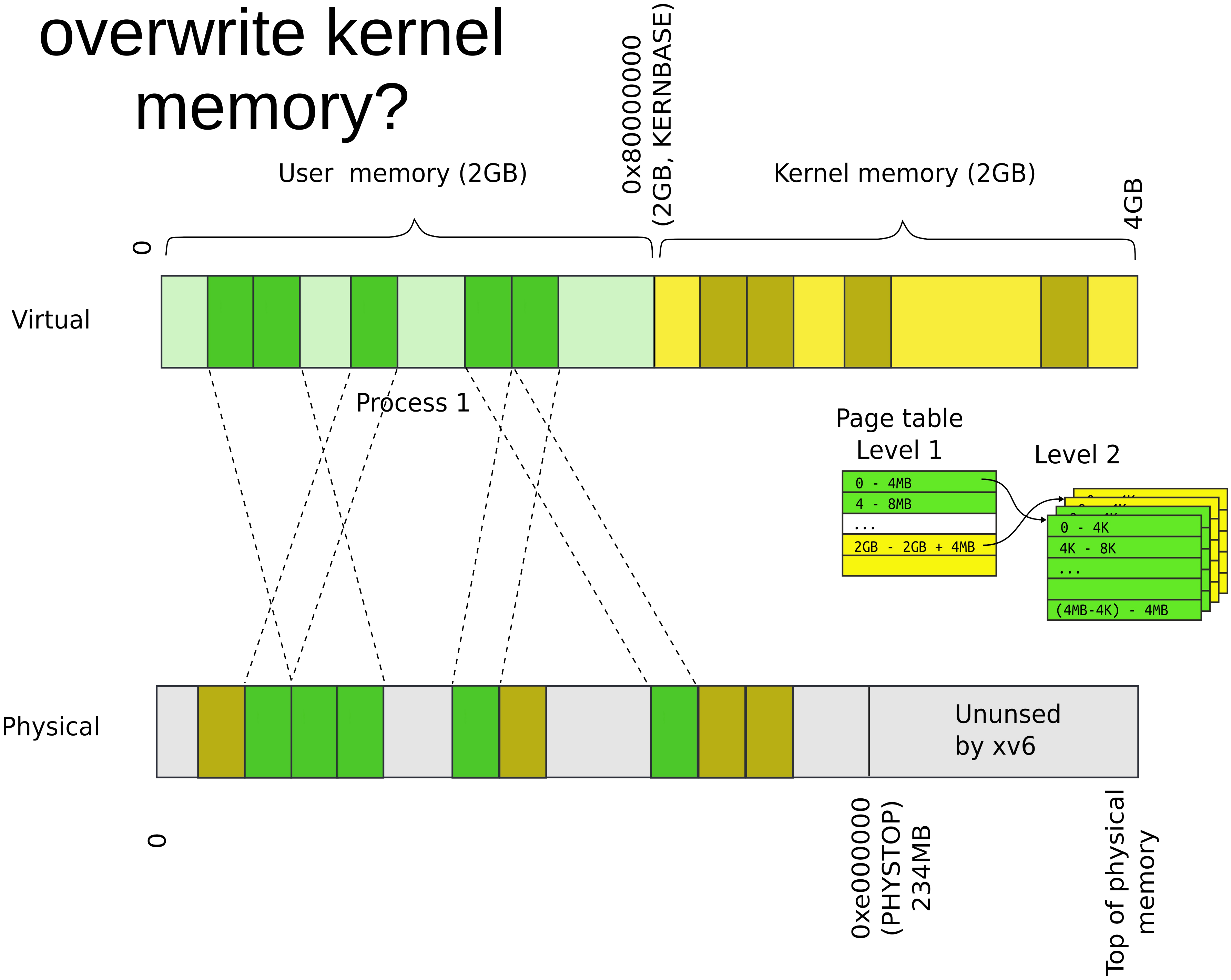
143A: Principles of Operating Systems

Lecture 09: First process

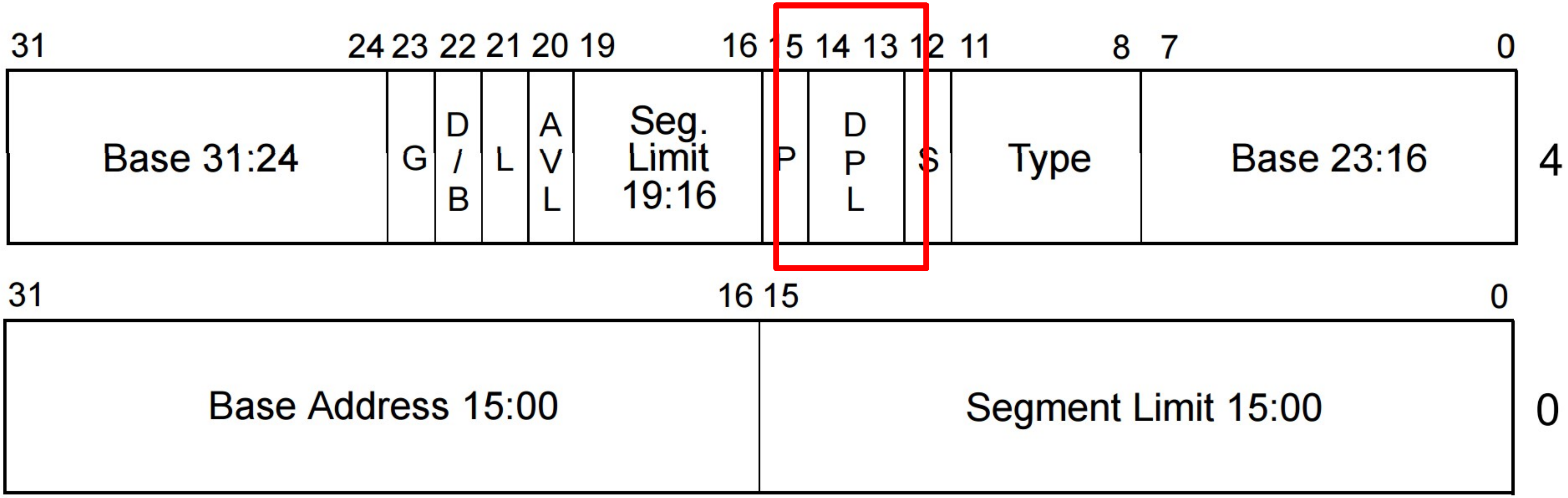
Anton Burtsev
January, 2017

