Executing a binary: `exec()`

Youjip Won

KAIST

setupkvm

```
setupkvm(void)
```

- Allocate a page (`kalloc()`) and generate mapping of the kernel address space.
- The returned page table is mapped to the kernel address space using `kmap`.
- It returns `0` if failed, the address of the page table otherwise.

```
} kmap[] = {
 { (void*)KERNBASE, 0,              EXTMEM,    PTE_W}, // I/O space
 { (void*)KERNLINK, V2P(KERNLINK), V2P(data), 0},     // kern text+rodata
 { (void*)data,     V2P(data),     PHYSTOP,   PTE_W}, // kern data+memory
 { (void*)DEVSPACE, DEVSPACE,      0,         PTE_W}, // more devices
};
```

allocuvm,loaduvm

```
allocuvm(pde_t *pgdir, uint oldsz, uint newsz)
```

- Set PTE to the `pgdir`, and make physical memory to grow from `oldsz` to `newsz`.
- Return `newsz` if succeeded, `0` otherwise.

```
loaduvm(pde_t *pgdir, char *addr,
    struct inode *ip, uint offset, uint sz)
```

- Load a program section of size `sz`.
- The program is loaded from `offset` in `ip` file to address `addr`.
- Return `0` if successful, `-1` otherwise.

walkpgdir

```
walkpgdir(pde_t *pgdir, const void *va, int alloc)
```

- Observe the virtual address `va` and find the location it should be mapped to through `pgdir`.
- Returns the virtual address of `pgdir` entry.
- If successful, returns `0`.
- When the option `alloc` is set to `1`, the mapping is created using a new page, by calling `kalloc`.

copyout

```
copyout(pde_t *pgdir, uint va, void *p, uint len)
```

- Copy `len` bytes from address `p` to address `pgdir->va`.

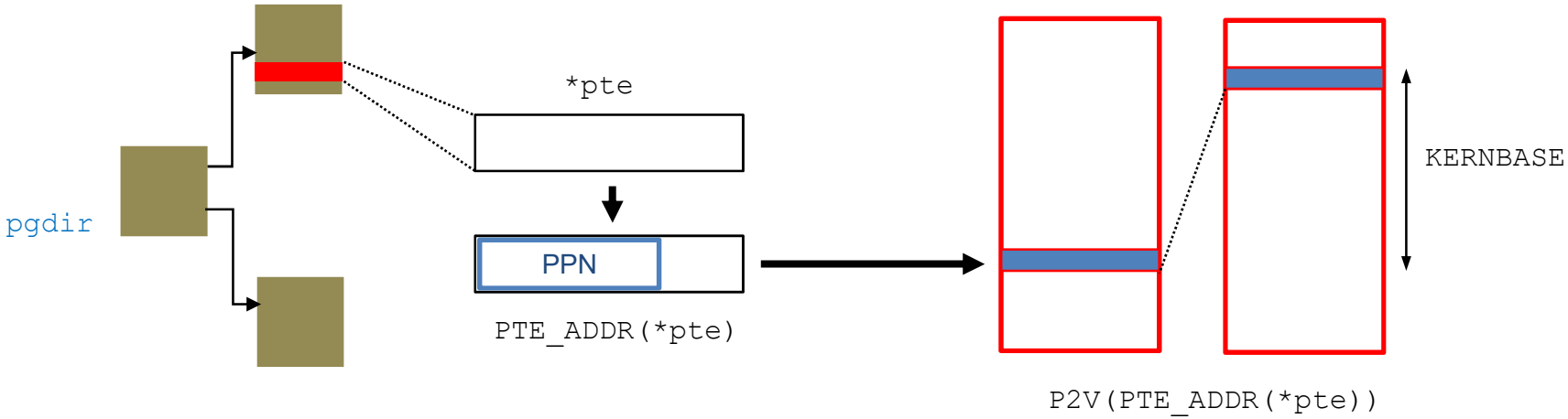
```
char *buf, *pa0;
uint n, va0;

buf = (char*)p;
while(len > 0){
  va0 = (uint)PGROUNDDOWN(va);
  pa0 = uva2ka(pgdir, (char*)va0);
  if(pa0 == 0)
    return -1;
  n = PGSIZE - (va - va0);
  if(n > len)
    n = len;
  memmove(pa0 + (va - va0), buf, n);
  len -= n;
  buf += n;
  va = va0 + PGSIZE;
}
```

copyout (Cont.)

```
uva2ka(pde_t *pgdir, char* uva)
```

- Returns the kernel address of a user address `uva`.



1. Find the page. (`walkpgdir()`)
2. Convert the physical address to kernel-space address (`P2V(PTE_ADDR(*pte))`)

namei,readi

```
namei(char *path)
```

- Returns an inode that matches the `path`.
- `namei` both supports relative and absolute `path`.

```
readi(struct inode *ip, char *dst, uint off, uint n)
```

- Reads the data from the disk of the inode `ip`, starting from the offset `off`.
- Read data is stored to the virtual address of `va`, as the size of `n`.

switchuvm

```
switchuvm(struct proc *p)
```

- Load the process `p`'s page table to the register `%cr3`.

```
  ...

 lcr3(V2P(p->pgdir));  // switch to process's address space
 popcli();
}
```

```
static inline void
lcr3(uint val)
{
  asm volatile("movl %0,%%cr3" : : "r" (val));
}
```

exec

```
exec("ls", "ls -l")
```

- Function Name : `ls`
- Argument : `"ls -l"`

```
main(int argc, char** argv)
```

`argc = 2`

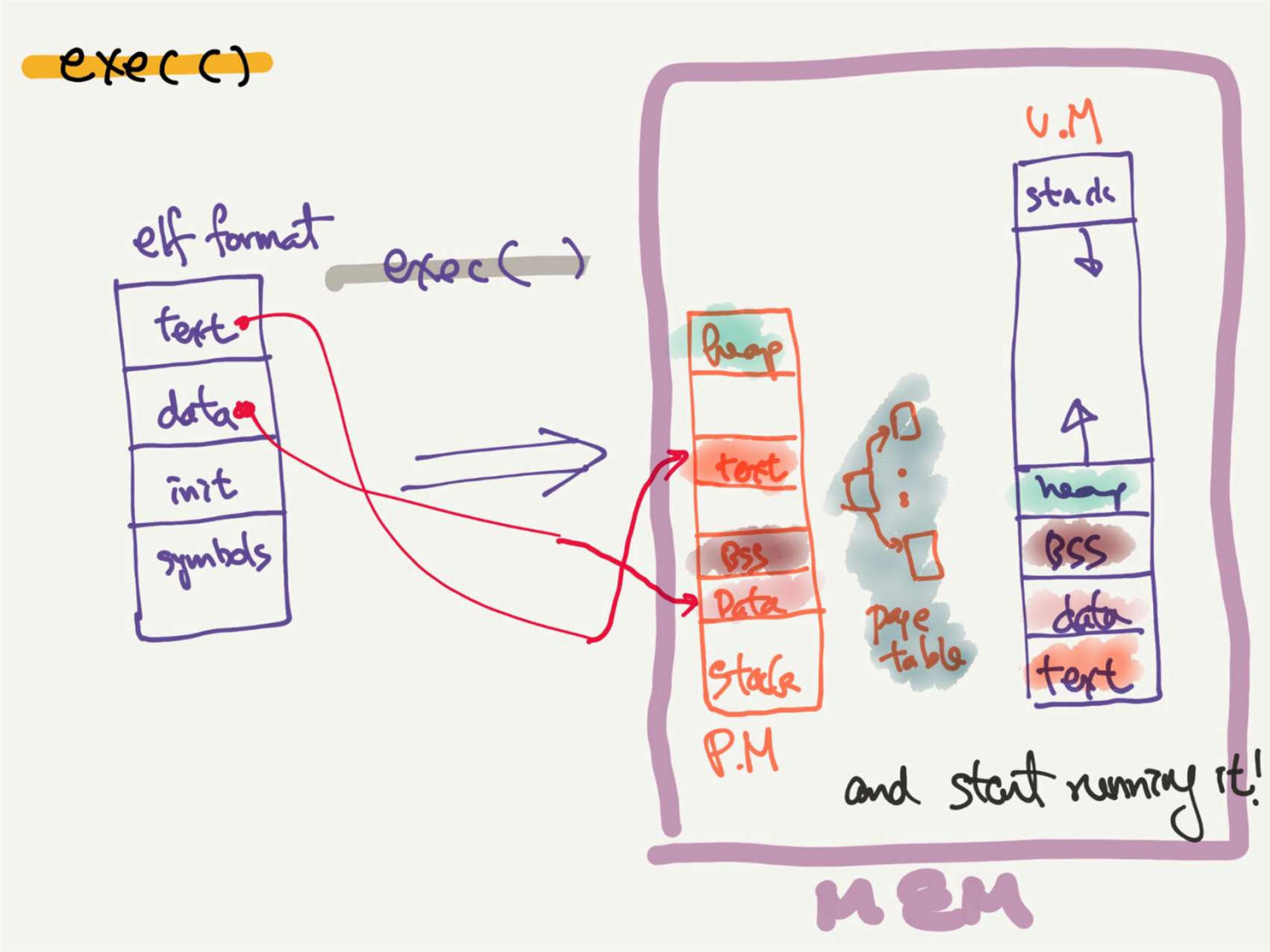
`argv[0] = "ls"`

`argv[1] = "-l"`

`exec()`: overview

1. Locate ELF file. (`namei`)
2. Setup page table for kernel region. (`setupkvm`)
3. For each section, allocate memory, set up page table for user space and load the section. (`allocuvm`, `loaduvm`)
4. Allocate stack with guard page.
5. Set up arguments at the stack.

* Be sure to check NOT to load an ELF segment to the kernel segment


exec()
elf format
exec()
text
data
init
symbols
V.M
stack
heap
text
BSS
Data
stack
P.M
page table
heap
BSS
data
text
and start running it!
MEM

exec

```
6609 int
6610 exec(char *path, char **argv)
6611 {
6612       char *s, *last;
6613       int i, off;
6614       uint argc, sz, sp, ustack[3+MAXARG+1];
6615       struct elfhdr elf;
6616       struct inode *ip;
6617       struct proghdr ph;
6618       pde_t *pgdir, *oldpgdir;
6619       struct proc *curproc = myproc();
6620
6621       begin_op();
6622
6623       if((ip = namei(path)) == 0){ //return inode for a file
6624             end_op();
6625             cprintf("exec: fail\n");
```

opens the named binary path using namei

exec (Cont.)

```
readi(inode, buffer, offset, size)
```

Read size byte starting from offset into buffer from file ip.

```
6626          return -1;
6627      }
6628      ilock(ip);
6629      pgdir = 0;
6630
6631      // Check ELF header
6632      if( readi(ip, (char*)&elf, 0, sizeof(elf)) != sizeof(elf))
6633          goto bad;
6634      if(elf.magic != ELF_MAGIC)
6635          goto bad;
6636
6637      if((pgdir = setupkvm()) == 0)
6638          goto bad;
```

Read ELF header.

