اعضا : سید احمد رکنی حسینی ۸۱۰۱۰۰۱۵۴ محمدعلی شاهین فر ۸۱۰۱۰۰۱۶۹ امیرحسین راحتی ۸۱۰۱۰۰۱۴۴

سوال اول : سیستم عامل **xv6** یک سیستم عامل آموزشی است که به شدت از سیستم عامل **Unix** الهام گرفته شده است.

این سیستم عامل به صورت ترکیبی از زبان **c** و اسمبلی نوشته شده است

معماری **xv6** از یک طرح لایه ای پیروی میکند که اجزای مختلفی مسئول عملکردهای خاص هستند.

۱. بوت لودر: سیستم با اجرای بوت لودر شروع می شود که سخت افزار را مقداردهی اولیه کرده و هسته **xv6**را در حافظه بارگذاری می کند

۲.کرنل : هسته **xv6** وظیفه مدیریت منابع سیستم و ارائه انتزاع برای برنامه های کاربر را بر عهده دارد. این عملکردهای اصلی سیستم عامل مانند **process management** ، **memory management** ، فایل های سیستمی (**system files**) و درایور های سخت افزاری دستگاه را پیاده سازی می کند.

Process management: کرنل وظیفه ایجاد و هندل کردن **process** ها را به عهده دارد و بوسیله یک **scheduler** ساده آنها را مدیریت میکند

Memory management: کرنل از حافظه مجازی پشتیبانی میکند که به کمک **page table** ها حافظه های مجازی را به حافظه‌های فیزیکی متصل میکند

File systems: از یک مدل ساده استفاده میکند که می‌تواند از روی فایل بخواند ، فایل ایجاد کند و روی فایل بنویسد

Device drivers: وظیفه هندل کردن و کنترل سخت افزار ها مثل کیبرد را دارد

system calls.3: در این سیستم عامل مجموعه ای از **system call** ها وجود دارد که به برنامه مورد نظر امکان ارتباط با کرنل را میدهد مانند ایجاد یک **process**، خواندن از فایل و ...

user programs.4: بوسیله **system call** ها با کرنل در ارتباط هستند و در سطح **user** اجرا میشوند.

سوال دوم :

۱. **Process control block**: یک ساختمان داده است که اطلاعاتی در مورد هر **process** را در خود نگه میدارد مثل **proc ID** و ...

۲.**Process switching** : کرنل باید استیت هر **process** را مدیریت کند.یعنی در لحظه مورد نظر یک **process** را متوقف کند و اطلاعات آن را ذخیره کند و **process** بعدی را لود و ادامه دهد.

۳.**Time scheduling**: وظیفه دارد ترتیب انجام **process** ها را مشخص کند. در این سیستم عامل از الگوریتم **round-robin** استفاده میشود.

در این الگوریتم مدت زمانی که به هر **process** اختصاص داده میشود هم محاسبه میشود.

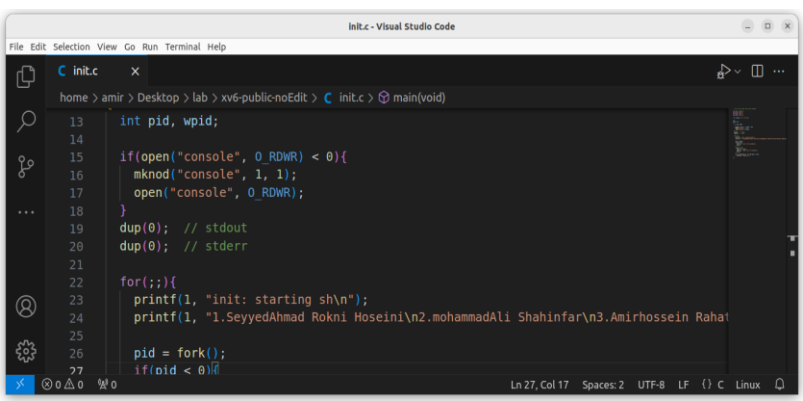
۴.**Process terminating** :زمانی که یک **process** تمام میشود با ارسال درخواستی به کرنل خواهان **terminate** آن میشویم که در اینجا کرنل آن را از بین میبرد و**PCB** مربوط به آن را ریست میکند و ...

