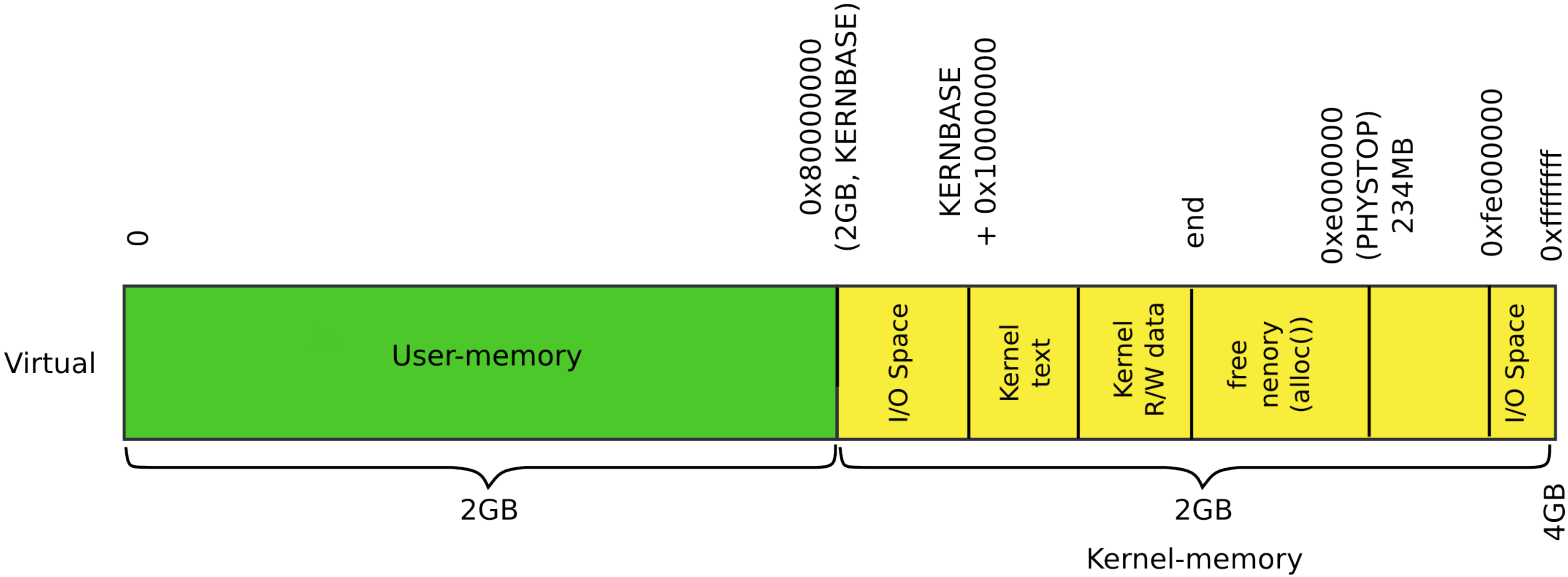
238P: Operating Systems

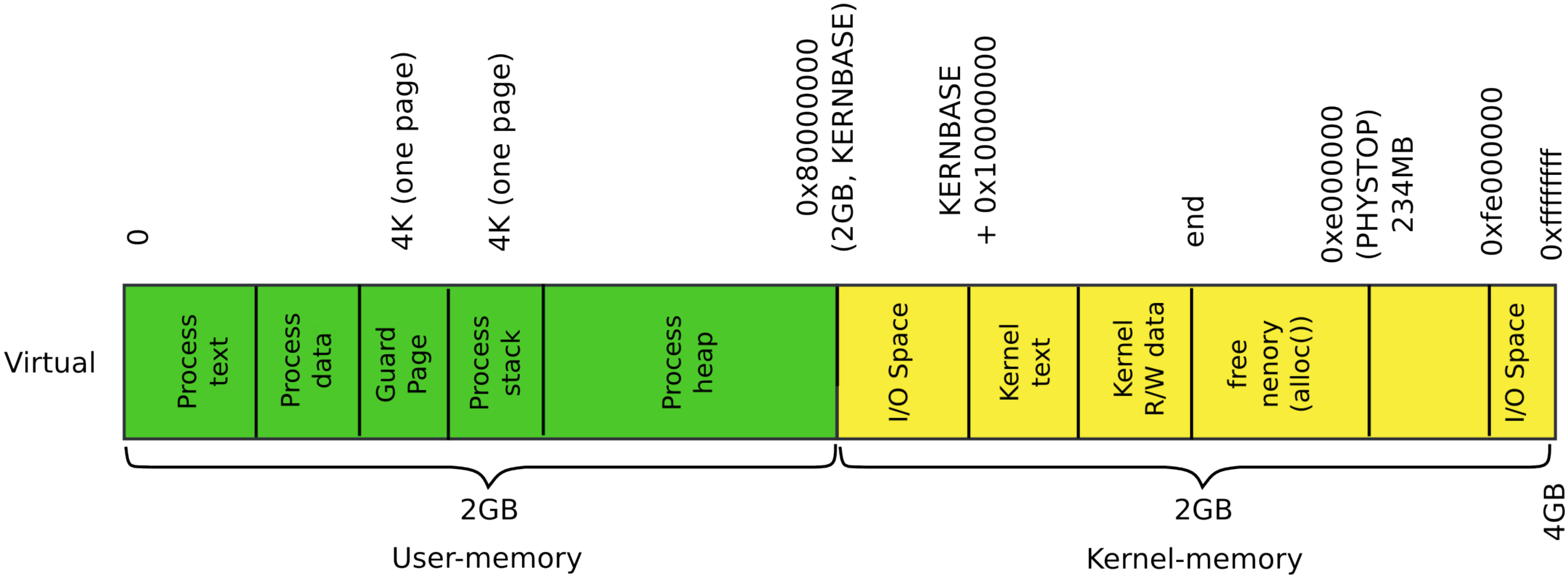
Lecture 9: Creating Processes

Anton Burtsev
February, 2018

Recap: kernel memory



Today: process memory



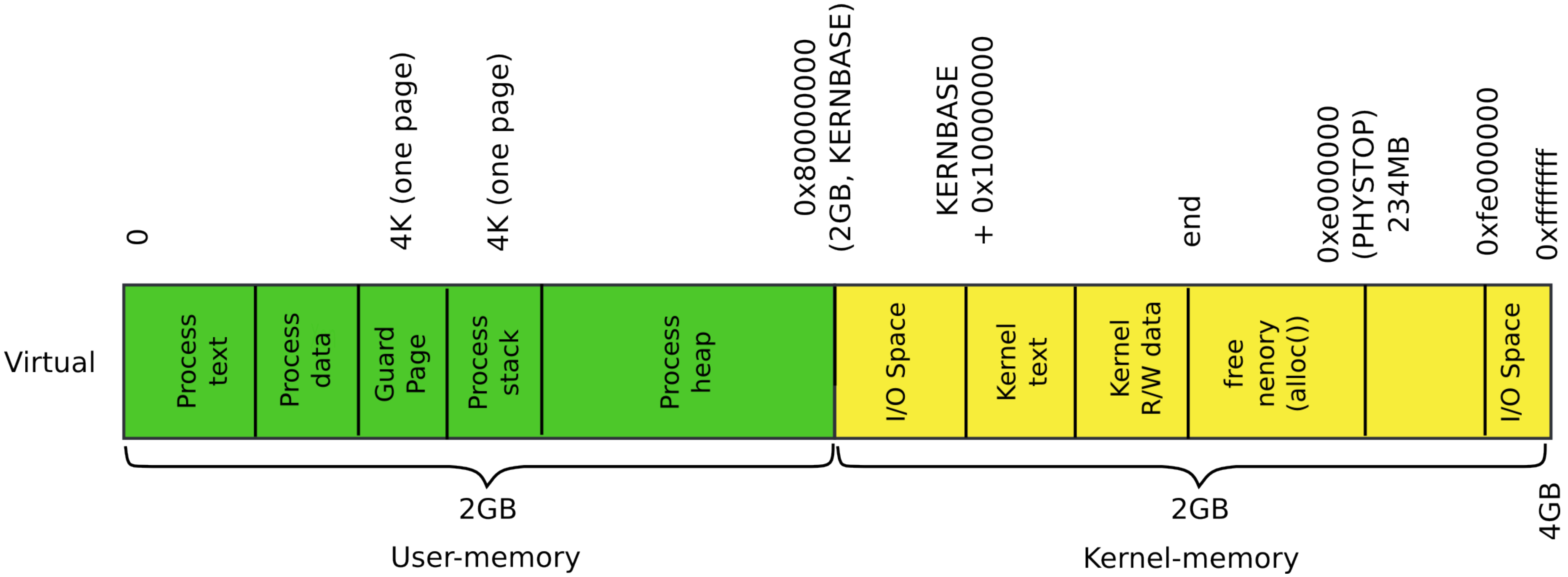
How does kernel creates new processes?

