Operating Systems

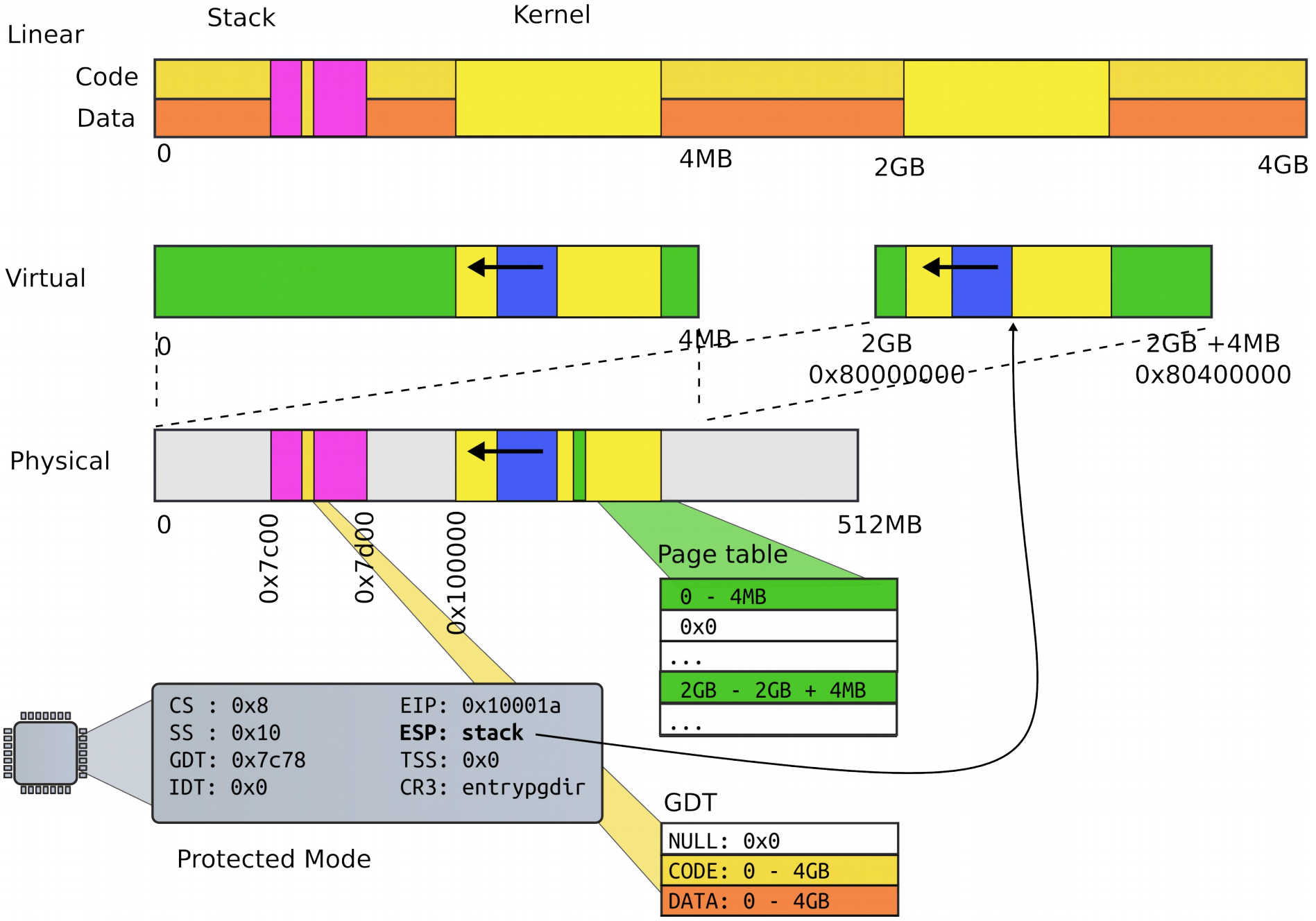
Lecture: Kernel Memory Allocator and Page Table

Anton Burtsev
November, 2021

Recap of the boot sequence

- Setup segments (data and code)
- Switched to protected mode
  - Loaded GDT (segmentation is on)
- Setup stack (to call C functions)
- Loaded kernel from disk
- Setup first page table
  - 2 entries [0 : 4MB] and [2GB : (2GB + 4MB)]
- Setup high-address stack
- Jumped to main()

State of the system after boot

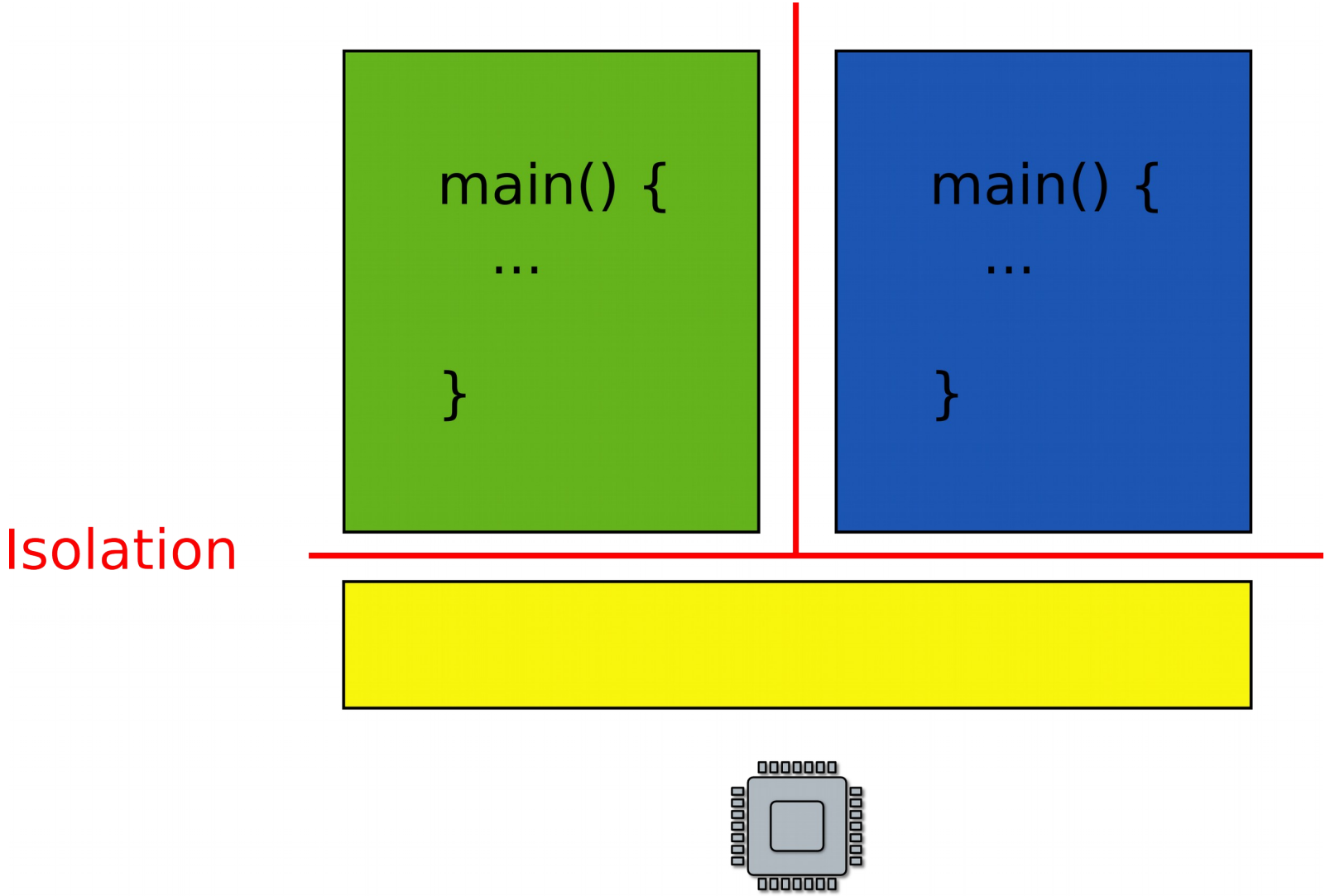


Running in main()

```
1313 // Bootstrap processor starts running C code here.
1314 // Allocate a real stack and switch to it, first
1315 // doing some setup required for memory allocator to work.
1316 int
1317 main(void)
1318 {
1319     kinit1(end, P2V(4*1024*1024)); // phys page allocator
1320     kvmalloc(); // kernel page table
1321     mpinit(); // detect other processors
1322     lapicinit(); // interrupt controller
1323     seginit(); // segment descriptors
1324     cprintf("\ncpu%d: starting xv6\n\n", cpunum());
...
1340 }
```

What's next?

We want to run multiple programs (processes)



But what is a process?

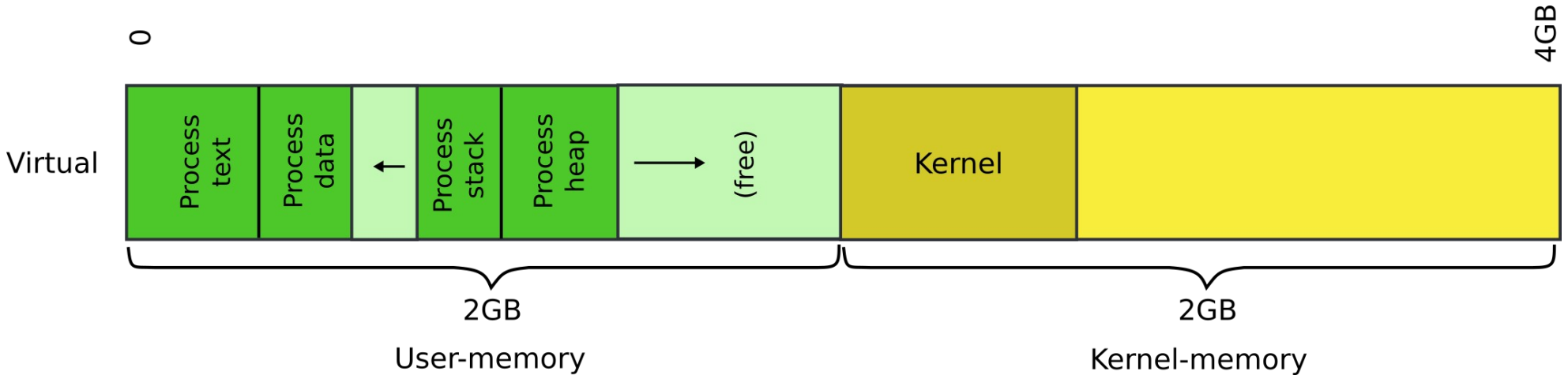
A couple of requirements

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

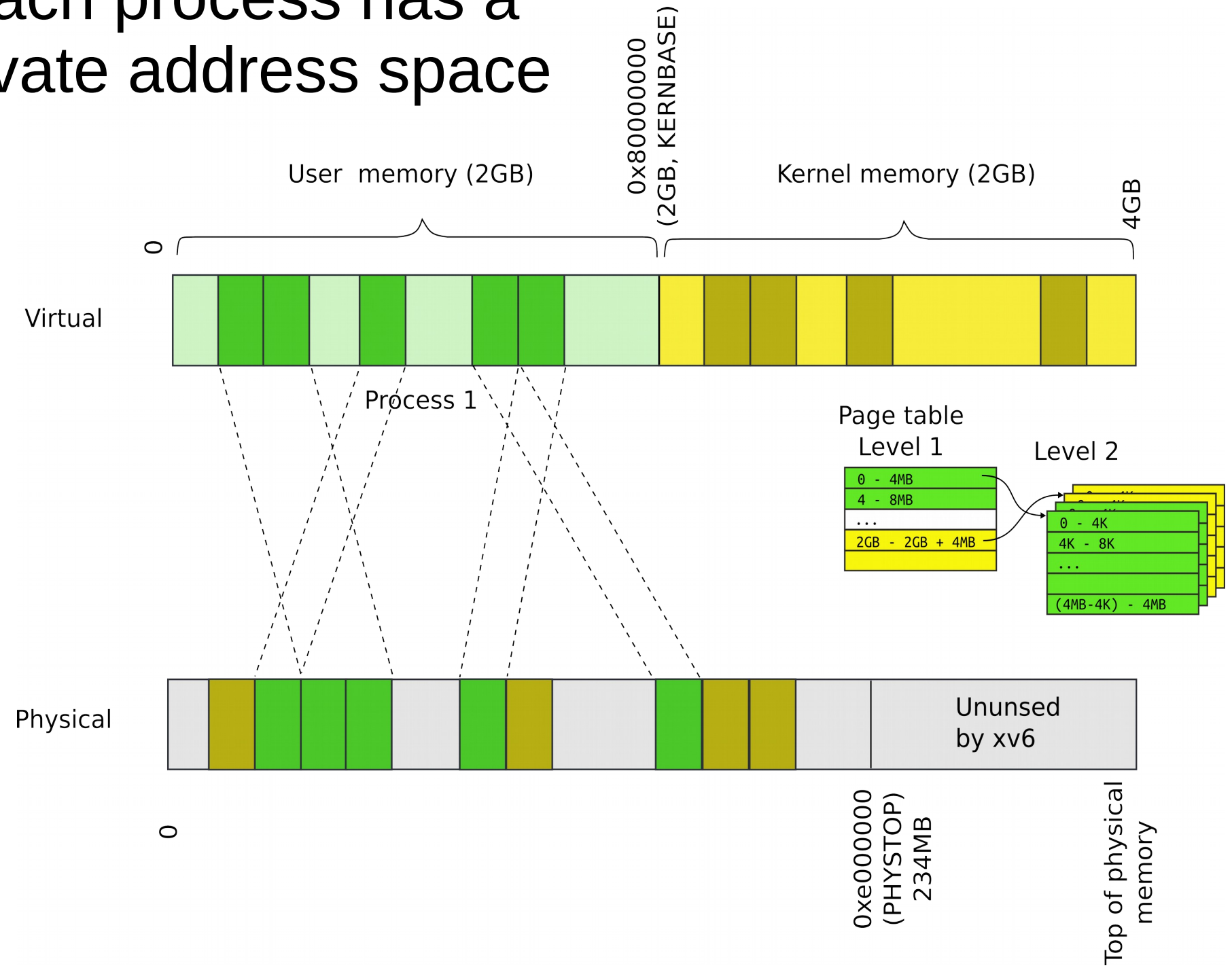
A couple of requirements

- 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 a 2GB/2GB private address space

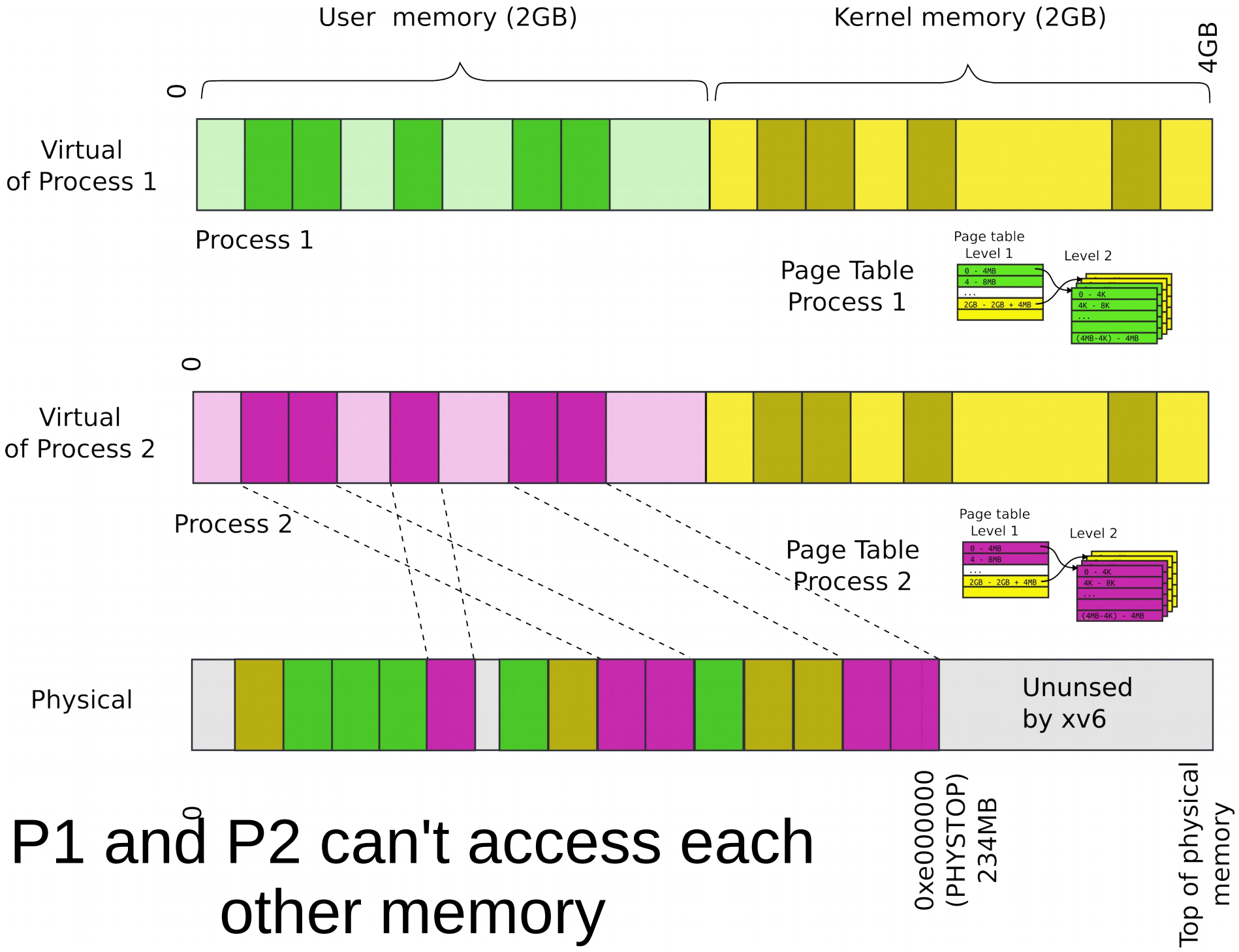


Each process has a private address space



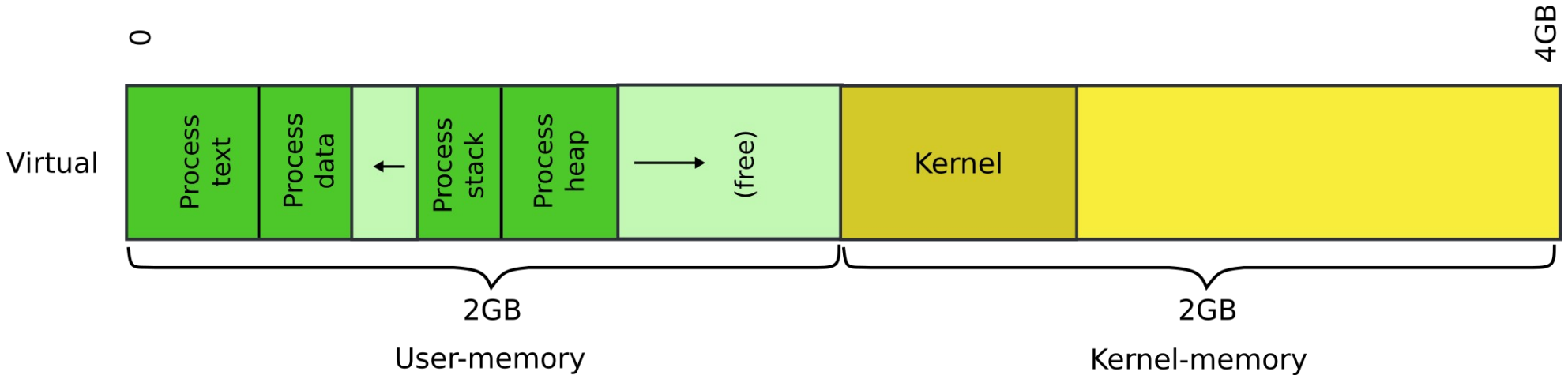
Each process maps the kernel

- 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!**

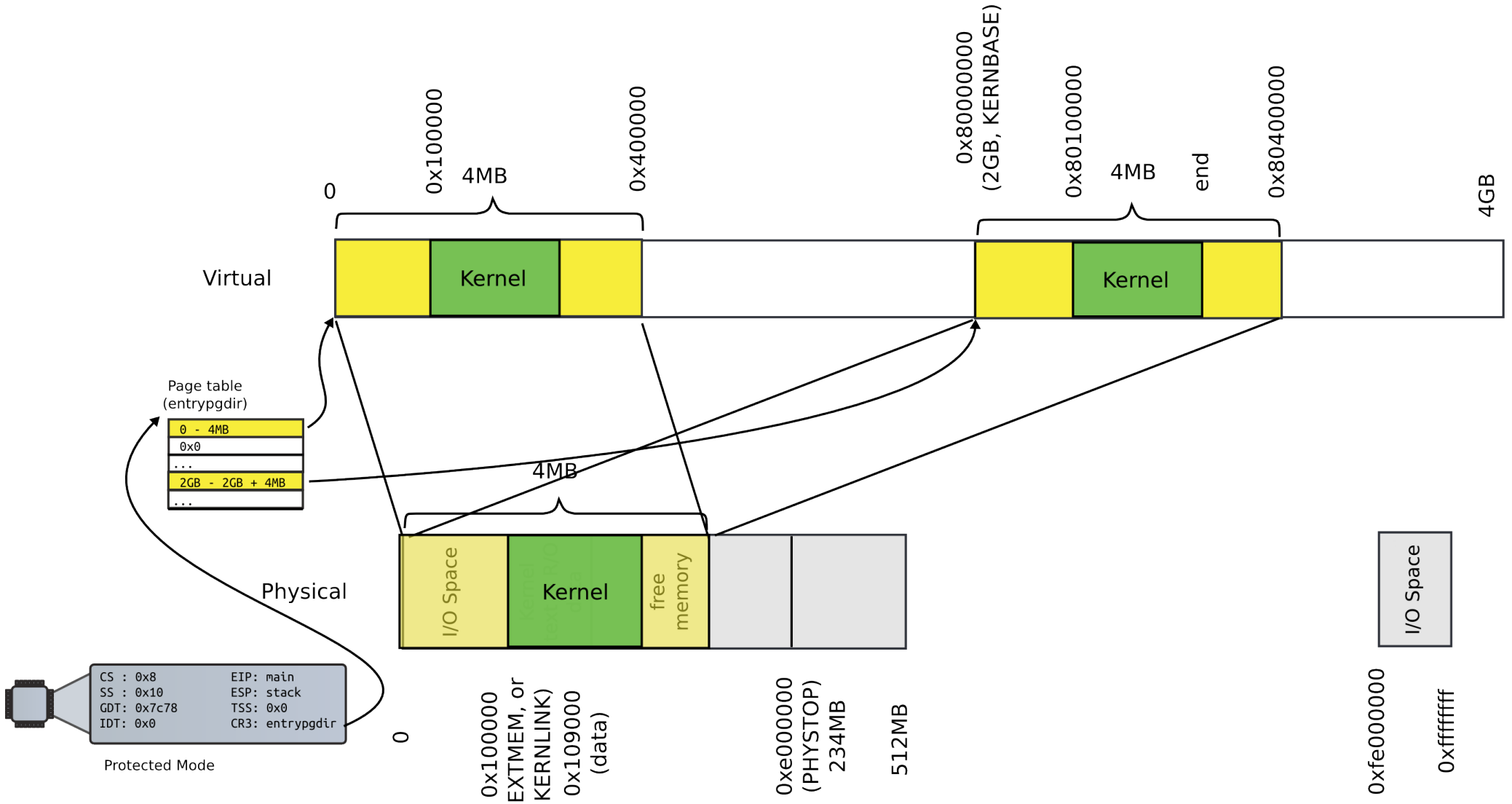


P1 and P2 can't access each other memory

Our goal: 2GB/2GB address space



Memory after boot



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 2 level, 4KB page table
  - Right now we have
    - One (statically allocated) page table
    - That has only two entries
  - And it is a page table for 4MB pages
- 4KB page table is a better choice
  - Xv6 processes are small
  - Wasting 4MB on a program that fits into 1KB is absurd

