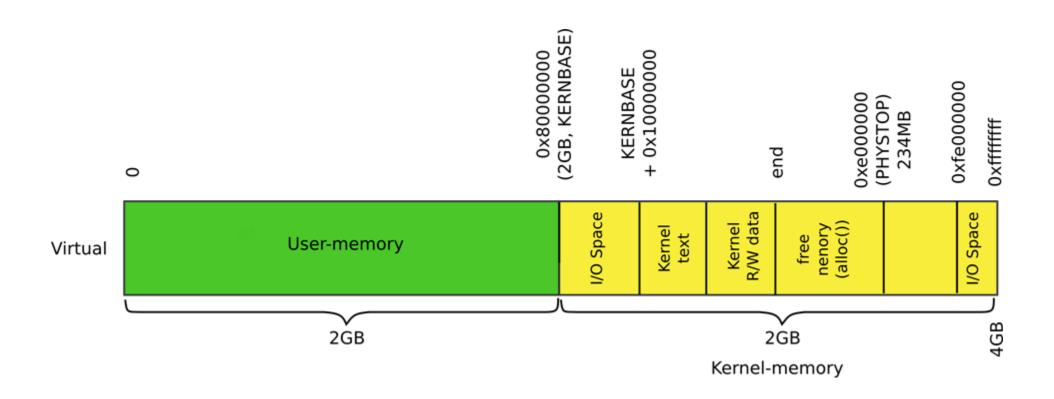
cs5460/6460: Operating Systems

Lecture: Creating Processes (exec())

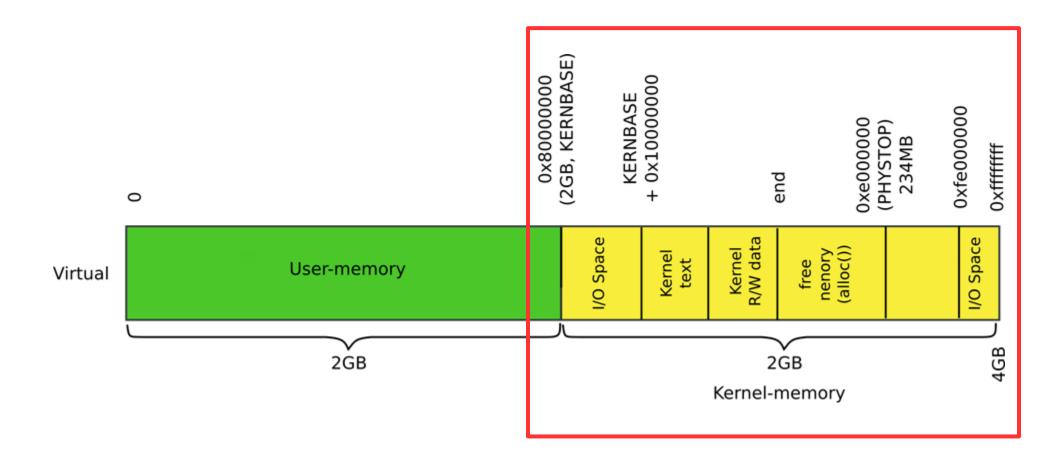
Anton Burtsev

March 2024

Recap: kernel memory



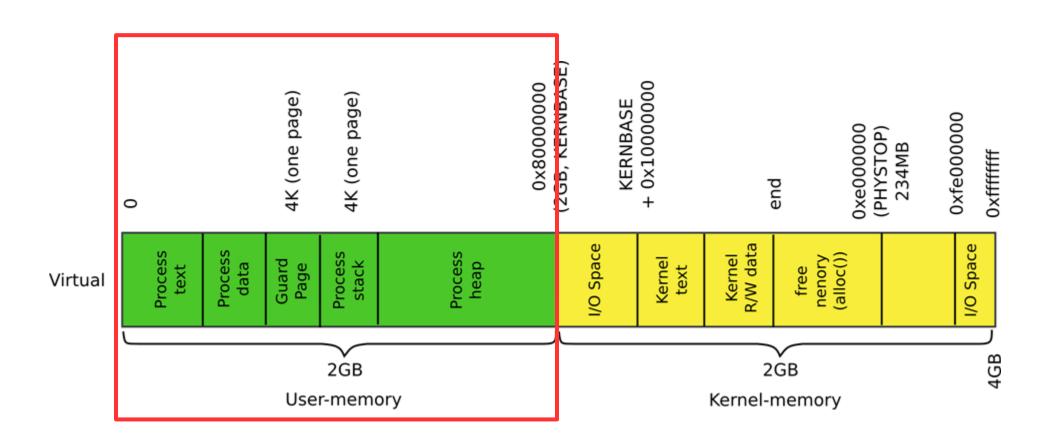
Recap: kernel memory



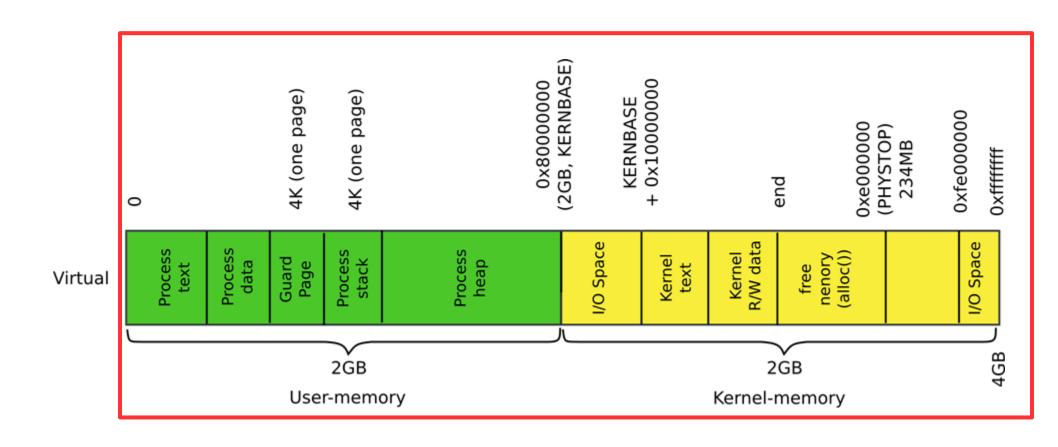
```
1317 main(void)
1318 {
1319 kinit1(end, P2V(4*1024*1024)); // phys page allocator
1320 kvmalloc(); // kernel page table
1321 mpinit(); // detect other processors
1322 lapicinit(); // interrupt controller
1323 seginit(); // segment descriptors
1324 cprintf("\ncpu%d: starting xv6\n\n", cpunum());
                // another interrupt controller
1325 picinit();
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)
```

main()

Today: process memory



Today: process memory



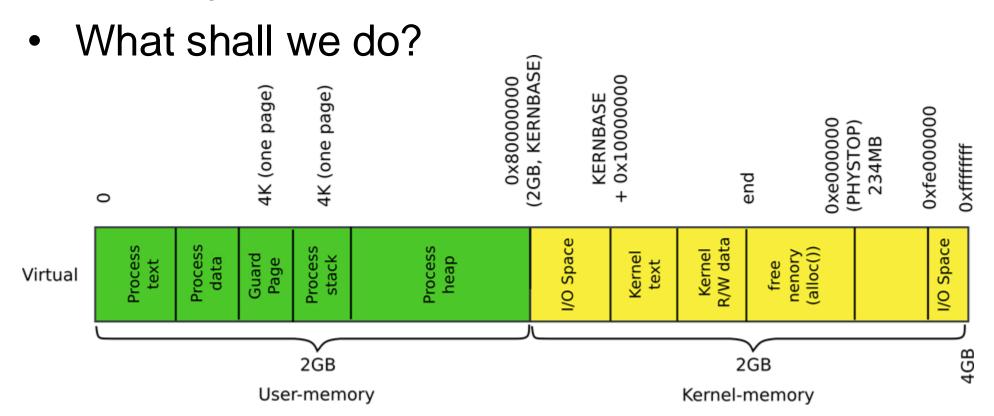
How does kernel creates new processes?

How does kernel creates new processes?

