CS248P: Operating Systems

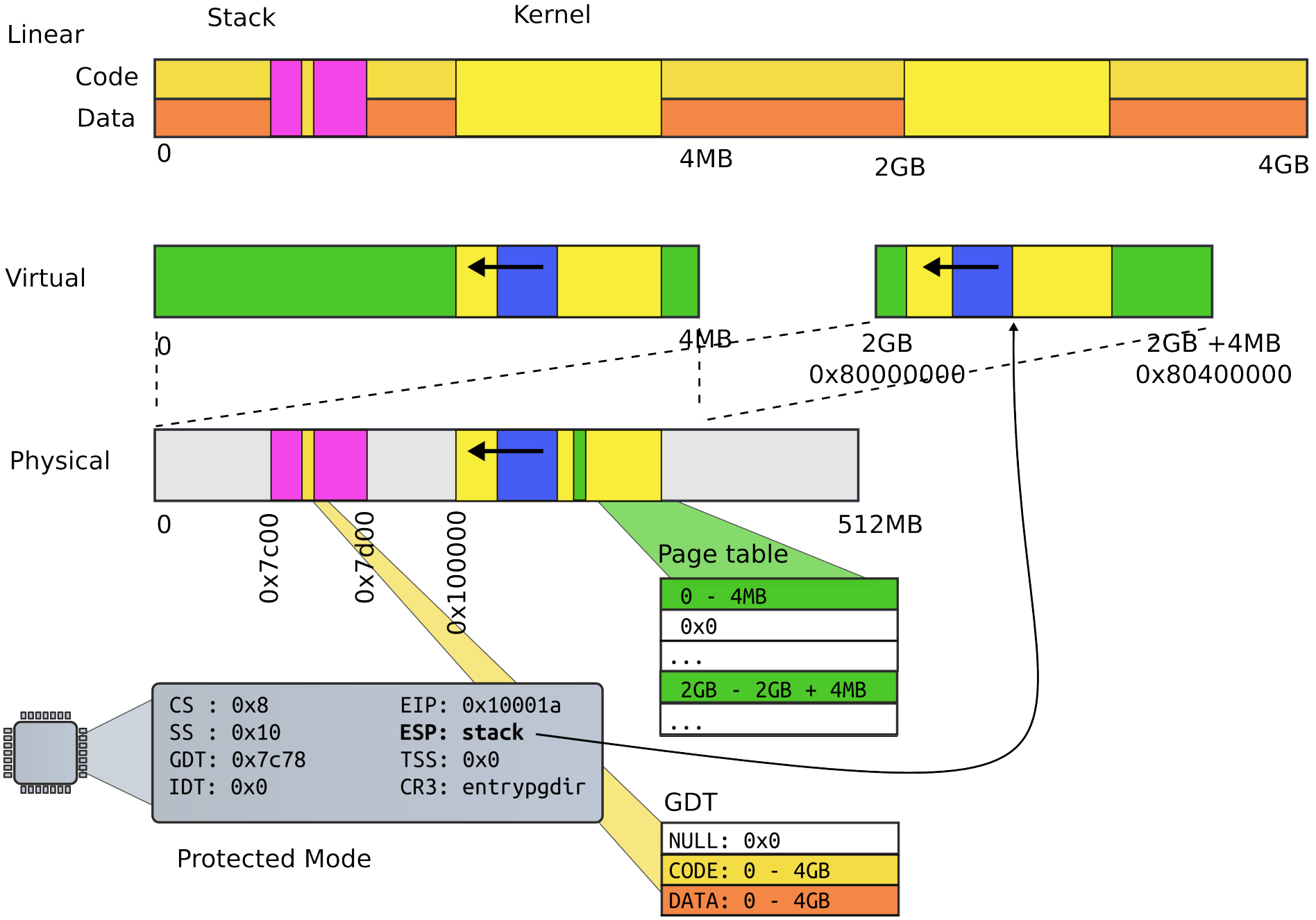
Lecture 8: Kernel Initialization

Anton Burtsev
February, 2018

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

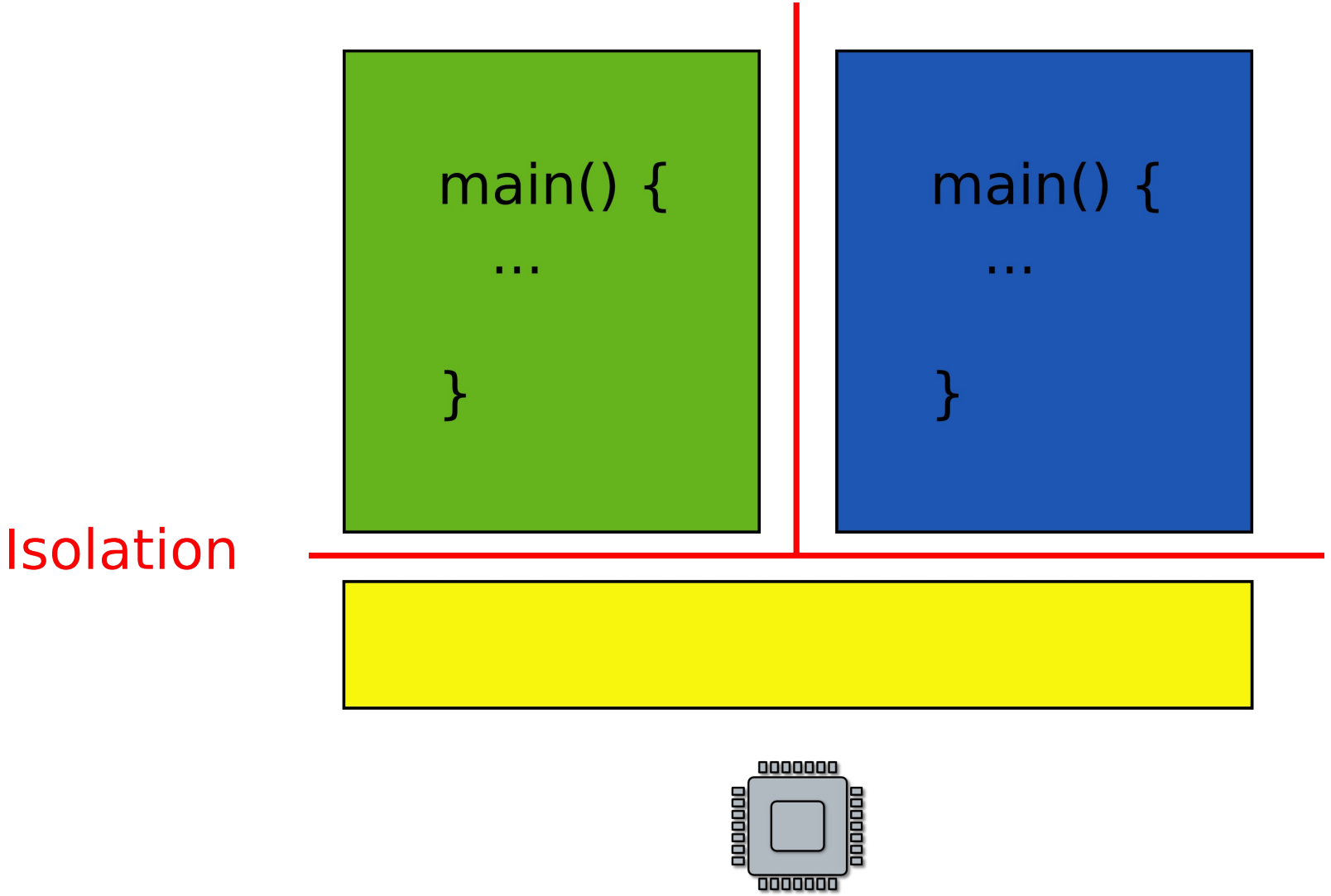


Running in main()

```
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)

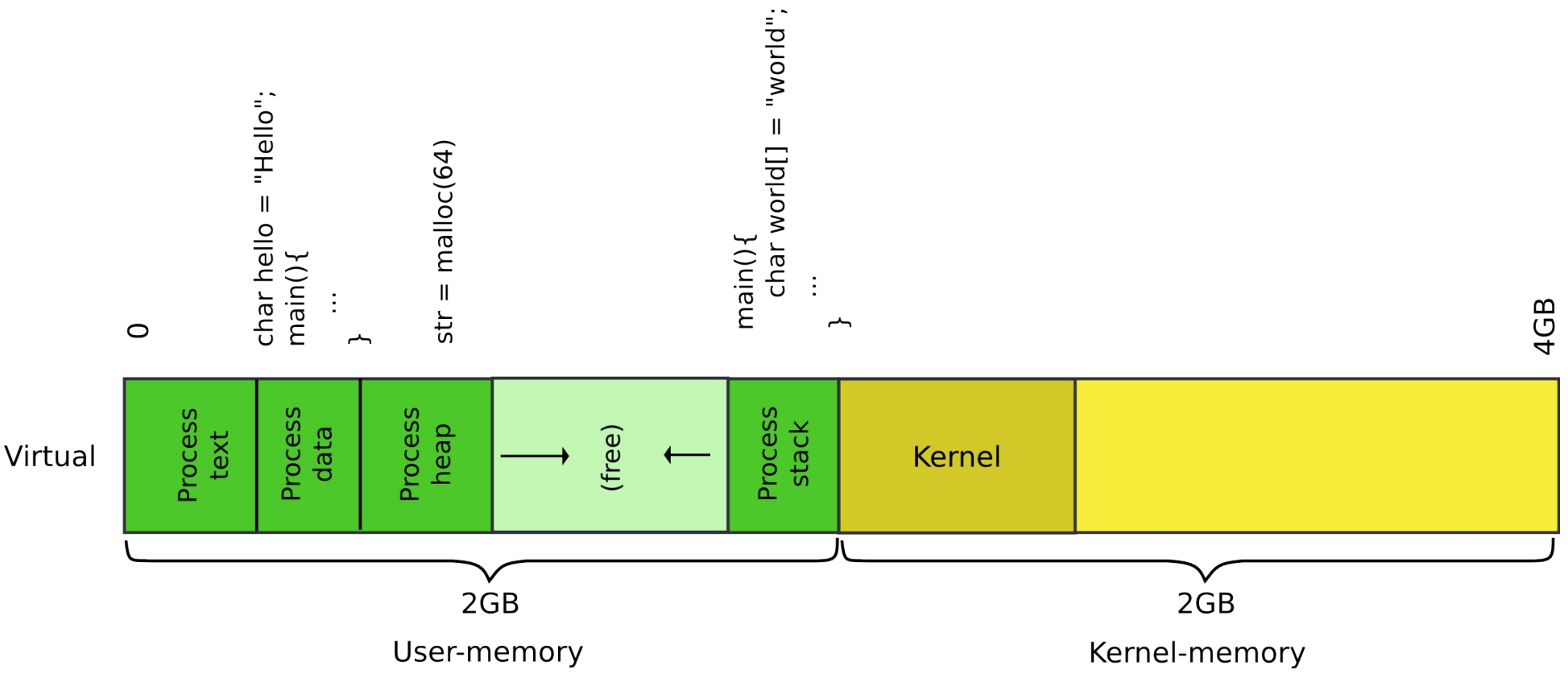


But what is a process?

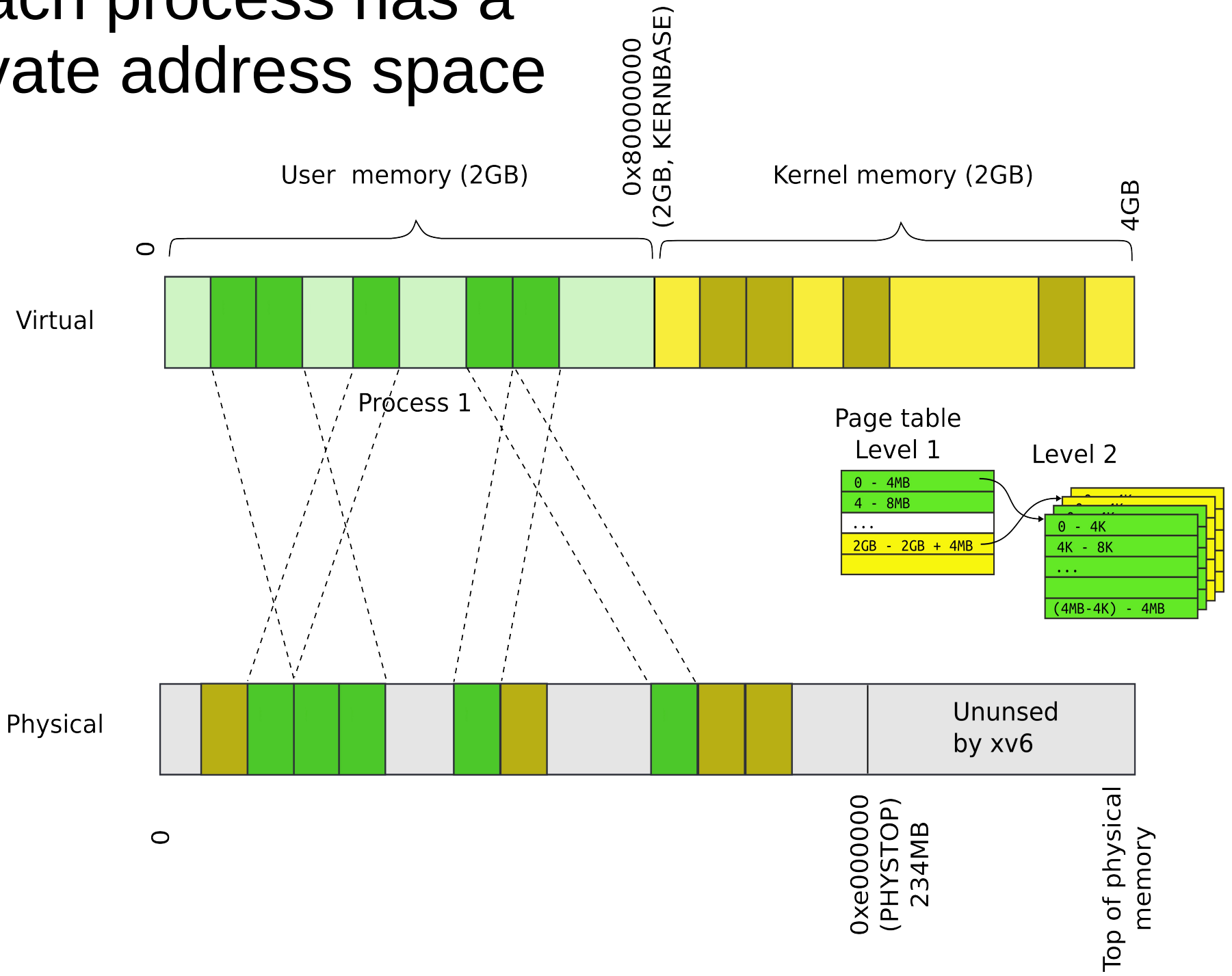
A couple of requirements

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

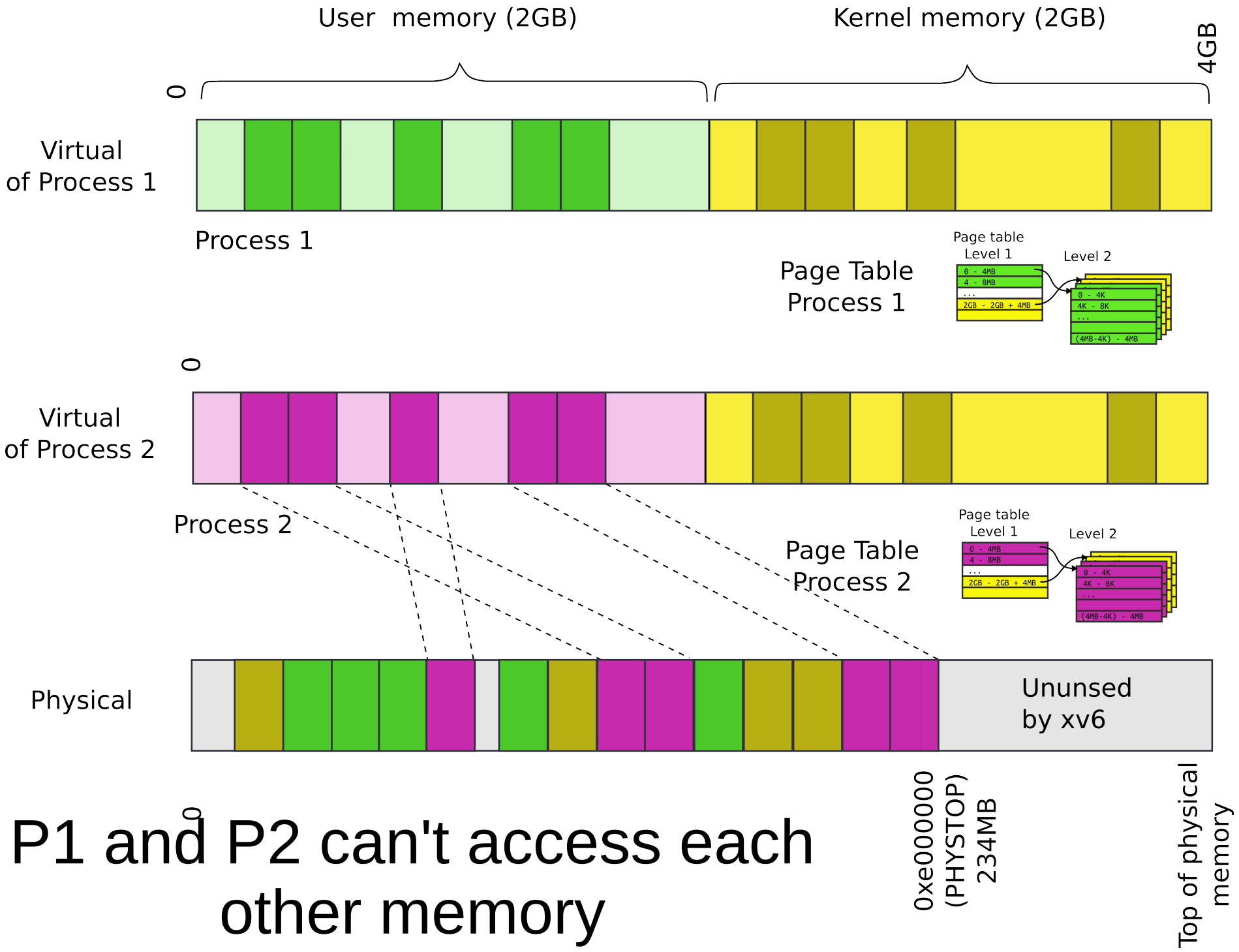


Each process has a private address space



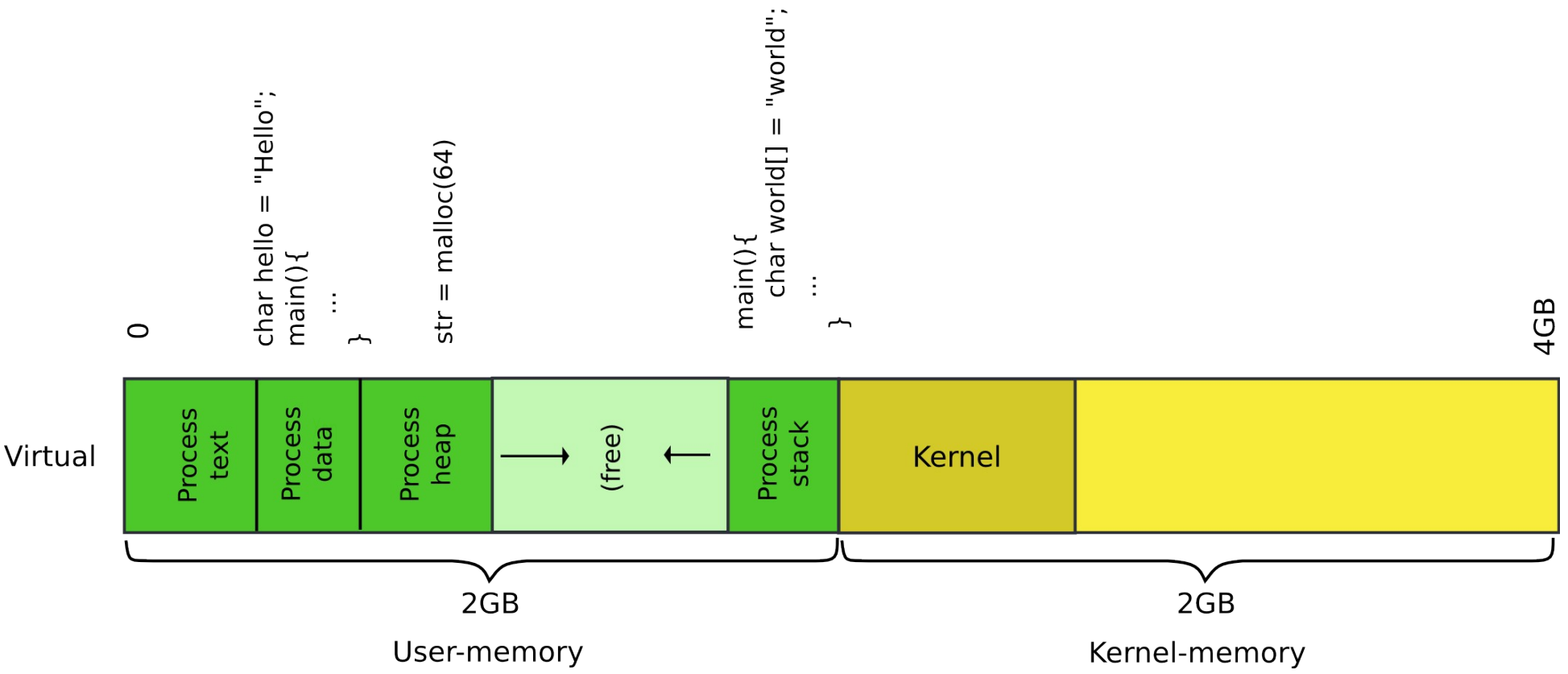
Each process maps the kernel