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



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

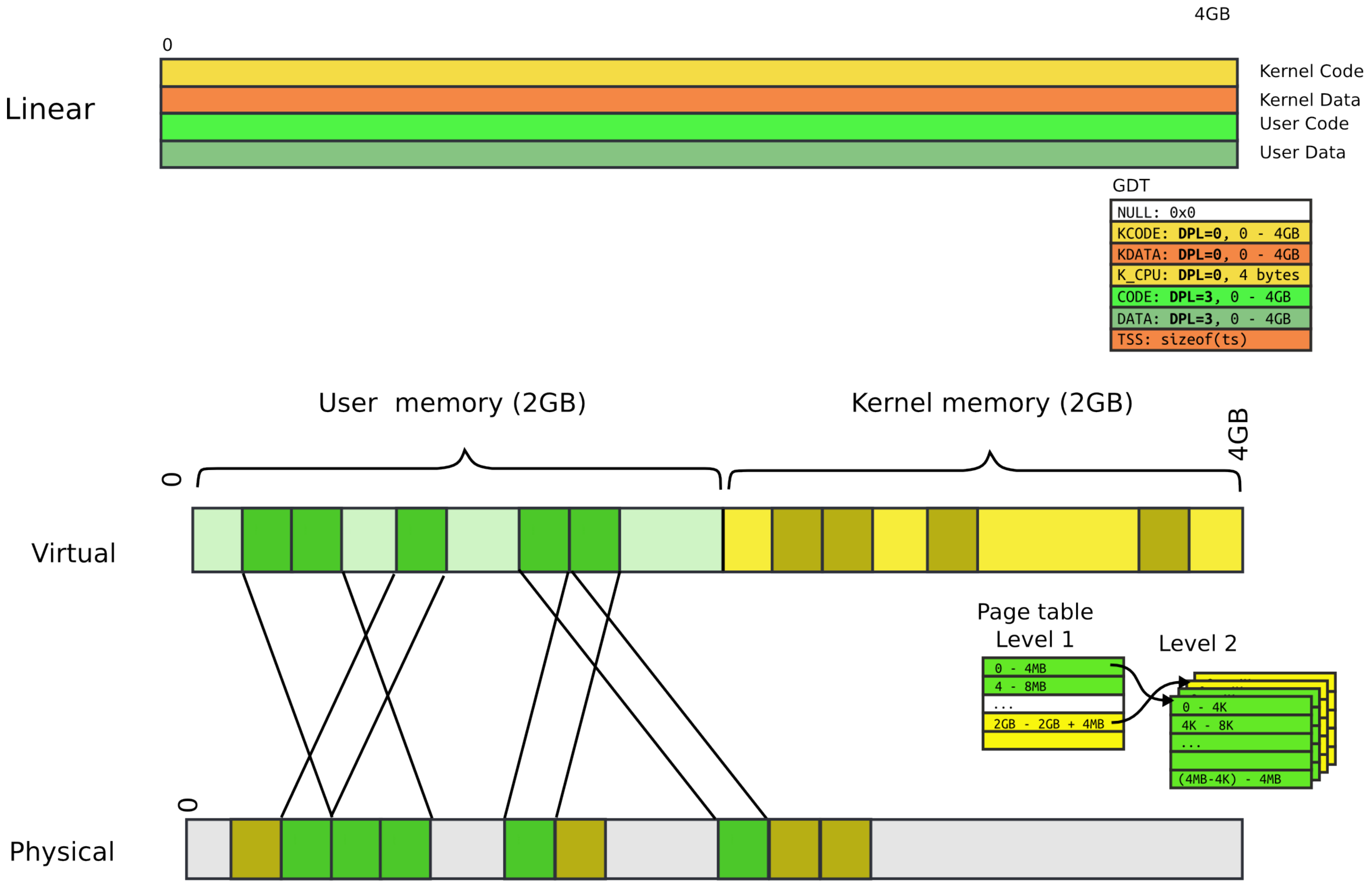
31 16	15 0	
Base Address 15:00	Segment Limit 15:00	0

