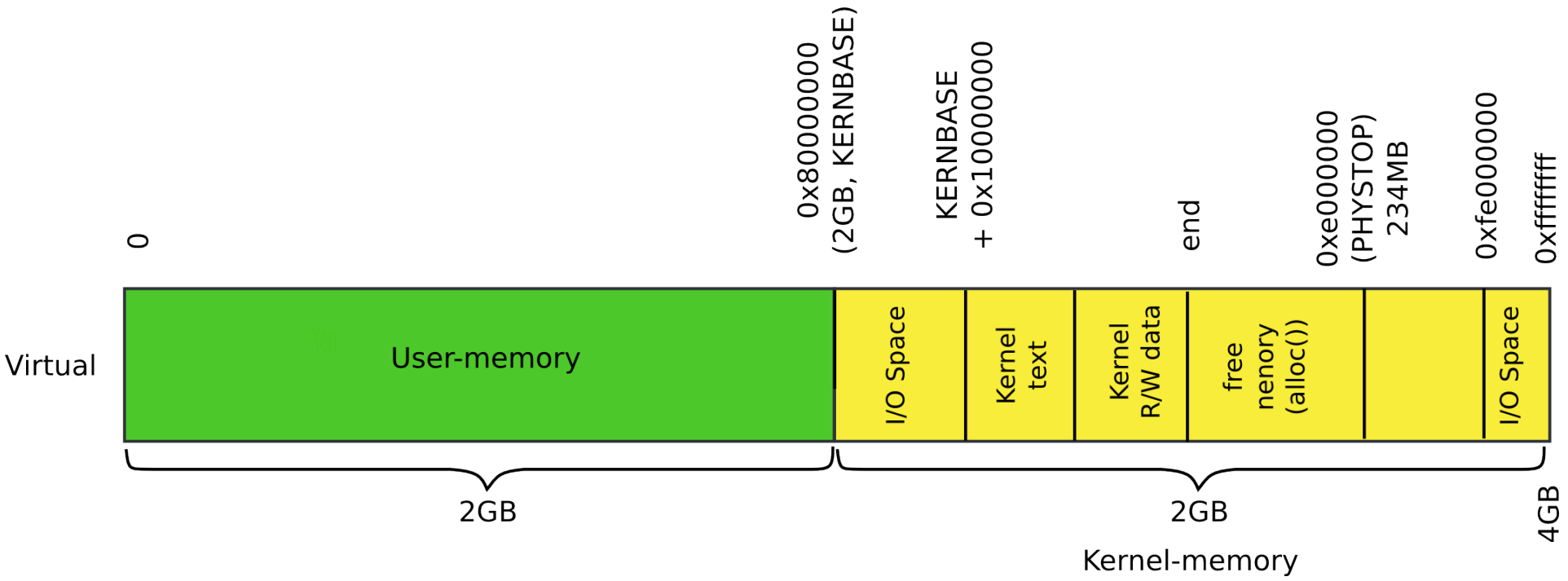
238P: Operating Systems

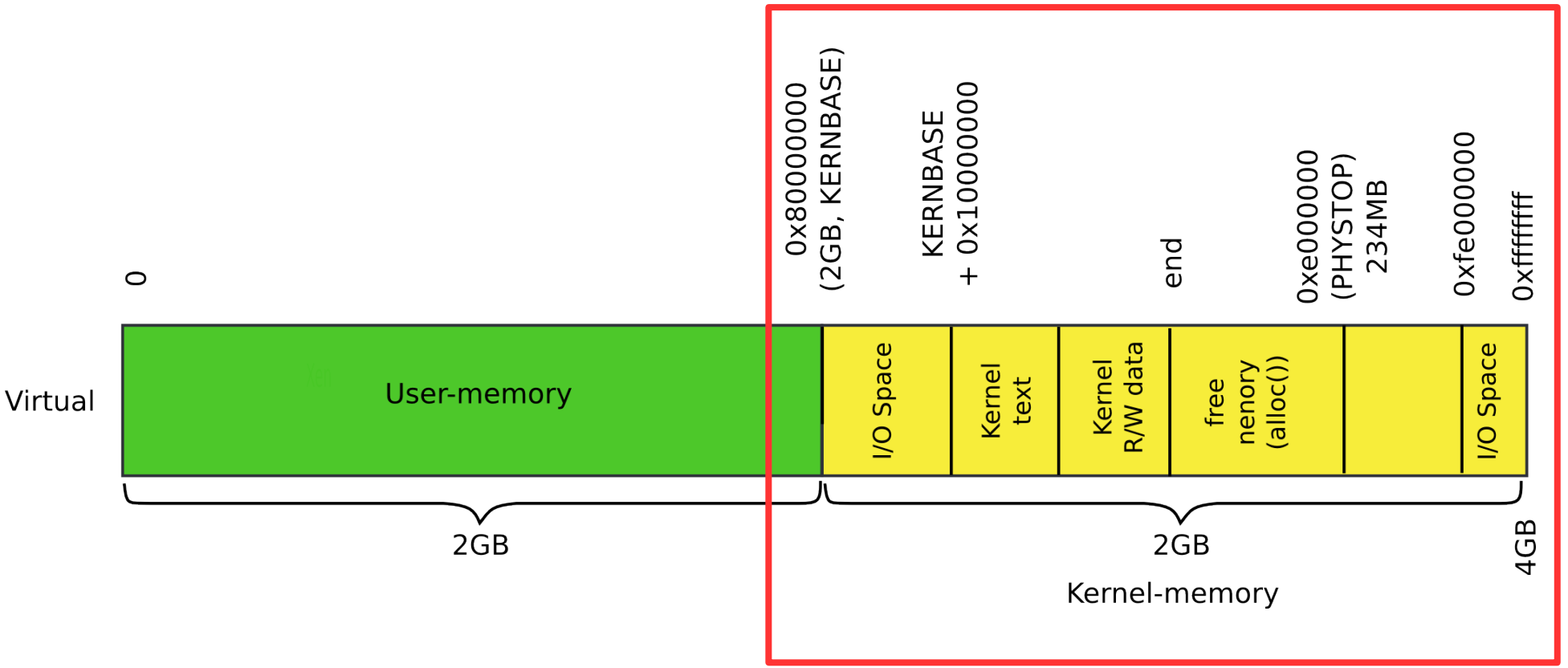
Lecture 8: Creating Processes (`exec()`)