Allocate a new page table and map kernel address space.

Program header

- `$ readelf -l [elf_file_name]`
- Two segments

```
// Program section header
struct proghdr {
  uint type;
  uint off;
  uint vaddr;
  uint paddr;
  uint filesz;
  uint memsz;
  uint flags;
  uint align;
};
```

```
jata@jata-VirtualBox:~/xv6-public$ readelf -l _sh

Elf file type is EXEC (Executable file)
Entry point 0x0
There are 2 program headers, starting at offset 52

Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  LOAD           0x000080 0x00000000 0x00000000 0x01872 0x018f0 RWE 0x20
  GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RWE 0x10

 Section to Segment mapping:
  Segment Sections...
   00     .text .rodata .eh_frame .data .bss
   01
jata@jata-VirtualBox:~/xv6-public$
```

exec (Cont.)

```
6641      sz = 0;
6642      for(i=0, off=elf.phoff; i<elf.phnum; i++, off+=sizeof(ph)){
6643          if(readi(ip, (char*)&ph, off, sizeof(ph)) != sizeof(ph))
6644              goto bad;
6645          if(ph.type != ELF_PROG_LOAD)
6646              continue;
6647          if(ph.memsz < ph.filesz)
6648              goto bad;
6649          if(ph.vaddr + ph.memsz < ph.vaddr)
6650              goto bad;
6651          if((sz = allocuvm(pgdir, sz, ph.vaddr + ph.memsz)) == 0)
6652              goto bad;
6653          if(ph.vaddr % PGSIZE != 0)
6654              goto bad;
```

Check program section header

Allocate memory for each ELF segment

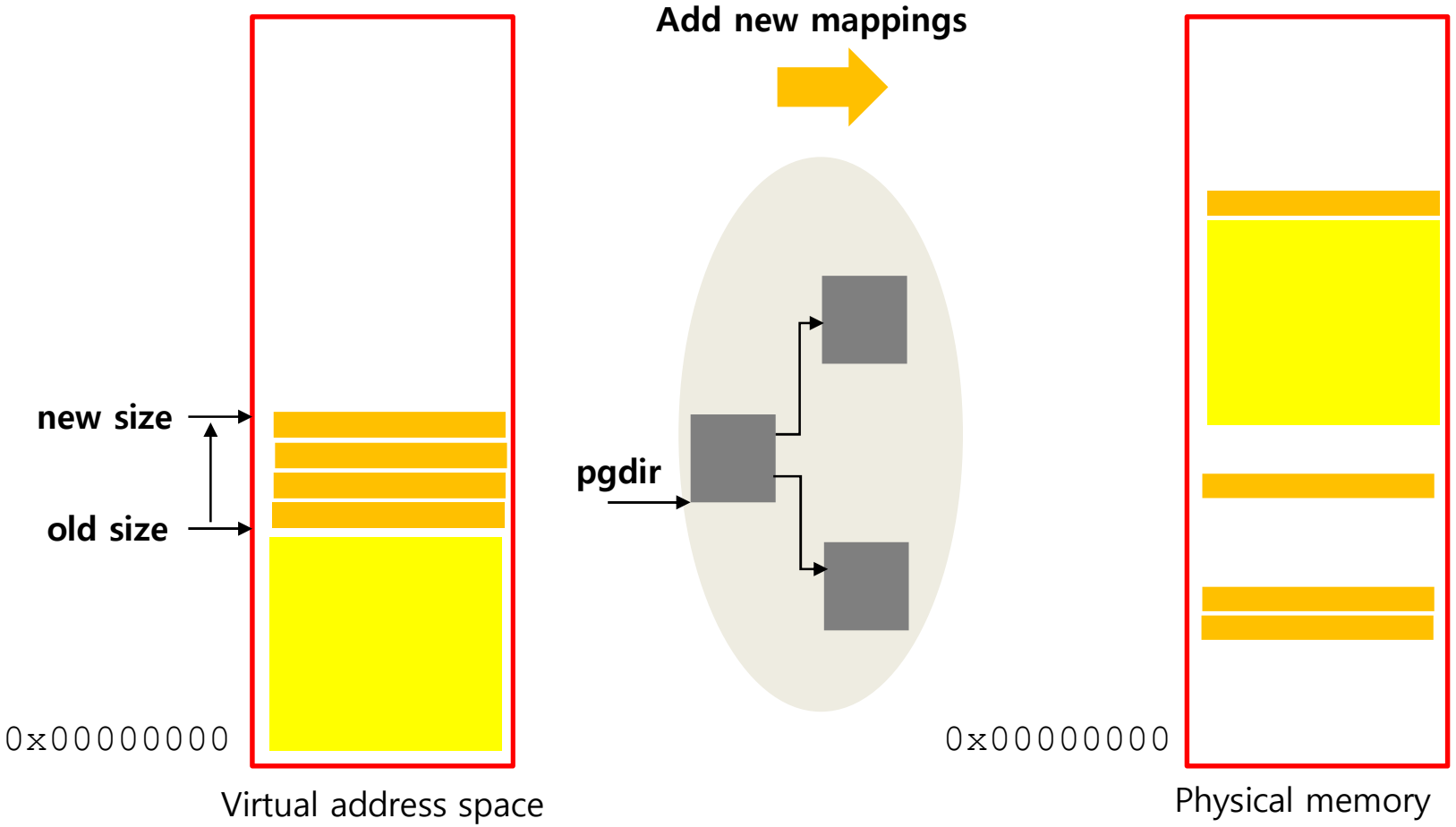
exec (Cont.)

```
6655            if(loaduvm(pgdir, (char*)ph.vaddr,
                        ip, ph.off, ph.filesz) < 0)
6656                 goto bad;
6657        }
6658        iunlockput(ip);
6659        end_op();
6660        ip = 0;
```

Load each segment into memory

allocuvm

```
1926 int
1927 allocuvm(pde_t *pgdir, uint oldsz, uint newsz)
```



allocuvm (Cont.)

```
1926 int
1927 allocuvm(pde_t *pgdir, uint oldsz, uint newsz)
1928 {
1929      char *mem;
1930      uint a;
1931
1932      if(newsz >= KERNBASE)
1933           return 0;
1934      if(newsz < oldsz)
1935           return oldsz;
1936
1937      a = PGROUNDUP(oldsz);
1938      for(; a < newsz; a += PGSIZE){
1939           mem = kalloc();
1940           if(mem == 0){
1941                cprintf("allocuvm out of memory\n");
```

Allocating memory.

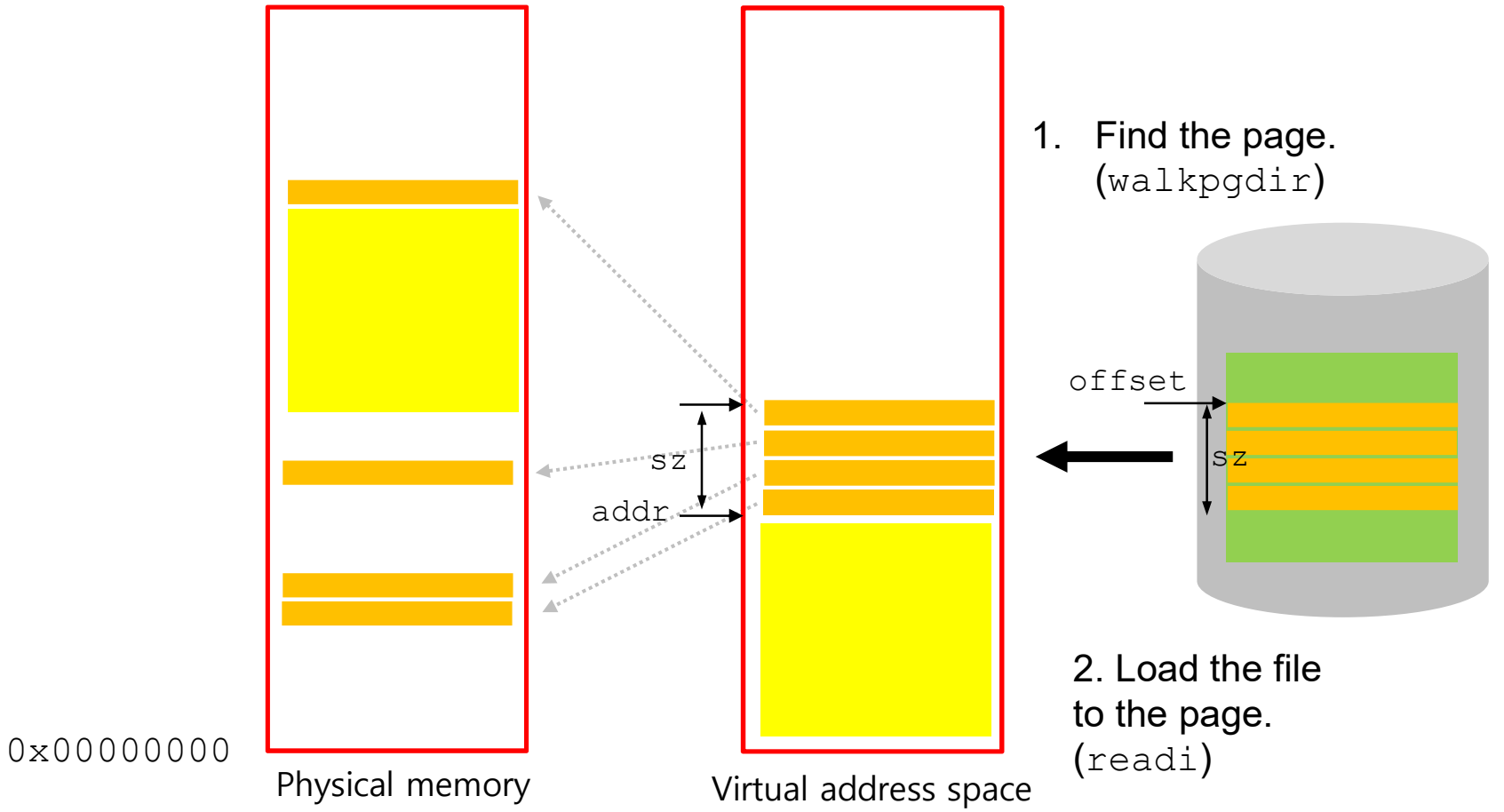
allocuvm (Cont.)

```
1942                deallocuvm(pgdir, newsz, oldsz);
1943                    return 0;
1944            }
1945        memset(mem, 0, PGSIZE);
1946        if(mappages(pgdir, (char*)a, PGSIZE, V2P(mem), PTE_W|PTE_U) < 0){
1947            cprintf("allocuvm out of memory (2)\n");
1948            deallocuvm(pgdir, newsz, oldsz);
1949            kfree(mem);
1950            return 0;
1951            }
1952        }
1953        return newsz;
1954 }
```

