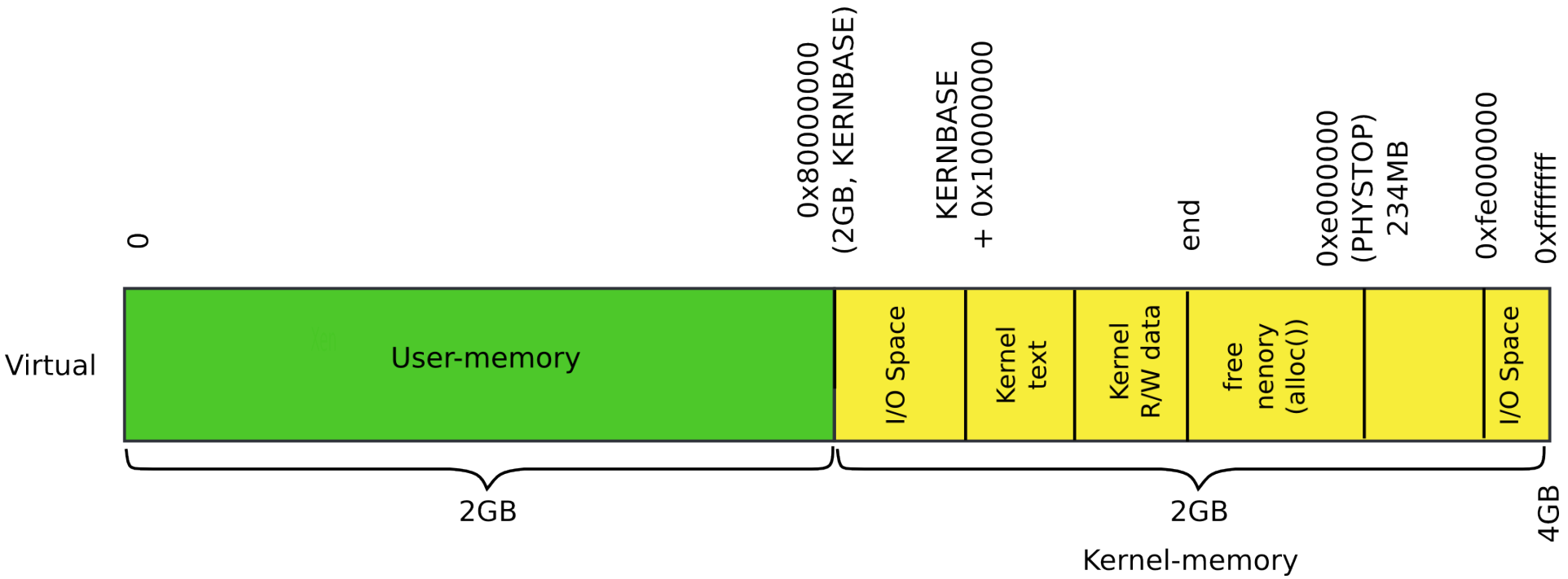
143A: Principles of Operating Systems

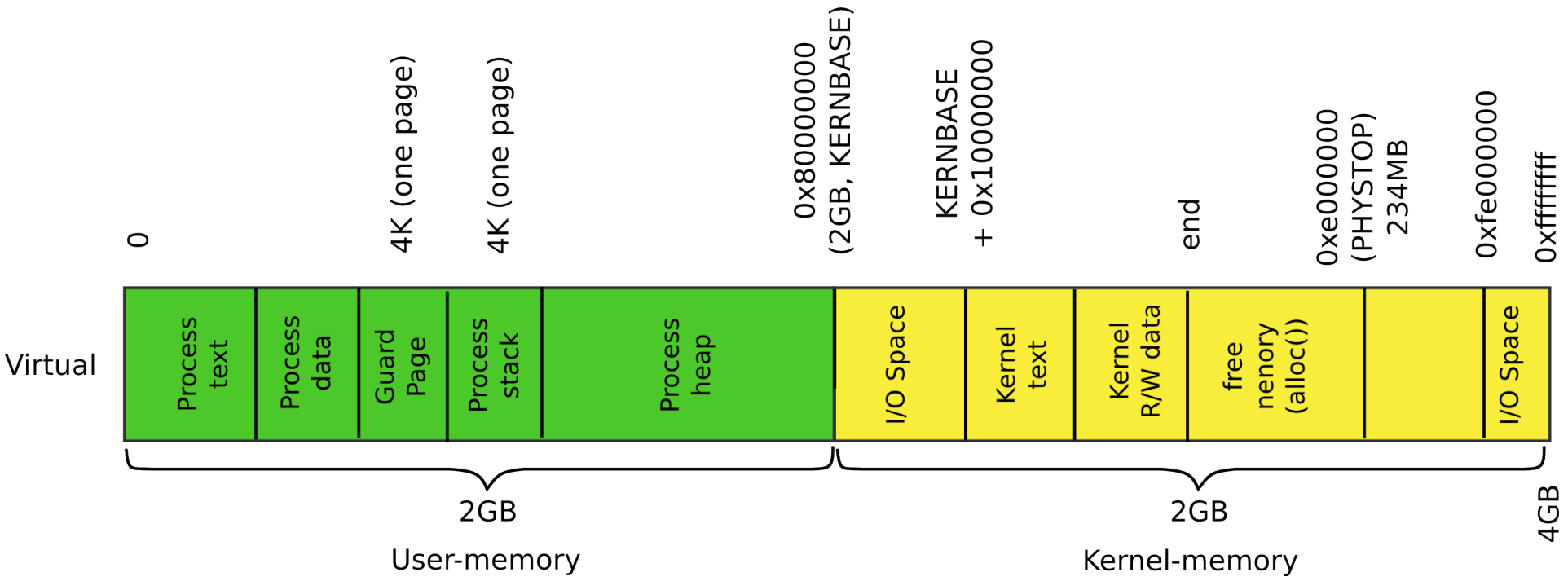
Lecture 11: Creating Processes

Anton Burtsev
October, 2017

Recap: kernel memory



Today: process memory



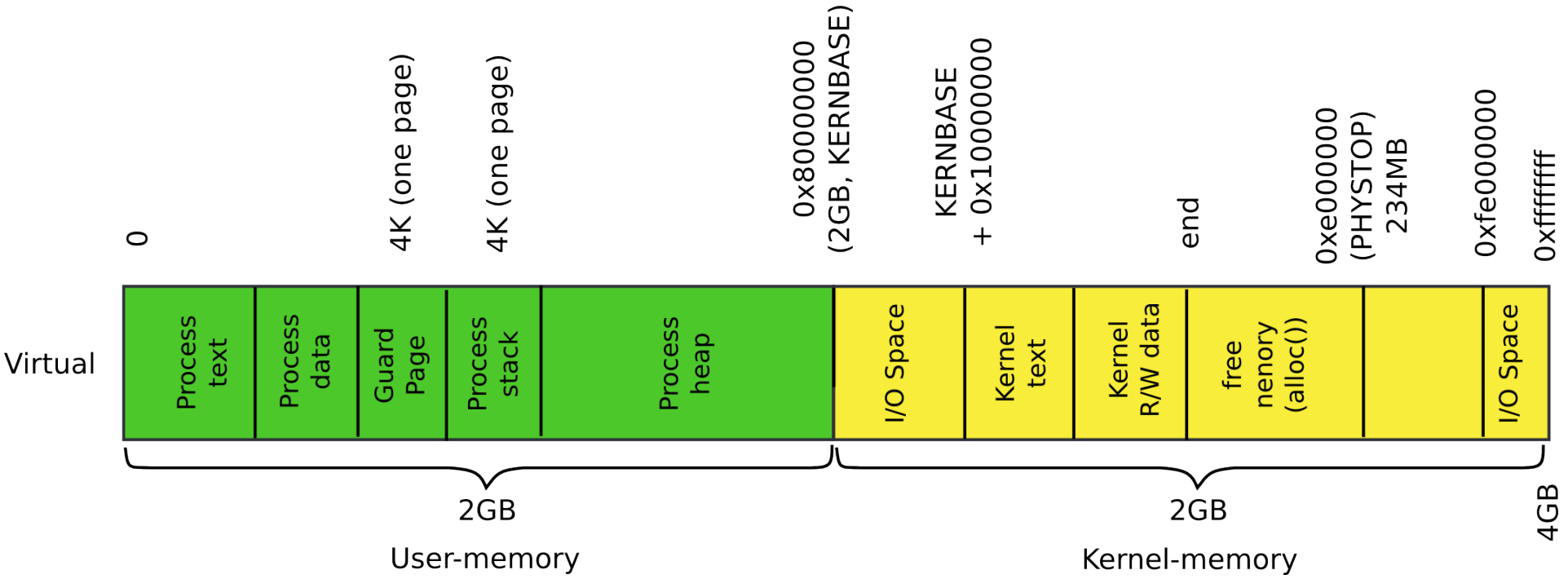
How does kernel creates new processes?

How does kernel creates new processes?

