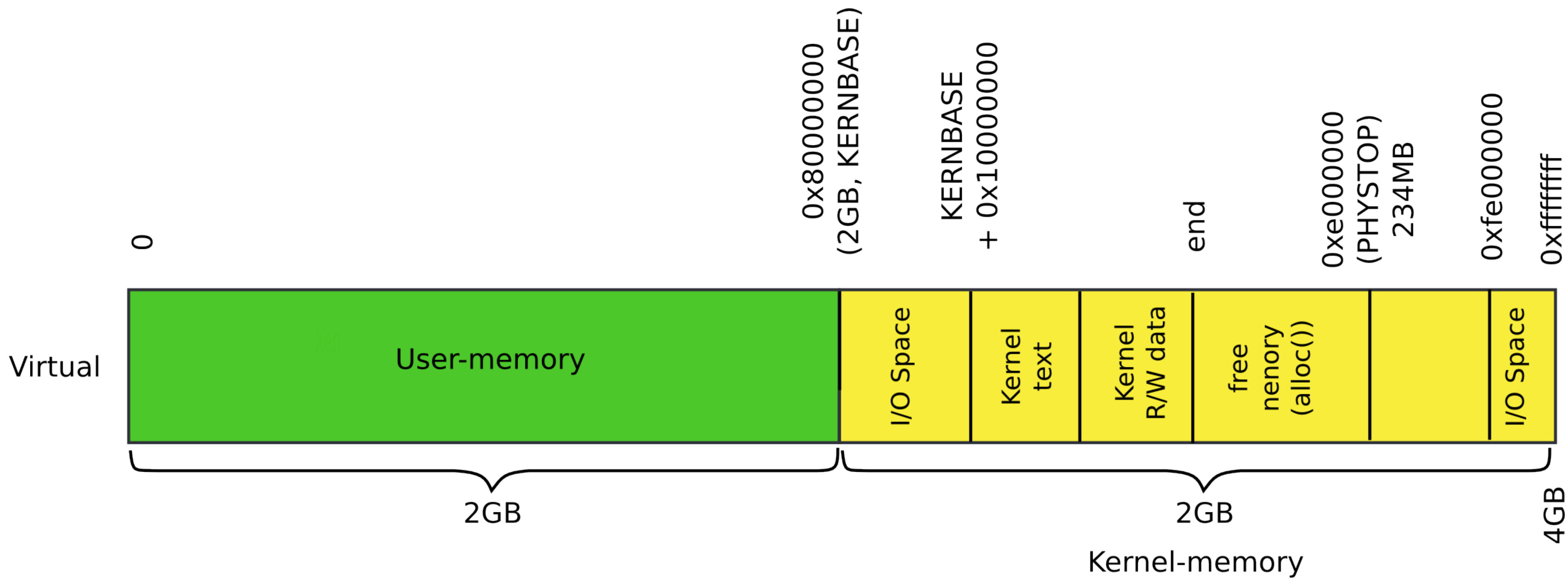
# Operating Systems

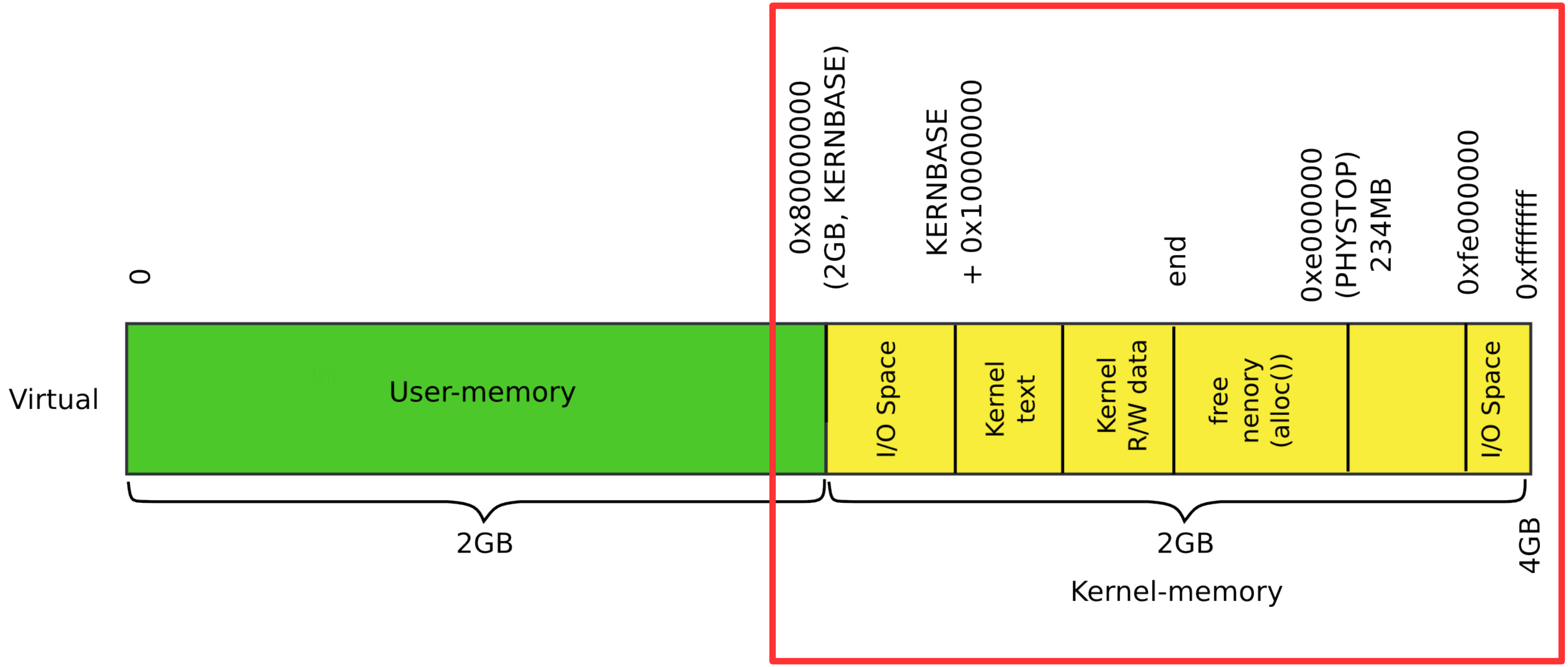
# Lecture: Creating Processes (`exec()`)

Anton Burtsev

# Recap: kernel memory



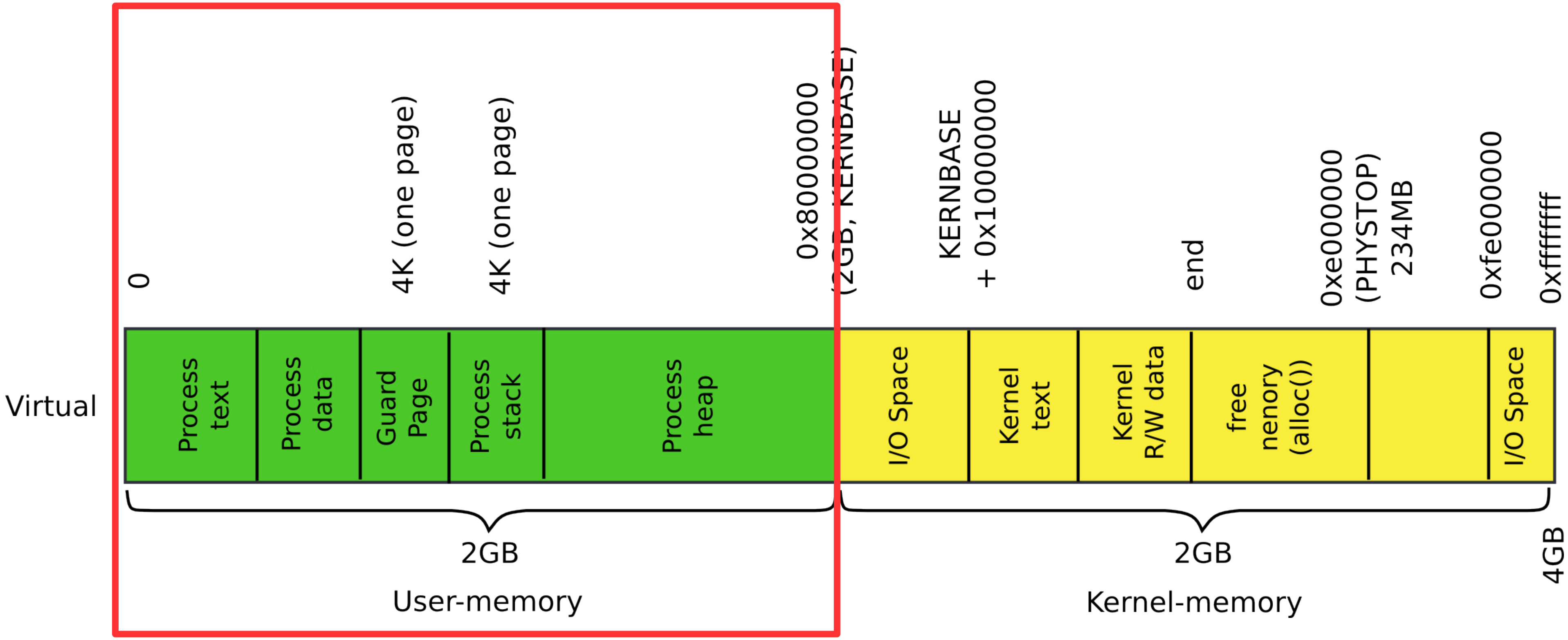
# Recap: kernel memory



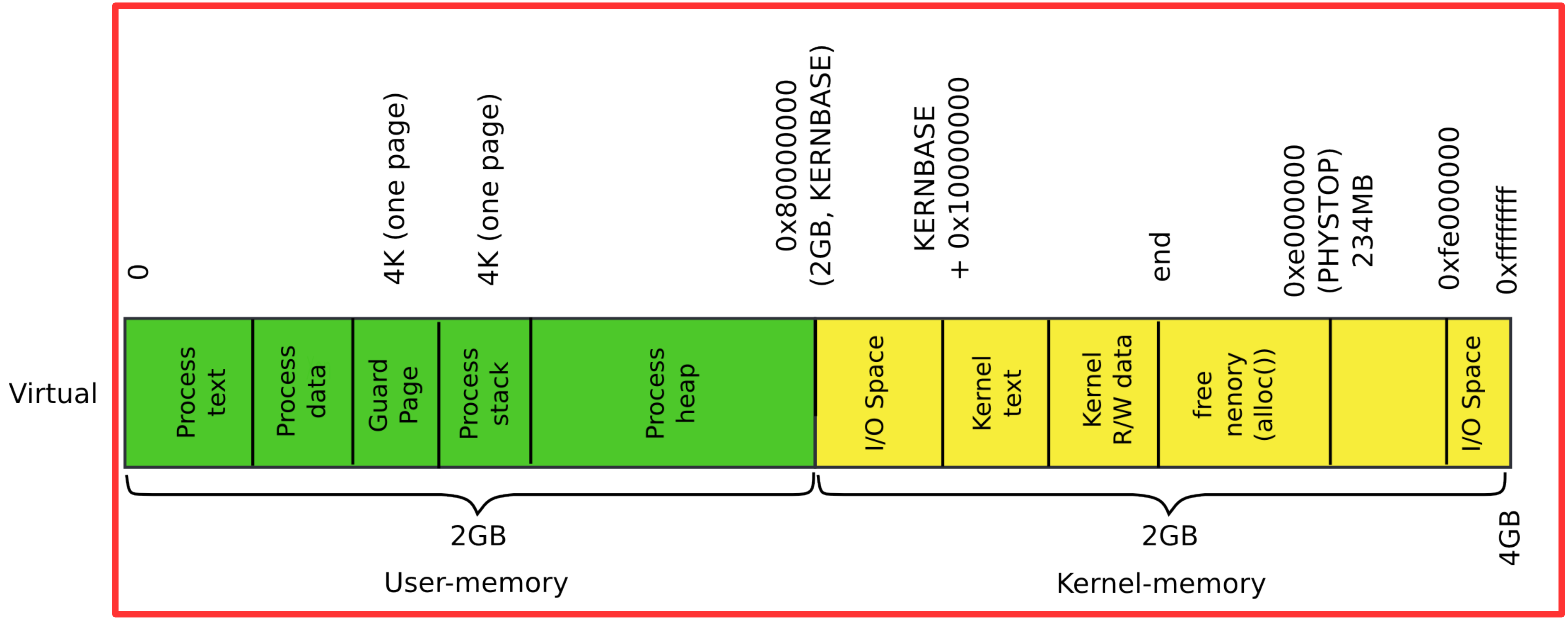
```c
main(void)
{
  kinit1(end, P2V(4*1024*1024)); // phys page allocator
  kvmalloc(); // kernel page table
  mpinit(); // detect other processors
  lapicinit();     // interrupt controller
  seginit();       // segment descriptors
  cprintf("\ncpu%d: starting xv6\n\n", cpunum());
  picinit();       // another interrupt controller
  ioapicinit();    // another interrupt controller
  consoleinit();   // console hardware
  uartinit();      // serial port
  pinit();         // process table
  tvinit();        // trap vectors
  binit();         // buffer cache
  fileinit();      // file table
  ideinit();       // disk
  if(!ismp)
    timerinit();   // uniprocessor timer
  startothers();   // start other processors
  kinit2(P2V(4*1024*1024), P2V(PHYSTOP)); // must come after startothers()
  userinit();      // first user process
  mpmain();        // finish this processor's setup
}
```

main()

# Today: process memory



# Today: process memory



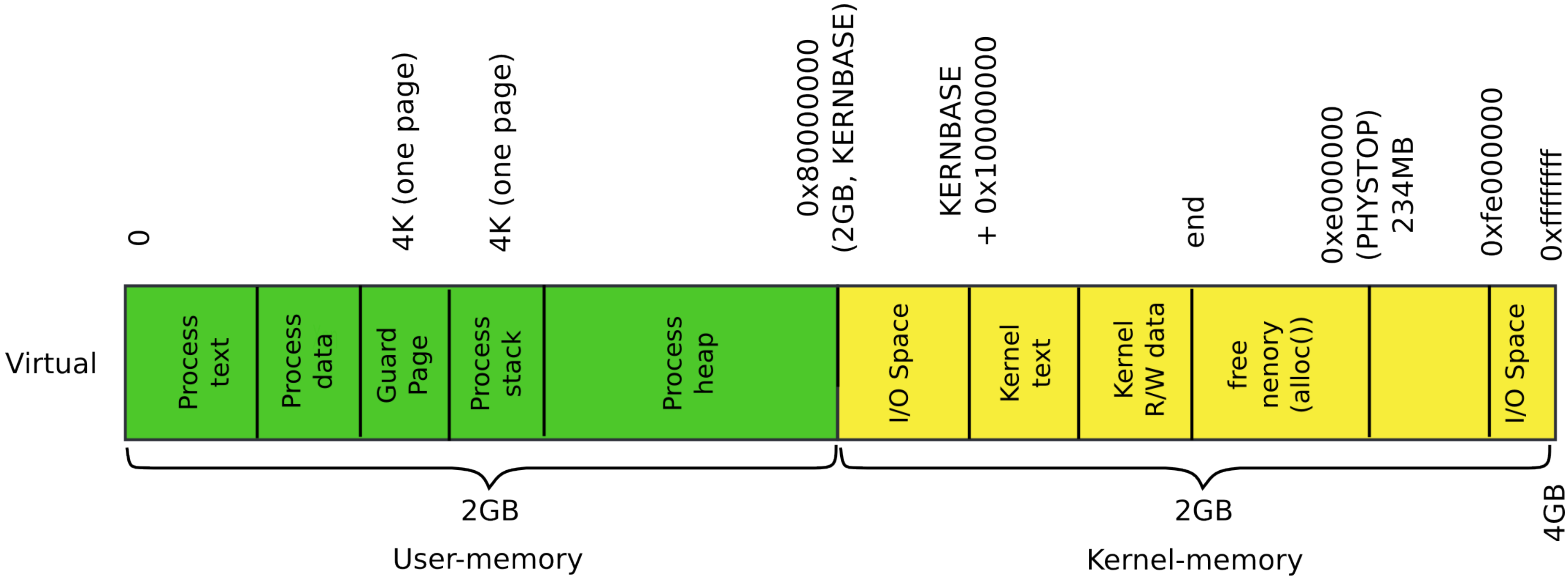
# How does kernel creates new processes?