Mapping virtual and physical addresses

Loading a segment: `loaduvm()`

```
int
loaduvm (pde_t *pgdir, char *addr, struct inode *ip,
         uint offset, uint sz)
```



Load the section into virtual memory

```
int
loaduvm(pde_t *pgdir, char *addr, struct inode *ip, uint offset, uint sz)
{
  uint i, pa, n;
  pte_t *pte;

  if((uint) addr % PGSIZE != 0)
    panic("loaduvm: addr must be page aligned");
  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;
}
```

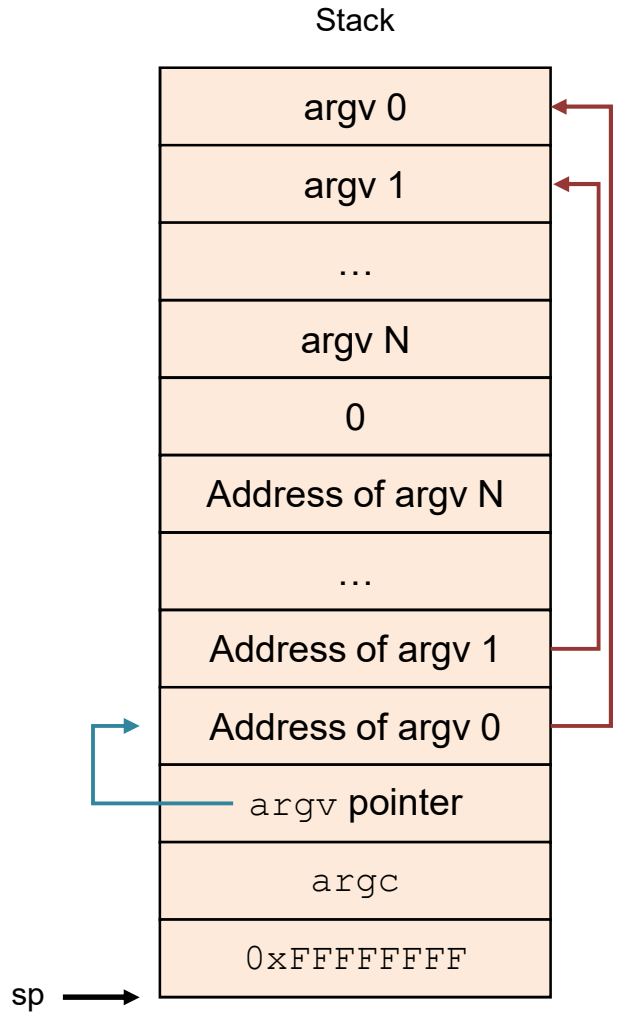
Search for the PTE

Load from the file and place it to the translated address.

Setting up the stack

- Stack
  - Parameters
  - Return address

for function calls.



exec: allocate stack.

```
6664      sz = PGROUNDUP(sz);
6665      if((sz = allocuvm(pgdir, sz, sz + 2*PGSIZE)) == 0)
6666           goto bad;
6667      clearpteu(pgdir, (char*)(sz - 2*PGSIZE));
6668      sp = sz;
6671      for(argc = 0; argv[argc]; argc++) {
6672           if(argc >= MAXARG)
6673                goto bad;
6674           sp = (sp - (strlen(argv[argc]) + 1)) & ~3;
6675           if(copyout(pgdir, sp, argv[argc],
                      strlen(argv[argc]) + 1) < 0)
6676                goto bad;
6677           ustack[3+argc] = sp;
6678      }
```

Allocate memory for stack and guard page

Disable the PTE_U flag in the page table to create guard page

Copy argument strings to the top of stack

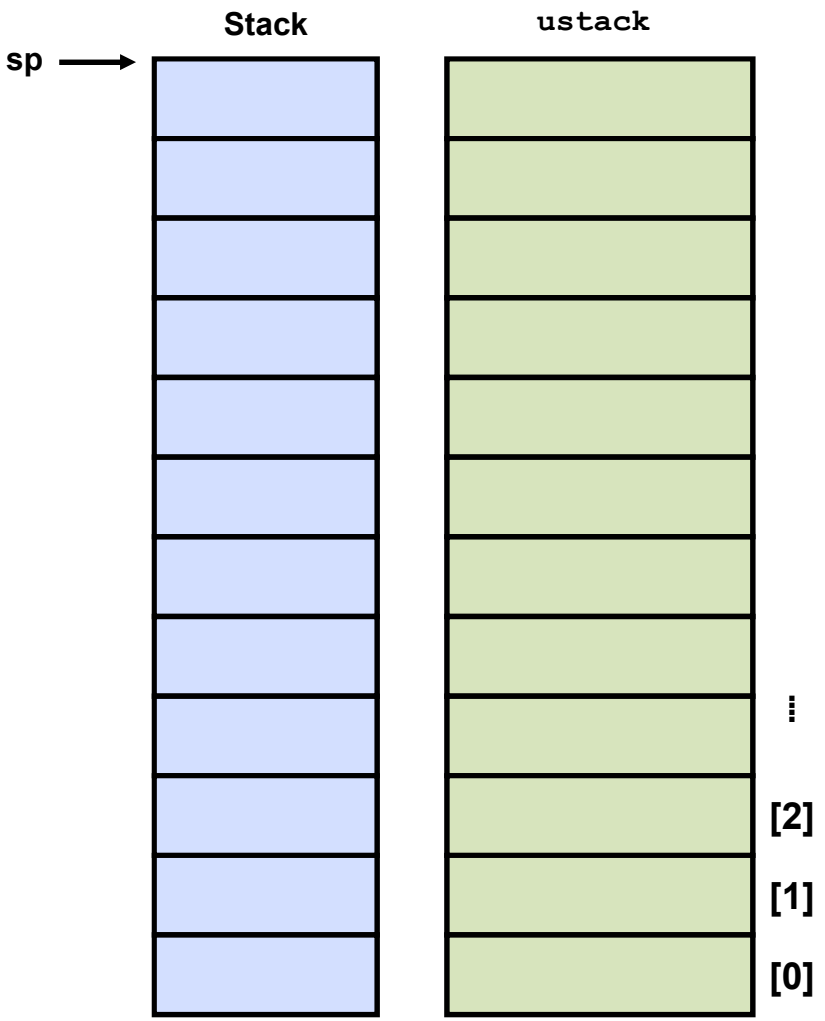
exec: allocate stack. (Cont.)

```
6679      ustack[3+argc] = 0;
6680
6681      ustack[0] = 0xffffffff; // fake return PC
6682      ustack[1] = argc;
6683      ustack[2] = sp - (argc+1)*4; // argv pointer
6684
6685      sp -= (3+argc+1) * 4;
6686      if(copyout(pgdir, sp, ustack, (3+argc+1)*4) < 0)
6687           goto bad;
```

Copy the ustack that saved fake return PC, argc, and argv pointer

exec: allocate stack. (Cont.)

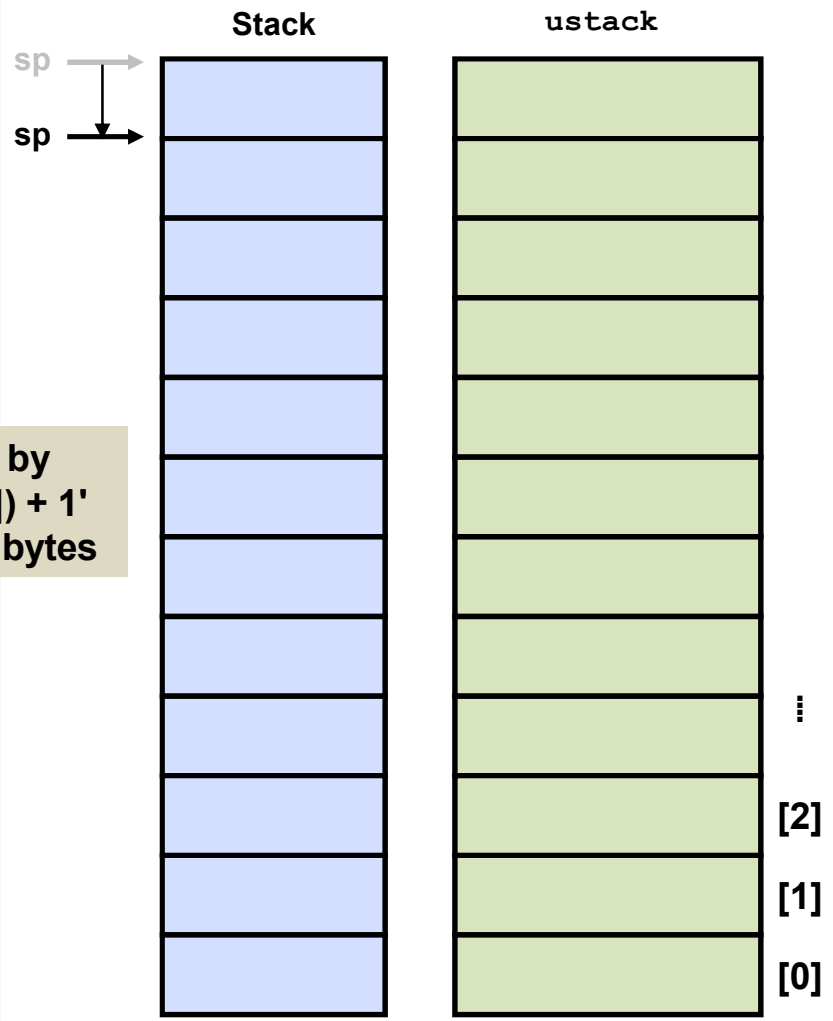
```
6671      for(argc = 0; argv[argc]; argc++) {
6672           if(argc >= MAXARG)
6673                goto bad;
6674           sp = (sp -
                (strlen(argv[argc])+1)) & ~3;
6675           if( copyout(pgdir, sp, argv[argc],
                strlen(argv[argc])+1) < 0)
6676                goto bad;
6677           ustack[3+argc] = sp;
6678      }
6679      ustack[3+argc] = 0;
6680
6681      ustack[0] = 0xffffffff;
                // fake return PC
6682      ustack[1] = argc;
6683      ustack[2] = sp - (argc+1)*4;
                // argv pointer
6684
6685      sp -= (3+argc+1) * 4;
6686      if(copyout(pgdir, sp, ustack,
                3+argc+1)*4) < 0)
6687           goto bad;
```



exec: allocate stack. (Cont.)

