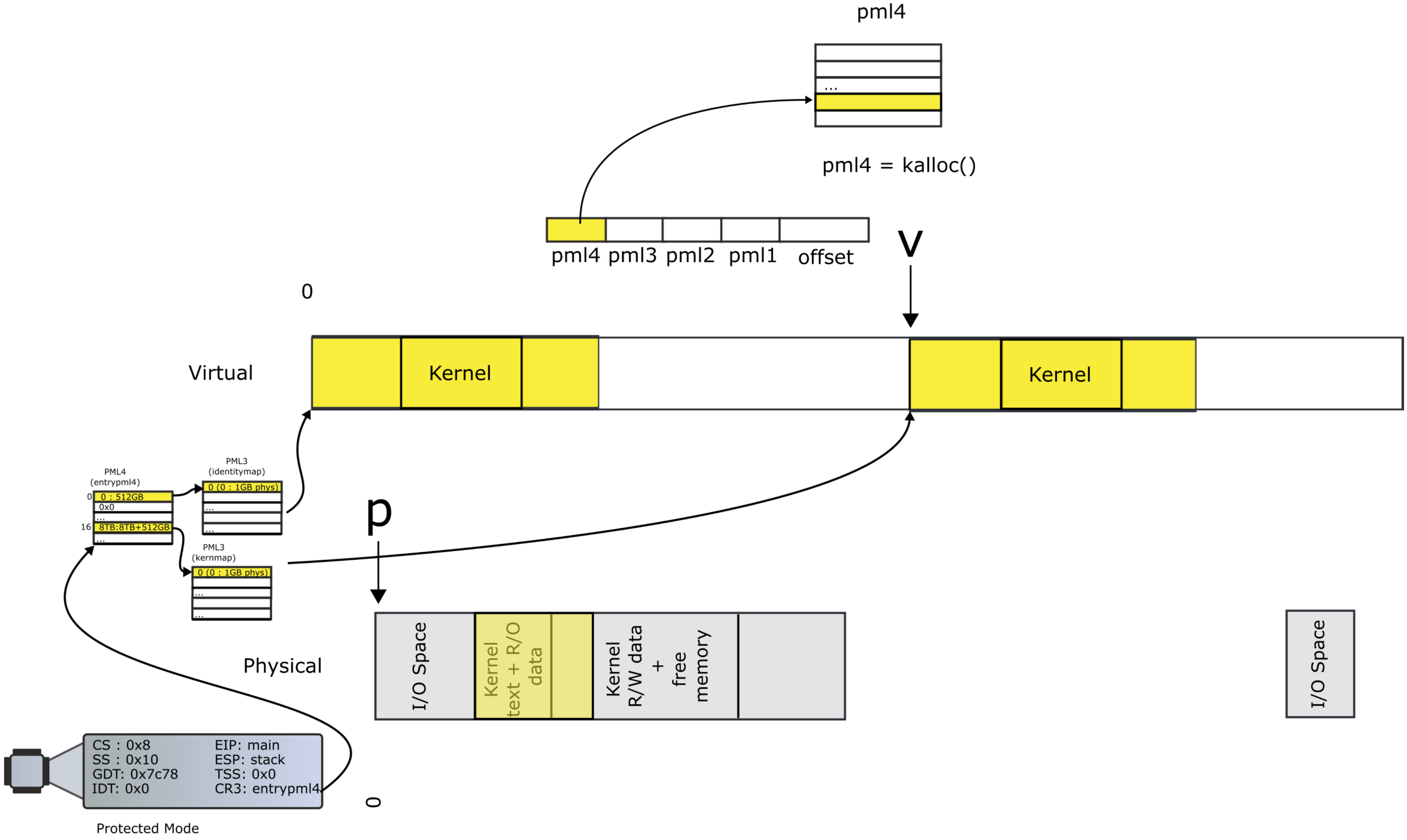
What is the address of this page?

e table directory



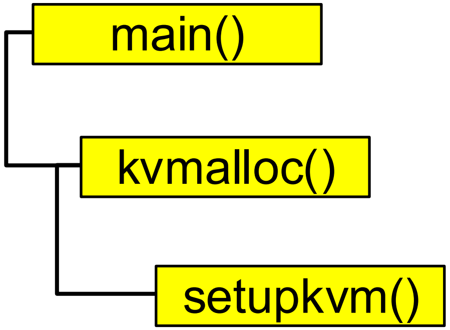
https://pollev.com/cs5460

Allocate page table directory



Iterate in a loop: map physical pages

```
setupkvm(void)
{
  pml4e_t *pml4;
  struct kmap *k;

  if((pml4 = (pml4e_t*)kalloc()) == 0)
    return 0;
  memset(pml4, 0, PGSIZE);
  if (P2V(PHYSTOP) > DEV_P2V(DEVSPACE))
    panic("PHYSTOP too high");
  for(k = kmap; k < &kmap[NELEM(kmap)]; k++)
    if(mappages(pml4, k->virt, k->phys_end - k->phys_start,
                (uint64)k->phys_start, k->perm) < 0) {
      freevm(pml4);
      return 0;
    }
  return pml4;
}
```



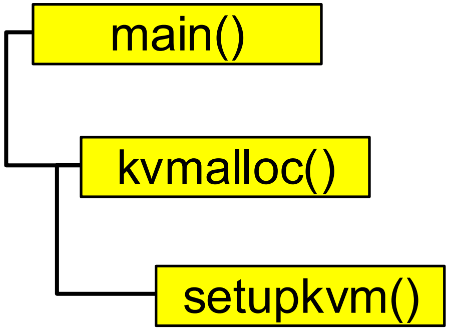
main()

kvmalloc()

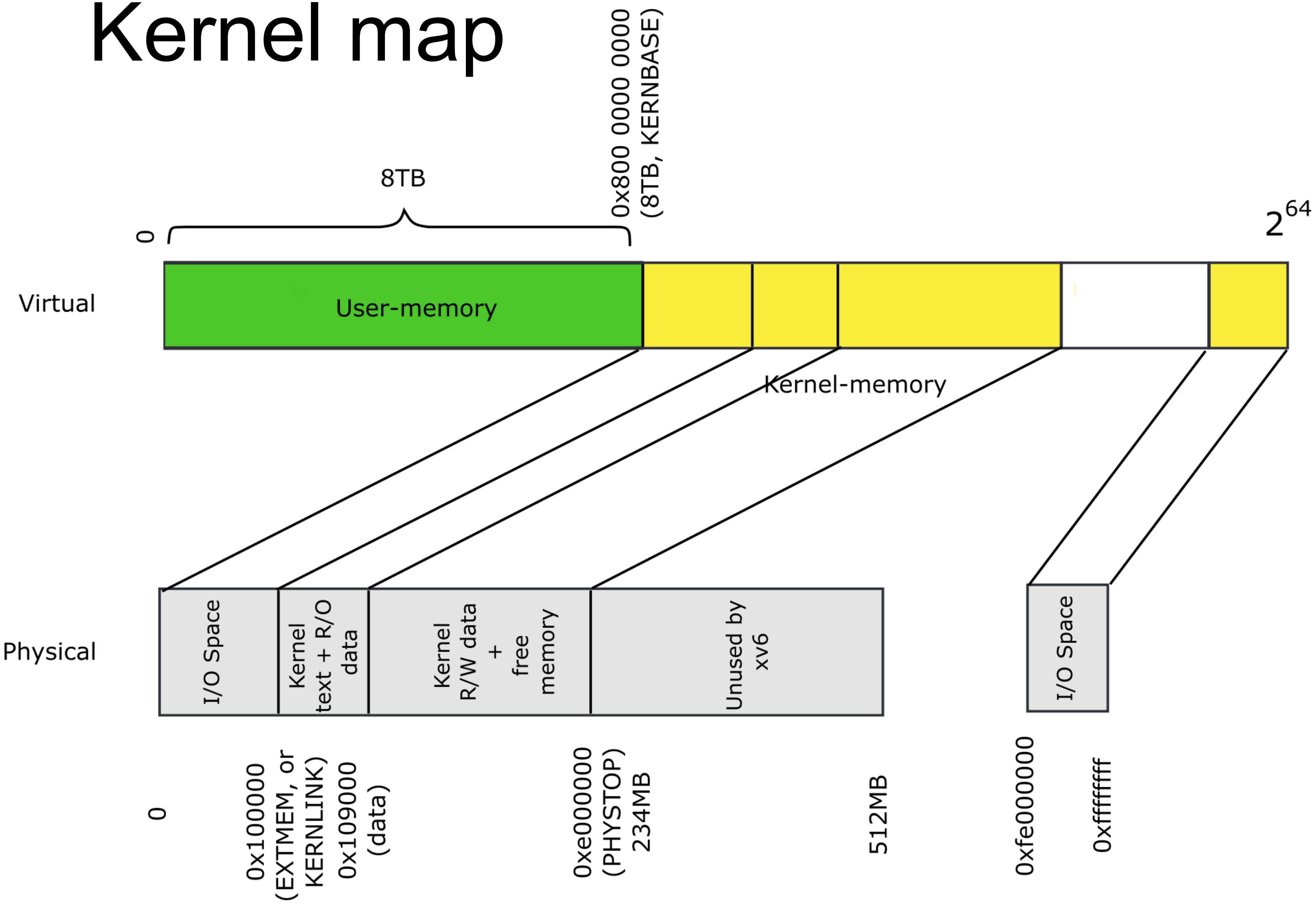
setupkvm()

