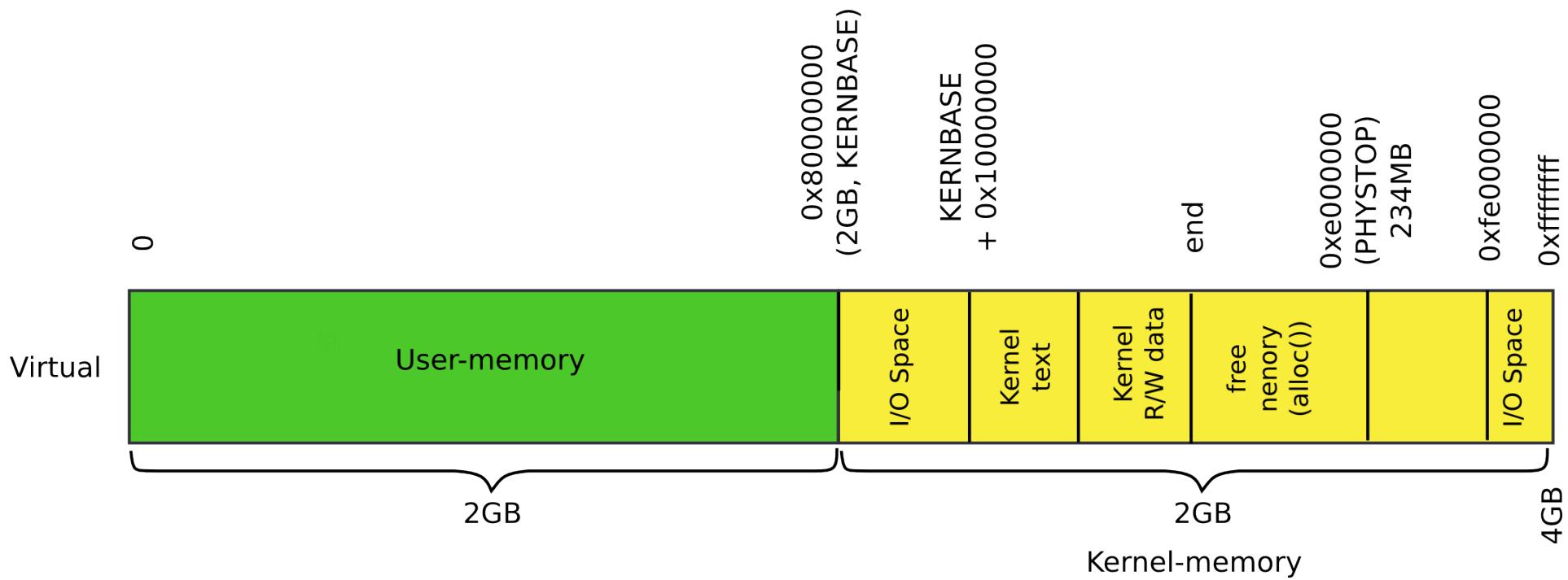


# cs5460/6460: Operating Systems

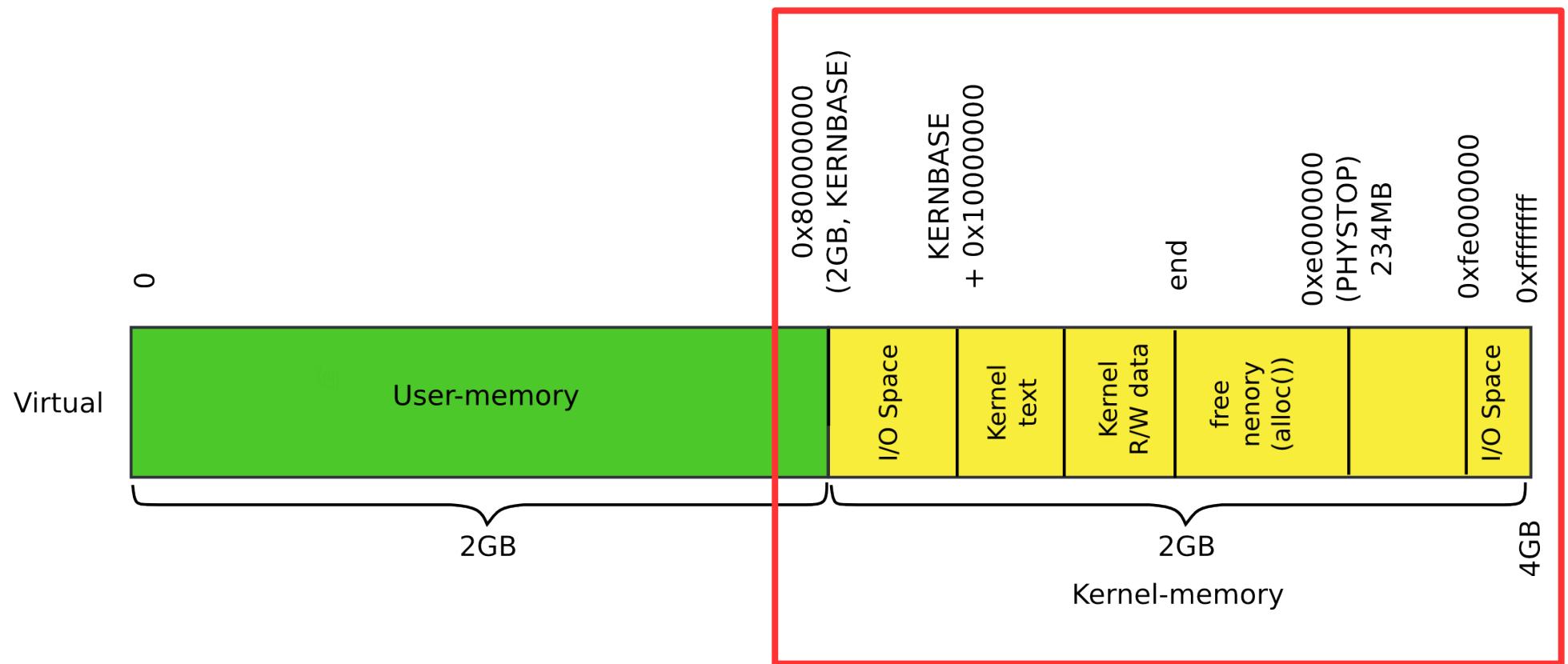
## Lecture: Creating Processes (`exec()`)