- Exec
  - exec(“/bin/ls”, argv);

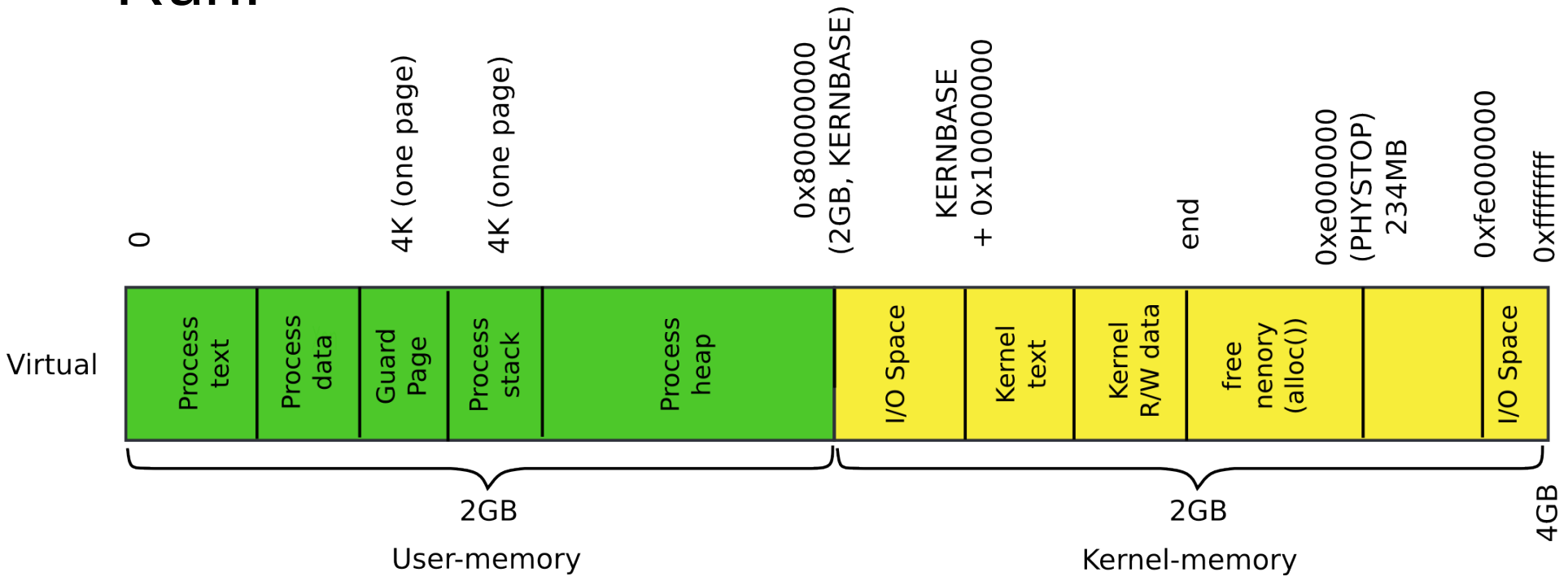
exec(): high-level outline

- We want to create the following layout
  - What shall we do?



exec(): high-level outline

- Load program from disk
- Create user-stack
- Run!



exec(): locate inode

```
6309 int
6310 exec(char *path, char **argv)
6311 {
...
6321   if((ip = namei(path)) == 0){
6322     end_op();
6323     return -1;
6324   }
6328   // Check ELF header
6329   if(readi(ip, (char*)&elf, 0, sizeof(elf)) <
                                          sizeof(elf))
6330     goto bad;
6331   if(elf.magic != ELF_MAGIC)
6332     goto bad;
```

exec(): check ELF header

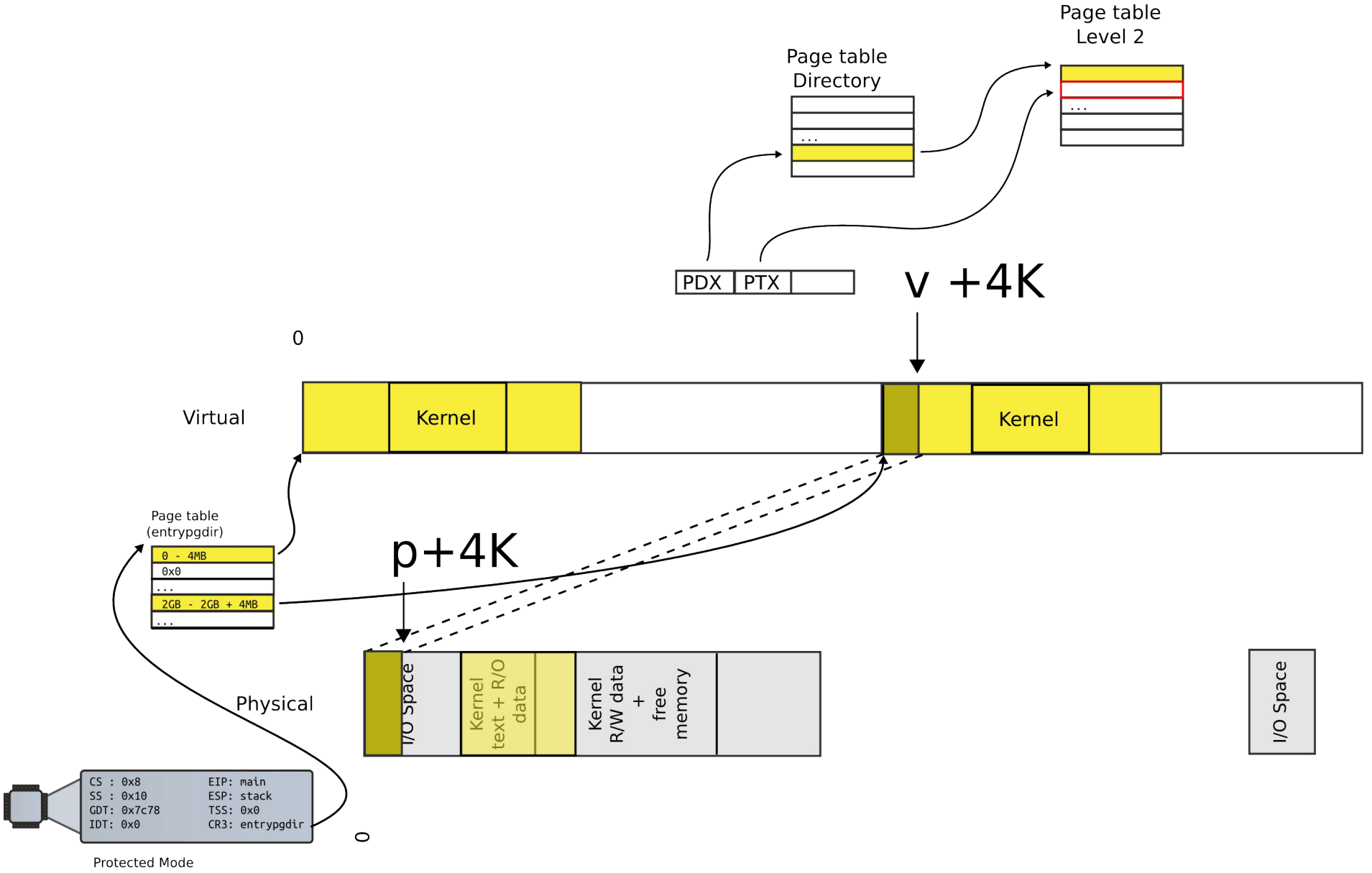
```
6309 int
6310 exec(char *path, char **argv)
6311 {
...
6321   if((ip = namei(path)) == 0){
6322     end_op();
6323     return -1;
6324   }
6328   // Check ELF header
6329   if(readi(ip, (char*)&elf, 0, sizeof(elf)) <
                                              sizeof(elf))
6330     goto bad;
6331   if(elf.magic != ELF_MAGIC)
6332     goto bad;
```

Exec(): Setup kernel address space()

```
6331    if(elf.magic != ELF_MAGIC)
6332      goto bad;
6333
6334    if((pgdir = setupkvm()) == 0)
6335      goto bad;
6336
```

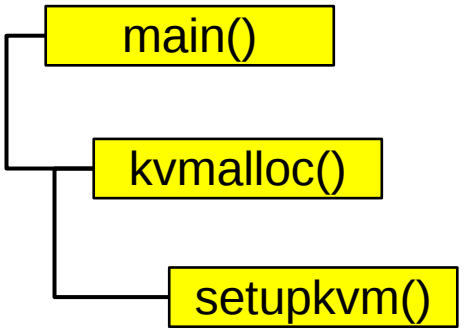
- Remember from last time?

setupkvm(): Move to next page



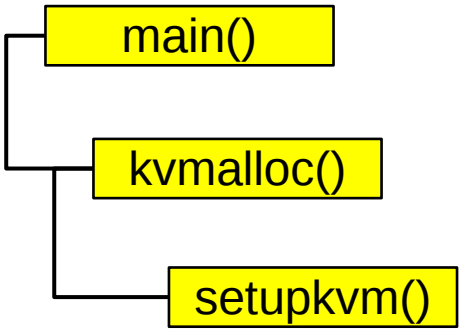
Allocate page table directory

```
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 }
```



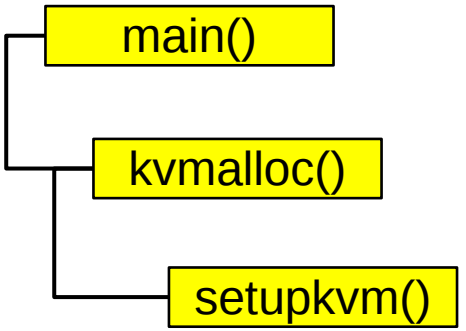
Iterate in a loop: remap physical pages

```
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 }
```