Iterate in a loop: map physical pages

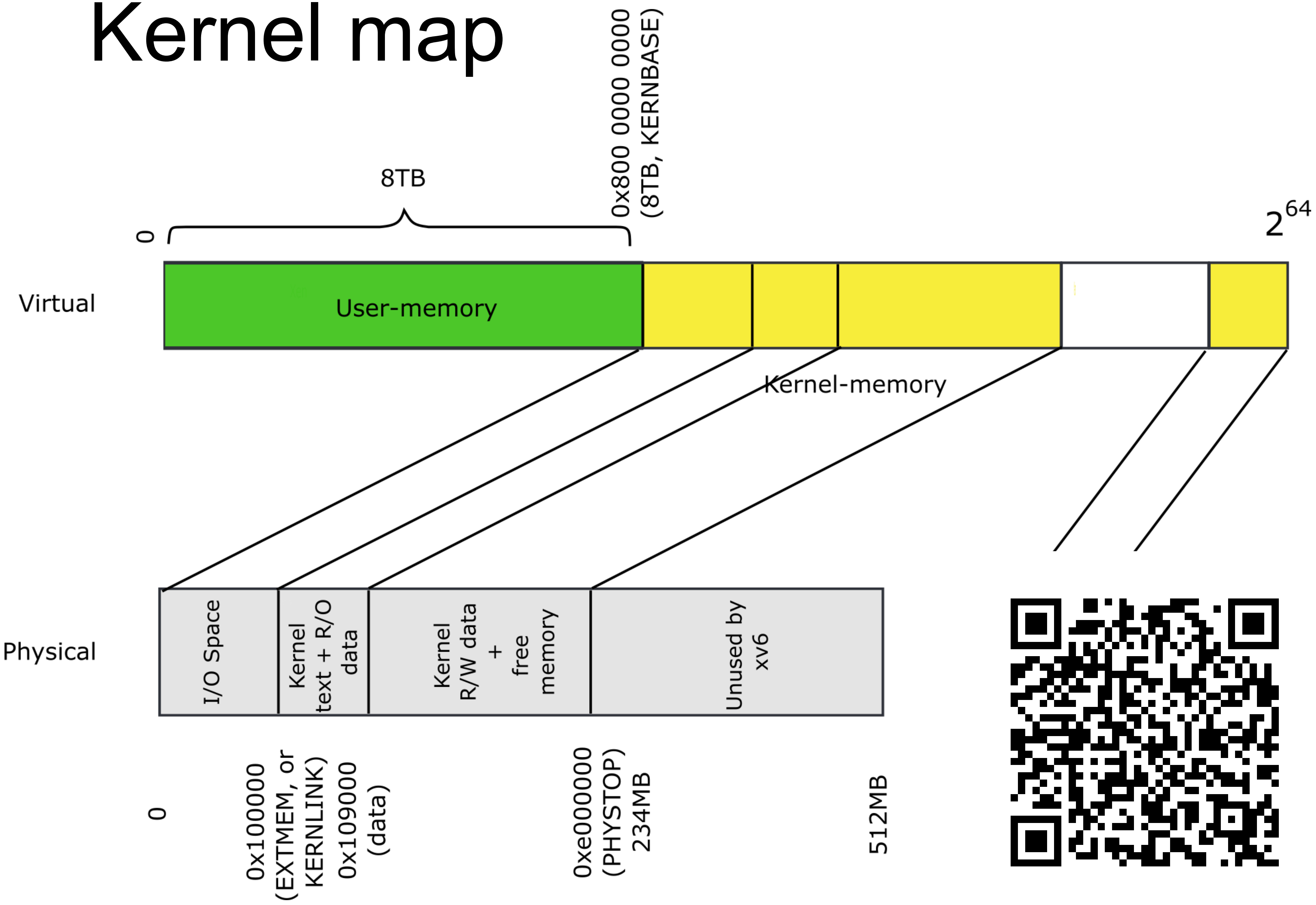
```
setupkvm(void)
{
  pml4e_t *pml4;
  struct kmap *k;

  if((pml4 = (pml4e_t*)kalloc()) == 0)
    return 0;
  memset(pml4, 0, PGSIZE);
  if (P2V(PHYSTOP) > DEV_P2V(DEVSPACE))
    panic("PHYSTOP too high");
  for(k = kmap; k < &kmap[NELEM(kmap)]; k++)
    if(mappages(pml4, k->virt, k->phys_end - k->phys_start,
            (uint64)k->phys_start, k->perm) < 0) {
      freevm(pml4);
      return 0;
    }
  return pml4;
}
```



Kernel map


8TB
0
0x800 0000 0000
(8TB, KERNBASE)
2^{64}
Virtual
User-memory
Kernel-memory
Physical
I/O Space
Kernel
text + R/O
data
Kernel
R/W data
+
free
memory
Unused by
xv6
I/O Space
0
0x100000
(EXTMEM, or
KERNLINK)
0x109000
(data)
0xe000000
(PHYSTOP)
234MB
512MB
0xfe000000
0xffffffff

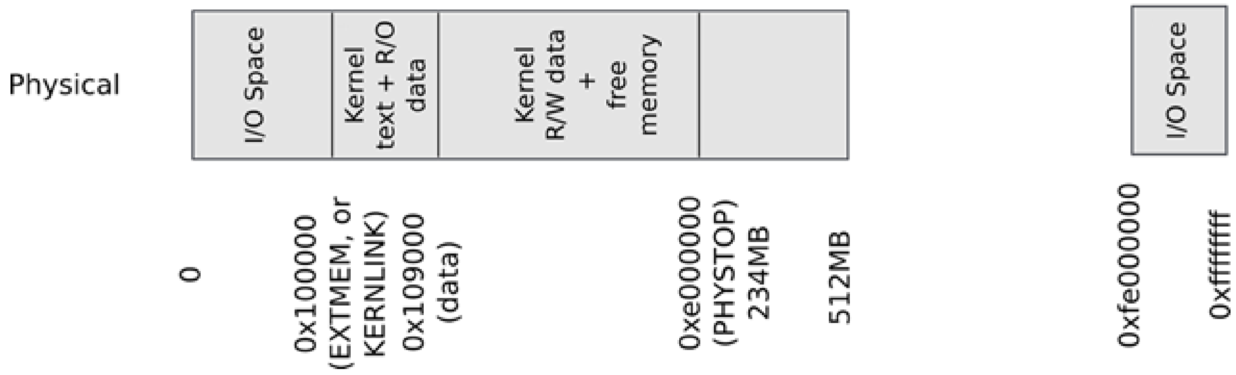
Kernel map



https://pollev.com/cs5460

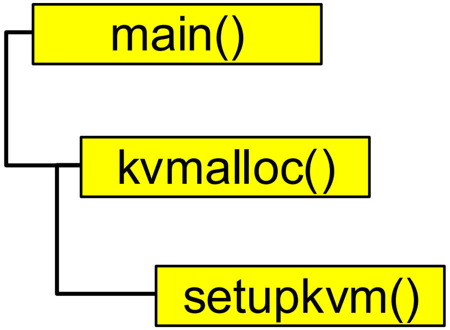
Kmap – kernel map

```
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},//text+rodata
    { (void*)data, V2P(data), PHYSTOP, PTE_W}, // kern data+memory
    { (void*)DEVSPACE, DEVSPACE, 0, PTE_W}, // more devices
};
```



Iterate in a loop: map physical pages

```
setupkvm(void)
{
  pml4e_t *pml4;
  struct kmap *k;

  if((pml4 = (pml4e_t*)kalloc()) == 0)
    return 0;
  memset(pml4, 0, PGSIZE);
  if (P2V(PHYSTOP) > DEV_P2V(DEVSPACE))
    panic("PHYSTOP too high");
  for(k = kmap; k < &kmap[NELEM(kmap)]; k++)
    if(mappages(pml4, k->virt, k->phys_end - k->phys_start,
           (uint64)k->phys_start, k->perm) < 0) {
      freevm(pml4);
      return 0;
    }
  return pml4;
}
```



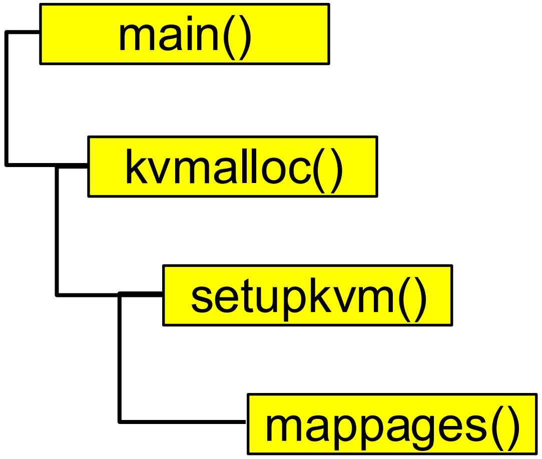
main()

kvmalloc()

setupkvm()

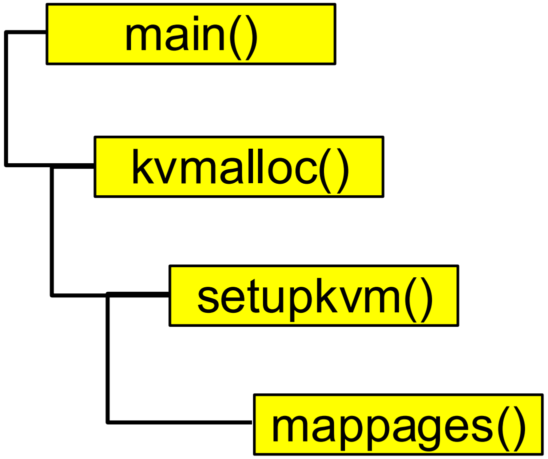
Inside mappages(

```
mappages(pml4e_t *pml4, void *va, uint size, uint64 pa, int perm)
{
  char *a, *last;
  pte_t *pte;

  a = (char*)PGROUNDDOWN((uint64)va);
  last = (char*)PGROUNDDOWN(((uint64)va) + size - 1);
  for(;;){
   if((pte = walkpml4(pml4, a, 1, 0)) == 0)
     return -1;
   if(*pte & PTE_P)
     panic("remap");
   *pte = pa | perm | PTE_P;
   if(a == last)
     break;
   a += PGSIZE;
   pa += PGSIZE;
  }
  return 0;
}
```



