Operating Systems
Lab Report 1A
Group 4

Group Members:
- Tarun Raj 220101104
- Ayush Savar 220101022
- Tanush Reddy Kolagatla 220101101
- Udbhav Gupta 220101106

Link to Patch and Code Files:
https://drive.google.com/drive/folders/1cJlkh522j971a7cWXdHqa3o66cnioJPN?usp=sharing

Part1
PC Bootstrap

- ## Exercise 1:

Complete code:

```
Welcome    ex1.c  U
ex1.c > main(int, char **)
#include <stdio.h>

int main(int argc, char **argv)
{
    int x = 1;
    printf("Hello x = %d\n", x);

    // In-line assembly to increment the value of x by 1
    __asm__(
        "addl $1, %0" // Assembly instruction to add 1 to the value of x
        : "=r"(x)     // Output operand: x is being modified
        : "0"(x)      // Input operand: initial value of x
    );
    // l in addl represents long

    printf("Hello x = %d after increment\n", x);

    if (x == 2)
    {
        printf("OK\n");
    }
    else
    {
        printf("ERROR\n");
    }

    return 0;
```

Output:

```
udbhav@Udbhav514:~/xv6-public$ cd "/home/udbhav/xv6-public/" && gcc ex1.c -o ex1 && "/home/udbhav/xv6-public/"ex1
Hello x = 1
Hello x = 2 after increment
OK
```

Explanation:

1) The task was to add inline assembly code to a provided C program which increments the value of given int variable x by 1.

The aim was achieved using the following lines of code:

```
__asm__(
        "addl $1, %0"
        : "=r(x)
        : "0"(x)
);
```

2) The added lines are explained as follows:

Instruction: "addl $1, %0"

- addl $1, %0: Adds the immediate value 1 to the operand referenced by %0.
- %0 is a placeholder that will be replaced with a register holding the value of x.

Output Operand: : "=r"(x)

- "=r": This indicates that the output will be stored in a register (r), and the result is assigned back to x.
- The "=r" constraint tells the compiler that x will be placed in a register, and this register will hold the result after the addl instruction.

Input Operand: : "0"(x)

- "0": This means that the input operand should be the same as the output operand (%0). It tells the compiler to use the same register for the input and output, effectively modifying x in place.

- ## Exercise 2

The task asks us to trace a few of the initial bootstrap instructions using si instruction in GDB.

Initial Instructions:

```
[f000:fff0]    0xffff0: ljmp   $0x3630,$0xf000e05b
0x0000fff0 in ?? ()
+ symbol-file kernel
warning: A handler for the OS ABI "GNU/Linux" is not built into this configuration
of GDB.  Attempting to continue with the default i8086 settings.

(gdb) si
[f000:e05b]    0xfe05b: cmpw   $0xffc8,%cs:(%esi)
0x0000e05b in ?? ()
(gdb) si
[f000:e062]    0xfe062: jne    0xd241d0b0
0x0000e062 in ?? ()
(gdb) si
[f000:e066]    0xfe066: xor    %edx,%edx
0x0000e066 in ?? ()
(gdb) si
[f000:e068]    0xfe068: mov    %edx,%ss
0x0000e068 in ?? ()
(gdb) si
[f000:e06a]    0xfe06a: mov    $0x7000,%sp
0x0000e06a in ?? ()
(gdb) si
[f000:e070]    0xfe070: mov    $0xfc1c,%dx
0x0000e070 in ?? ()
(gdb) si
[f000:e076]    0xfe076: jmp    0x5576cf2d
```

Explanation:

The si command is used to run through the code one line at a time while executing and displaying it on the terminal. The first instruction of the xv6 bootstrap process is:

`[f000:fff0] 0xffff0: ljmp $0x3630,$0xf000e05b`

- `ljmp` (Long Jump) is used to jump to a specific memory address.
- `$0x3630` is the segment selector.
- `$0xf000e05b` is the offset within that segment.
- `f000:fff0` is a segmented address (segment format), and `[f000:fff0]` indicates the contents of that memory address.