# How does kernel creates new processes?

- Exec
  - exec(“/bin/ls”, argv);

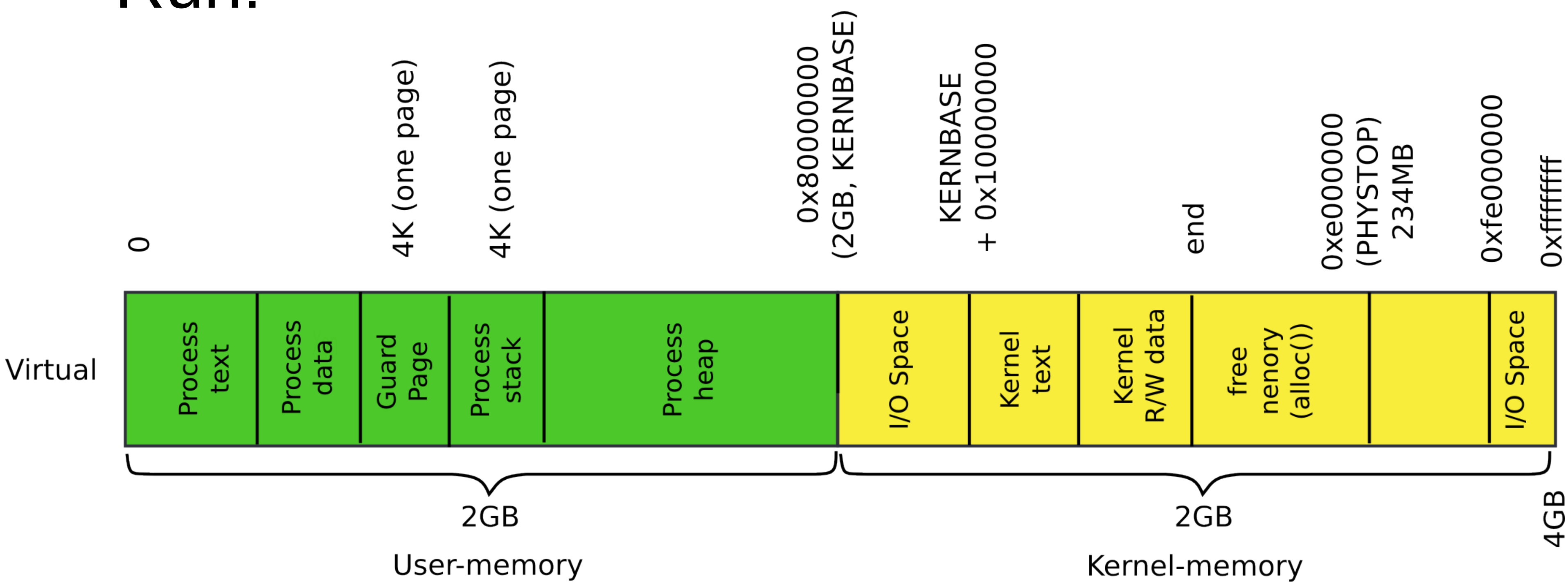
# exec(): high-level outline

- 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

```
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;
```

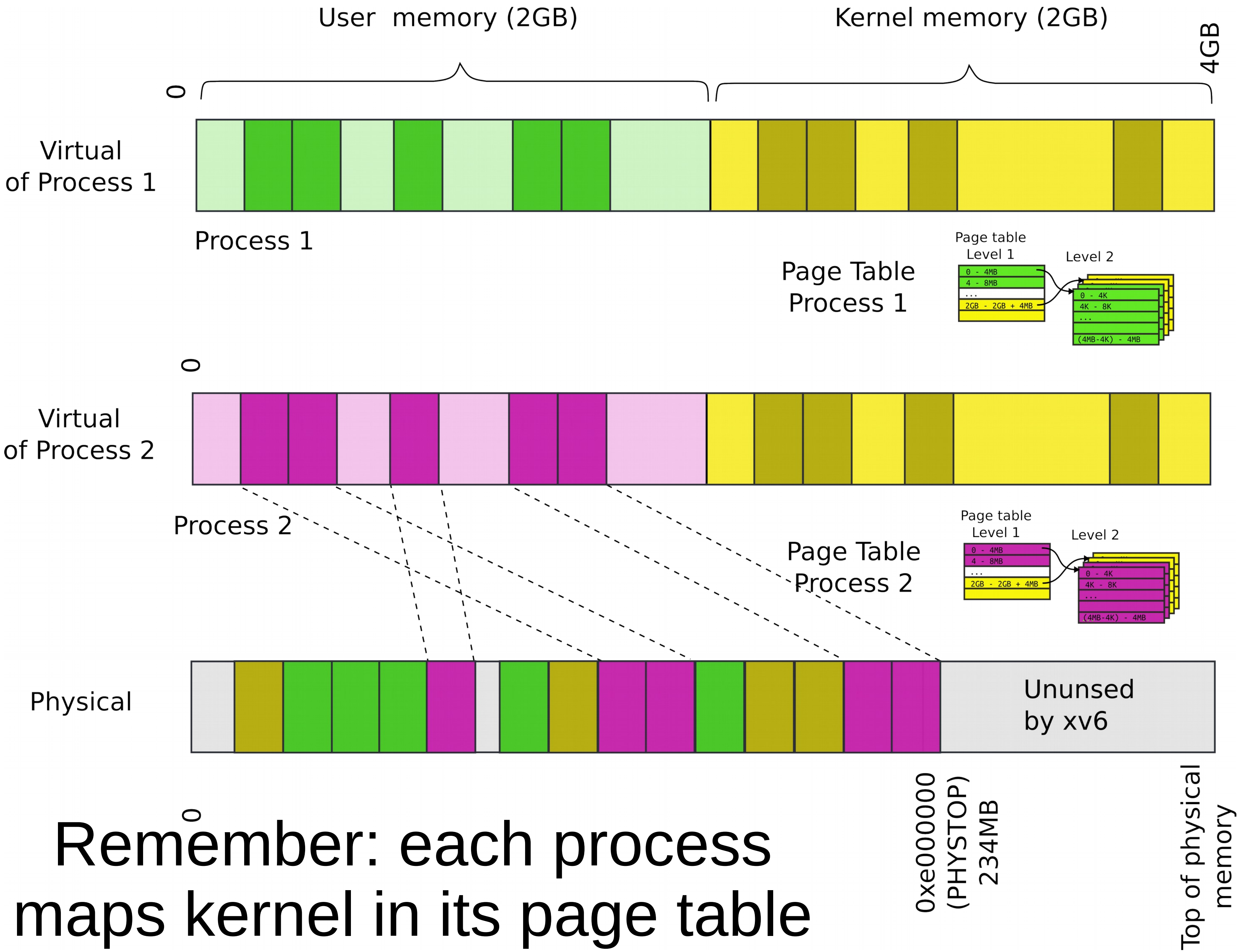
# exec(): check ELF header

```
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;
```

# 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



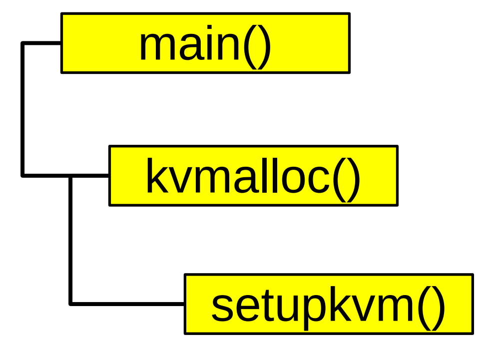

# Remember: each process maps kernel in its page table

# exec(): Setup kernel address space()

```
exec(char *path, char **argv)
{
...
  if(elf.magic != ELF_MAGIC)
    goto bad;
  if((pgdir = setupkvm()) == 0)
    goto bad;
...
```

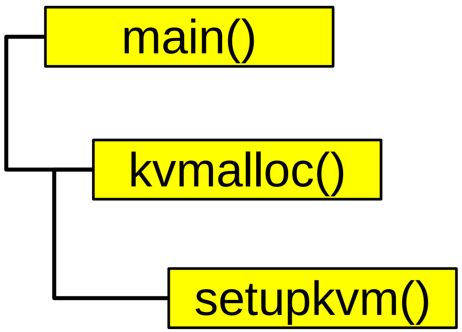
- Remember from the kernel memory allocator lecture?

# Recap: Allocate page table directory



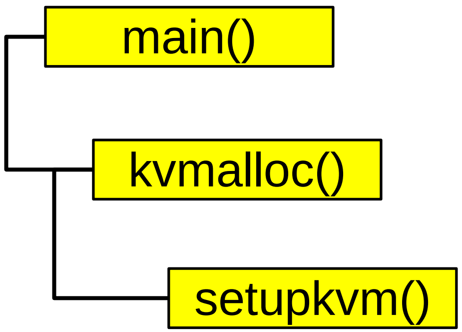

```
pde_t*
setupkvm(void)
{
  pde_t *pgdir;
  struct kmap *k;

  if((pgdir = (pde_t*)kalloc()) == 0)
    return 0;
  memset(pgdir, 0, PGSIZE);
...
  for(k = kmap; k < &kmap[NELEM(kmap)]; k++)
    if(mappages(pgdir, k->virt, k->phys_end - k->phys_start,
                (uint)k->phys_start, k->perm) < 0)
      return 0;
  return pgdir;
}
```

# Recap: Iterate in a loop: remap physical pages



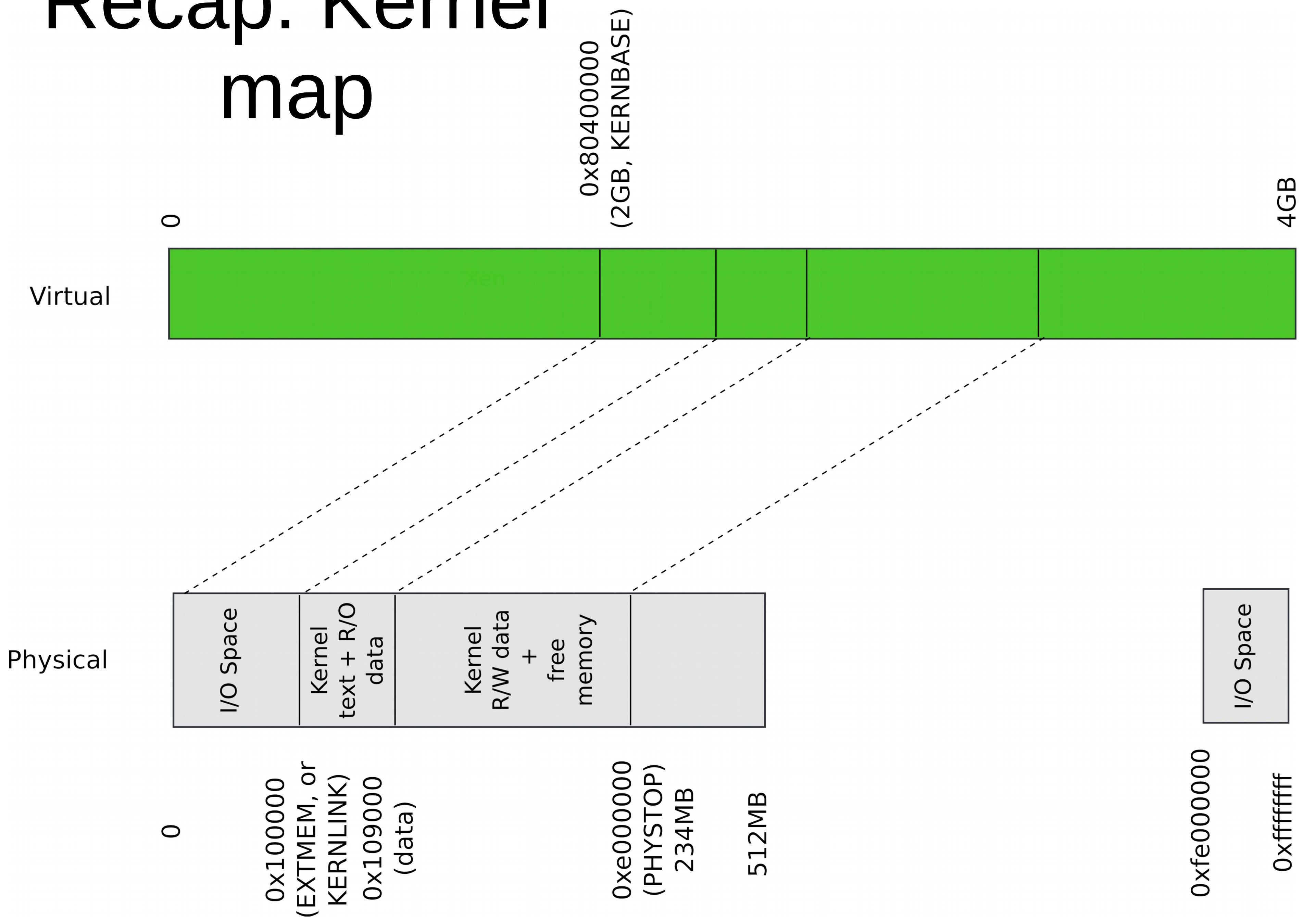

```
pde_t*
setupkvm(void)
{
  pde_t *pgdir;
  struct kmap *k;

  if((pgdir = (pde_t*)kalloc()) == 0)
    return 0;
  memset(pgdir, 0, PGSIZE);
...
  for(k = kmap; k < &kmap[NELEM(kmap)]; k++)
    if(mappages(pgdir, k->virt, k->phys_end - k->phys_start,
                (uint)k->phys_start, k->perm) < 0)
      return 0;
  return pgdir;
}
```