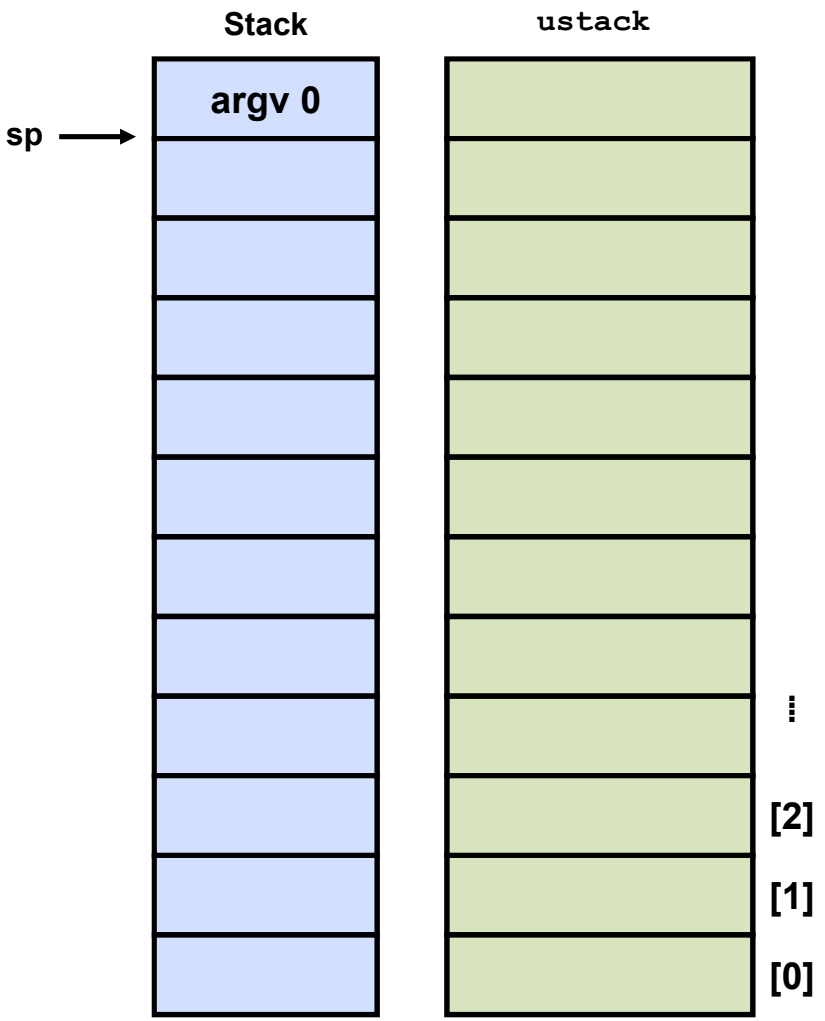
```
6671      for(argc = 0; argv[argc]; argc++) {
6672           if(argc >= MAXARG)
6673                goto bad;
6674           sp = (sp -
                   (strlen(argv[argc])+1)) & ~3;
6675           if( copyout(pgdir, sp, argv[argc],
                   strlen(argv[argc])+1) < 0)
6676                goto bad;
6677           ustack[3+argc] = sp;
6678      }
6679      ustack[3+argc] = 0;
6680
6681      ustack[0] = 0xffffffff;
                  // fake return PC
6682      ustack[1] = argc;
6683      ustack[2] = sp - (argc+1)*4;
                  // argv pointer
6684
6685      sp -= (3+argc+1) * 4;
6686      if(copyout(pgdir, sp, ustack,
                  3+argc+1)*4) < 0)
6687           goto bad;
```

Move down by 'strlen(argv[0]) + 1' aligned with 4 bytes



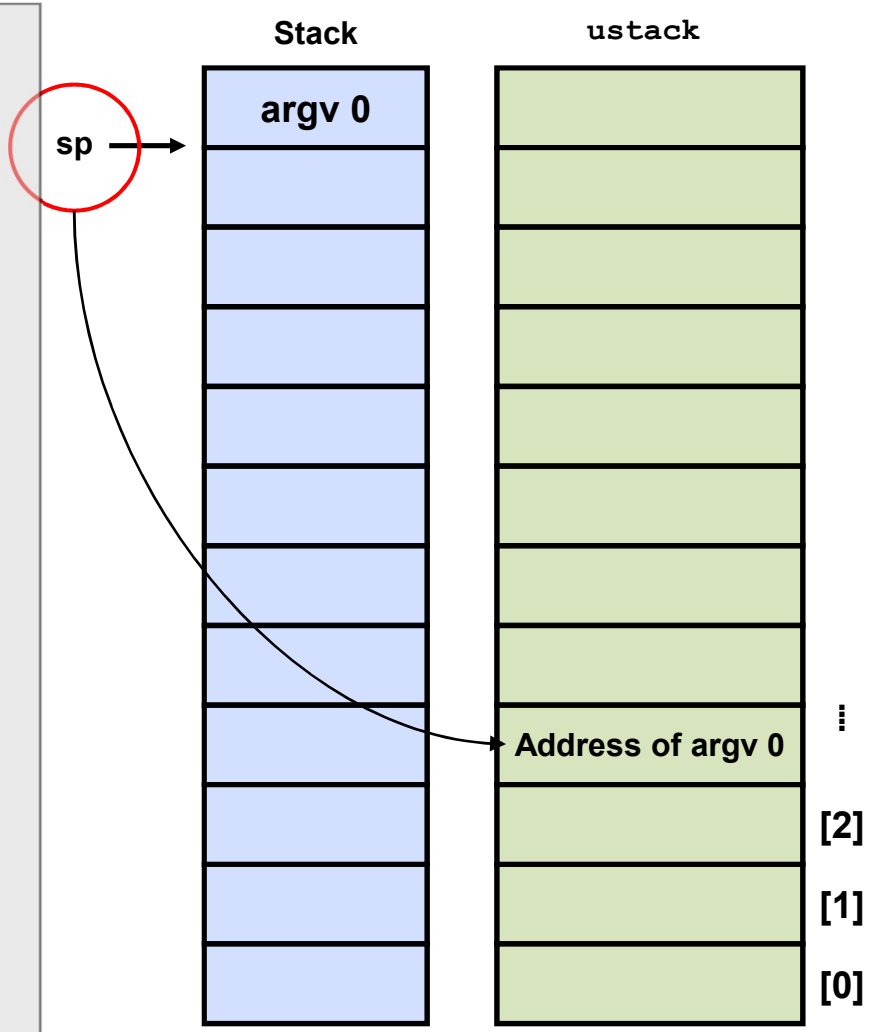
exec: allocate stack. (Cont.)

```
6671      for(argc = 0; argv[argc]; argc++) {
6672           if(argc >= MAXARG)
6673                goto bad;
6674           sp = (sp -
                   (strlen(argv[argc])+1)) & ~3;
6675           if( copyout(pgdir, sp, argv[argc],
                   strlen(argv[argc])+1) < 0)
6676                goto bad;
6677           ustack[3+argc] = sp;
6678      }
6679      ustack[3+argc] = 0;
6680
6681      ustack[0] = 0xffffffff;
                   // fake return PC
6682      ustack[1] = argc;
6683      ustack[2] = sp - (argc+1)*4;
                   // argv pointer
6684
6685      sp -= (3+argc+1) * 4;
6686      if(copyout(pgdir, sp, ustack,
                   3+argc+1)*4) < 0)
6687           goto bad;
```



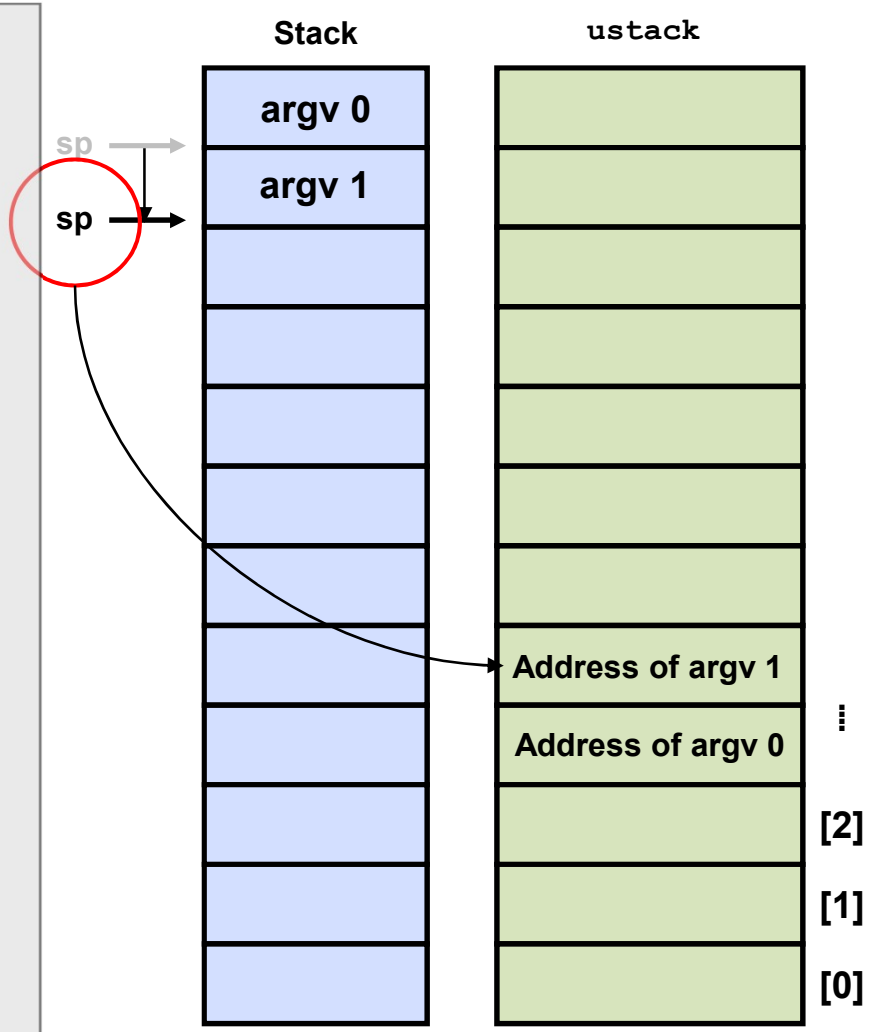
exec: allocate stack. (Cont.)

```
6671      for(argc = 0; argv[argc]; argc++) {
6672           if(argc >= MAXARG)
6673                goto bad;
6674           sp = (sp -
                   (strlen(argv[argc])+1)) & ~3;
6675           if( copyout(pgdir, sp, argv[argc],
                   strlen(argv[argc])+1) < 0)
6676                goto bad;
6677           ustack[3+argc] = sp;
6678      }
6679      ustack[3+argc] = 0;
6680
6681      ustack[0] = 0xffffffff;
                   // fake return PC
6682      ustack[1] = argc;
6683      ustack[2] = sp - (argc+1)*4;
                   // argv pointer
6684
6685      sp -= (3+argc+1) * 4;
6686      if(copyout(pgdir, sp, ustack,
                   3+argc+1)*4) < 0)
6687           goto bad;
```