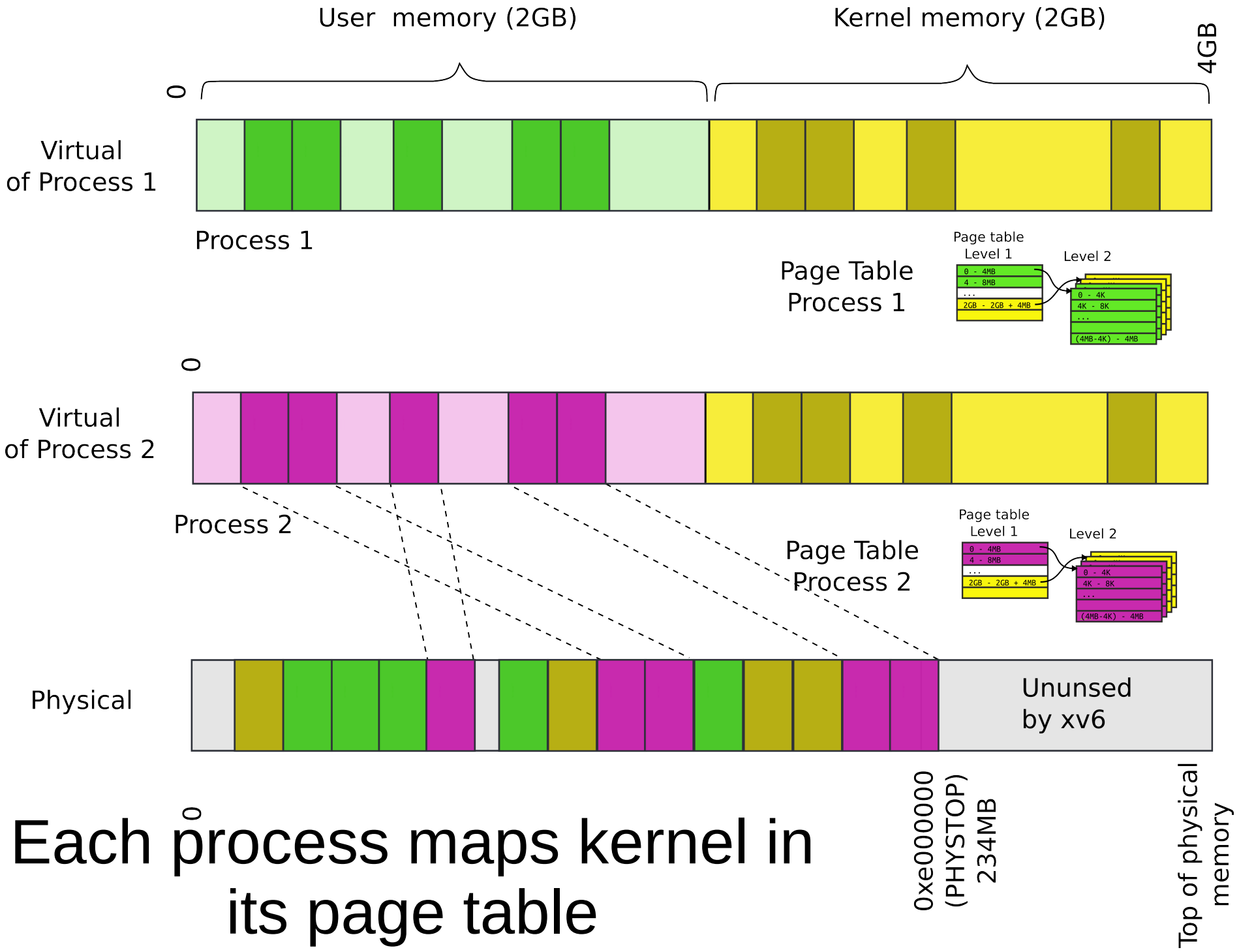


Iterate in a loop: remap physical pages

```
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 }
```



But why are we doing that?



Each process maps kernel in its page table

Setup kernel address space()

```
6333
6334    if((pgdir = setupkvm()) == 0)
6335      goto bad;
6336
```

- Remember from last time?

Load program into memory

```
6337 // Load program into memory.
6338 sz = 0;
6339 for(i=0, off=elf.phoff; i<elf.phnum; i++, off+=sizeof(ph)){
6340   if(readi(ip, (char*)&ph, off, sizeof(ph)) != sizeof(ph))
6341     goto bad;
...
6348   if((sz = allocuvm(pgdir, sz, ph.vaddr + ph.memsz)) == 0)
6349     goto bad;
6350   if(ph.vaddr % PGSIZE != 0)
6351     goto bad;
6352   if(loaduvm(pgdir, (char*)ph.vaddr, ip, ph.off, ph.filesz)
< 0)
6353     goto bad;
6354 }
```

- Set the address space size to 0

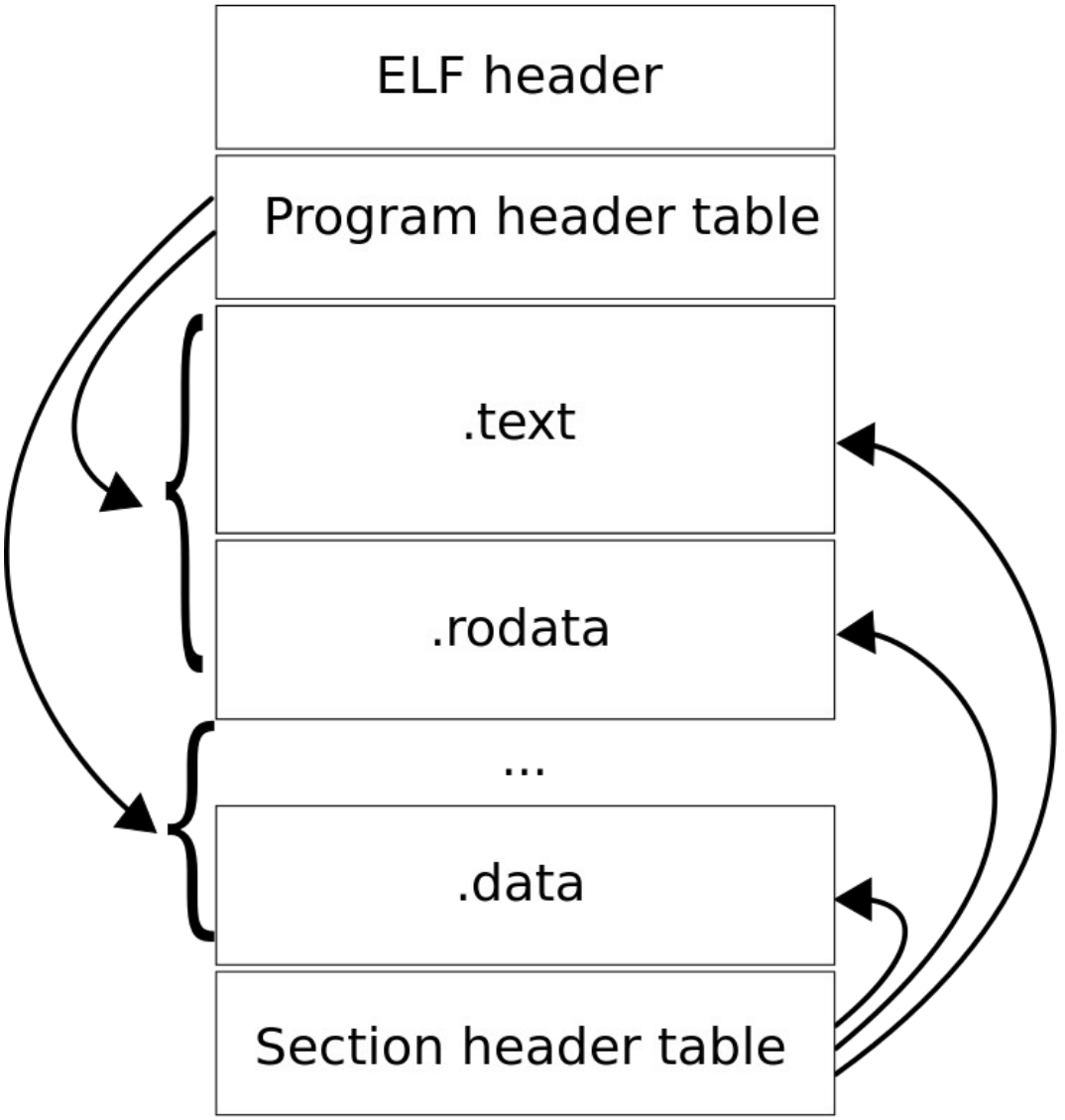
Load program into memory

```
6337 // Load program into memory.
6338 sz = 0;
6339 for(i=0, off=elf.phoff; i<elf.phnum; i++, off+=sizeof(ph)){
6340   if(readi(ip, (char*)&ph, off, sizeof(ph)) != sizeof(ph))
6341     goto bad;
...
6348   if((sz = allocuvm(pgdir, sz, ph.vaddr + ph.memsz)) == 0)
6349     goto bad;
6350   if(ph.vaddr % PGSIZE != 0)
6351     goto bad;
6352   if(loaduvm(pgdir, (char*)ph.vaddr, ip, ph.off, ph.filesz)
< 0)
6353     goto bad;
6354 }
```

- Loop over all program headers

ELF binary

- ELF header
- Program header table
  - Each entry describes a section of a program
  - Instruction, data



Load program into memory

```
6337 // Load program into memory.
6338 sz = 0;
6339 for(i=0, off=elf.phoff; i<elf.phnum; i++, off+=sizeof(ph)){
6340   if(readi(ip, (char*)&ph, off, sizeof(ph)) != sizeof(ph))
6341     goto bad;
...
6348   if((sz = allocuvm(pgdir, sz, ph.vaddr + ph.memsz)) == 0)
6349     goto bad;
6350   if(ph.vaddr % PGSIZE != 0)
6351     goto bad;
6352   if(loaduvm(pgdir, (char*)ph.vaddr, ip, ph.off, ph.filesz)
< 0)
6353     goto bad;
6354 }
```

- Start at the beginning of the program header table
  off = elf.phoff

Load program into memory

```
6337 // Load program into memory.
6338 sz = 0;
6339 for(i=0, off=elf.phoff; i<elf.phnum; i++, off+=sizeof(ph)){
6340   if(readi(ip, (char*)&ph, off, sizeof(ph)) != sizeof(ph))
6341     goto bad;
...
6348   if((sz = allocuvm(pgdir, sz, ph.vaddr + ph.memsz)) == 0)
6349     goto bad;
6350   if(ph.vaddr % PGSIZE != 0)
6351     goto bad;
6352   if(loaduvm(pgdir, (char*)ph.vaddr, ip, ph.off, ph.filesz)
< 0)
6353     goto bad;
6354 }
```

- Read one program header entry at a time

Load program into memory

```
6337 // Load program into memory.
6338 sz = 0;
6339 for(i=0, off=elf.phoff; i<elf.phnum; i++, off+=sizeof(ph)){
6340   if(readi(ip, (char*)&ph, off, sizeof(ph)) != sizeof(ph))
6341     goto bad;
...
6348   if((sz = allocuvm(pgdir, sz, ph.vaddr + ph.memsz)) == 0)
6349     goto bad;
6350   if(ph.vaddr % PGSIZE != 0)
6351     goto bad;
6352   if(loaduvm(pgdir, (char*)ph.vaddr, ip, ph.off, ph.filesz)
< 0)
6353     goto bad;
6354 }
```

- Read one program header entry at a time
- Each time increment offset (off)

Load program into memory

```
6337 // Load program into memory.
6338 sz = 0;
6339 for(i=0, off=elf.phoff; i<elf.phnum; i++, off+=sizeof(ph)){
6340   if(readi(ip, (char*)&ph, off, sizeof(ph)) != sizeof(ph))
6341     goto bad;
...
6348   if((sz = allocuvm(pgdir, sz, ph.vaddr + ph.memsz)) == 0)
6349     goto bad;
6350   if(ph.vaddr % PGSIZE != 0)
6351     goto bad;
6352   if(loaduvm(pgdir, (char*)ph.vaddr, ip, ph.off, ph.filesz)
< 0)
6353     goto bad;
6354 }
```

- Alloc pages for program section, e.g., text

Load program into memory

```
6337 // Load program into memory.
6338 sz = 0;
6339 for(i=0, off=elf.phoff; i<elf.phnum; i++, off+=sizeof(ph)){
6340   if(readi(ip, (char*)&ph, off, sizeof(ph)) != sizeof(ph))
6341     goto bad;
...
6348   if((sz = allocuvm(pgdir, sz, ph.vaddr + ph.memsz)) == 0)
6349     goto bad;
6350   if(ph.vaddr % PGSIZE != 0)
6351     goto bad;
6352   if(loaduvm(pgdir, (char*)ph.vaddr, ip, ph.off, ph.filesz)
< 0)
6353     goto bad;
6354 }
```