How does kernel creates new processes?

- Exec
  - exec(“/bin/ls”, argv);

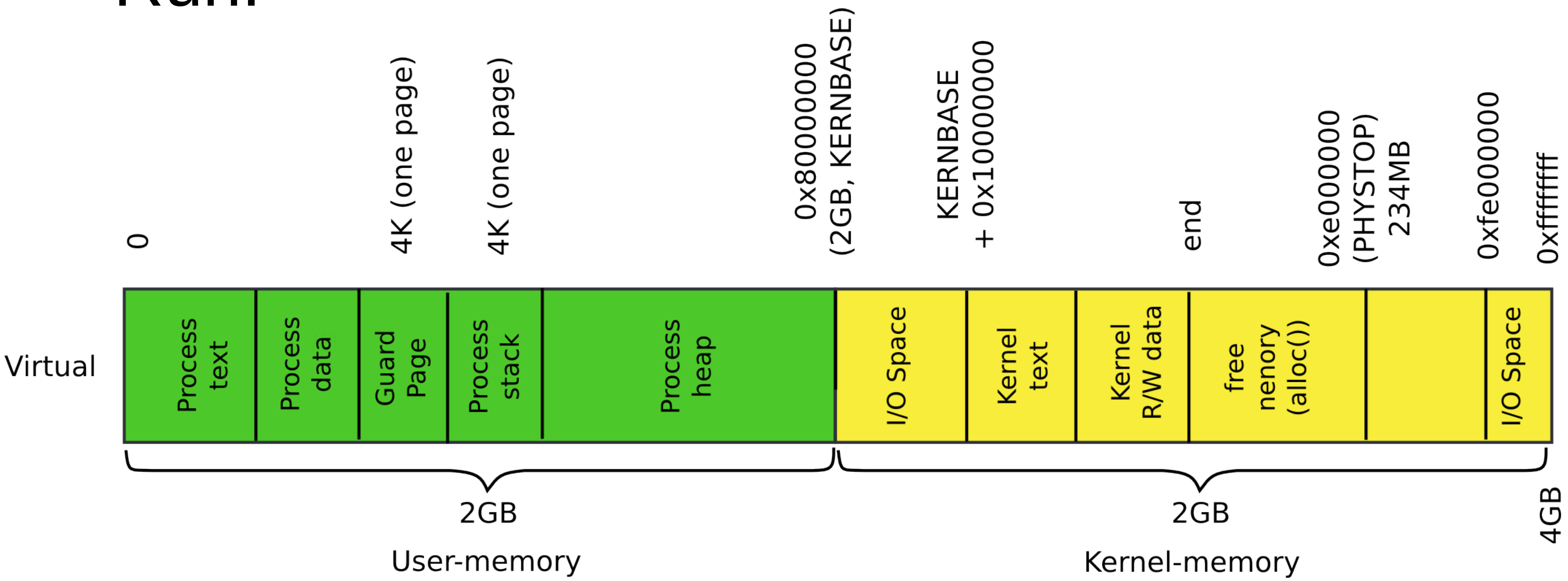
exec(): high-level outline

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



exec(): high-level outline

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



exec() – 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)

exec(): locate inode

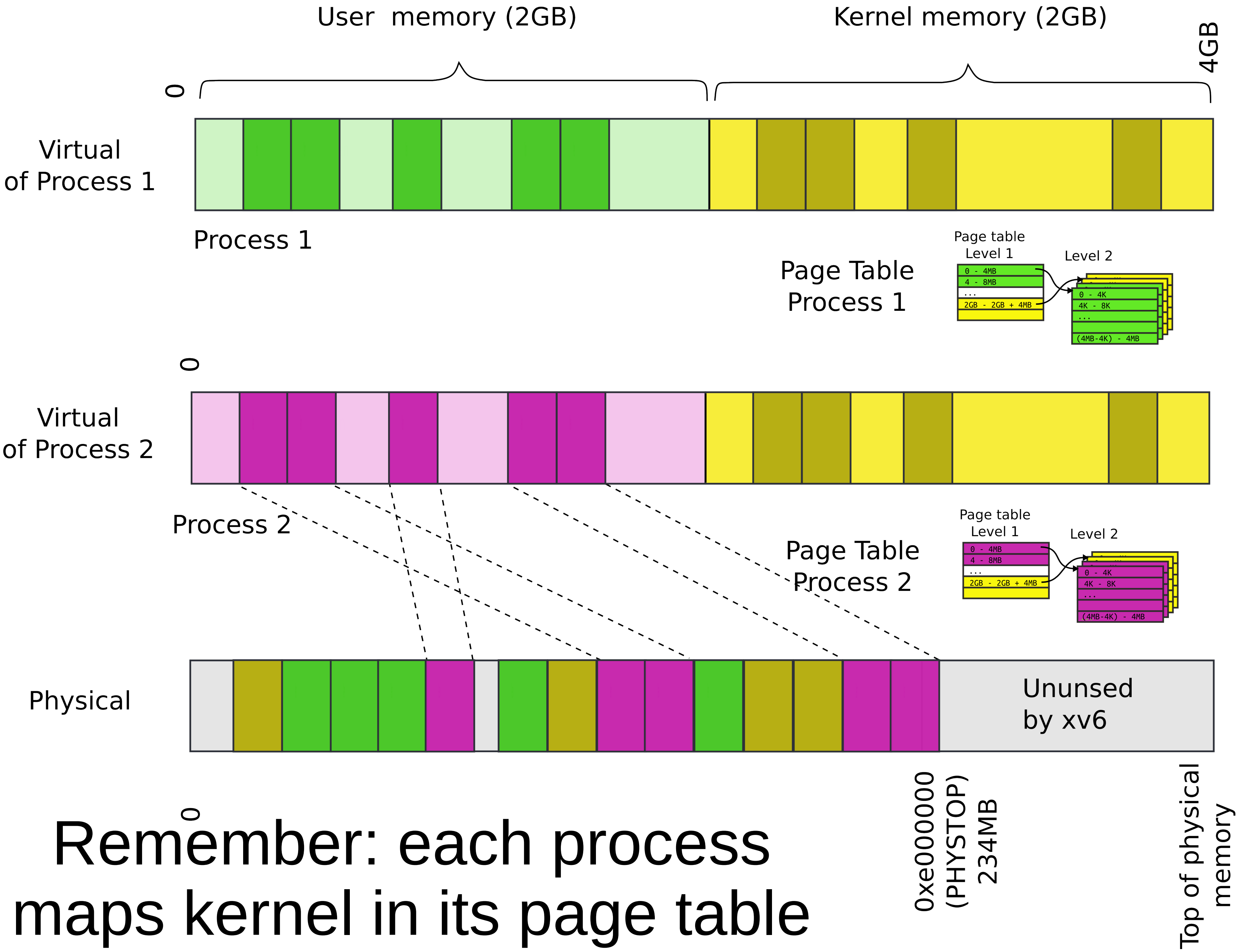
```
int
exec(char *path, char **argv)
{
...
  if((ip = namei(path)) == 0){
    end_op();
    return -1;
  }
  // Check ELF header
  if(readi(ip, (char*)&elf, 0, sizeof(elf)) <
                                      sizeof(elf))
    goto bad;
  if(elf.magic != ELF_MAGIC)
    goto bad;
```

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;
```

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**



Remember: each process maps kernel in its page table

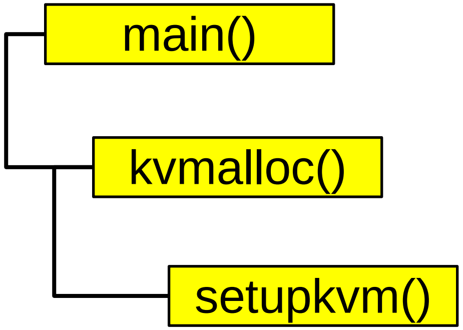
Exec(): Setup kernel address space()

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

```

- Remember from last time?