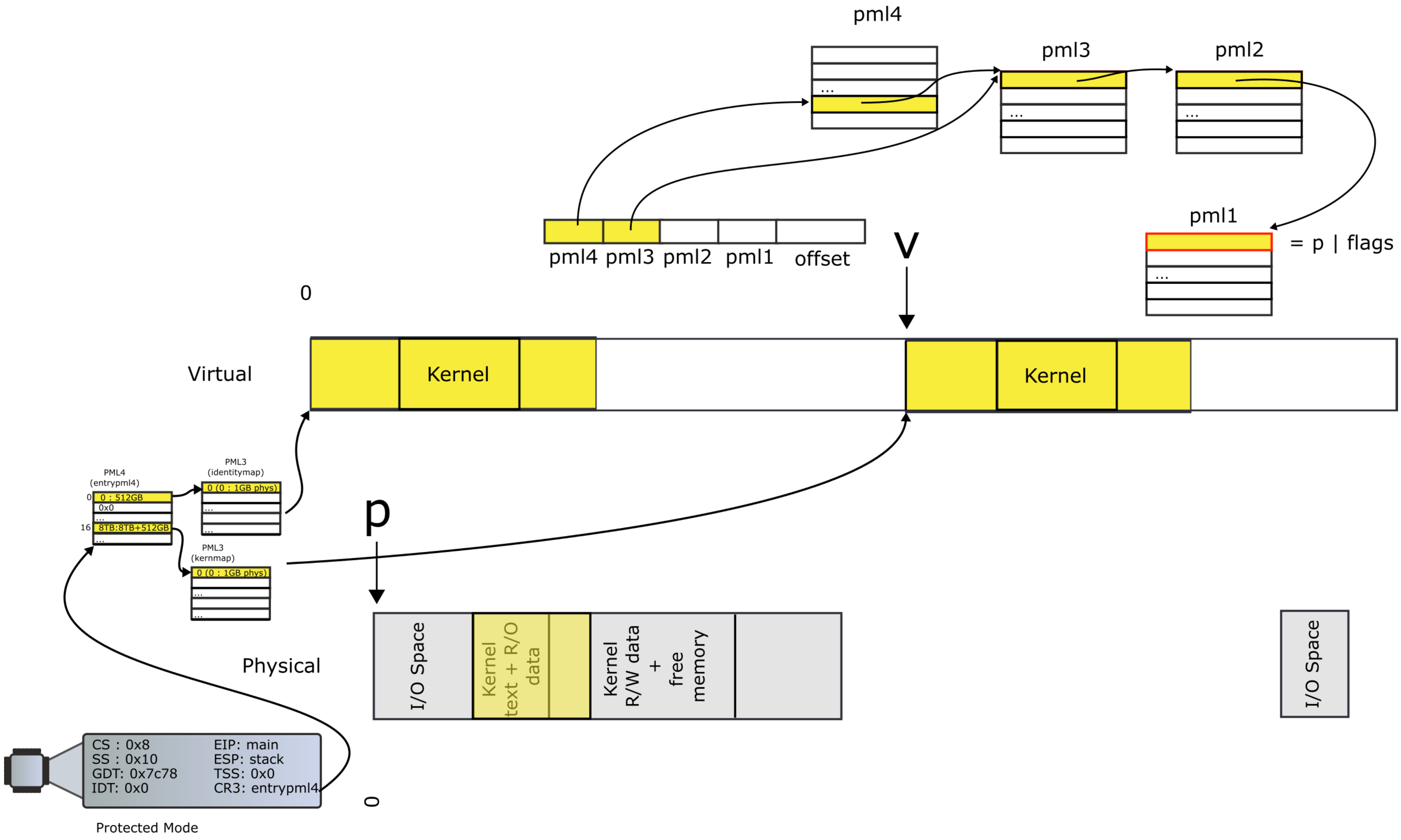
- Get the start (a) and end (last) pages fo the virtual address range we are mapping
- Then work in a loop mapping every page one by one

```
mappages(pml4e_t *pml4, void *va, uint size, uint64 pa, int perm)
{
  char *a, *last;
  pte_t *pte;

  a = (char*)PGROUNDDOWN((uint64)va);
  last = (char*)PGROUNDDOWN(((uint64)va) + size - 1);
  for(;;){
    if((pte = walkpml4(pml4, a, 1, 0)) == 0)
      return -1;
    if(*pte & PTE_P)
      panic("remap");
    *pte = pa | perm | PTE_P;
    if(a == last)
      break;
    a += PGSIZE;
    pa += PGSIZE;
  }
  return 0;
}
```



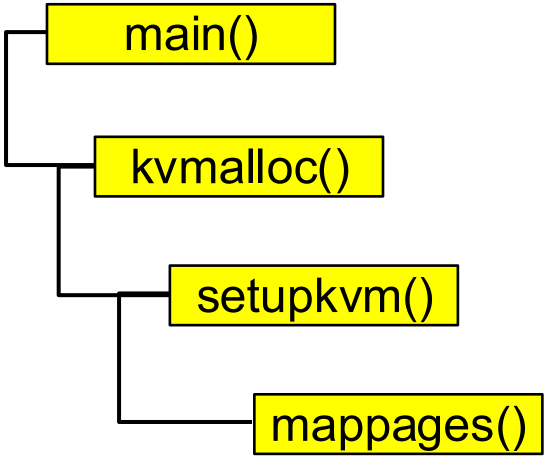
- First lookup the page table entry (pte) corresponding to the virtual address (a) we’re mapping

Locate the page table entry



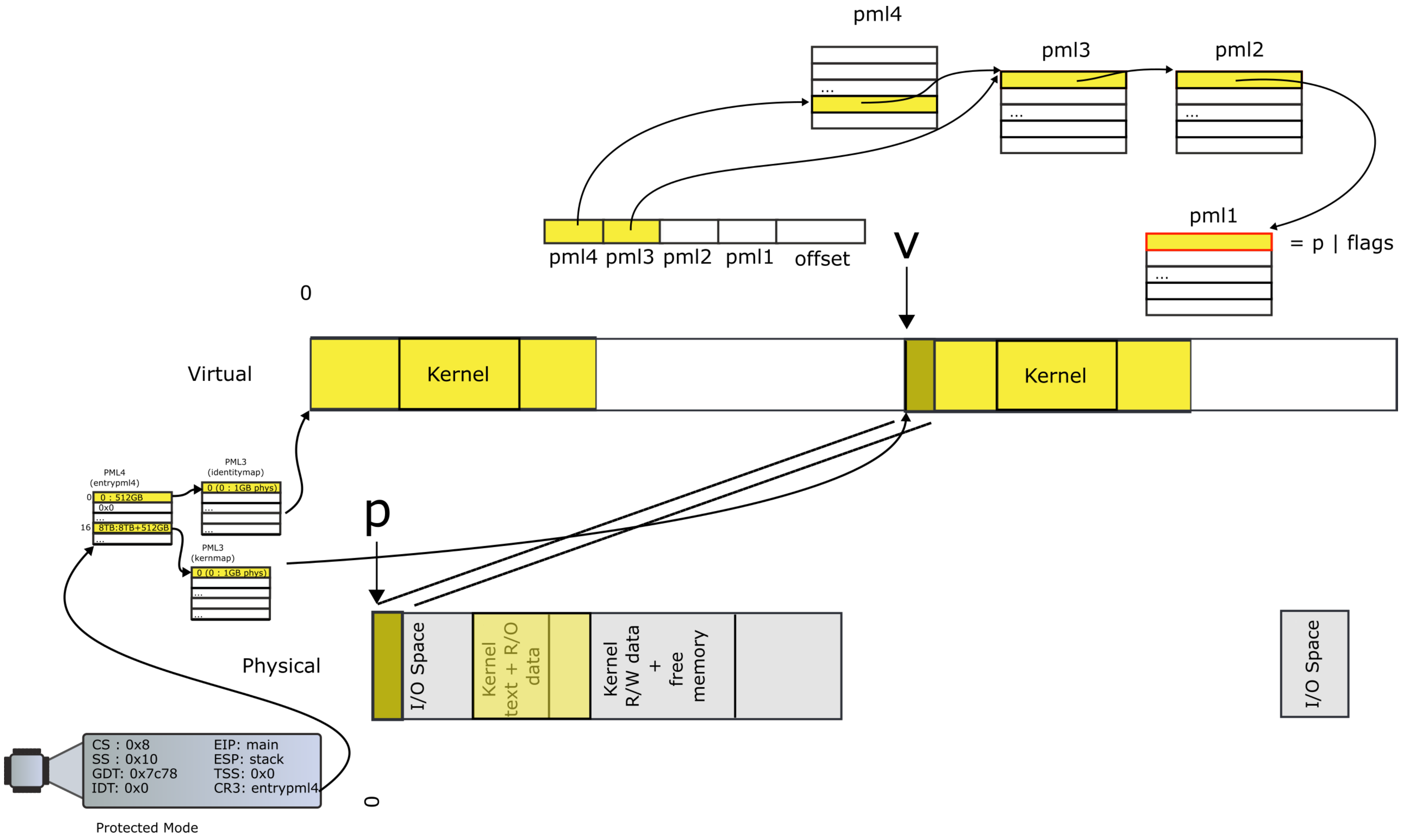
CS : 0x8
SS : 0x10
GDT: 0x7c78
IDT: 0x0
EIP: main
ESP: stack
TSS: 0x0
CR3: entrypml4

```
mappages(pml4e_t *pml4, void *va, uint size, uint64 pa, int perm)
{
  char *a, *last;
  pte_t *pte;

  a = (char*)PGROUNDDOWN((uint64)va);
  last = (char*)PGROUNDDOWN(((uint64)va) + size - 1);
  for(;;){
    if((pte = walkpml4(pml4, a, 1, 0)) == 0)
      return -1;
    if(*pte & PTE_P)
      panic("remap");
    *pte = pa | perm | PTE_P;
    if(a == last)
      break;
    a += PGSIZE;
    pa += PGSIZE;
  }
  return 0;
}
```

