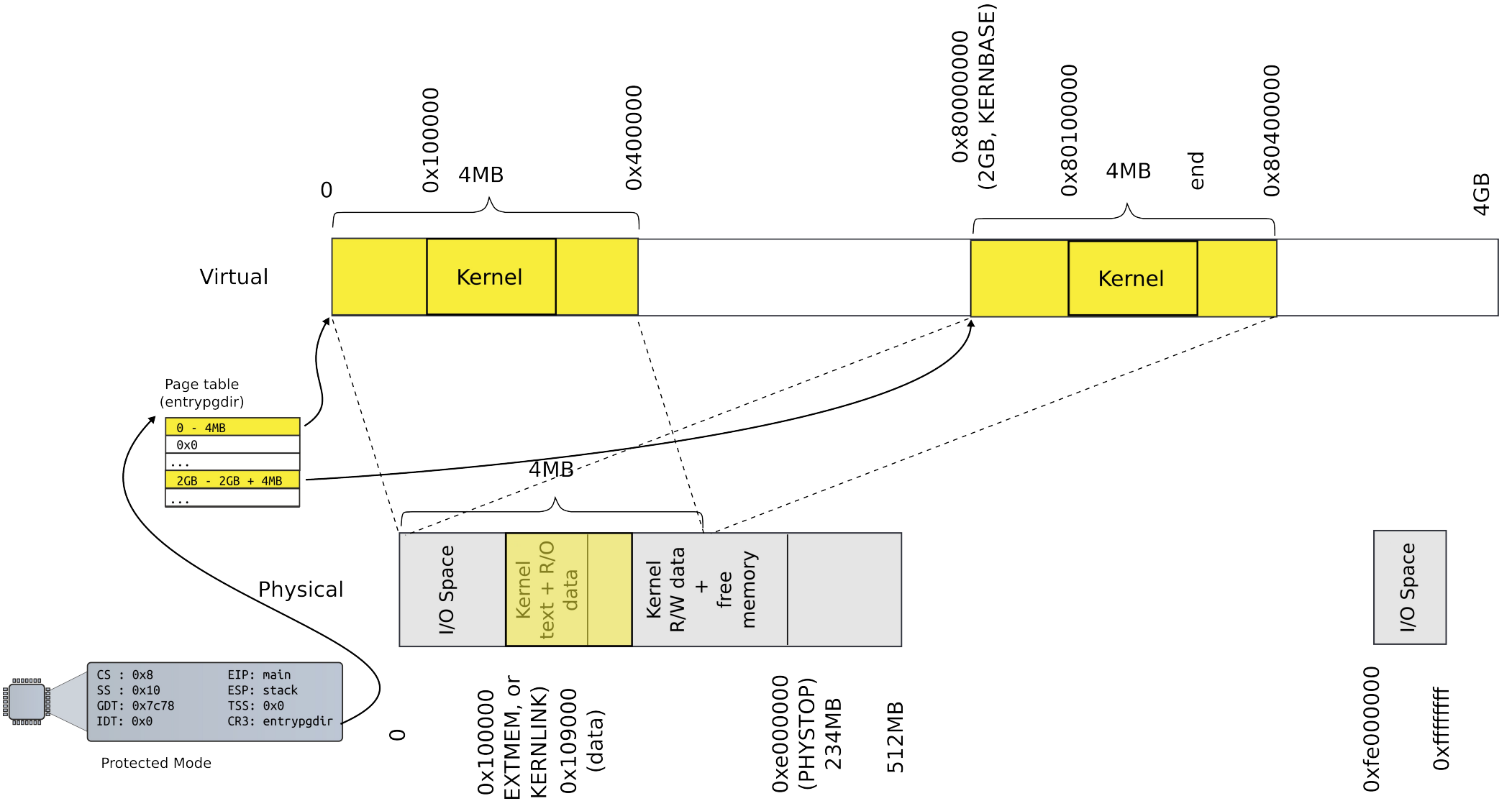
143A: Principles of Operating Systems

Lecture 10: Address spaces (4K page tables)

Anton Burtsev
October, 2017

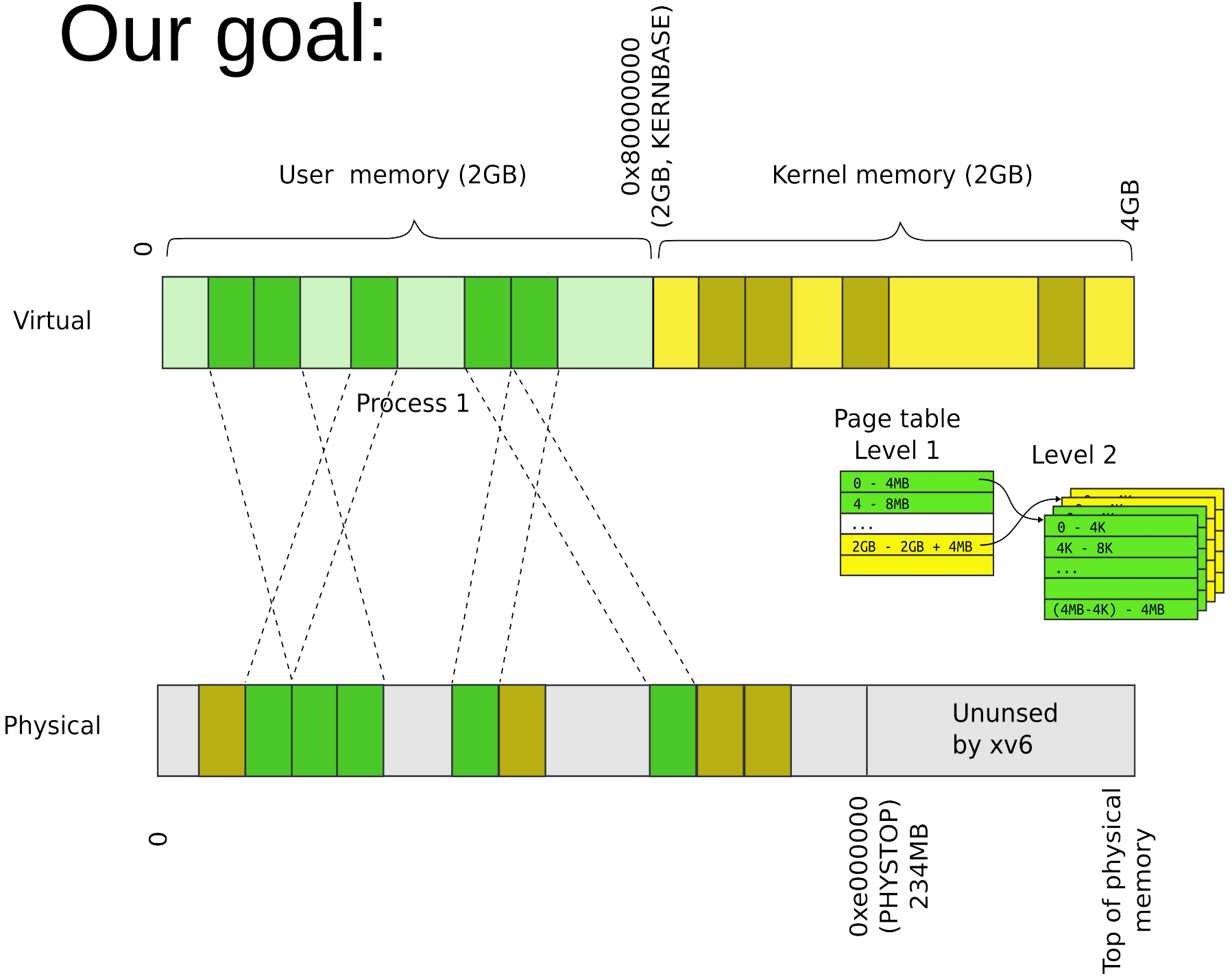
Memory after boot



Recap from last time

- We plan to move to 4KB page tables
  - Why do we need them?

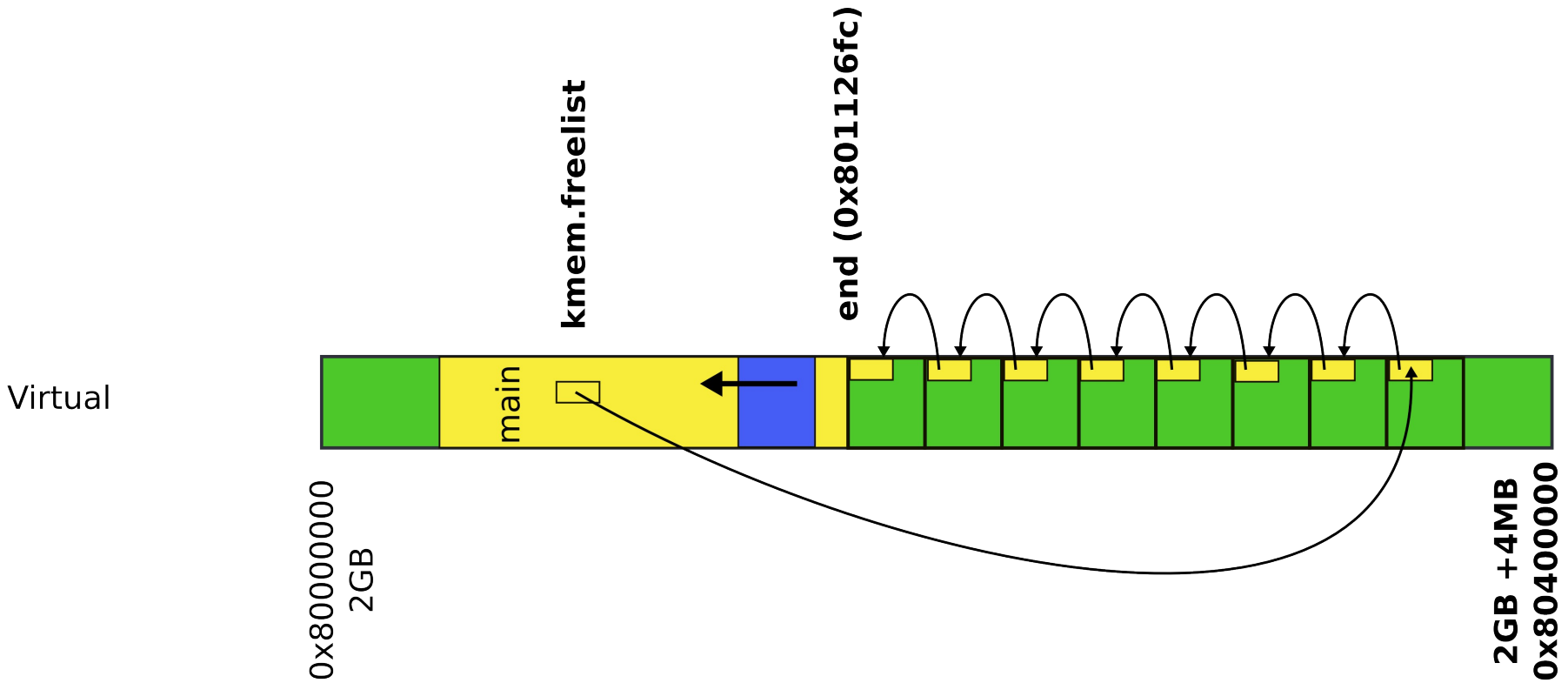
Our goal:

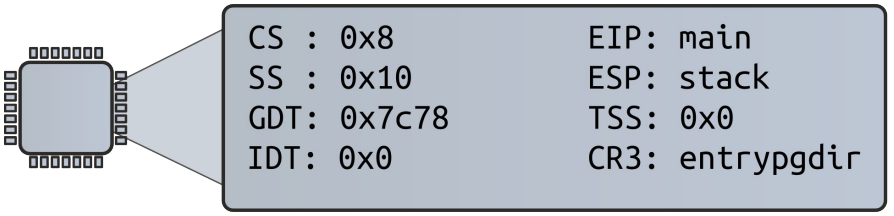

0x80000000
(2GB, KERNBASE)
User memory (2GB)
Kernel memory (2GB)
4GB
0
Virtual
Process 1
Page table
Level 1
Level 2
0 - 4MB
4 - 8MB
...
2GB - 2GB + 4MB
0 - 4K
4K - 8K
...
(4MB-4K) - 4MB
Physical
Ununsed
by xv6
0
0xe000000
(PHYSTOP)
234MB
Top of physical
memory

