The Linux Kernel 0.01 Commentary

Pramode C.E
Mobiuz Technologies

Gopakumar C.E
Cisco Systems

The Linux Kernel 0.01 Commentary

by Pramode C.E and Gopakumar C.E

This document describes the structure of a prehistoric Linux kernel as understood by the authors. Operating system newbies, hardware hackers or people with too much time in their hands than they would care to admit can use this document to learn more about 80386 architecture and the simple skeleton from which a great Operating Sytem (and a greater movement) was built.

The authors are NOT kernel hackers, and as such, the document is not guaranteed to be technically perfect. Reports of errors will be gratefully received and, time permitting, corrections would be incorporated in later versions. Flames will be redirected to /dev/null.

Happy Hacking!

Table of Contents

Chapter 1. Getting Started

1.1. Introduction

1.1.1. Copyright and License

1.1.2. Feedback and Corrections

Kindly forward feedback and corrections to *gopakumar_ce@yahoo.com*.

1.1.3. Acknowledgements

Gopakumar would like to thank the faculty and friends at the Government Engineering College, Trichur for introducing him to GNU/Linux and initiating a 'Free Software Drive' which ultimately resulted in the whole Computer Science curriculum being taught without the use of propreitary tools and platforms.

We express our gratitude towards those countless individuals who answer our queries on Internet newsgroups and mailing lists, those people who maintain this infrastructure, the hackers who write cool code just for the fun of writing it and everyone else who is a part of the great Free Software movement.

1.1.4. Reading 0.01 source

The Linux kernel version 0.01 is a tiny program (compared to the size of the current kernel). A newbie is faced with a problem - whether to read and understand a part of a big, sophisticated program or whether to read and understand almost in full a smaller and MUCH less sophisticated version. If you have of plenty of time at hand, you can do both! We started off by learning a bit of 80386 architecture and then poking around an ancient Linux kernel (you know, Linus too was a newbie!), trying to compile it on our 'modern' Red Hat Linux system (RH version 6.2 - we did these experiments some time back) and getting frustrated reading huge Intel manuals describing the mysteries of the 386 processor. This document might be useful for those sufficiently crazy folks out there (we know there would be many) who would like to conduct similar experiments.

1.1.5. Things you must know

We expect that you have/know:

- Programming experience in any UNIX environment, familiarity with UNIX system calls.

- Simple theoretical concepts about Operating Systems - like layers of an OS, scheduling, memory management etc.
- Good Knowledge of 8086 architecture, how interfacing is done etc.
- Basic idea about 80386 architecture - like switching to protected mode, how is protection enforced, how is paging done, etc.
- Basic concepts about what the output of assemblers, compilers, linkers etc will look like (ie their headers, symbol tables etc).

Well, it is quite difficult to enumerate exactly what you have to know and how much to know. The only way is to proceed with this book and learn the things that you don't know whenever you find it necessary. Please don't do the reverse process of learning the required things first and then proceeding with the actual intention. Learning for the sake of learning is boring and inefficient. The chapter on 80386 may not provide all the information that you need, it is presented just as a quick tour of 80386 which will greatly help you when you read any other detailed books on 80386. Anything else that you need to know about which you don't find in this book can be found in one of the below mentioned reference material.

1. The C programming language by K & R
2. Design of the Unix Operating System by Maurice.J.Bach
3. Advanced Programming in the Unix environment by W. Richard Stevens
4. The info and man pages (mainly on gcc, as, ld, as86, ld86)
5. The Intel 80386 manuals
6. The Indispensable PC Hardware Book by Hans-Peter Messmer

The book 2. is extremely essential because Linus has based the design of 0.01 along the lines specified by the author of the book. The book 3. is needed just for reference on UNIX programming. You can do even without it. The book 6. is simply a must if you don't have any other reference on the interfacing chips in a PC;It is worth having a copy. Also it is better to refer the 80386 manuals for information rather than any 80386 text books. If you have a pretty good net connection, you can download the Intel Manuals from the URL we have mentioned at the end of this sub-section. We just want architecture related information which is presented precisely and concisely in the manuals. Throughout the commentary, we will mention which of the above should be referred to find something that we have not explained. Also, all figures are from the Intel Architecture - Software Developer's Manual 3. That can be downloaded from the link http://x86.ddj.com/intel.doc/386manuals.htm.

1.1.6. How to read this document

We have tried to present sections in this document in the order that gives you a block-building approach to learning the .01 code. So at first, proceed with the chapters linearly. Later on you can adopt the random-walk approach. Again, going by the UNIX principles, we assume that the reader is intelligent and so not much coaxing and cajoling (If we were to go by that principle very strictly, then we should not have written this book because even without this book, you can easily read and understand the .01 kernel with some extra effort :-)). Again, it need not be mentioned that reading without experimenting will not help you any more than watching "Babie's Day Out". Well, finally the disclaimer - this book is a collection of our ideas. We are learning, so you will find mistakes. If following the instructions here leads to consequences which are not very pleasing to speculate - dont blame us.

Notes

1. http://x86.ddj.com/intel.doc/386manuals.htm

Chapter 2. Building Kernel 0.01

2.1. Getting 0.01 Up And Running

To get .01 running, it took us almost two weeks - going through various info and man pages and finally fixing that terrible trouble caused by the NT flag of the 386. The process by itself was highly educative. So don't be in a hurry and carefully understand what you are doing. Though we have provided a working version of 0.01, it would be good if you start with the original code and make modifications on your own and consult what we have written below only when you are really in trouble. We don't know whether we could have done things more easily, but ours' does work. We describe what we have done, briefly.

- From what Linus has written, we conjecture that the previous versions of gcc used to prefix an underscore ('_') to all the variable names declared in the program. So in the assembly files linked with the C code, those variables are accessed with the _xxx type of names. But the present gcc compilers doesn't do such things. So you have to go through all the assembly files, some header files and also portions where inline assembly is used and just delete the leading underscore from the variable name. Which files to be looked into will be known at the time of linking.
- Another minor change is the change of comment symbol from '|' in boot/boot.s to ';'.
- The options to the old linkers, assemblers etc are not the same as those for the new ones. So the Makefiles in all the directories had to be modified. Please read the original Makefiles and the ones that we have supplied and note the difference. The only difference is in the options to LD, AS, GCC etc. The only change that requires clarification is the -r and -T options to ld in the linux-0.01/Makefile. What we do here is we make the linker produce object files in a relocatable form (with all the symbols) and combine the various object files to produce the file 'system' also in relocatable form. Then we run a small linker script (the file 'script') using the -T option to ld. The script simply strips all the headers from the file called 'system', thus making the code in 'system' relative to absolute address 0x0 (because all relocation information is lost). Also it has also one other purpose - it also expands the bss block present in the relocatable form of system since when the file 'system' is loaded into memory, space has to present for variables intended to be in the bss block. In this particular case, if bss is made the last block in 'system', we can do without expanding it because the space in memory after the data and code in the file 'system' , will be free anyhow ('system' is loaded at 0x0). But still including it gives us an idea of the total size of the system from the size of the file 'system'. Otherwise, we would have to find out the size of the bss separately and add it to the file size (we need to know the 'system' size in boot/boot.s). Also a few unnecessary blocks like .note and .comment are removed. We have also added the -b option to as86 in the linux-0.01/Makefile to produce code relative to 0x0. More information on the options to linkers, assemblers, compilers etc can be found in the info pages. Information on as86 and ld86 can be got from the man pages. We did'nt find man pages for as86 and ld86 in our RedHat 5.2 installation, but it was there in the 6.2 version. The changes in the other Makefiles are minor and no clarification is needed.
- We have modified the linux-0.01/tools/build.c file. The original code was responsible for removing the headers of the object file produced by assembling boot.s, removing the headers of the 'system' file and generating a boot block and appending 'system' to it. But since our Makefiles tell the assemblers and linkers to produce raw binary output, we don't need another program to do it. So our tools/build.c simply produces a boot block of 512 bytes. The final 'Image' is produced by simply joining the boot block and 'system' with a 'cat'.

- The 0.01 uses a minix version 1 file system as its root file system. So make a primary partition in your hard disk. Set its type to 80 (Old Minix) using the fdisk in your Linux system. Suppose your partition is hda4. Then format the partition using the command 'mkfs.minix /dev/hda4 -n 14'. The -n 14 option specifies the maximum filename length to be 14 characters, which is what 0.01 assumes. Now create the directories bin and dev in the minix file system. Also create nodes corresponding to your root partition (hda4 in our example) and the console tty0. The major and minor numbers for these devices are present in the file linux-0.01/include/linux/fs.h. You can put a shell (sh) in the bin directory.
- The linux-0.01/include/linux/config.h file has to be edited to make 0.01 to mount its root file system. Use fdisk to find out the number of cylinders, sectors and heads of your hard disk. If fdisk shows the number of heads to be greater than 64, then bad luck, we don't know what to do. The original code had restricted the number of heads to 16, but we modified it to 64. When it is more than that, then it is not possible to use the CHS method to program the old style ide interface, because only 6 bits have been provided for specifying the number of heads. We did encounter such a problem and what we did was to wipe out the entire data on our disk, make the number of heads equal to 64 (using the advanced options in fdisk) and start all over again (so simple!!). So you can decide for yourself what to do. Also it is necessary that your partition ends within 1024 cylinders (approx. 2+ Gb with 64 heads, 63 sectors and 1024 cyl). This is because only 10 bits are reserved for the cylinder number in the CHS method of ide programming. With all these restrictions, we tried 0.01 on a variety of harddisk capacities like 1.2 Gb, 2.1 Gb, 4.3 Gb etc. But when we bought a new 10 Gb harddisk and tried this on it, it didn't work though all the restrictions were obeyed on the 10Gb harddisk (like heads <= 64, minix end cyl <= 1024 etc). So we just ran out of ideas and simply started a trial and error method with the harddisk code in the linux-0.01/kernel/hd.c file. Finally, when we commented one line in the hd_out function, it worked!!. That line is needed according to the ide programming manuals, but excluding it made .01 work!. Again, on all the disks that we tried, we had the disk as primary master ide. There is code in the hd.c for secondary ide also, but we have not tried it. So we assume that your hdd is also the primary master ide. Now if your harddisk configuration satisfies all these restrictions, note down the number of cylinders, heads and sectors in your harddisk. The we have to ake some changes in the linux-0.01/include/linux/config.h file. First change the ROOT_DEV symbol to correspond to your root partition. For example if your minix partition is hda4, then ROOT_DEV is 304, if it is hda3, then 303 etc. Only the LINUS_HD entry has to be modified. Then change the first of the two structures under the LINUS_HD macro check to reflect the configurations of your harddisk. We assume that your hdd is primary master which corresponds to the first structure. The WPCOM, LANDZ parameters can be got from your BIOS settings. We don't know whether they have any significance in the present day harddisks. We made CTL 8 when we read the comment provided above the structures. Again, we don't know whether it has any significance.
- In the file linux-0.01/include/string.h, many variables have been declared as register type __res __asm__(..), which the gcc compiler does not permit. So we simply changed it to type __res. We don't know whether it has any serious implications in the performance of the code.
- In the file linux-0.01/include/asm/segment.h, the value to be returned from a function is assigned via an arbitrary register specifier (=r). But gcc complains about that and so we have changed it to (=a), ie the ax register itself. We have also changed a few other (=r) specifications to (=a).
- In the file linux-0.01/kernel/hd.c, we have changed a register variable declaration to an ordinary declaration. Again the symbol NR_HD has been defined by us to be 1, the definition for NR_HD present in the original code gives it a value of 2 because the HD_TYPE sym-

bol is defined to be the two harddisk info structures in config.h. Again we assume here that our minix partition is present in the primary master ide disk. Also as we mentioned before, we have commented out one line in the hd_out() function to get it working on our 10Gb harddisk. Also we have changed the original head > 15 check in the hd_out() function to hd>63 to accommodate for 64 head harddisks.

- The 0.01 assumes an a.out format for executables, but giving options to gcc to produce the a.out format doesn't seem to work. So what we do is to compile our C program for the 0.01 (maybe a shell or anything we want to run in 0.01) and use ld to produce a raw binary output of the compiled code. Then we wrote a small piece of C code that reads the size of the binary and attaches a header to the binary containing details that the exec function in 0.01 expects to find. This header + binary is what gets executed in 0.01 when we call the execv system call from our program that gets executed in 0.01. The code that we wrote for this purpose is present in the file linux-0.01/bin/shell/header/header.c. Also the job of compiling our application program for the 0.01, then producing a raw binary using ld and then giving the size details of the file to our header code and producing the final binary file for 0.01, etc.. is done using a small script linux-0.01/bin/shell/mkbin. If we compile a file abcd.c using mkbin, then the binary of that file is present in the name abcd. Also mkbin does the job of linking the application program to the printf library and the system call archives. One thing that has to be taken care of while writing applications and producing binary using mkbin is that the main() function should be the first function to be defined in the application code.

- Finally we come to one of the two problems that left us clueless for almost two full days. As we mentioned before, we had run the 0.01 on different machines in our lab - not with the intention of testing it on different machines, but because no single machine was permanently available for use for this purpose; whichever machine was free, we would work on it. The 0.01 code first ran successfully on a brand new AMDK6, but when we tried it on an old Pentium I, it simply showed some protection fault. The code dump by the kernel showed that the fault was in the linux-0.01/boot/head.s file. A comparison with the head.S of the new kernel showed that clearing the NT flag may solve the problem. So just two instructions for making the flag register contents 0 (the same lines as in the new head.S) did solve the problem. So that is the piece of change that gave us great satisfaction that we did debug something!?. Maybe the problem was that the BIOS in the AMDK6 machine reset the NT flag before the kernel started, but the BIOS in the Pentium I machine didn't do it. So when the kernel did the first iret instruction (maybe in the move_to_user_mode()), the x86 thought that it is a task switch was to a previous task - but whose link was set to NULL when the task structure of each process was initialized.

- The second problem was that the kernel created a protection fault inside the fork system call (we think so, don't remember exactly). Reading the objdump of the area where the protection fault was generated identified the problem with the get_base code in linux-0.01/include/linux/sched.h file. The code generated the base address of a process in the edx register. That value had to be transferred to a variable __base, which the code returned. But the funny thing was that for transferring the contents of the edx register to the variable __base, the address of the variable __base was initially stored in the register edx itself and finally indirectly addressed to store the contents of edx (the base address). But since all the burden was on poor edx, by the time it was indirectly addressed, it contained some rubbish value, leading to a protection fault. Again after rummaging through the info pages (you can find info on inline asm under the C extensions section of gcc info) we changed the =d (__base) declaration present in the original code to =&d (__base) and this again solved the problem.

Chapter 3. Processor Architecture

3.1. The 386 Architecture

In this chapter, we deal with the architecture of the 386 processor. As we had mentioned in the introduction, this is not a full description of 386, it is intended only to help the reader to cope up with the rigor of the 386 manuals. So Keep the manuals nearby and start reading this chapter. We avoid explanations on basic things like register set, instruction set etc and present oly matter relevant to understanding the 0.01 code. Supplement this by referring the manuals whenever necessary.

3.1.1. Segmentation in 386

As in 8086, we can divide our programs into segments with a base address and also a limit - the limit feature being absent in 8086. In 8086, the base address is generated by multiplying the segment register contents by 16. Then the offset is added to it to get the absolute address. So how is the base address generated in 386? The mechanism is entirely different here. Here the contents of the segment register have no direct mathematical relation to the base address. For the same value in the segment register, say for the value 8 in the CS register, you can give any base address - 0x0 or 0xf0 or 0x67823ab or anything you like, where as in 8086, the base address corresponding to CS = 8 will be 16 * 8 = 0x80. So naturally what comes to the mind of a programmer is a mapping table that maps a segment register value to a base address. Of course, we may not use all possible values as the contents of a segment register in our program. That is we know that we will be using only say three values as the contents of the DS register, like - 0x0, 0x8, 0x10, 0x18. So again the solution that comes to the programmers mind is to use a fixed size array as the map table - like int MAP_TABLE[3], with MAP_TABLE[0] = base address of segment 0x0, MAP_TABLE[1] = base address of segment 0x8, and so on. A picturisation of this concept is given below.

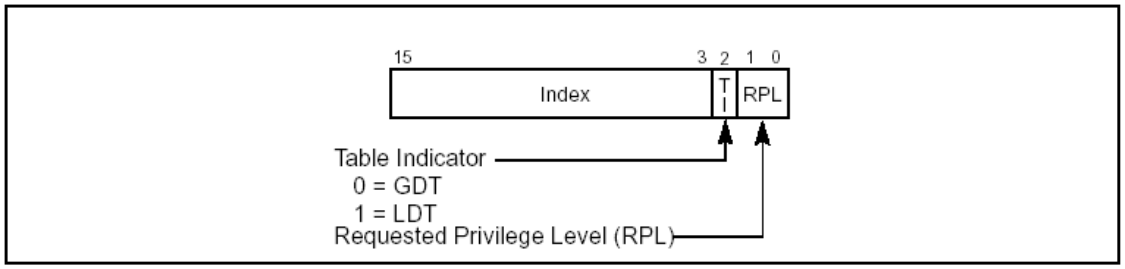

Figure 3-6. Segment Selector

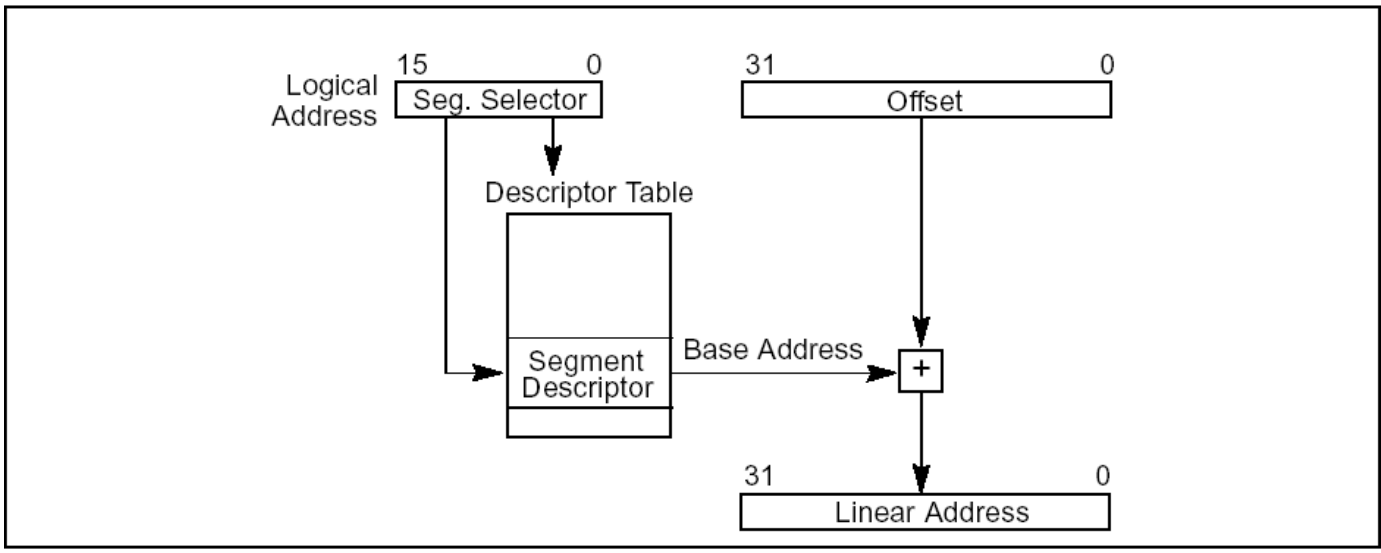

Figure 3-5. Logical Address to Linear Address Translation

Well, what is done in the 386 is also similar to this. The programmer should create a table and fill it entries with base addresses of the segment values that are used in the program. Such a table is called a Descriptor Table in the 386 nomenclature and the 'segment value' is formally known as the 'selector'. Now let us see in more detail about the descriptor tables.

3.1.1.1. The descriptor tables.

As in the 8086, the segment selector in 386 is also of 16 bit length (because the segment registers are 16 bit in length). But the interpretation of the contents of the segment registers is different in 386. The bits 0, 1 and 2 have special meanings. We will explain that later. From bit 4 onwards till bit 15, the value is interpreted as an index into the descriptor table to get the base address of the corresponding segment. For example the selector value 0xf3 is an index into the descriptor table of the index value (0xf3 >> 3), '>>' being the right shift operator in C. That is the reason why we started from 0x0 and went through 0x8, 0x10, 0x18 etc because we just ignored the least significant three bits. Now the address creation of 386 is becoming more and more clear. The 386 uses the selector in the segment register to get the base address of the segment being addressed from the descriptor table and adds the contents of the offset register used in the instruction to get the absolute memory address (wait, it is not fully complete, we will discuss paging later). Things like how the 386 knows where in memory has the programmer created the descriptor table etc will be explained later. Now let us see what bit 2 of a segment selector stands for. Well, now let us mention that the programmer can specify not just a single descriptor table, but two descriptor tables. Well why use two descriptor tables and not just a single large descriptor table equivalent to the concatenation of the two descriptor tables ? :-) The reason for that will be more clear when we get going with the kernel code. For now, just believe that the 386 programmer can tell the 386 that we are having two descriptor tables. One table is called the LDT (Local Descriptor Table) and the other table is called the GDT (Global Descriptor Table). So from which table do we get the base address of a selector, from LDT or GDT ? That is when we say the selector 0xf3, how does the 386 know whether the index (0xf3 >> 3) is into the LDT or the GDT ? There comes the role of the bit 2 of a selector. If the bit 2 of the selector is 0, it means we are asking the 386 to index (selector >> 3) into the GDT and if it 2 is 1, we are asking the 386

to index (selector >> 3) into the LDT. So if the selector is 0xf3, its bit 2 is 0, and so we are asking the 386 to use 0xf3 as an index to the (0xf3 >> 3) entry of the GDT. If we had wanted to use the same value (0xf3 >> 3) to index into the LDT, the selector should be 0xf3 | 0x4 = 0xf7.

Now let us get to the question of how the 386 knows where the programmer has kept the tables - LDT and GDT. The programmer can keep the LDT and GDT anywhere in memory. But after that, he/she has to specify in a structure (ie consecutive memory locations), the base address of the table and its limit. The format of the structure will be presented in the next section. Now there are instructions LGDT and LLDT of the 386 for specifying the location of the tables. To these instructions, we supply as operand, the base address of the structures that we created for the corresponding tables. From the structures, the instruction LGDT/LLDT can infer the base address and limit of GDT/LDT. Note that these structures need not be present permanently in memory - they can be discarded after the LGDT/LLDT instructions, but the LDT and the GDT has to be present in memory as long as any selector refers to those tables. If no descriptor in our programs refers to one of the tables, we need not keep that table in memory and can simply fill a structure with base address to some value and the limit to 0 and use LGDT/LLDT instruction with the base address of that structure as the operand to tell the 386 that we are not using GDT/LDT. Corresponding to each descriptor table, the 386 has got two registers - one for storing the base address of the table and the other for storing its limit. The format of those register will be presented in the next section. So what the LGDT/LLDT instruction does is to simply set the base address register and the limit register corresponding to the GDT/LDT by reading the contents of the structure whose address is specified along with the instruction. So after the LGDT/LLDT instructions, the 386 does know where to find the tables. So these instructions should be executed before any protected mode program begins to execute.

We mentioned that the descriptor tables contain the base addresses of segments. So what should be the size of one entry of a descriptor table ?, 4 bytes ?. No it is not 4 bytes, but 8 bytes. Apart from the base address of the segment represented by the indexing selector, the programmer may want to store in the descriptor table certain extra information like whether the segment is read only or read/write, whether it contains code or data etc. So in order to accommodate these extra information, the size of one table entry is 8 bytes. So a C language declaration for the table would be 'double GDT[10];' which declares a GDT for mapping 10 descriptor values 0x0, 0x8, 0x10, 0x18, ...The format of a descriptor will also be briefly explained in the next section. Refer the Intel manuals for complete information. Also since we have to specify a limit for our descriptor tables, we can use only a fixed number of selectors from each table, starting with 0x8 (0xc) for the GDT (LDT) and proceeding with the 'i'th selector (i >= 0) from 0x8(or 0xc, depending on which table is addressed) as long as the (base address of the table + i*8) does not exceed the limit of the table that we have specified. One table entry of a descriptor table is simply called a descriptor. Note that the selector is different from the descriptor. The selector is a 16 bit value present in a segment register. It is used to index into a descriptor table that contains 8 byte 'descriptors' containing base address and limit related information about he segment referred to by the selector.

3.1.1.2. LDT/GDT Descriptor Format

The format of the 8 byte descriptor is shown in the figure below.

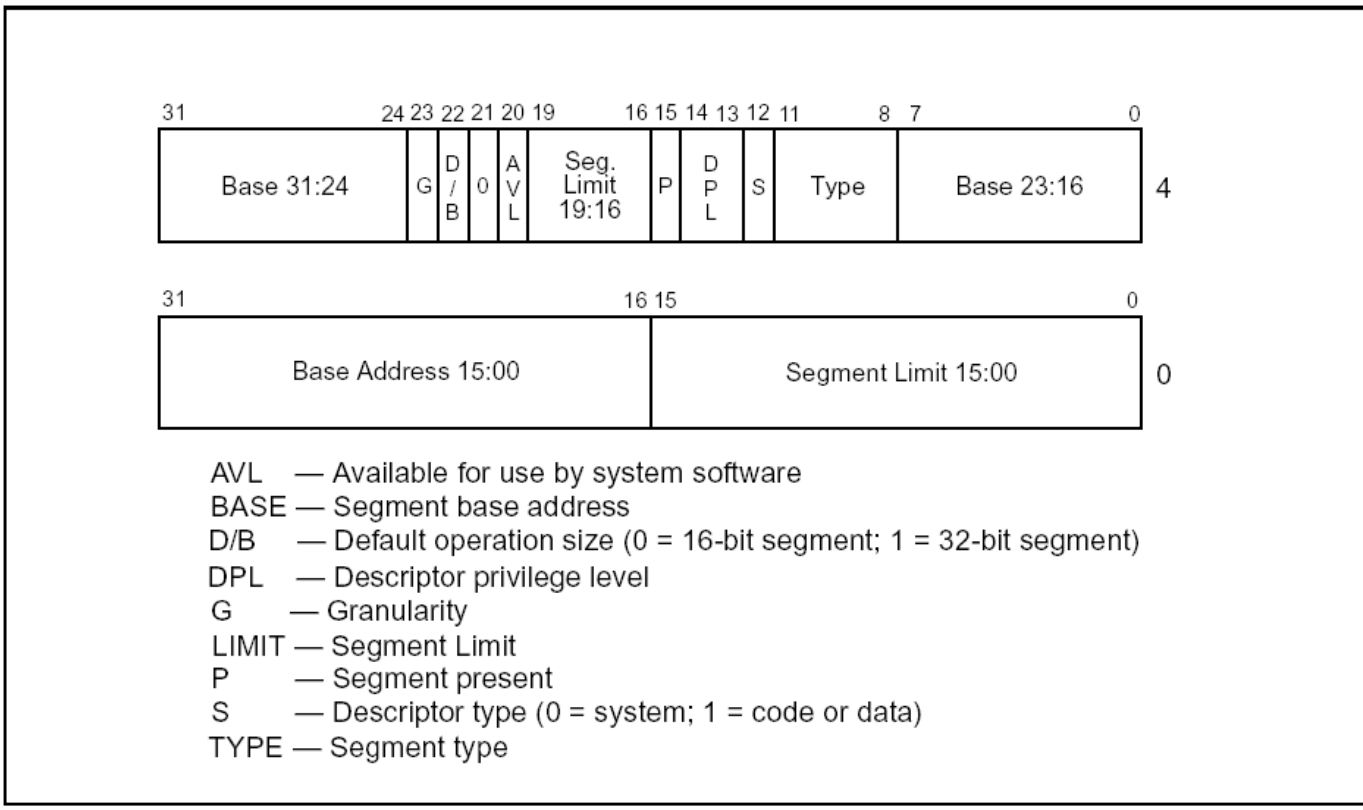

Figure 3-8. Segment Descriptor

The order in which the bytes are stored in memory is as follows. The first 32 bits (limit(16 bits) + base(16 bits)) occupy the first 4 contiguous bytes in memory starting with the first byte (bits 0 to 7) of the limit in the lowest memory location. The fifth contiguous byte stored is the base (bits 16 to 23) and in order till the last byte being the base (bits 24 to 31). We can see from the descriptor contents that the base and limit are present somewhere in the descriptor, though not in contiguous bytes. But one thing that we find is that there are only 20 bits for representing the segment limit. The G bit in the descriptor explains this. If the G bit is set to 1, then the 20 bit limit is assumed to be page granular, ie the limit denotes the number of pages used by the segment. That explains for the missing 12 bits which denotes the size of a 386 page. Now if the G bit is 0, the limit is simply the number of bytes used by the segment (relative to its base, of course). This is called byte granularity. So we know how to find the base address and limit of a segment from its descriptor. Now let us see what the S bit stands for. The S bit denotes whether the descriptor describes a system segment (S = 1) or a code/data segment (S = 0). The difference between system and non system segments will be explained in a short while. If our segment contains code or data/stack (stack is also data segment) then we should set S to 0. If S = 0, then the sixth byte has a lot of fields to denote tings like read/write permissions for the segment, privilege level of the segment etc. Refer the Intel manuals for exact details. Now let us discuss the format of the 'structure' that we mentioned in association with the LGDT instruction. Before using the LGDT instruction, we should have prepared a table somewhere in memory, containing descriptors of the format that we discussed above. Now the LGDT instruction needs to know the base address of this table and the limit of that table . The limit is specified as 16 bits and is byte granular. The base address is as usual, 32 bits. The format is that starting from the lowest address, the

limit should be stored followed by the base address, in the usual byte order of x86. Now the LGDT instruction is given the starting address of the location in memory where this structure is stored (in the format mentioned above). The LIDT instruction is also exactly similar to this. In the former sections, we had mentioned the LLDT instruction to be similar to the LGDT instruction. But that is not exactly true, there is a major difference between the two. As with a GDT, for the LDT also, we have to create a table somewhere in memory, containing descriptors of the format discussed in the beginning of this section. But the difference arises due to the restriction that the base address and limit information of the LDT has to be stored as a descriptor in the GDT!. This is where we get a chance to give an example of a system descriptor. After constructing the LDT, store the base address and limit of the LDT in the format of a descriptor in the GDT and set the S bit of this descriptor to 1 to denote that it is a special form of descriptor, not directly related to any code or data. Now find out the selector corresponding to that particular entry in the GDT. It is this selector that should be given as an operand to the LLDT instruction. So it is evident that the GDT should be present and active (by using the LGDT instruction) before the LLDT instruction can be specified. From the discussion above, it is clear that we can create any number of GDTs and LDTs in memory, but at a time, only one GDT and one LDT can be active. We can switch between GDTs and LDTs using the LGDT/LLDT instruction. Also, we can see that by specifying a limit for each segment, we enforce protection. Whenever the address used as offset with a particular selector exceeds the limit of the segment represented by the selector, the 386 generates a fault. For further information on LDTs and GDTs, refer the Intel manuals.

