CS 143A

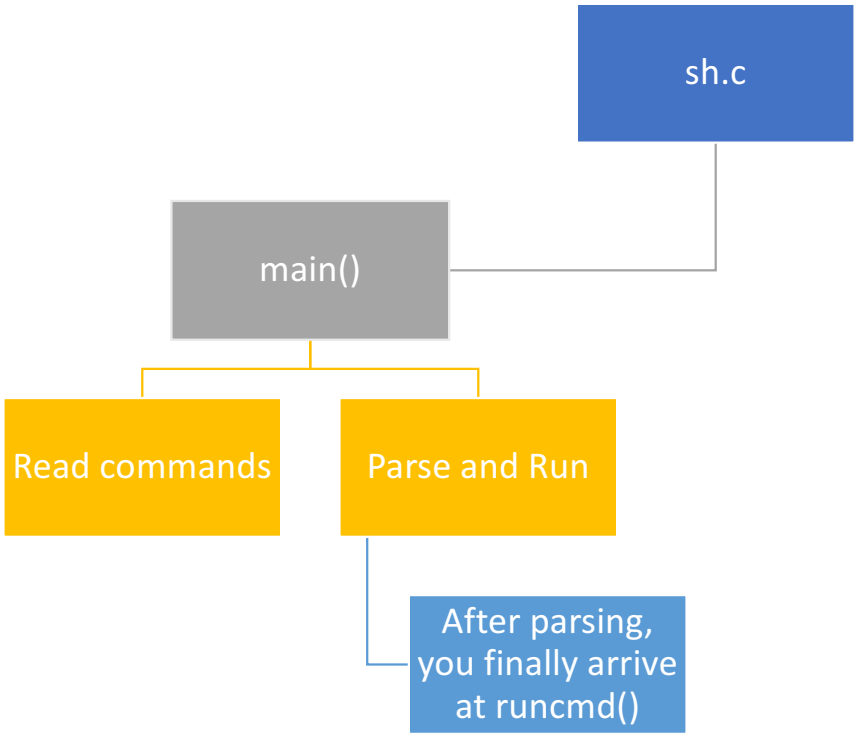
Principles of Operating Systems

Agenda

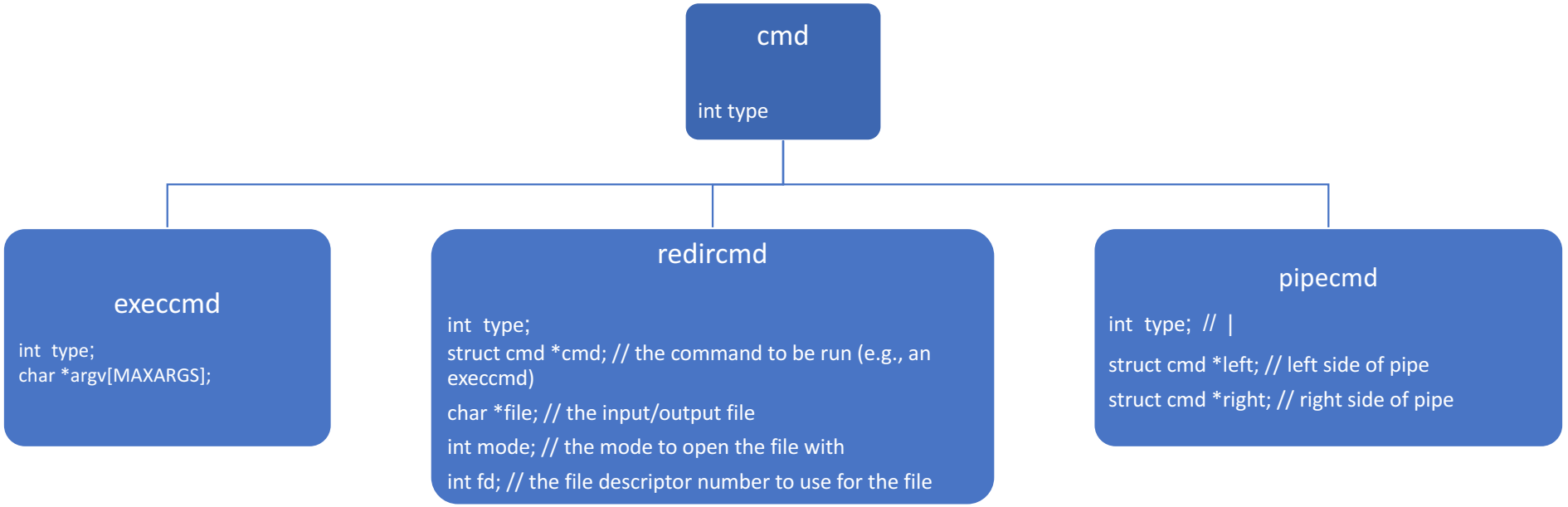
Homework 1 solution

Homework 2 solution

Homework 3 open questions (offline)



Homework 1: Program Structure



Homework 1 : Program Structure

What happens in an **execvp** call:

- Your shell searches in $PATH
- Typically, if the executable is found, a call to execve will be made.
- Linux prepares a program by allocating memory for it, opening it, scheduling it for execution, initializes memory structures, sets up its arguments and environment from the supplied arguments to the execvp call, finds a handler appropriate for loading the binary, and sets the current task (the execvp caller) as not executing.

Homework 1 Q1: **execcmd**

```
case ' ': ;
  ecmd = (struct execcmd*)cmd;
  if(ecmd->argv[0] == 0)
    exit(0);
  // Your code here ...
  if (execvp((const char*)ecmd->argv[0], ecmd->argv) == -1)
    perror("exec: ");
  break;
```

Homework 1 Q1: **execcmd**

Homework 1 Q1: execcmd Common Errors

```
char path_bin[100] = "/bin/";
char path_usr[100] = "/usr/bin/";

//try argv[0]
execv(ecmd->argv[0], ecmd->argv);
//try /bin/argv[0]
execv(strcat(path_bin, ecmd->argv[0]), ecmd->argv);
//try /usr/bin/argv[0]s
execv(strcat(path_usr, ecmd->argv[0]), ecmd->argv);

fprintf(stderr, " %s ecec error\n", ecmd->argv[0]);
```

Homework 1 Q1: execcmd Common Errors

```
case ' ':
  ecmd = (struct execcmd *)cmd;
  if (ecmd->argv[0] == 0)
    exit(0);
  // fprintf(stderr, "exec not implemented\n");
  // Your code here ...
  for (unsigned int i = 0; i < 3; ++i)
  {
    char path[256] = {0};
    strcpy(path, PATH[i]);
    strcat(path, ecmd->argv[0]);
    execv(path, ecmd->argv);
  }
  fprintf(stderr, "%s: Command not found.\n", ecmd->argv[0]);
  break;
```

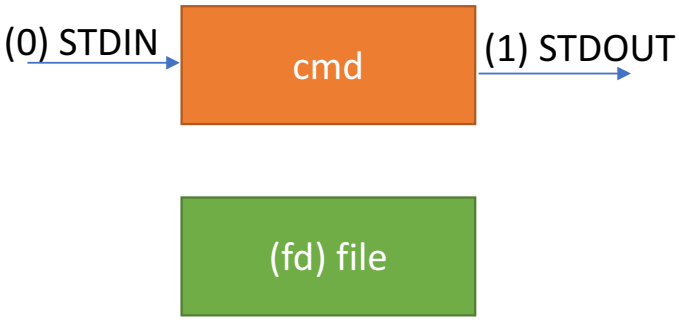
redircmd

int type;
struct cmd *cmd; // the command to be run (e.g., an execcmd)

char *file; // the input/output file

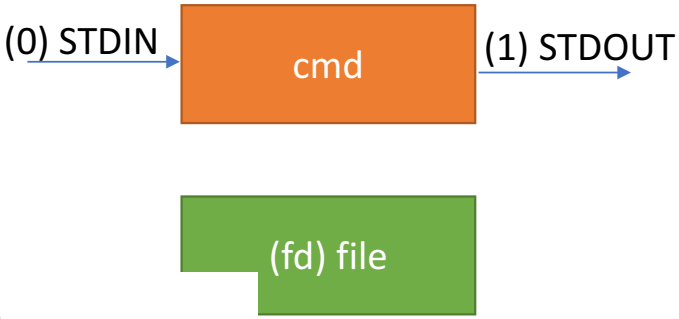
int mode; // the mode to open the file with

int fd; // the file descriptor number to use for the file



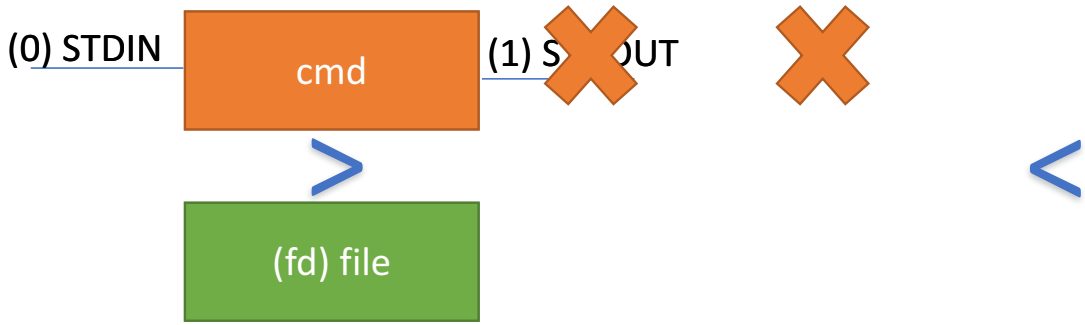
Default

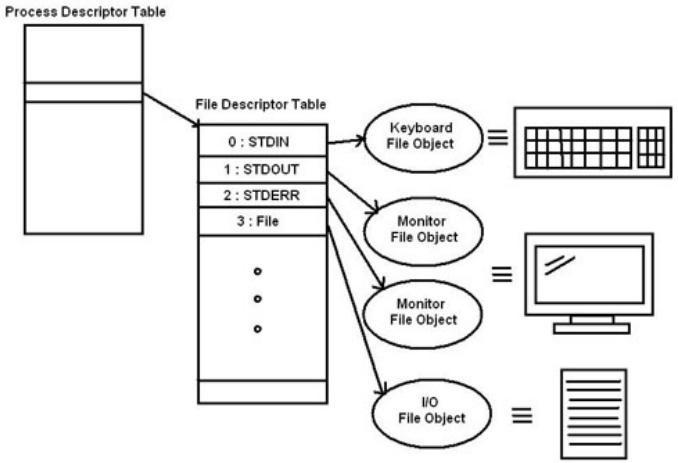
Homework 1 Q2: **redircmd**




```c
struct cmd*
redircmd(struct cmd *subcmd, char *file, int type)
{
  struct redircmd *cmd;

  cmd = malloc(sizeof(*cmd));
  memset(cmd, 0, sizeof(*cmd));
  cmd->type = type;
  cmd->cmd = subcmd;
  cmd->file = file;
  cmd->mode = (type == '<') ?  O_RDONLY : O_WRONLY|O_CREAT|O_TRUNC;
  cmd->fd = (type == '<') ? 0 : 1;
  return (struct cmd*)cmd;
}
```