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

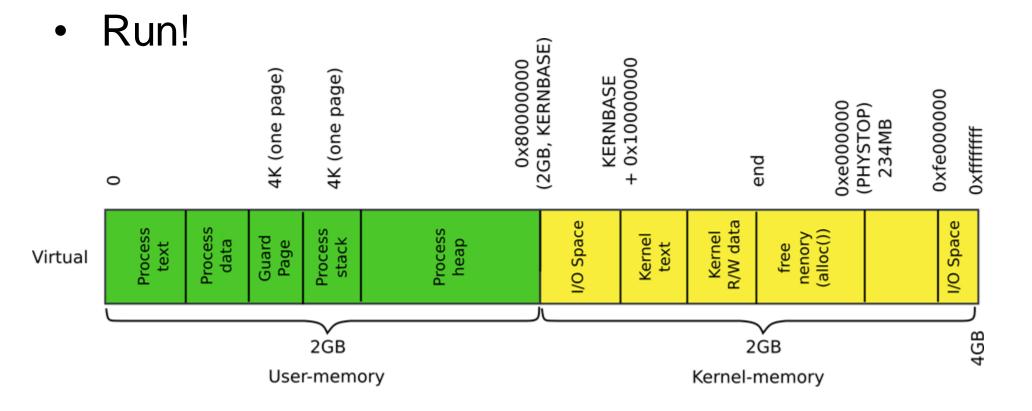
exec(): high-level outline

 We want to create the following memory layout for the process



exec(): high-level outline

- Load program from disk
- Create user-stack



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
 - I.e., a page table for the process

```
6309 int
6310 exec(char *path, char **argv)
6311 {
. . .
6321 if((ip = namei(path)) == 0){
       end_op();
6322
6323
       return -1;
6324 }
6328 // Check ELF header
6329 if(readi(ip, (char*)&elf, 0, sizeof(elf)) <
                       sizeof(elf))
6330
       goto bad;
6331 if(elf.magic != ELF_MAGIC)
       goto bad;
6332
```

exec(): locate inode

```
6309 int
6310 exec(char *path, char **argv)
6311 {
. . .
6321 if((ip = namei(path)) == 0){
      end_op();
6322
6323
      return -1;
6324 }
6328 // Check ELF header
6329 if(readi(ip, (char*)&elf, 0, sizeof(elf)) <
                       sizeof(elf))
6330
       goto bad;
      if(elf.magic != ELF_MAGIC)
6331
       goto bad;
6332
```

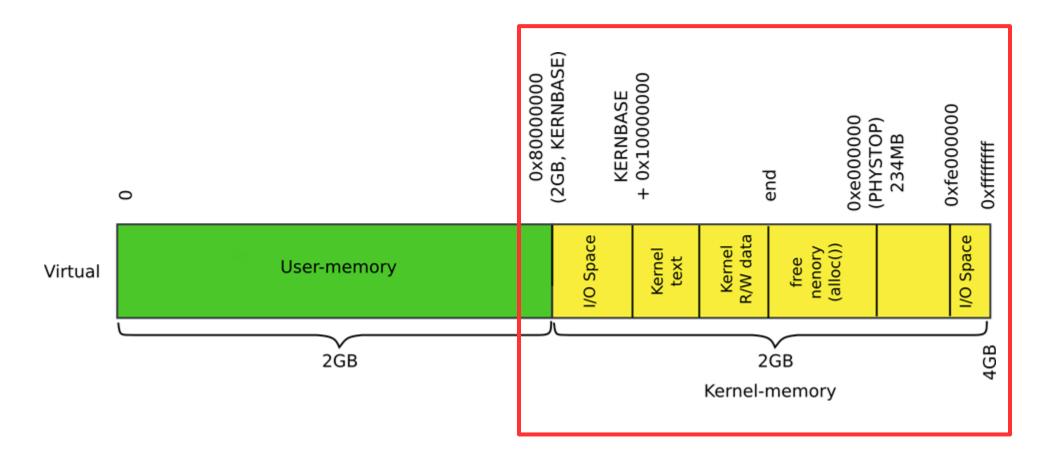
exec(): check ELF header

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

Each process maps kernel in its address space



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 setupkym(void)
1838 {
1839 pde t *pgdir;
1840 struct kmap *k;
1841
     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: Allocate page table directory

```
main()

kvmalloc()

setupkvm()
```

```
1836 pde_t*
1887 setupkym(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);
      for(k = kmap; k < kmap[NELEM(kmap)]; k++)
       if(mappages(pgdir, k->virt, k->phys_end - k->phys_start,
1848
1849
             (uint)k->phys start, k->perm) < 0
1850
        return 0;
1851 return pgdir;
1852 }
```

Recap: Iterate in a loop: remap physical pages

main()

kvmalloc()

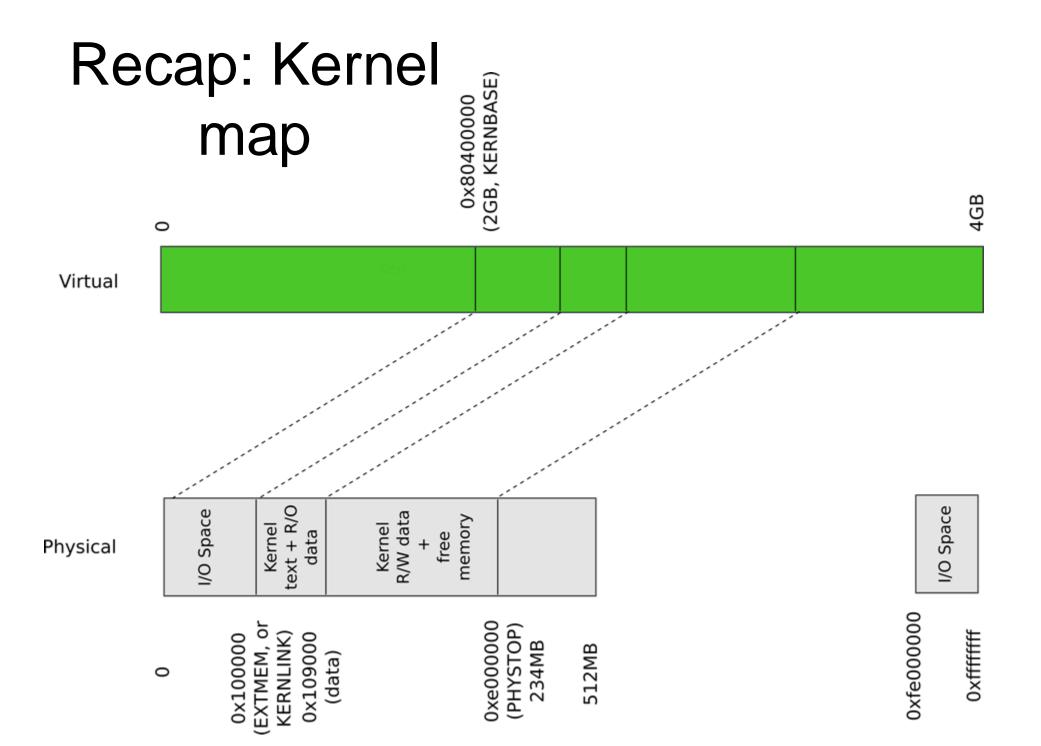
setupkvm()

```
Recap: Iterate in a
1836 pde_t*
1887 setupkym(void)
                                   loop: remap physical
1838 {
1839 pde t *pgdir;
                                                     pages
1840 struct kmap *k;
1841
1842 if((pgdir = (pde t^*)kalloc()) == 0)
1843
      return 0;
1844 memset(pgdir, 0, PGSIZE);
. . .
     for(k = kmap; k < &kmap[NELEM(kmap)]; k++)</pre>
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 }
```

main()

kvmalloc()

setupkvm()



Recap: Kmap – kernel map

```
1823 static struct kmap {
                                                                           Kernel
text + R/O
                                                                                       Kernel
R/W data
                                                                    I/O Space
                                                                                                memory
1824
                                                                                             free
         void *virt;
                                                    Physical
1825
         uint phys start;
                                                                                                  0xe000000
                                                                         EXTMEM, or
                                                                                                     PHYSTOP)
                                                                             KERNLINK)
                                                                       0×100000
                                                                               0×109000
                                                                                                       234MB
                                                                                                             512MB
1826
         uint phys end;
1827
         int perm;
1828 } kmap[] = {
         { (void*)KERNBASE, 0, EXTMEM, PTE W}, // I/O space
1829
1830
         { (void*)KERNLINK, V2P(KERNLINK), V2P(data), 0},//text+rodata
1831
         { (void*)data, V2P(data), PHYSTOP, PTE W}, // kern data+memory
         { (void*)DEVSPACE, DEVSPACE, 0, PTE W}, // more devices
1832
1833 };
```