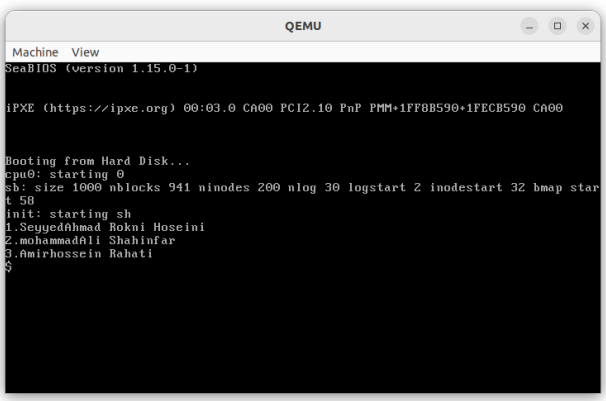
سوال چهارم :

fork(): create a process, copies the parent`s file descriptor table along with its memory, so that the child starts with exactly the same open file as a parent
exec(): load a file and execute it if they are separate, the shell can fork a child, use open, close, dup in the child to change the standard input and output file descriptor, and then exec, if they were combined into a single system call, some other scheme would be required for the shell to redirect standard input and output, or the program itself would have to understand how to redirect I/O

اضافه کردن متن به boot message:

در این قسمت با اضافه کردن یک تابع **printf** در فایل **init.c** میتوان پیام مورد نظر را به **boot message** اضافه کرد





اضافه کردن چند دستور:

برای اضافه کر دن دستورات **arrow up , arrow down** یک آرایه دوبعدی تعریف میکنیم که دو پوینتر به خانه های آن اشاره میکنند که یکی وضیفه نوشتن و یکی وظیفه خواندن از آن را دارد.با هربار زدن کلید اینتر روی این آرایه مینویسیم و با زدن کلید های بالا و پایین از این ارایه روی بافر اصلی دستورات نوشته میشود:

```
console.c M ×
console.c > consoleintr(int(* )(void))
236     case KEY_UP:
237       {
238         if ((hist.e <= HISTORY_SIZE  && hist.r == 0)
239         || ((hist.e % HISTORY_SIZE) == (hist.r-1) % HISTORY_SIZE))
240           break;
241
242         if (hist.r > 0)
243           hist.r--;
244
245         while(input.e != input.w && input.buf[(input.e-1) % INPUT_BUF] != '\n')
246         {
247           input.e--;
248           consputc(BACKSPACE);
249         }
250         for(int i = 0; i< INPUT_BUF ; i++)
251         {
252           char character = hist.buf[hist.r % HISTORY_SIZE][i];
253           if ((character == NULL) || (character == '\n'))
254             break;
255           input.buf[input.e % INPUT_BUF] = character;
256           input.e++;
257           consputc(character);
258         }
259       }
260      break;
```

```
console.c M ×
console.c > consoleintr(int(* )(void))
261     case KEY_DN:
262       {
263         if((hist.r % HISTORY_SIZE) == (hist.e % HISTORY_SIZE))        You, 3 minut
264           break;
265         else
266           hist.r++;
267
268         while(input.e != input.w && input.buf[(input.e-1) % INPUT_BUF] != '\n')
269         {
270           input.e--;
271           consputc(BACKSPACE);
272         }
273         for(int i = 0; i< INPUT_BUF ; i++)
274         {
275           char character = hist.buf[hist.r % HISTORY_SIZE][i];
276           if ((character == NULL) || (character == '\n'))
277             break;
278           input.buf[input.e % INPUT_BUF] = character;
279           input.e++;
280           consputc(character);
281         }
282       }
283      break;
```

```
console.c M ×
console.c > ...
199   struct History{
200     char buf[HISTORY_SIZE][INPUT_BUF];
201     uint r;  // Read index
202     uint e;  // Edit index
203   };
204
205   struct History hist = {.r = 0 , .e = 0};
```

کلاس **history** یک بافر دوبعدی و دو پوینتر دارد

```
consoleintr(int(* )(void))
int flag = 0;
if (input.e - 1 == input.w && (input.buf[input.e] == '\n' || input.buf[input.w] == '\n'))
{
  flag = 1;
  input.w = input.e;
}
if (flag == 0)
{
  for (int i = 0 ; i < INPUT_BUF ; i++)
  {
    hist.buf[(hist.e+1) % HISTORY_SIZE][i] = NULL;
    if (input.w != input.e)
    {
      hist.buf[hist.e % HISTORY_SIZE][i] = input.buf[input.w % INPUT_BUF];
      input.w++;
    }
    else
      hist.buf[hist.e % HISTORY_SIZE][i] = NULL;
  }
  hist.e++;
  hist.r = hist.e;
}
```

اجرای **arrow up , arrow down**:

ابتدا دستور **ls** ، بعد **echo** استفاده شده نهایتا با دو **arrow up** به دستور **ls**دسترسی داریم

```
QEMU
Machine  View
init: starting sh
$ ls
.              1 1 512
..             1 1 512
README         2 2 2286
cat            2 3 15500
echo           2 4 14380
forktest       2 5 8824
grep           2 6 18344
init           2 7 15000
kill           2 8 14468
ln             2 9 14364
ls             2 10 16932
mkdir          2 11 14488
rm             2 12 14468
sh             2 13 28524
stressfs       2 14 15400
usertests      2 15 62900
wc             2 16 15924
zombie         2 17 14048
console        3 18 0
$ echo hello
hello
$ ls
```

```
QEMU
Machine  View
console        3 18 0
$ echo hello
hello
$ ls
.              1 1 512
..             1 1 512
README         2 2 2286
cat            2 3 15500
echo           2 4 14380
forktest       2 5 8824
grep           2 6 18344
init           2 7 15000
kill           2 8 14468
ln             2 9 14364
ls             2 10 16932
mkdir          2 11 14488
rm             2 12 14468
sh             2 13 28524
stressfs       2 14 15400
usertests      2 15 62900
wc             2 16 15924
zombie         2 17 14048
console        3 18 0
$ echo hello
```

```
case C('L'):
  {
    for (int i = 0 ; i<1000 + count * 100;i++)
    {
      consputc(BACKSPACE);
      if (count > 0);
        count--;
    }
    consputc('$');
    consputc(' ');
  }
  break;
```

```
Machine  View
$ _
```

برای دستور**ctrl+L** تعداد خطوط چاپ شده را در متغیری گلوبال به نام **count** ذخیره میکنیم و با هربار **\n** ان را یکی زیاد میکنیم نهایتا با گرفتن **ctrl+L** به تعداد خطوط وکاراکتر های حدودی تخمین زده شده در صفحه آن را پاک میکنیم

برای دستورا **ctrl + B , ctrl + F** با تغییر مقدار متغیر **pos** و برگرداندن آن به حالت اول بار گرفتن اینتر یا **arrow up** و **down** میتوان عملیات مورد نظر را پیاده سازی کرد

افزودن برنامه در سطح کاربر:

برای این کار ، تغییراتی در **makefile** باید داده شود که دستور به عنوان یک دستور سطح کاربر شناخته و کامپایل شود.سپس کد مورد نظر را در یک فایل جداگانه قرار میدهیم و مسیر آن را به **makefile** میدهیم.

در این حالت مانند بقیه دستورات فایل های اسمبلی و ترجمه شده دستور جدید حین کامپایل ایجاد میشود. فایل **strdiff.c** :

```c
#define MAX_LENGTH 32
#define NULL '\0'

void compareStrings(const char *str1,
    const char *str2, char *result)
{
    int length = strlen(str1);
    if (length < strlen(str2))
        length = strlen(str2);
    for (int i = 0; i < length; i++)
        if (str1[i] <= str2[i])
            result[i] = '1';
        else
            result[i] = '0';
}

int main(int argc, char *argv[])
{
    if (argc < 3)
        exit();
    const char *str1 = argv[1] , *str2 = argv[2];
    char result[MAX_LENGTH];
    int resault_size = 0;
    compareStrings(str1, str2, result);
    for(int i = 0 ; i < MAX_LENGTH ; i++)
        if (result[i] == NULL)
        {
            resault_size = i;
            result[i] = '\n';
            break;
        }

    int fd = open("strdiff_result.txt", O_WRONLY);
    if (fd < 0)
    {
        printf(1 , "can't open file...\n");
        exit();
    }

    if (write(fd, result,
    (sizeof(char) * (resault_size+1))) < (sizeof(char) * resault_size))
    {
        printf(1 , "write error...\n");
        exit();
    }
    close(fd);
    exit();
}
```

```
QEMU
Machine  View
SeaBIOS (version 1.15.0-1)

iPXE (https://ipxe.org) 00:03.0 CA00 PCI2.


Booting from Hard Disk...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog
t 58
init: starting sh
$ strdiff amir reza
$ cat strdiff_result.txt
1010
$
```

سوال هشتم :

متغیر **UPROGS** همان **user programs** است که قرار است فایل های کد مربوط به آن ها کامپایل شوند و در **image** نهایی **xv6** قرار گیرد.

این متغیر لیستی از این برنامه های سطح کاربر را در خود دارد.

متغیر**ULIBS** هم **user libraries** است که **library** های کاربر را در خود نگه میدارد که در زمان کامپایل همراه با بقیه فایل های کاربر کامپایل و ایجاد شوند

سوال یازدهم :

آدرس های موجود در فایل های باینری کد ها مجازی هستند و باید به حافظه فیزیکی **map** شوند.