- 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!**

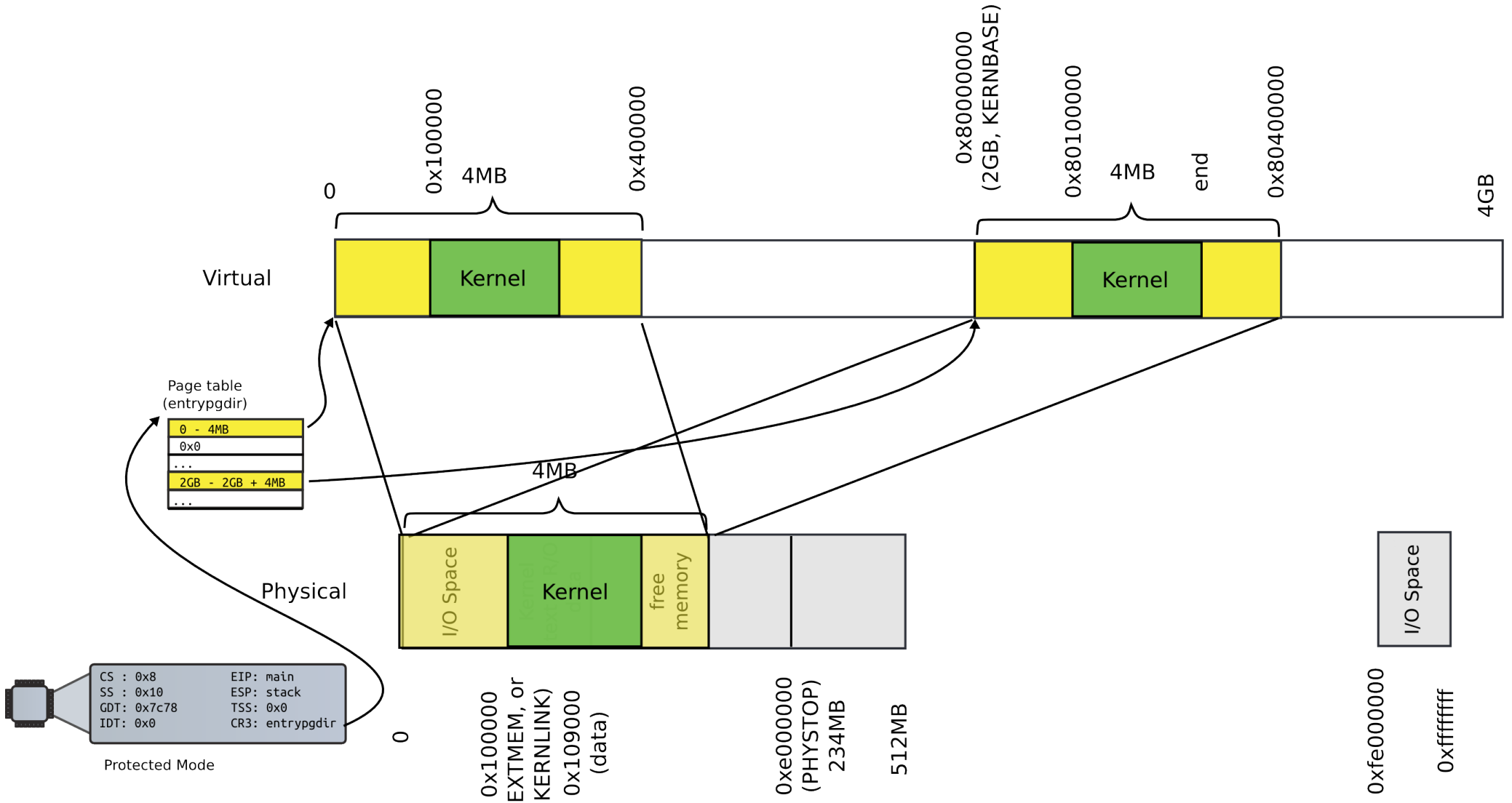


P1 and P2 can't access each other memory

Our goal: 2GB/2GB address space



Memory after boot



Outline

- Create the kernel address space
- Create the first process
  - User address space
- Implement fork() and exec()
  - To create other processes

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

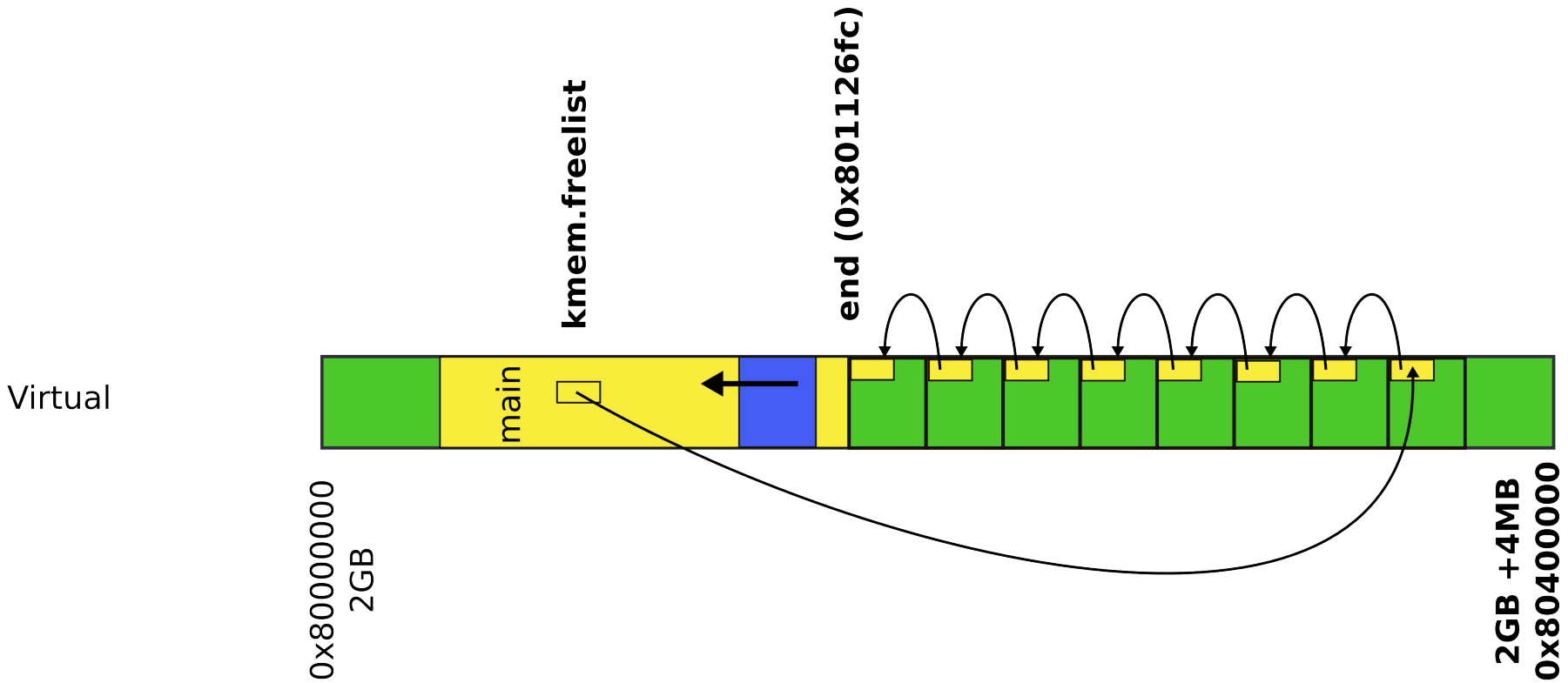
- But to create page tables we need memory
  - Where can it come from?

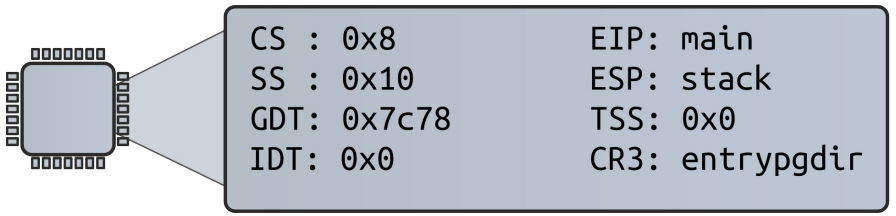
Simple memory allocator

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