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

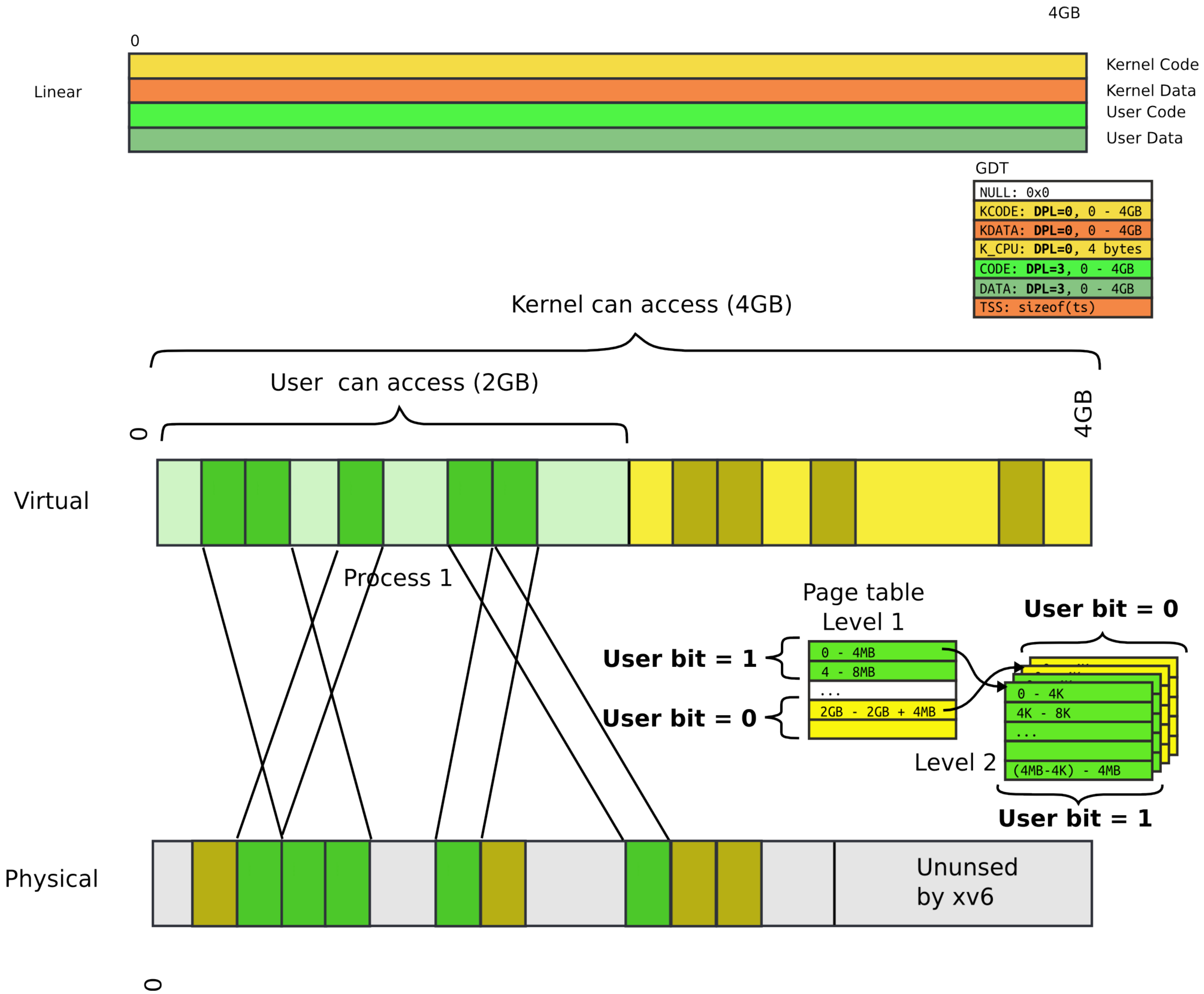


GDT

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)

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

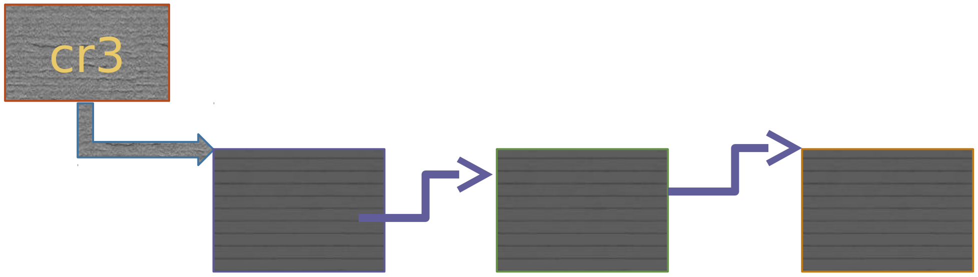


GDT

```
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)
```

TLB

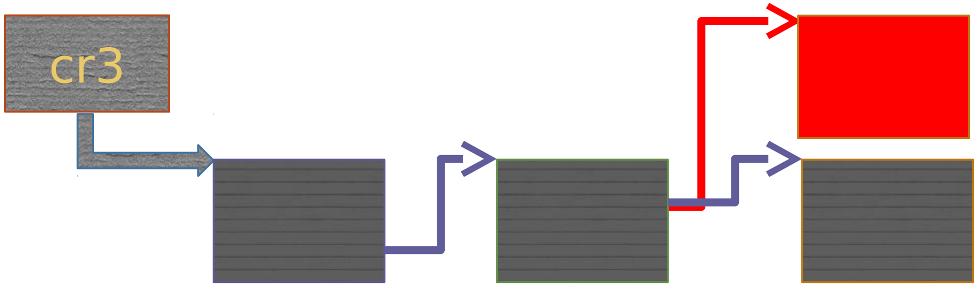
- CPU caches results of page table walks
  - In translation lookaside buffer (TLB)
- Walking page table is slow
  - Each memory access is 200-300 cycles on modern hardware
  - L3 cache access is 70 cycles



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.

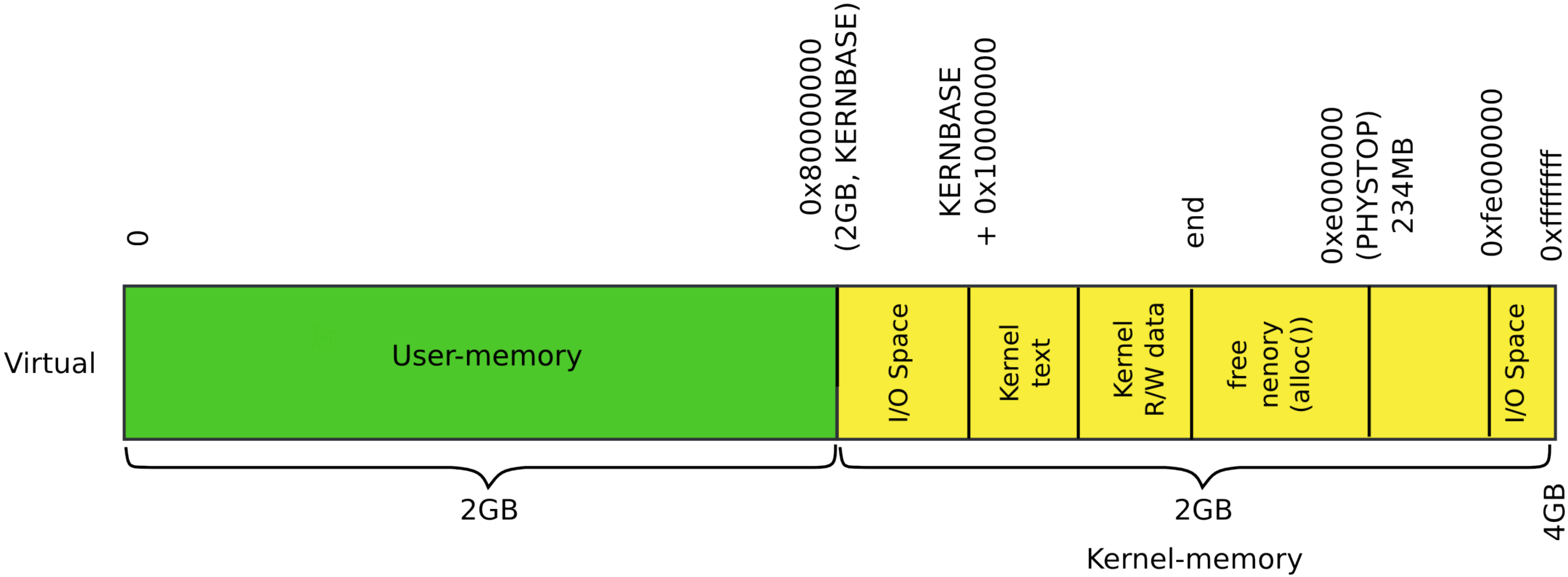
No
Change!!!