Recap from last time

- We plan to move to 4KB page tables
  - Why do we need them?
- We've created the kernel memory allocator
  - Can allocate space for page table directory and page tables

Physical page allocator



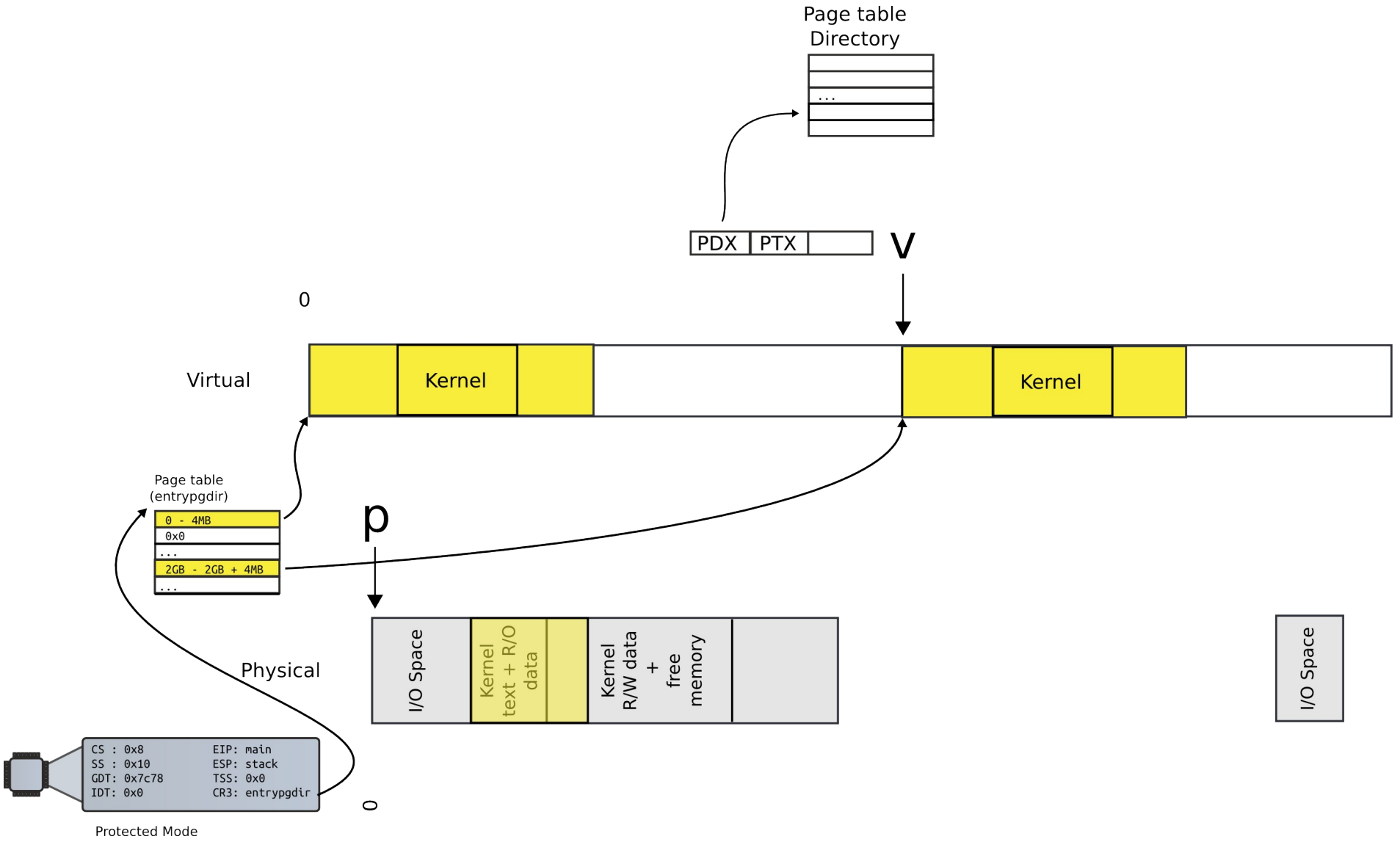


Protected Mode

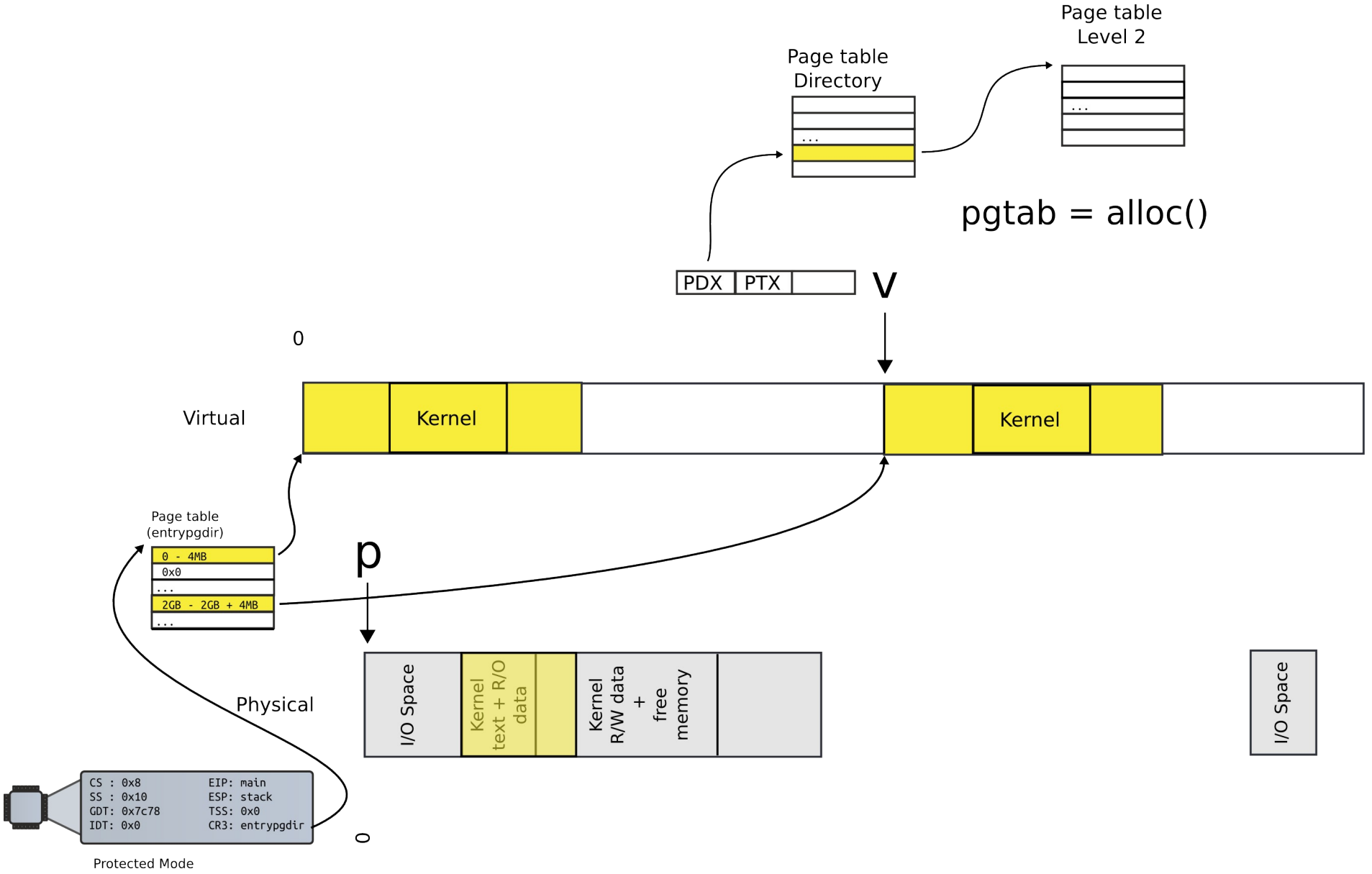
Outline

- 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

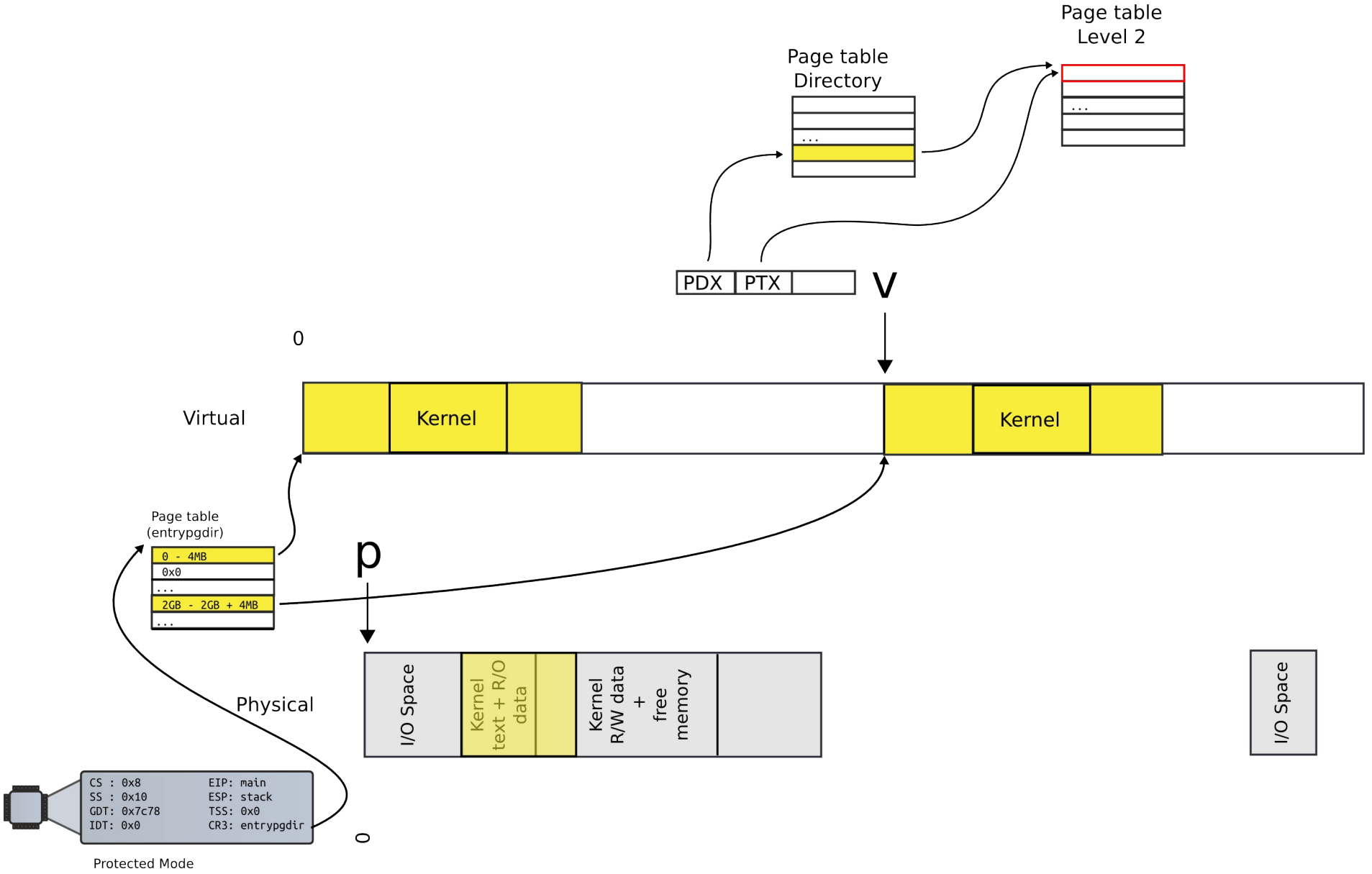
Allocate page table directory entry



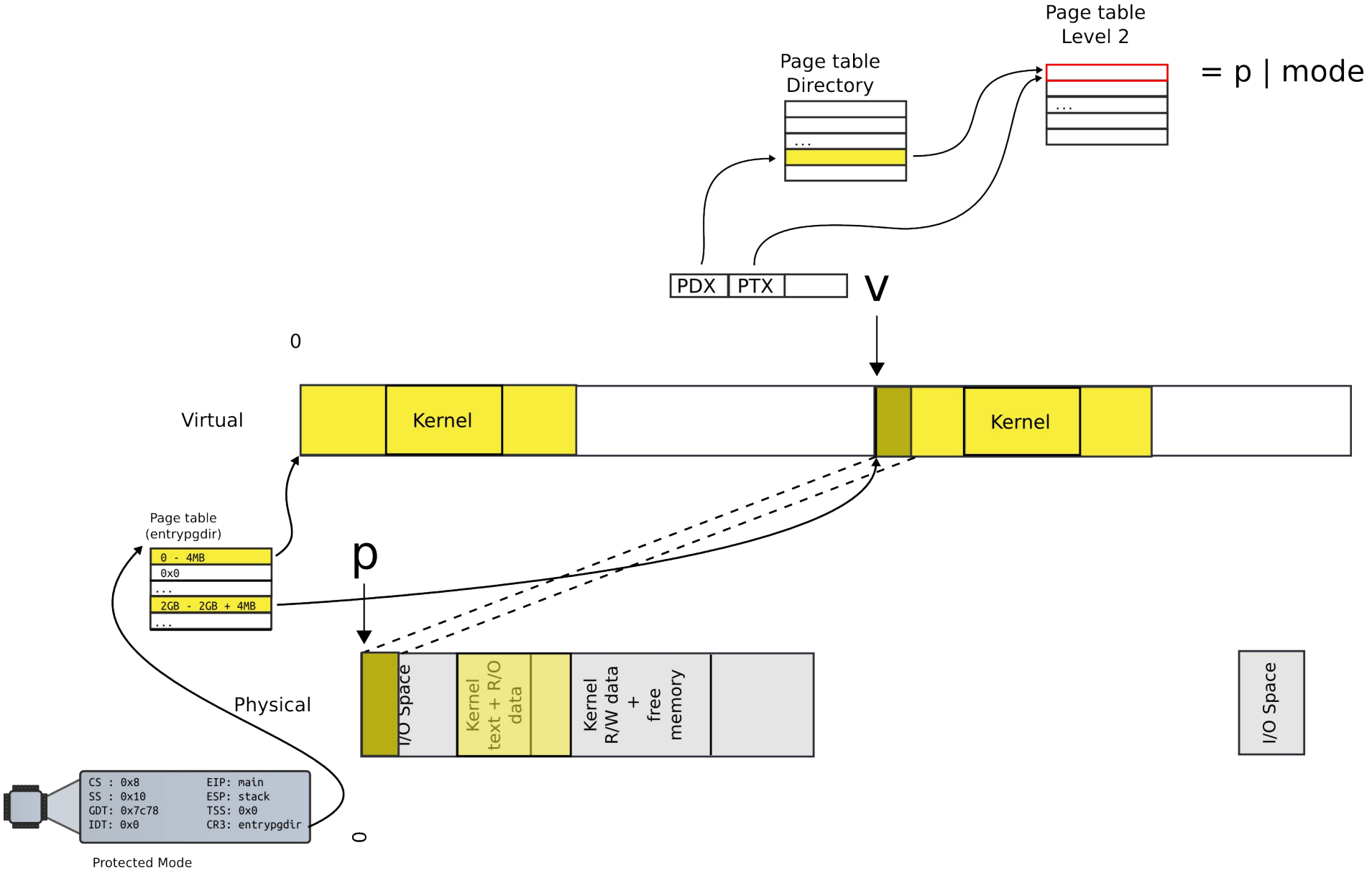