Anton Burtsev  
April 2023

# Recap: kernel memory



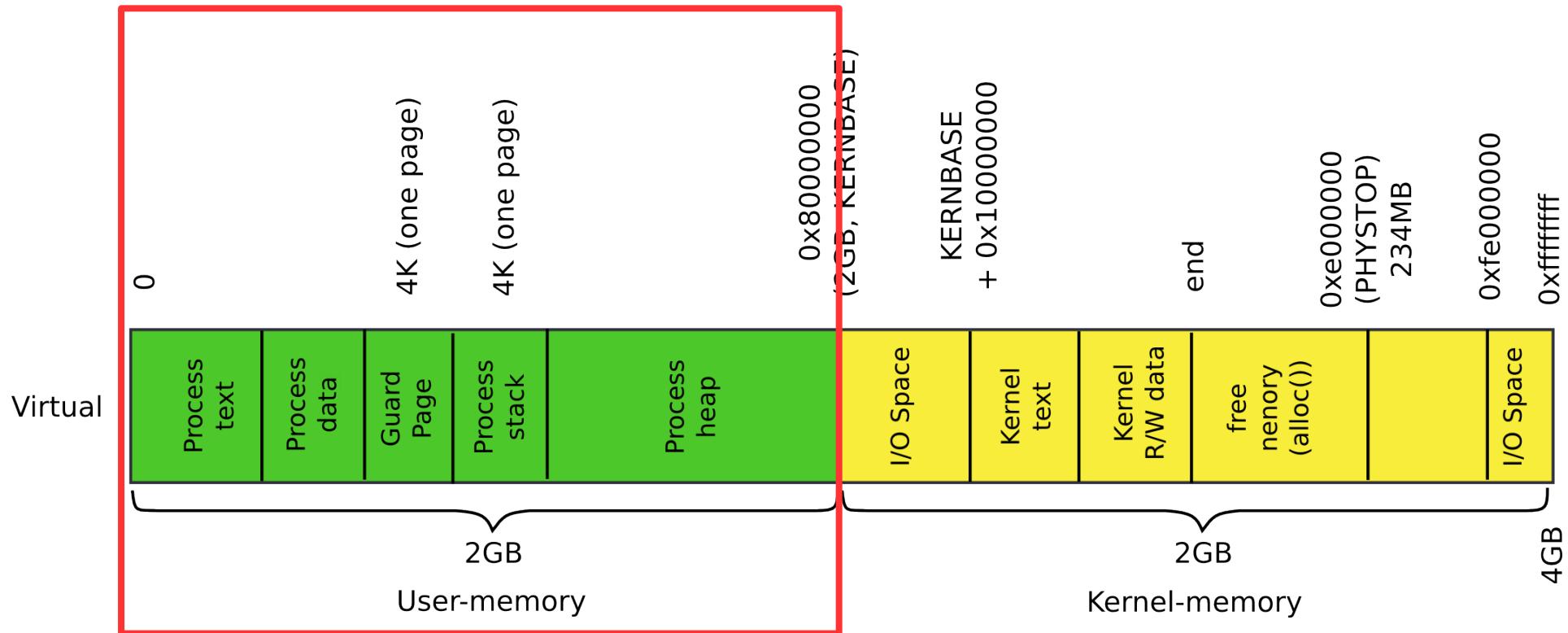
# Recap: kernel memory



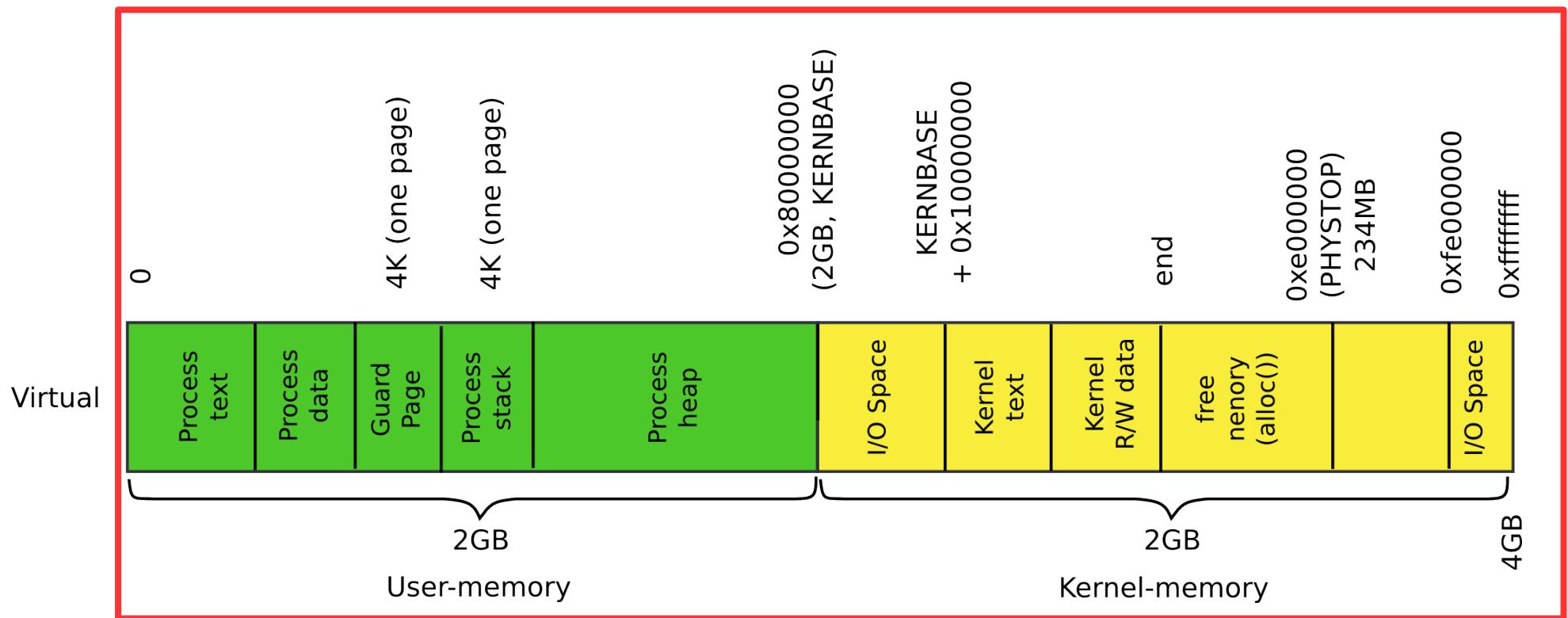
```
1317 main(void)
1318 {
1319     kinit1(end, P2V(4*1024*1024)); // phys page allocator
1320     kvmalloc(); // kernel page table
1321     mpinit(); // detect other processors
1322     lapicinit(); // interrupt controller
1323     seginit(); // segment descriptors
1324     cprintf("\ncpu%d: starting 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



# Today: process memory



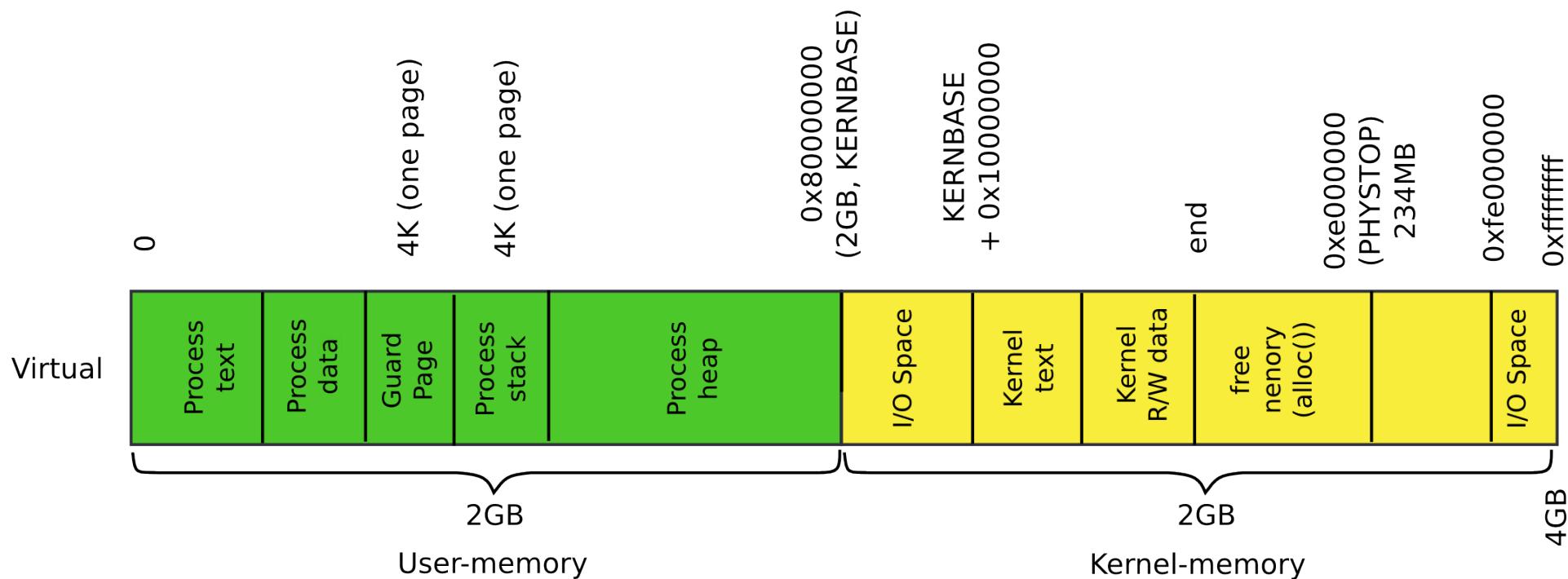
# How does kernel creates new processes?

# How does kernel creates new processes?

- Exec
  - `exec("/bin/ls", argv);`

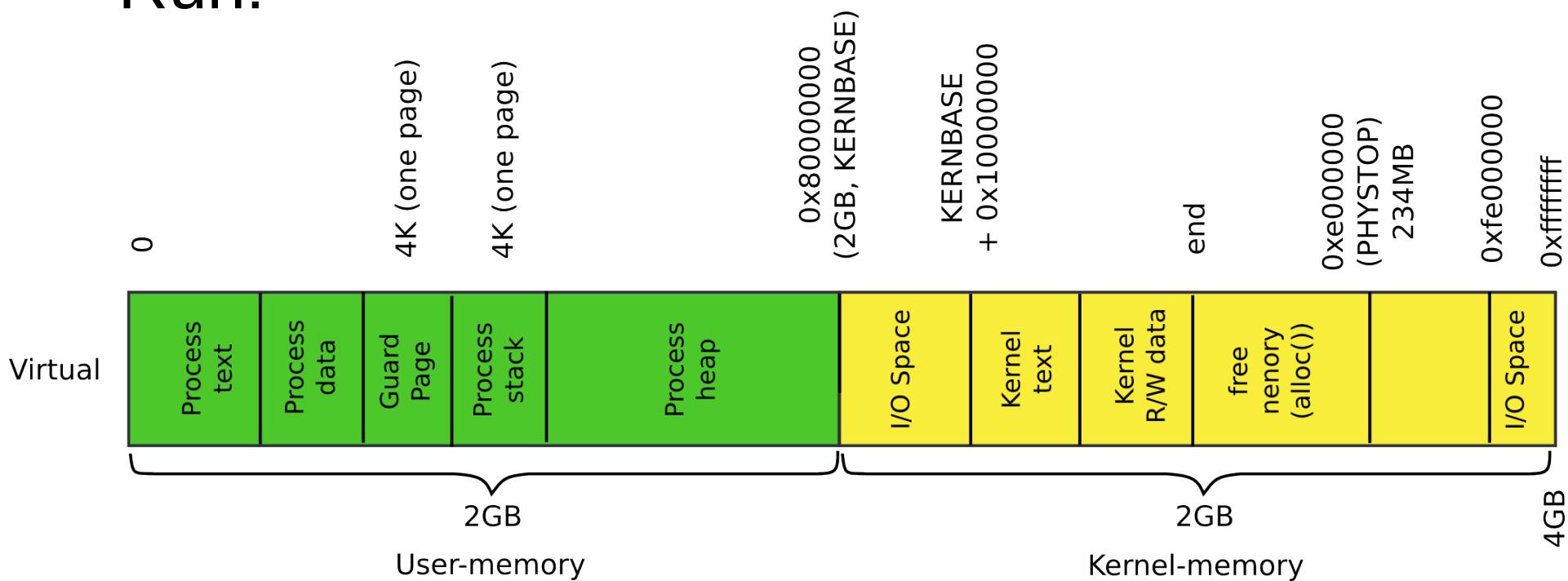
# exec(): high-level outline

- 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)) <
6330         sizeof(elf))
6331         goto bad;
6331     if(elf.magic != ELF_MAGIC)
6332         goto bad;
```

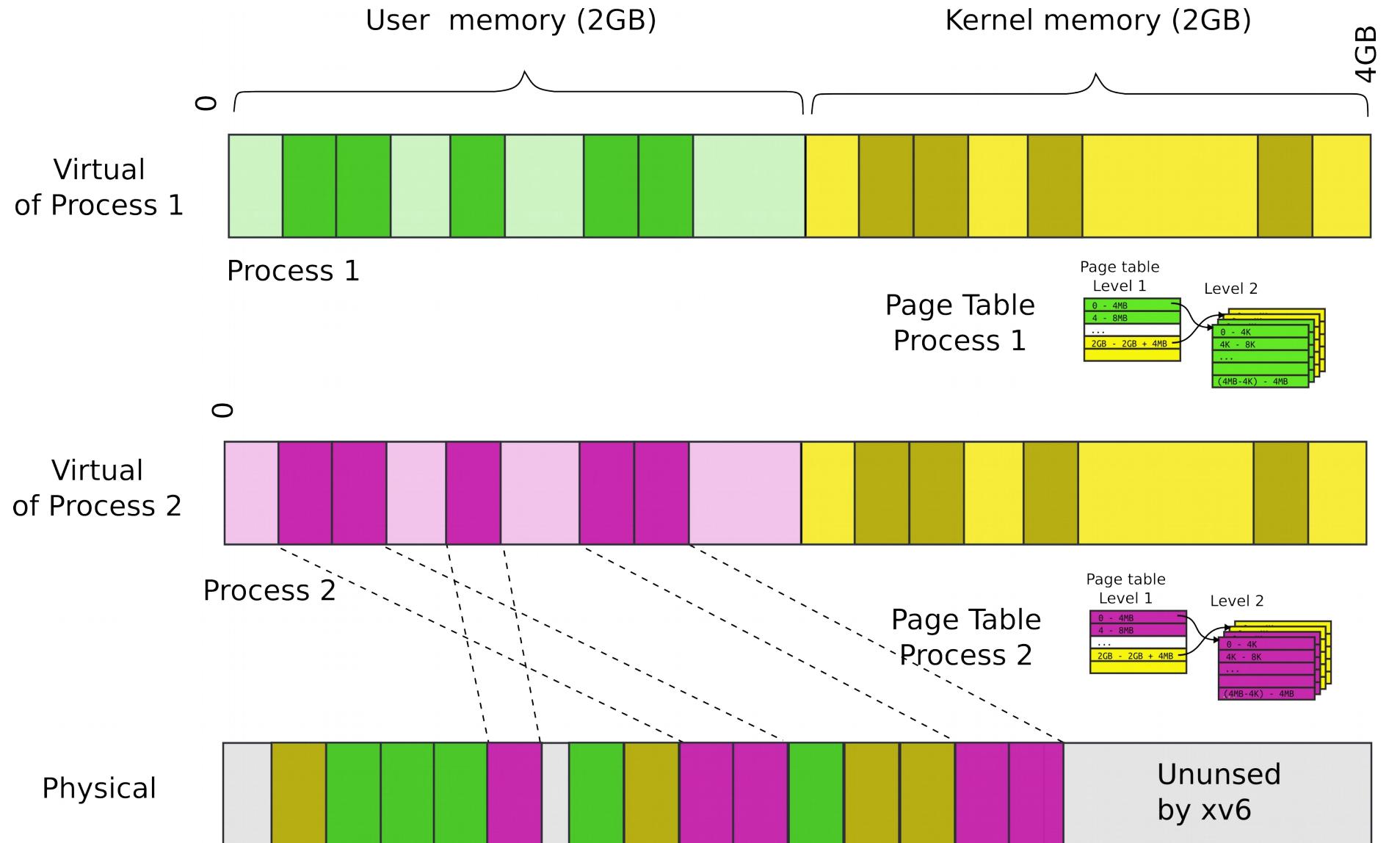
# exec(): check ELF header

```
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)) <
6330         sizeof(elf))
6331         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

0xe0000000  
(PHYSTOP)  
234MB

Top of physical  
memory

# 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;
...

```

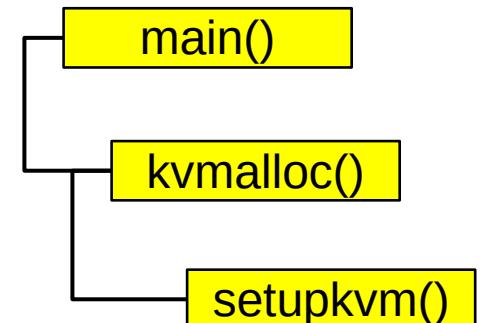
- Remember from the kernel memory allocator lecture?

```

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(pgd, 0, PGSIZE);
...
1847     for(k = kmap; k < &kmap[NELEM(kmap)]; k++)
1848         if(mappages(pgd, k->virt, k->phys_end - k->phys_start,
1849                     (uint)k->phys_start, k->perm) < 0)
1850             return 0;
1851     return pgd;
1852 }

```

# Recap: Allocate page table directory

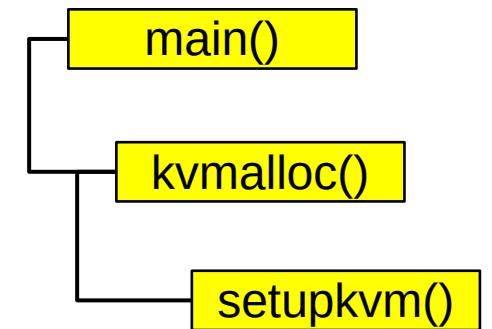


```

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

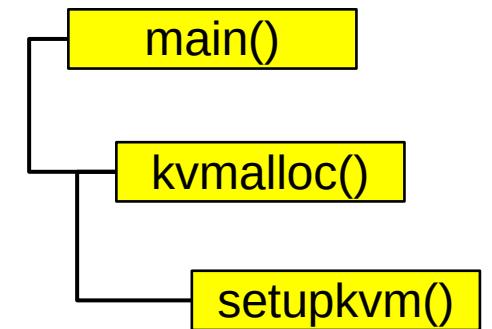