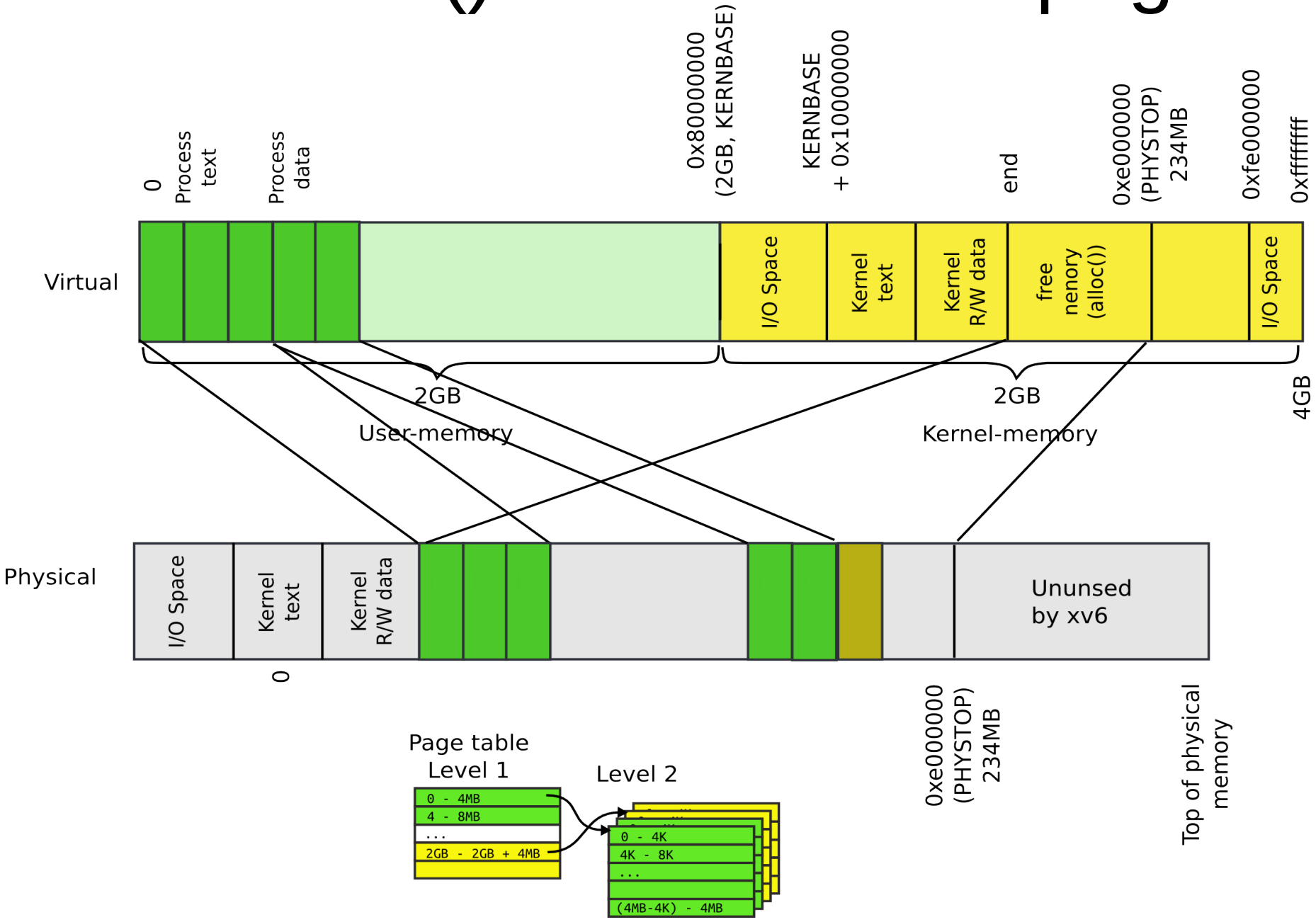
- Size of the user space is initially 0

Load program into memory

```
6337 // Load program into memory.
6338 sz = 0;
6339 for(i=0, off=elf.phoff; i<elf.phnum; i++, off+=sizeof(ph)){
6340   if(readi(ip, (char*)&ph, off, sizeof(ph)) != sizeof(ph))
6341     goto bad;
...
6348   if((sz = allocuvm(pgdir, sz, ph.vaddr + ph.memsz)) == 0)
6349     goto bad;
6350   if(ph.vaddr % PGSIZE != 0)
6351     goto bad;
6352   if(loaduvm(pgdir, (char*)ph.vaddr, ip, ph.off, ph.filesz)
< 0)
6353     goto bad;
6354 }
```

- Allocate memory for user pages
  - ph.vaddr is also 0

allocuvm(): allocate user pages



```
1953 allocuvm(pde_t *pgdir, uint oldsz, uint newsz)
1954 {
...
1958   if(newsz >= KERNBASE)
1959     return 0;
...
1964   for(; a < newsz; a += PGSIZE){
1965     mem = kalloc();
...
1971     memset(mem, 0, PGSIZE);
1972     if(mappages(pgdir, (char*)a, PGSIZE, V2P(mem), PTE_W|PTE_U) < 0){
...
1976       return 0;
1977     }
1978   }
1979   return newsz;
1980 }
```

Allocate user pages

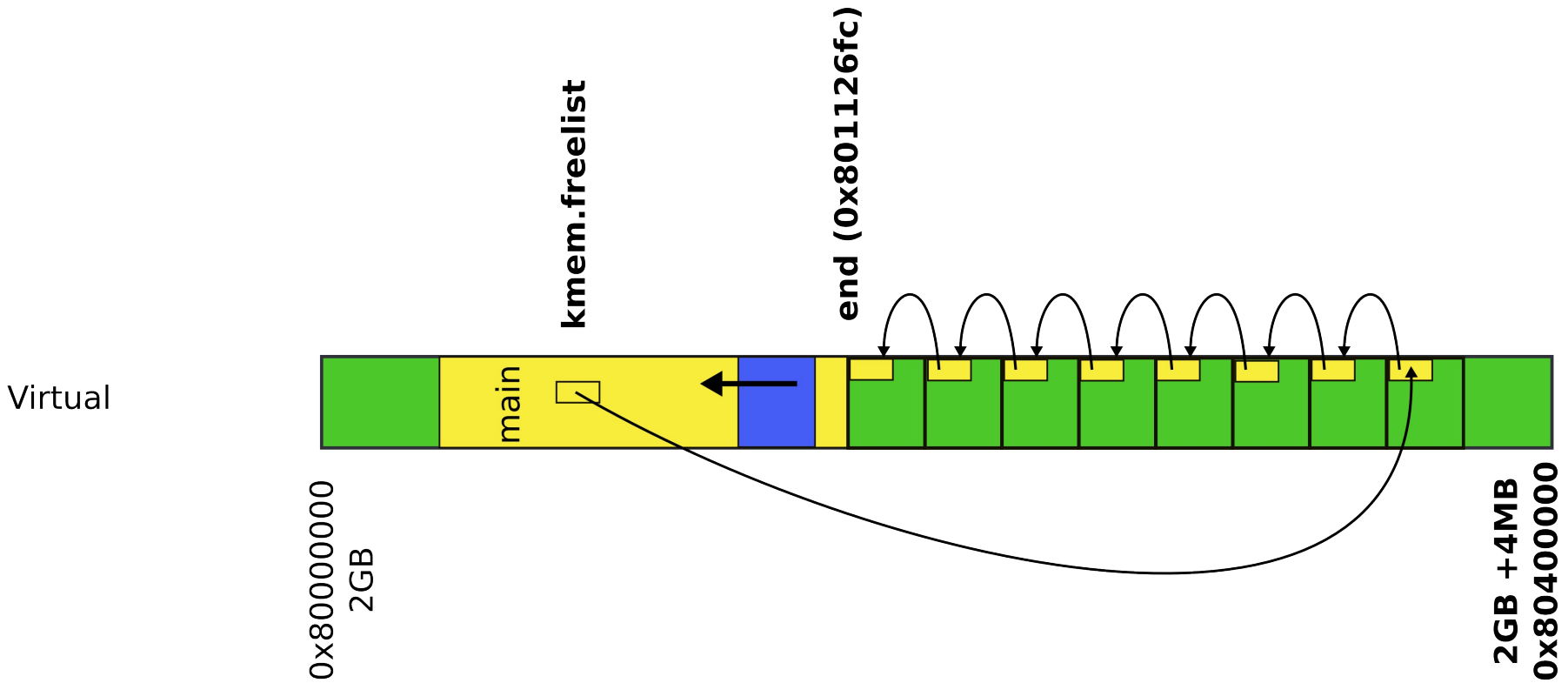
- New size can't be over 2GB

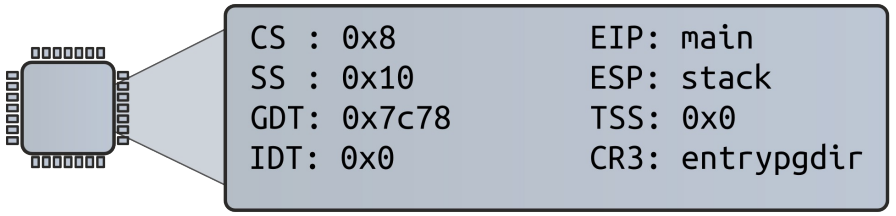
```
1953 allocuvm(pde_t *pgdir, uint oldsz, uint newsz)
1954 {
...
1958   if(newsz >= KERNBASE)
1959     return 0;
...
1964   for(; a < newsz; a += PGSIZE){
1965     mem = kalloc();
...
1971     memset(mem, 0, PGSIZE);
1972     if(mappages(pgdir, (char*)a, PGSIZE, V2P(mem), PTE_W|PTE_U) < 0){
...
1976       return 0;
1977     }
1978   }
1979   return newsz;
1980 }
```

Allocate user pages

- Allocate a page

Kernel memory allocator





Protected Mode

```
1953 allocuvm(pde_t *pgdir, uint oldsz, uint newsz)
1954 {
...
1958   if(newsz >= KERNBASE)
1959     return 0;
...
1964   for(; a < newsz; a += PGSIZE){
1965     mem = kalloc();
...
1971     memset(mem, 0, PGSIZE);
1972     if(mappages(pgdir, (char*)a, PGSIZE, V2P(mem), PTE_W|PTE_U) < 0){
...
1976       return 0;
1977     }
1978   }
1979   return newsz;
1980 }
```