Allocate next level page table



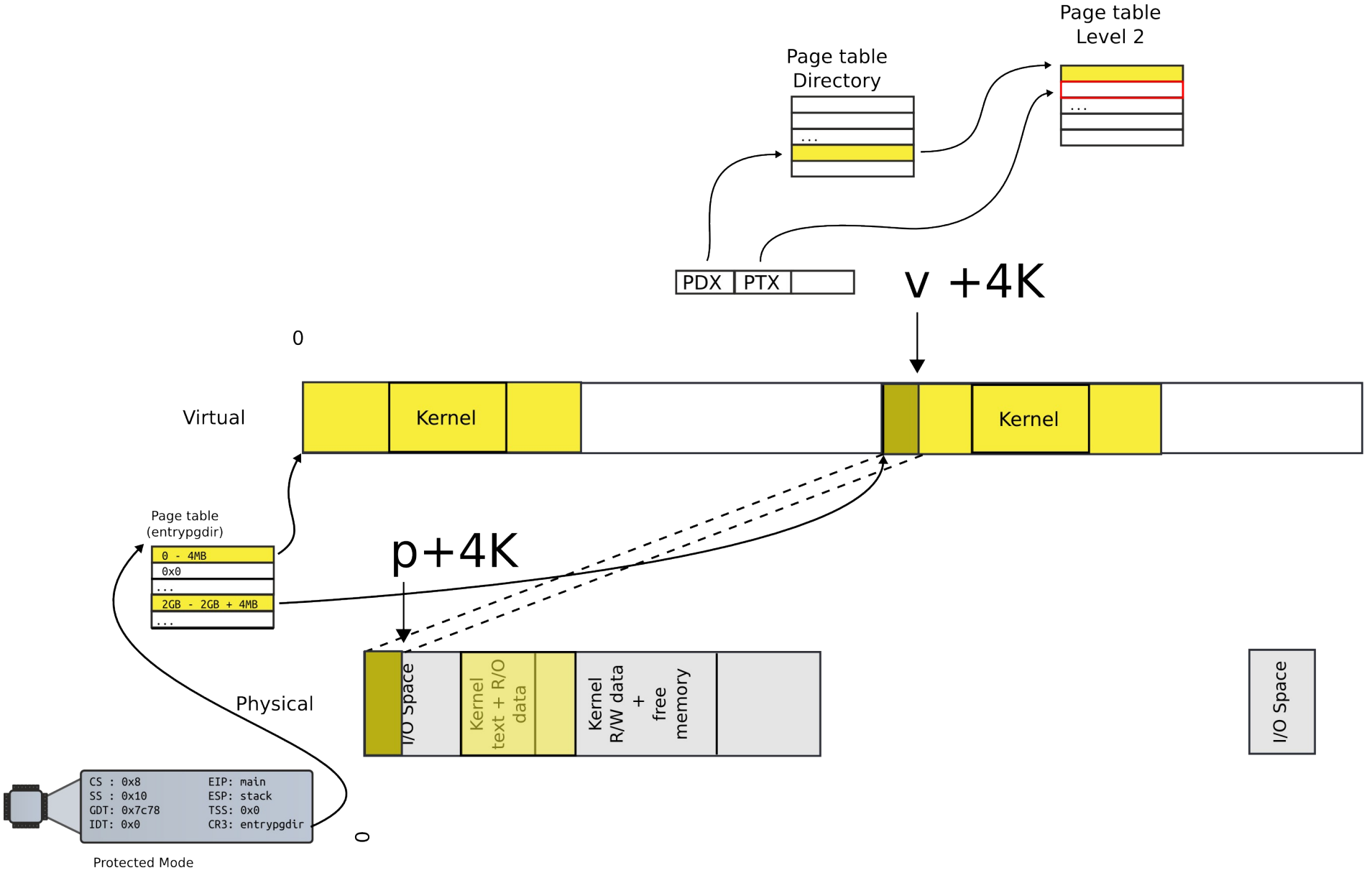
Locate PTE entry



Update mapping with physical addr



Move to next page

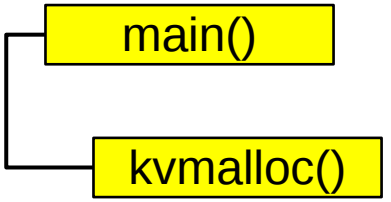


This is exactly what kernel is doing

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 }
```

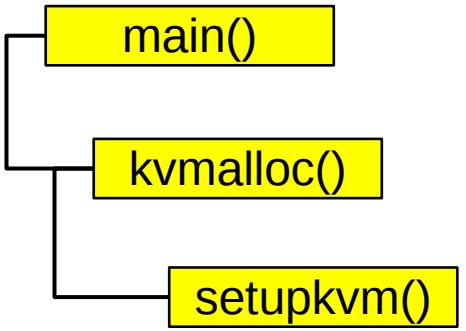
kvmalloc()




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

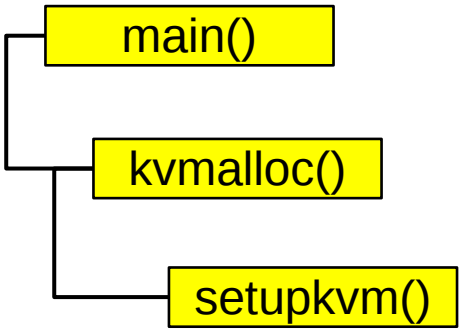
Allocate page table directory