Anton Burtsev
April, 2019

Recap: kernel memory



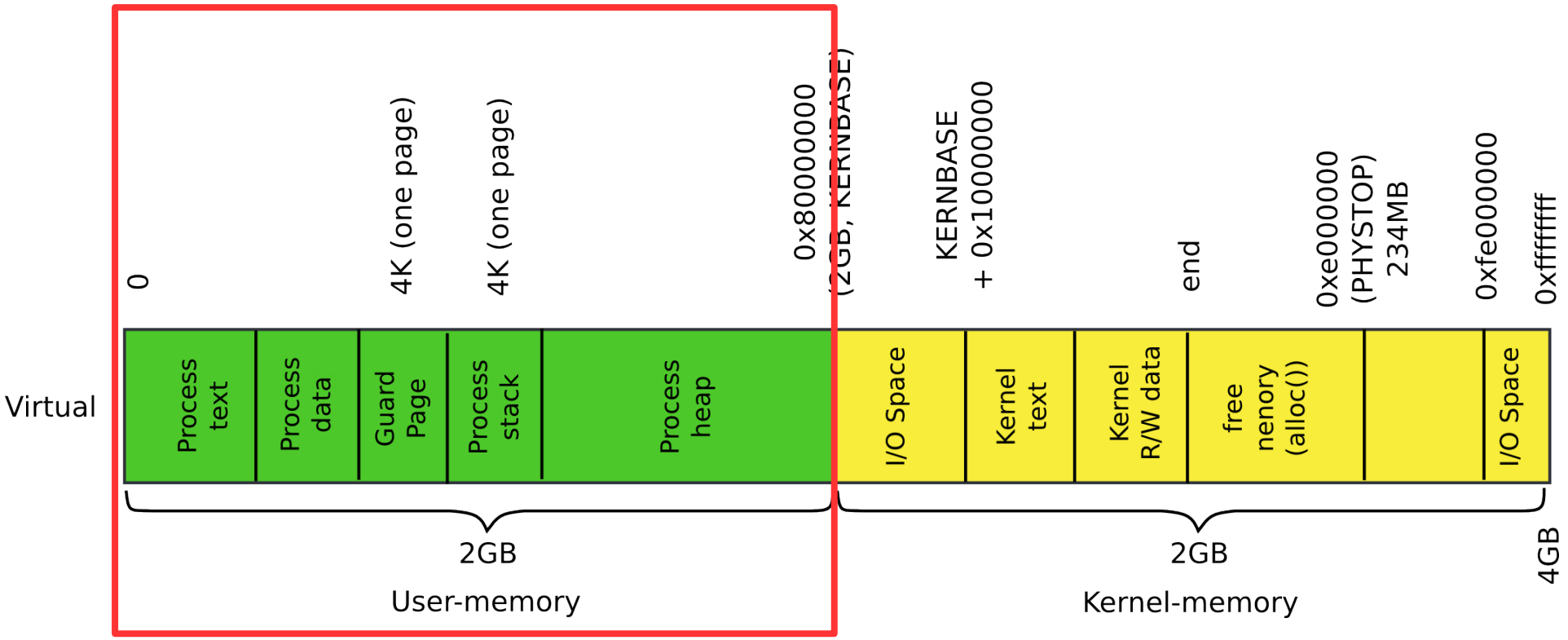
Recap: kernel memory



```
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()

Today: process memory



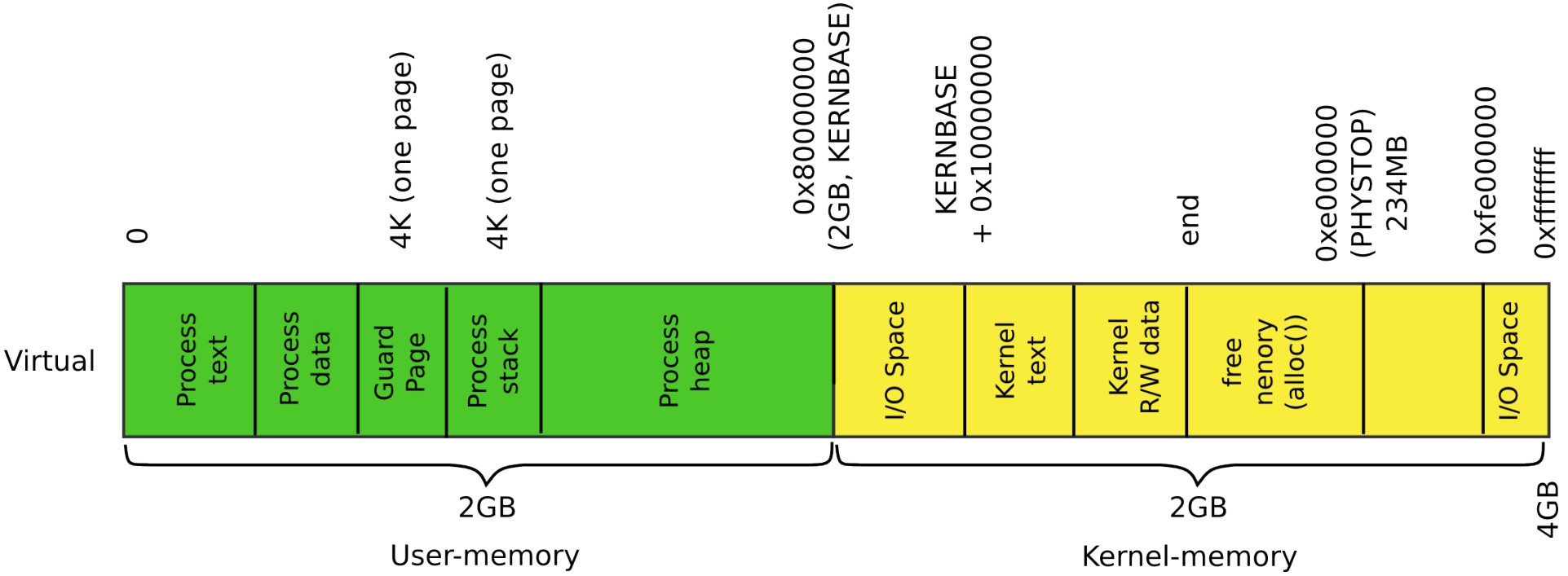
How does kernel creates new processes?

How does kernel creates new processes?

- Exec
  - exec(“/bin/ls”, argv);

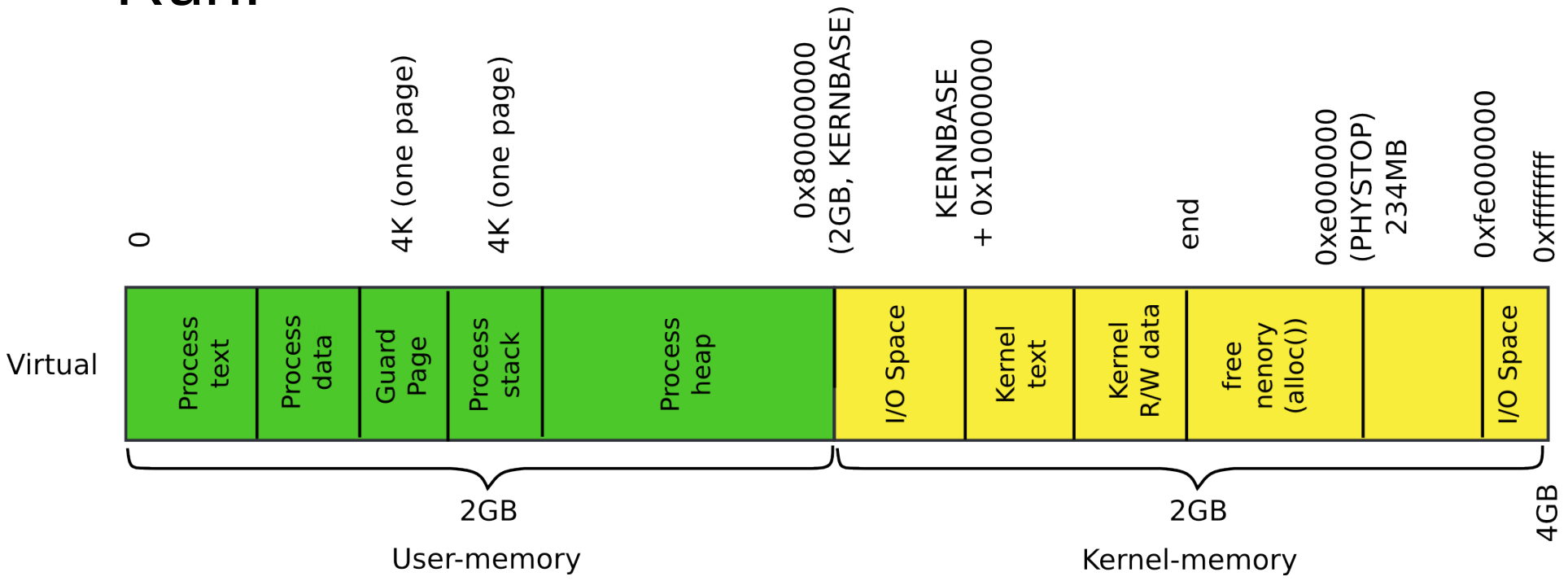
exec(): high-level outline

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



exec(): high-level outline

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



exec(): high-level overview

- Read process binary from disk
  - Locate a file that contains process binary
    - namei() takes a file path (“/bin/ls”) as an argument
    - Returns an inode
  - Read the file block by block
    - readi() reads the inode (file data) into memory
  - To read file in memory we need to construct the process address space
    - A page table specifically for the process

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;
```

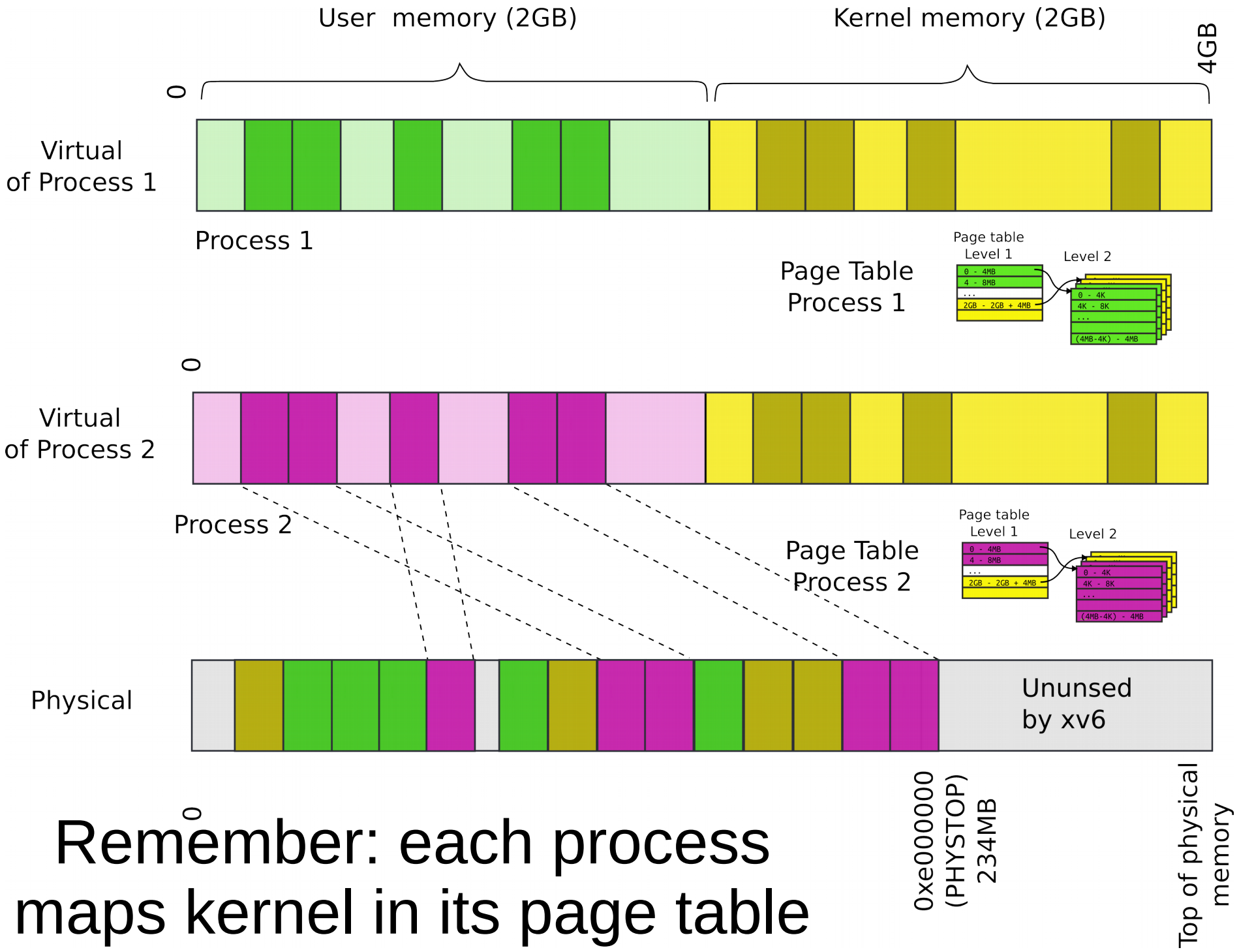
exec(): check ELF header

```
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;
```