I/O Space

0xfe0000000

0×fffffff

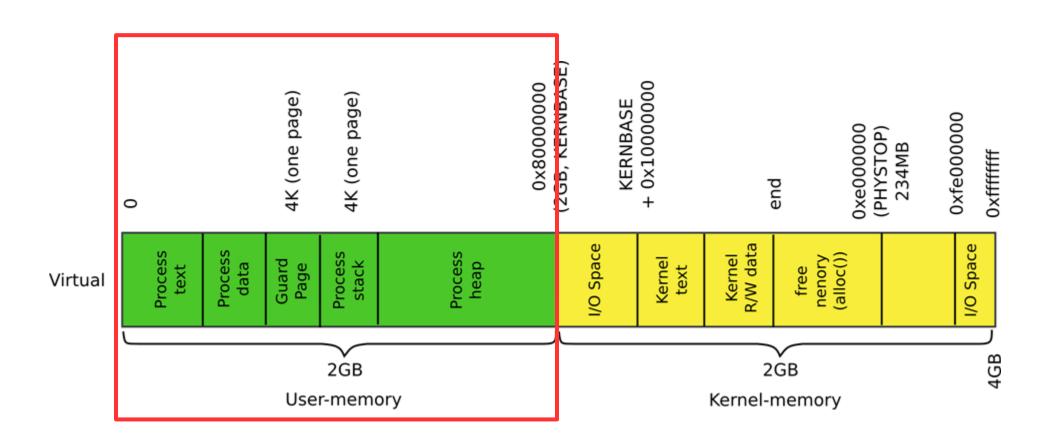
```
Recap: Iterate in a
1836 pde t*
1887 setupkym(void)
                                   loop: remap physical
1838 {
1839 pde t *pgdir;
                                                     pages
1840 struct kmap *k;
                                                                       main()
1841
1842 if((pgdir = (pde t^*)kalloc()) == 0)
                                                                       kvmalloc()
1843
      return 0;
1844 memset(pgdir, 0, PGSIZE);
                                                                          setupkvm()
     for(k = kmap; k < &kmap[NELEM(kmap)]; k++)</pre>
      if(mappages(pgdir, k->virt, k->phys_end - k->phys_start,
1848
1849
           (uint)k->phys start, k->perm) < 0
1850
       return 0;
1851 return pgdir;
1852 }
```

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

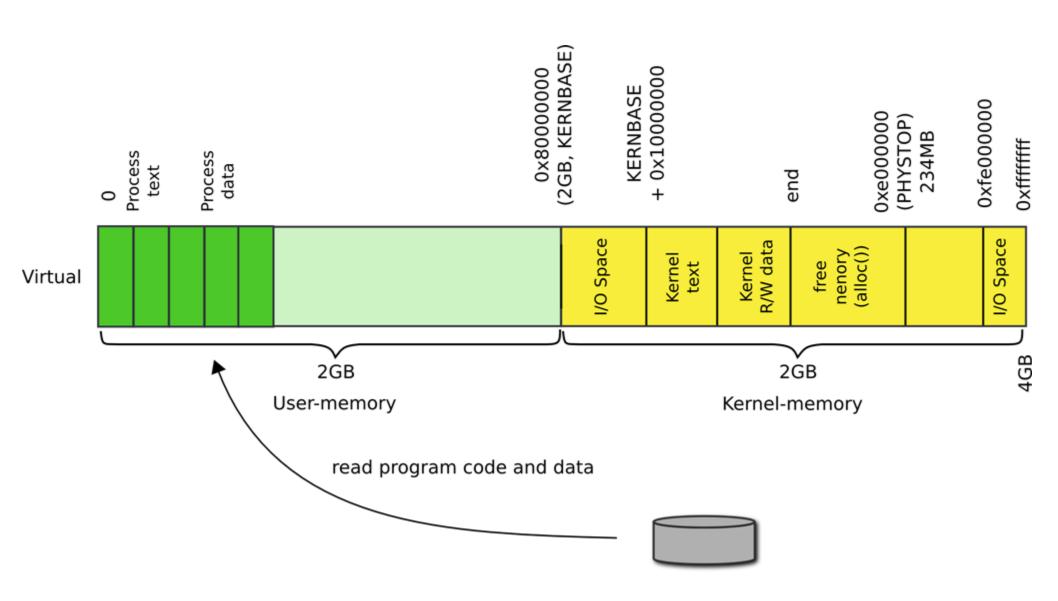
User part of the process memory



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)
                                                        Program loading loop
6311 {
6337 // Load program into memory.
6338 \text{ 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))
        goto bad;
6341
       if((sz = allocuvm(pgdir, sz, ph.vaddr + ph.memsz)) == 0)
6348
6349
        goto bad;
6350
       if(ph.vaddr % PGSIZE != 0)
6351
        goto bad;
       if(loaduvm(pgdir, (char*)ph.vaddr, ip, ph.off, ph.filesz) < 0)
6352
6353
        goto bad;
                                           Loop over all program
6354 }
                                            headers
```

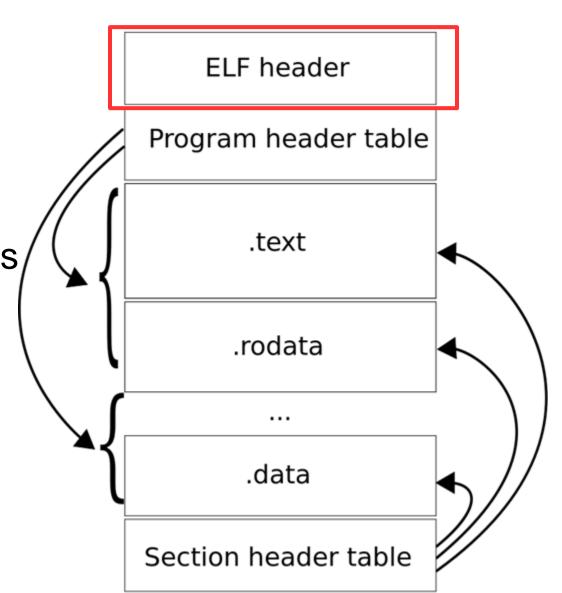
ELF object file

ELF header

Program header table

 Each entry describes a section of a program

Instruction, data



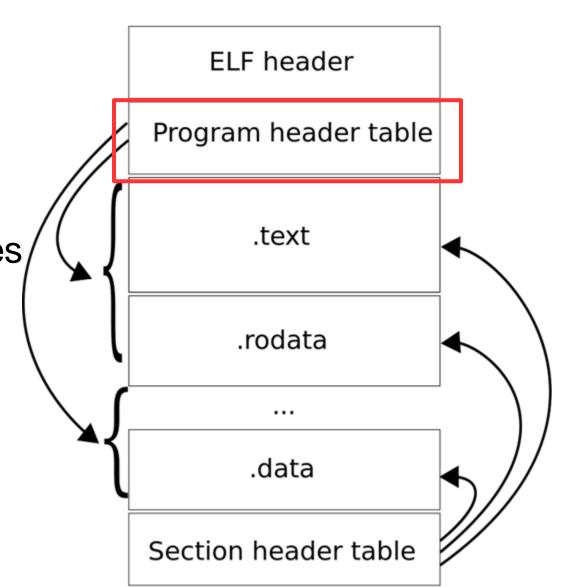
ELF object file

ELF header

Program header table

 Each entry describes a section of a program

Instruction, data



```
6337 // Load program into memory.
6338 \text{ 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))
       goto bad;
6341
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

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

```
6337 // Load program into memory.
                                                       Program loading loop
6338 \text{ 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))
       goto bad;
6341
6348 if((sz = allocuvm(pgdir, sz, ph.vaddr + ph.memsz)) == 0)
6349
       goto bad;
6350 if(ph.vaddr % PGSIZE != 0)
6351
       goto bad;
6352 if(loaduvm(pgdir, (char*)ph.vaddr, ip, ph.off, ph.filesz) < 0)
6353
       goto bad;
6354 }
```

 Read one program header entry at a time

```
6337 // Load program into memory.
                                                       Program loading loop
6338 \text{ 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))
       goto bad;
6341
6348 if((sz = allocuvm(pgdir, sz, ph.vaddr + ph.memsz)) == 0)
6349
       goto bad;
6350 if(ph.vaddr % PGSIZE != 0)
6351
       goto bad;
6352 if(loaduvm(pgdir, (char*)ph.vaddr, ip, ph.off, ph.filesz) < 0)
6353
       goto bad;
6354 }
```