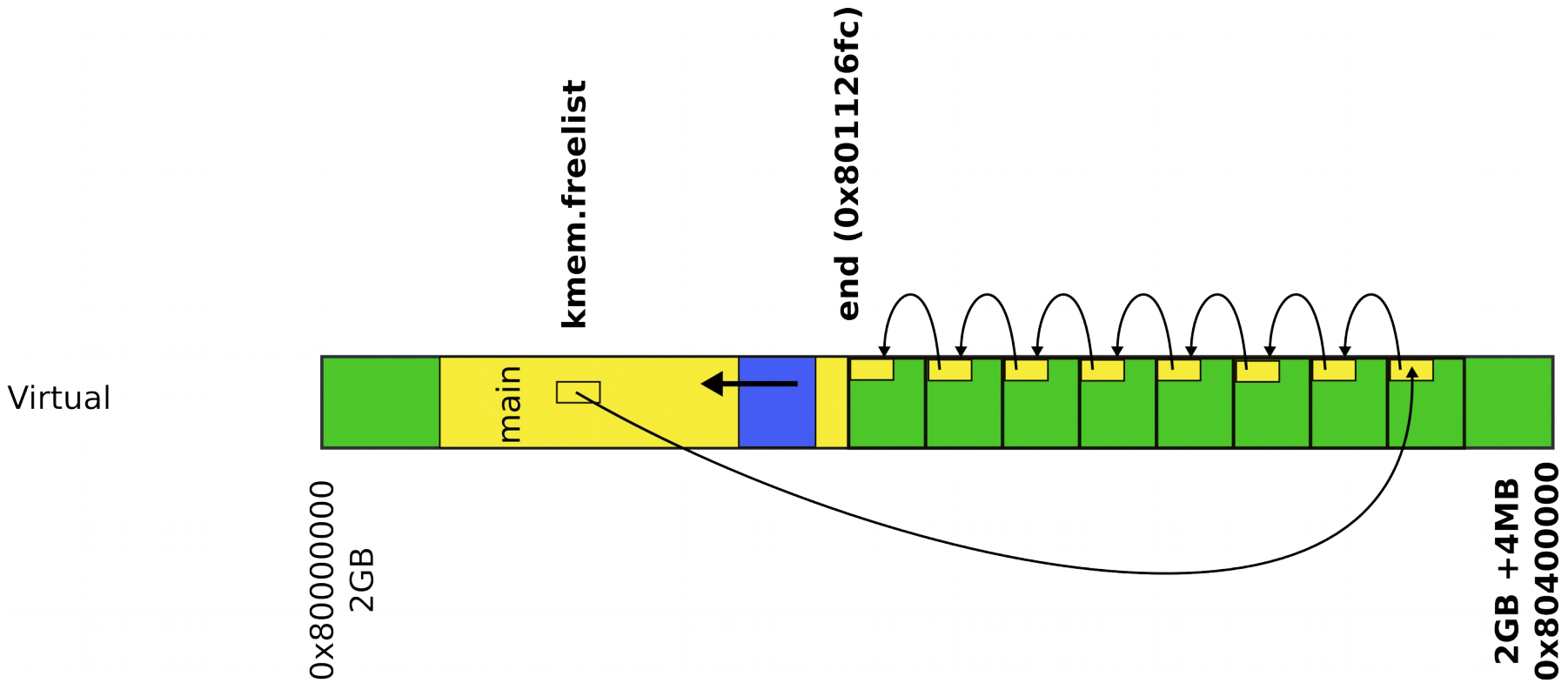
- But to create page tables we need memory
  - Where can it come from?

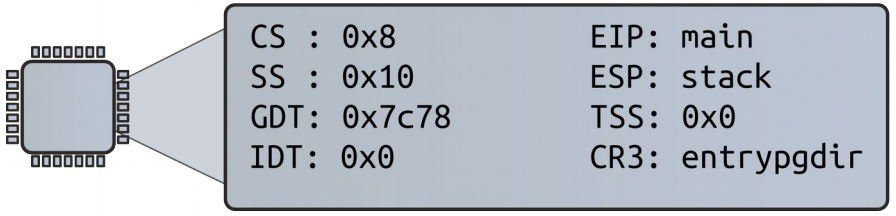
Simple memory allocator

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

What can it look like?