```
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 }
```



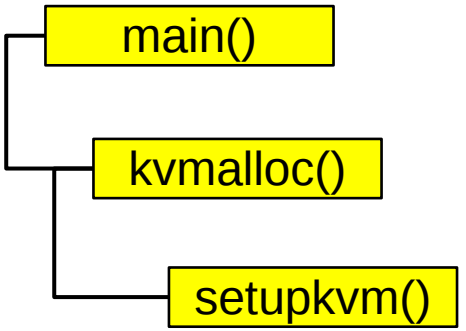
Iterate in a loop: remap physical pages

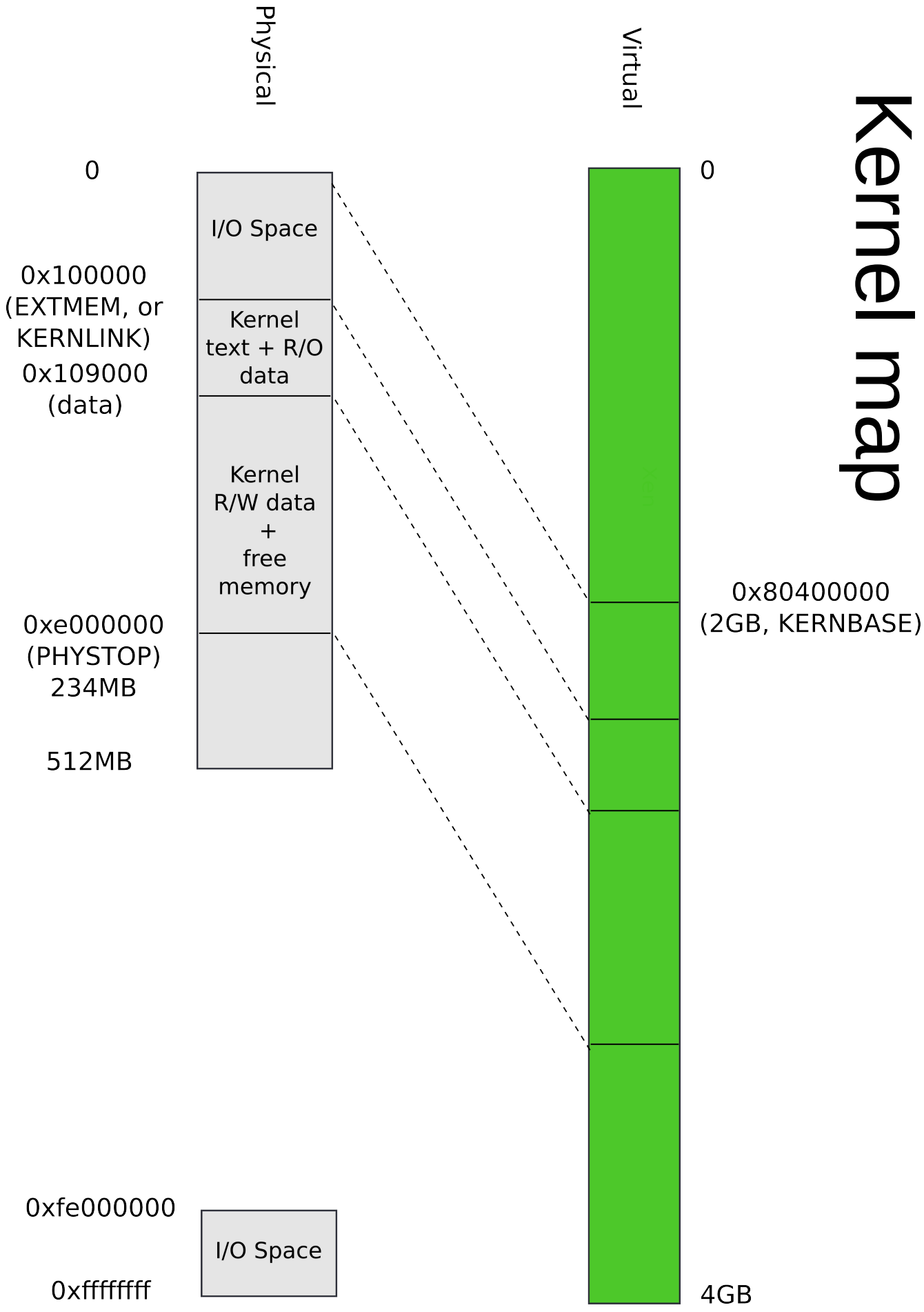
```
1836 pde_t*
1887 setupkvm(void)
1838 {
1839   pde_t *pgdir;
1840   struct kmap *k;
1841
1842   if((pgdir = (pde_t*)kalloc()) == 0)
1843     return 0;
1844   memset(pgdir, 0, PGSIZE);
...
1847   for(k = kmap; k < &kmap[NELEM(kmap)]; k++)
1848     if(mappages(pgdir, k->virt, k->phys_end - k->phys_start,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



Iterate in a loop: remap physical pages

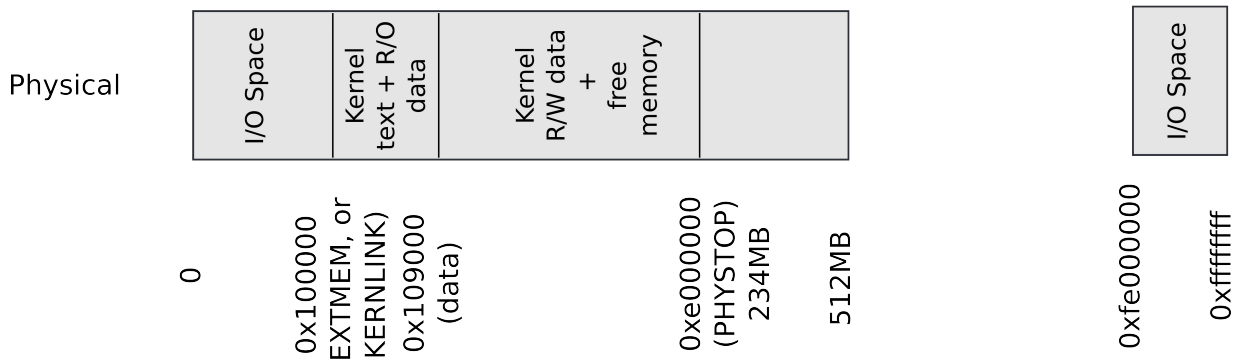
```
1836 pde_t*
1887 setupkvm(void)
1838 {
1839   pde_t *pgdir;
1840   struct kmap *k;
1841
1842   if((pgdir = (pde_t*)kalloc()) == 0)
1843     return 0;
1844   memset(pgdir, 0, PGSIZE);
...
1847   for(k = kmap; k < &kmap[NELEM(kmap)]; k++)
1848     if(mappages(pgdir, k->virt, k->phys_end - k->phys_start,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

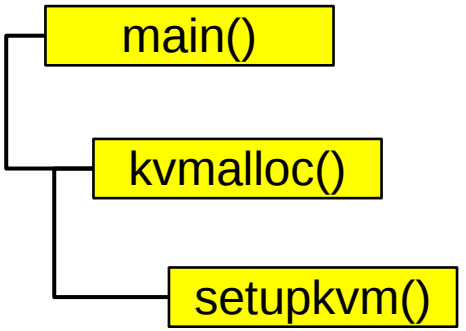
Kmap – kernel map

```
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 };
```



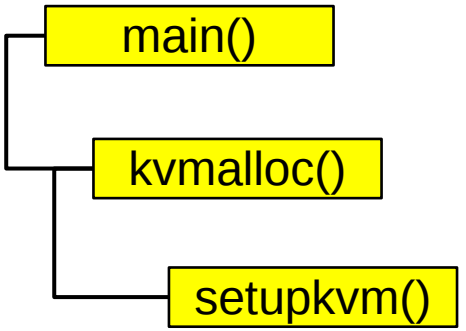
Iterate in a loop: remap physical pages

```
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 }
```

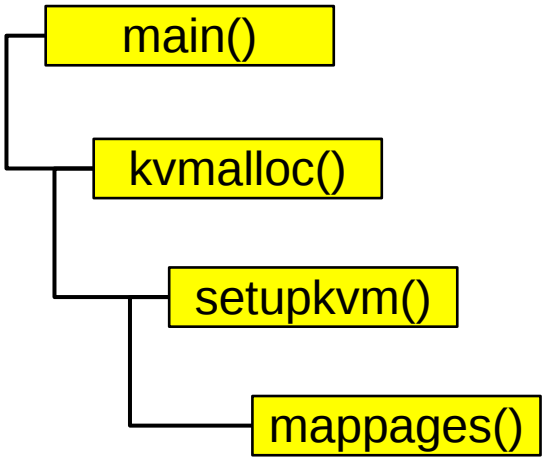


Remap physical pages

```
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 }
```

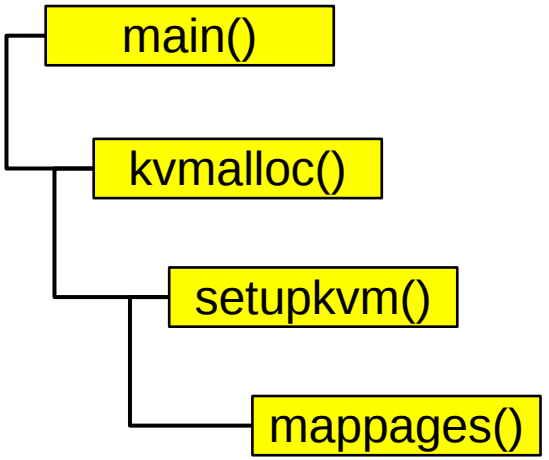


```
1779 mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm)
1780 {
1781   char *a, *last;
1782   pte_t *pte;
1783
1784   a = (char*)PGROUNDDOWN((uint)va);
1785   last = (char*)PGROUNDDOWN(((uint)va) + size - 1);
1786   for(;;){
1787     if((pte = walkpgdir(pgdir, a, 1)) == 0)
1788       return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



Create page table entries

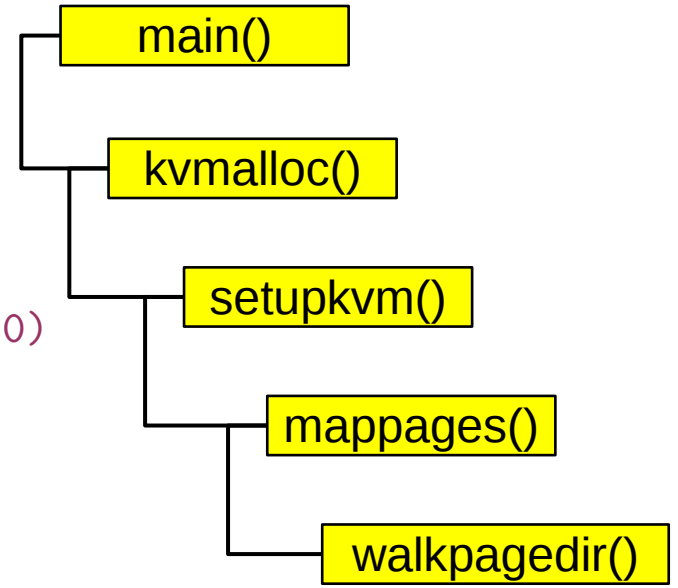
```
1779 mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm)
1780 {
1781   char *a, *last;
1782   pte_t *pte;
1783
1784   a = (char*)PGROUNDDOWN((uint)va);
1785   last = (char*)PGROUNDDOWN(((uint)va) + size - 1);
1786   for(;;){
1787     if((pte = walkpgdir(pgdir, a, 1)) == 0)
1788       return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

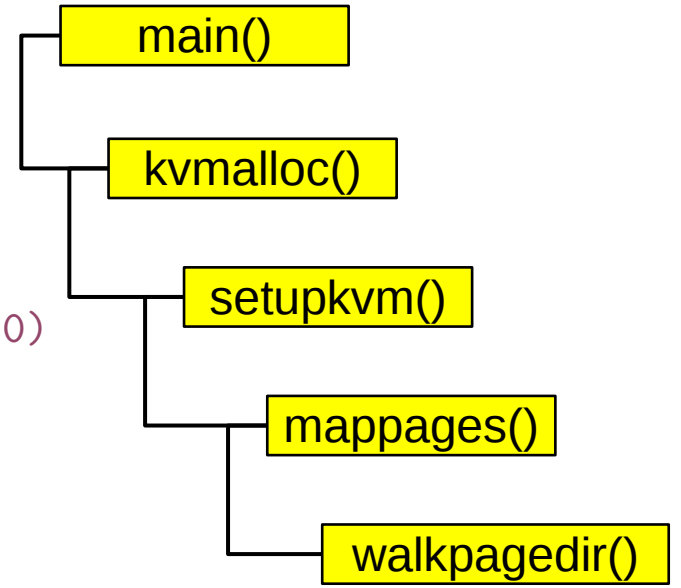


PDX()