- Read one program header entry at a time
- Each time increment offset by the size of the program header entry

```
6337 // Load program into memory.
                                                       Program loading loop
6338 \text{ 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))
       goto bad;
6341
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 }
```

Allocate pages for text, data, etc.

```
6337 // Load program into memory.
6338 \text{ 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

- sz is the size of the address space
- Initially it's 0

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

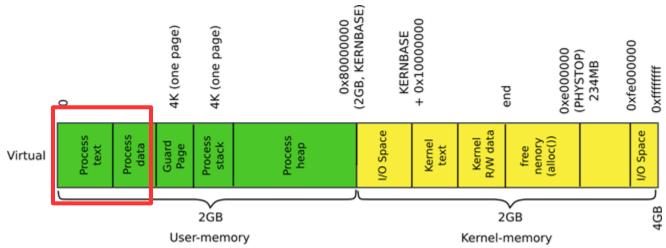
New size of the address space

```
6337 // Load program into memory.
                                                       Program loading loop
6338 \text{ 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))
       goto bad;
6341
6348 if((sz = allocuvm(pgdir, sz, ph.vaddr + ph.memsz)) == 0)
6349
       goto bad;
6350 if(ph.vaddr % PGSIZE != 0)
6351
       goto bad;
6352 if(loaduvm(pgdir, (char*)ph.vaddr, ip, ph.off, ph.filesz) < 0)
6353
       goto bad;
6354 }
```

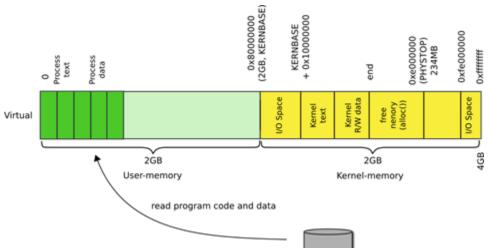
Load program section from disk

Two main functions

allocuvm() -- allocate and map user-memory

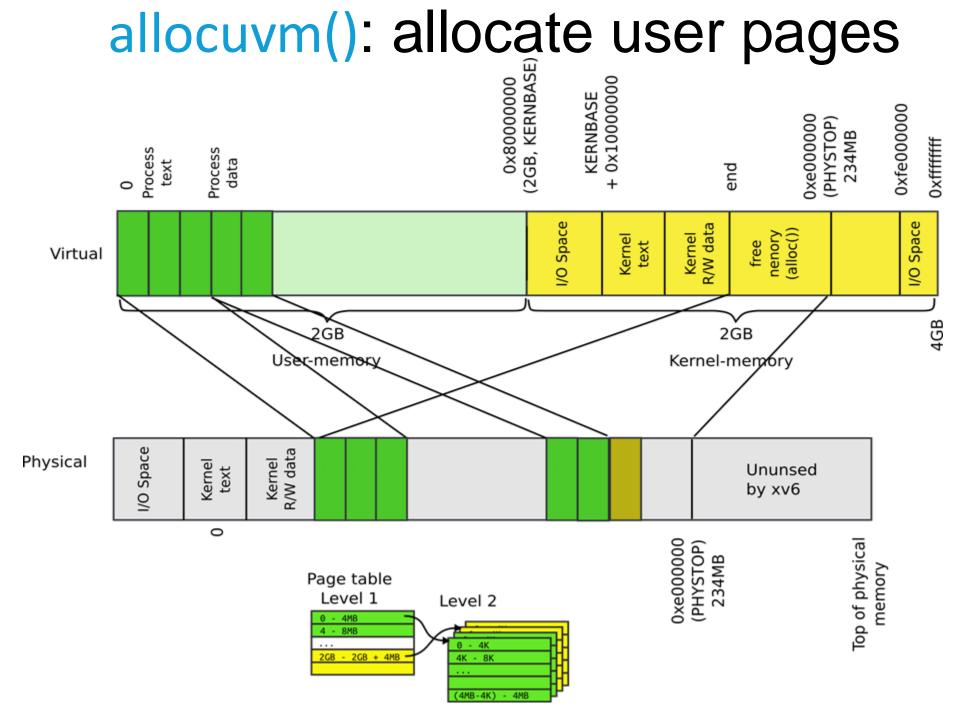


 loaduvm() -- load user-memory with data from disk



Lets take a closer look

allocuvm()



```
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){
      mem = kalloc();
1965
      memset(mem, 0, PGSIZE);
1971
      if(mappages(pgdir, (char*)a, PGSIZE, V2P(mem), PTE W|PTE U) < 0){
1972
1976
       return 0;
1977
1978 }
                                              2GB
1979 return newsz;
1980 }
```

New size can't be over

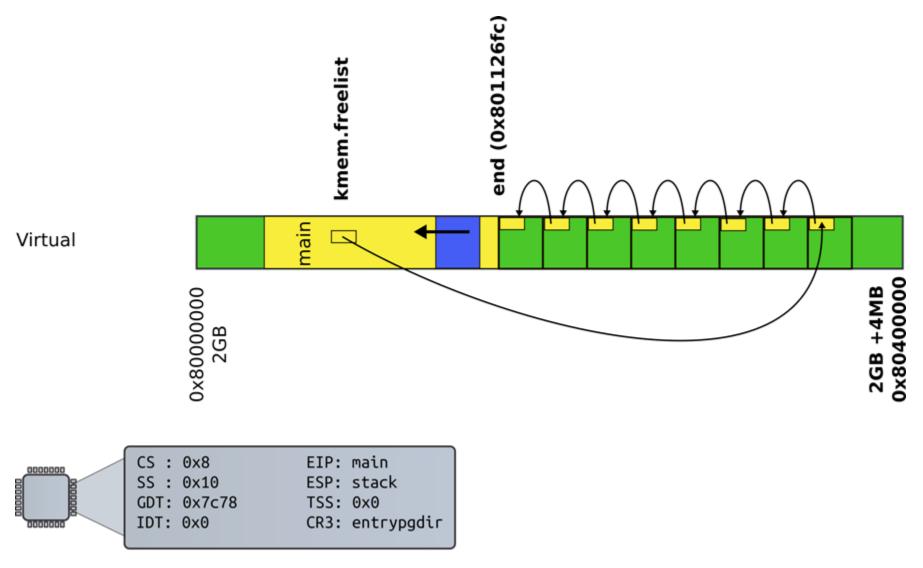
```
1953 allocuvm(pde t *pgdir, uint oldsz, uint newsz)
1954 {
1958 if(newsz >= KERNBASE)
1959
      return 0;
1963 a = PGROUNDUP(oldsz);
1964 for(; a < newsz; a += PGSIZE){
1965
      mem = kalloc();
1971
       memset(mem, 0, PGSIZE);
       if(mappages(pgdir, (char*)a, PGSIZE, V2P(mem), PTE W|PTE U) < 0){
1972
1976
        return 0;
1977
1978 }
1979 return newsz;
1980 }
```

Start with the old size rounded up to the next page

```
1953 allocuvm(pde t *pgdir, uint oldsz, uint newsz)
1954 {
                                                    Allocate user address
1958 if(newsz >= KERNBASE)
                                                                   space
1959
     return 0;
1963 a = PGROUNDUP(oldsz);
1964 for(; a < newsz; a += PGSIZE){
      mem = kalloc();
1965
      memset(mem, 0, PGSIZE);
1971
      if(mappages(pgdir, (char*)a, PGSIZE, V2P(mem), PTE W|PTE U) < 0){
1972
1976
       return 0;
1977
                                       Allocate a new page
1978 }
1979 return newsz;
1980 }
```

```
1953 allocuvm(pde t *pgdir, uint oldsz, uint newsz)
1954 {
                                                  Allocate user address
1958 if(newsz >= KERNBASE)
                                                                 space
1959
     return 0;
1963 a = PGROUNDUP(oldsz);
1964 for(; a < newsz; a += PGSIZE){
     mem = kalloc();
1965
     memset(mem, 0, PGSIZE);
1971
     if(mappages(pgdir, (char*)a, PGSIZE, V2P(mem), PTE W|PTE U) < 0){
1972
                                    Allocate a new page
1976
      return 0;
                                    Where does this memory come from?
1977
1978 }
                                    Poll: PollEv.com/antonburtsev
1979 return newsz;
1980 }
```

Kernel memory allocator



Protected Mode