3.1.2. Paging in 386

The complete addressing mechanism in the 386 consists of the segmentation unit plus the paging unit. We had mentioned about how segmentation is done in 386. Now let us move on to paging. Actually, paging is not a must in 386. It can be enabled/disabled via a bit in the cr0 register. The segmentation unit produces an address by adding the offset to the base address of the segment using the techniques we mentioned in the previous structures. If the paging is disabled, then the address that the 386 sends on its address bus (and thus to the memory chips) is the address produced by the segmentation units. But if paging is enabled, the address produced by the segmentation unit goes through the paging unit and the paging unit and it is the output of the paging unit that goes onto the address bus. The address from the paging unit (its output) may or may not be the same as the address from the segmentation unit (the input to the paging unit). So that was quite a dry and uninteresting explanation about paging. Also, the idea about paging and its necessity will not be clear until we read a working code that employs paging. So based on what we have read in the 0.01 code, let us explain about paging and its power briefly.

3.1.2.1. Using full 32 bits for addresses.

We know that the address bus in 386 is of 32 bit width. So we can attach maximum of 2^{32} = 4Gb of physical memory to the 386. But we all know that normally, we won't have that much of memory in our PCs. Suppose we have only 16 Mb of memory in our PC. So that means that whatever address that comes on the address bus of the 386, it has to be within the 16 Mb range. If it is exceeded then (what happens ? we don't know :-( ) the data we may get will be meaningless. So what about the nice 4Gb address range ? It almost always remains un-utilized, that is, no program ever uses addresses above 16Mb (or whatever is the available memory in our machine). This is the situation when there is no paging, that is the paging unit of 386 is disabled. Now what happens when paging is enabled ? Again our discussion will

finally lead to a sort of simple mapping table that we discussed in segmentation. Assume that the available range of memory is divided (logically) into equal sized units of memory called pages. The size that the 386 assumes for a page is 4Kb, ie 12 bits of address. Our intention is to be able to use any address that can be generated using 32 bits (ie Max 4Gb) in our programs, that is a base address for our segments anywhere in the 4Gb range, but somehow to translate those addresses to lie within the available memory that we have before the addresses are sent on to the address bus. Again the most simple and natural way of achieving this is to use an address map table. Again what is being mapped to what ? The address that we produce in our programs (can be called a virtual address, because that may not correspond to any data in the memory chips) are being mapped to the available range of addresses in our machine (can be called the physical address, because it SHOULD correspond to some location in the memory chips). But there is one slight difference from mapping descriptors. There we mapped each used descriptor to a particular base address. But here we will not map all addresses that we use to physical address. What we do is we divide our used address range into pages of the same size as we mentioned before (4Kb) and then map each of these pages to some page in the physical memory pages. The example below will simplify the concept.

3.1.2.1.1. Example.

Since we have 16Mb of memory, we divide it into pages of 4Kb. So the starting addresses of our physical pages will be 0x0, 0x1000, 0x2000, 0x3000,...., 0xfff000. That is we have 0xfff = 4K pages of 4 Kilo byte size each. Now suppose the virtual address that we use in our program ranges from 0xf0000000 to 0xf00fefff (it may be that the one and only segment in our program has base address 0xf0000000 and offsets upto 0x000fefff). Again, we divide this range also to 4Kb size address ranges. These addresses denoting the starting address of the corresponding page Will be 0xf0000000, 0xf00001000, 0xf00002000,...., 0xf00fe000. The biggest used address - 0xf00fefff will be the last address of the last page - 0xf00fe000. So we have used 0xfe = 254 pages = 1016 Kb in our program, ie slightly less than 1 Mb. Now suppose that this program is getting loaded into a contiguous area of physical memory with the starting physical address equal to 0x5000. So this means that the addresss 0xf0000000 has to correspond to 0x5000, the addresses from 0xf0000000 to 0xf0000fff will correspond to 0x5000 to 0x5fff. Again 0xf0001000 has to correspond to 0x6000 and so on. So this is what we keep in the map table. The map table will contain the starting address of the physical page that we have decided to map to our virtual page. So if we declare a C array to map the 254 virtual pages to 254 physical pages, it will be like 'int MAP_PAGES[254]' where MAP_PAGES[0] will contain 0x5000 corresponding to our virtual page 0xf0000000, MAP_PAGES[1] will contain 0x6000 corresponding to our virtual page 0xf0001000 and so on. So when the paging unit gets a virtual address from the segmentation unit, it checks the address to find out to which segment the address belongs. This can be done easily by masking the last 12 bits of the address. For example if the paging unit gets an address 0xf0023a69, then the paging unit knows that this address is in the virtual page whose starting address is 0xf0023000. So it checks the map table at index 0x23000/0x1000 (each index represents memory of size 0x1000 and so address 0x23000 is index 0x23000/0x1000) = 35. At index 35, the physical address we have stored is 0x5000 + 35 * 0x1000 = 0x28000 - this might not necessarily be true. Since for simplicity, we said that our physical address range in this particular example is contiguous, this calculation is true. Otherwise, we can store any physical address (multiple of 4K) at index 35 and the virtual address will correspond to that physical address. Now the actual physical address corresponding to 0xf0023a69 is 0x28000 + 0xa69 = 0x28a69. This is how the paging unit produces a physical address from a virtual address. Note that the memory used by our program has to be a multiple of one page size (4Kb), we cannot map virtual addresses that are not on page boundaries to a physical address on the page boundary. This introduces a small waste of memory which the classical OS textbooks

give nice names like internal (or external ??) fragmentation. As with segmentation, here too we have to explain how the 386 knows where the map table is present and what the format of an entry in the map table is etc. This will be explained in a later section. But before that, the explanation that we have provided above does not correspond exactly to the paging mechanism of the 3886. A small difference is that the 386 uses two levels of map tables (note the term 'two levels' and not 'two pages' as in segmentation). This is explained in the next section.

3.1.2.2. Two levels of paging in 386

Now let us see how exactly is paging done in the 386. The basic idea is as explained above. When the user program generates an address, the 386 consults a mapping table entry corresponding to that address. If only a single level of paging was used, the address in that entry would give us the physical address of the actual (physical) page in memory corresponding to our virtual address. But in 386, the address that we get from the table is not the physical address of the actual memory page corresponding to the virtual address, but the address we get from the table is the physical address of yet another table. Note that the address that we get corresponding to this second table is the PHYSICAL address of the start of the table, ie that address need not be paged again. Now to get the actual physical address corresponding to the virtual address that was generated by our program, we have to index again into this second table. The entry in this second table that we index into will contain the actual physical address corresponding to our virtual address. Now how the two indices are generated is explained briefly below. Think more about it and make the idea clear.

The virtual address that our program generates will be of 32 bit length. If we were to use only one level of table, we could have got the index by using the bits 12 to 31. That is the index would be (virtual address $>>$ 12). Now the total number of bits available for indexing is 20 (32 - 12). In the two level table scheme in 386, we use bits 22 to 31 (ie the leftmost 10 bits) to index into the first table and the bits 12 to 21 to index into the second table. We get the address of the second table from the first table entry corresponding to our index (bits 22 to 31). Now the physical page address of our virtual address will be present in the entry of the second table corresponding to the indexing bits 21 to 21. Now let us note that using 10 bits to index into each table will give us a total of 1024 entries in each table. Also each entry is of size 4 bytes. So the size of one table is 4Kb, which again is on a page boundary!. Now do we need all the 4 bytes in an entry of a table ?. No we don't. The addresses that we get from both the level of indexing are page aligned. The entry in the first table gives us the physical PAGE address of the second table, so it uses only 20 bits, again the second page table entry also gives us the physical PAGE address corresponding to our virtual address. So that is also using only 20 bits. So why the extra 12 bits ?. Well their purpose will be explained in the next section. Also the next section will explain how the 386 knows where the programmer has kept the tables and things like that.

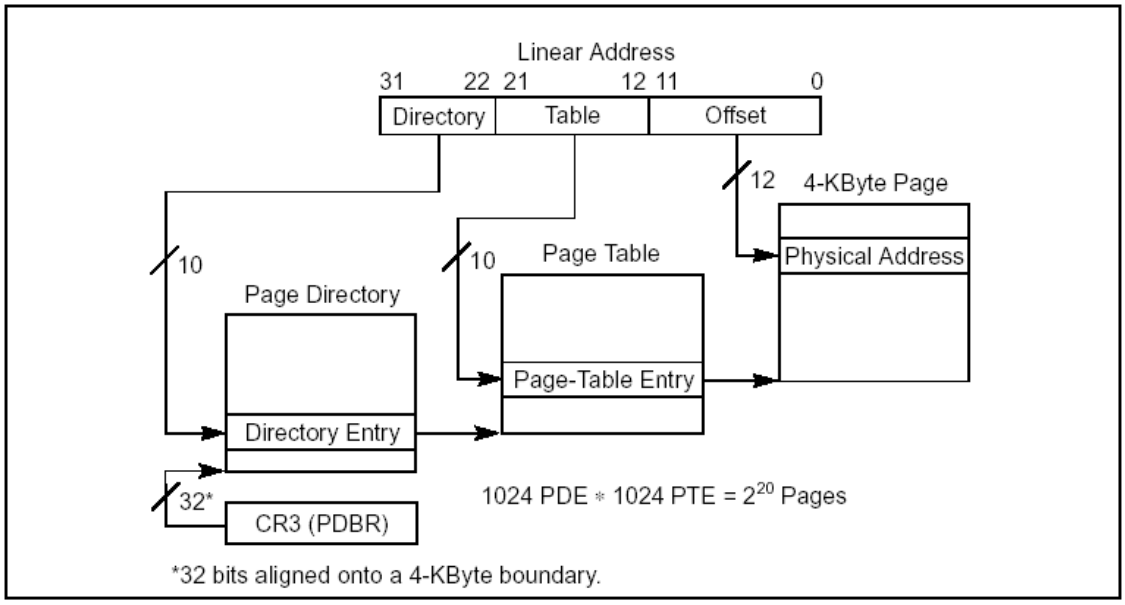

Figure 3-12. Linear Address Translation (4-KByte Pages)

3.1.2.3. Page table entry Format and more..

We mentioned about two levels of indexing in the 386 model of paging. The table used for the first level of indexing is called the page directory. As mentioned previously, it is one page in size and its address is stored in the cr3 register of the 386. The entries in the page directory are the addresses of the tables used in the second level of indexing and those tables are called the page tables. So at any instant of time there can be only one page directory and at most 1024 page tables (each page directory entry uses 4 bytes). The method used for indexing into both these tables are the same as that explained in the previous section. Paging is turned on/off by setting/resetting bit 31 of the cr0 register. Now let us see what is the format of an entry in the page directory or the page table.

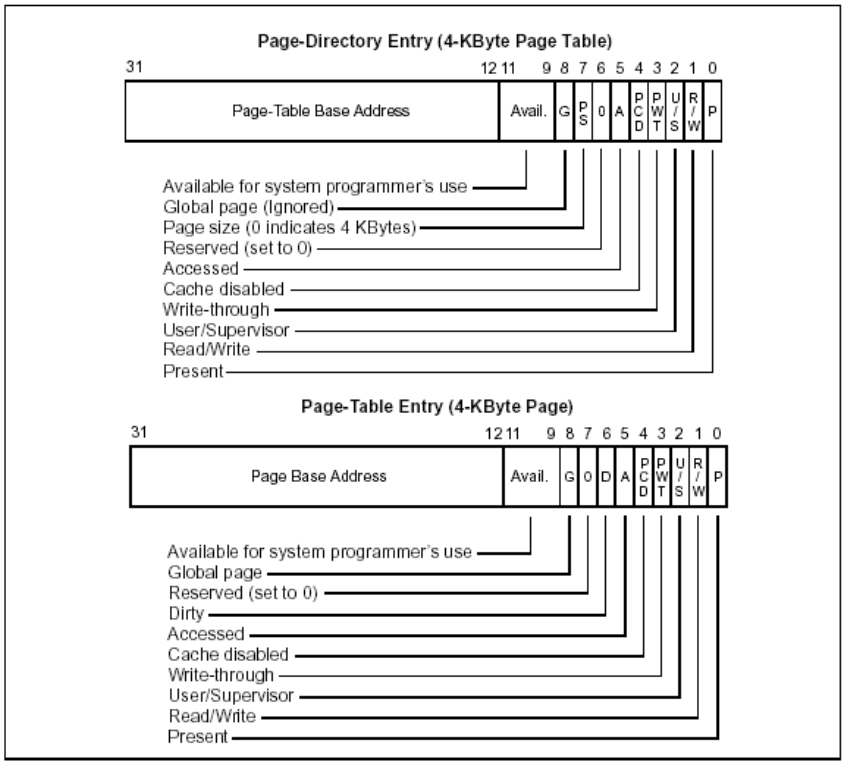

Figure 3-14. Format of Page-Directory and Page-Table Entries for 4-KByte Pages and 32-Bit Physical Addresses

The detailed explanation can be had from the intel manuals. Let us just see what the purpose of bits 0, 1 and 2 is. The bit 0 of a page directory entry is 0, it means that the corresponding page table is not present in memory and so the rest of the 31 bits can be used for storing other information like where on disk the page table is present. Similarly, if the bit 0 is 0 in the page table entry, it means that the corresponding page denoted by the virtual address is not present in memory and the other 31 bits can be used for any purpose - like giving details of where on disk the page is present. The bits 1 and 2 control the page level protection features. The bits 1 and 2 of a page directory entry control the protection features of all pages mapped by the page table corresponding to that page directory entry. The bits 1 and 2 in a page table entry control the protection features of the single page mapped by the corresponding page table entry. Now these bits are very useful for implementing swapping, demand loading, page sharing etc. If the bit 0 (P bit) of a page table entry is 0 and a use program tries to access that page, then the 386 automatically generates a 'page fault' which can be 'handled' by a page fault function to implement demand loading, swapping etc. More about faults and handlers in the next section. Similarly, page sharing can be implemented by setting a page to be read only and duplicating the page only when there is a fault on that page for a write onto that page. More about that in the kernel memory manager commentary.

From the above discussion, it is evident that each process can have its own separate page directories and page tables and so different processes can share the same virtual address by using different page directory/tables for different processes. It is not done in the kernel 0.01. All processes have the same page directory.

3.1.3. Interrupts and Exceptions.

Now let us move on to the idea of interrupts and exceptions in 386. Again, this topic alone can encompass a whole book, so we discuss only those parts that are essential to understand the 0.01 kernel. The working of interrupts in 386 is not much different from that in the 8086. The external devices are connected to an interrupt controller - the 8259 PIC or the recent APIC. When an external device generates an interrupt, after priority considerations, mask check and a lot of other things (Linus has described the 8259 as a 'hairy' device), the PIC sends an interrupt signal on the interrupt pin of the 386. Following this, the 8259 also sends one byte on the data bus of the 386 to help the 386 to identify which device caused the interrupt. In 8086, an address is generated from this byte (a simple multiple of the byte) and the 8086 transfers control to that address, just like a procedure call to that address. The things that happen after that is called an 'Interrupt Service Routine' or by any of the many names found in text books. So how is the working different in 386? Read on.

3.1.3.1. The IDT

The number of tables in 386 doesn't seem to end, is it not? IDT stands for the Interrupt Descriptor Table. Well as with the other tables we have seen so far, the entries in this table also contain addresses - but representing what ? The addresses in the IDT are the addresses of the Interrupt Service routines, the Exception Handlers etc. We will briefly explain about exceptions later. When the PIC transfers a byte to the 386, it uses that byte to index into the IDT. Then it executes the procedure whose address it finds in the IDT entry corresponding to the PIC byte. So simple!. The last instruction of this procedure will be an IRET and not a RET as with usual procedures. So before the kernel becomes fully functional, it has to set the various entries in the IDT with the addresses of the routines in the kernel to handle the corresponding hardware interrupts like hard disk, serial port, key board, timer etc.

3.1.3.2. Exceptions

As we saw in the sections on segmentation and paging, we can set various bits in the descriptors to indicate whether the segment is read only, whether a page is present or not, whether a segment contains code or data etc and umpteen more bits to indicate various other things. Now what is the use of setting these bits ? Suppose that we have specified a segment as read only. What happens if we write to that segment ? The 386 generates a FAULT. Each fault has an identifier byte like the interrupts. Again, 386 executes the procedure present in the IDT entry corresponding to the fault that was generated. The first 32 entries in the IDT are reserved by Intel for the exceptions and faults. That means that we should not put the addresses of our Interrupt Handlers in the first 32 entries. If we do so, then our hard disk handler might get executed on a division by zero exception !. The exception handler procedure might want to know what exactly was the reason for the exception or which particular instruction cause the exception etc. To facilitate this, 386 automatically pushes onto the stack a few bytes of information regarding the exception, before calling the handler. So the handler can get the needed information from the stack. Now how are exceptions helpful to the programmer ? Does it always indicate some error that is not expected ? Not always. For example, if we are intending to implement demand loading in our Operating System (which is introduced in the kernel 0.11) then we might load just a few pages of the program into the memory. then he/she might set the page table entries of the program to denote that the rest of the pages are not currently present in memory. So when the program tries to access a page not present in memory, 386 generates a page fault. This is where the handler comes handy. We can write a page fault handler that finds out which page is not present and loads it from the disk to the

corresponding location in memory. So that gives us the additional power of demand loading to the OS. This is how exceptions are treated as a positive feature rather than as an error condition to be panicked at. Refer the Intel manuals for more details on exceptions, the various types of exceptions - faults, traps, aborts etc. Again, what is the size of an entry of the IDT ? You guessed it this time - it is 8 bytes. So the next section will be a brief description of the IDT descriptor format.

3.1.3.3. IDT Descriptor Format

As with GDT, the IDT is also a table kept in memory whose entries are 8 byte descriptors. The format of the LIDT instruction is also the same as that of the LGDT instruction. Also the descriptor stored in an IDT is a system descriptor which can be of type task gates, interrupt gates or trap gates only. We have not mentioned at all about 'Gates' in 386. The idea of using a gate is to provide access to different 'parts' of the same section of code. For example, inside the kernel, we have written code for handling different interrupts like keyboard interrupt, harddisk interrupt etc. Instead of treating each of these procedures as a separate segment with base and limit, we would like to treat the collection of all these routines as a single segment with base and limit, and the access to these individual routines is specified as the base of the one encompassing segment plus an offset into the segment where we can find a particular procedure of our choice. The format of a gate descriptor in 386 is shown below.

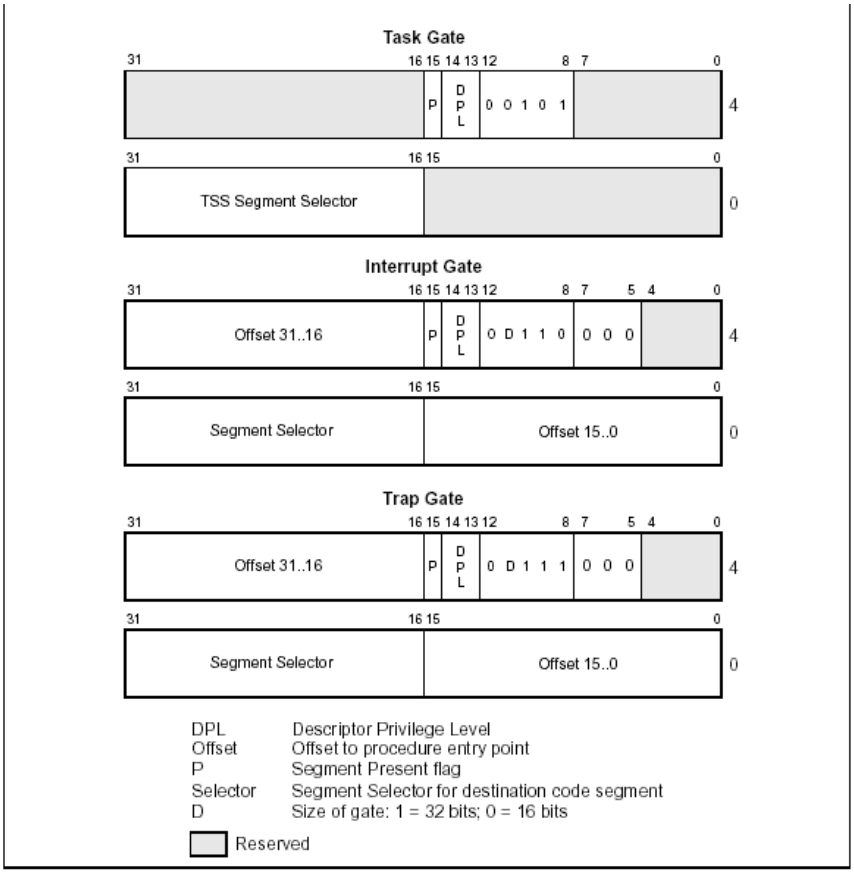

Figure 5-2. IDT Gate Descriptors

For handling all sorts of exceptions and interrupts, the 0.01 uses trap gates. The difference between trap gates and interrupt gates is that the interrupt gates disable interrupts by resetting the IE flag in the flag register whereas the trap gate does not. The only part of the descriptor that we need to know about is the selector field (we will mention briefly about the DPL field in the section on privilege levels). The selector field of the gate descriptor (the IDT entry) specifies another descriptor in the GDT/LDT which contains the base address and limit information of the one single segment which contains a collection of routines (interrupt/exception handlers). The offset field of the IDT entry specifies which specific procedure in the only single segment of handlers do we want to access using the particular entry in the IDT. Thus different IDT entries can have the same values of the selector and different offsets to denote different functions. For example, if the code for handling all the usual PC interrupts are collected as one single segment with say a selector or the segment as 0x60 (ie in the GDT), with offset for the timer interrupt procedure as 0x200, for the harddisk interrupt as 0x3040, for the keyboard interrupt as 0x6543 etc. Also let the timer interrupt use the 33rd slot in the IDT and the keyboard interrupt, the 34th slot. Then the 33rd entry (0x20th entry) of the IDT will have 0x60 in its selector field and 0x200 in its offset field. Similarly the 34th entry (0x21th entry) of the IDT will have 0x60 as its selector field and 0x3040 as its offset field. So when the timer interrupt comes, the 386 automatically looks up the GDT to find the segment corresponding to the selector 0x60, adds 0x200 to the base address of that segment and starts executing from that location. So this was a quick tour about interrupts and exceptions.

3.1.4. Tasks in 386

The 386 processor has been so designed as to give the Operating System as much hardware support as possible. The concept of tasks in 386 is one such. The implementation of multiple processes in UNIX can be done very easily using this feature of the 386. Suppose a particular process is under execution at the moment. Then a timer interrupt comes and the OS decides to schedule another process. But when the current process is rescheduled at a later stage, it should resume exactly from the point where it had stopped before getting scheduled out. So naturally, all the resources that the current process is using, like the registers, various tables etc should be saved before scheduling the new process. Also, these things should be restored before rescheduling the current process. Now, this saving of registers etc can be accomplished via routines in the kernel that save these things in some memory location belonging to the kernel. But if the time needed for executing the saving routine is reduced as far as possible, then 'task switching', ie scheduling a new process can be fat, thus providing better performance. Considering this factor, the 386 designers introduced the concept of a task so that saving and restoring of task related data can be done in hardware using instructions specially designed for that purpose. The following sections briefly describe how that is done.

3.1.4.1. The Task Table

The task table is again a contiguous area of memory that is used to save the information relating to the task that is currently executing. The information that 386 stores in this table automatically at the time of a task switch are the contents of ALL the registers of the 386 at the time of the task switch. Also stored in this table is a bit vector denoting which of the port numbers can be accessed by the owner of that task table, via I/O instructions. Well, this information doesn't change during task switch, because it is the programmer that stores this bit vector in the table. Also we can store any other task related information in this table, the only thing is that the saving and restoring of these information should be done by the code written by the programmer because the 386 does not know what we are storing or retrieving.

An example is the contents of the registers of the math co processor (if any) that we can store in the task table. This can be done very quickly by using specialized instructions for this purpose. The task table is called the Task State Structure(TSS) in the 386 literature. The detailed information regarding the TSS can be had from the Intel manuals.

3.1.4.2. Task switching

Each task must have an associated TSS. The details related to the TSS are similar to that of the LDT. As with LDT, the TSS also should be identified by a descriptor containing the base address and limit of the TSS. Also the type of the descriptor is either 0x9 or 0xB. The difference between the two types will be explained shortly. Again, as with the LDT, this descriptor should be present in the GDT. Also the current TSS in use can be found out via the Task State Segment Register (TR). The TR contains a selector corresponding to the GDT descriptor that identifies a TSS. So a TSS can be activated by using the LTR instruction by specifying the selector corresponding to the TSS that we want to make the current one. The LTR instruction does not cause any task switch, it just changes the memory area (ie the TSS) to which 386 stores data (register contents) when the next task switch occurs. So how do we switch tasks ? Task switching can be done via CALL, JMP, INT or IRET instruction. The CALL or JUMP instruction can be given the TSS selector or a task gate as the operand. Since we are not discussing task gates, we consider only task switch via TSS selector. When a CALL or an INT instruction is used to initiate a task switch, the following things are done

1. The new TSS (ie the one that gets activated after the task switch) will be marked busy. Here the two types for the TSS descriptor comes into view. A TSS that is not busy has type 0x9 and one that is busy has type 0xB. So initially, when we create a new TSS, its descriptor will be given a type of 0x9, since it is not busy initially.
2. The back link field of the new TSS is is set to the selector corresponding to the old TSS.
3. The NT bit of the new task gets set.