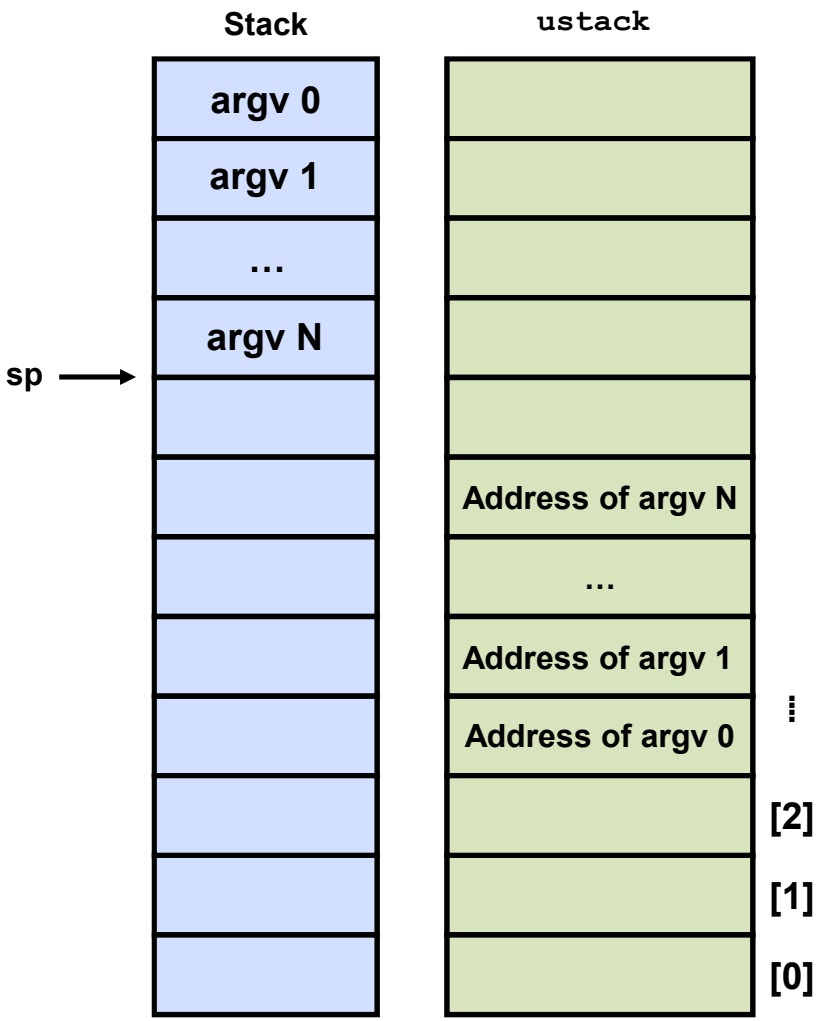
exec: allocate stack. (Cont.)

```
6671    for(argc = 0; argv[argc]; argc++) {
6672        if(argc >= MAXARG)
6673            goto bad;
6674        sp = (sp -
                (strlen(argv[argc])+1)) & ~3;
6675        if( copyout(pgdir, sp, argv[argc],
                strlen(argv[argc])+1) < 0)
6676            goto bad;
6677        ustack[3+argc] = sp;
6678    }
6679    ustack[3+argc] = 0;
6680
6681    ustack[0] = 0xffffffff;
              // fake return PC
6682    ustack[1] = argc;
6683    ustack[2] = sp - (argc+1)*4;
              // argv pointer
6684
6685    sp -= (3+argc+1) * 4;
6686    if(copyout(pgdir, sp, ustack,
              3+argc+1)*4) < 0)
6687        goto bad;
```



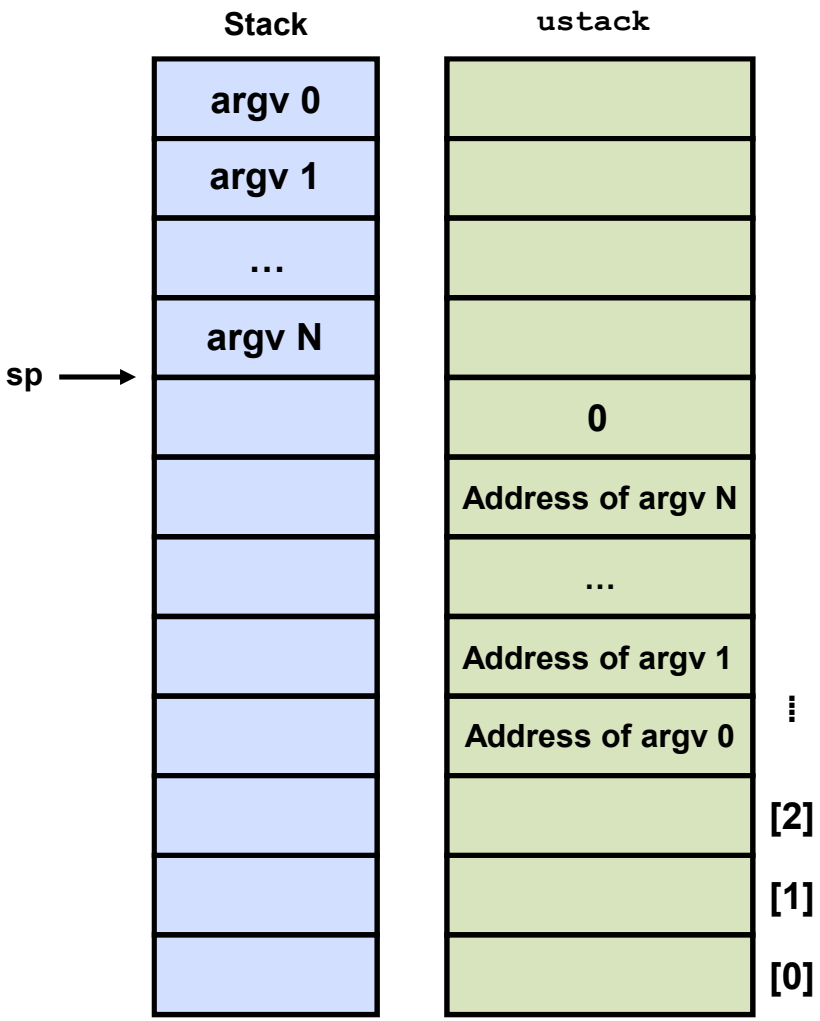
exec: allocate stack. (Cont.)

```
6671    for(argc = 0; argv[argc]; argc++) {
6672        if(argc >= MAXARG)
6673            goto bad;
6674        sp = (sp -
                (strlen(argv[argc])+1)) & ~3;
6675        if( copyout(pgdir, sp, argv[argc],
                strlen(argv[argc])+1) < 0)
6676            goto bad;
6677        ustack[3+argc] = sp;
6678    }
6679    ustack[3+argc] = 0;
6680
6681    ustack[0] = 0xffffffff;
                // fake return PC
6682    ustack[1] = argc;
6683    ustack[2] = sp - (argc+1)*4;
                // argv pointer
6684
6685    sp -= (3+argc+1) * 4;
6686    if(copyout(pgdir, sp, ustack,
                3+argc+1)*4) < 0)
6687        goto bad;
```



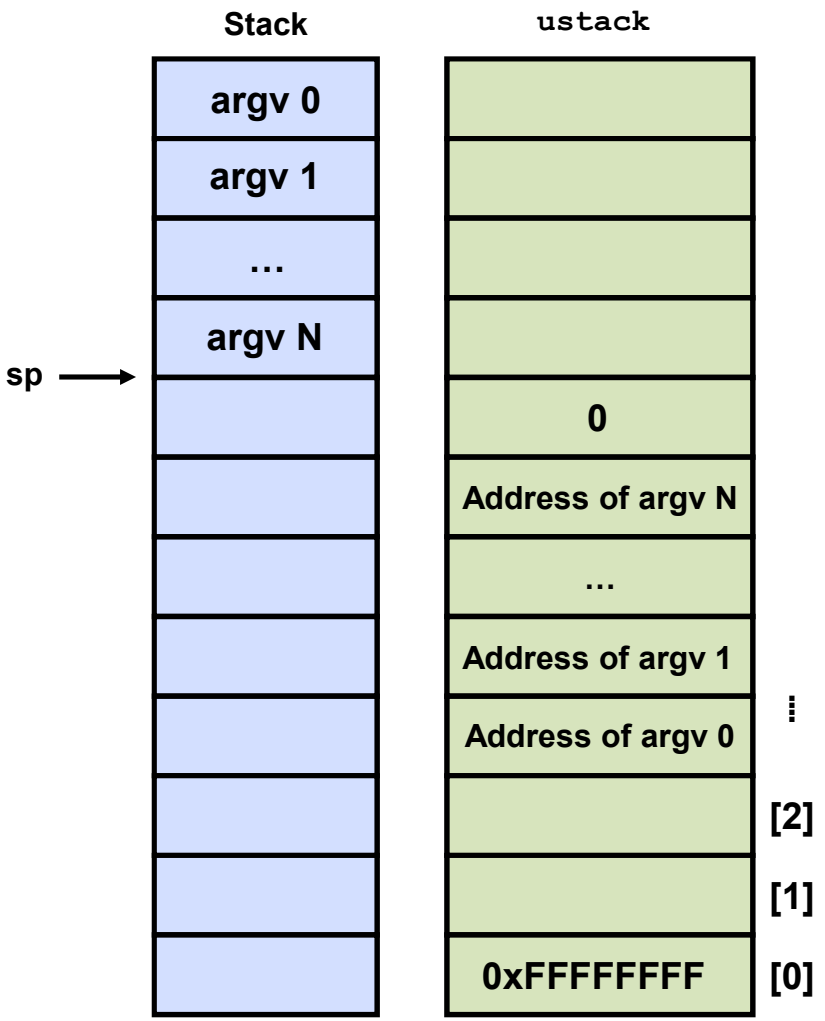
exec: allocate stack. (Cont.)

```
6671      for(argc = 0; argv[argc]; argc++) {
6672           if(argc >= MAXARG)
6673                goto bad;
6674           sp = (sp -
                   (strlen(argv[argc])+1)) & ~3;
6675           if( copyout(pgdir, sp, argv[argc],
                   strlen(argv[argc])+1) < 0)
6676                goto bad;
6677           ustack[3+argc] = sp;
6678      }
6679      ustack[3+argc] = 0;
6680
6681      ustack[0] = 0xffffffff;
                   // fake return PC
6682      ustack[1] = argc;
6683      ustack[2] = sp - (argc+1)*4;
                   // argv pointer
6684
6685      sp -= (3+argc+1) * 4;
6686      if(copyout(pgdir, sp, ustack,
                   3+argc+1)*4) < 0)
6687           goto bad;
```