What can it look like?

Page allocator

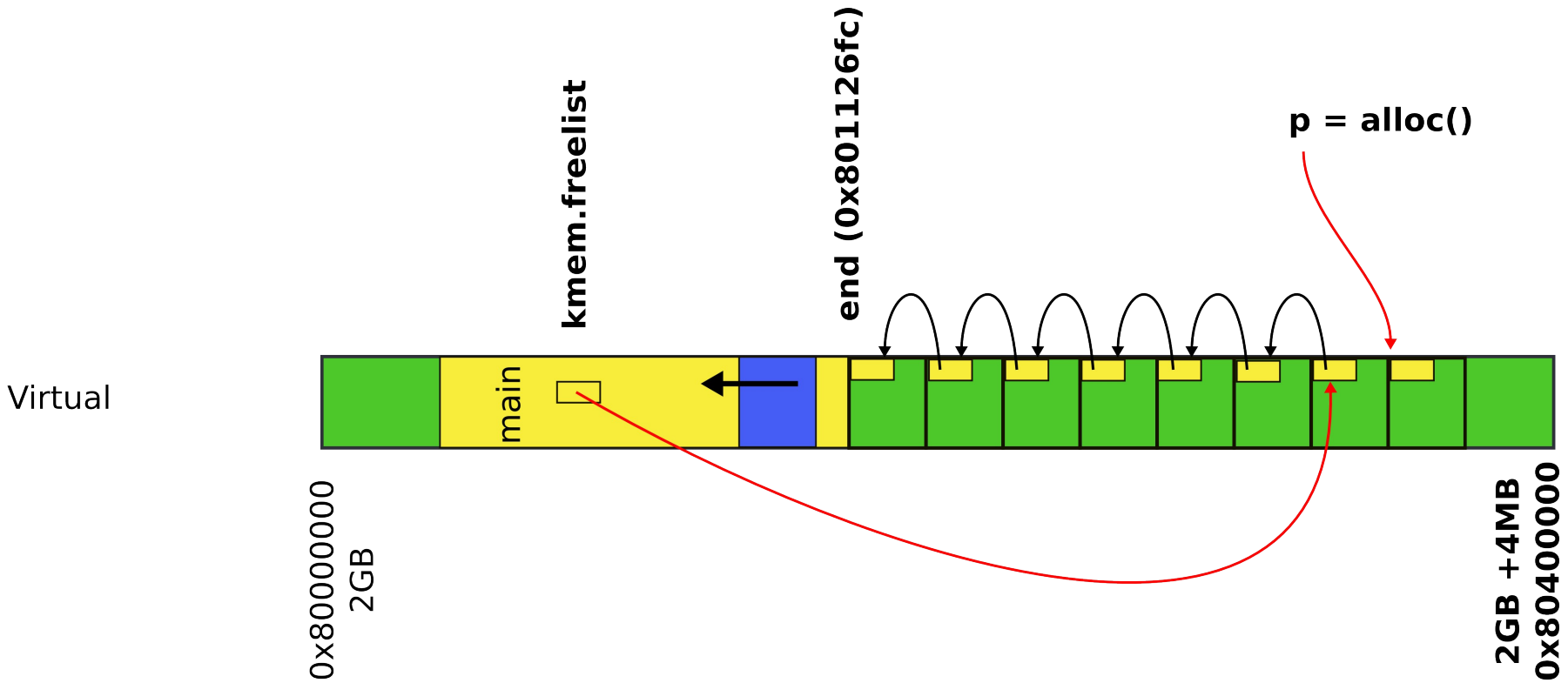


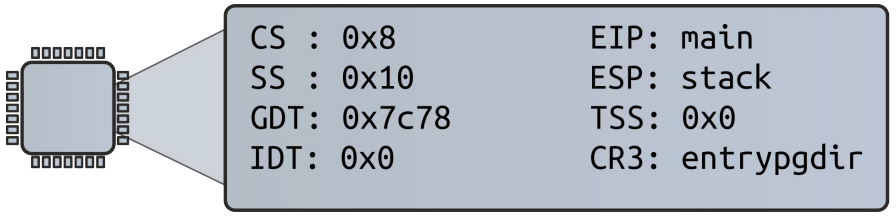


Protected Mode

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

Page allocator

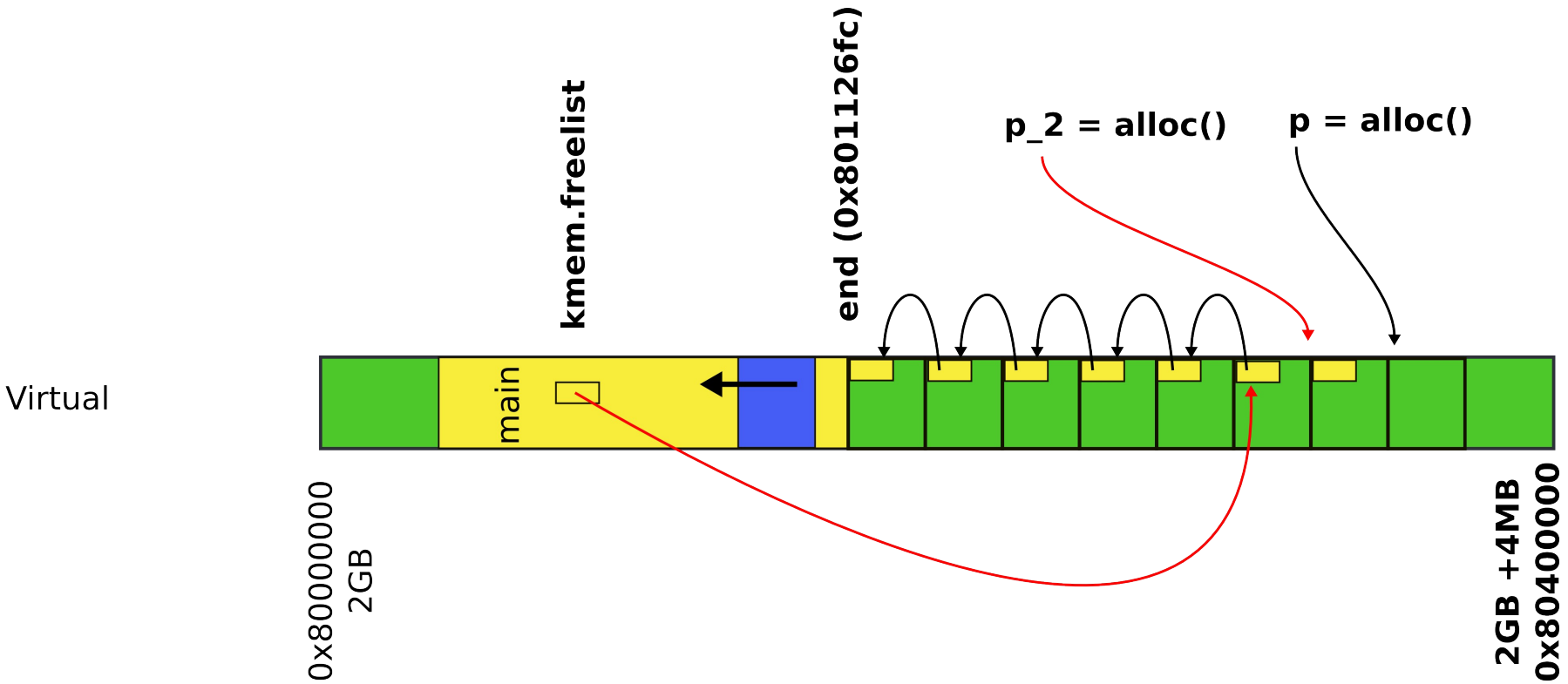


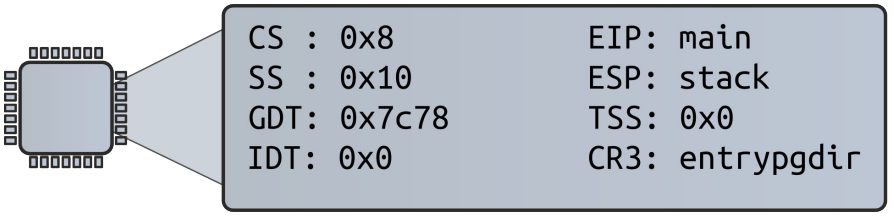


Protected Mode

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

Page allocator

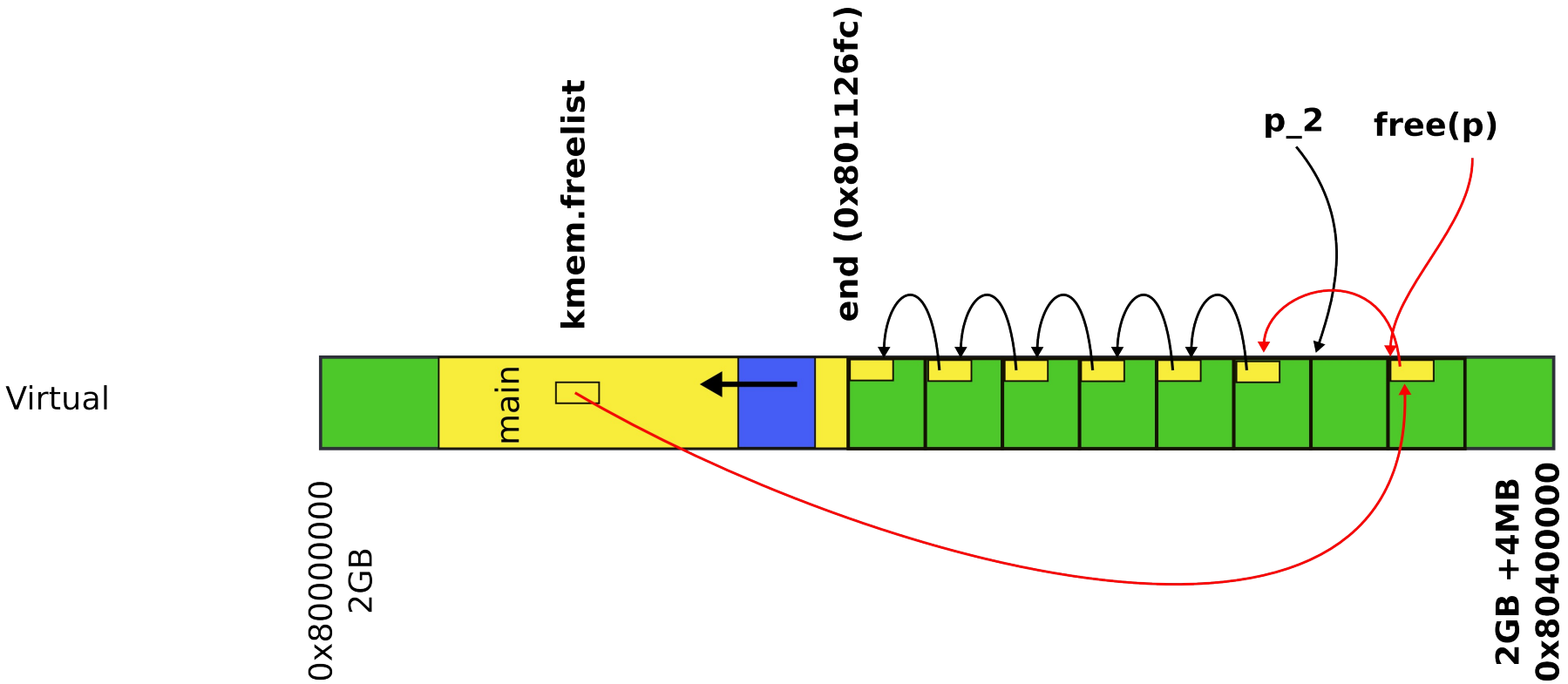


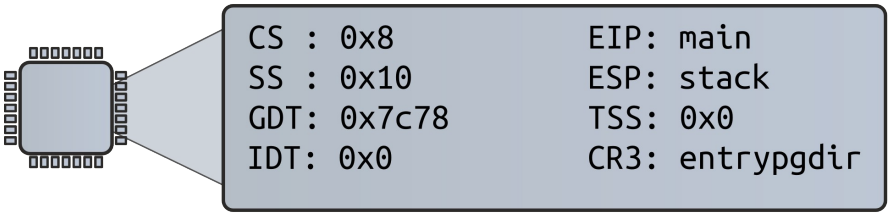


Protected Mode

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

Page allocator





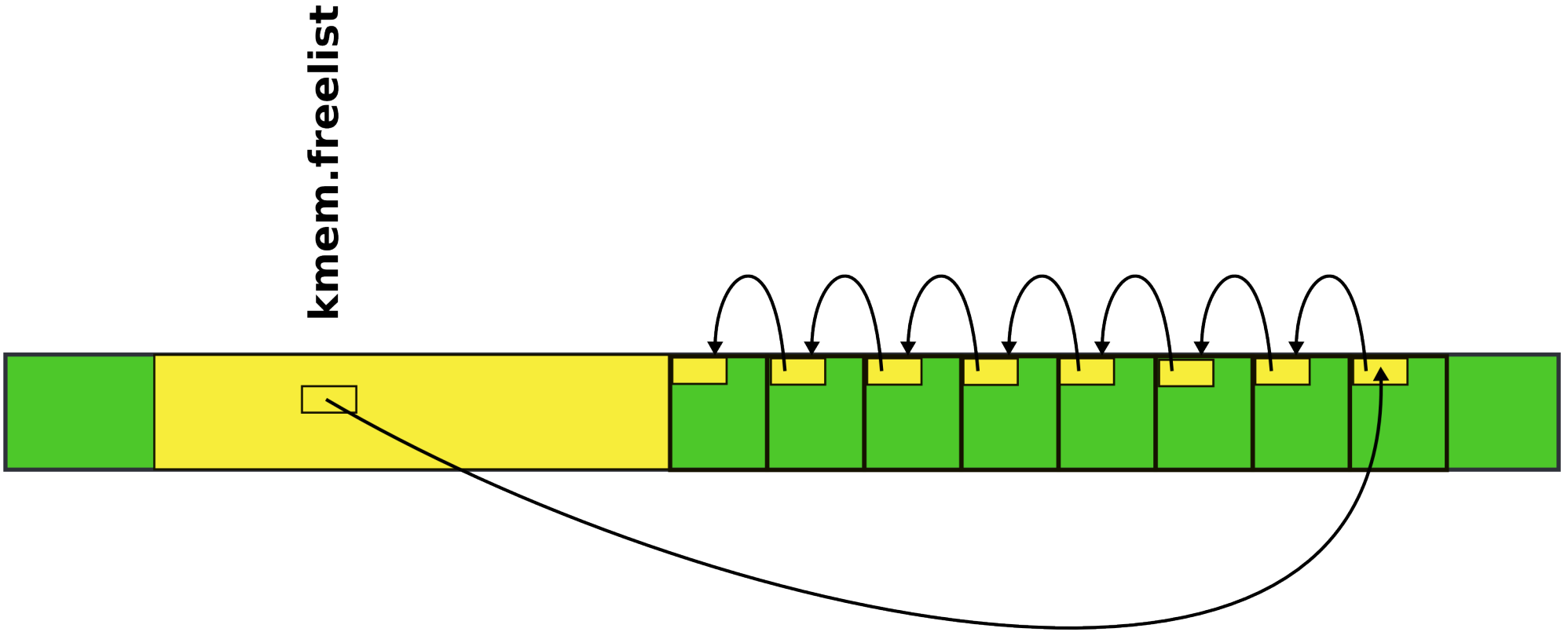
Protected Mode