```
1953 allocuvm(pde t *pgdir, uint oldsz, uint newsz)
1954 {
                                                  Allocate user address
1958 if(newsz >= KERNBASE)
1959
     return 0;
1963 a = PGROUNDUP(oldsz);
1964 for(; a < newsz; a += PGSIZE){
1965
     mem = kalloc();
     memset(mem, 0, PGSIZE);
1971
     if(mappages(pgdir, (char*)a, PGSIZE, V2P(mem), PTE W|PTE U) < 0){
1972
1976
      return 0;
1977
                                       Make sure the page is clean
1978 }
                                       Don't leak data
1979 return newsz;
1980 }
```

space

```
1953 allocuvm(pde t *pgdir, uint oldsz, uint newsz)
1954 {
                                                     Allocate user address
1958 if(newsz >= KERNBASE)
                                                                    space
1959
      return 0;
1963 a = PGROUNDUP(oldsz);
1964 for(; a < newsz; a += PGSIZE){
1965
      mem = kalloc();
      memset(mem, 0, PGSIZE);
1971
      if(mappages(pgdir, (char*)a, PGSIZE, V2P(mem), PTE W|PTE U) < 0){
1972
1976
       return 0;
1977
                                        Map the page
1978 }
1979 return newsz;
1980 }
```

```
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);
       if(mappages(pgdir, (char*)a, PGS ZE, V2P(mem), PTE_W|PTE_U) < 0){
1972
1976
        return 0;
1977
1978 }
1979 return newsz;
1980 }
```

Take page table directory as an argument

```
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){
      mem = kalloc();
1965
1971
       memset(mem, 0, PGSIZE);
       if(mappages(pgdir, (char*)a, PGS ZE, V2P(men)), PTE_W|PTE_U) < 0){
1972
1976
        return 0;
1977
1978 }
                                              map
1979 return newsz;
1980 }
```

 Virtual address of the page to map

```
1953 allocuvm(pde t *pgdir, uint oldsz, uint newsz)
1954 {
                                                   Allocate user address
1958 if(newsz >= KERNBASE)
                                                                  space
1959
     return 0;
1963 a = PGROUNDUP(oldsz);
1964 for(; a < newsz; a += PGSIZE){
     mem = kalloc();
1965
      memset(mem, 0, PGSIZE);
1971
      if(mappages(pgdir, (char*)a, PGSIZE, V2P(mem), PTE_W|PTE_U) < 0){
1972
1976
       return 0;
1977
                                      Size of the region
1978 }
                                       One page!
1979 return newsz;
1980 }
```

```
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);
       if(mappages(pgdir, (char*)a, PGSIZE, V2P(mem), PTE_W|PTE_U) < 0){
1972
1976
        return 0;
1977
1978 }
1979 return newsz;
1980 }
```

Physical address of the page we're mapping

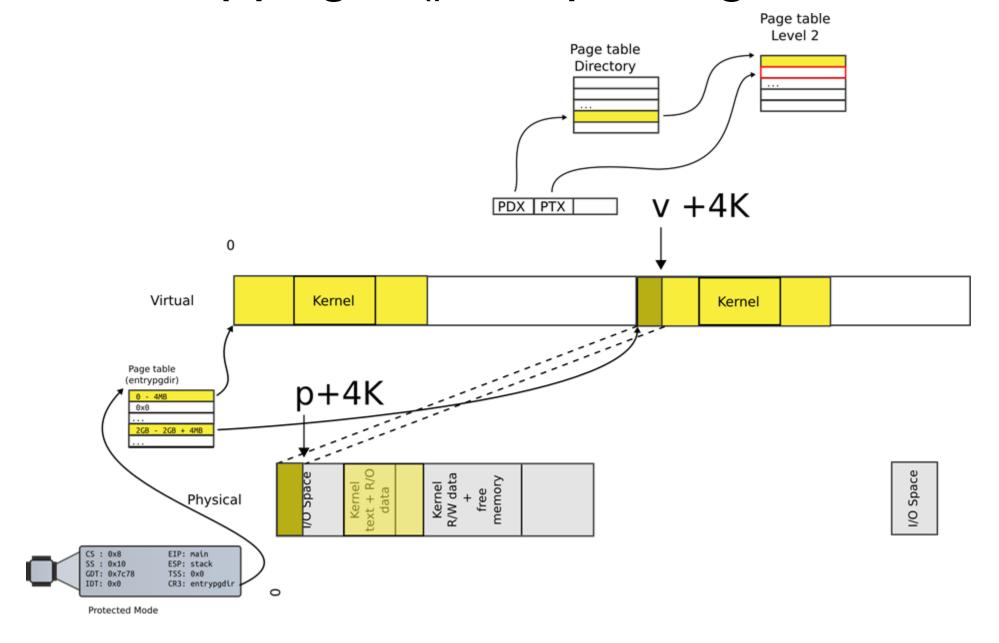
```
1953 allocuvm(pde t *pgdir, uint oldsz, uint newsz)
1954 {
                                                  Allocate user address
1958 if(newsz >= KERNBASE)
                                                                 space
1959
     return 0;
1963 a = PGROUNDUP(oldsz);
1964 for(; a < newsz; a += PGSIZE){
     mem = kalloc();
1965
     memset(mem, 0, PGSIZE);
1971
     if(mappages(pgdir, (char*)a, PGSIZE, V2P(mem), PTE W|PTE U) < 0){
1972
1976
      return 0;
1977
                                       Flags
1978 }
                                       Writable and user-accessible
1979 return newsz;
```

1980 }

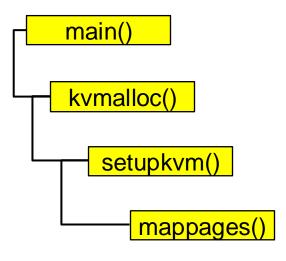
Who remembers mappages()?

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

mappages(): map a region



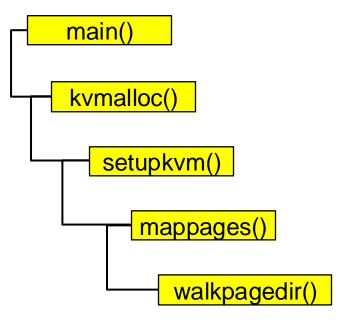
```
1779 mappages(pde t *pgdir, void *va, uint size, uint pa, int perm)
1780 {
1781 char *a, *last;
1782 pte t *pte;
1783
1784 a = (char*)PGROUNDDOWN((uint)va);
1785 last = (char^*)PGROUNDDOWN(((uint)va) + size - 1);
1786 for(;;){
1787
       if((pte = walkpgdir(pgdir, a, 1)) == 0)
1788
        return -1;
       if(*pte & PTE P)
1789
        panic("remap");
1790
       *pte = pa | perm | PTE P;
1791
       if(a == last)
1792
1793
        break;
       a += PGSIZE;
1794
1795
       pa += PGSIZE;
1796 }
1797 return 0;
1798 }
```



Lookup the page table entry

```
1754 walkpgdir(pde t *pgdir, const void *va, int alloc)
1755 {
       pde t*pde;
1756
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
       return &pgtab[PTX(va)];
1772
1773 }
```

Walk page table



