Task: Use only ATA0, embed the kernel into the file system

Carmi Merimovich

As distributed, the xv6 kernel assumes ATA0 holds xv6.img and ATA1 holds fs.img. As seen in lecture, xv6.img holds bootblock and kernel and fs.img contains the file system. As not seen in lecture fs.img is being built by the host program mkfs. (This is what a high level ‘formatting’ program does.) A whole drive holding only the kernel is a bit of an over kill. The aim of the task we present is to put the kernel into the file system, and let ATA0 holds the file system. Of course, the way we put the kernel into the file system should be simple in order to allow the bootmain.c to load it, and still be small enough to fit into the bootblock.

One point worth noting is that the makefile can build kernelmemfs which combines the file system and the kernel into one file using ld tricks. This, however, requires changing the ATA driver, i.e., the file ide.c, for this build, and is kind of a strange way to build the file system.

Let us begin. Begin with a fresh copy of xv6-rev7. We will first deal with loading the kernel into the file system, and then deal with changing the bootblock. The first step is to load the kernel into the file system. This looks quite easy. The mkfs program accepts list of files to load into fs.img, so just add kernel to the list.

1. In the makefile add to the end of the UPROGS macro kernel.
2. make xv6.
3. OOOOPPPPSSSSS! There is an error when mkfs runs.
4. Read the mkfs code. What is the error about?
5. Remove kernel from UPROGS.

Well, it seems the problem is the maximal file size is 12 + 512/4 blocks. This is quite a restriction, especially considering the kernel size is about 250 blocks. (Note that different versions of binutils and gcc (and compiling/linking options) yield different file size.) We can either change the structure of the file system (quite a task(!), or is it?) or split the `kernel` to smaller files! So we split the `kernel`. As long as the kernel size is up to 280 blocks, the way we proceed makes sesnse. Otherwise, splitting to three files is needed.

1. Add to the `makefile` the following lines.

   ```
   kernel0: kernel
       dd if=kernel of=kernel0 bs=512 count=125

   kernel1: kernel
       dd if=kernel of=kernel1 bs=512 skip=125
   ```

2. `make  xv6`. Everything should work as usual.

3. Look in the directory for the new files `kernel0` and `kernel1`. What are their sizes?

Now we can try again to load the `kernel` into the file system

1. In the `makefile` add to the end of the `UPROGS` macro both `kernel0` and `kernel1`.

2. `make xv6`. Everything should work.

3. `bochs -q` to `xv6`.

4. `ls` and see the new files `kernel0` and `kernel1`. What are their size?

Files in file system need not be physically continuous. Luckily, the `mkfs` utility loads files into the file system image almost continuously. Since we want the loading code to be as simple as possible, we want each of the `kernel` parts to be continuous. The reason `mkfs` loaded files are not physically continuous is the indirect block which is allocated for files larger than 12 blocks. This is done in a way similar to the way the kernel `filesystem` actually works. Since we *know* both `kernel` parts are larger than 12 blocks we will preallocate the the indirection block for these files. **Note. This is NOT a good solution. A better solution is suggested at the end.** The code in `mkfs` responsible for allocating the indirection block follows.

```
rinode(inum, &din);
if(xint(din.addrs[NDIRECT]) == 0){
        // printf("allocate indirect block\n");
```

```
        din.addrs[NDIRECT] = xint(freeblock++);
        usedblocks++;
    }
    winode(inum, &din);
```

Modify the the `mkfs` utiltity to preallocate the indirection block for the `kernel` parts.

1. Identify the loop in `mkfs` where file is read from the host and written to the image.
2. Before this loop add the following code:

   ```
   if ((strcmp(argv[i], "kernel0") == 0)   ||
               (strcmp(argv[i], "kernel1") == 0)) {
       struct dinode din;

       rinode(inum, &din);
       if (xint(din.addrs[NDIRECT]) == 0) {
           // printf("allocate indirect block\n");
           din.addrs[NDIRECT] = xint(freeblock++);
           usedblocks++;
       }
       winode(inum, &din);
   }
   ```

3. `make xv6`.
4. `bochs -q` into `xv6`. Everything should work.
5. Convince yourself indeed both kernel parts are physically continuous. (Say, by printing the block numbers, or checking if the gap between consecutive file blocks is indeed one.)

Now we need to find away to communicate the location of the `kernel` parts to `bootmain.c`. A reasonable way is have `mkfs` leave this information in a header file as follows:

```
#define kernel0_begin     (nnn)
#define kernel0_length    (nnn)
#define kernel1_begin     (nnn)
#define kernel1_length    (nnn)
```

So modify the `mkfs` to do this.

1. Identify the main loop of the `mkfs` program. (The loop which starts with i=2. Add the following code before this loop.

   ```
   FILE *kernel = fopen(”KernelBlocks.h”, ”w”);
   if (kernel == NULL) {
                  perror(”KernelBlocks.h”);
                  exit(1);
   }
   ```

2. After the line winode(inum, &din); of the indirection block allocation add the following code. (Do not forget a declarion for `b` in a reasonable place.)

   ```
       b = freeblock;
       fprintf(kernel, ”#define␣%s_begin␣␣(%d)\n”, argv[i], b);
   ```

3. After the read loop add the follwoing code.

   ```
     if ((strcmp(argv[i], ”kernel0”) == 0)   ||
     (strcmp(argv[i], ”kernel1”) == 0))
        fprintf(kernel, ”#define␣%s_length␣␣(%d)\n”,
                                     argv[i], freeblock - b);
   ```

4. After the main loop add

   ```
      fclose(kernel);
   ```

5. `make xv6`.

6. Read the file ‘KernelBlocks.h’ to see it is properly formatted and have the correct information.

At this point the file `fs.img` is to our liking. We are left with fixing `bootmain.c`.

1. In the `makefile` add to the dependencies of `bootblock` the file `KernelBlocks.h`.

2. Add `#include` to `KernelBlocks.h` to `bootmain.c`.

3. Add the following lines to the beginning of the function `readsect` in `bootmain.c`:

   ```
     if (offset < kernel0_length)
        offset += kernel0_begin - 1;
     else
        offset -= kernel0_length -  kernel1_begin - 1;
   ```

4. `make` xv6.

One last point. We read in lecture the boot code, and the one routine which accesses `ATA0`. Apparently, the `kernel` accesses `ATA1` when it needs to access the file system. This is controlled by the ROOTDEV constant in the `kernel`. Change it to 0. Finally we need to build `xv6.img` containing `fs.img`.

1. In the `makefile` replace the line ‘dd’ing `kernel` into `xv6.img` with the following:

   ```
   dd if=fs.img of=xv6.img skip=1 seek=1 conv=notrunc
   ```

2. `make xv6`.

3. `bochs -q` into it. **Boot should be succesful.**