

به نام خدا

گزارش پروژه ی اول آزمایشگاه درس سیستم عامل

اعضای گروه:

فاطمه حقیقی سید حمید تراشیون

آشنایی با سیستم عامل xv6

1- معماری سیستم عامل xv6 چیست؟ چه دلایلی در دفاع از نظر خود دارید؟

معماری این سیستم عامل از نوع Monolithic می باشد. در این نوع معماری تمام component ها و بخش های سیستم عامل در قسمت kernel قرار دارد. این component ها می توانند به طور مستقیم با یکدیگر توسط function call و با کرنل در ارتباط باشند. همچنین تمام اجزای داخل کرنل اجازه دسترسی نامحدود به سیستم کامپیوتر را دارند. این نوع معماری برای سیستم عامل های ساده استفاده می شود. چون اگر هسته ی سیستم پیچیده شود، این نوع معماری پیاده سازی سختی خواهد داشت.

2- مفهوم file descriptor در سیستم عامل های مبتنی بر unix چیست؟ این مساله چه مزیتی برای طراح سیستم عامل دارد؟

این مفهوم نشان دهنده ی یک object است که در آن می نویسیم یا از آن می خوانیم. هر فرآیند دارای تعدادی file descriptor می باشد که در pre-process table آن فرآیند قرار دارد. بر اساس یک قرارداد:

File descriptor = 0 is for standard input

File descriptor = 1 is for standard output

File descriptor = 2 use for fault

از جمله مزایای استفاده از file descriptor این است که، کسی که از file descriptor استفاده می کند تفاوتی بین فایل، pipe و device در نظر نمی گیرد و همه ی آن ها را شبیه به یک رشته ای از بایت ها می بیند. ارتباط متقابل file descriptor و سیستم کال fork باعث می شود پیاده سازی IO reduction برای طراح ساده تر شود.

3- فراخوانی های سیستمی fork و exec چه عملی انجام می دهند؟ از نظر طراحی، ادغام نکردن این دو چه مزیتی دارد؟

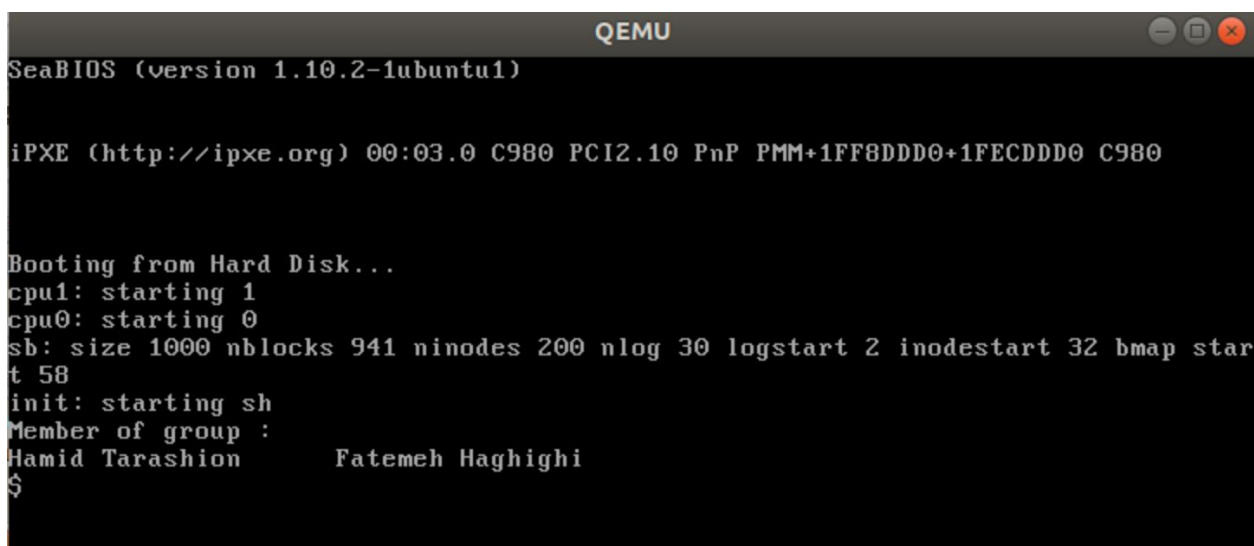
فراخوانی سیستمی fork که تنها فراخوانی سیستمی ای است که باعث ایجاد یک فرآیند جدید می شود، باعث می شود درون یک فرآیند، فرآیند دیگری بوجود بیاید که بین این دو فرآیند رابطه ی پدر و فرزندی برقرار است، که هر دو فرآیند به طور موازی اجرا می شوند مگر آنکه از فراخوانی سیستمی wait استفاده کنیم که در این حالت فرآیند پدر منتظر میماند تا اولین فرزند آن کارش تمام شود و