```
1953 allocuvm(pde t *pgdir, uint oldsz, uint newsz)
1954 {
                                                    Allocate user address
1958 if(newsz >= KERNBASE)
                                                                   space
1959
      return 0;
1963 a = PGROUNDUP(oldsz);
1964 for(; a < newsz; a += PGSIZE {
1965
      mem = kalloc();
      memset(mem, 0, PGSIZE);
1971
      if(mappages(pgdir, (char*)a, PGSIZE, V2P(mem), PTE W|PTE U) < 0){
1972
1976
       return 0;
1977
                                        Continue in a loop
1978 }
1979 return newsz;
1980 }
```

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

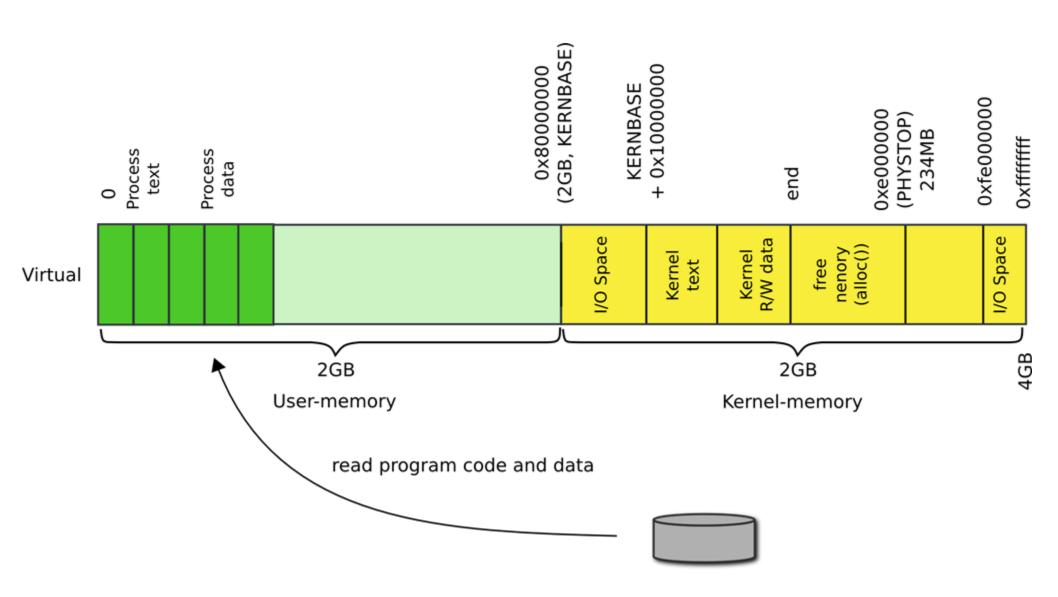
```
6337 // Load program into memory.
6338 \text{ sz} = 0;
6339 for(i=0, off=elf.phoff; i<elf.phnum; i++, off+=sizeof(ph)){
6340 if(readi(ip, (char*)&ph, off, sizeof(ph)) != sizeof(ph))
6341
       goto bad;
6348 if((sz = allocuvm(pgdir, sz, ph.vaddr + ph.memsz)) == 0)
6349
       goto bad;
6350 if(ph.vaddr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

Load program section from disk

Load program into

memory

loaduvm(): read program from disk



```
1918 loaduvm(pde_t *pgdir, char *addr, struct inode *i , uint offset, uint sz)
1919 {
1925 for(i = 0; i < sz; i += PGSIZE)
1926
       if((pte = walkpgdir(pgdir, addr+i, 0)) == 0)
        panic("loaduvm: address should exist");
1927
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)
       panic("loaduvm: address should exist");
1927
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)
         panic("loaduvm: address should exist");
1927
1928
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)
        panic("loaduvm: address should exist");
1927
1928
1929
       if(sz - i < PGSIZE)
1930
        n = sz - i;
1931
       else
1932
       n = PGSIZE;
       if(readi(ip, P2V(pa), offget+i, n) !=
1933
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)
        panic("loaduvm: address should exist");
1927
1928
1929
       if(sz - i < PGSIZE)
1930
        n = sz - i;
1931
       else
1932
       n = PGSIZE;
       if(readi(ip, P2V(pa), offset+i, n) != n)
1933
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 {
                                                     Load program into
1925 for(i = 0; i < sz; i += PGSIZE){
                                                               memory
1926
      if((pte = walkpgdir(pgdir, addr+i, 0)) == 0)
       panic("loaduvm: address should exist");
1927
1928
      pa = PTE_ADDR(*pte);
1929
1930
       n = sz - i;
1931
      else
1932
      n = PGSIZE;
      if(readi(ip, P2V(pa), offset+i, n) != n
1933
1934
       return -1;
1935 }
1936 return 0;
                                     Why can't we use addr
1937 }
                                      directly?
```

Drawing: two page tables

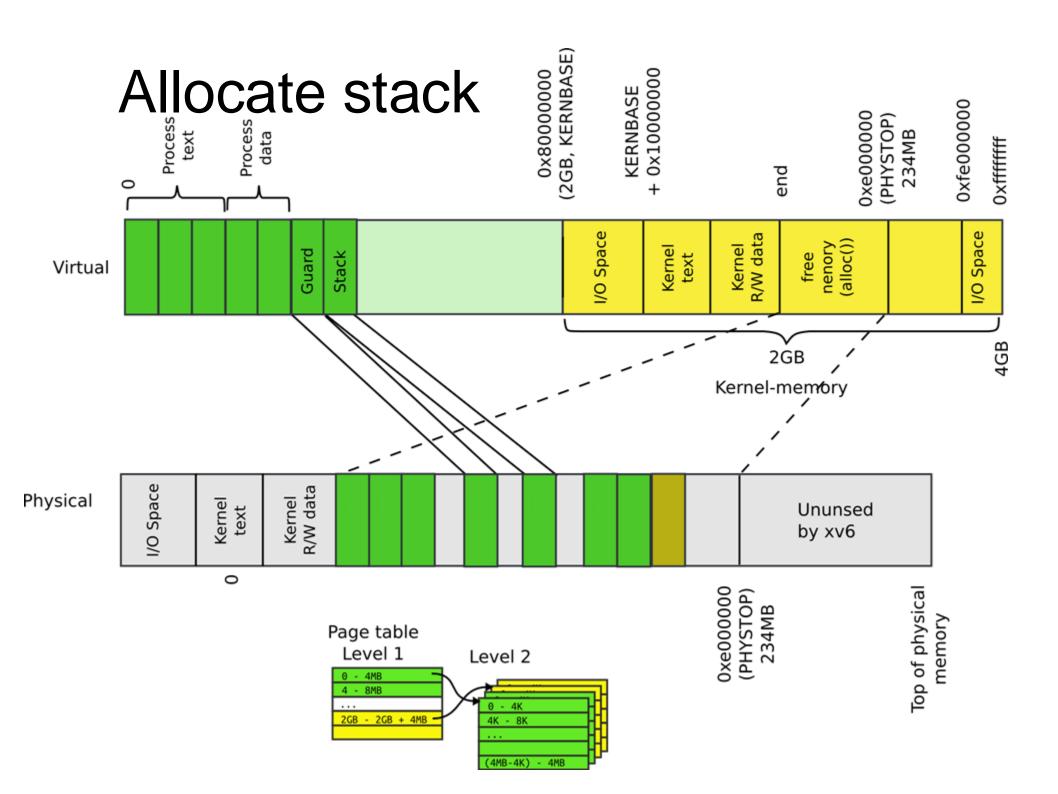
```
1918 loaduvm(pde t *pgdir, char *addr, struct inode *ip, uint offset, uint sz)
1919 {
1925 for(i = 0; i < sz; i += PGSIZE){
1926
       if((pte = walkpgdir(pgdir, addr+i, 0)) == 0)
       panic("loaduvm: address should exist");
1927
1928
       pa = PTE ADDR(*pte);
1929
       if(sz - i < PGSIZE)
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



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

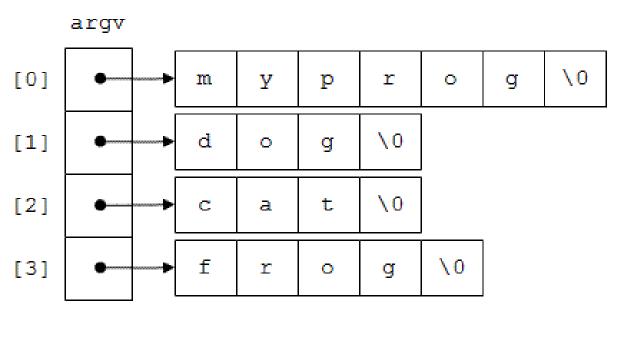
- Read process binary from disk
- Create process address space
- Load program into memory
- Allocate program stack
- Push program arguments on the stack

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 a pointer to "./program"
 - argv[1] would be a pointer to "hello"
 - argv[2] would be a pointer to "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++) {
                                                                   Copy elements of the
                                                                 array one by one on the
                                                                               stack
6371 sp = (sp - (strlen(argv[argc]) + 1)) & \sim3;
      if(cd pyout(pgdir, sp, argv[argc], strlen(argv[argc]) + 1) < 0)</pre>
6372
       goto bad;
6373
6374 ustack[3+argc] = sp;
                                                                         Remember we
6375 }
                                                                         can't use virtual
6376 ustack[3+argc] = 0;
                                                                         addresses directly
6377
6378 ustack[0] = 0xffffffff; // fake return PC
6379 ustack[1] = argc;
6380 ustack[2] = sp - (argc+1)*4; // argv pointer
6381
                                            Poll: PollEv.com/antonburtsev
6382 sp -= (3+argc+1) * 4;
6383 if(copyout(pgdir, sp, ustack, (3+argc+1)*4) < 0)
6384 goto bad;
```

```
6367 // Push argument strings, prepare rest of stack in ustack.
6368 for(argc = 0; argv[argc]; argc++) {
6371 sp = (sp - (strlen(argv[argc]) + 1)) & \sim3;
       if(copyout(pgdir, sp, argv[argc], strlen(argv[argc]) + 1) < 0)</pre>
6372
        goto bad;
6373
6374 ustack[3+argc] = sp;
6375 }
6376 ustack[3+argc] = 0;
6377
6378 ustack[0] = 0xffffffff; // fake return PC
6379 ustack[1] = argc;
6380 ustack[2] = sp - (argc+1)*4; // argv pointer
6381
6382 sp -= (3+argc+1) * 4;
6383 if(copyout(pgdir, sp, ustack, (3+argc+1)*4) < 0)
6384 goto bad;
```