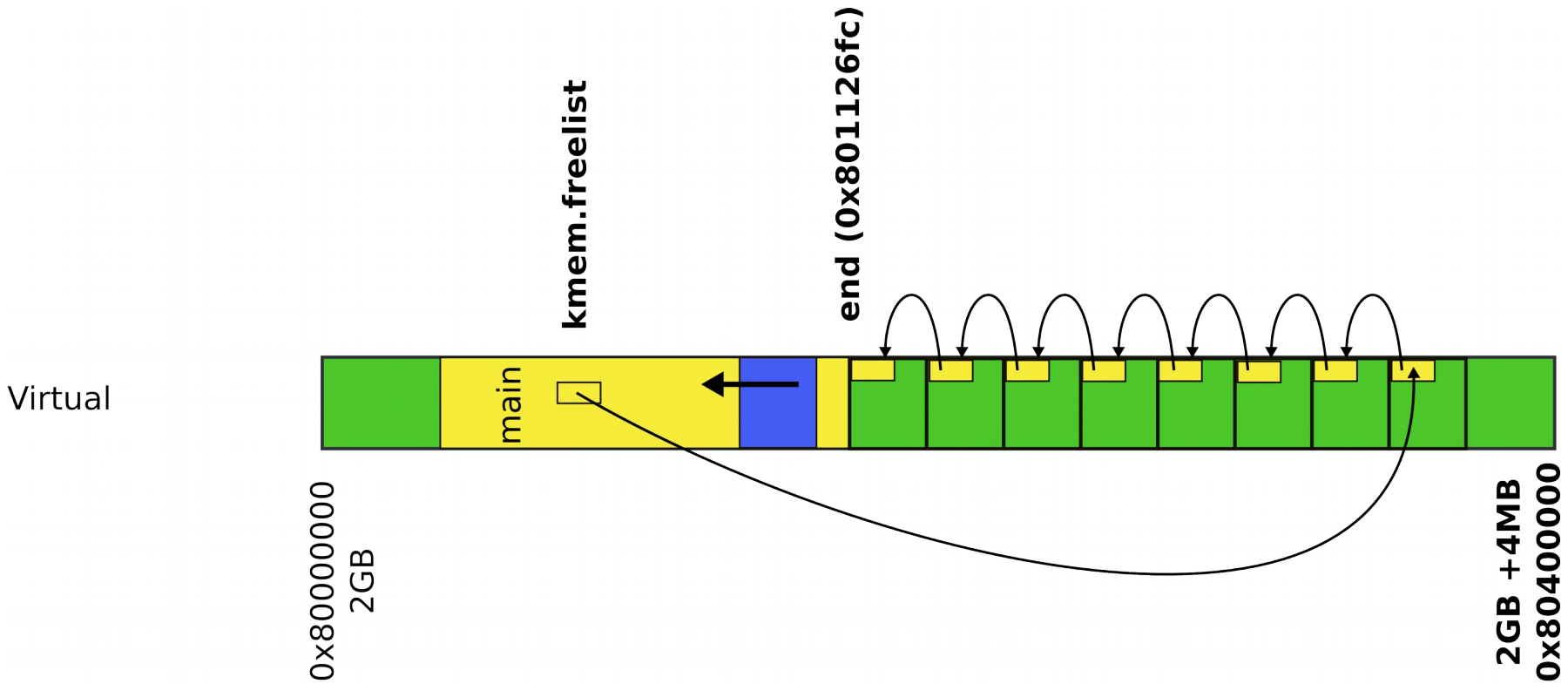
Page allocator

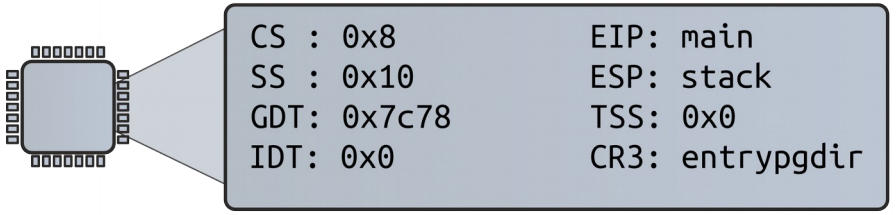




Protected Mode

Page allocator

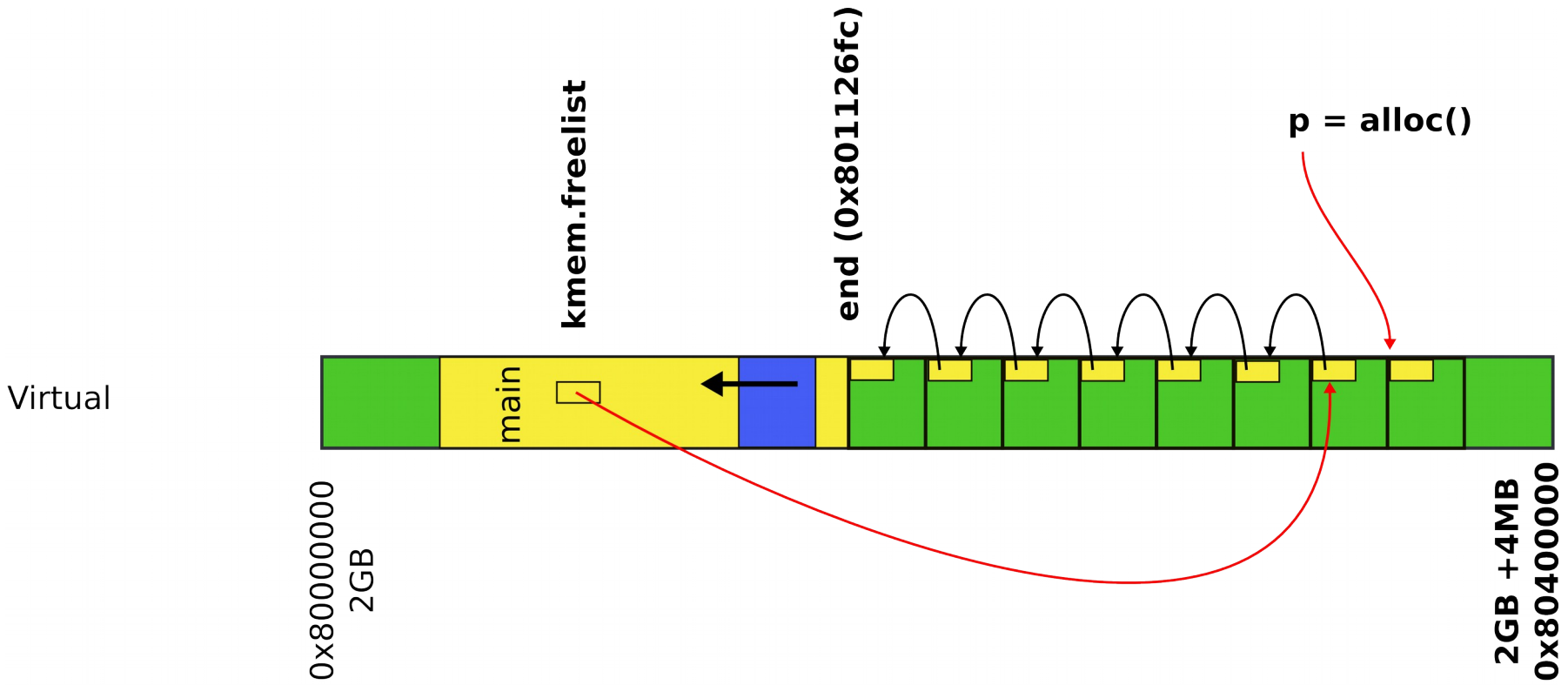


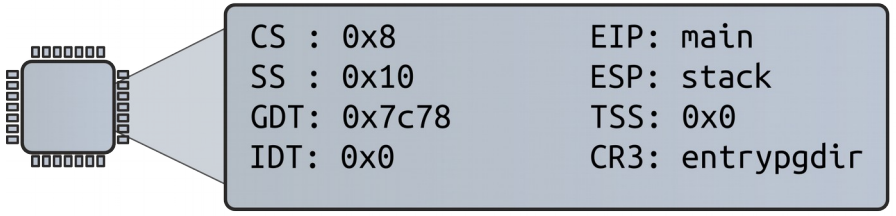


Protected Mode

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

Page allocator

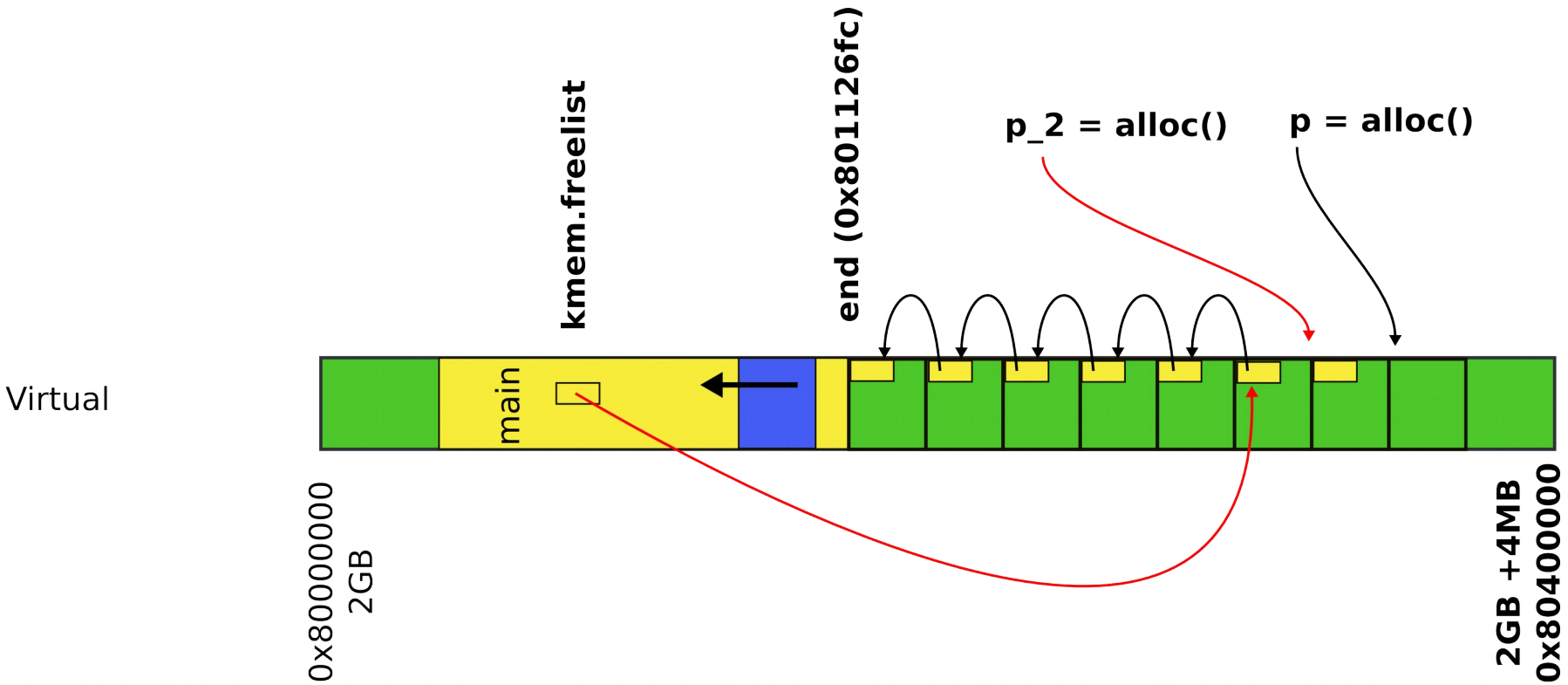


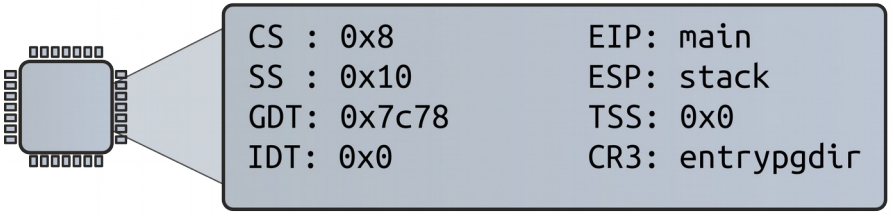


Protected Mode

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

Page allocator

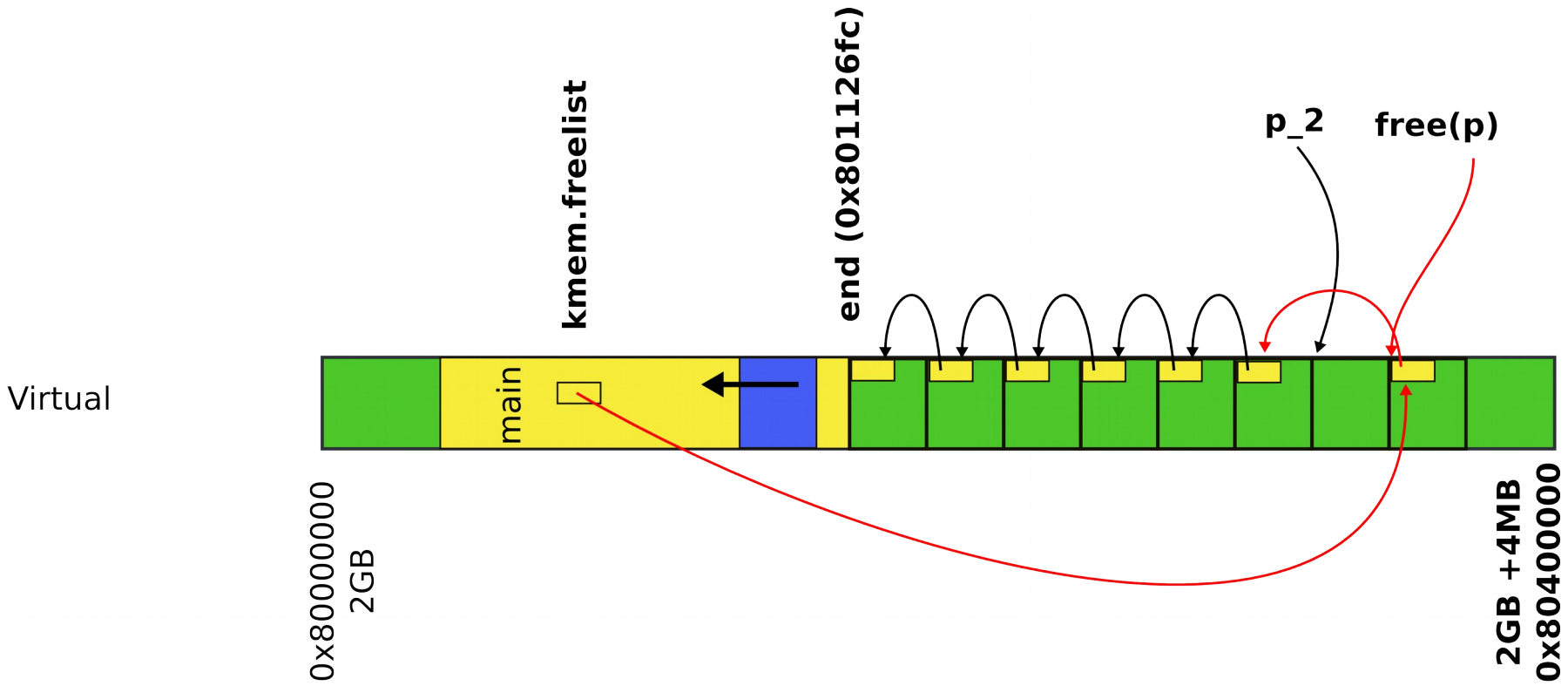


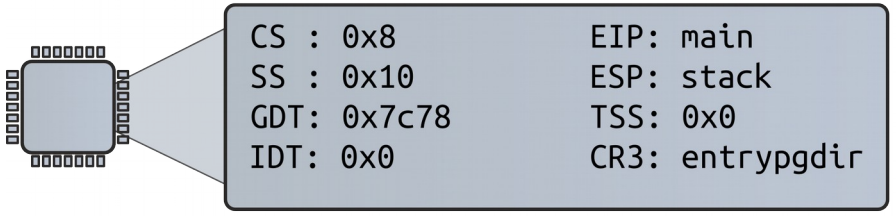


Protected Mode

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

Page allocator



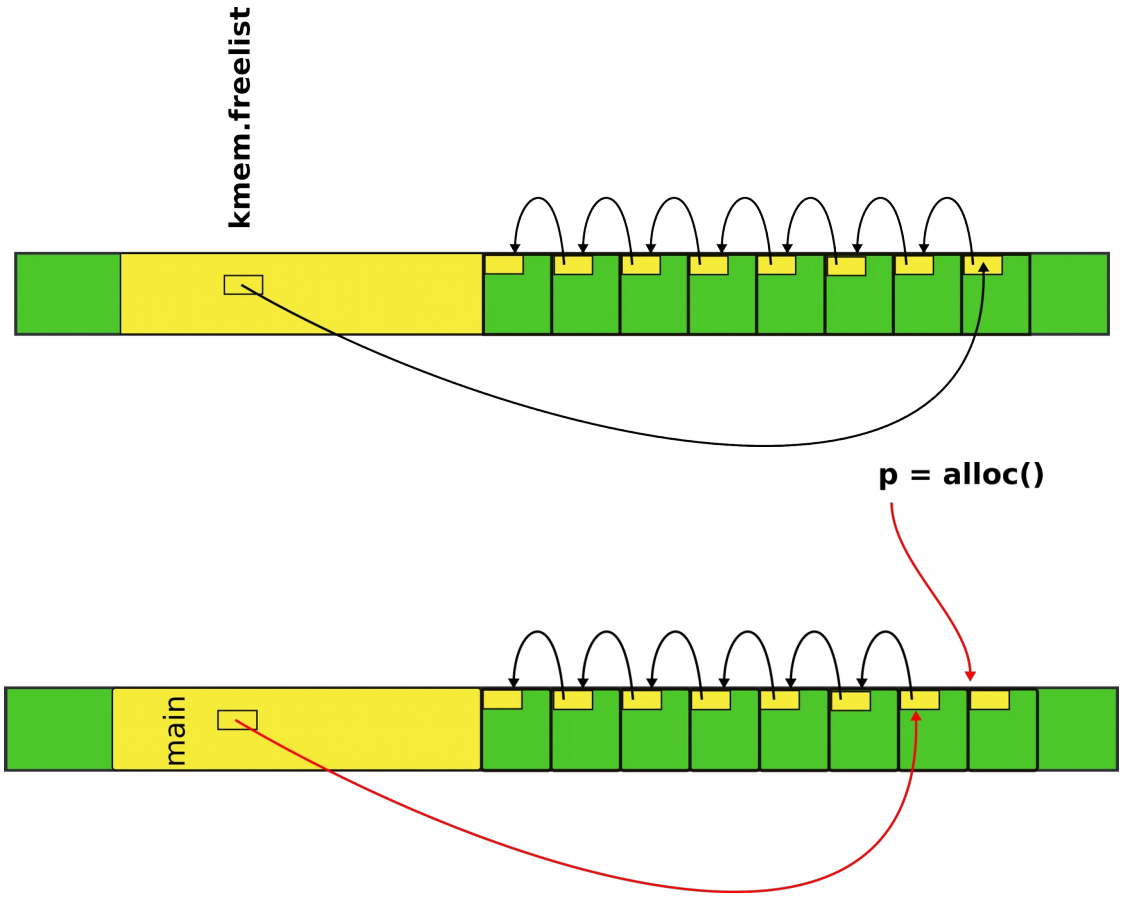


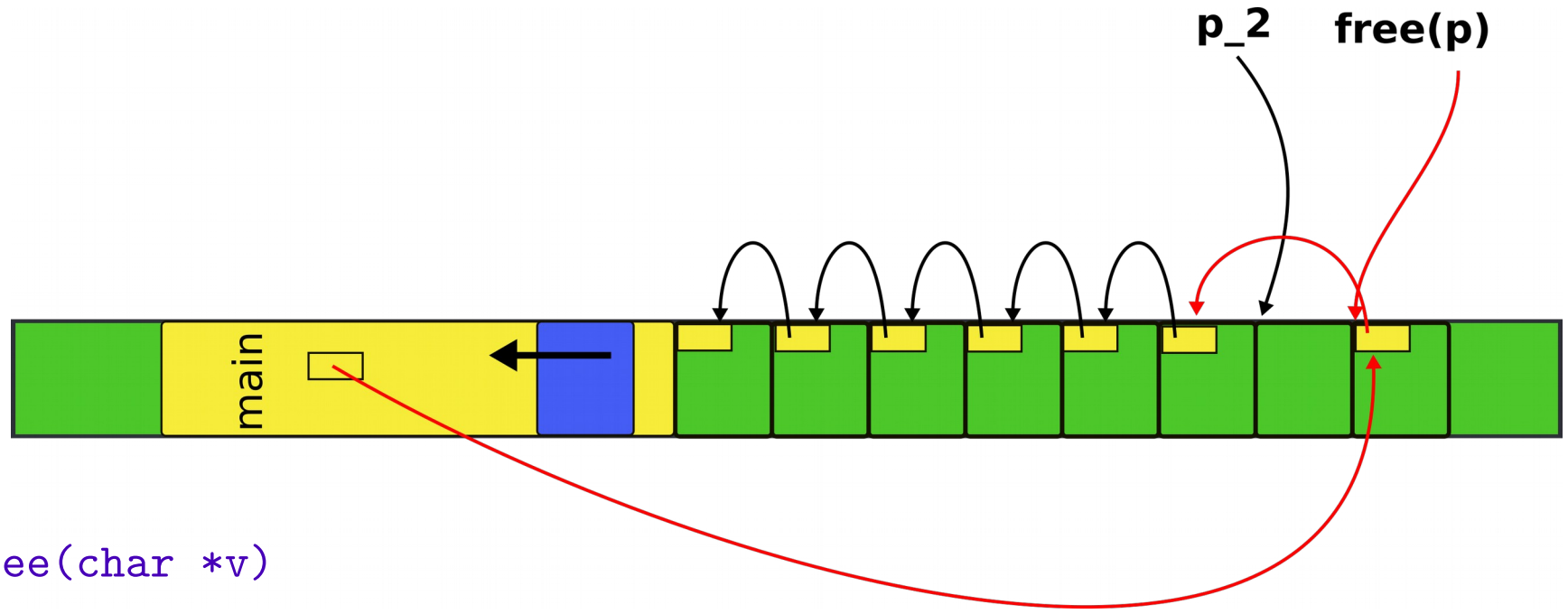
Protected Mode

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

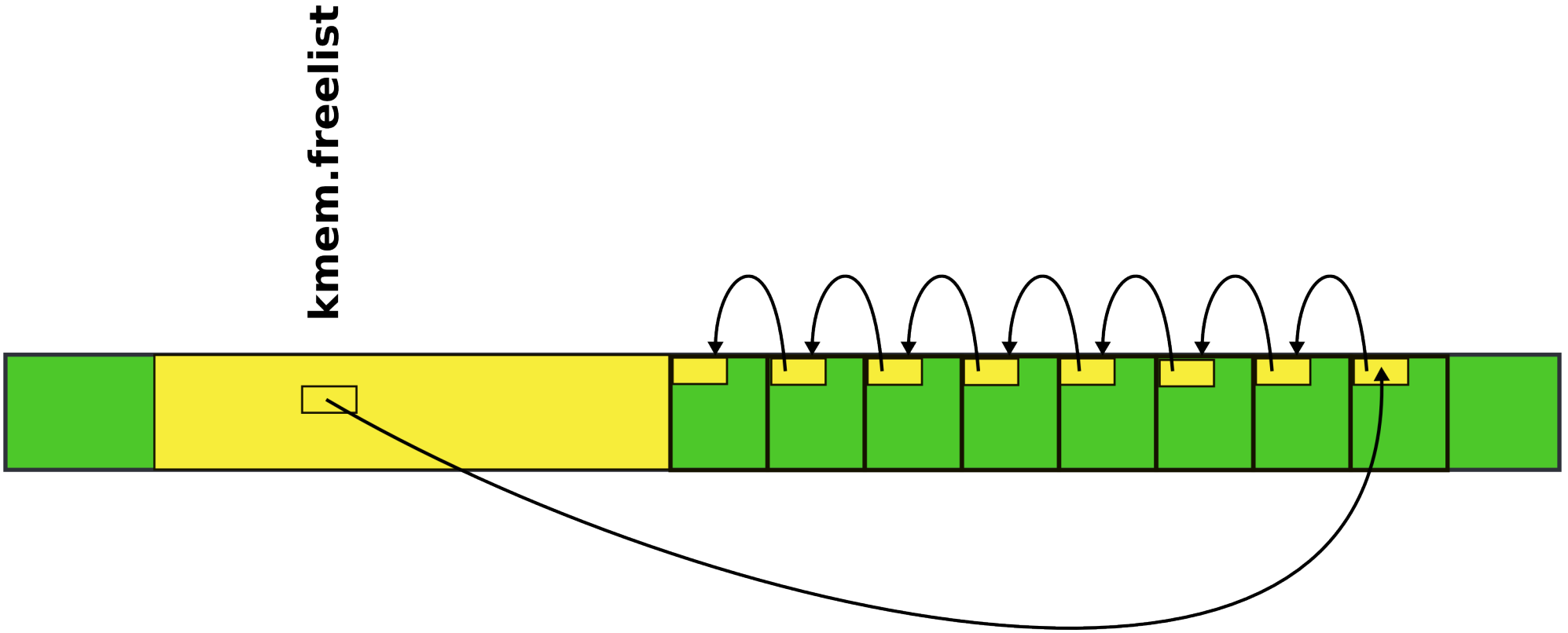
kalloc() - kernel allocator

```
3087 char*
3088 kalloc(void)
3089 {
3080   struct run *r;
...
3094   r = kmem.freelist;
3095   if(r)
3096     kmem.freelist = r->next;
…
3099   return (char*)r;
3099 }
```



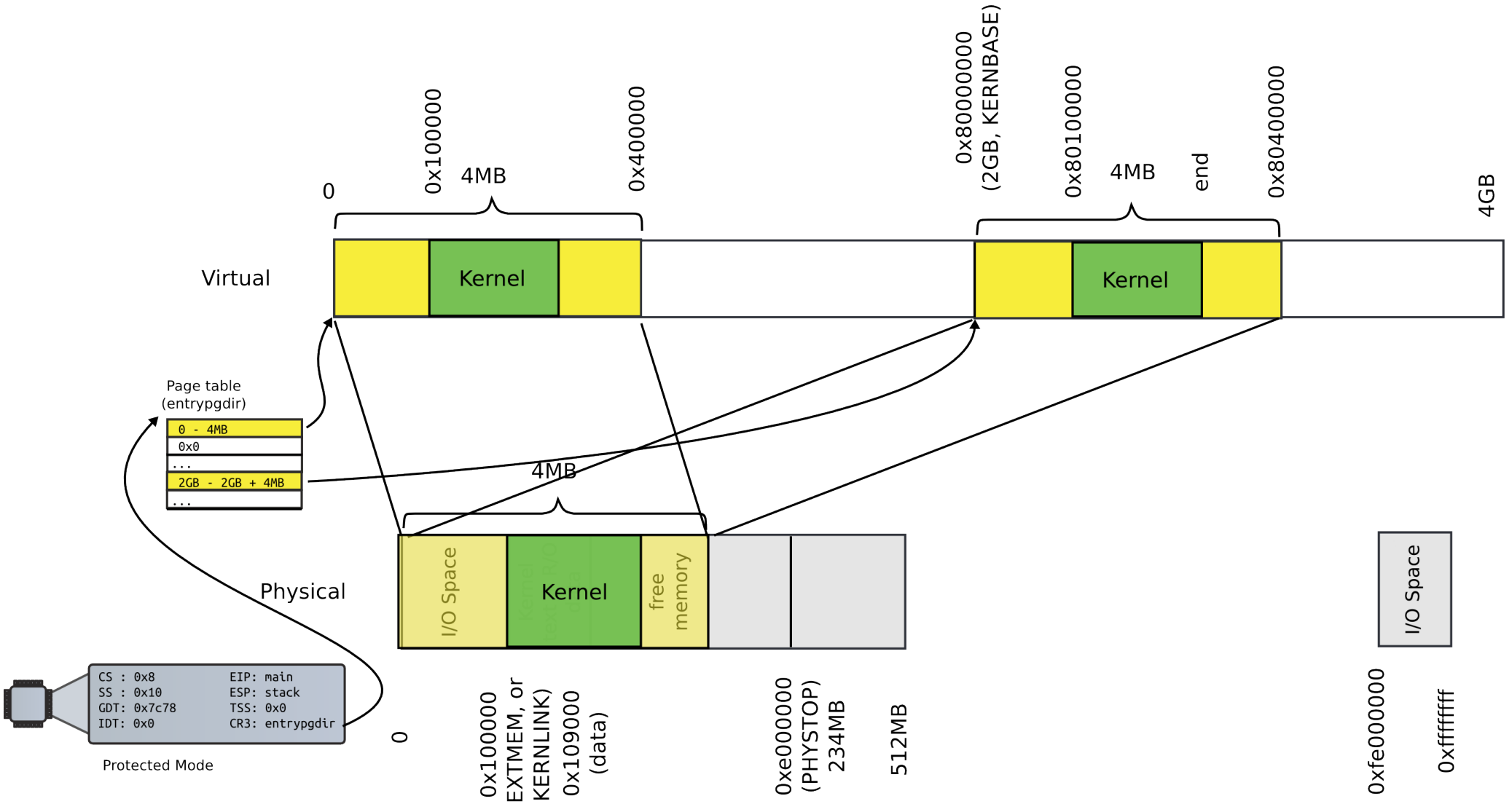



```
3065 kfree(char *v)
3066 {
3067   struct run *r;
...
3077   r = (struct run*)v;
3078   r->next = kmem.freelist;
3079   kmem.freelist = r;
...
2832 }
```

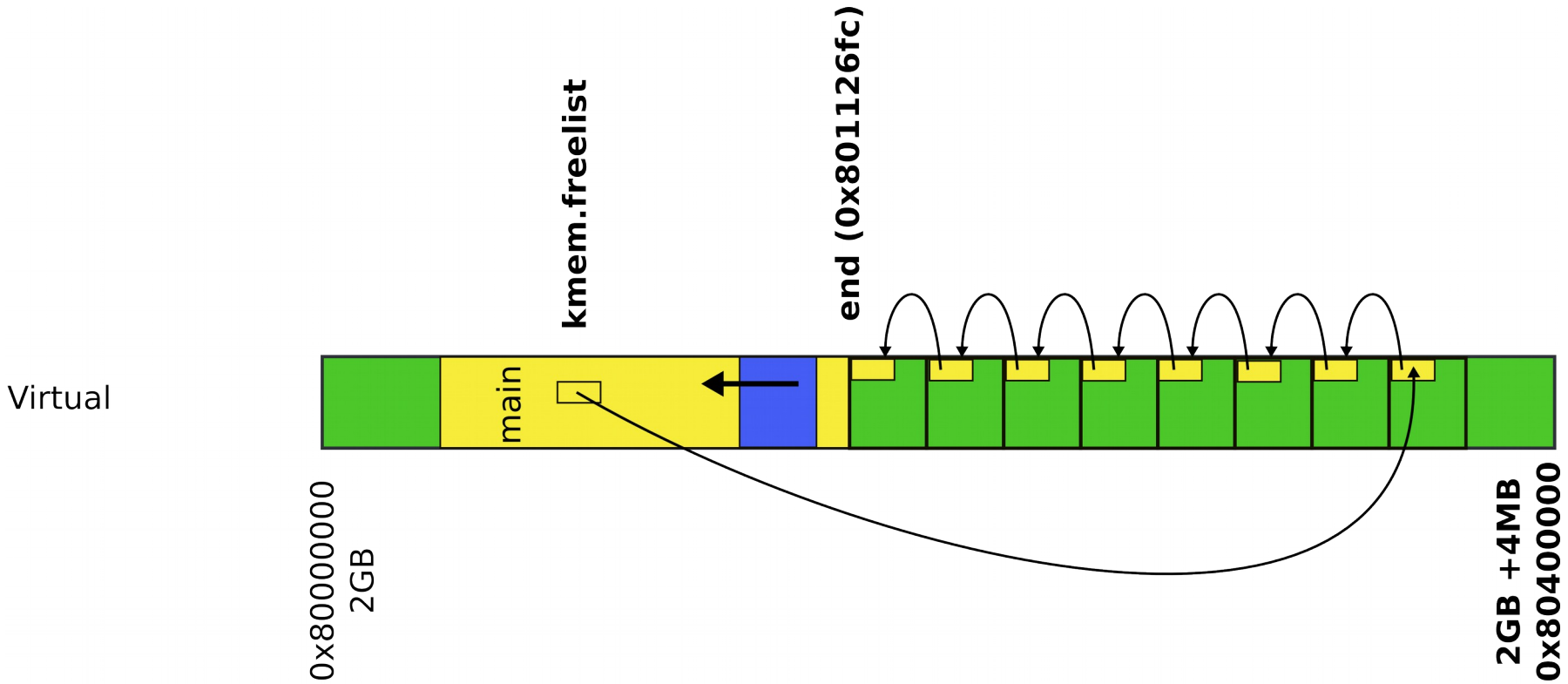


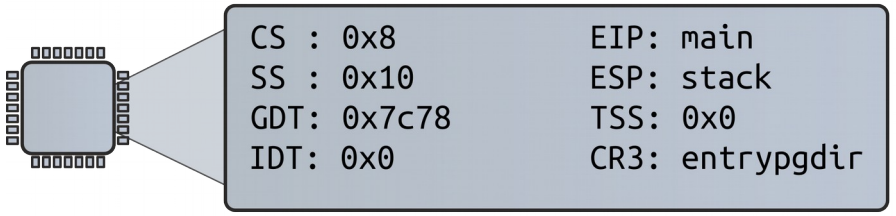
- Where can we get memory to keep the list itself?
  - Notice, the list is allocated within the pages
  - It has to write each page though to update the “next” pointer

There is a bit of free memory in the 4MB page we've mapped



Donate this free memory to the allocator





Protected Mode

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

kinit1(): donate free memory

```
1316 int
1317 main(void)
1318 {
1319     kinit1(end, P2V(4*1024*1024)); // phys page allocator
1320     kvmalloc(); // kernel page table
1321     mpinit(); // detect other processors
1322     lapicinit(); // interrupt controller
1323     seginit(); // segment descriptors
1324     cprintf("\ncpu%d: starting xv6\n\n", cpunum());
1325     picinit(); // another interrupt controller
1326     ioapicinit(); // another interrupt controller
1327     consoleinit(); // console hardware
1328     uartinit(); // serial port
...
1340 }
```

```
3030 kinit1(void *vstart, void *vend)
3031 {
...
3034   freerange(vstart, vend);
3035 }
```

Freerange()

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

freerange()

```
3051 freerange(void *vstart, void *vend)
3052 {
3053   char *p;
3054   p = (char*)PGROUNDUP((uint)vstart);
3055   for(; p + PGSIZE <= (char*)vend; p += PGSIZE)
3056     kfree(p);
3057 }
```

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

Where do we start?

```
1316 int
1317 main(void)
1318 {
1319     kinit1(end, P2V(4*1024*1024)); // phys page allocator
1320     kvmalloc(); // kernel page table
1321     mpinit(); // detect other processors
```

- What is this **`end`**?

```
1311 extern char end[];
```

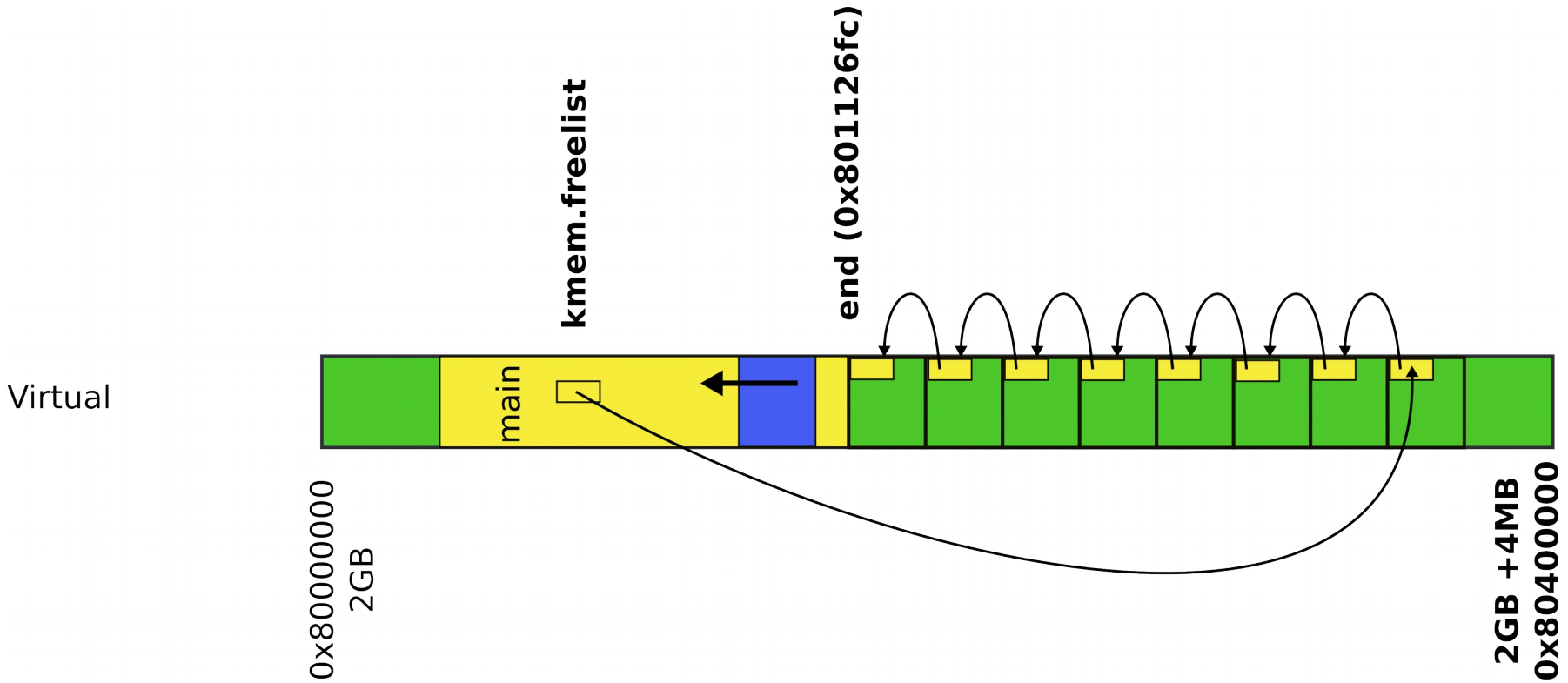
Where do we start?

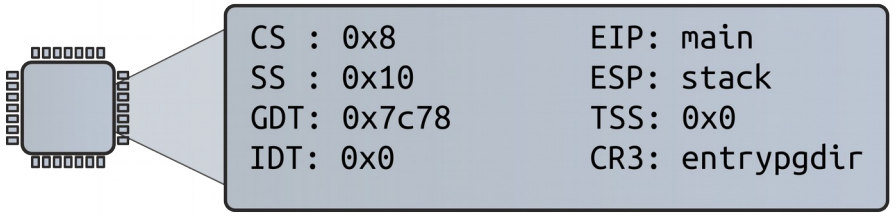
```
1316 int
1317 main(void)
1318 {
1319     kinit1(end, P2V(4*1024*1024)); // phys page allocator
1320     kvmalloc(); // kernel page table
1321     mpinit(); // detect other processors
```

- What is this **`end`**?

```
1311 extern char end[]; // first address after
                       kernel loaded from ELF file
```

Donate this free memory to the allocator





Protected Mode

Recap

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

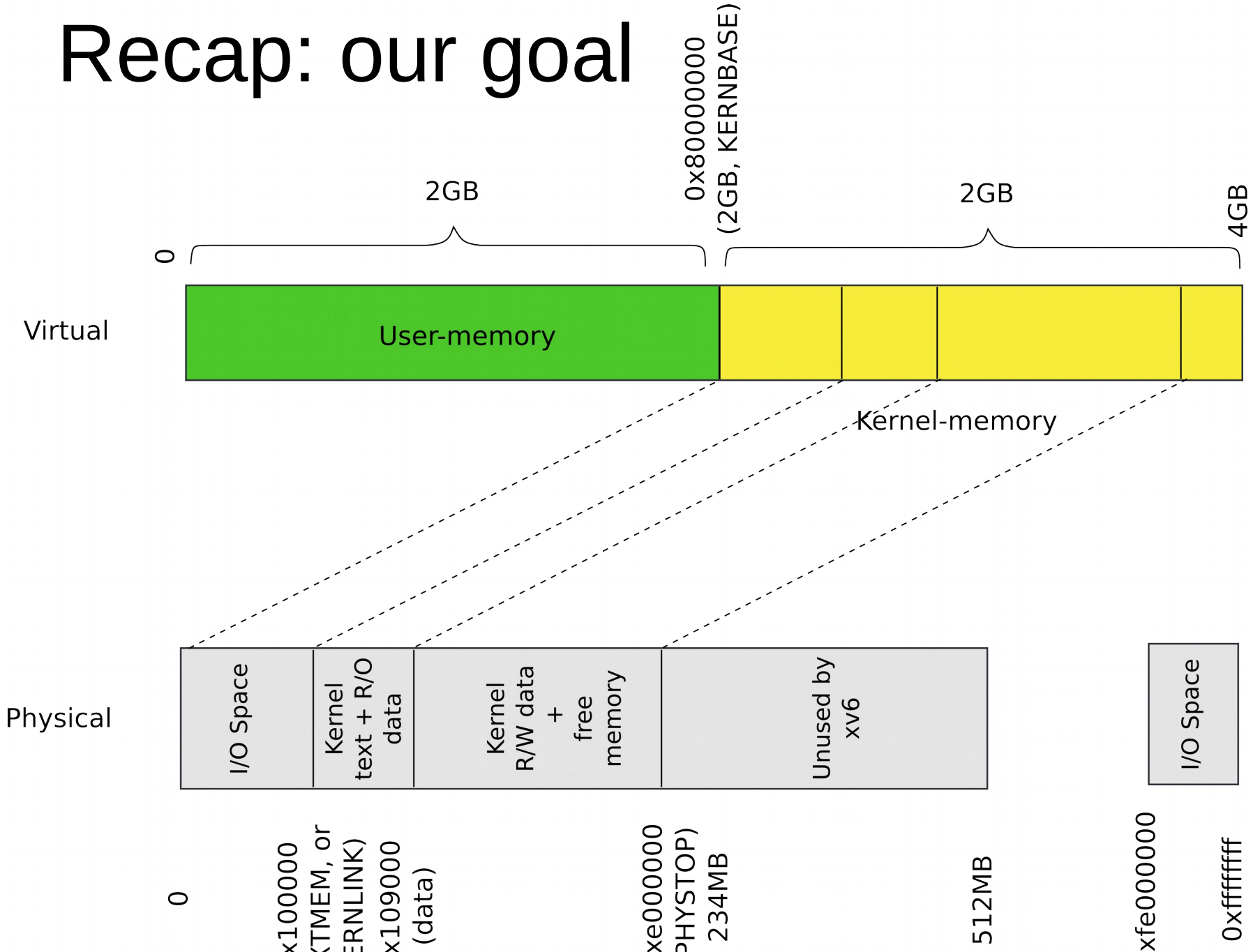
Kernel page table (4KB page tables)

Back to main(): Kernel address space

```
1316 int
1317 main(void)
1318 {
1319     kinit1(end, P2V(4*1024*1024)); // phys page allocator
1320     kvmalloc(); // kernel page table
1321     mpinit(); // detect other processors
1322     lapicinit(); // interrupt controller
1323     seginit(); // segment descriptors
1324     cprintf("\ncpu%d: starting xv6\n\n", cpunum());
1325     picinit(); // another interrupt controller
1326     ioapicinit(); // another interrupt controller
1327     consoleinit(); // console hardware
1328     uartinit(); // serial port
...
1340 }
```

- What do you think has to happen?
  - i.e., how to construct a kernel address space?