terminate شود. پس از آنکه فرآیند جدید بوجود آمد حافظه ی اختصاص داده شده به آن توسط محتوای فرآیند پدر، پر می شود و file descriptor فرزند یک کپی از file descriptor فرآیند پدر است. این فراخوانی سیستمی دو مقدار برمیگرداند، اگر در محیط فرآیند پدر باشیم pid پدر را برمی گرداند و اگر در محیط فرآیند فرزند باشیم، 0 برمی گرداند.

فراخوانی سیستمی دیگر exec می باشد که نام برنامه ای که می خواهیم آن را اجرا کنیم و ورودی های آن برنامه را به عنوان ورودی می گیرد. این فراخوانی سیستمی محتوای داخل حافظه را پاک می کند و حافظه ی برنامه ی جدید را جایگزین می کند. با این تفاوت که file descriptor برنامه ی قبلی را حذف نمی کند.

بعد از آنکه توسط فراخوانی سیستمی fork یک فرآیند جدید ایجاد کردیم، برای اجرا شدن برنامه ای که مورد نظر ما است باید فراخوانی سیستمی exec را صدا بزنیم. فراخوانی سیستمی exec، تمام حافظه را پاک می کند اما file descriptor قبلی را حفظ می کند. پیش از آنکه این فراخوانی سیستمی را صدا بزنیم، به وسیله ی دستوراتی مثل open, close میتوان file descriptor تغییر بدیم و standard input و standard output را مطابق برنامه ای که می خواهیم اجرا شود، تغییر دهیم. اگر فراخوانی های سیستمی fork و exec یکی بودند و میان آن ها تفاوتی وجود داشت، باید تغییر مربوط به standard input و standard output را به داخل هر برنامه ای که می خواستیم آن را اجرا کنیم، منتقل می کردیم و این باعث می شود طراحی های پیچیده تری برای shell داشته باشیم تا standard input و standard output را دوباره برنامه ریزی کنیم.

اضافه کردن متن به Boot Message



```
QEMU
SeaBIOS (version 1.10.2-1ubuntu1)

iPXE (http://ipxe.org) 00:03.0 C980 PCI2.10 PnP PMM+1FF8DDD0+1FECDDD0 C980

Booting from Hard Disk...
cpu1: starting 1
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
Member of group :
Hamid Tarashion      Fatemeh Haghighi
$
```

بوسیله ی اضافه کردن printf در اول for در فایل init.c صورت می گیرد.

اشکال زدایی

به دست آوردن آدرس مربوط به `_start`

```
fatemeh@asus-pro: ~/Documents/university/semester_6/OS/projects/lab/p1/xv6-public
File Edit View Search Terminal Help
→ xv6-public git:(master) X nm kernel | grep _start
8010a48c D _binary_entryother_start
8010a460 D _binary_initcode_start
0010000c T _start
→ xv6-public git:(master) X
```

سپس به وسیله ی دستور `make qemu-gdb` سیستم عامل را در وضعیت اشکال زدایی اجرا کردیم. در ترمینالی دیگر به وسیله ی دستور `gdb kernel` می توان سیستم عاملی را که در وضعیت اشکال زدایی اجرا کردیم به `gdb` متصل کنیم و سپس آن را اشکال زدایی کنیم. دستور مربوط به اتصال:

```
---Type <return> to continue, or q <return> to quit---
info "(gdb)Auto-loading safe path"
(gdb) target remote localhost:26000
Remote debugging using localhost:26000
0x00000fff0 in ?? ()
```

دستور مربوط به قرار دادن `breakpoint` در آدرس `_start` و ادامه دادن اجرا ی آن

```
(gdb) break _start
Breakpoint 1 at 0x10000c
(gdb) continue
Continuing.
```

بدست آوردن وضعیت رجیسترها در این حالت :

```
(gdb) info registers
eax          0x0          0
ecx          0x0          0
edx          0x1f0       496
ebx          0x10074     65652
esp          0x7bdc     0x7bdc
ebp          0x7bf8     0x7bf8
esi          0x10074     65652
edi          0x0          0
eip          0x10000c    0x10000c
eflags       0x46       [ PF ZF ]
cs           0x8          8
ss           0x10         16
ds           0x10         16
es           0x10         16
fs           0x0          0
gs           0x0          0
(gdb)
```