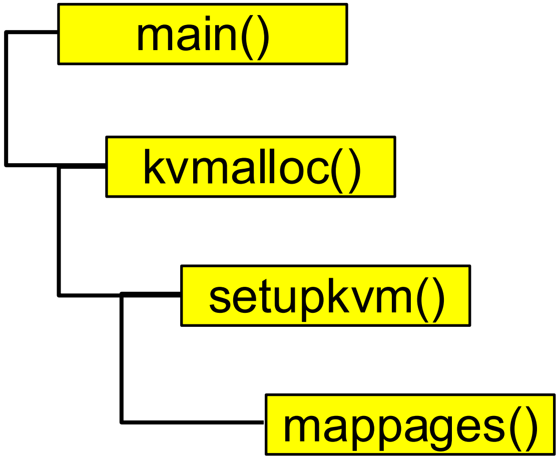


- Update the page directory entry (*pte) with the physical address (pa)

Update mapping with physical addr



```
mappages(pml4e_t *pml4, void *va, uint size, uint64 pa, int perm)
{
  char *a, *last;
  pte_t *pte;

  a = (char*)PGROUNDDOWN((uint64)va);
  last = (char*)PGROUNDDOWN(((uint64)va) + size - 1);
  for(;;){
    if((pte = walkpml4(pml4, a, 1, 0)) == 0)
      return -1;
    if(*pte & PTE_P)
      panic("remap");
    *pte = pa | perm | PTE_P;
    if(a == last)
      break;
    a += PGSIZE;
    pa += PGSIZE;
  }
  return 0;
}
```



main()

kvmalloc()

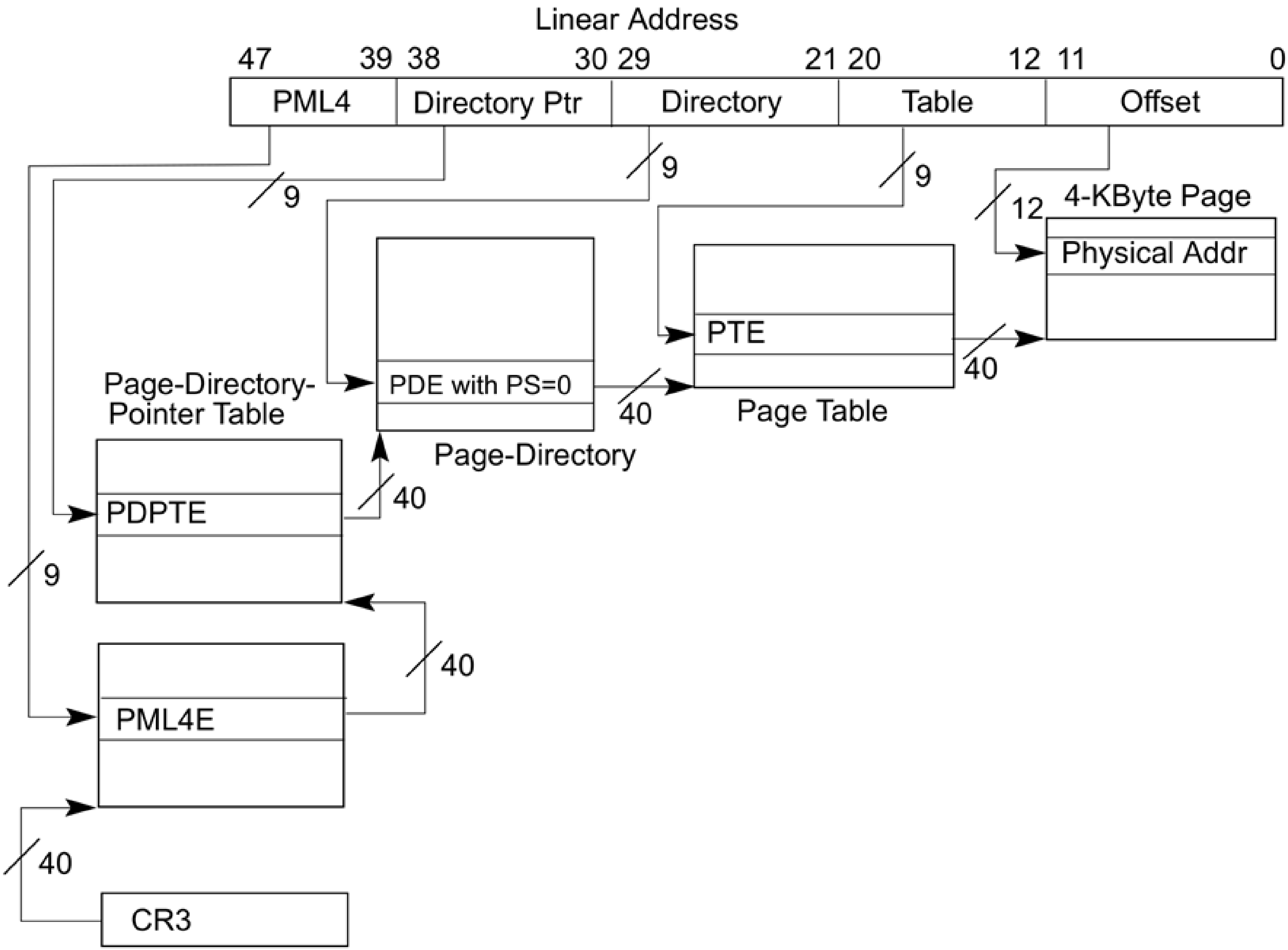
setupkvm()

mappages()

- But we need a function that locates the pte for us...

What should it look like?

- A function takes a virtual address
- Returns a page table directory entry that maps it


Linear Address
47
39
38
30
29
21
20
12
11
0
PML4
Directory Ptr
Directory
Table
Offset
9
9
9
12
4-KByte Page
Physical Addr
PTE
40
PDE with PS=0
40
Page Table
Page-Directory-
Pointer Table
Page-Directory
PDPTE
40
9
40
PML4E
40
CR3

```
// Return the address of the PTE in page table pml4
// that corresponds to virtual address va. If alloc!=0,
// create any required page table pages.
// On failure (return value equals 0), missing is set to the level
// in which the table was not present.
static pte_t *
walkpml4(pml4e_t *pml4, const void *va, int alloc, int *missing)
{
  return walkpglevel(pml4, va, alloc, 4, missing);
}
```

walkpml4(): walk page table

- Locate the page directory entry (*pde)

walkpml4(): walk page table

```
static pte_t *
walkpglevel(uint64 *pgdir, const void *va, int alloc, int level,
int *missing)
{
...
  switch (level){
  case 4:
    entry = &pgdir[PML4X(va)];
    break;
  case 3:
    entry = &pgdir[PDPTX(va)];
    break;
  case 2:
    entry = &pgdir[PDX(va)];
    break;
  case 1:
    return &pgdir[PTX(va)];
  default:
    panic("walkpglevel");
  }

  if(*entry & PTE_P){
    next_pgdir = (uint64*)P2V(PTE_ADDR(*entry));
  } else {
...
}
```

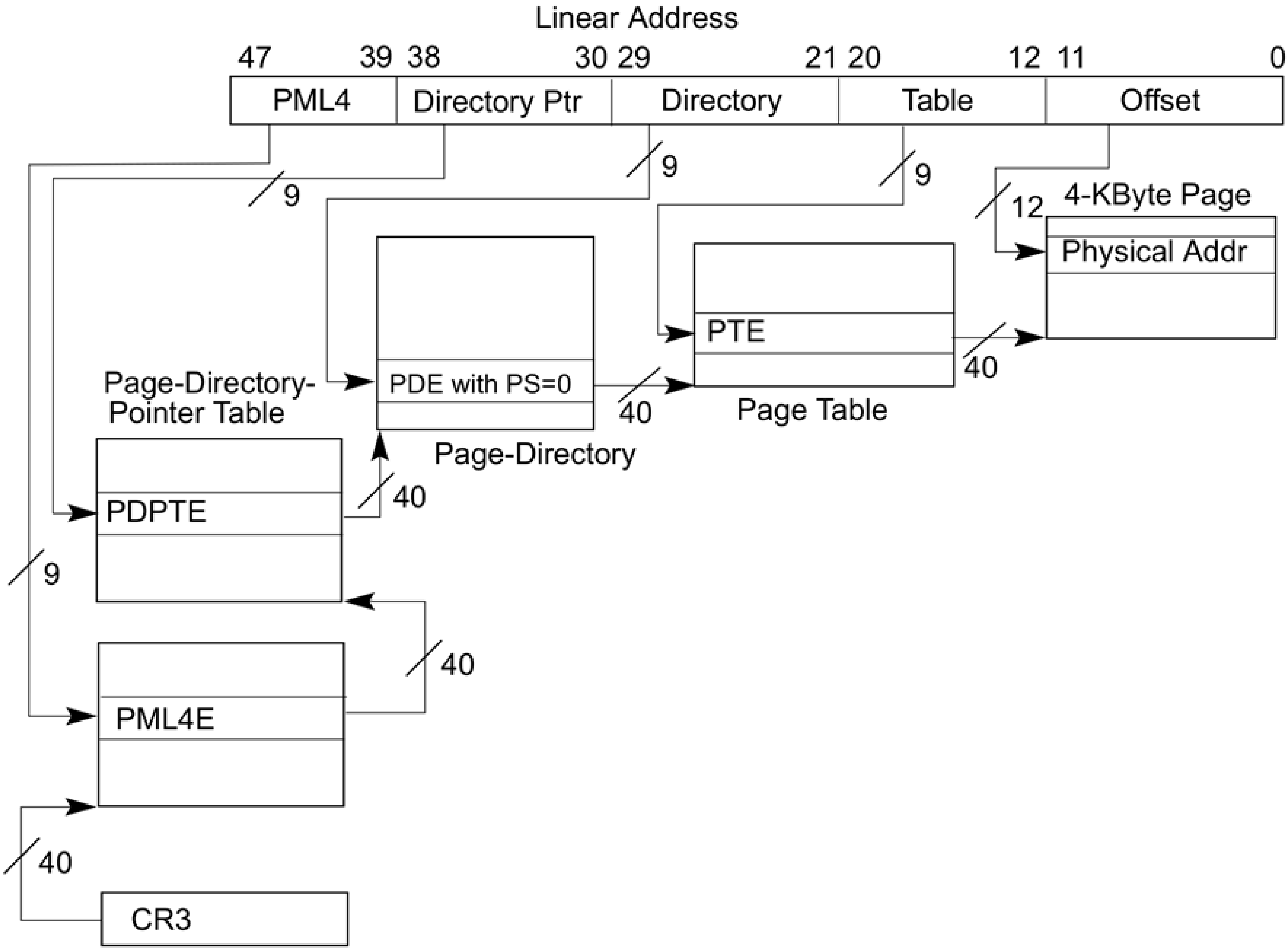
- Locate the page directory entry (*pde)