```

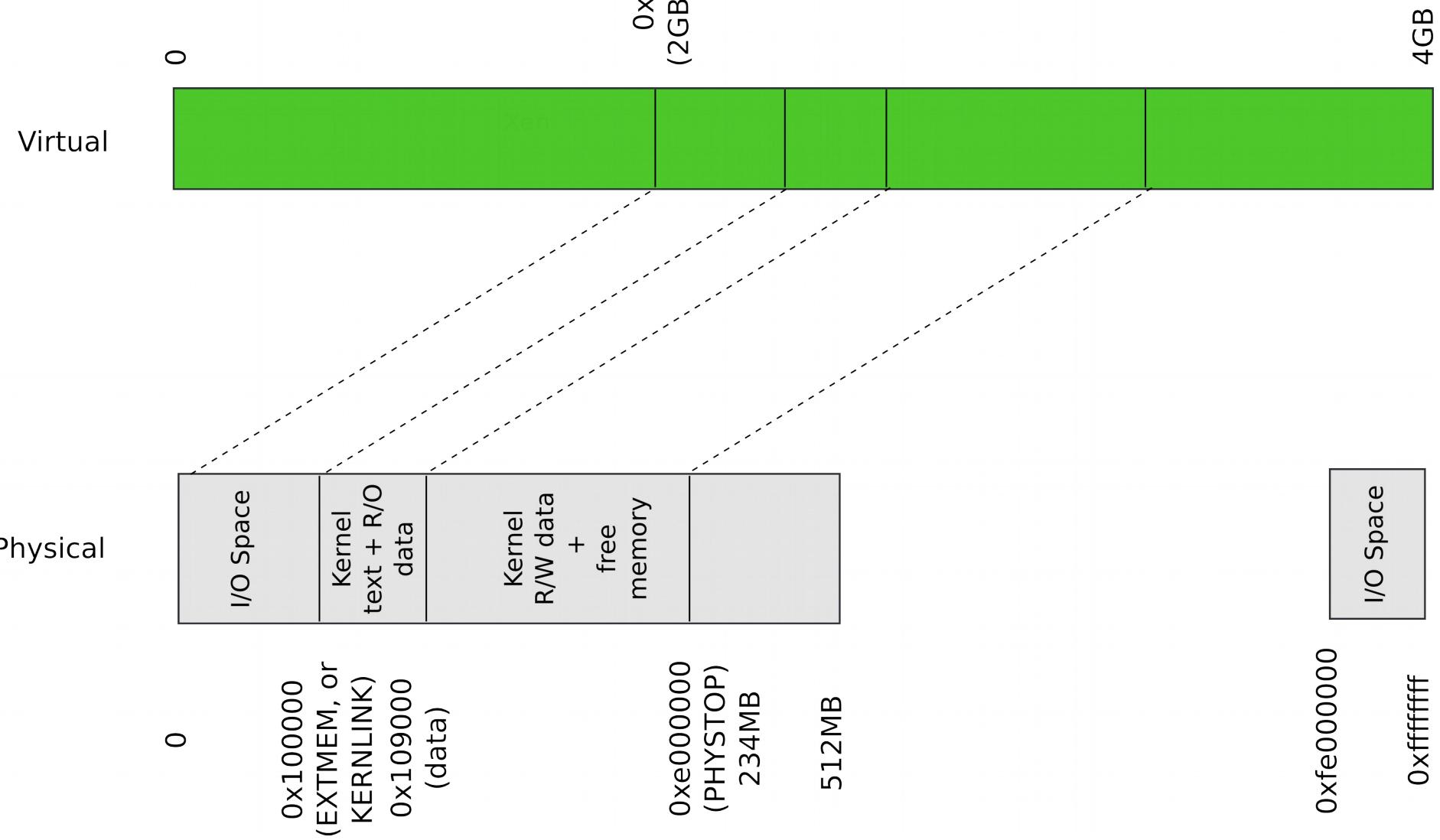
1836 pde_t*
1887 setupkvm(void)
1838 {
1839     pde_t *pgdir;
1840     struct kmap *k;
1841
1842     if((pgdir = (pde_t*)kalloc()) == 0)
1843         return 0;
1844     memset(pgdir, 0, PGSIZE);
...
1847     for(k = kmap; k < &kmap[NELEM(kmap)]; k++)
1848         if(mappages(pgdir, k->virt, k->phys_end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

```

# Recap: Iterate in a loop: remap physical pages



# Recap: Kernel map



# Recap: Kmap – kernel map

```
1823 static struct kmap {  
1824     void *virt;           Physical  
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 };
```

The diagram illustrates the kernel map structure. It consists of a horizontal bar divided into four segments. From left to right:

- I/O Space**: Physical address 0x100000 (EXTMEM, or KERNLINK), size 0x109000 (data).
- Kernel text + R/O data**: Physical address 0xe000000 (PHYSTOP), size 234MB.
- Kernel R/W data + free memory**: Physical address 0x109000 to 0xe000000.
- I/O Space**: Physical address 0xe000000 to 0xffffffff.

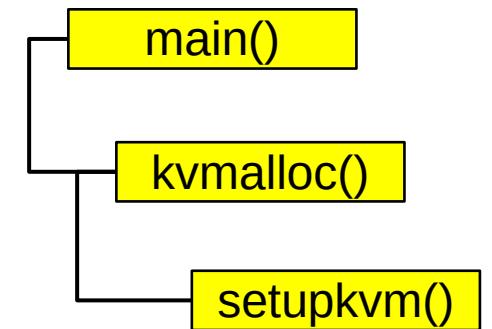
The total physical memory size is 512MB.

```

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

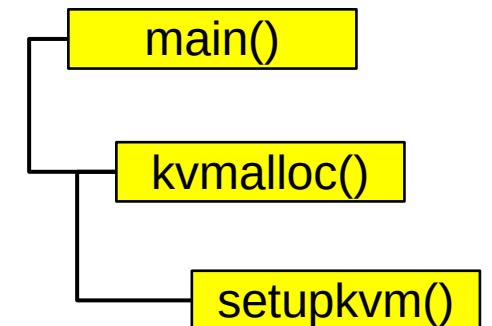


```

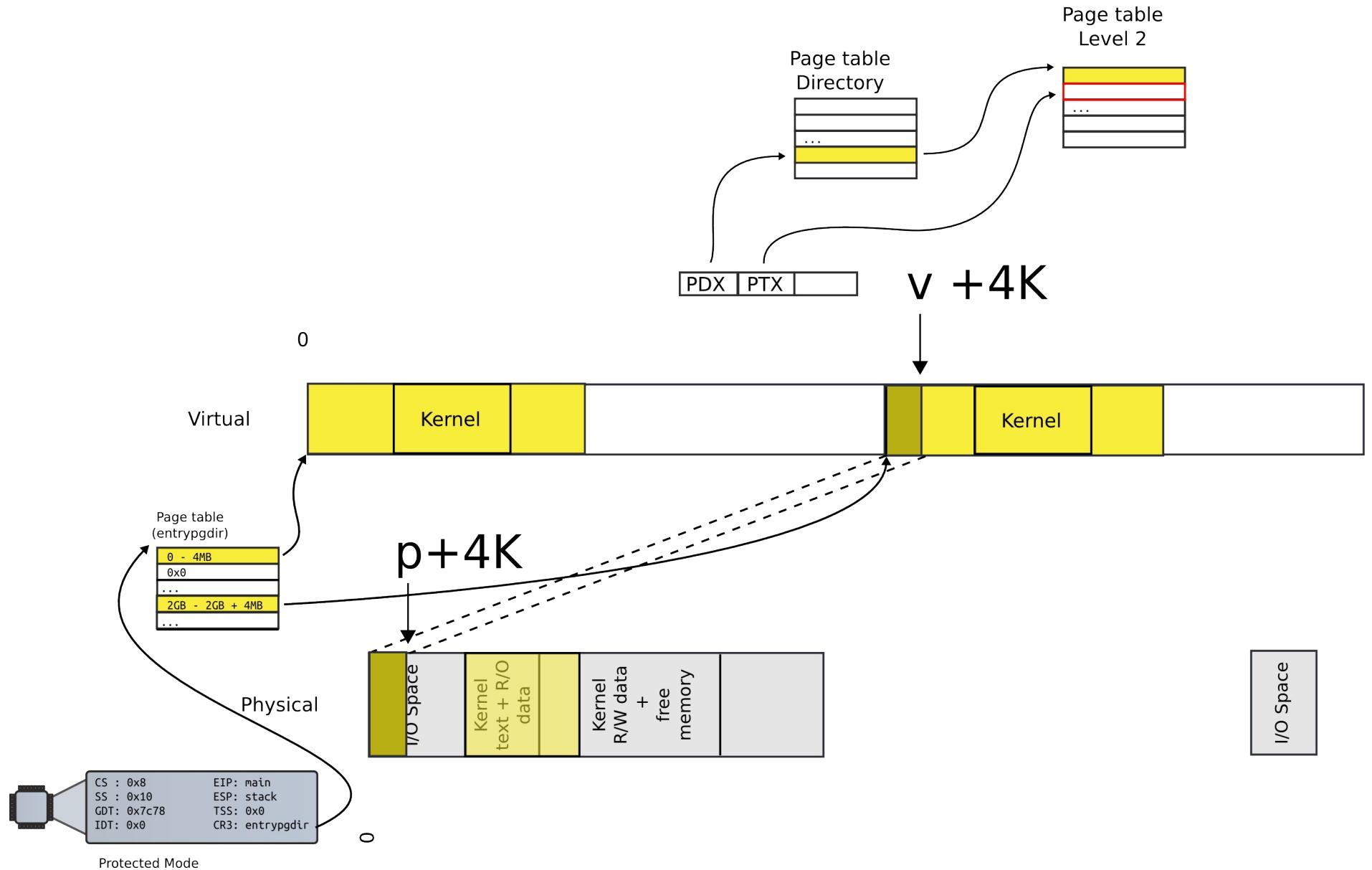
1836 pde_t*
1887 setupkvm(void)
1838 {
1839     pde_t *pgdir;
1840     struct kmap *k;
1841
1842     if((pgdir = (pde_t*)kalloc()) == 0)
1843         return 0;
1844     memset(pgdir, 0, PGSIZE);
...
1847     for(k = kmap; k < &kmap[NELEM(kmap)]; k++)
1848         if(mappages(pgdir, k->virt, k->phys_end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

```

# Recap: Remap physical pages



# setupkvm(): Move to next page



# exec(): Construct process address space

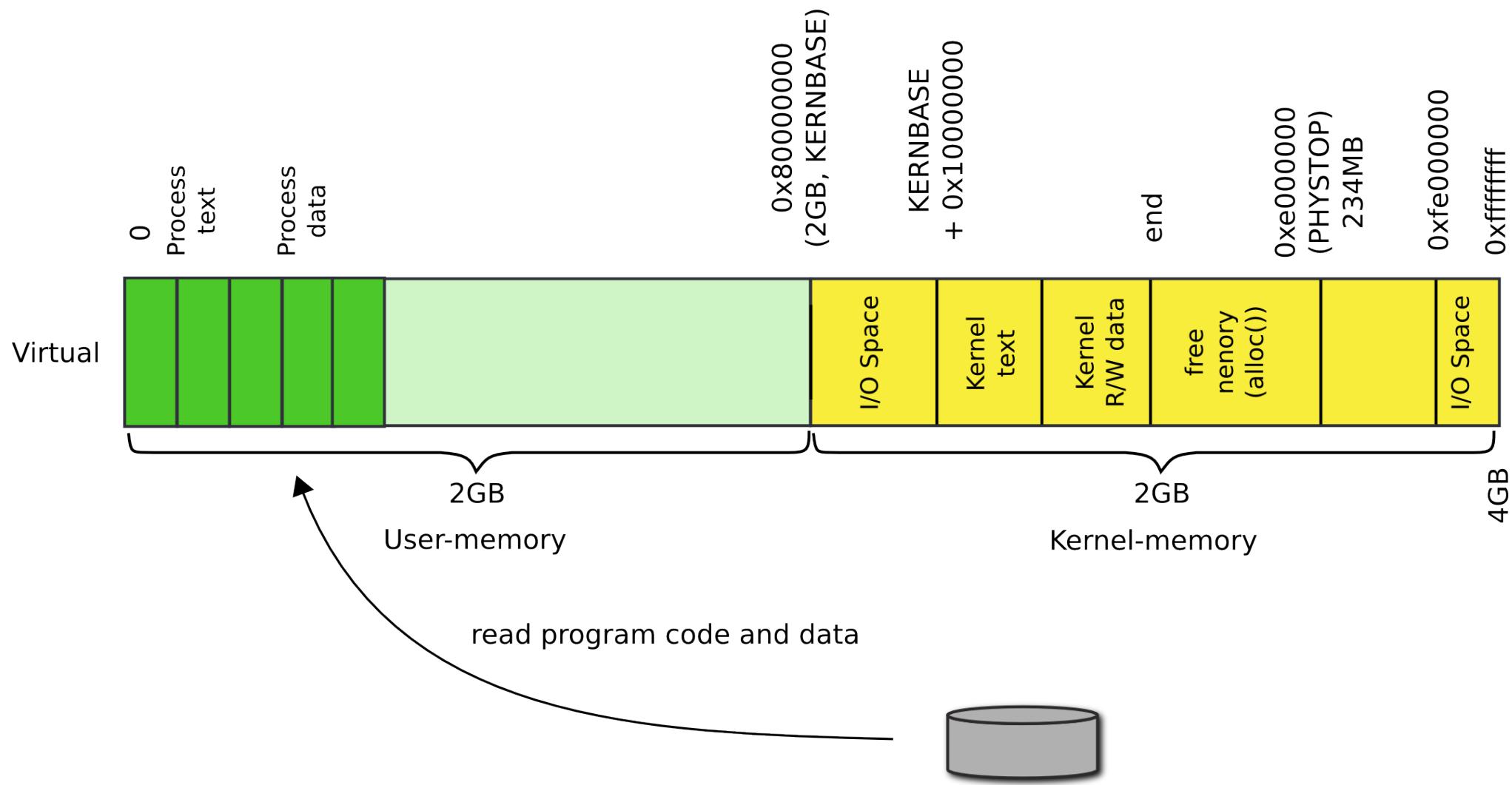
- Two step process
  - Create the kernel part of the address space
  - **Create the user part of the address space**