Homework 1 Q2: **redircmd**






```
case REDIR:
  rcmd = (struct redircmd*)cmd;
  close(rcmd->fd);
  if(open(rcmd->file, rcmd->mode) < 0){
    printf(2, "open %s failed\n", rcmd->file);
    exit();
  }
  runcmd(rcmd->cmd);
  break;
```

Homework 1 Q2: **redircmd**

Homework 1 Q2: **redircmd with proper permission bits**

- (400) S_IRUSRS : Read permission bit for the owner of the file.
- (200) S_IWUSRS: Write permission bit for the owner of the file.

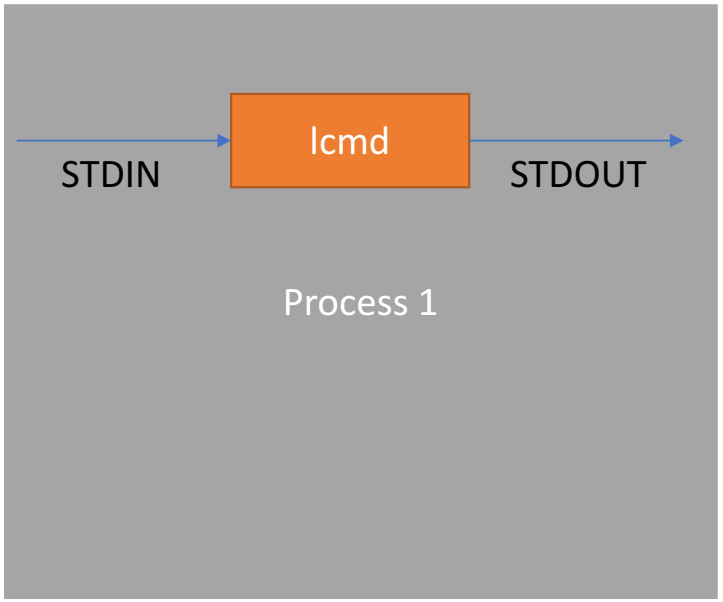
- (40) S_IRGRP : Read permission bit for the group owner of the file.
- (20) S_IWGRP : Write permission bit for the group owner of the file.

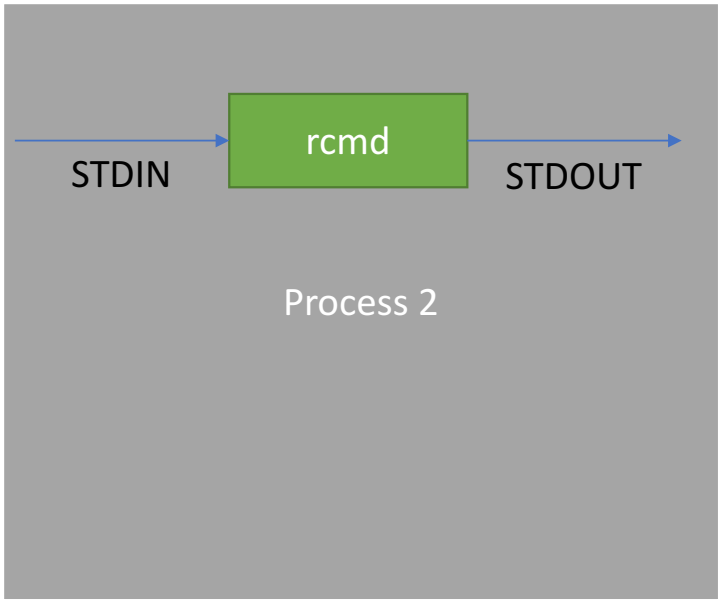
- (4) S_IROTH : Read permission bit for other users. Usually 04.
- (2) S_IWOTH : Write permission bit for other users. Usually 02.

```
case '>':
case '<':;
  int fd;
  rcmd = (struct redircmd *)cmd;

  if (rcmd->mode == O_RDONLY)
  {
    fd = open(rcmd->file, rcmd->mode);
  }
  else
  {
    // rw-rw-rw-
    fd = open(rcmd->file, rcmd->mode,
       S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH | S_IWOTH);
  }

  if (fd < 0)
  {
    perror("open:");
    break;
  }
  // close either 0 or 1 depending on the redir
  close(rcmd->fd);
  // dup the fd to the one that is closed
  dup2(fd, rcmd->fd);
  // run
  runcmd(rcmd->cmd);
  break;
```

Homework 1 Q3: **pipecmd**

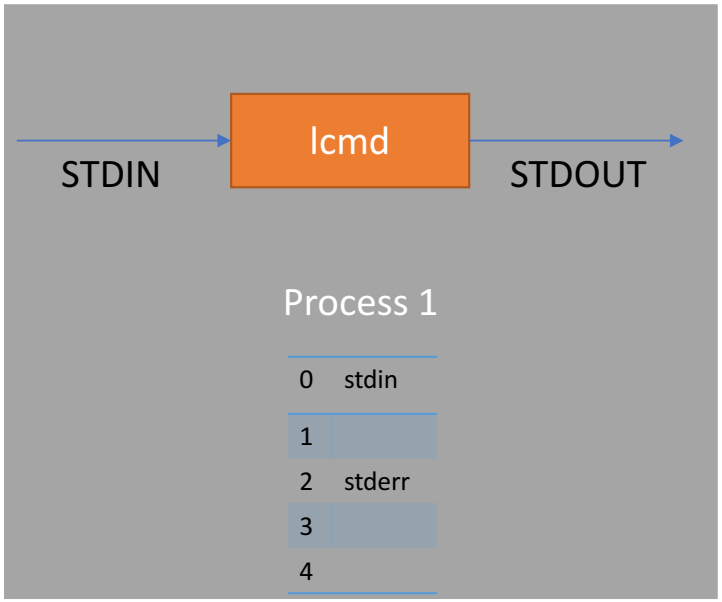
Homework 1 Q3: **pipecmd**

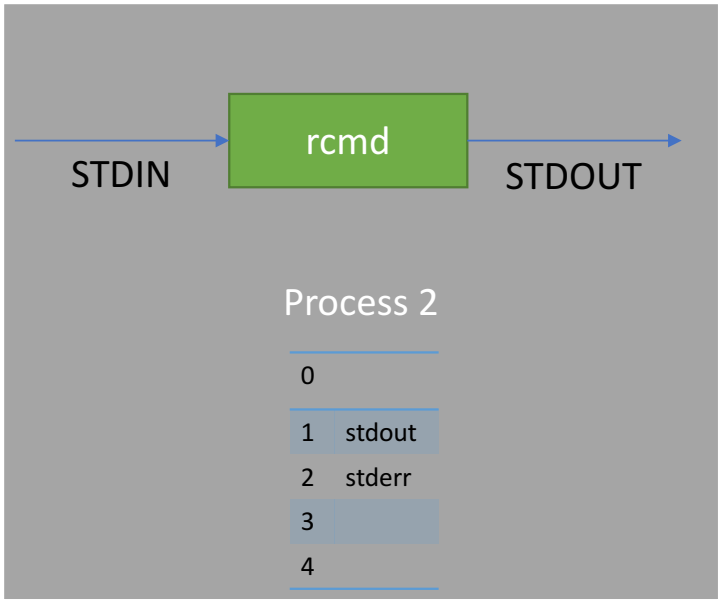





```
case PIPE:

  if(fork1() == 0){
    runcmd(pcmd->left);
  }
  if(fork1() == 0){
    runcmd(pcmd->right);
  }

  break;
```