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

kalloc() - kernel allocator

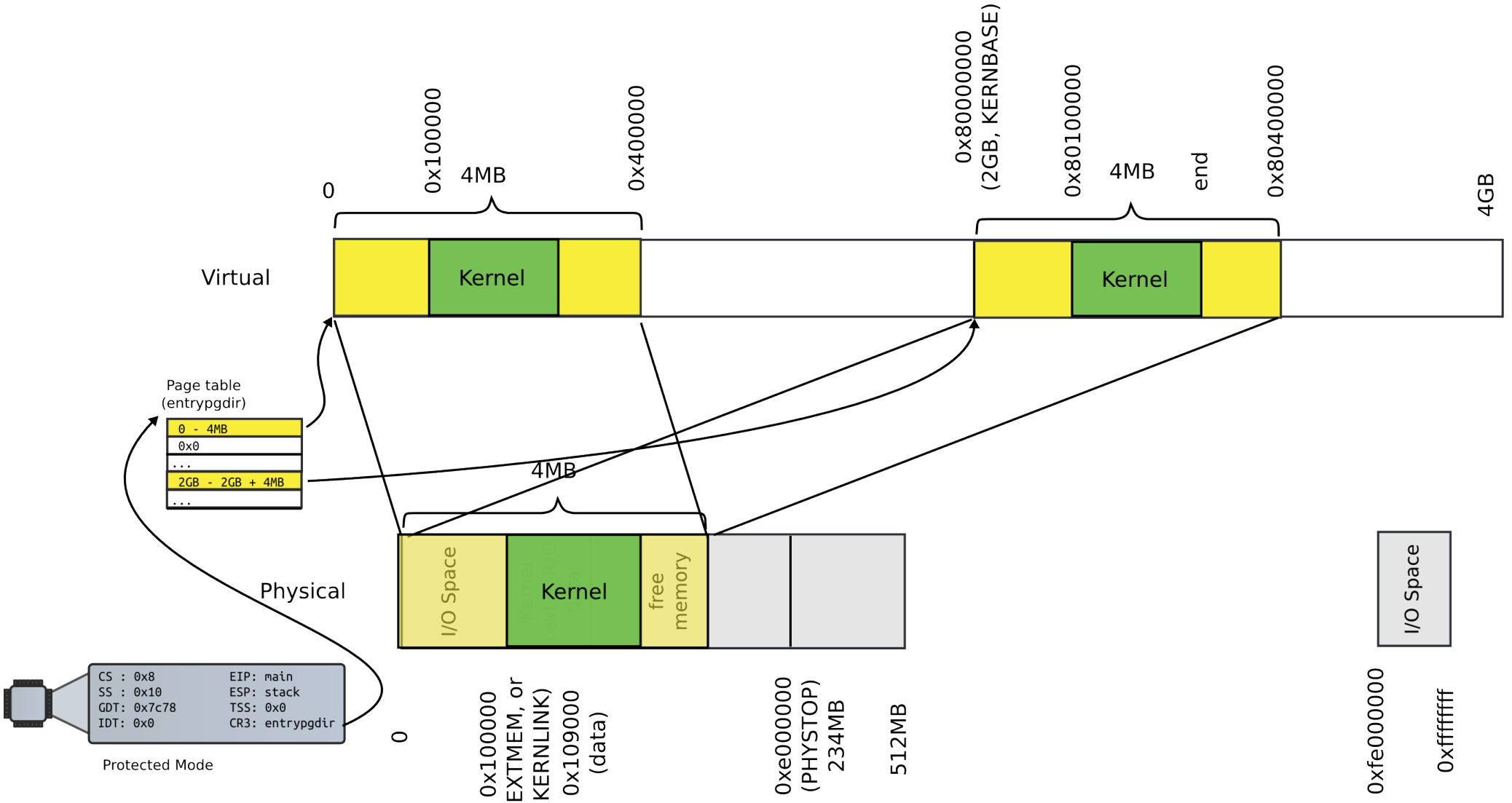
```
3087 char*
3088 kalloc(void)
3089 {
3080   struct run *r;
...
3094   r = kmem.freelist;
3095   if(r)
3096     kmem.freelist = r->next;
…
3099   return (char*)r;
3099 }
```

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

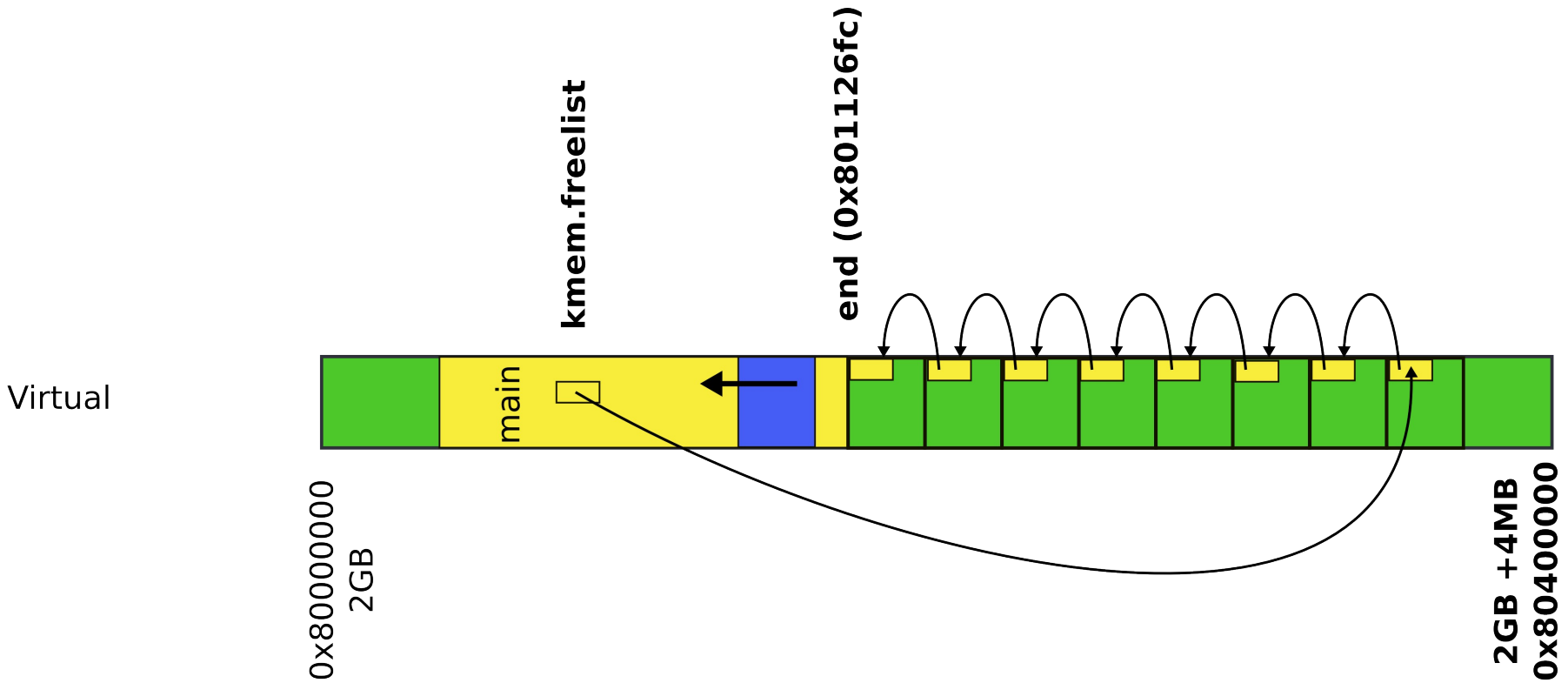


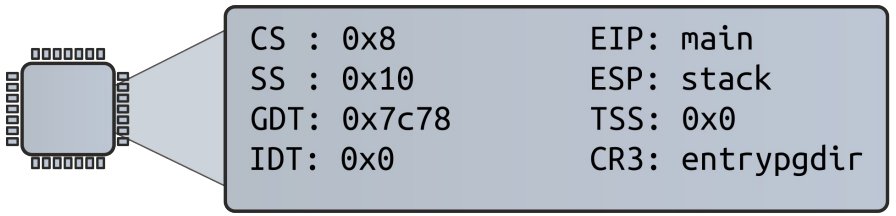
- Where can we get memory to keep the list itself?

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



Donate this free memory to the allocator





Protected Mode

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 }
```