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



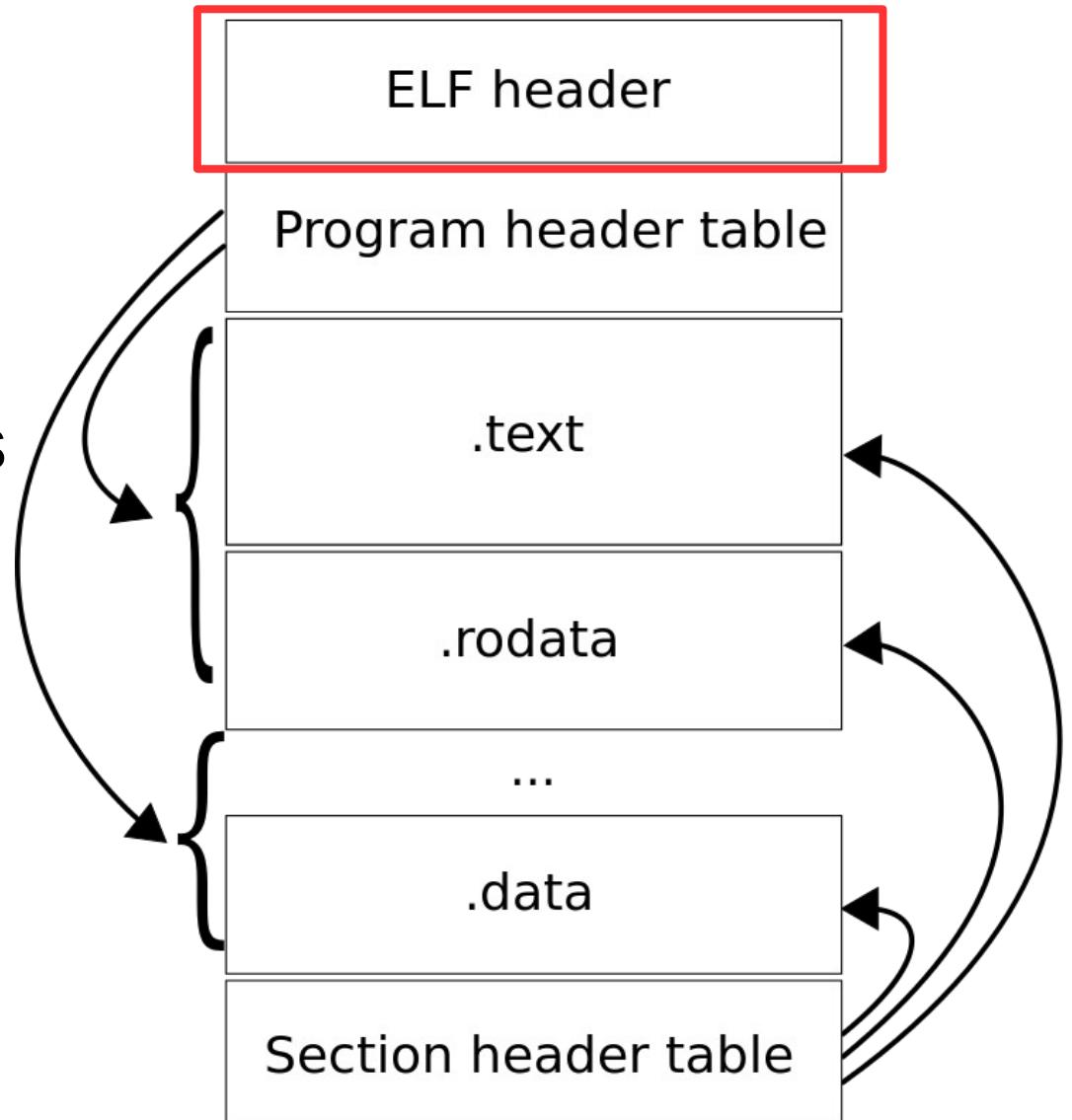
```
6310 exec(char *path, char **argv)
6311 {
...
6337     // Load program into memory.
6338     sz = 0;
6339     for(i=0, off=elf.phoff; i<elf.phnum; i++, off+=sizeof(ph)){
6340         if(readi(ip, (char*)&ph, off, sizeof(ph)) != sizeof(ph))
6341             goto bad;
...
6348     if((sz = allocuvm(pgdir, 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 }
```

## Program loading loop

- Loop over all program headers

# ELF object file

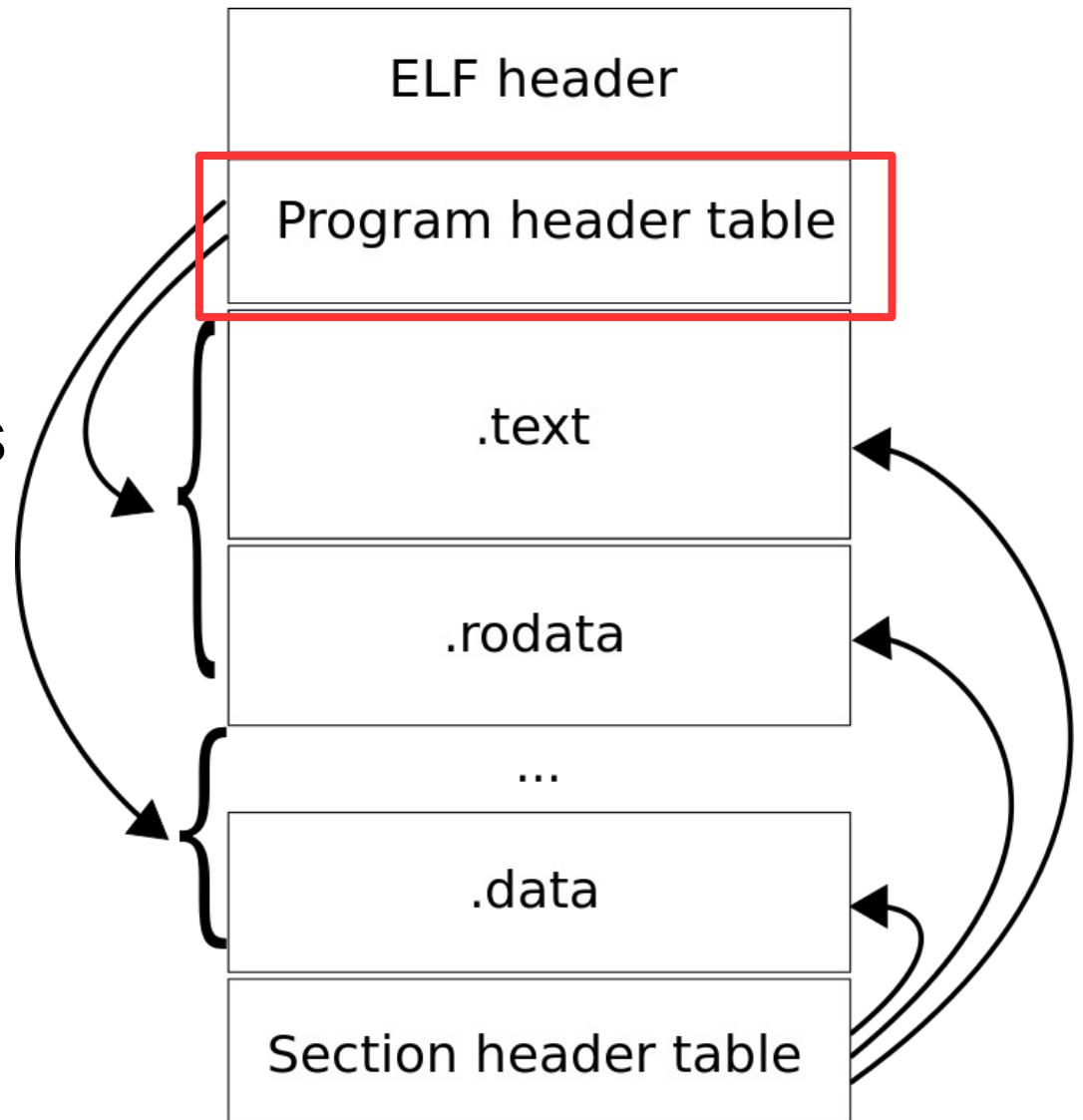
- **ELF header**
- Program header table
  - Each entry describes a section of a program
  - Instruction, data



# ELF object file

- ELF header
- **Program header table**

- Each entry describes a section of a program
- Instruction, data



```
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(pgdир, sz, ph.vaddr + ph.memsz)) == 0)  
6349         goto bad;  
6350     if(ph.vaddr % PGSIZE != 0)  
6351         goto bad;  
6352     if(loaduvm(pgdир, (char*)ph.vaddr, ip, ph.off, ph.filesz)  
< 0)  
6353         goto bad;  
6354 }
```

## Program loading loop

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

```
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(pgdир, sz, ph.vaddr + ph.memsz)) == 0)  
6349         goto bad;  
6350     if(ph.vaddr % PGSIZE != 0)  
6351         goto bad;  
6352     if(loaduvm(pgdир, (char*)ph.vaddr, ip, ph.off, ph.filesz)  
< 0)  
6353         goto bad;  
6354 }
```

## Program loading loop

- Read one program header entry at a time

```
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(pgdир, sz, ph.vaddr + ph.memsz)) == 0)  
6349         goto bad;  
6350     if(ph.vaddr % PGSIZE != 0)  
6351         goto bad;  
6352     if(loaduvm(pgdир, (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(pgdир, sz, ph.vaddr + ph.memsz)) == 0)  
6349         goto bad;  
6350     if(ph.vaddr % PGSIZE != 0)  
6351         goto bad;  
6352     if(loaduvm(pgdир, (char*)ph.vaddr, ip, ph.off, ph.filesz)  
< 0)  
6353         goto bad;  
6354 }
```

## Program loading loop

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

```
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(pgdир, sz, ph.vaddr + ph.memsz)) == 0)  
6349         goto bad;  
6350     if(ph.vaddr % PGSIZE != 0)  
6351         goto bad;  
6352     if(loaduvm(pgdир, (char*)ph.vaddr, ip, ph.off, ph.filesz)  
< 0)  
6353         goto bad;  
6354 }
```

Program loading loop

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

```
6337 // Load program into memory.      Program loading loop
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(pgdир, sz, ph.vaddr + ph.memsz)) == 0)
6349         goto bad;
6350     if(ph.vaddr % PGSIZE != 0)
6351         goto bad;
6352     if(loaduvm(pgdир, (char*)ph.vaddr, ip, ph.off, ph.filesz)
6353 < 0)
6354         goto bad;
6355 }
```