Create process address space

`exec()`: Construct process address space

- Two step process
  - Create the kernel part of the address space
  - Create the user part of the address space



Remember: each process maps kernel in its page table

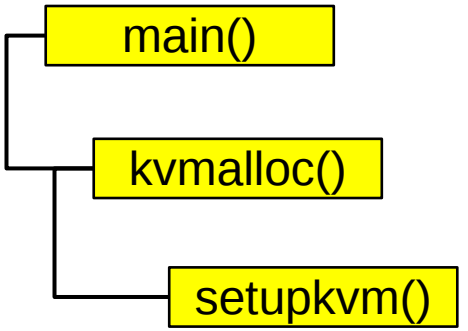
exec(): Setup kernel address space()

```
6310 exec(char *path, char **argv)
6311 {
...
6331   if(elf.magic != ELF_MAGIC)
6332     goto bad;
6334   if((pgdir = setupkvm()) == 0)
6335     goto bad;
...
```

- Remember from last time?

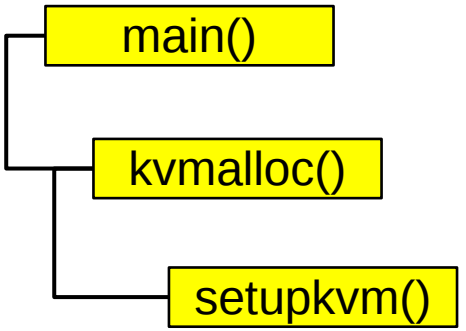
Recap: Allocate page table directory

```
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 }
```



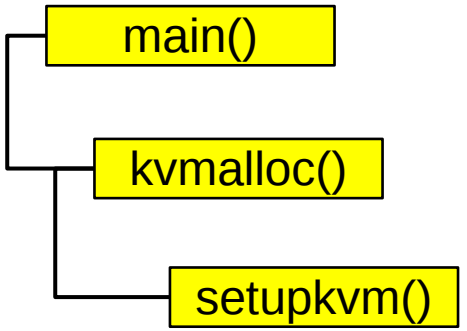
Recap: Iterate in a loop: remap physical pages

```
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 }
```

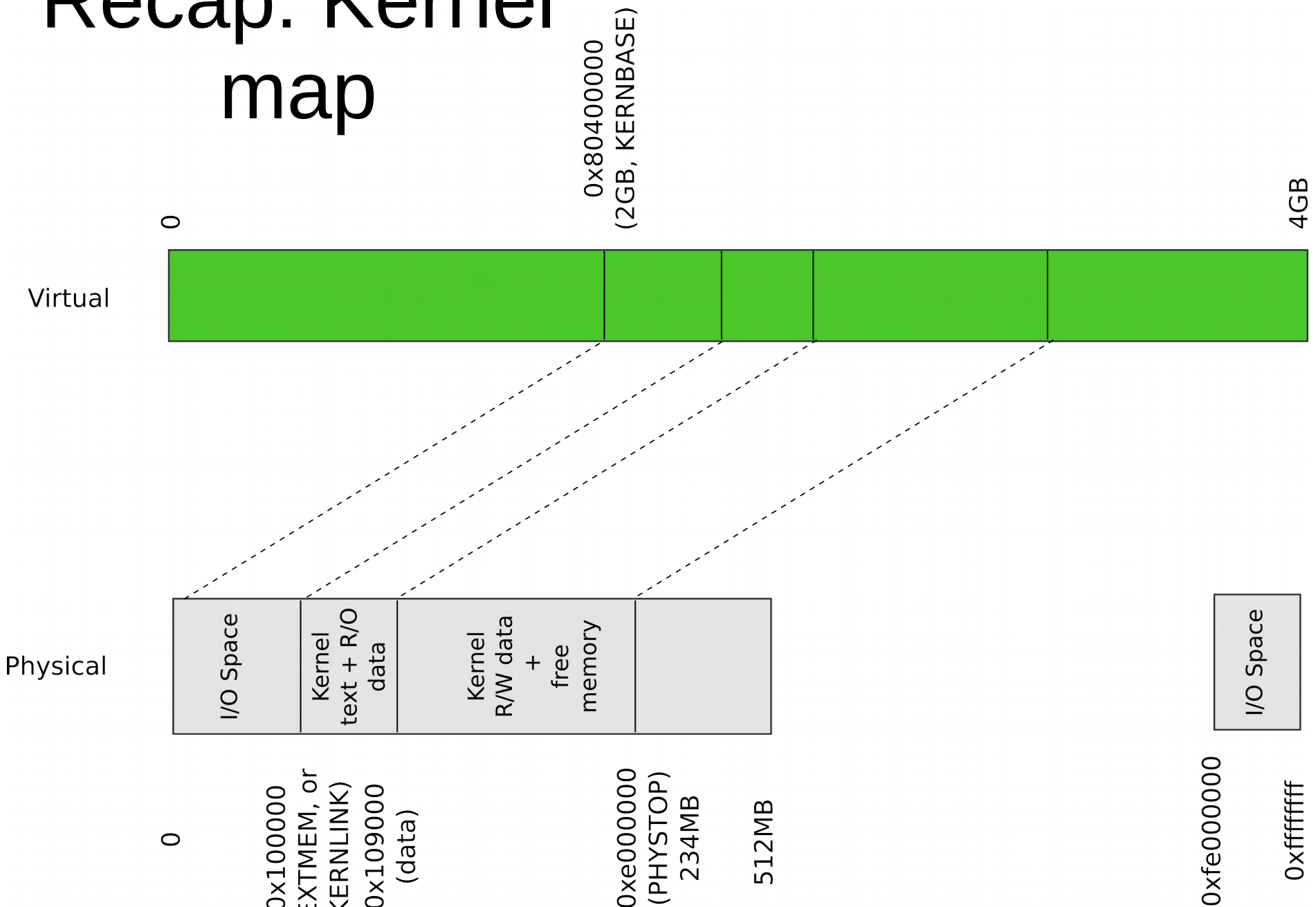


Recap: Iterate in a loop: remap physical pages

```
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 }
```



Recap: Kernel map

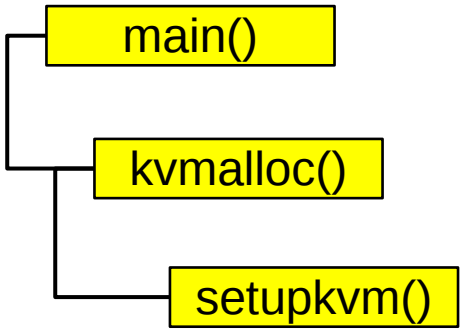


Recap: Kmap – kernel map

```
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 };
```

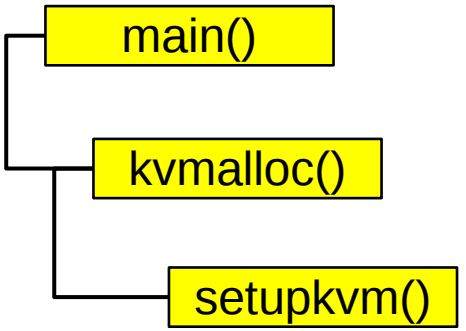
Recap: Iterate in a loop: remap physical pages

```
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 }
```