اما حافظه های موجود در فایل نهایی حافظه های واقعی هستند چون برای بوت باید حافظه های فیزیکی را تغییر دهیم

همچنین فایل بوت نهایی تماما باینری است

فایل بوت نهایی باید در کمترین حجم ممکن باشد تا بوت بدون مشکل و سریع داشته باشیم

نتیجه اجرای **objdump** : این دستور کد ماشین را به اسمبلی تبدیل میکند

reault.txt

```
bootblock.o:     file format elf32-i386


Disassembly of section .text:

00007c00 <start>:
    7c00:   fa                      cli
    7c01:   31 c0                   xor    %eax,%eax
    7c03:   8e d8                   mov    %eax,%ds
    7c05:   8e c0                   mov    %eax,%es
    7c07:   8e d0                   mov    %eax,%ss

00007c09 <seta20.1>:
    7c09:   e4 64                   in     $0x64,%al
    7c0b:   a8 02                   test   $0x2,%al
    7c0d:   75 fa                   jne    7c09 <seta20.1>
    7c0f:   b0 d1                   mov    $0xd1,%al
    7c11:   e6 64                   out    %al,$0x64
```

```
00007c13 <seta20.2>:
    7c13:   e4 64                   in     $0x64,%al
    7c15:   a8 02                   test   $0x2,%al
    7c17:   75 fa                   jne    7c13 <seta20.2>
    7c19:   b0 df                   mov    $0xdf,%al
    7c1b:   e6 60                   out    %al,$0x60
    7c1d:   0f 01 16                lgdtl  (%esi)
    7c20:   78 7c                   js     7c9e <readsect+0xe>
    7c22:   0f 20 c0                mov    %cr0,%eax
    7c25:   66 83 c8 01             or     $0x1,%ax
    7c29:   0f 22 c0                mov    %eax,%cr0
    7c2c:   ea                      .byte 0xea
    7c2d:   31 7c 08 00             xor    %edi,0x0(%eax,%ecx,1)

00007c31 <start32>:
    7c31:   66 b8 10 00             mov    $0x10,%ax
    7c35:   8e d8                   mov    %eax,%ds
    7c37:   8e c0                   mov    %eax,%es
    7c39:   8e d0                   mov    %eax,%ss
    7c3b:   66 b8 00 00             mov    $0x0,%ax
```

```
    7c3f:   8e e0                   mov    %eax,%fs
    7c41:   8e e8                   mov    %eax,%gs
    7c43:   bc 00 7c 00 00          mov    $0x7c00,%esp
    7c48:   e8 fc 00 00 00          call   7d49 <bootmain>
    7c4d:   66 b8 00 8a             mov    $0x8a00,%ax
    7c51:   66 89 c2                mov    %ax,%dx
    7c54:   66 ef                   out    %ax,(%dx)
    7c56:   66 b8 e0 8a             mov    $0x8ae0,%ax
    7c5a:   66 ef                   out    %ax,(%dx)

00007c5c <spin>:
    7c5c:   eb fe                   jmp    7c5c <spin>
    7c5e:   66 90                   xchg   %ax,%ax

00007c60 <gdt>:
    ...
    7c68:   ff                      (bad)
    7c69:   ff 00                   incl   (%eax)
    7c6b:   00 00                   add    %al,(%eax)
    7c6d:   9a cf 00 ff ff 00 00    lcall  $0x0,$0xffff00cf
```

```
    7c74:   00                      .byte 0x0
    7c75:   92                      xchg   %eax,%edx
    7c76:   cf                      iret
    ...

00007c78 <gdtdesc>:
    7c78:   17                      pop    %ss
    7c79:   00 60 7c                add    %ah,0x7c(%eax)
    ...

00007c7e <waitdisk>:
    7c7e:   f3 0f 1e fb             endbr32
    7c82:   ba f7 01 00 00          mov    $0x1f7,%edx
    7c87:   ec                      in     (%dx),%al
    7c88:   83 e0 c0                and    $0xffffffc0,%eax
    7c8b:   3c 40                   cmp    $0x40,%al
    7c8d:   75 f8                   jne    7c87 <waitdisk+0x9>
    7c8f:   c3                      ret

00007c90 <readsect>:
```

```
    7c90:   f3 0f 1e fb             endbr32
    7c94:   55                      push   %ebp
    7c95:   89 e5                   mov    %esp,%ebp
    7c97:   57                      push   %edi
    7c98:   53                      push   %ebx
    7c99:   8b 5d 0c                mov    0xc(%ebp),%ebx
    7c9c:   e8 dd ff ff ff          call   7c7e <waitdisk>
    7ca1:   b8 01 00 00 00          mov    $0x1,%eax
    7ca6:   ba f2 01 00 00          mov    $0x1f2,%edx
    7cab:   ee                      out    %al,(%dx)
    7cac:   ba f3 01 00 00          mov    $0x1f3,%edx
    7cb1:   89 d8                   mov    %ebx,%eax
    7cb3:   ee                      out    %al,(%dx)
    7cb4:   89 d8                   mov    %ebx,%eax
    7cb6:   c1 e8 08                shr    $0x8,%eax
    7cb9:   ba f4 01 00 00          mov    $0x1f4,%edx
    7cbe:   ee                      out    %al,(%dx)
    7cbf:   89 d8                   mov    %ebx,%eax
    7cc1:   c1 e8 10                shr    $0x10,%eax
    7cc4:   ba f5 01 00 00          mov    $0x1f5,%edx
```

```
    7cc9:   ee                      out    %al,(%dx)
    7cca:   89 d8                   mov    %ebx,%eax
    7ccc:   c1 e8 18                shr    $0x18,%eax
    7ccf:   83 c8 e0                or     $0xffffffe0,%eax
    7cd2:   ba f6 01 00 00          mov    $0x1f6,%edx
    7cd7:   ee                      out    %al,(%dx)
    7cd8:   b8 20 00 00 00          mov    $0x20,%eax
    7cdd:   ba f7 01 00 00          mov    $0x1f7,%edx
    7ce2:   ee                      out    %al,(%dx)
    7ce3:   e8 96 ff ff ff          call   7c7e <waitdisk>
    7ce8:   8b 7d 08                mov    0x8(%ebp),%edi
    7ceb:   b9 80 00 00 00          mov    $0x80,%ecx
    7cf0:   ba f0 01 00 00          mov    $0x1f0,%edx
    7cf5:   fc                      cld
    7cf6:   f3 6d                   rep insl (%dx),%es:(%edi)
    7cf8:   5b                      pop    %ebx
    7cf9:   5f                      pop    %edi
    7cfa:   5d                      pop    %ebp
    7cfb:   c3                      ret
```