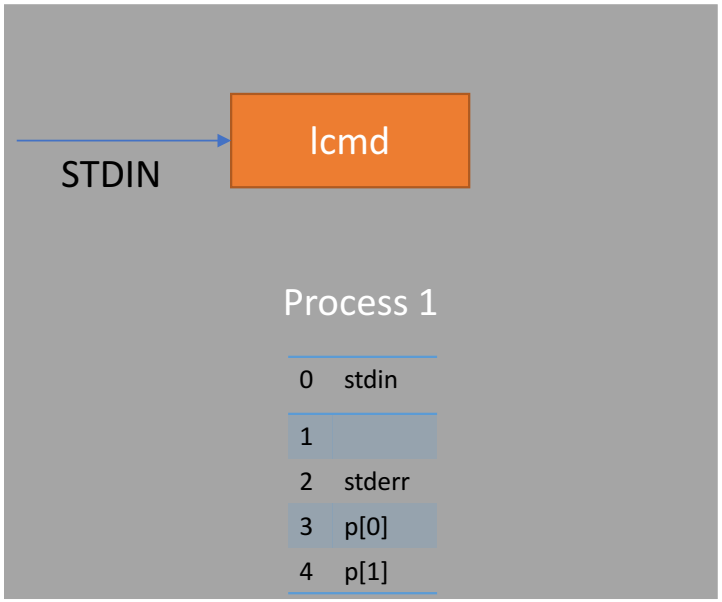
Homework 1 Q3: **pipecmd**

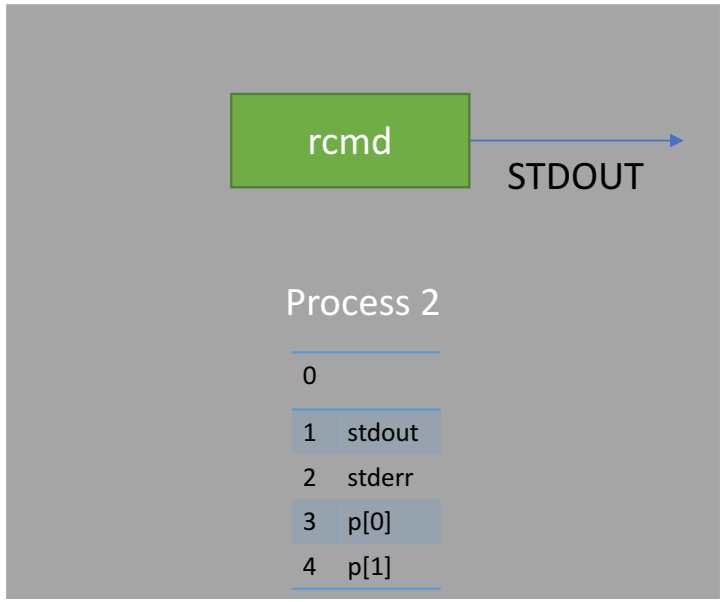





```
case PIPE:

  if(fork1() == 0){
    close(1);
    runcmd(pcmd->left);
  }
  if(fork1() == 0){
    close(0);
    runcmd(pcmd->right);
  }

  break;
```

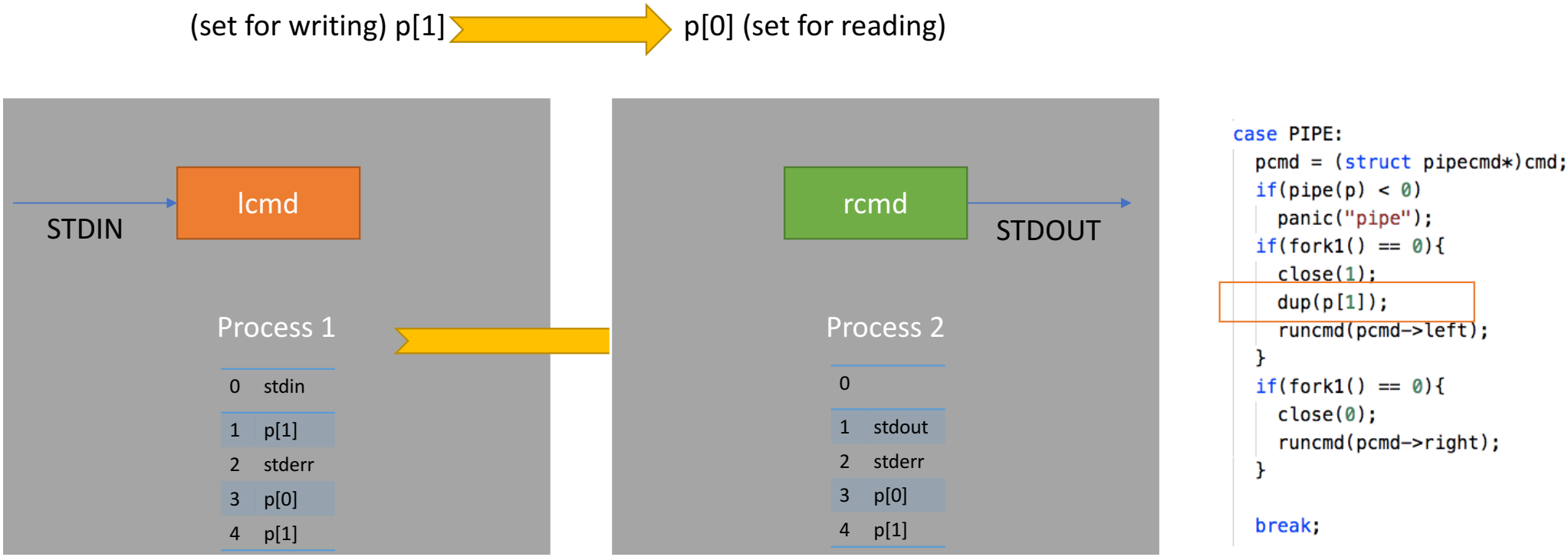
Homework 1 Q3: **pipecmd**





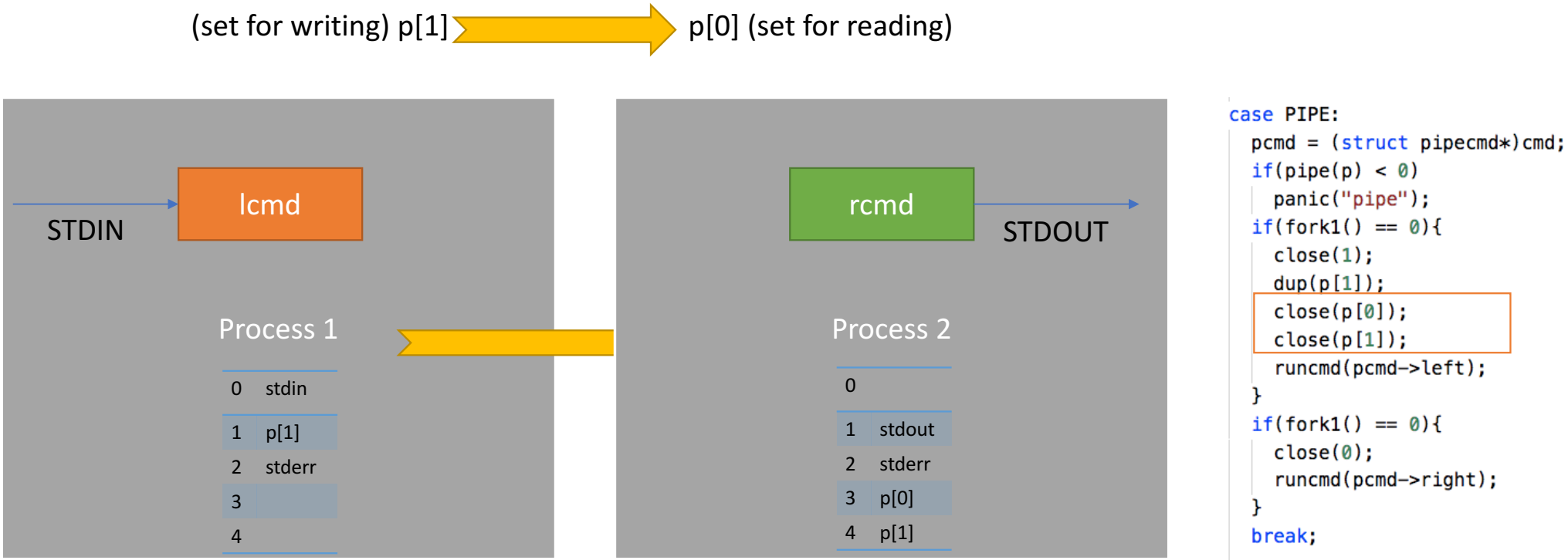

```
case PIPE:
  pcmd = (struct pipecmd*)cmd;
  if(pipe(p) < 0)
    panic("pipe");
  if(fork1() == 0){
    close(1);
    runcmd(pcmd->left);
  }
  if(fork1() == 0){
    close(0);
    runcmd(pcmd->right);
  }

  break;
```