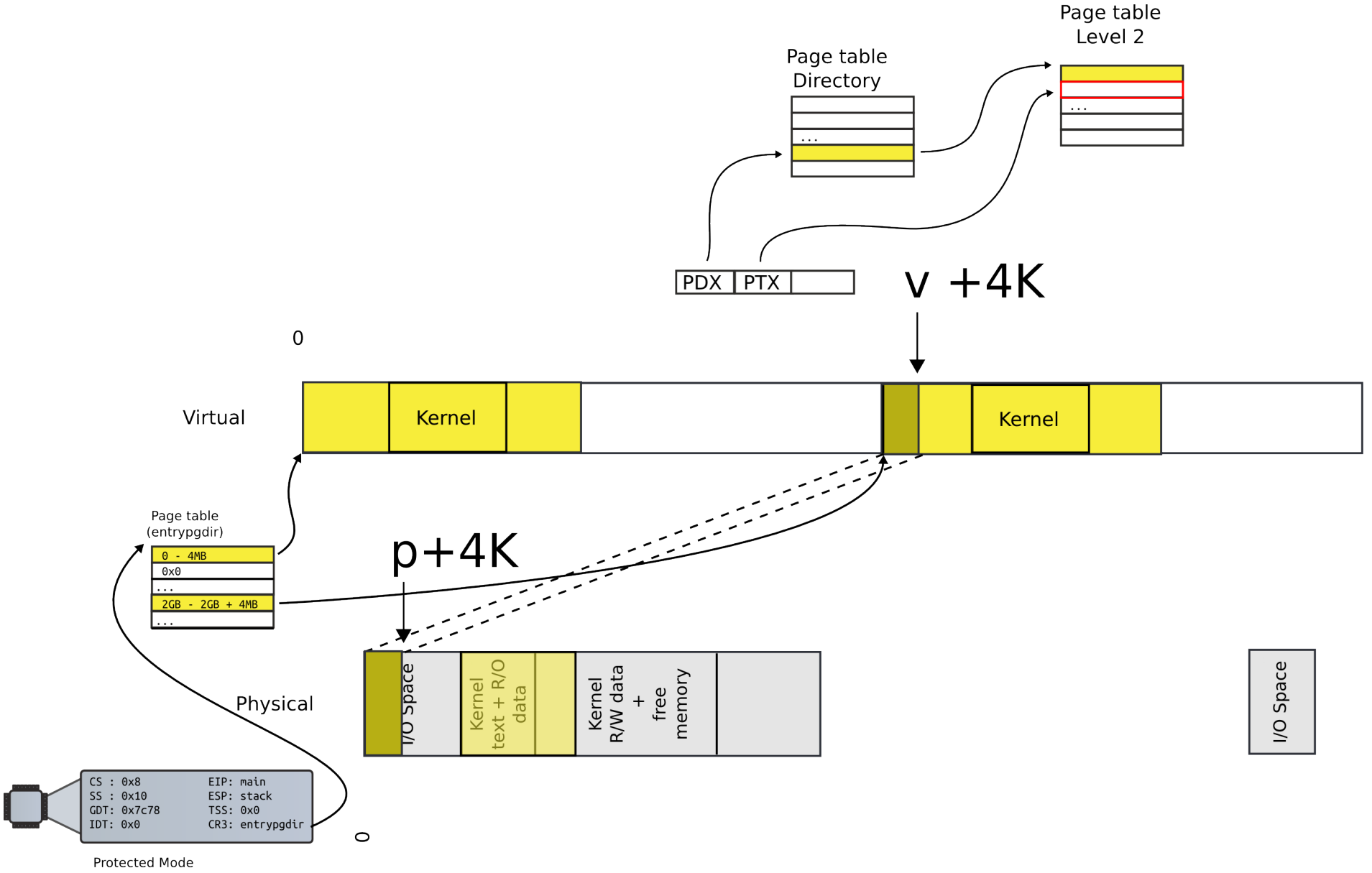


Recap: Remap physical pages

```
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 }
```



setupkvm(): Move to next page



`exec()`: Construct process address space

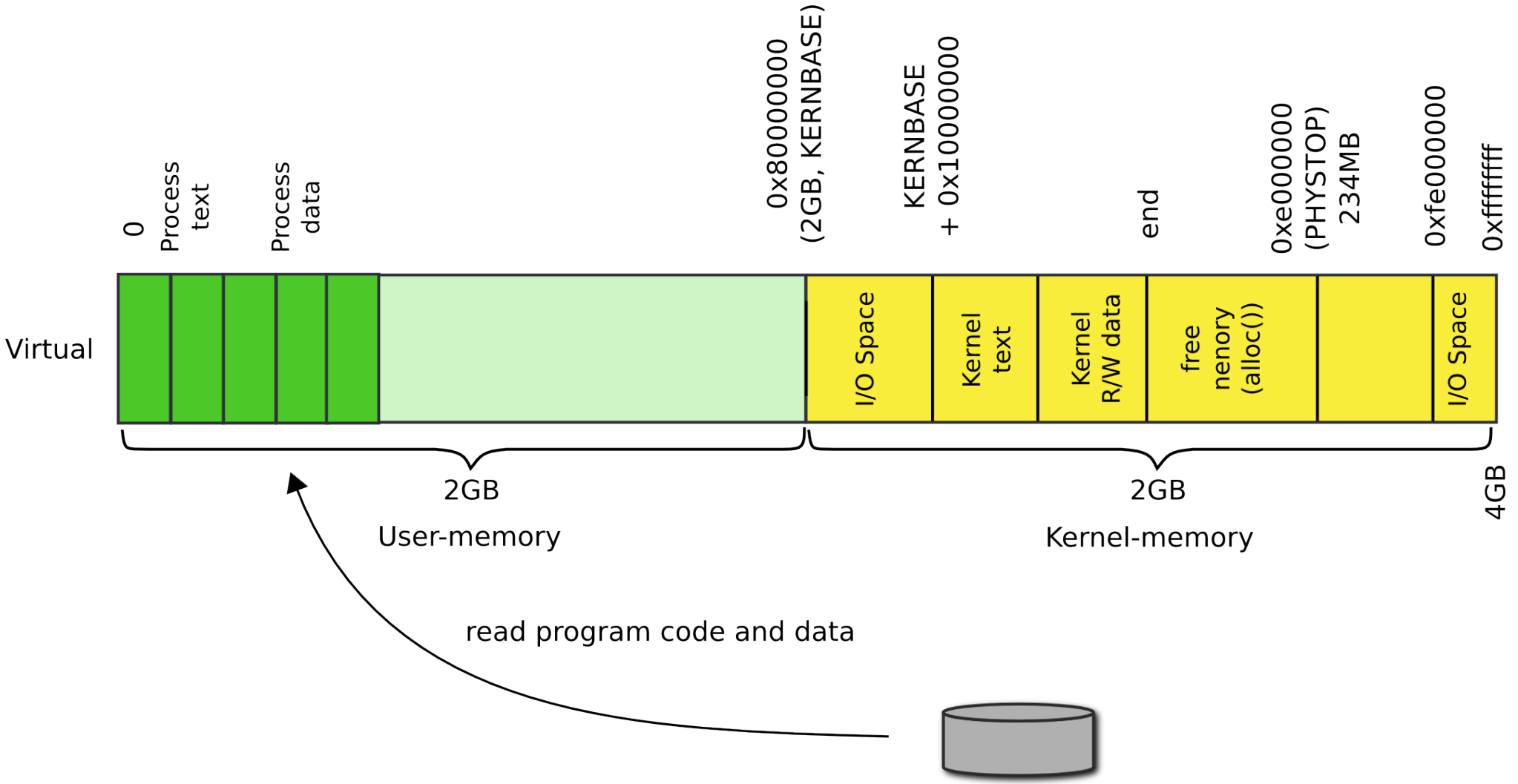
- Two step process
  - Create the kernel part of the address space
  - **Create the user part of the address space**

Create user part of the address space

exec(): create user part of the address space

- The goal is to fill in the page table entries
  - This can be naturally combined with loading the program from disk into memory
- At a high level iterate in a loop
  - On each step:
    - Allocate user-level pages for the program
    - Map them by filling in the page table entries
    - Read data from the inode into that memory