```
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

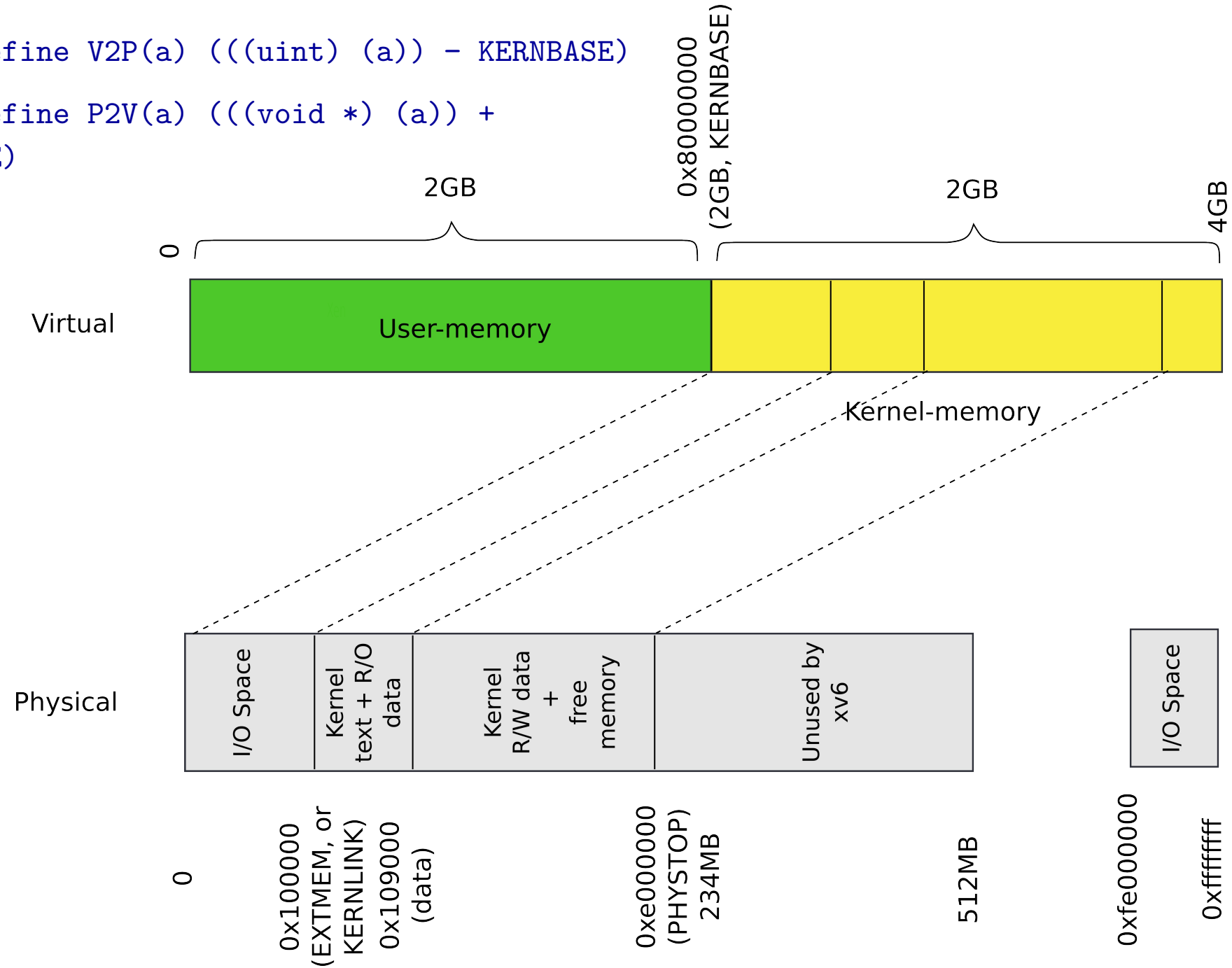
```
1754 walkpgdir(pde_t *pgdir, const void *va, int alloc)
1755 {
1756     pde_t *pde;
1757     pte_t *pgtab;
1758
1759     pde = &pgdir[PDX(va)];
1760     if(*pde & PTE_P){
1761         pgtab = (pte_t*)P2V(PTE_ADDR(*pde));
1762     } else {
1763         if(!alloc || (pgtab = (pte_t*)kalloc()) == 0)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

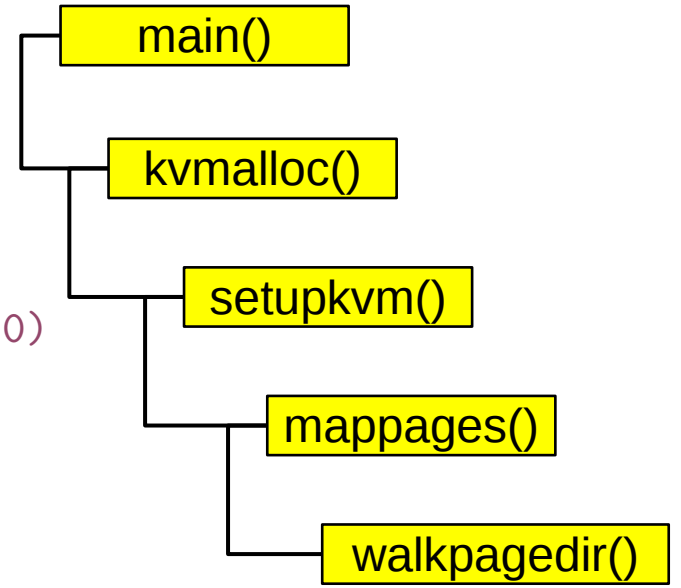
```
0206 // Key addresses for address space layout (see kmap in vm.c for
layout)

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)
```



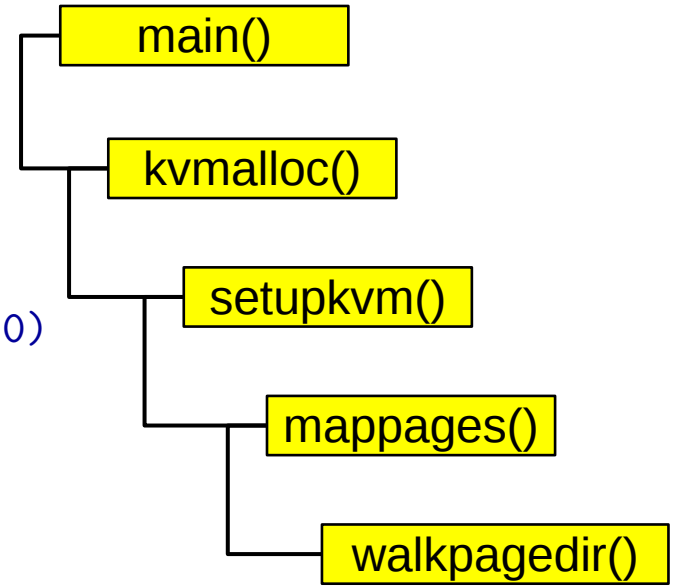
Walk page table

```
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 }
```

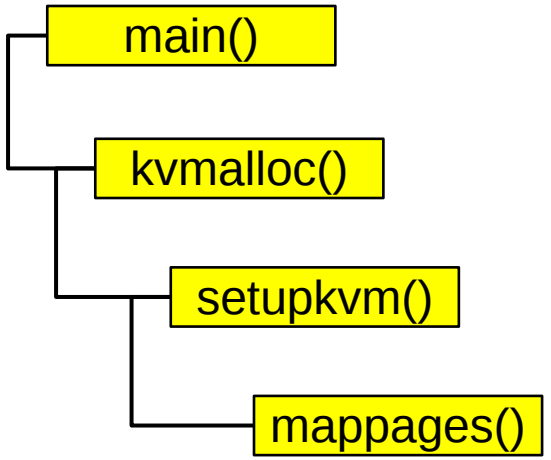


Walk page table

```
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 }
```

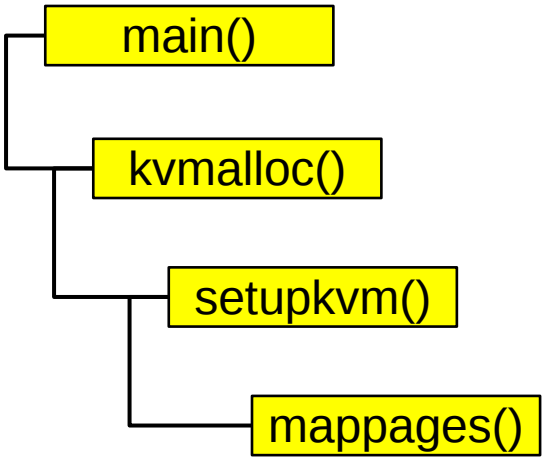


```
1779 mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm)
1780 {
1781   char *a, *last;
1782   pte_t *pte;
1783
1784   a = (char*)PGROUNDDOWN((uint)va);
1785   last = (char*)PGROUNDDOWN(((uint)va) + size - 1);
1786   for(;;){
1787     if((pte = walkpgdir(pgdir, a, 1)) == 0)
1788       return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



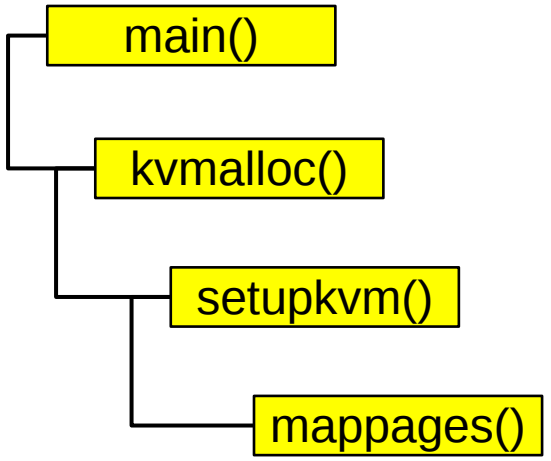
Look up the page table entry

```
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 }
```



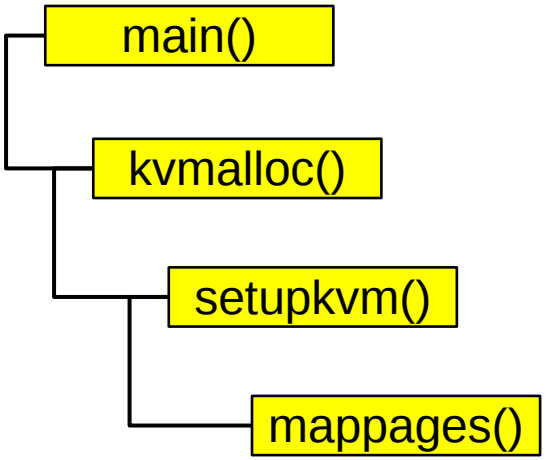
Page present (PTE_P) – panic

```
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

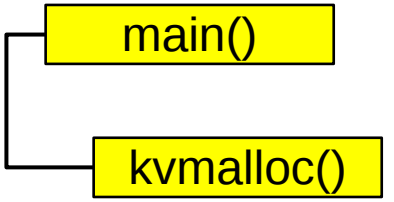
```
1779 mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm)
1780 {
1781   char *a, *last;
1782   pte_t *pte;
1783
1784   a = (char*)PGROUNDDOWN((uint)va);
1785   last = (char*)PGROUNDDOWN(((uint)va) + size - 1);
1786   for(;;){
1787     if((pte = walkpgdir(pgdir, a, 1)) == 0)
1788       return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

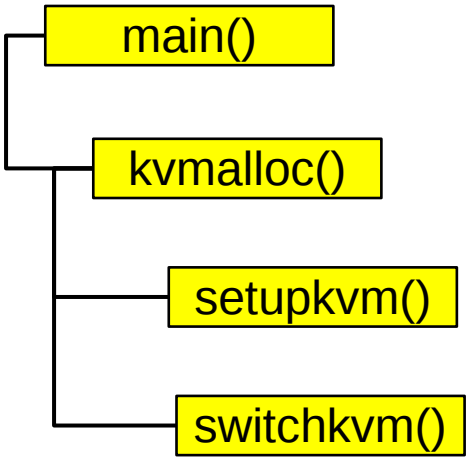
kvmalloc()

```
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 }
```