Homework 1 Q3: **pipecmd**



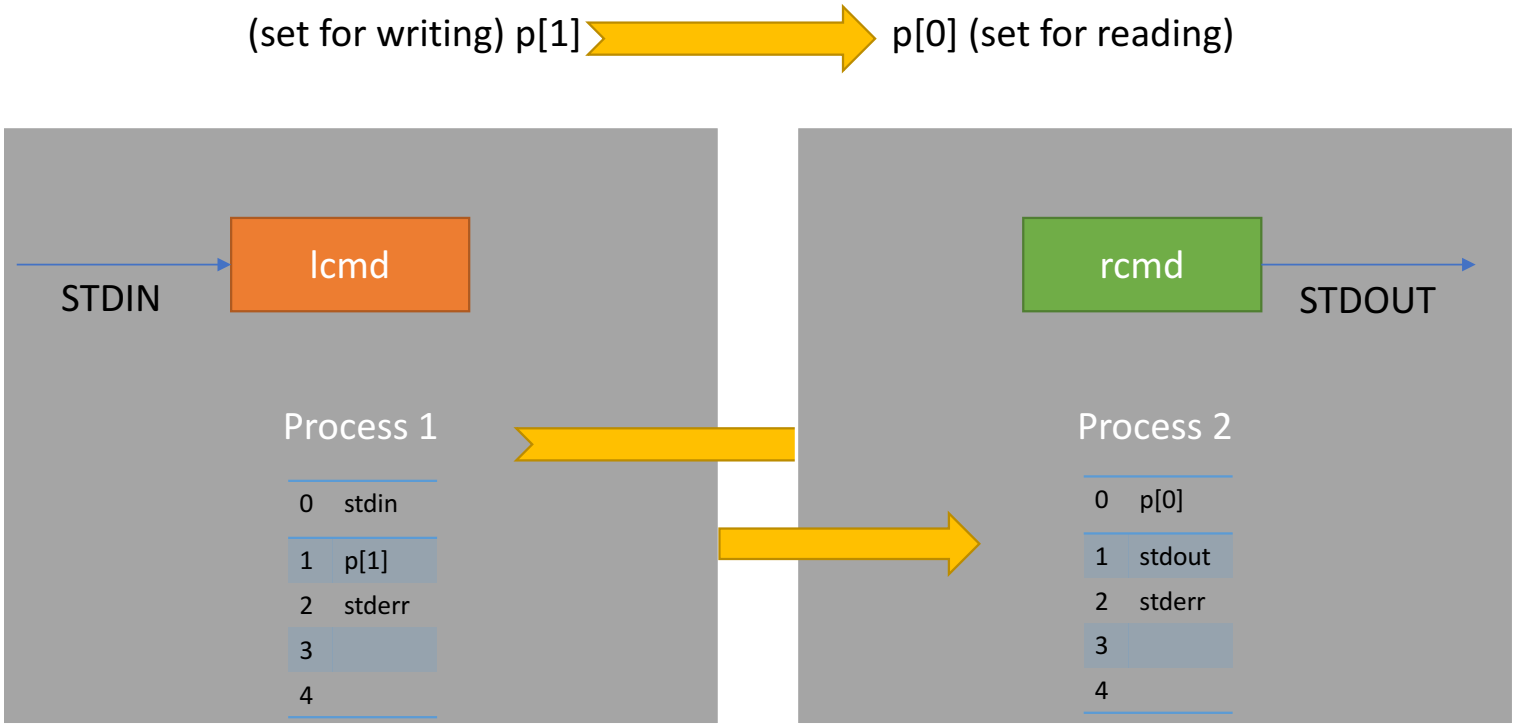

```
case PIPE:
  pcmd = (struct pipecmd*)cmd;
  if(pipe(p) < 0)
    panic("pipe");
  if(fork1() == 0){
    close(1);
    dup(p[1]);
    runcmd(pcmd->left);
  }
  if(fork1() == 0){
    close(0);
    runcmd(pcmd->right);
  }

  break;
```

Homework 1 Q3: **pipecmd**



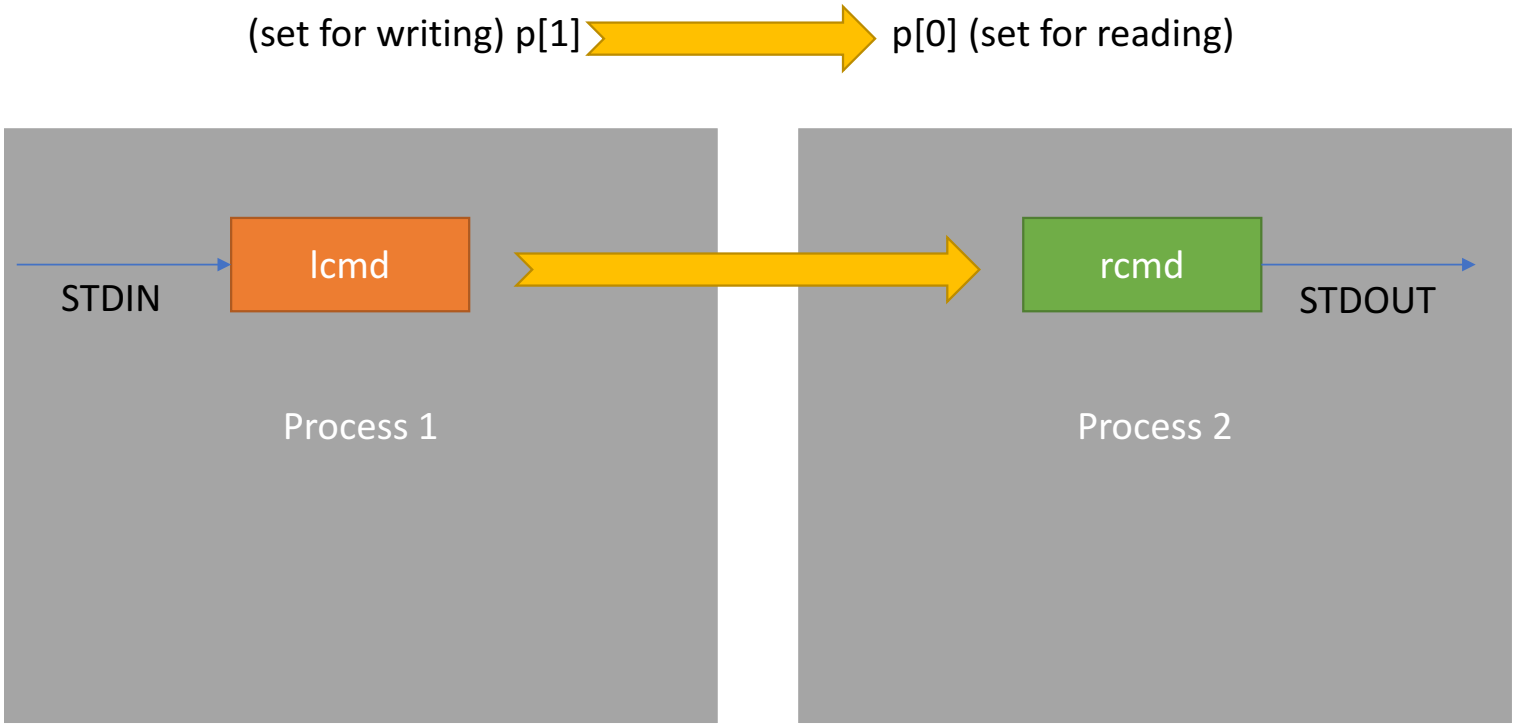

```
case PIPE:
  pcmd = (struct pipecmd*)cmd;
  if(pipe(p) < 0)
    panic("pipe");
  if(fork1() == 0){
    close(1);
    dup(p[1]);
    close(p[0]);
    close(p[1]);
    runcmd(pcmd->left);
  }
  if(fork1() == 0){
    close(0);
    runcmd(pcmd->right);
  }
  break;
```

Homework 1 Q3: **pipecmd**

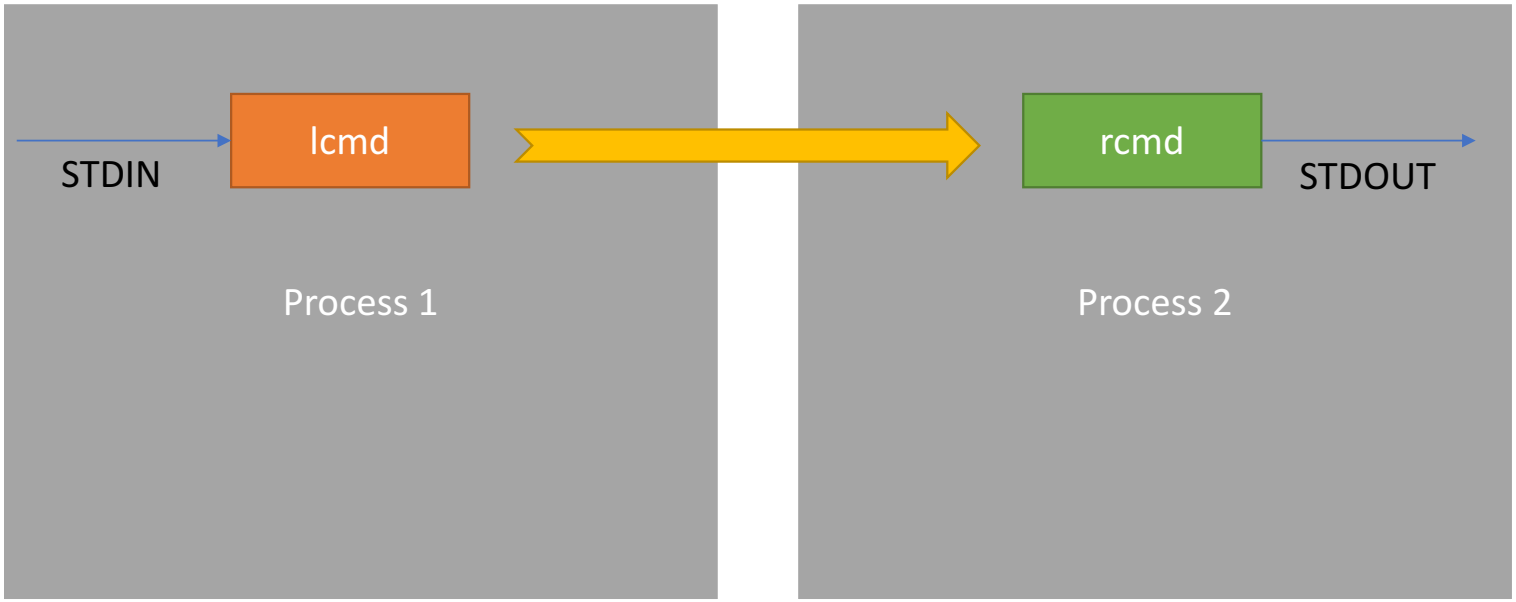



```
case PIPE:
  pcmd = (struct pipecmd*)cmd;
  if(pipe(p) < 0)
    panic("pipe");
  if(fork1() == 0){
    close(1);
    dup(p[1]);
    close(p[0]);
    close(p[1]);
    runcmd(pcmd->left);
  }
  if(fork1() == 0){
    close(0);
    dup(p[0]);
    close(p[0]);
    close(p[1]);
    runcmd(pcmd->right);
  }
  break;
```

Homework 1 Q3: **pipecmd**



Homework 1 Q3: **pipecmd**




```
case PIPE:
  pcmd = (struct pipecmd*)cmd;
  if(pipe(p) < 0)
    panic("pipe");
  if(fork1() == 0){
    close(1);
    dup(p[1]);
    close(p[0]);
    close(p[1]);
    runcmd(pcmd->left);
  }
  if(fork1() == 0){
    close(0);
    dup(p[0]);
    close(p[0]);
    close(p[1]);
    runcmd(pcmd->right);
  }
  close(p[0]);
  close(p[1]);
  break;
```

Homework 1 Q3: **pipecmd**


```
case PIPE:
  pcmd = (struct pipecmd*)cmd;
  if(pipe(p) < 0)
    panic("pipe");
  if(fork1() == 0){
    close(1);
    dup(p[1]);
    close(p[0]);
    close(p[1]);
    runcmd(pcmd->left);
  }
  if(fork1() == 0){
    close(0);
    dup(p[0]);
    close(p[0]);
    close(p[1]);
    runcmd(pcmd->right);
  }
  close(p[0]);
  close(p[1]);
  wait();
  wait();
  break;
```

Homework 1 Q3: **pipecmd**

- Can we do it with one fork ?