Copy elements of the array one by one on the stack

- Remember we can't use virtual addresses directly
- Still running on the page table of the old process

```
6367 // Push argument strings, prepare rest of stack in ustack.
6368 for(argc = 0; argv[argc]; argc++) {
      sp = (sp - (strlen(argv[argc]) + 1)) \& ~3;
6372
       if(copyout(pgdir, sp, argv[argc], strlen(argv[argc]) + 1) < 0)
6373
        goto bad;
6374 ustack[3+argc] = sp;
6375 }
6376 ustack[3+argc] = 0;
6377
6378 ustack[0] = 0xffffffff; // fake return PC
6379 ustack[1] = argc;
6380 ustack[2] = sp - (argc+1)*4; // argv pointer
6381
6382 sp -= (3+argc+1) * 4;
6383 if(copyout(pgdir, sp, ustack, (3+argc+1)*4) < 0)
6384 goto bad;
```

Push argc – number of arguments in the argv[]

```
6367 // Push argument strings, prepare rest of stack in ustack.
6368 for(argc = 0; argv[argc]; argc++) {
      sp = (sp - (strlen(argv[argc]) + 1)) \& ~3;
      if(copyout(pgdir, sp, argv[argc], strlen(argv[argc]) + 1) < 0)
6372
6373
       goto bad;
      ustack[3+argc] = sp;
6374
                                                 Push argv pointer – argv[]
6375 }
                                                       is on the stack itself
6376 ustack[3+argc] = 0;
6377
6378 ustack[0] = 0xffffffff; // fake return PC
6379 ustack[1] = argc;
6380 ustack[2] = sp = (argc+1)*4: // argy pointer
6381
6382 sp -= (3+argc+1) * 4;
6383 if(copyout(pgdir, sp, ustack, (3+argc+1)*4) < 0)
6384 goto bad;
```

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

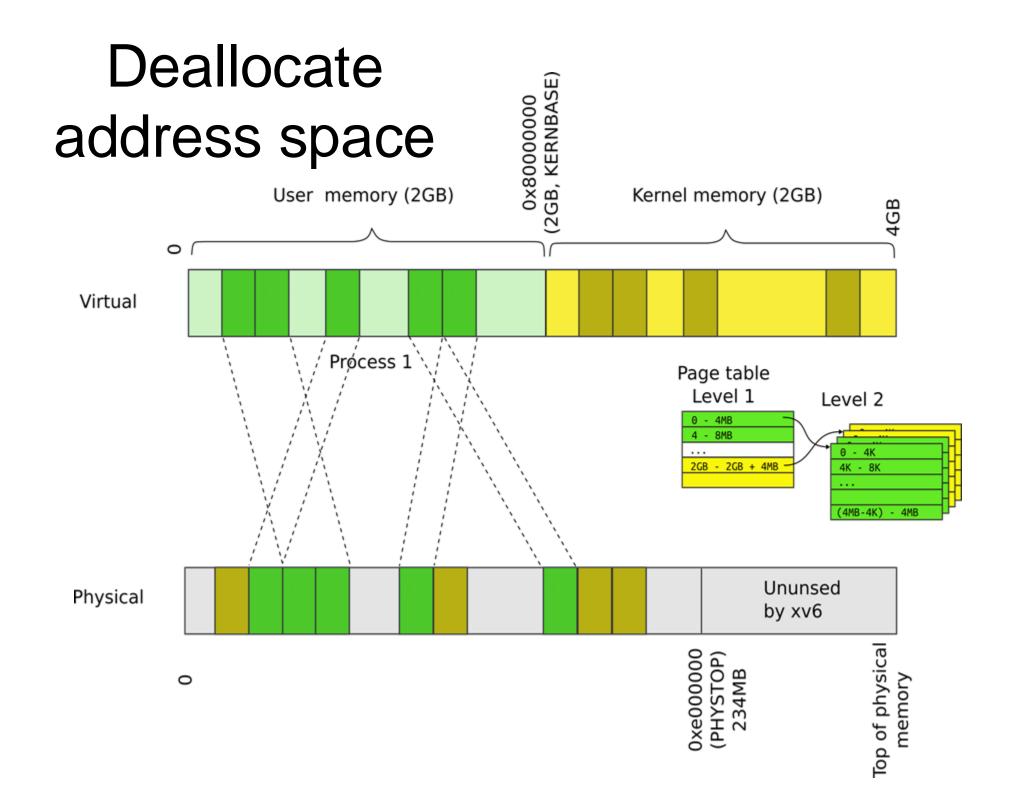
- Switch page tables
- Deallocate the old page table

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

Wait... which page table we are deallocating?

Wait... which page table we are deallocating?

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



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)
       panic("freevm: no pgdir");
2020
      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]));
        kfree(v);
2025
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){
       pte = walkpgdir(pgdir, (char*)a, 0);
1997
1998
       if(!pte)
1999
        a += (NPTENTRIES - 1) * PGSIZE;
2000
       else if((*pte & PTE P) != 0){
2001
        pa = PTE ADDR(*pte);
2002
        if(pa == 0)
         panic("kfree");
2003
        char *v = P2V(pa);
2004
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){
       pte = walkpgdir(pgdir, (char*)a, 0);
1997
1998
       if(!pte)
        a += (NPTENTRIES - 1) * PGSIZE;
1999
       else if((*pte & PTE_P) != 0){
2000
2001
        pa = PTE_ADDR(*pte);
2002
        if(pa == 0)
         panic("kfree");
2003
        char *v = P2V(pa);
2004
2005
        kfree(v);
2006
        *pte = 0;
2007
2008 }
2009 return newsz;
2010 }
```

Deallocate a page

```
2015 freevm(pde_t *pgdir)
                                   Deallocate Level 2
2016 {
2017 uint i;
2018
2019 if(pgdir == 0)
      panic("freevm: no pgdir");
2020
2021 deallocuvm(pgdir, KERNBASE, 0);
2022 for(i = 0; i < NPDENTRIES; i++){
      if(pgdir[i] & PTE_P){
2023
       char * v = P2V(PTE_ADDR(pgdir[i]));
2024
        kfree(v);
2025
2026
2027 }
2028 kfree((char*)pgdir);
2029 }
```

```
2015 freevm(pde_t *pgdir)
                                Deallocate page table
2016 {
2017 uint i;
2018
2019 if(pgdir == 0)
      panic("freevm: no pgdir");
2020
     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]));
       kfree(v);
2025
2026
2027 }
2028 kfree((char*)pgdir);
2029 }
```

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
                                                                                 main()
1323 seginit(); // segment descriptors
1330 tvinit(); // trap vectors
1338 userinit(); // first user process
1339 mpmain(); // finish this processor's setup
1340 }
```