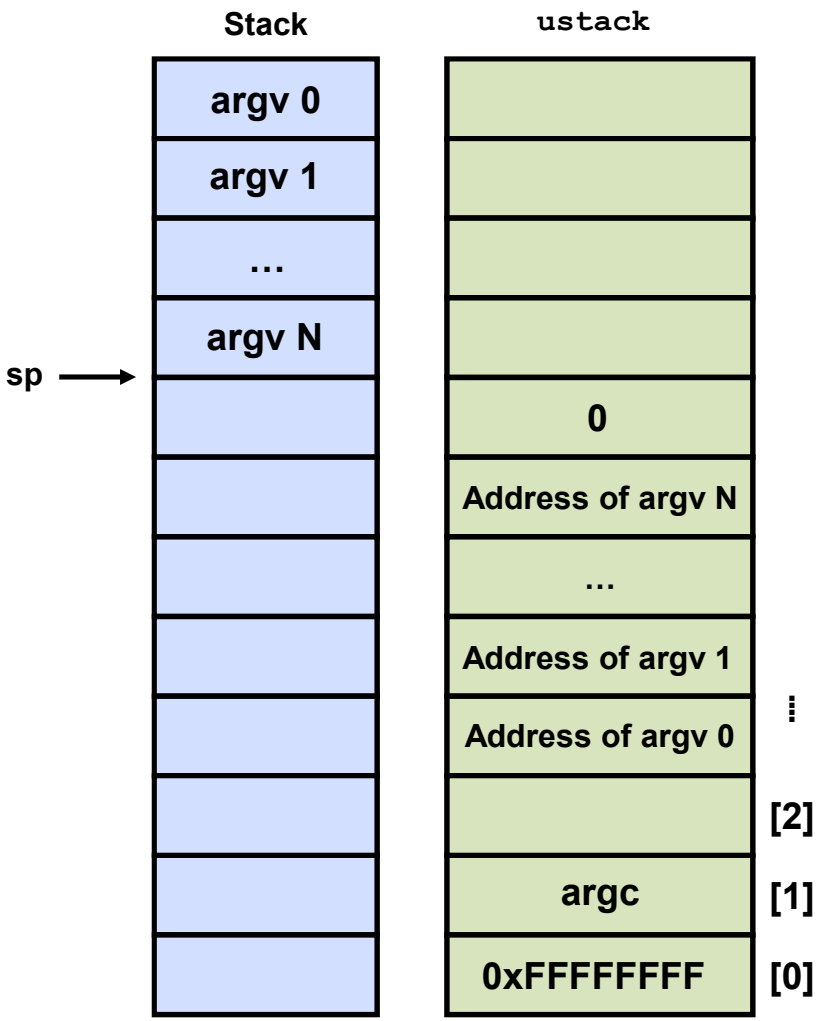
exec: allocate stack. (Cont.)

```
6671      for(argc = 0; argv[argc]; argc++) {
6672           if(argc >= MAXARG)
6673                goto bad;
6674           sp = (sp -
                (strlen(argv[argc])+1)) & ~3;
6675           if( copyout(pgdir, sp, argv[argc],
                strlen(argv[argc])+1) < 0)
6676                goto bad;
6677           ustack[3+argc] = sp;
6678      }
6679      ustack[3+argc] = 0;
6680
6681      ustack[0] = 0xffffffff;
                // fake return PC
6682      ustack[1] = argc;
6683      ustack[2] = sp - (argc+1)*4;
                // argv pointer
6684
6685      sp -= (3+argc+1) * 4;
6686      if(copyout(pgdir, sp, ustack,
                3+argc+1)*4) < 0)
6687           goto bad;
```



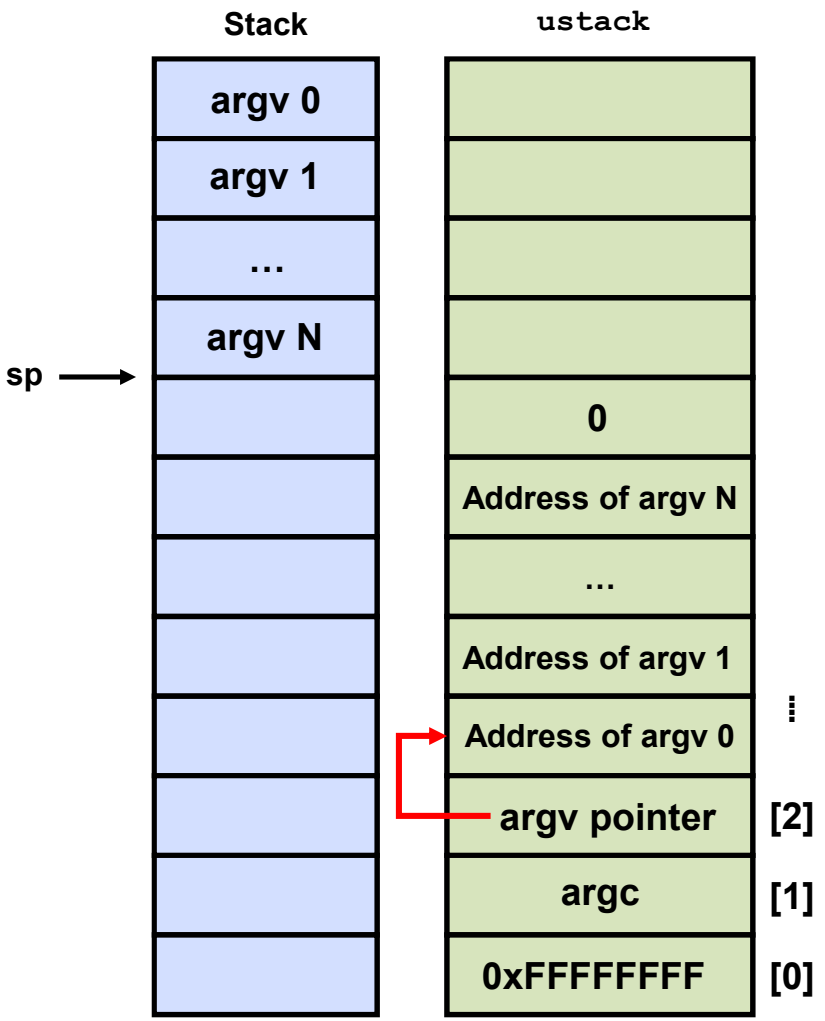
exec: allocate stack. (Cont.)

```
6671      for(argc = 0; argv[argc]; argc++) {
6672           if(argc >= MAXARG)
6673                goto bad;
6674           sp = (sp -
                (strlen(argv[argc])+1)) & ~3;
6675           if( copyout(pgdir, sp, argv[argc],
                strlen(argv[argc])+1) < 0)
6676                goto bad;
6677           ustack[3+argc] = sp;
6678      }
6679      ustack[3+argc] = 0;
6680
6681      ustack[0] = 0xffffffff;
                // fake return PC
6682      ustack[1] = argc;
6683      ustack[2] = sp - (argc+1)*4;
                // argv pointer
6684
6685      sp -= (3+argc+1) * 4;
6686      if(copyout(pgdir, sp, ustack,
                3+argc+1)*4) < 0)
6687           goto bad;
```



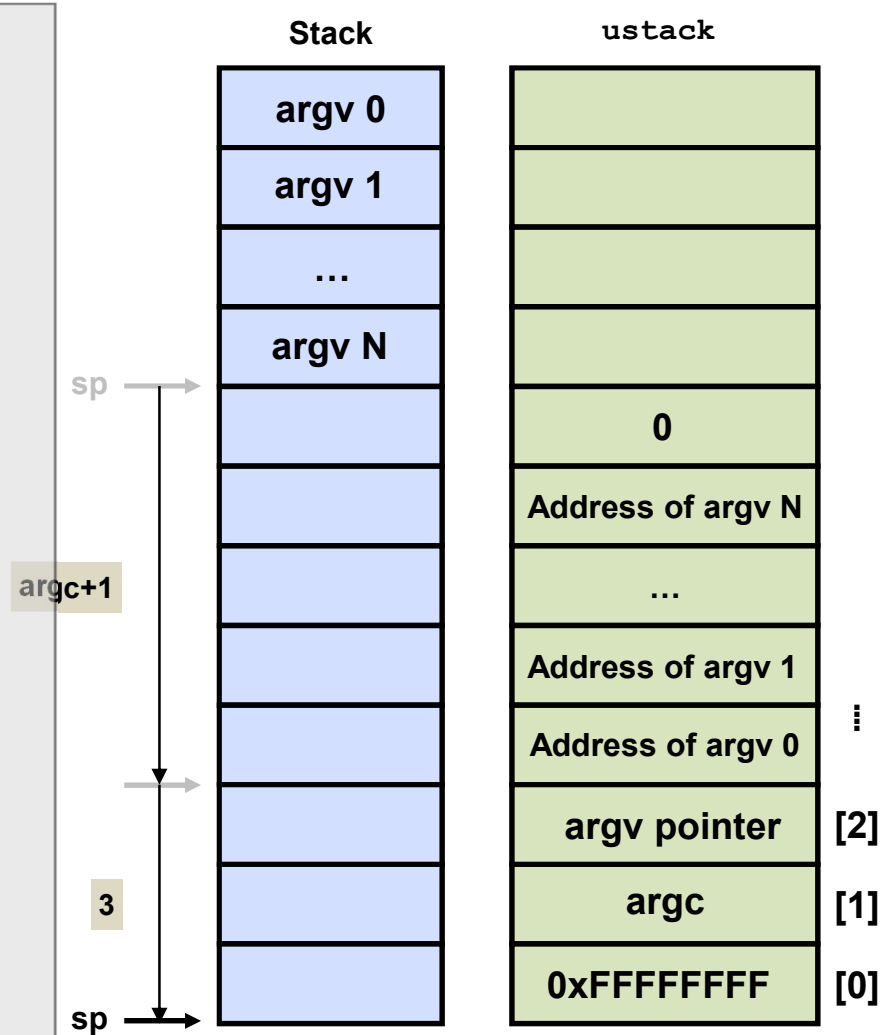
exec: allocate stack. (Cont.)

```
6671      for(argc = 0; argv[argc]; argc++) {
6672           if(argc >= MAXARG)
6673                goto bad;
6674           sp = (sp -
                (strlen(argv[argc])+1)) & ~3;
6675           if( copyout(pgdir, sp, argv[argc],
                strlen(argv[argc])+1) < 0)
6676                goto bad;
6677           ustack[3+argc] = sp;
6678      }
6679      ustack[3+argc] = 0;
6680
6681      ustack[0] = 0xffffffff;
                // fake return PC
6682      ustack[1] = argc;
6683      ustack[2] = sp - (argc+1)*4;
                // argv pointer
6684
6685      sp -= (3+argc+1) * 4;
6686      if(copyout(pgdir, sp, ustack,
                3+argc+1)*4) < 0)
6687           goto bad;
```