walkpglevel(): walk page table

```
walkpglevel(uint64 *pgdir, const void *va, int alloc, int level,
int *missing)
{
…

  if(*entry & PTE_P){
    next_pgdir = (uint64*)P2V(PTE_ADDR(*entry));
  } else {
    if(!alloc || (next_pgdir = (uint64*)kalloc()) == 0){
      if (missing)
        *missing = level;
        return 0;
      }
    // Make sure all those PTE_P bits are zero.
    memset(next_pgdir, 0, PGSIZE);
    // The permissions here are overly generous, but they can
    // be further restricted by the permissions in the page table
    // entries, if necessary.
    *entry = V2P(next_pgdir) | PTE_P | PTE_W | PTE_U;
  }
  return walkpglevel(next_pgdir, va, alloc, level - 1, missing);
}
```

- Check if page table is allocated (present)

Check if levels are allocated

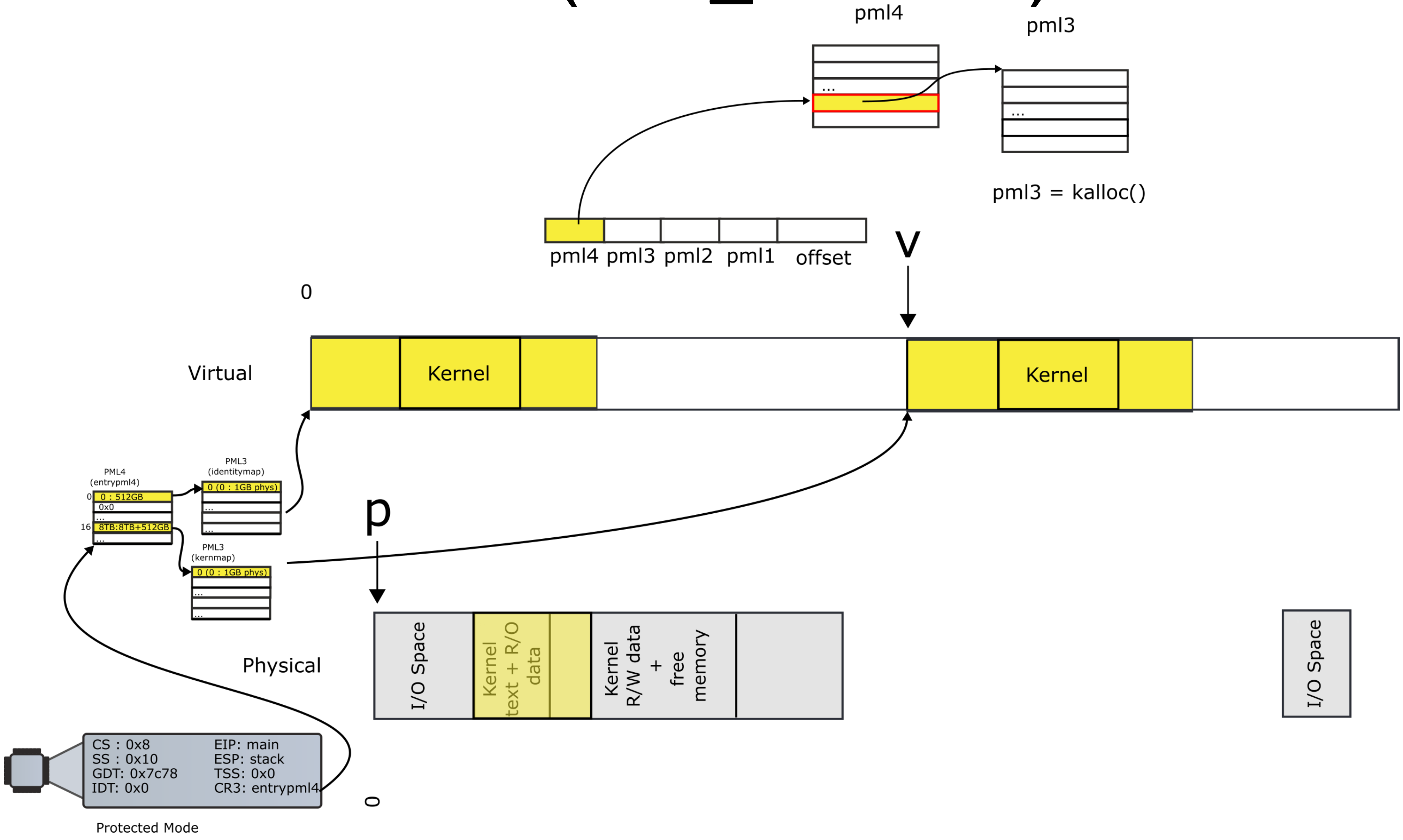


walkpgdir(): walk page table

```
walkpglevel(uint64 *pgdir, const void *va, int alloc, int level,
int *missing)
{
…

  if(*entry & PTE_P){
    next_pgdir = (uint64*)P2V(PTE_ADDR(*entry));
  } else {
    if(!alloc || (next_pgdir = (uint64*)kalloc()) == 0){
      if (missing)
        *missing = level;
        return 0;
      }
    // Make sure all those PTE_P bits are zero.
    memset(next_pgdir, 0, PGSIZE);
    // The permissions here are overly generous, but they can
    // be further restricted by the permissions in the page table
    // entries, if necessary.
    *entry = V2P(next_pgdir) | PTE_P | PTE_W | PTE_U;
  }
  return walkpglevel(next_pgdir, va, alloc, level - 1, missing);
}
```

- Allocate if needed

See if the next page table level exists (PTE_P is set)



CS : 0x8
SS : 0x10
GDT: 0x7c78
IDT: 0x0

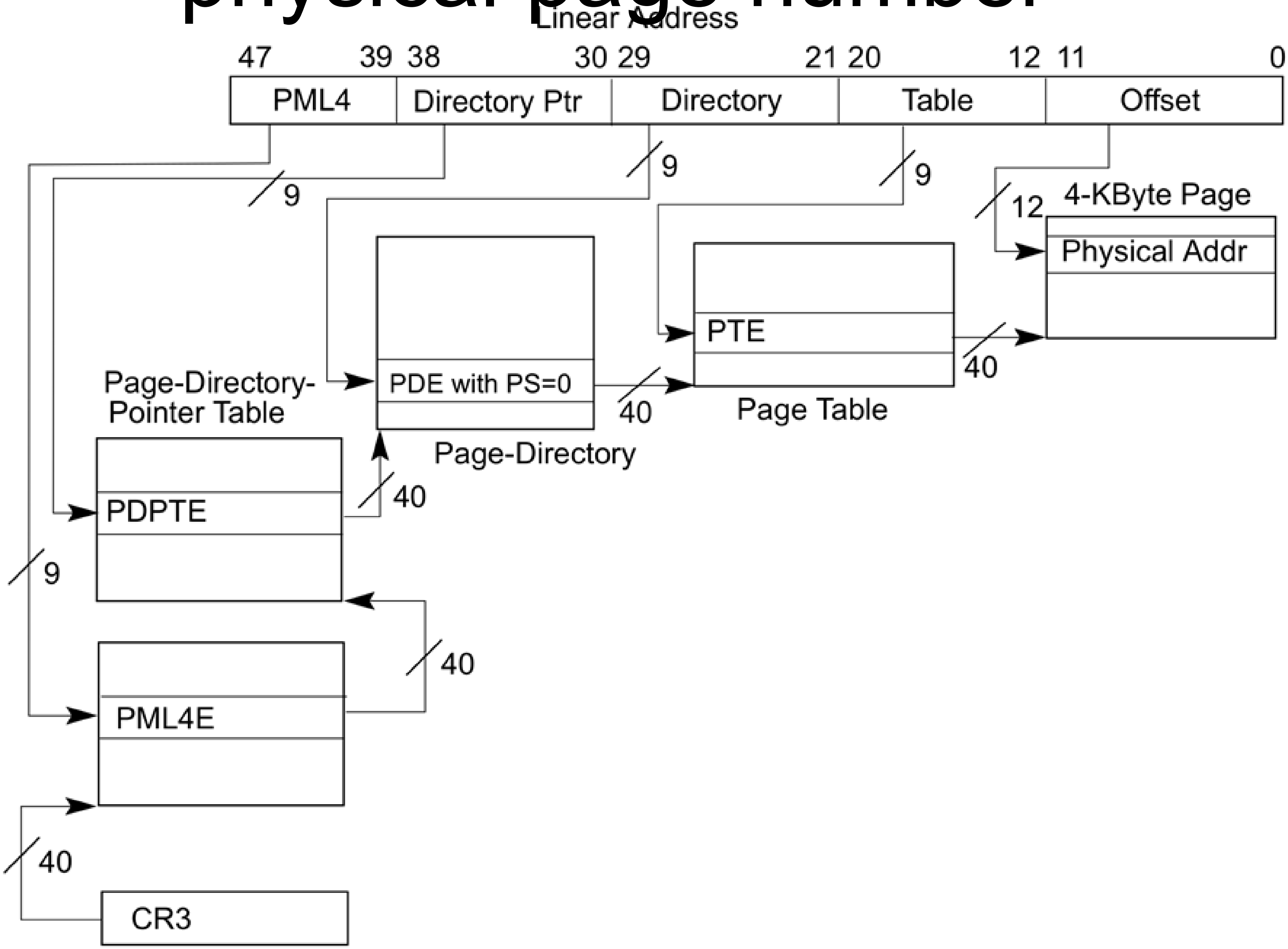
EIP: main
ESP: stack
TSS: 0x0
CR3: entrypml4