در معماری x86 رجیسترهای esp و ebp نشانگر چه چیزی هستند؟ در شرایط فعلی مقدار آن ها چیست و چه نتیجه ای از مقدار کنونی آن ها می گیرید؟
رجیستر ۳۲ بیتی ebp یک اشاره گری است که به آخرین خانه ی پر stack اشاره می کند و در واقع شروع یک framework است. رجیستر esp اشاره گری به اولین خانه ی خالی stack می باشد. 28. خانه بین رجیستر های esp و ebp قرار دارد.

اجرای یک برنامه ی سطح کاربر

تصویر زیر بیانگر یک برنامه سطح کاربر به نام middle.c است که ۷ مقدار از ورودی گرفته و یک فایل به نام result.txt می سازد و میانه ی اعداد را در آن فایل می نویسد و نیز pid برنامه ی فعلی را نمایش می دهد.

تابع sort استفاده شده در این برنامه:

```
1 #include "types.h"
2 #include "stat.h"
3 #include "fcntl.h"
4 #include "user.h"
5 #define ARRAY_SIZE 7
6 void sort_func(int* array)
7 {
8     int temp;
9     for(int i = 0; i < ARRAY_SIZE; i++)
10    {
11        temp = 0;
12        for(int j = i + 1; j < ARRAY_SIZE; j++)
13        {
14            if(array[j] < array[i])
15            {
16                temp = array[j];
17                array[j] = array[i];
18                array[i] = temp;
19            }
20        }
21    }
22 }
23
24 }
```

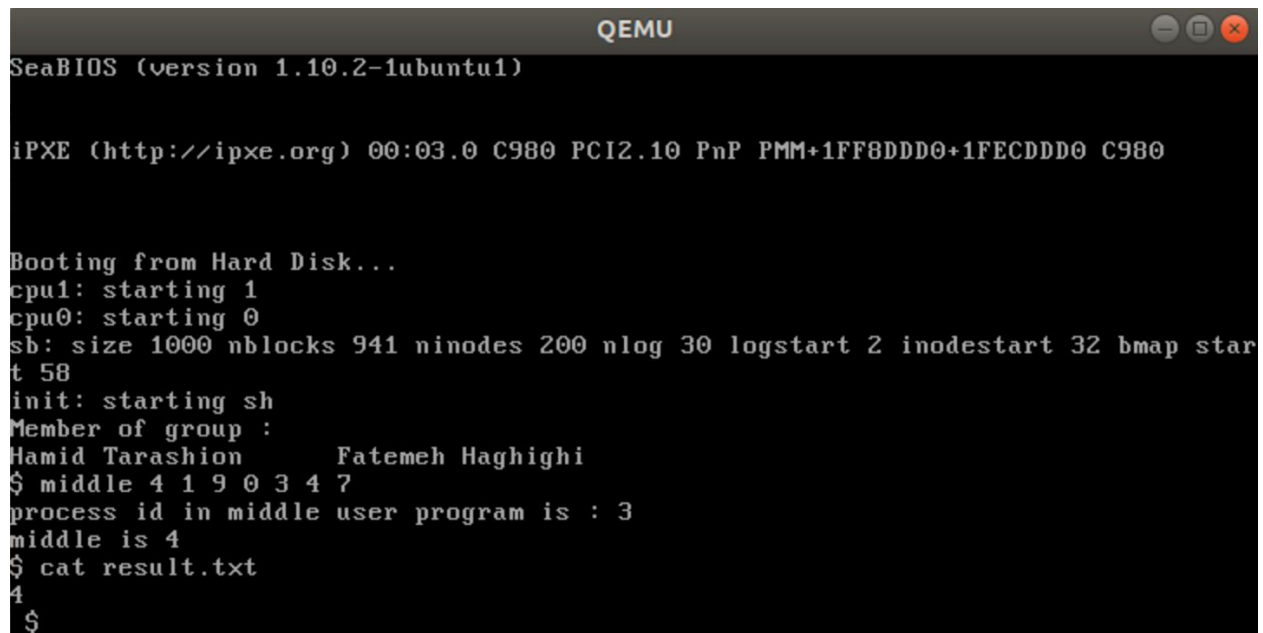
بدنه ی اصلی برنامه:

```

25 int main(int argc, char* argv[])
26 {
27     int pid = 0;
28     pid = getpid();
29     printf(1, "process id in middle user program is : %d\n", pid);
30     int array[ARRAY_SIZE];
31     for(int i = 0; i < ARRAY_SIZE; i++)
32         array[i] = atoi(argv[i + 1]);
33     sort_func(array);
34     char out[15];
35     int num = array[3];
36     int i = 0;
37     while(num > 0)
38     {
39         out[i] = (char)((num % 10) + '0');
40         num = num / 10;
41         i++;
42     }
43     printf(1, "middle is %s\n", out);
44
45     int file = open("result.txt", O_CREATE | O_WRONLY);
46     if(file != -1)
47     {
48         write(file, out, strlen(out) + 1);
49         write(file, "\n", strlen("\n") + 1);
50     }
51
52     exit();
53 }