Allocate user pages

- Set page to 0

```
1953 allocuvm(pde_t *pgdir, uint oldsz, uint newsz)
1954 {
...
1958   if(newsz >= KERNBASE)
1959     return 0;
...
1964   for(; a < newsz; a += PGSIZE){
1965     mem = kalloc();
...
1971     memset(mem, 0, PGSIZE);
1972     if(mappages(pgdir, (char*)a, PGSIZE, V2P(mem), PTE_W|PTE_U) < 0){
...
1976       return 0;
1977     }
1978   }
1979   return newsz;
1980 }
```

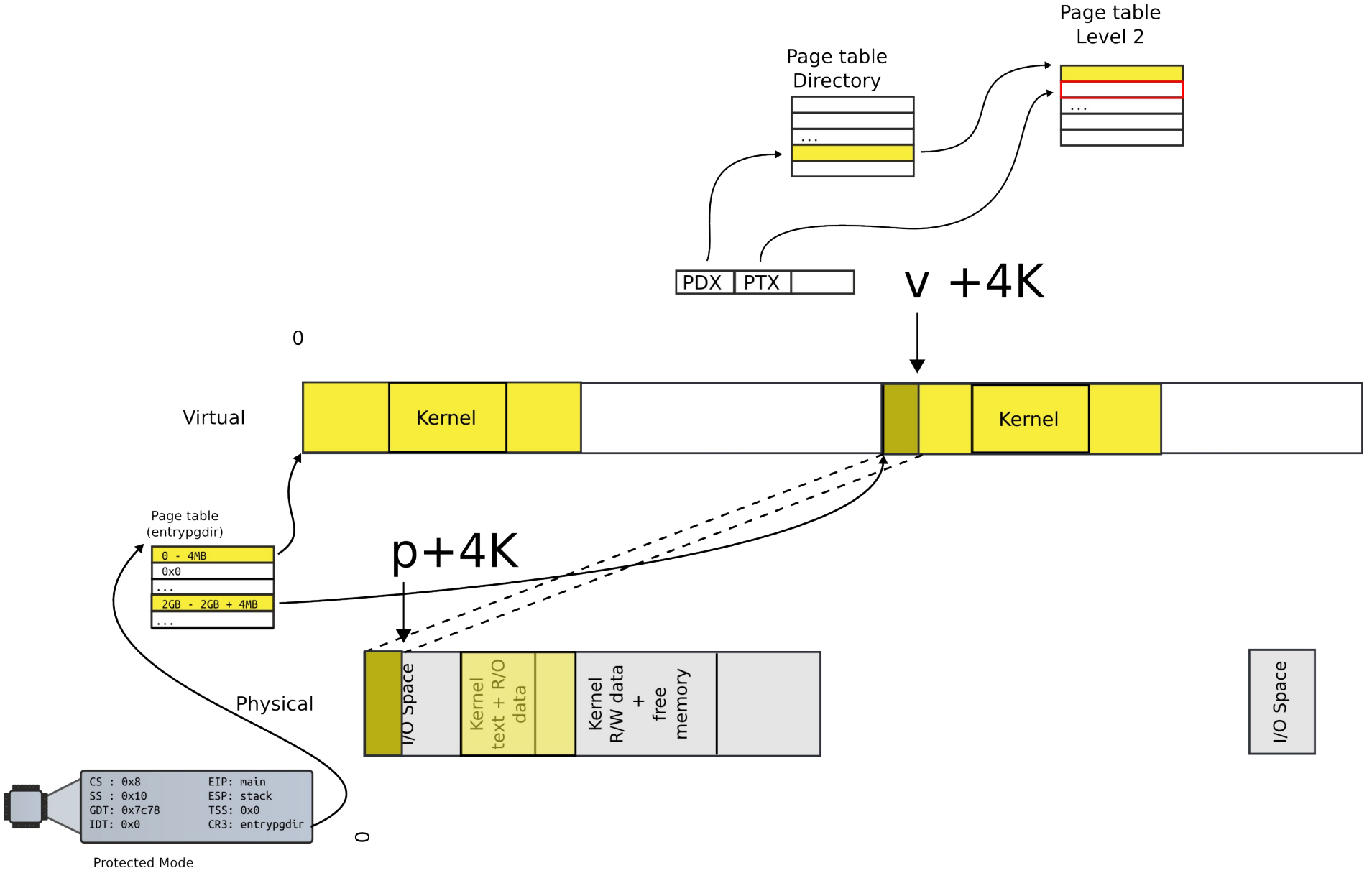
Allocate user pages

- Map the page

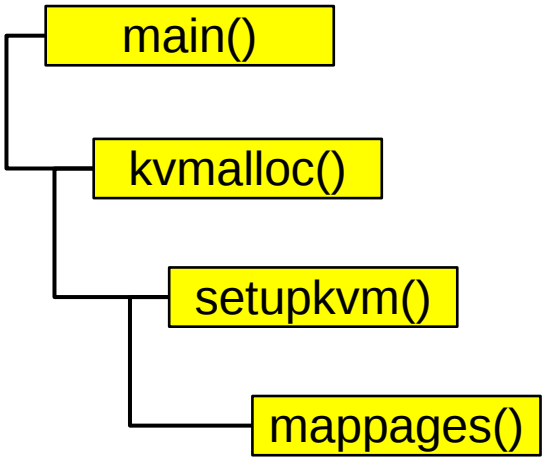
Who remembers mappages()?

- Remember we want a region of memory to be mapped
  - i.e., appear in the page table

mappages(): map a region



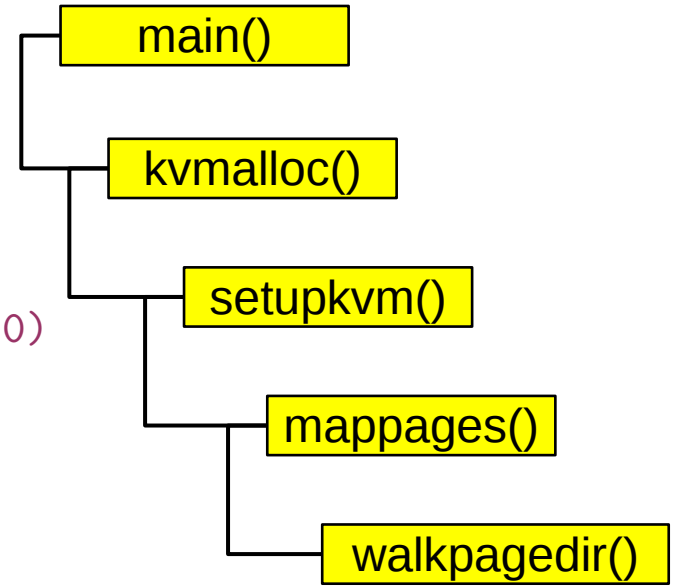
```
1779 mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm)
1780 {
1781   char *a, *last;
1782   pte_t *pte;
1783
1784   a = (char*)PGROUNDDOWN((uint)va);
1785   last = (char*)PGROUNDDOWN(((uint)va) + size - 1);
1786   for(;;){
1787     if((pte = walkpgdir(pgdir, a, 1)) == 0)
1788       return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

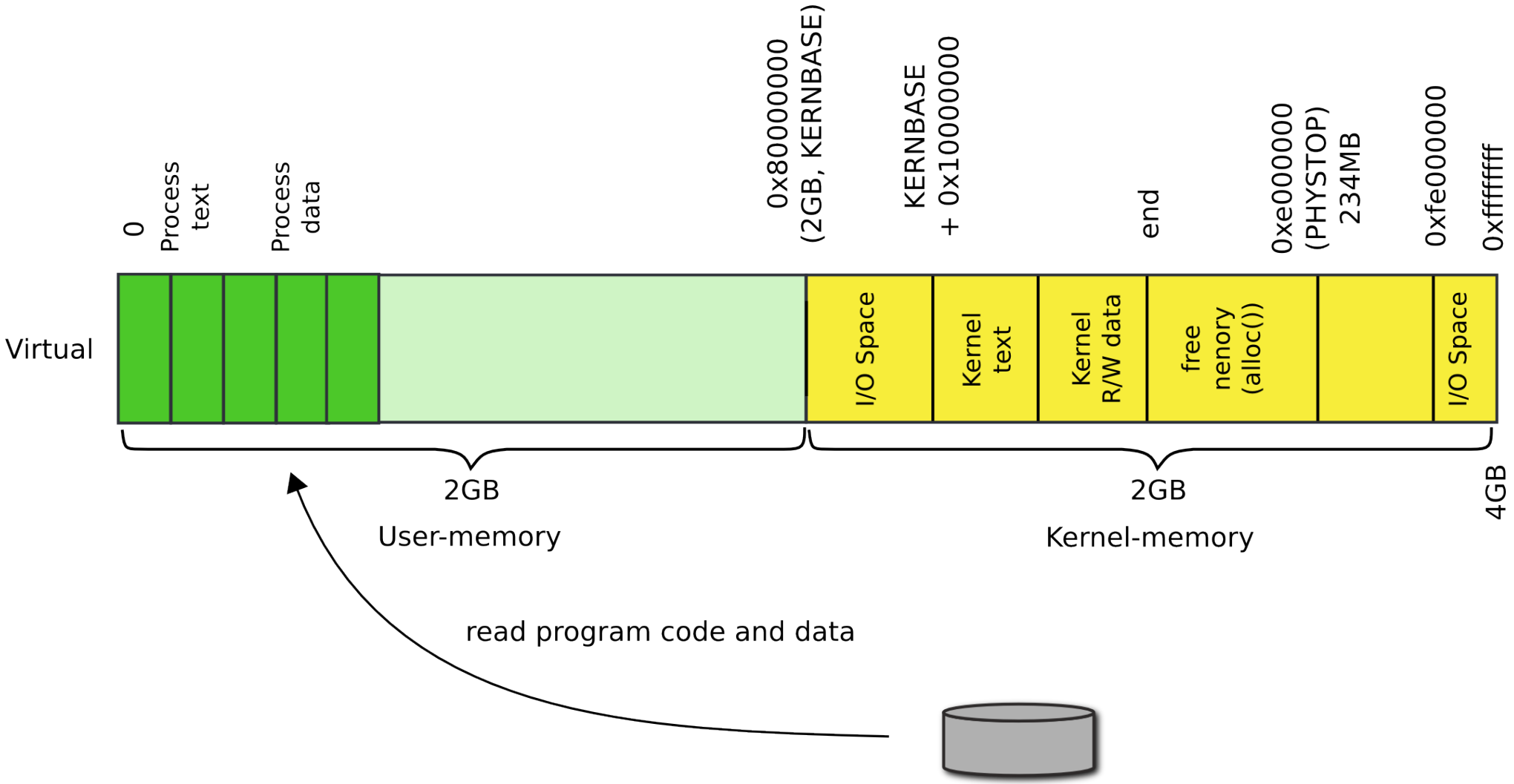


Load program into memory