walkpgdir(): walk page table

```
walkpgdir(pde_t *pgdir, const void *va, int alloc)
{
    pde_t *pde;
    pte_t *pgtab;

    pde = &pgdir[PDX(va)];
    if(*pde & PTE_P){
        pgtab = (pte_t*)P2V(PTE_ADDR(*pde));
    } else {
        if(!alloc || (pgtab = (pte_t*)kalloc()) == 0)
            return 0;
        // Make sure all those PTE_P bits are zero.
        memset(pgtab, 0, PGSIZE);
...
        *pde = V2P(pgtab) | PTE_P | PTE_W | PTE_U;
    }
    return &pgtab[PTX(va)];
}
```

- If exists, get the address of the next level

PDE contains bits which represent physical page number



Getting level 2 page

```
1761    pgtab = (pte_t*)P2V(PTE_ADDR(*pde));
```

- We need two things
- Convert from 20 bits of physical page number to physical address of the page
- `PTE_ADDR(*pde)`
- Convert from physical address of that page to virtual address
- `P2V(…)`
  - We can’t access physical addresses directly
  - We can only access virtual addresses
  - Registers, mov instructions, etc. contain virtual addresses
  - Physical address have to be mapped by the current page table

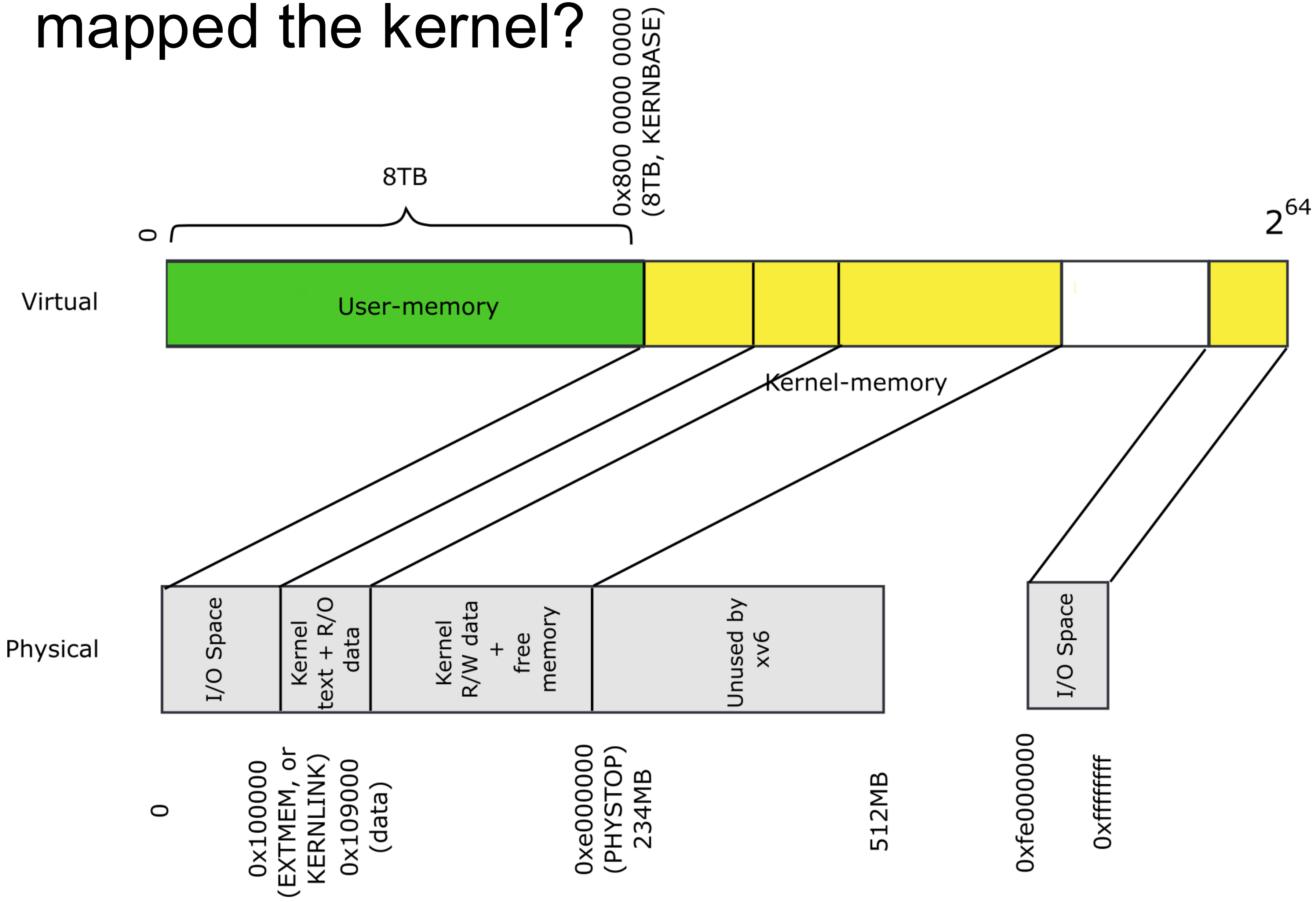
Step 1

- Convert from bits of physical page number to physical address of the page
- PTE_ADDR(*pde)
- This is trivial

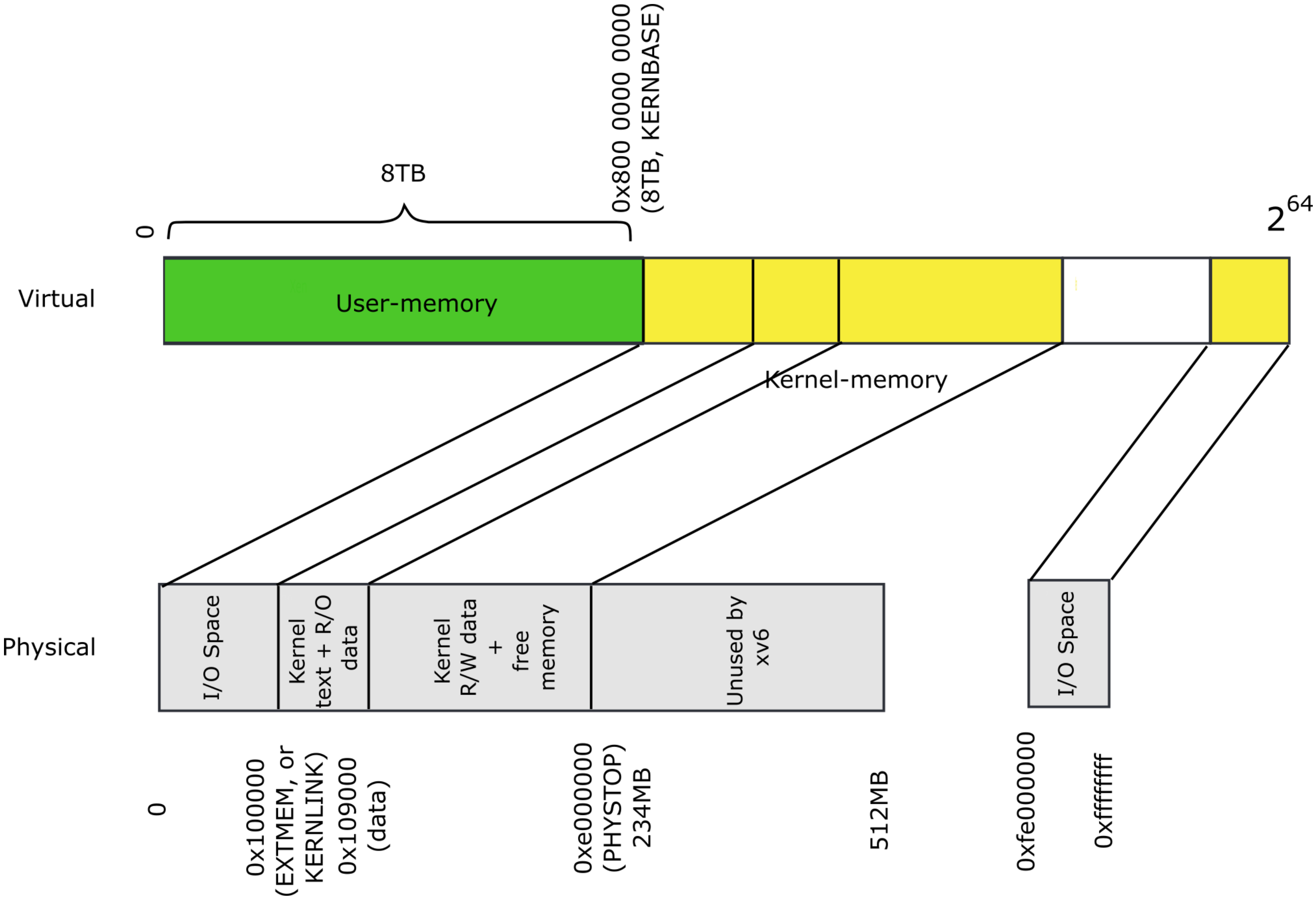
Step 2

- Convert from physical address of that page to virtual address
- P2V(...)
- This seems a bit tricky