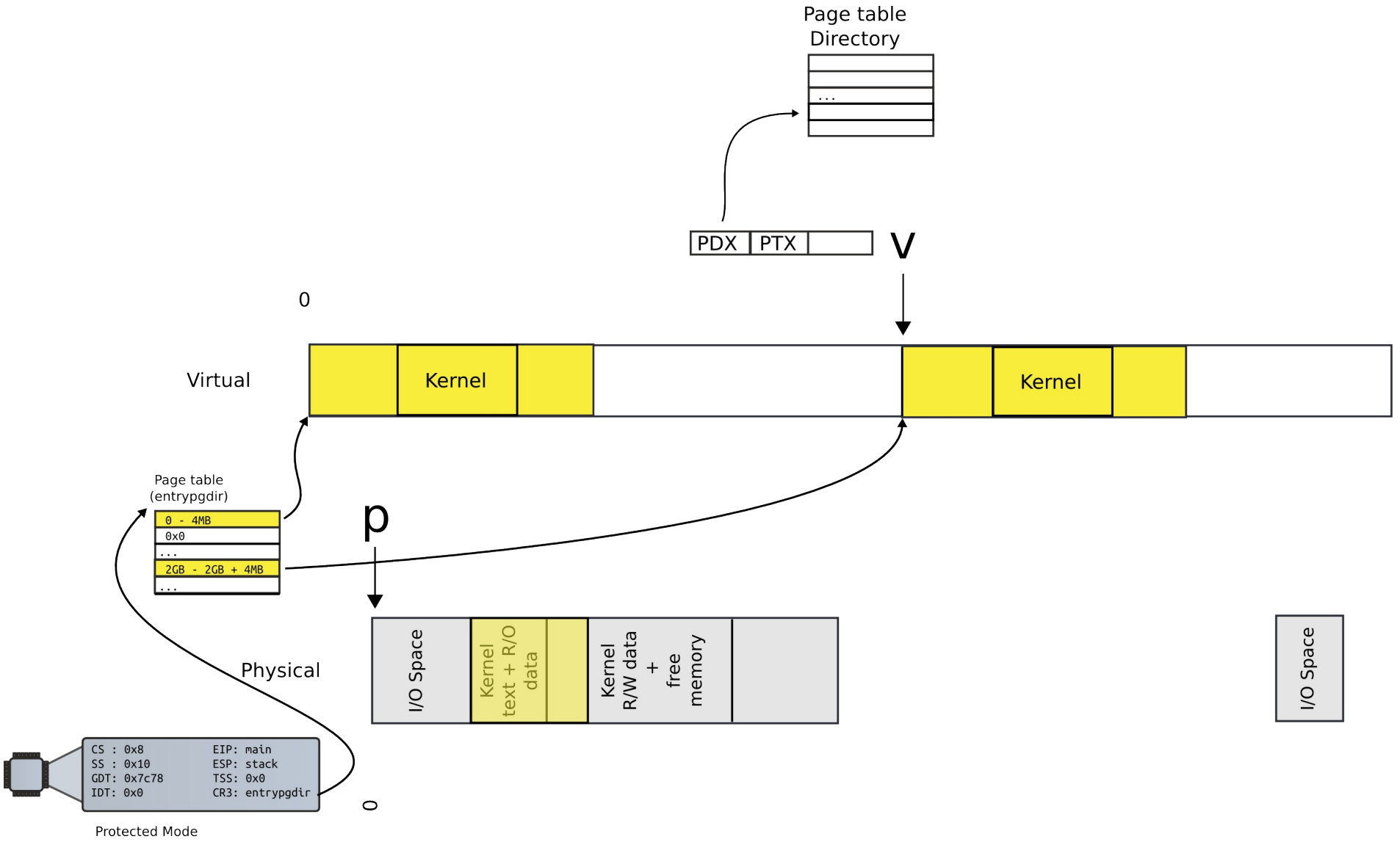
Recap: our goal



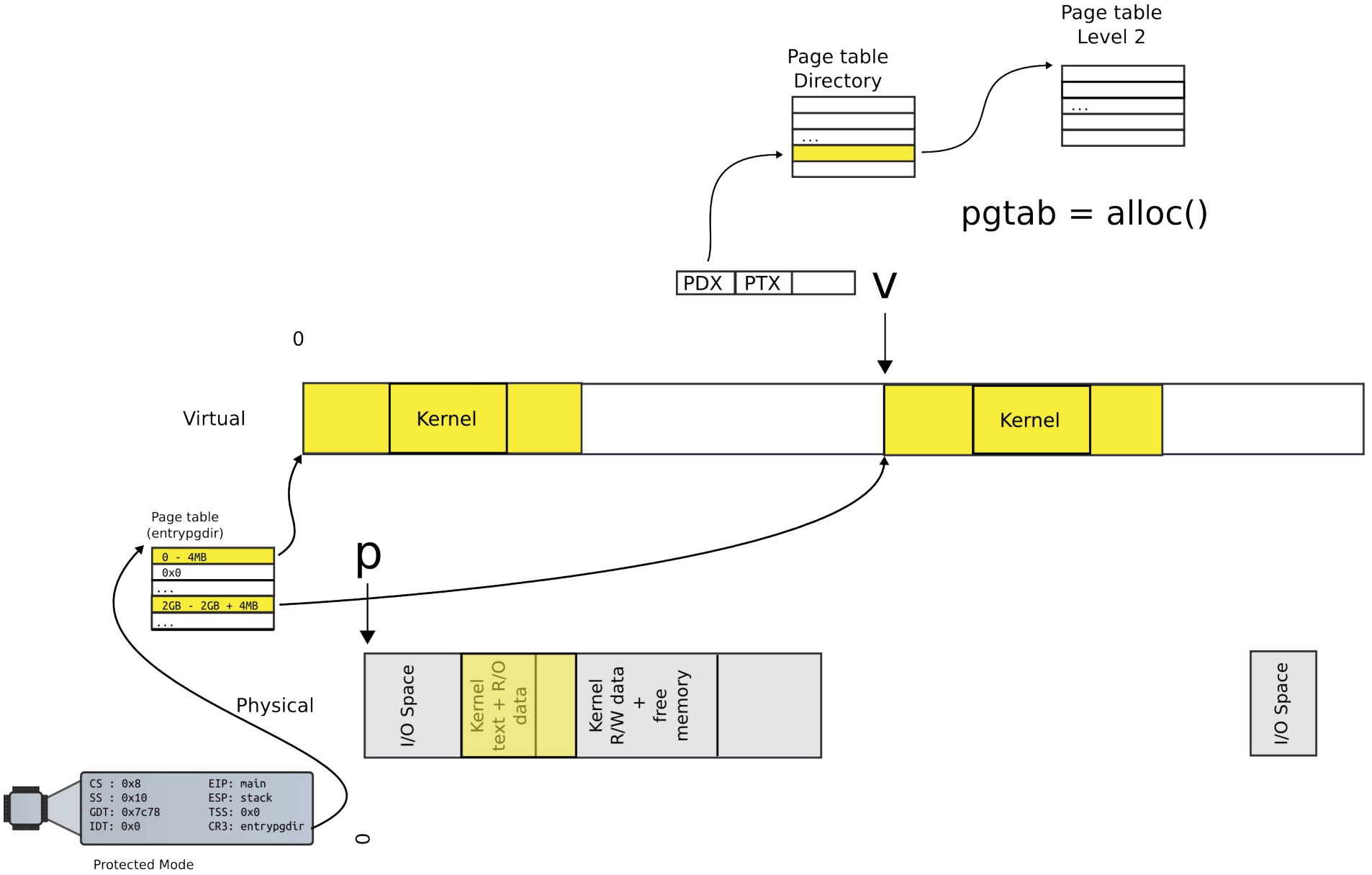
Outline

- 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
- We've created the kernel memory allocator
  - Can allocate space for page table directory and page tables

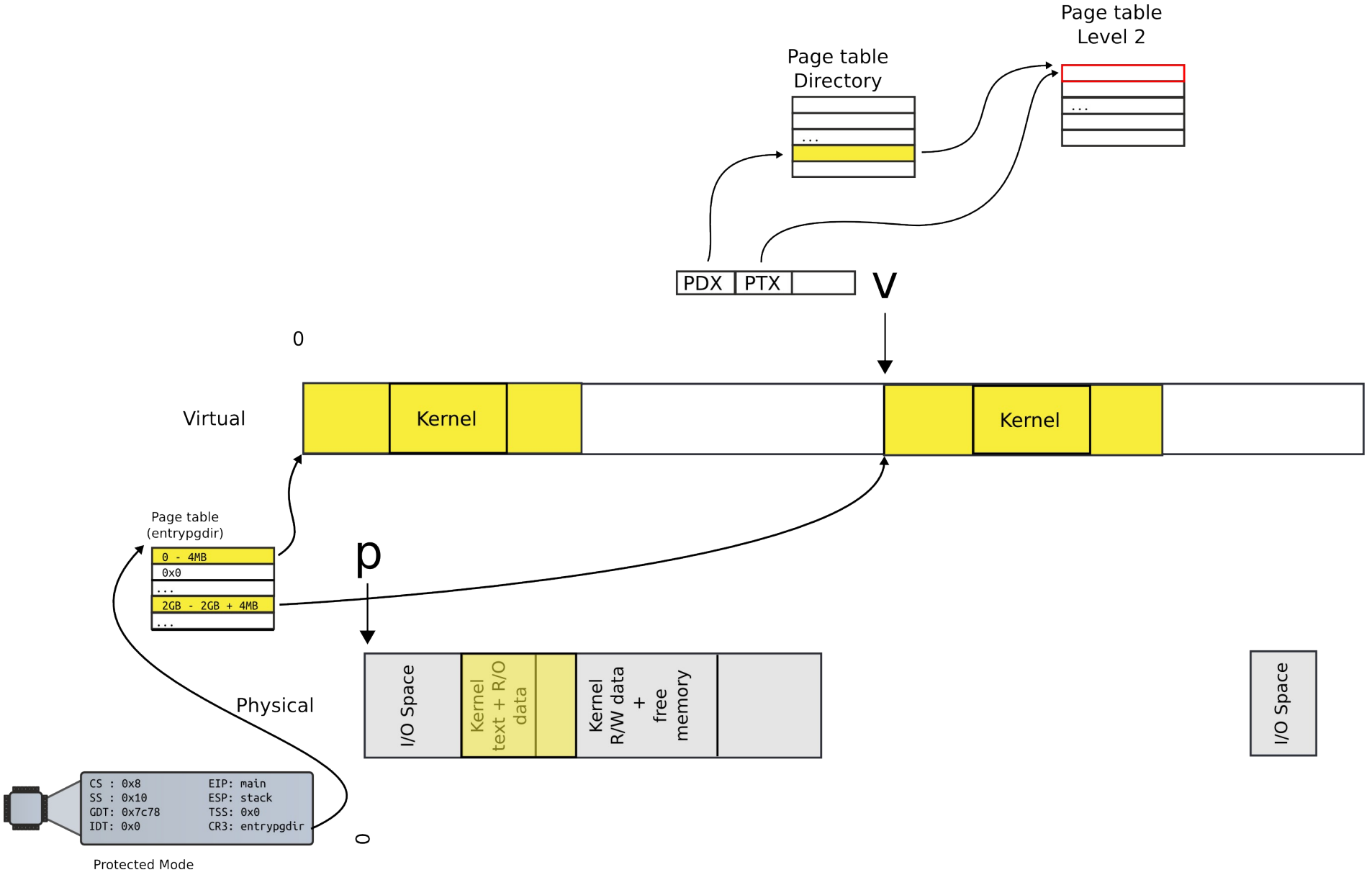
Allocate page table directory entry



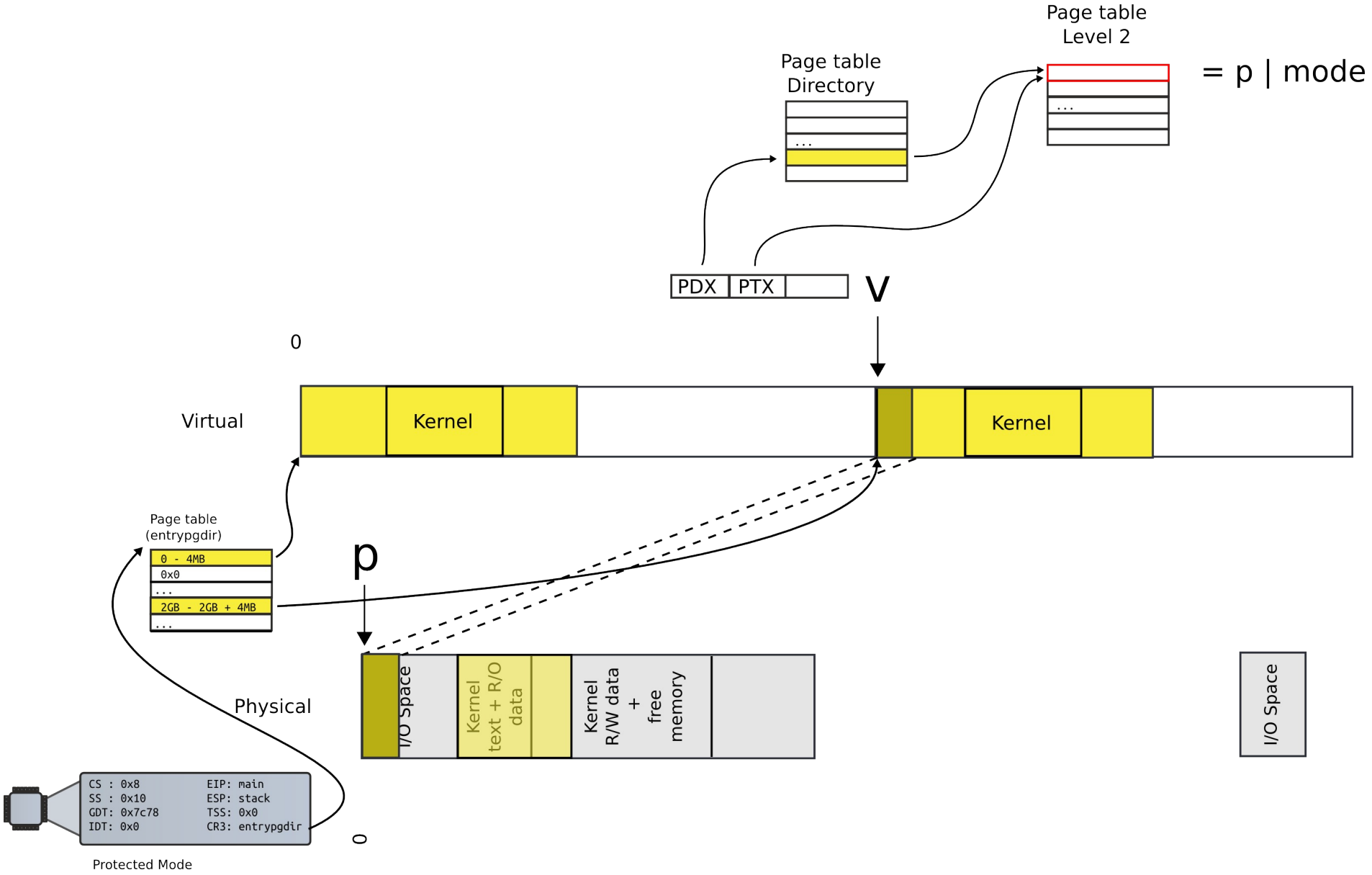
Allocate next level page table



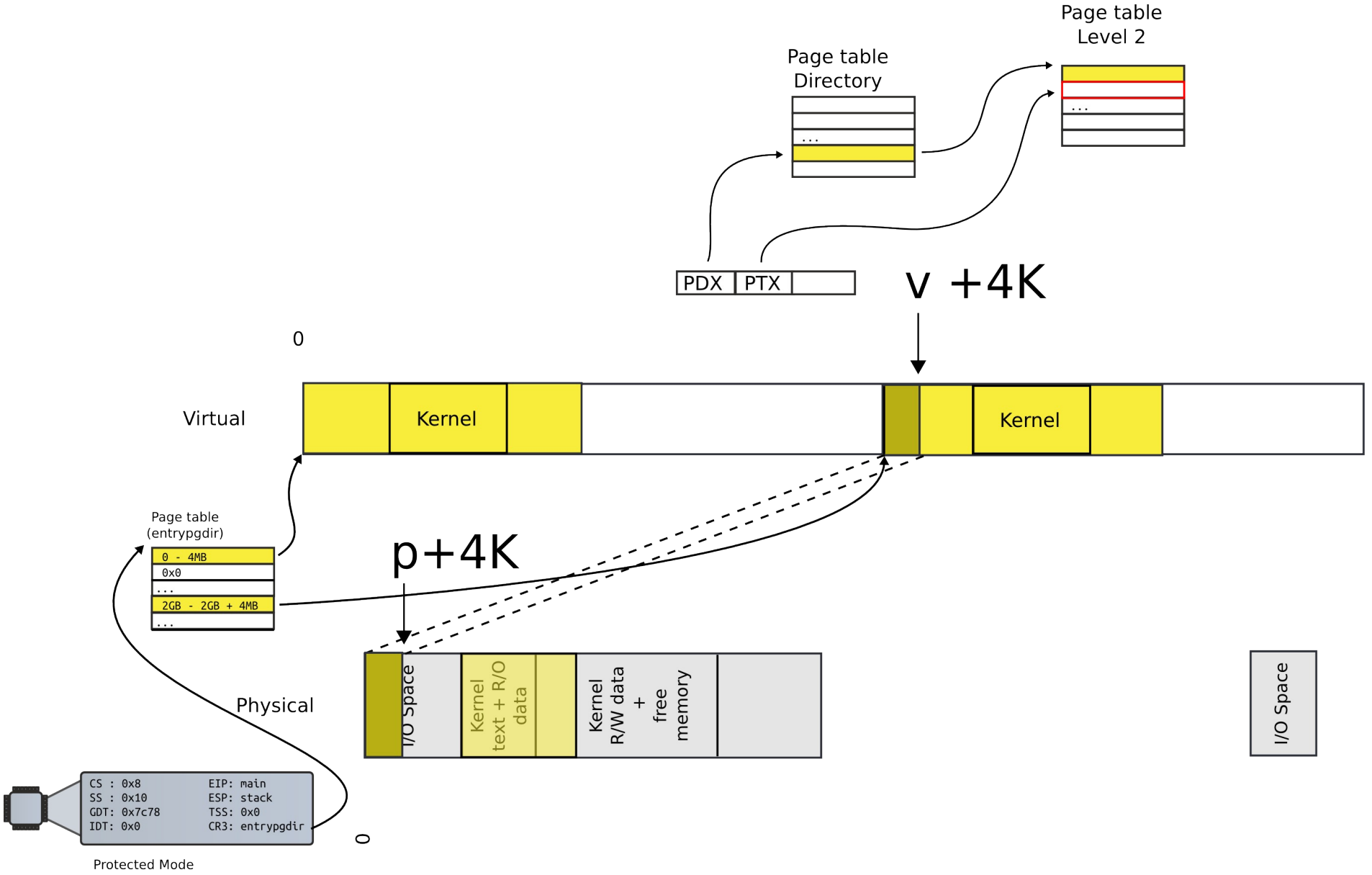
Locate PTE entry



Update mapping with physical addr



Move to next page

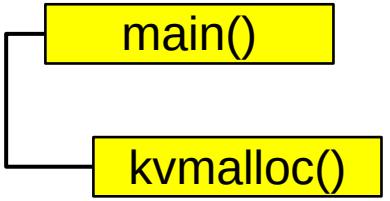


This is exactly what kernel is doing

Allocate page tables

```
1316 int
1317 main(void)
1318 {
1319     kinit1(end, P2V(4*1024*1024)); // phys page allocator
1320     kvmalloc(); // kernel page table
1321     mpinit(); // detect other processors
1322     lapicinit(); // interrupt controller
1323     seginit(); // segment descriptors
1324     cprintf("\ncpu%d: starting xv6\n\n", cpunum());
1325     picinit(); // another interrupt controller
1326     ioapicinit(); // another interrupt controller
1327     consoleinit(); // console hardware
1328     uartinit(); // serial port
...
1340 }
```

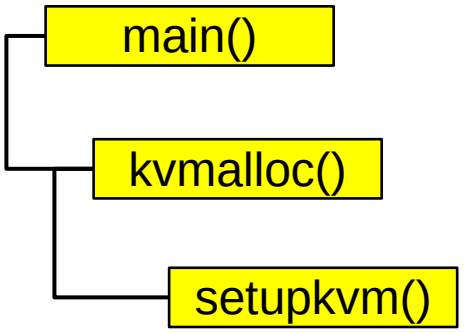
kvmalloc()



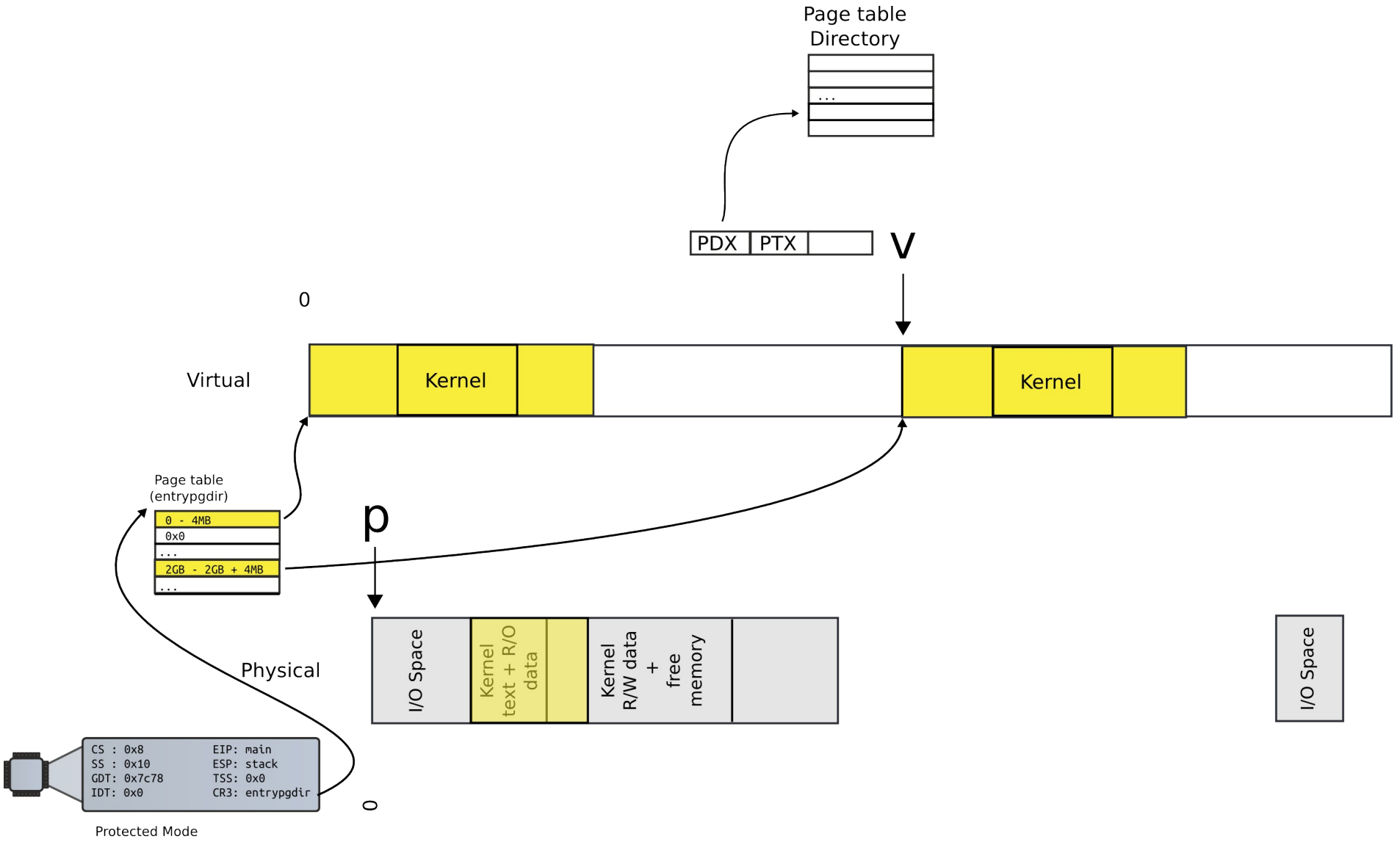

```
1857 kvmalloc(void)
1858 {
1859   kpgdir = setupkvm();
1860   switchkvm();
1861 }
```