Invalidating TLB

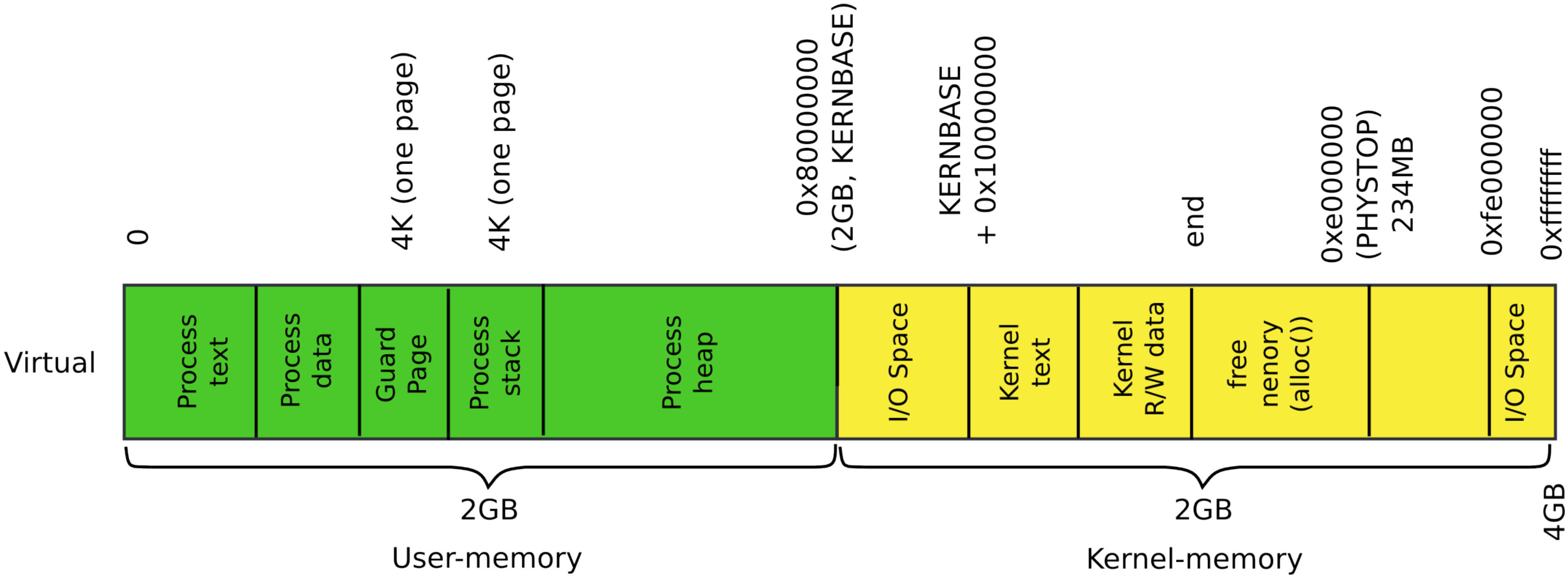
- After every page table update, OS needs to manually invalidate cached values
- Modern CPUs have “tagged TLBs”,
  - Each TLB entry has a “tag” – identifier of a process
  - No need to flush TLBs on context switch
- On Intel this mechanism is called
  - Process-Context Identifiers (PCIDs)

Creating Processes

Recap: kernel memory layout



Process memory layout



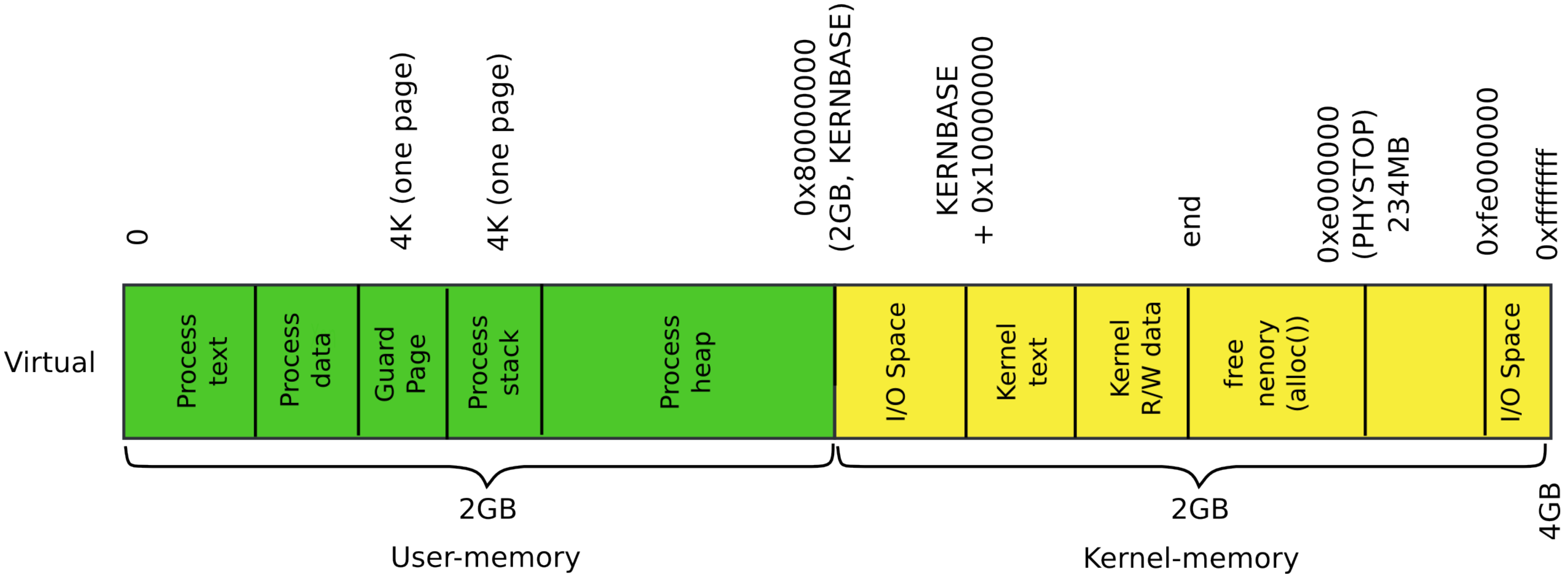
How does kernel creates new processes?

How does kernel creates new processes?