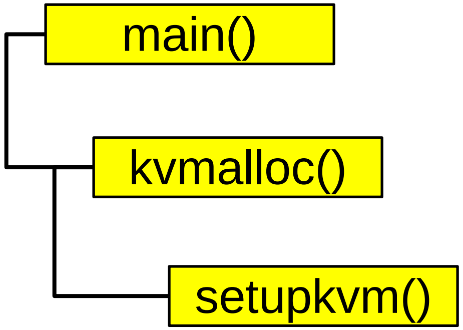
Recap: Allocate page table directory

```
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;
}
```



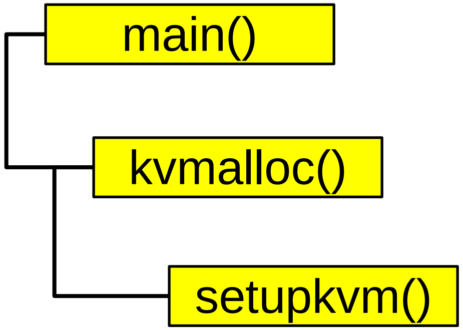
Recap: Iterate in a loop: remap physical pages

```
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;
}
```

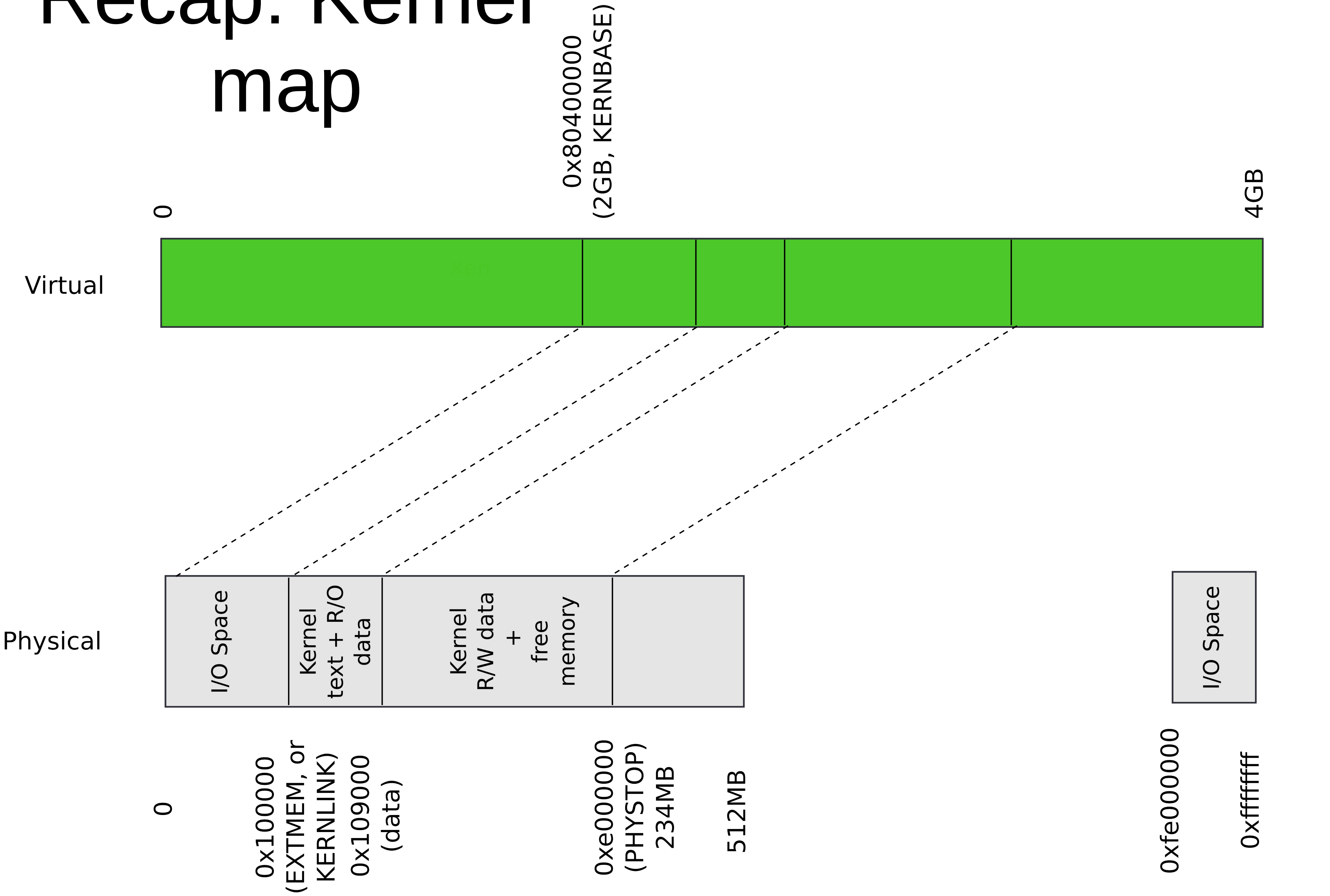


Recap: Iterate in a loop: remap physical pages

```
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;
}
```

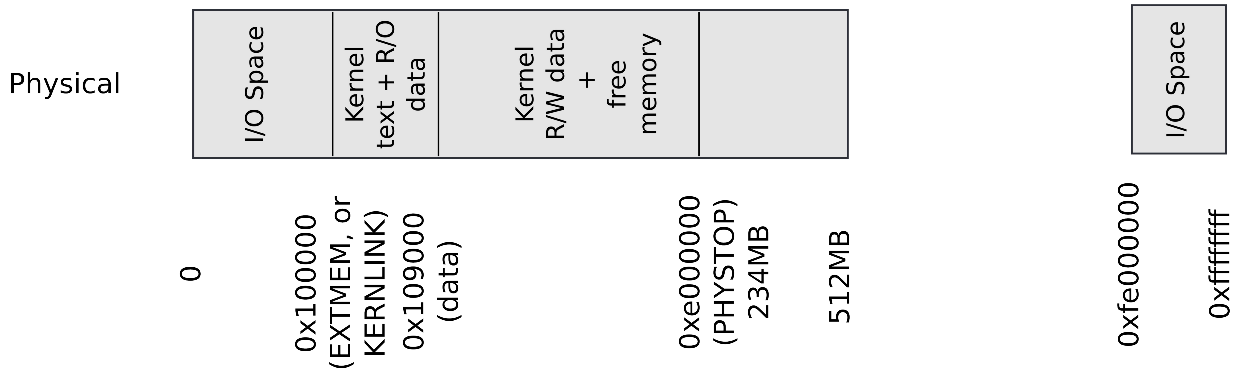


Recap: Kernel map



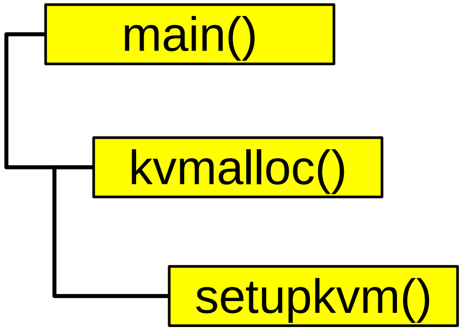
Recap: Kmap – kernel map

```
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
};
```



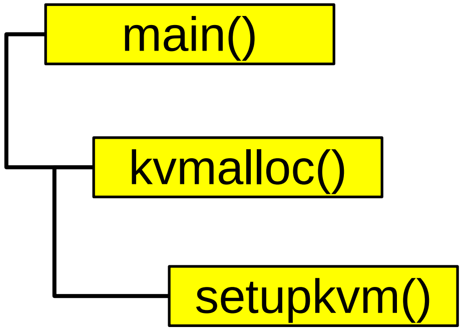
Recap: Iterate in a loop: remap physical pages

```
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;
}
```