Allocate page table directory

```
1836 pde_t*
1837 setupkvm(void)
1838 {
1839   pde_t *pgdir;
1840   struct kmap *k;
1841
1842   if((pgdir = (pde_t*)kalloc()) == 0)
1843     return 0;
1844   memset(pgdir, 0, PGSIZE);
...
1847   for(k = kmap; k < &kmap[NELEM(kmap)]; k++)
1848     if(mappages(pgdir, k->virt, k->phys_end - k->phys_start,
1849                 (uint)k->phys_start, k->perm) < 0)
1850       return 0;
1851   return pgdir;
1852 }
```

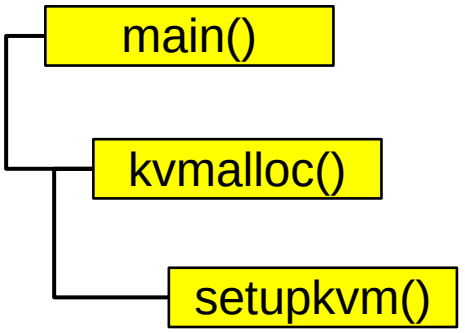


Allocate page table directory



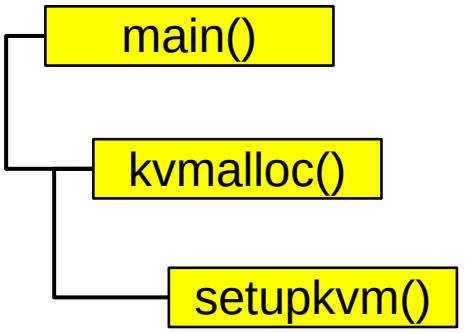
Iterate in a loop: map physical pages

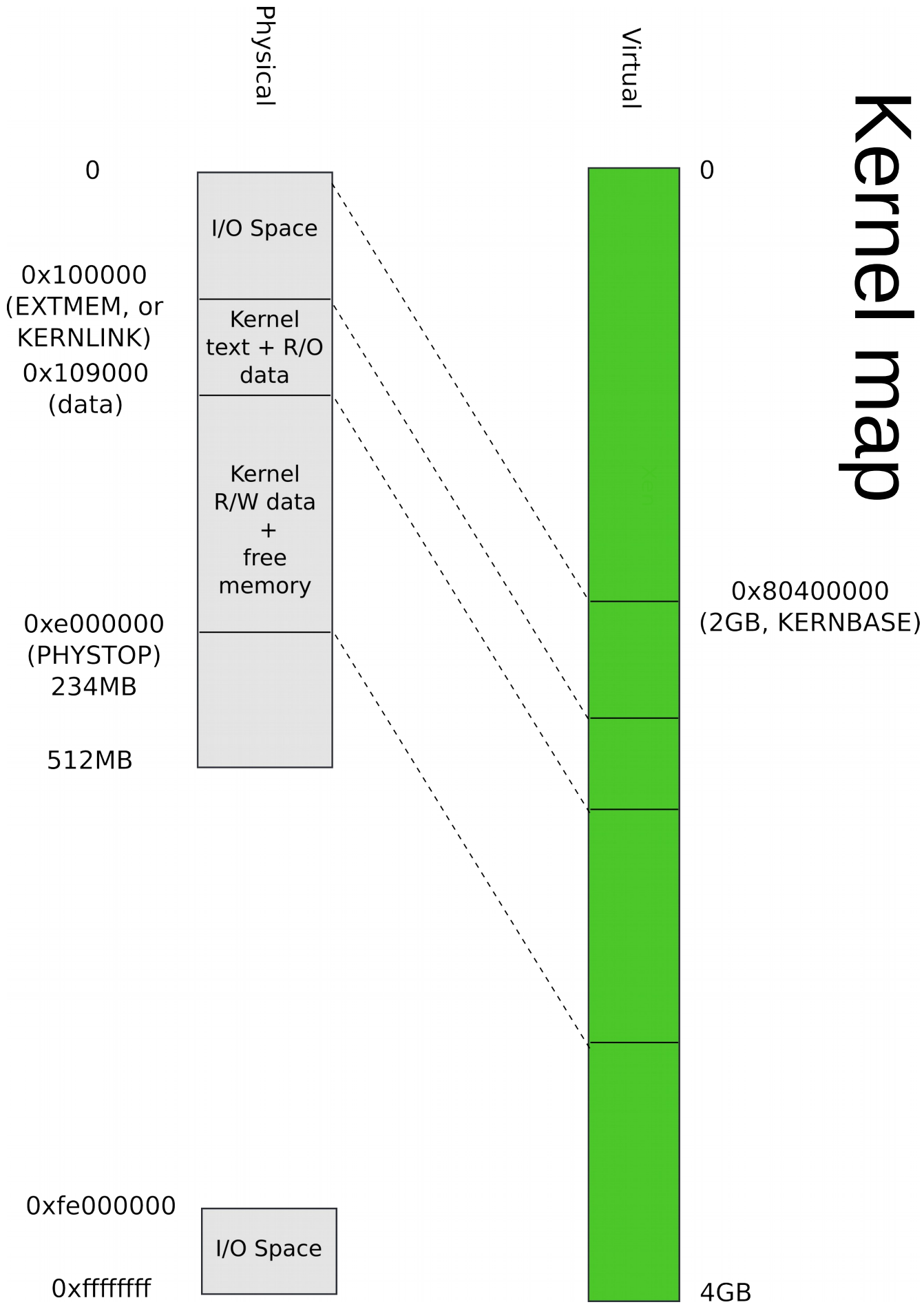
```
1836 pde_t*
1887 setupkvm(void)
1838 {
1839   pde_t *pgdir;
1840   struct kmap *k;
1841
1842   if((pgdir = (pde_t*)kalloc()) == 0)
1843     return 0;
1844   memset(pgdir, 0, PGSIZE);
...
1847   for(k = kmap; k < &kmap[NELEM(kmap)]; k++)
1848     if(mappages(pgdir, k->virt, k->phys_end - k->phys_start,
1849                 (uint)k->phys_start, k->perm) < 0)
1850       return 0;
1851   return pgdir;
1852 }
```



Iterate in a loop: map physical pages

```
1836 pde_t*
1887 setupkvm(void)
1838 {
1839   pde_t *pgdir;
1840   struct kmap *k;
1841
1842   if((pgdir = (pde_t*)kalloc()) == 0)
1843     return 0;
1844   memset(pgdir, 0, PGSIZE);
...
1847   for(k = kmap; k < &kmap[NELEM(kmap)]; k++)
1848     if(mappages(pgdir, k->virt, k->phys_end - k->phys_start,
1849                 (uint)k->phys_start, k->perm) < 0)
1850       return 0;
1851   return pgdir;
1852 }
```



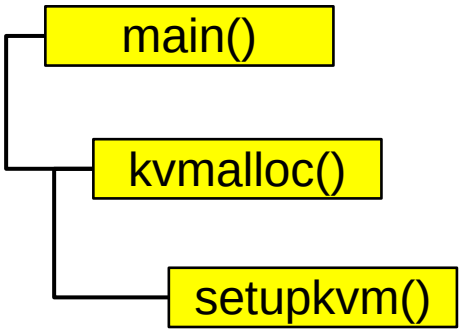

Kernel map
Physical
Virtual
0
I/O Space
0x100000
(EXTMEM, or
KERNLINK)
Kernel
text + R/O
data
0x109000
(data)
Kernel
R/W data
+
free
memory
0xe000000
(PHYSTOP)
234MB
512MB
0xfe000000
I/O Space
0xffffffff
0
0x80400000
(2GB, KERNBASE)
4GB

Kmap – kernel map

```
1823 static struct kmap {
1824     void *virt;
1825     uint phys_start;
1826     uint phys_end;
1827     int perm;
1828 } kmap[] = {
1829     { (void*)KERNBASE, 0, EXTMEM, PTE_W}, // I/O space
1830     { (void*)KERNLINK, V2P(KERNLINK), V2P(data), 0},//text+rodata
1831     { (void*)data, V2P(data), PHYSTOP, PTE_W}, // kern data+memory
1832     { (void*)DEVSPACE, DEVSPACE, 0, PTE_W}, // more devices
1833 };
```

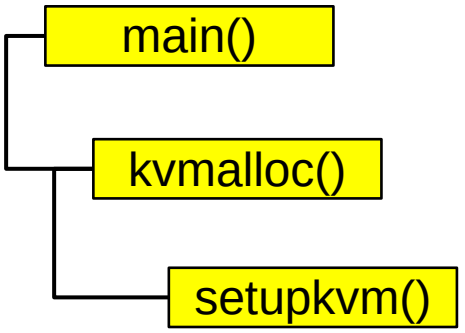
Iterate in a loop: map physical pages

```
1836 pde_t*
1887 setupkvm(void)
1838 {
1839   pde_t *pgdir;
1840   struct kmap *k;
1841
1842   if((pgdir = (pde_t*)kalloc()) == 0)
1843     return 0;
1844   memset(pgdir, 0, PGSIZE);
...
1847   for(k = kmap; k < &kmap[NELEM(kmap)]; k++)
1848     if(mappages(pgdir, k->virt, k->phys_end - k->phys_start,
1849                 (uint)k->phys_start, k->perm) < 0)
1850       return 0;
1851   return pgdir;
1852 }
```

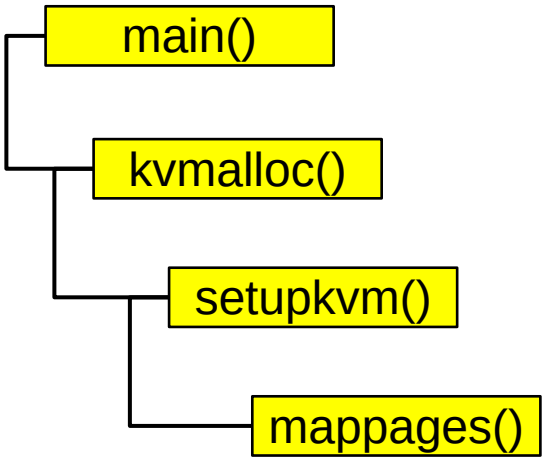


Map a region of memory

```
1836 pde_t*
1887 setupkvm(void)
1838 {
1839   pde_t *pgdir;
1840   struct kmap *k;
1841
1842   if((pgdir = (pde_t*)kalloc()) == 0)
1843     return 0;
1844   memset(pgdir, 0, PGSIZE);
...
1847   for(k = kmap; k < &kmap[NELEM(kmap)]; k++)
1848     if(mappages(pgdir, k->virt, k->phys_end - k->phys_start,
1849                 (uint)k->phys_start, k->perm) < 0)
1850       return 0;
1851   return pgdir;
1852 }
```

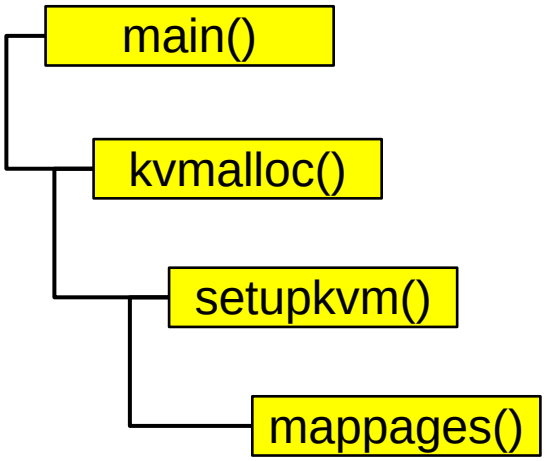


```
1779 mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm)
1780 {
1781   char *a, *last;
1782   pte_t *pte;
1783
1784   a = (char*)PGROUNDDOWN((uint)va);
1785   last = (char*)PGROUNDDOWN(((uint)va) + size - 1);
1786   for(;;){
1787     if((pte = walkpgdir(pgdir, a, 1)) == 0)
1788       return -1;
1789     if(*pte & PTE_P)
1790       panic("remap");
1791     *pte = pa | perm | PTE_P;
1792     if(a == last)
1793       break;
1794     a += PGSIZE;
1795     pa += PGSIZE;
1796   }
1797   return 0;
1798 }
```



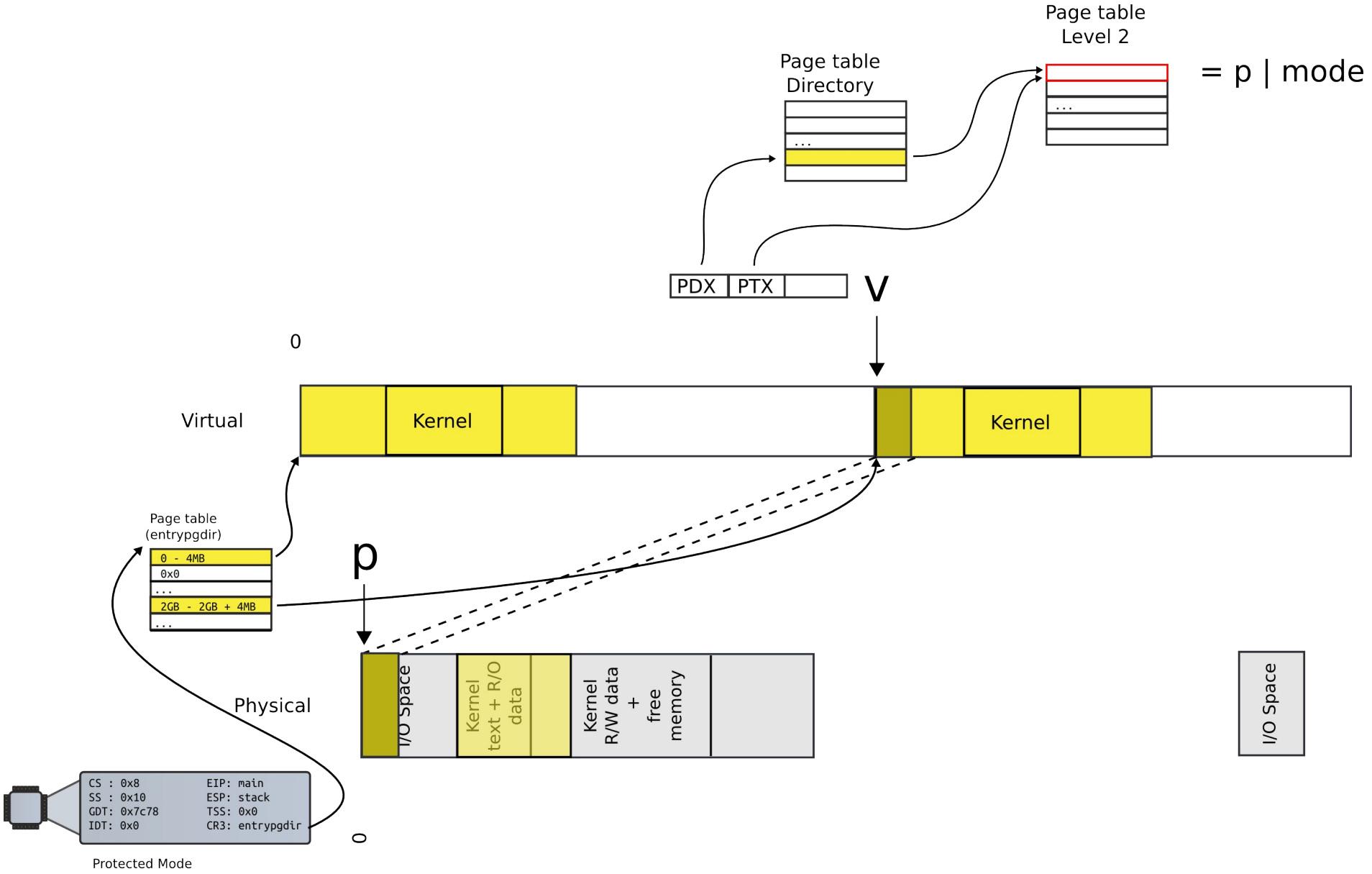
- Get the start (`a`) and end (`last`) pages fo the virtual address range we are mapping
- Then work in a loop mapping every page one by one

```
1779 mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm)
1780 {
1781   char *a, *last;
1782   pte_t *pte;
1783
1784   a = (char*)PGROUNDDOWN((uint)va);
1785   last = (char*)PGROUNDDOWN(((uint)va) + size - 1);
1786   for(;;){
1787     if((pte = walkpgdir(pgdir, a, 1)) == 0)
1788       return -1;
1789     if(*pte & PTE_P)
1790       panic("remap");
1791     *pte = pa | perm | PTE_P;
1792     if(a == last)
1793       break;
1794     a += PGSIZE;
1795     pa += PGSIZE;
1796   }
1797   return 0;
1798 }
```

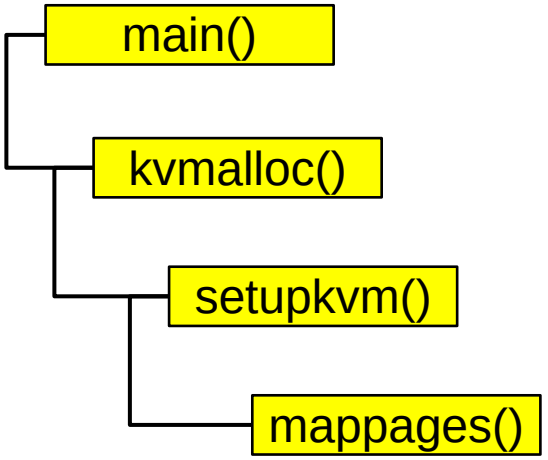


- First lookup the page table directory entry (`pte`) corresponding to the virtual address (`a`) we're mapping

Locate the page table entry

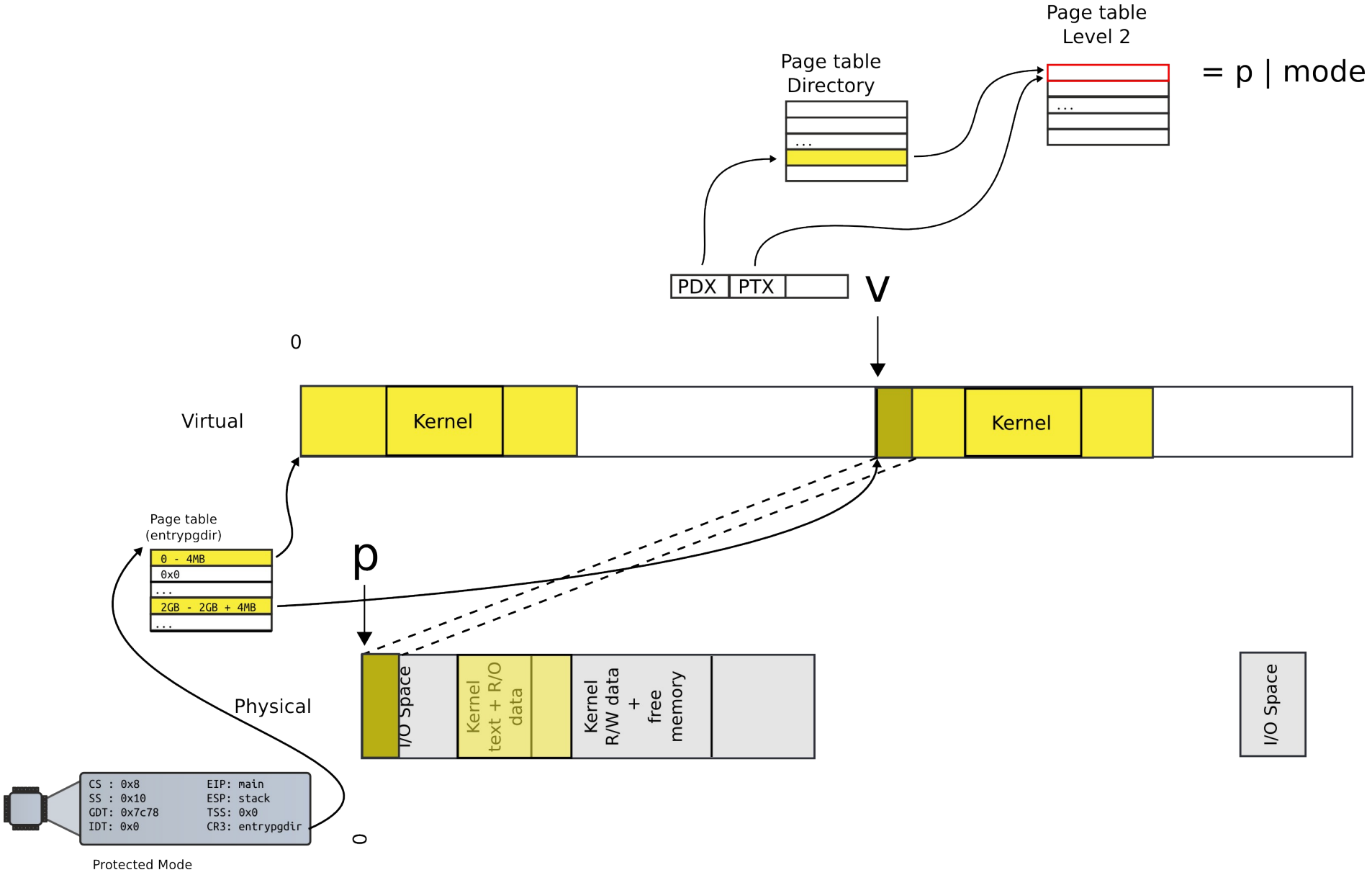


```
1779 mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm)
1780 {
1781   char *a, *last;
1782   pte_t *pte;
1783
1784   a = (char*)PGROUNDDOWN((uint)va);
1785   last = (char*)PGROUNDDOWN(((uint)va) + size - 1);
1786   for(;;){
1787     if((pte = walkpgdir(pgdir, a, 1)) == 0)
1788       return -1;
1789     if(*pte & PTE_P)
1790       panic("remap");
1791     *pte = pa | perm | PTE_P;
1792     if(a == last)
1793       break;
1794     a += PGSIZE;
1795     pa += PGSIZE;
1796   }
1797   return 0;
1798 }
```

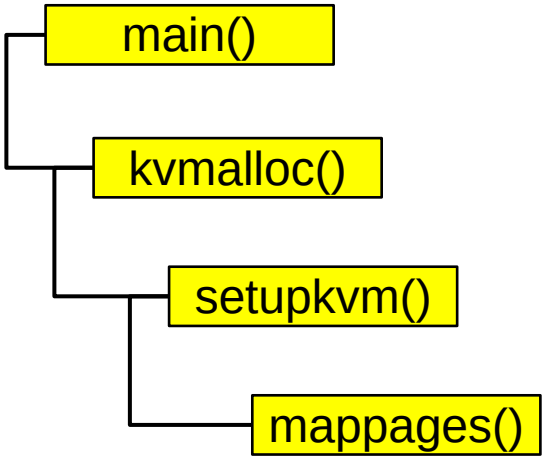


- Update the page directory entry (`*pte`) with the physical address (`pa`)