- Exec
  - exec(“/bin/ls”, argv);

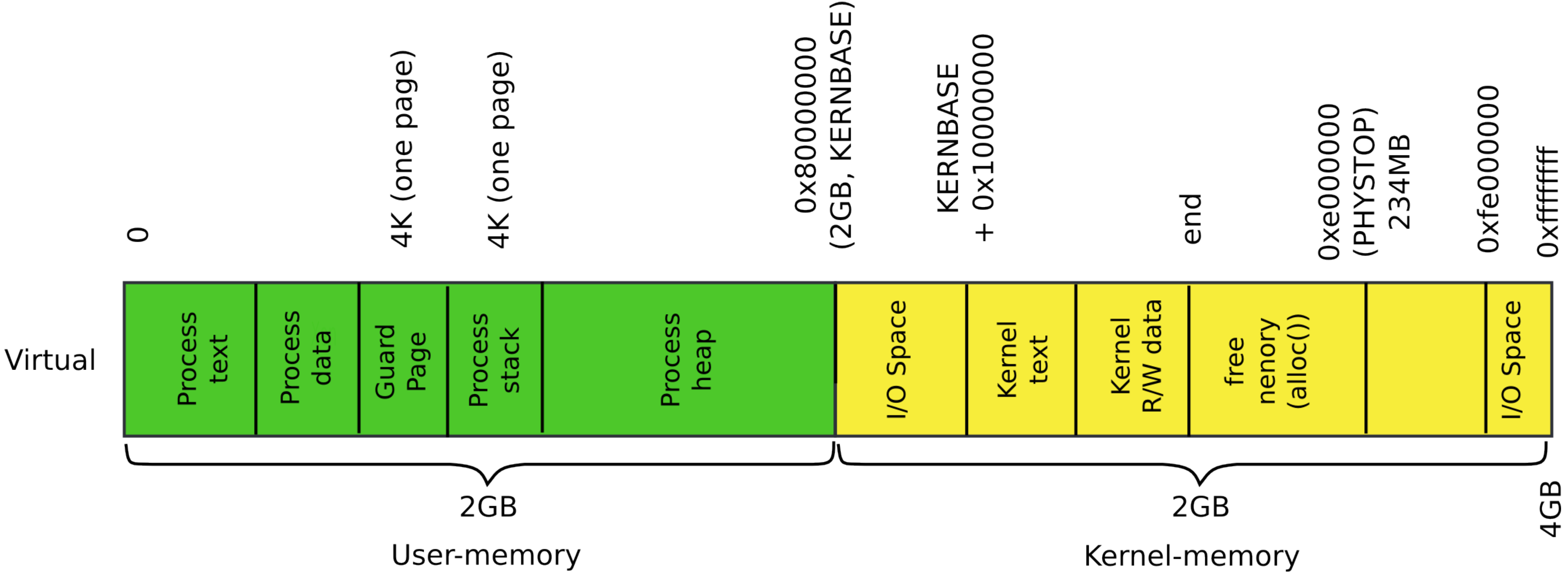
exec(): high-level outline

- 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

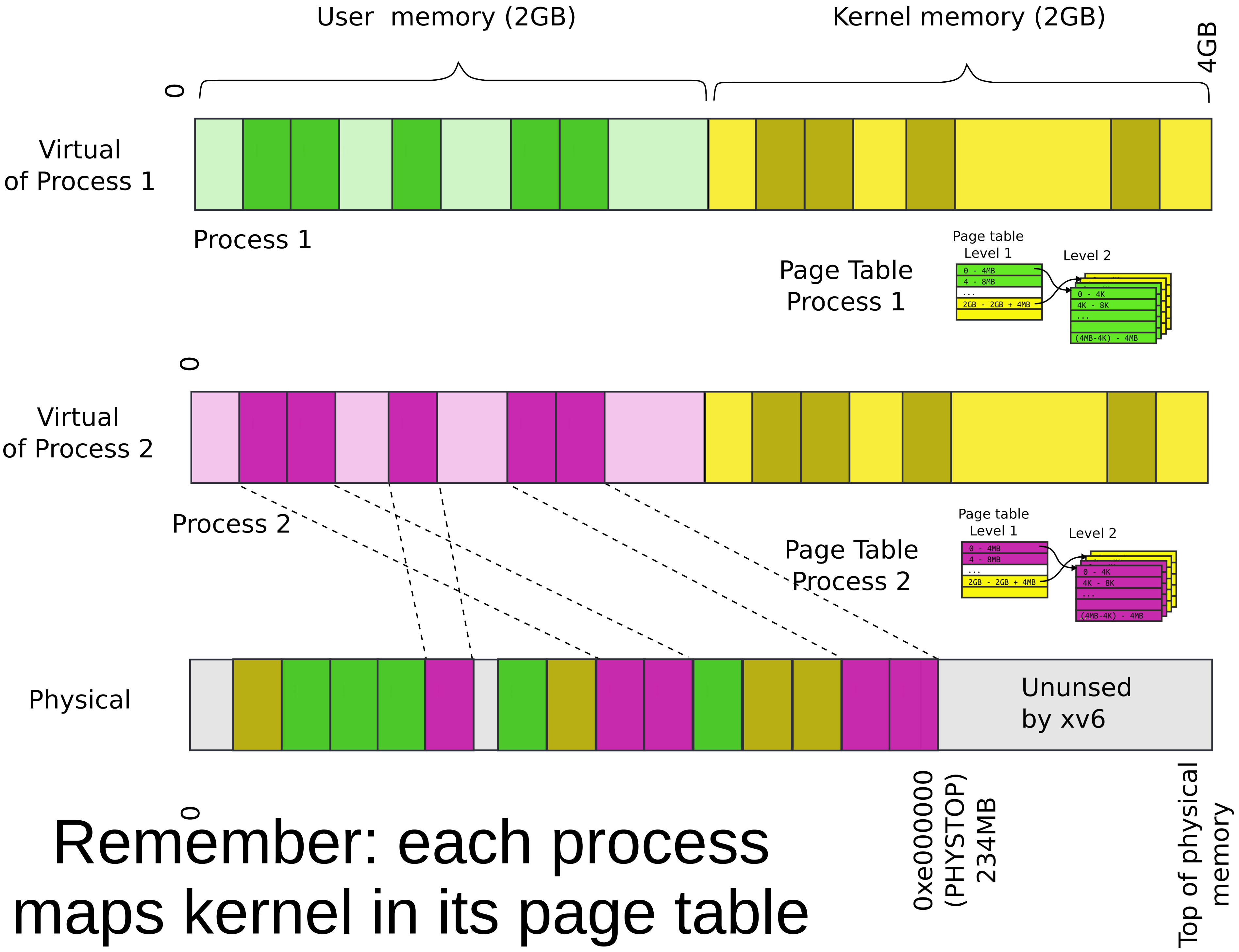
```
int
exec(char *path, char **argv)
{
...
  if((ip = namei(path)) == 0){
    end_op();
    return -1;
  }
  // Check ELF header
  if(readi(ip, (char*)&elf, 0, sizeof(elf)) <
                                      sizeof(elf))
    goto bad;
  if(elf.magic != ELF_MAGIC)
    goto bad;
```