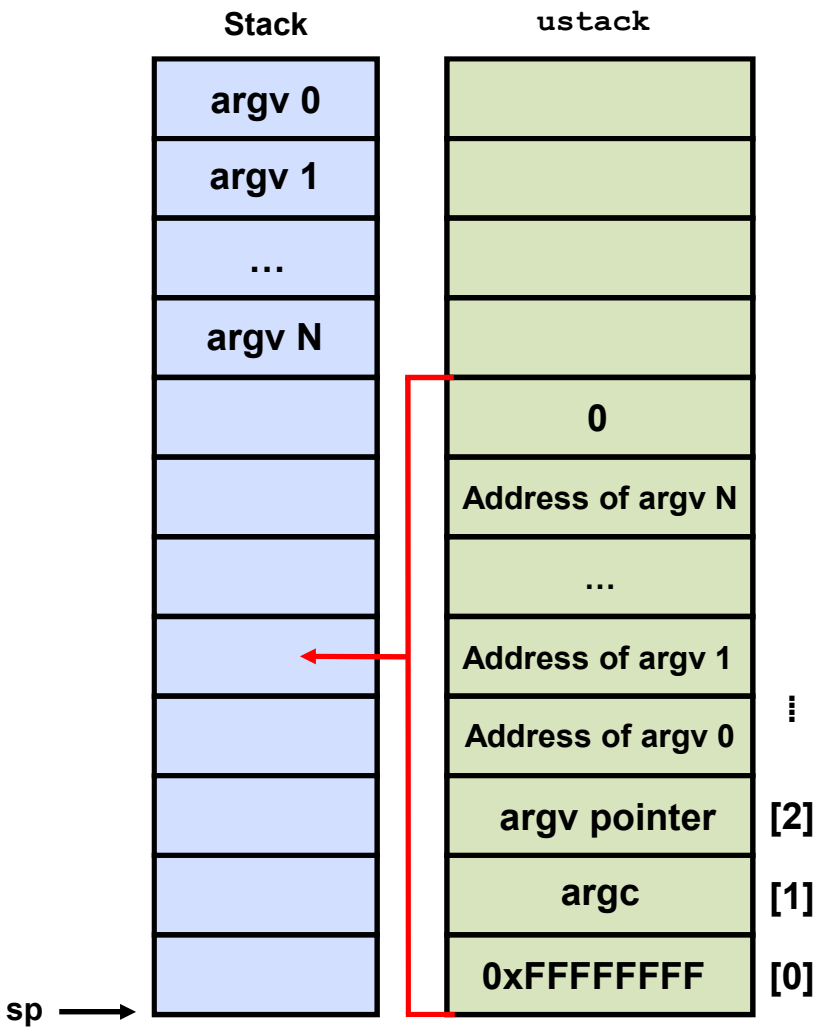
exec: allocate stack. (Cont.)

```
6671      for(argc = 0; argv[argc]; argc++) {
6672           if(argc >= MAXARG)
6673                goto bad;
6674           sp = (sp -
                   (strlen(argv[argc])+1)) & ~3;
6675           if( copyout(pgdir, sp, argv[argc],
                   strlen(argv[argc])+1) < 0)
6676                goto bad;
6677           ustack[3+argc] = sp;
6678      }
6679      ustack[3+argc] = 0;
6680
6681      ustack[0] = 0xffffffff;
                   // fake return PC
6682      ustack[1] = argc;
6683      ustack[2] = sp - (argc+1)*4;
                   // argv pointer
6684
6685      sp -= (3+argc+1) * 4;
6686      if(copyout(pgdir, sp, ustack,
                   3+argc+1)*4) < 0)
6687           goto bad;
```



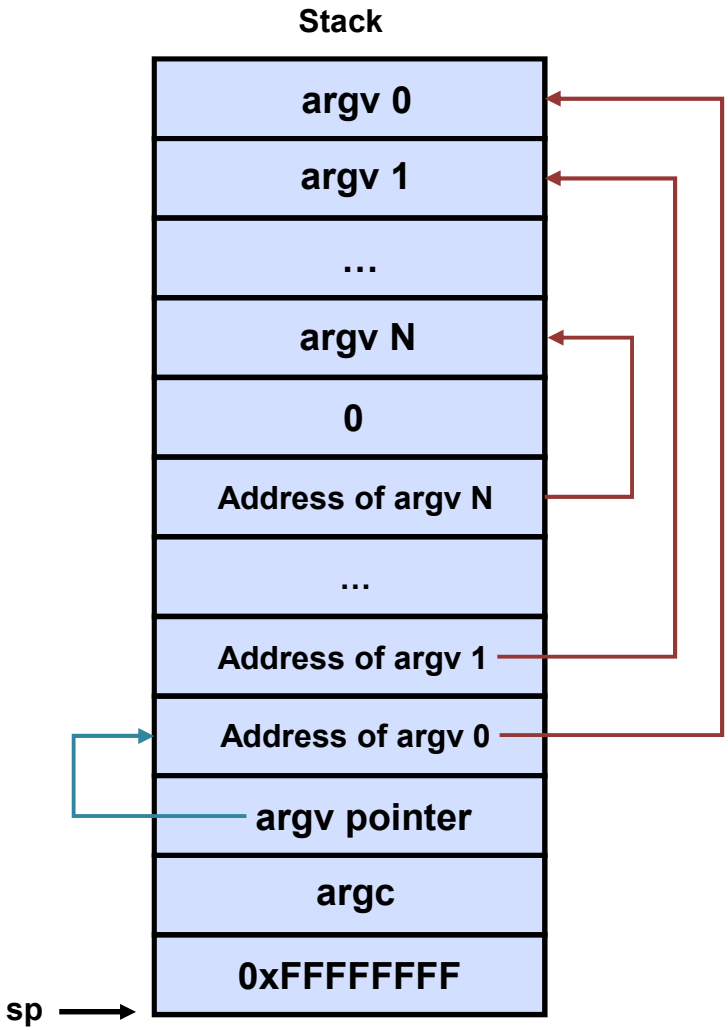
exec: allocate stack. (Cont.)

```
6671      for(argc = 0; argv[argc]; argc++) {
6672           if(argc >= MAXARG)
6673                goto bad;
6674           sp = (sp -
                   (strlen(argv[argc])+1)) & ~3;
6675           if( copyout(pgdir, sp, argv[argc],
                   strlen(argv[argc])+1) < 0)
6676                goto bad;
6677           ustack[3+argc] = sp;
6678      }
6679      ustack[3+argc] = 0;
6680
6681      ustack[0] = 0xffffffff;
                  // fake return PC
6682      ustack[1] = argc;
6683      ustack[2] = sp - (argc+1)*4;
                  // argv pointer
6684
6685      sp -= (3+argc+1) * 4;
6686      if(copyout(pgdir, sp, ustack,
                  3+argc+1)*4) < 0)
6687           goto bad;
```



exec: allocate stack. (Cont.)

```
6671      for(argc = 0; argv[argc]; argc++) {
6672           if(argc >= MAXARG)
6673                goto bad;
6674           sp = (sp -
                (strlen(argv[argc])+1)) & ~3;
6675           if( copyout(pgdir, sp, argv[argc],
                strlen(argv[argc])+1) < 0)
6676                goto bad;
6677           ustack[3+argc] = sp;
6678      }
6679      ustack[3+argc] = 0;
6680
6681      ustack[0] = 0xffffffff;
                // fake return PC
6682      ustack[1] = argc;
6683      ustack[2] = sp - (argc+1)*4;
                // argv pointer
6684
6685      sp -= (3+argc+1) * 4;
6686      if(copyout(pgdir, sp, ustack,
                3+argc+1)*4) < 0)
6687           goto bad;
```



exec

```
6689      // Save program name for debugging.
6690      for(last=s=path; *s; s++)
6691           if(*s == '/')
6692                last = s+1;
6693      safestrcpy(curproc->name, last, sizeof(curproc->name));
6695      // Commit to the user image.
6696      oldpgdir = curproc->pgdir;
6697      curproc->pgdir = pgdir;
6698      curproc->sz = sz;
6699      curproc->tf->eip = elf.entry; // main
6700      curproc->tf->esp = sp;
6701      switchuvm(curproc);
6702      freevm(oldpgdir);
6703      return 0;
6705 bad:
6706      if(pgdir)
6707           freevm(pgdir);
6708      if(ip){
```