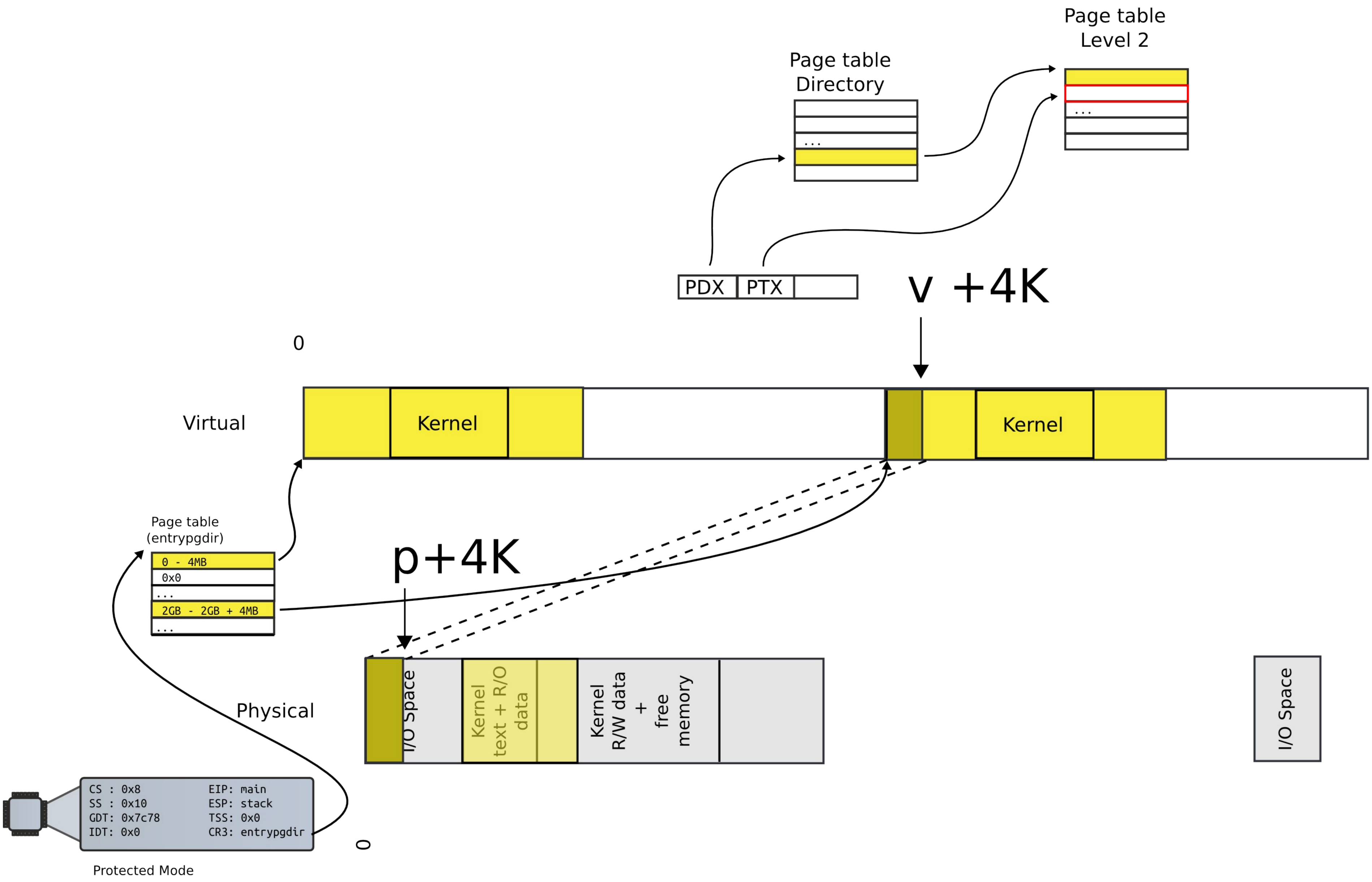


main()

kvmalloc()

setupkvm()

setupkvm(): Move to next page



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**

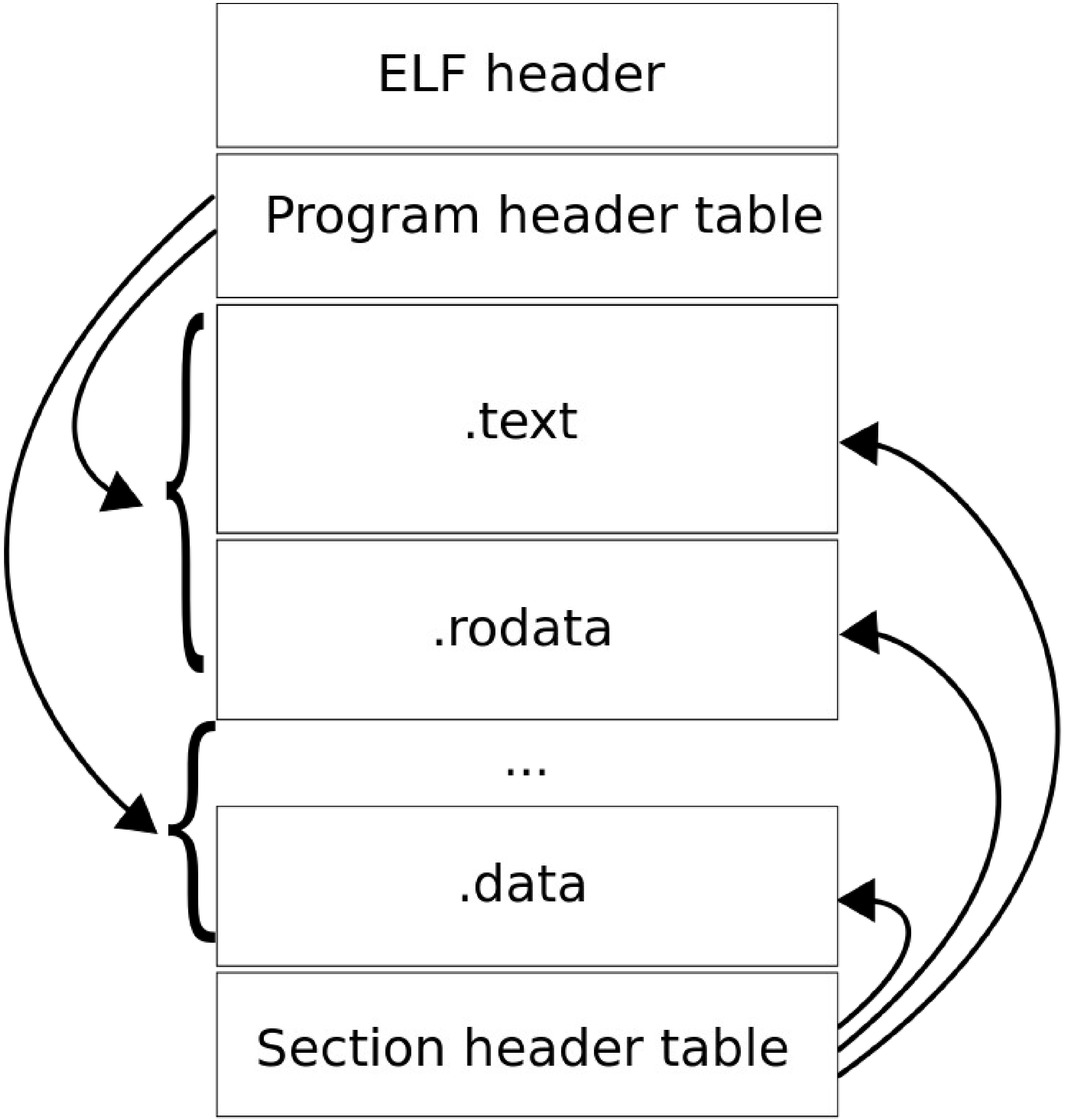
Load program into memory

```
// 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;
}
```

- Loop over all program headers

ELF binary

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



Load program into memory

```
// 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

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;
}
```

- Read one program header entry at a time

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;
}
```

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

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;
}
```

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

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;
}
```

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

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;
}
```