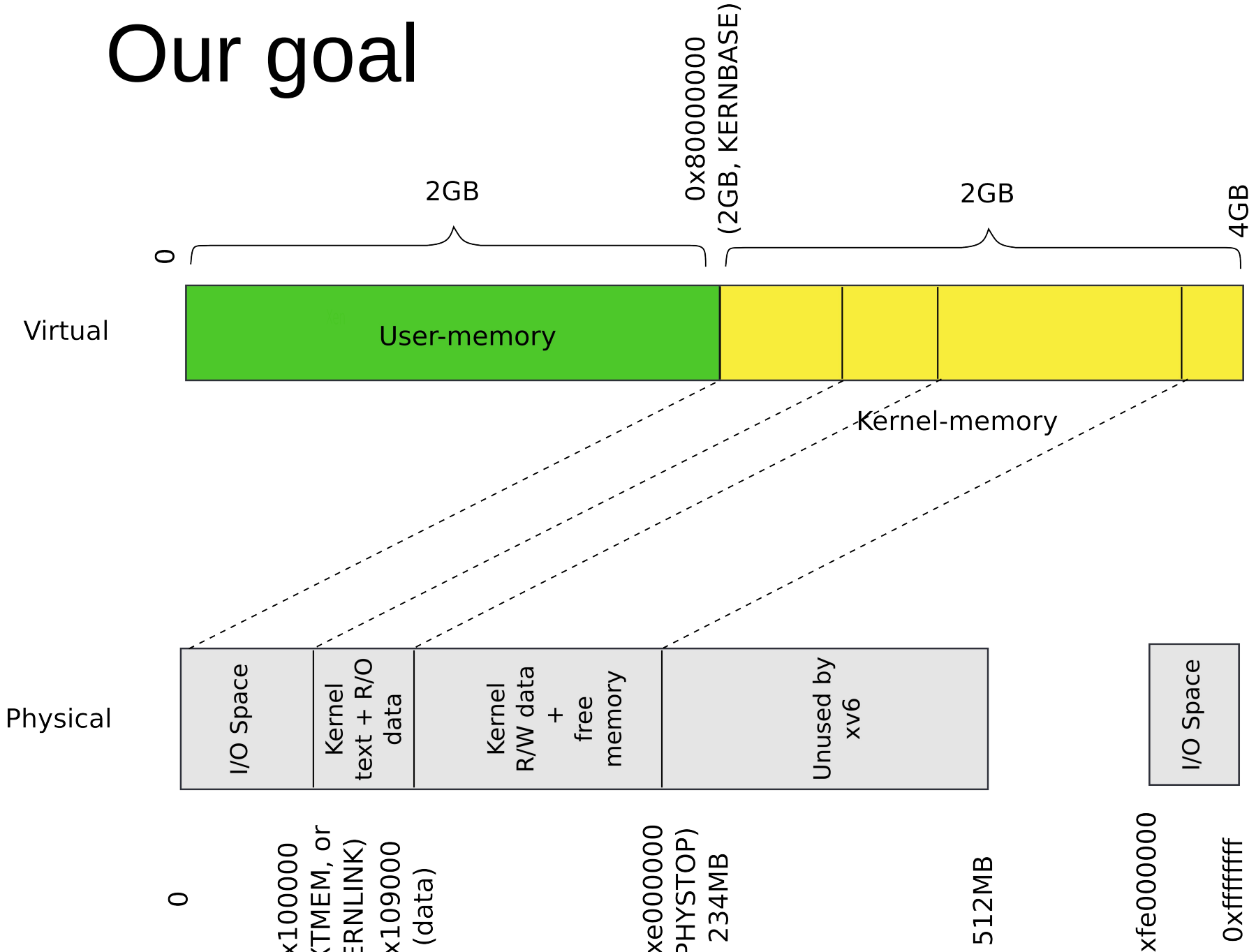


Conclusion

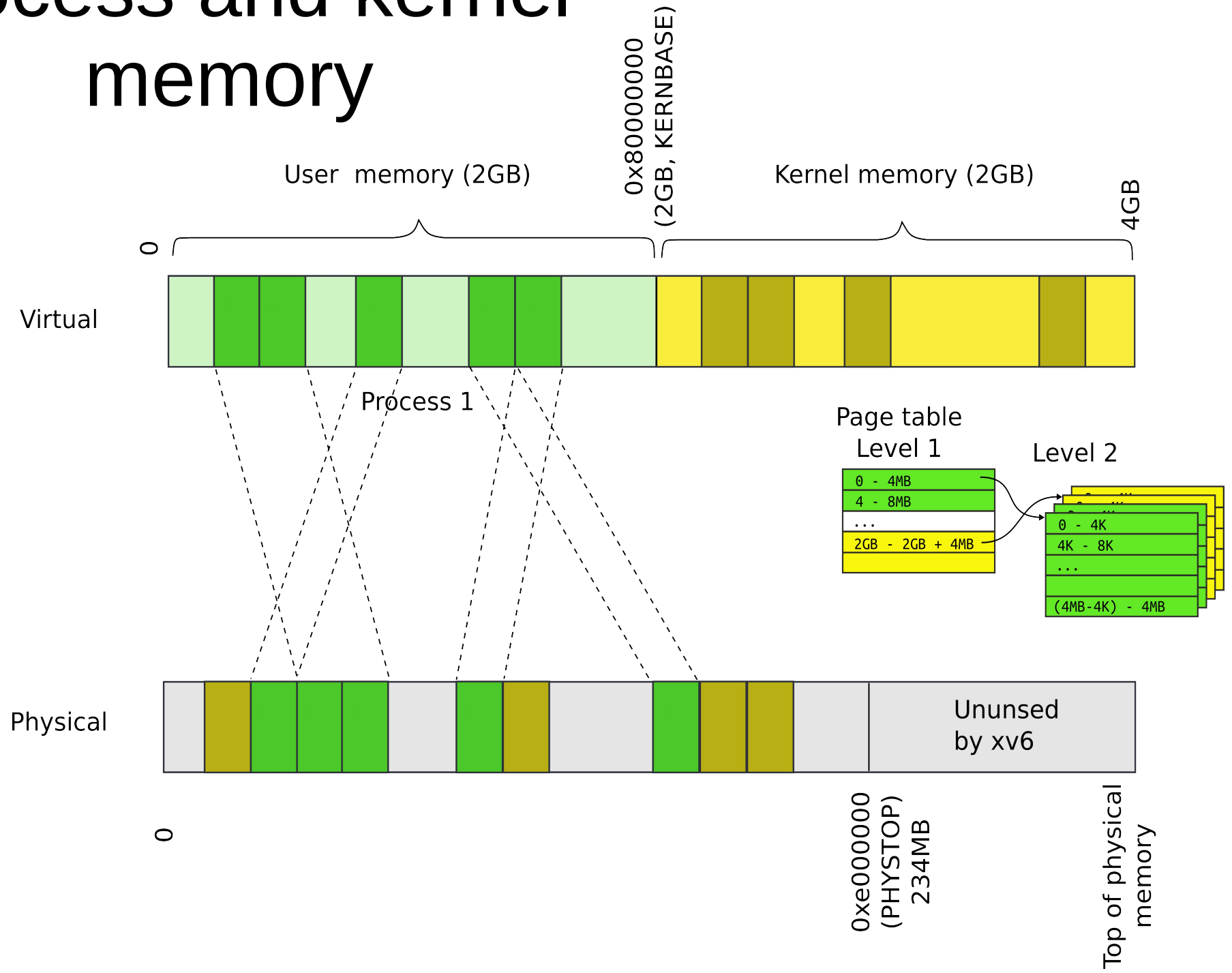
- Kernel has a 4KB page table

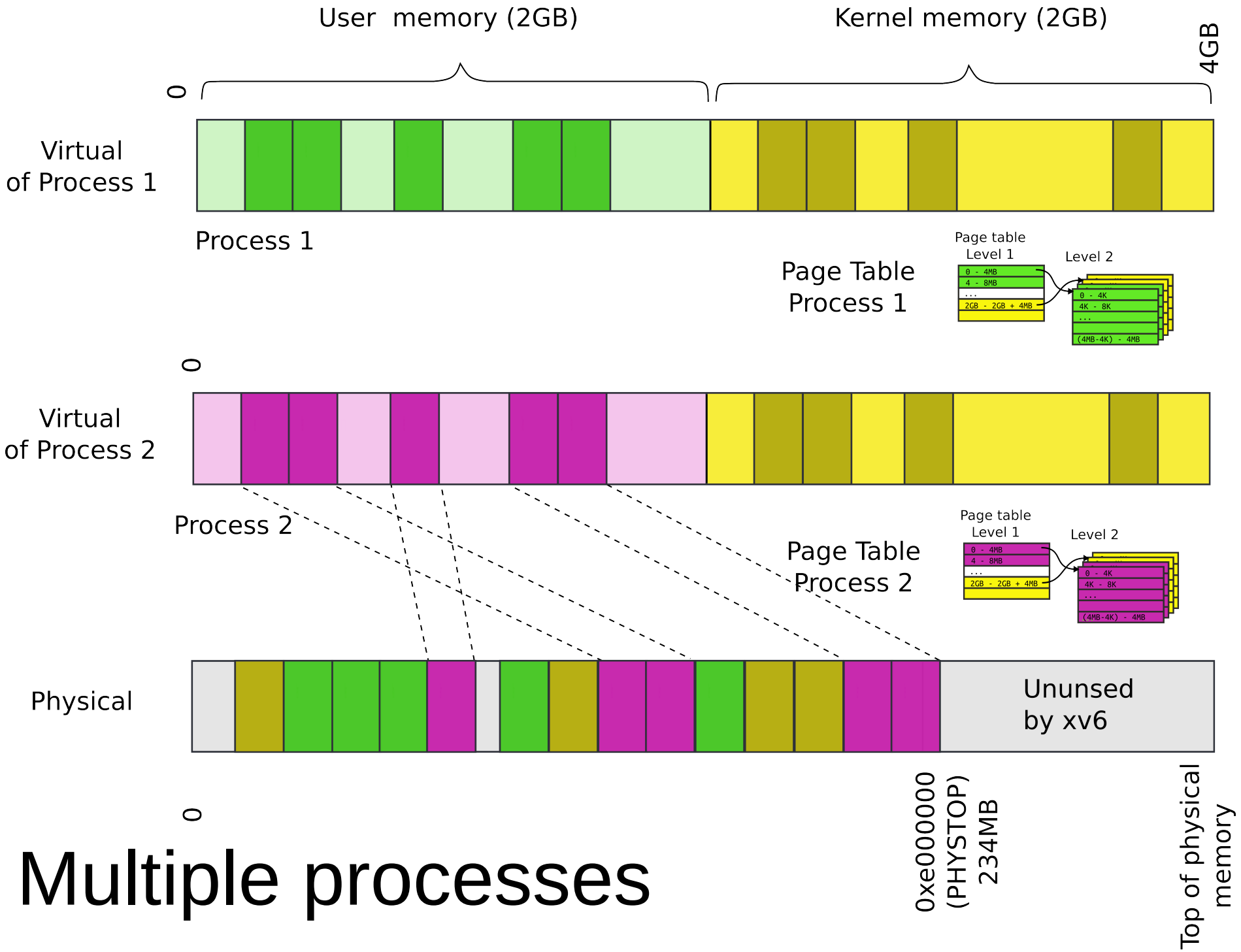
Thank you!