High-level idea



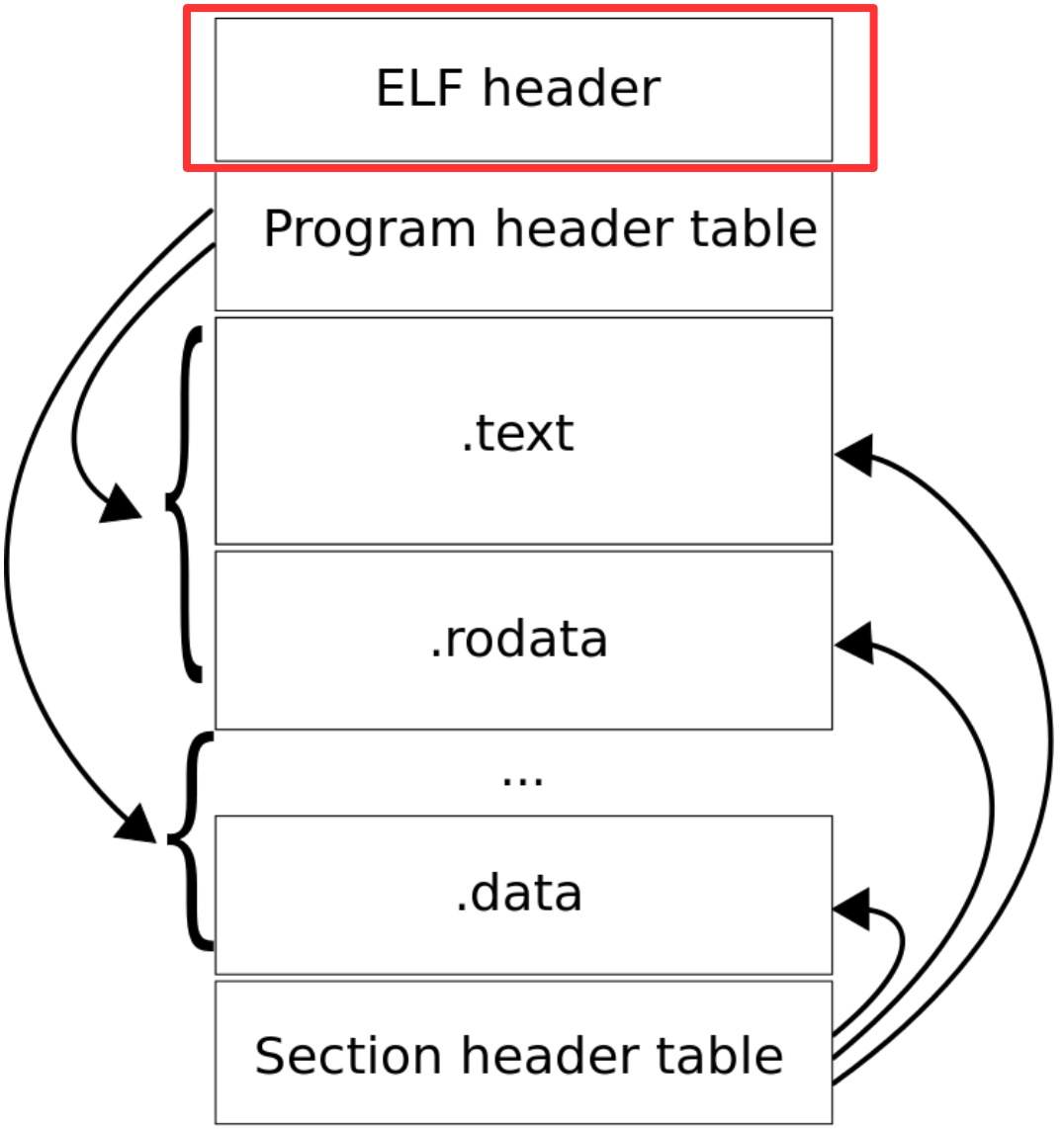
Program loading loop

```
6310 exec(char *path, char **argv)
6311 {
...
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

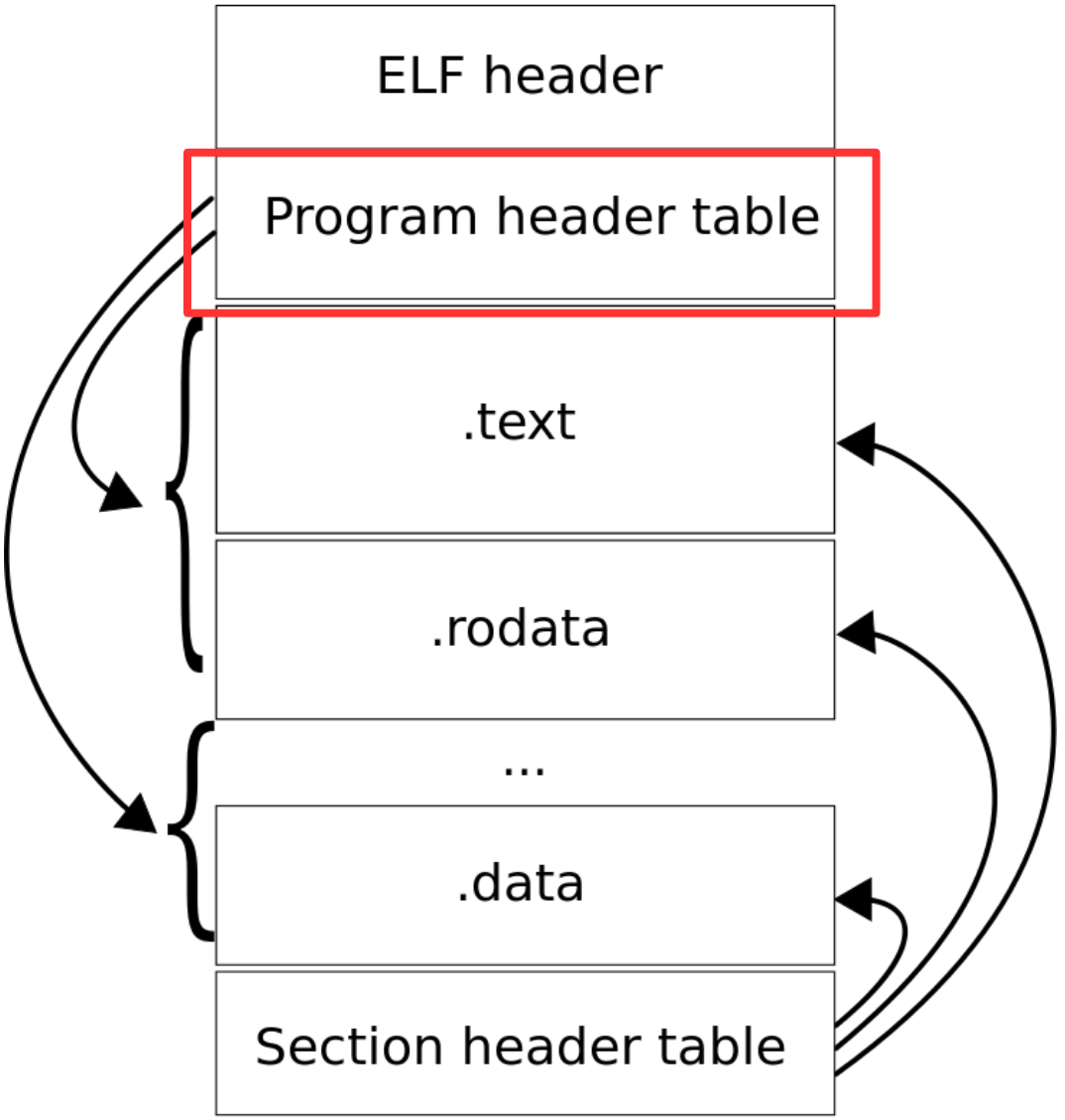
ELF object file

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



ELF object file

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



Program loading loop

```
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

Program loading loop

```
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

Program loading loop

```
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)

Program loading loop

```
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

Program loading loop

```
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 }
```

- Current size of the user address space
- Initially it's 0

Program loading loop

```
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 }
```

- New size of the address space

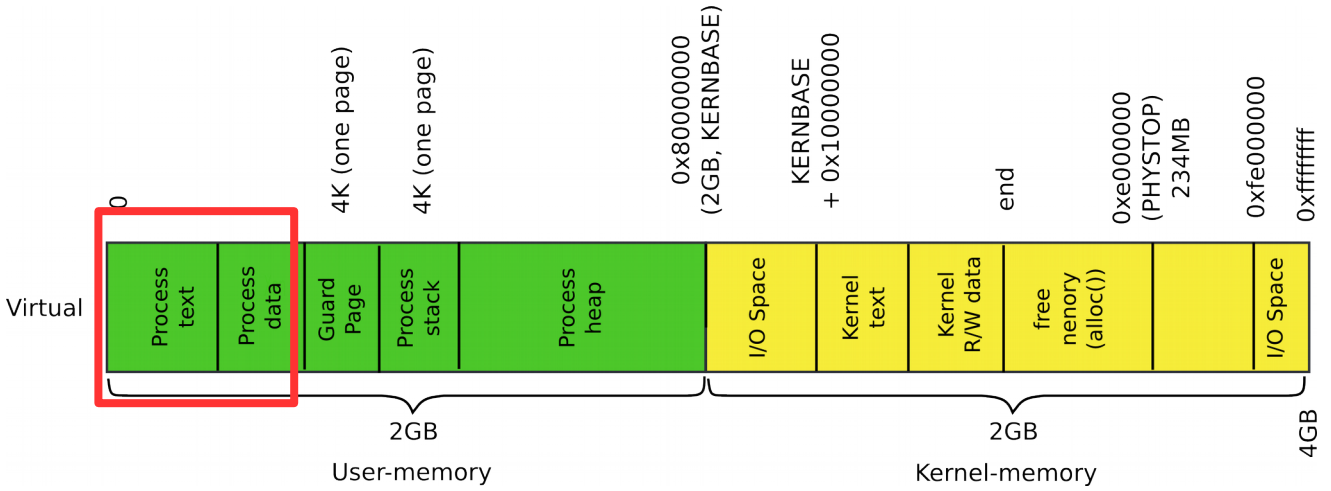
Program loading loop