سوال دوازدهم :

objcopy used to generate a raw binary file. when objcopy generates a raw binary file, it will essentially produce a memory dump pf the contests of the input object file. all symbols and relocation information will be discarded. the memory dump will start at the load address of the lowest section copied into the output file., the result data file is a stripped version of initcode.out . it is a memory dump of the contents, the hex dump is the assembly instruction in little endian format make binary files bootable

سوال چهاردهم :

۱.General purpose registers: مانند همه سیستم عامل های دیگر تعدادی رجیستر همه کاره وجود دارد که وظیفه نگه داری حاصل عملیات های ریاضی و ... دارد.این رجیستر ها همچنین میتوانند آدرس های مموری را هم در خود نگه دارند.این رجیستر ها برای جابجایی دیتا بین پردازنده و سیستم عامل استفاده میشوند.

۲.Segment registers: در معماری x86 برای مدیریت حافظه و دسترسی به حافظه استفاده میشوند . در سیستم عامل xv6 کمتر از آن ها استفاده میشود ولی برای ذخیره آدرس های حافظه کاربرد دارند

۳.Control registers:رجیستر های خاص منظوره برای ذخیره flag های پردازنده است برای مثال فعال کردن memory protection

۴.Status registers: برای ذخیره حالت های پردازنده و برنامه ها بعد انجام یک instruction استفاده میشود برای مثال بررسیoverflowدر محاسبات

سوال هجدهم:

the kernel has different entry points for different bootloaders on some architectures, for example on x86, the entry point is in arch/x86/boot/header.S

سوال نوزدهم:

because the paging hardware doesn't know how to translate virtual addresses yet, it doesn't have a page table yet

سوال بیست ودوم :

دلیل اصلی استفاده از این حالت این است که بعضی دستورات با سطح دسترسی کرنل و بعضی با سطح دسترسی کاربر انجام میشود و قطعی دسترسی این دو سطح تفاوت زیادی با هم دارند.برای چک کردن این موضوع از این flag استفاده میشود تا مثلا دستور در سطح کاربر نتواند به اطلاعاتی که به آن دسترسی ندارد آسیب بزند.

سوال بیست وسوم :