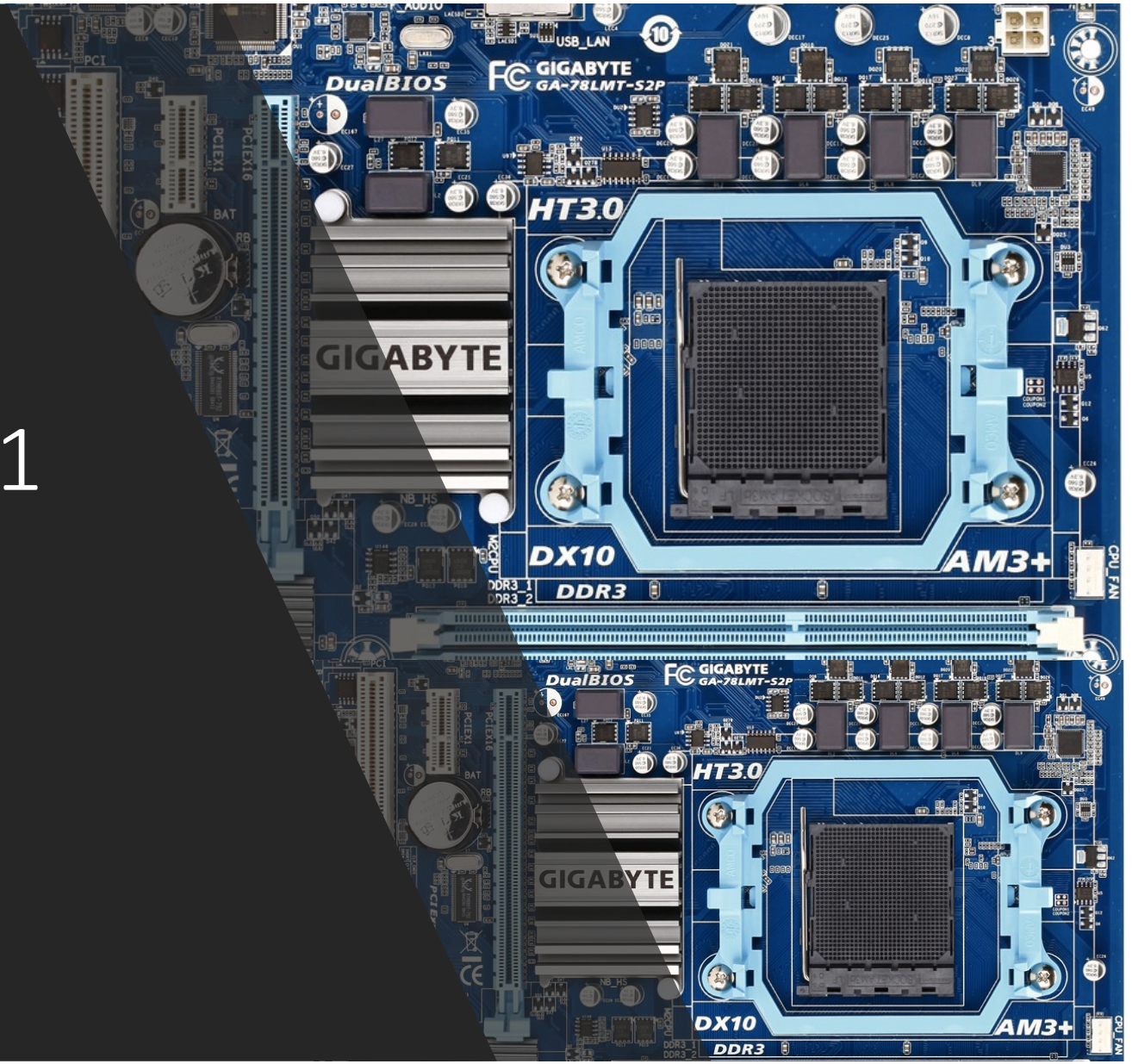
No, that's incorrect, since the left child might fill the pipe before finishing it's input, e.g., if you cat a file that is longer than a pipe. Although it might work for some instances, it won't work for all cases.

So both left and right have to run in parallel

Homework 1 Q3: **pipecmd Common Problems**

Homework 2: Q1

XV6 - Bootup



Can BIOS load kernel directly ?

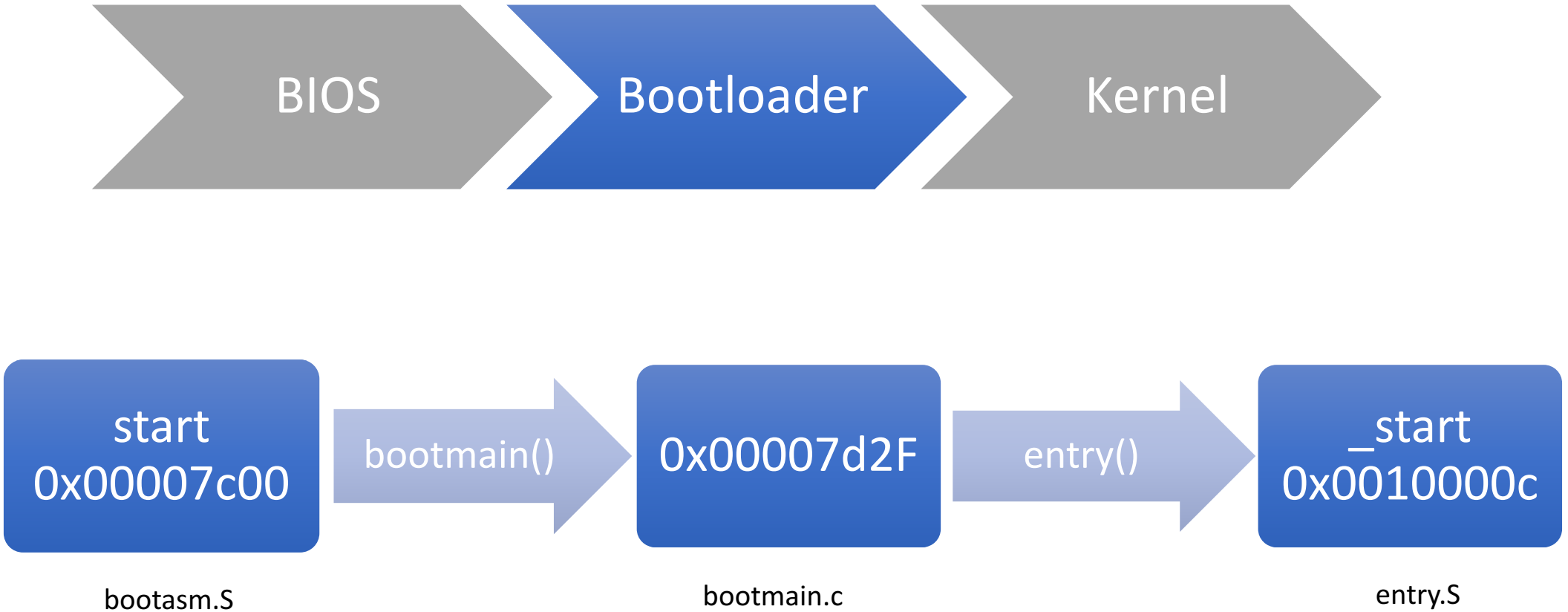
Loading a kernel involves different steps

These steps are different for each Operating Systems

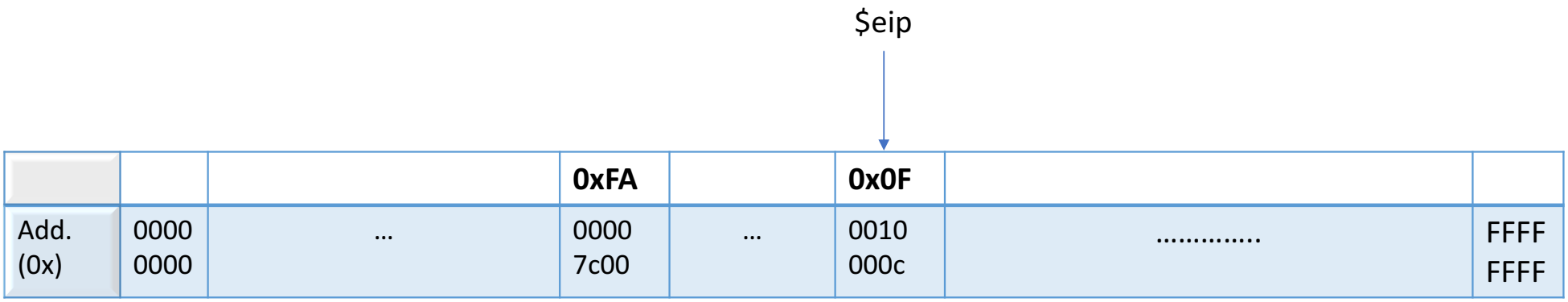
The boot loader step also lets you start one of several operating systems off different places on your disk.