The instructions we explored:

1. `cmpw $0xffc8,%cs:(%esi):` Compare the word at %cs:(%esi) with 0xffc8.
2. `jne 0xd241d0b0:` Jump to 0xd241d0b0 if the comparison is not equal.
3. `xor %edx,%edx:` Clear edx (set to 0).
4. `mov %edx,%ss:` Set the stack segment (ss) to 0.
5. `mov $0x7000,%sp:` Set the stack pointer (sp) to 0x7000.
6. `mov $0xfc1c,%dx:` Load dx with 0xfc1c.
7. `jmp $0x5576cf2d:` Jump to address 0x5576cf2d.

Part2
PC Bootstrap

- ## Exercise 3

Identifying Loop:

```
for(; ph < eph; ph++){
  pa = (uchar*)ph->paddr;
  readseg(pa, ph->filesz, ph->off);
  if(ph->memsz > ph->filesz)
    stosb(pa + ph->filesz, 0, ph->memsz - ph->filesz);
}
```

In the above code,
Loop instruction starts at:

```
7d7d: 39 f3                    cmp     %esi, %ebx
```

And ends at:

```
7d94: 76 eb                    jbe     7d81 <bootmain+0x44>
```

Upon entering the for loop, the first operation performed is a comparison between the values of ph and eph. The loop will only continue executing as long as ph is less than eph. The final instruction in the loop occurs when ph and eph are equal, signalling the end of the loop. At this point, the loop execution terminates, and control is transferred to the next instruction at address `0x7d91`. Consequently, the jump instruction serves as the concluding operation of the for loop.

After Loop Termination:
The subsequent instruction after the loop is:

```
7d81: ff 15 18 00 01 00           call    *0x10018
```

Marking a breakpoint at this address 0x7d81 and then executing further instructions resulted in this:

```
(gdb) b *0x7d81
Breakpoint 1 at 0x7d81
(gdb) c
Continuing.
The target architecture is set to "i386".
=> 0x7d81:      call   *0x10018

Thread 1 hit Breakpoint 1, 0x00007d81 in ?? ()
(gdb) si
=> 0x10000c:    mov    %cr4,%eax
0x0010000c in ?? ()
(gdb) si
=> 0x10000f:    or     $0x10,%eax
0x0010000f in ?? ()
(gdb) si
=> 0x100012:    mov    %eax,%cr4
0x00100012 in ?? ()
(gdb) si
=> 0x100015:    mov    $0x10a000,%eax
0x00100015 in ?? ()
(gdb) si
=> 0x10001a:    mov    %eax,%cr3
0x0010001a in ?? ()
(gdb) si
=> 0x10001d:    mov    %cr0,%eax
0x0010001d in ?? ()
(gdb) si
=> 0x100020:    or     $0x80010000,%eax
0x00100020 in ?? ()
(gdb) si
=> 0x100025:    mov    %eax,%cr0
0x00100025 in ?? ()
(gdb) si
=> 0x100028:    mov    $0x801164d0,%esp
0x00100028 in ?? ()
(gdb) si
=> 0x10002d:    mov    $0x80103060,%eax
0x0010002d in ?? ()
(gdb) si
=> 0x100032:    jmp    *%eax
0x00100032 in ?? ()
(gdb) si
=> 0x80103060 <main>:   lea    0x4(%esp),%ecx
main () at main.c:20
20        kinit1(end, P2V(4*1024*1024)); // phys page allocator
(gdb) si
=> 0x80103064 <main+4>: and    $0xfffffff0,%esp
0x80103064      20        kinit1(end, P2V(4*1024*1024)); // phys page allocator
(gdb) si
=> 0x80103067 <main+7>: push   -0x4(%ecx)
0x80103067      20        kinit1(end, P2V(4*1024*1024)); // phys page allocator
(gdb) 
```