# Recap: Iterate in a loop: remap physical pages



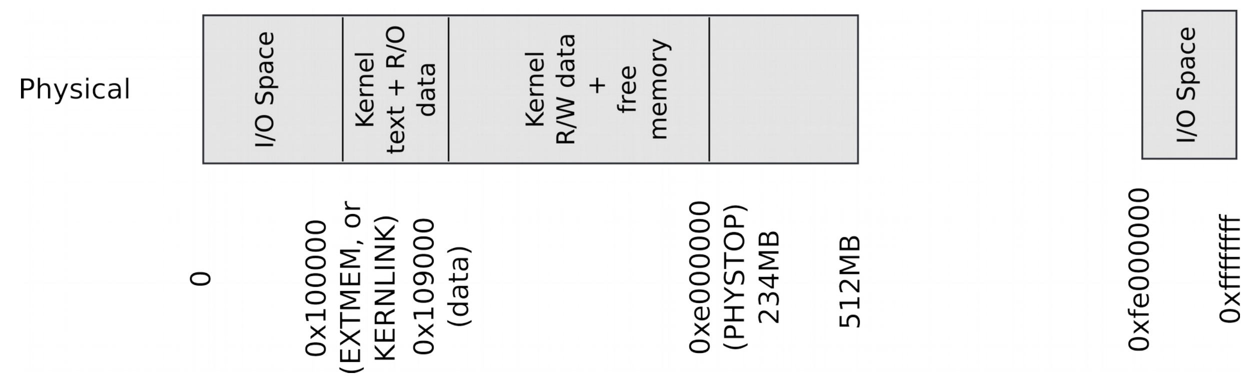

```
pde_t*
setupkvm(void)
{
  pde_t *pgdir;
  struct kmap *k;

  if((pgdir = (pde_t*)kalloc()) == 0)
    return 0;
  memset(pgdir, 0, PGSIZE);
...
  for(k = kmap; k < &kmap[NELEM(kmap)]; k++)
    if(mappages(pgdir, k->virt, k->phys_end - k->phys_start,
                (uint)k->phys_start, k->perm) < 0)
      return 0;
  return pgdir;
}
```

# Recap: Kernel map



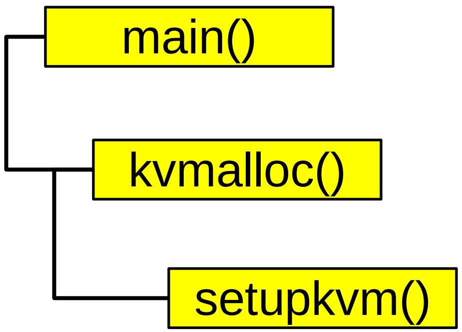
# Recap: Kmap – kernel map

```c
static struct kmap {
    void *virt;
    uint phys_start;
    uint phys_end;
    int perm;
} kmap[] = {
    { (void*)KERNBASE, 0, EXTMEM, PTE_W}, // I/O space
    { (void*)KERNLINK, V2P(KERNLINK), V2P(data), 0},//text+rodata
    { (void*)data, V2P(data), PHYSTOP, PTE_W}, // kern data+memory
    { (void*)DEVSPACE, DEVSPACE, 0, PTE_W}, // more devices
};
```

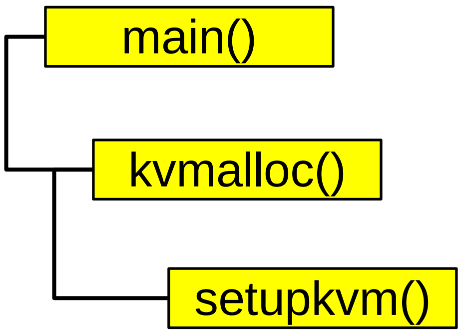


# Recap: Iterate in a loop: remap physical pages

```
pde_t*
setupkvm(void)
{
  pde_t *pgdir;
  struct kmap *k;

  if((pgdir = (pde_t*)kalloc()) == 0)
    return 0;
  memset(pgdir, 0, PGSIZE);
...
  for(k = kmap; k < &kmap[NELEM(kmap)]; k++)
    if(mappages(pgdir, k->virt, k->phys_end - k->phys_start,
                (uint)k->phys_start, k->perm) < 0)
      return 0;
  return pgdir;
}
```

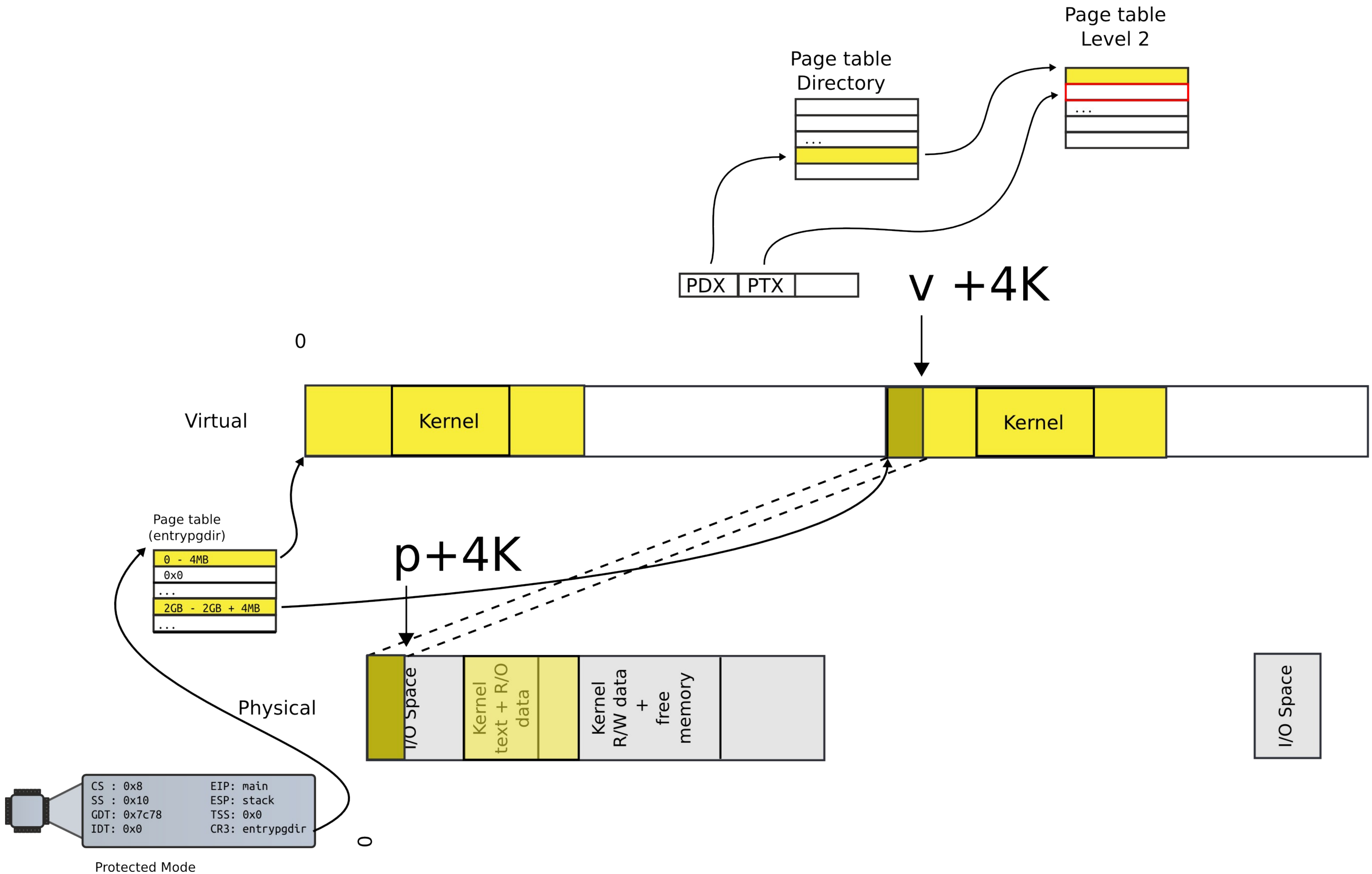


# Recap: Remap physical pages




```
pde_t*
setupkvm(void)
{
  pde_t *pgdir;
  struct kmap *k;

  if((pgdir = (pde_t*)kalloc()) == 0)
    return 0;
  memset(pgdir, 0, PGSIZE);
...
  for(k = kmap; k < &kmap[NELEM(kmap)]; k++)
    if(mappages(pgdir, k->virt, k->phys_end - k->phys_start,
                (uint)k->phys_start, k->perm) < 0)
      return 0;
  return pgdir;
}
```

# setupkvm(): Move to next page



# `exec()`: Construct process address space

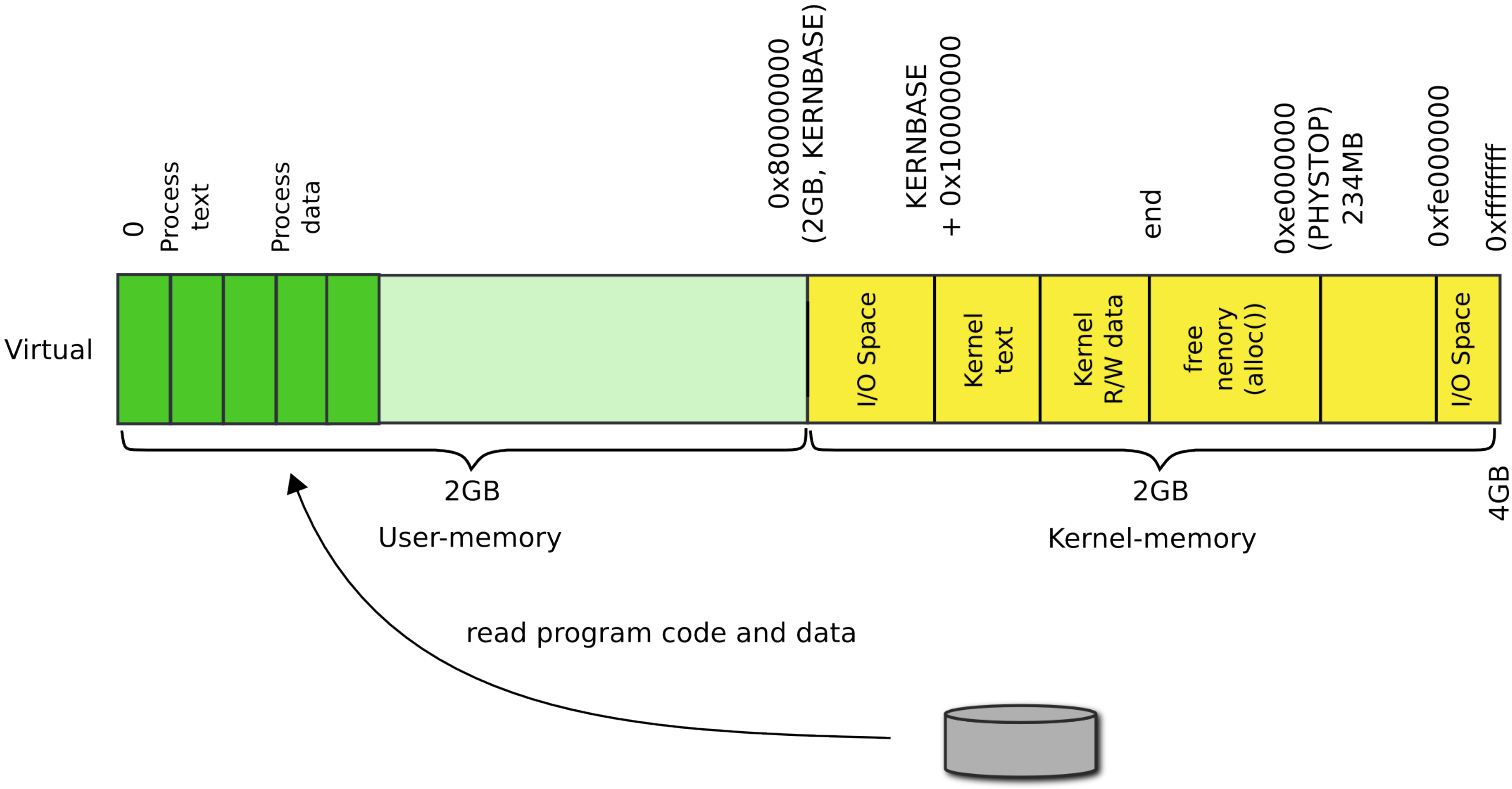
- Two step process
  - Create the kernel part of the address space
  - **Create the user part of the address space**

# 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



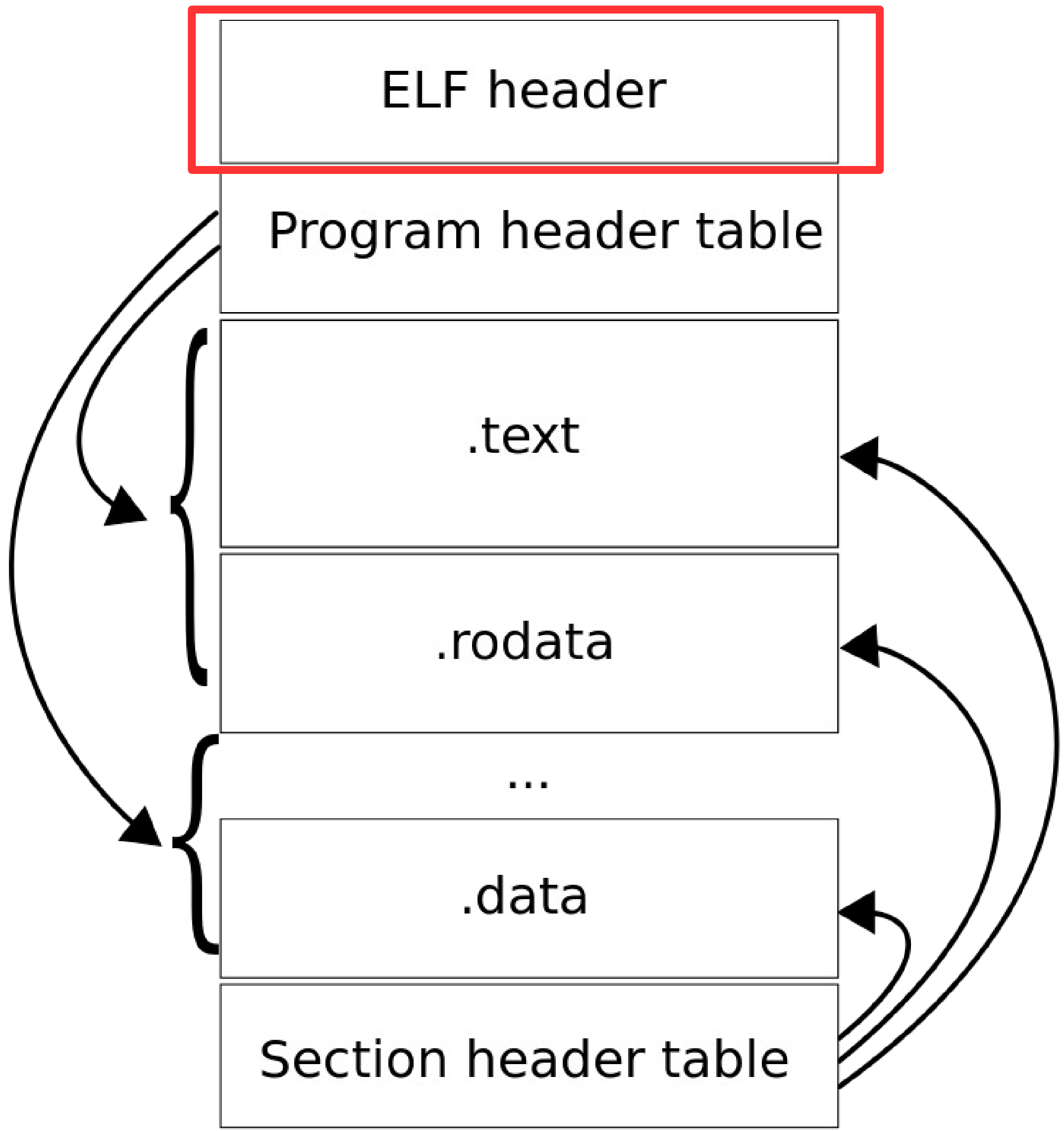
# Program loading loop

```
exec(char *path, char **argv)
{
...
  // Load program into memory.
  sz = 0;
  for(i=0, off=elf.phoff; i<elf.phnum; i++, off+=sizeof(ph)){
    if(readi(ip, (char*)&ph, off, sizeof(ph)) != sizeof(ph))
      goto bad;
...
    if((sz = allocuvm(pgdir, sz, ph.vaddr + ph.memsz)) == 0)
      goto bad;
    if(ph.vaddr % PGSIZE != 0)
      goto bad;
    if(loaduvm(pgdir, (char*)ph.vaddr, ip, ph.off, ph.filesz) < 0)
      goto bad;
  }
```