```
6337 // Load program into memory.
6338 sz = 0;
6339 for(i=0, off=elf.phoff; i<elf.phnum; i++, off+=sizeof(ph)){
6340   if(readi(ip, (char*)&ph, off, sizeof(ph)) != sizeof(ph))
6341     goto bad;
...
6348   if((sz = allocuvm(pgdir, sz, ph.vaddr + ph.memsz)) == 0)
6349     goto bad;
6350   if(ph.vaddr % PGSIZE != 0)
6351     goto bad;
6352   if(loaduvm(pgdir, (char*)ph.vaddr, ip, ph.off, ph.filesz)
< 0)
6353     goto bad;
6354 }
```

- Load program section from disk

loaduvm(): read program from disk



```
1918 loaduvm(pde_t *pgdir, char *addr, struct inode *ip, uint offset, uint
sz)
1919 {
...
1925   for(i = 0; i < sz; i += PGSIZE){
1926     if((pte = walkpgdir(pgdir, addr+i, 0)) == 0)
1927       panic("loaduvm: address should exist");
1928     pa = PTE_ADDR(*pte);
1929     if(sz - i < PGSIZE)
1930       n = sz - i;
1931     else
1932       n = PGSIZE;
1933     if(readi(ip, P2V(pa), offset+i, n) != n)
1934       return -1;
1935   }
1936   return 0;
1937 }
```

Load program into memory

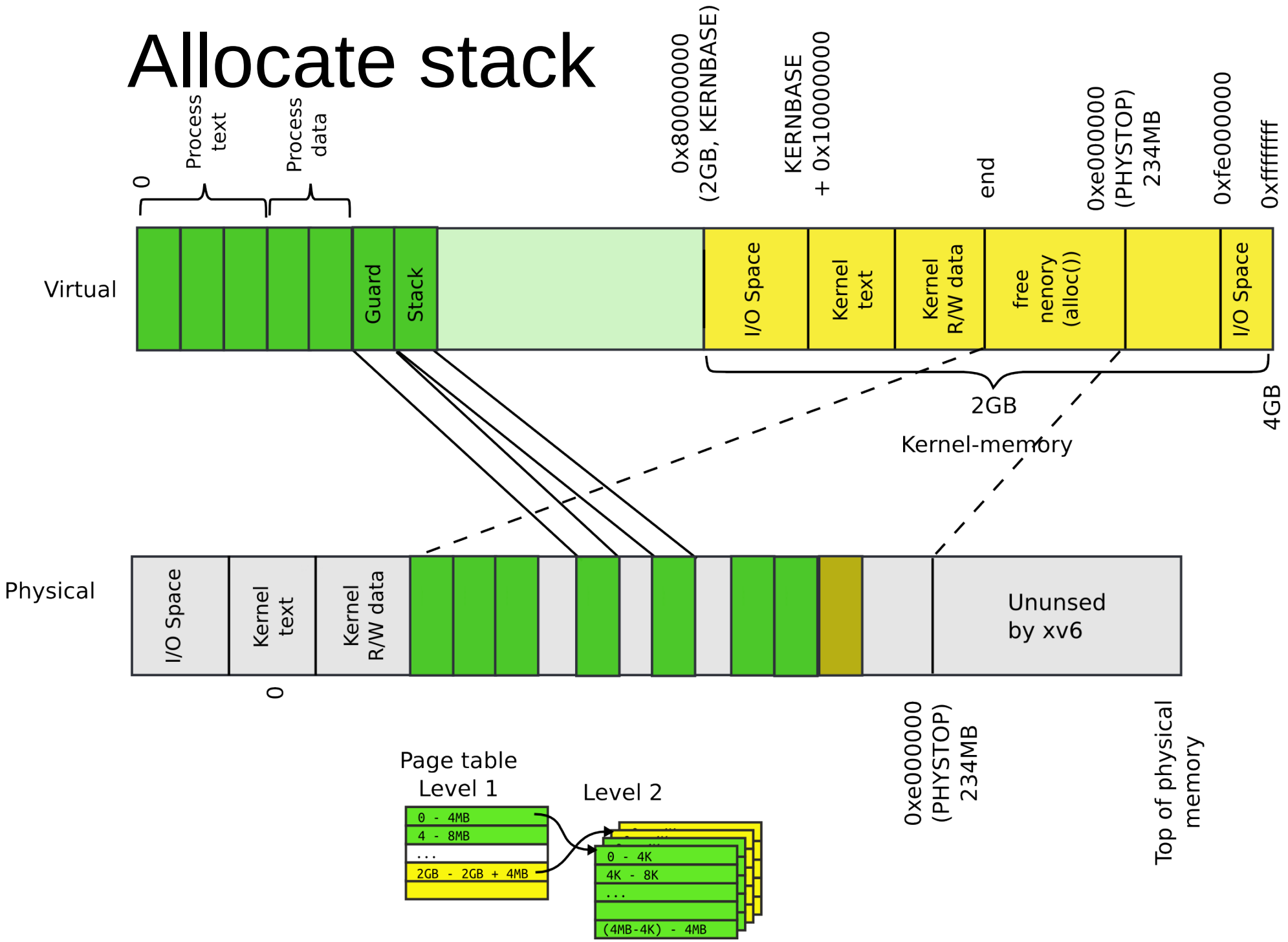
- Allocate a page

```
1918 loaduvm(pde_t *pgdir, char *addr, struct inode *ip, uint offset, uint
sz)
1919 {
...
1925   for(i = 0; i < sz; i += PGSIZE){
1926     if((pte = walkpgdir(pgdir, addr+i, 0)) == 0)
1927       panic("loaduvm: address should exist");
1928     pa = PTE_ADDR(*pte);
1929     if(sz - i < PGSIZE)
1930       n = sz - i;
1931     else
1932       n = PGSIZE;
1933     if(readi(ip, P2V(pa), offset+i, n) != n)
1934       return -1;
1935   }
1936   return 0;
1937 }
```

Load program into memory

- Read the page from disk

Allocate stack



exec(): allocate process' stack

- Allocate two pages
  - One will be stack
  - Mark another one as inaccessible

```
6361    sz = PGROUNDUP(sz);
6362    if((sz = allocuvm(pgdir, sz, sz + 2*PGSIZE)) == 0)
6363      goto bad;
6364    clearpteu(pgdir, (char*)(sz - 2*PGSIZE));
6365    sp = sz;
```

```
6367  // Push argument strings, prepare rest of stack in ustack.
6368  for(argc = 0; argv[argc]; argc++) {
...
6371    sp = (sp - (strlen(argv[argc]) + 1)) & ~3;
6372    if(copyout(pgdir, sp, argv[argc], strlen(argv[argc]) + 1) < 0)
6373      goto bad;
6374    ustack[3+argc] = sp;
6375  }
6376  ustack[3+argc] = 0;
6377
6378  ustack[0] = 0xffffffff; // fake return PC
6379  ustack[1] = argc;
6380  ustack[2] = sp - (argc+1)*4; // argv pointer
6381
6382  sp -= (3+argc+1) * 4;
6383  if(copyout(pgdir, sp, ustack, (3+argc+1)*4) < 0)
6384   goto bad;
```

Push program arguments on the stack

Remember arguments to main()?

- `int main(int argc, char **argv);`
- If you run
  - `./program hello world`
- Then:
  - argc would be 3.
  - argv[0] would be "./program".
  - argv[1] would be "hello".
  - argv[2] would be "world".

Arguments to main() are passed on the stack

- Copy argument strings at the top of the stack
  - One at a time
- Record pointers to them in ustack
  - Which will be an argument list (argv list)

Switch page tables

- Switch page tables
- Deallocate old page table

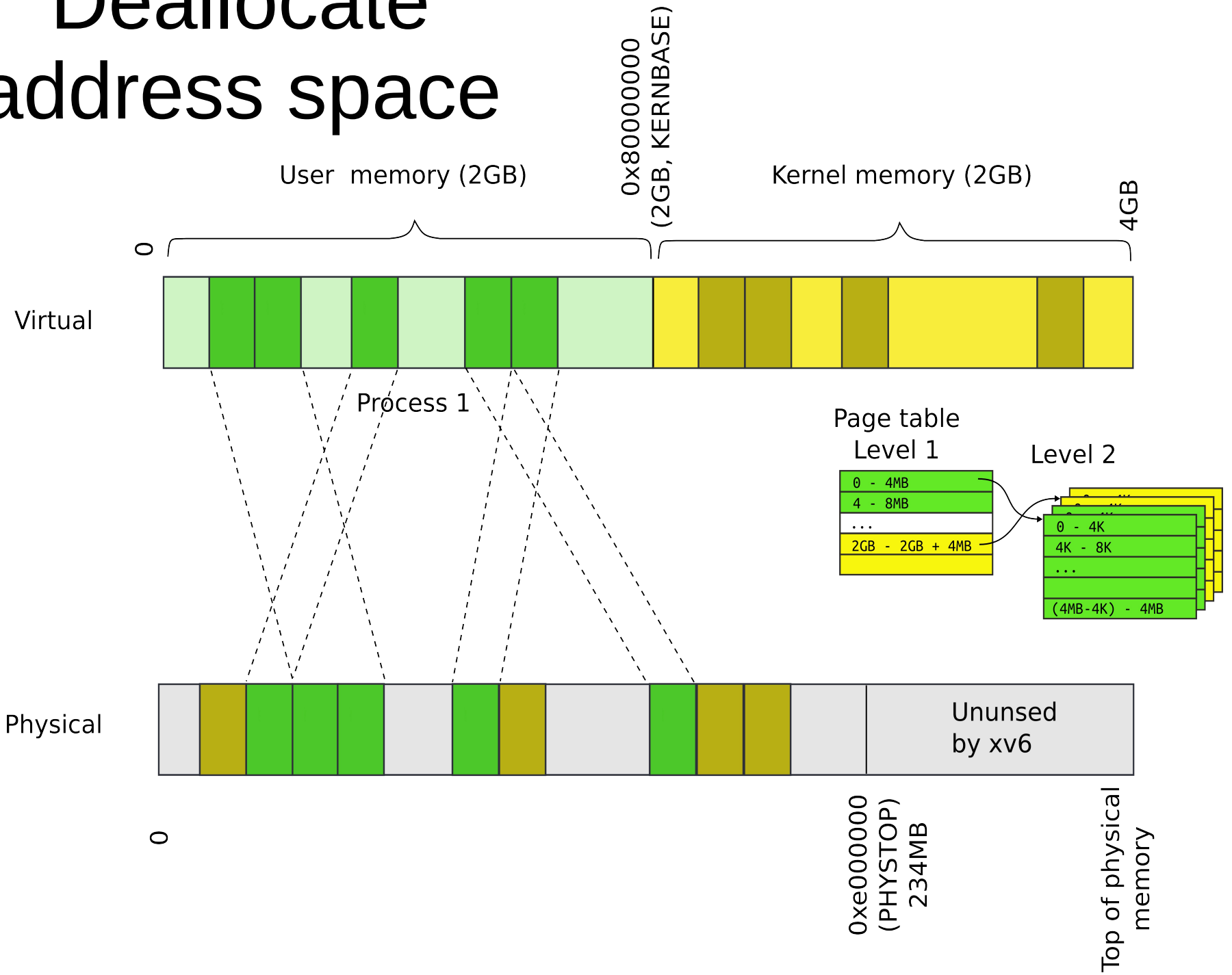
```
6398    switchuvm(proc);
6399    freevm(oldpgdir);
6400    return 0;
```

Wait... which page table we are deallocating?

Wait... which page table we are deallocating?

- Remember exec() replaces content of an already existing process
  - That process had a page table
  - We have to deallocate it

Deallocate address space



Outline: deallocate process address space

- Walk the page table
  - Deallocate all pages mapped by the page table
- Deallocate pages that contain Level 2 of the page-table
- Deallocate page directory