0x00007d2F

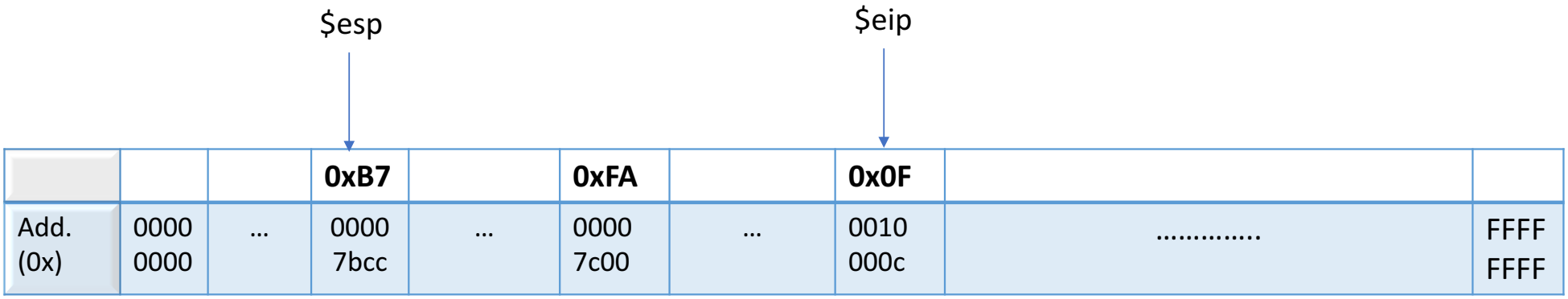
bootmain.c


BIOS
Bootloader
Kernel
start
0x00007c00
bootmain()
0x00007d2F
entry()
_start
0x0010000c
bootasm.S
bootmain.c
entry.S

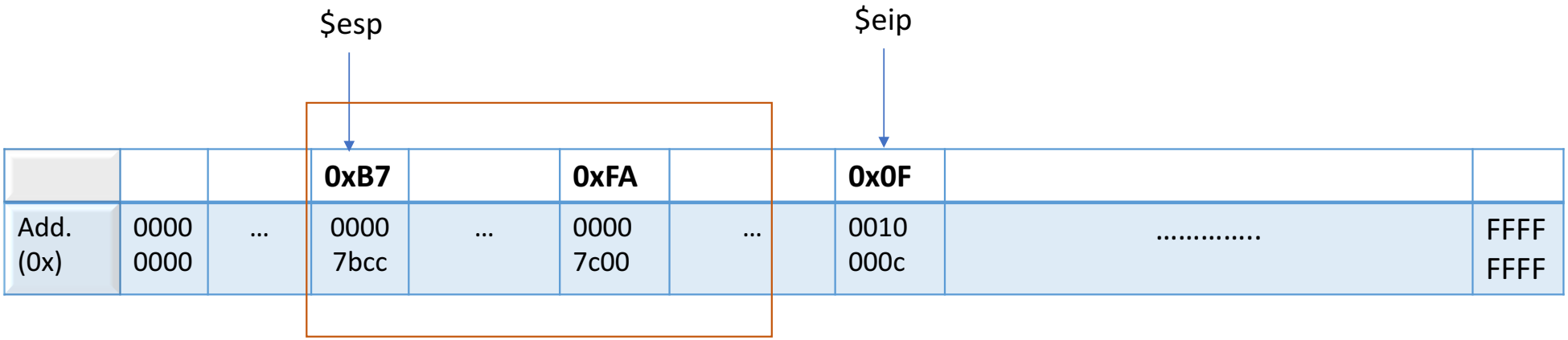
Homework 2: Q1



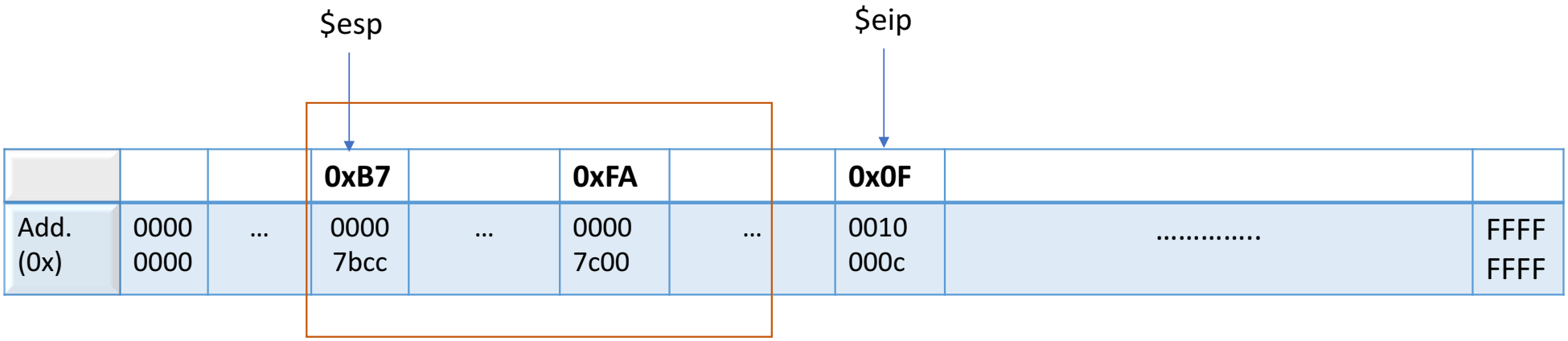
Homework 2: Q1



Homework 2: Q1



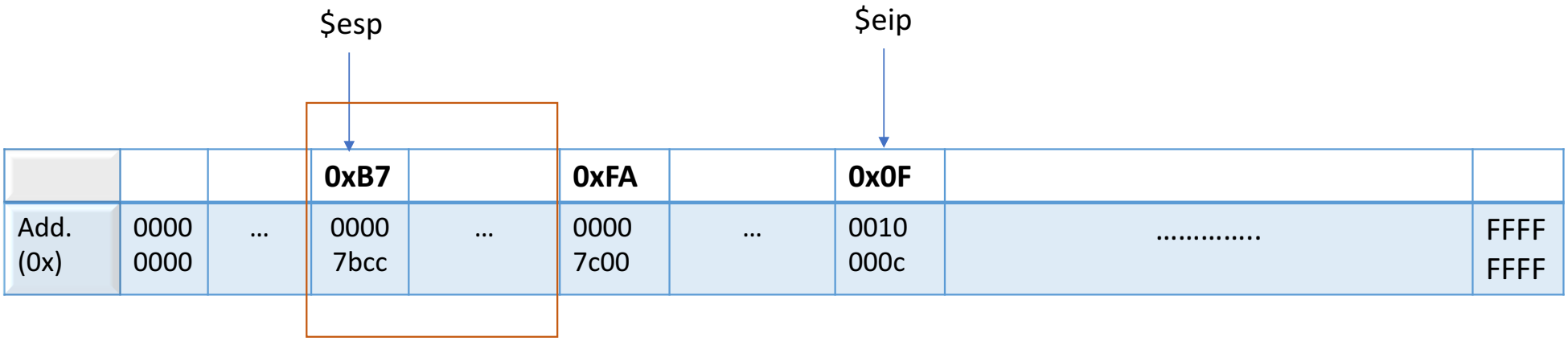
Homework 2: Q1



x/24x

x/24xw

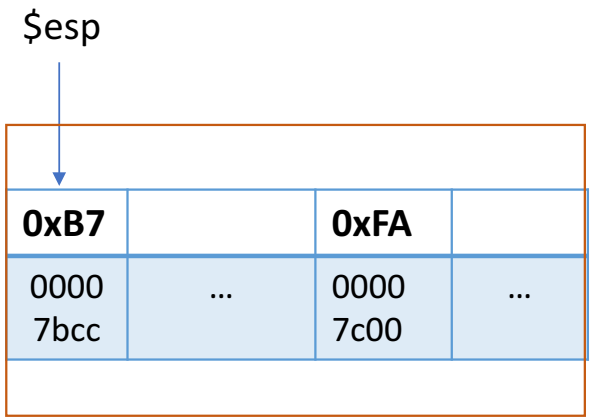
Homework 2: Q1



x/12x

x/12xw

Homework 2: Q1




```
0x7bcc: 0x00007db7      0x00000000      0x00000000      0x00000000
0x7bdc: 0x00000000      0x00000000      0x00000000      0x00000000
0x7bec: 0x00000000      0x00000000      0x00000000      0x00000000
0x7bfc: 0x00007c4d      0x8ec031fa      0x8ec08ed8      0xa864e4d0
0x7c0c: 0xb0fa7502      0xe464e6d1      0x7502a864      0xe6dfb0fa
0x7c1c: 0x16010f60      0x200f7c78      0xc88366c0      0xc0220f01
```