Update mapping with physical addr



```
1779 mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm)
1780 {
1781   char *a, *last;
1782   pte_t *pte;
1783
1784   a = (char*)PGROUNDDOWN((uint)va);
1785   last = (char*)PGROUNDDOWN(((uint)va) + size - 1);
1786   for(;;){
1787     if((pte = walkpgdir(pgdir, a, 1)) == 0)
1788       return -1;
1789     if(*pte & PTE_P)
1790       panic("remap");
1791     *pte = pa | perm | PTE_P;
1792     if(a == last)
1793       break;
1794     a += PGSIZE;
1795     pa += PGSIZE;
1796   }
1797   return 0;
1798 }
```

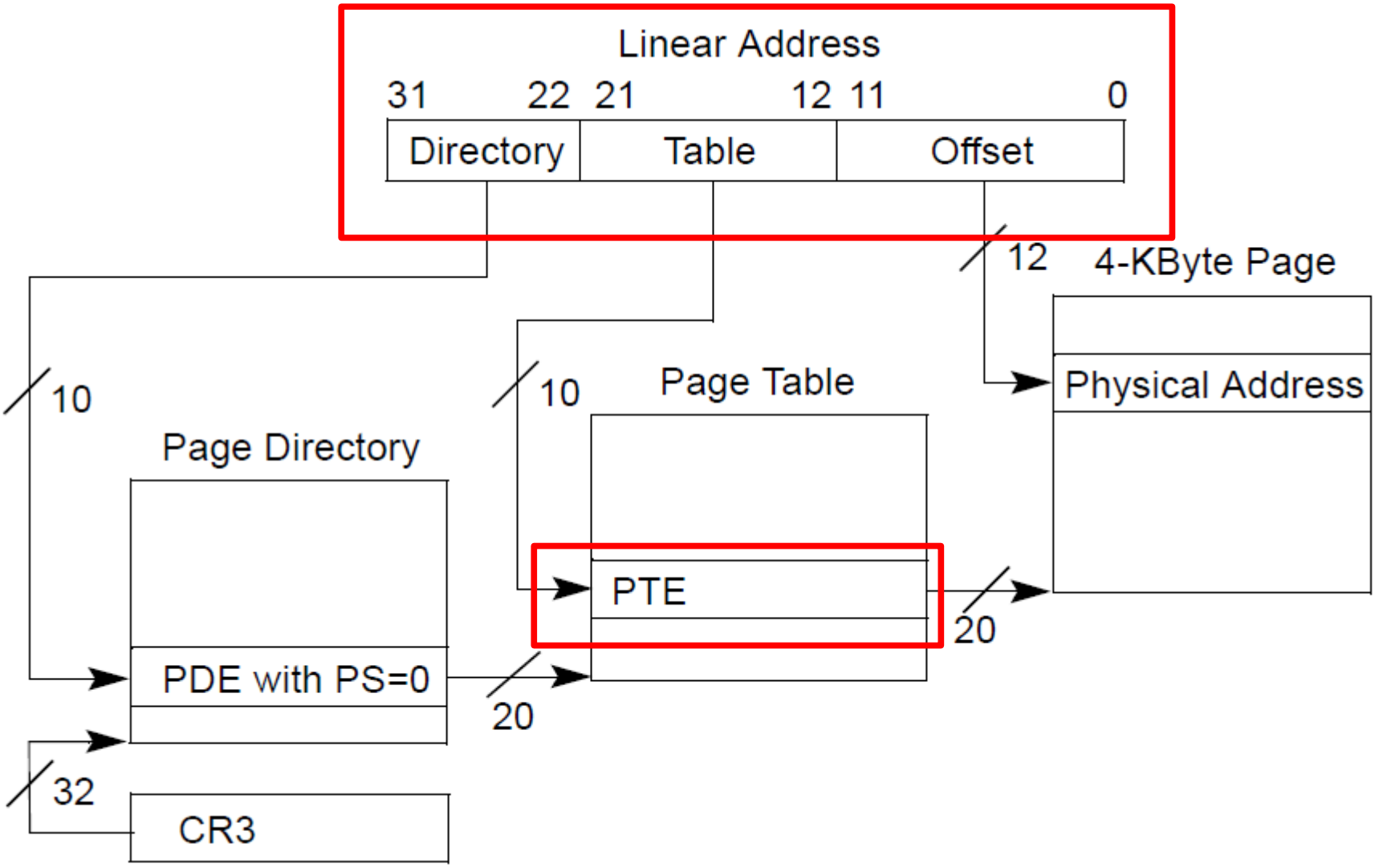


- But we need a function that locates the `pte` for us...

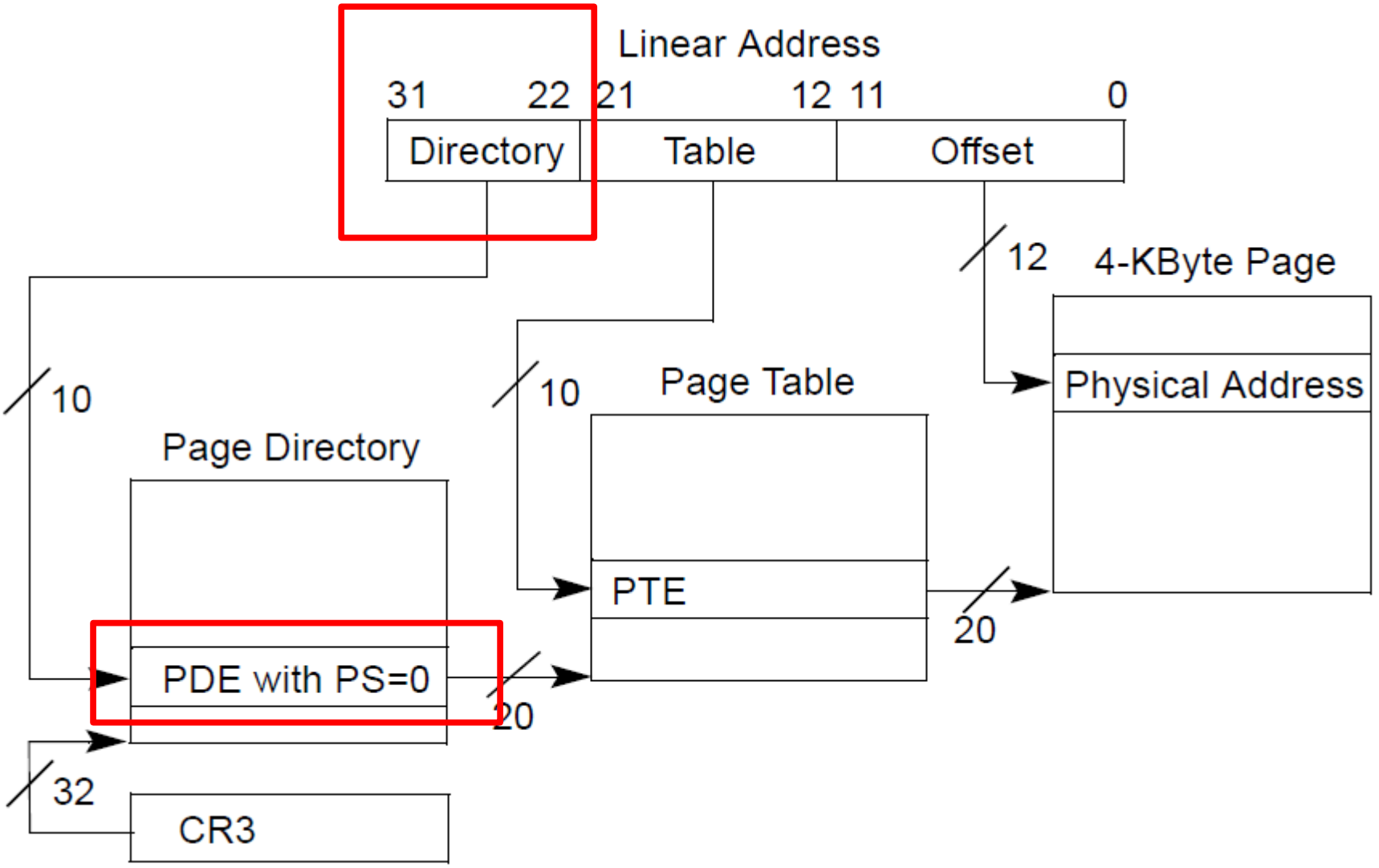
What should it look like?

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

Recap of the page table

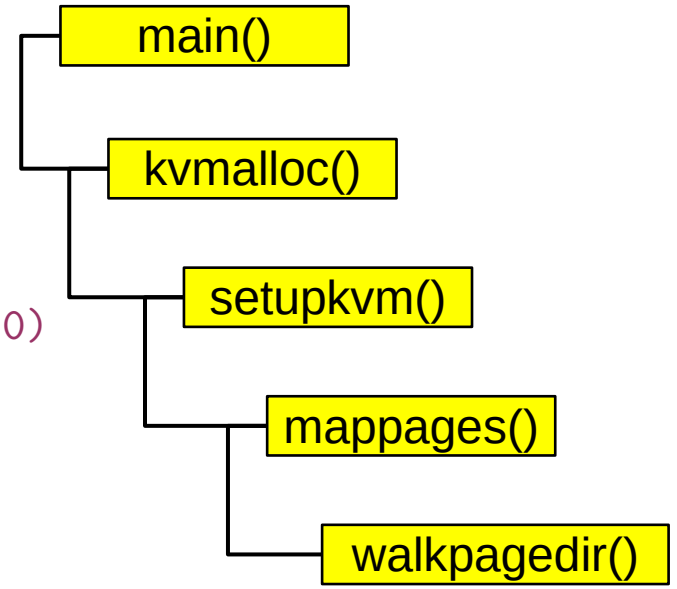


Locate the PDE frist



```
1754 walkpgdir(pde_t *pgdir, const void *va, int alloc)
1755 {
1756     pde_t *pde;
1757     pte_t *pgtab;
1758
1759     pde = &pgdir[PDX(va)];
1760     if(*pde & PTE_P){
1761         pgtab = (pte_t*)P2V(PTE_ADDR(*pde));
1762     } else {
1763         if(!alloc || (pgtab = (pte_t*)kalloc()) == 0)
1764             return 0;
1765         // Make sure all those PTE_P bits are zero.
1766         memset(pgtab, 0, PGSIZE);
...
1770         *pde = V2P(pgtab) | PTE_P | PTE_W | PTE_U;
1771     }
1772     return &pgtab[PTX(va)];
1773 }
```

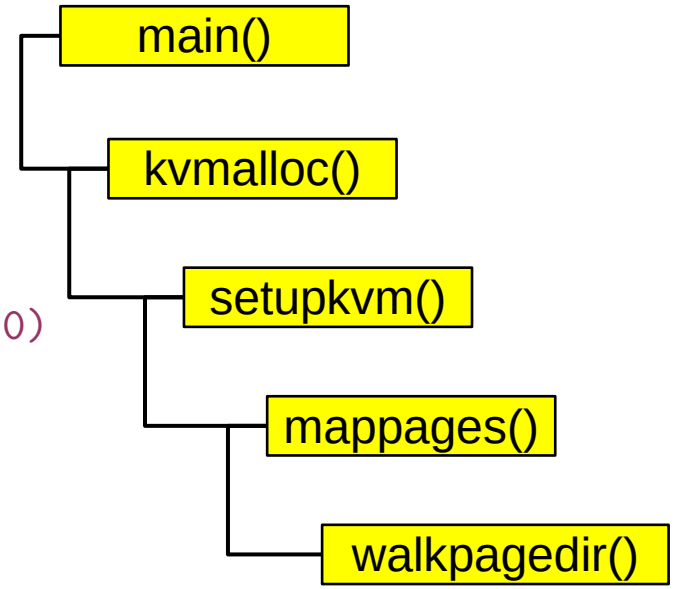
Locate the page table directory entry for this virtual address



- Locate the page directory entry (`*pde`)

```
1754 walkpgdir(pde_t *pgdir, const void *va, int alloc)
1755 {
1756     pde_t *pde;
1757     pte_t *pgtab;
1758
1759     pde = &pgdir[PDX(va)];
1760     if(*pde & PTE_P){
1761         pgtab = (pte_t*)P2V(PTE_ADDR(*pde));
1762     } else {
1763         if(!alloc || (pgtab = (pte_t*)kalloc()) == 0)
1764             return 0;
1765         // Make sure all those PTE_P bits are zero.
1766         memset(pgtab, 0, PGSIZE);
...
1770         *pde = V2P(pgtab) | PTE_P | PTE_W | PTE_U;
1771     }
1772     return &pgtab[PTX(va)];
1773 }
```

Locate the page table directory entry for this virtual address



- Locate the page directory entry (`*pde`)

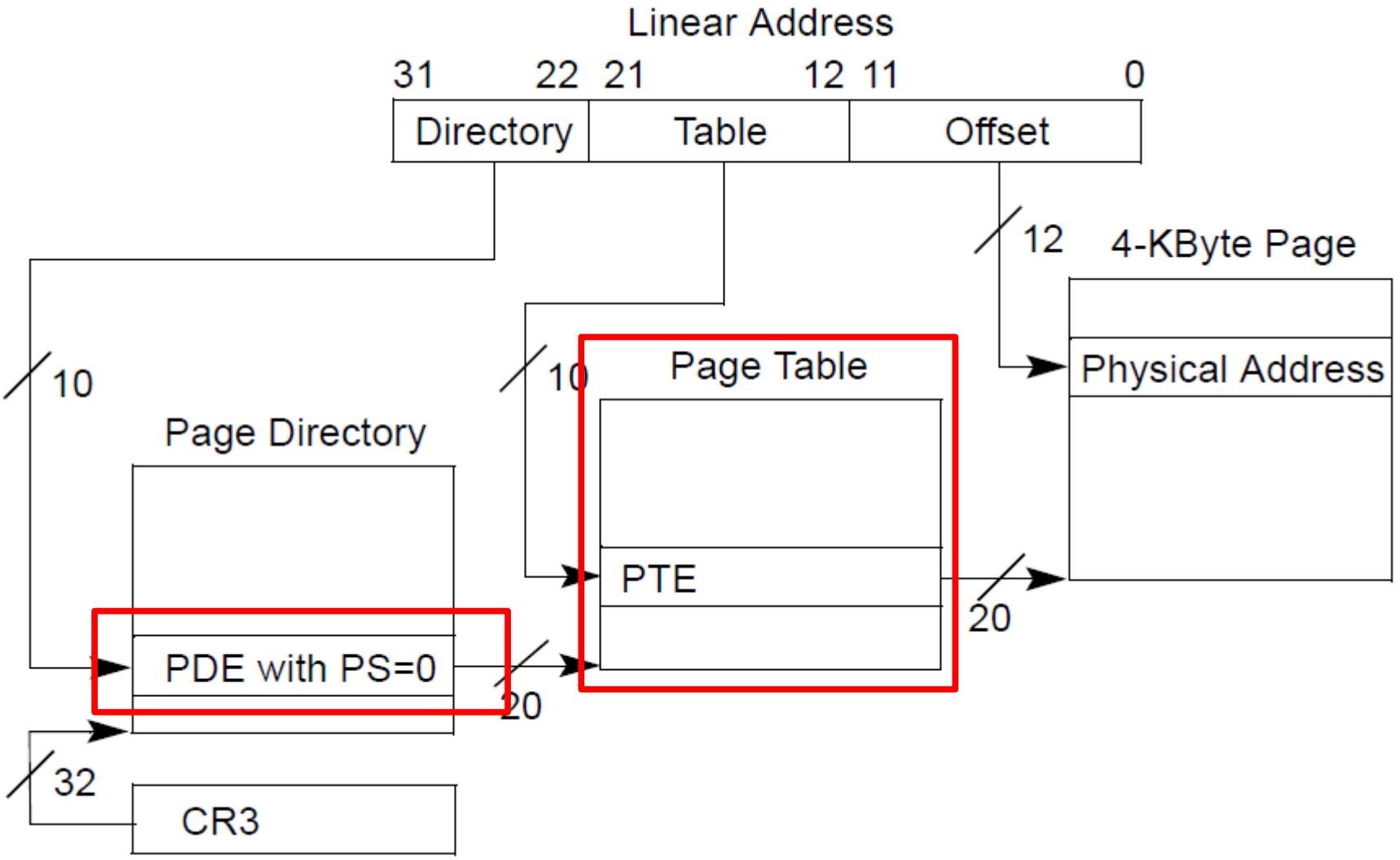
PDX()

```
0855 // +--------10------+-------10-------+---------12----------+
0856 // | Page Directory |    Page Table   | Offset within Page  |
0857 // |     Index      |      Index      |                     |
0858 // +----------------+-----------------+---------------------+
0859 // \--- PDX(va) --/   \--- PTX(va) --/
0860
0861 // page directory index
0862 #define PDX(va) (((uint)(va) >> PDXSHIFT) & 0x3FF)
0863
0864 // page table index
0865 #define PTX(va) (((uint)(va) >> PTXSHIFT) & 0x3FF)
…
0876 #define PTXSHIFT 12 // offset of PTX in a linear address
0877 #define PDXSHIFT 22 // offset of PDX in a linear address
```

PDX()

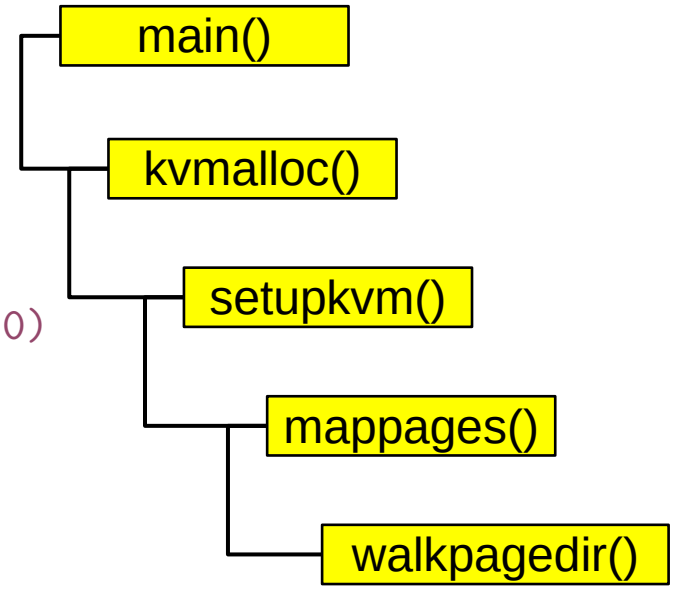
```
0855 // +--------10------+-------10-------+----------12----------+
0856 // | Page Directory |    Page Table   | Offset within Page  |
0857 // |     Index      |      Index      |                     |
0858 // +----------------+-----------------+---------------------+
0859 // \--- PDX(va) --/  \--- PTX(va) --/
0860
0861 // page directory index
0862 #define PDX(va) (((uint)(va) >> PDXSHIFT) & 0x3FF)
0863
0864 // page table index
0865 #define PTX(va) (((uint)(va) >> PTXSHIFT) & 0x3FF)
…
0876 #define PTXSHIFT 12 // offset of PTX in a linear address
0877 #define PDXSHIFT 22 // offset of PDX in a linear address
```

Check if level 2 page table is allocated

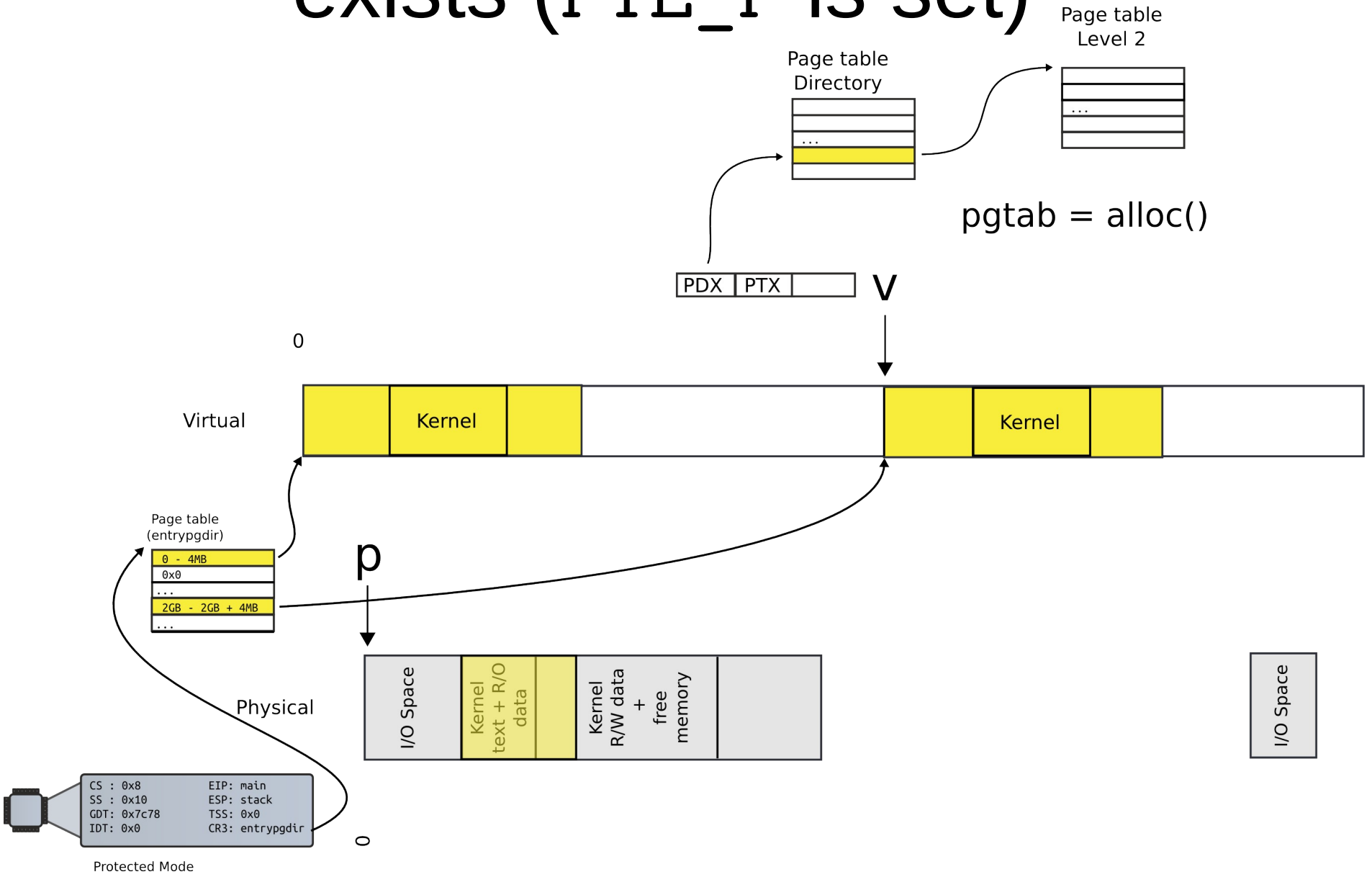


```
1754 walkpgdir(pde_t *pgdir, const void *va, int alloc)
1755 {
1756     pde_t *pde;
1757     pte_t *pgtab;
1758
1759     pde = &pgdir[PDX(va)];
1760     if(*pde & PTE_P){
1761         pgtab = (pte_t*)P2V(PTE_ADDR(*pde));
1762     } else {
1763         if(!alloc || (pgtab = (pte_t*)kalloc()) == 0)
1764             return 0;
1765         // Make sure all those PTE_P bits are zero.
1766         memset(pgtab, 0, PGSIZE);
...
1770         *pde = V2P(pgtab) | PTE_P | PTE_W | PTE_U;
1771     }
1772     return &pgtab[PTX(va)];
1773 }
```