Now we are in a position to explain in more detail the role of the NT bit which we vaguely mentioned in the section on compiling the kernel 0.01. As we saw above, the NT bit gets set on a task switch via a CALL or an INT (hardware or software INT). Now the IRET instruction also has a double role. If we execute the IRET instruction, the 386 checks whether the NT flag is currently set. If it is set, then the IRET instruction is interpreted as a request to switch to the previous task. The selector of the previous TSS is store in the back link field of the current TSS. So the IRET instruction switches back to the task represented by that selector, thus moving one step back along a chain of nested tasks (moving one step back from the last task in the chain). But what happens in the following scenario ? The NT bit is currently set. Then an interrupt comes (external hardware interrupt or an exception). That interrupt is not intended to cause a task switch. So the handler corresponding to that interrupt is executed. At the end of the handler, there will always be an IRET instruction. And Lo!, the IRET instruction finds the NT flag set and causes a task switch. But that was not the intention of this particular IRET. Its intention was just to return from the handler. So how is this problem overcome? Well, interrupts that are not intended to cause a task switch always turns off the NT bit automatically. It can be restored after the IRET of the handler because the original flag register will get popped with the IRET instruction of the handler. So another IRET intended to cause a task switch can effect a task switch if the NT flag is set.

Phew :-(, that was a rather complicated process. Luckily, the kernel 0.01 does not cause those complications. There is no need for a chain of tasks in 0.01 and so there is no need to 'return'

to the previous task in the chain via an IRET instruction and so there is no reason why we should be cared about the NT bit. We (0.01) would prefer that the NT bit is always off. Don't misunderstand that we are using only one task in the 0.01 and that is why we don't want a chain of tasks, NO. If we keep a chain of tasks, then the problem is that we cannot switch to an arbitrary task in the chain, we will have to follow the chain of tasks backward. But for scheduling processes, we would like to switch arbitrarily between processes and their corresponding tasks. So a chain of tasks implementation is not suited for process scheduling. So we always switch between tasks independently without making one process the back link of another and marking the new one busy and so on. So what do we do? Well, there is another way of switching tasks that avoid all these nuances. That is by using the JMP instruction with the selector of a TSS as its operand. In this mode of task switching, there are no chains and busy markings, just store the contents of the registers to the current TSS and load the registers with the contents of the new TSS and load the TR with the selector of the new TSS, thus making the new TSS active - that is all. This is the method of task switching that is used in 0.01. So now, the concept of task switching should be clear and that takes us very close to the technique used in scheduling processes. Refer the manuals for any further details that you would like to know.

3.1.5. Privilege Levels And The Stack

Till now, we have been extremely unfair on two bits that popped up every now and then in segment selectors and descriptors. Well, these two bits are used for conveying privilege level information. The 386 has four privilege levels (and so two bits). The privilege level 0 (00 binary) is the most privileged level and the privilege level 3 (11 binary) is the least privileged level. In the most privileged level, we can execute ALL instructions of the 386 and access all memory areas, execute code in any segment etc. But in the other privilege levels, we cant execute a lot of instructions like the LGDT, LLDT, LTR, I/O instructions etc. Suppose that the kernel executes in privilege level 0 (because it has to be able to do anything) and the rest of the programs execute in privilege level 3. Also suppose that we could access the GDT and execute the LLDT instruction from this privilege level (3). Then what we can create a new LDT somewhere in the memory allocated to us and we can store in that LDT descriptors that will help us to access the memory of other programs (by knowing their base address). Then we can make an entry in the GDT corresponding to this new LDT and then execute the LLDT instruction. Well, after that we can simply wreak havoc. So naturally, 386 prevents privilege levels other than 0 from doing such mischiefs. So we cannot access the memory of other programs unless there is some bug in the kernel code. So that is one form of protection. The privilege level that the 386 is in currently (called the Current Privilege Level, CPL) is equal to the privilege level of the code segment that it is executing presently. Now privilege related rules are not as simple as it is stated above. It is extremely complicated and just by reading the various rules, we will never come to understand their purpose or their requirement. Only when we come across a need for a particular protection feature while programming, can we understand perfectly the rules about privilege given in the manuals.

3.1.5.1. Privilege Levels Used In 0.01

The 0.01 uses only two levels of privilege - 0 and 3, ie the most privileged and the least privileged levels. Also since the 0.01 does not make use of any complex facilities provided by the 386, the privilege rules that need to be kept in mind while reading the 0.01 kernel are very simple (thank God). We shall briefly state how the various privilege levels are used in the 0.01. We come across privilege levels in segment selectors (the bits 0 and 1) and in segment

descriptors. The general rule use in the 0.01 is that for all the selectors and descriptors related to the kernel, make the two bits 00, ie a privilege level of 0. For all other programs (user programs) make the two bits of their selectors and descriptors equal to 11, ie a privilege level of 3.

Now as examples, the kernel related things are - the code, data and stack segment descriptors of the kernel code which are kept in the GDT, the code, data and stack segment selectors used in the kernel code, the LDT descriptor kept in the GDT, the LDT selector used to refer to an LDT, the TSS descriptor kept in GDT, the TSS selector used to refer to a TSS, the descriptors present in the IDT, the descriptors present in the GDT - wherever the privilege bits are present in these items, make those bits 00. Thus all these kernel related things have the highest privilege. This means that the data items among those mentioned in the list above cannot be accessed by any code that is not in privilege level 0 and that the code items among those mentioned above when executed will make the CPL equal to 0 and thus those code items alone can access the data items at privilege level 0.

The items that should be kept at privilege level 3 are only a few in number. They are - the descriptors in the LDT and their corresponding selectors. In 0.01, the descriptors in the LDT denote the code, data and stack segments of the user program corresponding to that LDT. So naturally, they should be at privilege level 3. Also the selectors that are use in user level programs to correspond to the user code, data and stack (whose descriptors are in the LDT) should also have the two bits set to 11 corresponding to privilege level 3. So in brief, this means that nothing in the kernel space can be accessed by the user space programs. As mentioned above, the kernel keeps its code, data and stack segment descriptors in the GDT. If the user program uses a selector to index into the GDT (which it can easily do by setting the Table Indicator bit of the selector, bit 2 to 0), then the 386 processor generates a 'Protection' fault because the user program is at privilege level 3 and the descriptors in the GDT are at level 0.

3.1.5.2. Stack Change With Privilege Change

If we examine the fields in the TSS, we can find that there are three pairs of entires ss0:esp0, ss1:esp1 and ss2:esp2. Also there is the entry for the ss:esp registers also, along with the other registers of the 386. So totally, there are four entries for the stack register pairs. The reader can guess that the number of pairs (four) has some relation to the number of privilege levels. It is exactly so. The four pairs of the stack registers are the values of the stack segment register and the stack pointer register in the corresponding privilege levels. Well, to make things clear, this means that for the SAME task, the stack registers have different values in different privilege levels. This means that the SAME task has different stacks in different privilege levels. This arrangement is needed for protection related reasons. So if a particular task is using different privilege levels, then it should allocate space in memory for different stacks for each level. It can use just a single stack only and use the same pair of stack registers for all the levels, but that compromises on protection. So initially, ie when a new task is getting ready to run for the first time, the TSS of that task should contain the segment base and offset of the stacks that it intends to use in different levels in the stack registers corresponding to that level. After the task starts running, when it switches to a higher privilege level, the old values of the ss and esp registers (ie those of the lower privilege level) get pushed on to the new stack before the task starts executing in the higher privilege level. The new stack base and offset is got from the TSS entries corresponding to the new privilege level, and these values are loaded into the ss:esp registers. When the task RETurns (or IRETurns) to the lower privilege level, the stack registers in the lower privilege level are restored from the values stored in the higher privilege level stack when we switched to a higher privilege level. Also, there are a lot of other things

like copying of parameters from one stack to the other and so on when we use call gates. For our purpose of understanding the 0.01 kernel, we need not know all those details. So that was all we wanted to know about the relationship between stacks and privilege levels.

With this, we end our discussion on the 386 architecture. Almost all the basic ideas about the 386 architecture needed for understanding the 0.01 kernel have been presented in this section. The good thing about this section is that we have presented things in the order in which the reader can understand them. This may not be possible if the reader starts reading the manuals directly. Now the reader can supplement his/her knowledge by referring the manuals. Also, be careful not to spend too much time trying to understand the architecture, because reading about the architecture alone can't produce a clear idea. You have to see some code where the architecture details are made use of. Also do not spend much time on architectural features that are not used in 0.01. The only features used are the ones that we spoke about in this section. So now, let us move on to our final aim - the kernel 0.01 code!! Hold your breath and tighten your seat belts :-).

Chapter 4. The Big Picture

4.1. Step Wise Refinement Of The 0.01 Kernel - Step 1

We will use the 'method of step wise refinement' (by Niklaus Wirth) for explaining the kernel code. First we will describe the picture as a whole, then we will break it up into smaller and smaller parts until we feel that we do know what each line of code is meant for. So this book is not just an expanded set of comments for each procedure in each of the files in the source tree, but it also talks about how those procedures work together. We hope that the granularity that we have selected in explaining the code is just right. Well, our knowledge on different topics in the 0.01 code may vary and as a result, the explanation for the various topics may not be all of the same flavor. The hardware details have been completely avoided, just an overall idea of each hardware device is provided. But it is good if the reader makes himself acquainted with the full source code for each device in the kernel 0.01. Refer for detailed hardware description. So let us move on to the commentary. Also, we assume that the reader is using the source distribution that we have provided. Well, as we mentioned before, the changes we have made are meager and of the grade of a few simple syntax changes.

4.1.1. The Source Tree

The entire kernel code consists mainly of three components - the file system, the kernel (composing of scheduler, system calls, signals etc) and the memory manager. These components are present in the directories linux-0.01/fs, linux-0.01/kernel and linux-0.01/mm respectively. The linux-0.01/init contains a single file 'main.c' that gets the kernel working. The linux-0.01/boot directory contains files (two in number) that contains a boot loader for the kernel (boot.s) and the initialization parts of the kernel in the protected mode (head.s). The linux-0.01/tools directory contains files that build a kernel image from the raw binary of the kernel that we get after a 'make'. The linux-0.01/include directory contains several files and sub directories. The contents of each of these files and subdirectories will be explained as and when needed. And finally, there is the linux-0.01/lib directory. It contains code for the system calls that are to be called from user level programs. It is not needed to make the kernel image. But we have to use the object code in that directory when we write a shell or any other utility that uses system calls. So this is how the kernel 0.01 code is organized.

4.1.2. The Makefiles

A few words about the make files. The fs, kernel and mm directories have separate make files. These make files result in the production of relocatable object files fs.o, kernel.o and mm.o respectively. This is combined with main.o and head.o to get the actual kernel in the raw binary format - the file linux-0.01/tools/system. Now the boot sector is got by assembling and linking the boot.s file to get a raw binary file (8086 code) called 'boot'. This boot is acted upon by the 'build' binary (generated from the linux-0.01/tools/build.c) to extend the size of 'boot' to 512 bytes and add the necessary bytes to indicate that it is a boot sector image. This boot sector is combined with the kernel to generate the file 'Image' in the linux-0.01/ directory. This image can be copied on to a floppy in the raw form (using 'dd', rawrite etc). The reader may often need to refer the Makefiles to understand which files are needed for what and so on.

4.1.3. The Big Picture

Now let us see how exactly various pieces of code are interrelated in making the kernel work. Only the relevant parts have been explained, the other things like printing messages onto the screen etc. have been simply ignored. Again, this section is a brief overview, so don't be worried about the exact details of the actual code at this stage. We have an image of the kernel in your floppy, we insert the floppy in our drive and turn on our machine. Now what ?

4.1.3.1. The Boot Sector

The BIOS cares only about the contents of the first sector (512 bytes) of your floppy. If the bytes 511 and 512 have the values 0x55 and 0xaa respectively, the BIOS tells itself that the first 510 bytes contain some executable code. If the two flags are not present, the BIOS reasons that your floppy is not a 'boot floppy' and so it looks for the next bootable device (may be a harddisk or a CD Rom). Now if your floppy has been found bootable by the BIOS, it simply loads the first 512 bytes (one sector) into a fixed location in the memory, namely location 0x7c00 and starts executing from that location onwards (by making the 16 *cs + ip equal to 0x7c00). So whatever we put in the first sector of the floppy (after setting the flags 0x55 and 0xaa), it gets executed. So what do we put in the boot sector when we create the 0.01 image ? Well, the code that we put in the boot sector in the 0.01 image is generated from the linux-0.01/boot/boot.s file. The explanation of what it does follows. The moment it starts executing its first instruction, it is in location 0x7c00. But it somewhat dislikes this location and moves (copies) itself to location 0x90000. Well, we don't know why, but we tried modifying the same code so that it would not move itself to 0x90000, but it would continue running from 0x7c00 and accomplish all the other things exactly as it would have done, had it been moved to 0x90000 - :0(our code didn't work. We didn't try experimenting too much with that and made ourself satisfied with the 0x90000 move!. After moving to 0x90000, it starts running the rest of the code, that is, the code present after that needed for the copying operation. Of course, if that code was to be executed again, it would not execute anything else :-). Now we had appended the raw binary form of our kernel to the bootsector. So starting from the second sector onwards in our floppy, we have the binary of our kernel. The boot sector code reads the entire kernel (it knows how large the kernel is)into the memory location starting at 0x10000 using BIOS interrupts to read the floppy. Then it moves this kernel image to memory location starting from 0x0. Hey, why the trouble ? Why don't we directly load the kernel image to 0x0 ? Well, the answer is that the interrupt vector table is present starting from 0x0 and so if we overwrite it with the kernel image, what will happen to our poor floppy reading interrupt ? So the extra trouble is worth it. So far, our code has been executing in the real mode (16 bit mode, no protection ^_^). But for multitasking, protection etc etc.., we want to move to the protected mode. For that purpose, our boot sector code sets up a simple GDT and IDT and switches to protected mode. After this our bootsector code instructs the processor to start executing from location 0x0 - which contains the 32 bit code for our kernel, with the initial part of the code corresponding to the linux-0.01/boot/head.s file. And with that our poor bootsector code has completed its mission and is no longer needed.

4.1.3.2. Kernel Initialization - Phase I

This phase corresponds to the code in the linux-0.01/boot/head.s file. Starting from this phase, the 386 is working in protected mode. To initialize the protected mode, the real mode code had used a rudimentary GDT and IDT. So in this phase, the actual GDT is activated and a partial IDT (space for IDT allocated, but its entries are all 'inactive') is put up. We have to remember that the real mode code transfered control to the kernel, with the interrupts disabled (using

'cli' instruction). So there is no harm in leaving the IDT 'half baked'. Also the stack that the kernel will be using initially is also activated. Then the paging mechanism is initialized and activated. After this, control is transfered to the code in the linux-0.01/init/main.c file. Well, all the code upto this point was written in assembly language(8086 assembly in boot.s and 386 assembly in head.s). It is evident why it is written in assembly because we are dealing with machine dependent manipulations, which can't be done using any standard C constructs. Of course, if it was a matter of life and death if you don't use C, then you could have done it by using the inline assembly features of GCC (well, it is almost the same as writing pure assembly code). But still there are problems like stack not being initialized etc, because the C compiler assumes a stack to be present. So it is better written in assembly. Again, the boot.s binary code was intended to fit into the boot sector, ie 512 bytes. In 0.01, the boot.s binary is less than 512 bytes, so it can fit into the boot sector. But what if boot.s binary exceeds 512 bytes ? In newer kernels, it happens so. What is done is to break up boot.s into two files - boot.s and setup.s, both containing 8086 code. The boot.s goes into the boot sector. The setup.s comes next, followed by the kernel (starting from head.s). When the boot.s finishes its part, it transfers control to setup.s (whereas it transfers controls to head.s in 0.01) and finally, setup.s transfers control to head.s. The reason why boot.s increases in size is that it may contain code for reading several parameters from the BIOS, doing some tests etc, which are not done in 0.01. For example, the parameters like the type of the floppy and harddisk are hard coded into the kernel, which may have been read from the BIOS. Now let us move on to the second phase of initializations.

4.1.3.3. Kernel Initialization - Phase II

This phase corresponds to the code in the linux-0.01/init/main.c file. We had mentioned that we left the IDT incomplete in the previous phase. The main task of tis phase is to properly fill up the IDT entries with the handlers for various interrupts and exceptions. After that the interrupts are enabled (using 'cli' instruction). Now if an interrupt comes, it will be properly handled because we have installed the proper interrupt handlers in the IDT. With this, the kernel is completely functional and ready to provide services to the user programs ? But where does the user programs come from ? We know that in UNIX, any program is loaded using the exec system call. Usually, exec is used in conjunction with fork. For example, if we type a command at the shell prompt, the shell forks and the child shell execs the command, so that there is again the shell for typing other commands. So after initializing the interrupts, the kernel 'forks' itself to create a new copy. Also, the shell and other programs are residing in a 'file system' (the minix version 1 file system in our case). So the file system has to be identified, its parameters (like number of free blocks etc) have to be determined etc. So the copy of the kernel does this job of file system initialization. This copy of the kernel has pid 1 (the init process in UNIX) and its parent, the original kernel has pid 0 (the swapper). Now the init process generates a few other processes like a file system checker and the shell by the usual technique of fork-exec combination. So with this, we get a shell running in our machine. Now we can type commands at the shell prompt and the shell will fork-exec the command and so on. Thus the 0.01 has become fully operational. Remember that till the point where the first exec was executed by the init process, there was no need to load any file from disk to memory by the kernel, because the code that we were executing till then was loaded by the boot strap loader. Now what to explain next ? We had written about programs being run using the fork-exec pair, there being a file system in which our files are residing etc. But what happens when a system call like fork or exec is executed, or how does a new file get loaded into memory when the exec system call is executed ? These are the things that we are going to talk about further.

4.1.3.4. System Calls - The Interface To The Kernel

Now that we have got the kernel up and a process running (like the shell), what next ? Can the process do everything by itself ? By the term 'itself', we mean the user level functions inside the process. For example, the process can do things like finding the values of mathematical functions like sine, cosine etc by using its own functions. But if the process wants to write to a file, can it do that by using only its own functions ? Suppose that it could have done so. But if there are more than one processes executing and more than one process tries to write to the same file, then unpredictable things would happen. So these things have to be done by somebody who can take care of such problems. That is exactly the idea behind system calls. When we consider a server - client process (say the X Window server) then the server is a separate process and the clients are separate processes. But it is not correct to consider the kernel as a process - ie the kernel is not a separate entity. The kernel has meaning only when it is attached to some process. The kernel can be thought of as a set of functions and data that are common to all processes. The functions in the kernel are there for doing certain things for the processes, such things that affect other processes also and so require a co-ordination among all the processes(like the file write mentioned above). Now the kernel being a set of global data and routines, a particular process can know about the status of all the other processes via the data in the kernel. So using these kernel data, the processes can achieve coordination while doing things. Again, all this discussion was to ward off the idea that a kernel is one "running" entity and the user programs make use of the kernel by "requesting" it to do some service for the user program. That is NOT the idea. The kernel is a common set of routines and data that ANY process can access. Well, it might be somewhat demoralizing for us to think of the kernel as the "slave" of the processes rather than the "master" of all. But that is the truth. But suppose that a process has the freedom to access and modify any data in the kernel. Then a malicious process can modify data in the kernel that is related to the status of some other process in such a manner that the other process may be made to suffer in some manner (like being denied the right to write into a file, though it has the write permissions). So the kernel data should NOT be allowed to be modified (or even accessed) by user processes without some restrictions. Now how can we enforce restrictions ? Naturally, the best restriction is to keep the kernel data at a higher privilege level (0 in our case) and user programs at a lower privilege level (3 in our case) which is what we are doing in 0.01. So the user progress cannot access the kernel data using the user code. This is where system calls come into view. System calls are nothing but a few routines inside the kernel that does some service for a process such that the effect that the service has on the other processes can also be taken care of by manipulating the common data in the kernel. The privilege levels of the system call procedures are so set that they can modify the kernel data in any manner they want (not only the kernel data, they can even modify the user data). Also, the system calls are written very carefully so that the system call themselves are not making any mistakes while manipulating the kernel data. Thus if the user processes are allowed to call the system call procedures only, then we can be sure that no malicious process can do any mischief because the only way they can modify the kernel data is via system calls and the system calls themselves are coded to work perfectly. Now how does a user process call these system calls ? There can be many methods of implementing system calls. The common method is to use software interrupts. We know that the handler corresponding to an interrupt can be activated either by a hardware interrupt corresponding to the entry in the IDT or by an INT NN instruction where the NN corresponds to the entry in the IDT. So in Linux, we use the interrupt 0x80 for system calls. That is, we make the entry 0x80 in the IDT correspond to a particular function in the kernel (by suitably setting the IDT descriptor). When we use the INT 0x80 instruction (from the user program), then the function in the kernel corresponding to the IDT entry gets executed. But does that mean that Linux has just one system call ? No, this particular function is also given an argument (it can have other arguments also) via the 'ax' register to denote which

system call we want and depending on the argument, this particular function calls different functions in the kernel (something like a 'switch' on the argument). Well, it could have been that each system call (ie kernel procedure) was allocated a different entry in the IDT. But this would take up a large number of entries in the IDT and adding a new system call would also be cumbersome. So a system call is nothing but a usual function call. After the system call is executed it returns to the point that it was called, like a normal function (well, not always, more when we explain the actual code).

Now, the whole procedure in short. After the kernel is up (by the word 'up' we mean that the kernel code and data is in memory and not anything like the kernel is "running") and a process starts running, the process uses system calls to temporarily switch to a higher privilege level, it manipulates some common kernel data and returns back, again after some time, it uses a system call and this process continues. This is what happens after the system is booted and processes start running. The processes start running in the user mode, momentarily switches to kernel mode, back again to user mode, then to kernel mode and so on until the process completes its execution. Also we have to understand that when a process calls a system call and the system call modifies some kernel data, then it is effectively the process that has called the system call, that is modifying the kernel data. So let us wind up this section on system calls by highlighting the following fact - THE KERNEL IS NOT A SEPARATE ENTITY. IT CANNOT "RUN" INDEPENDENTLY. WHENEVER IT RUNS, IT IS ATTACHED TO SOME PROCESS, THAT IS IT NEVER RUNS WITHOUT UNLESS SOME PROCESS MAKES IT RUN VIA SYSTEM CALLS. It is not the system calls alone that makes the kernel run, but also interrupts and exceptions. That will be clear only when we see the actual code.

4.1.3.5. Multi Tasking And The Timer Interrupt

Ok guys...have you ever read a few adventures in "Sherlock Holmes" that were written by Watson ? This happened after Conon Doyle "killed" Sherlock Holmes in one adventure (We don't remember the name). You can find a distinct difference in the writing style of "Watson" and "Sherlock Holmes". Similarly, from now onwards, you can find a DISTINCT difference in writing style because one of the authors has moved out of college and is no longer a college student :-(. So may be you will find less of "college" style from now onwards (what is that style ?). Also, this work is being taken up after a long gap of more than 8 months. Also now as the authors are more involved in the MIPS family of CPUs and have lost touch with the x86 family, the assembly level description from here onwards might not be very detailed. But by God's grace, much of the machine architecture part of this work was completed before we moved on to MIPS, so that will not be a very big negative impact on this commentary. The MIPS architecture is so simple and beautiful that we are planning to port 0.01 to an embedded system based on R4k. May be after that work is over, we can write an "addendum" to this work describing the MIPS changes! So let us move ahead with Timer Interrupt. The Timer Interrupt is the "heart" of any preemptive multi tasking operating system. Well, what is a preemptive multi tasking Operating System - www.dictionary.com defines preemption as "Prior seizure of, appropriation of, or claim to something, such as property". Prior seizure is the exact description for preemption from the OS perspective - this means that we do not allow anybody (any process) to do his work for as long a time as it wants, we ask the process to temporarily stop doing its job and allow others to continue even if the process being suspended has not completely finished its work. So a preemptive multitasking OS is one in which many processes can run simultaneously (not exactly simultaneous, but..), but nobody is allowed to play the "king". So how does the timer help in that ? There is a timer chip/circuit in EVERY board that has serious intentions of running an OS (not even multitasking OS).

This chip/circuit can be programmed to emit signals periodically without the help of the processor. This signal is an interrupt source to the processor. If our memory is correct, the 8251 (or compatibles) is the timer used in Intel boards. It can of course, be programmed by the processor to control the “periodicity” of its period, various operating modes etc.. Also, IOMC (If Our Memory Is Correct :-), the timer interrupt is interrupt line 0 on the 8259 and has the highest priority - that means that whatever other interrupt is waiting, the 8259 first informs the x86 about the timer interrupt. Ok, lots said about the hardware part of the timer interrupt. So what does the processor/OS do on getting the timer interrupt ? As mentioned in the section on IDT, the timer interrupt causes the x86 to automatically switch to privileged mode and call the timer interrupt service routine. Now what does the timer ISR do ? Speaking in extremely simple terms, the timer interrupt will increment a counter corresponding to the current process which will indicate how long this process has been running, Then it will see whether the count has exceeded some default value which means that the current process has to be temporarily stopped and somebody else has to be selected to run. Now, how do we select “somebody else” to run ? Very simple - the kernel has ALL the data structures which describe the FULL details about a process when it switched from the unprivileged to the privileged mode - IOMC, this information is automatically stored by the x86 in the TSS. So to get somebody else running, all we have to do is to point the Task Register to the TSS of the new process that is going to run. Now, how long does this continue ? This will continue as long as you switch off your PC or you have to push the “big red button” because your system is hung. The kernel will have a list of data structures corresponding to each process which contains data like how long the process has been running etc. When we create a new process (fork()), then an extra element is added to the list, if we kill a process (exit()), then one member goes off the list. If there are no process to run, then the “idle task” will run. The “idle task” is a process (the only process) created by the kernel (the kernel can do all dirty tricks) just for this purpose - to run when nobody else is ready to run. So after the timer interrupt, if a new process is scheduled (not necessarily), then the TSS loaded will be that of the new process and it will resume from the state where it was suspended previously. So this makes it very clear how a “timer interrupt” can ensure fairness to all processes.

4.1.3.6. Other Interrupts

Well, apart from the timer interrupt, we have lots of other interrupts like keyboard interrupt (in the PC), harddisk interrupt etc...There is nothing much to be said about these from the hardware perspective. They just behave like timer interrupt - they also get into the kernel mode, start executing their own ISRs and go back to process level (may be a different process) after its execution is over. But the difference is that these other interrupts are not “periodic”, ghosh!, we don’t want a hard disk interrupt to be periodic!

4.1.3.7. TLB exceptions

The two items that we discussed above namely - system calls and timer interrupt are two of the most important in the general category of exceptions. These two will be present in almost all the advanced architectures of processors. Now let us move on to the other types of exceptions of which the TLB exception comes with the next level of importance. With the timer interrupt, the system call exception, the TLB exception and a few other interrupts for the various IO devices, we come to a conclusion on all that is needed for a processor that supports preemptive multitasking. So now let us get into more detail on the TLB exception, which is the “heart” of the “virtual memory” management in all modern OSes. In subsection 3.2, we had explained in detail about paging in 386. When the 386 finds that corresponding to an address, there is NO

"physical" page present, or if it finds that somebody is trying to write to a page that is "read only", then the 386 gets a page fault (TLB exception is a term used in MIPS :-). Then as usual, it switches to privilege mode, changes to kernel stack, blah..blah... and starts executing the page fault exception handler. Now the code in the exception handler is responsible for all the other "virtual memory" activities. For example, suppose we have a shortage of memory and so we did not allocate "physical pages" to the "whole" process when it was created (by fork()) - may be we allocated only one physical page just enough to run the first few instructions corresponding to the process (starting with main() ??). So when the program counter (IP) increments (or due to a jump instruction) and gets a value of address corresponding to which there is no "physical page" in memory (that is, the "going to be run" instruction in not in memory), what will it do - should the process crash ;-) because its instructions is not in memory ? No - it need not. So what happens is that the 386 gets a page fault (no page present) and it switches to kernel mode. Now the kernel has to do two things - it has to find a "physical" page to give to this process AND it has to load the code corresponding to the instructions that is not in memory, from a medium like hard disk (concept of swapping!!!). Now suppose it somehow gets a free page and also it loads the "missing" code into that page from the harddisk (the harddisk operations also require memory, but that is usually "reserved" by the kernel and not used by anybody else) - so now the process can again start executing untill it finds that another of its instructions is not in memory :-(But what happens when memory is so less that the kernel does not find a free page to give to the poor process!! ? Then the kernel does "swapping" - that is, it "steals" one page from some other process (the algorithm for which is a subject of HUGE research) and gives it to the poor process that needs it. But then what happens to the valuable data in the page that was "stolen" ? Won't the other process from which the page was stolen suffer because of this ? Yes, it will - so to prevent that, the kernel writes the data in that "stolen" page to a location in the hard disk (swap space) and marks the page table entry corresponding to the stolen page (in the process from which it was stolen) as "page not present". Now the story repeats - if this generous process from which the page was stolen needs to execute some code (or access data) in its stolen page, then it will also generate a page fault, then again the same story repeats (now the previously "generous" process becomes the "poor" process). If this stealing of pages and writing to the disks happen very frequently, then the system performance will HEAVILY degrade and this is called thrashing! So this is how the page faults help in implementing "virtual memory" - allocate as much address space as needed to various processes even though there are no "physical pages" corresponding to those addresses - the absence of "physical pages" will be taken care of by page faults + swapping etc...