exec(): check ELF header

```
int
exec(char *path, char **argv)
{
...
  if((ip = namei(path)) == 0){
    end_op();
    return -1;
  }
  // Check ELF header
  if(readi(ip, (char*)&elf, 0, sizeof(elf)) <
                                      sizeof(elf))
    goto bad;
  if(elf.magic != ELF_MAGIC)
    goto bad;
```

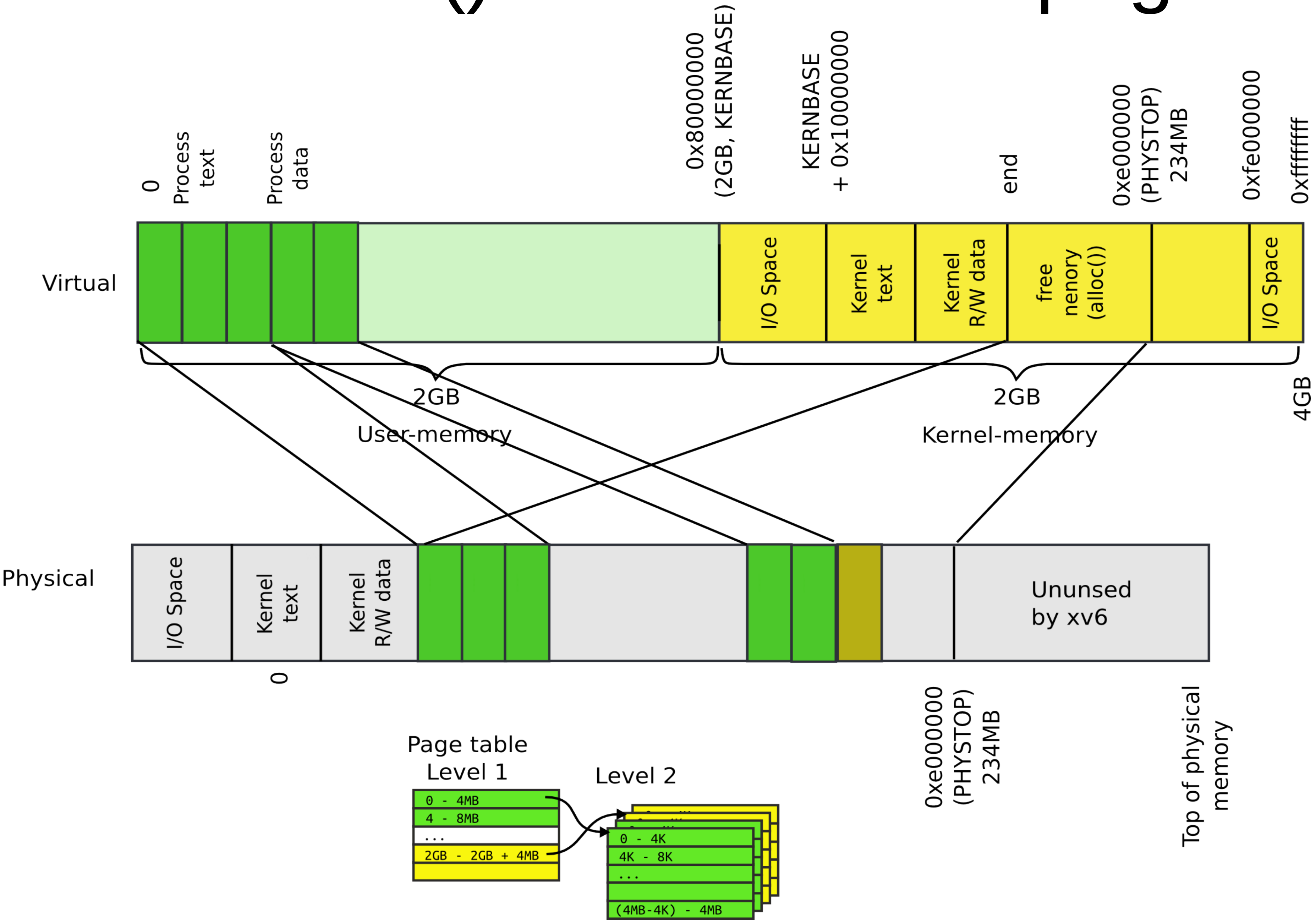
Setup kernel address space()

```
  if((pgdir = setupkvm()) == 0)
    goto bad;
```