```

خروجی برنامه و محتوای فایل `result.txt`:



```

QEMU
SeaBIOS (version 1.10.2-1ubuntu1)

iPXE (http://ipxe.org) 00:03.0 C980 PCI2.10 PnP PMM+1FF8DDD0+1FECDDD0 C980

Booting from Hard Disk...
cpu1: starting 1
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap star
t 58
init: starting sh
Member of group :
Hamid Tarashion      Fatemeh Haghighi
$ middle 4 1 9 0 3 4 7
process id in middle user program is : 3
middle is 4
$ cat result.txt
4
$

```

مقدمه ای درباره سیستم عامل و `xv6`

1. سه وظیفه اصلی سیستم عامل را بیان نمایید.

1- مدیریت منابع کامپیوتر مانند واحد پردازش مرکزی، حافظه، دیسک درایو و چاپگر

2- ایجاد کردن رابط کاربری بین لایه ی سخت افزار و لایه ی کاربر و برنامه ها

3- اجرا و ارائه خدمات برای نرم افزاری های کاربردی

2. فایل های اصلی سیستم عامل xv6 در صفحه یک کتاب xv6 لیست شده اند. به طور مختصر هر گروه را توضیح دهید.

: Basic Headers

در این دسته یک سری تعارف اولیه، struct ها و define های اولیه قرار گرفته است که بعد از آن ها استفاده خواهیم کرد.

: Entering xv6

در این دسته قطعه کدهایی قرار گرفته است که در واقع شروع اجرای کرنل هستند و از بعد از اجرا شدن این دسته کدها، مدیریت و کنترل به سیستم عامل سپرده می شود. زیرا در این دسته، سیستم عامل در حافظه لود شده و حال می تواند کار خود را شروع کند.

: Lock

یکی از مزیت های سیستم های multiprocessor، کارایی بالای آن هاست، اما مشکلاتی هم در این سیستم ها وجود دارد. مکانیزم lock به ما کمک می کند یکی از این مشکلات را حل کنیم. زمانی که دو پردازنده همزمان از یک متغیر استفاده کنند، ممکن است جوابی که در نهایت در متغیر باقی می ماند اشتباه باشد. در این زمان مکانیزم coordination به ما در حل این مشکل کمک می کند. Lock یک object است که دارای state 2 می باشد. (Acquire(L و Release(L). در state اول تابع ذکر شده قفل L را lock می کند تا دیگر پردازنده ها نتوانند تا زمانی که کار پردازنده ی اول با آن متغیر تمام نشده است، از آن متغیر استفاده کنند.

در state دوم، تابع ذکر شده L را unlock می کند. این مکانیزم باعث می شود سرعت موازی کار کردن پردازنده ها پایین بیاید اما مطمئن خواهیم بود نتیجه درست است.

: Processes

در این قسمت اطلاعات مربوط به فرآیندها نوشته می شود. و ویژگی هر کدام مشخص می شود مانند اسم، id، پدر، state، context، فعال سازی و مدیریت فرآیندها و ...

: System calls

در این دسته سیستم کال ها نوشته می شوند. همچنین مکانیزم های interrupt و trap و ... در این قسمت هندل می شود. مدیریت فرآیندها هم در این بخش انجام می شود.

: File system

مدیریت کردن فایل ها و فراخوانی های سیستمی و مدیریت file descriptor ها و ... در این قسمت هندل می شود.

: Pipe

در این قسمت ساختار pipe و مدیریت کردن آن و خواندن و نوشتن در آن هندل می شود.

: String operation

در این قسمت در وهله ی اول یک سری ویژگی های رشته مانند اسم و طول و ... مشخص و تعیین می شوند. همچنین یک سری توابع تعریف می شوند که روی دسته ای از رشته ها اعمال می شوند.