Back to kinit1()

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


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 }
```

Wait! 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[];
```

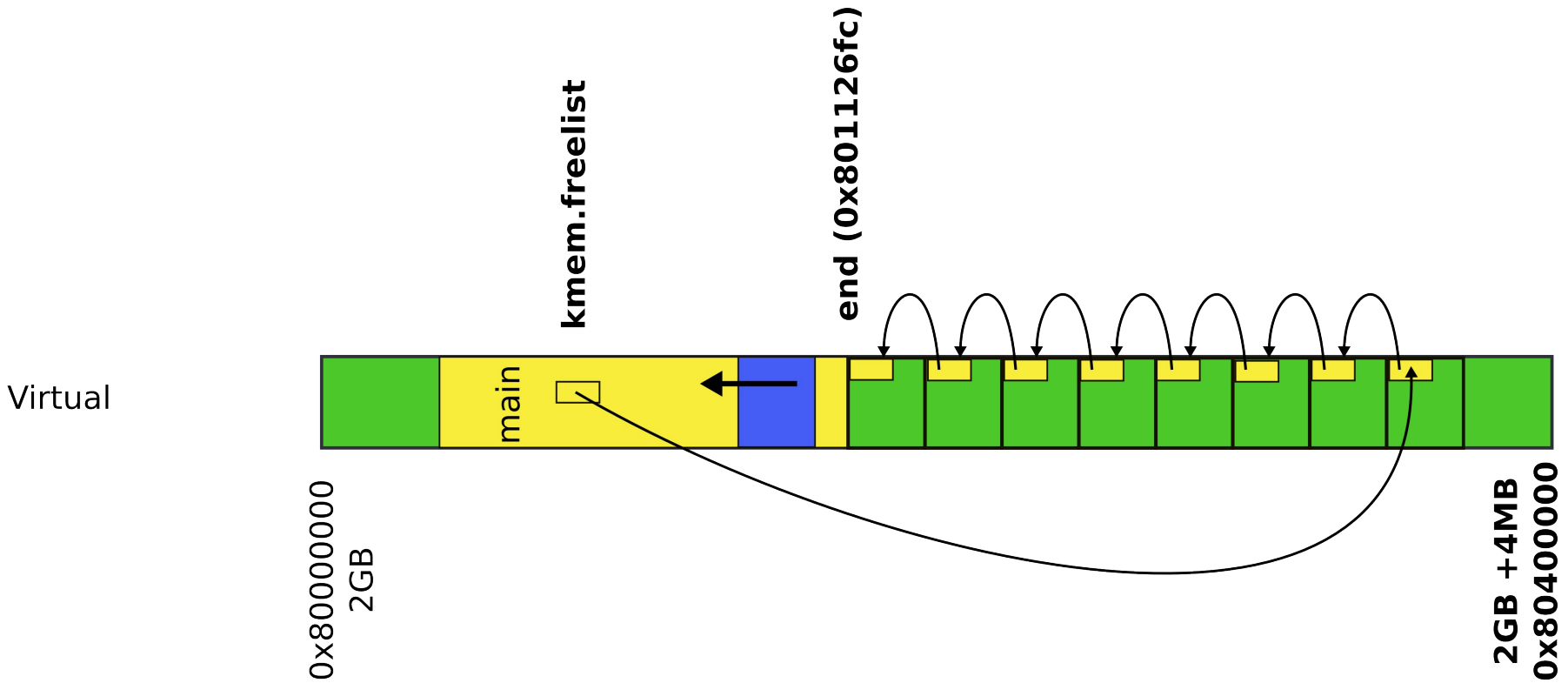
Wait! Where do we start?

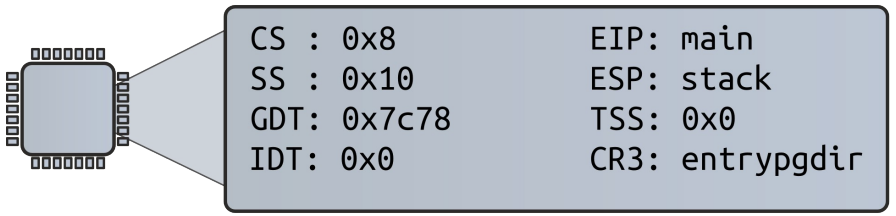
```
1316 int
1317 main(void)
1318 {
1319     kinit1(end, P2V(4*1024*1024)); // phys page allocator
1320     kvmalloc(); // kernel page table
1321     mpinit(); // detect other processors
```

- 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

Conclusion

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

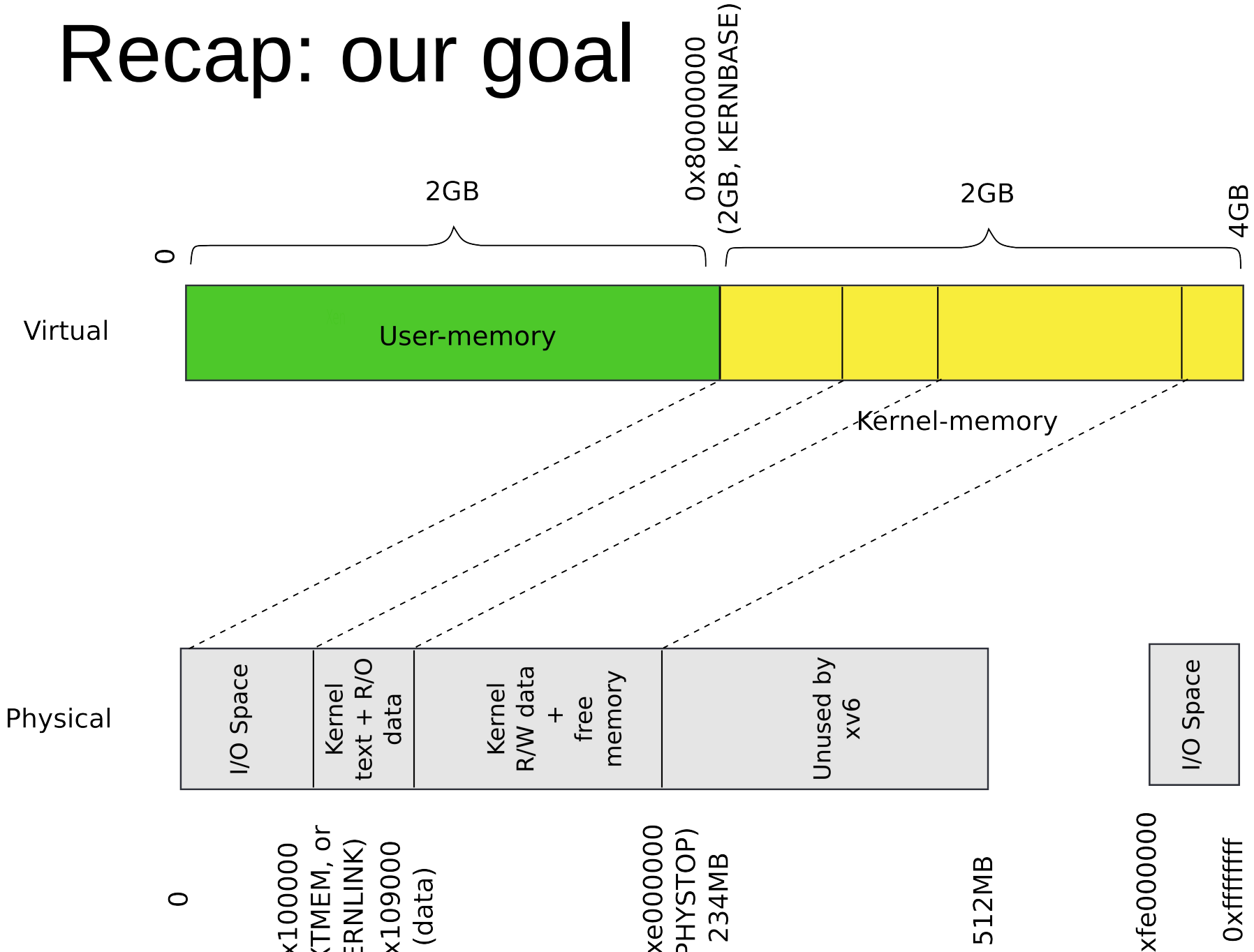
Kernel address space

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?