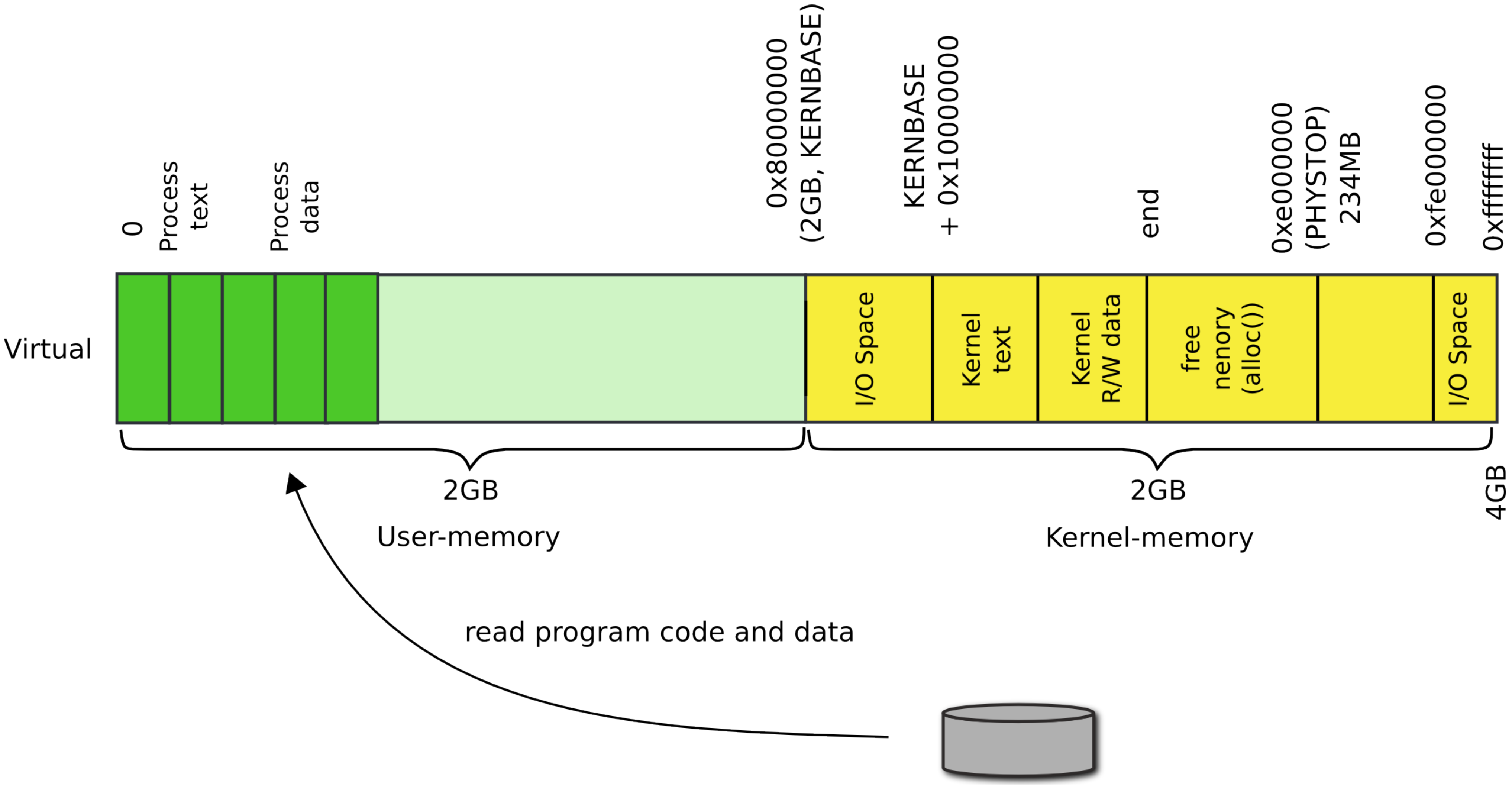


Remember: each process maps kernel in its page table

allocuvm(): allocate user pages



loaduvm(): read program from disk

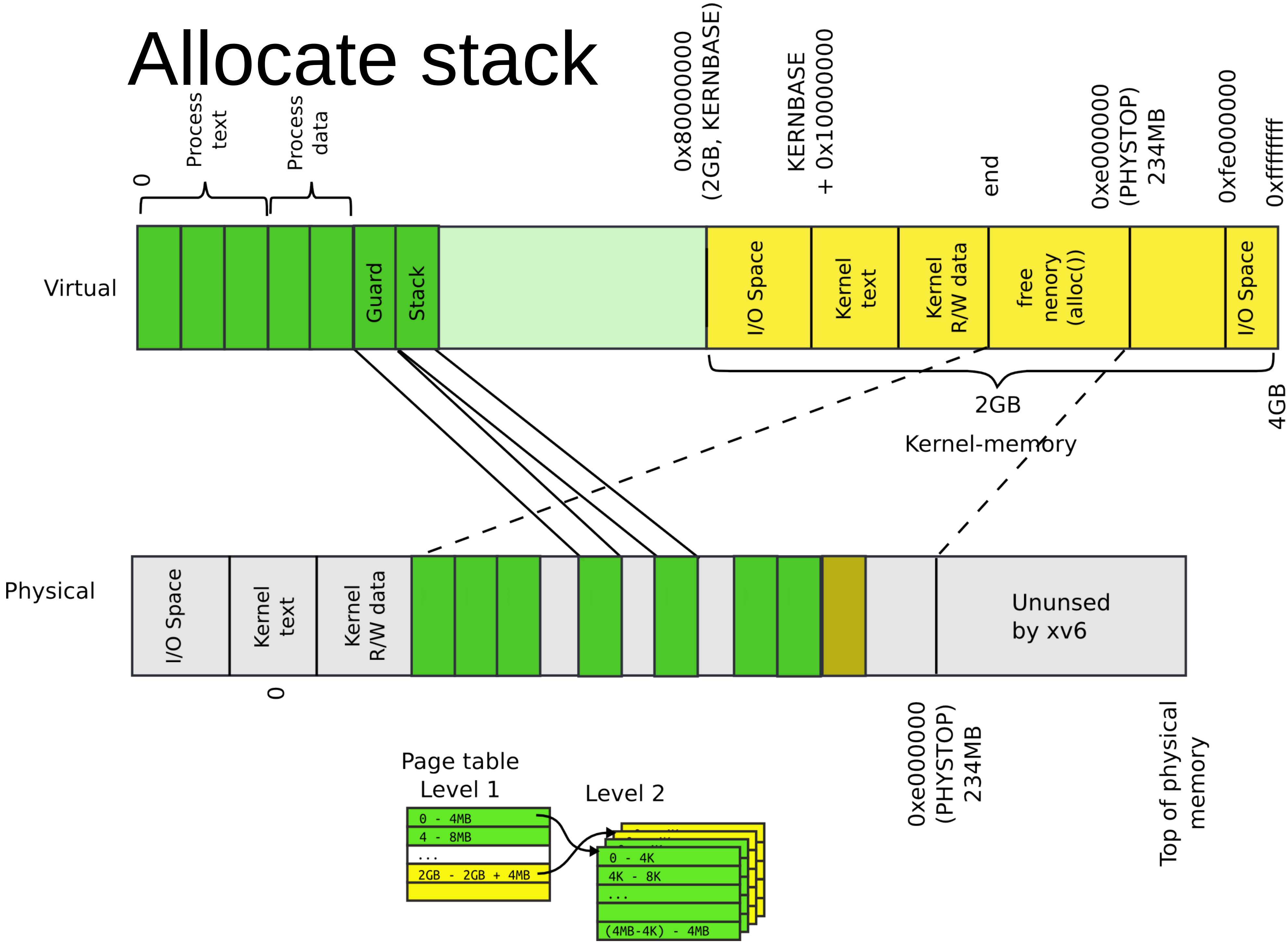


exec(): allocate process' stack

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

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

Allocate stack



Switch page tables

- Switch page tables
- Deallocate old page table

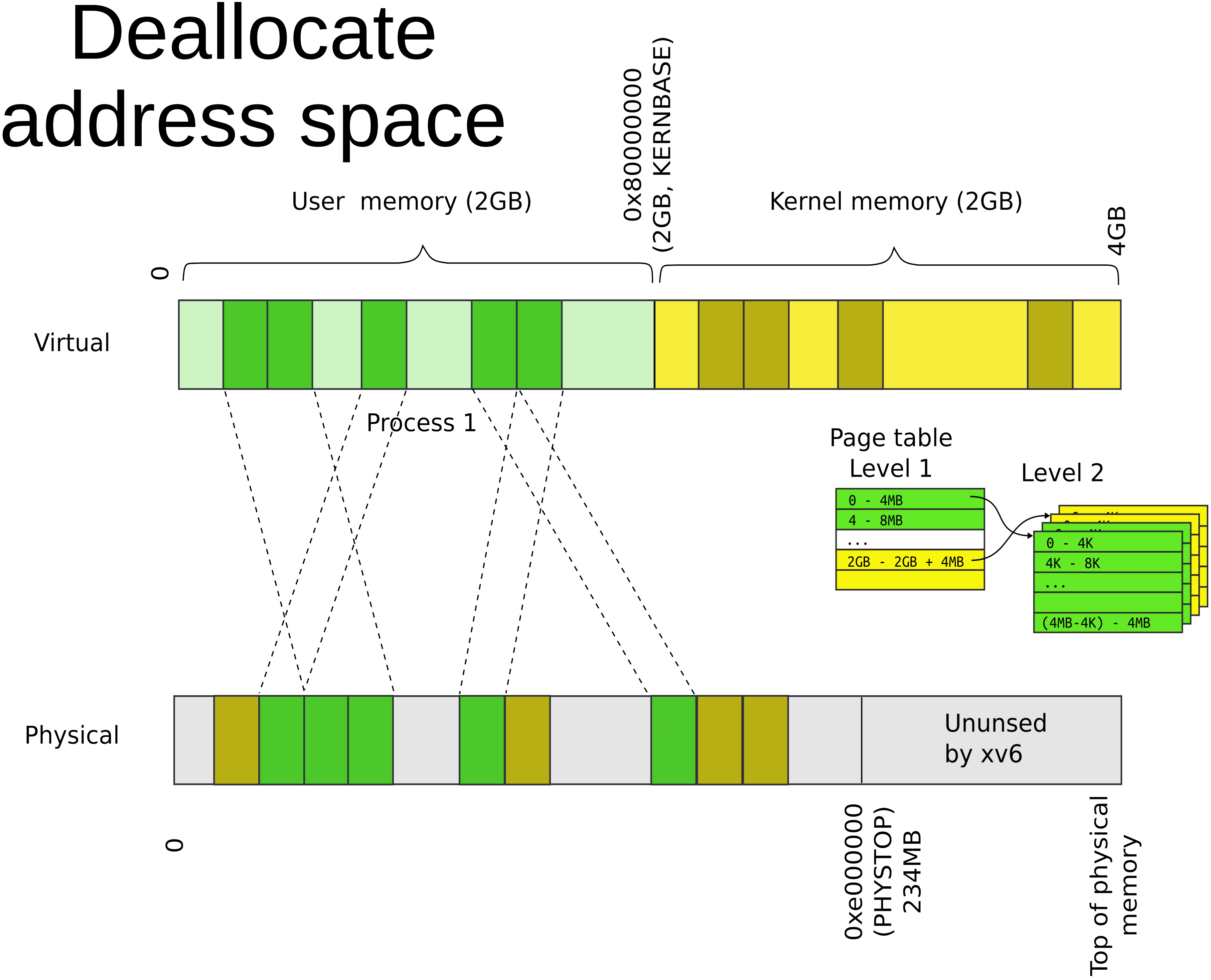
```
switchuvm(proc);
freevm(oldpgdir);
return 0;
```

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

```
freevm(pde_t *pgdir)
{
  uint i;

  if(pgdir == 0)
    panic("freevm: no pgdir");
  deallocuvm(pgdir, KERNBASE, 0);
  for(i = 0; i < NPDENTRIES; i++){
    if(pgdir[i] & PTE_P){
      char * v = P2V(PTE_ADDR(pgdir[i]));
      kfree(v);
    }
  }
  kfree((char*)pgdir);
}
```

Deallocate user address space

```
deallocuvm(pde_t *pgdir, uint oldsz, uint newsz)
{
...
  a = PGROUNDUP(newsz);
  for(; a < oldsz; a += PGSIZE){
    pte = walkpgdir(pgdir, (char*)a, 0);