- Loop over all program headers

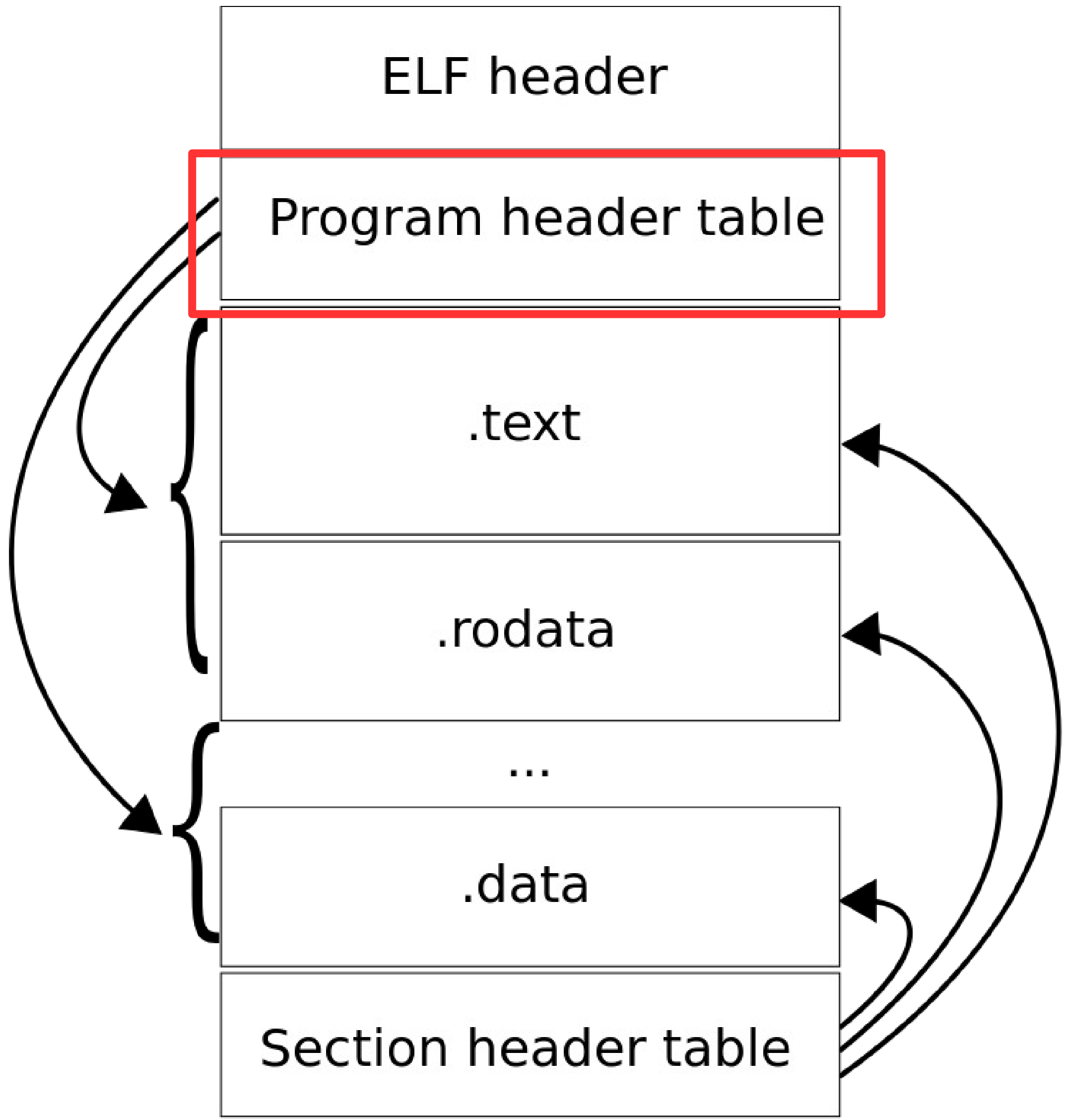
# ELF object file

- **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

```
// Load program into memory.
sz = 0;
for(i=0, off=elf.phoff; i<elf.phnum; i++, off+=sizeof(ph)){
  if(readi(ip, (char*)&ph, off, sizeof(ph)) != sizeof(ph))
    goto bad;
...
  if((sz = allocuvm(pgdir, sz, ph.vaddr + ph.memsz)) == 0)
    goto bad;
  if(ph.vaddr % PGSIZE != 0)
    goto bad;
  if(loaduvm(pgdir, (char*)ph.vaddr, ip, ph.off, ph.filesz) < 0)
    goto bad;
}
```

- Start at the beginning of the program header table

  off = elf.phoff

# Program loading loop

```
// Load program into memory.
sz = 0;
for(i=0, off=elf.phoff; i<elf.phnum; i++, off+=sizeof(ph)){
  if(readi(ip, (char*)&ph, off, sizeof(ph)) != sizeof(ph))
    goto bad;
...
  if((sz = allocuvm(pgdir, sz, ph.vaddr + ph.memsz)) == 0)
    goto bad;
  if(ph.vaddr % PGSIZE != 0)
    goto bad;
  if(loaduvm(pgdir, (char*)ph.vaddr, ip, ph.off, ph.filesz) < 0)
    goto bad;
}
```

- Read one program header entry at a time

# Program loading loop

```
// Load program into memory.
sz = 0;
for(i=0, off=elf.phoff; i<elf.phnum; i++, off+=sizeof(ph)){
  if(readi(ip, (char*)&ph, off, sizeof(ph)) != sizeof(ph))
    goto bad;
...
  if((sz = allocuvm(pgdir, sz, ph.vaddr + ph.memsz)) == 0)
    goto bad;
  if(ph.vaddr % PGSIZE != 0)
    goto bad;
  if(loaduvm(pgdir, (char*)ph.vaddr, ip, ph.off, ph.filesz) < 0)
    goto bad;
}
```

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

# Program loading loop

```
// Load program into memory.
sz = 0;
for(i=0, off=elf.phoff; i<elf.phnum; i++, off+=sizeof(ph)){
  if(readi(ip, (char*)&ph, off, sizeof(ph)) != sizeof(ph))
    goto bad;
...
  if((sz = allocuvm(pgdir, sz, ph.vaddr + ph.memsz)) == 0)
    goto bad;
  if(ph.vaddr % PGSIZE != 0)
    goto bad;
  if(loaduvm(pgdir, (char*)ph.vaddr, ip, ph.off, ph.filesz)
< 0)
    goto bad;
}
```

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

# Program loading loop

```
// Load program into memory.
sz = 0;
for(i=0, off=elf.phoff; i<elf.phnum; i++, off+=sizeof(ph)){
  if(readi(ip, (char*)&ph, off, sizeof(ph)) != sizeof(ph))
    goto bad;
...
  if((sz = allocuvm(pgdir, sz, ph.vaddr + ph.memsz)) == 0)
    goto bad;
  if(ph.vaddr % PGSIZE != 0)
    goto bad;
  if(loaduvm(pgdir, (char*)ph.vaddr, ip, ph.off, ph.filesz) < 0)
    goto bad;
}
```

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

# Program loading loop

```
// Load program into memory.
sz = 0;
for(i=0, off=elf.phoff; i<elf.phnum; i++, off+=sizeof(ph)){
  if(readi(ip, (char*)&ph, off, sizeof(ph)) != sizeof(ph))
    goto bad;
...
  if((sz = allocuvm(pgdir, sz, ph.vaddr + ph.memsz)) == 0)
    goto bad;
  if(ph.vaddr % PGSIZE != 0)
    goto bad;
  if(loaduvm(pgdir, (char*)ph.vaddr, ip, ph.off, ph.filesz) < 0)
    goto bad;
}
```

- New size of the address space

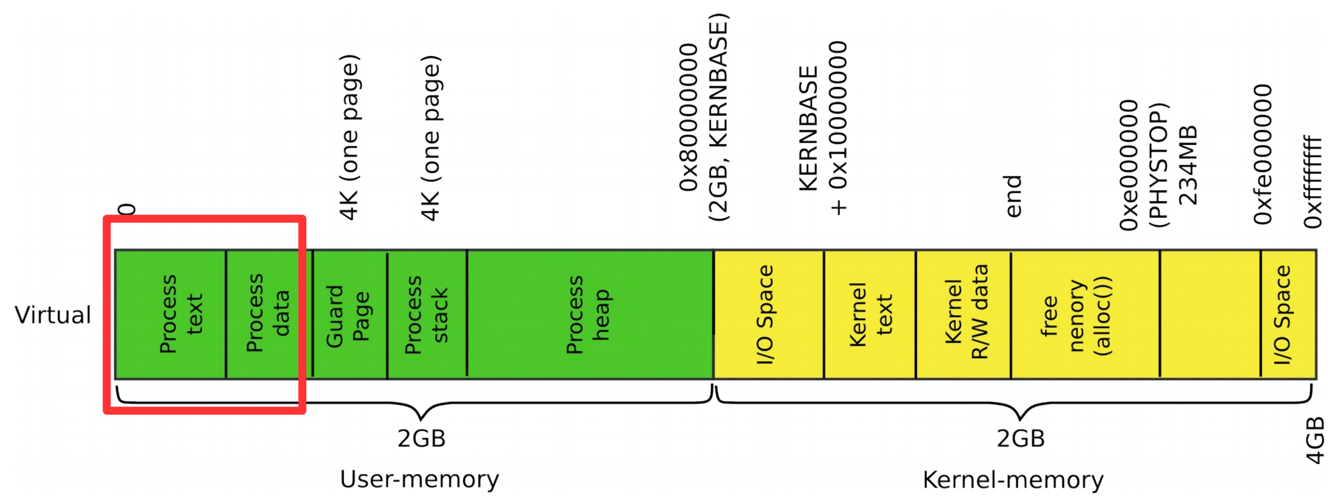
# Program loading loop

```
// Load program into memory.
sz = 0;
for(i=0, off=elf.phoff; i<elf.phnum; i++, off+=sizeof(ph)){
  if(readi(ip, (char*)&ph, off, sizeof(ph)) != sizeof(ph))
    goto bad;
...
  if((sz = allocuvm(pgdir, sz, ph.vaddr + ph.memsz)) == 0)
    goto bad;
  if(ph.vaddr % PGSIZE != 0)
    goto bad;
  if(loaduvm(pgdir, (char*)ph.vaddr, ip, ph.off, ph.filesz) < 0)
    goto bad;
}
```