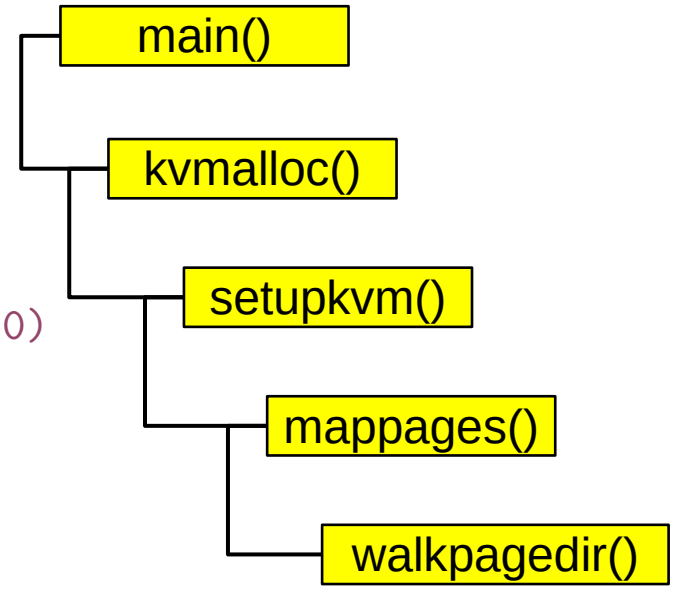
Check if the level 2 page is allocated already



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

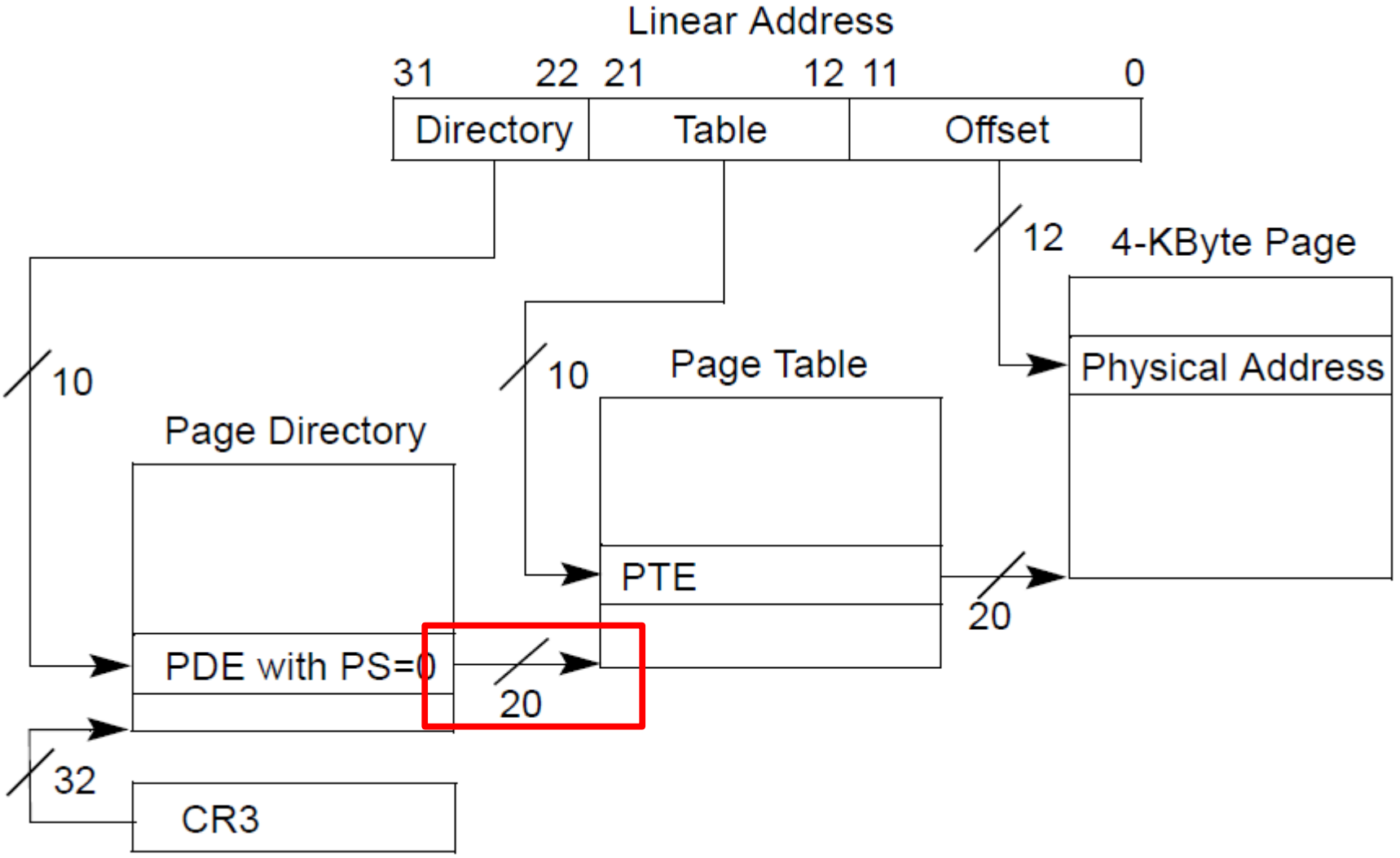


```
1754 walkpgdir(pde_t *pgdir, const void *va, int alloc)
1755 {
1756     pde_t *pde;
1757     pte_t *pgtab;
1758
1759     pde = &pgdir[PDX(va)];
1760     if(*pde & PTE_P){
1761         pgtab = (pte_t*)P2V(PTE_ADDR(*pde));
1762     } else {
1763         if(!alloc || (pgtab = (pte_t*)kalloc()) == 0)
1764             return 0;
1765         // Make sure all those PTE_P bits are zero.
1766         memset(pgtab, 0, PGSIZE);
...
1770         *pde = V2P(pgtab) | PTE_P | PTE_W | PTE_U;
1771     }
1772     return &pgtab[PTX(va)];
1773 }
```



- If yes, locate the page (`pgtab`) containing the level 2 page table

PDE contains 20 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 has to be mapped by the current page table

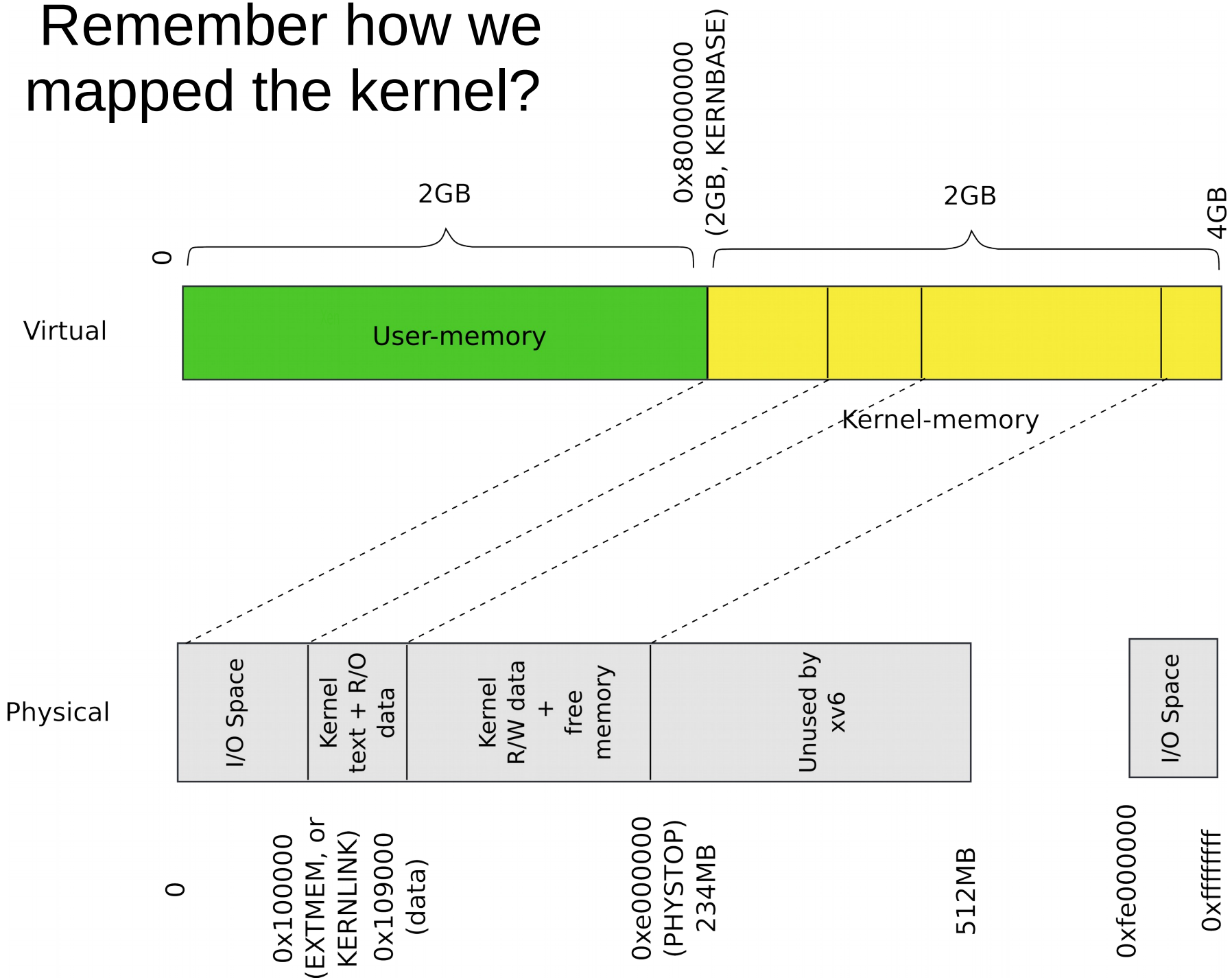
Step 1

- Convert from 20 bits of physical page number to physical address of the page
  - `PTE_ADDR(*pde)`
  - This is trivial

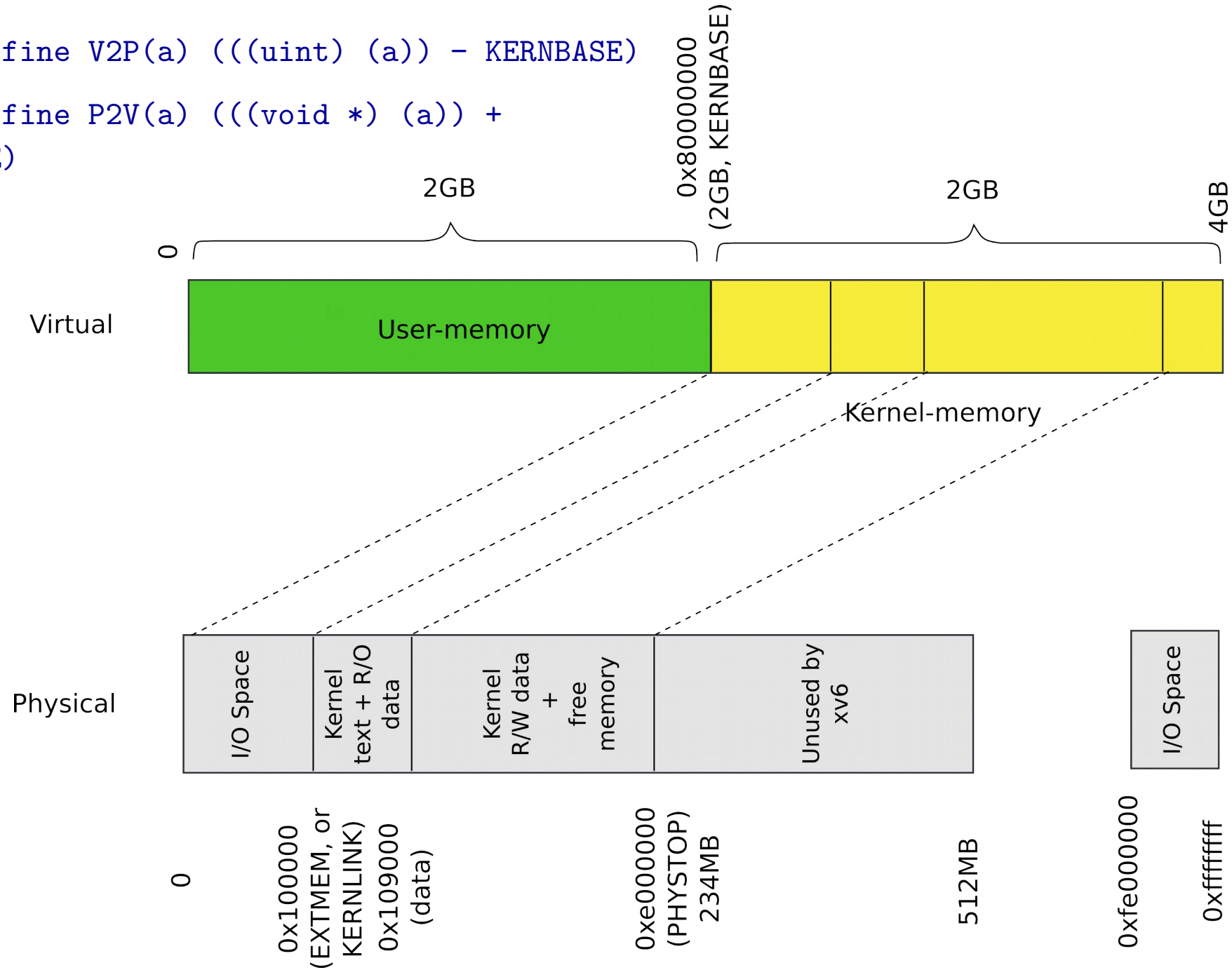
Step 2

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

Remember how we mapped the kernel?

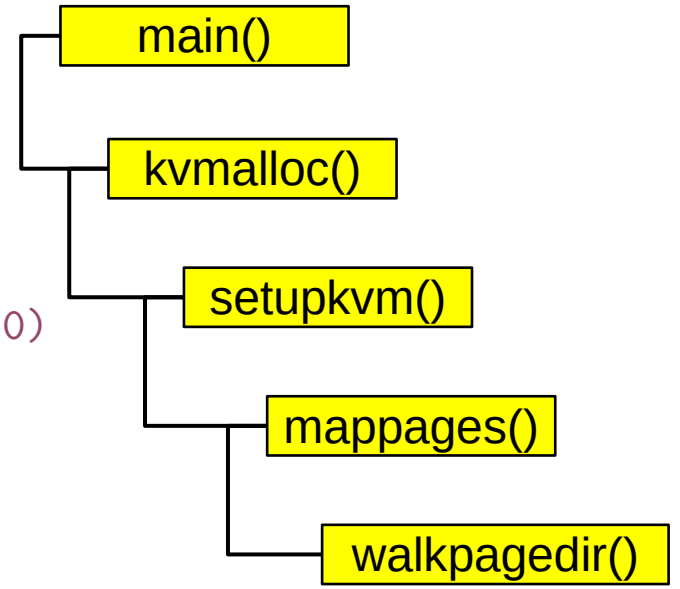


```
0207 #define KERNBASE 0x80000000 // First
kernel virtual address

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

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



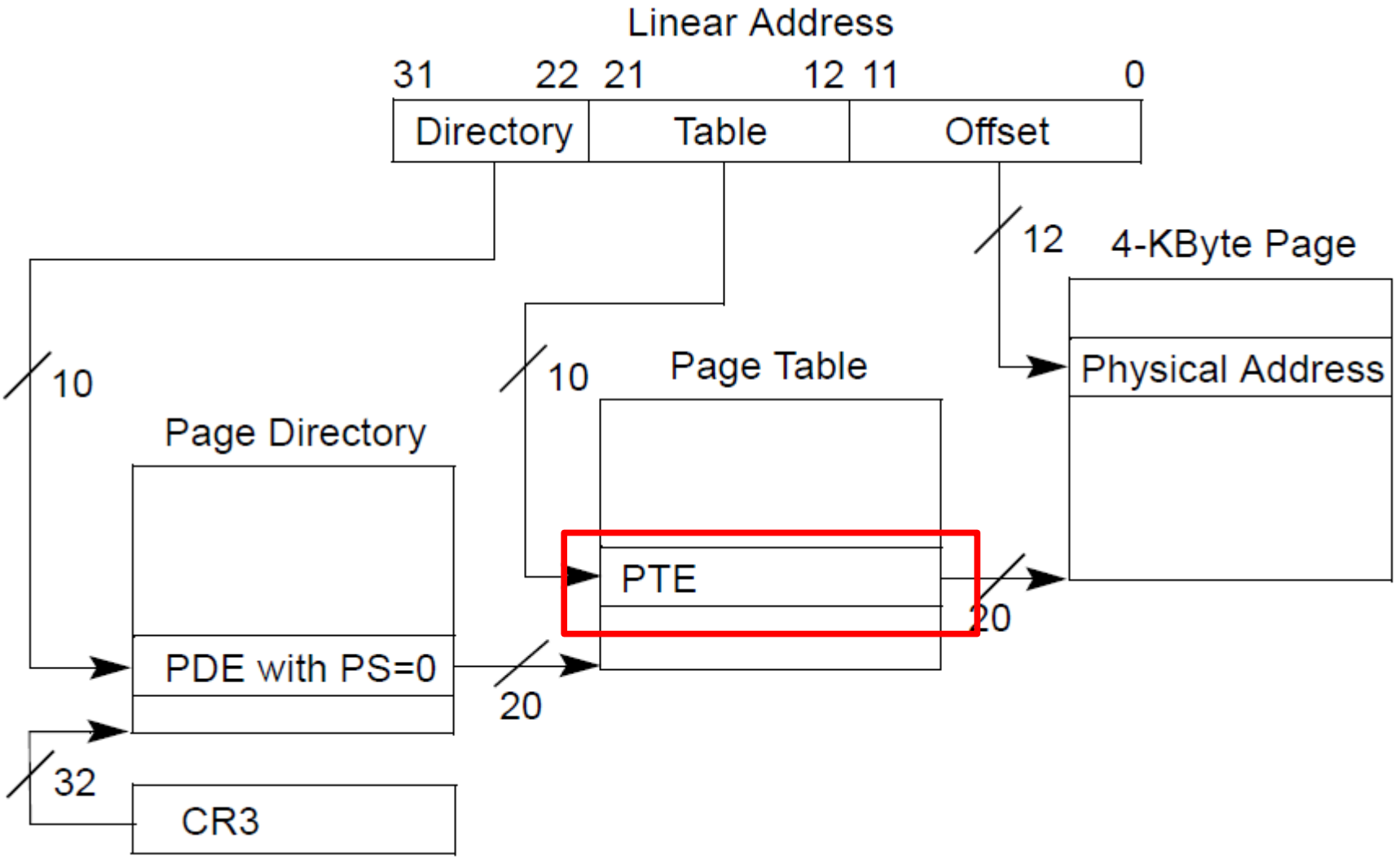
Walk page table

```
1754 walkpgdir(pde_t *pgdir, const void *va, int alloc)
1755 {
1756     pde_t *pde;
1757     pte_t *pgtab;
1758
1759     pde = &pgdir[PDX(va)];
1760     if(*pde & PTE_P){
1761         pgtab = (pte_t*)P2V(PTE_ADDR(*pde));
1762     } else {
1763         if(!alloc || (pgtab = (pte_t*)kalloc()) == 0)
1764             return 0;
1765         // Make sure all those PTE_P bits are zero.
1766         memset(pgtab, 0, PGSIZE);
...
1770         *pde = V2P(pgtab) | PTE_P | PTE_W | PTE_U;
1771     }
1772     return &pgtab[PTX(va)];
1773 }
```



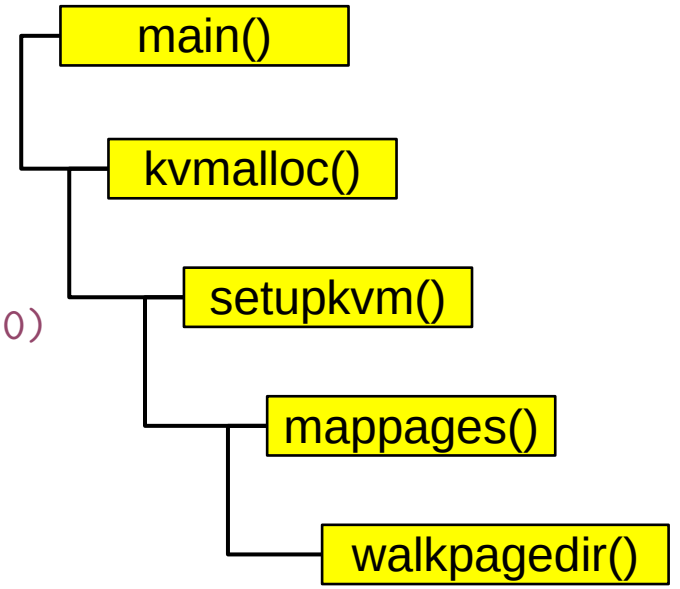
- Page table Level 2 exists
- Return the PTE entry

Return a pointer to PTE



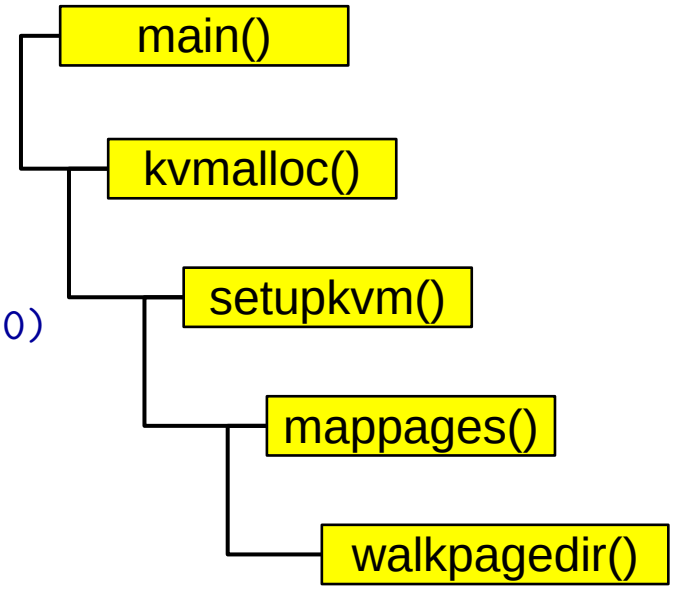
Walk page table