Our goal


0
2GB
0x80000000
(2GB, KERNBASE)
2GB
4GB
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

Process and kernel memory





Multiple processes

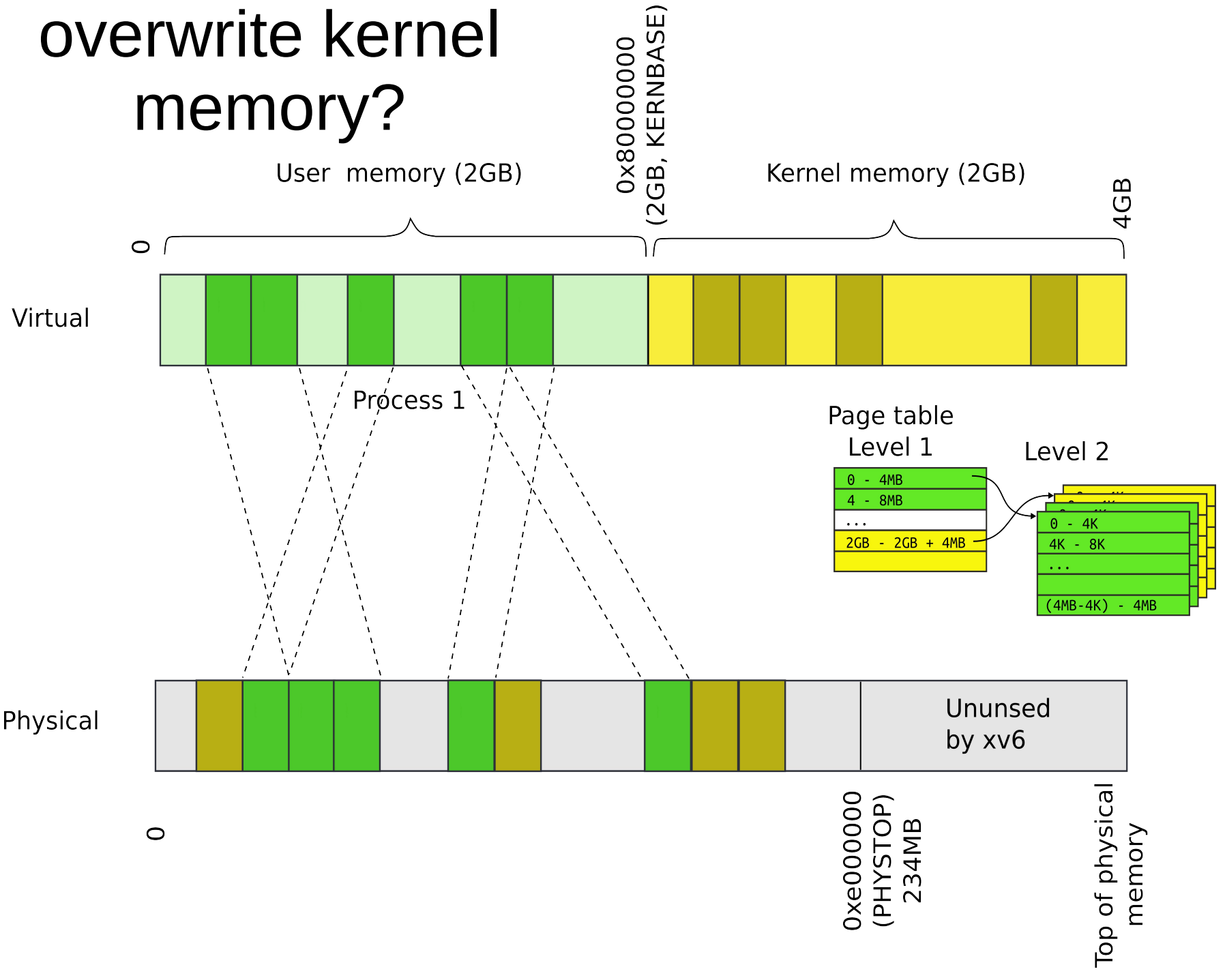
Walk page table

```
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 }
```

```
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 }
```

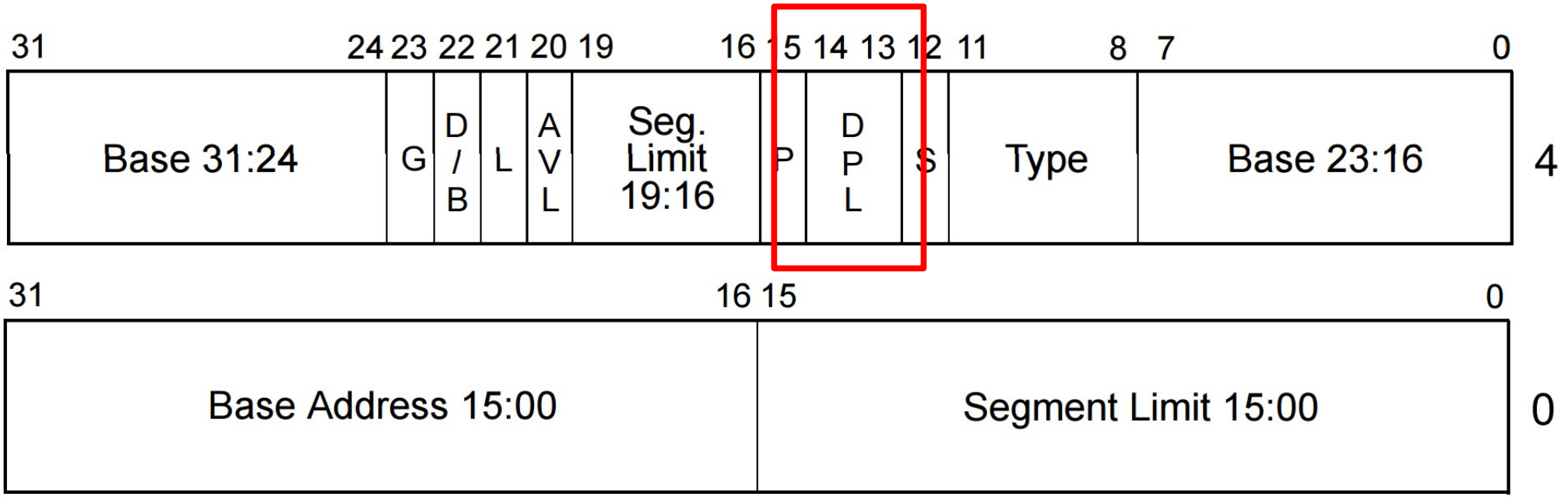
Create page table entries

Can a process overwrite kernel memory?



Privilege levels

- Each segment has a privilege level
  - DPL (descriptor privilege level)
  - 4 privilege levels ranging 0-3

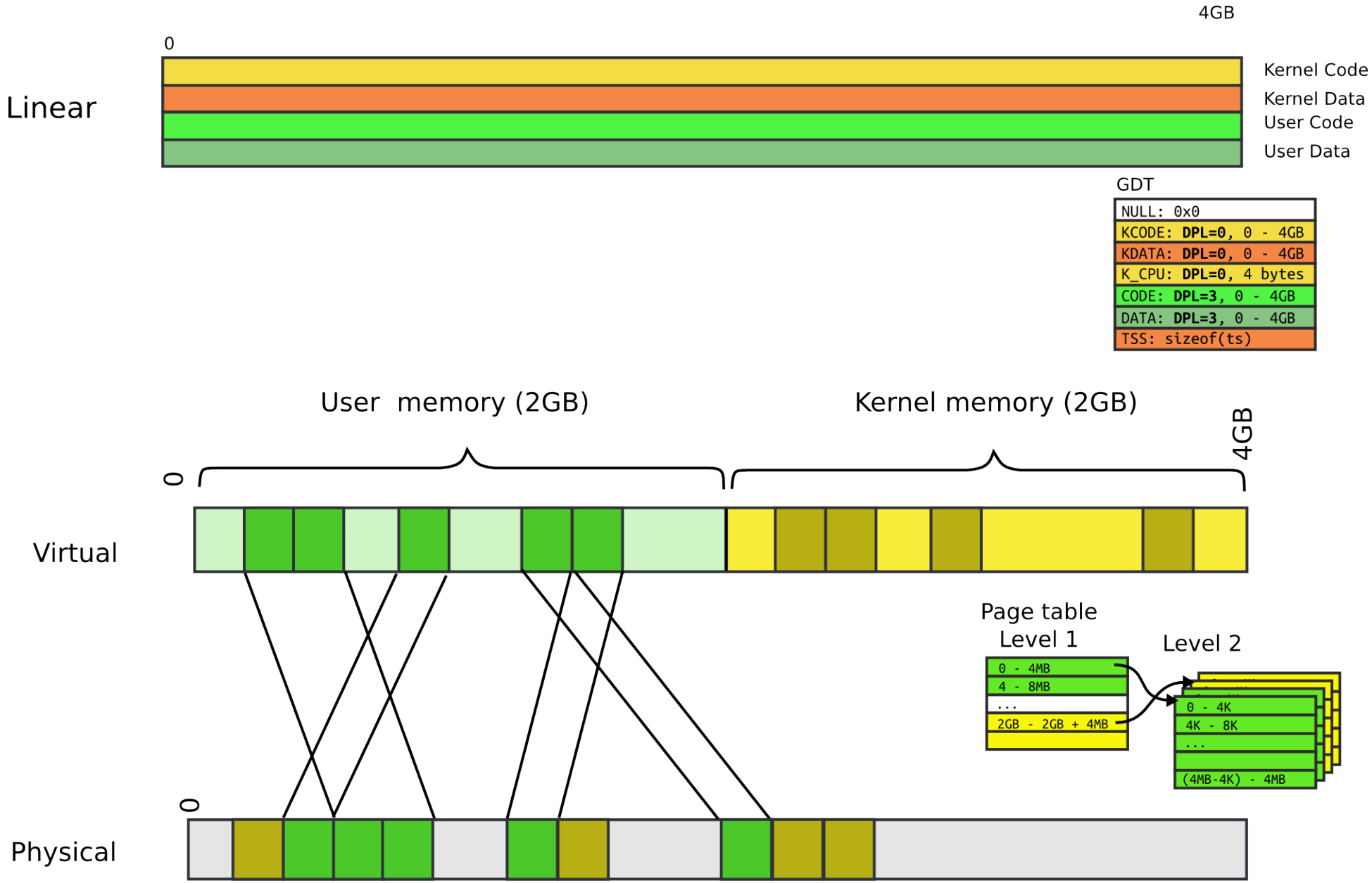


Privilege levels

- Currently running code also has privilege level
  - “Current privilege level” (CPL): 0-3
  - Can access only less privileged segments
    - E.g., 0 can access 1, 2, 3
- Some instructions are “privileged”
  - Can only be invoked at CPL = 0
  - Examples:
    - Load GDT
    - MOV <control register>
      - E.g. reload a page table by changing CR3

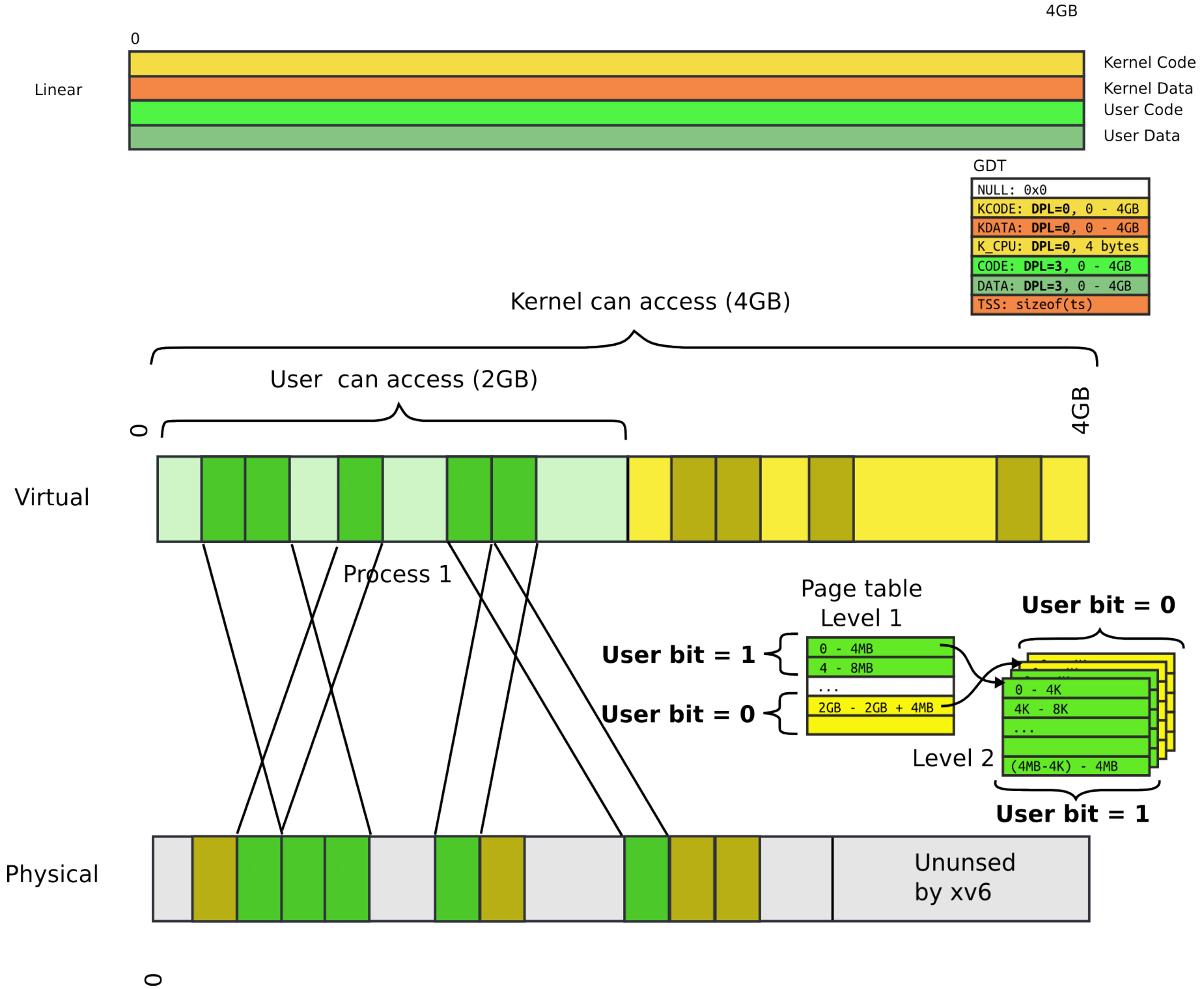
Real world

- Only two privilege levels are used in modern OSes:
  - OS kernel runs at 0
  - User code runs at 3
- This is called “flat” segment model
  - Segments for both 0 and 3 cover entire address space
- But then... how the kernel is protected?
  - Page tables