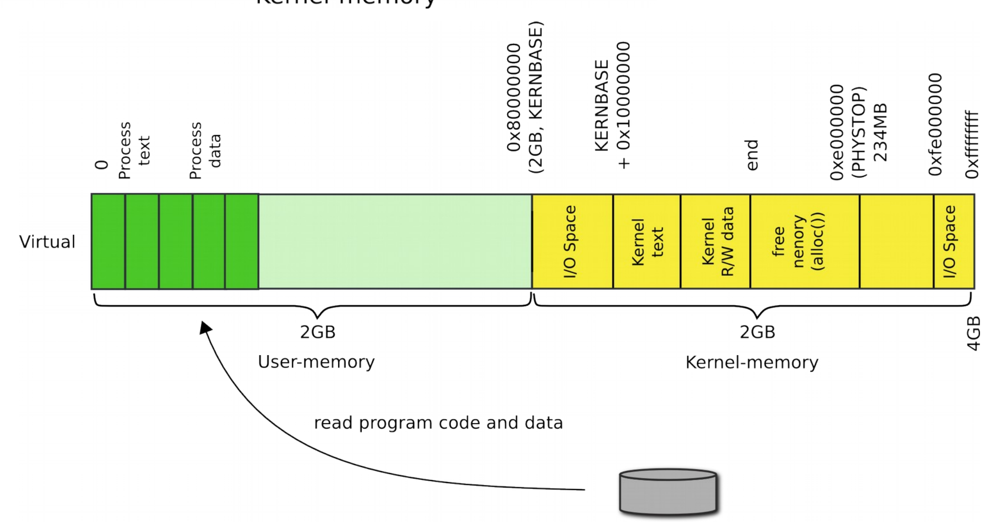
- Load program section from disk

# Two main functions

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

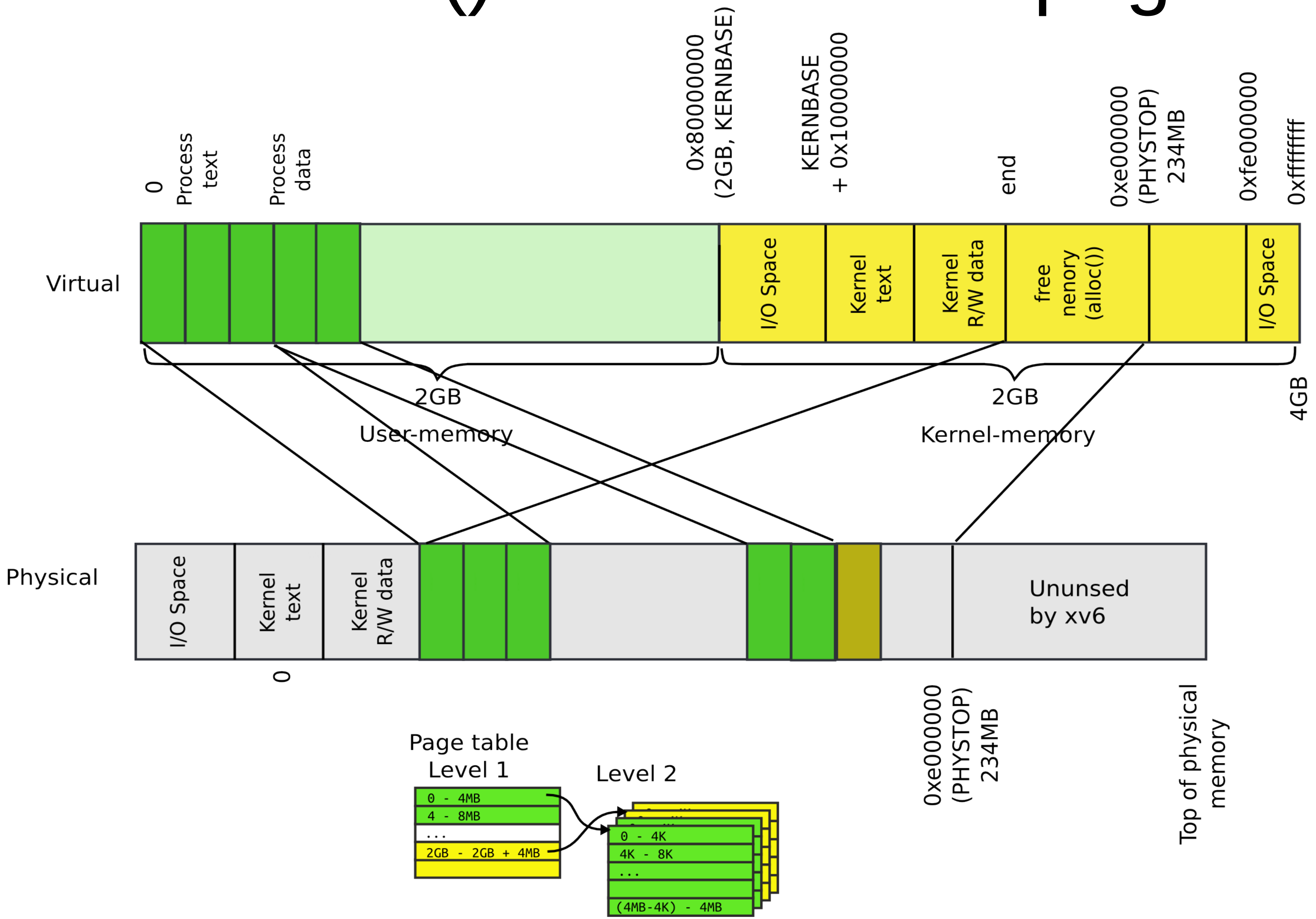



- loaduvm() -- load user-memory with data from disk



# Lets take a closer look allocuvm()

# allocuvm(): allocate user pages



# Allocate user address space

```
allocuvm(pde_t *pgdir, uint oldsz, uint newsz)
{
...
  if(newsz >= KERNBASE)
    return 0;
…
  a = PGROUNDUP(oldsz);
  for(; a < newsz; a += PGSIZE){
    mem = kalloc();
...
    memset(mem, 0, PGSIZE);
    if(mappages(pgdir, (char*)a, PGSIZE, V2P(mem), PTE_W|PTE_U) < 0){
...
      return 0;
    }
  }
  return newsz;
}
```

- New size can't be over 2GB

# Allocate user address space

```
allocuvm(pde_t *pgdir, uint oldsz, uint newsz)
{
...
  if(newsz >= KERNBASE)
    return 0;
…
  a = PGROUNDUP(oldsz);
  for(; a < newsz; a += PGSIZE){
    mem = kalloc();
...
    memset(mem, 0, PGSIZE);
    if(mappages(pgdir, (char*)a, PGSIZE, V2P(mem), PTE_W|PTE_U) < 0){
...
      return 0;
    }
  }
  return newsz;
}
```

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

```
allocuvm(pde_t *pgdir, uint oldsz, uint newsz)
{
...
  if(newsz >= KERNBASE)
    return 0;
...
  for(; a < newsz; a += PGSIZE){
    mem = kalloc();
...
    memset(mem, 0, PGSIZE);
    if(mappages(pgdir, (char*)a, PGSIZE, V2P(mem), PTE_W|PTE_U) < 0){
...
      return 0;
    }
  }
  return newsz;
}
```

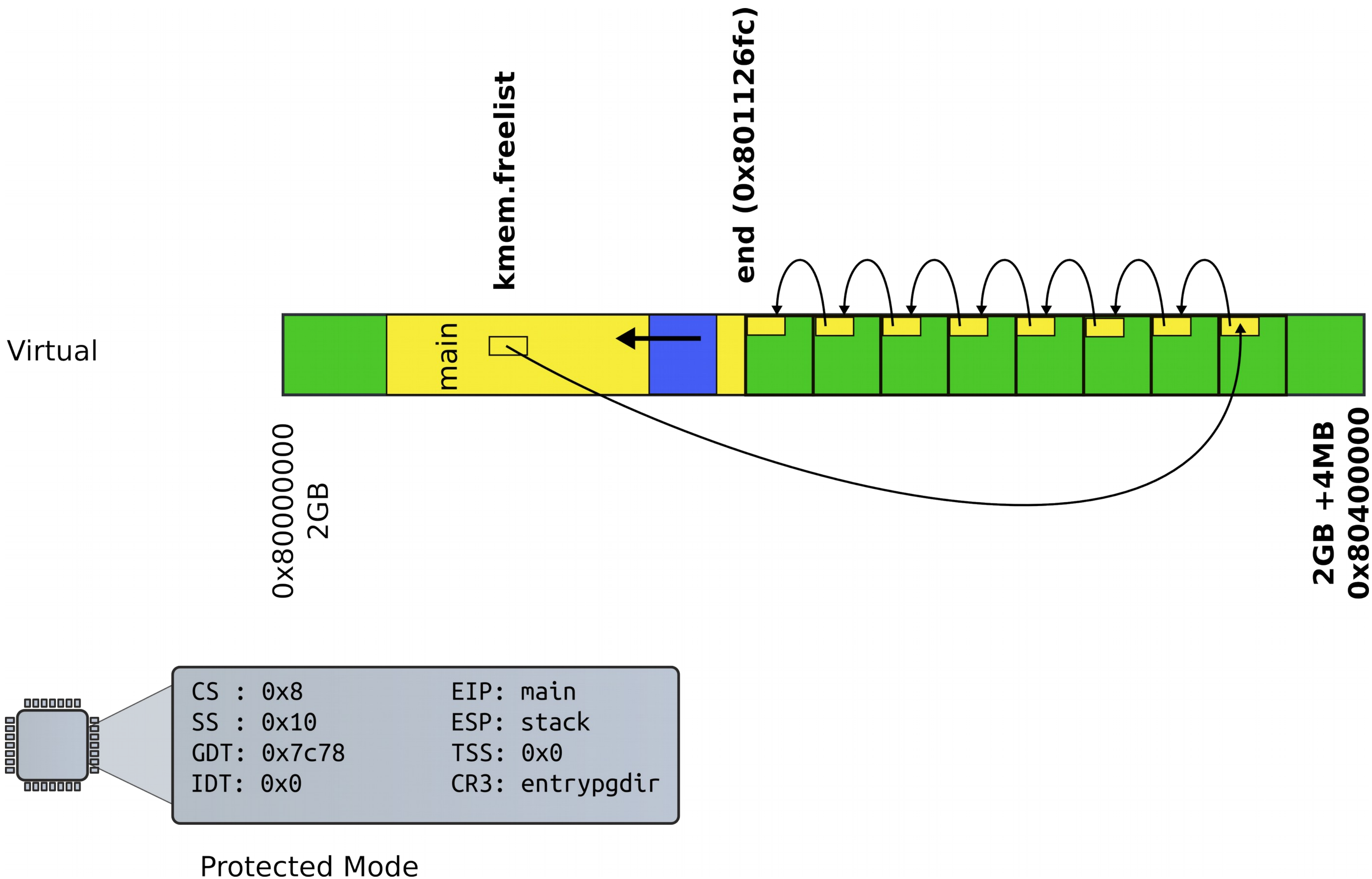
# Allocate user address space

- Allocate a new page

```
allocuvm(pde_t *pgdir, uint oldsz, uint newsz)
{
...
  if(newsz >= KERNBASE)
    return 0;
...
  for(; a < newsz; a += PGSIZE){
    mem = kalloc();
...
    memset(mem, 0, PGSIZE);
    if(mappages(pgdir, (char*)a, PGSIZE, V2P(mem), PTE_W|PTE_U) < 0){
...
      return 0;
    }
  }
  return newsz;
}
```

Where does this memory come from?

# Kernel memory allocator



# 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 }
```

- Set page to 0

# Allocate user address space

```
allocuvm(pde_t *pgdir, uint oldsz, uint newsz)
{
...
  if(newsz >= KERNBASE)
    return 0;
...
  for(; a < newsz; a += PGSIZE){
    mem = kalloc();
...
    memset(mem, 0, PGSIZE);
    if(mappages(pgdir, (char*)a, PGSIZE, V2P(mem), PTE_W|PTE_U) < 0){
...
      return 0;
    }
  }
  return newsz;
}
```

- Map the page

# Allocate user address space

```
allocuvm(pde_t *pgdir, uint oldsz, uint newsz)
{
...
  if(newsz >= KERNBASE)
    return 0;
...
  for(; a < newsz; a += PGSIZE){
    mem = kalloc();
...
    memset(mem, 0, PGSIZE);
    if(mappages(pgdir, (char*)a, PGSIZE, V2P(mem), PTE_W|PTE_U) < 0){
...
      return 0;
    }
  }
  return newsz;
}
```

- Take the page directory as an argument

# Allocate user address space

```
allocuvm(pde_t *pgdir, uint oldsz, uint newsz)
{
...
  if(newsz >= KERNBASE)
    return 0;
...
  for(; a < newsz; a += PGSIZE){
    mem = kalloc();
...
    memset(mem, 0, PGSIZE);
    if(mappages(pgdir, (char*)a, PGSIZE, V2P(mem), PTE_W|PTE_U) < 0){
...
      return 0;
    }
  }
  return newsz;
}
```

- Virtual address where to map the page

# Allocate user address space

```
allocuvm(pde_t *pgdir, uint oldsz, uint newsz)
{
...
  if(newsz >= KERNBASE)
    return 0;
...
  for(; a < newsz; a += PGSIZE){
    mem = kalloc();
...
    memset(mem, 0, PGSIZE);
    if(mappages(pgdir, (char*)a, PGSIZE, V2P(mem), PTE_W|PTE_U) < 0){
...
      return 0;
    }
  }
  return newsz;
}
```

- Size of the region
  - One page!

# Allocate user address space

```
allocuvm(pde_t *pgdir, uint oldsz, uint newsz)
{
...
  if(newsz >= KERNBASE)
    return 0;
...
  for(; a < newsz; a += PGSIZE){
    mem = kalloc();
...
    memset(mem, 0, PGSIZE);
    if(mappages(pgdir, (char*)a, PGSIZE, V2P(mem), PTE_W|PTE_U) < 0){
...
      return 0;
    }
  }
  return newsz;
}
```

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

## Allocate user address space

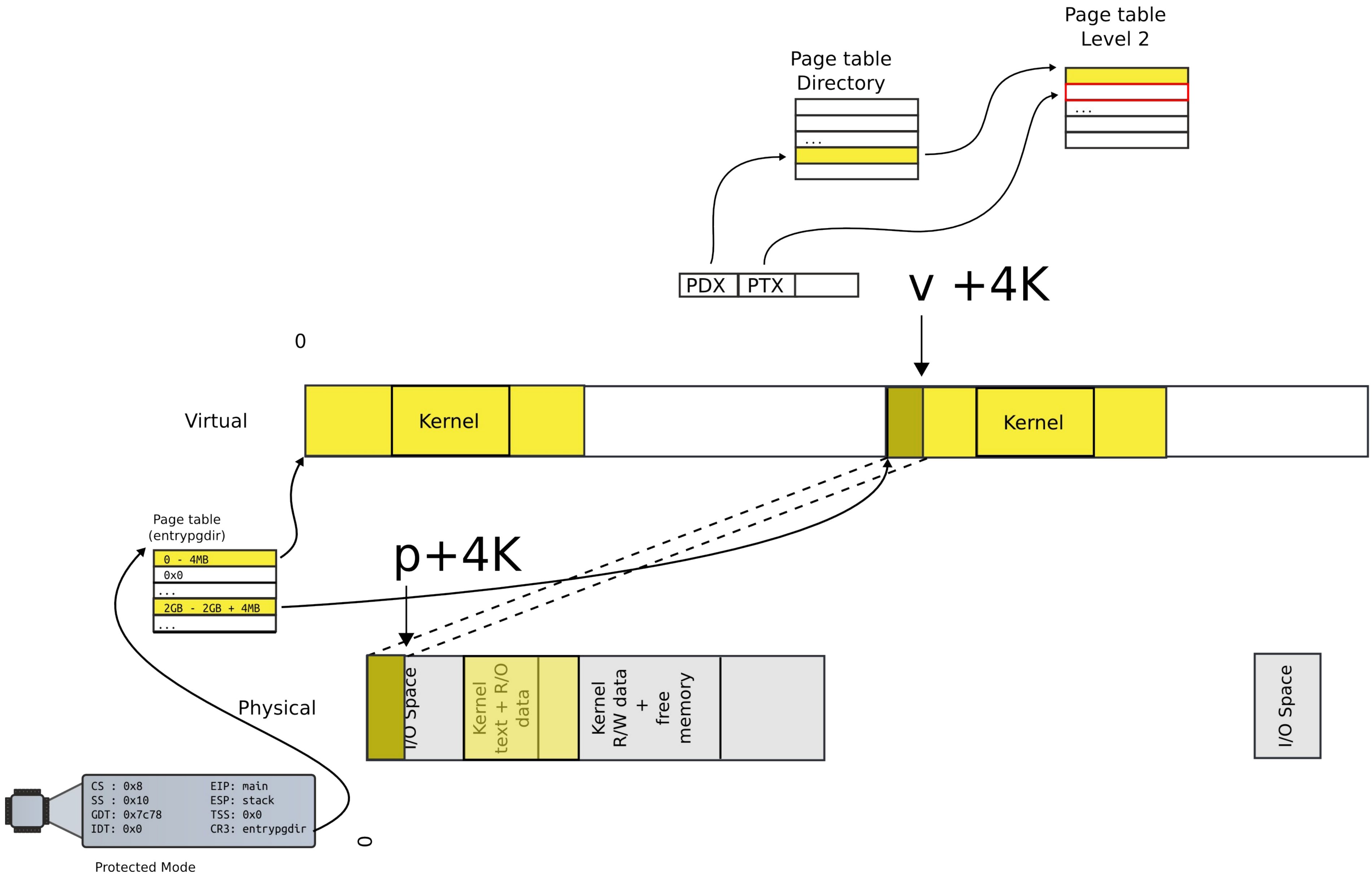
```
allocuvm(pde_t *pgdir, uint oldsz, uint newsz)
{
...
  if(newsz >= KERNBASE)
    return 0;
...
  for(; a < newsz; a += PGSIZE){
    mem = kalloc();
...
    memset(mem, 0, PGSIZE);
    if(mappages(pgdir, (char*)a, PGSIZE, V2P(mem), PTE_W|PTE_U) < 0){
...
      return 0;
    }
  }
  return newsz;
}
```

- Flags
  - Writable and user-accessible

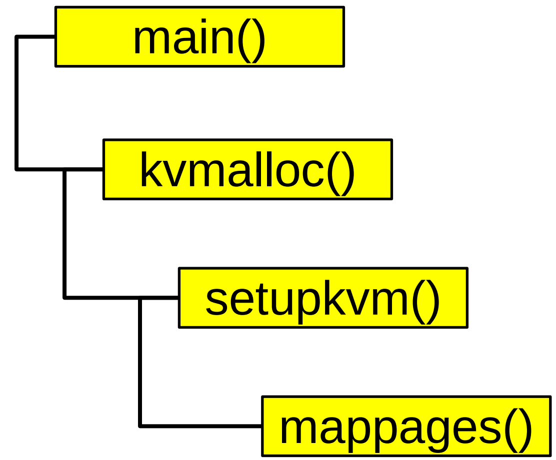
# Who remembers mappages()?

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

# mappages(): map a region



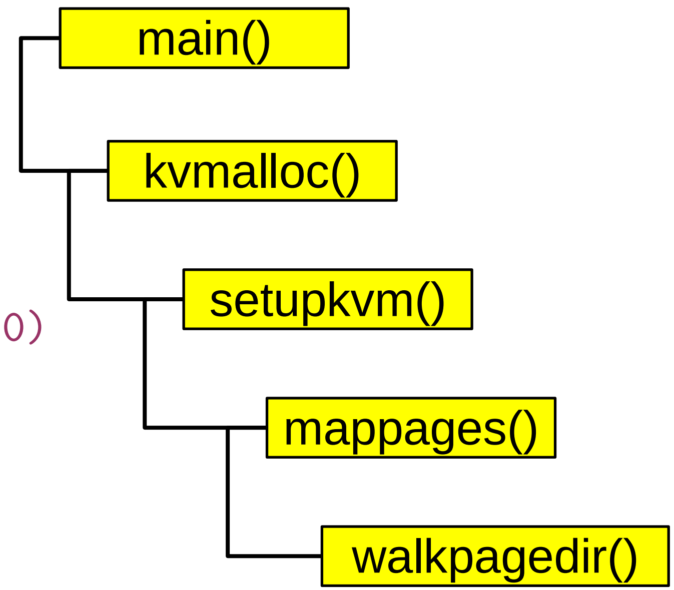
```c
mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm)
{
  char *a, *last;
  pte_t *pte;