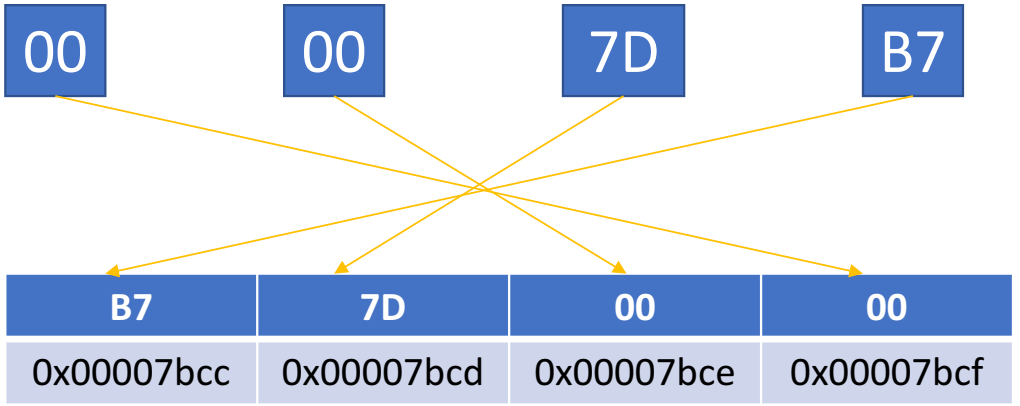
Homework 2: Q1

0x00007bcc	0x00007bcd	0x00007bce	0x00007bcf

0xB7		**0xFA**	
0000 7bcc	…	0000 7c00	…

```
0x7bcc: 0x00007db7  0x00000000  0x00000000  0x00000000
0x7bdc: 0x00000000  0x00000000  0x00000000  0x00000000
0x7bec: 0x00000000  0x00000000  0x00000000  0x00000000
0x7bfc: 0x00007c4d  0x8ec031fa  0x8ec08ed8  0xa864e4d0
0x7c0c: 0xb0fa7502  0xe464e6d1  0x7502a864  0xe6dfb0fa
0x7c1c: 0x16010f60  0x200f7c78  0xc88366c0  0xc0220f01
```

Homework 2: Q1



0xB7		0xFA	
0000 7bcc	…	0000 7c00	…

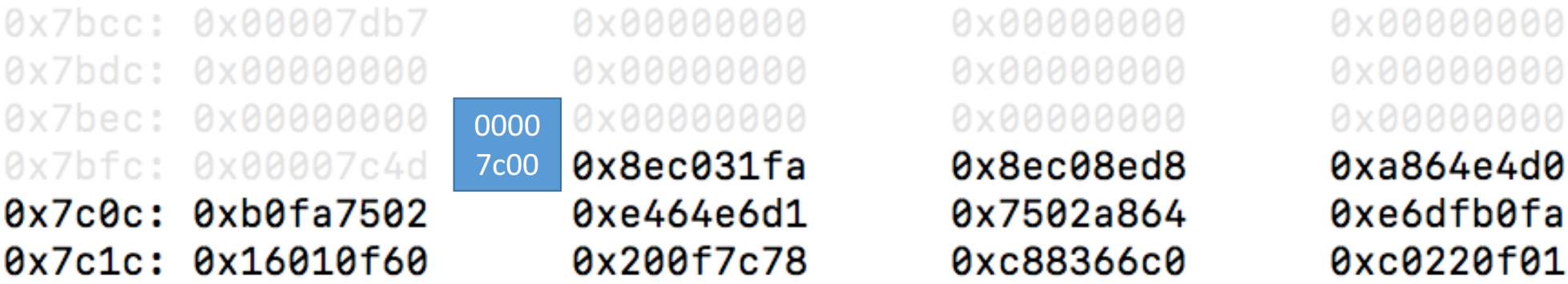
```
0x7bcc: 0x00007db7  0x00000000  0x00000000  0x00000000
0x7bdc: 0x00000000  0x00000000  0x00000000  0x00000000
0x7bec: 0x00000000  0x00000000  0x00000000  0x00000000
0x7bfc: 0x00007c4d  0x8ec031fa  0x8ec08ed8  0xa864e4d0
0x7c0c: 0xb0fa7502  0xe464e6d1  0x7502a864  0xe6dfb0fa
0x7c1c: 0x16010f60  0x200f7c78  0xc88366c0  0xc0220f01
```

Homework 2: Q1

```
(gdb) x/1xw $esp
0x7bcc: 0x00007db7
(gdb) x/4xb $esp
0x7bcc: 0xb7    0x7d    0x00    0x00
```

0xB7		0xFA	
0000 7bcc	…	0000 7c00	…

```
0x7bcc: 0x00007db7  0x00000000  0x00000000  0x00000000
0x7bdc: 0x00000000  0x00000000  0x00000000  0x00000000
0x7bec: 0x00000000  0x00000000  0x00000000  0x00000000
0x7bfc: 0x00007c4d  0x8ec031fa  0x8ec08ed8  0xa864e4d0
0x7c0c: 0xb0fa7502  0xe464e6d1  0x7502a864  0xe6dfb0fa
0x7c1c: 0x16010f60  0x200f7c78  0xc88366c0  0xc0220f01
```


0x7bcc: 0x00007db7 0x00000000 0x00000000 0x00000000
0x7bdc: 0x00000000 0x00000000 0x00000000 0x00000000
0x7bec: 0x00000000 0x00000000 0x00000000 0x00000000
0x7bfc: 0x00007c4d 0x8ec031fa 0x8ec08ed8 0xa864e4d0
0x7c0c: 0xb0fa7502 0xe464e6d1 0x7502a864 0xe6dfb0fa
0x7c1c: 0x16010f60 0x200f7c78 0xc88366c0 0xc0220f01
0000
7c00

Instructions opcodes

```
.globl start
start:
  cli                           # BIOS enabled interrupts; disable
    7c00: fa                      cli

  # Zero data segment registers DS, ES, and SS.
  xorw    %ax,%ax               # Set %ax to zero
    7c01: 31 c0                   xor    %eax,%eax
  movw    %ax,%ds               # -> Data Segment
    7c03: 8e d8                   mov    %eax,%ds
  movw    %ax,%es               # -> Extra Segment
    7c05: 8e c0                   mov    %eax,%es
  movw    %ax,%ss               # -> Stack Segment
    7c07: 8e d0                   mov    %eax,%ss
```

```
0x7bcc: 0x00007db7    0x00000000    0x00000000    0x00000000
0x7bdc: 0x00000000    0x00000000    0x00000000    0x00000000
0x7bec: 0x00000000    0x00000000    0x00000000    0x00000000
0x7bfc: 0x00007c4d    0x8ec031fa    0x8ec08ed8    0xa864e4d0
0x7c0c: 0xb0fa7502    0xe464e6d1    0x7502a864    0xe6dfb0fa
0x7c1c: 0x16010f60    0x200f7c78    0xc88366c0    0xc0220f01
```

0000
7c00


```
0x7bcc: 0x00007db7  0x00000000  0x00000000  0x00000000
0x7bdc: 0x00000000  0x00000000  0x00000000  0x00000000
0x7bec: 0x00000000  0x00000000  0x00000000  0x00000000
0x7bfc: 0x00007c4d  0x8ec031fa  0x8ec08ed8  0xa864e4d0
0x7c0c: 0xb0fa7502  0xe464e6d1  0x7502a864  0xe6dfb0fa
0x7c1c: 0x16010f60  0x200f7c78  0xc88366c0  0xc0220f01
```

bootasm.S

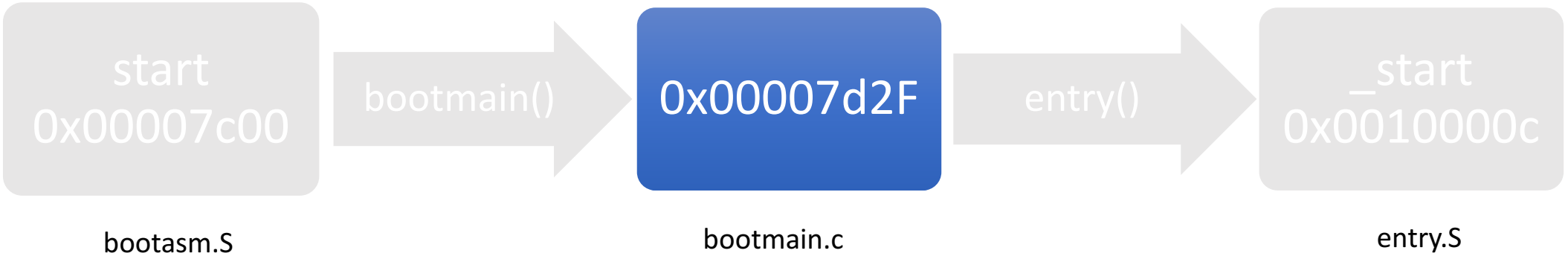
bootmain.c

entry.S

Address of the next instruction to execute after return from call

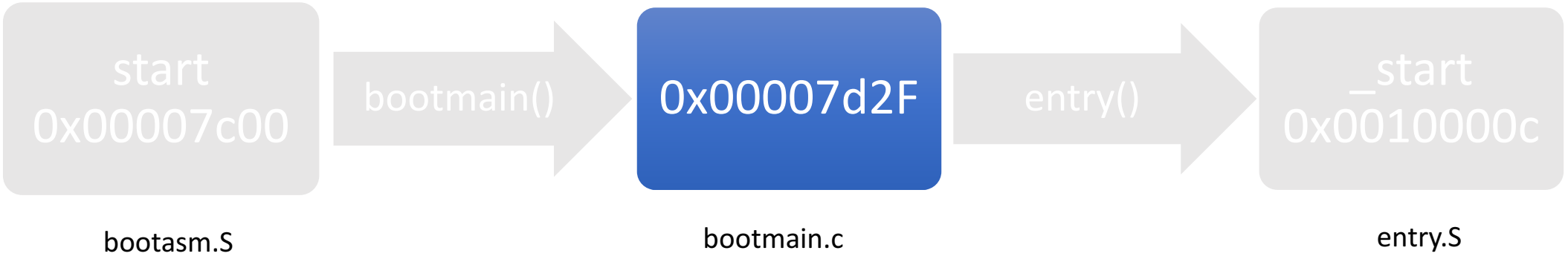
```
call    bootmain
  7c48: e8 e2 00 00 00          call   7d2f <bootmain>

# If bootmain returns (it shouldn't), trigger a Bochs
# breakpoint if running under Bochs, then loop.
movw    $0x8a00, %ax            # 0x8a00 -> port 0x8a00
  7c4d: 66 b8 00 8a             mov    $0x8a00,%ax
```

```
0x7bcc: 0x00007db7
0x7bdc: 0x00000000
0x7bec: 0x00000000
0x7bfc: 0x00007c4d
0x7c0c: 0xb0fa7502
0x7c1c: 0x16010f60
```