Upon examining the code in `bootasm.S`,

```
  # Switch from real to protected mode.  Use a bootstrap GDT that makes
  # virtual addresses map directly to physical addresses so that the
  # effective memory map doesn't change during the transition.
  lgdt    gdtdesc
  movl    %cr0, %eax
  orl     $CR0_PE, %eax
  movl    %eax, %cr0

//PAGEBREAK!
  # Complete the transition to 32-bit protected mode by using a long jmp
  # to reload %cs and %eip.  The segment descriptors are set up with no
  # translation, so that the mapping is still the identity mapping.
  ljmp    $(SEG_KCODE<<3), $start32

.code32  # Tell assembler to generate 32-bit code now.
start32:
  # Set up the protected-mode data segment registers
  movw    $(SEG_KDATA<<3), %ax    # Our data segment selector
```

Cause of switch from 16-bit to 32-bit

We conclude that the instruction `movw $(SEG_KDATA<<3), %ax` is the first to be executed in 32-bit mode, while the `ljmp $(SEG_KCODE<<3), $start32` instruction finalises the transition to 32-bit protected mode.

First and Last Instructions

Further examination of `bootasm.S`, `bootmain.c`, and `bootblock.asm` reveals that `bootasm.S` transitions the system into 32-bit mode before calling `bootmain.c`, which then loads the kernel via the ELF header and ultimately transfers control to the kernel through the `entry()` function. Consequently, the final instruction executed by the bootloader is `entry()`. When examining `bootblock.asm` for this, we identify the corresponding instruction:

```
7d81: ff 15 18 00 01 00          call    *0x10018
```

Which is a call instruction. Since dereferencing operator(*) has been used, this instruction changes control to 0x10018.

To find the starting address of the kernel, we can look at the contents of "objdump -f kernel", and we need to check the instruction stored at the relevant address to get the beginning instruction of the kernel, by using the command "x/1i 0x0010000c"

```
(gdb) b *0x7d81
Breakpoint 1 at 0x7d81
(gdb) c
Continuing.
The target architecture is set to "i386".
=> 0x7d81:      call   *0x10018

Thread 1 hit Breakpoint 1, 0x00007d81 in ?? ()
(gdb) si
=> 0x10000c:    mov    %cr4,%eax
0x0010000c in ?? ()
```

The beginning instruction will be:

```
0x10000c:    mov        %cr4,%eax
```

Kernel Loading and Sectors

The following code in `bootmain.c` is used by xv6 to load the kernel.

```
ph = (struct proghdr*)((uchar*)elf + elf->phoff);
eph = ph + elf->phnum;
for(; ph < eph; ph++){
  pa = (uchar*)ph->paddr;
  readseg(pa, ph->filesz, ph->off);
  if(ph->memsz > ph->filesz)
    stosb(pa + ph->filesz, 0, ph->memsz - ph->filesz);
}

```

First, xv6 loads the ELF headers of the kernel into a memory location specified by the `elf` pointer. It then determines the starting address of the first segment to be loaded using `ph`, which is calculated by adding an offset (`elf->phoff`) to the base address (`elf`). Additionally, an end pointer `eph` is maintained, which points to the memory location right after the last segment.

The bootloader processes each segment by iterating while `ph < eph`. For each segment, `pa` represents the address where the segment should be loaded. The segment is then loaded at this address using `readseg`, with parameters `pa`, `ph->filesz`, and `ph->off`. After loading, if the allocated memory exceeds the size of the data copied, the extra memory is initialised to zeros.

The bootloader continues to load segments as long as `ph < eph` holds true. The `ph` and `eph` values are determined by the `phoff` and `phnum` attributes in the ELF header. Therefore, the information in the ELF header guides the bootloader on how many segments to read.

- ## Exercise 4:

Upon running the command "objdump -h kernel",

```
udbhav@Udbhav514:~/xv6-public$ objdump -h kernel

kernel:     file format elf32-i386

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         00007188  80100000  00100000  00001000  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .rodata       000009cb  801071a0  001071a0  000081a0  2**5
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  2 .data         00002516  80108000  00108000  00009000  2**12
                  CONTENTS, ALLOC, LOAD, DATA
  3 .bss          0000afb0  8010a520  0010a520  0000b516  2**5
                  ALLOC
  4 .debug_line   00006aaf  00000000  00000000  0000b516  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  5 .debug_info   00010e14  00000000  00000000  00011fc5  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  6 .debug_abbrev 00004496  00000000  00000000  00022dd9  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  7 .debug_aranges 000003b0  00000000  00000000  00027270  2**3
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  8 .debug_str    00000def  00000000  00000000  00027620  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  9 .debug_loclists 000050b1  00000000  00000000  0002840f  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
 10 .debug_rnglists 00000845  00000000  00000000  0002d4c0  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
 11 .debug_line_str 00000132  00000000  00000000  0002dd05  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
 12 .comment      0000002b  00000000  00000000  0002de37  2**0
                  CONTENTS, READONLY
```