  a = (char*)PGROUNDDOWN((uint)va);
  last = (char*)PGROUNDDOWN(((uint)va) + size - 1);
  for(;;){
    if((pte = walkpgdir(pgdir, a, 1)) == 0)
      return -1;
    if(*pte & PTE_P)
      panic("remap");
    *pte = pa | perm | PTE_P;
    if(a == last)
      break;
    a += PGSIZE;
    pa += PGSIZE;
  }
  return 0;
}
```




# Lookup the page table entry

# Walk page table

```c
walkpgdir(pde_t *pgdir, const void *va, int alloc)
{
    pde_t *pde;
    pte_t *pgtab;

    pde = &pgdir[PDX(va)];
    if(*pde & PTE_P){
        pgtab = (pte_t*)P2V(PTE_ADDR(*pde));
    } else {
        if(!alloc || (pgtab = (pte_t*)kalloc()) == 0)
            return 0;
        // Make sure all those PTE_P bits are zero.
        memset(pgtab, 0, PGSIZE);
...
        *pde = V2P(pgtab) | PTE_P | PTE_W | PTE_U;
    }
    return &pgtab[PTX(va)];
}
```



# Allocate user address space

```
allocuvm(pde_t *pgdir, uint oldsz, uint newsz)
{
...
  if(newsz >= KERNBASE)
    return 0;
...
  for(; a < newsz; a += PGSIZE){
    mem = kalloc();
...
    memset(mem, 0, PGSIZE);
    if(mappages(pgdir, (char*)a, PGSIZE, V2P(mem), PTE_W|PTE_U) < 0){
...
      return 0;
    }
  }
  return newsz;
}
```

- Continue in a loop
  - Map pages one by one

Now the second function: `loaduvm()`

# exec() – create a new process

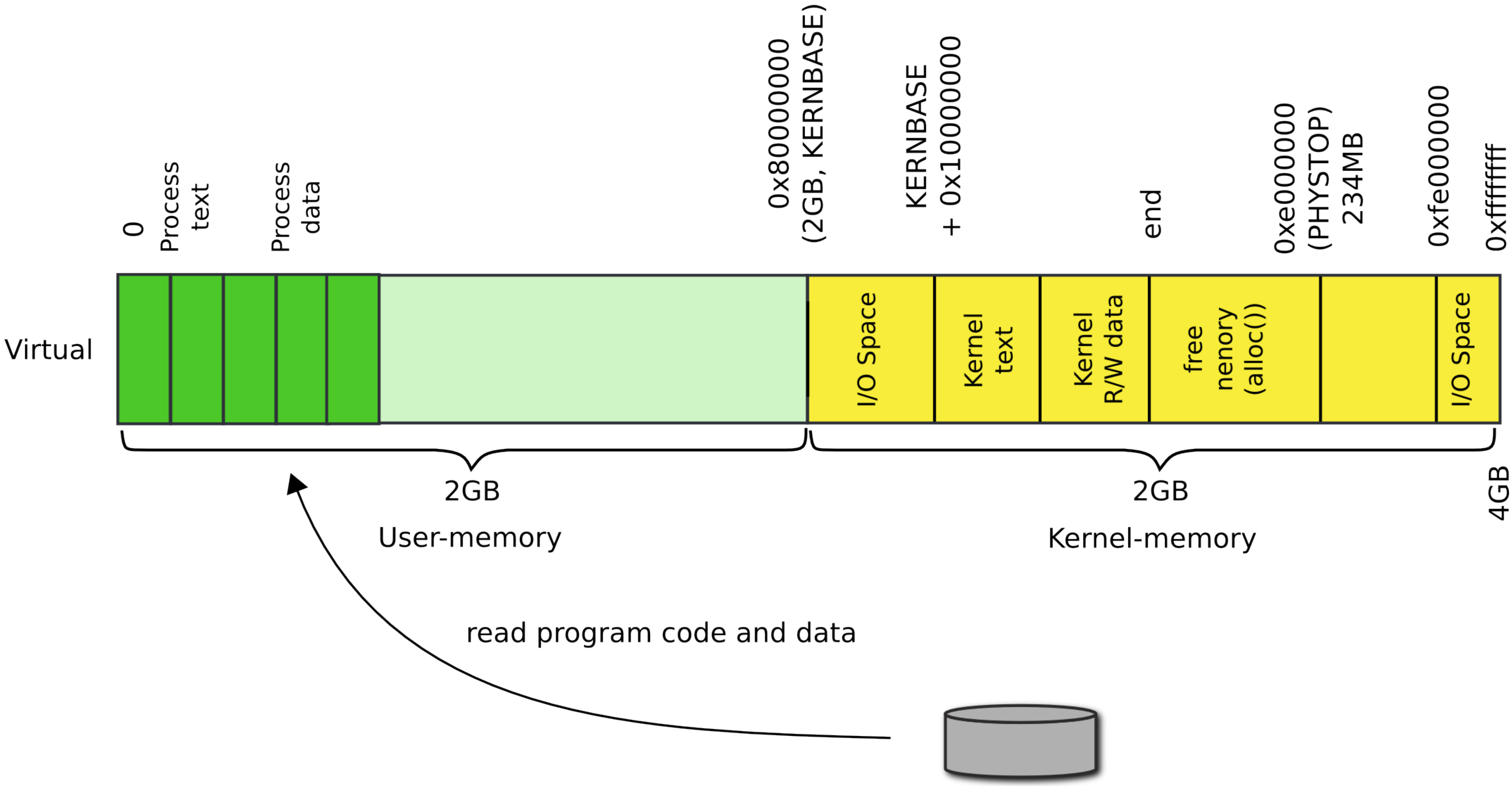
- Read process binary from disk
  - namei() takes a file path (“/bin/ls”) as an argument
  - Returns an inode
  - readi() reads the inode (file data)
- Create process address space
  - Create a page table
  - Map only kernel space
- **Load program into memory**
  - Allocate user-level pages for the program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

# Load program into memory

```
// Load program into memory.
sz = 0;
for(i=0, off=elf.phoff; i<elf.phnum; i++, off+=sizeof(ph)){
  if(readi(ip, (char*)&ph, off, sizeof(ph)) != sizeof(ph))
    goto bad;
...
  if((sz = allocuvm(pgdir, sz, ph.vaddr + ph.memsz)) == 0)
    goto bad;
  if(ph.vaddr % PGSIZE != 0)
    goto bad;
  if(loaduvm(pgdir, (char*)ph.vaddr, ip, ph.off, ph.filesz) < 0)
    goto bad;
}
```

- Load program section from disk

# loaduvm(): read program from disk



# Load program into memory

```
loaduvm(pde_t *pgdir, char *addr, struct inode *ip, uint offset, uint sz)
{
...
  for(i = 0; i < sz; i += PGSIZE){
    if((pte = walkpgdir(pgdir, addr+i, 0)) == 0)
      panic("loaduvm: address should exist");
    pa = PTE_ADDR(*pte);
    if(sz - i < PGSIZE)
      n = sz - i;
    else
      n = PGSIZE;
    if(readi(ip, P2V(pa), offset+i, n) != n)
      return -1;
  }
  return 0;
}
```

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

# Load program into memory

```
loaduvm(pde_t *pgdir, char *addr, struct inode *ip, uint offset, uint sz)
{
...
  for(i = 0; i < sz; i += PGSIZE){
    if((pte = walkpgdir(pgdir, addr+i, 0)) == 0)
      panic("loaduvm: address should exist");
    pa = PTE_ADDR(*pte);
    if(sz - i < PGSIZE)
      n = sz - i;
    else
      n = PGSIZE;
    if(readi(ip, P2V(pa), offset+i, n) != n)
      return -1;
  }
  return 0;
}
```

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

# Load program into memory

```
loaduvm(pde_t *pgdir, char *addr, struct inode *ip, uint offset, uint sz)
{
...
  for(i = 0; i < sz; i += PGSIZE){
    if((pte = walkpgdir(pgdir, addr+i, 0)) == 0)
      panic("loaduvm: address should exist");
    pa = PTE_ADDR(*pte);
    if(sz - i < PGSIZE)
      n = sz - i;
    else
      n = PGSIZE;
    if(readi(ip, P2V(pa), offset+i, n) != n)
      return -1;
  }
  return 0;
}
```

- Resolve pte into physical address

# Load program into memory

```
loaduvm(pde_t *pgdir, char *addr, struct inode *ip, uint offset, uint sz)
{
...
  for(i = 0; i < sz; i += PGSIZE){
    if((pte = walkpgdir(pgdir, addr+i, 0)) == 0)
      panic("loaduvm: address should exist");
    pa = PTE_ADDR(*pte);
    if(sz - i < PGSIZE)
      n = sz - i;
    else
      n = PGSIZE;
    if(readi(ip, P2V(pa), offset+i, n) != n)
      return -1;
  }
  return 0;
}
```

- Then use the virtual address of that physical page

# Load program into memory

```
loaduvm(pde_t *pgdir, char *addr, struct inode *ip, uint offset, uint sz)
{
...
  for(i = 0; i < sz; i += PGSIZE){
    if((pte = walkpgdir(pgdir, addr+i, 0)) == 0)
      panic("loaduvm: address should exist");
    pa = PTE_ADDR(*pte);
    if(sz - i < PGSIZE)
      n = sz - i;
    else
      n = PGSIZE;
    if(readi(ip, P2V(pa), offset+i, n) != n)
      return -1;
  }
  return 0;
}
```

- Wait... virtual address of a page?