: Low-level Hardware

IO و interrupt ها و را هندل می کند.

: Bootloader

در bootloader از bootstrap کدهای مربوط به بوت لودر را می خوانیم و آن را اجرا می کنیم. سپس کدهای مربوط به سیستم عامل را که در bootmain.c قرار دارند را در حافظه ذخیره می کنیم و در آخر کار که سیستم عامل به طور کامل لود شد، مدیریت و کنترل را به سیستم عامل می سپاریم.

کامپایل سیستم عامل xv6

4. در Makefile متغیر هایی به نام های UPROGS و ULIB تعریف شده است. کاربرد آنها

چیست؟

متغیر UPROGS مربوط به برنامه های کاربر هست، یعنی برنامه های کاربر توسط مجموعه ای از دستوراتی که UPROGS شامل آنها می شود، ساخته می شود.

پس طبق Makefile هر برنامه ی xv6 از دستورات / cat / echo / forktest / grep / init / kill / ln / ls / mkdir / rm / sh / stressfs / usertests / wc / zombie تشکیل شده است.

متغیر ULIB مربوط به فایل های کتابخانه کاربر هست، یعنی اینکه هر برنامه ی xv6 شامل چه

کتابخانه هایی می شود و طبق makefile کتابخانه هایمان شامل ulib.o usys.o printf.o umalloc.o هست.

اجرا بر روی شبیه ساز QEMU

5. دستور make qemu -n را اجرا نمایید. دو دیسک به عنوان ورودی به شبیه ساز داده شده است.

محتوای آنها چیست؟ (راهنمایی: این دیسک ها حاوی سه خروجی اصلی فرایند بیلد هستند.)

دیسک های ورودی شامل xv6.img و fs.img می باشد. دیسک xv6.img شامل بوت لودر xv6 می باشد که بایوس شبیه ساز qemu آن را بوسیله ی یک دیسک سخت مجازی بارگزاری می کند. فایل

fs.img شامل فایل های ابتدایی و اولیه شبیه ساز qemu می باشد. در واقع همان فایل سیستم هایی که ما در برنامه آنها را تغییر می دهیم و امتحان می کنیم.

مراحل بوت سیستم عامل xv6

8. بوت سیستم توسط فایل های bootasm.S و bootmain.c صورت می گیرد. چرا تنها از کد c استفاده نشده؟

پردازنده های x86، در real mood شروع به کار می کنند. BIOS سیستم کد boot selector را در حافظه لود می کند و سپس boot loader را از خانه به آدرس 0x7c00 در حافظه لود می کند و سپس به آن آدرس می رود و boot loader را اجرا می کند. در کد boot loader دو قطعه کد یکی به زبان اسمبلی و دیگری به زبان c نوشته شده است. قطعه کد اسمبلی برنامه را از real mode به protected mode میبرد. چون در real mode آدرس رجیسترهای ما 16 بیتی هستند و در protected mode این رجیسترها 32 بیتی هستند. در نتیجه می توان آدرس های بیشتری با آنها ساخت. سپس در کد به زبان c برنامه ی سیستم عامل را در حافظه ذخیره می کند تا بعد از آن کنترل و مدیریت سیستم را به سیستم عامل بدهد. وظیفه ی اول boot loader یا همان تغییر مود، در زبان اسمبلی نوشته شده است control-transfer و calling-convention در زبان c قابل اجرا نیست.

9. یک ثابت عام منظوره، یک ثابت قطعه، یک ثابت وضعیت و یک ثابت کنترلی در معماری xv6 نام برده و وظیفه ی هر یک را به طور مختصر توضیح دهید.

ثبات عام منظوره : این رجیستر در حافظه ی داده به طور موقت و برای دسترسی به حافظه استفاده می شود. مانند رجیستر EBX، معمولا در یک تابع به مقداری که زیاد استفاده می شود مقدار دهی می شود تا سرعت محاسبات را بالا ببرد.

ثبات قطعه : شامل CS، DS، SS، EX است. برای آدرس دهی در 20 real mode بیت داریم که غیر از رجیستر های 16 بیتی، 4 بیت اضافی وجود دارد که این ثبات های قطعه آن بیت های اضافه را مهیا می کنند و این رجیسترها در protected mode، یک ایندکس برای segment descriptor table هستند. برای مثال SS برای خواندن و نوشتن در استک است. ثبات وضعیت :

در این رجیسترها مقدارهایی نوشته می شود که وضعیت فعلی پردازنده که همان نتیجه ی عملیات قبل است را نشان می دهند.