Chapter 5. Flow of control through the Kernel

5.1. Step Wise Refinement Of The 0.01 Kernel - Step 2

In Step 1, we have seen a BIG picture of what is happening - all in one picture. Now let us make the picture a bit more smaller. Here we will not explain in as great detail as we did in Step 1. We will briefly mention what files contain what code and what is the code sequence followed during and after the 0.01 kernel boots up. So after this Step is completed, we will be familiar with the flow of code from the “file name” level - that is, we will be able to say - Ok, when an interrupt comes, the code in this file is executed, after that it goes to the code in this file etc... In Step 3, we will be explaining the code in each file so that we will be able to trace the code path by “function names” rather than “file names” :-)

5.1.1. Normal Activities In a Running OS

What does the OS do after booting up ? The main activities are servicing interrupts and exceptions - via which all the other abstract activities take place. For example, after the OS is up and we get a shell running, the user may type some command corresponding to which the keyboard generates interrupts. The shell reads those characters, interprets them as a command and if necessary, makes a system call (like fork or exec) which is also one form of exception. Or when a process is running, it might get a page fault and service it. Or it may get a timer interrupt as a result of which a new process might get scheduled. So again, to sum up, the main activities are processing interrupts and exceptions.

5.1.2. linux/boot Directory

5.1.2.1. Boot Up - boot/boot.s

We compile the 0.01 kernel and get an image which is ready to run from address 0x00000000 (or any other address - it is the choice of the programmer). But who will get this code loaded into memory from address 0x0 (or whatever is the base address) ? Here is where the boot ROM comes into picture. The 386 processor when powered up, starts executing by default from a fixed address (We think 0xfffffff0, verify from the x86 manuals :-), and again, there will be just one jump instruction (hard wired) at that location to the starting address of the ROM. Now the ROM starting address is fixed in all the IBM PC family of machines (which again is somewhere around 320K to 640K if we remember right). Now starts the actual boot ROM code. The boot ROM code checks a few devices (like floppy, hard drive etc..) in a predefined order. Suppose in our case, the boot ROM code initially considers the floppy drive. If the first sector of the floppy disk (the first 512 bytes) “looks like” a valid program that can be executed (the “validity” is verified by the last two bytes of the first sector which should be 0xaa and 0x55 to indicate that the initial 510 bytes are executable binary). Now this 510 bytes of program is called the “boot loader”. The program in the boot ROM loads the boot loader into again a “fixed” location in the RAM and transfers controls (ie jumps) to the beginning of the boot loader. Now the boot loader starts executing. The boot loader is again a program which is part of the 0.01 source. Boot loaders can be programmed according to our choice, but usually the boot loader will have a “standard” way of working and so the main kernel (which does not include the boot loader) will have to be placed on the floppy (or whatever medium is used) according to what the boot loader expects. Now the job of the boot loader (in 0.0.1) is to load the actual kernel from where it is placed in the floppy disk to the base

address of the kernel (0x0 in our case). After loading the actual kernel to its base address, the boot loader transfers control (ie jumps) to the start of the kernel code. In our case (0.01), it also switches from the 8086 mode to the “protected mode” before it transfers control to the kernel. The code that contains the boot loader that does the tasks mentioned above is present in the file boot/boot.s.

5.1.2.2. The “head” Of The Kernel - boot/head.s

The Kernel code starts with the code in the file head.s - so it is called “head”.s :-). As we had mentioned in the previous section, the kernel code starts in protected mode. The boot.s sets up a dummy IDT,GDT, LDT etc.. before it transfers control to the kernel code. But we are not going to work with the dummy IDT etc.., we need our choice and freedom in organizing the IDT,GDT,LDTs etc.. So one thing that need to be done after switching to kernel mode and before the main kernel code starts executing is to setup a “good” IDT,GDT etc.. Again, another task that needs to be done before the main kernel code starts is to setup paging, page tables etc. With all these setup and ready to be used, we can safely turn on the interrupts (because IDT is setup) and start executing the main kernel code. So it is exactly this part of the code that is contained in the file boot/head.s. Now one question is why does this code have to be written in assembly - why not in C ? Well, the answer is that if we put in a lot of effort, we can manage to write this in C, but such “low level” coding looks more clear in assembly. Of course, if we want to write values into certain registers as part of the “initialization”, we will have to use some inline assembly statements. So more than 75% of our C code will be inline assembly. Why clutter C code with inline assembly rather than using assembly directly ?

5.1.3. linux/init Directory

5.1.3.1. The “main” Kernel - init/main.c

Well, main.c contais the function main(). We jump from the code in head.s to the main in init/main.c. Now what does the code in main.c do ? Again, before the actual kernel starts running, we have to do all the “initialization” stuff like initializing the console, hard disk, initializing the file system structures etc..After this, the kernel is “fully ready” and can start proper multitasking. So now that the kernel is completely ready, it forks a few processes - the init process, a process for scanning the file system,a shell etc.. (details in the next refinement :-). After this we can say that Linux0.01 is fully “operational”. A shell is running and it will get scheduled once in a while. Then the commands that we type will be interpreted by the shell which may gain lead to another fork or exec and that continues...

5.1.4. linux/kernel Directory

5.1.4.1. Software int 0x80 - kernel/system_call.s

The next few sections speak about files containing the kernel code that is executed when a process issues a system call to the kernel or the kernel gets some exception like divide by zero etc. This file contains the code that decides what the kernel does when it gets a software

interrupt 0x80. That is, basically the code that executes after the software interrupt $0x80 in the 386. Software int $0x80 is used by all the system calls in linux. This also contains the code for signal handling (which is quite an interesting way oh handling signals), the code for fork system call, exec system call, the timer interrupt and the hard disk interrupt.

5.1.4.2. System Calls - kernel/sys.c

This contains the C code for a few other system calls.

5.1.4.3. Exceptions - kernel/asm.s, kernel/traps.c

asm.s contains code for certain exceptions, traps.c sets up the descriptors for each of these exceptions, sets the address of the exception vectors in asm.s in the descriptors etc. - ie it does the initialization part for these exceptions.

5.1.4.4. Console Functions - kernel/console.c, kernel/tty_io.c

Well, nothing much to say except that they are related to console handling!!

5.1.4.5. Miscellaneous Files - kenel/panic.c, kernel/mktime.c, kernel/vsprintf.c, kernel/printk.c

Well, these are not playing much important role in the core kernel, but of course they are very essential (like vsprintf.c - it gives us a “printf” function, if you don’t mind go through the nice comment in the beginning of vsprintf.c now itself!!). And of course, without printk.c, we cannot print messages from the kernel mode (printf is for use from the process level).

5.1.4.6. Device Handling Code - kernel/hd.c, kernel/keyboard.s, kernel/rs_io.s

Well, as the names indicate, these contain the code to handle harddisk interrupts, keyboard interupts and serial port (rs232) interrupts. Well, they handle all the buffering/queuing jobs etc that needs to be done to handle these devices!!

5.1.4.7. Important System Calls - fork() & exit() - kernel/fork.c, kernel/exit.c

The importance of the fork system call can never be over estimated!!! The code for fork is a real beauty to a person who is seeing OS code for the first time!! The exit system call is also important, but its code is not as beautiful as that for fork - fork is a beast of beauty :-)

5.1.4.8. The “heart” of the Kernel - scheduler - os/sched.c

With this file, we end the description of the files in the linux/kernel directory. Now as mentioned in one of the previous sections, after the kernel is fully functional, if the user types something or a mouse attached to the serial port is “moved”, then we get an interrupt and some function in one of the files we discussed above gets executed. If during the execution of a process, if it issues a system call, then again one of the files explained above comes

into picture. Now when does this scheduler come into picture ? We had mentioned about the timer interrupt - the “heart beat” of any multitasking OS. The code for handling this is also in kernel/sched.c. The timer interrupt code keeps an account of how long each process has been executing. If it finds that the currently executing process has exceeded its time quota, then the scheduler code comes into picture and another process gets scheduled. The scheduler also runs in situations like a process reading from the hard disk - here the data might not be readily available, so the kernel schedules another process and “wakes up” this process when the data is ready. Also, a process might issue a pause system call to temporarily suspend the process - then again the scheduler runs to schedule some other process! The kernel/sched.c also contains code for a few other system calls.

5.1.5. linux/mm Directory

Well, this is another directory that contains some other beautiful piece of code. This directory contains code for the memory management - that is paging (with respect to 0.01).

5.1.5.1. Page Fault Handler - mm/page.s

As we had mentioned earlier, page faults can occur due to two reasons in 0.01. One is that we may try to write to a page that is read only (a shared page - like code segment ??). The other is a “page not present” situation - here we will have to get a new page(s) and fill up the page table/page directory entry which is connected with the page fault. The assembly code in this file is executed on a page fault - it just distinguishes between the two reasons for the page fault and calls the C functions in memory.c to do the rest of the job.

5.1.5.2. C code for Page Not Present and Page Not Writable - mm/memory.c

As explained in the above section, this file contains the C code that is executed corresponding to the page fault exceptions for page not present and page is read only.

5.1.6. linux/fs Directory

Ok, the file system is one thing which we are not very much interested in and so not much knowledged about! So please don’t expect much from the file system explanation. We will organize the files in the fs directory in a “collective” fashion starting with the collection that performs the most “low level” operations on the fs (here by fs we basically mean the hard disk Minix V1 file system - 0.01 uses Minix V1).

5.1.6.1. fs/super.c

This is the starting point for the file system on the hard disk. It reads the super block, recognizes the organization of the files system and initializes kernel variables like the root inode etc.

5.1.6.2. fs/buffer.c

This deals with the next layer above the raw read and write mechanisms - reads and writes are (or rather, has to be) buffered for optimum performance. So the buffers might correspond to disk blocks, they might have to be written back before being freed etc.. These things are done by the code in this file.

5.1.6.3. fs/bitmap.c, fs/inode.c, fs/namei.c, fs/truncate.c

The kernel first mounts the root file system (in fs/super.c) and assigns a root inode for the starting point of the file system. Now all further accesses to the file system will have to be through inodes. We may have to create new inodes, delete them, add more blocks to a file etc.. These operations need functions that can manipulate inodes. These four files provide functions for exactly that purpose.

5.1.6.4. fs/block_dev.c, fs/file_dev.c, fs/char_dev.c

The three types of devices supported in 0.01 are block devices (like /dev/hda), file devices (like /root/test.c) and character devices (like /dev/tty0). Now the code to read and write from each of these devices is present in the corresponding files. Finally the file_dev.c and block_dev.c will use common disk read/write functions, but they present different layers for abstraction though their end mode of operation will be the same.

5.1.6.5. fs/file_table.c

This is just an array of data related to the currently open files in the system (regardless of whatever the type of file - block, file or char).

5.1.6.6. fs/open.c, fs/read_write.c

Now this is the highest level of abstraction inside the kernel. Whatever be the file type (block, file or char), we have to “open” it before using it. The system call code for this is in open.c. Now after opening a file, we can read and write from the file using just the file descriptor. The code for this is in read_write.c

5.1.6.7. fs/fcntl.c, fs/ioctl.c, fs/tty_ioctl.c, fs/stat.c

These contain code for system calls that indirectly affect the operation of open/read/write etc, an example being changing the parameters like blocking or non blocking read.

5.1.6.8. fs/pipe.c

This code is for the “pipe” device for IPC - ie write an one end and read from the other end.

5.1.6.9. fs/exec.c

Now, this is the second most beautiful system call implementation after fork. This file contains the code that implements the exec system call. But this is more complicated than the fork system call - of course, the fork just needs to copy from memory whereas in exec we need to get a file from the disk into memory!

5.1.7. linux/lib Directory

Well, this code actually is not part of the kernel. It is finally bundled up as an archive for use by application (user level) programs. These files (for example dup.c) is the code for giving a system call (which finally is int 0x80) from the user level. Each file corresponds to one system call. There are more system calls - if we need any more, then we can just write a similar file with similar syntax.

5.1.8. linux/include Directory

As the name indicates, these are the files that are “included” by the C files. But the header files also contain very interesting pieces of code - like string.h containing pure assembly code for many string utilities like string copy. This directory has further sub directories like include/asm/ which contains again files with assembly code (or macros) for utilities like segment descriptor manipulation etc.

5.1.9. linux/tools Directory

This directory contains one file build.c. This file has code to “strip” the final object code that we get (the format depending on the compiler we use - like ELF, a.out etc..) from all the “information” fields and make it a pure binary and append it to the code in boot.s (which by default the linker ld86 converts into a “pure” binary format). It is this final “pure” binary file that we do a sector by sector write (raw write) to the floppy disk starting from the first sector (occupied by the boot loader).

Chapter 6. Journey to the Center of the Code

6.1. Step Wise Refinement Of The 0.01 Kernel - Step 3

So now we enter the real “Journey To The Center Of The Code”.We will try to proceed in the same order as that given in the above section on description of files.We will take each of those files and will explain in the “necessary” detail, each of the functions and pieces of code present in those files. You can expect a very good coverage of all pieces of code except the file system - our description on file system code will be very meager as we are not much knowledged and interested in that.We would also advice that the reader keep a copy of the Hardware Bible (or any equivalent) and the 386 manuals nearby when going through the code because that will help more in increasing the grasp of the code :-) Again, since the code is not a very huge one like Minix, what we will do is to place our commentary inline with the code - so you can find code and commentary intermixed!

6.1.1. linux/boot

We will be giving quite a detailed explanation about the files in this directory.

6.1.1.1. linux/boot/boot.s

```
  1
  2 So here goes boot.s
  3 |       boot.s
  4 |
  5 | boot.s is loaded at 0x7c00 by the bios-startup
  6 | routines, and moves itself
  7 | out of the way to address 0x90000, and jumps there.
  8
```

* *By 0x7c00, we mean the “combined” value after CS:IP. Remember that the x86 is still in the real mode.We don’t remember what the exact values for CS and IP will be (if there are exact values), which is immaterial also. Now why does it move itself to 0x90000 ? Well, it cant load itself in the “lower” address regions (like 0x0?) because the BIOS might store some information like the ISR table in low memory and we need the help of BIOS to get the actual kernel image (the image names “system” we get after compilation) loaded into memory. Also, it has to be well out of the way of the actual kernel image that gets loaded at 0x10000 to be sure that the kernel does not over write the boot loader. Also, it has to be remembered that the BIOS chip has address range within the first Mega Byte. So refer the Hardware manual, find the address range of the BIOS chip and make sure that all the addresses where the boot loader images and the kernel image is loaded do not overlap with the BIOS. For this the size of each of the image has also to be considered - the boot loader is fixed at 512 bytes, the kernel as Linus says in his comment below will not be more than 512Kb :-)))*

```
  1
  2 | It then loads the system at 0x10000, using BIOS interrupts. There-
after
  3
```

* *Again, it needs to be taken care that till the “full” kernel is not in memory, the BIOS’s information should not be wiped off. So temporarily load the image at 0x10000.*

```
  1
```

```
2 | it disables all interrupts, moves the system down to 0x0000, changes
3
```

* *Now that the whole image is in memory, we no longer need BIOS. So we can load the image wherever we want and so we choose location 0x0.*

```
1
2 | to protected mode, and calls the start of system. System then must
3 | RE-initialize the protected mode in it's own tables, and enable
4 | interrupts as needed.
5
```

* *After the whole kernel is in memory, we have to switch to protected mode - in 0.01, this is also done by the bootloader boot.s. It need not be done by the bootloader, the kernel can also do it. But the boot loader should not forget that it is a "boot loader" and not the kernel. So it should do just the right amount of job. So it just uses some dummy IDT, GDT etc.. and uses that to switch into the protected mode and jump to the kernel code. Now the kernel code can decide how to map its memory, how do design the GDT etc.. independently of the boot loader. so even if the kernel changes, the boot loader can be the same.*

```
1
2 | NOTE! currently system is at most 8*65536 bytes long.
3 | This should be no | problem, even in the future.
4 | want to keep it simple. This 512 kB | kernel size
5 | should be enough - in fact more would mean we'd have to move
6 | not just these start-up routines, but also do something about the cache-

7 | memory (block IO devices). The area left over in the lower 640 kB is meant
8 | for these. No other memory is assumed to be "physical", ie all memory
9 | over 1Mb is demand-paging. All addresses under 1Mb are guaranteed to match
10 | their physical addresses.
11 |
12
```

* *More about paging in the further sections. Anyway, the gist of what is written above is that the kernel code is within the first One Mega Byte and the mapping for Kernel code is one to one - that is an address 0x4012 referred inside the kernel will get translated to 0x4012 itself by the paging mechanism and similarly for all addresses. But for user processes,we have mentioned in the section on paging that address 0x3134 may correspond to "physical" address 0x200000 .*

```
1
2 | NOTE1 abouve is no longer valid in it's entirety. cache-memory is allocate
3 | above the 1Mb mark as well as below. Otherwise it is mainly correct.
4 |
5 | NOTE 2! The boot disk type must be set at compile-time, by setting
6 | the following equ. Having the boot-up procedure hunt for the right
7 | disk type is severe brain-damage.
8 | The loader has been made as simple as possible (had to, to get it
9 | in 512 bytes with the code to move to protected mode), and continuos
10 | read errors will result in a unbreakable loop. Reboot by hand. It
11 | loads pretty fast by getting whole sectors at a time whenever possible.
12
13 | 1.44Mb disks:
14 sectors = 18
15 | 1.2Mb disks:
16 | sectors = 15
17 | 720kB disks:
```

```
18 | sectors = 9
19
20 .globl begtext, begdata, begbss, endtext, enddata, endbss
21 .text
22 begtext:
23 .data
24 begdata:
25 .bss
26 begbss:
27 .text
28
29 BOOTSEG = 0x07c0
30 INITSEG = 0x9000
31 SYSSEG  = 0x1000                          | system loaded at 0x10000 (65536).
32 ENDSEG  = SYSSEG + SYSSIZE
33
34 entry start
35 start:
36         mov     ax,#BOOTSEG
37         mov     ds,ax
38         mov     ax,#INITSEG
39         mov     es,ax
40         mov     cx,#256
41         sub     si,si
42         sub     di,di
43         rep
44         movw
45         jmpi    go,INITSEG
46
```

* *The bootloader starts executing from entry point start because that will be the first byte on the floppy! In the above piece of code, the boot loader copies 512 bytes starting at BOOTSEG (0x7c00) to location INITSEG (0x9000). That is, the bootloader copies "itself" to 0x90000.*

```
 1
 2 go:     mov     ax,cs
 3         mov     ds,ax
 4         mov     es,ax
 5         mov     ss,ax
 6         mov     sp,#0x400               | arbitrary value >>512
 7
 8         mov     ah,#0x03        | read cursor pos
 9         xor     bh,bh
10         int     0x10
11
12         mov     cx,#24
13         mov     bx,#0x0007      | page 0, attribute 7 (normal)
14         mov     bp,#msg1
15         mov     ax,#0x1301      | write string, move cursor
16         int     0x10
17
18 | ok, we've written the message, now
19 | we want to load the system (at 0x10000)
20
```

* *Just to print a boot up message on the screen*

```
 1
 2
 3          mov     ax,#SYSSEG
 4          mov     es,ax           | segment of 0x010000
 5          call    read_it
 6          call    kill_motor
 7
 8
```

* *Here is where we read the kernel image from the floppy disk and load it to 0x10000. The routine read_it will be explained later.*

```
 1
 2 | if the read went well we get current cursor position ans save it for
 3 | posterity.
 4
 5          mov     ah,#0x03        | read cursor pos
 6          xor     bh,bh
 7          int     0x10            | save it in known place, con_init fetches
 8          mov     [510],dx        | it from 0x90510.
 9
10 | now we want to move to protected mode ...
11
12          cli                     | no interrupts allowed !
13
14 | first we move the system to it's rightful place
15
16          mov     ax,#0x0000
17          cld                     | 'direction'=0, movs moves forward
18 do_move:
19          mov     es,ax           | destination segment
20          add     ax,#0x1000
21          cmp     ax,#0x9000
22          jz      end_move
23          mov     ds,ax           | source segment
24          sub     di,di
25          sub     si,si
26          mov     cx,#0x8000
27          rep
28          movsw
29          j       do_move
30
31
```

* *Finally, we copy the kernel from 0x10000 to 0x0!!*

```
 1
 2 | then we load the segment descriptors
 3
 4 end_move:
 5
 6          mov     ax,cs           | right, forgot this at first. didn't work :-
   )
 7          mov     ds,ax
 8          lidt    idt_48          | load idt with 0,0
 9          lgdt    gdt_48          | load gdt with whatever appropriate
10
```

```
11
```

* *Prepare ourselves to switch to the protected mode. For this, GDT and IDT has to be initialized. We have a dummy IDT and GDT called idt_48 and gdt_48 respectively which enables us to jump to the protected mode.*

```
 1
 2 | that was painless, now we enable A20
 3
 4         call    empty_8042
 5         mov     al,#0xD1                | command write
 6         out     #0x64,al
 7         call    empty_8042
 8         mov     al,#0xDF                | A20 on
 9         out     #0x60,al
10         call    empty_8042
11
12
```

* *Refer to the hardware Manual about this A20 (Address Line 20) line controlled by the Keyboard controller. Actually, the A20 line is used in real mode in 32 bit systems to get access to more memory even when in real mode - the key board controller can be made to drive this Address Line 20 high in order to access more memory above the 1Mb limit. But then why not ask the keyboard controller to introduce an A21, A22 etc... :-) so that we can go on accessing the entire 4Gb memory range even when in real mode ? Well, we don't have any answer to this as of now (We will add something here if we get to know the answer later). But remember that this A20 extension will allow the "extra" memory to be accessed only as data and not as an executable memory area because it is NOT the processor who is driving the A20 line, but it is the keyboard controller who has to be programmed via I/O registers to drive the A20.*

```
 1
 2 | well, that went ok, I hope. Now we have to reprogram the inter-
rupts :-(
 3 | we put them right after the intel-reserved hardware interrupts, at
 4 | int 0x20-0x2F. There they won't mess up anything. Sadly IBM really
 5 | messed this up with the original PC, and they haven't been able to
 6 | rectify it afterwards. Thus the bios puts interrupts at 0x08-0x0f,
 7 | which is used for the internal hardware interrupts as well. We just
 8 | have to reprogram the 8259's, and it isn't fun.
 9
10         mov     al,#0x11                | initialization sequence
11         out     #0x20,al                | send it to 8259A-1
12         .word   0x00eb,0x00eb           | jmp $+2, jmp $+2
13         out     #0xA0,al                | and to 8259A-2
14         .word   0x00eb,0x00eb
15         mov     al,#0x20                | start of hardware int's (0x20)
16         out     #0x21,al
17         .word   0x00eb,0x00eb
18         mov     al,#0x28                | start of hardware int's 2 (0x28)
19         out     #0xA1,al
20         .word   0x00eb,0x00eb
21         mov     al,#0x04                | 8259-1 is master
22         out     #0x21,al
23         .word   0x00eb,0x00eb
24         mov     al,#0x02                | 8259-2 is slave
25         out     #0xA1,al
26         .word   0x00eb,0x00eb
27         mov     al,#0x01                | 8086 mode for both
```

```
28          out     #0x21,al
29          .word   0x00eb,0x00eb
30          out     #0xA1,al
31          .word   0x00eb,0x00eb
32          mov     al,#0xFF                | mask off all interrupts for now
33          out     #0x21,al
34          .word   0x00eb,0x00eb
35          out     #0xA1,al
36
37  | well, that certainly wasn't fun :-(. Hopefully it works, and we don't
38  | need no steenking BIOS anyway (except for the initial loading :-
    ).
39  | The BIOS-routine wants lots of unnecessary data, and it's less
40  | "interesting" anyway. This is how REAL programmers do it.
41  |
42
```

* *Well, we know that the IBM PC series uses the Intel 8259 interrupt controller which can handle upto 8 (or 15 in cascaded mode) interrupts with varios forms of priority etc. On getting an interrupt from 8259, the x86 identifies the source of the interrupt be reading a value from one of the registers of 8259. If that value is say xy, then the x86 issues a software interrupt "int xy" to execute the ISR for that interrupt source. Again in the 32 bite x86 series, the first 32 software interrupts are "reserved" by Intel for uses like excepetions such as divide by zero (not very sure, refer the Intel manual). So we program the 8259 with values starting from 0x20 (ie 32) corresponding to interrupt line 0 on the 8259.*

```
1
2   | Well, now's the time to actually move into protected mode. To make
3   | things as simple as possible, we do no register set-up or anything,
4   | we let the gnu-compiled 32-bit programs do that. We just jump to
5   | absolute address 0x00000, in 32-bit protected mode.
6
7           mov     ax,#0x0001      | protected mode (PE) bit
8           lmsw    ax              | This is it!
9           jmpi    0,8             | jmp offset 0 of segment 8 (cs)
10
11
```

* *And finally, we move to protected mode by setting the bit 0 of the concerned register (name we forgot!!) using the lmsw instruction. Also, the Intel manual states that the transition to protected mode will be complete only with a jump instruction following that! So we jump to offset 0 in Code Segment number 8 which has been set to start from absolute physical address 0x0 (again, note that all this code is part of the boot loader and so is running from 0x90000). Now what does this mean ? What is the code at 0x0 ? It is the 0.01 Kernel Code!!!!! So we finally start executing the kernel code and we never come back to the bootloader code unless we do a reset and the kernel has to be loaded again :-(*

```
1
2   | This routine checks that the keyboard command queue is empty
3   | No timeout is used - if this hangs there is something wrong with
4   | the machine, and we probably couldn't proceed anyway.
5   empty_8042:
6           .word   0x00eb,0x00eb
7           in      al,#0x64        | 8042 status port
8           test    al,#2           | is input buffer full?
9           jnz     empty_8042      | yes - loop
10          ret
11
```

```
12 | This routine loads the system at address 0x10000, making sure
13 | no 64kB boundaries are crossed. We try to load it as fast as
14 | possible, loading whole tracks whenever we can.
15 |
16 | in:   es - starting address segment (normally 0x1000)
17 |
18 | This routine has to be recompiled to fit another drive type,
19 | just change the "sectors" variable at the start of the file
20 | (originally 18, for a 1.44Mb drive)
21 |
22
```

* *This particular piece of code below seems to appear complicated to many - so let us give a pseudo language description of the control transfer that happens below. /* We copy the kernel using es:[bx] indexing mode. We start with es = 0x0 and bx = 0x0, we go on incrementing bx till bx = 0xffff. Then we add 0x1000 es and again make bx = 0x0. Now, why do we add 0x1000 to es ? - we know that the x86 addressing is es * 4 + bx. Now bx has already used all the four bits (0xffff). So to avoid overlap of address (and thus overwriting the code/data), we need to ensure that es * 4 has always the lower four bits as zero. Now if the number is 0x?000, then number * 4 is always 0x?0000. */ es = 0x0; bx = 0x0; sread = 1; /* We have already read the first sector which is the bootloader */ head = track = 0; /* We have two heads */ /* Assume we are reading from track number "track", head number "head" and that we have already read "sread" sectors on this track. Also assume that the last segment that we will need to use (depending on the size of the image) is ENDSEG. We will explain how to calculate this later */ die: some error occured while reading from the floppy, loop here for ever !! rp_read: if (es >= ENDSEG) {we have loaded the full kernel into memory, return to the point where we were called } ok1_read: Calculate the number of sectors that can be read into the remaining area in the current segment (es). ok2_read: Now call read_track which will use the BIOS routines to load the requested number of sectors into es:[bx]. if (all the sectors in the current track has NOT been loaded into memory) {goto ok3_read} if (all the sectors in the current track has been loaded into memory) {this means we have read a full track. So find out which head needs to be used for the next read (head = 0 or 1). if (head is 1) {then we have to read the "other side" of the same track. Go to ok3_read} else if (head is 0) {then we have to read from the first head of the "next track". Fall through to ok4_read} ok4_read: Increment the value of "track" variable. ok3_read: Update the value of "sread" variable with the number of sectors read till now. if (there is more space in the current segment) {goto rp_read} else {es = es + 0x1000; bx = 0x0; goto rp_read} Now one question is "even if there is space left in the current segment, ie bx <0xffff, what happens if the space left in the current segment is not enough to hold one sector of data ?". The answer is that such a situation will not arise because the sector size is 512 bytes and the segment size is a multiple of 512. Now we will give short comments in between where things are not clear.*

```
 1
 2 sread:  .word 1                 | sectors read of current track
 3 head:   .word 0                 | current head
 4 track:  .word 0                 | current track
 5 read_it:
 6         mov ax,es
 7         test ax,#0x0fff
 8 die:    jne die                 | es must be at 64kB boundary
 9         xor bx,bx               | bx is starting address within segment
10 rp_read:
11         mov ax,es
12         cmp ax,#ENDSEG          | have we loaded all yet?
13         jb ok1_read
14         ret
15
```