```
loaduvm(pde_t *pgdir, char *addr, struct inode *ip, uint offset, uint sz)
{
...
  for(i = 0; i < sz; i += PGSIZE){
    if((pte = walkpgdir(pgdir, addr+i, 0)) == 0)
      panic("loaduvm: address should exist");
    pa = PTE_ADDR(*pte);
    if(sz - i < PGSIZE)
      n = sz - i;
    else
      n = PGSIZE;
    if(readi(ip, P2V(pa), offset+i, n) != n)
      return -1;
  }
  return 0;
}
```

# Load program into memory

- Why can't we use addr directly?

# Drawing: two page tables

# Load program into memory

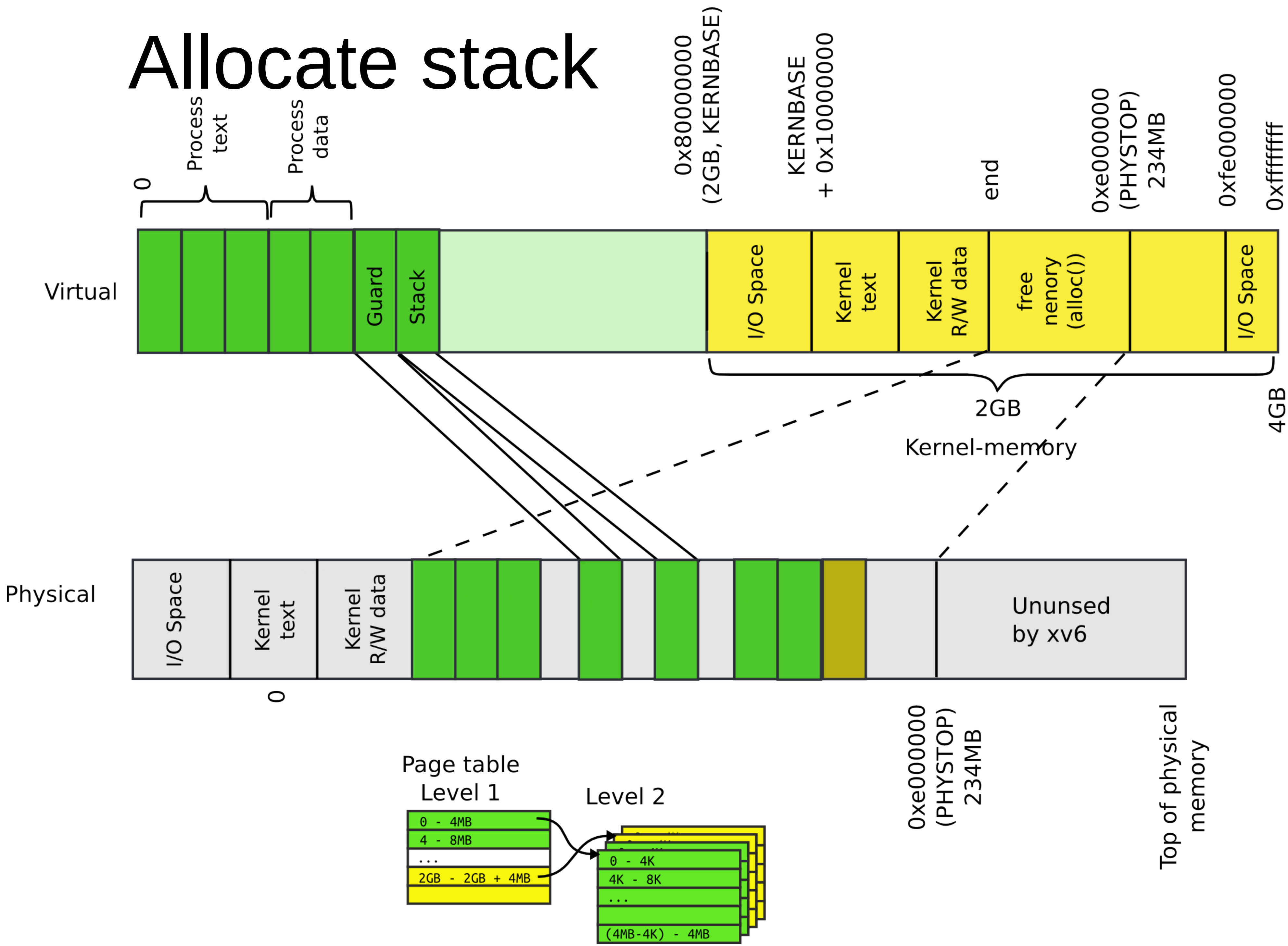
```
loaduvm(pde_t *pgdir, char *addr, struct inode *ip, uint offset, uint sz)
{
...
  for(i = 0; i < sz; i += PGSIZE){
    if((pte = walkpgdir(pgdir, addr+i, 0)) == 0)
      panic("loaduvm: address should exist");
    pa = PTE_ADDR(*pte);
    if(sz - i < PGSIZE)
      n = sz - i;
    else
      n = PGSIZE;
    if(readi(ip, P2V(pa), offset+i, n) != n)
      return -1;
  }
  return 0;
}
```

- 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

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

# exec() – create a new process

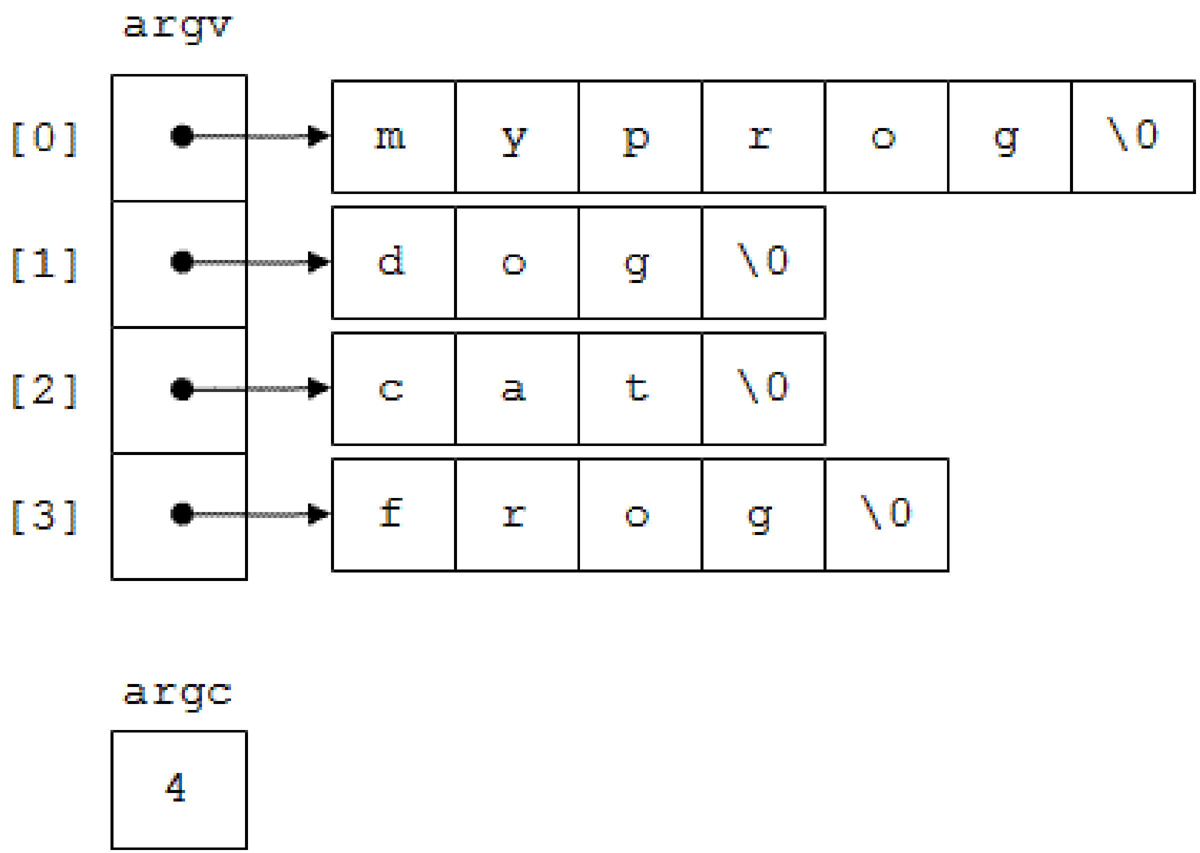
- Read process binary from disk
- Create process address space
- Load program into memory
- Allocate program stack
- **Push program arguments on the stack**

# 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
```



# 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)

```
// Push argument strings, prepare rest of stack in ustack.
for(argc = 0; argv[argc]; argc++) {
...
  sp = (sp - (strlen(argv[argc]) + 1)) & ~3;
  if(copyout(pgdir, sp, argv[argc], strlen(argv[argc]) + 1) < 0)
    goto bad;
  ustack[3+argc] = sp;
}
ustack[3+argc] = 0;

ustack[0] = 0xffffffff; // fake return PC
ustack[1] = argc;
ustack[2] = sp - (argc+1)*4; // argv pointer

sp -= (3+argc+1) * 4;
if(copyout(pgdir, sp, ustack, (3+argc+1)*4) < 0)
 goto bad;
```

Copy elements of the array one by one on the stack

- Remember we can't use virtual addresses directly

```
// Push argument strings, prepare rest of stack in ustack.
for(argc = 0; argv[argc]; argc++) {
...
  sp = (sp - (strlen(argv[argc]) + 1)) & ~3;
  if(copyout(pgdir, sp, argv[argc], strlen(argv[argc]) + 1) < 0)
    goto bad;
  ustack[3+argc] = sp;
}
ustack[3+argc] = 0;

ustack[0] = 0xffffffff; // fake return PC
ustack[1] = argc;
ustack[2] = sp - (argc+1)*4; // argv pointer

sp -= (3+argc+1) * 4;
if(copyout(pgdir, sp, ustack, (3+argc+1)*4) < 0)
 goto bad;
```

Push argc – number of arguments in the argv[]

```
// Push argument strings, prepare rest of stack in ustack.
for(argc = 0; argv[argc]; argc++) {
...
  sp = (sp - (strlen(argv[argc]) + 1)) & ~3;
  if(copyout(pgdir, sp, argv[argc], strlen(argv[argc]) + 1) < 0)
    goto bad;
  ustack[3+argc] = sp;
}
ustack[3+argc] = 0;

ustack[0] = 0xffffffff; // fake return PC
ustack[1] = argc;
ustack[2] = sp - (argc+1)*4; // argv pointer

sp -= (3+argc+1) * 4;
if(copyout(pgdir, sp, ustack, (3+argc+1)*4) < 0)
 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

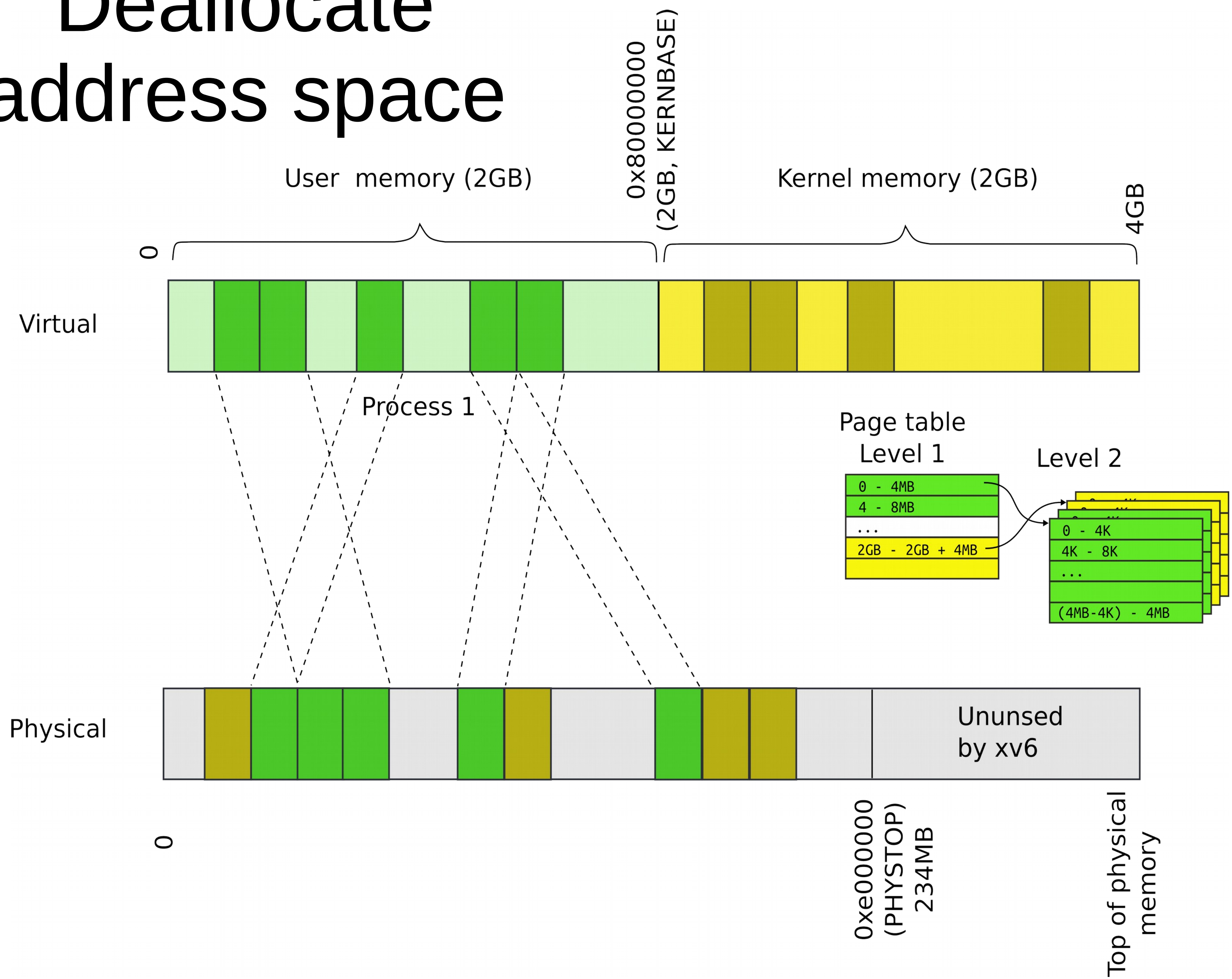
```
int
exec(char *path, char **argv)
{
…
  switchuvm(proc);
  freevm(oldpgdir);
  return 0;
...
```

# 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

```c
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

```
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 a page

# 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);
}
```

# Deallocate page table directory itself

```c
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

# Creating the first process

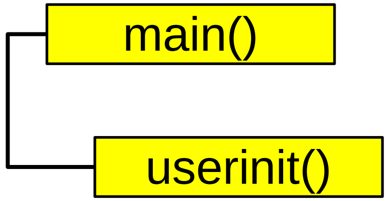
```
main(void)
{
  kinit1(end, P2V(4*1024*1024)); // phys page allocator
  kvmalloc(); // kernel page table
  mpinit(); // detect other processors
...
  seginit(); // segment descriptors
...
  tvinit(); // trap vectors
...
  userinit(); // first user process
  mpmain(); // finish this processor's setup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

main()

# Userinit() – create first process

- Allocate process structure
  - Information about the process

```
userinit(void)
{
  struct proc *p;
  extern char _binary_initcode_start[],
              _binary_initcode_size[];
...
  p = allocproc();
  initproc = p;
  if((p->pgdir = setupkvm()) == 0)
    panic("userinit: out of memory?");
  inituvm(p->pgdir, _binary_initcode_start,
          (int)_binary_initcode_size);
  p->sz = PGSIZE;
  memset(p->tf, 0, sizeof(*p->tf));
...
}
```

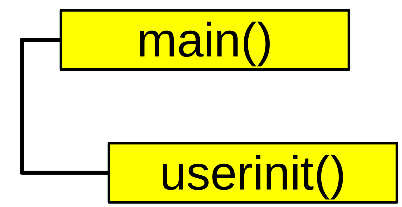


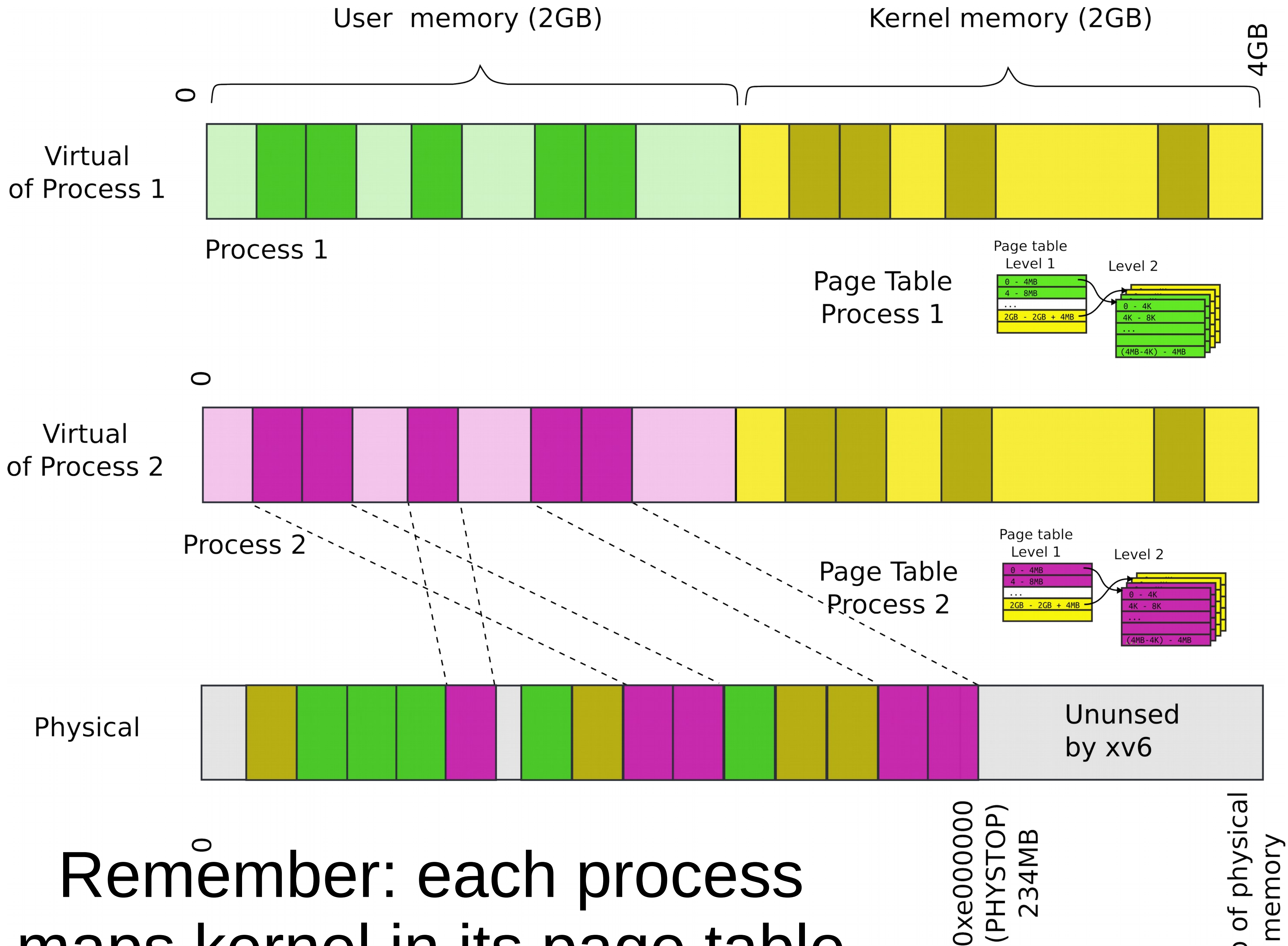
```
struct proc {
  uint sz; // Size of process memory (bytes)
  pde_t* pgdir; // Page table
  char *kstack; // Bottom of kernel stack for this process
  enum procstate state; // Process state
  volatile int pid; // Process ID
  struct proc *parent; // Parent process
  struct trapframe *tf; // Trap frame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

# Userinit() – create first process

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