    if(!pte)
      a += (NPTENTRIES - 1) * PGSIZE;
    else if((*pte & PTE_P) != 0){
      pa = PTE_ADDR(*pte);
      if(pa == 0)
        panic("kfree");
      char *v = P2V(pa);
      kfree(v);
      *pte = 0;
    }
  }
  return newsz;
}
```

Walk page table and get pte

Deallocate a page

```c
deallocuvm(pde_t *pgdir, uint oldsz, uint newsz)
{
...
  a = PGROUNDUP(newsz);
  for(; a < oldsz; a += PGSIZE){
    pte = walkpgdir(pgdir, (char*)a, 0);
    if(!pte)
      a += (NPTENTRIES - 1) * PGSIZE;
    else if((*pte & PTE_P) != 0){
      pa = PTE_ADDR(*pte);
      if(pa == 0)
        panic("kfree");
      char *v = P2V(pa);
      kfree(v);
      *pte = 0;
    }
  }
  return newsz;
}
```

Deallocate Level 2

```
freevm(pde_t *pgdir)
{
  uint i;

  if(pgdir == 0)
    panic("freevm: no pgdir");
  deallocuvm(pgdir, KERNBASE, 0);
  for(i = 0; i < NPDENTRIES; i++){
    if(pgdir[i] & PTE_P){
      char * v = P2V(PTE_ADDR(pgdir[i]));
      kfree(v);
    }
  }
  kfree((char*)pgdir);
}
```

Recap

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

Questions?