Remember how we mapped the kernel?



```
0207 #define KERNBASE 0x80000000 // First kernel virtual address

0210 #define V2P(a) (((uint) (a)) − KERNBASE)

0211 #define P2V(a) (((void *) (a)) + KERNBASE)
```

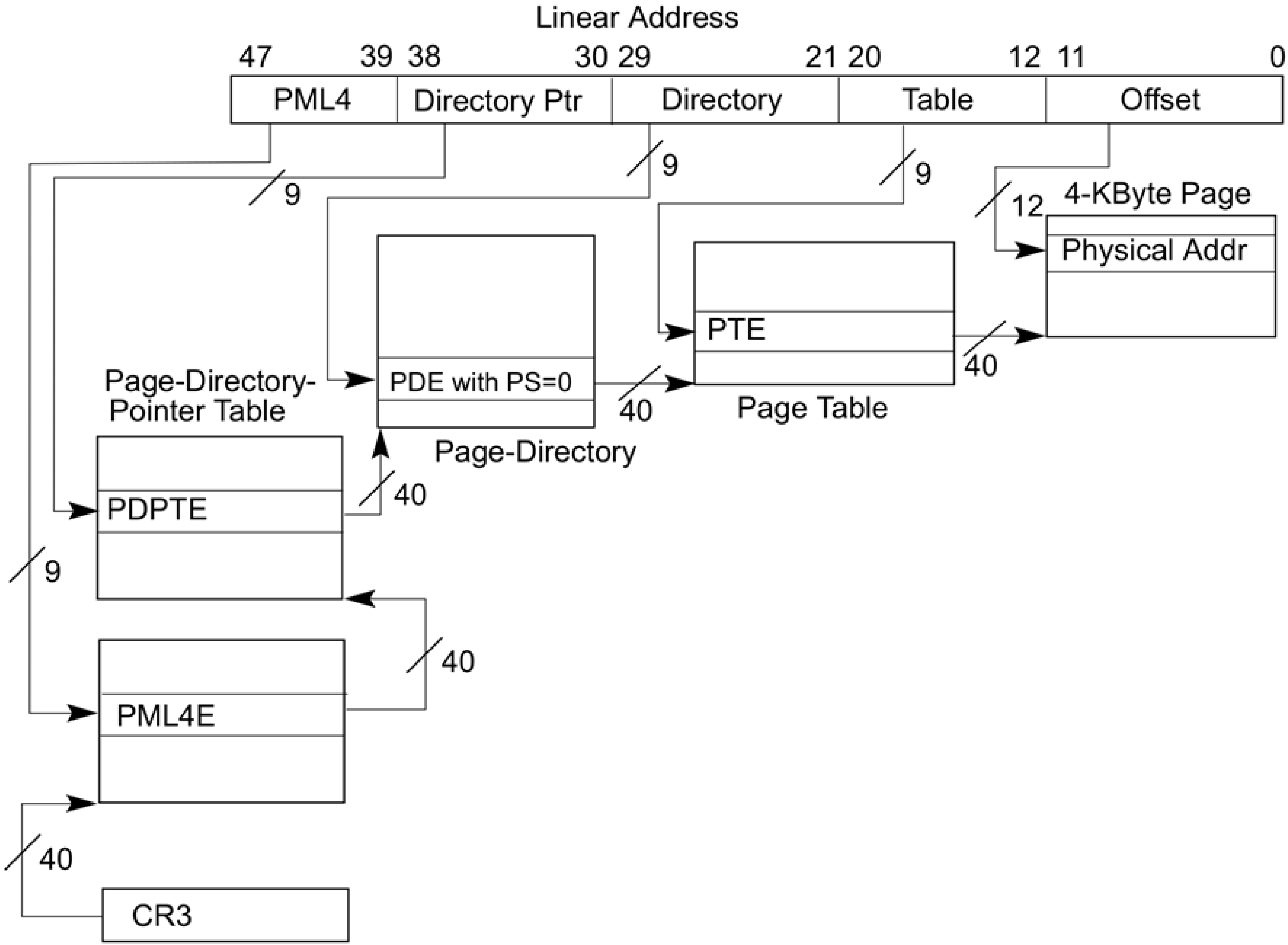


walkpgdir(): walk page table

```
walkpglevel(uint64 *pgdir, const void *va, int alloc, int level,
int *missing)
{
…

  if(*entry & PTE_P){
    next_pgdir = (uint64*)P2V(PTE_ADDR(*entry));
  } else {
    if(!alloc || (next_pgdir = (uint64*)kalloc()) == 0){
      if (missing)
        *missing = level;
        return 0;
      }
    // Make sure all those PTE_P bits are zero.
    memset(next_pgdir, 0, PGSIZE);
    // The permissions here are overly generous, but they can
    // be further restricted by the permissions in the page table
    // entries, if necessary.
    *entry = V2P(next_pgdir) | PTE_P | PTE_W | PTE_U;
  }
  return walkpglevel(next_pgdir, va, alloc, level - 1, missing);
}
```

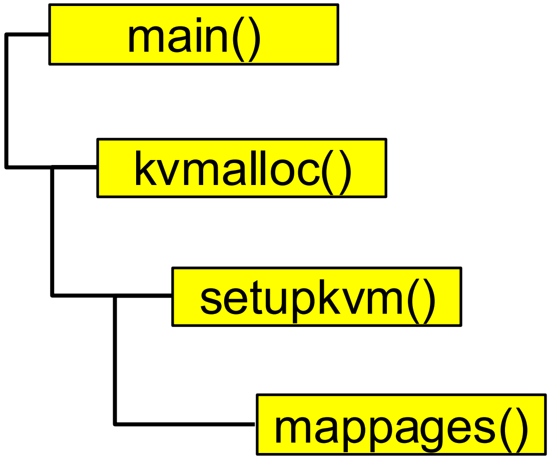
- Go to the next level

Return a pointer to PTE



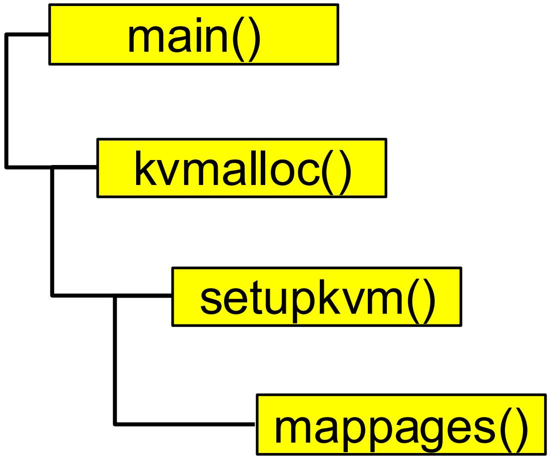
Back to mappages() function that maps a region of virtual memory into continuous region of physical memory

```
mappages(pml4e_t *pml4, void *va, uint size, uint64 pa, int perm)
{
  char *a, *last;
  pte_t *pte;

  a = (char*)PGROUNDDOWN((uint64)va);
  last = (char*)PGROUNDDOWN(((uint64)va) + size - 1);
  for(;;){
    if((pte = walkpml4(pml4, a, 1, 0)) == 0)
      return -1;
    if(*pte & PTE_P)
      panic("remap");
    *pte = pa | perm | PTE_P;
    if(a == last)
      break;
    a += PGSIZE;
    pa += PGSIZE;
  }
  return 0;
}
```



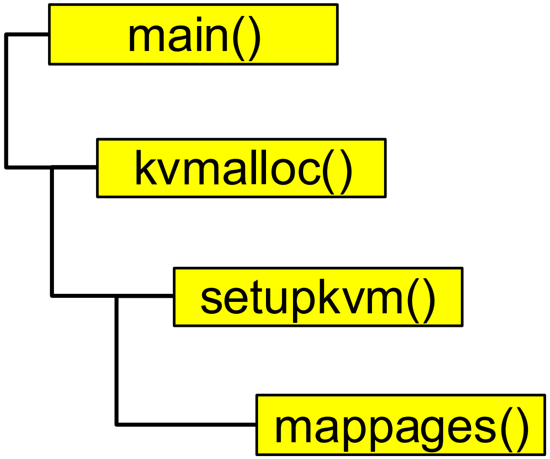
Remember we just discussed walkpml4()

```
mappages(pml4e_t *pml4, void *va, uint size, uint64 pa, int perm)
{
  char *a, *last;
  pte_t *pte;

  a = (char*)PGROUNDDOWN((uint64)va);
  last = (char*)PGROUNDDOWN(((uint64)va) + size - 1);
  for(;;){
    if((pte = walkpml4(pml4, a, 1, 0)) == 0)
      return -1;
    if(*pte & PTE_P)
      panic("remap");
    *pte = pa | perm | PTE_P;
    if(a == last)
      break;
    a += PGSIZE;
    pa += PGSIZE;
  }
  return 0;
}
```



Page present (PTE_P) – panic

```
mappages(pml4e_t *pml4, void *va, uint size, uint64 pa, int perm)
{
  char *a, *last;
  pte_t *pte;

  a = (char*)PGROUNDDOWN((uint64)va);
  last = (char*)PGROUNDDOWN(((uint64)va) + size - 1);
  for(;;){
    if((pte = walkpml4(pml4, a, 1, 0)) == 0)
      return -1;
    if(*pte & PTE_P)
      panic("remap");
    *pte = pa | perm | PTE_P;
    if(a == last)
      break;
    a += PGSIZE;
    pa += PGSIZE;
  }
  return 0;
}
```



main()