* *How do we find out ENDSEG ? Well, what we used to do was to compile an image with some value for SYSSIZE (ENDSEG = SYSSEG + SYSSIZE) and after compilation, see what the size of the image is and calculate SYSSIZE accordingly and recompile!! Well, this is possible because the compilation time is too less. What would be done is to find out the location in the image where this SYSSIZE is used and just use some small C program to overwrite that location with the value of the SYSSIZE calculated from the final image.*

```
1
2 ok1_read:
3         mov ax,#sectors
4         sub ax,sread
5         mov cx,ax
6         shl cx,#9
7
```

* *shl cx,#9 multiplies cx by 512 - the size of the sector.*

```
1
2         add cx,bx
3         jnc ok2_read
4         je ok2_read
5         xor ax,ax
6         sub ax,bx
7
```

* *We want to find how many bytes are "left" in the current segment. For this, what we should do is 0x10000 - bx which is effectively 0x0 - bx !!!*

```
1
2         shr ax,#9
3
```

* *Convert bytes to sectors*

```
1
2 ok2_read:
3         call read_track
4         mov cx,ax
5         add ax,sread
6         cmp ax,#sectors
7         jne ok3_read
8         mov ax,#1
9         sub ax,head
10        jne ok4_read
11        inc track
12 ok4_read:
13        mov head,ax
14        xor ax,ax
15 ok3_read:
16        mov sread,ax
17        shl cx,#9
18        add bx,cx
19        jnc rp_read
20        mov ax,es
21        add ax,#0x1000
22        mov es,ax
```

```
23          xor bx,bx
24          jmp rp_read
25
26
```

* *Rest of the code above can be directly mapped to what we have written in the pseudo code.*

```
 1
 2 read_track:
 3          push ax
 4          push bx
 5          push cx
 6          push dx
 7          mov dx,track
 8          mov cx,sread
 9          inc cx
10          mov ch,dl
11          mov dx,head
12          mov dh,dl
13          mov dl,#0
14          and dx,#0x0100
15          mov ah,#2
16          int 0x13
17          jc bad_rt
18          pop dx
19          pop cx
20          pop bx
21          pop ax
22          ret
23 bad_rt: mov ax,#0
24          mov dx,#0
25          int 0x13
26          pop dx
27          pop cx
28          pop bx
29          pop ax
30          jmp read_track
31
32 /*
33  * This procedure turns off the floppy drive motor, so
34  * that we enter the kernel in a known state, and
35  * don't have to worry about it later.
36  */
37 kill_motor:
38          push dx
39          mov dx,#0x3f2
40          mov al,#0
41          outb
42          pop dx
43          ret
44
45 gdt:
46          .word   0,0,0,0          | dummy
47
48          .word   0x07FF           | 8Mb - limit=2047 (2048*4096=8Mb)
49          .word   0x0000           | base address=0
50          .word   0x9A00           | code read/exec
```

```
51          .word   0x00C0          | granularity=4096, 386
52
53          .word   0x07FF          | 8Mb - limit=2047 (2048*4096=8Mb)
54          .word   0x0000          | base address=0
55          .word   0x9200          | data read/write
56          .word   0x00C0          | granularity=4096, 386
57
58
```

* *This is the "dummy" gdts that we were speaking about. This just maps the lower 8Mb of addresses to the lower 8Mb of physical memory (by setting base address = 0x0 and limit = 8Mb). We create two gdt entries one for code segment and one for data segment as we can find from the read/exec and read/write attributes. The code segment is entry number 1 (assuming to start from 0), but with the first few extra bits in the segment descriptor for indicating priority level etc.., the code segment will be actually 8 when it gets loaded into cs. Again, refer to the intel manual to find out how exactly the entry number 1 becomes 8 when loaded into cs. Similarly, you can find out what will be the value of a segment descriptor for the data segment, the data segment entry being number 2. The exact layout of the hex values can be understood only by reading the Intel manuals.*

```
1
2 idt_48:
3           .word   0                       | idt limit=0
4           .word   0,0                     | idt base=0L
5
6
```

* *We believe the interrupts are disabled as of now and so we don't need a proper IDT. That explains all the zeroes in idt_48 above. The values in idt_48 are loaded into the register pointing to the IDT using lidt instruction. Again, what each of those zeroes mean will have to be understood by going through the Intel Manual.*

```
1
2 gdt_48:
3           .word   0x800           | gdt limit=2048, 256 GDT entries
4           .word   gdt,0x9         | gdt base = 0X9xxxx
5
6
```

* *This is for presenting all gdt related info in the fashion expected by the lgdt instruction.*

```
1
2 msg1:
3           .byte 13,10
4           .ascii "Loading system ..."
5           .byte 13,10,13,10
6
7
```

* *Modify the above to print your own message :0)*

```
1
2 .text
3 endtext:
4 .data
5 enddata:
```

```
 6 .bss
 7 endbss:
 8
```

6.1.1.2. linux/boot/head.s

Now let us look into head.s. This file contains code for three main items before the kernel can start functioning as a fully multitasking one. The GDT, IDT and paging has to be initialized. So here goes...

```
 1
 2 /*
 3  *  head.s contains the 32-bit startup code.
 4  *
 5  * NOTE!!! Startup happens at absolute address 0x00000000, which is also whe
 6  * the page directory will exist. The startup code will be overwrit-
ten by
 7  * the page directory.
 8  */
 9
```

* *We jump to startup_32 from boot.s. The comment says that the "startup" code will be overwritten by the page tables - what does that mean ? The code below to setup the GDT and IDT are not needed after the setup is done. So after that code is executed, four pages of memory starting from 0x0 are used for paging purposes as page directory and page tables. That is what we mean by "overwriting" the code.*

```
 1
 2 .text
 3 .globl _idt,_gdt,_pg_dir
 4 _pg_dir:
 5 startup_32:
 6         movl $0x10,%eax
 7         mov %ax,%ds
 8         mov %ax,%es
 9         mov %ax,%fs
10         mov %ax,%gs
11
```

* *Till this point, we are relying on the GDT which was setup in boot.s (direct memory mapping). 0x10 is the Data Segment.*

```
 1
 2         lss _stack_start,%esp
 3
```

* *We will need stacks for function calls. So we setup the ess and esp using values from the structure stack_start which is declared in kernel/sched.c. Again, the format of the structure will be understood only by looking into the intel manual and seeing what is the format of the argument for lss and what lss does!*

```
 1
 2         call setup_idt
 3         call setup_gdt
 4
```

* *Now setup IDT and GDT according to the preferences of the kernel.*

```
1
2         movl $0x10,%eax         # reload all the segment registers
3         mov %ax,%ds             # after changing gdt. CS was already
4         mov %ax,%es             # reloaded in 'setup_gdt'
5         mov %ax,%fs
6         mov %ax,%gs
7         lss _stack_start,%esp
8
```

* *Again modify the segment registers to reflect the "new" values in the GDT and IDT.*

```
1
2         xorl %eax,%eax
3 1:      incl %eax               # check that A20 really IS enabled
4         movl %eax,0x000000
5         cmpl %eax,0x100000
6         je 1b
7         movl %cr0,%eax          # check math chip
8         andl $0x80000011,%eax   # Save PG,ET,PE
9         testl $0x10,%eax
10        jne 1f                  # ET is set - 387 is present
11        orl $4,%eax             # else set emulate bit
12 1:     movl %eax,%cr0
13        jmp after_page_tables
14
```

* *Now the "jmp after_page_tables". All the memory till the label after_page_tables will be used for paging and will be over written.*

```
1
2
3 /*
4  *  setup_idt
5  *
6  *  sets up a idt with 256 entries pointing to
7  *  ignore_int, interrupt gates. It then loads
8  *  idt. Everything that wants to install itself
9  *  in the idt-table may do so themselves. Interrupts
10 *  are enabled elsewhere, when we can be relatively
11 *  sure everything is ok. This routine will be over-
12 *  written by the page tables.
13 */
14 setup_idt:
15        lea ignore_int,%edx
16        movl $0x00080000,%eax
17        movw %dx,%ax            /* selector = 0x0008 = cs */
18        movw $0x8E00,%dx        /* interrupt gate - dpl=0, present */
19
20        lea _idt,%edi
21        mov $256,%ecx
22 rp_sidt:
23        movl %eax,(%edi)
24        movl %edx,4(%edi)
25        addl $8,%edi
26        dec %ecx
```

```
27          jne rp_sidt
28          lidt idt_descr
29          ret
30
31
```

* *The comments by Linus for this function are pretty self evident. The IDT with 256 entries is represented by the variable _idt. Now after initializing _idt with "ignore_int" (the actual interrupts will be initialized later at various points), the lidt instruction is used with idt_descr as parameter, whose format again can be understood by looking into the manual.*

```
 1
 2 /*
 3  *  setup_gdt
 4  *
 5  *  This routines sets up a new gdt and loads it.
 6  *  Only two entries are currently built, the same
 7  *  ones that were built in init.s. The routine
 8  *  is VERY complicated at two whole lines, so this
 9  *  rather long comment is certainly needed :-).
10  *  This routine will beoverwritten by the page tables.
11  */
12 setup_gdt:
13          lgdt gdt_descr
14          ret
15
16 .org 0x1000
17 pg0:
18
19 .org 0x2000
20 pg1:
21
22 .org 0x3000
23 pg2:            # This is not used yet, but if you
24                  # want to expand past 8 Mb, you'll have
25                  # to use it.
26 .org 0x4000
27
```

* *Let us explain a bit about the page directory and the page tables. The Intel architecture uses two levels of paging - one page directory and 1024 page tables. The page directory starts from 0x0 and extends till 0x1000 (4K). In 0.01, we use two page tables which start at 0x10000 (pg0) and 0x20000 (pg1) respectively. These page tables are respectively the first and second entries in the page directory. So we can see that the total memory that can be mapped by two page tables is 2 * 1024 pages = 8Mb (one page = 4K). Now the page table starting at 0x30000 till 0x40000 (pg2) is not in use in 0.01.Again, one point to be noted is that these page directory/tables are for use ONLY by the kernel. Each process will have to setup its' own page directory and page tables (TODO:not very sure, will correct/confirm this later)*

```
 1
 2 after_page_tables:
 3          pushl $0                # These are the parameters to main :-
)
 4          pushl $0
 5          pushl $0
 6          pushl $L6               # return address for main, if it de-
cides to.
```

```
 7         pushl $_main
 8         jmp setup_paging
 9 L6:
10         jmp L6                  # main should never return here, but
11                                 # just in case, we know what hap-
   pens.
12
```

* *After setting up paging, we jump to main, where the actual “C” code for the kernel starts.*

```
 1
 2
 3 /* This is the default interrupt "handler" :-) */
 4 .align 2
 5 ignore_int:
 6         incb 0xb8000+160                # put something on the screen
 7         movb $2,0xb8000+161             # so that we know something
 8         iret                            # happened
 9
10
11 /*
12  * Setup_paging
13  *
14  * This routine sets up paging by setting the page bit
15  * in cr0. The page tables are set up, identity-mapping
16  * the first 8MB. The pager assumes that no illegal
17  * addresses are produced (ie >4Mb on a 4Mb machine).
18  *
19  * NOTE! Although all physical memory should be identity
20  * mapped by this routine, only the kernel page functions
21  * use the >1Mb addresses directly. All "normal" functions
22  * use just the lower 1Mb, or the local data space, which
23  * will be mapped to some other place - mm keeps track of
24  * that.
25  *
26  * For those with more memory than 8 Mb - tough luck. I’ve
27  * not got it, why should you :-) The source is here. Change
28  * it. (Seriously - it shouldn’t be too difficult. Mostly
29  * change some constants etc. I left it at 8Mb, as my machine
30  * even cannot be extended past that (ok, but it was cheap :-)
31  * I’ve tried to show which constants to change by having
32  * some kind of marker at them (search for "8Mb"), but I
33  * won’t guarantee that’s all :-( )
34  */
35 .align 2
36 setup_paging:
37         movl $1024*3,%ecx
38         xorl %eax,%eax
39         xorl %edi,%edi                  /* pg_dir is at 0x000 */
40         cld;rep;stosl
41
```

* *Fill the page directory, pg0 and pg1 with zeroes!!*

```
 1
 2         movl $pg0+7,_pg_dir             /* set present bit/user r/w */
```

```
  3         movl $pg1+7,_pg_dir+4           /*  --------- " " -------
-- */
  4
```

* *Fill the first two entries of the page directory with pointers to pg0 and pg1 and the necessary bits for paging (which again can be found from the manuals).*

```
  1
  2         movl $pg1+4092,%edi
  3         movl $0x7ff007,%eax             /*  8Mb - 4096 + 7 (r/w user,p) */
  4         std
  5 1:      stosl                   /* fill pages backwards - more ef-
ficient :-) */
  6         subl $0x1000,%eax
  7         jge 1b
  8
```

* *Now the page tables has to be filled with the physical address corresponding to the virtual address that the page table entry represents. For example, the value in the last entry of the second page table pg1 should represent the starting address of the "last" page in the physical memory whose starting address is 8Mb - 4096 (one page = 4096 bytes). Again, the second last entry of pg1 should be the starting address of the second last physical memory page = 8Mb - 4096 -4096. (4096 = 0x10000). Again, the necessary bits for paging purposes (7 = r/w user, p) are also added to the address - refer manuals.*

```
  1
  2         xorl %eax,%eax          /* pg_dir is at 0x0000 */
  3         movl %eax,%cr3          /* cr3 - page directory start */
  4         movl %cr0,%eax
  5         orl $0x80000000,%eax
  6         movl %eax,%cr0          /* set paging (PG) bit */
  7         ret                     /* this also flushes prefetch-queue */
  8
```

* *Set the Kernel page directory to start at 0x0 and then turn on paging!*

```
  1
  2 .align 2
  3 .word 0
  4 idt_descr:
  5         .word 256*8-1           # idt contains 256 entries
  6         .long _idt
  7 .align 2
  8 .word 0
  9 gdt_descr:
 10         .word 256*8-1           # so does gdt (not that that's any
 11         .long _gdt              # magic number, but it works for me :^)
 12
 13         .align 3
 14 _idt:   .fill 256,8,0           # idt is uninitialized
 15
 16 _gdt:   .quad 0x0000000000000000        /* NULL descriptor */
 17         .quad 0x00c09a00000007ff        /* 8Mb */
 18         .quad 0x00c09200000007ff        /* 8Mb */
 19         .quad 0x0000000000000000        /* TEMPORARY - don't use */
 20         .fill 252,8,0                   /* space for LDT's and TSS's etc */
 21
```

* *The GDT has at present, just three entries. The NULL descriptor, the Code Segment and the Data Segment - these two segments are for use by the kernel and so it covers the entire address from 0x0 to 8Mb which is again directly mapped to the first 8Mb of phyisical memory by the kernel page tables.*

6.1.2. linux/kernel

We will not be explaining all files in this directory. Some files like asm.s are not having any serious content to be explained, and for some files like keyboard.s and console.c, we never had the patience enough to go through and understand what was happening inside them.

6.1.2.1. linux/kernel/system_call.s

This is a very crucial file.

```
 1
 2 /*
 3  *  system_call.s  contains the system-call low-level handling routines.
 4  * This also contains the timer-interrupt handler, as some of the code is
 5  * the same. The hd-interrupt is also here.
 6  *
 7  * NOTE: This code handles signal-recognition, which happens every time
 8  * after a timer-interrupt and after each system call. Ordinary interrupts
 9  * don't handle signal-recognition, as that would clutter them up totally
10  * unnecessarily.
11  *
12  * Stack layout in 'ret_from_system_call':
13  *
14  *       0(%esp) - %eax
15  *       4(%esp) - %ebx
16  *       8(%esp) - %ecx
17  *       C(%esp) - %edx
18  *      10(%esp) - %fs
19  *      14(%esp) - %es
20  *      18(%esp) - %ds
21  *      1C(%esp) - %eip
22  *      20(%esp) - %cs
23  *      24(%esp) - %eflags
24  *      28(%esp) - %oldesp
25  *      2C(%esp) - %oldss
26  */
27
```

* *Note that the code in this file will be running on a kernel stack - why is that ? That is because the system call causes the process to switch from priority 3 to priority 0. And the 386 architecture recommends different stacks for different priority levels. Now each process has its own separate memory area for kernel stack which is created when we fork a process. We will see that in fork.c Also, registers ess, esp, eflags and cs are automatically pushed onto the kernel stack by 386 when it changes privilege level*

```
 1
 2
 3 SIG_CHLD        = 17
 4 EAX             = 0x00
 5 EBX             = 0x04
 6 ECX             = 0x08
```

```
 7 EDX             = 0x0C
 8 FS              = 0x10
 9 ES              = 0x14
10 DS              = 0x18
11 EIP             = 0x1C
12 CS              = 0x20
13 EFLAGS          = 0x24
14 OLDESP          = 0x28
15 OLDSS           = 0x2C
16
17 state   = 0             # these are offsets into the task-struct.
18 counter = 4
19 priority = 8
20 signal  = 12
21 restorer = 16           # address of info-restorer
22 sig_fn  = 20            # table of 32 signal addresses
23
24 nr_system_calls = 67
25
26 .globl _system_call,_sys_fork,_timer_interrupt,_hd_interrupt,_sys_execve
27
28 .align 2
29 bad_sys_call:
30         movl $-1,%eax
31         iret
32 .align 2
33 reschedule:
34         pushl $ret_from_sys_call
35         jmp _schedule
36 .align 2
37 _system_call:
38
```

* *You come here (_system_call) when the user level process gives an int 0x86. How is that - of course, we have set the corresponding IDT entry to point to _system_call.*

```
1
2         cmpl $nr_system_calls-1,%eax
3         ja bad_sys_call
4
```

* *The system call number (which identifies what system call code to be executed in response to int 0x86) is passed as parameter in register eax (we will see this later when we discuss files in lib/ directory). See if the system call number is in the valid range.*

```
1
2         push %ds
3         push %es
4         push %fs
5         pushl %edx
6         pushl %ecx              # push %ebx,%ecx,%edx as parameters
7         pushl %ebx              # to the system call
8
```

* *Note that now we have to push only eax (which is done a few instructions below) to get the stack layout mentioned in the comments above.*

```
1
2         movl $0x10,%edx          # set up ds,es to kernel space
3         mov %dx,%ds
4         mov %dx,%es
5         movl $0x17,%edx          # fs points to local data space
6
```

* *Remember that fs points to the "user space". will use fs heavily to access data in user space.*

```
1
2         mov %dx,%fs
3         call _sys_call_table(,%eax,4)
4
```

* *This is how we map the system call number to the actual function that needs to be executed. _sys_call_table is defined in include/linux/sys.h*

```
1
2         pushl %eax
3         movl _current,%eax
4         cmpl $0,state(%eax)              # state
5         jne reschedule
6         cmpl $0,counter(%eax)            # counter
7         je reschedule
8
```

* *The task (process) related information for each process is stored in a structure. The address of that structure for the currently running process (task) is stored in the variable current. Depending on certain information in that structure, decide whether it is time to schedule another process (preemption). We will see more about this is sched.c*

```
1
2 ret_from_sys_call:
3
```

* *Now this is one part of the code where we (our code) were stuck up a bit without knowing what is happening. Finally when we came to know about it and fixed the issue to get signals working properly, it was a great joy. In this implementation, we handle signals to a process just before we return from a system call that is made by that process. That is, when we do a signal() system call to send a signal to some other process, the kernel just sets a bit in an array (for signals). Now when the victim process does a system call, after the system call is completed, the kernel checks if there is any signal pending to that process. If so, arrange things such that when this process returns from the system call, the first thing it executes is the signal handler for that signal (or a default signal handler for that signal if there is no signal handler installed). After executing the signal handler, the process continues from the point where it did a system call. Now how does the kernel "make" the process to execute a signal handler ? This was where we were stuck up. Our process would execute a signal handler allright, but it would crash after handling the signal. This part we will explain at the point where it is done in the code (again, a few instructions later). Except for that part, the explanation above explains rest of the code below.*

```
1
2         movl _current,%eax               # task[0] cannot have signals
3         cmpl _task,%eax
4         je 3f
5         movl CS(%esp),%ebx               # was old code segment supervisor
6         testl $3,%ebx                    # mode? If so - don't check signals
```

```
 7          je 3f
 8          cmpw $0x17,OLDSS(%esp)          # was stack segment = 0x17 ?
 9          jne 3f
10
```

* *Check whether we originally came from "user mode". But we don't know why this is done - kernel code does not use system calls. So how can anybody other than the process mode task get here ?*

```
 1
 2 2:       movl signal(%eax),%ebx          # signals (bitmap, 32 signals)
 3          bsfl %ebx,%ecx                  # %ecx is signal nr, return if none
 4          je 3f
 5          btrl %ecx,%ebx                  # clear it
 6          movl %ebx,signal(%eax)
 7          movl sig_fn(%eax,%ecx,4),%ebx   # %ebx is signal handler address
 8          cmpl $1,%ebx
 9          jb default_signal               # 0 is default signal han-
dler - exit
10          je 2b                           # 1 is ignore - find next signal
11          movl $0,sig_fn(%eax,%ecx,4)     # reset signal handler address
12          incl %ecx
13
```

* *A little idea of the signal API in unix will make the above piece of code obvious*

```
 1
 2          xchgl %ebx,EIP(%esp)            # put new return address on stack
 3          subl $28,OLDESP(%esp)
 4          movl OLDESP(%esp),%edx          # push old return address on stack
 5
```

* *Now what is the trick to make the process execute the signal handler ? How does the execution get back to the place in the process where the system call was issued ? This happens when the EIP value stored on the kernel stack is popped during the privilege level change that occurs during an iret instruction. So what will happen if we replace the EIP address in the kernel stack with with the address of the signal handler ? This is exactly what we have done above. But why have we increased the stack size by 28 ? Now when the kernel returns to the process, it executes the signal handler because the address of the signal handler is popped into EIP. Now what happens after the signal handler finishes executing ? We have to continue from the point where the process executed a system call. This could have been done by placing the original value of EIP (when we entered the system call) on top of the process stack (accessible via fs register) so that after the signal handler finishes executing and it issues a ret instruction, the top of the stack is popped into EIP and the process continues. But here we introduce another function called "restorer" whose address is placed on the top of the process stack so that when the signal handler finishes executing, this "restorer" starts executing. Also, we are passing 5 parameters to restorer which come below the address of the restorer (which is on the stack top). Below that comes the actual EIP where we have to continue executing. So totally, we increase the size of the stack by 1 + 5 + 1 = 7 entries = 7*4 = 28 bytes.*

```
 1
 2          pushl %eax                      # but first check that it's ok.
 3          pushl %ecx
 4          pushl $28
 5          pushl %edx
 6          call _verify_area
 7          popl %edx
 8          addl $4,%esp
```

```
 9          popl %ecx
10          popl %eax
11
```

* *Why are the above instructions needed ? We are going to increase the size of the stack. Also we should not get a page fault from the kernel. So which means, if we increase the process stack size by 28, there should actually be a "physical" page corresponding to that address - if no page is there correspoding to that address and the kernel tries to write something there, then the kernel will get a page fault and crash. So to prevent this, we call verify_area which checks whether there is a physical page corresponding to that address, if no page is there, it gets one page and puts it there corresponding to that address.*

```
 1
 2          movl restorer(%eax),%eax
 3          movl %eax,%fs:(%edx)            # flag/reg restorer
 4
```

* *Put the address of restorer on top of the process stack so that after the signal handler executes and returns, the restorer starts executing.*

```
 1
 2          movl %ecx,%fs:4(%edx)           # signal nr
 3          movl EAX(%esp),%eax
 4          movl %eax,%fs:8(%edx)           # old eax
 5          movl ECX(%esp),%eax
 6          movl %eax,%fs:12(%edx)          # old ecx
 7          movl EDX(%esp),%eax
 8          movl %eax,%fs:16(%edx)          # old edx
 9          movl EFLAGS(%esp),%eax
10          movl %eax,%fs:20(%edx)          # old eflags
11
```

* *The 5 parameters to restorer.*

```
 1
 2          movl %ebx,%fs:24(%edx)          # old return addr
 3
```

* *The original return address where the process has to continue.*

```
 1
 2 3:       popl %eax
 3          popl %ebx
 4          popl %ecx
 5          popl %edx
 6          pop %fs
 7          pop %es
 8          pop %ds
 9          iret
10
```

* *This iret pops the EIP and starts using the process stack (ESP saved on the kernel stack). The EIP has been modified to be the signal handler address. The process stack has been increased in size by 28 (and the ESP has been subtracted by 28 to reflect that) and the top of the process stack is the restorer address.*

```
 1
 2
```

```
 3 default_signal:
 4         incl %ecx
 5         cmpl $SIG_CHLD,%ecx
 6         je 2b
 7         pushl %ecx
 8         call _do_exit           # remember to set bit 7 when dump-
ing core
 9         addl $4,%esp
10         jmp 3b
11
12
```

* *Default signal will kill the process!! A few more words about the restorer - where does the kernel get the address of the restorer from ? - it is passed in register dx when the process issues a system call and the kernel stores it in the restorer field of the task structure. This happens in function sys_signal in sched.c. So what does the restorer function look like ? For our kernel, our restorer was this restorer () { __asm("addl $20, %%esp\n\t" \ "ret"::); } That is our restorer adds 20 to the stack pointer - that is, it "ignores" all the parameters so that the top of the stack is the original eip where the process has to continue execution. After that we do a ret instruction which gets the process running again. We don't know what is the purpose of having a restorer because the actual eip could have been placed on the stack top instead of the restorer address, but this is how it works!! The rest of the code is for a few system calls and a few interrupts. What most of them does is to just setup a stack frame and call a C routine. The stack frame acts as the parameters to the C routine. Note that in the timer interrupt also, we are jumping to ret_from_sys_call which means that after each timer interrupt, the signals for the "current" process (that is, the process which was executing while the timer interrupt came) will get serviced.*

```
 1
 2 .align 2
 3 _timer_interrupt:
 4         push %ds                # save ds,es and put kernel data space
 5         push %es                # into them. %fs is used by _system_call
 6         push %fs
 7         pushl %edx              # we save %eax,%ecx,%edx as gcc doesn't
 8         pushl %ecx              # save those across function calls. %ebx
 9         pushl %ebx              # is saved as we use that in ret_sys_call
10         pushl %eax
11         movl $0x10,%eax
12         mov %ax,%ds
13         mov %ax,%es
14         movl $0x17,%eax
15         mov %ax,%fs
16         incl _jiffies
17         movb $0x20,%al          # EOI to interrupt controller #1
18         outb %al,$0x20
19         movl CS(%esp),%eax
20         andl $3,%eax            # %eax is CPL (0 or 3, 0=supervisor)
21         pushl %eax
22         call _do_timer          # 'do_timer(long CPL)' does every-
thing from
23         addl $4,%esp            # task switching to accounting ...
24         jmp ret_from_sys_call
25
26 .align 2
27 _sys_execve:
28         lea EIP(%esp),%eax
29         pushl %eax
```

```
30          call _do_execve
31          addl $4,%esp
32          ret
33
34 .align 2
35 _sys_fork:
36          call _find_empty_process
37          testl %eax,%eax
38          js 1f
39          push %gs
40          pushl %esi
41          pushl %edi
42          pushl %ebp
43          pushl %eax
44          call _copy_process
45          addl $20,%esp
46 1:       ret
47
48 _hd_interrupt:
49          pushl %eax
50          pushl %ecx
51          pushl %edx
52          push %ds
53          push %es
54          push %fs
55          movl $0x10,%eax
56          mov %ax,%ds
57          mov %ax,%es
58          movl $0x17,%eax
59          mov %ax,%fs
60          movb $0x20,%al
61          outb %al,$0x20          # EOI to interrupt controller #1
62          jmp 1f                  # give port chance to breathe
63 1:       jmp 1f
64 1:       outb %al,$0xA0          # same to controller #2
65          movl _do_hd,%eax
66          testl %eax,%eax
67          jne 1f
68          movl $_unexpected_hd_interrupt,%eax
69 1:       call *%eax              # "interesting" way of handling intr.
70          pop %fs
71          pop %es
72          pop %ds
73          popl %edx
74          popl %ecx
75          popl %eax
76          iret
77
78
```