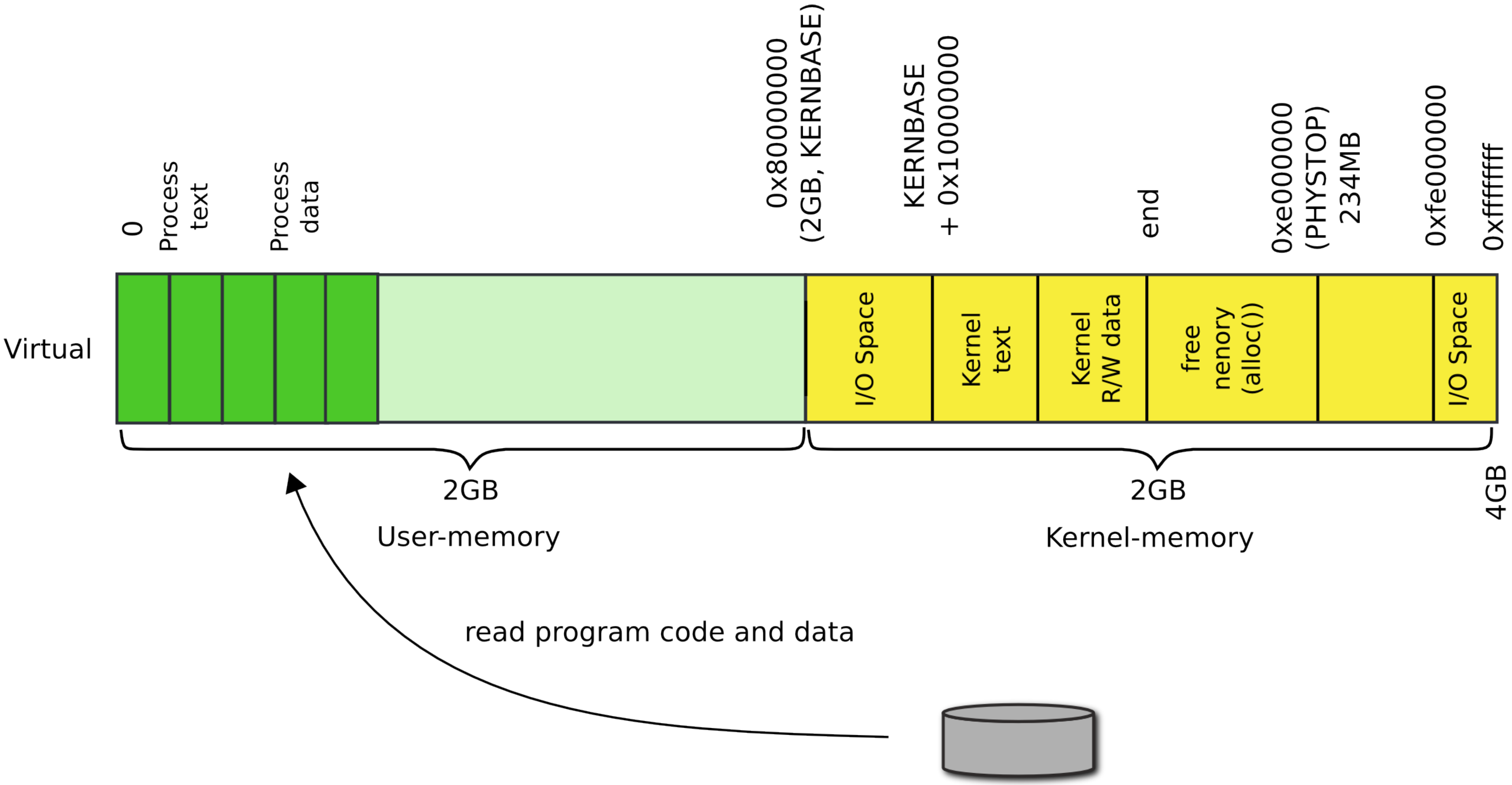
- New size of the address space

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

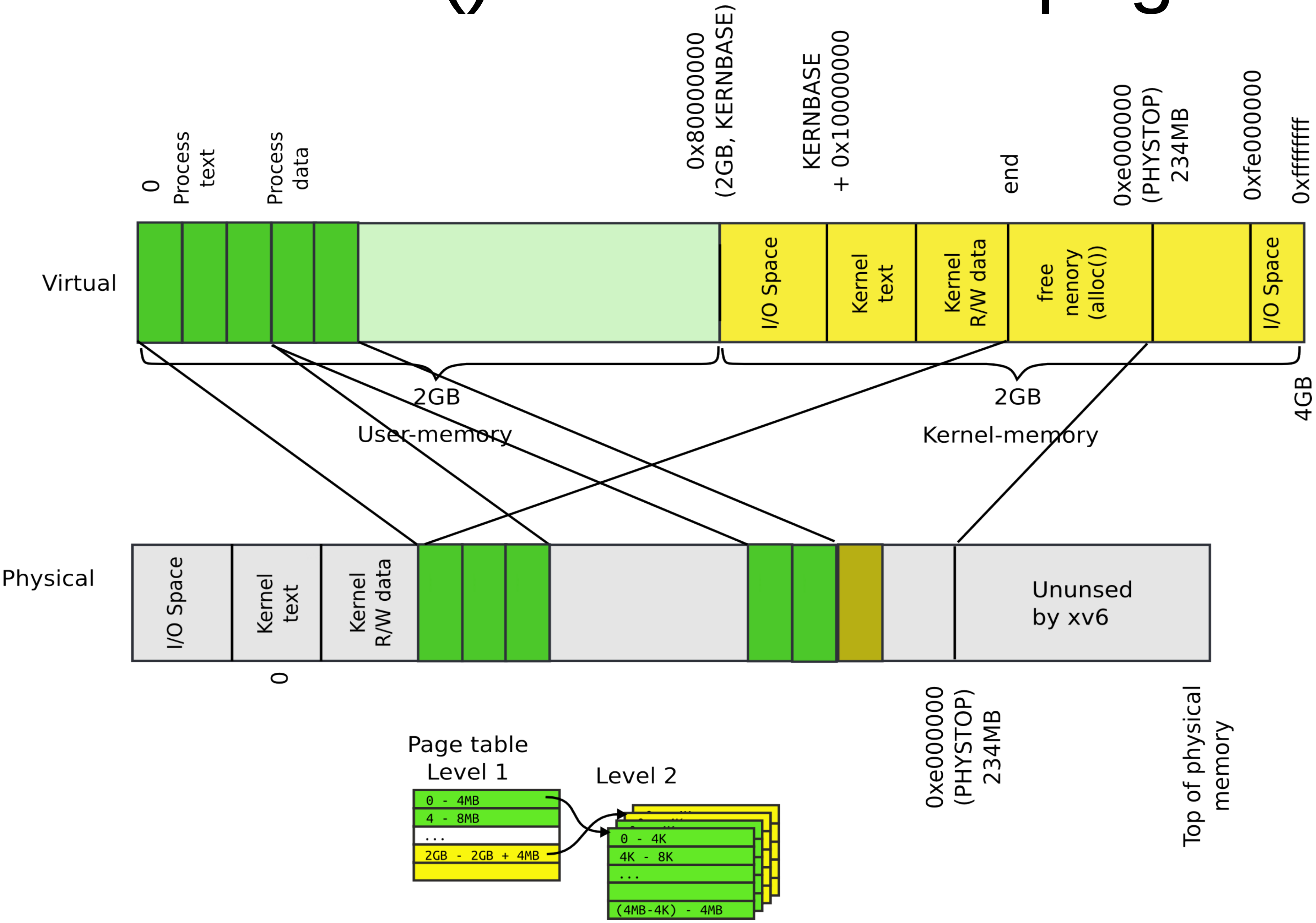


Lets take a closer look allocuvm()