```
2015 freevm(pde_t *pgdir)
2016 {
2017   uint i;
2018
2019   if(pgdir == 0)
2020     panic("freevm: no pgdir");
2021   deallocuvm(pgdir, KERNBASE, 0);
2022   for(i = 0; i < NPDENTRIES; i++){
2023     if(pgdir[i] & PTE_P){
2024       char * v = P2V(PTE_ADDR(pgdir[i]));
2025       kfree(v);
2026     }
2027   }
2028   kfree((char*)pgdir);
2029 }
```

Deallocate user address space

```
1987 deallocuvm(pde_t *pgdir, uint oldsz, uint newsz)
1988 {
...
1995   a = PGROUNDUP(newsz);
1996   for(; a < oldsz; a += PGSIZE){
1997     pte = walkpgdir(pgdir, (char*)a, 0);
1998     if(!pte)
1999       a += (NPTENTRIES - 1) * PGSIZE;
2000     else if((*pte & PTE_P) != 0){
2001       pa = PTE_ADDR(*pte);
2002       if(pa == 0)
2003         panic("kfree");
2004       char *v = P2V(pa);
2005       kfree(v);
2006       *pte = 0;
2007     }
2008   }
2009   return newsz;
2010 }
```

Walk page table and get pte

```
1987 deallocuvm(pde_t *pgdir, uint oldsz, uint newsz)
1988 {
...
1995   a = PGROUNDUP(newsz);
1996   for(; a < oldsz; a += PGSIZE){
1997     pte = walkpgdir(pgdir, (char*)a, 0);
1998     if(!pte)
1999       a += (NPTENTRIES - 1) * PGSIZE;
2000     else if((*pte & PTE_P) != 0){
2001       pa = PTE_ADDR(*pte);
2002       if(pa == 0)
2003         panic("kfree");
2004       char *v = P2V(pa);
2005       kfree(v);
2006       *pte = 0;
2007     }
2008   }
2009   return newsz;
2010 }
```

Deallocate a page

Deallocate Level 2