- New size of the address space

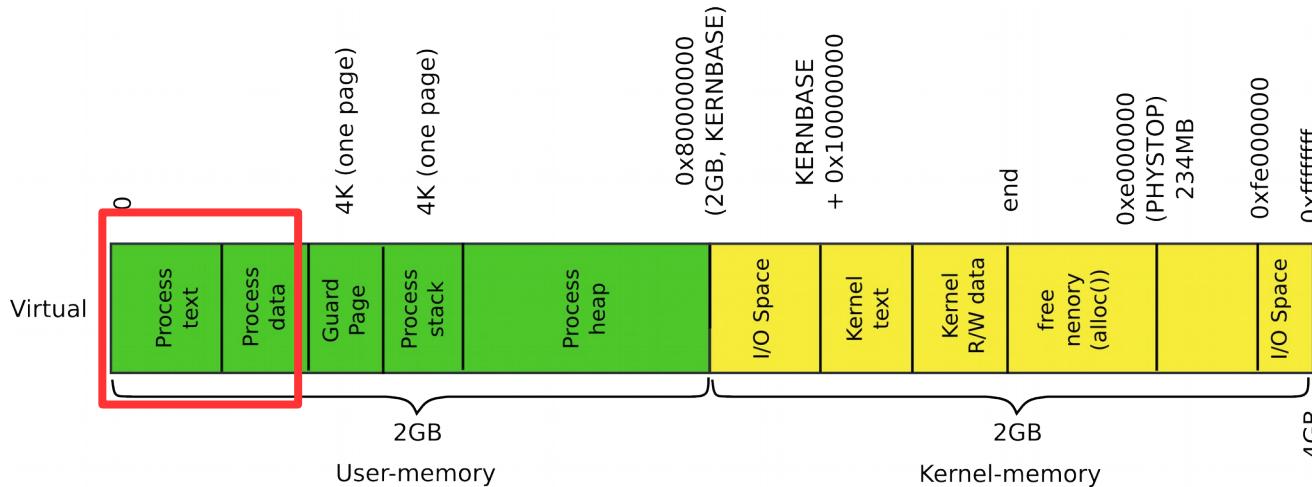
```
6337 // Load program into memory.  
6338 sz = 0;  
  
6339 for(i=0, off=elf.phoff; i<elf.phnum; i++, off+=sizeof(ph)){  
6340     if(readi(ip, (char*)&ph, off, sizeof(ph)) != sizeof(ph))  
6341         goto bad;  
  
...  
  
6348     if((sz = allocuvm(pgdир, sz, ph.vaddr + ph.memsz)) == 0)  
6349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

## Program loading loop

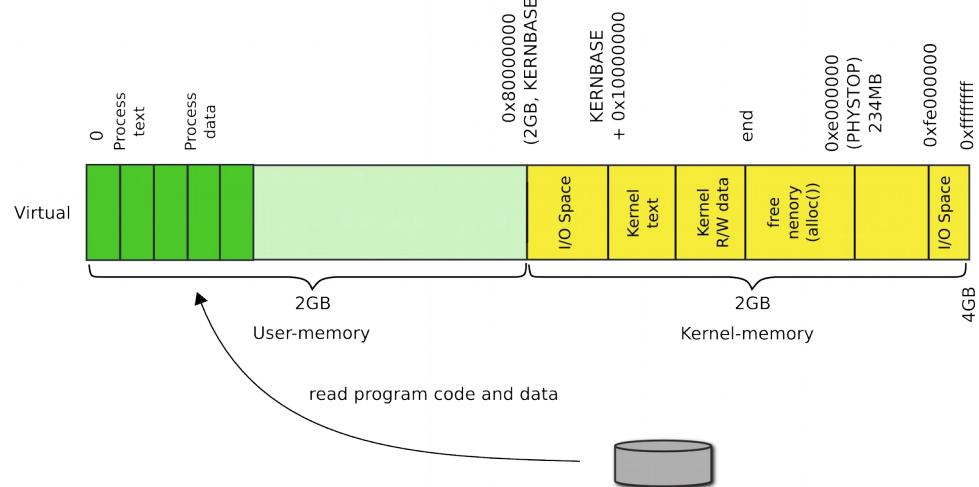
- Load program section from disk

# Two main functions

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

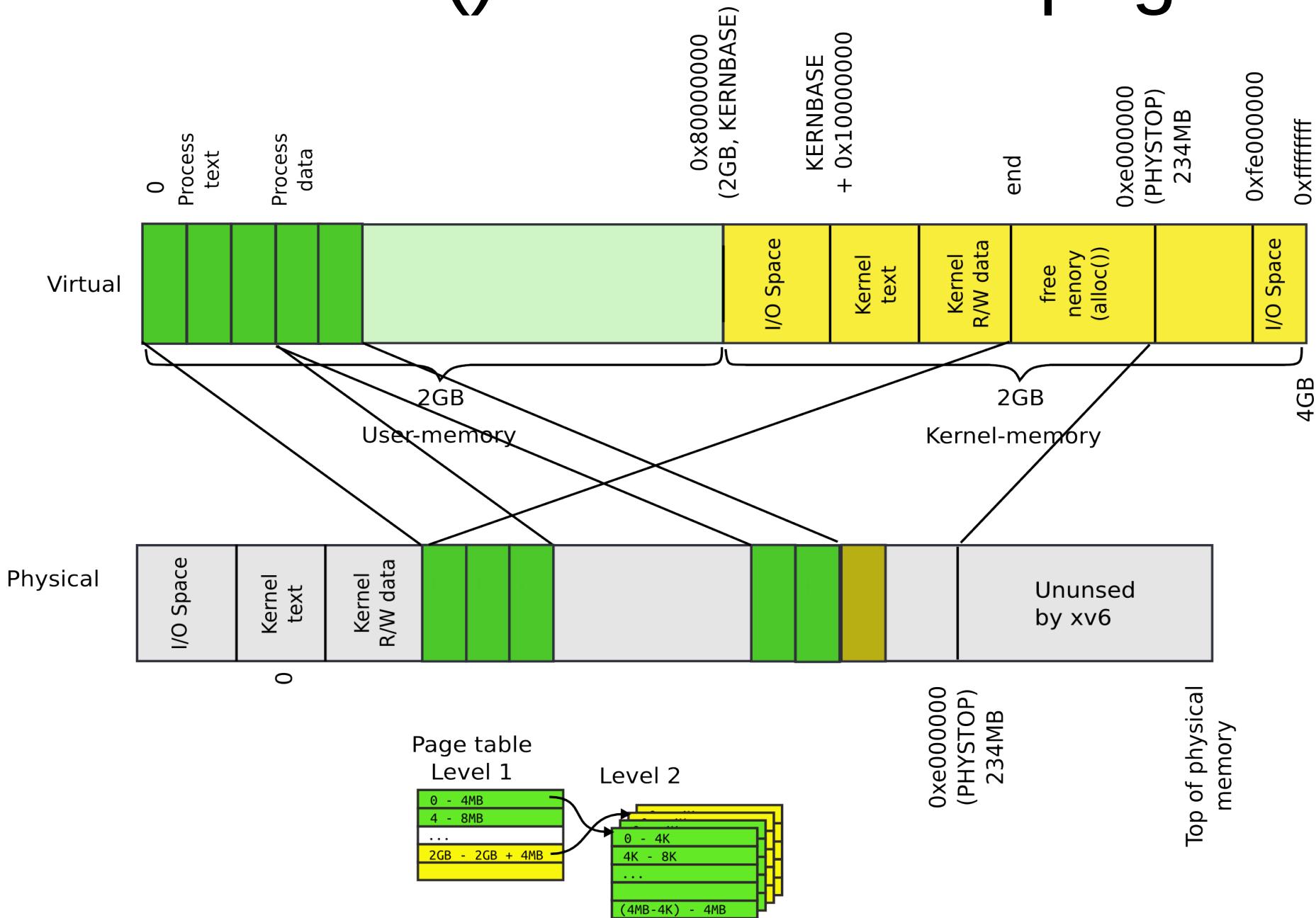


- `loaduvvm()` -- load user-memory with data from disk



Lets take a closer look  
allocuvm()

# allocuvm(): allocate user pages



```
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 }
```

## Allocate user address space

- New size can't be over 2GB

```
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 }
```

## Allocate user address space

- 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 }
```

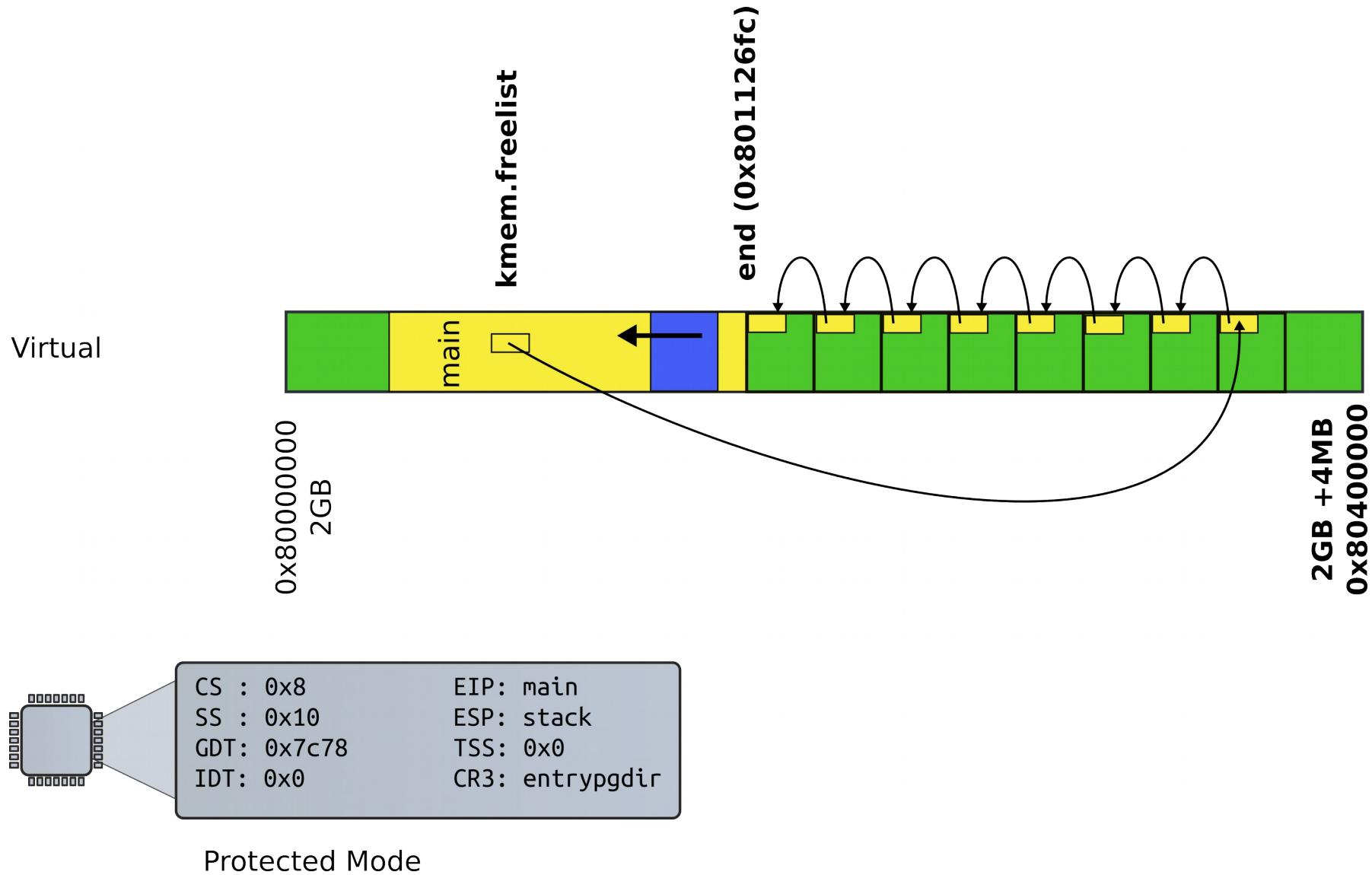
Allocate user address space

- 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



```
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

```
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

- 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

```
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

- 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!

```
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 }
```

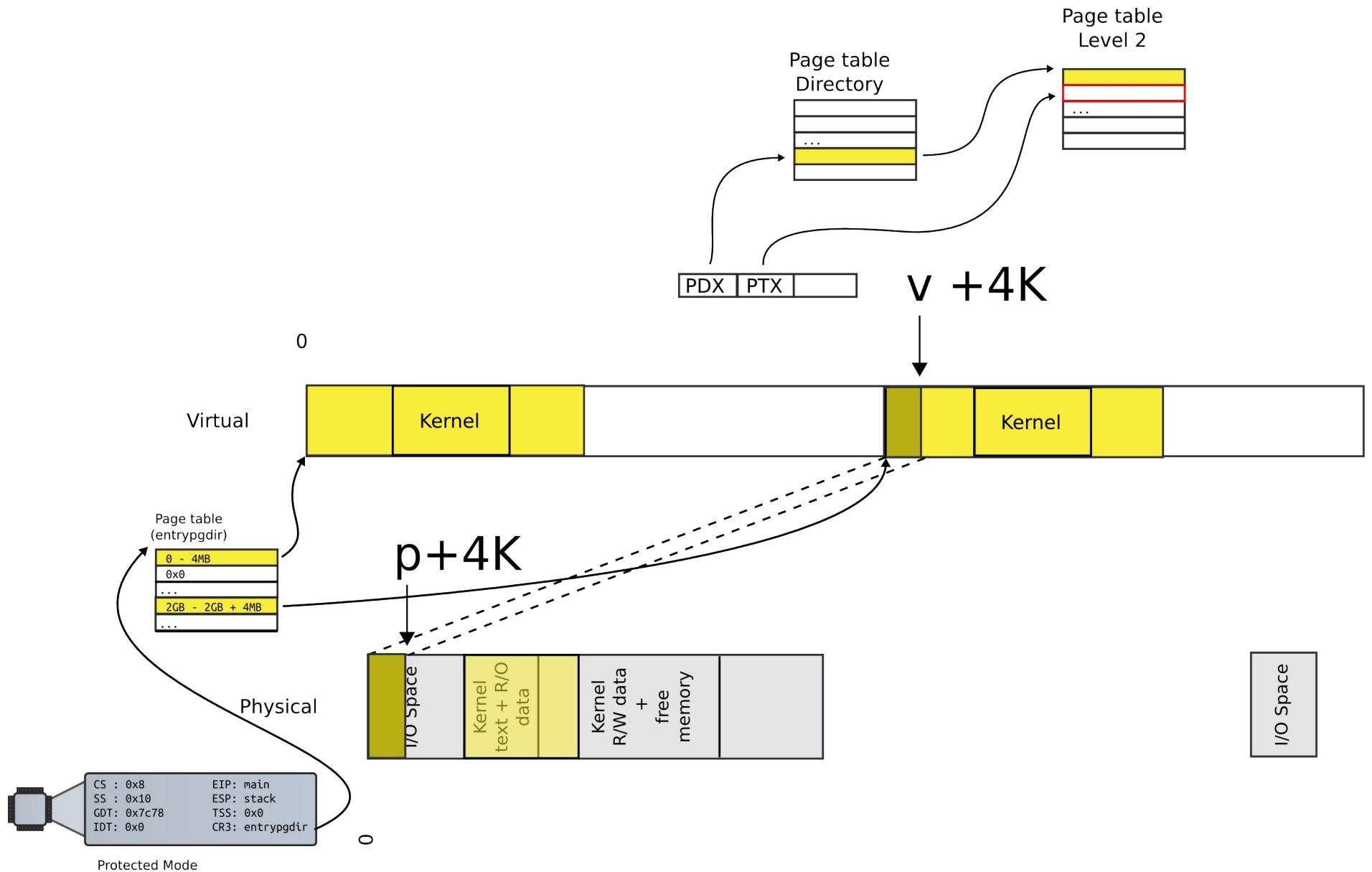
Allocate user address space

- Flags
  - Writable and user-accessible

# Who remembers mappages()?

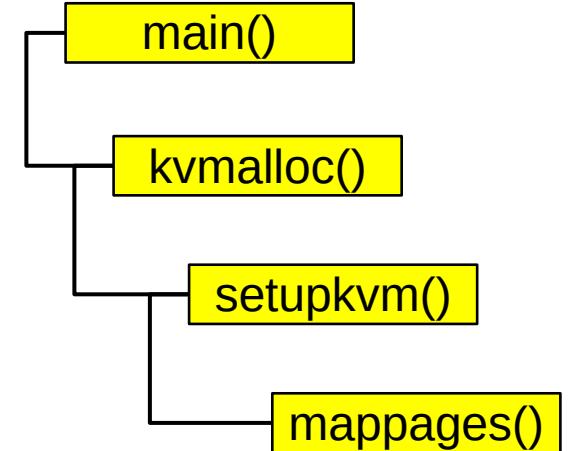
- Remember we want a region of memory to be mapped
  - i.e., appear in the page table

# 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(pgd, 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 }
```