6.1.2.2. linux/kernel/fork.c

This file uses many functions from the mm/ directory which will be explained later.

```
 1
 2 /*
 3  *  'fork.c' contains the help-routines for the 'fork' system call
 4  * (see also system_call.s), and some misc functions ('verify_area').
 5  * Fork is rather simple, once you get the hang of it, but the memory
 6  * management can be a bitch. See 'mm/mm.c': 'copy_page_tables()'
 7  */
 8 #include <errno.h>
 9
10 #include <linux/sched.h>
11 #include <linux/kernel.h>
12 #include <asm/segment.h>
13 #include <asm/system.h>
14
15 extern void write_verify(unsigned long address);
16
17 long last_pid=0;
18
19 void verify_area(void * addr,int size)
20 {
21         unsigned long start;
22
23         start = (unsigned long) addr;
24         size += start & 0xfff;
25         start &= 0xfffff000;
26         start += get_base(current->ldt[2]);
27         while (size>0) {
28                 size -= 4096;
29                 write_verify(start);
30                 start += 4096;
31         }
32 }
33
```

* *The purpose of the above function is to check whether there is a "physical" page corresponding to the virtual address range of addr to addr+size. Some address adjustment is done to page align the address and the corresponding size of the area is also increased accordingly. Note that the address (addr) passed to the function verify_area corresponds to the "logical address" (not even the virtual address) as assumed by the correposnding process. Now what does this mean ??? In Unix, we assign a "linear" block of virtual address to each process. In 0.01, the starting address of process number nr (nr need not be the process id) is nr * 0x4000000. That means the maximum virtual address size of one process in 0.01 is 0x4000000 = 64Mb. So the "base" address of process nr is nr * 0x4000000. But again, the process is not worried about any "base" address. The linker produces all the addresses in the program with respect to 0x0 (is base address 0x0), of course, the linker can be instructed to use a different base address, in that case it will add the base address to all the addresses that it generated with respect to base 0x0. Anyway, in our case, the binaries that we load are assumed to have been produced by the linker with base address assumed to be 0x0. So that means the address "addr" passed to verify area has to be added with nr * 0x4000000 to get the "actual virtual address" (phew!!) of the process. Now the function write_verify takes a page aligned "virtual" address and checks if there is a "physical" page corresponding to that virtual address. If no "physical" page is present, it puts a "physical" page corresponding to that virtual address. The function write_verify will be explained in mm/. Also, see more comments about this when you go through our comments on mm/memory.c - remember about the above explanation when you go through memory.c because the above explanation is not perfectly true.*

```
 1
 2 int copy_mem(int nr,struct task_struct * p)
 3 {
 4         unsigned long old_data_base,new_data_base,data_limit;
 5         unsigned long old_code_base,new_code_base,code_limit;
 6
 7         code_limit=get_limit(0x0f);
 8         data_limit=get_limit(0x17);
 9         old_code_base = get_base(current->ldt[1]);
10         old_data_base = get_base(current->ldt[2]);
11         if (old_data_base != old_code_base)
12                 panic("We don't support separate I&D");
13         if (data_limit < code_limit)
14                 panic("Bad data_limit");
15         new_data_base = new_code_base = nr * 0x4000000;
16         set_base(p->ldt[1],new_code_base);
17         set_base(p->ldt[2],new_data_base);
18         if (copy_page_tables(old_data_base,new_data_base,data_limit)) {
19                 free_page_tables(new_data_base,data_limit);
20                 return -ENOMEM;
21         }
22         return 0;
23 }
24
```

* *The above function is used by copy_process (the below function). When we do a fork, the child has to have a copy of the data of the parent, but the code can be shared. So this is the purpose of the above function. As mentioned previously, Unix considers the whole memory range for a process as a single block of virtual address. Different segment descriptors for code and data have the same base address, only diference will be in the read/write properties etc.. for those segments. So again, the size and base address of both code and data segments should be the same which is checked first in the above function. Then the base address for the new process is calculated as nr * 0x4000000. Also the TSS (which is stored as a member of the structure task_struct *p is also updated with the new data and code base. Now what does copy_page_tables do ? It is the main function which does the most important job in fork(). From the address new_data_base to new_data_base + data_limit, it checks the page directories - if the page directory does not contain a page for the page table, it gets a new page and puts its address in the corresponding page directory entry. Then for each entry in the page table, it checks whether there is a page for the corresponding entry in the page table for the parent process, if there is an entry, it sets bits in the new page table entry (and in the old page table entry) in such a fashion that the same page is shared "read only" by both processes. Now after the child/parent is scheduled, whoever tries to write to any of its pages, it will get a page fault. At that instance, the kernel (page fault handler) will get a new "physical" page, copy the contents of the "faulted" page to the new page and sets its permissions as read/write. Thus fork() implements a COW - Copy On Write. More about memory handling code in mm/. If copy_page_tables is unsuccesful for some reason like no more free physical pages present, then all the pages allocated till then for the new process is deallocated.*

```
 1
 2
 3 /*
 4  *  Ok, this is the main fork-routine. It copies the system process
 5  * information (task[nr]) and sets up the necessary registers. It
 6  * also copies the data segment in it's entirety.
 7  */
 8 int copy_process(int nr,long ebp,long edi,long esi,long gs,long none,
 9                 long ebx,long ecx,long edx,
10                 long fs,long es,long ds,
```

```
11                  long eip,long cs,long eflags,long esp,long ss)
12 {
13
```

* *This function is called from the assembly routine _sys_fork in system_call.s. But remember that _sys_fork is a member of sys_call_table[] and so it is called after it starts executing from _system_call in system_call.s. The reason we explained this much is to help the reader to understand how we got all the parameters for copy_process. They are all the elements on the stack before the call to copy_process.*

```
 1
 2          struct task_struct *p;
 3          int i;
 4          struct file *f;
 5
 6          p = (struct task_struct *) get_free_page();
 7          if (!p)
 8                  return -EAGAIN;
 9          *p = *current;  /* NOTE! this doesn't copy the supervisor stack */
10          p->state = TASK_RUNNING;
11          p->pid = last_pid;
12          p->father = current->pid;
13          p->counter = p->priority;
14          p->signal = 0;
15          p->alarm = 0;
16          p->leader = 0;          /* process leadership doesn't in-
herit */
17          p->utime = p->stime = 0;
18          p->cutime = p->cstime = 0;
19          p->start_time = jiffies;
20          p->tss.back_link = 0;
21
```

* *Now we get a new page for the task structure of the child process. Then we copy all fields from the parent's task structure to the child's task structure (*p = *current) and modify the necessary fields for the child, set the childs TSS with the correct values of segment descriptors, register values etc.*

```
1
2          p->tss.esp0 = PAGE_SIZE + (long) p;
3
```

* *Now the interesting part is the kernel stack. Note the line p->tss.esp0 = PAGE_SIZE + (long) p; - this means that the kernel stack starts from the bottom of the page allocated for the task structure. So obviously, the kernel stack size is maximum PAGE_SIZE - sizeof (task_struct). After that, it will overwrite the task structure.*

```
 1
 2          p->tss.ss0 = 0x10;
 3          p->tss.eip = eip;
 4          p->tss.eflags = eflags;
 5          p->tss.eax = 0;
 6          p->tss.ecx = ecx;
 7          p->tss.edx = edx;
 8          p->tss.ebx = ebx;
 9          p->tss.esp = esp;
10          p->tss.ebp = ebp;
11          p->tss.esi = esi;
12          p->tss.edi = edi;
```

```
13          p->tss.es = es & 0xffff;
14          p->tss.cs = cs & 0xffff;
15          p->tss.ss = ss & 0xffff;
16          p->tss.ds = ds & 0xffff;
17          p->tss.fs = fs & 0xffff;
18          p->tss.gs = gs & 0xffff;
19          p->tss.ldt = _LDT(nr);
20
```

* *The _LDT(nr) gets an entry for the LDT descriptor in the GDT and stores it in the TSS.*

```
1
2           p->tss.trace_bitmap = 0x80000000;
3           if (last_task_used_math == current)
4                   __asm__("fnsave %0"::"m" (p->tss.i387));
5           if (copy_mem(nr,p)) {
6                   free_page((long) p);
7                   return -EAGAIN;
8           }
9
```

* *Then it calls copy_process which was explained previously. After that it does some files/descriptor related things. Finally, it sets the address of the child's TSS and the address of the child's LDT (both are present in the task structure) in the GDT.*

```
1
2           for (i=0; i<NR_OPEN;i++)
3                   if (f=p->filp[i])
4                           f->f_count++;
5           if (current->pwd)
6                   current->pwd->i_count++;
7           if (current->root)
8                   current->root->i_count++;
9           set_tss_desc(gdt+(nr<<1)+FIRST_TSS_ENTRY,&(p->tss));
10          set_ldt_desc(gdt+(nr<<1)+FIRST_LDT_ENTRY,&(p->ldt));
11
```

* *Now update the GDT entries for this child process with the address of its TSS and LDT both of which are present in the task structure. When a task switch occurs (by "jumping" to the TSS entry in the GDT which we will see later when we explain some utility macros and functions), the processor updates all the registers etc.. with values from the TSS.*

```
1
2           task[nr] = p;   /* do this last, just in case */
3
```

* *Finally, everything is over. Now update the task[] array with the task structure address of the child.*

```
1
2           return last_pid;
3   }
4   int find_empty_process(void)
5   {
6           int i;
7
8           repeat:
```

```
 9                 if ((++last_pid)<0) last_pid=1;
10                 for(i=0 ; i<NR_TASKS ; i++)
11                         if (task[i] && task[i]->pid == last_pid) goto repeat
12         for(i=1 ; i<NR_TASKS ; i++)
13                 if (!task[i])
14                         return i;
15         return -EAGAIN;
16 }
17
```

* *Pretty self explanatory.*

6.1.2.3. linux/kernel/sched.c

Here comes the scheduler of the kernel. Somehow, we could not understand properly some parts of the code in the file - like why certain code was written the way it is written. So we are explaining our views about the code in this file, you should think more about this and try to understand the code yourself.

```
 1
 2 /*
 3  * 'sched.c' is the main kernel file. It contains scheduling primitives
 4  * (sleep_on, wakeup, schedule etc) as well as a number of simple system
 5  * call functions (type getpid(), which just extracts a field from
 6  * current-task
 7  */
 8 #include <linux/sched.h>
 9 #include <linux/kernel.h>
10 #include <signal.h>
11 #include <linux/sys.h>
12 #include <asm/system.h>
13 #include <asm/io.h>
14 #include <asm/segment.h>
15
16 #define LATCH (1193180/HZ)
17
18 extern void mem_use(void);
19
20 extern int timer_interrupt(void);
21 extern int system_call(void);
22
23 union task_union {
24         struct task_struct task;
25         char stack[PAGE_SIZE];
26 };
27
```

* *The size of the above union will be at least PAGE_SIZE.*

```
 1
 2
 3 static union task_union init_task = {INIT_TASK,};
 4
 5 long volatile jiffies=0;
 6 long startup_time=0;
 7 struct task_struct *current = &(init_task.task), *last_task_used_math = NULL
```

```
 8
 9 struct task_struct * task[NR_TASKS] = {&(init_task.task), };
10
11 long user_stack [ PAGE_SIZE>>2 ] ;
12
```

* *Remeber that in head.s, the user_stack was used as the kernel stack initially till all the multitasking etc.. was setup.*

```
 1
 2
 3 struct {
 4         long * a;
 5         short b;
 6         } stack_start = { & user_stack [PAGE_SIZE>>2] , 0x10 };
 7
```

* *This was also used in head.s*

```
 1
 2 /*
 3  *  'math_state_restore()' saves the current math information in the
 4  * old math state array, and gets the new ones from the current task
 5  */
 6 void math_state_restore()
 7 {
 8         if (last_task_used_math)
 9                 __asm__("fnsave %0"::"m" (last_task_used_math->tss.i387));
10         if (current->used_math)
11                 __asm__("frstor %0"::"m" (current->tss.i387));
12         else {
13                 __asm__("fninit"::);
14                 current->used_math=1;
15         }
16         last_task_used_math=current;
17 }
18
19 /*
20  *  'schedule()' is the scheduler function. This is GOOD CODE! There
21  * probably won't be any reason to change this, as it should work well
22  * in all circumstances (ie gives IO-bound processes good response etc).
23  * The one thing you might take a look at is the signal-handler code here.
24  *
25  *   NOTE!!  Task 0 is the 'idle' task, which gets called when no other
26  * tasks can run. It can not be killed, and it cannot sleep. The 'state'
27  * information in task[0] is never used.
28  */
29 void schedule(void)
30 {
31         int i,next,c;
32         struct task_struct ** p;
33
34 /* check alarm, wake up any interruptible tasks that have got a sig-
   nal */
35
36         for(p = &LAST_TASK ; p > &FIRST_TASK ; --p)
37                 if (*p) {
```

```
38                          if ((*p)->alarm && (*p)->alarm < jiffies) {
39                                  (*p)->signal |= (1<<(SIGALRM-
1));
40                                  (*p)->alarm = 0;
41                          }
42                          if ((*p)->signal && (*p)->state==TASK_INTERRUPTIBLE)
43                                  (*p)->state=TASK_RUNNING;
44                  }
45
```

* *Check for alarms. If the alarm time has been exceeded, set the SIGALRM signal for that process. Now check for signals. If a signal is pending to an "Interruptible" process, then the signal is a sufficient reason for that process to be woken up, but if a signal is pending to a "Non Interruptible" process, then the signal is not a sufficient reason for the process to be woken up. When a process "sleeps", it can be an interruptible or a non interruptible sleep. We will see more about this below.*

```
1
2
3 /* this is the scheduler proper: */
4
5         while (1) {
6                 c = -1;
7                 next = 0;
8                 i = NR_TASKS;
9                 p = &task[NR_TASKS];
10                while (--i) {
11                        if (!*--p)
12                                continue;
13                        if ((*p)->state == TASK_RUNNING && (*p)->counter > c)
14                                c = (*p)->counter, next = i;
15                }
16
```

* *The (*p)->counter is initially assigned a value that indicates how many clock ticks the process can run before being preempted. If counter is zero, then that means that the process has finished its quota of execution time and that it has to be rescheduled. A process can go on without being preempted for the number of clock ticks it was initially assigned, but the scheduler might also be invoked due to reasons other than the time quota expiry - like a process deciding to "sleep", a process waiting for data from hard disk etc.. So the above algorithm selects the process that has executed the least amount of time (higer value of counter).*

```
1
2                 if (c) break;
3                 for(p = &LAST_TASK ; p > &FIRST_TASK ; --p)
4                         if (*p)
5                                 (*p)->counter = ((*p)->counter >> 1) +
6                                                 (*p)->priority;
7         }
8
```

* *The above few lines is one point where we are not very sure about the algorithm. The above lines get executed when all the "running" processes have their counter value equal to zero. For those processes, their counter becomes (reinitialized) equal to their to their priority value. But for the "not running" processes, their counter values are reinitialised to "double" of their existing counter plus their priority. Why is this ??? Why are the sleeping processes given more time slot ?? So if a process goes on sleeping again and again, will its time quota go on increasing ? we don't know - figure it out!!!*

```
1
2         switch_to(next);
3
```

* *This is an assembly routine to switch context to the new process selected to be executed. We will see this when we discuss the "utlity" routines.*

```
1
2 }
3
4 int sys_pause(void)
5 {
6         current->state = TASK_INTERRUPTIBLE;
7         schedule();
8         return 0;
9 }
10
```

* *This system call (pause) is used to suspend the current process till it gets a signal (type "man pause" on your unix shell). Since a signal should be able to get this process running, it state is marked as "interruptible".*

```
1
2
3 void sleep_on(struct task_struct **p)
4 {
5         struct task_struct *tmp;
6
7
```

* *This function (and the next one) is used by other parts of the kernel to make a process to wait (suspend); till a data structure (in the kernel) whose contents the process wanted to get is "ready" to be read - like say the process asked for some data to be read from a sector on the harddisk - that sector might be mapped to a buffer in kernel space, but the buffer currently does not have the contents of the required sector. So the kernel suspends the current process and marks it as waiting "on" the buffer. When a dma interrupt notifies that the read from the hard disk is over, then the kernel calls wake_up on the process who was waiting for the buffer. Now the problem is that lots of processes might request for the same sector of data, so each process has to be marked as waiting on the same buffer - whenever the buffer is ready, all those proceses should be woken up. How can we do this ? - we can use a linked list in the buffer structure to which we can link each process (task struct) that is waiting on the buffer. When the buffer is ready, we can go through each of the processes in the linked list and wake them up. Yes, yes that is one way of doing it!! But Linus has done this in a way that will not "click" easily for novice programmers! Now hos does Linus do it differently ? Read below :-)*

```
1
2         if (!p)
3                 return;
4         if (current == &(init_task.task))
5                 panic("task[0] trying to sleep");
6         tmp = *p;
7         *p = current;
8
```

* *We will make the explanation short. Suppose there is already one process waiting on the buffer. A pointer to that task structure is sotred in a variable in the buffer structure. And the "p" that we are passed is a pointer to that variable which stores the address of the task structure of the process waiting on the buffer. Wht is done here is to copy the task structure of the process currently waiting on the buffer to "tmp". Then we make the*

"currently" waiting process to be the current process. Note that tmp is a local variable, so tmp will be on the stack. Also understand that the kernel stack is different for each process. So tmp will not be overwritten. So that is enough of an explanation.

```
1
2          current->state = TASK_UNINTERRUPTIBLE;
3
```

* *Now make the process to be sleeping "uninterruptably".*

```
1
2          schedule();
3
```

* *Schedule another process to run.*

```
1
2          if (tmp)
3                  tmp->state=0;
4
```

* *How did we reach here ? May be we got a dma interrupt saying that the buffer is ready. So the interrupt handler will set the state of the "currently" waiting process (the task structure address stored in the buffer structure) to be running. So the next time the scheduler runs, the process pointed by the buffer structure will (may) start running. So that is one way we reach here. What if we are not the "currently" waiting process ? Then what happens is that the "currently" waiting process will execute the above two lines which says that "hey man..you have woken up, so wake up the process who was waiting on this buffer before you started waiting on the buffer". This chain continues (like a push/pop sequence) and all the processes that were waiting on the buffer gets woken up - again, one thing we don't understand here is that if we had used linked lists, then we could have woken up all the waiting processes in one shot, but here we have to wait for the process "after" us to get scheduled for us to be woken up though theoretically we need not wait on any other process since the buffer is ready. So why is this ?? We don't know - figure it out ;-)*

```
1
2 }
3
4 void interruptible_sleep_on(struct task_struct **p)
5 {
6          struct task_struct *tmp;
7
8
```

* *This function is almost exactly same as the above function except that the sleep is interruptible - that is the sleeping process can be woken up by an interrupt. What does this function do ? We are still not very clear about this, but what we understood is that if "ANY" one process that is in the chain of "waiting" processes is woken up by a signal, that process inturn causes "ALL" the waiting proceses also to be woken up!!!*

```
1
2          if (!p)
3                  return;
4          if (current == &(init_task.task))
5                  panic("task[0] trying to sleep");
6          tmp=*p;
7          *p=current;
8
```

* *Till this part, it is exactly same as the previous function.*

```
1
2 repeat: current->state = TASK_INTERRUPTIBLE;
3         schedule();
4
```

* *How did we reach here (after schedule()) ? May be we were woken up by a signal ?*

```
1
2         if (*p && *p != current) {
3                 (**p).state=0;
4
```

* *What are we doing here ? We check to see if the “last” process that was waiting on the buffer is ourselves, if so, wake up the “last” process. Again set our state to “interruptible sleep” and go back to the sleeping state.*

```
1
2                 goto repeat;
3         }
4         *p=NULL;
5         if (tmp)
6                 tmp->state=0;
7 Finally the &ldquo;top most&rdquo; process wakes up and then fol-
lows the usual sequence of waking up the &ldquo;just below&rdquo; pro-
cess and the chain continues like that. What happens if the &ldquo;top most&rdqu
cess gets changed in between ? - that is another new process starts wait-
ing on the same buffer ? Then again, the same sequence mentioned in the pre-
vious comments will take place - the previously &ldquo;top most&rdquo; pro-
cess wakes up and finds that it is not the &ldquo;top most&rdquo; any-
more. So it wakes up the topmost process and goes to sleep again.  Now again, th
derstand here - first why does the &ldquo;middle&rdquo; process (not the &ldquo;
most process and go to sleep again ??? Why does it not continue to be &ldquo;awa
ning). Second is why does waking up one process (by a signal) have to re-
sult in the waking up of all the other processes in the chain - one ob-
vious reason is that the &ldquo;push/pop&rdquo; architecture necessi-
tates that if one process is woken up, then at least all the processes &ldquo;be
fore) it should be woken up; but then is that the correct way of imple-
mentation ??? We don’t know. Figure it out yourself :0)
8 }
9
10 void wake_up(struct task_struct **p)
11 {
12         if (p && *p) {
13                 (**p).state=0;
14                 *p=NULL;
15         }
16 }
17
18 void do_timer(long cpl)
19 {
20         if (cpl)
21                 current->utime++;
22         else
23                 current->stime++;
24         if ((--current->counter)>0) return;
25         current->counter=0;
```

```
26          if (!cpl) return;
27          schedule();
28  }
29
```

* *Self explanatory.*

```
 1
 2  int sys_alarm(long seconds)
 3  {
 4          current->alarm = (seconds>0)?(jiffies+HZ*seconds):0;
 5          return seconds;
 6  }
 7
 8  int sys_getpid(void)
 9  {
10          return current->pid;
11  }
12
13  int sys_getppid(void)
14  {
15          return current->father;
16  }
17
18  int sys_getuid(void)
19  {
20          return current->uid;
21  }
22
23  int sys_geteuid(void)
24  {
25          return current->euid;
26  }
27
28  int sys_getgid(void)
29  {
30          return current->gid;
31  }
32
33  int sys_getegid(void)
34  {
35          return current->egid;
36  }
37
38  int sys_nice(long increment)
39  {
40          if (current->priority-increment>0)
41                  current->priority -= increment;
42          return 0;
43  }
44
45  int sys_signal(long signal,long addr,long restorer)
46  {
47          long i;
48
49          switch (signal) {
50                  case SIGHUP: case SIGINT: case SIGQUIT: case SIGILL:
```

```
51                  case SIGTRAP: case SIGABRT: case SIGFPE: case SIGUSR1:
52                  case SIGSEGV: case SIGUSR2: case SIGPIPE: case SIGALRM:
53                  case SIGCHLD:
54                          i=(long) current->sig_fn[signal-1];
55                          current->sig_fn[signal-1] = (fn_ptr) addr;
56                          current->sig_restorer = (fn_ptr) restorer;
57                          return i;
58                  default: return -1;
59          }
60 }
61
```

* *Here comes the signal system call that we were discussing about in system_call.s Go through it carefully and see that it is indeed doing the things that we said it will do in system_call.s*

```
 1
 2
 3 void sched_init(void)
 4 {
 5          int i;
 6          struct desc_struct * p;
 7
 8          set_tss_desc(gdt+FIRST_TSS_ENTRY,&(init_task.task.tss));
 9          set_ldt_desc(gdt+FIRST_LDT_ENTRY,&(init_task.task.ldt));
10          p = gdt+2+FIRST_TSS_ENTRY;
11          for(i=1;i<NR_TASKS;i++) {
12                  task[i] = NULL;
13                  p->a=p->b=0;
14                  p++;
15                  p->a=p->b=0;
16                  p++;
17          }
18          ltr(0);
19          lldt(0);
20          outb_p(0x36,0x43);              /* binary, mode 3, LSB/MSB, ch 0 */
21          outb_p(LATCH & 0xff , 0x40);    /* LSB */
22          outb(LATCH >> 8 , 0x40);        /* MSB */
23          set_intr_gate(0x20,&timer_interrupt);
24          outb(inb_p(0x21)&~0x01,0x21);
25          set_system_gate(0x80,&system_call);
26 }
27
```

* *Initialise things related to init task (task 0), then the timer interrupt gate and system call gate. One thing we did not mention was the init_task. It is also called the idle task (see main.c). It runs when there is nothing else to run. The task structure for init_task is a predefined structure named INIT_TASK (find out which file it is in!!). Go through the fields to see what it looks like. Now how do you get this init_task running ? - remember a fork() in main.c ?? The kernel forks just before the lines - for(;;) pause(); - in main.c. So the code that was running till this line as the "kernel initialisation" now starts running as an "official" task 0. This idea of "kernel" doing a fork might not be very clear. Think more about this and it will become clear.*

6.1.2.4. linux/kernel/hd.c

We will not be explaining this file in much detail since the basic code flow here is simple.