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

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

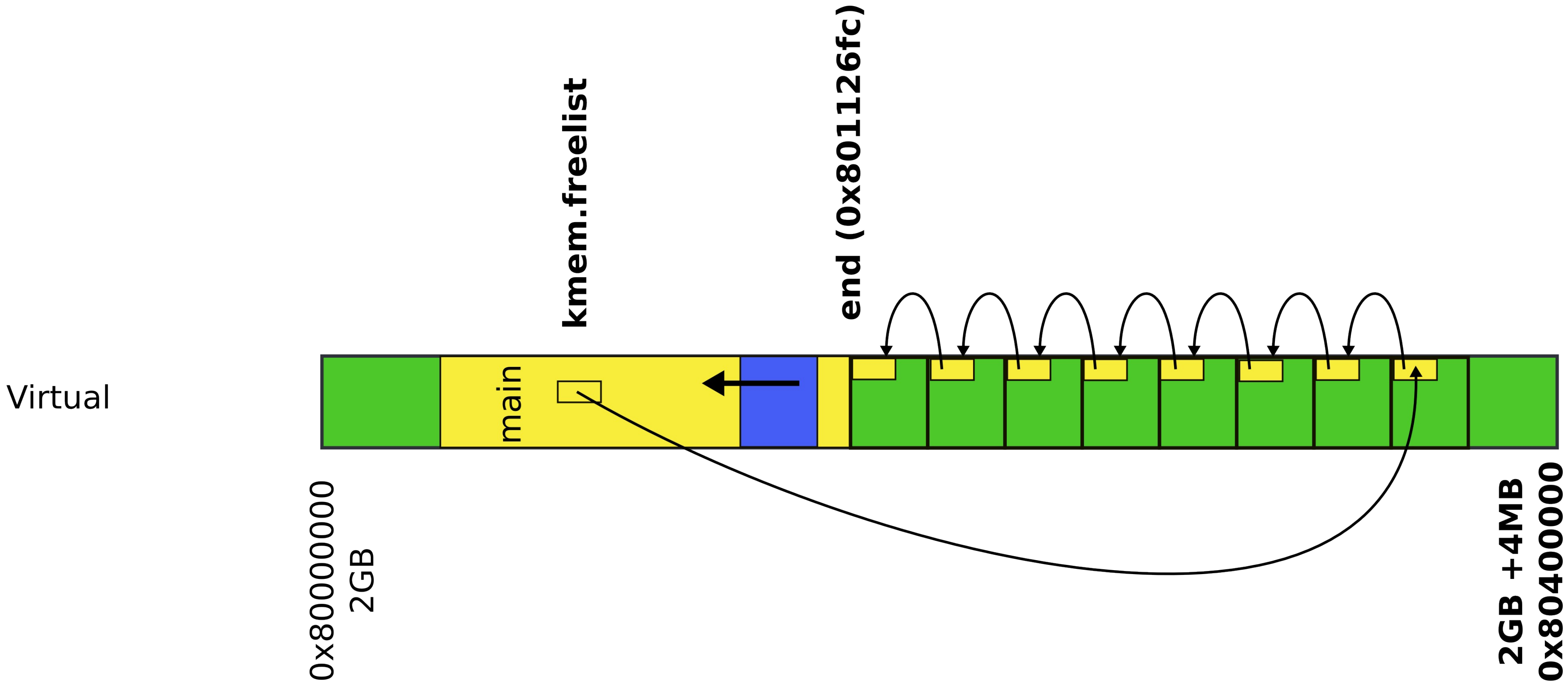
Allocate user address space

```
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;
}
```

- Allocate a new page

Where does this memory come from?

Kernel memory allocator



```
CS : 0x8          EIP: main
SS : 0x10         ESP: stack
GDT: 0x7c78       TSS: 0x0
IDT: 0x0          CR3: entrypgdir
```

Protected Mode

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;
}
```

- 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

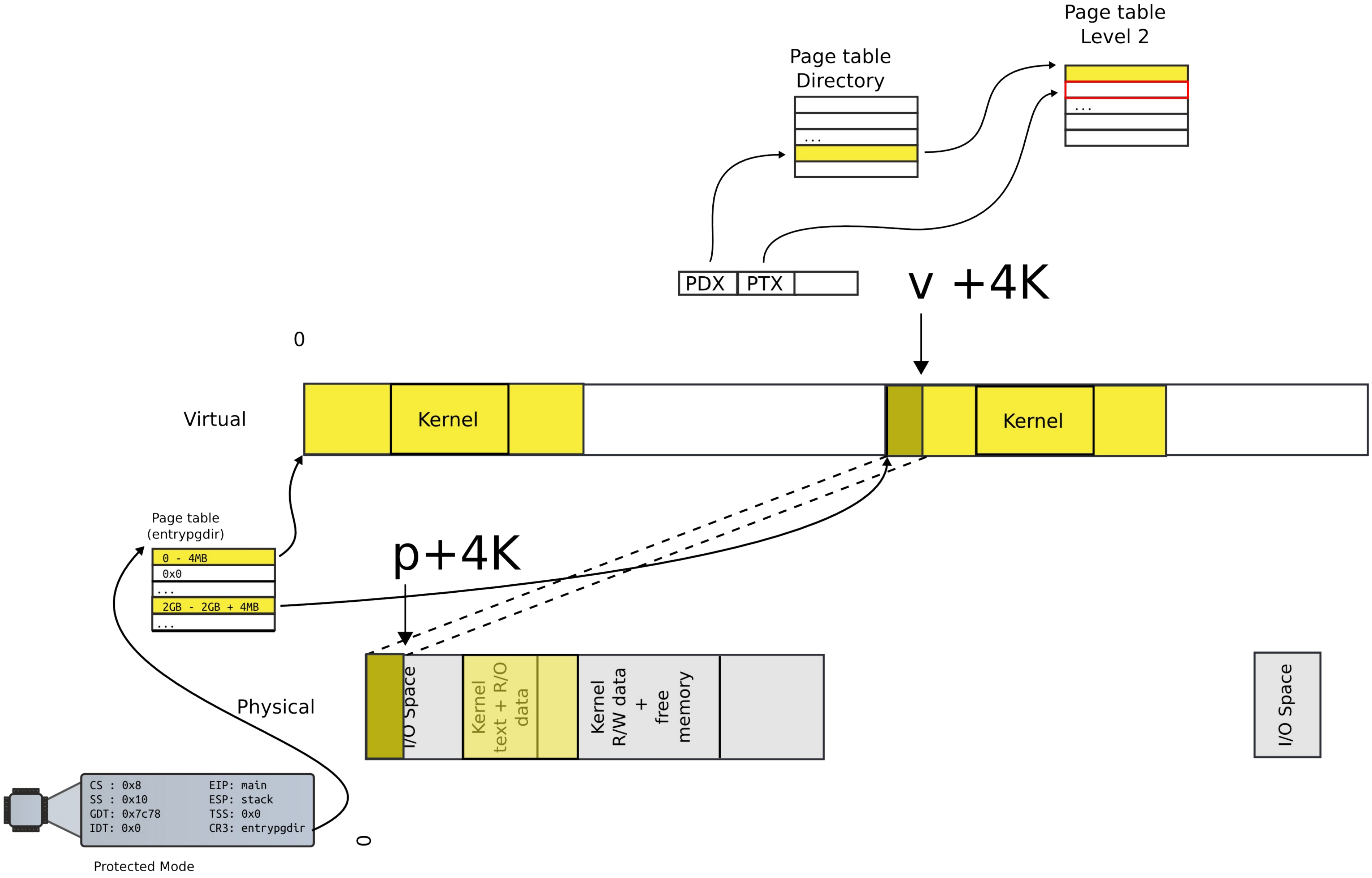
```
allocuvm(pde_t *pgdir, uint oldsz, uint newsz)
{
...
  if(newsz >= KERNBASE)
    return 0;
...
  for(; a < newsz; a += PGSIZE){
    mem = kalloc();
...
    memset(mem, 0, PGSIZE);
    if(mappages(pgdir, (char*)a, PGSIZE, V2P(mem), PTE_W|PTE_U) < 0){
...
      return 0;
    }
  }
  return newsz;
}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

- Flags
  - Writable and user-accessible

Who remembers mappages()?

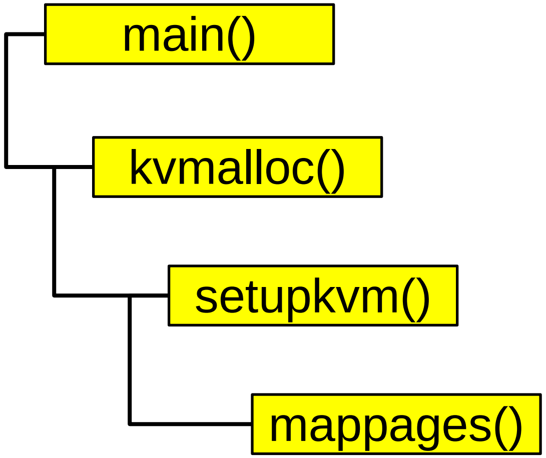
- Remember we want a region of memory to be mapped
  - i.e., appear in the page table

mappages(): map a region



```
CS : 0x8          EIP: main
SS : 0x10         ESP: stack
GDT: 0x7c78       TSS: 0x0
IDT: 0x0          CR3: entrypgdir
```