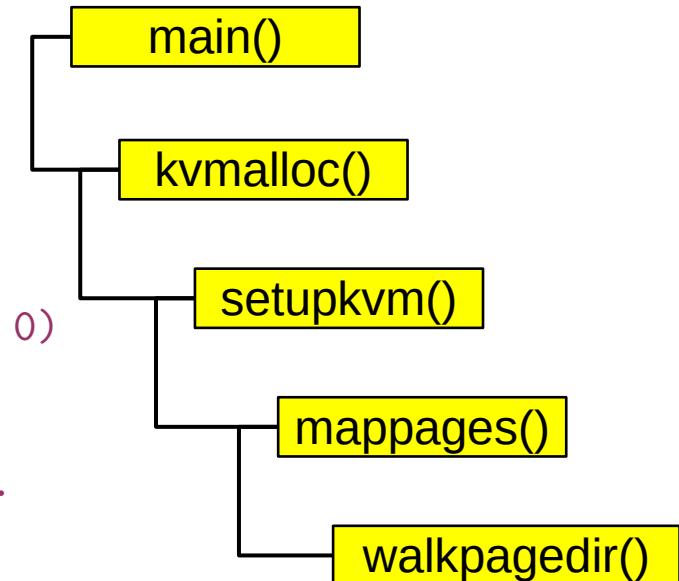


# Lookup the page table entry

```

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



```
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

- Continue in a loop
- Map pages one by one

Now the second function: `loaduvm()`

# exec() – create a new process

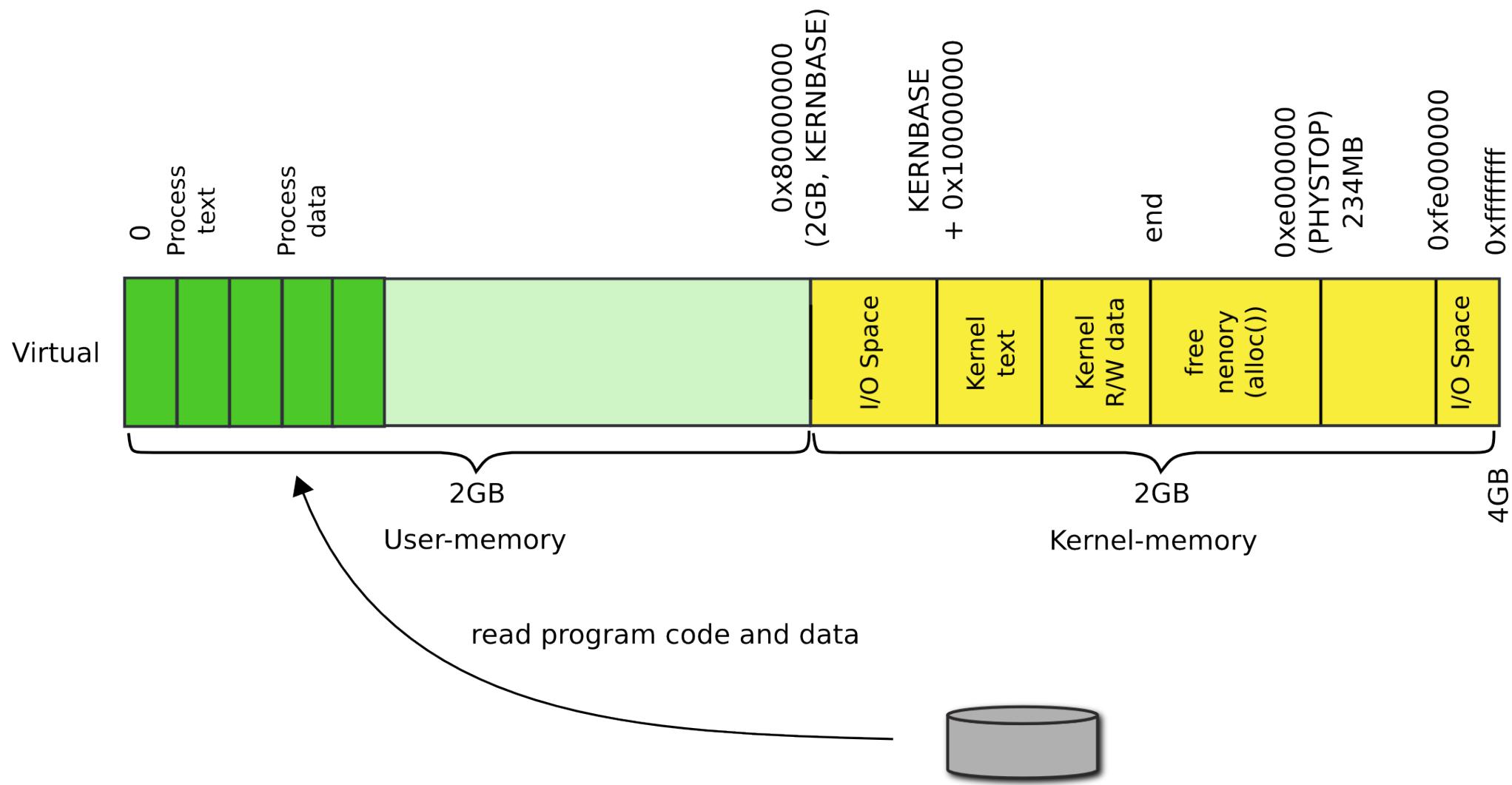
- Read process binary from disk
  - namei() takes a file path (“/bin/ls”) as an argument
  - Returns an inode
  - readi() reads the inode (file data)
- Create process address space
  - Create a page table
  - Map only kernel space
- **Load program into memory**
  - Allocate user-level pages for the program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

```
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(pgdир, sz, ph.vaddr + ph.memsz)) == 0)  
6349         goto bad;  
6350     if(ph.vaddr % PGSIZE != 0)  
6351         goto bad;  
6352     if(loaduvm(pgdир, (char*)ph.vaddr, ip, ph.off, ph.filesz)  
< 0)  
6353         goto bad;  
6354 }
```

## Load program into memory

- 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

```
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

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

- Resolve pte into physical address

```
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

- Then use the virtual address of that physical 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

- Wait... virtual address of 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

- Why can't we use addr directly?

# Drawing: two 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 }
```

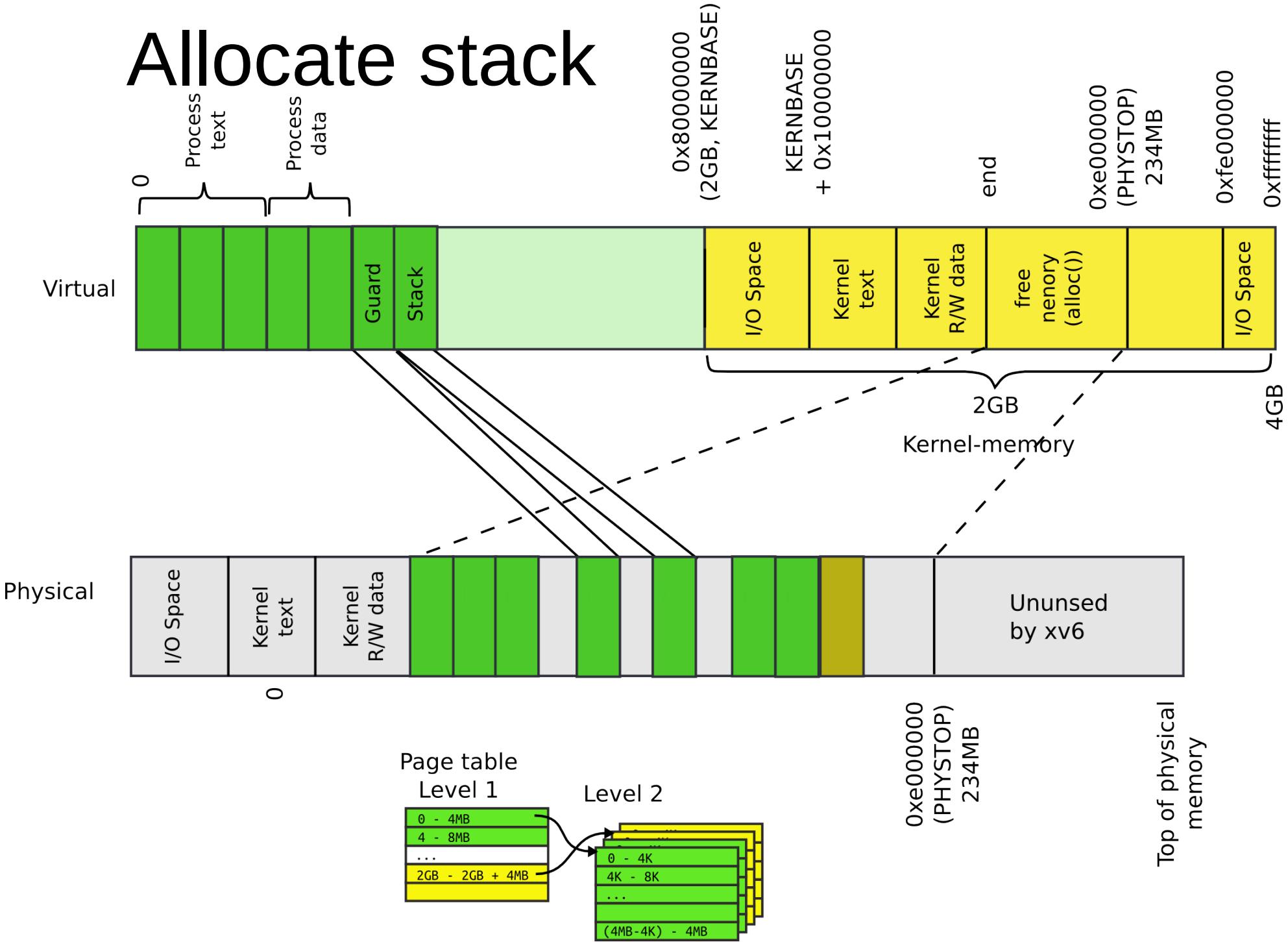
## Load program into memory

- 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

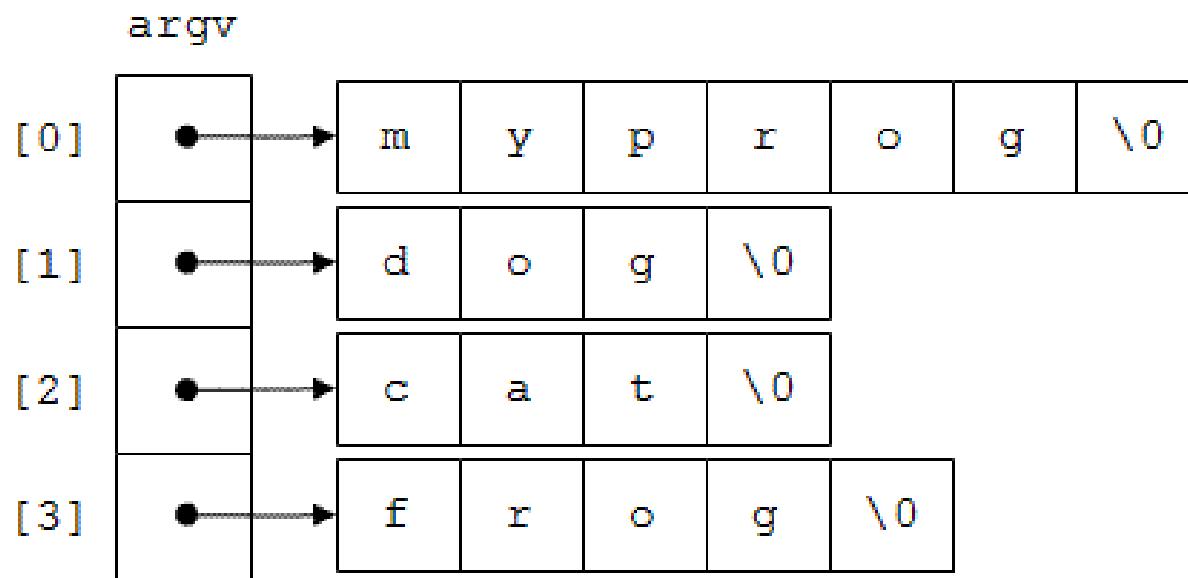
- Read process binary from disk
- Create process address space
- Load program into memory
- Allocate program stack
- **Push program arguments on the stack**

# 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"`.

# Argv and argc