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[],
          _binary_initcode_size[];
• • •
      p = allocproc();
2509
     initproc = p;
2510
2511 if((p->pgdir = setupkvm()) == 0)
      panic("userinit: out of memory?");
2512
      inituvm(p->pgdir, _binary_initcode_start,
        (int) binary initcode size);
2514 p->sz = PGSIZE;
2515 memset(p->tf, 0, sizeof(*p->tf));
. . .
2530 }
```

main()
userinit()

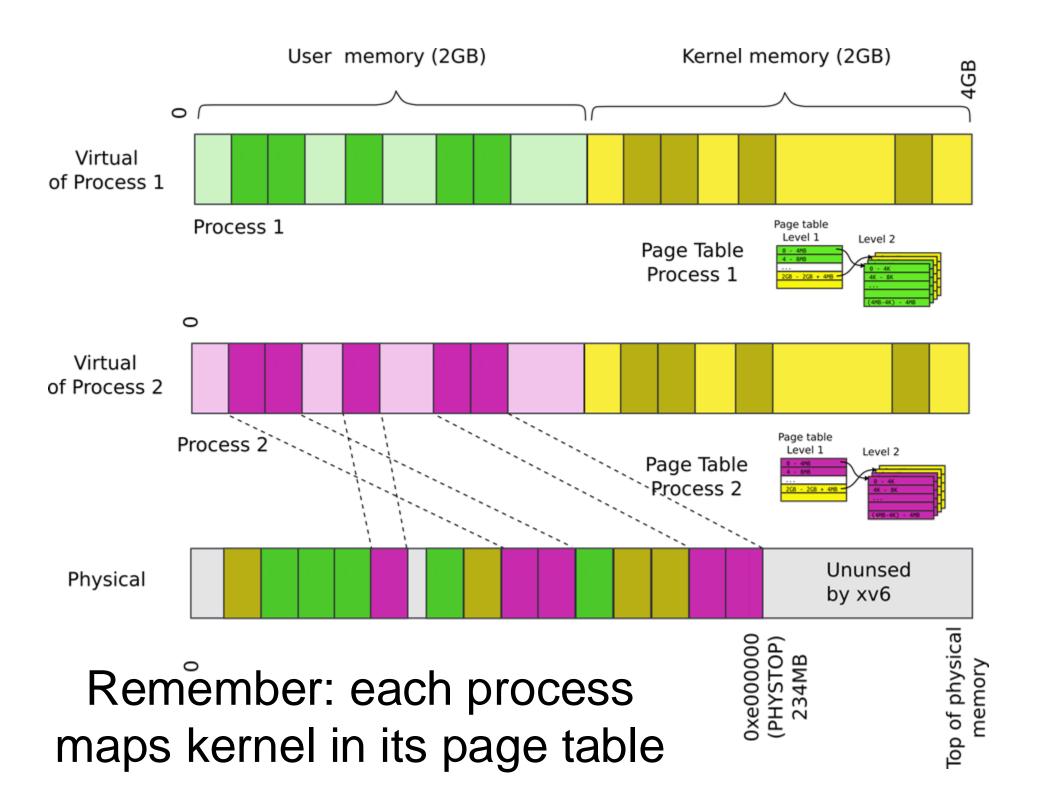
```
2103 struct proc {
2104 uint sz; // Size of process memory (bytes)
2105 pde t* pgdir; // Page table
2106 char *kstack; // Bottom of kernel stack for this process
2107 enum procstate state; // Process state
2108 volatile int pid; // Process ID
2109 struct proc *parent; // Parent process
2110 struct 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[NOFILE]; // 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[],
          _binary_initcode_size[];
• • •
      p = allocproc();
2509
2510
     initproc = p;
2511 if((p->pgdir = setupkvm()) == 0)
       panic("userinit: out of memory?");
2512
      inituvm(p->pgdir, _binary_initcode_start,
        (int) binary initcode size);
2514 p->sz = PGSIZE;
2515 memset(p->tf, 0, sizeof(*p->tf));
. . .
2530 }
```

main()
userinit()



Userinit() – create first process

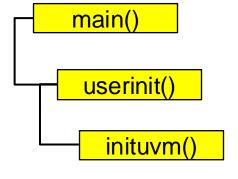
- Allocate process structure
- Information about the process
- Create a page table
- Map only kernel space
- Allocate a page for the user init code
- Map this page

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

main()

userinit()

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



- Allocate one page
- Map this page

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

What is this init?

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

2530 }

 Init was passed as the text section of the init process

```
1903 inituvm(pde_t *pgdir, char *init, uint sz)
1904 {
                                                         main()
1905 char *mem;
                                                          userinit()
1906
                                                           inituvm()
1907 if(sz \ge PGSIZE)
1908
     panic("inituvm: more than a page");
1909 mem = kalloc();
1910 memset(mem, 0, PGSIZE);
     mappages(pgdir, 0, PGSIZE, V2P(mem),
1911
        PTE W|PTE U);
1912 memmove(mem, init, sz);
1913 }
```

- Can we use virtual addresses here directly?
- We're not running on the page table of the init process
- Poll: <u>PollEv.com/antonburtsev</u>

```
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
```

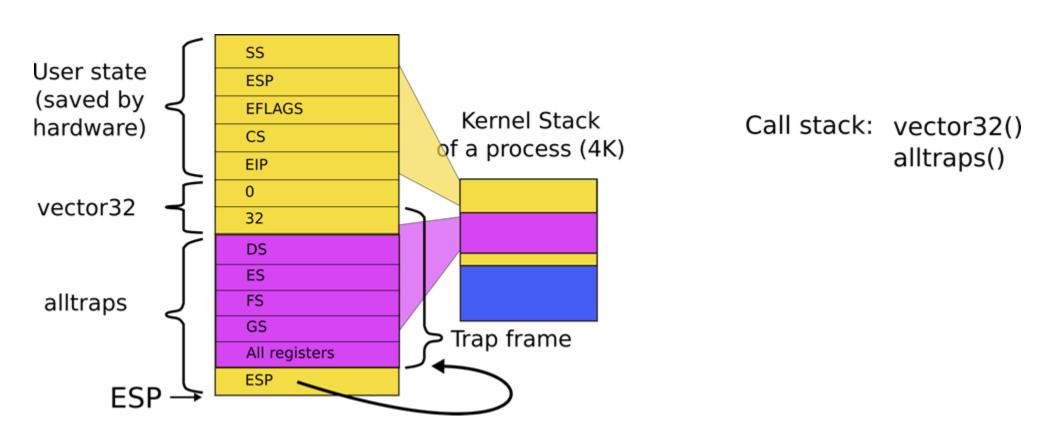
initcode.S: call exec("/init", argv);

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



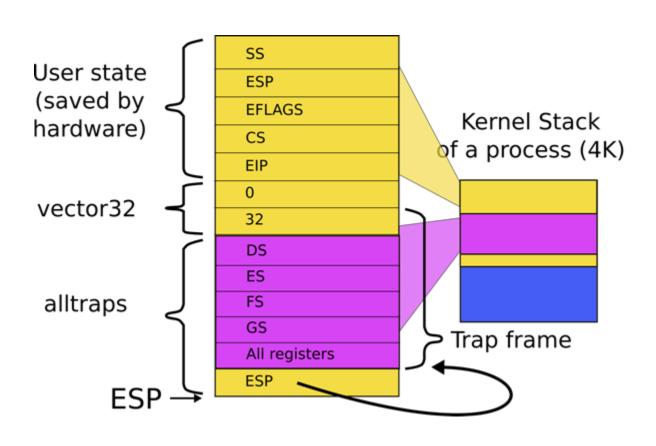
```
2103 struct proc {
2104 uint sz; // Size of process memory (bytes)
2105 pde t* pgdir; // Page table
2106 char *kstack; // Bottom of kernel stack for this process
2107 enum procstate state; // Process state
2108 volatile int pid; // Process ID
2109 struct proc *parent; // Parent process
2110 struct 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[NOFILE]; // 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 {
. . .
     inituvm(p->pgdir, _binary_initcode_start,
       (int)_binary_initcode_size);
2514 p->sz = PGSIZE;
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
. . .
2530 }
```

Kernel stack after interrupt/syscall



Call stack: vector32()

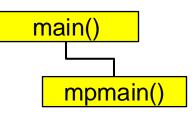
alltraps()

```
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;
```

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

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

```
1260 // Common CPU setup code.
1261 static void
1262 mpmain(void)
1263 {
1264 cprintf("cpu%d: starting\n", cpu->id);
1265 idtinit(); // load idt register
1266 xchg(&cpu->started, 1);
1267 scheduler(); // start running processes
1268 }
```





```
8510 main(void)
8511 {
8521 for(;;){
8522
       printf(1, "init: starting sh\n");
       pid = fork();
8523
8524
       if(pid < 0){
8525
         printf(1, "init: fork failed\n");
8526
         exit();
8527
       if(pid == 0){
8528
         exec("sh", argv);
8529
         printf(1, "init: exec sh failed\n");
8530
8531
         exit();
8532
8533
       while((wpid=wait()) >= 0 && wpid != pid)
         printf(1, "zombie!\n");
8534
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!