Save old page directory

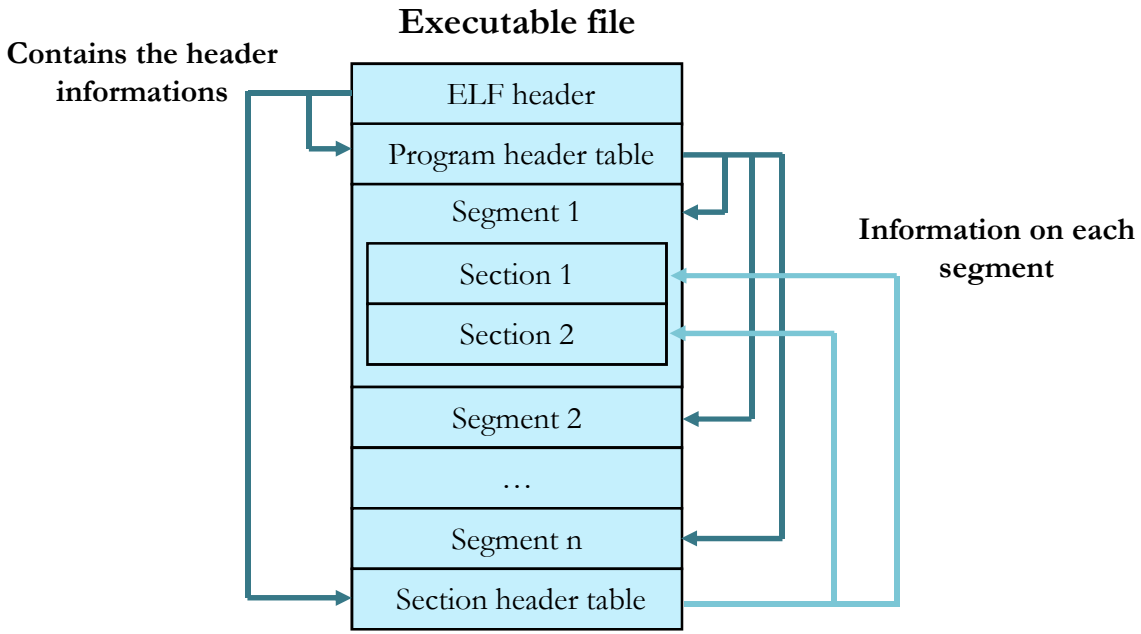
Setup new page table,
return point of user mode(main),
new user stack pointer

Install new image, free old one

Appendix A: ELF

Executable Linkable Format

- **E**xecutable **L**inkable **F**ormat
- Store the Code, Data, Relocation information, Symbol table, Debugging information as the separate sections.
- ELF header contains the information on the segments.
- A segment has multiple sections.



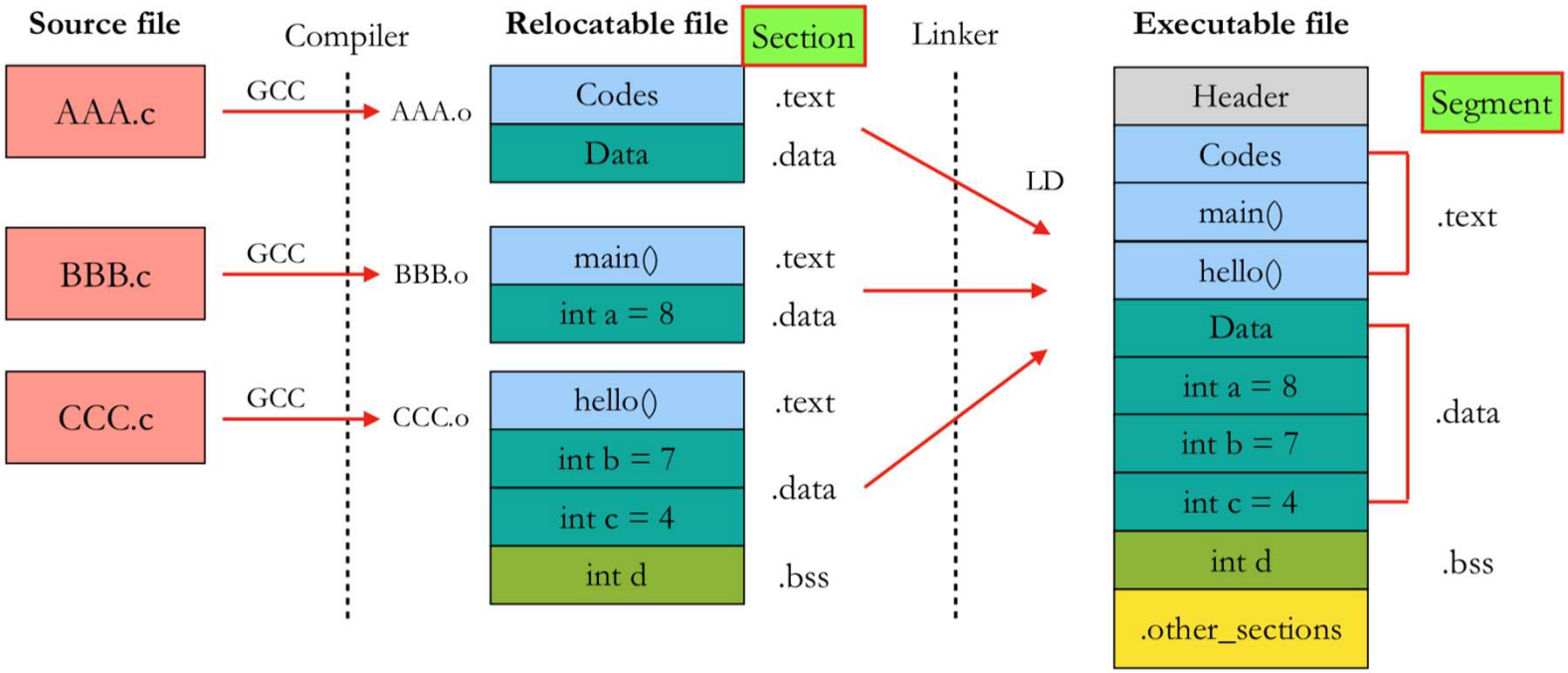
ELF Types

- Relocatable file (*.o)
  - It consists of code and data that can be linked with other object file.
- Executable file
  - ELF file that can be executed.
- Shared Object file (*.so)
  - it can be linked with other executable file or shared object file.

```
kms@linux:~/xv6-public$ file kernel
kernel: ELF 32-bit LSB  executable, Intel 80386, version 1 (SYSV), statically linked, not stripped
kms@linux:~/xv6-public$ file main.o
main.o: ELF 32-bit LSB  relocatable, Intel 80386, version 1 (SYSV), not stripped
```

Executable Linkable Format

- Section : contains the information required for linking
  - includes all information required for Object file (*.o)
- Segment : contains the information required for executing a program
  - a segment consists of sections with similar properties



More on ELF

- ELF Header consists of total 14 fields

  `readelf -h [elf_file_name]: list the ELF header content`

- Defined at `[Kernel Directory]/include/uapi/linux/elf.h`

```
kms@linux:~/xv6-public$ readelf -h kernel
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Intel 80386
  Version:                           0x1
  Entry point address:               0x10000c
  Start of program headers:          52 (bytes into file)
  Start of section headers:          164640 (bytes into file)
  Flags:                             0x0
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         2
  Size of section headers:           40 (bytes)
  Number of section headers:         18
  Section header string table index: 15
```

Program header

- `$ readelf -l [elf_file_name]`
- two segments

```
jata@jata-VirtualBox:~/xv6-public$ readelf -l _sh

Elf file type is EXEC (Executable file)
Entry point 0x0
There are 2 program headers, starting at offset 52

Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  LOAD           0x000080 0x00000000 0x00000000 0x01872 0x018f0 RWE 0x20
  GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RWE 0x10

 Section to Segment mapping:
  Segment Sections...
   00     .text .rodata .eh_frame .data .bss
   01
jata@jata-VirtualBox:~/xv6-public$
```

Section header

- `$ readelf -S [elf_file_name]`
- `off`: file offset
- `addr`: virtual memory address
- `ES`: entry size

```
jata@jata-VirtualBox:~/xv6-public$ readelf -S _sh
There are 17 section headers, starting at offset 0x580c:

Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .text             PROGBITS        00000000 000080 0011b6 00 WAX  0   0 16
  [ 2] .rodata           PROGBITS        000011b8 001238 000101 00   A  0   0  4
  [ 3] .eh_frame         PROGBITS        000012bc 00133c 0005a8 00   A  0   0  4
  [ 4] .data             PROGBITS        00001864 0018e4 00000e 00  WA  0   0  4
  [ 5] .bss              NOBITS          00001880 0018f2 000070 00  WA  0   0 32
  [ 6] .comment          PROGBITS        00000000 0018f2 00002a 01  MS  0   0  1
  [ 7] .debug_aranges    PROGBITS        00000000 001920 0000a8 00      0   0  8
  [ 8] .debug_info       PROGBITS        00000000 0019c8 001760 00      0   0  1
  [ 9] .debug_abbrev     PROGBITS        00000000 003128 00070e 00      0   0  1
  [10] .debug_line       PROGBITS        00000000 003836 000516 00      0   0  1
  [11] .debug_str        PROGBITS        00000000 003d4c 000306 01  MS  0   0  1
  [12] .debug_loc        PROGBITS        00000000 004052 000f7d 00      0   0  1
  [13] .debug_ranges     PROGBITS        00000000 004fcf 0000d8 00      0   0  1
  [14] .symtab           SYMTAB          00000000 0050a8 000500 10     15  23  4
  [15] .strtab           STRTAB          00000000 0055a8 0001c3 00      0   0  1
  [16] .shstrtab         STRTAB          00000000 00576b 0000a0 00      0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings), I (info),
  L (link order), O (extra OS processing required), G (group), T (TLS),
  C (compressed), x (unknown), o (OS specific), E (exclude),
  p (processor specific)
jata@jata-VirtualBox:~/xv6-public$
```

Structure of ELF header

```
#define Elf32_Addr        unsigned long
#define Elf32_Off         unsigned long
#define Elf32_Half        unsigned short int
#define Elf32_Word        unsigned int
#define Elf32_Sword       int
#define Elf32_Xword       unsigned long
#define Elf32_Sxword      long

typedef struct {
    unsigned char  e_ident[16];     // ELF id
    Elf32_Half     e_type;          // object file type
    Elf32_Half     e_machine;       // machine type
    Elf32_Word     e_version;       // object file version
    Elf32_Addr     e_entry;         // address of entry point
    Elf32_Off      e_phoff;         // offset of program header
    Elf32_Off      e_shoff;         // offset of section header table
    Elf32_Word     e_flags;         // processor dependency flag
    Elf32_Half     e_ehsize;        // size of elf header
    Elf32_Half     e_phentsize;     // size of program header entry
    Elf32_Half     e_phnum;         // the number of program header entries
    Elf32_Half     e_shentsize;     // the size of the section header entry
    Elf32_Half     e_shnum;         // the number of section header entry
    Elf32_Half     e_shstrndx;      // index of the section header that
contain
                                    //the section name
}Elf32_Ehdr;
```

Header size = 52 byte (32bit CPU)