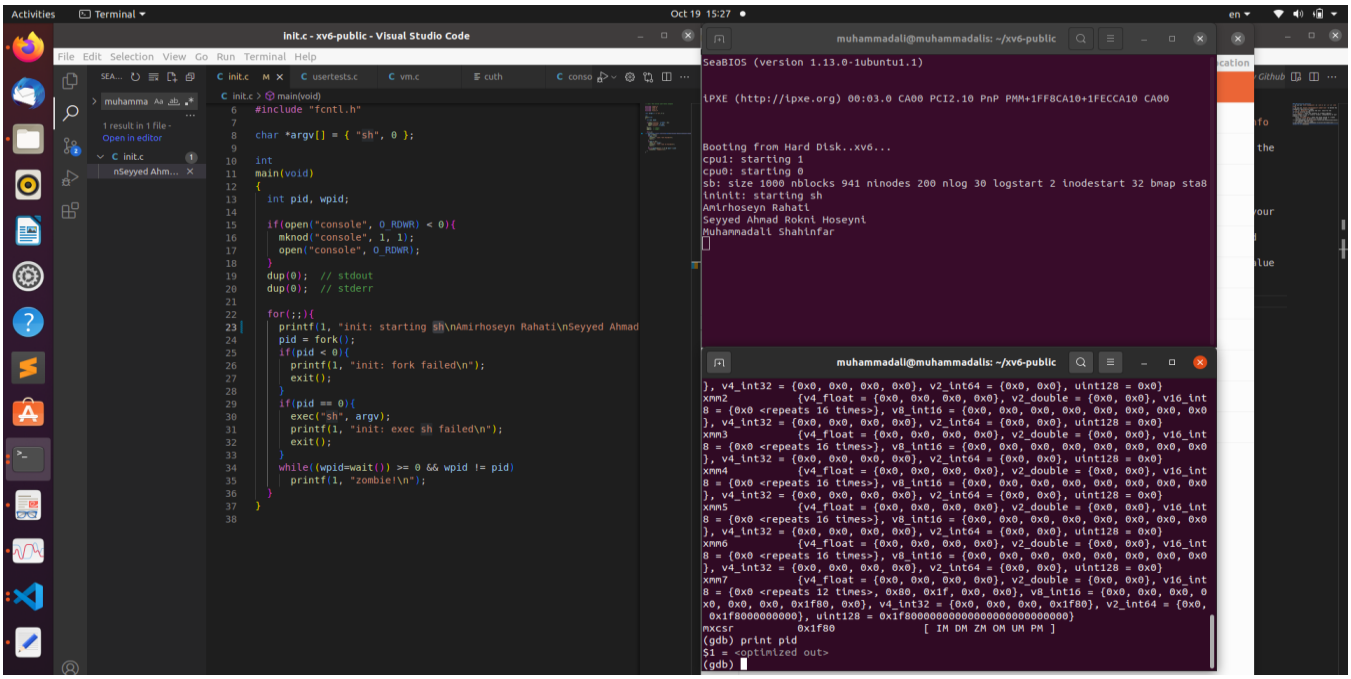
p-> trapframe: holds saved user threads registers
p-> context: holds saved kernel threads registers
p-> kstack: points to the threads kernel stack
p-> state: running, runnable, sleeping
p-> lock: protects state, and other things in Linux, we have /proc/$PID

سوال بیست وهفتم :

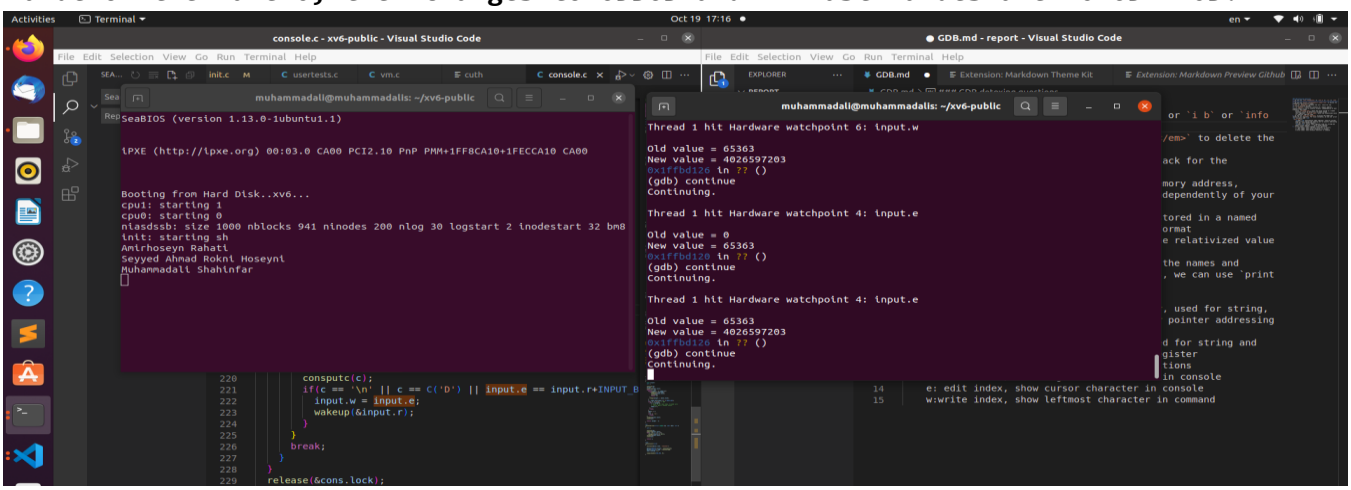
scheduler threads are one per core, each has stack + context, separate scheduler stack makes it easier to handle exit() and gets off kernel stack, allowing another core to run the last thread in parallel. boot and bootloader are same between all cores. after loading OS in disk, bootloader leave the process to cores. scheduling and executing user-level program are done in Cores