```c
userinit(void)
{
  struct proc *p;
  extern char _binary_initcode_start[],
              _binary_initcode_size[];
...
  p = allocproc();
  initproc = p;
  if((p->pgdir = setupkvm()) == 0)
    panic("userinit: out of memory?");
  inituvm(p->pgdir, _binary_initcode_start,
          (int)_binary_initcode_size);
  p->sz = PGSIZE;
  memset(p->tf, 0, sizeof(*p->tf));
...
}
```



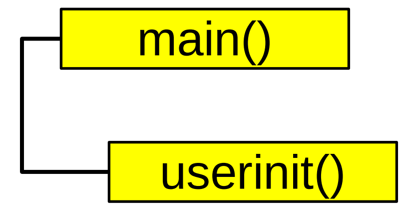



# Remember: each process maps kernel in its page table

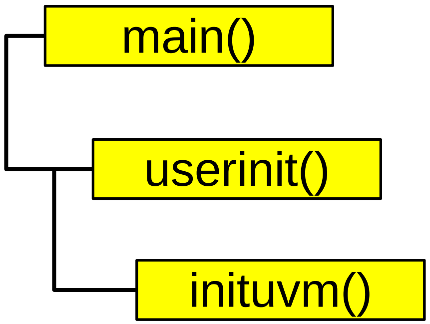
# Userinit() – create first process

- 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**

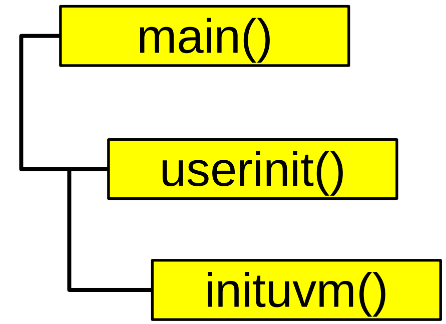
```c
userinit(void)
{
  struct proc *p;
  extern char _binary_initcode_start[],
              _binary_initcode_size[];
...
  p = allocproc();
  initproc = p;
  if((p->pgdir = setupkvm()) == 0)
    panic("userinit: out of memory?");
  inituvm(p->pgdir, _binary_initcode_start,
         (int)_binary_initcode_size);
  p->sz = PGSIZE;
  memset(p->tf, 0, sizeof(*p->tf));
...
}
```



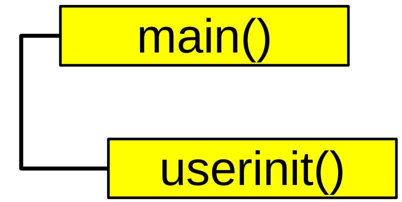
```c
inituvm(pde_t *pgdir, char *init, uint sz)
{
  char *mem;

  if(sz >= PGSIZE)
    panic("inituvm: more than a page");
  mem = kalloc();
  memset(mem, 0, PGSIZE);
  mappages(pgdir, 0, PGSIZE, V2P(mem),
               PTE_W|PTE_U);
  memmove(mem, init, sz);
}
```



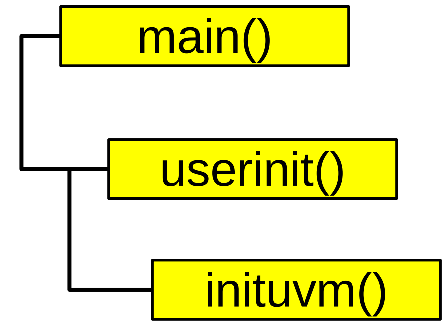
```c
inituvm(pde_t *pgdir, char *init, uint sz)
{
  char *mem;

  if(sz >= PGSIZE)
    panic("inituvm: more than a page");
  mem = kalloc();
  memset(mem, 0, PGSIZE);
  mappages(pgdir, 0, PGSIZE, V2P(mem),
               PTE_W|PTE_U);
  memmove(mem, init, sz);
}
```



```c
userinit(void)
{
  struct proc *p;
  extern char _binary_initcode_start[],
              _binary_initcode_size[];
...
  p = allocproc();
  initproc = p;
  if((p->pgdir = setupkvm()) == 0)
    panic("userinit: out of memory?");
  inituvm(p->pgdir, _binary_initcode_start,
         (int)_binary_initcode_size);
  p->sz = PGSIZE;
  memset(p->tf, 0, sizeof(*p->tf));
...
}
```



```c
inituvm(pde_t *pgdir, char *init, uint sz)
{
  char *mem;

  if(sz >= PGSIZE)
    panic("inituvm: more than a page");
  mem = kalloc();
  memset(mem, 0, PGSIZE);
  mappages(pgdir, 0, PGSIZE, V2P(mem),
           PTE_W|PTE_U);
  memmove(mem, init, sz);
}
```



# initcode.S: call exec(“/init”, argv);

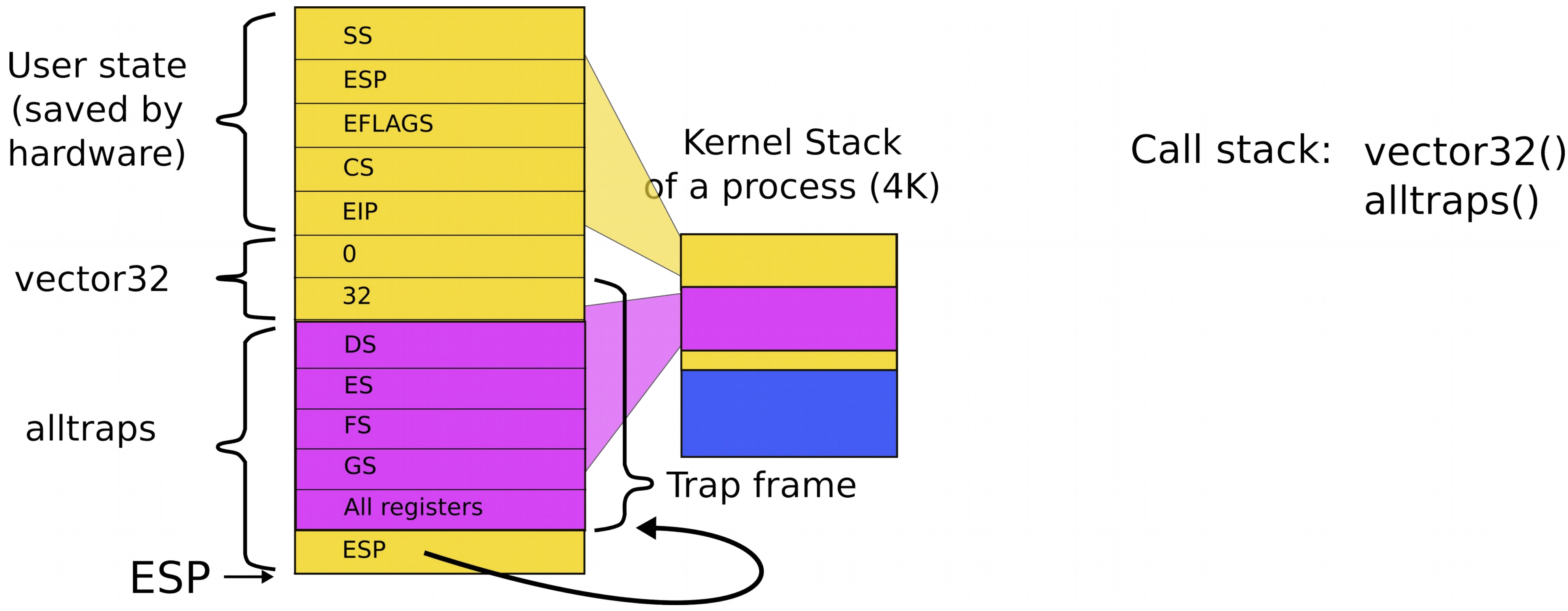
```
start:
  pushl $argv
  pushl $init
  pushl $0 // where caller pc would be
  movl $SYS_exec, %eax
  int $T_SYSCALL

...
# char init[] = "/init\0";
init:
  .string "/init\0"

# char *argv[] = { init, 0 };
.p2align 2
argv:
  .long init
  .long 0
```

# 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



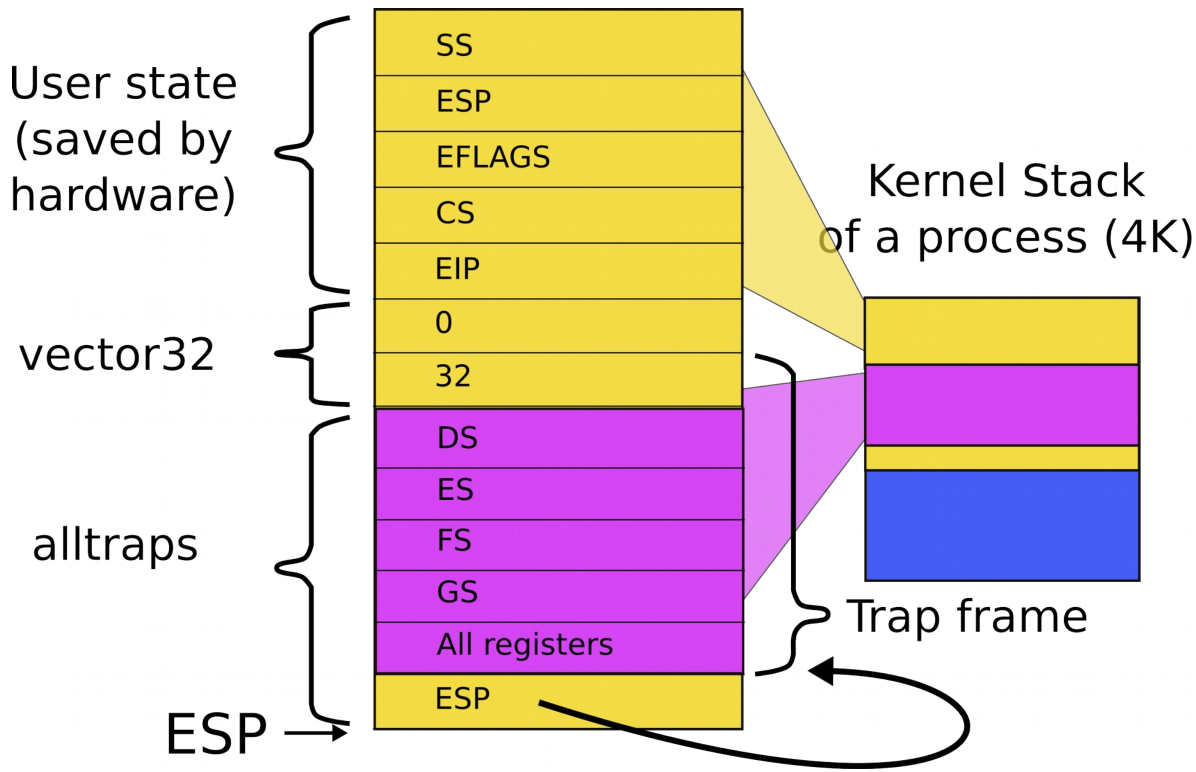
```
struct proc {
  uint sz; // Size of process memory (bytes)
  pde_t* pgdir; // Page table
  char *kstack; // Bottom of kernel stack for this process
  enum procstate state; // Process state
  volatile int pid; // Process ID
  struct proc *parent; // Parent process
  struct trapframe *tf; // Trap frame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

Trap frame is on the kernel stack of the process

```
allocproc(void)
{
...
  // Allocate kernel stack.
  if((p->kstack = kalloc()) == 0){
    p->state = UNUSED;
    return 0;
  }
  sp = p->kstack + KSTACKSIZE;

  // Leave room for trap frame.
  sp -= sizeof *p->tf;
  p->tf = (struct trapframe*)sp;

...
}
```

```
userinit(void)
{
...
  inituvm(p->pgdir, _binary_initcode_start,
               (int)_binary_initcode_size);
  p->sz = PGSIZE;
  memset(p->tf, 0, sizeof(*p->tf));
  p->tf->cs = (SEG_UCODE << 3) | DPL_USER;
  p->tf->ds = (SEG_UDATA << 3) | DPL_USER;
  p->tf->es = p->tf->ds;
  p->tf->ss = p->tf->ds;
  p->tf->eflags = FL_IF;
  p->tf->esp = PGSIZE;
  p->tf->eip = 0; // beginning of initcode.S
...
}
```

# Kernel stack after interrupt/syscall




Call stack: vector32()
alltraps()

```
userinit(void)
{
...
  memset(p->tf, 0, sizeof(*p->tf));
  p->tf->cs = (SEG_UCODE << 3) | DPL_USER;
  p->tf->ds = (SEG_UDATA << 3) | DPL_USER;
  p->tf->es = p->tf->ds;
  p->tf->ss = p->tf->ds;
  p->tf->eflags = FL_IF;
  p->tf->esp = PGSIZE;
  p->tf->eip = 0; // beginning of initcode.S

  safestrcpy(p->name, "initcode", sizeof(p->name));
  p->cwd = namei("/");

  p->state = RUNNABLE;
...
}
```

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

```c
main(void)
{
...
  if(open("console", O_RDWR) < 0){
    mknod("console", 1, 1);
    open("console", O_RDWR);
  }
  dup(0); // stdout
  dup(0); // stderr

  for(;;){
    printf(1, "init: starting sh\n");
    pid = fork();
    if(pid < 0){
      printf(1, "init: fork failed\n");
      exit();
    }
    if(pid == 0){
      exec("sh", argv);
      printf(1, "init: exec sh failed\n");
      exit();
    }
    while((wpid=wait()) >= 0 && wpid != pid)
      printf(1, "zombie!\n");
  }
}
```

- 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!