```
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 }
```

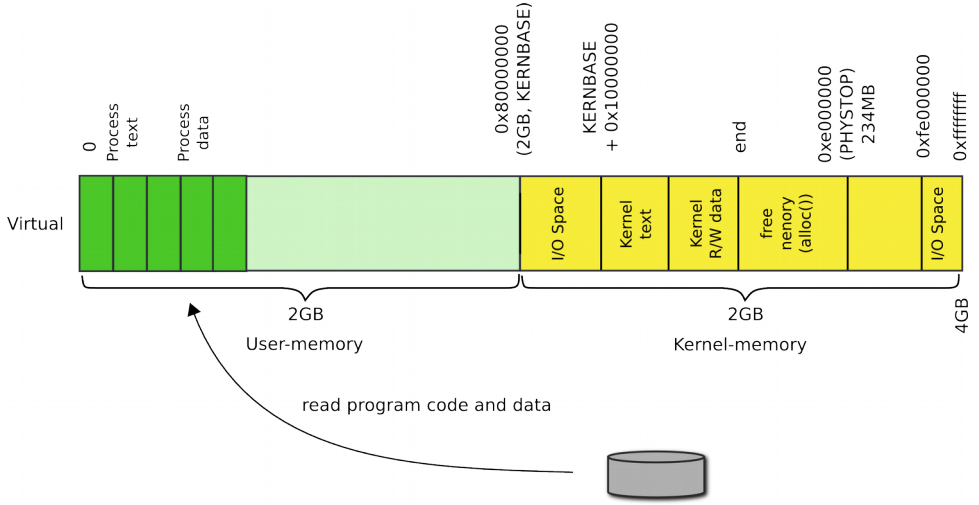
- Load program section from disk

Two main functions

- allocuvm() -- allocate and map user-memory

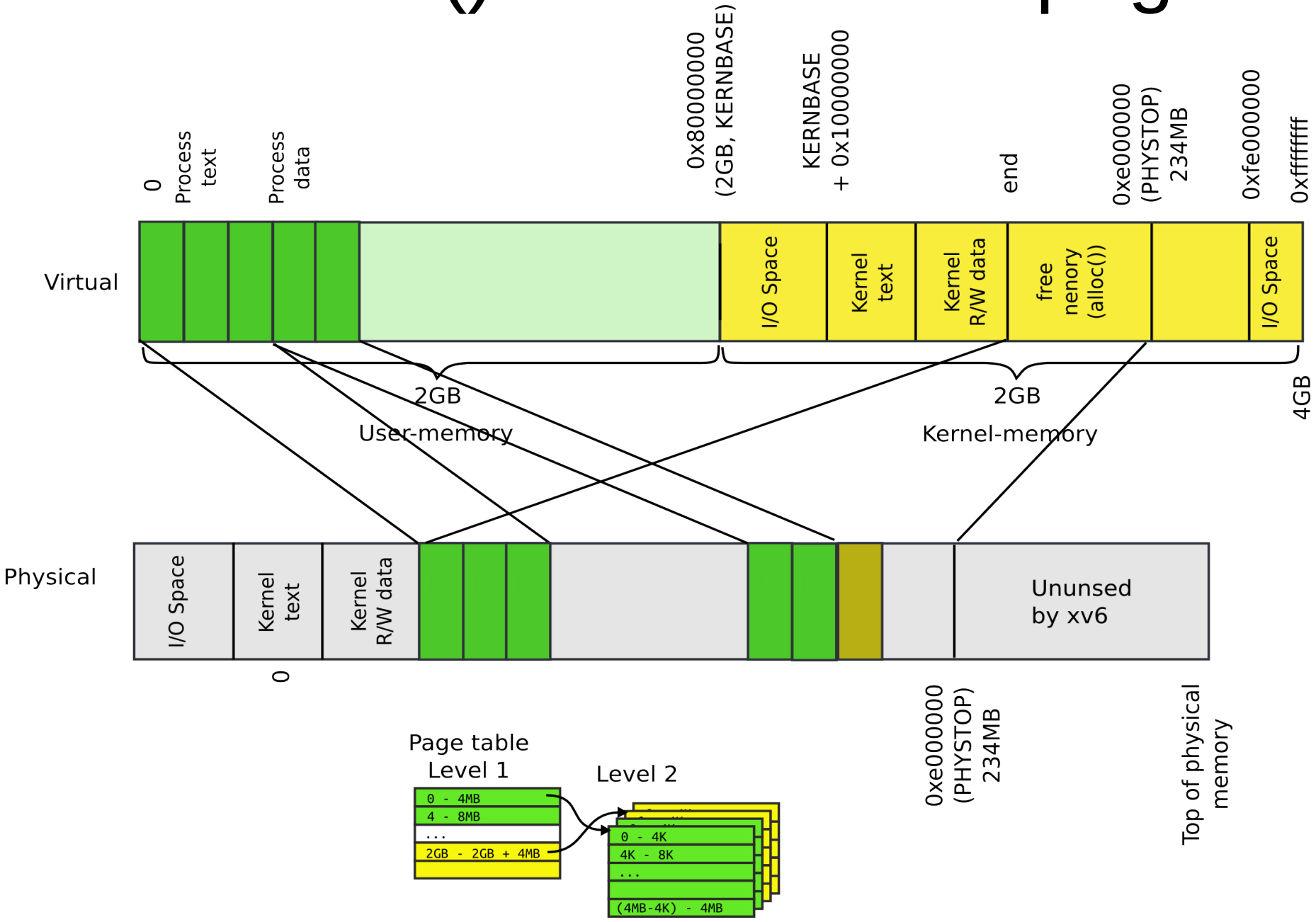


- loaduvm() -- load user-memory with data from disk



Lets take a closer look
allocuvm()

allocuvm(): allocate user pages



Allocate user address space

```
1953 allocuvm(pde_t *pgdir, uint oldsz, uint newsz)
1954 {
...
1958   if(newsz >= KERNBASE)
1959     return 0;
…
1963   a = PGROUNDUP(oldsz);
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 }
```

- New size can't be over 2GB

Allocate user address space

```
1953 allocuvm(pde_t *pgdir, uint oldsz, uint newsz)
1954 {
...
1958   if(newsz >= KERNBASE)
1959     return 0;
…
1963   a = PGROUNDUP(oldsz);
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 }
```

- Start with the old size rounded up to the nearest page

```
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 }
```

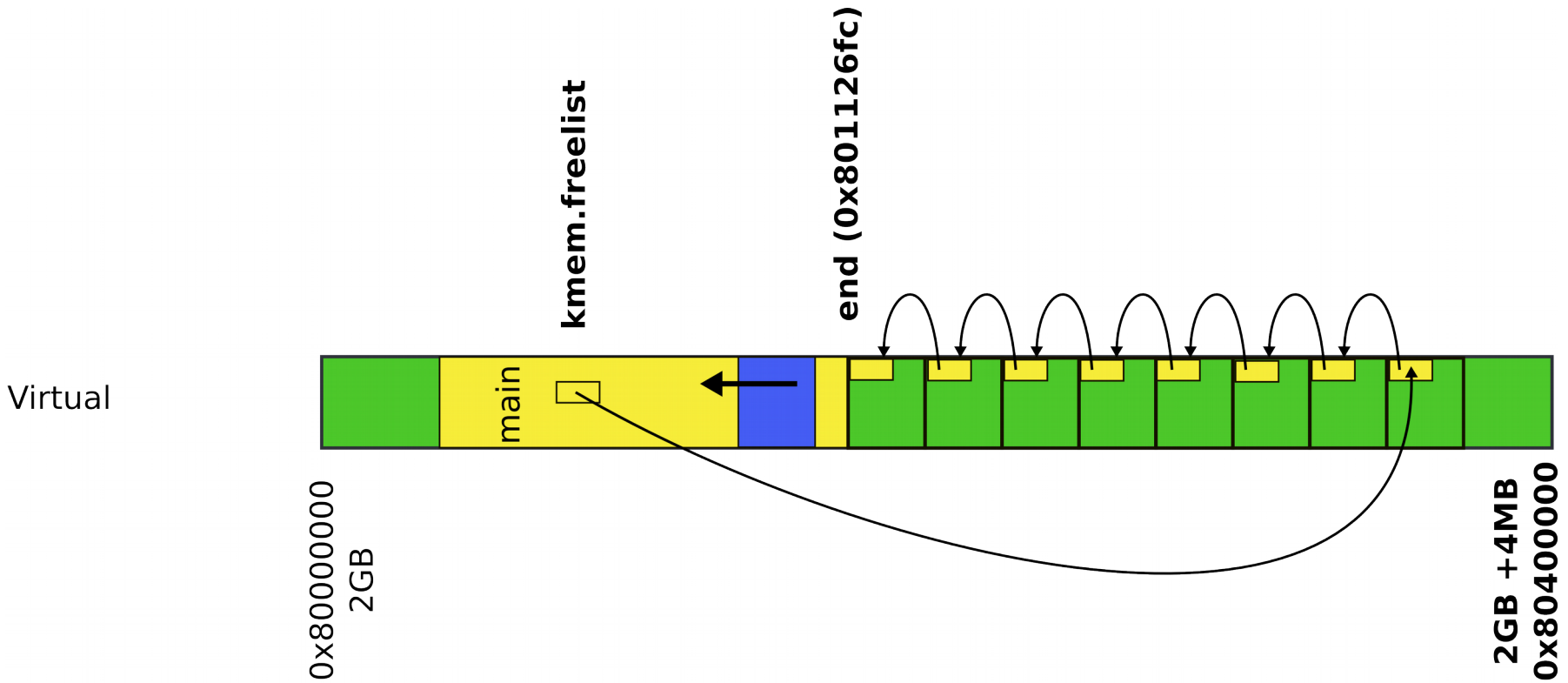
Allocate user address space

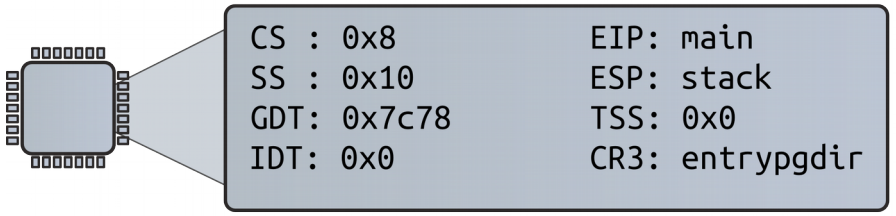
- Allocate a new page

```
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 }
```

Where does this memory come from?

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 address space

- 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 address space

- Map the page

Allocate user address space

```
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 }
```

- Take the page directory as an argument

```
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 address space

- Virtual address where to map the page

Allocate user address space

```
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 }
```