We notice that the VMA of the .text section and LMA of the .text section are not the same, showing that it loads and executes from separate distinct addresses.

Upon running the command "objdump -h bootblock.o",

```
udbhav@Udbhav514:~/xv6-public$ objdump -h bootblock.o

bootblock.o:     file format elf32-i386

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         000001c3  00007c00  00007c00  00000074  2**2
                  CONTENTS, ALLOC, LOAD, CODE
  1 .eh_frame     000000b0  00007dc4  00007dc4  00000238  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  2 .comment      0000002b  00000000  00000000  000002e8  2**0
                  CONTENTS, READONLY
  3 .debug_aranges 00000040  00000000  00000000  00000318  2**3
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  4 .debug_info   00000585  00000000  00000000  00000358  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  5 .debug_abbrev 0000023c  00000000  00000000  000008dd  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  6 .debug_line   00000283  00000000  00000000  00000b19  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  7 .debug_str    00000206  00000000  00000000  00000d9c  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  8 .debug_line_str 00000041  00000000  00000000  00000fa2  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  9 .debug_loclists 0000018d  00000000  00000000  00000fe3  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
 10 .debug_rnglists 00000033  00000000  00000000  00001170  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
udbhav@Udbhav514:~/xv6-public$
```

We notice that the VMA of the .text section and LMA of the .text section are the same, showing that it loads and executes from the same address.

- ## Exercise 5:

The task is to change the boot loader's link address and observe changes. Initially the address was set to 0x7C00, which we changed to 0x7C08. The BIOS remains unchanged hence it ran smoothly for both versions and handed over control to the boot loader. The differences were compared hereafter as shown below using si command. We set a breakpoint at 0x7C00, and differences were observed a few lines after the breakpoint.

```
[   0:7c1b] => 0x7c1b:  out    %al,$0x60
0x00007c1b in ?? ()
(gdb) si
[   0:7c1d] => 0x7c1d:  lgdtl  (%esi)
0x00007c1d in ?? ()
(gdb) si
[   0:7c22] => 0x7c22:  mov    %cr0,%eax
0x00007c22 in ?? ()
(gdb) si
[   0:7c25] => 0x7c25:  or     $0x1,%ax
0x00007c25 in ?? ()
(gdb) si
[   0:7c29] => 0x7c29:  mov    %eax,%cr0
0x00007c29 in ?? ()
(gdb) si
[   0:7c2c] => 0x7c2c:  ljmp   $0xb866,$0x87c31
0x00007c2c in ?? ()
(gdb) si
The target architecture is set to "i386".
=> 0x7c31:      mov    $0x10,%ax
0x00007c31 in ?? ()
(gdb) si
=> 0x7c35:      mov    %eax,%ds
0x00007c35 in ?? ()
(gdb)
```

Before changing Makefile

```
(gdb) si
[   0:7c1b] => 0x7c1b:  out    %al,$0x60
0x00007c1b in ?? ()
(gdb) si
[   0:7c1d] => 0x7c1d:  lgdtl  (%esi)
0x00007c1d in ?? ()
(gdb) si
[   0:7c22] => 0x7c22:  mov    %cr0,%eax
0x00007c22 in ?? ()
(gdb) si
[   0:7c25] => 0x7c25:  or     $0x1,%ax
0x00007c25 in ?? ()
(gdb) si
[   0:7c29] => 0x7c29:  mov    %eax,%cr0
0x00007c29 in ?? ()
(gdb) si
[   0:7c2c] => 0x7c2c:  ljmp   $0xb866,$0x87c39
0x00007c2c in ?? ()
(gdb) si
[f000:e05b]    0xfe05b: cmpw   $0xffc8,%cs:(%esi)
0x0000e05b in ?? ()
(gdb) si
[f000:e062]    0xfe062: jne    0xd241d0b0
0x0000e062 in ?? ()
(gdb) si
```