4GB
0
Linear
Kernel Code
Kernel Data
User Code
User Data
GDT
NULL: 0x0
KCODE: DPL=0, 0 - 4GB
KDATA: DPL=0, 0 - 4GB
K_CPU: DPL=0, 4 bytes
CODE: DPL=3, 0 - 4GB
DATA: DPL=3, 0 - 4GB
TSS: sizeof(ts)
User memory (2GB)
Kernel memory (2GB)
4GB
0
Virtual
Page table
Level 1
Level 2
0 - 4MB
4 - 8MB
...
2GB - 2GB + 4MB
0 - 4K
4K - 8K
...
(4MB-4K) - 4MB
0
Physical

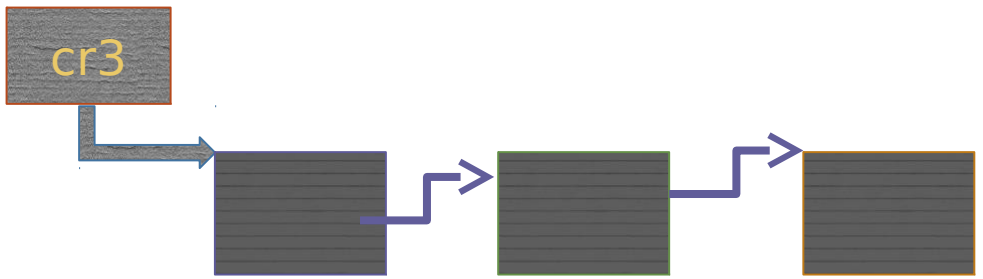
Page table: user bit

- Each entry (both Level 1 and Level 2) has a bit
  - If set, code at privilege level 3 can access
  - If not, only levels 0-2 can access
- Note, only 2 levels, not 4 like with segments
- All kernel code is mapped with the user bit clear
  - This protects user-level code from accessing the kernel


4GB
0
Linear
Kernel Code
Kernel Data
User Code
User Data
GDT
NULL: 0x0
KCODE: DPL=0, 0 - 4GB
KDATA: DPL=0, 0 - 4GB
K_CPU: DPL=0, 4 bytes
CODE: DPL=3, 0 - 4GB
DATA: DPL=3, 0 - 4GB
TSS: sizeof(ts)
Kernel can access (4GB)
User can access (2GB)
0
4GB
Virtual
Process 1
Page table
Level 1
User bit = 1
User bit = 0
0 - 4MB
4 - 8MB
...
2GB - 2GB + 4MB
User bit = 0
0 - 4K
4K - 8K
...
(4MB-4K) - 4MB
Level 2
User bit = 1
Physical
Ununsed
by xv6
0

TLB

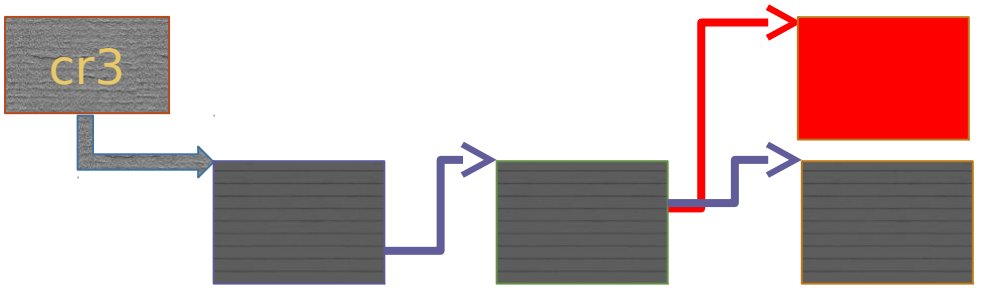
- CPU caches results of page table walks
  - In translation lookaside buffer (TLB)
- Walking page table is slow
  - Each memory access is 200-300 cycles on modern hardware
  - L3 cache access is 70 cycles



Virt	Phys
0xf0231000	0x1000
0x00b31000	0x1f000
0xb0002000	0xc1000
-	-

TLB

- TLB is a cache (in CPU)
  - It is not coherent with memory
  - If page table entry is changes, TLB remains the same and is out of sync



Virt	Phys
0xf0231000	0x1000
0x00b31000	0x1f000
0xb0002000	0xc1000
-	-

Same Virt Addr.

No Change!!!

Invalidating TLB

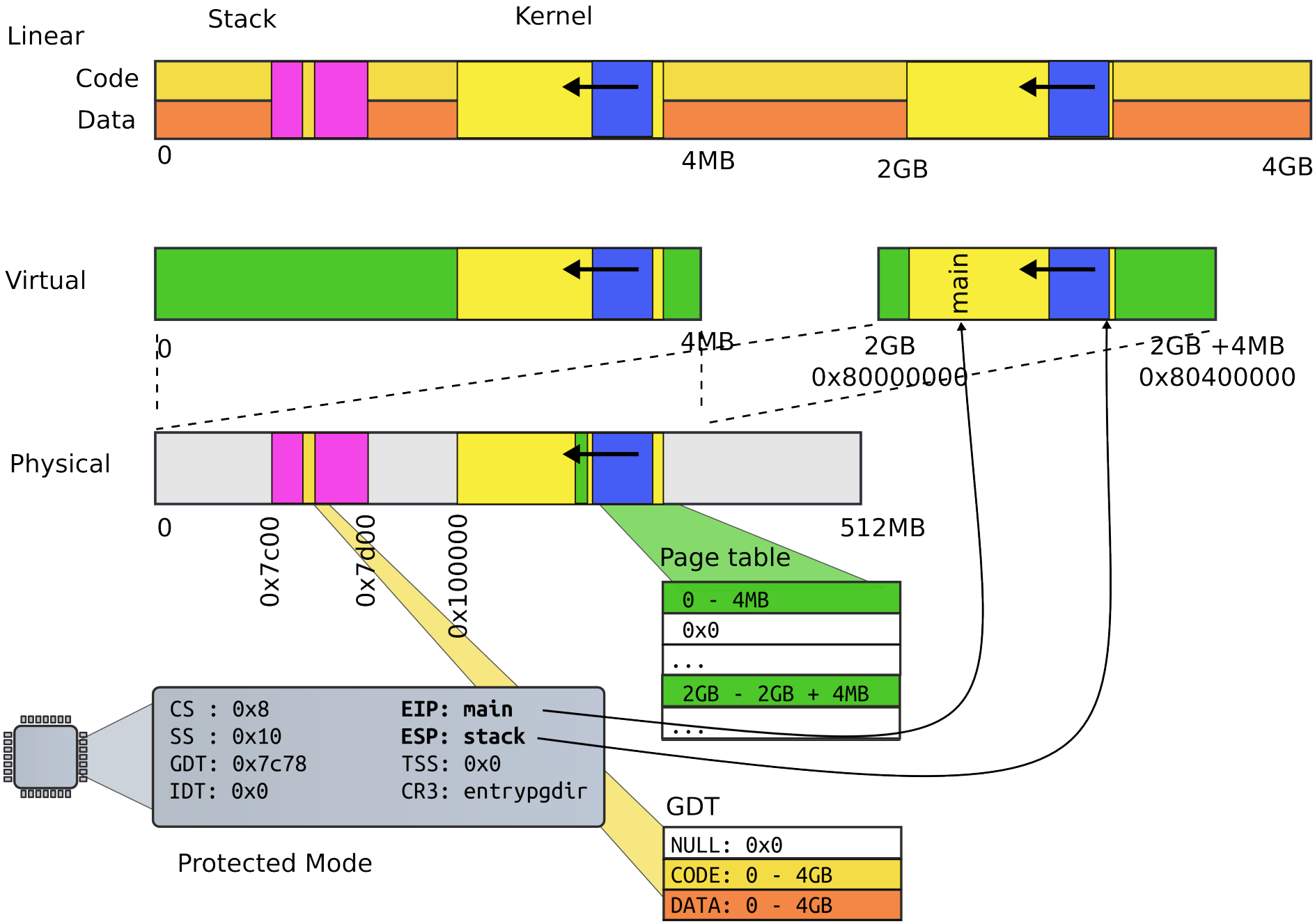
- After every page table update, OS needs to manually invalidate cached values
- Modern CPUs have “tagged TLBs”,
  - Each TLB entry has a “tag” – identifier of a process
  - No need to flush TLBs on context switch
- On Intel this mechanism is called
  - Process-Context Identifiers (PCIDs)

Questions?

```
1157 # Set up the stack pointer.
1158 movl $(stack + KSTACKSIZE), %esp
1159
1160 # Jump to main(), and switch to executing at
1161 # high addresses. The indirect call is needed
because
1162 # the assembler produces a PC-relative instruction
1163 # for a direct jump.
1164 mov $main, %eax
1165 jmp *%eax
1166
1167 .comm stack, KSTACKSIZE
```

How come `$main` makes sense?

Why is it there...0x80000000 + something?



Makefile

```
bootblock: bootasm.S bootmain.c
        $(CC) $(CFLAGS) -fno-pic -O -nostdinc -I. -c bootmain.c
        $(CC) $(CFLAGS) -fno-pic -nostdinc -I. -c bootasm.S
        $(LD) $(LDFLAGS) -N -e start -Ttext 0x7C00 -o bootblock.o bootasm.o bootmain.o
        $(OBJDUMP) -S bootblock.o > bootblock.asm
        $(OBJCOPY) -S -O binary -j .text bootblock.o bootblock


kernel: $(OBJS) entry.o entryother initcode kernel.ld
        $(LD) $(LDFLAGS) -T kernel.ld -o kernel entry.o $(OBJS) -b binary initcode
                                                                    entryother
        $(OBJDUMP) -S kernel > kernel.asm
        $(OBJDUMP) -t kernel | sed '1,/SYMBOL TABLE/d; s/ .* / /; /^$$/d' > kernel.sym
```

kernel.ld: Linker script

```
OUTPUT_FORMAT("elf32-i386", "elf32-i386", "elf32-i386")
OUTPUT_ARCH(i386)
ENTRY(_start)

SECTIONS
{
        /* Link the kernel at this address: "." means the current address */
        /* Must be equal to KERNLINK */
        . = 0x80100000;
….
        .bss : {
                *(.bss)
        }
        PROVIDE(end = .);
}
```