1. Push the value of EBP (0) onto the stack, and then copy the value of ESP into EBP

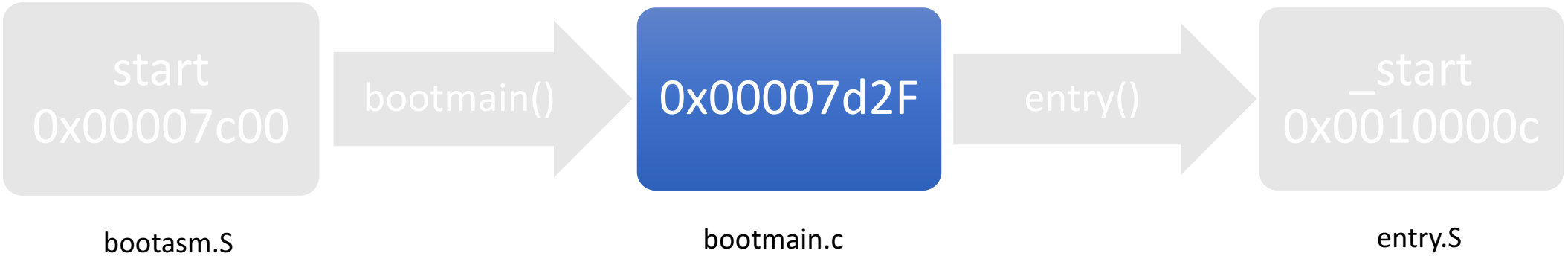
```
0x7bcc: 0x00007db7  0x00000000  0x00000000  0x00000000
0x7bdc: 0x00000000  0x00000000  0x00000000  0x00000000
0x7bec: 0x00000000  0x00000000  0x00000000  0x00000000
0x7bfc: 0x00007c4d  0x8ec031fa  0x8ec08ed8  0xa864e4d0
0x7c0c: 0xb0fa7502  0xe464e6d1  0x7502a864  0xe6dfb0fa
0x7c1c: 0x16010f60  0x200f7c78  0xc88366c0  0xc0220f01
```



1. Push the value of EBP (0) onto the stack, and then copy the value of ESP into EBP

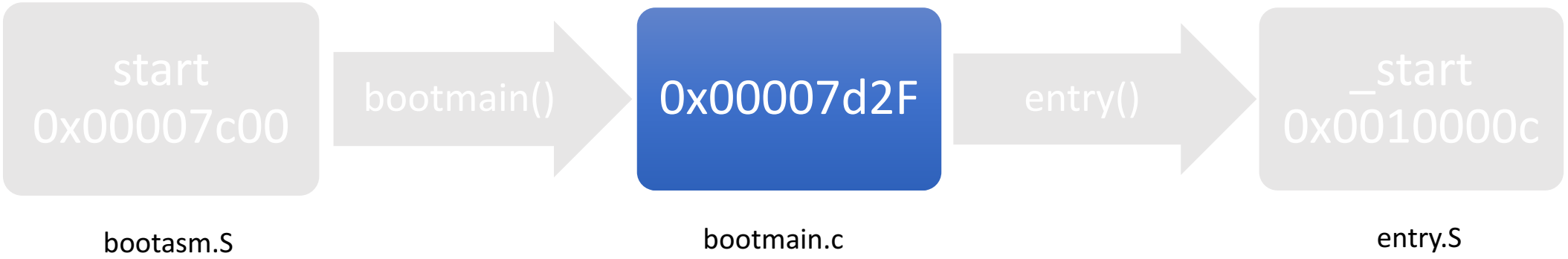
2. Saves three registers: %edi, %esi, and %ebx

```
0x7bcc: 0x00007db7  0x00000000  0x00000000  0x00000000
0x7bdc: 0x00000000  0x00000000  0x00000000  0x00000000
0x7bec: 0x00000000  0x00000000  0x00000000  0x00000000
0x7bfc: 0x00007c4d  0x8ec031fa  0x8ec08ed8  0xa864e4d0
0x7c0c: 0xb0fa7502  0xe464e6d1  0x7502a864  0xe6dfb0fa
0x7c1c: 0x16010f60  0x200f7c78  0xc88366c0  0xc0220f01
```



sub $0x1c,%esp

```
0x7bcc: 0x00007db7  0x00000000  0x00000000  0x00000000
0x7bdc: 0x00000000  0x00000000  0x00000000  0x00000000
0x7bec: 0x00000000  0x00000000  0x00000000  0x00000000
0x7bfc: 0x00007c4d  0x8ec031fa  0x8ec08ed8  0xa864e4d0
0x7c0c: 0xb0fa7502  0xe464e6d1  0x7502a864  0xe6dfb0fa
0x7c1c: 0x16010f60  0x200f7c78  0xc88366c0  0xc0220f01
```



- Why 28bytes and not 20bytes ?
- It's an optimization. The compiler allocates the space for both local variables and for the arguments to the *readseg((uchar*)elf, 4096, 0)* function call that is about to be called

```
0x7bcc: 0x00007db7  0x00000000  0x00000000  0x00000000
0x7bdc: 0x00000000  0x00000000  0x00000000  0x00000000
0x7bec: 0x00000000  0x00000000  0x00000000  0x00000000
0x7bfc: 0x00007c4d  0x8ec031fa  0x8ec08ed8  0xa864e4d0
0x7c0c: 0xb0fa7502  0xe464e6d1  0x7502a864  0xe6dfb0fa
0x7c1c: 0x16010f60  0x200f7c78  0xc88366c0  0xc0220f01
```

bootasm.S

bootmain.c

entry.S

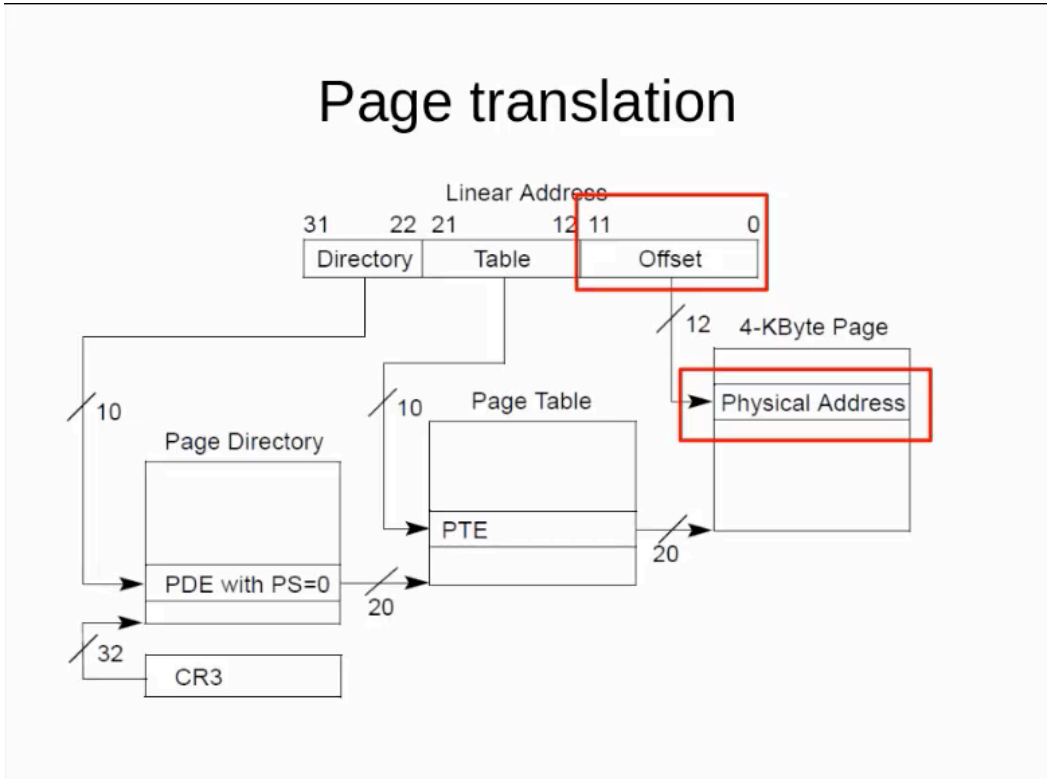
```
0x7bcc: 0x00007db7
0x7bdc: 0x00000000
0x7bec: 0x00000000
0x7bfc: 0x00007c4d
0x7c0c: 0xb0fa7502
0x7c1c: 0x16010f60
```

```
// Call the entry point from the ELF header.
// Does not return!
entry = (void(*)(void))(elf->entry);
entry();
  7db1: ff 15 18 00 01 00     call   *0x10018
}
  7db7: 83 c4 1c              add    $0x1c,%esp
```


```
0x7bcc: 0x00007db7  0x00000000  0x00000000  0x00000000
0x7bdc: 0x00000000  0x00000000  0x00000000  0x00000000
0x7bec: 0x00000000  0x00000000  0x00000000  0x00000000
0x7bfc: 0x00007c4d  0x8ec031fa  0x8ec08ed8  0xa864e4d0
0x7c0c: 0xb0fa7502  0xe464e6d1  0x7502a864  0xe6dfb0fa
0x7c1c: 0x16010f60  0x200f7c78  0xc88366c0  0xc0220f01
```

Homework 2
Question 2

Lecture 5&6 : Address Translation



- Virtual address being divided into three parts and the widths of each part
- Page directory indexing (with flags
- Page table indexing
- Final physical address composition

References

- http://web.cs.ucla.edu/classes/fall08/cs111/scribe/4/index.html
- https://linux.die.net/man/3/open
- https://www.gnu.org/software/libc/manual/html_node/Permission-Bits.html