```
z123456@turing:~$ myprog dog cat frog
```



argc

4

# 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(pkdir, 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(pkdir, sp, ustack, (3+argc+1)*4) < 0)  
6384 goto bad;
```

Copy elements of the array one by one on the stack

- Remember we can't use virtual addresses directly

```
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(pkdir, 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(pkdir, 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(pkdir, 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(pkdir, 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

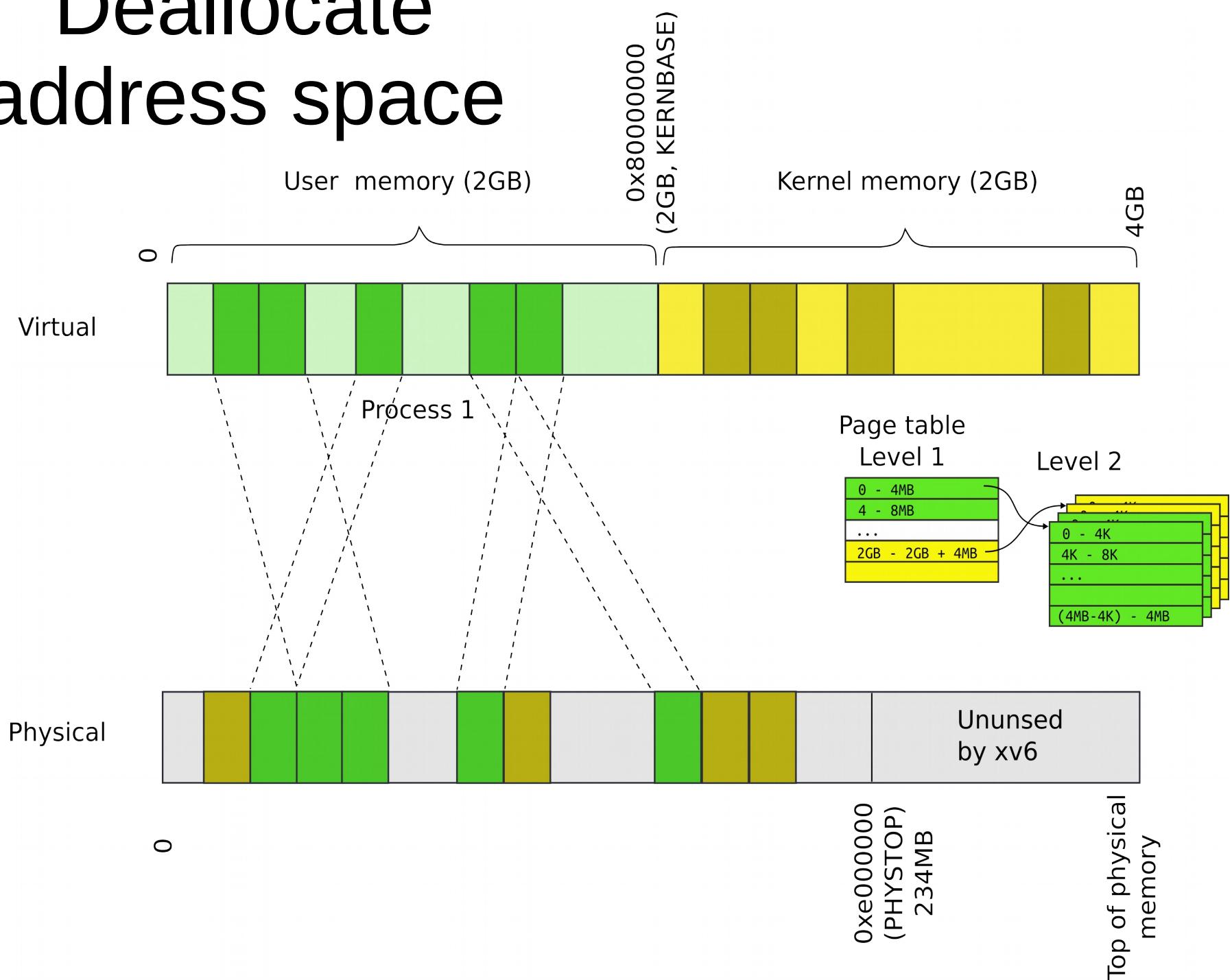
```
6309 int
6310 exec(char *path, char **argv)
6311 {
...
6398     switchuvm(proc);
6399     freevm(oldpgdir);
6400     return 0;
...
6410
```

Wait... which page table we are  
deallocating?

# Wait... which page table we are deallocated?

- 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     deallocuvvm(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 }
```

```
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 < NPENTRIES; 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

# Recap

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

# Creating the first process

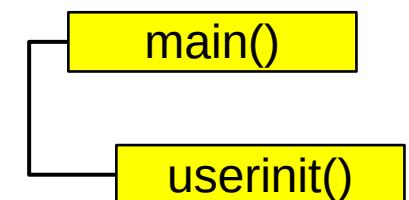
```
1317 main(void)
1318 {
1319     kinit1(end, P2V(4*1024*1024)); // phys page allocator
1320     kvmalloc(); // kernel page table
1321     mpinit(); // detect other processors
...
1323     seginit(); // segment descriptors
...
1330     tvinit(); // trap vectors
...
1338     userinit(); // first user process
1339     mpmain(); // finish this processor's setup
1340 }
```

main()

# Userinit() – create first process

- Allocate process structure
  - Information about the process

```
2502 userinit(void)
2503 {
2504     struct proc *p;
2505     extern char _binary_initcode_start[],
2506             _binary_initcode_size[];
...
2509     p = allocproc();
2510     initproc = p;
2511     if((p->pgdir = setupkvm()) == 0)
2512         panic("userinit: out of memory");
2513     inituvm(p->pgdir, _binary_initcode_start,
2514             (int)_binary_initcode_size);
2515     p->sz = PGSIZE;
2516     memset(p->tf, 0, sizeof(*p->tf));
...
2530 }
```

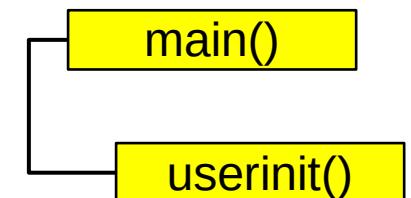


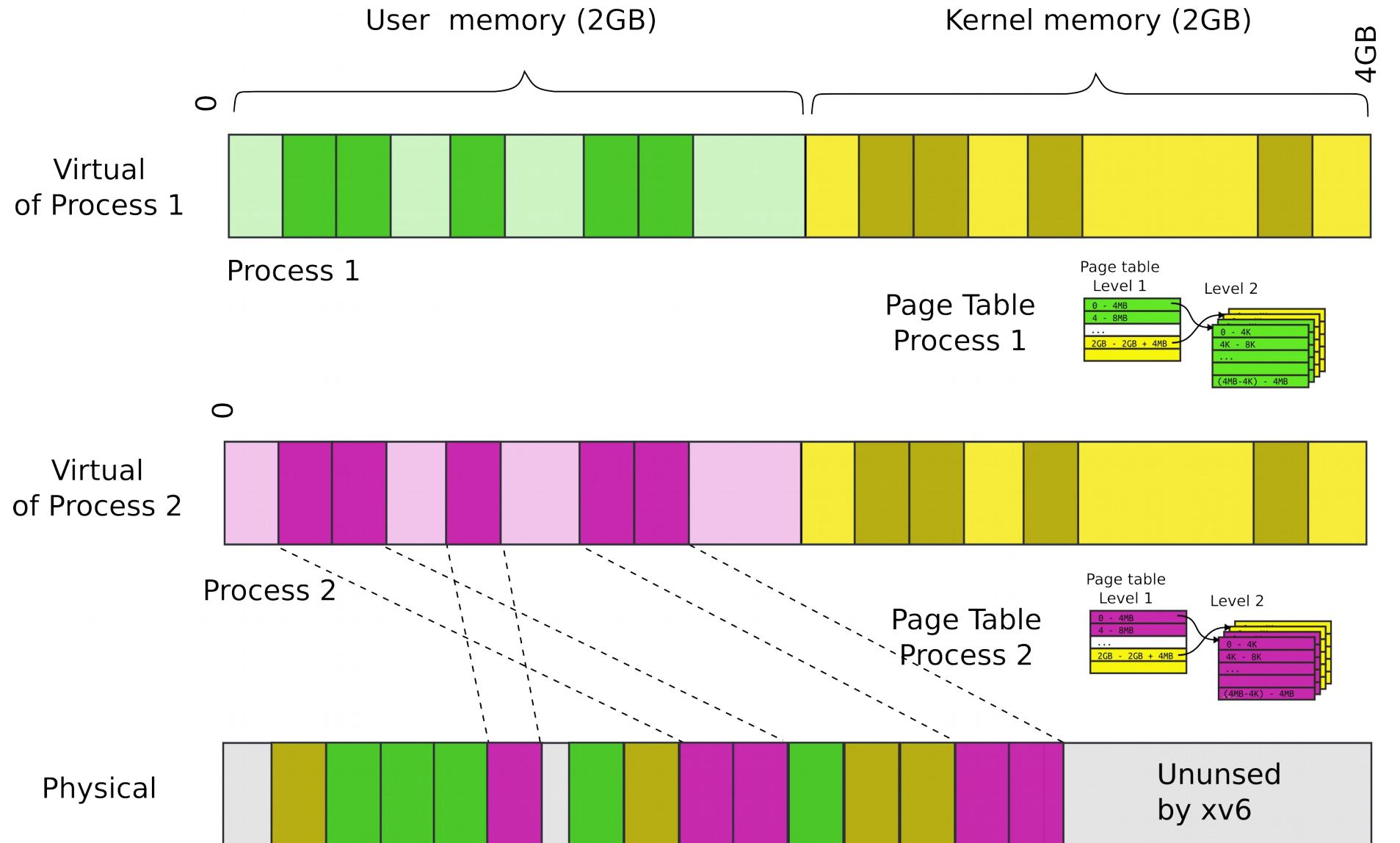
```
2103 struct proc {  
2104     uint sz; // Size of process memory (bytes)  
2105     pde_t* pgdir; // Page table  
2106     char *kstack; // Bottom of kernel stack for this process  
2107     enum procstate state; // Process state  
2108     volatile int pid; // Process ID  
2109     struct proc *parent; // Parent process  
2110     struct trapframe *tf; // Trap frame for current syscall  
2111     struct context *context; // swtch() here to run  
2112     void *chan; // If non-zero, sleeping on chan  
2113     int killed; // If non-zero, have been killed  
2114     struct file *ofile[NFILE]; // Open files  
2115     struct inode *cwd; // Current directory  
2116     char name[16]; // Process name (debugging)  
2117 };
```

# Userinit() – create first process

- Allocate process structure
  - Information about the process
- **Create a page table**
  - Map only kernel space

```
2502 userinit(void)
2503 {
2504     struct proc *p;
2505     extern char _binary_initcode_start[],
2506             _binary_initcode_size[];
...
2509     p = allocproc();
2510     initproc = p;
2511     if((p->pgdir = setupkvm()) == 0)
2512         panic("userinit: out of memory");
2513     inituvm(p->pgdir, _binary_initcode_start,
2514             (int)_binary_initcode_size);
2515     p->sz = PGSIZE;
2516     memset(p->tf, 0, sizeof(*p->tf));
...
2530 }
```





Remember: each process maps kernel in its page table

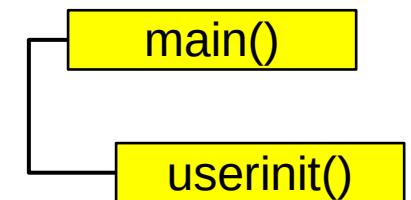
0xe0000000  
(PHYSTOP)  
234MB

Top of physical  
memory

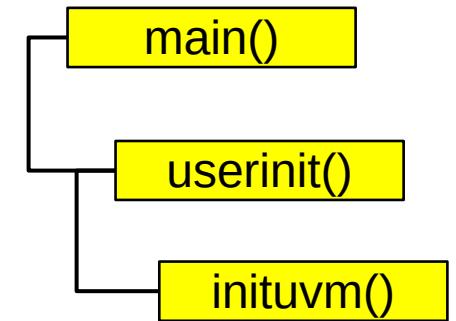
# Userinit() – create first process

- Allocate process structure
  - Information about the process
- Create a page table
  - Map only kernel space
- **Allocate a page for the user init code**
  - **Map this page**

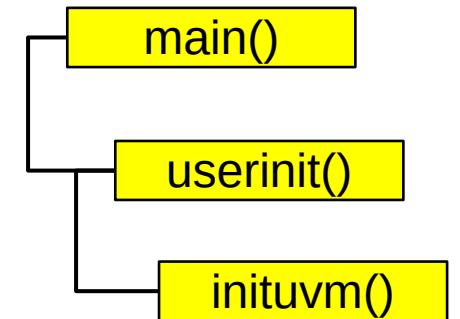
```
2502 userinit(void)
2503 {
2504     struct proc *p;
2505     extern char _binary_initcode_start[],
2506             _binary_initcode_size[];
...
2509     p = allocproc();
2510     initproc = p;
2511     if((p->pgdir = setupkvm()) == 0)
2512         panic("userinit: out of memory?");
2513     inituvm(p->pgdir, _binary_initcode_start,
2514              (int)_binary_initcode_size);
2515     p->sz = PGSIZE;
2516     memset(p->tf, 0, sizeof(*p->tf));
...
2530 }
```



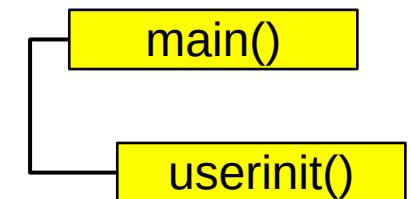
```
1903 inituvm(pde_t *pgdir, char *init, uint sz)
1904 {
1905     char *mem;
1906
1907     if(sz >= PGSIZE)
1908         panic("inituvm: more than a page");
1909     mem = kalloc();
1910     memset(mem, 0, PGSIZE);
1911     mappages(pgdir, 0, PGSIZE, V2P(mem),
1912               PTE_W|PTE_U);
1913 }
```



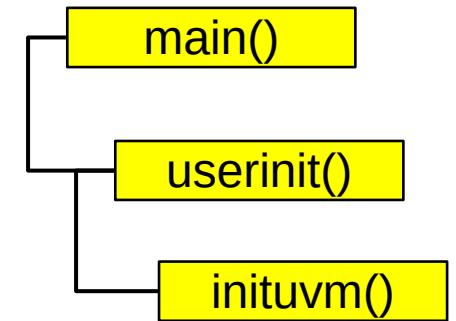
```
1903 inituvm(pde_t *pgdir, char *init, uint sz)
1904 {
1905     char *mem;
1906
1907     if(sz >= PGSIZE)
1908         panic("inituvm: more than a page");
1909     mem = kalloc();
1910     memset(mem, 0, PGSIZE);
1911     mappages(pgdir, 0, PGSIZE, V2P(mem),
1912               PTE_W|PTE_U);
1913 }
```