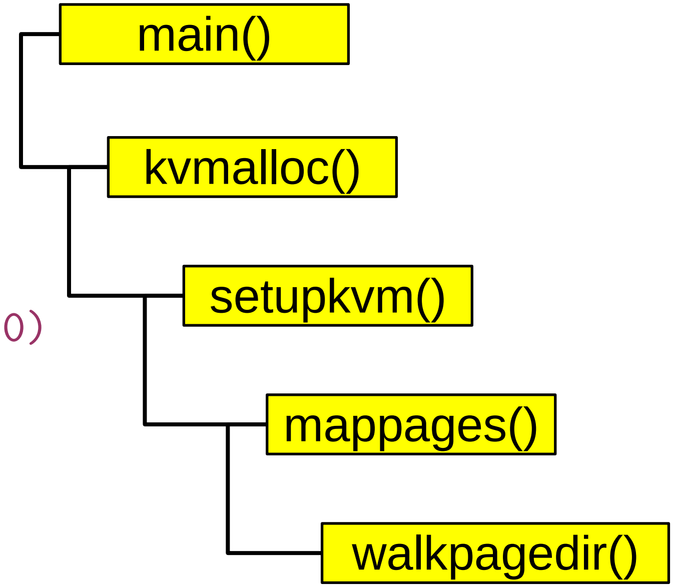
```
mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm)
{
  char *a, *last;
  pte_t *pte;

  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

```
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 back to loaduvm()

exec() – create a new process

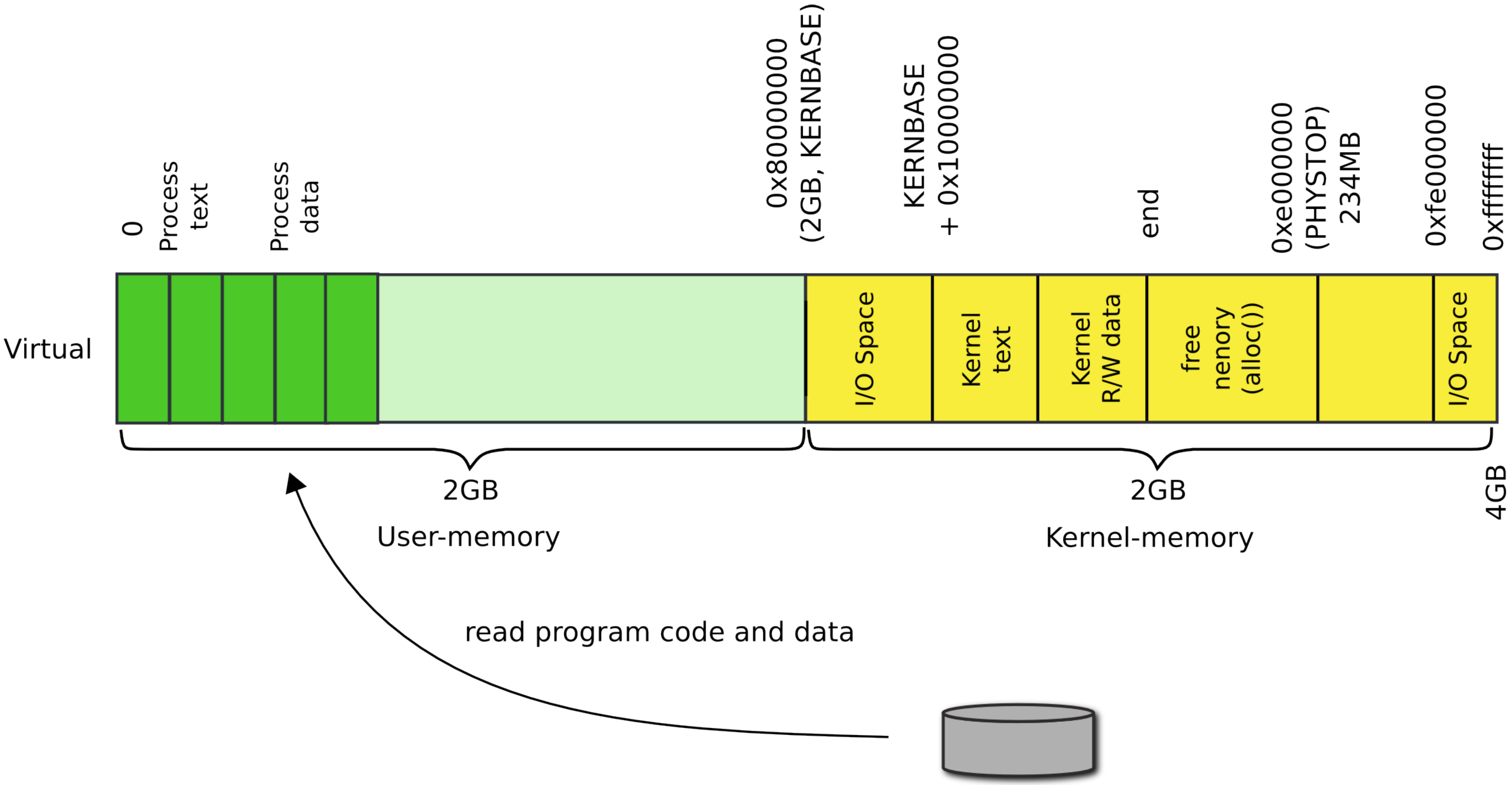
- Read process binary from disk
  - namei() takes a file path (“/bin/ls”) as an argument
  - Returns an inode
  - readi() reads the inode (file data)
- Create process address space
  - Create a page table
  - Map only kernel space
- **Load program into memory**
  - Allocate user-level pages for the program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