Debugging with GDB:

1. we can use info breakpoints or info b or i b or info break
2. we can use delete breakpoint number to delete the breakpoint specified by number
3. bt or backtrace Displays the call stack for the currently selected thread
4.x address: shows the contents of a memory address, examine memory in any of several formats, independently of your programs data types print variable name: shows the value stored in a named variable, print by default as variable format info registers register name: print the relativized value of register name
5.we can use info all-registers to print the names and values of all registers. for local variables, we can use print variable name



EDI: destination index register, used for string, memory array copying and setting and far pointer addressing with ES, called saved register
ESI: source index register used for string and memory array copying , called saved register
6.struct input: contain console cursor positions
    r: read index, show rightmost character in console
    e: edit index, show cursor character in console
    w:write index, show leftmost character in command
by adding watchpoint, we saw that variable changes when console run and initial value of them are 0, then changes to 65363 and in last values are 4026597203.

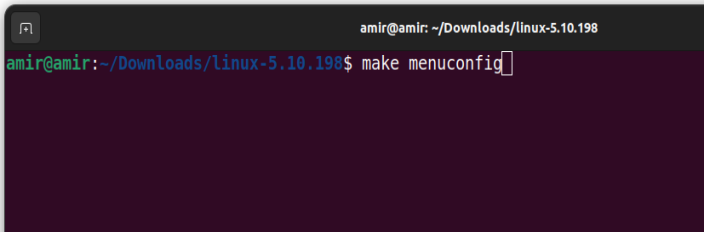


7. layout src show stop point location on user level code, layout asm show stop point in assembly level
8. by using up and down command, we can swap between functions in stack trace.

Linux kernel

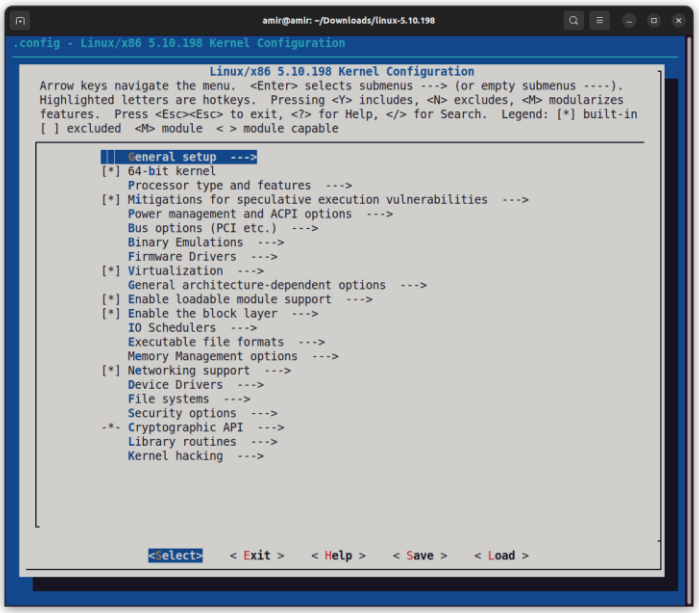
در این بخش ورژن ۵.۱۰ و ۶.۵ کرنل لینوکس مورد استفاده قرار میگیرد.

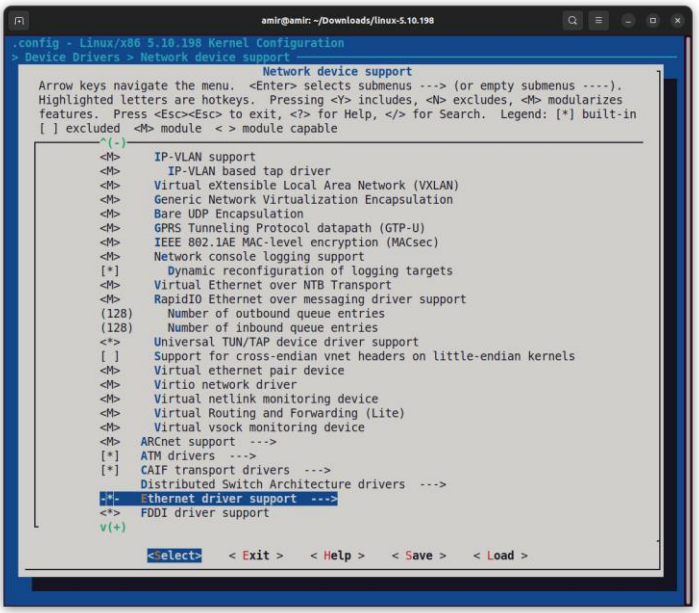
با زدن دستور <make menuconfig> فایل کانفیگ پیش فرض ایجاد میشود



سپس یک رابط کاربری متنی باز میشود که میتوان تنظیمات پیشفرض را مشاهده کرد.