```
2502 userinit(void)
2503 {
2504     struct proc *p;
2505     extern char _binary_initcode_start[],
2506             _binary_initcode_size[];
...
2509     p = allocproc();
2510     initproc = p;
2511     if((p->pgdir = setupkvm()) == 0)
2512         panic("userinit: out of memory?");
2513     inituvm(p->pgdir, _binary_initcode_start,
2514             (int)_binary_initcode_size);
2515     p->sz = PGSIZE;
2516     memset(p->tf, 0, sizeof(*p->tf));
...
2530 }
```



```
1903 inituvm(pde_t *pgdir, char *init, uint sz)
1904 {
1905     char *mem;
1906
1907     if(sz >= PGSIZE)
1908         panic("inituvm: more than a page");
1909     mem = kalloc();
1910     memset(mem, 0, PGSIZE);
1911     mappages(pgdir, 0, PGSIZE, V2P(mem),
1912               PTE_W|PTE_U);
1913 }
```



```
8409 start:  
8410     pushl $argv  
8411     pushl $init  
8412     pushl $0 // where caller pc would be  
8413     movl $SYS_exec, %eax  
8414     int $T_SYSCALL  
  
8415  
  
...  
  
8422 # char init[] = "/init\0";  
  
8423 init:  
8424     .string "/init\0"  
  
8425  
  
8426 # char *argv[] = { init, 0 };  
8427 .p2align 2  
  
8428 argv:  
8429     .long init  
8430     .long 0
```

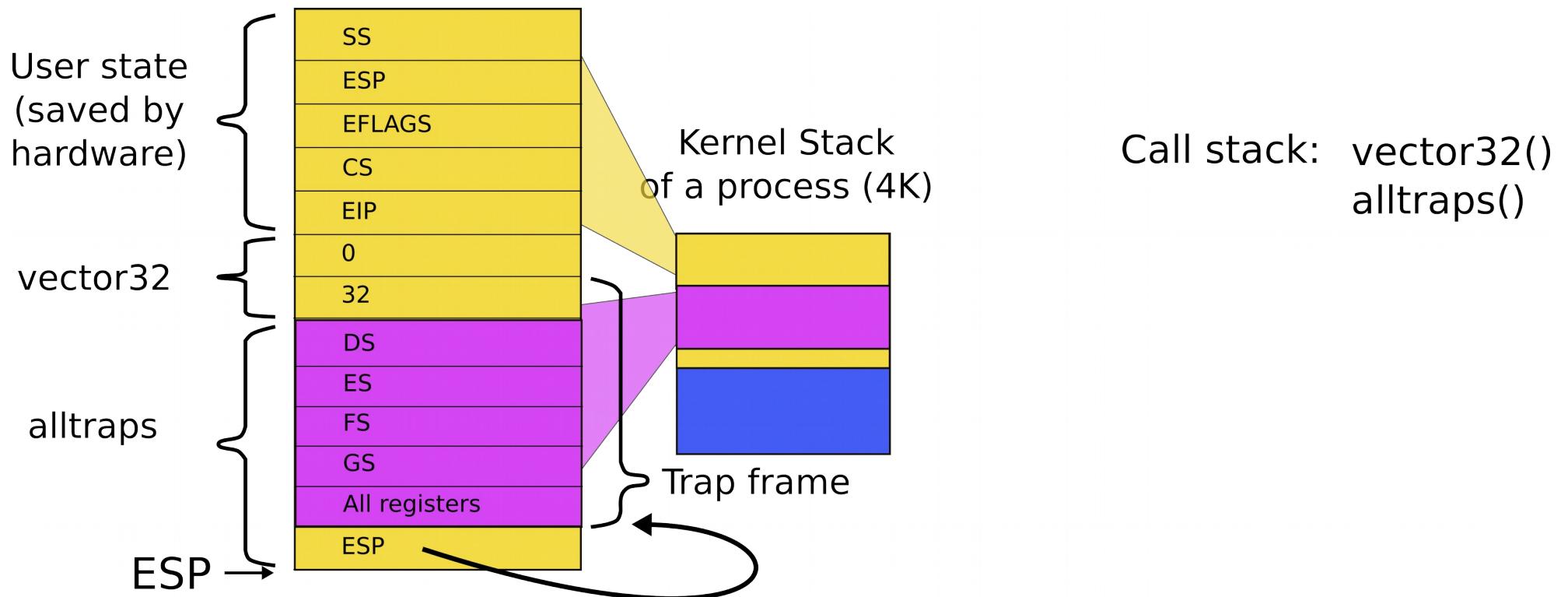
initcode.S: call  
exec("/init", argv);

# userinit() – create first process

- Allocate process structure
  - Information about the process
- Create a page table
  - Map only kernel space
- Allocate a page for the user init code
  - Map this page
- **Configure trap frame for “iret”**

We need to configure the following kernel

- The stack of a process after interrupt/syscall



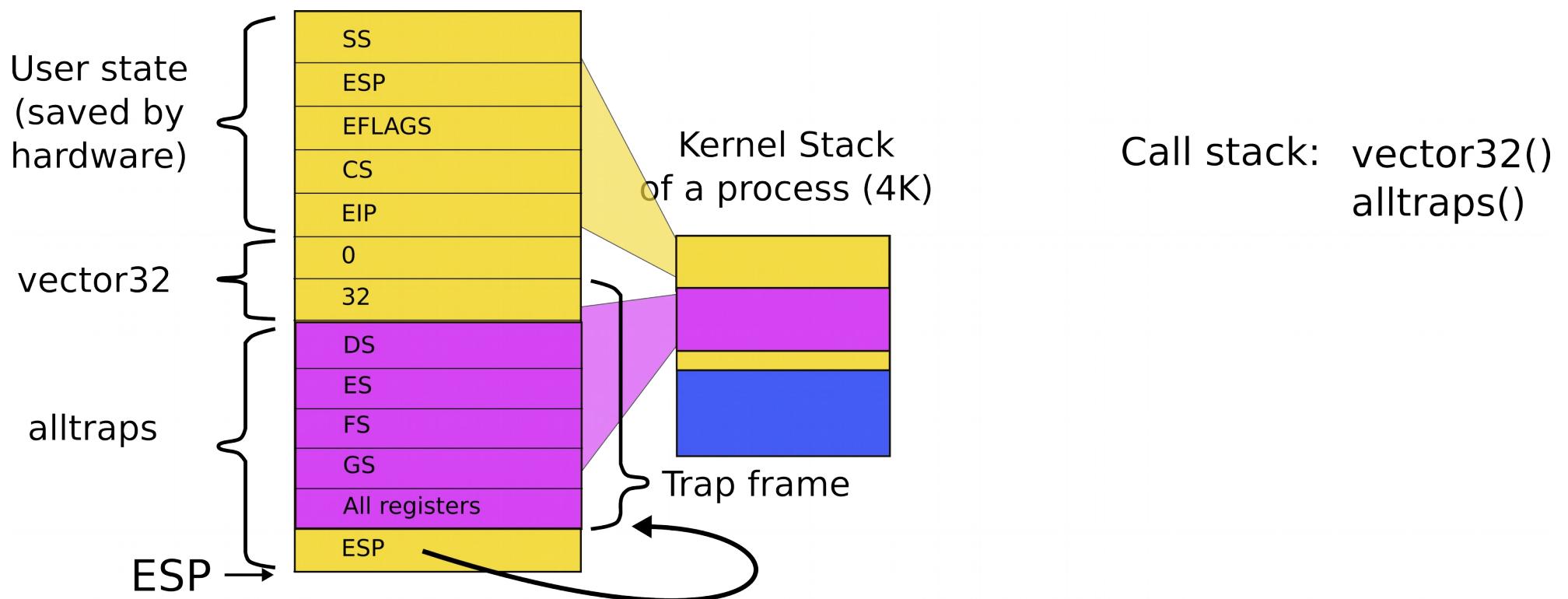
```
2103 struct proc {  
2104     uint sz; // Size of process memory (bytes)  
2105     pde_t* pgdir; // Page table  
2106     char *kstack; // Bottom of kernel stack for this process  
2107     enum procstate state; // Process state  
2108     volatile int pid; // Process ID  
2109     struct proc *parent; // Parent process  
2110     struct trapframe *tf; // Trap frame  
2111     struct context *context; // swtch() here to run  
2112     void *chan; // If non-zero, sleeping on chan  
2113     int killed; // If non-zero, have been killed  
2114     struct file *ofile[NFILE]; // Open files  
2115     struct inode *cwd; // Current directory  
2116     char name[16]; // Process name (debugging)  
2117 };
```

```
2456 allocproc(void)
2457 {
...
2470     // Allocate kernel stack.
2471     if((p->kstack = kalloc()) == 0){
2472         p->state = UNUSED;
2473         return 0;
2474     }
2475     sp = p->kstack + KSTACKSIZE;
2476
2477     // Leave room for trap frame.
2478     sp -= sizeof *p->tf;
2479     p->tf = (struct trapframe*)sp;
2480
...
2492 }
```

Trap frame is on the  
kernel stack of the process

```
2502 userinit(void)
2503 {
...
2513     inituvm(p->pgdir, _binary_initcode_start,
2514             (int)_binary_initcode_size);
2515     p->sz = PGSIZE;
2516     memset(p->tf, 0, sizeof(*p->tf));
2517     p->tf->cs = (SEG_UCODE << 3) | DPL_USER;
2518     p->tf->ds = p->tf->ds;
2519     p->tf->ss = p->tf->ds;
2520     p->tf->eflags = FL_IF;
2521     p->tf->esp = PGSIZE;
2522     p->tf->eip = 0; // beginning of initcode.S
...
2530 }
```

# Kernel stack after interrupt/syscall



```
2502 userinit(void)
2503 {
...
2515     memset(p->tf, 0, sizeof(*p->tf));
2516     p->tf->cs = (SEG_UCODE << 3) | DPL_USER;
2517     p->tf->ds = (SEG_UDATA << 3) | DPL_USER;
2518     p->tf->es = p->tf->ds;
2519     p->tf->ss = p->tf->ds;
2520     p->tf->eflags = FL_IF;
2521     p->tf->esp = PGSIZE;
2522     p->tf->eip = 0; // beginning of initcode.S
2523
2524     safestrcpy(p->name, "initcode", sizeof(p->name));
2525     p->cwd = namei("/");
2526
2527     p->state = RUNNABLE;
...
2530 }
```

Wait, we mapped process memory, created trap frame, but it doesn't really run...

```
8510 main(void)
8511 {
...
8514     if(open("console", 0_RDWR) < 0){
8515         mknod("console", 1, 1);
8516         open("console", 0_RDWR);
8517     }
8518     dup(0); // stdout
8519     dup(0); // stderr
8520
8521     for(; ;){
8522         printf(1, "init: starting sh\n");
8523         pid = fork();
8524         if(pid < 0){
8525             printf(1, "init: fork failed\n");
8526             exit();
8527         }
8528         if(pid == 0){
8529             exec("sh", argv);
8530             printf(1, "init: exec sh failed\n");
8531             exit();
8532         }
8533         while((wpid=wait()) >= 0 && wpid != pid)
8534             printf(1, "zombie!\n");
8535     }
8536 }
```

- First process **exec("init")**
- /init starts /sh
  - fork() and **exec("sh")**

# Summary

- We've finally learned how the first process came to life

Also we know:

- How OS boots and initializes itself
- How each process is constructed (`exec()`)
- How OS switches between processes

Thank you!