* *The basic code path is like this - the read and request to the harddisk are implemented as a FIFO linked list. But only a maximum of NR_REQUEST outstanding requests are allowed at any time. The first request on the head of the list (be it a request to read or write data - both are on the same linked list) is processed by writing the details like the head,sector,number of bytes etc.. to be read or written to a port in the harddisk controller (all these details will be present in the request structure). Depending on whether the request was a read or write, a function pointer do_hd is set to the address of a function that handles an interrupt from the harddisk controller for a read and write respectively. For example, if the request is a read, the hd controller will issue an interrupt when the data is ready to be copied into the buffer. Then in the interrupt path, the function do_hd is called - do_hd has already been set to read_intr when the read request at the top of the list was processed. Similarly when we issue a write command to the hd controller port and the hd controller is ready to accept data, it issues an interrupt and then the function do_hd (which has been set to write_intr) will copy the data from the buffer to the memory in the hd controller. Now when a new read/write request needs to be issued, we will check whether there can be more requests, if not then we will ask the process to "wait" (suspend) till the request list has a free entry. If we get a free entry, we will put that request in a "sorted" form in the list to improve read/write efficiency.*

```
 1
 2 #include <linux/config.h>
 3 #include <linux/sched.h>
 4 #include <linux/fs.h>
 5 #include <linux/kernel.h>
 6 #include <linux/hdreg.h>
 7 #include <asm/system.h>
 8 #include <asm/io.h>
 9 #include <asm/segment.h>
10
11 /*
12  * This code handles all hd-interrupts, and read/write requests to
13  * the hard-disk. It is relatively straigthforward (not obvious maybe,
14  * but interrupts never are), while still being efficient, and never
15  * disabling interrupts (except to overcome possible race-condition).
16  * The elevator block-seek algorithm doesn't need to disable interrupts
17  * due to clever programming.
18  */
19
20 /* Max read/write errors/sector */
21 #define MAX_ERRORS      5
22 #define MAX_HD          2
23 #define NR_REQUEST      32
24
25 /*
26  *  This struct defines the HD's and their types.
27  *  Currently defined for CP3044's, ie a modified
28  *  type 17.
29  */
30 static struct hd_i_struct{
31         int head,sect,cyl,wpcom,lzone,ctl;
32         } hd_info[]= { HD_TYPE };
33
34 #define NR_HD ((sizeof (hd_info))/(sizeof (struct hd_i_struct)))
35
36 static struct hd_struct {
```

```
37          long start_sect;
38          long nr_sects;
39  } hd[5*MAX_HD]={{0,0},};
40
41  static struct hd_request {
42          int hd;         /* -1 if no request */
43          int nsector;
44          int sector;
45          int head;
46          int cyl;
47          int cmd;
48          int errors;
49          struct buffer_head * bh;
50          struct hd_request * next;
51  } request[NR_REQUEST];
52
53  #define IN_ORDER(s1,s2) \
54  ((s1)->hd<(s2)->hd || (s1)->hd==(s2)->hd && \
55  ((s1)->cyl<(s2)->cyl || (s1)->cyl==(s2)->cyl && \
56  ((s1)->head<(s2)->head || (s1)->head==(s2)->head && \
57  ((s1)->sector<(s2)->sector))))
58
59  static struct hd_request * this_request = NULL;
60
61  static int sorting=0;
62
63  static void do_request(void);
64  static void reset_controller(void);
65  static void rw_abs_hd(int rw,unsigned int nr,unsigned int sec,unsigned int he
66          unsigned int cyl,struct buffer_head * bh);
67  void hd_init(void);
68
69  #define port_read(port,buf,nr) \
70  __asm__("cld;rep;insw"::"d" (port),"D" (buf),"c" (nr):"cx","di")
71
72  #define port_write(port,buf,nr) \
73  __asm__("cld;rep;outsw"::"d" (port),"S" (buf),"c" (nr):"cx","si")
74
75  extern void hd_interrupt(void);
76
77  static struct task_struct * wait_for_request=NULL;
78
79  static inline void lock_buffer(struct buffer_head * bh)
80  {
81          if (bh->b_lock)
82                  printk("hd.c: buffer multiply locked\n");
83          bh->b_lock=1;
84  }
85
```

* *The buffer is "locked" when the buffer is allocated for use by a read/write request to the hard disk.*

```
1
2
3   static inline void unlock_buffer(struct buffer_head * bh)
4   {
5           if (!bh->b_lock)
```

```
6                 printk("hd.c: free buffer being unlocked\n");
7         bh->b_lock=0;
8         wake_up(&bh->b_wait);
9 }
10
```

* *The buffer is "unlocked" when the read/write request to the harddisk is completely processed by the hardisk and after the data is completely copied to/from the buffer.*

```
1
2 static inline void wait_on_buffer(struct buffer_head * bh)
3 {
4         cli();
5         while (bh->b_lock)
6                 sleep_on(&bh->b_wait);
7         sti();
8 }
9
```

* *Hmm...we have discussed the sleep_on in quite detail. The process has to sleep till the data is ready in the buffer, ie the lock for the buffer is released.*

```
1
2
3 void rw_hd(int rw, struct buffer_head * bh)
4 {
5         unsigned int block,dev;
6         unsigned int sec,head,cyl;
7
8         block = bh->b_blocknr << 1;
9         dev = MINOR(bh->b_dev);
10        if (dev >= 5*NR_HD || block+2 > hd[dev].nr_sects)
11                return;
12        block += hd[dev].start_sect;
13        dev /= 5;
14        __asm__("divl %4":"=a" (block),"=d" (sec):"0" (block),"1" (0),
15                "r" (hd_info[dev].sect));
16        __asm__("divl %4":"=a" (cyl),"=d" (head):"0" (block),"1" (0),
17                "r" (hd_info[dev].head));
18        rw_abs_hd(rw,dev,sec+1,head,cyl,bh);
19 }
20
21 /* This may be used only once, enforced by 'static int callable' */
22 int sys_setup(void)
23 {
24        static int callable = 1;
25        int i,drive;
26        struct partition *p;
27
28        if (!callable)
29                return -1;
30        callable = 0;
31        for (drive=0 ; drive<NR_HD ; drive++) {
32                rw_abs_hd(READ,drive,1,0,0,(struct buffer_head *) start_buffe
33                if (!start_buffer->b_uptodate) {
34                        printk("Unable to read partition table of drive %d\n\
```

```
35                                  drive);
36                          panic("");
37                  }
38                  if (start_buffer->b_data[510] != 0x55 || (unsigned char)
39                      start_buffer->b_data[511] != 0xAA) {
40                          printk("Bad partition table on drive %d\n\r",drive);
41                          panic("");
42                  }
43                  p = 0x1BE + (void *)start_buffer->b_data;
44                  for (i=1;i<5;i++,p++) {
45                          hd[i+5*drive].start_sect = p->start_sect;
46                          hd[i+5*drive].nr_sects = p->nr_sects;
47                  }
48          }
49          printk("Partition table%s ok.\n\r",(NR_HD>1)?"s":"");
50          mount_root();
51          return (0);
52  }
53
54  /*
55   * This is the pointer to a routine to be executed at every hd-interrupt.
56   * Interesting way of doing things, but should be rather practical.
57   */
58  void (*do_hd)(void) = NULL;
59
60  static int controller_ready(void)
61  {
62          int retries=1000;
63
64          while (--retries && (inb(HD_STATUS)&0xc0)!=0x40);
65          return (retries);
66  }
67
68  static int win_result(void)
69  {
70          int i=inb(HD_STATUS);
71
72          if ((i & (BUSY_STAT | READY_STAT | WRERR_STAT | SEEK_STAT | ERR_STAT)
73                  == (READY_STAT | SEEK_STAT))
74                  return(0); /* ok */
75          if (i&1) i=inb(HD_ERROR);
76          return (1);
77  }
78
79  static void hd_out(unsigned int drive,unsigned int nsect,unsigned int sect,
80                  unsigned int head,unsigned int cyl,unsigned int cmd,
81                  void (*intr_addr)(void))
82  {
83          register int port asm("dx");
84
85          if (drive>1 || head>15)
86                  panic("Trying to write bad sector");
87          if (!controller_ready())
88                  panic("HD controller not ready");
89          do_hd = intr_addr;
90          outb(_CTL,HD_CMD);
91          port=HD_DATA;
```

```
92          outb_p(_WPCOM,++port);
93          outb_p(nsect,++port);
94          outb_p(sect,++port);
95          outb_p(cyl,++port);
96          outb_p(cyl>>8,++port);
97          outb_p(0xA0|(drive<<4)|head,++port);
98          outb(cmd,++port);
99  }
100
```

* *This function writes the command to be issued to the harddisk to the necessary ports of the hd controller. Also it sets the do_hd variable to the read or write handling function which will be passed as parameter.*

```
1
2
3   static int drive_busy(void)
4   {
5           unsigned int i;
6
7           for (i = 0; i < 100000; i++)
8                   if (READY_STAT == (inb(HD_STATUS) & (BUSY_STAT | READY_STAT)))
9                           break;
10          i = inb(HD_STATUS);
11          i &= BUSY_STAT | READY_STAT | SEEK_STAT;
12          if (i == READY_STAT | SEEK_STAT)
13                  return(0);
14          printk("HD controller times out\n\r");
15          return(1);
16  }
17
18  static void reset_controller(void)
19  {
20          int     i;
21
22          outb(4,HD_CMD);
23          for(i = 0; i < 1000; i++) nop();
24          outb(0,HD_CMD);
25          for(i = 0; i < 10000 && drive_busy(); i++) /* nothing */;
26          if (drive_busy())
27                  printk("HD-controller still busy\n\r");
28          if((i = inb(ERR_STAT)) != 1)
29                  printk("HD-controller reset failed: %02x\n\r",i);
30  }
31
32  static void reset_hd(int nr)
33  {
34          reset_controller();
35          hd_out(nr,_SECT,_SECT,_HEAD-1,_CYL,WIN_SPECIFY,&do_request);
36  }
37
38  void unexpected_hd_interrupt(void)
39  {
40          panic("Unexpected HD interrupt\n\r");
41  }
42
43  static void bad_rw_intr(void)
44  {
```

```
45          int i = this_request->hd;
46
47          if (this_request->errors++ >= MAX_ERRORS) {
48                  this_request->bh->b_uptodate = 0;
49                  unlock_buffer(this_request->bh);
50                  wake_up(&wait_for_request);
51                  this_request->hd = -1;
52                  this_request=this_request->next;
53          }
54          reset_hd(i);
55  }
56
57  static void read_intr(void)
58  {
59          if (win_result()) {
60                  bad_rw_intr();
61                  return;
62          }
63          port_read(HD_DATA,this_request->bh->b_data+
64                  512*(this_request->nsector&1),256);
65          this_request->errors = 0;
66          if (--this_request->nsector)
67                  return;
68          this_request->bh->b_uptodate = 1;
69          this_request->bh->b_dirt = 0;
70          wake_up(&wait_for_request);
71          unlock_buffer(this_request->bh);
72          this_request->hd = -1;
73          this_request=this_request->next;
74          do_request();
75  }
76
```

* *do_hd was set to read_intr when a read request was issued. This copies the data to the buffer and unlocks the buffer which again wakes up all the processes that were waiting to get the data from this buffer. Now that one request has been processed completely, we can enqueue a new request to the queue. So we also wake up all the processes that were waiting for a free entry on the request list. Finally, it tries to process the next request on the head of the request queue (do_request()).*

```
 1
 2
 3  static void write_intr(void)
 4  {
 5          if (win_result()) {
 6                  bad_rw_intr();
 7                  return;
 8          }
 9          if (--this_request->nsector) {
10                  port_write(HD_DATA,this_request->bh->b_data+512,256);
11                  return;
12          }
13          this_request->bh->b_uptodate = 1;
14          this_request->bh->b_dirt = 0;
15          wake_up(&wait_for_request);
16          unlock_buffer(this_request->bh);
17          this_request->hd = -1;
18          this_request=this_request->next;
```

```
        do_request();
}

```

* *This functions exactly similar to read_intr.*

```


static void do_request(void)
{
        int i,r;

        if (sorting)
                return;
        if (!this_request) {
                do_hd=NULL;
                return;
        }
        if (this_request->cmd == WIN_WRITE) {
                hd_out(this_request->hd,this_request->nsector,this_request-
>
                        sector,this_request->head,this_request->cyl,
                        this_request->cmd,&write_intr);
                for(i=0 ; i<3000 && !(r=inb_p(HD_STATUS)&DRQ_STAT) ; i++)
                        /* nothing */ ;
                if (!r) {
                        reset_hd(this_request->hd);
                        return;
                }
                port_write(HD_DATA,this_request->bh->b_data+
                        512*(this_request->nsector&1),256);
        } else if (this_request->cmd == WIN_READ) {
                hd_out(this_request->hd,this_request->nsector,this_request-
>
                        sector,this_request->head,this_request->cyl,
                        this_request->cmd,&read_intr);
        } else
                panic("unknown hd-command");
}

```

* *This is the function that takes the first entry on the request queue and depending on whether it is a read or write request, it passes read_intr or write_intr as parameter to hd_out. Now there is one interesting part - if(sorting) return; - that will be explained next.*

```


/*
 * add-request adds a request to the linked list.
 * It sets the 'sorting'-variable when doing something
 * that interrupts shouldn't touch.
 */
static void add_request(struct hd_request * req)
{
        struct hd_request * tmp;

        if (req->nsector != 2)
```

```
13                 panic("nsector!=2 not implemented");
14 /*
15  * Not to mess up the linked lists, we never touch the two first
16  * entries (not this_request, as it is used by current interrups,
17  * and not this_request->next, as it can be assigned to this_request).
18  * This is not too high a price to pay for the ability of not
19  * disabling interrupts.
20  */
21         sorting=1;
22         if (!(tmp=this_request))
23                 this_request=req;
24         else {
25                 if (!(tmp->next))
26                         tmp->next=req;
27                 else {
28                         tmp=tmp->next;
29                         for ( ; tmp->next ; tmp=tmp->next)
30                                 if ((IN_ORDER(tmp,req) ||
31                                     !IN_ORDER(tmp,tmp->next)) &&
32                                     IN_ORDER(req,tmp->next))
33                                         break;
34                         req->next=tmp->next;
35                         tmp->next=req;
36                 }
37         }
38         sorting=0;
39
```

* *This function is responsible for adding new requests to the request list. But remeber that the request list is modified by read_intr and write_intr. They set this_request to this_request->next after the currrent read/write is over. So naturally, we are not supposed to change these two values without blocking the harddisk interrupt. Saying that we should not change this_requset->next might not be very obvious - but imagine the situation where the tmp above is assigned this_request, but immediately after the assignment. we got an interrupt and this_request is assigned this_request->next. Now we comeback and assign the new request req to the "old" this_request->next, but the old this_request is no more in the list and so the new request gets lost!! Again, we don't know what happens if this_request->next is NULL - then an interrupt comes and this_request becomes this_request->next == NULL!! Now we come back to this function and assign the new request to tmp->next which is "old" this_request->next. So does not the new request get lost ? We don't know - find it out yourself ^_^. So we chainging only from this_request->next->next. But how does that help ? Before we "anchor" tmp to this_request (tmp = this_request), we set a variable soting to one. So suppose an interrupt comes before sorting is assigned one!! No worry - even if any number of interrupts come before sorting is assigned one, there are no issues because we have not "anchored" the value of tmp to this_request. But what if an interrupt comes after sorting = 1 ?? Then in the interrupt, this_request is modified to this_request->next - that does not cause any proble because we are not touching the first two values. Now the read_intr/write_intr calls do_request t processe the next request on the head of the queue. Now if the next request is processed, then "as soon as" that request is processed another harddisk interrupt may come (for the completion of the request) and again this_request may get modified - but this we can't handle because we have given provision only for the first two entries. So what do_request does is to return without processing any requests if it finds that sorting is set to one.*

```
1
2 /*
3  * NOTE! As a result of sorting, the interrupts may have died down,
4  * as they aren't redone due to locking with sorting=1. They might
5  * also never have started, if this is the first request in the queue,
```

```
 6  * so we restart them if necessary.
 7  */
 8         if (!do_hd)
 9                 do_request();
10
```

* *If this was our first request, process it.*

```
 1
 2 }
 3
 4 void rw_abs_hd(int rw,unsigned int nr,unsigned int sec,unsigned int head,
 5         unsigned int cyl,struct buffer_head * bh)
 6 {
 7         struct hd_request * req;
 8
 9         if (rw!=READ && rw!=WRITE)
10                 panic("Bad hd command, must be R/W");
11         lock_buffer(bh);
12 repeat:
13         for (req=0+request ; req<NR_REQUEST+request ; req++)
14                 if (req->hd<0)
15                         break;
16         if (req==NR_REQUEST+request) {
17                 sleep_on(&wait_for_request);
18                 goto repeat;
19         }
20         req->hd=nr;
21         req->nsector=2;
22         req->sector=sec;
23         req->head=head;
24         req->cyl=cyl;
25         req->cmd = ((rw==READ)?WIN_READ:WIN_WRITE);
26         req->bh=bh;
27         req->errors=0;
28         req->next=NULL;
29         add_request(req);
30         wait_on_buffer(bh);
31 }
32
```

* *This is the outermost layer of the wrapper. This calls the required functions mentioned above to do a read or write request appropriately and waits till the lock on the buffer is released.*

```
 1
 2
 3 void hd_init(void)
 4 {
 5         int i;
 6
 7         for (i=0 ; i<NR_REQUEST ; i++) {
 8                 request[i].hd = -1;
 9                 request[i].next = NULL;
10         }
11         for (i=0 ; i<NR_HD ; i++) {
12                 hd[i*5].start_sect = 0;
13                 hd[i*5].nr_sects = hd_info[i].head*
```

```
14                                      hd_info[i].sect*hd_info[i].cyl;
15          }
16          set_trap_gate(0x2E,&hd_interrupt);
17          outb_p(inb_p(0x21)&0xfb,0x21);
18          outb(inb_p(0xA1)&0xbf,0xA1);
19  }
20
```

6.1.2.5. linux/kernel/exit.c

We are not explaining this file because there is nothing complicated in this file. sys_exit just frees the page tables allocated for the process, marks the process as a “ZOMBIE” and sends a SIGCHLD to the parent. The sys_waitpid process waits till the child with the specified pid has terminated!!

6.1.3. linux/mm

This directory contains the minimal support needed for virtual memory support. The code here is basically to handle a page fault because of the two reasons - no “page present” and “page is read only”.

6.1.3.1. linux/mm/page.s

Refer the Intel manual to see how to identify the “cause” of a page fault. All that the below function does is to identify the cause of the page fault and call a “no page” handling routine or “page read only” routine.

```
 1
 2 /*
 3  * page.s contains the low-level page-exception code.
 4  * the real work is done in mm.c
 5  */
 6
 7 .globl _page_fault
 8
 9 _page_fault:
10          xchgl %eax,(%esp)
11          pushl %ecx
12          pushl %edx
13          push %ds
14          push %es
15          push %fs
16          movl $0x10,%edx
17          mov %dx,%ds
18          mov %dx,%es
19          mov %dx,%fs
20          movl %cr2,%edx
21
```

* *Get the “virtual” address that caused the page fault.*

```
 1
```

```
2          pushl %edx
3          pushl %eax
4          testl $1,%eax
5          jne 1f
6
```

* *Find out the cause of the page fault.*

```
1
2          call _do_no_page
3          jmp 2f
4 1:       call _do_wp_page
5
```

* *Call a “no page” handler or “page read only” handler depending on the cause of the page fault.*

```
1
2 2:       addl $8,%esp
3          pop %fs
4          pop %es
5          pop %ds
6          popl %edx
7          popl %ecx
8          popl %eax
9          iret
10
```

6.1.3.2. linux/mm/memory.c

The main functions in this file are for getting a free “physical” page, freeing pages in a “range” of virtual address and copying pages from one range of virtual address to another range.

```
1
2 #include <signal.h>
3
4 #include <linux/config.h>
5 #include <linux/head.h>
6 #include <linux/kernel.h>
7 #include <asm/system.h>
8
9 int do_exit(long code);
10
11 #define invalidate() \
12 __asm__("movl %%eax,%%cr3"::"a" (0))
13
14 #if (BUFFER_END < 0x100000)
15 #define LOW_MEM 0x100000
16 #else
17 #define LOW_MEM BUFFER_END
18 #endif
19
```

* *The lower one Mb - 0x100000 is considered as a “special” area used by the kernel - that is why we have special macros for that!!*

```
1
2
3 /* these are not to be changed - thay are calculated from the above */
4 #define PAGING_MEMORY (HIGH_MEMORY - LOW_MEM)
5 #define PAGING_PAGES (PAGING_MEMORY/4096)
6 #define MAP_NR(addr) (((addr)-LOW_MEM)>>12)
7
8 #if (PAGING_PAGES < 10)
9 #error "Won't work"
10 #endif
11
12 #define copy_page(from,to) \
13 __asm__("cld ; rep ; movsl"::"S" (from),"D" (to),"c" (1024):"cx","di","si")
14
15 static unsigned short mem_map [ PAGING_PAGES ] = {0,};
16
```

* *The size of the array corresponds to the number of physical pages actually present in the system. A value of n in mem_map[i] denotes that physical page i is being shared (used) by n processes - if n == 0, then that page is free for use.*

```
1
2 /*
3  * Get physical address of first (actually last :-) free page, and mark it
4  * used. If no free pages left, return 0.
5  */
6 unsigned long get_free_page(void)
7 {
8 register unsigned long __res asm("ax");
9
10 __asm__("std ; repne ; scasw\n\t"
11
```

* *Find out the first free page (count 0) using the aray mem_map.*

```
1
2         "jne 1f\n\t"
3         "movw $1,2(%%edi)\n\t"
4
```

* *We got a page, mark the count for the page as one (1).*

```
1
2         "sall $12,%%ecx\n\t"
3         "movl %%ecx,%%edx\n\t"
4         "addl %2,%%edx\n\t"
5
```

* *Calculate the actual physical address for the procured free page. The page "number" is present in ecx. So the physical address is calculated as (4k * ecx) + 0x100000.*

```
1
2         "movl $1024,%%ecx\n\t"
3         "leal 4092(%%edx),%%edi\n\t"
4         "rep ; stosl\n\t"
5
```

* *Fill the entire 4k page with zeroes!!*

```
1
2         "movl %%edx,%%eax\n"
3
```

* *Return the physical address of the free page that we got!*

```
1
2         "1:"
3         :"=a" (__res)
4         :"0" (0),"i" (LOW_MEM),"c" (PAGING_PAGES),
5         "D" (mem_map+PAGING_PAGES-1)
6         :"di","cx","dx");
7 return __res;
8 }
9
10 /*
11  * Free a page of memory at physical address 'addr'. Used by
12  * 'free_page_tables()'
13  */
14 void free_page(unsigned long addr)
15 {
16         if (addr<LOW_MEM) return;
17         if (addr>HIGH_MEMORY)
18                 panic("trying to free nonexistent page");
19         addr -= LOW_MEM;
20         addr >>= 12;
21         if (mem_map[addr]--) return;
22         mem_map[addr]=0;
23         panic("trying to free free page");
24 }
25
26 /*
27  * This function frees a continuos block of page tables, as needed
28  * by 'exit()'. As does copy_page_tables(), this handles only 4Mb blocks.
29  */
30
```

* *Here we have to remember that the 386 has got a two level page heirarchy - a page directory, a page table and finally the actual page. So if we have to free the physical pages in a range of address, we have start from the page directory, see if a page table is present, if a page table is present, free all the available (present) pages in the page table, then free the page table too and mark the page directory entry accordingly. Repeat this process till the whole virtual address range is covered. Also, remember that we free only blocks of 4Mb.*

```
1
2 int free_page_tables(unsigned long from,unsigned long size)
3 {
4         unsigned long *pg_table;
5         unsigned long * dir, nr;
6
7         if (from & 0x3fffff)
8                 panic("free_page_tables called with wrong alignment");
9         if (!from)
10                panic("Trying to free up swapper memory space");
11        size = (size + 0x3fffff) >> 22;
12
```

* *Calculate the number of 4Mb blocks to be freed. This means that we calculate the number of page directory entries to be freed. Remeber that one page directory entry can map 4Mb. Now we mainly use this function while we terminate one process (exit(), exec() etc..), so freeing in 4Mb blocks of virtual address is acceptbale - remember that the total virtual address range assigned to one process is 64Mb!!!*

```
 1
 2         dir = (unsigned long *) ((from>>20) & 0xffc); /* _pg_dir = 0 */
 3
```

* *Refer to the Intel manual if you have forgotten how to calculate the index into the page directory from the virtual address. Now that you have got the index into the page directory, add that to the base address of the page directory to get the address of the page directory entry. But again, the page directory base is 0x0 (refer head.s)!!!*

```
 1
 2         for ( ; size-->0 ; dir++) {
 3                 if (!(1 & *dir))
 4                         continue;
 5                 pg_table = (unsigned long *) (0xfffff000 & *dir);
 6                 for (nr=0 ; nr<1024 ; nr++) {
 7                         if (1 & *pg_table)
 8                                 free_page(0xfffff000 & *pg_table);
 9                         *pg_table = 0;
10                         pg_table++;
11                 }
12                 free_page(0xfffff000 & *dir);
13                 *dir = 0;
14         }
15         invalidate();
16         return 0;
17 }
18
19 /*
20  *  Well, here is one of the most complicated functions in mm. It
21  * copies a range of linerar addresses by copying only the pages.
22  * Let's hope this is bug-free, 'cause this one I don't want to de-
   bug :-)
23  *
24  * Note! We don't copy just any chunks of memory - addresses have to
25  * be divisible by 4Mb (one page-directory entry), as this makes the
26  * function easier. It's used only by fork anyway.
27  *
28  * NOTE 2!! When from==0 we are copying kernel space for the first
29  * fork(). Then we DONT want to copy a full page-directory entry, as
30  * that would lead to some serious memory waste - we just copy the
31  * first 160 pages - 640kB. Even that is more than we need, but it
32  * doesn't take any more memory - we don't copy-on-write in the low
33  * 1 Mb-range, so the pages can be shared with the kernel. Thus the
34  * special case for nr=xxxx.
35  */
36 int copy_page_tables(unsigned long from,unsigned long to,long size)
37 {
38
```

* *This function is used only by the fork() system call. That again explains why we copy in terms of 4Mb blocks!! The basic algorithm is like this (similar to free_page_tables) - calculate the number of page directory entries*

to be copied. Then get one page for the page table and assign it to the destination page directory entry. Then (for normal processes), for each page table entry in the source page table, set the SAME physical address to the corresponding destination page table entry also and mark BOTH the page table entries as "read only". Continue doing this till all the necessary page directory entries have been copied. But the only exception is for the first fork() which is the fork() by the kernel - when the kernel forks(), it is just to create a new process - that new process will not be using any of the kernels' data structures or buffers. So why try to copy an entire address range ? So the only thing we need to ensure is that the kernel code is copied to the child's address space also (since the fork() by the kernel and all the following code are also part of the kernel code, if the chil has to run, that code has to be copied). The kernel code will come well within thefirst 640kB. So when the task0 is forking, we copy only 640Kb.

```
 1
 2         unsigned long * from_page_table;
 3         unsigned long * to_page_table;
 4         unsigned long this_page;
 5         unsigned long * from_dir, * to_dir;
 6         unsigned long nr;
 7
 8         if ((from&0x3fffff) || (to&0x3fffff))
 9                 panic("copy_page_tables called with wrong alignment");
10         from_dir = (unsigned long *) ((from>>20) & 0xffc); /* _pg_dir = 0 */
11         to_dir = (unsigned long *) ((to>>20) & 0xffc);
12         size = ((unsigned) (size+0x3fffff)) >> 22;
13
```

* *Calculate the number of page directory entries to be copied.*

```
 1
 2         for( ; size-->0 ; from_dir++,to_dir++) {
 3                 if (1 & *to_dir)
 4                         panic("copy_page_tables: already exist");
 5                 if (!(1 & *from_dir))
 6                         continue;
 7                 from_page_table = (unsigned long *) (0xfffff000 & *from_dir);
 8                 if (!(to_page_table = (unsigned long *) get_free_page()))
 9                         return -1;      /* Out of memory, see free-
ing */
10
```

* *Get a new page to be used as the destination page table.*

```
 1
 2                 *to_dir = ((unsigned long) to_page_table) | 7;
 3                 nr = (from==0)?0xA0:1024;
 4
```

* *Is the task0 forking ? Then copy only 640Kb.*

```
 1
 2                 for ( ; nr-- > 0 ; from_page_table++,to_page_table++) {
 3                         this_page = *from_page_table;
 4                         if (!(1 & this_page))
 5                                 continue;
 6
```

* *Get the physical page from the source page table entry.*

```
1
2                           this_page &= ~2;
3                           *to_page_table = this_page;
4
```

* *Mark the destination as read only (this_page &= ~2).*

```
1
2                           if (this_page > LOW_MEM) {
3                                   *from_page_table = this_page;
4
```

* *If the source page is not a kernel page, mark the source page also as "read only". Of course, we should not set the kernel pages as read only - this will cause a page fault from the kernel and page faults in kernel are not allowed.*

```
1
2                                   this_page -= LOW_MEM;
3                                   this_page >>= 12;
4                                   mem_map[this_page]++;
5
```

* *Again, if the source page is not a kernel page, increase the reference count to that page.*

```
1
2                           }
3                   }
4           }
5           invalidate();
6           return 0;
7   }
8
9   /*
10   * This function puts a page in memory at the wanted address.
11   * It returns the physical address of the page gotten, 0 if
12   * out of memory (either when trying to access page-table or
13   * page.)
14   */
15  unsigned long put_page(unsigned long page,unsigned long address)
16  {
17          unsigned long tmp, *page_table;
18
19  /* NOTE !!! This uses the fact that _pg_dir=0 */
20
21          if (page < LOW_MEM || page > HIGH_MEMORY)
22                  printk("Trying to put page %p at %p\n",page,address);
23          if (mem_map[(page-LOW_MEM)>>12] != 1)
24                  printk("mem_map disagrees with %p at %p\n",page,address);
25          page_table = (unsigned long *) ((address>>20) & 0xffc);
26          if ((*page_table)&1)
27                  page_table = (unsigned long *) (0xfffff000 & *page_table);
28          else {
29                  if (!(tmp=get_free_page()))
30                          return 0;
31                  *page_table = tmp|7;
32                  page_table = (unsigned long *) tmp;
33          }
```

```
34          page_table[(address>>12) & 0x3ff] = page | 7;
35          return page;
36  }
37
```

* *Not a very difficult function to understand.*

```
1
2
3  void un_wp_page(unsigned long * table_entry)
4  {
5
```

* *Remeber that during a fork(), bith the source and destination pages are shared and BOTH are marked as read only. Now suppose the parent tries to write to the shared page. Then it gets a page fault and we reach here. Then we get a new page for the parent and copies the contents of the old page to the new page and sets the new page as writable and decrements the reference count of the page (it becomes 1). But remeber that the child still has the old page as readonly. So when the child tries to write to the old page, it gets a page fault and comes here. But here the code says that if the page is referred to by only one process and still it is marked read only when we get a page fault, then mark the page as writable. So that solves all issues!!*

```
1
2          unsigned long old_page,new_page;
3
4          old_page = 0xfffff000 & *table_entry;
5          if (old_page >= LOW_MEM && mem_map[MAP_NR(old_page)]==1) {
6                  *table_entry |= 2;
7                  return;
8          }
9
```

* *If the page reference count is only 1, it means that the other "page sharing" has already page faulted and got a new page, so now we are the sole owner of this page. So set the permission of the page to writable.*

```
1
2          if (!(new_page=get_free_page()))
3                  do_exit(SIGSEGV);
4          if (old_page >= LOW_MEM)
5                  mem_map[MAP_NR(old_page)]--;
6          *table_entry = new_page | 7;
7          copy_page(old_page,new_page);
8
```

* *If reference count $>$ 1, get a new page, mark it as writable and copy the contents of the old page to the new one. Also, decrement the reference count for the old page.*

```
1
2  }
3
4  /*
5   * This routine handles present pages, when users try to write
6   * to a shared page. It is done by copying the page to a new address
7   * and decrementing the shared-page counter for the old page.
8   */
9  void do_wp_page(unsigned long error_code,unsigned long address)
10 {
```

```
11          un_wp_page((unsigned long *)
12                  (((address>>10) & 0xffc) + (0xfffff000 &
13                  *((unsigned long *) ((address>>20) &0xffc)))));
14
```

* *Translate the virtual address to the page table entry and call un_wp_page.*

```
 1
 2
 3 }
 4
 5 void write_verify(unsigned long address)
 6 {
 7
```

* *This function sees whether the page present at the specified address (if at all a page is present) is readonly. If so, this puts a new page at that address and makes it writable. Now remeber the explanation that we gave about verify_area in fork.c ? There we mentioned that write_verify will put a new page if there is no page corresponding to that address. But from the above explanation, that is not true though we feel it should have been so. In this case what will verify_area do if there is no physical page present at the mentioned virtual address ? It will not do anything!! So what if we access that address from the kernel space ?? Won't we get a page fault ?? We don't know - figure it out yourself ;-)*

```
 1
 2          unsigned long page;
 3
 4          if (!( (page = *((unsigned long *) ((address>>20) & 0xffc)) )&1))
 5                  return;
 6          page &= 0xfffff000;
 7          page += ((address>>10) & 0xffc);
 8          if ((3 & *(unsigned long *) page) == 1)  /* non-writeable, present *,
 9                  un_wp_page((unsigned long *) page);
10          return;
11 }
12
13 void do_no_page(unsigned long error_code,unsigned long address)
14 {
15          unsigned long tmp;
16
17          if (tmp=get_free_page())
18                  if (put_page(tmp,address))
19                          return;
20          do_exit(SIGSEGV);
21 }
22
23 void calc_mem(void)
24 {
25          int i,j,k,free=0;
26          long * pg_tbl;
27
28          for(i=0 ; i<PAGING_PAGES ; i++)
29                  if (!mem_map[i]) free++;
30          printk("%d pages free (of %d)\n\r",free,PAGING_PAGES);
31          for(i=2 ; i<1024 ; i++) {
32                  if (1&pg_dir[i]) {
33                          pg_tbl=(long *) (0xfffff000 & pg_dir[i]);
```

```
34                          for(j=k=0 ; j<1024 ; j++)
35                                  if (pg_tbl[j]&1)
36                                          k++;
37                          printk("Pg-dir[%d] uses %d pages\n",i,k);
38                  }
39          }
40 }
41
```

6.1.4. linux/fs

Well, the file system is one area where we are uncomfortable with the code. So we will be explaining only exec.c. The rest of the file system code we will not be explaining “file wise” - most of the file system code is more or less of algorithms and has nothing much to do with the working of the kernel. So we will give a brief idea of how the file system and the /dev/ devices work in general and move forward to the exec.c explanation.