- Size of the region
  - One page!

```
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 address space

- Physical address of the page we're mapping
  - V2P!

Allocate user address space

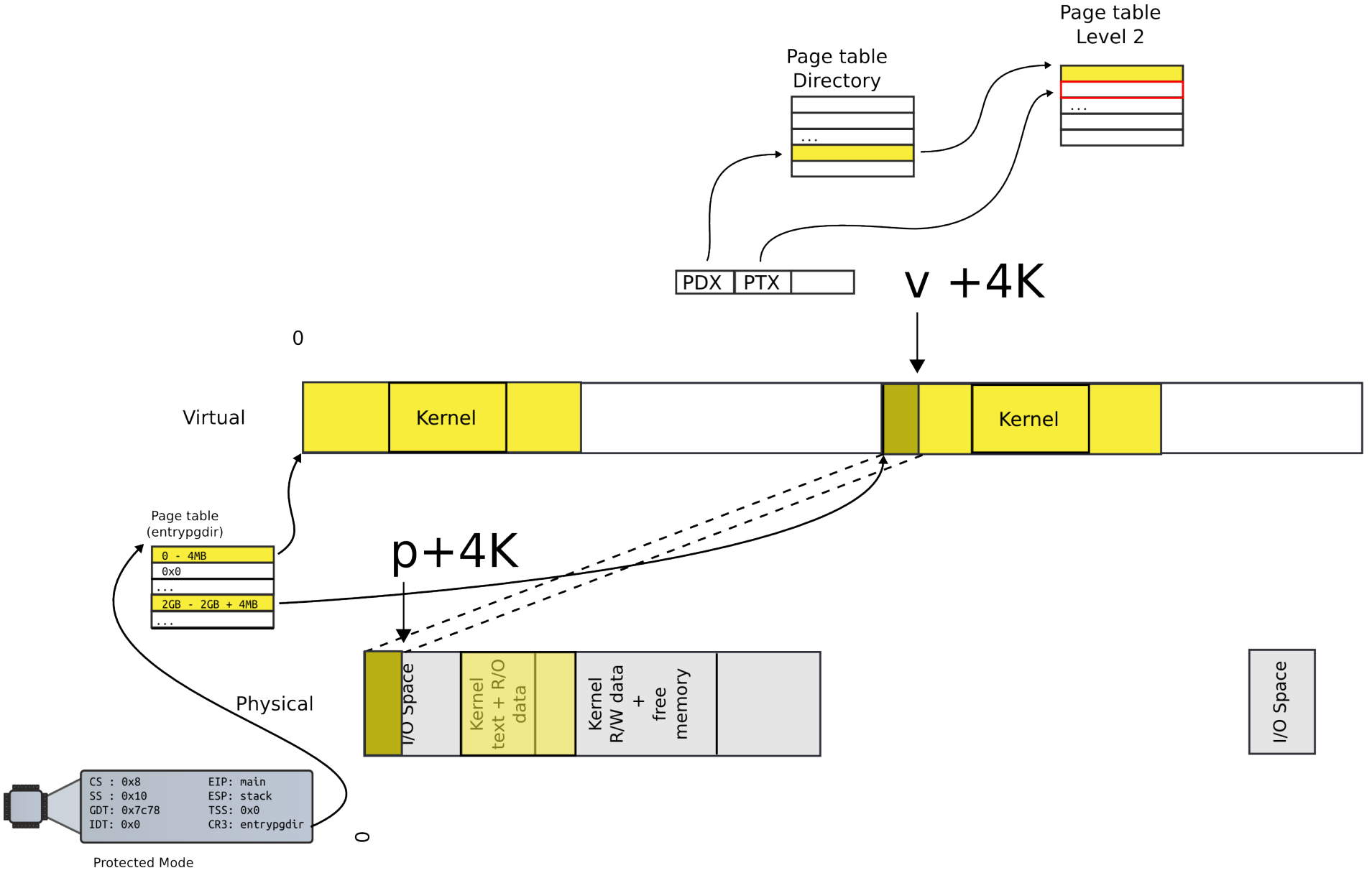
```
1953 allocuvm(pde_t *pgdir, uint oldsz, uint newsz)
1954 {
...
1958   if(newsz >= KERNBASE)
1959     return 0;
...
1964   for(; a < newsz; a += PGSIZE){
1965     mem = kalloc();
...
1971     memset(mem, 0, PGSIZE);
1972     if(mappages(pgdir, (char*)a, PGSIZE, V2P(mem), PTE_W|PTE_U) < 0){
...
1976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

- Flags
  - Writable and user-accessible

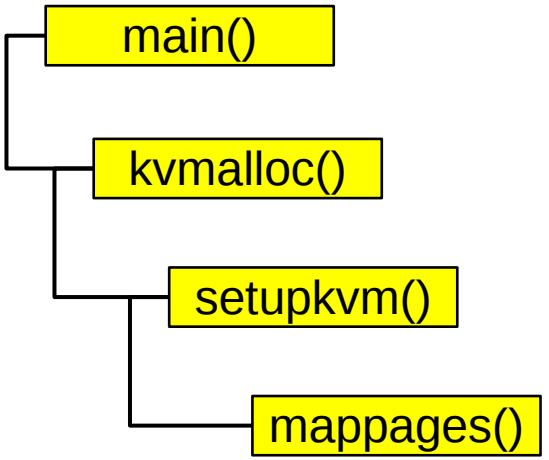
Who remembers mappages()?

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

mappages(): map a region



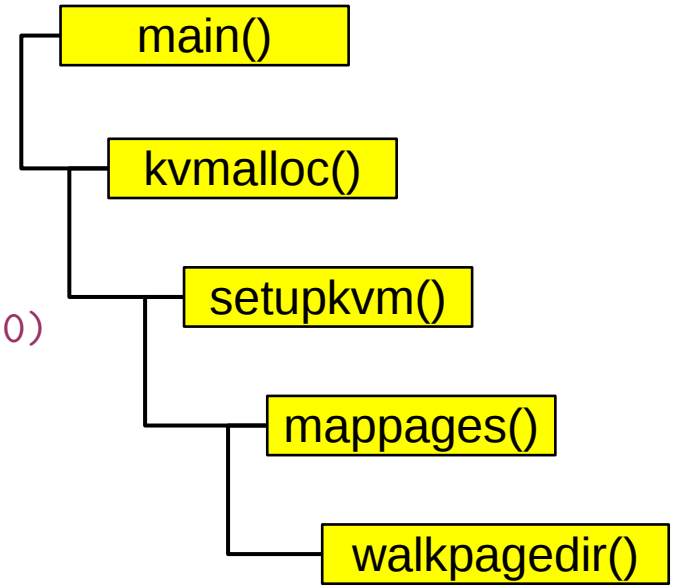
```
1779 mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm)
1780 {
1781   char *a, *last;
1782   pte_t *pte;
1783
1784   a = (char*)PGROUNDDOWN((uint)va);
1785   last = (char*)PGROUNDDOWN(((uint)va) + size - 1);
1786   for(;;){
1787     if((pte = walkpgdir(pgdir, a, 1)) == 0)
1788       return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



Allocate user address space

```
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 }
```

- Continue in a loop
  - Map pages one by one

Now the second function: `loaduvm()`

exec() – create a new process

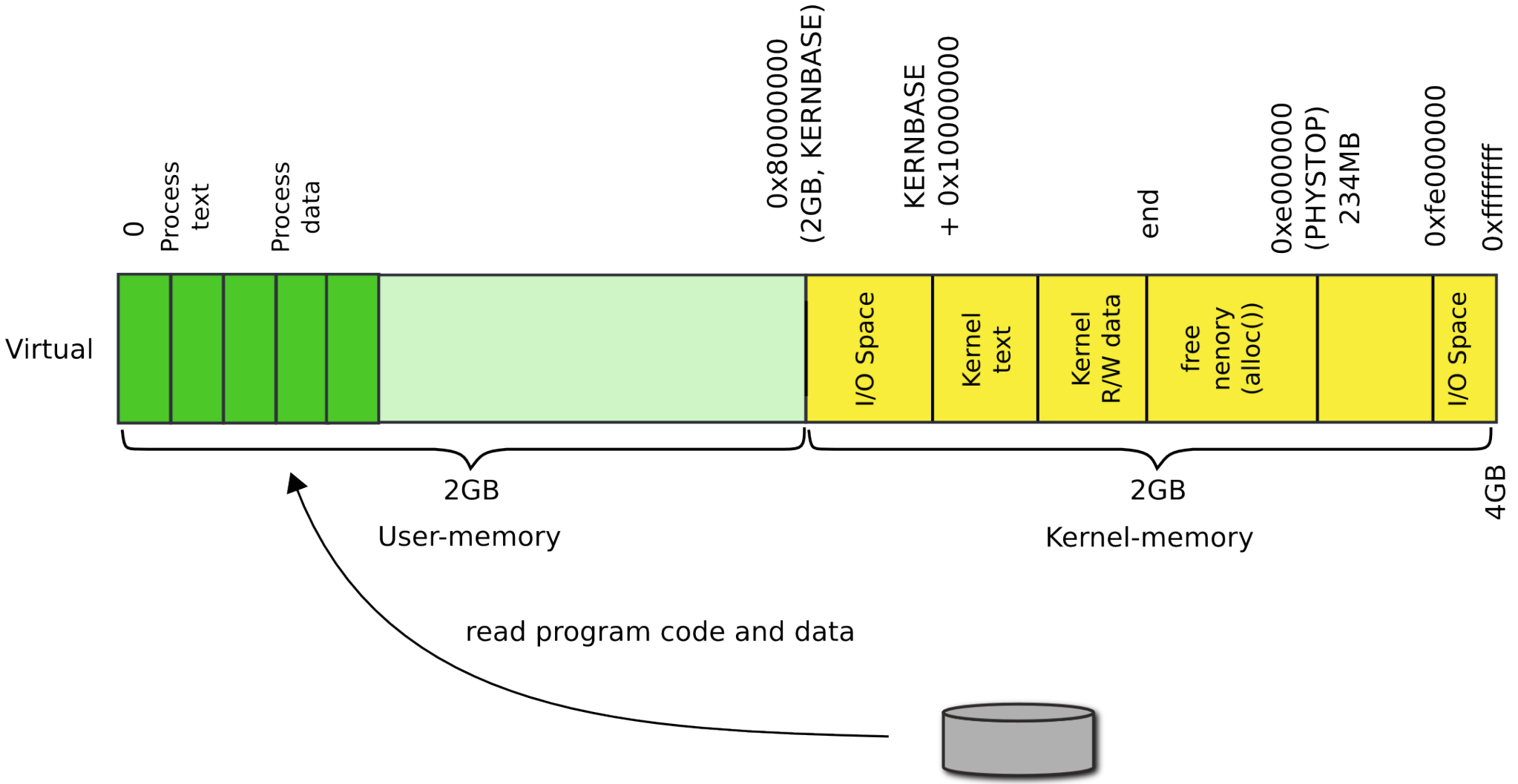
- Read process binary from disk
  - namei() takes a file path (“/bin/ls”) as an argument
  - Returns an inode
  - readi() reads the inode (file data)
- Create process address space
  - Create a page table
  - Map only kernel space
- **Load program into memory**
  - Allocate user-level pages for the program
  - **Read data from the inode into that memory**

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

- Locate pte
  - addr is virtual address where the program has to be loaded

Load program into memory