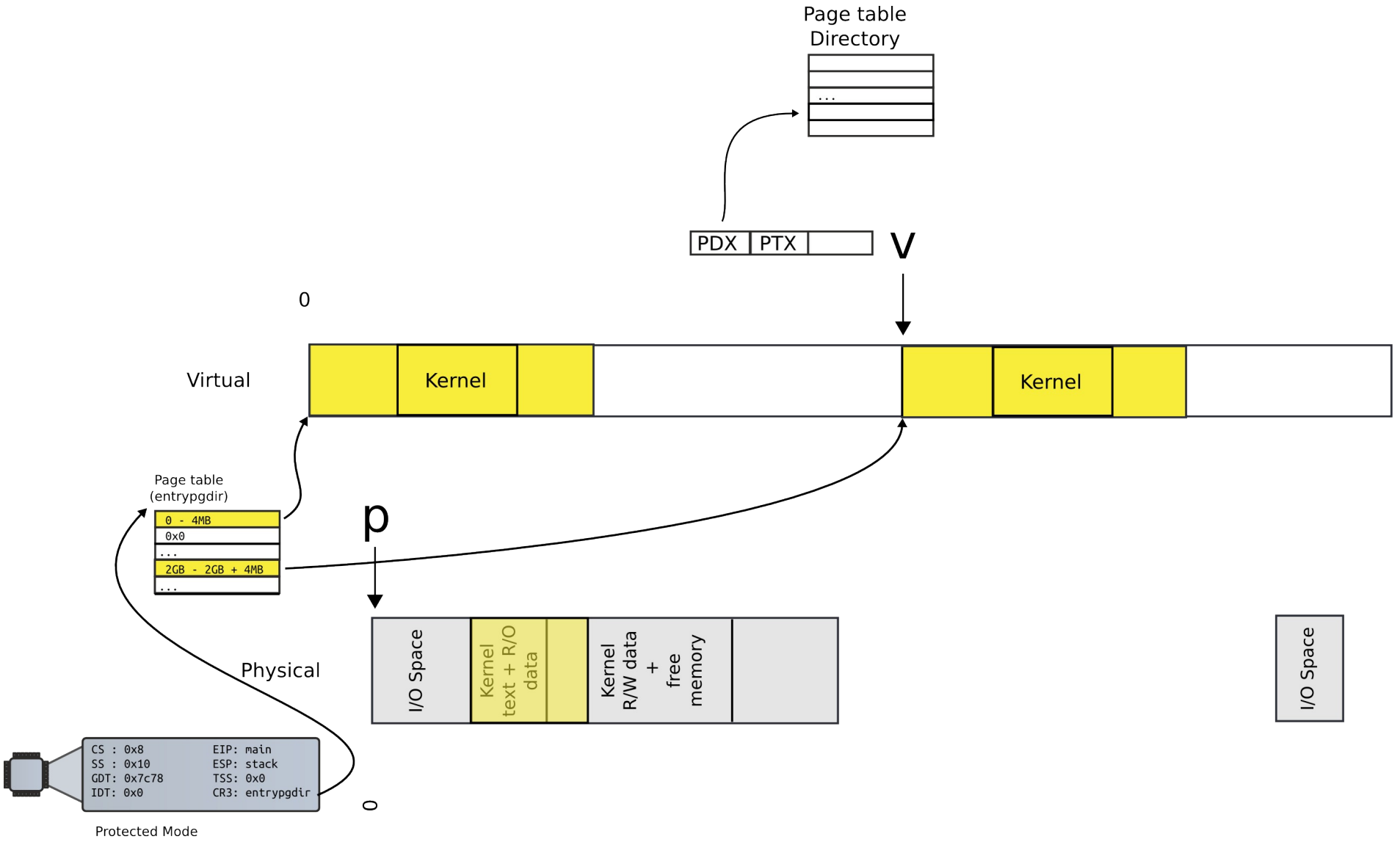
Recap: our goal



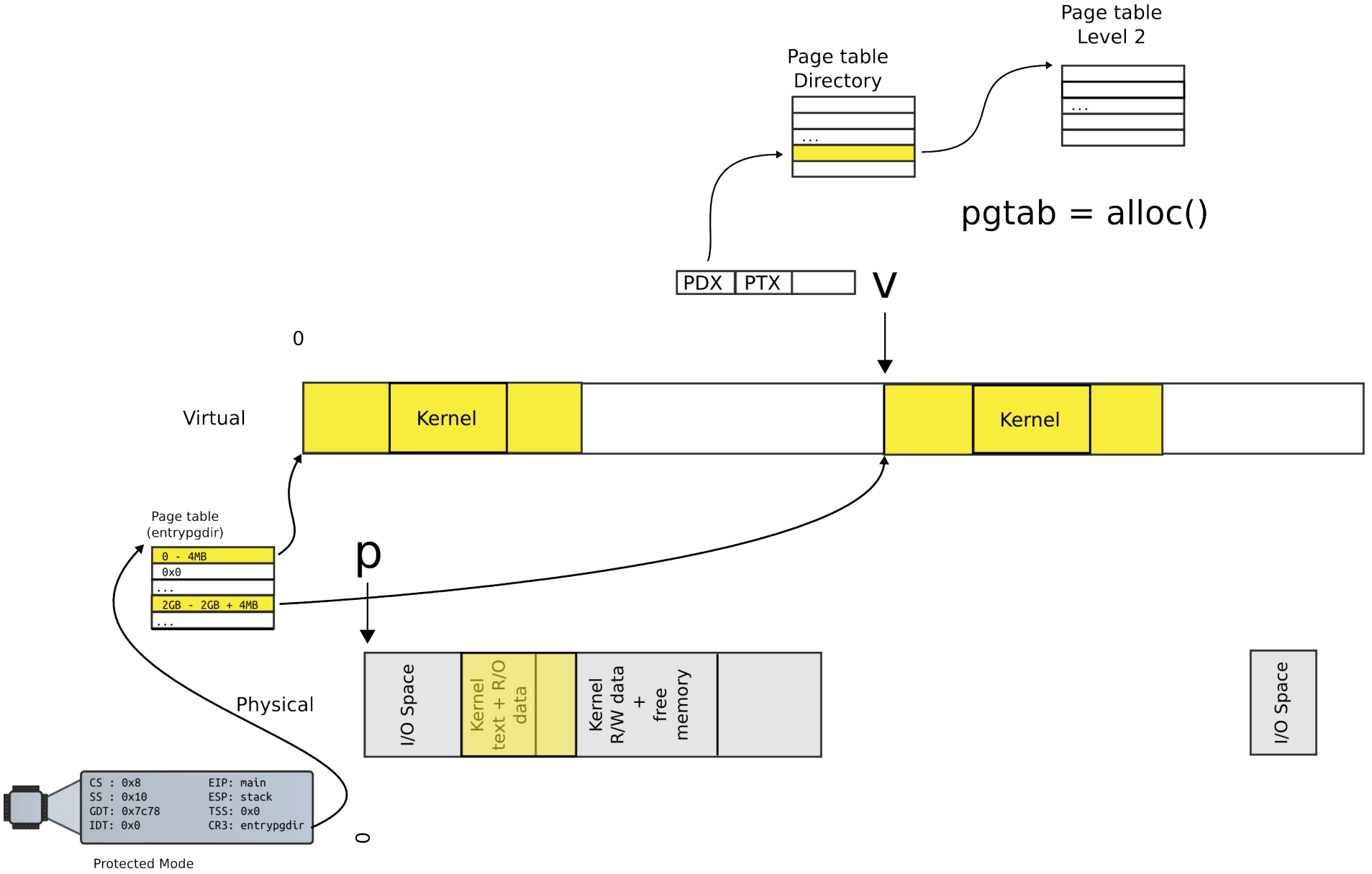
Outline

- 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

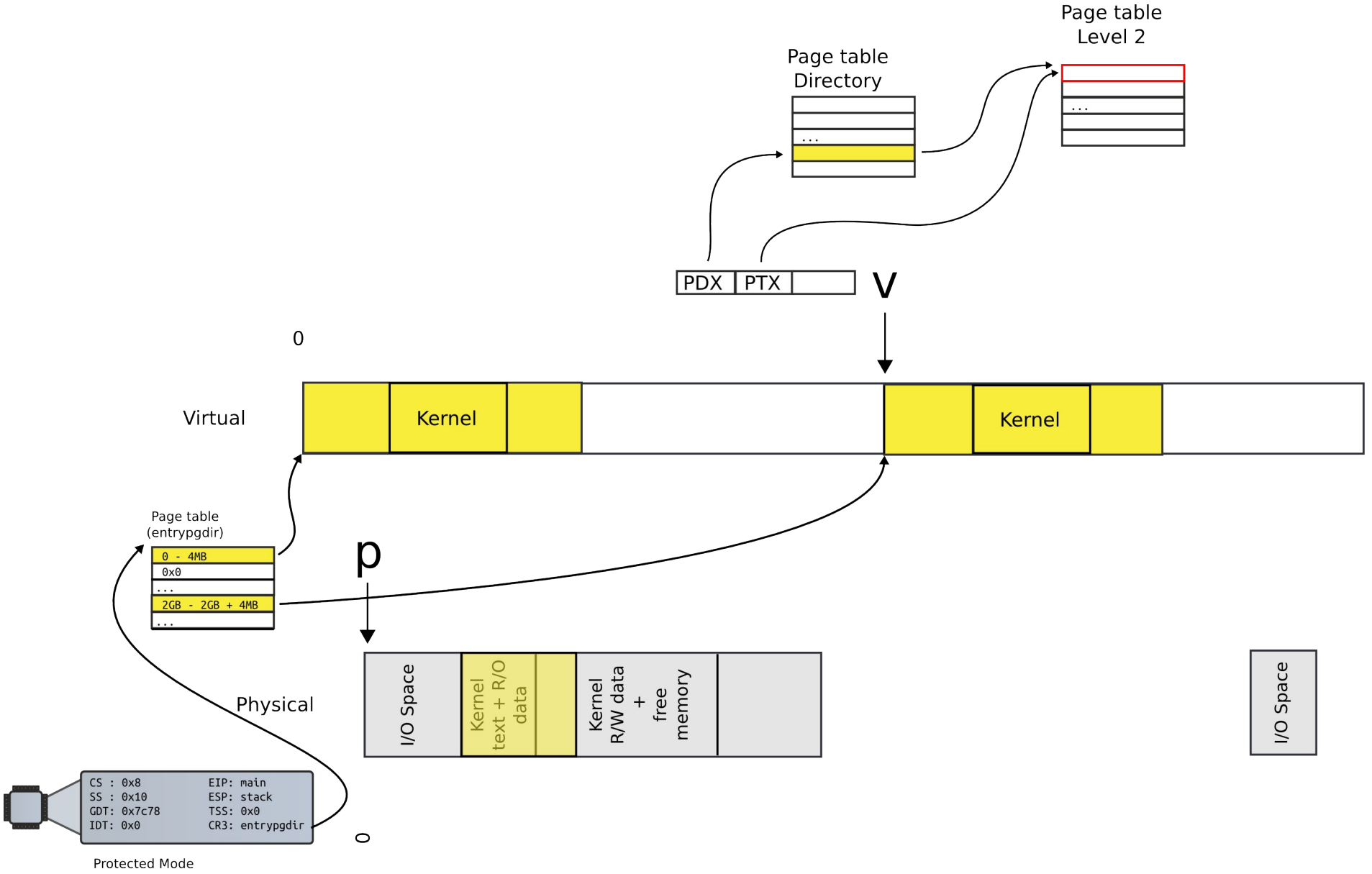
Allocate page table directory entry



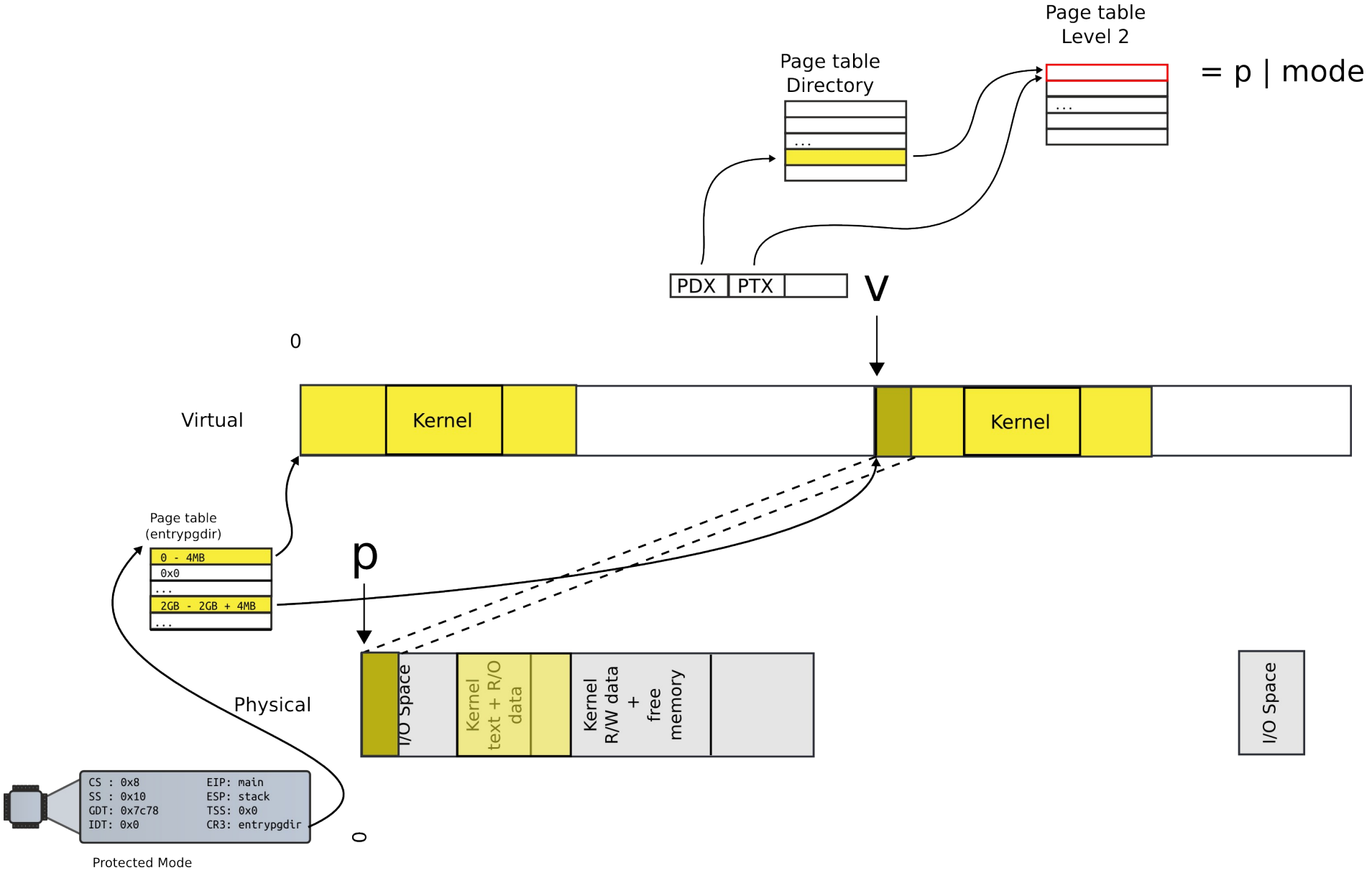
Allocate next level page table



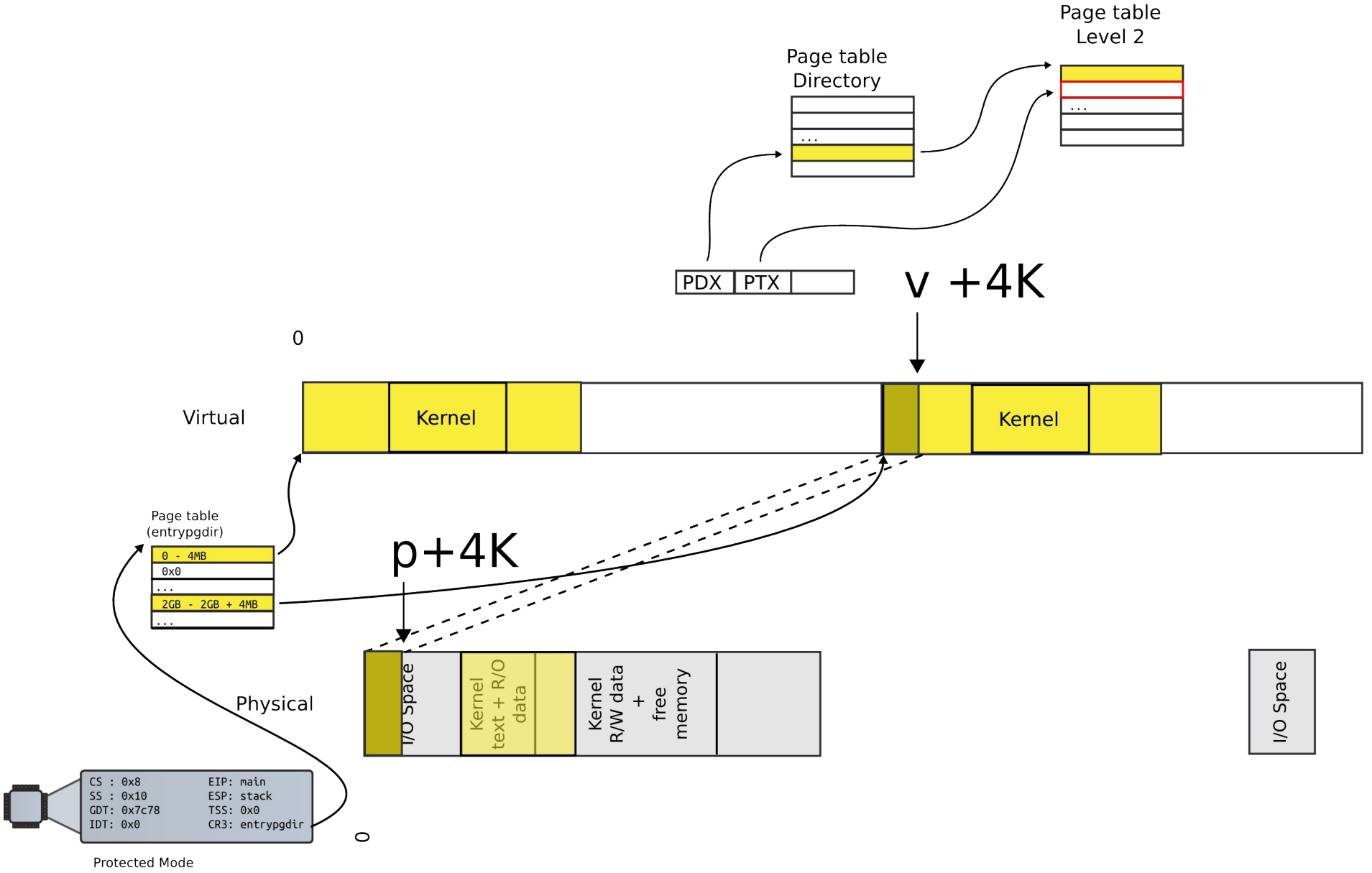
Locate PTE entry



Update mapping with physical addr



Move to next page

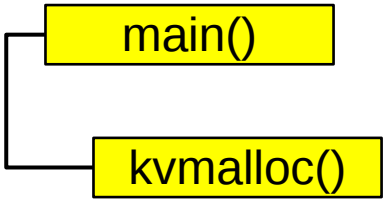


This is exactly what kernel is doing

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 }
```

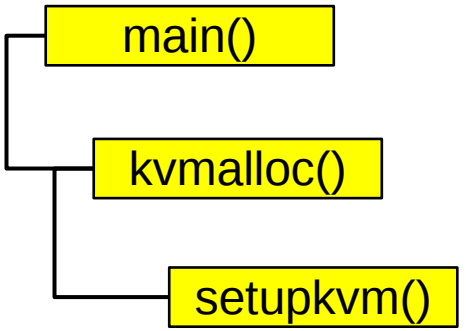
kvmalloc()




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

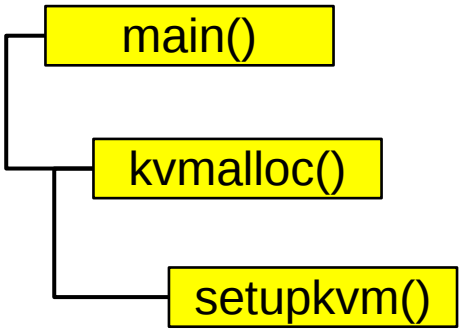
Allocate page table directory