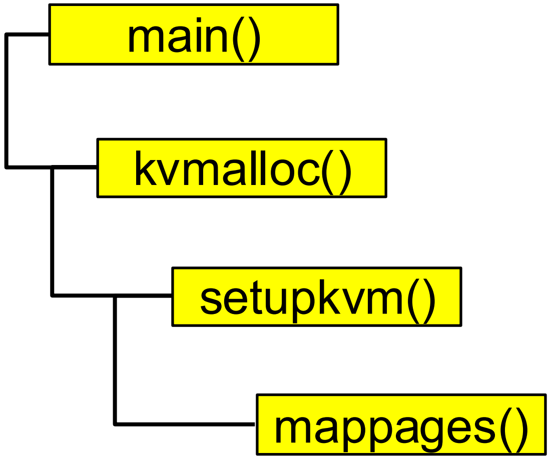
kvmalloc()

setupkvm()

mappages()

- Update page table entry
- Where does *pte point?
    - pa – physical address of the page

```
mappages(pml4e_t *pml4, void *va, uint size, uint64 pa, int perm)
{
  char *a, *last;
  pte_t *pte;

  a = (char*)PGROUNDDOWN((uint64)va);
  last = (char*)PGROUNDDOWN(((uint64)va) + size - 1);
  for(;;){
    if((pte = walkpml4(pml4, a, 1, 0)) == 0)
      return -1;
    if(*pte & PTE_P)
      panic("remap");
    *pte = pa | perm | PTE_P;
    if(a == last)
      break;
    a += PGSIZE;
    pa += PGSIZE;
  }
  return 0;
}
```



main()

kvmalloc()

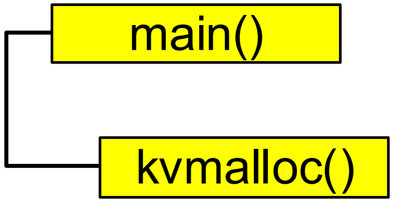
setupkvm()

mappages()

- Move to the next page

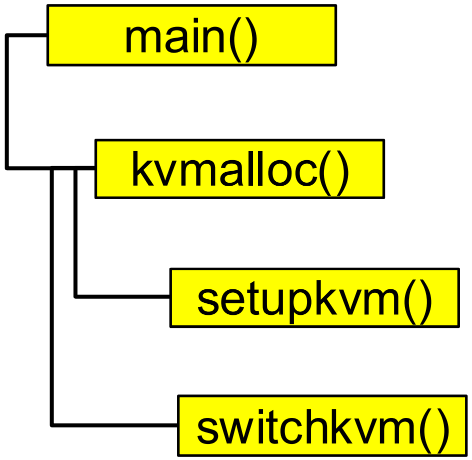
kvmalloc()

```
// Allocate one page table for the machine for the kernel address
// space for scheduler processes.
void
kvmalloc(void)
{
  kpml4 = setupkvm();
  switchkvm();
}
```



Switch to the new page table

```
// Switch h/w page table register to the kernel-only page table,
// for when no process is running.
void
switchkvm(void)
{
  lcr3(V2P(kpml4)); // switch to the kernel page table
}
```



Recap

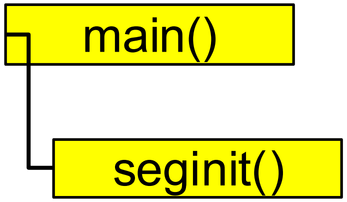
- Kernel has a memory allocator
- Kernel has a its own address space
- It uses 4KB page tables
- It is ready to create processes

```
int
main(void)
{
  void *kinit1_end = (1024*1024*1024 > PHYSTOP) ? P2V(PHYSTOP)
                                                : P2V(1024*1024*1024);
  kinit1(end, kinit1_end); // phys page allocator
  kvmalloc(); // kernel page table
  mpinit(); // detect other processors
  lapicinit(); // interrupt controller
  seginit(); // segment descriptors
  picinit(); // disable pic
  ioapicinit(); // another interrupt controller
...
```

main()

Initialize GDT

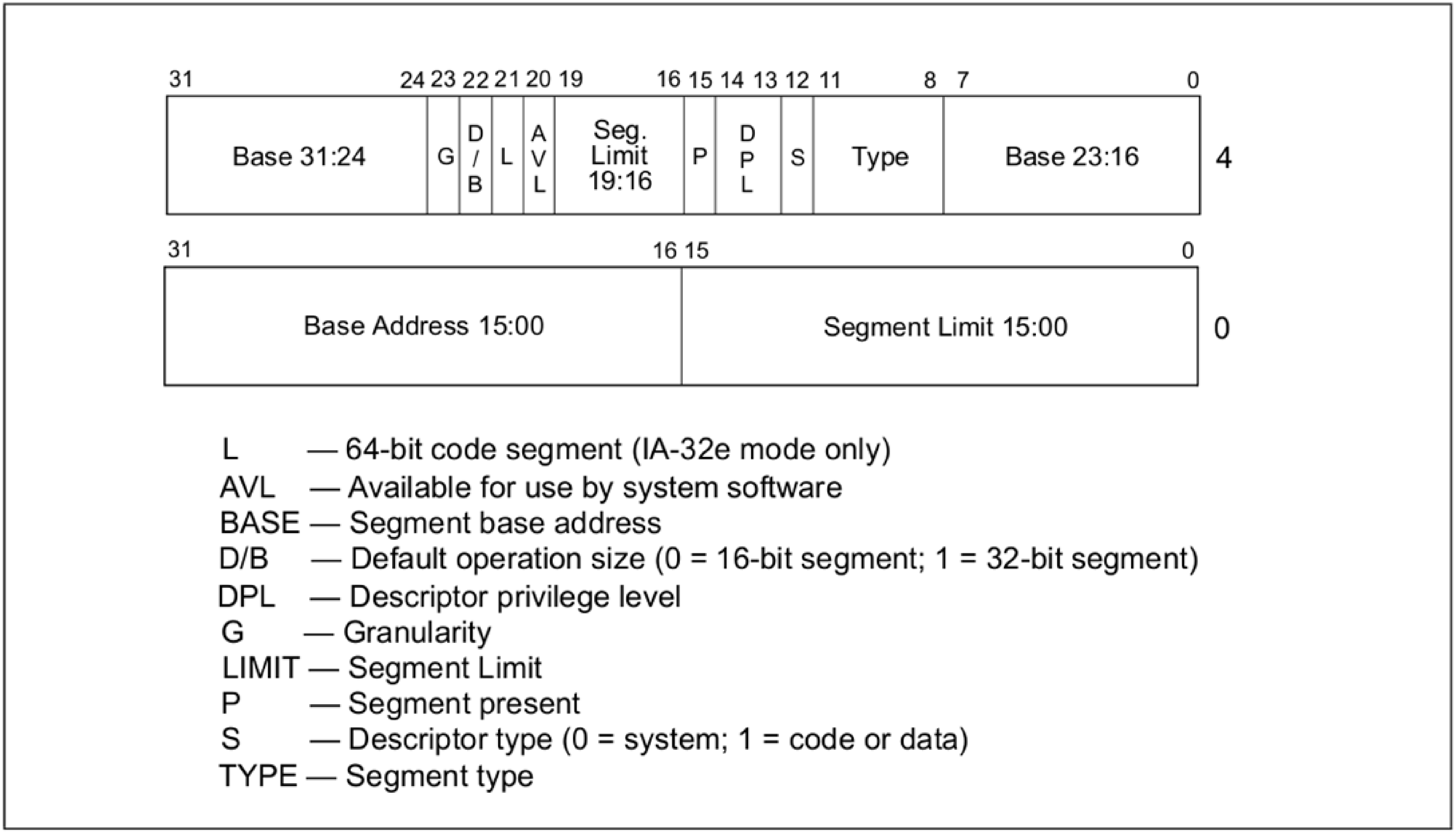
```
// Set up CPU’s kernel segment descriptors.
// Run once on entry on each CPU.
void
seginit(void)
{
  struct cpu *c;

  // Map "logical" addresses to virtual addresses using identity map.
  // Cannot share a CODE descriptor for both kernel and user
  // because it would have to have DPL_USR, but the CPU forbids
  // an interrupt from CPL=0 to DPL=3.
  c = &cpus[cpuid()];
  c–>gdt[SEG_KCODE] = SEG(STA_X|STA_R, 0, 0xffffffff, 0);
  c–>gdt[SEG_KDATA] = SEG(STA_W, 0, 0xffffffff, 0);
  c–>gdt[SEG_UCODE] = SEG(STA_X|STA_R, 0, 0xffffffff, DPL_USER);
  c–>gdt[SEG_UDATA] = SEG(STA_W, 0, 0xffffffff, DPL_USER);
  lgdt(c–>gdt, sizeof(c–>gdt));
}
```



Struct CPU

```
// Per−CPU state
struct cpu {
  uchar apicid;                // Local APIC ID
  struct context *scheduler;   // swtch() here to enter scheduler
  struct taskstate ts;         // Used by x86 to find stack for interrupt
  struct segdesc gdt[NSEGS];   // x86 global descriptor table
  volatile uint started;       // Has the CPU started?
  int ncli;                    // Depth of pushcli nesting.
  int intena;                  // Were interrupts enabled before pushcli?
  struct proc *proc;           // The process running on this cpu or null
};

extern struct cpu cpus[NCPU];
```

Segment descriptor



31 24	23	22	21	20	19 16	15	14 13	12	11 8	7 0	
Base 31:24	G	D/B	L	AVL	Seg. Limit 19:16	P	DPL	S	Type	Base 23:16	4

31 16	15 0	
Base Address 15:00	Segment Limit 15:00	0

L — 64-bit code segment (IA-32e mode only)
AVL — Available for use by system software
BASE — Segment base address
D/B — Default operation size (0 = 16-bit segment; 1 = 32-bit segment)
DPL — Descriptor privilege level
G — Granularity
LIMIT — Segment Limit
P — Segment present
S — Descriptor type (0 = system; 1 = code or data)
TYPE — Segment type

Segment Descriptor

```
// Segment Descriptor
struct segdesc {
  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
};
```

Thank you!

(Next time: interrupts!)