```
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 }
```

- Pte (page table entry) of the physical page backing up the virtual page

Load program into memory

```
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 }
```

- Resolve pte into physical address

Load program into memory

```
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 }
```

- Then use the virtual address of that physical page

Load program into memory

```
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 }
```

- Wait... virtual address of a page?

Load program into memory

```
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 }
```

- Why can't we use addr directly?

Drawing: process and kernel page tables

```
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 }
```

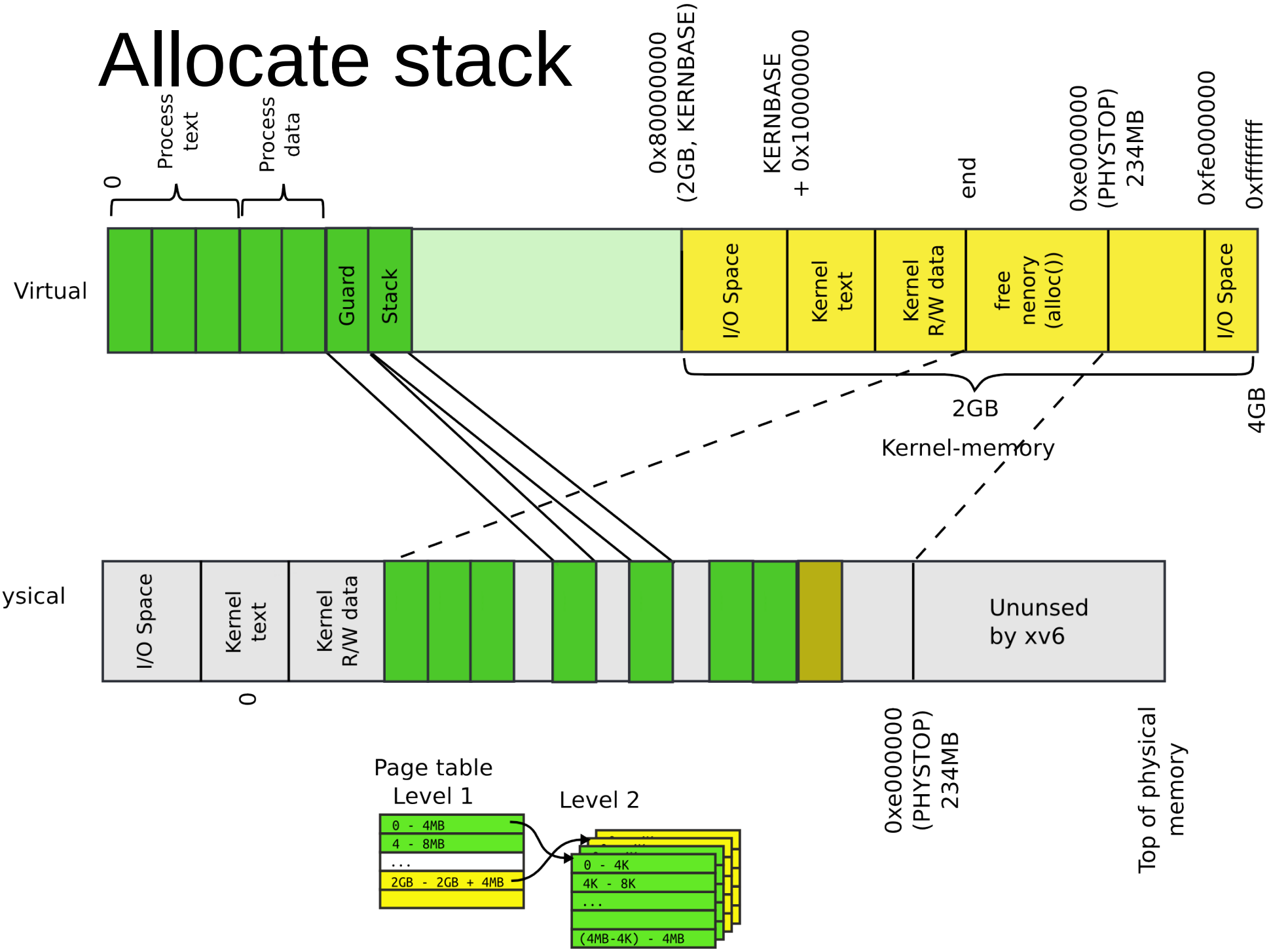
Load program into memory

- Read the page from disk

exec() – create a new process

- Read process binary from disk
- Create process address space
- Load program into memory
- **Allocate program stack**

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;
```

exec() – create a new process

- Read process binary from disk
- Create process address space
- Load program into memory
- Allocate program stack
- **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)

```
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;
```

Copy elements of the array one by one on the stack

```
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 argc – number of arguments in the argv[]

```
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 argv pointer – argv[] is on the stack itself

exec() – create a new process

- Read process binary from disk
- Create process address space
- Load program into memory
- Allocate program stack
- Push program arguments on the stack
- **Switch page tables**

exec(): switch page tables

- Switch page tables
- Deallocate old page table

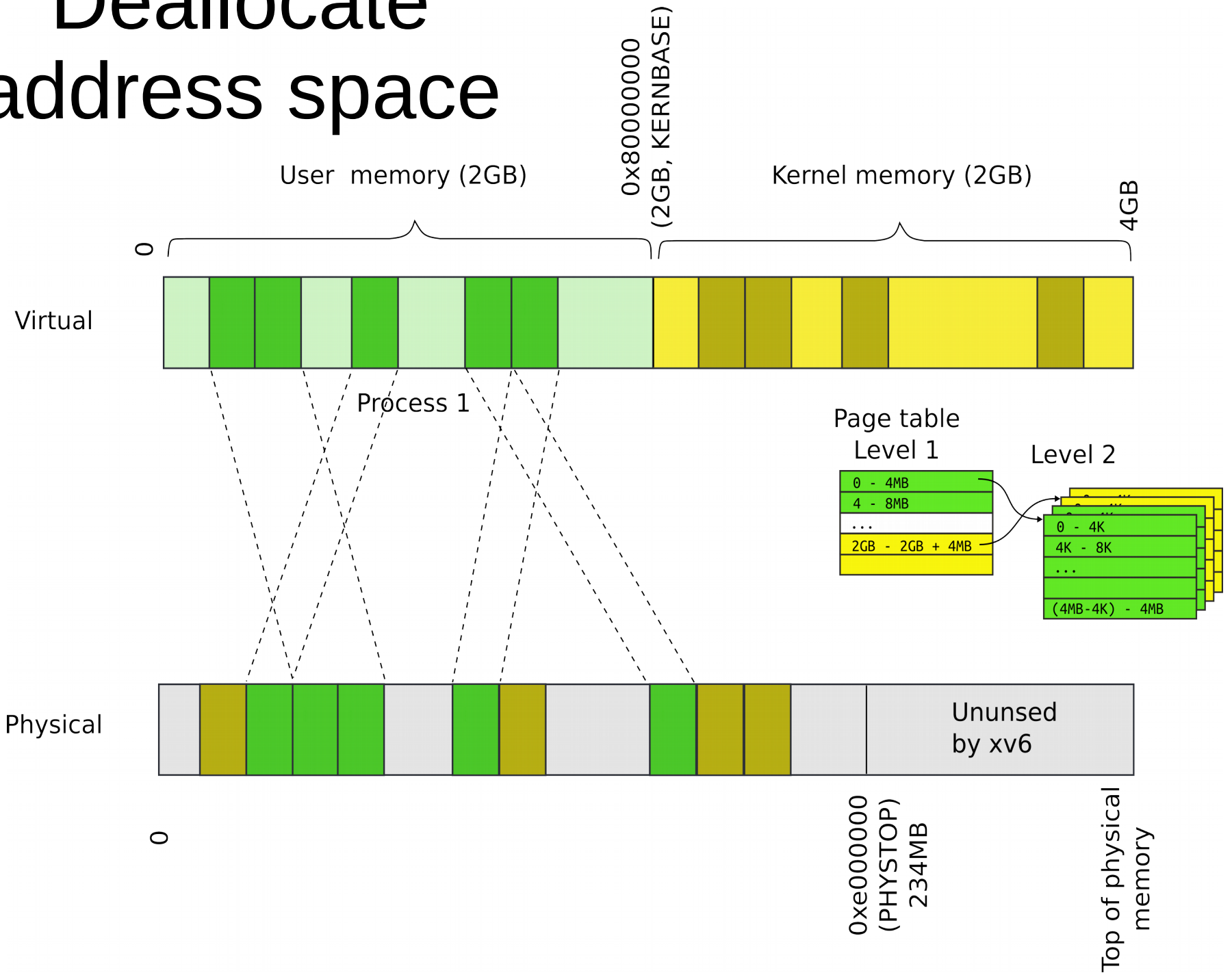
```
6309 int
6310 exec(char *path, char **argv)
6311 {
…
6398   switchuvm(proc);
6399   freevm(oldpgdir);
6400   return 0;
...
6410
```

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 }
```

Deallocate page table directory itself

```
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!