```
2015 freevm(pde_t *pgdir)
2016 {
2017   uint i;
2018
2019   if(pgdir == 0)
2020     panic("freevm: no pgdir");
2021   deallocuvm(pgdir, KERNBASE, 0);
2022   for(i = 0; i < NPDENTRIES; i++){
2023     if(pgdir[i] & PTE_P){
2024       char * v = P2V(PTE_ADDR(pgdir[i]));
2025       kfree(v);
2026     }
2027   }
2028   kfree((char*)pgdir);
2029 }
```

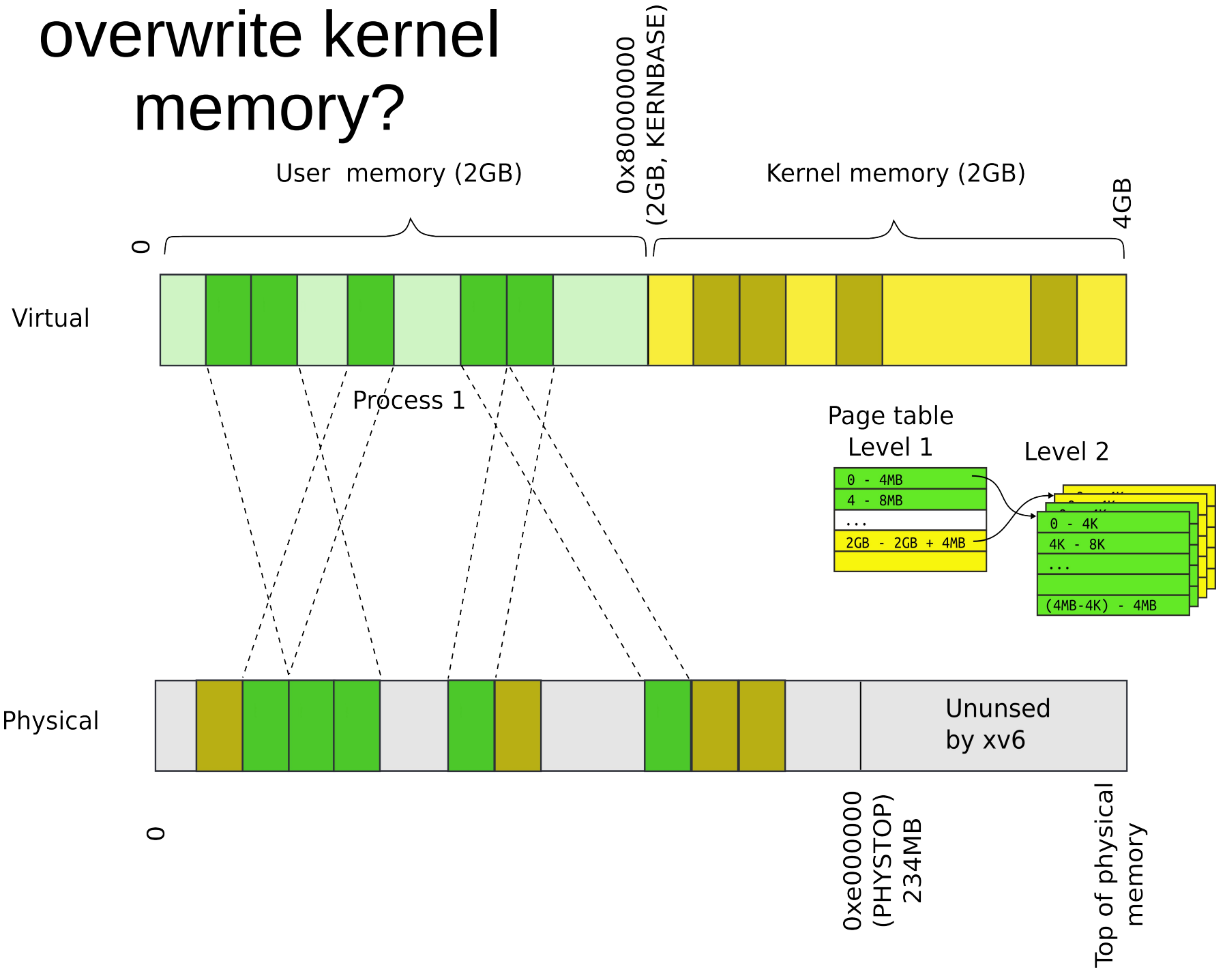
Recap

- We know how exec works!
- We can create new processes

Thank you!

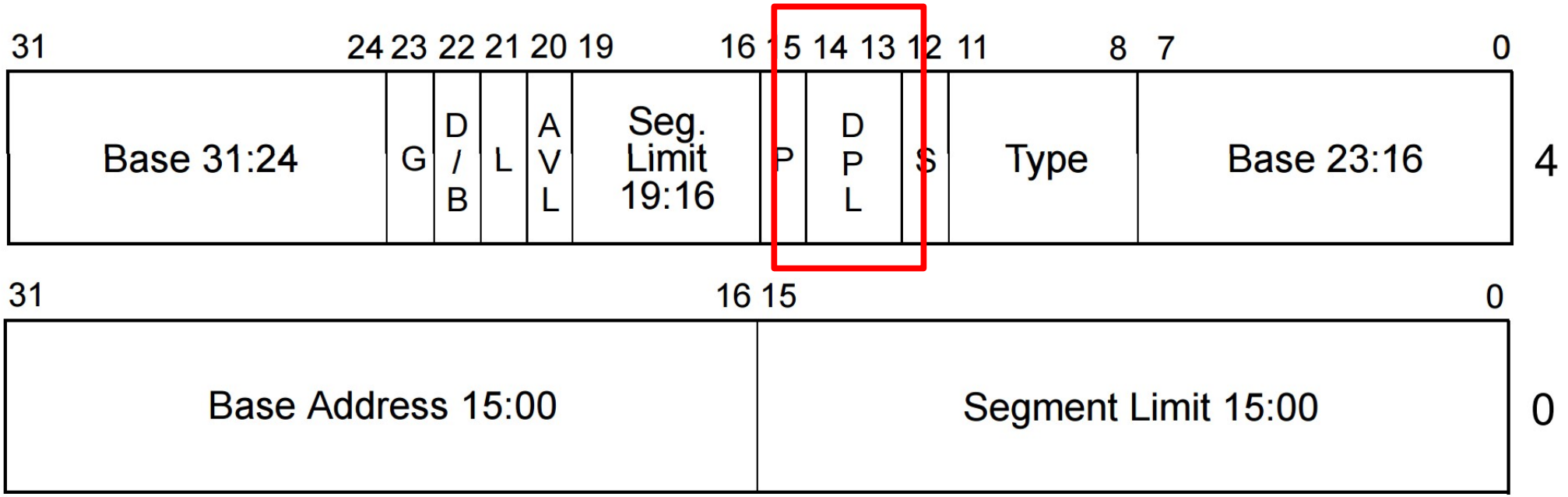
Backup

Recap: Can a process overwrite kernel memory?



Privilege levels

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

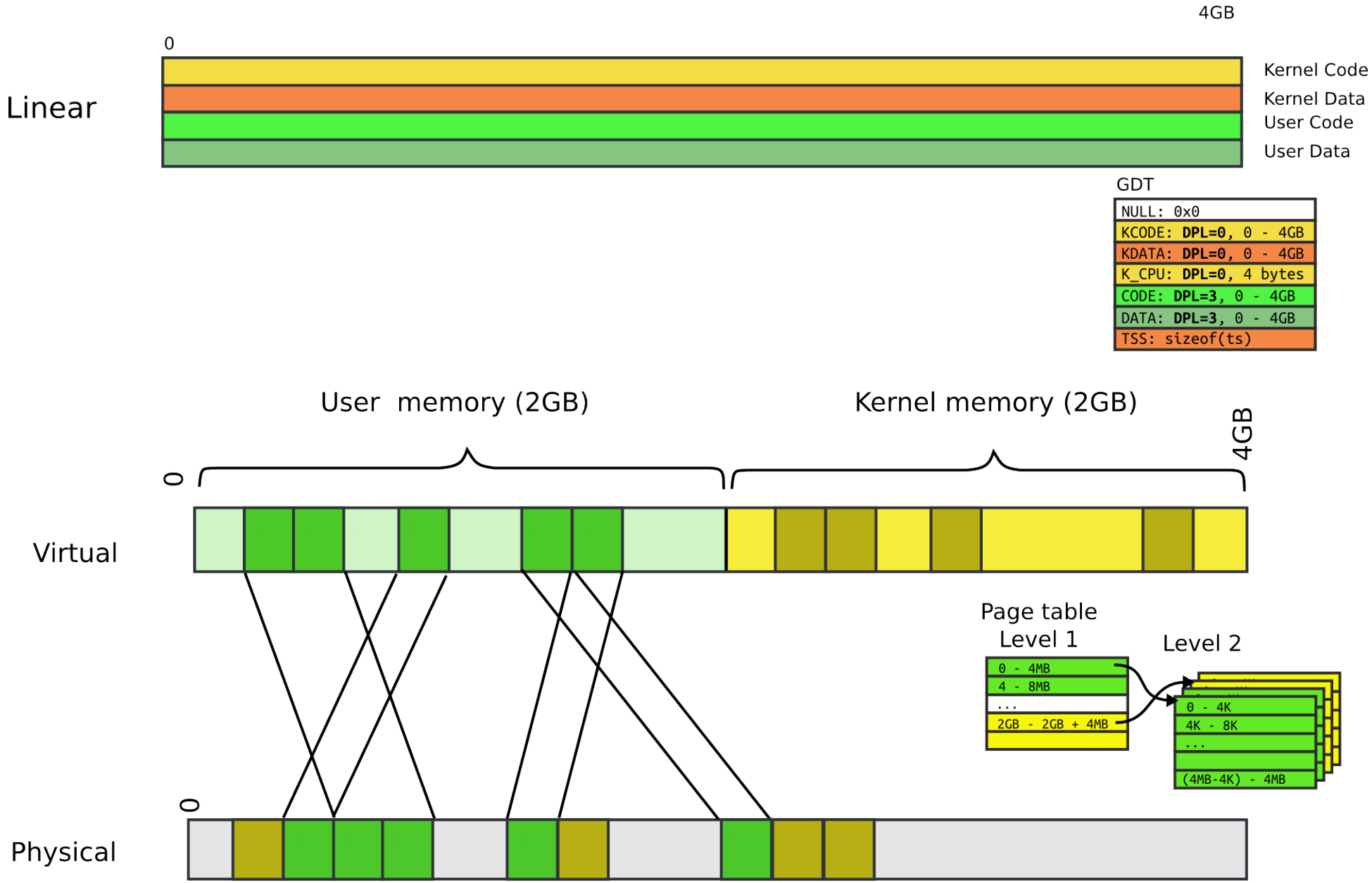


Privilege levels

- 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

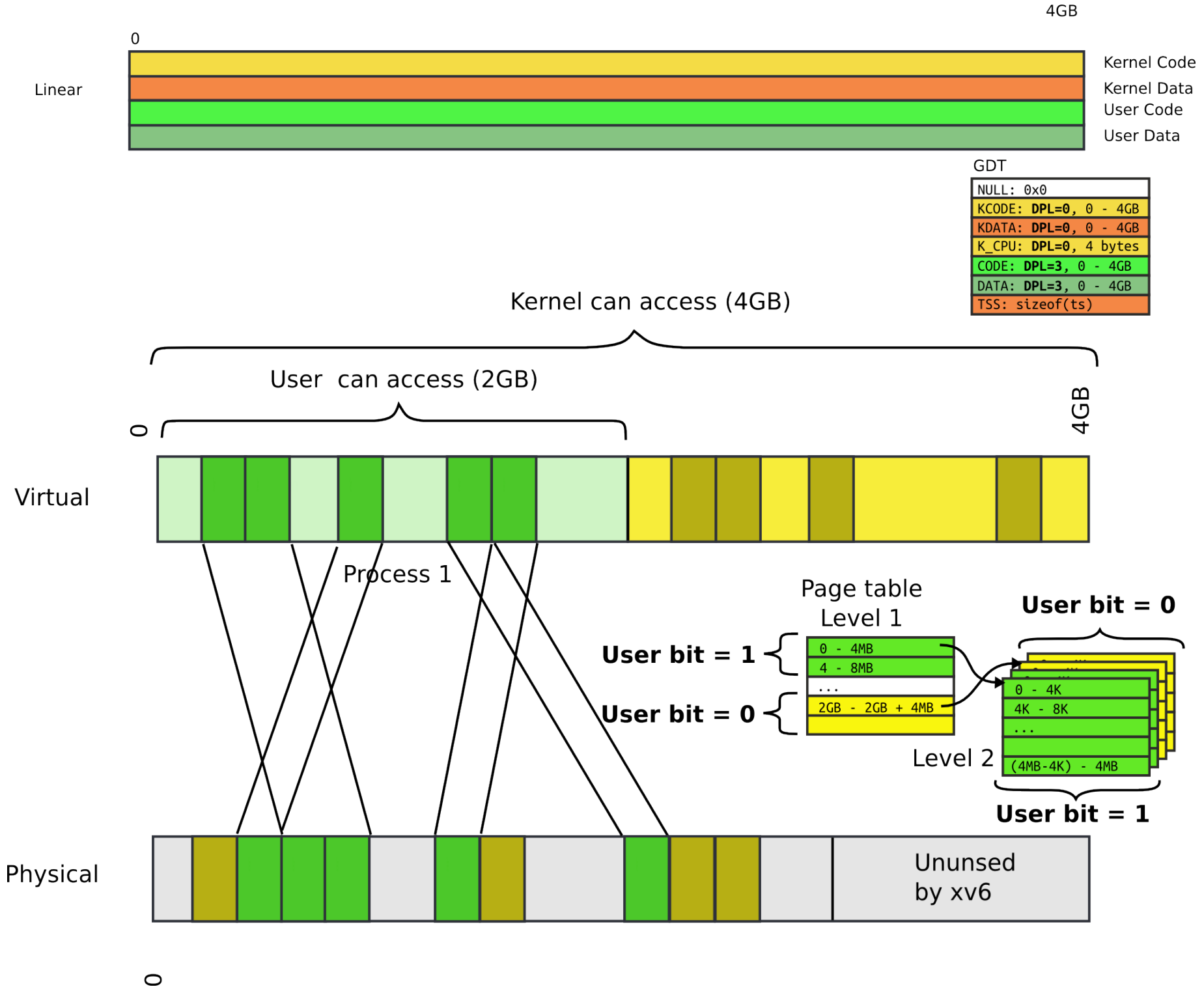
Real world

- 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

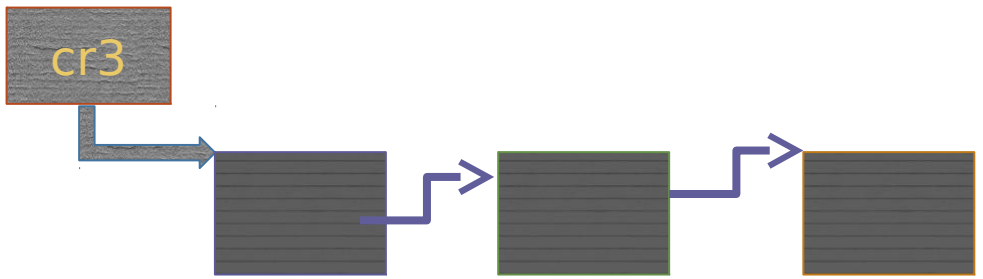
Page table: user bit

- 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

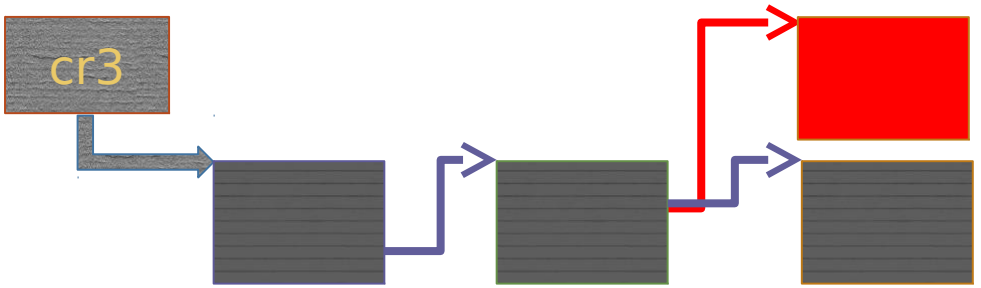
- CPU caches results of page table walks
  - In translation lookaside buffer (TLB)
- Walking page table is slow
  - Each memory access is 200-300 cycles on modern hardware
  - L3 cache access is 70 cycles



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)