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;
}
```

- Why can't we use addr directly?

Load program into memory

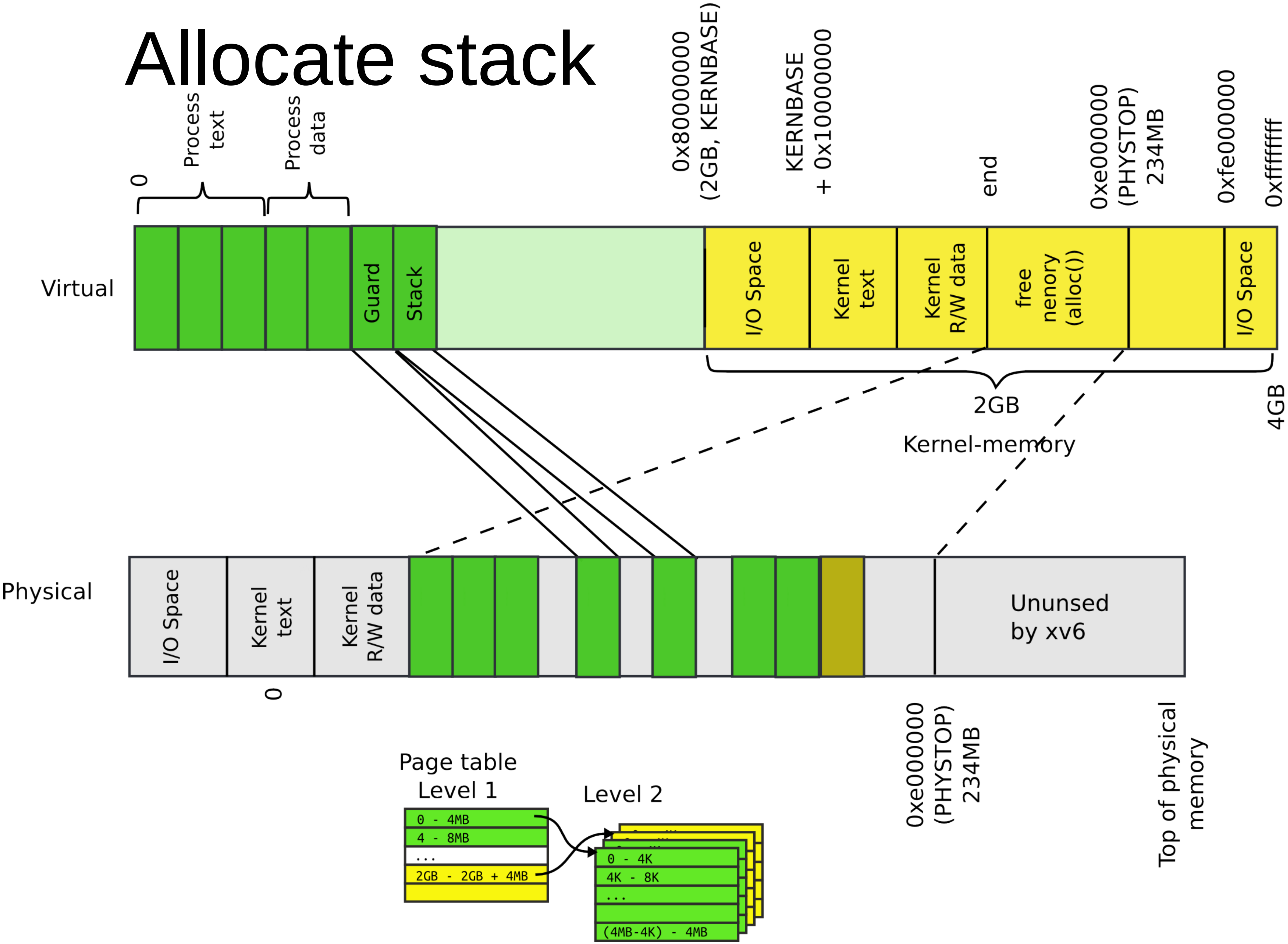
```
loaduvm(pde_t *pgdir, char *addr, struct inode *ip, uint offset, uint sz)
{
...
  for(i = 0; i < sz; i += PGSIZE){
    if((pte = walkpgdir(pgdir, addr+i, 0)) == 0)
      panic("loaduvm: address should exist");
    pa = PTE_ADDR(*pte);
    if(sz - i < PGSIZE)
      n = sz - i;
    else
      n = PGSIZE;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

```
// 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 program arguments
on the stack

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
- Allocate program stack
- Push program arguments on the stack
- **Switch page tables**

Switch page tables

- Switch page tables
- Deallocate old page table

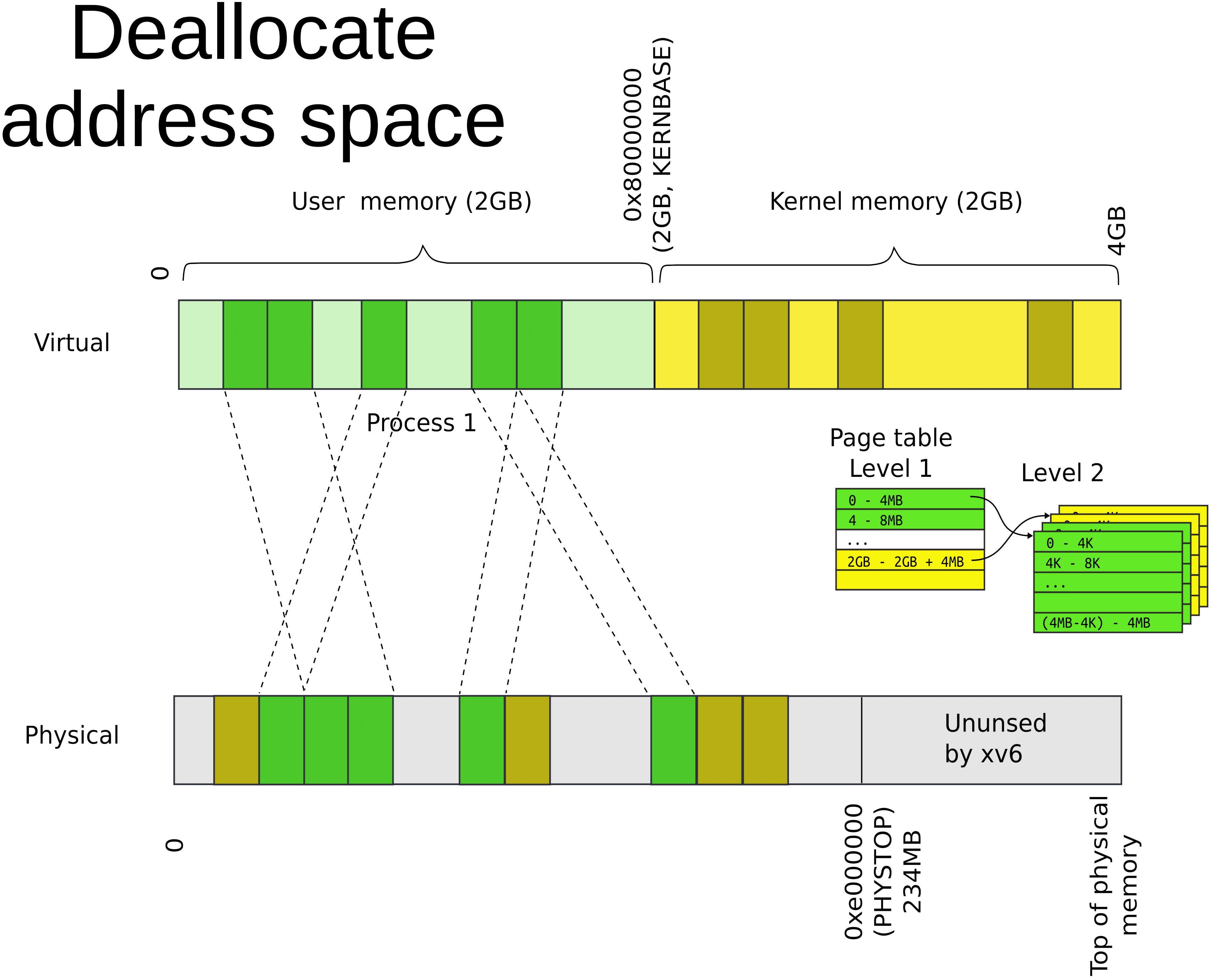
```
switchuvm(proc);
freevm(oldpgdir);
return 0;
```

Wait... which page table we are deallocating?

Wait... which page table we are deallocating?

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

Deallocate address space



Outline: deallocate process address space

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

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

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

Deallocate user address space

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

Walk page table and get pte

```
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

```
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

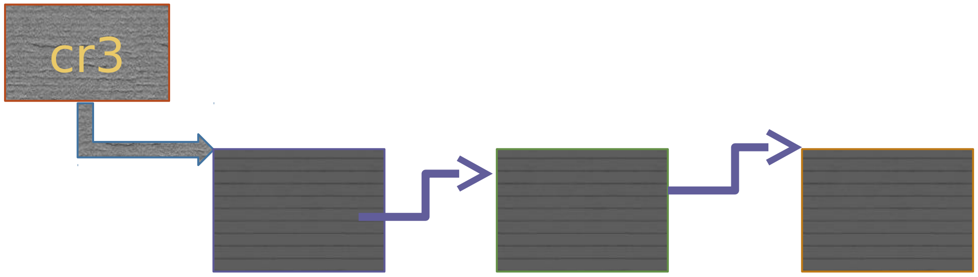
- We know how exec works!
- We can create new processes

Thank you!

Backup

TLB

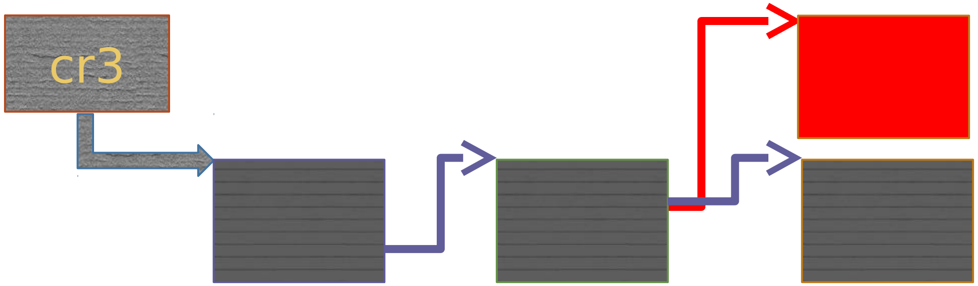
- CPU caches results of page table walks
  - In translation lookaside buffer (TLB)
- Walking page table is slow
  - Each memory access is 150-200 cycles on modern hardware
  - L3 cache access is 40 cycles



Virt	Phys
0xf0231000	0x1000
0x00b31000	0x1f000
0xb0002000	0xc1000
-	-

TLB

- TLB is a cache (in CPU)
  - It is not coherent with memory
  - If page table entry is changes, TLB remains the same and is out of sync



Virt	Phys
0xf0231000	0x1000
0x00b31000	0x1f000
0xb0002000	0xc1000
-	-

Same
Virt Addr.

No
Change!!!

Invalidating TLB

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