```
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 }
```



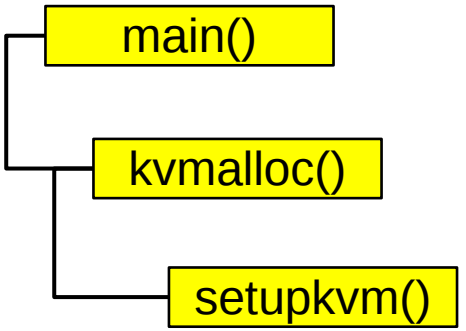
Iterate in a loop: map physical pages

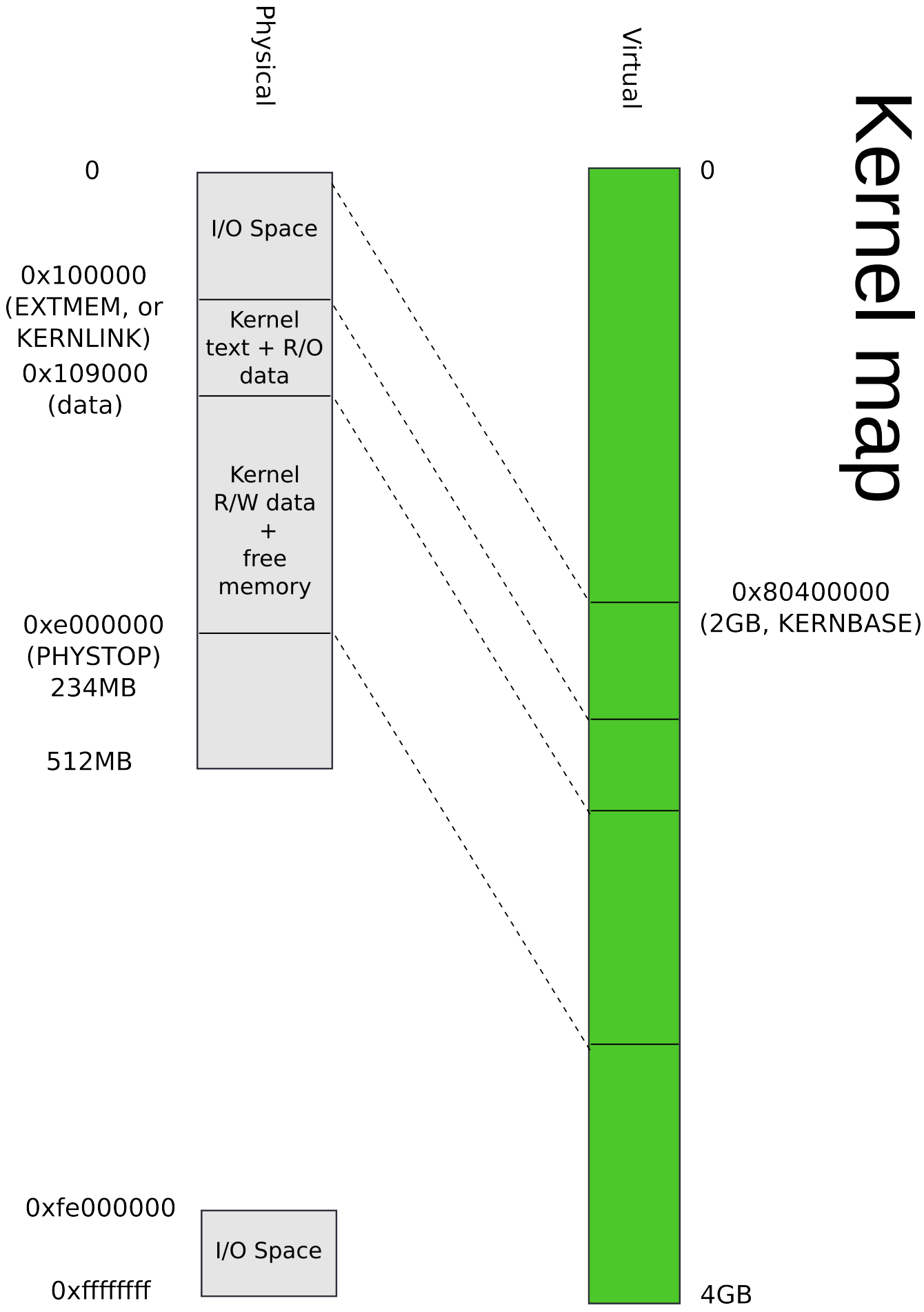
```
1836 pde_t*
1887 setupkvm(void)
1838 {
1839   pde_t *pgdir;
1840   struct kmap *k;
1841
1842   if((pgdir = (pde_t*)kalloc()) == 0)
1843     return 0;
1844   memset(pgdir, 0, PGSIZE);
...
1847   for(k = kmap; k < &kmap[NELEM(kmap)]; k++)
1848     if(mappages(pgdir, k->virt, k->phys_end - k->phys_start,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



Iterate in a loop: map physical pages

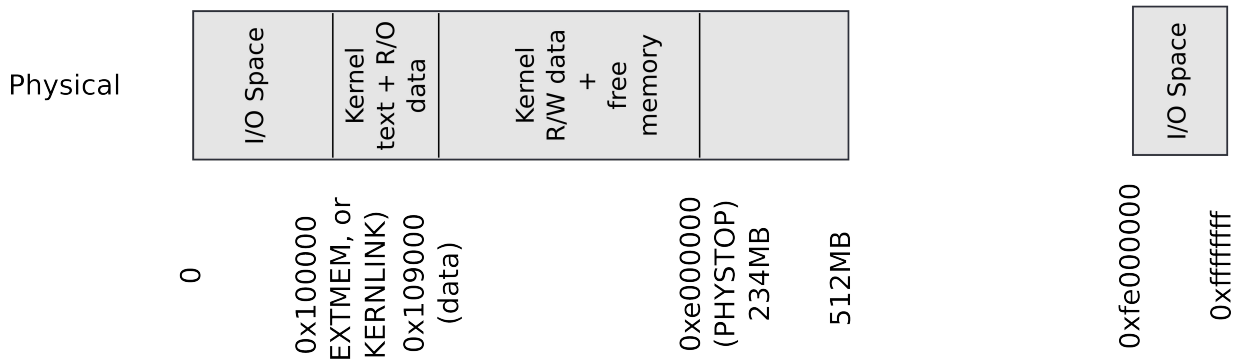
```
1836 pde_t*
1887 setupkvm(void)
1838 {
1839   pde_t *pgdir;
1840   struct kmap *k;
1841
1842   if((pgdir = (pde_t*)kalloc()) == 0)
1843     return 0;
1844   memset(pgdir, 0, PGSIZE);
...
1847   for(k = kmap; k < &kmap[NELEM(kmap)]; k++)
1848     if(mappages(pgdir, k->virt, k->phys_end - k->phys_start,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

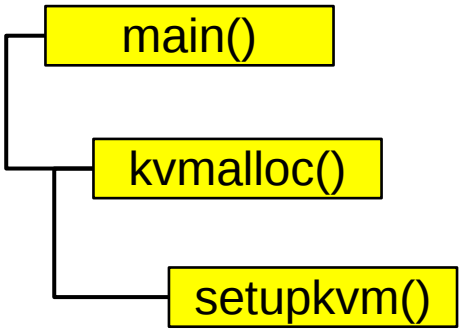
Kmap – kernel map

```
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 };
```



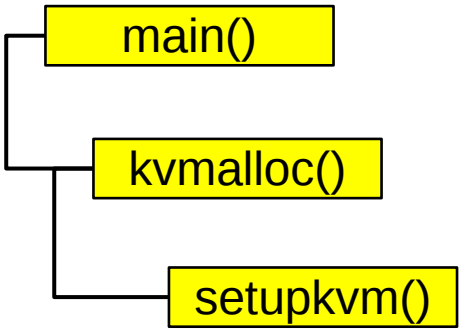
Iterate in a loop: map physical pages

```
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 }
```

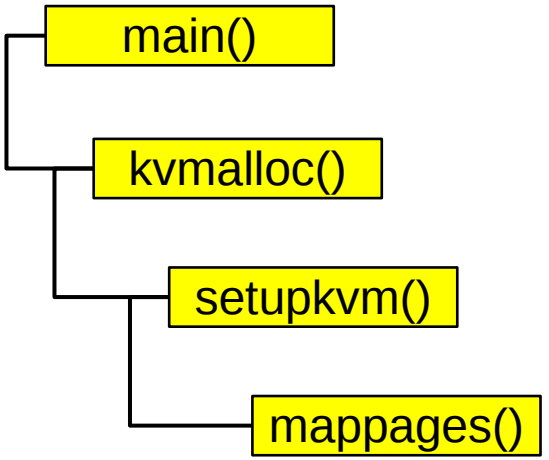


Map a region of memory

```
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 }
```

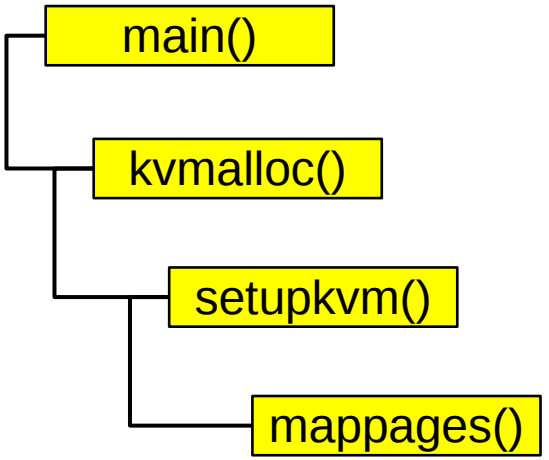


```
1779 mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm)
1780 {
1781   char *a, *last;
1782   pte_t *pte;
1783
1784   a = (char*)PGROUNDDOWN((uint)va);
1785   last = (char*)PGROUNDDOWN(((uint)va) + size - 1);
1786   for(;;){
1787     if((pte = walkpgdir(pgdir, a, 1)) == 0)
1788       return -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) virtual address

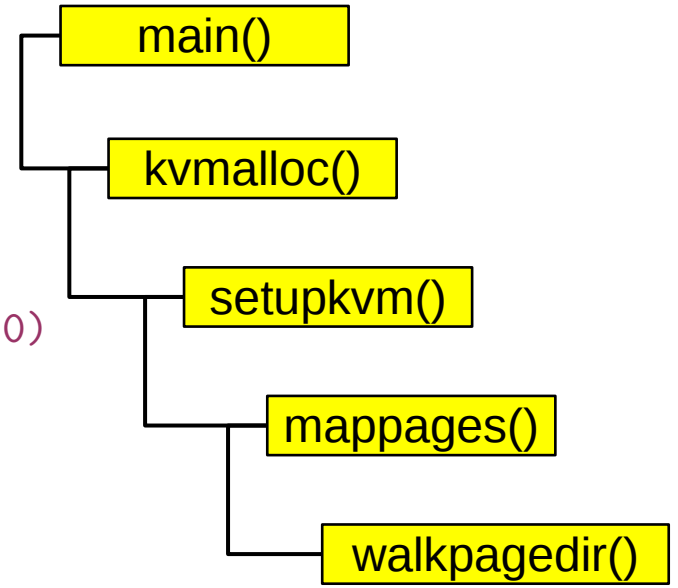
```
1779 mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm)
1780 {
1781   char *a, *last;
1782   pte_t *pte;
1783
1784   a = (char*)PGROUNDDOWN((uint)va);
1785   last = (char*)PGROUNDDOWN(((uint)va) + size - 1);
1786   for(;;){
1787     if((pte = walkpgdir(pgdir, a, 1)) == 0)
1788       return -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 }
```