- The lowermost part of the file system code is the code to read a specified location on the hard disk into a buffer in kernel space. The code for this has been discussed clearly in kernel/hd.c.
- Now the whole of Unix works on the idea of a “root” file system. The kernel as a whole has a root directory and each process can have a different root directory. Now the partition on the harddisk which corresponds to the root file system is hard coded in the kernel by using the macro ROOT_DEV in include/linux/config.h. So during boot up, what the kernel does is to read the partition table which has a standard format and locate the begining of the root partition. Then it reads in the superblock for the minix file system and loctates the location of the inode of the root directory (refer Maurice.J.Bach or Tanenbaum for details on inodes, directory structure etc..). Now once we get the location of the root inode, then it is easy to locate any other element in the file system.
- Once the root inode has been located, the next level is the code to locate inodes given the file name. This is also a pure algorithmic task given the organisation of blocks in the file system.
- Once the file name to block translation code is also there, then the next piece of code that is needed is the code which deals with open, read, write etc.. for files and devices. This is the place where distinctions like block device, char device, normal file, pipe device etc.. assumes importance. The type of each file (b, c, p etc..) is again stored in the inode. So once you get the inode of the file to be opened, read, write or closed, we know what device it is from its’ major number (which corresponds to c, b, p etc..). Take for example the open system call. Depending on the type of the file, the open system call invokes different functions for block device, char device etc.. So in the kernel, we will have different sets of functions for open, read, write, close, ioctl etc.. for “each” type of device. These functions may call other functions in the layers mentioned previously depending on what device it is. For example a read on a block devie will use the lower most layer of functions discussed in the beginning that just gets you the data from a specified location on the disk. But a read for a normal file may make use of a function that translates the file name to a disk block and then calls the lower layers.

So with the above four categories of code, all the necessary functions for the file system can be implemented. Again, all the files in the fs/ directory (except exec.c) can be categorised

into either of the above four categories. So with that, we are stopping our discussion on file systems. Now let us explain exec.c

6.1.4.1. linux/fs/exec.c

In this file, the only function that has “system code” is the do_execve function. The rest are just helper functions. But the function do_execve is pretty straight forward, almost all the other functions are pretty complicated by the lack of explanation. So let us look into those functions.

```
 1
 2    #include <errno.h>
 3 #include <sys/stat.h>
 4 #include <a.out.h>
 5
 6 #include <linux/fs.h>
 7 #include <linux/sched.h>
 8 #include <linux/kernel.h>
 9 #include <linux/mm.h>
10 #include <asm/segment.h>
11
12 extern int sys_exit(int exit_code);
13 extern int sys_close(int fd);
14
15 /*
16  * MAX_ARG_PAGES defines the number of pages allocated for arguments
17  * and envelope for the new program. 32 should suffice, this gives
18  * a maximum env+arg of 128kB !
19  */
20 #define MAX_ARG_PAGES 32
21
```

* *We reserve 32 pages for argv, envp and the tables needed to represent them. First of all, let us explain from where the argv and envp comes into existence. It is very simple - when we call the execve function, we pass the argv and envp pointers as arguements to the function ! As simple as that. So what happens when we type a command and arguments at the shell prompt ? Well, that is also simple - the shell takes those strings an puts it into a two dimensional array and passes it as arguements to the execve call. So what are the implications of this ? This means that the argv and envp address that the kernel gets are in the user space (data segment) - so when ever the kernel needs to access that data, kernel will have to use LDT index 0x17. Or in C code, the kernel will have to call get_user_fs or put_user_fs.*

```
 1
 2
 3 #define cp_block(from,to) \
 4 __asm__("pushl $0x10\n\t" \
 5         "pushl $0x17\n\t" \
 6         "pop %%es\n\t" \
 7         "cld\n\t" \
 8         "rep\n\t" \
 9         "movsl\n\t" \
10         "pop %%es" \
11         ::"c" (BLOCK_SIZE/4),"S" (from),"D" (to) \
12         :"cx","di","si")
13
14
```

* *The above function does a "fast" copy by utilizing the x86 repeat instruction. In short, the rep instruction copies data from ds:si to es:di. So the es used should denote the user segment. That is why initially we do a push 0x17 and pop to es. At the end of the function, we restore es to 0x10 (kernel segment). The above function copies data from the kernel space to the user space. The first arguement is a kernel offset and the second one is a user space offset.*

```
 1
 2 /*
 3  * read_head() reads blocks 1-6 (not 0). Block 0 has already been
 4  * read for header information.
 5  */
 6 int read_head(struct m_inode * inode,int blocks)
 7 {
 8         struct buffer_head * bh;
 9         int count;
10
11         if (blocks>6)
12                 blocks=6;
13         for(count = 0 ; count<blocks ; count++) {
14                 if (!inode->i_zone[count+1])
15                         continue;
16                 if (!(bh=bread(inode->i_dev,inode->i_zone[count+1])))
17                         return -1;
18                 cp_block(bh->b_data,count*BLOCK_SIZE);
19                 brelse(bh);
20         }
21         return 0;
22 }
23
24 int read_ind(int dev,int ind,long size,unsigned long offset)
25 {
26         struct buffer_head * ih, * bh;
27         unsigned short * table,block;
28
29         if (size<=0)
30                 panic("size<=0 in read_ind");
31         if (size>512*BLOCK_SIZE)
32                 size=512*BLOCK_SIZE;
33         if (!ind)
34                 return 0;
35         if (!(ih=bread(dev,ind)))
36                 return -1;
37         table = (unsigned short *) ih->b_data;
38         while (size>0) {
39                 if (block=*(table++))
40                         if (!(bh=bread(dev,block))) {
41                                 brelse(ih);
42                                 return -1;
43                         } else {
44                                 cp_block(bh->b_data,offset);
45                                 brelse(bh);
46                         }
47                 size -= BLOCK_SIZE;
48                 offset += BLOCK_SIZE;
49         }
50         brelse(ih);
51         return 0;
```

```
52 }
53
54 /*
55  * read_area() reads an area into %fs:mem.
56  */
57 int read_area(struct m_inode * inode,long size)
58 {
59         struct buffer_head * dind;
60         unsigned short * table;
61         int i,count;
62
63         if ((i=read_head(inode,(size+BLOCK_SIZE-1)/BLOCK_SIZE)) ||
64             (size -= BLOCK_SIZE*6)<=0)
65                 return i;
66         if ((i=read_ind(inode->i_dev,inode->i_zone[7],size,BLOCK_SIZE*6)) ||
67             (size -= BLOCK_SIZE*512)<=0)
68                 return i;
69         if (!(i=inode->i_zone[8]))
70                 return 0;
71         if (!(dind = bread(inode->i_dev,i)))
72                 return -1;
73         table = (unsigned short *) dind->b_data;
74         for(count=0 ; count<512 ; count++)
75                 if ((i=read_ind(inode->i_dev,*(table++),size,
76                     BLOCK_SIZE*(518+count))) || (size -= BLOCK_SIZE*512)<=0)
77                         return i;
78         panic("Impossibly long executable");
79 }
80
```

* *The above three functions are self explanatory. Because of the multiple indexing levels in the file system block organisation (We forgot the number of levels for minix v1.0, refer Tanenbaum), we need to jump to different levels depending on what offset into the file we are reading. Also note that the function reads from the file and copies it into memory starting at offset zero. Actually this function is used for loading the file to be executed into memory. One more thing to be mentioned at this point is that Linux 0.01 is not capable of swapping or demand loading or any circus of that sort. It is a very simple piece of code and requires the whole program to be in memory (only the program and static piece of data are needed - it alloocates memory pages on demand). So it makes sense to load the whole file from offset zero. That is, if the file is of size 3K, then load the file from offset 0 to 3K in memory (the task has its own base address). Now the next few functions are pretty cryptic. All these functions are inter-related. So at this point, let us give an explanation about argv, envp and how they can be accessed by the "to be" executed program after it starts execution. The system allows a maximum of (MAX_ARG_PAGES - 4) for storing the argv and envp data. The kernel allows a maximum of 64Mb contiguous virtual address space (includnig the initial code + static data and future expansion like malloc and stack space) for any process. Now the envp and argv are placed at the bottom of this virtual space. On top of that comes the stack, then there will be a unused space after the last byte of the stack and then above that will come the dynamically expanded data, then the static data, then the code. It looks somewhat as below. High memory means memory with numerically larger address and low memory is its opposite (so 0x0 is low memory and 0x4000000 is high memory). 0x0 : Code - Static Data - Dynamic Data - Free Space - Expanding Stack - envp - argv - argc - tables - argv contents - envp contents: 0x4000000 So the below functions are all meant to arrange memory in this fashion. What do we mean by argv contents ? The argv is a two dimensional array, right ? Say it is argv[3][..] = {"/bin/ls", "-la", "/home/guest"} and let envp[2][] = {"HOME=/home/guest", "PATH=/bin"} So argv and envp contents are laid out as below in memory Low memory: /bin/ls NULL -la NULL /home/guest NULL HOME=/home/guest NULL PATH=/bin NULL : 0x4000000 Now what do we mean by the "tables" which come after the argv and envp contents ?*

*It is for these tables that we allocated 4 out of MAX_ARG_PAGES. The tables for the above argv and envp will look as below. Low mem: arg_address1 arg_address2 NULL env_address1 env_address2 env_address3 NULL - argv contents - envp contents: 0x4000000 argv_address1 is the address of the "/" in the beginning of /bin, argv_address2 points to the beginning "-" in -la etc.. Now how does the rest of the stuff look like ? Low mem: 3 address1 address2 - tables - etc...etc.. : High mem Now 3 denots argc. address1 is the address in memory where arg_address1 is stored. Actually, in the program to be executed, this will be the the address of argv in the argv[][] that is used as the arguement in main - int main (int argc, char *argv[][], char *envp[][]). address2 is the address in memory where env_address1 is stored and this address2 becomes the address envp passed to main. So that was a pretty detailed explanation of a trivial stuff :-(! Now let us explain the above function. Actually, the explanation of functions is in the reverse order because it was written in the reverser order ! Suppose that the the argv and envp contents are already laid out in memory and we have been given the address (arguement p) which is the address of "/" in /bin. Also, we are given the argc and envc count. Now what do we have to do ? We have to create the tables - that is the final step. Now how do we do it ? See below.*

```
1
2 /*
3  * create_tables() parses the env- and arg-strings in new user
4  * memory and creates the pointer tables from them, and puts their
5  * addresses on the "stack", returning the new stack pointer value.
6  */
7 static unsigned long * create_tables(char * p,int argc,int envc)
8 {
9         unsigned long *argv,*envp;
10        unsigned long * sp;
11
12        sp = (unsigned long *) (0xfffffffc & (unsigned long) p);
13        sp -= envc+1;
14
```

* *Leave space for the enp table - here we store env_addres1 env_addres2*

```
1
2         envp = sp;
3
```

* *So this becomes the envp in main(..)*

```
1
2         sp -= argc+1;
3
```

* *Leave space for argv. Here we store env_addres1 env_addres2 env_addres3.*

```
1
2         argv = sp;
3
```

* *So this becomes the argv in main(...).*

```
1
2         put_fs_long((unsigned long)envp,--sp);
3         put_fs_long((unsigned long)argv,--sp);
4         put_fs_long((unsigned long)argc,--sp);
5
```

* *Now put the values 3 argv envp in memory.*

```
1
2         while (argc-->0) {
3                 put_fs_long((unsigned long) p,argv++);
4                 while (get_fs_byte(p++)) /* nothing */ ;
5         }
6
```

* *Remeber that p points to the beginning of "/" in /bin. So p is the env_addres1 and we write that to address argv - note that argv is a user space address and so we use put_fs_long. Now the loop is self explanatory - we get the env_address2 and so on from the loop.*

```
1
2         put_fs_long(0,argv);
3
```

* *Put a NULL after the argv table - that is why we added a 1 while allocating space for argv table.*

```
1
2         while (envc-->0) {
3                 put_fs_long((unsigned long) p,envp++);
4                 while (get_fs_byte(p++)) /* nothing */ ;
5         }
6         put_fs_long(0,envp);
7
```

* *Now after constructing argv table, construct envp table along the same lines.*

```
1
2         return sp;
3
```

* *Note that we had calculated the value of sp in the beginning and never modified it after that. sp points just above the argc (3). Now this address is the starting address of the stack for this process.*

```
1
2 }
3
4 /*
5  * count() counts the number of arguments/envelopes
6  */
7 static int count(char ** argv)
8 {
9         int i=0;
10        char ** tmp;
11
12        if (tmp = argv)
13                while (get_fs_long((unsigned long *) (tmp++)))
14                        i++;
15
16        return i;
17 }
18
```

* *Given argc or envp which were passed as arguements to execve system call, find the first dimentsion (argc/envc) of those two-d arrays.*

```
1
2
3 /*
4  * 'copy_string()' copies argument/envelope strings from user
5  * memory to free pages in kernel mem. These are in a format ready
6  * to be put directly into the top of new user memory.
7  */
8 static unsigned long copy_strings(int argc,char ** argv,unsigned long *page,
9                 unsigned long p)
10 {
11         int len,i;
12         char *tmp;
13
```

* *This is a pretty big function. What it does is this - the argument p is the amount of memory out of (MAX_ARG_PAGES - 4)*PAGESIZE that remains to be used for copying the contents of arguement argv (or envp) contents into memory. At the point where this funtion is first called, there are no pages in memory for copying the contents of argv or envp. We may need upto max of MAX_ARG_PAGES. So we have an array page[MAX_ARG_PAGES] (second argument) which stores the address of the ith page allocated for copying argv or envp contents (i < MAX_AARG_PAGES). Say if 9 pages have already been allocated for copying and say we need another page again, then we allocate one more page and store its address in the page[i] index. Also, note that we start filling the page[] array from the last index (MAX_ARG_PAGE) down to 0. This is because, in the end, after copying both argv and envp contents, we put these pages into memory from 0x4000000 upwards using index MAX_ARG_PAGES to 0. ie page[MAX_ARG_PAGES] will contain the address of the page that will be put at address 0x4000000 (using put_page).*

```
1
2
3         while (argc-- > 0) {
4
```

* *Loop over each first dimension in the two-d array - argv[argc][..].*

```
1
2                 if (!(tmp = (char *)get_fs_long(((unsigned long *) argv)+argc
3                         panic("argc is wrong");
4
```

* *Oh! We encountered a NULL in argv[argc][] before expected ?? That is a bug.*

```
1
2                 len=0;          /* remember zero-padding */
3                 do {
4                         len++;
5                 } while (get_fs_byte(tmp++));
6
```

* *So we got the second dimension address argv[argc] which points to one string like "/bin/ls". Note that we are starting with argc and going down to 0. Similarly, we are proceeding from high address p ((MAX_ARG_PAGES - 4) * PAGESIZE) to lower address. So the last string in argv[] will be in high memory and the first one in low memory. So now we got the length of the string in argv[argc]*

```
1
2                   if (p-len < 0)          /* this shouldn't happen -
128kB */
3                           return 0;
4
```

* *p - len < 0 means we don't have space to store this string :-(*

```
1
2                   i = ((unsigned) (p-len)) >> 12;
3
```

* *Remember that we are proceeding backwards from MAX_ARG_PAGES to 0. So the above calculation says that - "hey, we have already filled memory upto address p, now we need to fill the contents of argv[argc] from p-len to p. So what is the page number for that ? So it gets you the page number.*

```
1
2                   while (i<MAX_ARG_PAGES && !page[i]) {
3                           if (!(page[i]=get_free_page()))
4                                   return 0;
5                           i++;
6                   }
7
```

* *So we know that at this pont, filling has been done till address p - say page 10. now filling has to be done from p-len to p - say that corresponds to pages 5 to 10. So the above loop allocates kernel pages for that purpose.*

```
1
2                   do {
3                           --p;
4                           if (!page[p/PAGE_SIZE])
5                                   panic("nonexistent page in exec.c");
6                           ((char *) page[p/PAGE_SIZE])[p%PAGE_SIZE] =
7                                   get_fs_byte(--tmp);
8                   } while (--len);
9
```

* *Now that we have got the necessary number of pages, fill it !! Think about the above modulo arithmetic and you will understand what is happening. Noe that page[p/PAGE_SIZE] contains the address got from _get_free_page(). This address is still a kernel address and has not been translated into a virtual address. That is why we are assigning to it directly without any put_fs. But the argv that we have been passed is a user space address. So to get the characters from it, we use get_fs_byte.*

```
1
2           }
3           return p;
4
```

* *Now we used up some space out of ((MAX_ARG_PAGES - 4) * PAGE_SIZE) in the above process. How much of the total is remaining ? Return that number so that the next fellow who calls this same function can start from that left over space and proceed to zero.*

```
1
2 }
```

```
 3
 4 static unsigned long change_ldt(unsigned long text_size,unsigned long * page)
 5 {
 6         unsigned long code_limit,data_limit,code_base,data_base;
 7         int i;
 8
 9         code_limit = text_size+PAGE_SIZE -1;
10         code_limit &= 0xFFFFF000;
11         data_limit = 0x4000000;
12
```

* *We give max 64Mb virtual space to each process.*

```
 1
 2         code_base = get_base(current->ldt[1]);
 3         data_base = code_base;
 4         set_base(current->ldt[1],code_base);
 5         set_limit(current->ldt[1],code_limit);
 6         set_base(current->ldt[2],data_base);
 7         set_limit(current->ldt[2],data_limit);
 8
```

* *Renmeber ? In fork(), we allocated new virtual address for the process. But in exec, we use the same address space as of the process that did the exec - that is logical because the process that called exec is going to be over written by the new one. So why not use its own address ? Yes, we can.*

```
 1
 2 /* make sure fs points to the NEW data segment */
 3         __asm__("pushl $0x17\n\tpop %%fs"::);
 4         data_base += data_limit;         for (i=MAX_ARG_PAGES-1 ; i>=0 ; i-
-) {
 5                 data_base -= PAGE_SIZE;
 6                 if (page[i])
 7                         put_page(page[i],data_base);
 8         }
 9
```

* *Now it is here that we put the pages in the page[] array which contains the addresses of the pages containging argv and envp contents. Note that we start from the last page, page[MAX_ZRG_PAGES] and proceed to zero and we put the pages from the end of the space allocated for the task and proceed to low memory.*

```
 1
 2         return data_limit;
 3
```

* *Well, data_limit has been assigned 0x4000000. So return 0x4000000.*

```
 1
 2 }
 3
 4 /*
 5  * 'do_execve()' executes a new program.
 6  */
 7 int do_execve(unsigned long * eip,long tmp,char * filename,
 8         char ** argv, char ** envp)
 9 {
```

```
10          struct m_inode * inode;
11          struct buffer_head * bh;
12          struct exec ex;
13          unsigned long page[MAX_ARG_PAGES];
14          int i,argc,envc;
15          unsigned long p;
16
17
```

* *Now comes the real function. This is pretty simple thanks to the other complicated functions.*

```
 1
 2          if ((0xffff & eip[1]) != 0x000f)
 3                  panic("execve called from supervisor mode");
 4          for (i=0 ; i<MAX_ARG_PAGES ; i++)       /* clear page-table */
 5                  page[i]=0;
 6          if (!(inode=namei(filename)))           /* get executables in-
ode */
 7                  return -ENOENT;
 8          if (!S_ISREG(inode->i_mode)) {  /* must be regular file */
 9                  iput(inode);
10                  return -EACCES;
11          }
12          i = inode->i_mode;
13          if (current->uid && current->euid) {
14                  if (current->euid == inode->i_uid)
15                          i >>= 6;
16                  else if (current->egid == inode->i_gid)
17                          i >>= 3;
18          } else if (i & 0111)
19                  i=1;
20          if (!(i & 1)) {
21                  iput(inode);
22                  return -ENOEXEC;
23          }
24
```

* *All the initialisations and permission checks.*

```
 1
 2          if (!(bh = bread(inode->i_dev,inode->i_zone[0]))) {
 3                  iput(inode);
 4                  return -EACCES;
 5          }
 6          ex = *((struct exec *) bh->b_data);     /* read exec-header */
 7          brelse(bh);
 8          if (N_MAGIC(ex) != ZMAGIC || ex.a_trsize || ex.a_drsize ||
 9                  ex.a_text+ex.a_data+ex.a_bss>0x3000000 ||
10                  inode->i_size < ex.a_text+ex.a_data+ex.a_syms+N_TXTOFF(ex)) {
11                  iput(inode);
12                  return -ENOEXEC;
13          }
14
```

* *Read the first block of the executable and see whether it is properly ni the a.out fomat. We remember that this did not somehow work for us and we got frustrated and gave up to use "raw" binary format - the main() starts at offset 0 in the file - pretty simple :-)*

```
1
2         if (N_TXTOFF(ex) != BLOCK_SIZE)
3                 panic("N_TXTOFF != BLOCK_SIZE. See a.out.h.");
4         argc = count(argv);
5         envc = count(envp);
6
```

* *Here is where we get the number of strings in argc and envp*

```
1
2         p = copy_strings(envc,envp,page,PAGE_SIZE*MAX_ARG_PAGES-4);
3         p = copy_strings(argc,argv,page,p);
4
```

* *Copy the envp and argv into memory. Note that envp is copied first. So envp comes in the highest address of the memory and argv comes above that.*

```
1
2         if (!p) {
3                 for (i=0 ; i<MAX_ARG_PAGES ; i++)
4                         free_page(page[i]);
5                 iput(inode);
6                 return -1;
7         }
8
```

* *If we did not get enough pages, free whatever was allocated and quit exec.*

```
1
2 /* OK, This is the point of no return */
3         for (i=0 ; i<32 ; i++)
4                 current->sig_fn[i] = NULL;
5         for (i=0 ; i<NR_OPEN ; i++)
6                 if ((current->close_on_exec>>i)&1)
7                         sys_close(i);
8         current->close_on_exec = 0;
9         free_page_tables(get_base(current->ldt[1]),get_limit(0x0f));
10        free_page_tables(get_base(current->ldt[2]),get_limit(0x17));
11        if (last_task_used_math == current)
12                last_task_used_math = NULL;
13        current->used_math = 0;
14
```

* *Free up memories of the “going-to-die” task.*

```
1
2         p += change_ldt(ex.a_text,page)-MAX_ARG_PAGES*PAGE_SIZE;
3
```

* *What are we doing above with += ? Till now the address p was from 0 to MAX_ARG_PAGES*PAGE_SIZE. Till this point we just copied the strings into memory and so we were not cared whether we used kernel addres or user address - we had to just get the strings into memory. But now we need the exact offset in memory where this is going to be copied - why ? Because we are going to construct tables below. And in tables, we need the actual user memory offset because the tables are going to be used by the user process. So we do the arithmetic above - the p += 0x4000000 - MAX_ARG_PAGES*PAGE_SIZE*

```
1
2         p = (unsigned long) create_tables((char *)p,argc,envc);
3
```

* *Now create the tables.*

```
1
2         current->brk = ex.a_bss +
3                 (current->end_data = ex.a_data +
4                 (current->end_code = ex.a_text));
5         current->start_stack = p & 0xfffff000;
6
```

* *Now whatever address is in p, assign that as the start of the stack.*

```
1
2         i = read_area(inode,ex.a_text+ex.a_data);
3
```

* *Now read the whole binary into memory.*

```
1
2         iput(inode);
3         if (i<0)
4                 sys_exit(-1);
5         i = ex.a_text+ex.a_data;
6         while (i&0xfff)
7                 put_fs_byte(0,(char *) (i++));
8         eip[0] = ex.a_entry;            /* eip, magic happens :-) */
9         eip[3] = p;                     /* stack pointer */
10
```

* *Now the above two lines is a very important piece of code and one that will confuse the person who is new to the concepts of stack and arguments in C. When we call a function in C, the top most element of the stack is the fisrt argument to the function, the next element of the stack is the second argument and so on. So what is the topmost element on the stack when do_execve is called ? Go to file kernel/system_call.s and trace out all the push instructions from lable system_call till call *sys_call_table instruction. We can see that the topmost elemet pushed on the stack is register ebx. Now how is do_execve called ? The call *sys_call_table(,%eax,4) instruction calls sys_execve. Now go to sys_execve. What are they doing there ? They are pusing something again onto the stack. What is that something ? Look into the Stack Layout given at the top of the file system_call.s. There we find that EIP(%esp) is the address in kernel stack where the address to which we have to return after the system call is stored. 3*4 bytes below that we have the address of the stack to be used on return. So what is the first parameter to do_execve ? It is the address on the kernel stack where we store the address to which we have to return to. So what are we doing in the above mentioned C assignment statements ? We are modifying the kernel stack to use the "new" address and "new" stack after the system call so that the "new" task will automatically start executing !! Wow that was a good piece of cryptyic code :-). Again, what is this "tmp" as the second parameter ? Any guesses ? After ebx, we pushed only the EIP(%esp), right ? NO.. NOT right, after the exb push, we did a call instruction which pushes something onto the stack. We don't remember now and are too lazy to consult the manuals :-) - there is some rule that if you do a call from the privileged (kernel) mode, then you push only the eip (and not the esp) or some rule like that - we don't remember. That is why we need only one long tmp as the second argument instead of one more tmp1 or something like that as the second arguement (oe even a long long as the second argument). We don't want to use the eip pushed by the kernel.*

```
1
2
3           /* OOOPS... Dangerous */
4           exec_graph(current);
5
6
```

* *To be honest, we don't remember what the exec_graph is :-(*

```
1
2           return 0;
3 }
4
```

6.2. The End

6.2.1. Do try and get 0.01 up and kicking

So we have come to the end of the book. Maybe if we had written the material without the one year or so break in between, we might have done better. The main parts like the processor descritpion, the description of the core kernel etc was written before the one year break when the enthusiasm level was very high. Anyway, we were not at all interested in the file system code and so even if we had written it without break, we would have excluded that. We request the seriosuly interested readers to try bringing up the kernel by themselves - it will help you learn a lot.