After changing Makefile

The output of objdump -h bootmain.io after change has been displayed below:

```
udbhav@Udbhav514:~/xv6-public$ objdump -f kernel

kernel:     file format elf32-i386
architecture: i386, flags 0x00000112:
EXEC_P, HAS_SYMS, D_PAGED
start address 0x0010000c
```

- **Exercise 6:**

```
of GDB.  Attempting to continue with the default i8086 settings.

(gdb) b* 0x7c00
Breakpoint 1 at 0x7c00
(gdb) c
Continuing.
[   0:7c00] => 0x7c00:  cli

Thread 1 hit Breakpoint 1, 0x00007c00 in ?? ()
(gdb) x/8x 0x10000
0x10000:        0x00000000      0x00000000      0x00000000      0x00000000
0x10010:        0x00000000      0x00000000      0x00000000      0x00000000
(gdb) b *0x10000c
Breakpoint 2 at 0x10000c
(gdb) c
Continuing.
The target architecture is set to "i386".
=> 0x10000c:    mov    %cr4,%eax

Thread 1 hit Breakpoint 2, 0x0010000c in ?? ()
(gdb) x/8x 0x10000
0x10000:        0x464c457f      0x00010101      0x00000000      0x00000000
0x10010:        0x00030002      0x00000001      0x0010000c      0x00000034
(gdb) 
```

In this experiment, we will examine 8 bytes of memory at address 0x00100000 at two distinct points in time: first when the BIOS transitions to the boot loader, and second when the boot loader transitions to the kernel.

Setting Breakpoints

1. First Breakpoint: BIOS to Boot Loader Transition
   - Set the first breakpoint at address 0x7c00. This address marks the moment the BIOS transfers control to the boot loader.
2. Second Breakpoint: Boot Loader to Kernel Transition
   - Set the second breakpoint at address 0x0010000c. This address indicates when the boot loader hands control over to the kernel.

Memory Examination

Once the breakpoints are set, use the command x/8x 0x00100000 to examine the 8 bytes of memory at address 0x00100000 at the specified breakpoints.

Lab Report 1B

- **Exercise 1:**

Kernel mode and user mode are the two modes in an operating system. A system call is used to ask the kernel for permission to access to RAM or any hardware resource that may be required by a program in user mode. Upon provoking a system call, it switches from user mode to kernel mode.

In xv6, if we aim to build our own system call, relevant changes must be made in these files: syscall.c, syscall.h, usys.h, user.S, sysproc.c.

Let us begin with syscall.h. This file consists of the system call and its respective numbering. Let's add our new system call at the end.

```
#define SYS_banana 22
```

Moving on to the syscall.c file, it contains an array of function pointers by the name of *syscalls* which use the relevant numbers in the syscall.h file as pointers to system calls, which have been defined elsewhere. Let us add the pointer for our new system call inside this array syscalls.

```
[SYS_banana] sys_banana
```

Notice that the number for our SYS_banana system call is 22. So when it is called by a user program, the function pointer sys_banana which has the index 22, will call the system call function. Now, we have to implement this system call to get everything up and running. We shall do the actual implementation of the system call function in sysproc.c and reference it here. Make sure to include the following line because we are defining the function in another file and need to use it here :

```
extern int sys_banana(void)
```