```
1754 walkpgdir(pde_t *pgdir, const void *va, int alloc)
1755 {
1756     pde_t *pde;
1757     pte_t *pgtab;
1758
1759     pde = &pgdir[PDX(va)];
1760     if(*pde & PTE_P){
1761         pgtab = (pte_t*)P2V(PTE_ADDR(*pde));
1762     } else {
1763         if(!alloc || (pgtab = (pte_t*)kalloc()) == 0)
1764             return 0;
1765         // Make sure all those PTE_P bits are zero.
1766         memset(pgtab, 0, PGSIZE);
...
1770         *pde = V2P(pgtab) | PTE_P | PTE_W | PTE_U;
1771     }
1772     return &pgtab[PTX(va)];
1773 }
```



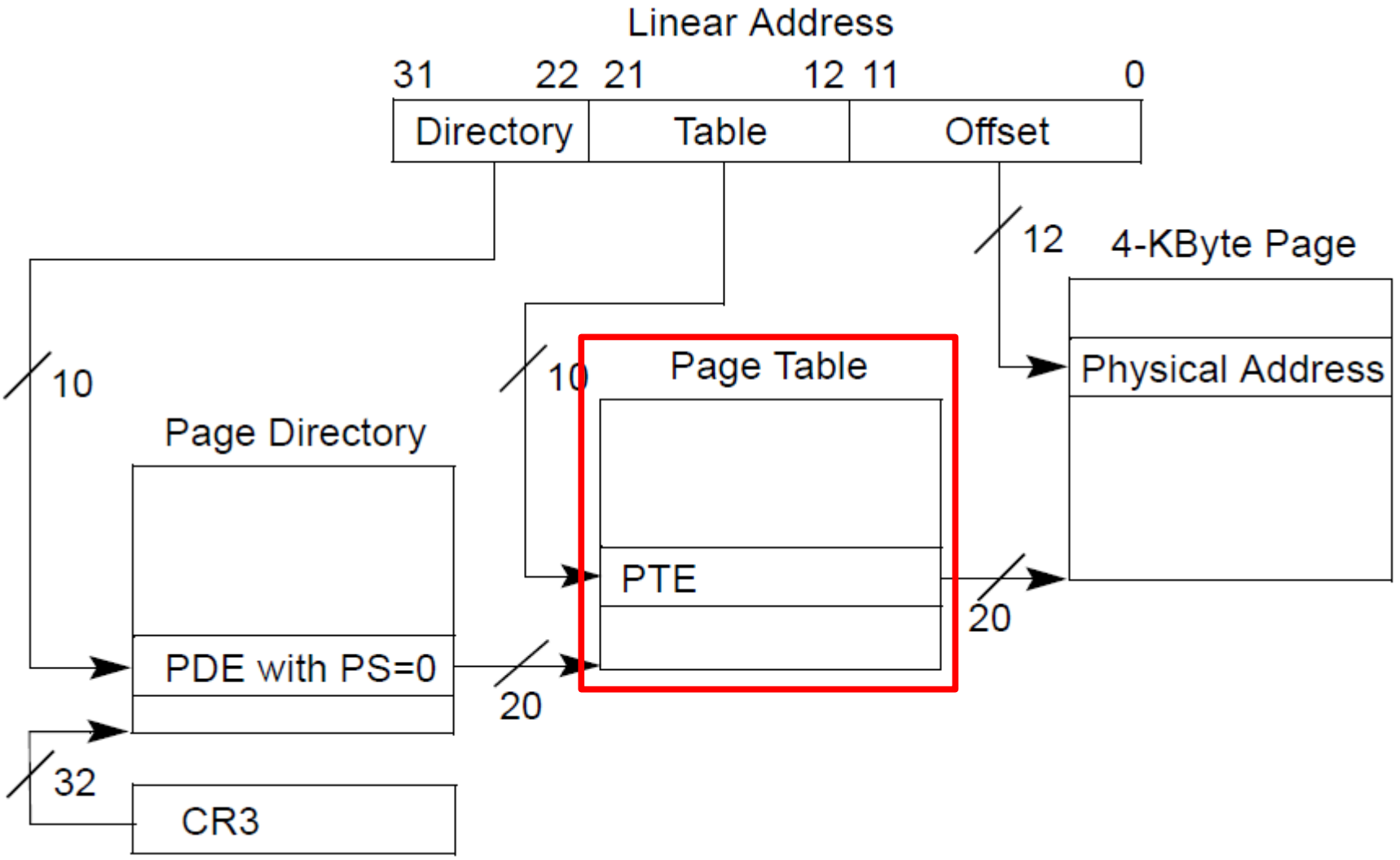
- Page table Level 2 exists
- Return the PTE entry

```
1754 walkpgdir(pde_t *pgdir, const void *va, int alloc)
1755 {
1756     pde_t *pde;
1757     pte_t *pgtab;
1758
1759     pde = &pgdir[PDX(va)];
1760     if(*pde & PTE_P){
1761         pgtab = (pte_t*)P2V(PTE_ADDR(*pde));
1762     } else {
1763         if(!alloc || (pgtab = (pte_t*)kalloc()) == 0)
1764             return 0;
1765         // Make sure all those PTE_P bits are zero.
1766         memset(pgtab, 0, PGSIZE);
...
1770         *pde = V2P(pgtab) | PTE_P | PTE_W | PTE_U;
1771     }
1772     return &pgtab[PTX(va)];
1773 }
```

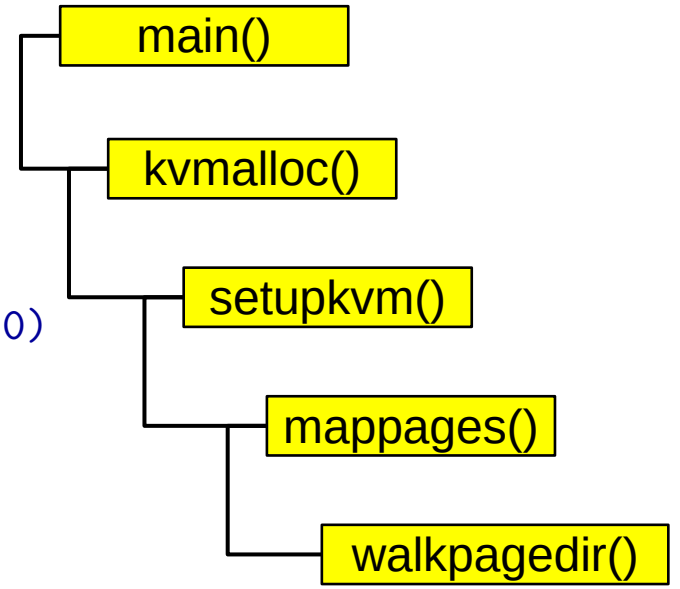


- Page table Level 2 does not exist
  - Allocate one

Level 2 page table is not allocated

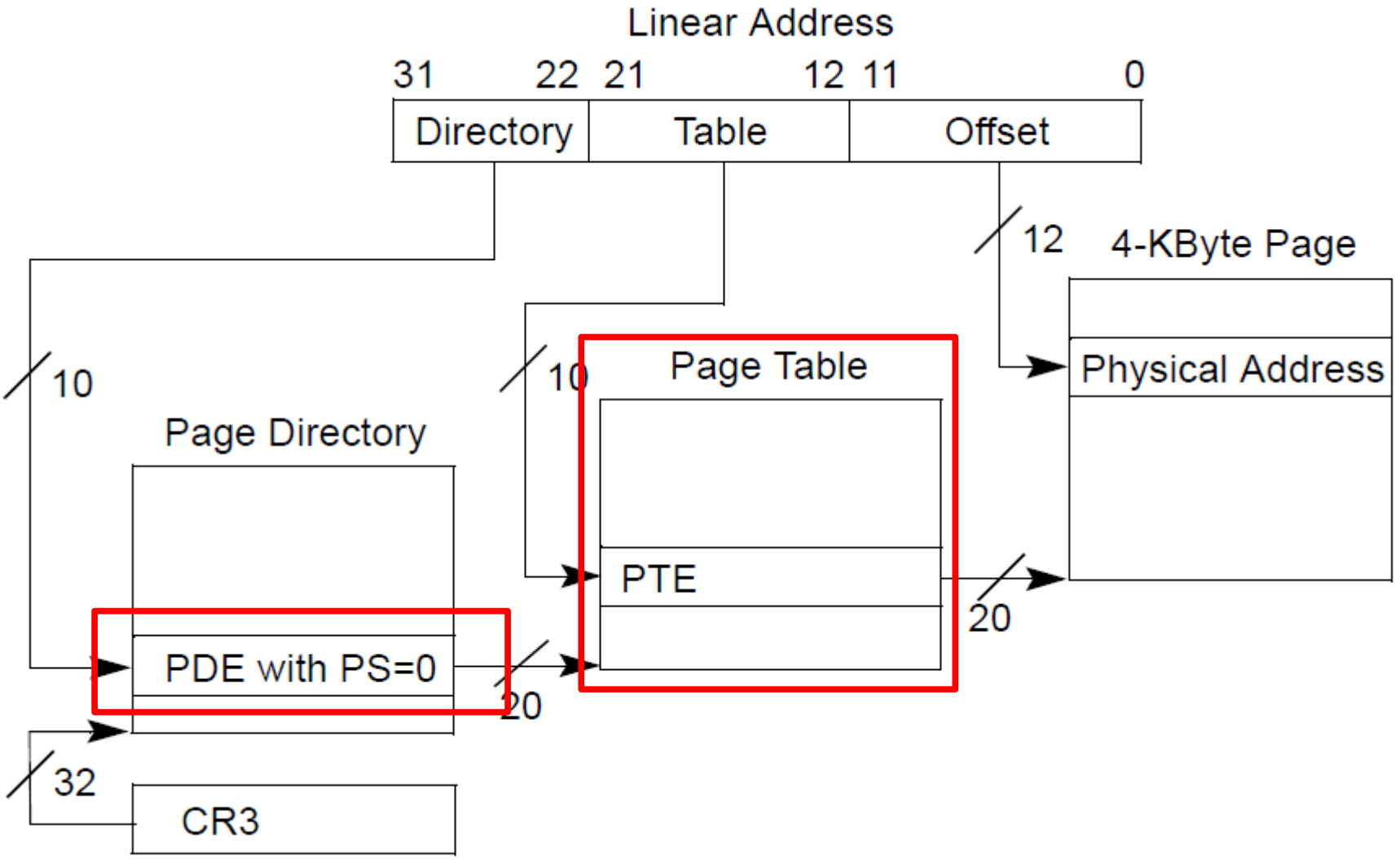


```
1754 walkpgdir(pde_t *pgdir, const void *va, int alloc)
1755 {
1756     pde_t *pde;
1757     pte_t *pgtab;
1758
1759     pde = &pgdir[PDX(va)];
1760     if(*pde & PTE_P){
1761         pgtab = (pte_t*)P2V(PTE_ADDR(*pde));
1762     } else {
1763         if(!alloc || (pgtab = (pte_t*)kalloc()) == 0)
1764             return 0;
1765         // Make sure all those PTE_P bits are zero.
1766         memset(pgtab, 0, PGSIZE);
...
1770         *pde = V2P(pgtab) | PTE_P | PTE_W | PTE_U;
1771     }
1772     return &pgtab[PTX(va)];
1773 }
```



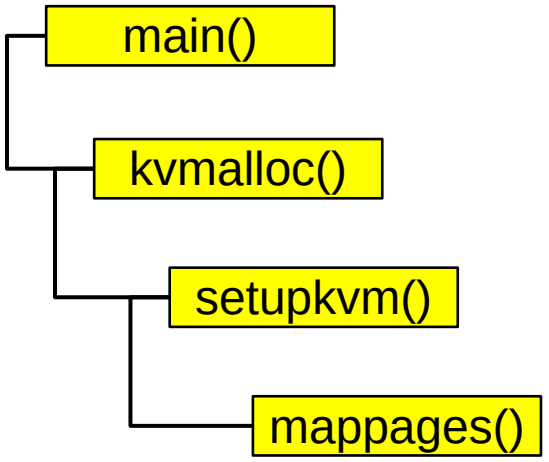
- Allocate the new page
- Initialize it with zeros
- Update the page directory entry (`*pde`)

Now the level 2 page table is allocated



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

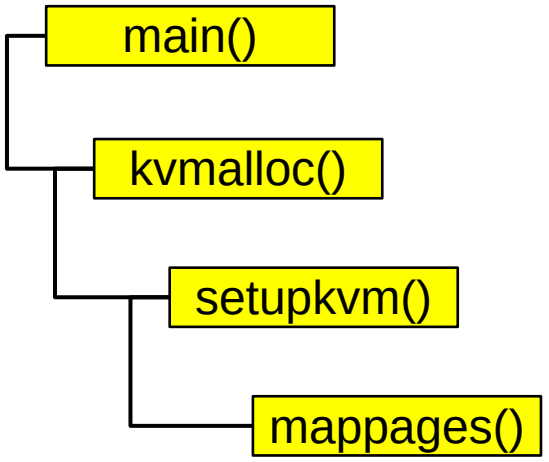
```
1779 mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm)
1780 {
1781   char *a, *last;
1782   pte_t *pte;
1783
1784   a = (char*)PGROUNDDOWN((uint)va);
1785   last = (char*)PGROUNDDOWN(((uint)va) + size - 1);
1786   for(;;){
1787     if((pte = walkpgdir(pgdir, a, 1)) == 0)
1788       return -1;
1789     if(*pte & PTE_P)
1790       panic("remap");
1791     *pte = pa | perm | PTE_P;
1792     if(a == last)
1793       break;
1794     a += PGSIZE;
1795     pa += PGSIZE;
1796   }
1797   return 0;
1798 }
```



Remember we just discussed

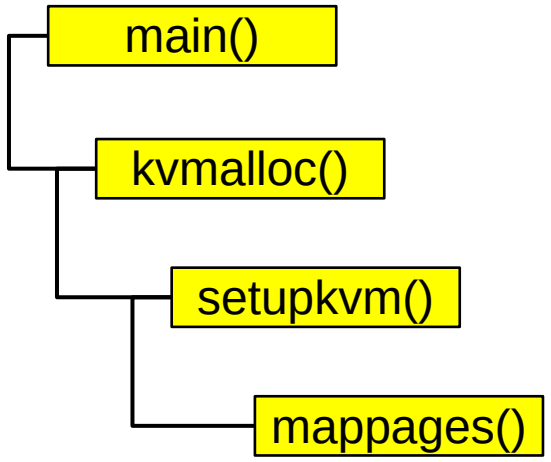
`walkpgdir()`

```
1779 mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm)
1780 {
1781   char *a, *last;
1782   pte_t *pte;
1783
1784   a = (char*)PGROUNDDOWN((uint)va);
1785   last = (char*)PGROUNDDOWN(((uint)va) + size - 1);
1786   for(;;){
1787     if((pte = walkpgdir(pgdir, a, 1)) == 0)
1788       return -1;
1789     if(*pte & PTE_P)
1790       panic("remap");
1791     *pte = pa | perm | PTE_P;
1792     if(a == last)
1793       break;
1794     a += PGSIZE;
1795     pa += PGSIZE;
1796   }
1797   return 0;
1798 }
```



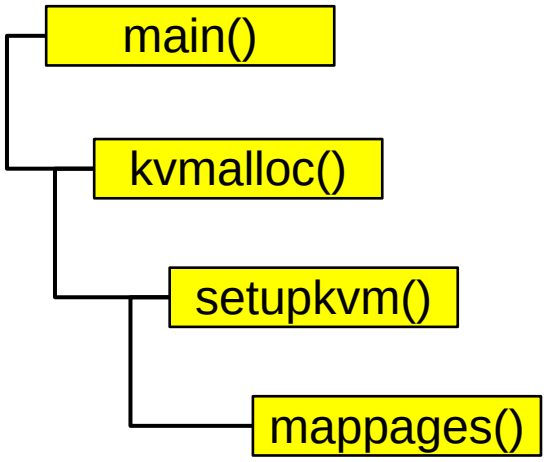
Page present (PTE_P) – panic

```
1779 mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm)
1780 {
1781   char *a, *last;
1782   pte_t *pte;
1783
1784   a = (char*)PGROUNDDOWN((uint)va);
1785   last = (char*)PGROUNDDOWN(((uint)va) + size - 1);
1786   for(;;){
1787     if((pte = walkpgdir(pgdir, a, 1)) == 0)
1788       return -1;
1789     if(*pte & PTE_P)
1790       panic("remap");
1791     *pte = pa | perm | PTE_P;
1792     if(a == last)
1793       break;
1794     a += PGSIZE;
1795     pa += PGSIZE;
1796   }
1797   return 0;
1798 }
```



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

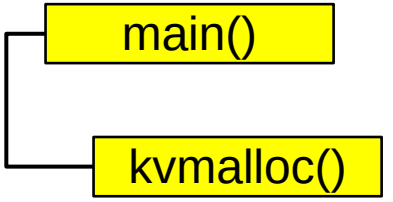
```
1779 mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm)
1780 {
1781   char *a, *last;
1782   pte_t *pte;
1783
1784   a = (char*)PGROUNDDOWN((uint)va);
1785   last = (char*)PGROUNDDOWN(((uint)va) + size - 1);
1786   for(;;){
1787     if((pte = walkpgdir(pgdir, a, 1)) == 0)
1788       return -1;
1789     if(*pte & PTE_P)
1790       panic("remap");
1791     *pte = pa | perm | PTE_P;
1792     if(a == last)
1793       break;
1794     a += PGSIZE;
1795     pa += PGSIZE;
1796   }
1797   return 0;
1798 }
```



- Move to the next page

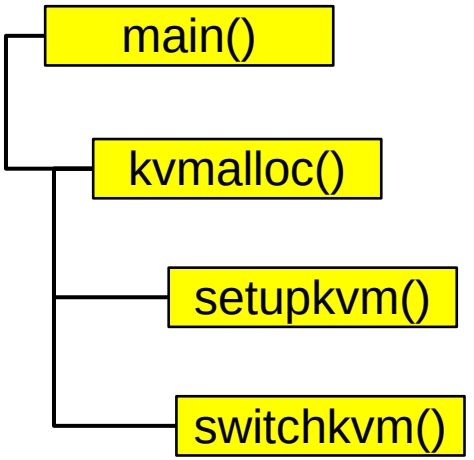
kvmalloc()

```
1757 kvmalloc(void)
1758 {
1759   kpgdir = setupkvm();
1760   switchkvm();
1761 }
```



Switch to the new page table

```
1765 void
1766 switchkvm(void)
1767 {
1768   lcr3(v2p(kpgdir));
1769 }
```



Recap

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


WHERE IS JEFF!!!!

```
1317 main(void)
1318 {
1319   kinit1(end, P2V(4*1024*1024)); // phys page allocator
1320   kvmalloc(); // kernel page table
1321   mpinit(); // detect other processors
1322   lapicinit();     // interrupt controller
1323   seginit();       // segment descriptors
1324   cprintf("\ncpu%d: starting xv6\n\n", cpunum());
1325   picinit();       // another interrupt controller
1326   ioapicinit();    // another interrupt controller
1327   consoleinit();   // console hardware
1328   uartinit();      // serial port
1329   pinit();         // process table
1330   tvinit();        // trap vectors
1331   binit();         // buffer cache
1332   fileinit();      // file table
1333   ideinit();       // disk
1334   if(!ismp)
1335     timerinit();   // uniprocessor timer
1336   startothers();   // start other processors
1337   kinit2(P2V(4*1024*1024), P2V(PHYSTOP)); // must come after startothers()
1338   userinit();      // first user process
1339   mpmain();        // finish this processor's setup
1340 }
```

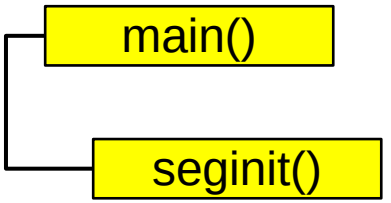
main()

```
1317 main(void)
1318 {
1319   kinit1(end, P2V(4*1024*1024)); // phys page allocator
1320   kvmalloc(); // kernel page table
1321   mpinit(); // detect other processors
1322   lapicinit();     // interrupt controller
1323   seginit();       // segment descriptors
1324   cprintf("\ncpu%d: starting xv6\n\n", cpunum());
1325   picinit();       // another interrupt controller
1326   ioapicinit();    // another interrupt controller
1327   consoleinit();   // console hardware
1328   uartinit();      // serial port
1329   pinit();         // process table
1330   tvinit();        // trap vectors
1331   binit();         // buffer cache
1332   fileinit();      // file table
1333   ideinit();       // disk
1334   if(!ismp)
1335     timerinit();   // uniprocessor timer
1336   startothers();   // start other processors
1337   kinit2(P2V(4*1024*1024), P2V(PHYSTOP)); // must come after startothers()
1338   userinit();      // first user process
1339   mpmain();        // finish this processor's setup
1340 }
```

main()

Initialize GDT

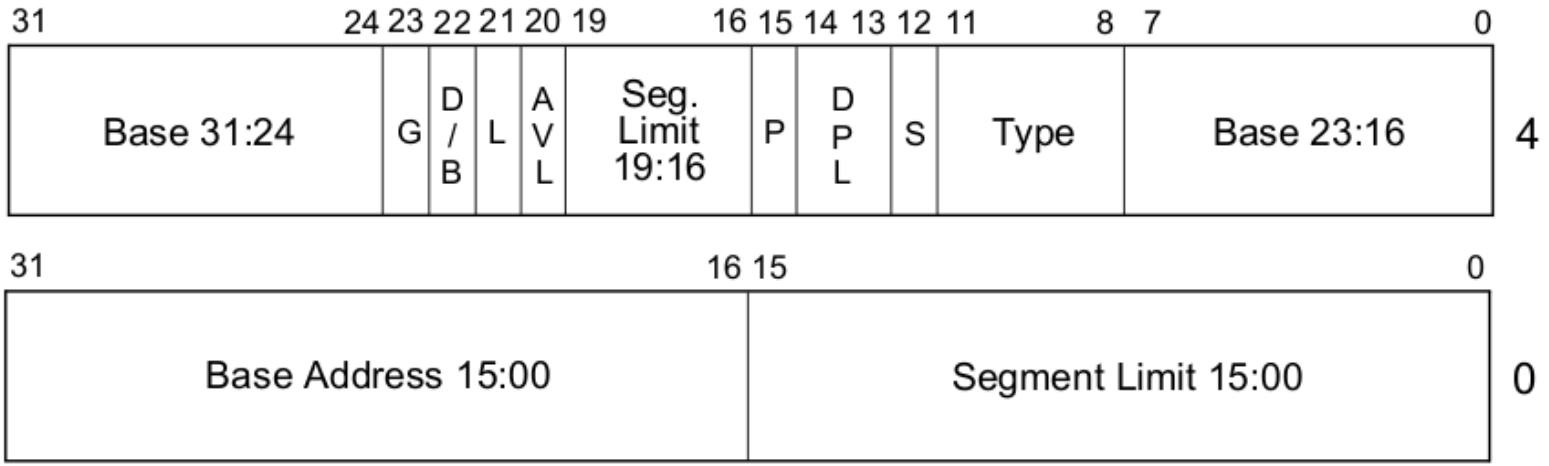
```
1712 // Set up CPU's kernel segment descriptors.
1713 // Run once on entry on each CPU.
1714 void
1715 seginit(void)
1716 {
1717   struct cpu *c;
1718
1719   // Map "logical" addresses to virtual addresses using identity map.
1720   // Cannot share a CODE descriptor for both kernel and user
1721   // because it would have to have DPL_USR, but the CPU forbids
1722   // an interrupt from CPL=0 to DPL=3.
1723   c = &cpus[cpuid()];
1724   c->gdt[SEG_KCODE] = SEG(STA_X|STA_R, 0, 0xffffffff, 0);
1725   c->gdt[SEG_KDATA] = SEG(STA_W, 0, 0xffffffff, 0);
1726   c->gdt[SEG_UCODE] = SEG(STA_X|STA_R, 0, 0xffffffff, DPL_USER);
1727   c->gdt[SEG_UDATA] = SEG(STA_W, 0, 0xffffffff, DPL_USER);
1728   lgdt(c->gdt, sizeof(c->gdt));
1729 }
```



Struct CPU

```
2300 // Per-CPU state
2301 struct cpu {
2302   uchar apicid;                // Local APIC ID
2303   struct context *scheduler;   // swtch() here to enter scheduler
2304   struct taskstate ts;         // Used by x86 to find stack for interrupt
2305   struct segdesc gdt[NSEGS];   // x86 global descriptor table
2306   volatile uint started;       // Has the CPU started?
2307   int ncli;                    // Depth of pushcli nesting.
2308   int intena;                  // Were interrupts enabled before pushcli?
2309   struct proc *proc;           // The process running on this cpu or null
2310 };
2311
2312 extern struct cpu cpus[NCPU];
```

Segment descriptor



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

```
0724 // Segment Descriptor
0725 struct segdesc {
0726   uint lim_15_0 : 16;  // Low bits of segment limit
0727   uint base_15_0 : 16; // Low bits of segment base address
0728   uint base_23_16 : 8; // Middle bits of segment base address
0729   uint type : 4;       // Segment type (see STS_ constants)
0730   uint s : 1;          // 0 = system, 1 = application
0731   uint dpl : 2;        // Descriptor Privilege Level
0732   uint p : 1;          // Present
0733   uint lim_19_16 : 4;  // High bits of segment limit
0734   uint avl : 1;        // Unused (available for software use)
0735   uint rsv1 : 1;       // Reserved
0736   uint db : 1;         // 0 = 16-bit segment, 1 = 32-bit segment
0737   uint g : 1;          // Granularity: limit scaled by 4K when set
0738   uint base_31_24 : 8; // High bits of segment base address
0739 };
```

Thank you!
(Next time: interrupts!)