Lookup the page table entry

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 }
```

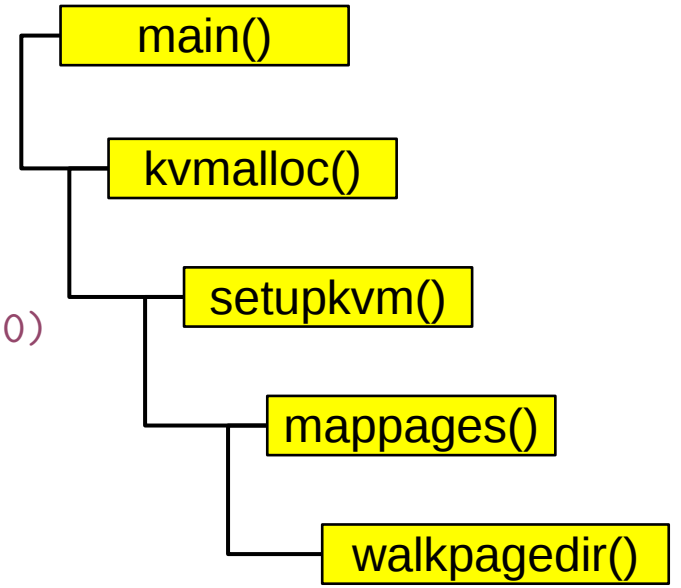


PDX()

```
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
```

Walk page table

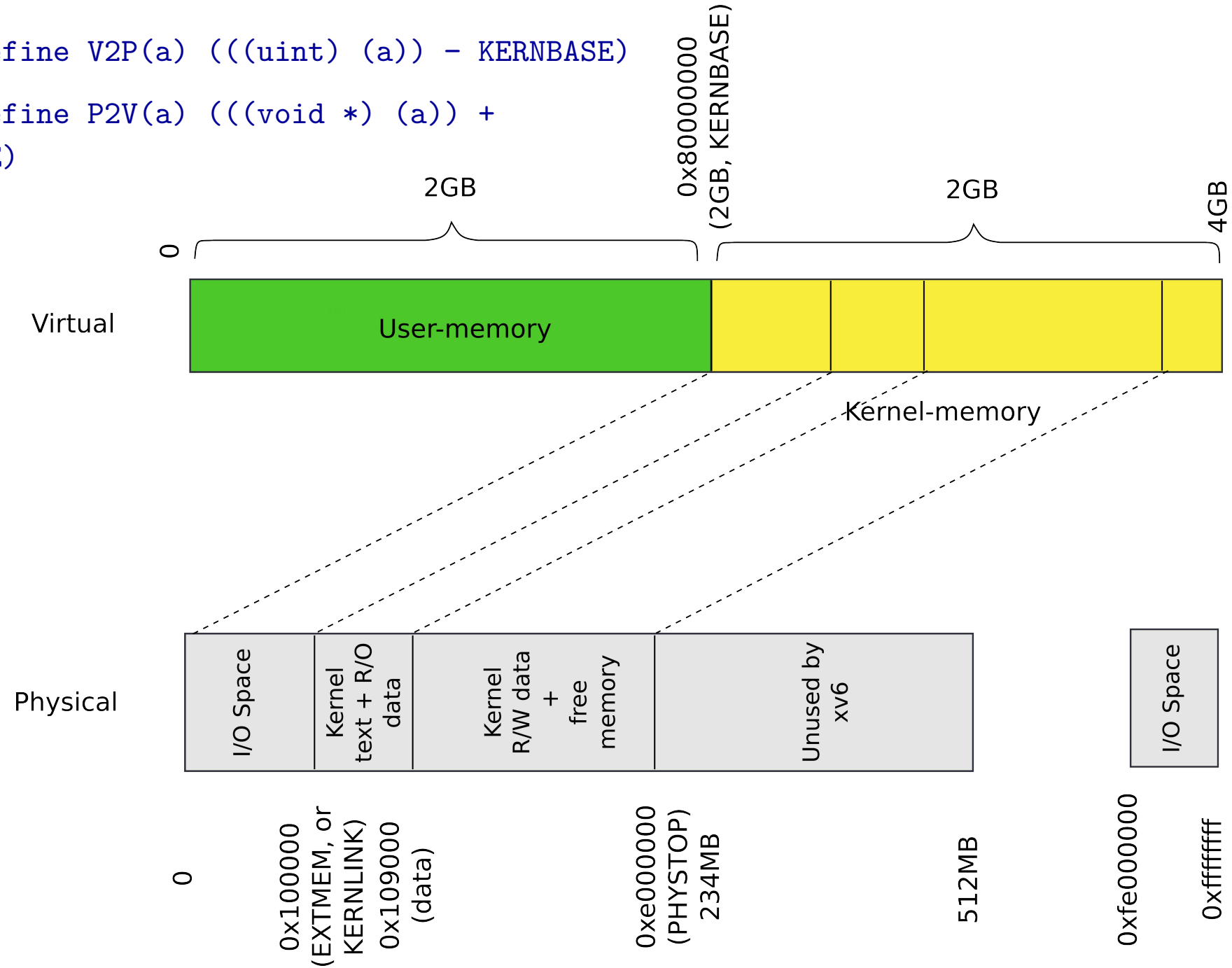
```
1754 walkpgdir(pde_t *pgdir, const void *va, int alloc)
1755 {
1756     pde_t *pde;
1757     pte_t *pgtab;
1758
1759     pde = &pgdir[PDX(va)];
1760     if(*pde & PTE_P){
1761         pgtab = (pte_t*)P2V(PTE_ADDR(*pde));
1762     } else {
1763         if(!alloc || (pgtab = (pte_t*)kalloc()) == 0)
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 }
```



P2V and V2P

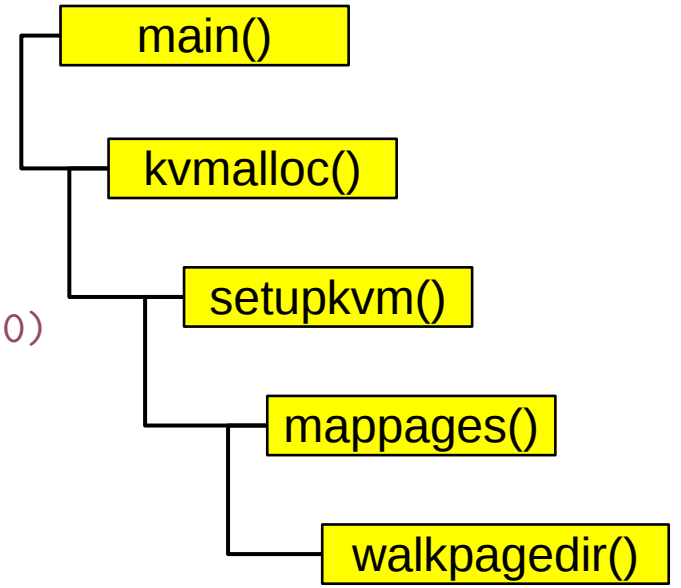
```
0206 // Key addresses for address space layout (see kmap in vm.c for
layout)

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

0208 #define KERNLINK (KERNBASE+EXTMEM) // Address where kernel is linked

0209

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

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

```
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

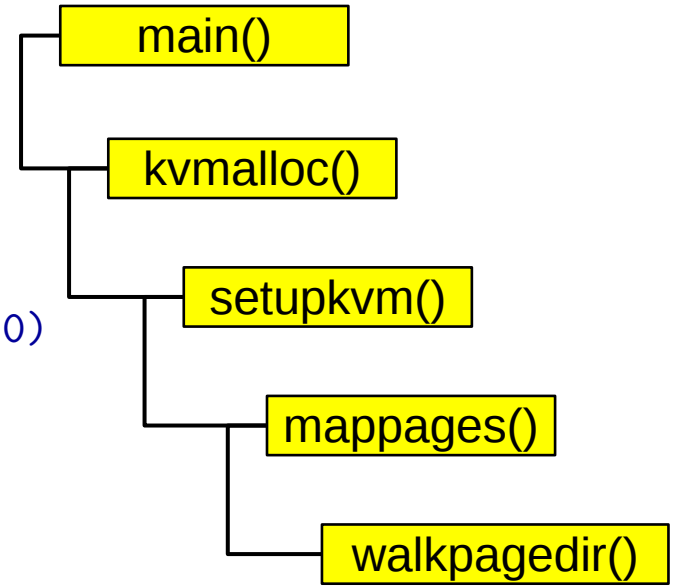
```
1754 walkpgdir(pde_t *pgdir, const void *va, int alloc)
1755 {
1756     pde_t *pde;
1757     pte_t *pgtab;
1758
1759     pde = &pgdir[PDX(va)];
1760     if(*pde & PTE_P){
1761         pgtab = (pte_t*)P2V(PTE_ADDR(*pde));
1762     } else {
1763         if(!alloc || (pgtab = (pte_t*)kalloc()) == 0)
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
  - Look it up

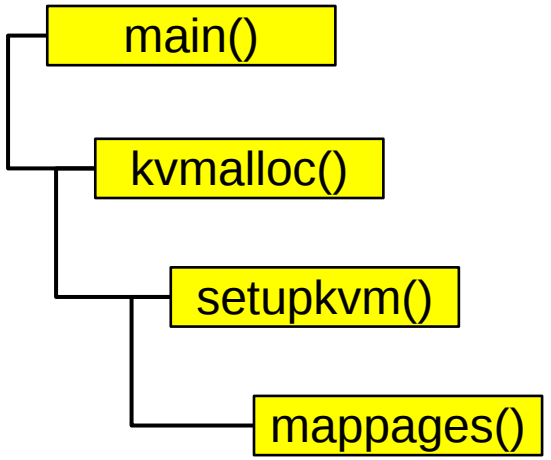
Walk page table

```
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 }
```



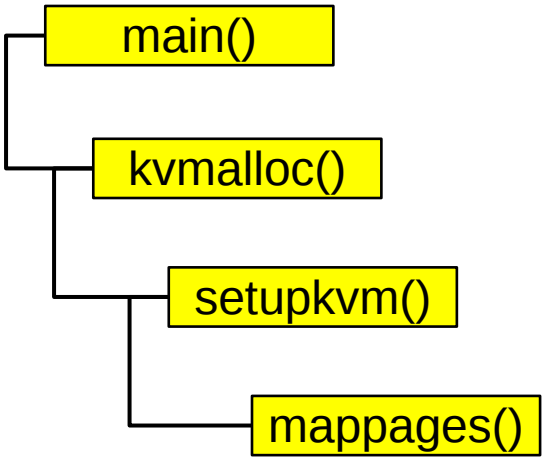
- Page table Level 2 doe not exist
  - Allocate one

```
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 }
```



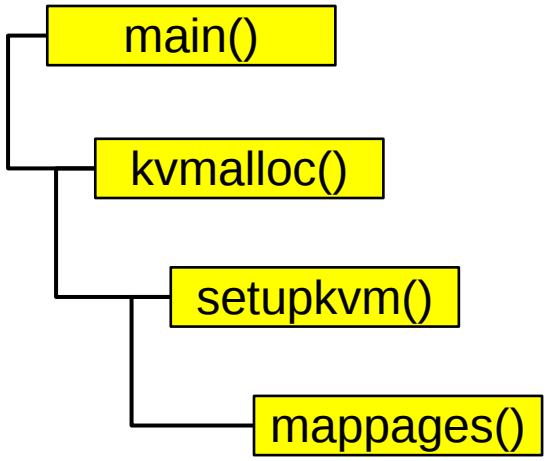
Look up the page table entry

```
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 }
```



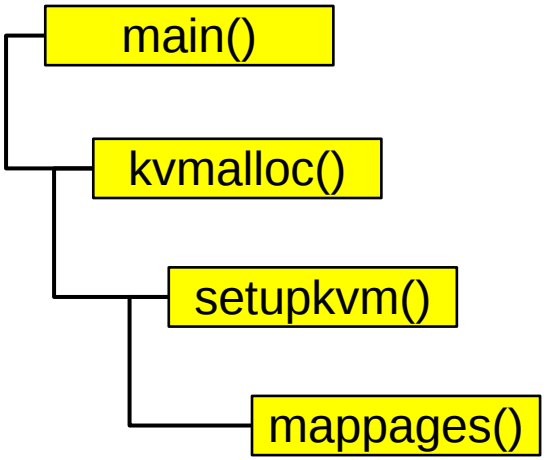
Page present (PTE_P) – panic

```
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 it point (*pte)?
  - pa – physical address of the page

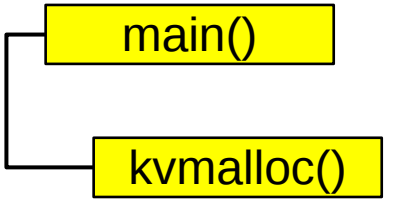
```
1779 mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm)
1780 {
1781   char *a, *last;
1782   pte_t *pte;
1783
1784   a = (char*)PGROUNDDOWN((uint)va);
1785   last = (char*)PGROUNDDOWN(((uint)va) + size - 1);
1786   for(;;){
1787     if((pte = walkpgdir(pgdir, a, 1)) == 0)
1788       return -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

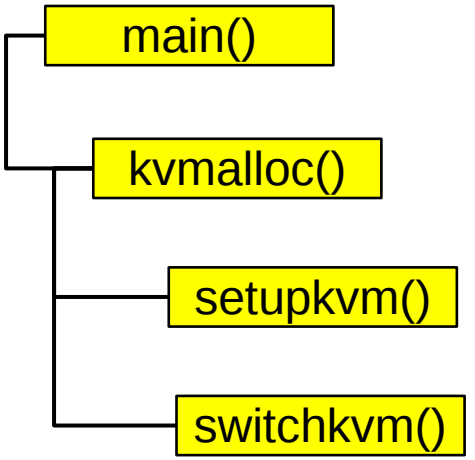
kvmalloc()

```
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 }
```



Conclusion

- Kernel has a its own address space
  - It uses 4KB page tables

Thank you!