Following is the relevant code for sys_banana system call function:

```c
// return how many clock tick interrupts have occurred
// since start.
int sys_uptime(void)
{
  uint xticks;

  acquire(&tickslock);
  xticks = ticks;
  release(&tickslock);
  return xticks;
}
int sys_banana(void)
{
  void *buf;
  uint size;

  argptr(0, (void *)&buf, sizeof(buf));
  argptr(1, (void *)&size, sizeof(size));

  char bababa[] = "\n\
                                                  :=+***=\n\
                                                 +@@@@@@@=\n\
                                                  -#@@@+#\n\
                                                   .#=-+=\n\
                                                   .#--=*\n\
                                                   -*---*=.\n\
                                                  .*=----=++:\n\
                                                 .*=--------+*.\n\
                                                -*-----==----=*=\n\
                                               -*-------=------++\n\
                                              .#--------=-------++\n\
                             ::-::.           #=--------=--------#:\n\
                            :: .. -::        =*--------==--------=*\n\
                             -, : : =       .#=--------=---=+==--=#\n\
                             -:.....=.::.   *+===-----==-======---#\n\
                             =      ..  =  :#.-*+==---==+.=*=+=--=#\n\
                             =::.   ::::  :+.:+@@++--==+ =@@@=*--+=\n\
                             *+-:-:.     -*+ +@@++=-===+ #@@+-+--#.\n\
                            ++= .       ++-+-*####+==-=+-####*+-++\n\
                            +*.  ..:--=*=---+++++*==++==*++++*-=# ::.     ::.\n\
                            -*####**+++------=++===------=++=-=#.:: -..-:..::-.\n\
                                  -*+------------==------------#:  -. :.::....::\n\
                               .=*=------------==------------*:   ::     :.::::.\n\
                             -++=------------==------------=#*     =  :  .=.\n\
                          .=*=--------------==------------=*-+*    -:..-::\n\
                       :=++---------------==--------------*+  .#+:   *+-:\n\
                    .=*+----------------==-------------=*:     ++*:=@+\n\
                 .=++----------------===-------------++        .=**-\n\
               -++----------------===---------------=*+.\n\
            .+*=-------------====---------------=*#+.\n\
         .=++-------------====-----------------+*=.-++\n\
        -++=---------=====----------------=*+:    ++\n\
        =@@+----=====------------------+**:        *@.\n\
         +@@=----------------------=+++==@=        =+-\n\
          .+*+==--------------==+++=-.   -@=        =+-\n\
             .:-====+++++++===-:        :=@*:      :++=:\n\
                                       .=-*=-=    =-=*-=:\n\
                                     --+++=--+:  .+--=+++--.\n\
                                   -+----=----=  -=---=----=-\n\
                                   *--------=+=:.=+=--------+.\n\
                                   +======---:-  ::---======*.\n\
                                    .:::::::.       .:::::::.  \n\n";

  if (sizeof(bababa) > size)
    return -1;

  strncpy((char *)buf, bababa, size);
  return sizeof(bababa);
}
```

Now we need to add a means for user program to call this system call, which can be done by adding the following lines

To user.h:

```
int banana(void *, uint)
```

To usys.S:

```
SYSCALL(banana)
```

- ## Exercise 2:

The only remaining obstacle is adding a user program to call the custom system call! Add a file named bananacheck.c in the xv6 folder with the following code, which is just a simple system call:

```
#include "types.h"
#include "stat.h"
#include "user.h"

int main(void)
{
    static char buf[5000];
    printf(1, banana sys call returns %d\n, banana((void *)buf, 5000);
    printf(1, "%s", buf);
    exit();
}
```

Add bananacheck.c into the file makefile under UPROGS and EXTRA. Now execute the following commands on the terminal and voila!

```
make clean
make
make qemu-gdb
bananacheck
```

Here are our results:

You can also check the existence of the program by running the following command:

```
ls
```

We obtained the following output:

```
udbhav@Udbhav514:~/xv6-public$ make qemu
qemu-system-i386 -serial mon:stdio -drive file=fs.img,index=1,media=disk,format=raw -dri
xv6...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$ ls
.              1 1 512
..             1 1 512
README         2 2 2286
cat            2 3 15476
echo           2 4 14356
forktest       2 5 8804
grep           2 6 18320
init           2 7 14976
kill           2 8 14440
ln             2 9 14340
ls             2 10 16908
mkdir          2 11 14468
rm             2 12 14448
sh             2 13 28504
stressfs       2 14 15372
usertests      2 15 62876
wc             2 16 15904
zombie         2 17 14024
bananacheck    2 18 14288
console        3 19 0
$
```

Notice that bananacheck is present in this list.