سمت راست صفحه اصلی تنظیمات و سمت چپ تنظیمات پیشفرض مربوط به شبکه و اینترنت است





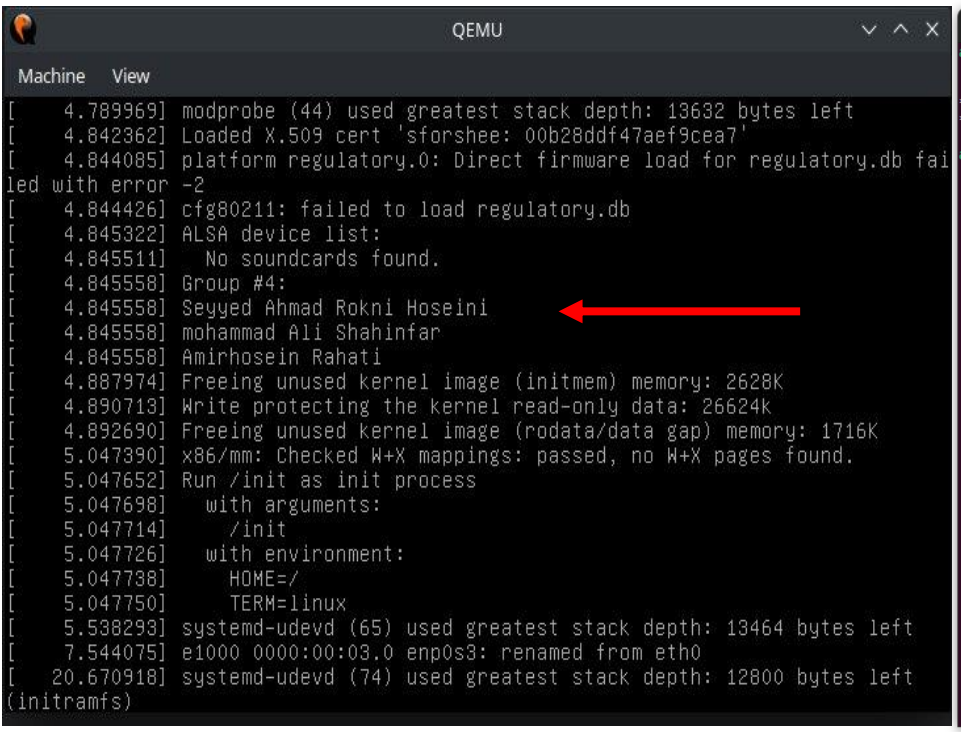
با اضافه کردن یک دستور **printk** در فایل **main.c** در **arch/x86/boot** میتوان رشته مورد نظر را اضافه کرد. (تابع **init_do_initcalls**)

نهایتا با زدن دستور **make –j8** کرنل شروع به کامپایل شدن میکنیم سپس به وسیله دستور زیر فایل ایمیج نهایی را میسازیم .

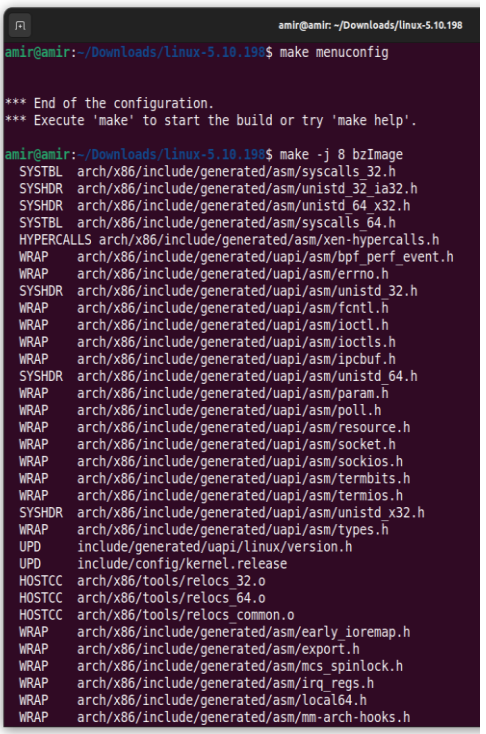
```
mkinitramfs -o initrd.img-6.5.7
```

آن را به **qemu** میدهیم.نهایتا با زدن دستور **dmesg** میتوان نام اعضا را مشاهده کرد.

```
qemu-system-x86_64 -kernel bzImage -initrd initrd.img-6.5.7 -m 1024
```



خروجی بعد دستور dmesg



روند کامپایل