به عنوان مثال، بیت شماره ی 0 : trap : 8 : zero - 6 : carry

ثبات کنترلی : این رجیسترها، شامل بیت هایی هستند که کنترل کننده یا تغییر دهنده ی رفتار کلی

پردازنده هستند. برای مثال CR0: تعیین مدها و وضعیت هایی مثل paging، cache disable، write-protect و ...

13. کد معادل entry.S را در هسته ی لینوکس بیابید.

Entry.s در سیستم عامل xv6 معادل header.S در سیستم عامل لینوکس است. کار این کدها این است که بعد از اینکه boot loader کارش را انجام داد و real mode را به protected mode تبدیل کرد و سپس سیستم عامل را در حافظه لود کرد، این تکه کد اجرا می شود. در واقع با این تکه کد وارد کرنل می شویم و کنترل و مدیریت را به دست سیستم عامل می سپاریم. کارهایی که در این کدها انجام می شود، به چند دسته تقسیم می شوند. در وهله ی اول کرنل شروع به اجرا شدن می کند. Multiboot header اجرا می شود. Multiboot header نوعی data structure در kernel image هستند که یک سری اطلاعات برای multiboot-compliant فراهم می کند. کی مثال برای شروع کار در نظر می گیرد سپس مکانیزم paging را اجرا می کند که همان پیچ پیچ کردن و تبدیل آدرس های مجازی به فیزیکی و است. سپس اشاره گرهایی به استک را مقدار دهی می کند و در مرحله ی آخر سیستم کال main را فراخوانی می کند.

اجرای هسته ی xv6

14. چرا این آدرس فیزیکی است؟

جدولی در هر سیستم وجود دارد که در آن آدرس های مجازی به آدرس های فیزیکی تبدیل می شوند. یک رجیستر به نام cr3 وجود دارد که در آن آدرس فیزیکی آن جدول نوشته شده است. حال برای آنکه این آدرس به آدرس مجازی تبدیل شود، برای ترجمه کردن این آدرس، در سخت افزار paging ، نیاز به آن جدول داریم. اما ما آن page table را نداریم. در نتیجه در یک loop می افتم.

16. مختصری راجع به فضای آدرس مجازی هسته توضیح دهید.

آدرس مجازی یک عدد باینری در حافظه ی مجازی است که به وسیله ی آن process می تواند مستقل از سایر process ها از حافظه ی اصلی استفاده کند. از جمله مزیت های این نوع آدرس دهی این است که process می تواند از فضای بیشتری نسبت به آن چه در ذخیره سازی اولیه وجود دارد، استفاده کند به این صورت که به طور موقت برخی از محتویات را به یک هارد دیسک یا درایو فلش داخلی منتقل می کند.

اجرای نخستین برنامه ی سطح کاربر

22. کدام بخش از آماده سازی سیستم، بین تمامی هسته های پردازنده مشترک و کدام بخش اختصاصی است؟ زمان بند روی کدام هسته اجرا می شود؟

محتویات اولیه ی حافظه ی `process` اول، فرم کامپایل شده ی فایل `initcode.S` می باشد. با توجه به آنکه قسمتی از کرنل وظیفه ی ساخت `process` جدید را بر عهده دارد، `linker` یا وصل کننده کد باینری این عملیات را داخل کرنل قرار می دهد و دو سمبل به خصوص تعریف می کند. این سمبل ها عبارت اند از `binary_initcode_size_` و `binary_initcode_start_` که نشان دهنده ی سایز و محل قرار گیری کد باینری می باشد. `Userinit` این کد باینری را به وسیله ی صدا زدن `inituvm` در حافظه ی یک `process` جدید ذخیره می کند که به این `process` جدید یک صفحه یا `page` از آدرس فیزیکی اختصاص داده می شود و `inituvm` آدرس مجازی صفر را به آن حافظه اختصاص می دهد و کد باینری را در آن صفحه از حافظه کپی می کند.

درباره ی سوال قسمت دوم از آنجایی که معماری `xv6` اینتل و پراسس های `symmetric` روی همه ی پروسسور ها به برنامه زمانبند (`scheduler`) مثل لینوکس نصب می باشد، که مانند یک درخت رد بلک هست که هر تسکی که باید به عنوان تسک بعدی انتخاب بشود تو چپ ترین نود قرار دارد.

23. برنامه ی معادل `initcode.S` در هسته ی لینوکس چیست؟

`init.s`